

PRACA DYPLOMOWA INŻYNIERSKA

Aplikacja wspomagająca zarządzanie budżetem

Application for Budget Management Support

Dawid Ziora

Nr albumu: 136700

Kierunek: Informatyka

Forma studiów: stacjonarne

Poziom studiów: I

Promotor pracy:

dr inż. Bartosz Kowalczyk

Praca przyjęta dnia:

Podpis promotora:

Częstochowa, 2025

Spis treści

1	Wstęp	5
1.1	Cel pracy	6
1.2	Zakres pracy	6
1.3	Struktura pracy	7
2	Przegląd technologii stosowanych w aplikacjach klient-serwer	9
2.1	Architektura systemu	9
2.2	Java	11
2.3	React	13
2.4	MongoDB	13
3	Narzędzia i metodyka pracy	15
3.1	Narzędzia stosowane w procesie wytwarzania aplikacji webowej . . .	15
4	Architektura aplikacji	17
4.1	Wymagania funkcjonalne	17
4.2	Wymagania нефункционалне	18
4.3	Diagramy UML	19
4.4	Diagramy przypadków użycia	19
4.5	Diagramy sekwencji	20
4.6	Diagram klas	20
5	Implementacja systemu	23
5.1	Backend – logika biznesowa i API	23
5.2	Frontend – interfejs użytkownika	34
5.3	Baza danych - implementacja i przepływ danych	37
5.4	Integracja warstw	38
6	Prezentacja aplikacji	41
6.1	Środowisko deweloperskie	41

6.2 Formularze autoryzacji	41
6.3 Ustawienia	42
6.4 Funkcjonalność aplikacji udostępniona użytkownikowi zalogowanemu	43
7 Podsumowanie i wnioski	49
Streszczenie	53
Abstract	55
Bibliografia	57
Spis rysunków	61
Spis tabel	63
Spis listingów	65

1. Wstęp

W gospodarstwach domowych jednym z kluczowych obszarów decyzyjnych mających bezpośredni wpływ na poziom zaspokojenia potrzeb jest strefa finansowa. W jej ramach podejmowane są decyzje związane z konsumpcją, oszczędzaniem bądź inwestowaniem.

Jednym z celów wykorzystania aplikacji wspomagających zarządzanie budżetem domowym jest wspieranie procesów decyzyjnych konsumentów [18]. Aplikacje finansowe podnoszą świadomość społeczeństwa dotyczącą finansów oraz zwiększają chęć do planowania i kontroli budżetu domowego. Usługi finansowe są dostępne w prawie każdym miejscu niemalże dwadzieścia cztery godziny na dobę. Aplikacje te najczęściej cechują się interfejsem łatwym w obsłudze, kategoryzacją i śledzeniem wydatków, możliwością przypominania o rachunkach oraz zapewniają ochronę informacji personalnych.

Wiele aplikacji jest tworzonych z myślą o użytkownikach, korzystających z urządzeń mobilnych. Są to często aplikacje natywne, czyli stworzonymi dla konkretnej platformy mobilnej, ale istnieją również aplikacje hybrydowe, które mogą działać zarówno na Androidzie, jak i iOS-ie [12] oraz PWA (ang. *Progressive Web Apps* — Progresywne Aplikacje Internetowe), które pozwalają uruchomić aplikacje na większości systemów operacyjnych oraz zapisać ją w pamięci podręcznej co umożliwia korzystanie z niej offline [13]. Aplikacje bankowe potrzebują dostępu do wielu funkcji systemu operacyjnego, aby zapewnić bezpieczeństwo i wydajność. Duża część banków znajdujących się aktualnie na rynku oferuje w swoich aplikacjach narzędzia, które wspomagają zarządzanie finansami klienta.

Banki oferują też możliwość korzystania z bankowości internetowej, która udostępnia podobne narzędzia co aplikacje natywne w przeglądarce internetowej. Pierwszy bank internetowy pojawił się na rynku polskim w październiku 1998 r. i należał do Banku Gospodarczego S.A. [23].

W ankiecie przeprowadzonej dla AcademyBank wskazano, że 83,1% osób śledzi swoje wydatki, z czego 45,3% używa do tego narzędzi cyfrowych. Jednym z rozwiązań jest korzystanie z aplikacji budżetowych. Użycie tych narzędzi zadeklarowało 20,9%

ankietowanych. Z biegiem czasu te liczby będą tylko rosnąć, ponieważ na rynku będzie pojawiać się coraz więcej narzędzi umożliwiających proste zarządzanie własnymi finansami. Niemal 80% osób korzystających z platform do zarządzania budżetem deklaruje, że korzysta z nich przynajmniej raz w tygodniu [4].

W celu analizy jakości aplikacji do zarządzania finansami osobistymi przeprowadzono badanie na grupie 301 Polaków, z których 288 przyznało, że korzysta z aplikacji do wspomagania budżetem, a tylko 13, że nie korzysta. Główne czynności, do których Polacy wykorzystują aplikacje to Kontrola budżetu domowego (88,54%), Weryfikacja wydatków z ostatniego miesiąca (86,11%), Sprawdzanie salda (48,26%) oraz planowanie wydatków na kilka miesięcy (45,83%) [29].

Dane zebrane przez stronę CoinLaw przedstawiają, że aplikacje do wspomagania budżetem domowym są najczęściej wykorzystywane w przedziale wiekowym od 27 do 42 lat (91%). Kolejna grupa to osoby w wieku od 43 do 58 lat (80%). W przedziale od 18 do 26 roku życia jest to 68% [11]. Aktualna wielkość rynku dla technologii finansowych jako usługi (ang. *Fintech as a Service*) wynosi około 470,94 miliardów dolarów i przy aktualnym tempie wzrostu może urosnąć nawet do 906,14 miliardów dolarów do 2030 roku [15].

1.1. Cel pracy

Celem pracy jest zbudowanie aplikacji do zarządzania budżetem domowym. Aplikacja ma za zadanie wspomagać użytkownika w kontroli wydatków, analizie danych finansowych oraz zarządzaniu finansami osobistymi.

1.2. Zakres pracy

Niniejsza praca została utworzona z wykorzystaniem architektury klient-serwer, która umożliwia tworzenie aplikacji webowych. Aplikacja charakteryzuje się wysoką wydajnością, niezawodnością i czytelnym interfejsem użytkownika. Strona udostępnia przyciski nawigujące do poszczególnych części interfejsu. Użytkownicy mogą dodawać transakcje dla swoich portfeli oraz kont bankowych. Transakcje dla portfeli dzielą się na kategorie, natomiast konta bankowe mają możliwość wykonywania przelewów między kontami. Podsumowanie danych finansowych przedstawiają wykresy liniowe, słupkowe i kołowe.

Warstwa serwerowa została zaimplementowana w języku Java z wykorzystaniem frameworka Spring Boot. Zastosowano język Java w wersji 21, który zapewnia obsługę logiki biznesowej, komunikację z bazą danych oraz interfejs API w technologii REST.

Warstwa prezentacji została zrealizowana przy pomocy technologii React + Vite. Język, wykorzystany po stronie frontendu to TypeScript, który wspiera statyczne typowanie. Elementy, z których zbudowany jest interfejs użytkownika, są wykorzystane z popularnego reactowego frameworka UI o nazwie Ant Design. Składa się on ze wstępnie zbudowanych komponentów, które łatwo jest wyświetlić na stronie internetowej.

Dane są przechowywane w nierelacyjnej bazie danych MongoDB, która zawiera dokumenty JSON z ważnymi informacjami, które są związane z użytkownikami aplikacji.

Aplikacja uruchamiana jest z poziomu platformy docker.

1.3. Struktura pracy

Pracę otwiera Wstęp, w którym uzasadniono istotność podejmowanego tematu. Przedstawiono w nim dane o aplikacjach wspomagających zarządzanie budżetem domowym oraz zakres funkcjonalności, jakie powinna dostarczać aplikacja.

W rozdziale drugim opisano architekturę aplikacji oraz technologie wykorzystane w procesie jej tworzenia. Zaprezentowano biblioteki oraz frameworki wraz z opisaniami ich zastosowań. Wyłuszczone, dlaczego aplikacja została zaimplementowana z wykorzystaniem danych języków programowania oraz w jaki sposób odbywa się komunikacja poszczególnych części systemu.

W rozdziale trzecim przedstawiono narzędzia, które zostały wykorzystane w procesie tworzenia oprogramowania m.in.: środowiska programistyczne, system kontroli wersji, narzędzia do konteneryzacji i testowania komunikacji.

W rozdziale czwartym skupiono się na wymaganiach, które musi spełnić projekt, na diagramach przedstawiających klasy oraz sposobie działania wybranych funkcji aplikacji. Opisano, czym są poszczególne diagramy, do czego służą oraz sposób, w jaki powinno się czytać zawarte w nich informacje.

W rozdziale piątym zaprezentowano informację o sposobie implementacji systemu. Opisano działanie poszczególnych warstw aplikacji, w jaki sposób odbywa się przepływ danych, funkcje oferowane przez system oraz komunikaty zwracane przez serwer.

W rozdziale szóstym zawarto informację, w jaki sposób użytkownik powinien poruszać się po aplikacji. Przedstawiono proces uruchamiania systemu oraz kroki, jakie należy podjąć, by efektywnie poruszać się po interfejsie użytkownika. Opisano kluczowe elementy warstwy prezentacji takie jak formularze, ustawienia, portfele użytkownika i funkcję systemu.

W rozdziale siódmym opisano efekty pracy oraz wnioski. Zawarto w nim informację, czy system spełnia założone cele oraz co osiągnął autor. Przedstawiono możliwości przyszłego rozwoju aplikacji.

2. Przegląd technologii stosowanych w aplikacjach klient-serwer

W procesie wytwarzania oprogramowania częstym problemem jest wybór odpowiednich rozwiązań, które pozwolą w najlepszy sposób osiągnąć zamierzone cele. Znajomość kompatybilnych technologii w znaczącym stopniu przyspiesza tworzenie aplikacji webowych. Projekt można podzielić na warstwę prezentacji, która odpowiada za interfejs graficzny oraz interakcję z użytkownikiem, warstwę serwową odpowiadającą za logikę biznesową aplikacji i komunikację z bazą danych oraz warstwę dostępu do danych, która odpowiada za trwałe przechowywanie rekordów i udostępnianie ich inną warstwą.

2.1. Architektura systemu

Jednym ze sposobów projektowania aplikacji sieciowych jest model klient-serwer. Klient nie posiada danych oraz funkcji, dlatego musi kontaktować się z programem posiadającym dostęp do danych oraz usług. Program ten nazywany jest serwerem.

Klient zleca serwerowi żądanie pewnej usługi, następnie serwer analizuje polecenia i wysyła odpowiedź. Model ten jest powszechnie używany w aplikacjach dostępnych za pośrednictwem internetu [21].

Przykładowe zastosowanie w aplikacji webowej

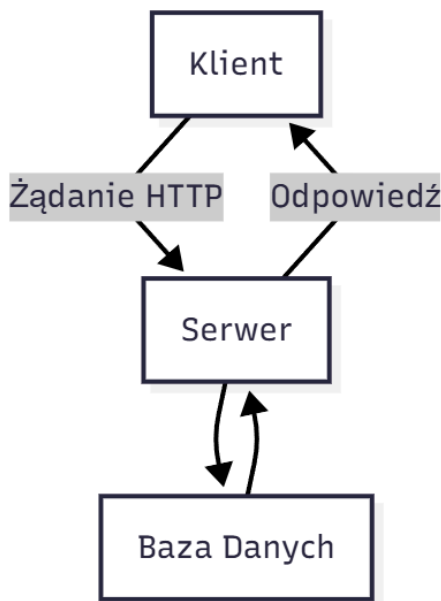
Próba logowania na stronę WWW jest równoważna z wysłaniem do serwera wiadomości z typem operacji oraz podanymi w formularzu wartościami. Serwer otrzymując taką wiadomość, wykonuje niezbędne testy, na przykład: sprawdza poprawność otrzymanych danych. Po pomyślnym dokonaniu sprawdzeń serwer wykonuje żądaną operację i wysyła odpowiedź klientowi [21].

Kody odpowiedzi HTTP

Odpowiedź serwera to numeryczna dana wysłana przez serwer, która ma na celu poinformowanie użytkownika o realizacji pewnego zadania. Kody dzielą się na pięć klas, informacyjne, które zaczynają się od 1xx, powodzenia zaczynające się od 2xx, przekierowania (3xx), błędu aplikacji klienta (4xx), błędu serwera HTTP (5xx) [14].

Tabela 2.1. Przykładowe opisy odpowiedzi serwera

Kod	Opis słowny	Znaczenie
100	Continue	Informuje, że żądanie jest kontynuowane
200	OK	Odpowiedź została zwrócona przez serwer (najczęściej zwracany nagłówek)
201	Created	Wysłany dokument został zapisany poprawnie
204	No content	Zrealizowano żądanie bez konieczności zwracania treści
300	Multiple Choices	Istnieje kilka sposobów obsłużenia danego żądania
302	Found	Zasób jest chwilowo dostępny pod innym adresem
400	Bad Request	Żądanie zawiera nieprawidłowość w postaci błędu użytkownika
401	Unauthorized	Żądany zasób wymaga uwierzytelnienia
403	Forbidden	Zapytanie zostało rozpoznane przez serwer, ale konfiguracja bezpieczeństwa zabrania zwrócić żądany zasób
404	Not Found	odnalezienie zasobu nie powiodło się na podstawie podanego adresu URL
500	Internal Server Error	Serwer napotkał niespodziewane trudności
502	Bad Gateway	Serwer pośredniczący otrzymał niepoprawną odpowiedź od serwera nadrzędnego
503	Service Unavailable	Serwer nie jest w stanie zrealizować żądania użytkownika ze względu na przeciążenia



Rysunek 2.1. Diagram przedstawiający architekturę klient-serwer

2.2. Java

Java jest obiekowym wieloplatformowym językiem programowania. Jest ona jednym z najpopularniejszych języków dla deweloperów oprogramowania [16]. Korzysta z wirtualnej maszyny Java (JVM), która może zostać zainstalowane na większości komputerów i urządzeń przenośnych. Została ona stworzona z myślą „napisz raz, uruchamiaj w dowolnym miejscu”. W projekcie jest odpowiedzialna za logikę biznesową aplikacji, przetwarzanie danych oraz komunikację z bazą danych [19].

Spring Boot

Technologia, która pomaga utworzyć szkielet projektu. Umożliwia dołączenie skonfigurowanych frameworków Javy w postaci zależności (ang. *dependencies*). Upraszcza to sposób tworzenia aplikacji w środowisku Java.

Umożliwia dołączenie kodu odpowiedzialnego za budowanie aplikacji webowych, który zawiera m.in. REST API. Wspiera również takie mechanizmy jak Wstrzykiwanie zależności (ang. *Dependency Injection*) oraz AOP (ang. *Aspect-Oriented Programming*) [25].

Maven

W projekcie wykorzystano narzędzie Apache Maven, które pełni rolę automatyzacji budowy i zarządzania zależnościami. Pozwala zdefiniować biblioteki i frameworki w jednym centralnym pliku `pom.xml` (ang. *Project Object Model*). Wykorzystuje repozytoria, w których przechowywane są biblioteki. Główny plik zawiera takie informacje jak:

- `groupId`, `artifactId`, `version` – identyfikatory projektu,
- sekcję `dependencies` – listę bibliotek, które mają zostać automatycznie pobrane,
- sekcję `build` – ustawienia dotyczące budowania aplikacji,
- sekcje `plugins` – narzędzia wspierające proces budowania.

W przypadku aplikacji opartej na Spring Boot, Maven pobiera również tzw. startery np.:

- `spring-boot-starter-web` – uruchamia aplikację serwerową (np. Tomcat) i pozwala tworzyć kontrolery REST,
- `spring-boot-starter-data-mongodb` - umożliwia korzystanie z bazy mongo,
- `spring-boot-starter-security` – dodaje mechanizmy autoryzacji i uwierzytelniania,
- `spring-boot-starter-test` – pozwala na testowanie aplikacji,
- `spring-boot-starter-validation` – obsługuje walidację encji w klasach, umożliwia m.in. ustawienie wymaganej długości łańcucha znaków lub określa czy pole może być puste.

Posiada również możliwość dodawania bibliotek oraz zestawów narzędzi jak:

- Lombok – umożliwia redukcję kodu szablonowego (ang. *boilerplate code*), poprzez automatyczne tworzenie podstawowych konstruktorów i metod,
- `spring-boot-devtools` – dostarcza narzędzia pozwalające m.in. na przeładowanie w czasie rzeczywistym.

Cała konfiguracja środowiska sprowadza się do utworzenia odpowiedniego pliku z zależnościami, a Spring Boot automatycznie na jego podstawie konfiguruje się przy uruchomieniu [2].

2.3. React

React to jedna z bibliotek JavaScript służąca do tworzenia interfejsów użytkownika (ang. *User Interface*) w aplikacjach webowych. Jest oparta na koncepcji komponentów, które można łatwo ze sobą łączyć i w czytelny sposób wyświetlać na stronie. React pozwala renderować widoki na podstawie mechanizmu Virtual DOM (ang. *Document Object Model*). Obiekt DOM umożliwia dostęp do struktury strony w celu jej modyfikacji. W przypadku modelu wirtualnego minimalizuje operacje na drzewie rzeczywistym [22].

Vite

W celu usprawnienia procesu tworzenia aplikacji zastosowano narzędzie Vite, które pełni funkcję bundlera, czyli łączy ze sobą wiele plików m.in. kody źródłowe i zależności. Rozwiązanie to oferuje szybkie uruchamianie środowiska, optymalizacja kodu oraz przeładowanie kodu na bieżąco (ang. *Hot Module Replacement*), który pozwala wyświetlać zmiany bez konieczności ponownego budowania całej aplikacji [28].

TypeScript

Komponenty oraz elementy interfejsu użytkownika są przygotowane w języku TypeScript, który jest nadzbiorem języka JavaScript. Wprowadza on statyczne typowanie. Takie podejście pozwala określać i sprawdzać typy w czasie kompilacji. Umożliwia wprowadzanie typów dla zmiennych, funkcji i obiektów. Dzięki temu kod jest bardziej czytelny i zrozumiały [26].

Połączenie Reacta, Vite oraz TypeScript zapewnia wydajne środowisko dla pracy programisty, lepszą kontrolę nad danymi, a dzięki dobraniu odpowiedniego systemu projektowania (ang. *design system*) redukuje ilość komponentów.

2.4. MongoDB

MongoDB to nierelacyjna baza typu NoSQL. W przeciwieństwie do klasycznych baz relacyjnych MongoDB nie korzysta z tabel, lecz przechowuje dane w postaci dokumentów BSON (*Binary JSON*), które strukturą przypominają obiekty JSON. Jeden dokument może mieć wiele różnych pól, co pozwala bardziej elastycznie

modelować dane. Baza ta korzysta z kolekcji (ang. *collections*), które są odpowiednikami tabel w klasycznych bazach relacyjnych. Każdy wpis w bazie posiada własny unikatowy identyfikator `_id`. Wspiera ona wiele operacji, takich jak sortowanie, filtrowanie, czy grupowanie [17].

Zaimplementowanie ich w kodzie programu wymaga odpowiednio utworzonego repozytorium wraz z operacjami (listing. 2.1). Połączenie bazy w aplikacji Java odbywa się za pomocą modułu Spring Data MongoDB. To właśnie ten moduł odpowiada za możliwość tworzenia repozytorium.

```
1 @RepositoryRestResource(collectionResourceRel = "users", path = "
   users")
2 public interface UserRepository extends MongoRepository<User,
   String> {
3     Optional<User> findByLogin(String login);
4     List<User> findByLoginContainingIgnoreCase(String loginPart);
5 }
```

Listing 2.1. Przykładowe repozytorium w Javie

Powyższy fragment kodu pozwala znaleźć użytkownika po loginie lub wyszukać wszystkich użytkowników zawierających podany ciąg znaków w swojej nazwie.

```
1 {
2     "id": "68d56ad95a544e07c8ebaa54",
3     "login": "ADMIN",
4     "password": "$2a$10$.
       itTB4jFiMBPdZSDebVE40bt18FpDiT7CHovqCtq8dcUnFMoe1gem",
5     "role": "ADMIN"
6 }
```

Listing 2.2. Przykładowy dokument z kolekcji

3. Narzędzia i metodyka pracy

W rozdziale przedstawiono zestaw narzędzi zastosowany podczas tworzenia aplikacji webowej. Opisane rozwiązania pozwoliły na efektywne tworzenie, testowanie i wdrażanie aplikacji.

3.1. Narzędzia stosowane w procesie wytwarzania aplikacji webowej

Stosowanie narzędzi usprawnia pracę programisty. Przykładowe narzędzia to: zintegrowane środowiska programistyczne (ang. *Integrated Development Enviroment* — IDE), systemy kontroli wersji, narzędzia wspomagające testowanie oraz konteneryzacje.

Środowiska programistyczne

Środowiska programistyczne to narzędzia, które umożliwiają deweloperowi edycję kodu. Jednym z najpopularniejszych rozwiązań są zintegrowane środowiska programistyczne, które składają się z edytora kodu, kompilatora lub interpretera oraz debuggera. W projekcie zostały wykorzystane takie edytory jak: IntelliJ IDEA Ultimate [7] oraz WebStorm [8]. Oba te narzędzia zostały wydane przez producenta oprogramowania JetBrains.

System kontroli wersji

Oprogramowanie służące do obsługi kontroli wersji nazywa się Git. Pozwala na tworzenie lokalnych lub zdalnych repozytoriów, do których zapisywane są zmiany w projekcie poprzez polecenie `commit`. W dowolnym momencie możemy śledzić modyfikacje plików lub powrócić do wcześniejszej wersji projektu przy pomocy komendy `rollback`. Aby przesłać zmiany do zdalnego repozytorium na stronie trzeciej (np.: github, bitbucket), należy wykorzystać opcję `push`, natomiast pobranie wymaga operacji `pull`.

Git umożliwia również pracę w grupach dzięki systemowi tworzenia gałęzi (ang. *branches*), które pozwalają na równoległą pracę całego zespołu. Zakończone części kodu mogą być scalone poleceniem `merge` z główną gałęzią [6].

Testowanie komunikacji

W celu sprawdzenia komunikacji wykorzystuje się narzędzie Postman, które umożliwia wysyłanie żądań HTTP do serwera i sprawdzanie otrzymanych odpowiedzi. Pozwala to na weryfikację punktów końcowych aplikacji. Narzędzie to daje możliwość tworzenia kolekcji z zapytaniami, które można używać wielokrotnie bez konieczności ponownego tworzenia zapytań wraz z danymi. Posiada również możliwość zapisywania ciasteczek [20].

Konteneryzacja

Możliwość pracy na różnych urządzeniach bez konieczności instalowania oprogramowania umożliwia narzędzie Docker, którego głównym elementem jest silnik (ang. *Docker Engine*) działający w formie usługi. Pozwala na budowanie, uruchamianie i zarządzanie kontenerami. Środowisko umożliwia tworzenie obrazów, na bazie których powstają izolowane środowiska do wykonywania konkretnych zadań. Narzędzie pozwala na konteneryzację zaawansowanych projektów, korzystające z wielu usług, dzięki plikowi `docker-compose.yml`, w którym definiuje się zależności między kontenerami, konfigurację sieci oraz woluminy [5].

Testowanie oprogramowania

Testy jednostkowe są wykonywane przy pomocy biblioteki JUnit w wersji 5, która oferuje możliwość tworzenia testów parametryzowanych oraz narzędzie Mockito, które umożliwia tworzenie wirtualnych repozytoriów i klas. Najważniejsze dane, które należy sprawdzać w testach to wartości graniczne, puste ciągi lub wartości null. Testy są wykonywane przed uruchomieniem aplikacji, dla wszystkich plików, których nazwa kończy się na `test`. W przypadku gdy jakiś test zakończy się niepowodzeniem, zostanie wyświetlona stosowana informacja na ten temat [9]

4. Architektura aplikacji

W rozdziale przedstawiono wymagania funkcjonalne i нефункционалне. Zilustrowano również klasy oraz funkcje programu na diagramach języka UML.

4.1. Wymagania funkcjonalne

Wymagania funkcjonalne określają zachowanie systemu i jego reakcji na określone zadania [30]. Stanowią one podstawę dla wykonawcy oprogramowania do opracowania odpowiednich metod oraz funkcjonalności. Precyzyjne zdefiniowanie wymagań z perspektywy biznesowej pozwala na dopasowanie tworzonych rozwiązań do potrzeb organizacji oraz celów strategicznych projektu. Takie rozwiązanie umożliwia wdrożenie systemu zgodnego z oczekiwaniami zleceniodawcy.

Tabela 4.1. Wymagania funkcjonalne systemu do zarządzania budżetem domowym

Nr	Wymaganie funkcjonalne	Opis
1.	Rejestracja użytkownika	System umożliwia utworzenie konta użytkownika.
2.	Logowanie do systemu	System weryfikuje dane użytkownika i przyznaje ciasteczko umożliwiające dostęp do aplikacji.
3.	Wylogowanie	System usuwa ciasteczko użytkownika.
4.	Sprawdzenie dostępnych środków	Użytkownik może sprawdzić balans w portfelach, kontach bankowych i łączny balans.
5.	Tworzenie transakcji	Użytkownik może utworzyć wpłatę lub wypłatę z konta bankowego i portfelu.
6.	Historia transakcji	Aplikacja wyświetla listę transakcji.
7.	Tworzenie przelewu	Użytkownik może przelać finanse na inne konto bankowe.

8.	Podsumowanie finansów	System wyświetla przychody i wydatki w określonym czasie.
9.	Kategoryzowanie transakcji	Użytkownik podaje kategorię dla swoich transakcji w portfelu.
10.	Prezentacja wydatków na wykresie	System wyświetla dane o transakcjach w postaci wykresów.
11.	Kantor walut	System przelicza finanse do innych walut.
12.	Uzupełnienie informacji o użytkowniku	Użytkownik może przypisać dane kontaktowe do konta.

4.2. Wymagania niefunkcjonalne

Wymaganiami niefunkcjonalnymi nazywamy wszystkie potrzeby, które nie odnoszą się bezpośrednio do funkcjonalności produktu, lecz określają jego właściwości jakościowe [30]. Dotyczą one m.in. takich zagadnień, jak: czas reakcji systemu, termin dostarczenia produktu, wyglądu interfejsu użytkownika, czy bezpieczeństwa. Stanowią kluczowy element dla wykonawcy oprogramowania, przy projektowaniu architektury, doborze technologii oraz planowaniu procesu wdrożenia. Mają istotne znaczenie z perspektywy biznesowej, ponieważ wpływają na satysfakcję użytkowników, konkurencyjność systemu oraz koszty jego utrzymania. Dla zamawiających gwarantują odpowiednią jakość, stabilność i wydajność oprogramowania.

Tabela 4.2. Wymagania niefunkcjonalne systemu do zarządzania budżetem domowym

Nr	Wymaganie niefunkcjonalne	Opis
1.	Niezawodność	System w krótkim czasie odzyskuje pełną funkcjonalność w przypadku wystąpienia błędu.
2.	Dostępność	System ma być dostępna przez 99,9% czasu działania serwera.
3.	Wydajność	System reaguje na żądanie w czasie krótszym niż 1 sekunda. Z systemu jednocześnie może korzystać więcej niż jeden użytkownik.

4.	Bezpieczeństwo	System zapewnia, że dane są dostępne tylko dla osób upoważnionych. Autoryzacja użytkowników odbywa się poprzez podanie odpowiedniego loginu oraz hasła. Hasła muszą być przechowywane w formie zaszyfrowanej.
5.	Wdrożenie	Aplikacja serwerowa musi być uruchomiona w środowisku JVM oraz korzystać z wersji Java 21, Warstwa prezentacji uruchamiana w środowisku NodeJs z wykorzystaniem 9 wersji biblioteki React. Dane przechowywane w postaci dokumentów bazy danych MongoDB w wersji 8, Aplikacja uruchomiana jest w środowisku systemu operacyjnego Windows 11 oraz Linux.

4.3. Diagramy UML

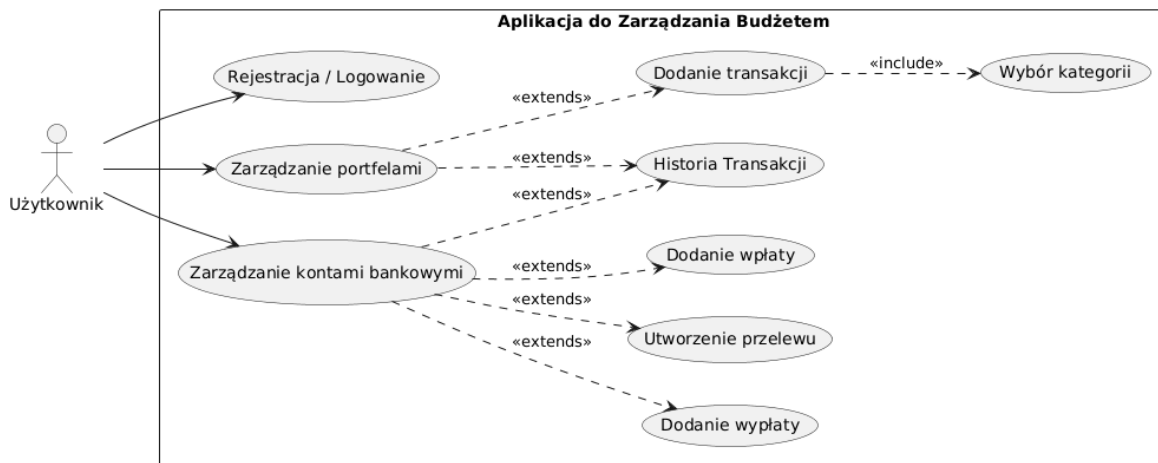
Jednym z kluczowych etapów w procesie tworzenia oprogramowania jest wizualizacja systemu [27]. Język UML (ang. *Unified Modeling Language*), czyli ujednolicony język modelowania nie jest językiem programowania, chociaż może uprościć proces tworzenia aplikacji, generując kod na podstawie diagramów. UML definiuje dwie podstawowe składowe: notację elementów na diagramach oraz ich semantykę [24].

Tworzenie diagramów

Diagramy UML można utworzyć graficznie, na przykład przez stronę draw.io, bądź tekstowo odpowiednim językiem takim jak PlantUML lub mermaid. Można też skorzystać z narzędzi takich jak enterprise architect, które posiada dodatkowe opcje, takie jak tworzenie kodu programu na bazie diagramów.

4.4. Diagramy przypadków użycia

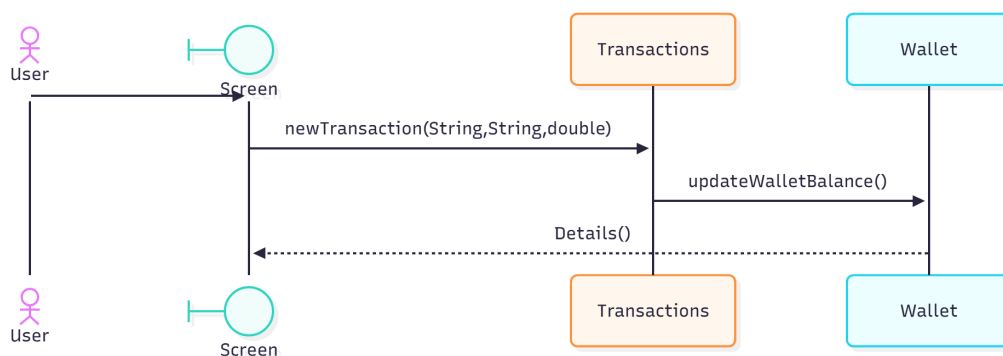
Diagram przypadków użycia (ang. *Use Case Diagram*) prezentuje usługi, które system świadczy aktorom, bez wskazywania rozwiązań technologicznych [24]. Stanowi on podstawę do modelowania szczegółowych części systemu.



Rysunek 4.1. Diagram przypadków użycia

4.5. Diagramy sekwencji

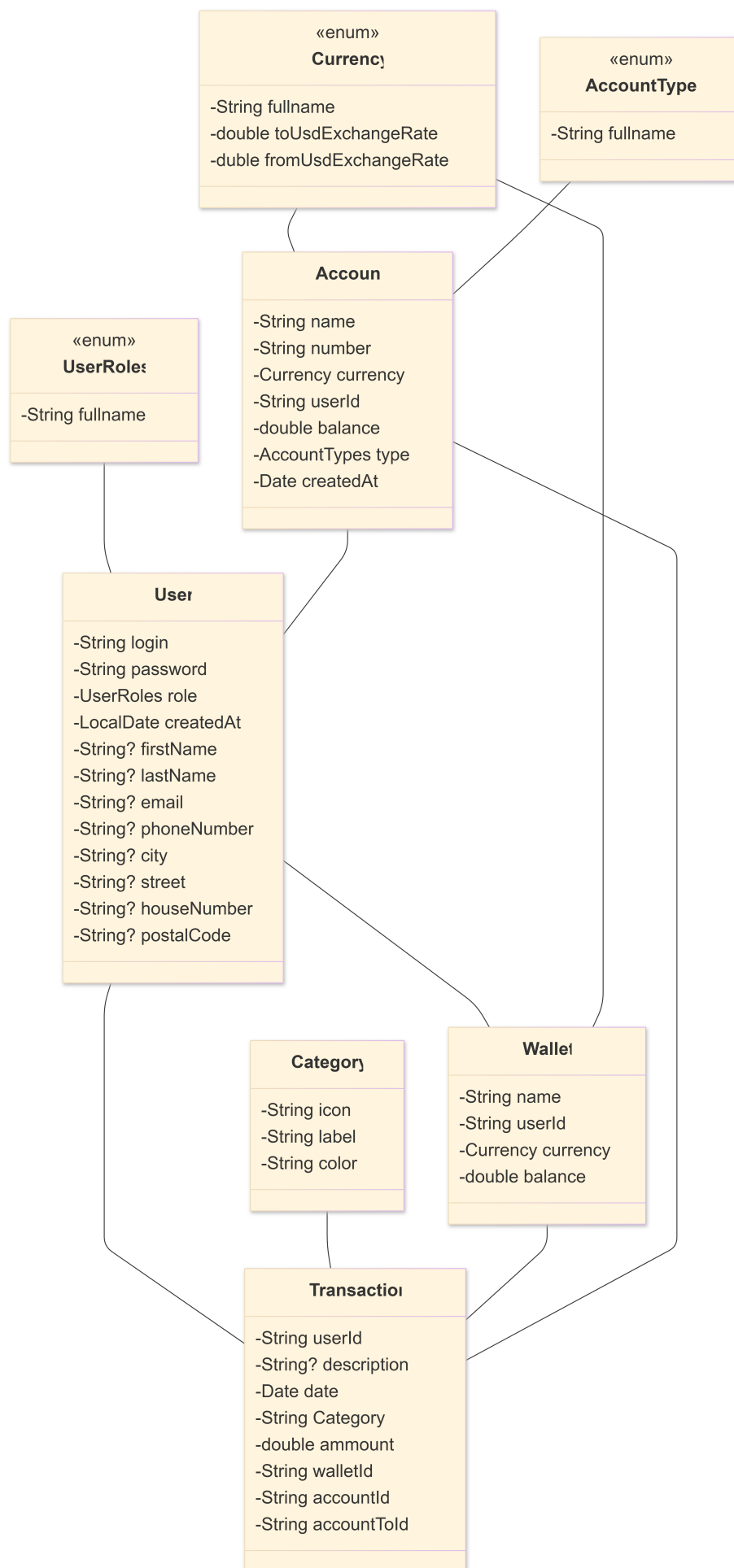
Diagram sekwencji (ang. *sequence diagram*) jest uzupełnieniem diagramu klas, który reprezentuje statyczną strukturę systemu. Diagram sekwencji w dynamiczny sposób przedstawia zachowanie klas, interfejsów oraz możliwe zastosowanie metod.



Rysunek 4.2. Diagram sekwencji - Nowa transakcja portfela

4.6. Diagram klas

Przedstawia strukturę klas oraz zależności między nimi. Składa się on



Rysunek 4.3. Diagram klas

5. Implementacja systemu

Rozdział przedstawia proces implementacji systemu. Omówiono w nim wykorzystane technologie oraz poszczególne warstwy aplikacji. Celem tego rozdziału jest zaprezentowanie działania wykorzystanej architektury oraz korelacje między poszczególnymi elementami systemu.

5.1. Backend – logika biznesowa i API

Warstwa serwerowa odpowiada za przetwarzanie żądań od warstwy prezentacji oraz za komunikację z bazą danych. Elementy, z których jest skonstruowana to m.in. kontrolery, repozytoria, czy pliki konfiguracyjne. Została zrealizowana przy pomocy frameworka Spring Boot.

Spring Boot i ekosystem bibliotek

Framework spring boot oferuje moduły oraz biblioteki wykorzystane do budowy REST API, dostępu do baz danych, bezpieczeństwa i testowania. Wykorzystywane biblioteki są deklarowane w pliku `pom.xml`. Są one wstępnie skonfigurowane. Takie rozwiązanie znacząco zwiększa tempo rozwoju aplikacji, pozwala na lepszą pracę w grupach oraz redukuje ilość kodu szablonowego (ang. *boilerplate code*).

Struktura katalogów

Projekt posiada uporządkowaną strukturę katalogów składającą się z:

- Config – pliki konfiguracyjne i zabezpieczające,
- Controller – kontrolery udostępniające endpointy i walidujące,
- Dto – modele danych do transferu (ang. *Data Transfer Object*),
- Model – opisują strukturę dokumentów,
- Repository – obsługują zapytania z bazą danych,
- Service – logika biznesowa aplikacji.

Zarządzanie zależnościami i startery

Spring boot łączy biblioteki w postaci starterów. Pozwala to zminimalizować ryzyko konfliktów wersji oraz skrócić czas konfiguracji. Kluczowe startery wykorzystywane w aplikacji webowej to:

- spring-boot-starter-web – Warstwa HTTP/REST oparta na spring MVC, która udostępnia wbudowany serwer Tomcat.
- spring-boot-starter-data-mongodb – Pozwala na dostęp do nierelacyjnej bazy MongoDB.
- spring-boot-starter-validation – Implementuje biblioteki do walidacji danych wejściowych.
- spring-boot-starter-security – Mechanizm uwierzytelniania, autoryzacji i ochrony endpointów.
- spring-boot-starter-test – Środowisko testowe do testów jednostkowych i integracyjnych.

Warstwa webowa

Umożliwia komunikację z wykorzystywaniem standardu REST API (ang. *Representational State Transfer Application Programming Interface*). W warstwie kontrolerów tworzone są endpointy na podstawie modeli, z którymi są związane. Kontroler może implementować obsługę żądań HTTP, takich jak:

- POST – Przesyłanie danych na serwer i utworzenie zasobu po stronie serwera.
- GET – Pobieranie danych z serwera.
- PUT – Aktualizowanie danych na serwerze.
- DELETE – Usuwanie danych z serwera.

Mapowanie endpointów w kontrolerze zaprezentowano w listingu 5.1.


```

1 @RestController
2
3 @PostMapping("/signup")           // Dostęp pod POST /auth/signup
4 @GetMapping("/wallet/periodTransactions") // Dostęp pod GET /
    Transaction/wallet/periodTransaction
5 @PutMapping("/wallet/updateTransaction") // Dostęp pod PUT /
    Transaction/wallet/updateTransaction
6 @DeleteMapping("/wallet/deleteTransaction") // Dostęp pod DELETE /
    Transaction/wallet/deleteTransaction

```

Listing 5.1. Przykładowe Endpointy

Implementacja powyższych endpointów przedstawiono w listingach: 5.8, 5.7, 5.9, 5.10.

Dostęp do danych

Połączenie z bazą danych zostało zrealizowane z wykorzystaniem `spring-boot-starter-data-mongodb`, który udostępnia spójny model pracy z dokumentową bazą MongoDB. Pozwala mapować obiekty klas na dokumenty i tworzyć repozytoria oparte na interfejsach.

Utworzenie klasy mapowanej do dokumentu wymaga oznaczenia adnotacją `@Document`, a klucza głównego `@Id`. Identyfikator przeważnie określa się typem `String` (listing 5.2).

Warstwa repozytoriów opiera się na interfejsach rozszerzających `MongoRepository` `<T, ID>`. Repozytoria pozwalają na podstawie nazw generować implementację metod wyszukiwujących (listing 5.3).

```

1 @Document
2 public class Account {
3     @Id
4     private String id;
5     private String name;
6     @Size(min = 25, max = 25)
7     @Indexed(unique = true)
8     private String number;
9     private String userId;

```

Listing 5.2. Fragment modelu Account

```

1 @RepositoryRestResource
2 public interface AccountRepository extends MongoRepository<Account,
    String> {
3     List<Account> findByUserId(String userId);
4     Optional<Account> findByNumber(String number);
5 }

```

Listing 5.3. Repozytorium dla klasy Account

Bezpieczeństwo

Spring Security zapewnia mechanizm uwierzytelniania JWT (ang. *JSON Web Token*) i filtrowania żądań. Token umożliwia autoryzowane połączenie z serwerem dzięki mechanizmowi ciasteczek. Token bezpieczeństwa składa się z trzech części (tab. 5.1), które zwyczajowo wyglądają w następujący sposób `xxxxx.yyyyy.zzzzz`. Filtrowanie pozwala m.in. na określenie żądań, które mogą zostać wykonane bez konieczności autoryzacji użytkownika.

Tabela 5.1. Struktura tokenu JWT [10]

Nazwa	Struktura	Opis
Nagłówek (ang. <i>Header</i>)	{ "alg": "HS256", "typ": "JWT" }	Nagłówek składa się z wykorzystanego algorytmu oraz z typu użytego tokena.
Ładunek (ang. <i>Payload</i>)	{ "sub": "1234567890", "name": "John Doe" "admin": "true" }	Ładunek zawiera roszczenia (ang. <i>claims</i>), czyli oświadczenia dotyczące podmiotu i dodatkowe dane. Typy roszczeń to: rejestrowane (ang. <i>registered</i>), publiczne (ang. <i>public</i>) i prywatne (ang. <i>private</i>).

Podpis (ang. <i>Signature</i>)	<pre> HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload) , secretPassword) </pre>	Podpis składa się z zaszyfrowanego nagłówka, ładunku i hasła, które są później przetwarzane przez wybrany algorytm.
---------------------------------	---	---

Testowanie

Narzędzia do wykonywania testów dostarcza `spring-boot-starter-test`. Testy jednostkowe wykonuje się z wykorzystaniem biblioteki JUnit, a dane do testów są tworzone przy pomocy narzędzia Mockito. Mock to obiekt klasy, od której zależy część kodu, ale w środku jest pusta. Podczas korzystania z tej pustej klasy używa się składni `when(...).thenReturn(...)`, w której `when` oznacza „Kiedy zostanie wywołana ta metoda...”, a `thenReturn` „...zwróć ten obiekt”.

Pierwszym krokiem podczas testowania jest przygotowanie danych testowych, czyli argumentów, które przyjmuje test (listing 5.4).

```

1 private static Stream<Arguments> walletTransactionsProvider() {
2     return Stream.of(
3         Arguments.of(100.0, 50.0, 150.0, "Test_Category", "
           Test_Description"),
4         Arguments.of(200.0, -20.0, 180.0, "Groceries", null),
5         Arguments.of(0.0, 10.5, 10.5, "Salary", "June Salary")
6     );
7 }

```

Listing 5.4. Strumień argumentów

Wykonywanie wielu testów dla jednej metody jest możliwe dzięki mechanizmowi testów parametryzowanych, które są opisane adnotacją `@ParametrizedTest`. Pozwalają one określić zbiór na przykład podanie metody, która zwraca dane `@MethodSource("nazwa")`. Poniższy test sprawdza, czy dodanie nowej transakcji się powiodło. Każdemu mockowi należy zdefiniować zachowanie (ang. *stubbing*), czyli opisać im reakcję na wywołanie danych metod. Gdy zachowania zostały opisane, wywoływana jest funkcja, która korzysta z tych metod. Po jej wykonaniu odbywa się sprawdzenie za pomocą asercji oraz metody `verify`, czy warunki zostały spełnione.

```

1  @ParameterizedTest
2  @MethodSource("walletTransactionsProvider")
3  void walletNewTransactionTestOK(
4      double initialBalance,
5      double amount,
6      double expectedBalance,
7      String category,
8      String description
9  )
10 //Stubbing
11 String walletId = "Test_Wallet_ID";
12 String userId = "Test_User_ID";
13 Wallet wallet = new Wallet();
14 wallet.setId(walletId);
15 wallet.setUserId(userId);
16 wallet.setBalance(initialBalance);
17 User user = new User();
18 user.setId(userId);
19 WalletNewTransactionDTO dto = new WalletNewTransactionDTO();
20 dto.setWalletId(walletId);
21 dto.setAmount(amount);
22 dto.setCategory(category);
23 dto.setDescription(description);
24 when(walletRepository.findById(walletId)).thenReturn(Optional.of(
25     wallet));
26 when(userRepository.findById(userId)).thenReturn(Optional.of(user));
27 when(transactionRepository.save(any(Transaction.class)))
28     .thenReturn(inv -> inv.getArgument(0, Transaction.class));
29 //Verifying
30 Transaction result = transactionService.walletNewTransaction(dto);
31 assertThat(wallet.getBalance()).isEqualTo(expectedBalance);
32 verify(walletService).updateWallet(wallet);
33 verify(transactionRepository).save(any(Transaction.class));
34 assertThat(result.getWalletId()).isEqualTo(walletId);
35 assertThat(result.getUserId()).isEqualTo(userId);
36 assertThat(result.getAmount()).isEqualTo(amount);
37 assertThat(result.getCategory()).isEqualTo(category);
38 assertThat(result.getDescription()).isEqualTo(description);

```

Listing 5.5. Deklaracja testu

Ważnym aspektem testów jest również sprawdzanie, czy kontroler działa poprawnie. Udostępnia on endpointy oraz przeprowadza walidację, więc sprawdzenie, czy endpoint działa oraz, czy podanie błędnych danych jest obsługiwane, może być sprawdzone tylko na poziomie kontrolera.

Do tego rodzaju testów należy uruchomić część springa. Ładowanie części programu nazywa się testem wycinkowym (ang. *Slice test*). Ładowana jest wyłącznie warstwa webowa, czyli kontrolery, walidatory i zabezpieczenia (listing 5.11).

Serwisy, a kontrolery

Podział na warstwę kontrolerów i warstwę serwisów umożliwia rozdzielenie obsługi ruchu od logiki biznesowej. Kontrolery przyjmują i walidują żądania oraz przesyłają odpowiedź (ang. *Response entity*). Serwisy zawierają logikę, czyli opisuje działanie metod. Metody te można wywołać z dowolnego miejsca, a nie tylko z przez REST API. Umożliwia to pisanie testów jednostkowych dla samej logiki bez uwzględniania protokołu HTTP. Takie rozwiązanie umożliwia też ponowne wykorzystanie kodu. Operacje na bazie danych są odizolowane od kontrolera. Odpowiada za nie serwis.

Konfiguracja i uruchamianie

Konfiguracja środowiska odbywa się w pliku `application.properties`. Definicja plików `Dockerfile` oraz `Docker-compose.yml` pozwala na spójne uruchamianie usług w różnych środowiskach.

Przykładowe fragmenty kodu

Rejestracja

Funkcja rejestracji przyjmuje jako argument przygotowany model danych do autoryzacji, który posiada poprawnie uzupełnione ciało (`String username`, `String password`). Jeżeli podane dane są poprawne, zostaje wywołana metoda w serwisie odpowiedzialna za rejestrację. Tworzy nowego użytkownika oraz szyfruje jego hasło.

```
1 // Metoda w serwisie
2 public User signup(AuthDto authDto) {
3     User user = new User()
4     .setLogin(authDto.getUsername())
5     .setPassword(passwordEncoder.encode(authDto.getPassword()))
```

```

6     .setRole(UserRoles.USER);
7     return userRepository.save(user);}
8
9 // Endpoint w kontrolerze
10 @PostMapping("/signup")
11 public ResponseEntity<User> signup(@RequestBody AuthDto authDto) {
12     User registeredUser = authenticationService.signup(authDto);
13     return ResponseEntity.ok(registeredUser);
14 }

```

Listing 5.6. Rejestracja użytkownika

Dane o transakcjach w przedziale czasu

Pobranie transakcji z zadanego przedziału czasu jest potrzebne do analizy sytuacji finansowej użytkownika. Przygotowane dane z pewnego okresu można następnie zobrazować na wykresach oraz wyświetlić zmiany balansu portfeli lub kont bankowych.

```

1 // Metoda w serwisie
2 public Collection<Transaction> periodTransactions(String walletId
3     , Date startDate, Date endDate){
4     return transactionRepository.findByWalletIdAndDateBetween(
5         walletId, startDate, endDate);
6 }
7
8 // Endpoint w kontrolerze
9 @GetMapping("/wallet/periodTransactions")
10 public ResponseEntity<List<Transaction>> getPeriodTransactions
11     (@RequestParam String walletId,
12     @RequestParam @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
13         Date startDate,
14     @RequestParam @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
15         Date endDate) {
16     Date from = startDate;
17     Date to = endDate;
18     List<Transaction> transactions = (List<Transaction>)
19         transactionService.periodTransactions(walletId, from, to);
20     return ResponseEntity.ok(transactions);
21 }

```

Listing 5.7. Wyświetlanie transakcji od do

Dane do wykresów

Otrzymane dane z pewnego okresu czasu są przygotowywane do prezentacji za pomocą mechanizmu strumieni. Strumienie przetwarzają, filtrują i dokonują operacji na zbiorze danych.

```
1 // Metoda w serwisie
2 public List<WalletBarChartDto> getBarChartData(PeriodChangeDto
    wallet){
3     Collection<Transaction> transactions = transactionRepository.
        findByWalletIdAndDateBetween(
4         wallet.getId(),
5         wallet.getFrom(),
6         wallet.getTo()
7     );
8
9     return getWalletBarChartDtos(wallet, transactions);
10 }
11 // fragment Metody pomocniczej
12 double income = dailyTransactions.stream()
13     .filter(t -> t.getAmount() > 0)
14     .mapToDouble(Transaction::getAmount)
15     .sum();
16
17 double expenses = Math.abs(dailyTransactions.stream()
18     .filter(t -> t.getAmount() < 0)
19     .mapToDouble(Transaction::getAmount)
20     .sum());
21
22 // Endpoint w kontrolerze
23 @PostMapping("/wallet/barChartData")
24 public ResponseEntity<List<WalletBarChartDto>> getBarChartData (
    @RequestBody PeriodChangeDto periodData) {
25     List<WalletBarChartDto> barData = transactionService.
        getBarChartData(periodData);
26     return ResponseEntity.ok(barData);
27 }
```

Listing 5.8. Wykorzystanie strumieni

Usuwanie transakcji

Transakcje które znalazły się w portfelu użytkownika mogą zostać usunięte przez klienta w dowolnym momencie.

```
1 // Metoda w serwisie
2 public Transaction deleteWalletTransaction(String transactionId) {
3     Transaction transaction = transactionRepository.findById(
4         transactionId).orElseThrow(()->new EntityNotFoundException("Nie
5         znaleziono transakcji"));
6     Wallet wallet = walletService.getWalletById(transaction.getWalletId(
7         ));
8     wallet.setBalance(wallet.getBalance() - (transaction.getAmount()));
9     walletService.updateWallet(wallet);
10    transactionRepository.delete(transaction);
11    return transaction;
12 }
13 // Endpoint w kontrolerze
14 @DeleteMapping("/wallet/deleteTransaction")
15 public ResponseEntity<Transaction> deleteTransaction(@RequestParam
16     String id) {
17     Transaction deletedTransaction = transactionService.
18         deleteWalletTransaction(id);
19     return ResponseEntity.ok(deletedTransaction);
20 }
```

Listing 5.9. Usuwanie transakcji z portfela

Edycja transakcji

Podanie błędnych danych w transakcji może zostać poprawione dzięki funkcji edycji transakcji.

```
1 // Metoda w serwisie
2 public Transaction updateWalletTransaction(Transaction transaction)
3     {
4     Transaction existingTransaction = transactionRepository.findById(
5         transaction.getId()).orElse(null);
6     if (existingTransaction == null) {
7         throw new RuntimeException("Transaction not found");
8     }
9 }
```



```

7   Wallet wallet = walletService.getWalletById(existingTransaction.
    getWalletId());
8   double balanceDifference = transaction.getAmount() -
    existingTransaction.getAmount();
9   wallet.setBalance(wallet.getBalance() + balanceDifference);
10  walletService.updateWallet(wallet);
11  existingTransaction.setAmount(transaction.getAmount());
12  existingTransaction.setDescription(transaction.getDescription());
13  existingTransaction.setCategory(transaction.getCategory());
14  return transactionRepository.save(existingTransaction);
15 }
16 // Endpoint w kontrolerze
17 @PutMapping("/wallet/updateTransaction")
18 public ResponseEntity<Transaction> updateTransaction(@Valid
    @RequestBody Transaction transaction){
19     Transaction updatedTransaction = transactionService.
        updateWalletTransaction(transaction);
20     return ResponseEntity.ok(updatedTransaction);
21 }

```

Listing 5.10. Edytowanie transakcji w portfelu

Slice Test

Testowanie kontrolera z wykorzystaniem narzędzia MockMvc, które pozwala testować aplikację webową oraz adnotacji @WithMockUser, która symuluje zalogowanego użytkownika.

```

1   @Autowired
2   private MockMvc mockMvc;
3
4   @MockitoBean
5   private TransactionService transactionService;
6   @ParameterizedTest
7   @ValueSource(strings = {
8       "{\"category\": null, \"amount\": 100.0, \"description\": \"
        Test transaction\"}",
9       "{\"category\": \"Groceries\", \"amount\": \"\", \"description
        \": null}",
10      "{\"category\": \"Groceries\", \"amount\": \"0\", \"description
        \": null}",

```

```

11     "{\"category\": \"Groceries\", \"amount\": null, \"description
    \": null}"
12 })
13 @WithMockUser
14 public void createTransactionNoCat(String content) throws
    Exception{
15     this.mockMvc.perform(MockMvcRequestBuilders.post("/Transaction/
        wallet/newTransaction")
16         .contentType("application/json")
17         .content(content)
18         .param("walletId", "wallet123"))
19         .andExpect(status().isBadRequest());
20 }

```

Listing 5.11. Test wycinkowy

5.2. Frontend – interfejs użytkownika

Warstwa prezentacji została zrealizowana z wykorzystaniem biblioteki React oraz narzędzie budujące Vite. Vite zapewnia szybkie uruchamianie środowiska deweloperskiego oraz prostą integrację z TypeScript. React dostarcza deklaratywny model budowy, który pozwala opisać, jak powinien wyglądać interfejsu użytkownika w danym stanie [1].

Struktura projektu

Struktura katalogów projektu została zaprojektowana w sposób pozwalający rozdzielić warstwy odpowiedzialności. Składa się z takich modułów jak:

- App – Przechowuje układ strony i trasy (ang. *routes*).
- Components – Zawiera elementy, które znajdują się na stronach.
- Models – Zawiera typy utworzone na potrzeby działania funkcji.
- Pages – Widoki stron.
- Services – Zawiera funkcję odpowiedzialne za komunikację z backendem.
- Styles – Przechowuje style w formacie .css.

Interfejs użytkownika

Warstwa prezentacji stanowi interfejs, za pośrednictwem którego użytkownik końcowy wchodzi w interakcje z systemem. To jedyna warstwa aplikacji widoczna bezpośrednio dla odbiorcy, dlatego kluczowe było zaprojektowanie jej w sposób intuicyjny i przejrzysty.

Aplikacja została zrealizowana w architekturze SPA (ang. *Single Page application*). Dzięki temu interfejs nie wymaga pełnego odświeżania w przejściach między widokami. Stałe elementy są zdefiniowane w pliku `AppLayout`, a widoki stron są dynamicznie renderowane wewnątrz kontenera `Outlet` (listing 5.12).

Elementy interfejsu użytkownika, które można wykorzystać ponownie, znajdują się w katalogu `components`. Wykorzystują gotowe komponenty z biblioteki `Ant Design`. Dzięki takiemu rozwiązaniu aplikacja jest spójna oraz optymalizuje zarządzanie kodem. Komponenty są używane w widokach (katalog `pages`), które odpowiadają za elementy wyświetlane użytkownikowi.

Aplikacja oferuje również możliwość zmiany motywu z jasnego na ciemny oraz tłumaczenie treści strony internetowej.

```
1 export function AppLayout() {
2   return (
3     <Flex gap="middle" wrap>
4       <Layout className={"app-layout"}>
5         <Navbar/>
6         <Content className={"app-layout-content"}>
7           <Outlet />
8         </Content>
9         <MyFooter/>
10      </Layout>
11    </Flex>
12  );
13 }
```

Listing 5.12. Układ strony

Komunikacja z API

Komunikacja z backendem odbywa się dzięki bibliotece `axios`, która wysyła żądania do serwera. W pliku `axios.ts` zawarta jest konfiguracja zapytań,

która definiuje adres wykorzystywany do komunikacji oraz czy wysłać uwierzytelnienie (ang. *Credentials*).

```
1 export const api = axios.create({
2   baseURL: 'http://localhost:8080',
3   withCredentials: true,
4   headers: {
5     'Content-Type': 'application/json',
6   },});
```

Listing 5.13. Konfiguracja biblioteki axios

Następnie obiekt api możemy wykorzystać do wykonywania zapytań.

```
1 export const newWalletTransaction = async (data:
   payloadNewTransaction, id: String)=>{
2   try {
3     const response = await api.post('/Transaction/wallet/
       newTransaction?walletId=${id}', data);
4     return response.data;
5   } catch (error) {
6     throw error;
7   }
8 }
```

Listing 5.14. Funkcja wyszukująca wszystkie konta użytkownika

Stan aplikacji i interakcje

Stany są jednym z kluczowych funkcji, które wykorzystuje się w aplikacjach opartych o React. Wykorzystują funkcję haków (ang. *hook*), które pozwalają używać stanów bez konieczności posiadania klasy. Do zarządzania stanami możemy użyć funkcji `useState` lub `useEffect`. Funkcja `useState` pozwala nam ustawić jakiś status podczas działania strony. Użycie efektu pozwala wykonać jakąś funkcję lub działanie i wpłynąć na to, jaki status zostanie ustawiony, na przykład wybranie daty na stronie zmieni status pola `date`. Pozwala również określić, od czego dany efekt jest zależny, w takim wypadku przy zmianie wartości zostanie ponownie wykonany.

```
1 const [ExpensesAndIncomes, setExpensesAndIncomes] = useState<any>(  
   null);  
2
```

```

3 useEffect(() => {
4   if(!wallet.id || !timeFrom || !timeTo) return;
5   const fetchPeriod = async () => {
6     try {
7       const response = await periodWalletBalance({
8         walletId: id!,
9         startDate: timeFrom!,
10        endDate: timeTo!
11      });
12      setExpensesAndIncomes(response);
13    } catch (e: any) {
14      console.error(e?.message || "Error fetching period
15        transactions");
16    }
17  };
18  fetchPeriod();
19 }, [timeFrom, timeTo, wallet.id]);

```

Listing 5.15. Wykorzystanie stanów

5.3. Baza danych - implementacja i przepływ danych

Dane przechowywane w nierelacyjnej bazie dokumentowej MongoDB. Wspiera ona wydajne zapytania pod typowe przypadki użycia aplikacji budżetowej na przykład: filtrowanie, agregacje po dacie, kategoriach lub użytkownikach.

Przykładowy przepływ danych

Dane wyświetlane użytkownikowi końcowemu muszą zostać pobrane z bazy, przetworzone przez serwer, i dopiero wtedy zostaną ukazane użytkownikowi.

1. Użytkownik wywołuje operację logowania. Parametry trafiają do API jako JSON.
2. Warstwa kontrolerów waliduje dane.
3. Backend zleca operację repozytorium.
4. Baza danych analizuje zapytanie oraz zwraca dokumenty w formacie BSON.
5. Dane są serializowane do JSON i zwracane do frontendu.

```

1 {
2   "id": "6907778371da826d8bac2eb9",
3   "name": "Portfel_Codzienny",
4   "userId": "68d56ad95a544e07c8ebaa54",
5   "currency": "PLN",
6   "balance": 1367.99
7 }

```

Listing 5.16. Przykładowy rekord z kolekcji portfeli

5.4. Integracja warstw

Integracja trzech warstw systemu została zaprojektowana tak, aby zapewnić spójny przepływ danych, określić jednoznaczną specyfikację API oraz przewidywać zachowania w środowiskach deweloperskich.

Umowa API

Podstawą integracji jest stabilna specyfikacja API [3]. Frontend korzysta z określonego punktu wejścia, a żądania i odpowiedzi wykorzystują obiekty klas w javie oraz odpowiadające im typy w TypeScript w celu komunikacji. Dane wejściowe są walidowane w kontrolerach i zapisywane bądź odczytywane z MongoDB. Takie podejście zapewnia jednoznaczność komunikacji między warstwami i ułatwia wprowadzanie zmian.

Autoryzacja i pochodzenie żądań

Mechanizm bezpieczeństwa oparto o tokeny JWT, na których podstawie określana jest tożsamość użytkownika. W środowisku deweloperskim, gdzie frontend i backend działają pod różnymi portami, zezwala się na połączenia z innych domen CORS (ang. *Cross-Origin Resource Sharing*) w kontrolowanym zakresie.

```

1 CorsConfigurationSource corsConfigurationSource() {
2   CorsConfiguration configuration = new CorsConfiguration();
3   configuration.setAllowedOrigins(List.of("http://localhost:5173"));
4   ;
5   configuration.setAllowedMethods(List.of("GET", "POST", "PUT", "
6     DELETE"));
7 }

```

```

5  configuration.setAllowedHeaders(List.of("Content-Type"));
6  configuration.setAllowCredentials(true);
7
8  UrlBasedCorsConfigurationSource source = new
    UrlBasedCorsConfigurationSource();
9  source.registerCorsConfiguration("/**", configuration);
10
11  return source;
12 }

```

Listing 5.17. Konfiguracja CORS

Obsługa błędów

Błędy, które występują w aplikacji, są obsługiwane dzięki globalnemu plikowi z obsługą błędów. Zostają tam wysyłane wyjątki i zwracane stosowne odpowiedzi oraz kody błędów. Komunikaty o wystąpieniu błędu mają ujednolicony format, co pozwala łatwo prezentować informację o błędach po stronie klienta.

Konteneryzacja i uruchamianie wieloskładnikowe

Konteneryzacja umożliwia spójne uruchamianie wielu warstw niezależnie od systemu operacyjnego. Aplikacja backendowa posiada utworzony obraz pozwalający na jej płynne uruchamianie i edycję (listing 5.18). Baza danych MongoDB działa w dedykowanym kontenerze z trwałym woluminem, w którym przechowywane są dokumenty. Połączenie kontenerów odbywa się dzięki konfiguracji zawartej w pliku `docker-compose.yml` (listing 5.19), który zawiera w sobie informację o obrazach, portach, sieciach, woluminach, zależnościach oraz zmiennych środowiskowych.

```

1 FROM maven:3.9.10-eclipse-temurin-21
2 WORKDIR /app
3 COPY pom.xml .
4 COPY src ./src
5 EXPOSE 8080
6 CMD ["mvn", "spring-boot:run"]

```

Listing 5.18. Budowanie obrazu - Dockerfile

```
1 app:           // nazwa kontenera
2   build:       // budowanie obrazu na podstawie dockerfile
3     context: .
4     dockerfile: Dockerfile
5   ports:       // określenie portów
6     - "8080:8080"
7   environment: // zmienne środowiskowe
8     DB_HOST: mongo
9     DB_PORT: 27017
10    DB_NAME: budget_management_db
11   networks:    // określenie sieci
12     - default
13   volumes:      // zadeklarowanie współdzielonego folderu
14     - ./app
15   depends_on:   // określenie zależności od innych kontenerów
16     - mongo
```

Listing 5.19. Fragment pliku konfiguracyjnego compose.yml

6. Prezentacja aplikacji

Celem tego rozdziału jest zaprezentowanie funkcjonalności w warstwie prezentacji. Interfejs użytkownika składa się z elementów, które pozwalają w płynny sposób poruszać się po aplikacji oraz dostosowywać ją do własnych potrzeb.

6.1. Środowisko deweloperskie

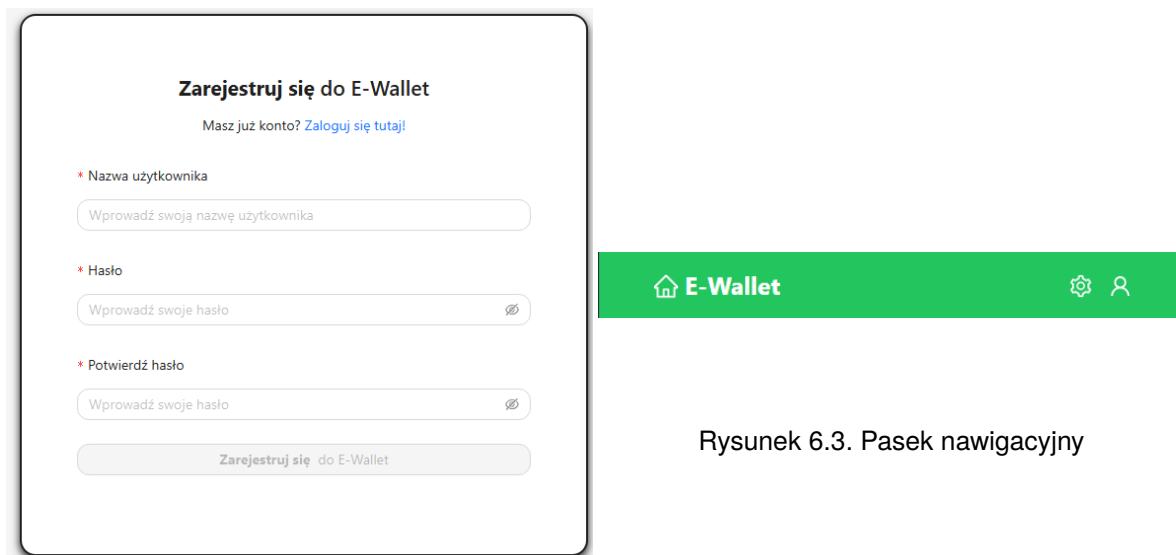
Aplikacja webowa uruchamiana jest w środowisku deweloperskim. Urządzenie, na którym uruchomiona jest aplikacja, działa jak serwer i klient jednocześnie. Odwołanie do lokalnego hosta (*localhost*) jest wymagane, by uruchomić aplikację, która znajduje się domyślnie pod adresem `localhost:5173/`, gdzie liczba po dwukropku oznacza port, z którego korzysta aplikacja. W dalszej części rozdziału adres zostanie zastąpiony słowem „HOST”.

6.2. Formularze autoryzacji

Po pierwszym załadowaniu aplikacji `www` użytkownik zostanie przekierowany przez system do formularza logowania znajdującego się pod adresem `HOST/auth/login`. Zawiera on dwa pola wymagane, w których użytkownik powinien podać nazwę użytkownika oraz hasło, i pole wyboru umożliwiające zapisanie danych (rys. 6.1). Użytkownik, który nie posiada konta, ma możliwość utworzenia go poprzez kliknięcie odnośnika *Zarejestruj się!*, który przenosi do formularza rejestracji.

The image displays two versions of a login form for an application called 'E-wallet'. The left form is in English, titled 'Login to E-wallet', and the right form is in Polish, titled 'Zaloguj się do E-Wallet'. Both forms are visually identical in layout and structure. Each form includes a header with the title and a link for users who don't have an account yet. Below this, there are two input fields: one for the username (labeled 'Username' in English and 'Nazwa użytkownika' in Polish) and one for the password (labeled 'Password' in English and 'Hasło' in Polish). Both fields have placeholder text. Below the password field is a checkbox labeled 'Remember me' (with a green checkmark icon) and a 'Login to E-wallet' button. The Polish version has a 'Zaloguj się do E-Wallet' button. A link to the registration page is provided in both languages: 'Sign Up here!' in English and 'Zarejestruj się!' in Polish.

Rysunek 6.1. Formularz logowania po polsku i angielsku

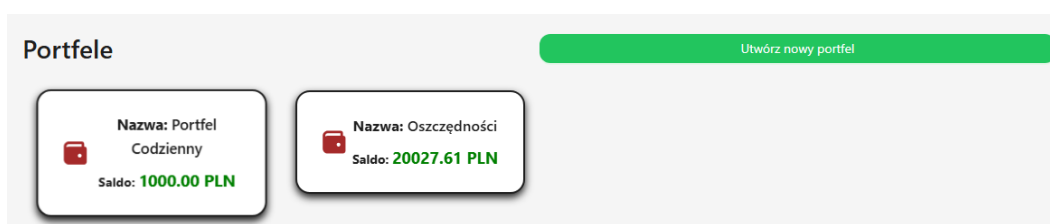


Rysunek 6.3. Pasek nawigacyjny

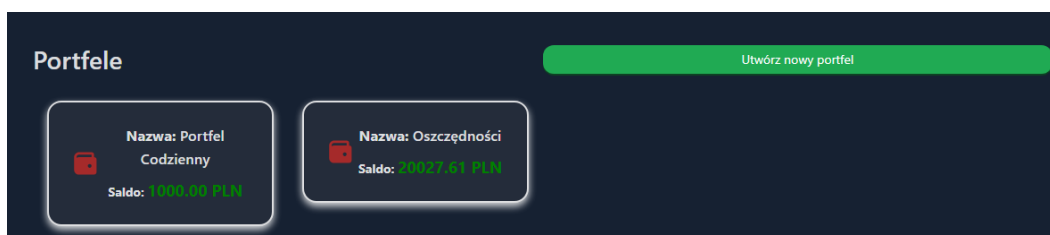
Rysunek 6.2. Formularz rejestracji

6.3. Ustawienia

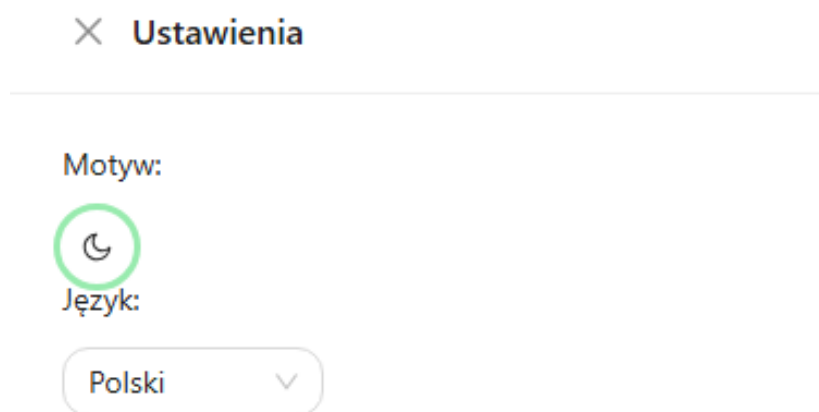
Aplikacja webowa oferuje możliwość personalizacji strony poprzez zakładkę ustawień (rys. 6.6) znajdującą się po prawej stronie na pasku nawigacyjnym (ang. *navbar*; rys. 6.3). Aplikacja pozwala na ustawienie ciemnego oraz jasnego wyglądu interfejsu użytkownika (rys. 6.4, 6.5) oraz zmianę języka na polski lub angielski (rys. 6.1).



Rysunek 6.4. Motyw Jasny



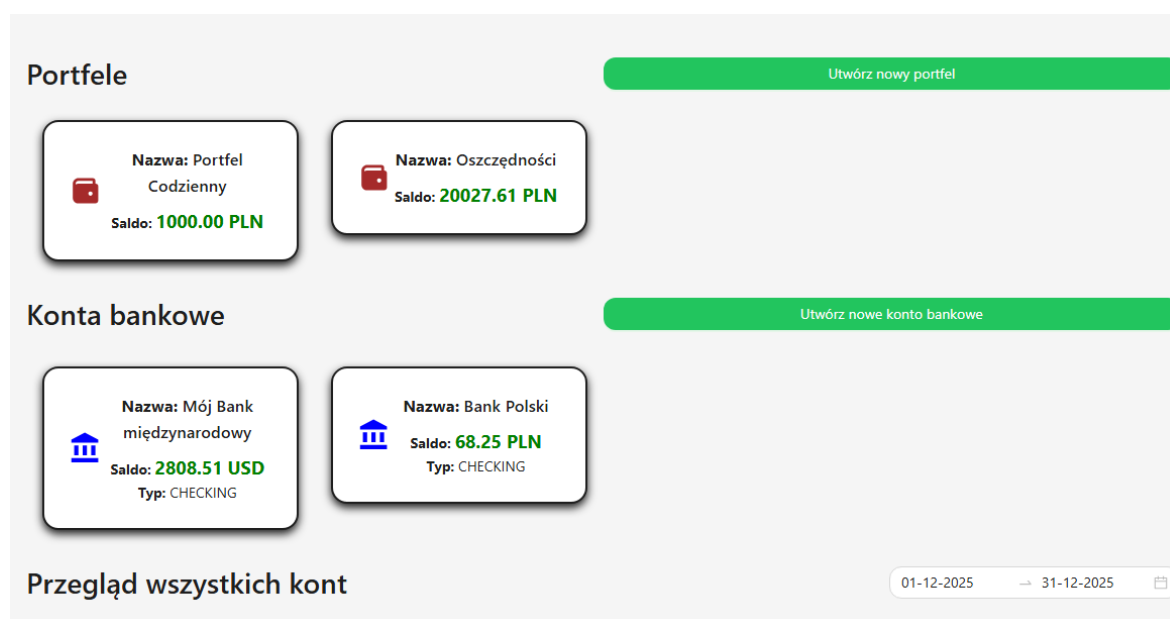
Rysunek 6.5. Motywy Ciemny



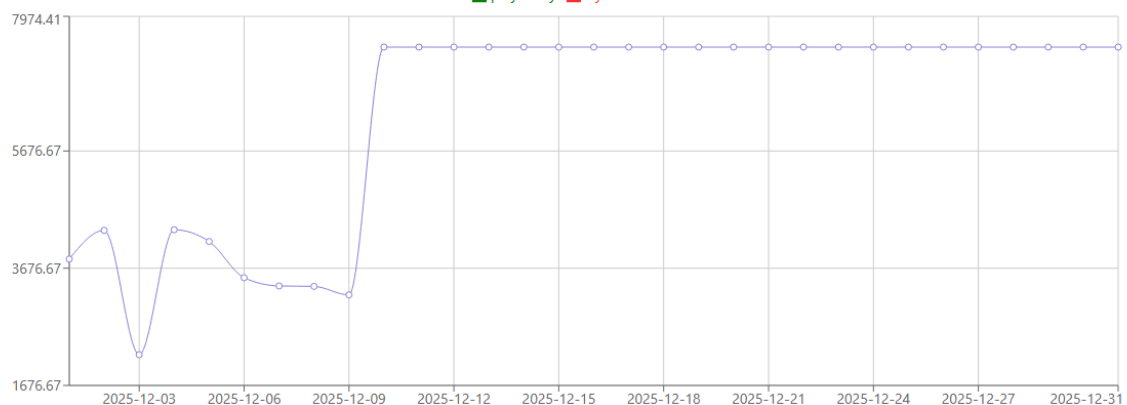
Rysunek 6.6. Ustawienia użytkownika

6.4. Funkcjonalność aplikacji udostępniona użytkownikowi zalogowanemu

Pomyślna próba logowania spowoduje przekierowanie do strony głównej (rys. 6.7, 6.8). Wyświetlone zostaną wszystkie portfele i konta bankowe użytkownika oraz podsumowanie danych na wykresach. Z poziomu strony głównej można również utworzyć nowy portfel lub dodać konto bankowe (rys. 6.9, 6.10).



Rysunek 6.7. Portfele i konta



Rysunek 6.8. Podsumowanie danych użytkownika

Rysunek 6.9. Formularz utworzenia portfela

Rysunek 6.10. Formularz utworzenia konta bankowego

Ustawienia konta użytkownika

Klient ma możliwość dodania informacji kontaktowych do konta. Podanie danych jest opcjonalne.

Ustawienia użytkownika

Imię: Jan

Nazwisko: Nowak

Email: jan.nowak@mail.com

Telefon: 123456789

Miejscowość: Warszawa

Ulica: 3 Maja

Nr domu: 13

Kod pocztowy: 00-001

Zapisz zmiany

Rysunek 6.11. Dane kontaktowe użytkownika

Portfel użytkownika

Klient zarządza swoim portfelem poprzez panel portfela. Aplikacja wyświetla takie informacje jak nazwa portfela, waluta, balans konta i podsumowanie okresowe (rys. 6.12), formularze do dodawania przychodów oraz wydatków konta (rys. 6.13), historię transakcji (rys. 6.15) i analizę finansową na wykresach liniowych, słupkowych i kołowych (rys. 6.16). Informacje są domyślnie wyświetlane z bieżącego miesiąca, jednak użytkownik może dowolnie dostosować okres, z którego chce przeprowadzić analizę wydatków. Każda transakcja posiada kategorię, która została nadana przez klienta.

Wallet			
01-12-2025 → 31-12-2025			
Dane o portfelu			
Nazwa: Oszczędności		Waluta: PLN	
Aktualne saldo portfela	Zmiana salda w okresie	Wydatki w okresie	Przychód w okresie
20027.61 PLN	787.61 PLN	-3762.39 PLN	4550.00 PLN

Rysunek 6.12. Główny panel informacyjny








Dodaj transakcję

* Kwota:

Opis:

* Kategoria: ▼

Dodaj

-  **Transport**
-  **Jedzenie**
-  **Rachunki**
-  **Zakupy**
-  **Sport i Hobby**
-  **Inne**
-  **Wypłata**

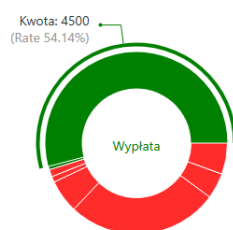
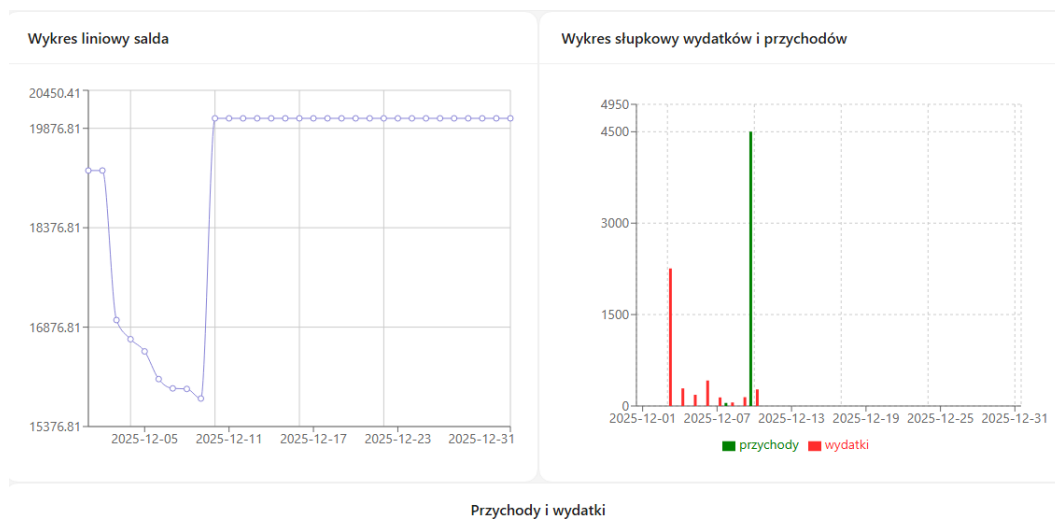
Rysunek 6.13. formularz dodawania transakcji

Rysunek 6.14. Dostępne kategorie

Historia transakcji od 01-12-2025 do 31-12-2025

Kategoria	Opis	Kwota	Data
Jedzenie	Kawa na mieście Starbucks	-22.00 PLN	2025-12-10
<div>Edytuj transakcję</div> <div>Usuń transakcję</div>			
Inne	Prezent urodzinowy	-250.00 PLN	2025-12-10
Wypłata	Wynagrodzenie za listopad	+4500.00 PLN	2025-12-10
Inne	Fryzjer męski	-70.00 PLN	2025-12-09
Transport	Przejazd Uberem	-24.50 PLN	2025-12-09
Inne	Kurs programowania Udemy	-49.99 PLN	2025-12-09
Inne	Zwrot za paliwo od znajomego	+50.00 PLN	2025-12-08
Jedzenie	Pizza na telefon	-58.00 PLN	2025-12-08
Rachunki	Rachunek za prąd	-140.50 PLN	2025-12-07
Sport i Hobby	Wyjście do kina z dziewczyną	-85.00 PLN	2025-12-06
Zakupy	Szybkie zakupy Żabka	-34.20 PLN	2025-12-06
Zakupy	Nowe słuchawki MediaExpert	-299.00 PLN	2025-12-06
Transport	Bilet miesięczny MPK	-110.00 PLN	2025-12-05
Inne	Leki przeciwbólowe Apteka	-45.90 PLN	2025-12-05
Jedzenie	Lunch w pracy - Kebab	-28.00 PLN	2025-12-05
Inne	Netflix subskrypcja	-43.00 PLN	2025-12-04
Transport	Paliwo Orlen - pełny бак	-245.80 PLN	2025-12-04
Rachunki	Czynsz za mieszkanie	-2100.00 PLN	2025-12-03
Zakupy	Zakupy spożywcze Biedronka	-156.50 PLN	2025-12-03

Rysunek 6.15. Przykładowa historia transakcji



Rysunek 6.16. Podsumowanie w formie wykresów

Konta bankowe

Strona konta bankowego różni się od portfeli brakiem możliwości kategoryzowania transakcji. Posiada natomiast możliwość tworzenia przelewów. Do utworzenia przelewu wymagana jest znajomość numeru konta odbiorcy.

Konto bankowe 01-12-2025 → 31-12-2025

Szczegóły konta bankowego

Nazwa: Mój Bank międzynarodowy **Numer:** 7167451447831507421770864 **Saldo:** 2808.51 USD **Typ:** CHECKING

Nowa transakcja

Wpłata Wyplata Przelew

* Numer konta odbiorcy:

* Kwota:

Opis:

Wykonaj przelew

Historia transakcji

Kwota	Opis	Data
-200.00 USD	Paliwo Orlen	6.12.2025
-15.99 USD	Abonament Netflix	5.12.2025
2420.00 USD	Wynagrodzenie za listopad	4.12.2025
-50.00 USD	Wyplata z bankomatu	3.12.2025
200.00 USD	Zwrot podatku	3.12.2025
15.00 USD	Za basen	3.12.2025
500.00 USD	Premia kwartalna	2.12.2025
-10.00 USD	-	2.12.2025
-120.50 USD	Zakupy spozywcz Biedronka	1.12.2025
100.00 USD	Prezent urodzinowy	1.12.2025

Rysunek 6.17. Detale konta bankowego

7. Podsumowanie i wnioski

W teraźniejszych czasach ludzie coraz częściej korzystają z portfeli elektronicznych. Pozwalają one na proste zarządzanie finansami bez konieczności wychodzenia z domu. Dzięki nim można kategoryzować swoje wydatki, przeglądać szczegółowe statystyki, oszczędzać pieniądze i usprawnić zarządzanie finansami.

Celem niniejszej pracy było zaprojektowanie i implementacja aplikacji webowej wspomagającej zarządzanie finansami osobistymi. Aplikacja została utworzona z wykorzystaniem różnych technologii i narzędzi.

System powstał i w pełni spełnia założone cele. Umożliwia dodawanie i usuwanie środków, śledzenie wydatków, kategoryzowanie i ogólne zarządzanie budżetem. Interfejs użytkownika pozwala intuicyjnie poruszać się po elementach aplikacji, a funkcje i metody opisane po stronie serwera, dają możliwość na wydajne przetwarzanie danych.

Wnioski

Nowoczesne technologie pozwalają na tworzenie skalowalnych i łatwych w rozwoju aplikacji. Narzędzia programistyczne są niezbędne w procesie tworzenia oprogramowania.

Odpowiednie przygotowanie architektury aplikacji pozwala na implementację klas, metod i funkcji, które zostały zaplanowane na początku tworzenia projektu. W procesie wytwarzania oprogramowania może okazać się, że niektóre zaplanowane rozwiązania nie będą odpowiednie dla danego systemu.

Język Java wraz z frameworkiem Spring sprawdza się doskonale przy tworzeniu aplikacji webowej. Zawiera wiele bibliotek, które czynią to rozwiązanie jednym z najlepszych.

Testowanie oprogramowania pozwala ograniczyć ilość błędów systemu.

Biblioteka React wraz z narzędziami deweloperskimi pozwala na sprawny rozwój interfejsu użytkownika. Dostępność wielu gotowych komponentów sprawia, że warstwa prezentacji jest przejrzysta i czytelna.

Środowiska deweloperskie pozwalają na szybkie sprawdzanie wprowadzanych zmian dzięki funkcji przeładowania na gorąco.

Nierelacyjne bazy danych pozwalają na dynamiczną edycję struktury kolekcji, bez konieczności edycji istniejących dokumentów.

Konteneryzacja umożliwia tworzenie bezpiecznych środowisk, w których może działać system.

Kontrola wersji aplikacji pozwala na bezpieczny rozwój projektu, dzięki udostępnionej możliwości zapisywania zmian.

Osiągnięcia autora

Podczas tworzenia projektu autor rozwinął swoje umiejętności techniczne oraz analityczne w zakresie tworzenia aplikacji webowych. Poszerzył zakres wiedzy w dziedzinie informatyki i inżynierii oprogramowania. Opracował takie zagadnienia jak:

- implementacja architektury warstwowej,
- obsługa komunikacji warstw,
- wzorce DTO,
- globalna obsługa wyjątków,
- bezpieczeństwo bezstanowe,
- implementacja tokenów JWT,
- testowanie aplikacji z wykorzystaniem mocków,
- wizualizacja danych analitycznych,
- konteneryzacja środowiska,
- modelowanie danych w bazie NoSql,
- implementacje interfejsu użytkownika z wykorzystaniem biblioteki React,
- wykorzystanie frameworka Spring,
- opracowanie diagramów UML,
- język TypeScript.

Przyszły rozwój aplikacji

W przyszłości aplikację będzie można rozszerzyć o takie funkcjonalności, jak:

- prognozowania wydatków z wykorzystaniem algorytmów uczenia maszynowego,
- dodawanie kart płatniczych,
- możliwość ustawienia awatara użytkownika,
- utworzenie raportów z danymi w formacie PDF,
- konwersja danych do formatu XML,
- rozszerzenie tłumaczeń strony,
- wsparcie dla większej liczby walut,
- integracja z istniejącymi API różnych banków.

Streszczenie

Celem niniejszej pracy jest zaprojektowanie i zaimplementowanie aplikacji wspomagającej zarządzanie budżetem domowym. System umożliwia użytkownikom tworzenie wirtualnych portfeli, kontrolowanie przychodów i wydatków oraz przeprowadzanie analizy finansowej za pomocą interaktywnych wykresów. W części teoretycznej przybliżono tematykę aplikacji bankowych oraz technologii wykorzystanych do ich tworzenia.

W pracy szczegółowo opisano proces wytwórczy oprogramowania, począwszy od analizy wymagań funkcjonalnych i нефункциональных, aż po implementację. Logikę działania systemu przedstawiono za pomocą diagramów języka UML. Projekt zrealizowano w architekturze klient-serwer, wykorzystując framework Spring Boot w warstwie backendowej, bibliotekę React w warstwie prezentacji oraz nie-relacyjną bazę danych MongoDB.

Praca zawiera również instrukcję obsługi, która przeprowadza użytkownika przez dostępne funkcjonalności systemu.

Słowa kluczowe: Bankowość, Budżet domowy, Zarządzanie finansami, Aplikacja internetowa, Inżynieria oprogramowania, Java, Rest API, Spring, React, MongoDB

Abstract

The thesis focuses on designing and implementing a home budget management application. The system allows users to create virtual wallets, track their income and expenditure, and perform financial analysis using interactive charts. The theoretical section introduces the subject of banking applications and the technologies used to develop them.

The thesis provides a detailed description of the software development process, from analysing functional and non-functional requirements to the final implementation. The system's logic is illustrated using UML diagrams. The project was implemented using a client-server architecture, with the Spring Boot framework for the backend, the React library for the frontend, and MongoDB as the non-relational database.

The thesis also includes a user manual that guides the user through the available system's functionalities.

Key words: Banking, Home budget, Financial management, Web application, Software engineering, Java, Rest API, Spring, React, MongoDB

Bibliografia

- [1] Sidorenko A. React is declarative - what does it mean? <https://alexsidorenko.com/blog/react-is-declarative-what-does-it-mean>, 2021. stan na dzień: 18.10.2025.
- [2] Apache. Apache maven documentation. <https://maven.apache.org/guides/index.html>. stan na dzień: 14.10.2025.
- [3] AppMaster. Umowa api. <https://appmaster.io/pl/glossary/umowa-api>, 2023. stan na dzień: 18.10.2025.
- [4] Academy Bank. *Banking Trends in 2025 & Beyond: Budgeting Apps for Financial Success*. <https://www.academybank.com/article/banking-trends-in-2025-and-beyond-budgeting-apps-for-financial-success>. stan na dzień: 13.10.2025.
- [5] Docker. Docker documentation. <https://docs.docker.com/>. stan na dzień: 29.12.2025.
- [6] Git. Git documentation. <https://git-scm.com/docs>. stan na dzień: 29.12.2025.
- [7] JetBrains. IntelliJ idea documentation. <https://www.jetbrains.com/help/idea/getting-started.html>. stan na dzień: 29.12.2025.
- [8] JetBrains. Webstorms documentation. <https://www.jetbrains.com/help/webstorm/getting-started-with-webstorm.html>. stan na dzień: 29.12.2025.
- [9] JUnit5. Junit5 documentation. <https://docs.junit.org/5.5.0/user-guide/index.html>. stan na dzień: 29.12.2025.
- [10] JWT. Introduction to json web tokens. <https://www.jwt.io/introduction#what-is-json-web-token>. stan na dzień: 17.10.2025.
- [11] Elad B., Kinder K. *Fintech Adoption Statistics 2025: What's Driving It (And What's Holding It Back)*. <https://coinlaw.io/fintech-adoption-statistics/>, 2025. stan na dzień: 14.10.2025.

- [12] Rożnowski K. Aplikacje natywne – czym są i jak działają? <https://semcore.pl/aplikacje-natywne-czym-sa-i-jak-dzialaja/>, 2025. stan na dzień: 28.12.2025.
- [13] Gudalewska M. Co to jest pwa (progressive web app) i jak działa? <https://auroracreation.pl/blog/co-to-pwa-i-jak-dziala/>. stan na dzień: 28.12.2025.
- [14] Mazurek M. Kody odpowiedzi http: Lista. <https://jakwybrachosting.pl/kody-http/>. stan na dzień: 24.10.2025.
- [15] markets and markets. *Fintech as a Service Market*. <https://www.marketsandmarkets.com/Market-Reports/fintech-as-a-service-market-9388805.html>, 2025. stan na dzień: 29.12.2025.
- [16] Azure Microsoft. Co to jest java? <https://azure.microsoft.com/pl-pl/resources/cloud-computing-dictionary/what-is-java-programming-language#:~:text=Jak%20dzia%C5%82a%20j%C4%99zyk%20Java?%20Kod%20Java%20jest,i%20udost%C4%99pnia%20%C5%9Brodowisko%20uruchomieniowe%20dla%20aplikacji%20Java>. stan na dzień: 14.10.2025.
- [17] MongoDB. MongoDB documentation. <https://www.mongodb.com/docs/>. stan na dzień: 29.12.2025.
- [18] Abgilas D. Aishwarya D. Lipsa D. Siddharth M. Supriyo Ghose Nitin B. Robo-advisory: An investor's perception. *International Journal of Psychosocial Rehabilitation*, 23(3), 2019.
- [19] Oracle. *JDK 21 Documentation*. <https://docs.oracle.com/en/java/javase/21/>. stan na dzień: 14.10.2025.
- [20] Postman. Postman documentation. <https://learning.postman.com/docs/introduction/overview/>. stan na dzień: 29.12.2025.
- [21] Majkowski A., Rak R. *Systemy pomiarowe*, pages 35–37. Ośrodek kształcenia na Odległość OKNO, 2018.

- [22] React. React documentation. <https://react.dev/learn>. stan na dzień: 29.12.2025.
- [23] Parys T. Bankowość internetowa jako nowa forma świadczenia usług bankowych. *Wydział Zarządzania Uniwersytet Warszawski*, 2016.
- [24] Sobestiańczyk T. Standard uml 2.3 w zarządzaniu wytwarzaniem oprogramowania. *Uniwersytet Jana Kochanowskiego w Kielcach*, 2012.
- [25] VMware Tanzu. Spring boot documentation. <https://docs.spring.io/spring-boot/documentation.html>. stan na dzień: 14.10.2025.
- [26] TypeScriptLang. Typescript documentation. <https://www.typescriptlang.org/docs/handbook/jsdoc-supported-types.html#documentation>. stan na dzień: 29.12.2025.
- [27] Sinan Si Alhir, (tłum.) Jarczyk A. *UML, Wprowadzenie*. Helion, 2003.
- [28] Vite. Vite documentation. <https://vite.dev/guide/>. stan na dzień: 29.12.2025.
- [29] Warchlewska A. Waliszewski K. Aplikacje pfm jako fintech w finansach osobistych. In Perez K., editor, *Innowacje finansowe w gospodarce 4.0*, pages 120–140. Uniwersytet Ekonomiczny w Poznaniu, 1st edition, 2021. <https://doi.org/10.18559/978-83-8211-083-8>.
- [30] Łuczak K. Identyfikowanie wymagań użytkowników jako podstawa projektowania user experience. *Uniwersytet Ekonomiczny we wrocławiu*, 2022.

Spis rysunków

2.1	Diagram przedstawiający architekturę klient-serwer	11
4.1	Diagram przypadków użycia	20
4.2	Diagram sekwencji - Nowa transakcja portfela	20
4.3	Diagram klas	21
6.1	Formularz logowania po polsku i angielsku	41
6.2	Formularz rejestracji	42
6.3	Pasek nawigacyjny	42
6.4	Motyw Jasny	42
6.5	Motywy Ciemny	42
6.6	Ustawienia użytkownika	43
6.7	Portfele i konta	43
6.8	Podsumowanie danych użytkownika	44
6.9	Formularz utworzenia portfela	44
6.10	Formularz utworzenia konta bankowego	44
6.11	Dane kontaktowe użytkownika	45
6.12	Główny panel informacyjny	45
6.13	formularz dodawania transakcji	46
6.14	Dostępne kategorie	46
6.15	Przykładowa historia transakcji	46
6.16	Podsumowanie w formie wykresów	47
6.17	Detale konta bankowego	47

Spis tabel

2.1	Przykładowe opisy odpowiedzi serwera	10
4.1	Wymagania funkcjonalne systemu do zarządzania budżetem domowym	17
4.2	Wymagania niefunkcjonalne systemu do zarządzania budżetem domo- wym	18
5.1	Struktura tokenu JWT [10]	26

Spis listingów

2.1	Przykładowe repozytorium w Javie	14
2.2	Przykładowy dokument z kolekcji	14
5.1	Przykładowe Endpointy	25
5.2	Fragment modelu Account	25
5.3	Repozytorium dla klasy Account	26
5.4	Strumień argumentów	27
5.5	Deklaracja testu	28
5.6	Rejestracja użytkownika	29
5.7	Wyświetlanie transakcji od do	30
5.8	Wykorzystanie strumieni	31
5.9	Usuwanie transakcji z portfela	32
5.10	Edytowanie transakcji w portfelu	32
5.11	Test wycinkowy	33
5.12	Układ strony	35
5.13	Konfiguracja biblioteki axios	36
5.14	Funkcja wyszukująca wszystkie konta użytkownika	36
5.15	Wykorzystanie stanów	36
5.16	Przykładowy rekord z kolekcji portfeli	38
5.17	Konfiguracja CORS	38
5.18	Budowanie obrazu - Dockerfile	39
5.19	Fragment pliku konfiguracyjnego compose.yml	40