Spis treści

1 Wprowadzenie – funkcjonalność systemu

System PersonalFinanceManager to aplikacja webowa typu full stack, która umożliwia użytkownikowi kompleksowe zarządzanie finansami osobistymi. Składa się z trzech głównych komponentów działających w odseparowanych kontenerach Docker: backendu w technologii .NET (WebAPI), frontendowego interfejsu użytkownika stworzonego w Node.js z wykorzystaniem ExpressJS oraz bazy danych Microsoft SQL Server. Dzięki tej architekturze system można łatwo uruchamiać i przenosić pomiędzy różnymi środowiskami, a poszczególne elementy są od siebie niezależne, co sprzyja utrzymaniu i dalszemu rozwojowi aplikacji.

Głównym zadaniem systemu jest umożliwienie użytkownikowi śledzenia swojego portfela — zarówno w kontekście aktualnych zasobów, jak i historii wydatków, wpływów oraz zmian kursów walut. Użytkownik może tworzyć konta finansowe (np. gotówka, konto bankowe, karta kredytowa), przypisywać do nich kategorie (np. „żywność", „transport", „rozrywka") oraz rejestrować transakcje: zarówno wpływy, jak i wydatki. Wszystkie dane są przechowywane w bazie SQL, co pozwala na wydajne filtrowanie i agregowanie informacji, na przykład do generowania raportów.

System udostępnia funkcję przeliczania walut na podstawie kursów pobieranych z zewnętrznych API. Użytkownik może podać walutę źródłową, walutę docelową oraz kwotę — a aplikacja nie tylko obliczy wartość końcową, lecz także zwróci źródło danych (np. NBP lub inne API), datę kursu oraz kurs zastosowany do przeliczenia. Wspierane są zarówno kursy aktualne, jak i archiwalne, co może mieć duże znaczenie przy analizie historycznej sytuacji finansowej użytkownika.

Dzięki mechanizmowi uwierzytelniania JWT (JSON Web Token), system zapewnia bezpieczny dostęp do danych. Każdy użytkownik musi się zalogować, by korzystać z funkcjonalności aplikacji — a jego token autoryzuje dostęp do konkretnych zasobów. Zabezpieczenia te pozwalają na obsługę wielu kont użytkowników i prywatność danych. Dzięki zintegrowanemu Swaggerowi możliwe jest testowanie i eksploracja API przez programistów i testerów bez konieczności pisania klienta HTTP.

Na poziomie interfejsu użytkownika aplikacja oferuje czytelny przegląd kont, sald oraz historii transakcji. Użytkownik ma możliwość dodawania nowych operacji finansowych i śledzenia ich w czasie, z przypisaniem do wcześniej zdefiniowanych kategorii. Choć obecna wersja interfejsu nie wykorzystuje jeszcze wszystkich dostępnych funkcji udostępnionych przez backendowe API, stanowi solidną podstawę do dalszego rozwoju. Jednocześnie dobrze obrazuje potencjał systemu oraz kierunki, w jakich może być on rozbudowywany w przyszłości.

System został przygotowany z myślą o dalszej rozbudowie. Dzięki oddzieleniu warstw frontend, backend i bazy danych, można go łatwo rozwijać o dodatkowe funkcjonalności, takie jak budżetowanie, powiadomienia, integrację z bankami czy eksport danych do plików. Ponadto wykorzystanie technologii kontenerowych pozwala na łatwe wdrożenia – zarówno w środowiskach lokalnych, jak i chmurowych.

Podsumowując, PersonalFinanceManager to narzędzie, które łączy w sobie prostotę obsługi z elastyczną, skalowalną architekturą. Umożliwia użytkownikom świadome zarządzanie finansami osobistymi, oferując nie tylko możliwość rejestrowania danych, ale też ich przetwarzania, analizowania i przeliczania – w bezpiecznym i przemyślanym środowisku.

2 Architektura systemu

System został zaprojektowany jako aplikacja wielowarstwowa, uruchamiana w trzech oddzielnych kontenerach: backendowego API w technologii .NET, bazy danych MS SQL Server oraz frontendowego interfejsu użyt-

kownika zbudowanego w Node.js i Express.js. Kontenery te są połączone wewnętrzną siecią Docker, co umożliwia ich bezpośrednią komunikację bez konieczności wystawiania usług na zewnątrz. Cały zestaw można uruchomić praktycznie jednym poleceniem za pomocą Docker Compose, co upraszcza zarówno proces wdrażania, jak i testowania aplikacji w środowiskach deweloperskich i produkcyjnych. Taka architektura zapewnia przejrzystość, modularność i łatwą skalowalność projektu.

2.1 Baza danych Microsoft SQL

Pierwszym z kontenerów wchodzących w skład systemu jest baza danych oparta na silniku Microsoft SQL Server. Jest to relacyjna baza danych, dobrze znana z niezawodności, wysokiej wydajności oraz pełnego wsparcia dla zaawansowanych zapytań SQL, transakcji i relacji między tabelami. W projekcie baza została przygotowana z wykorzystaniem plików skryptów SQL, które definiują strukturę danych – tabele, klucze główne i obce, indeksy oraz inne niezbędne elementy schematu. Kontener z bazą uruchamia się jako pierwszy, ponieważ pozostałe elementy systemu, a w szczególności backendowe API, wymagają aktywnego i dostępnego połączenia z serwerem bazy danych. Dzięki temu zapewniona jest spójność danych i poprawna inicjalizacja serwisów zależnych od kontekstu bazodanowego.

2.2 API

Drugim i zarazem kluczowym kontenerem w architekturze systemu jest kontener z backendowym API, zbudowanym w technologii .NET. To właśnie ten komponent odpowiada za całą logikę biznesową aplikacji oraz pełni rolę pośrednika pomiędzy bazą danych a interfejsem użytkownika. API zostało zaprojektowane zgodnie z architekturą REST, a jego struktura umożliwia łatwe rozszerzanie i utrzymywanie w przyszłości.

Po uruchomieniu, kontener API nawiązuje połączenie z bazą danych Microsoft SQL, wykorzystując zdefiniowany w konfiguracji connection string. Dopiero po uzyskaniu dostępu do aktywnej bazy danych możliwa jest poprawna inicjalizacja wszystkich usług — takich jak zarządzanie użytkownikami, kontami, kategoriami czy transakcjami. Aplikacja wykorzystuje mechanizm wstrzykiwania zależności (Dependency Injection), który pozwala na łatwe podmienianie implementacji poszczególnych serwisów, na przykład podczas testowania lub integracji z zewnętrznymi API.

Ważnym elementem tej warstwy jest także autentykacja i autoryzacja użytkowników z wykorzystaniem tokenów JWT. API weryfikuje tożsamość użytkowników i udostępnia zasoby jedynie uprawnionym osobom, zgodnie z zasadami bezpieczeństwa stosowanymi w nowoczesnych aplikacjach sieciowych. Dodatkowo, w projekcie wykorzystano narzędzie Swagger, które automatycznie generuje dokumentację interfejsu API i umożliwia jego testowanie z poziomu przeglądarki.

Kontener API został również przygotowany z myślą o współpracy z serwisami zewnętrznymi – do pobierania danych o kursach walut uruchamiane są osobne usługi HTTP, które komunikują się z zewnętrznymi źródłami, jednocześnie buforując dane w pamięci podręcznej, by nie przekraczać limitów narzuconych przez dostawców. Całość działa na serwerze Kestrel, który został skonfigurowany do nasłuchiwania zarówno na porcie 80, jak i 443, dzięki czemu aplikacja może być łatwo dostosowana do pracy z HTTPS i wystawienia na zewnątrz w środowisku produkcyjnym.

2.2.1 Endpointy

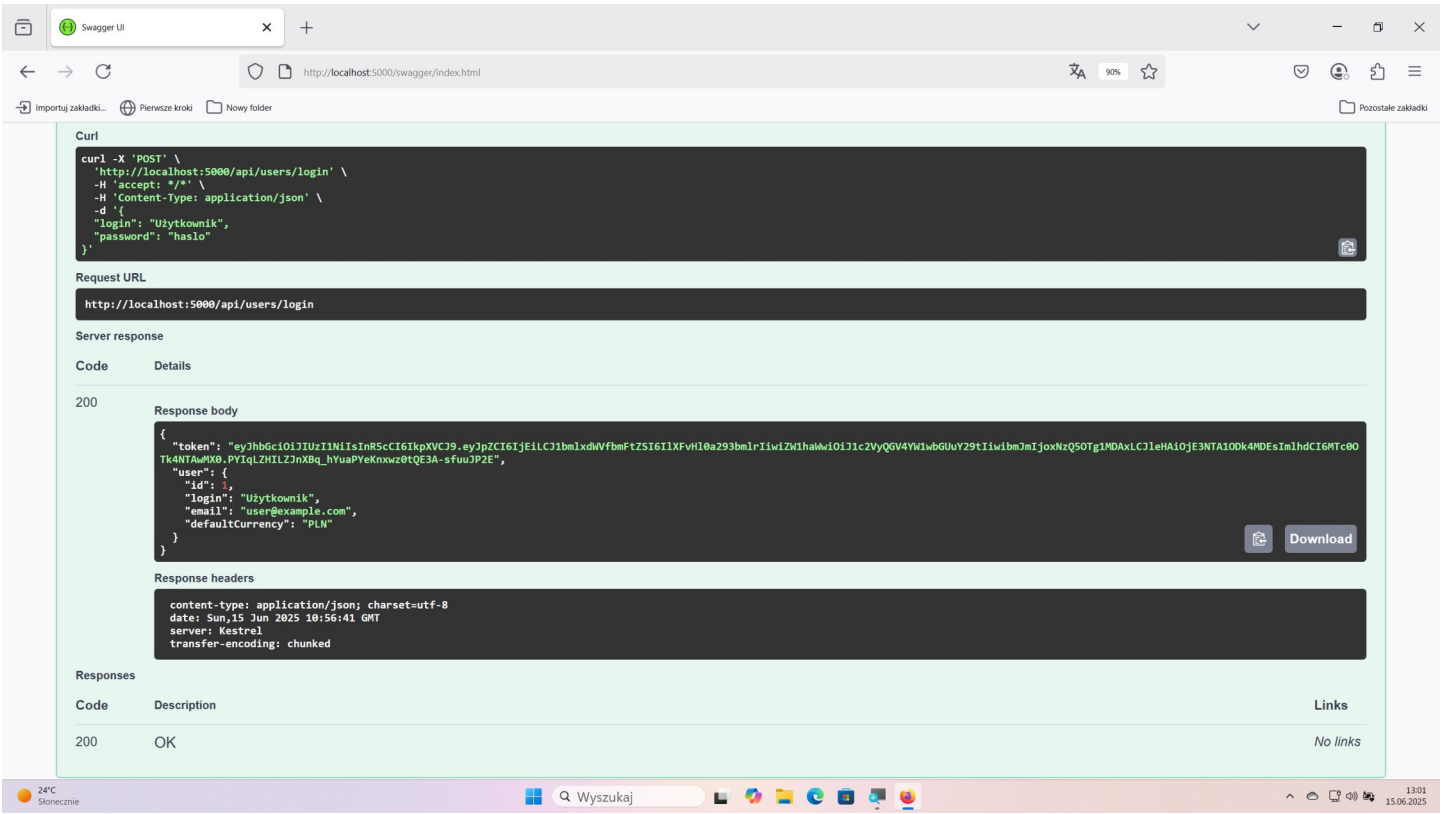
Poniższa tabela przedstawia endpointy, z których można korzystać w ramach funkcjonalności aplikacji.

Metoda	Ścieżka	Funkcjonalność
GET	/api/accounts	Pobierz wszystkie konta
POST	/api/accounts	Utwórz nowe konto
GET	/api/accounts/{id}	Pobierz konto po ID
PUT	/api/accounts/{id}	Aktualizuj konto
DELETE	/api/accounts/{id}	Usuń konto
POST	/api/accounts/{accountId}/ permissions	Dodaj uprawnienie do konta
DELETE	/api/accounts/{accountId}/ permissions/{appUserId}	Usuń uprawnienie z konta
GET	/api/categories	Pobierz wszystkie kategorie
POST	/api/categories	Utwórz nową kategorię
GET	/api/categories/{id}	Pobierz kategorię po ID
PUT	/api/categories/{id}	Aktualizuj kategorię
DELETE	/api/categories/{id}	Usuń kategorię
GET	/api/rates/Historical	Pobierz historyczne kursy walut
GET	/api/rates/current	Pobierz bieżące kursy walut
GET	/api/accounts/{accountId}/ transactions	Pobierz transakcje dla konta
POST	/api/accounts/{accountId}/ transactions	Dodaj transakcję do konta
GET	/api/accounts/{accountId}/ transactions/{transactionId}	Pobierz transakcję po ID
PUT	/api/accounts/{accountId}/ transactions/{transactionId}	Aktualizuj transakcję
DELETE	/api/accounts/{accountId}/ transactions/{transactionId}	Usuń transakcję
POST	/api/users/register	Rejestracja użytkownika
POST	/api/users/login	Logowanie użytkownika

Metoda	Ścieżka	Funkcjonalność
GET	/api/users/me	Pobierz dane zalogowanego użytkownika
PUT	/api/users/me	Aktualizuj dane zalogowanego użytkownika
DELETE	/api/users/me	Usuń konto użytkownika

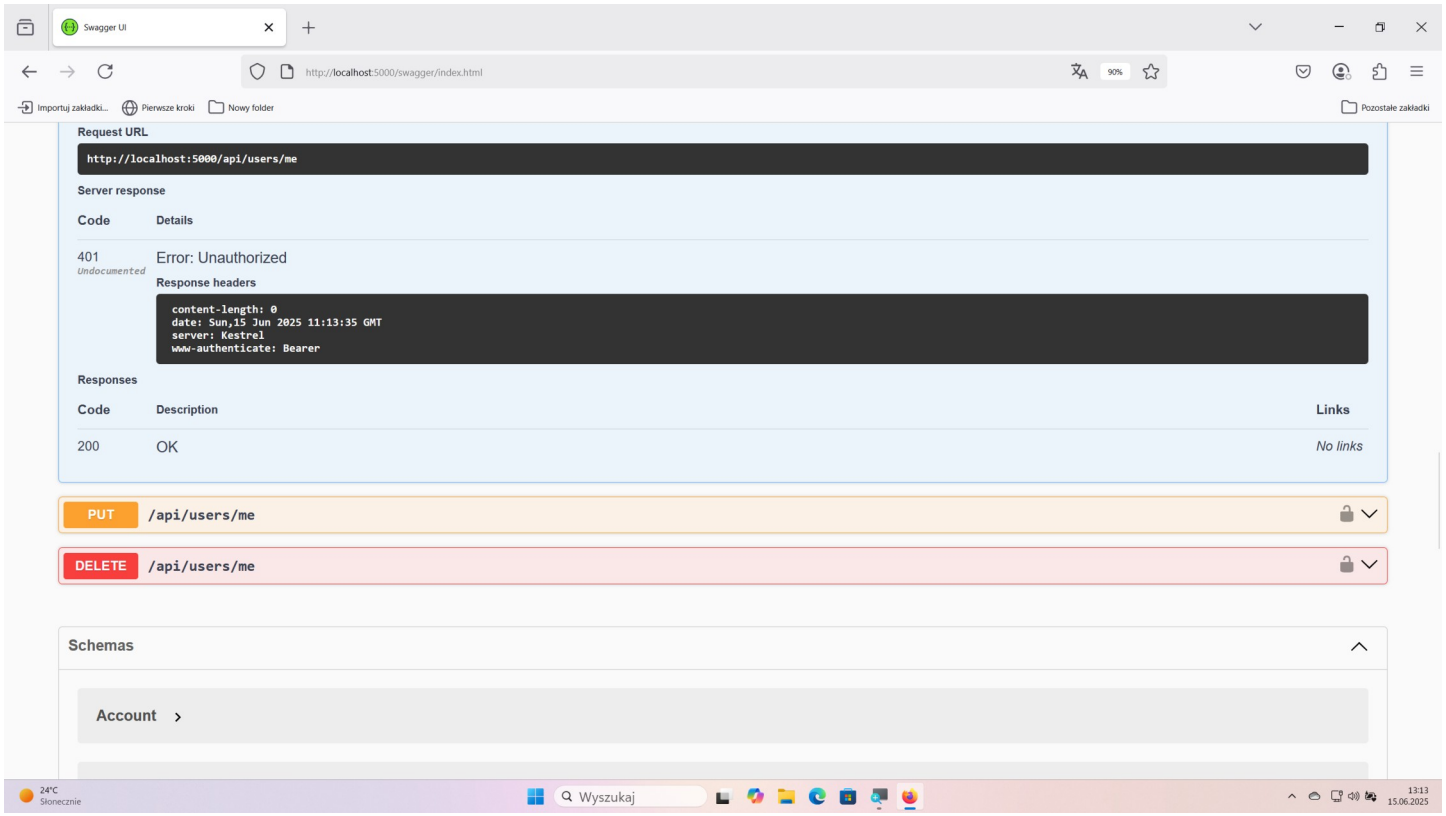
2.2.2 Autoryzacja i autentykacja

W aplikacji zastosowano mechanizm autentykacji oparty na tokenach JWT (JSON Web Token), który pozwala na bezpieczne uwierzytelnianie użytkowników w środowisku bezstanowym. Po zalogowaniu użytkownik otrzymuje token JWT, który zawiera zaszyfrowane informacje identyfikujące go w systemie. Token ten jest przesyłany przez klienta w nagłówku Authorization przy każdym kolejnym żądaniu do chronionych zasobów API. Dzięki temu serwer może weryfikować tożsamość użytkownika bez konieczności utrzymywania sesji po stronie backendu.



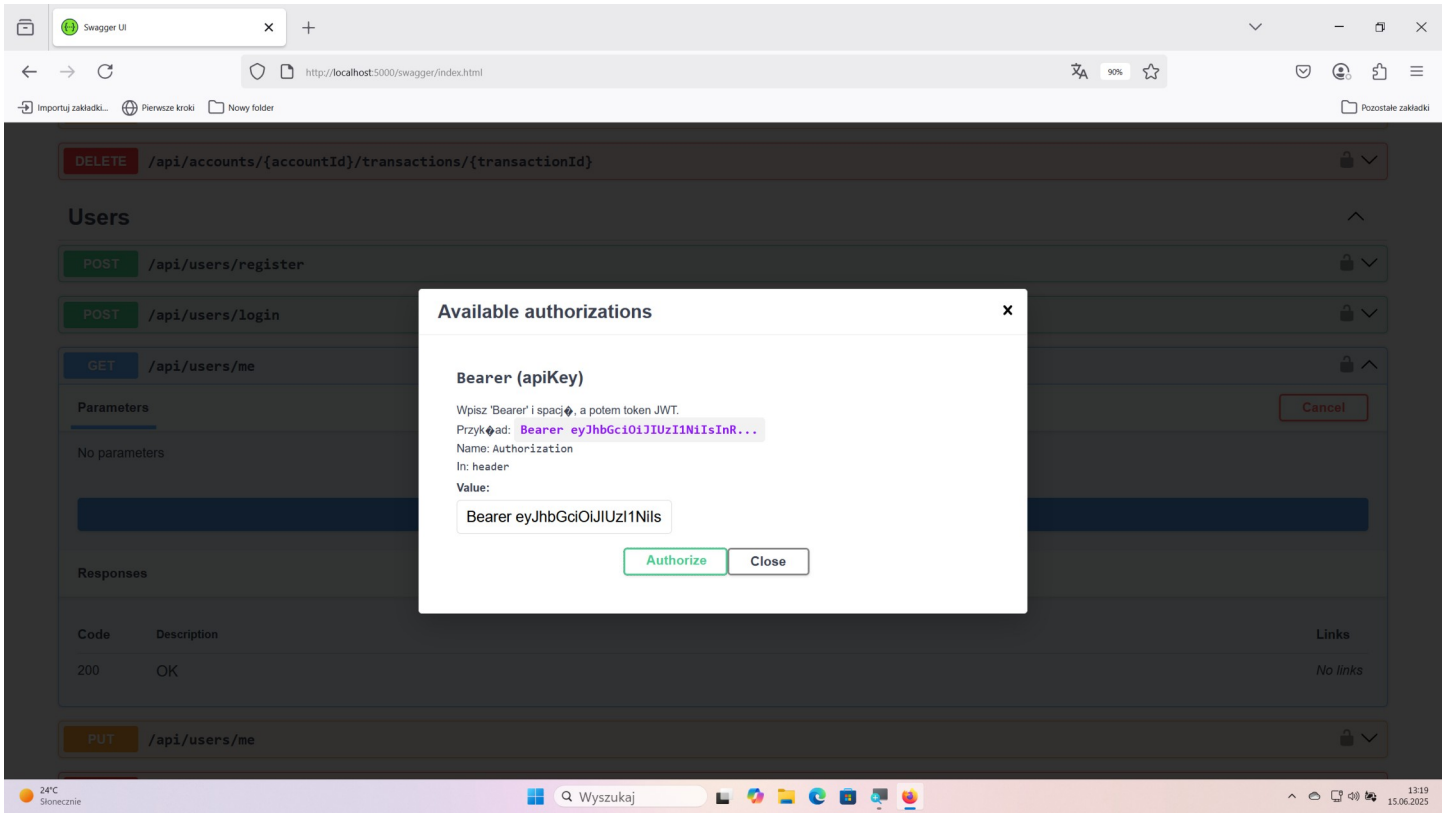
Screenshot 1: Użytkownik zalogował się. W odpowiedzi otrzymał wygenerowany przez serwer token JWT.

Autentykacja została skonfigurowana przy użyciu biblioteki JwtBearer z pakietu Microsoft.AspNetCore.Authentication.JwtBearer. W konfiguracji określono, że aplikacja nie weryfikuje wystawcy (Issuer) ani odbiorcy (Audience) tokena, co upraszcza proces walidacji, jednak nadal aktywnie weryfikowany jest klucz podpisujący (IssuerSigningKey) oraz czas życia tokena (ValidateLifetime). Klucz symetryczny używany do walidacji podpisu JWT jest pobierany z konfiguracji aplikacji i kodowany przy użyciu UTF-8.



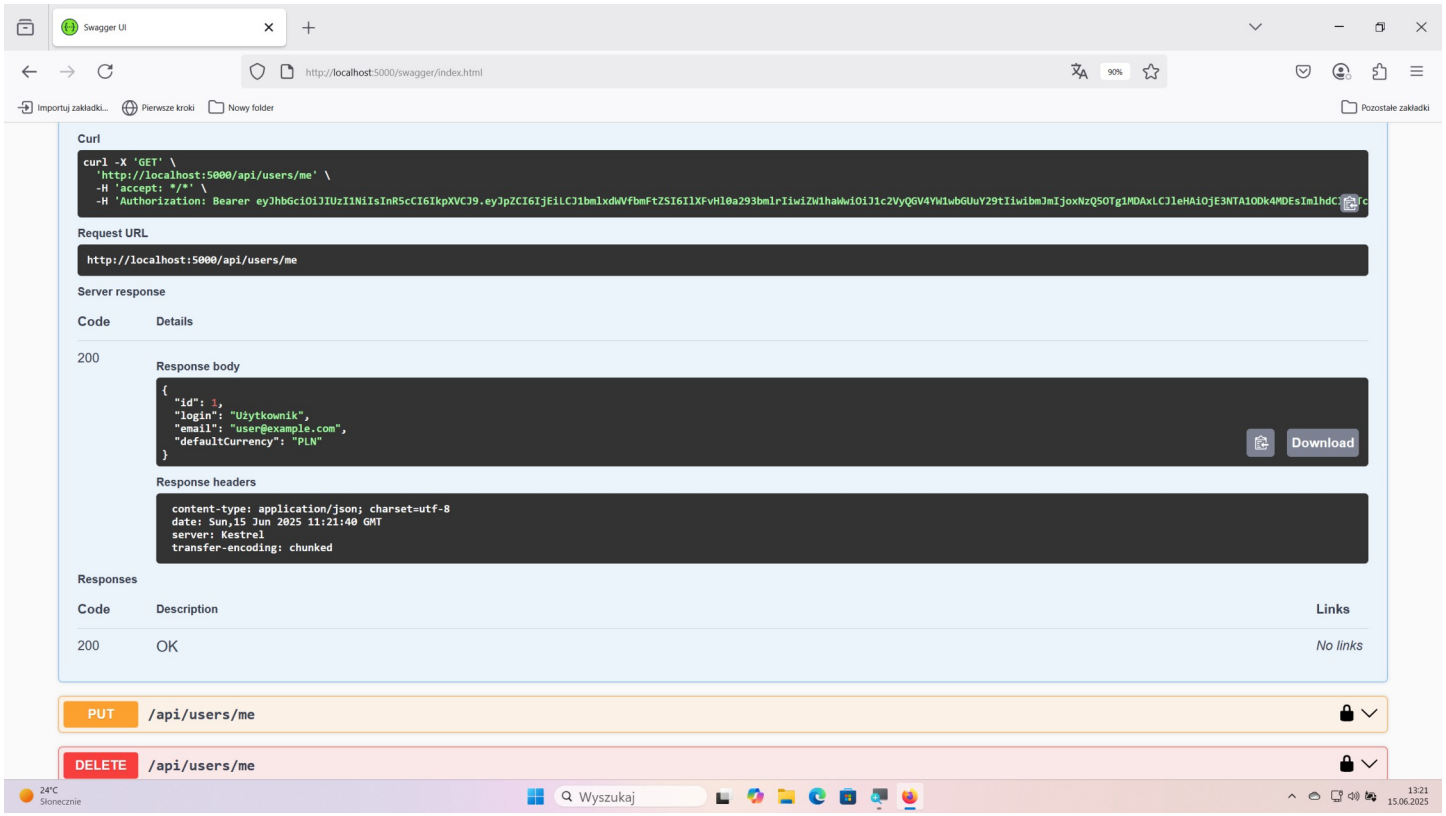
Screenshot 2: Niezalogowany użytkownik otrzymuje odpowiedź "negatywną" serwera.

Autoryzacja działa na podstawie polityki domyślnej, która wymaga, by wszystkie żądania do chronionych zasobów były poprzedzone poprawnie podpisanym tokenem JWT. Jeśli klient nie dołączy tokena lub przekaże nieprawidłowy, otrzyma odpowiedź z kodem 401 Unauthorized. Dzięki zastosowaniu atrybutu [Authorize] w kontrolerach lub konkretnych akcjach, programiści mogą precyzyjnie kontrolować dostęp do poszczególnych części API w zależności od tego, czy użytkownik jest zalogowany. Dodatkowo, możliwe jest rozszerzenie autoryzacji o role i polityki dostępu, jeśli zajdzie taka potrzeba.



Screenshot 3: Dodawanie tokenu Bearer do nagłówków zapytań w swaggerze

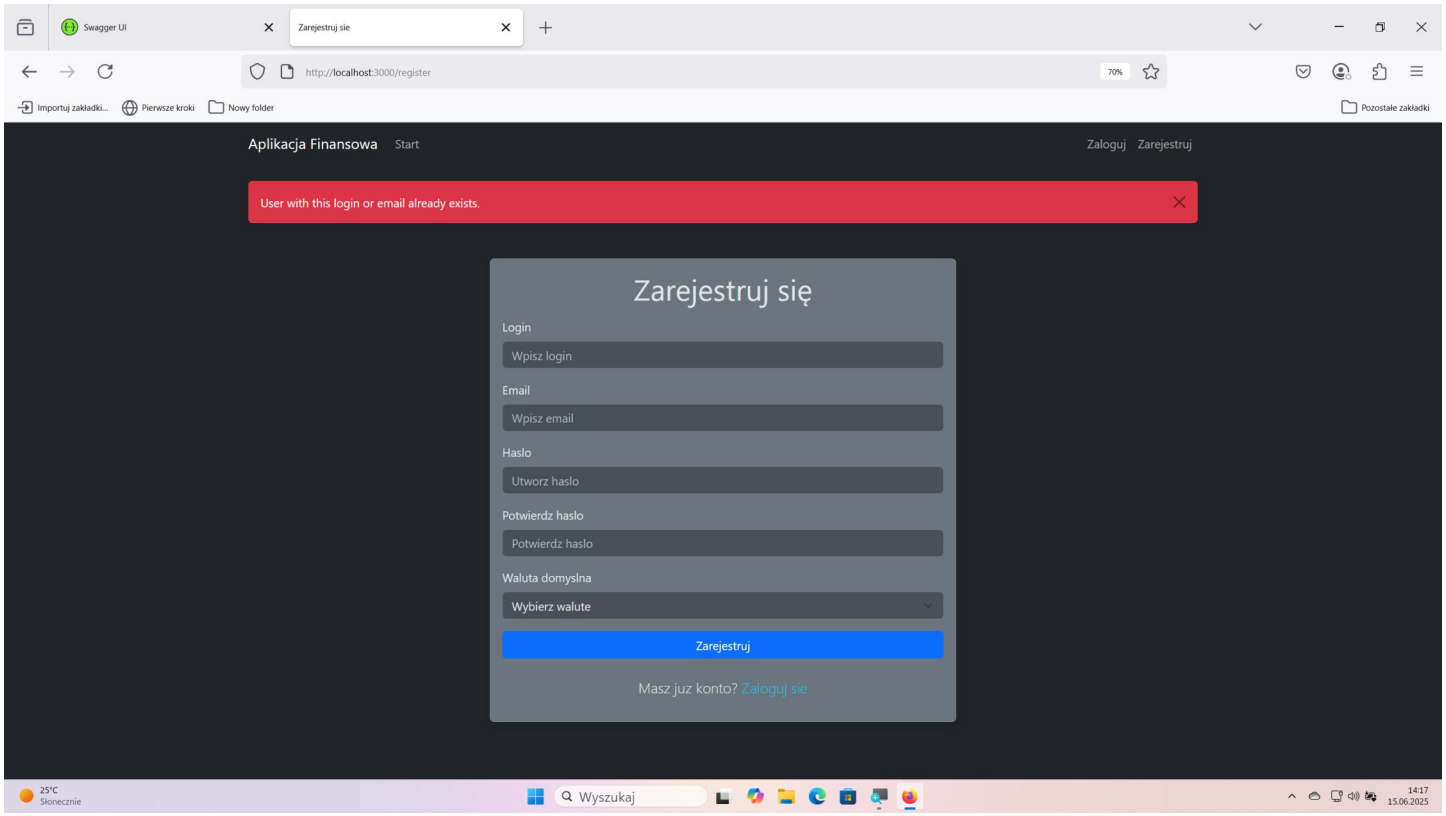
W projekcie uwzględniono również integrację Swaggera z mechanizmem autentykacji JWT, co umożliwia testowanie chronionych endpointów bezpośrednio z poziomu dokumentacji API. W konfiguracji SwaggerGen dodano definicję schematu bezpieczeństwa Bearer, który wymaga podania tokena w nagłówku żądania. Ułatwia to pracę deweloperską i pozwala na szybkie testowanie oraz prezentowanie działania zabezpieczonych zasobów bez konieczności używania zewnętrznych narzędzi do wysyłania żądań HTTP.



Screenshot 4: Token Bearer jest dodany do nagłówków zapytań. Użytkownik jest traktowany jako zalogowany

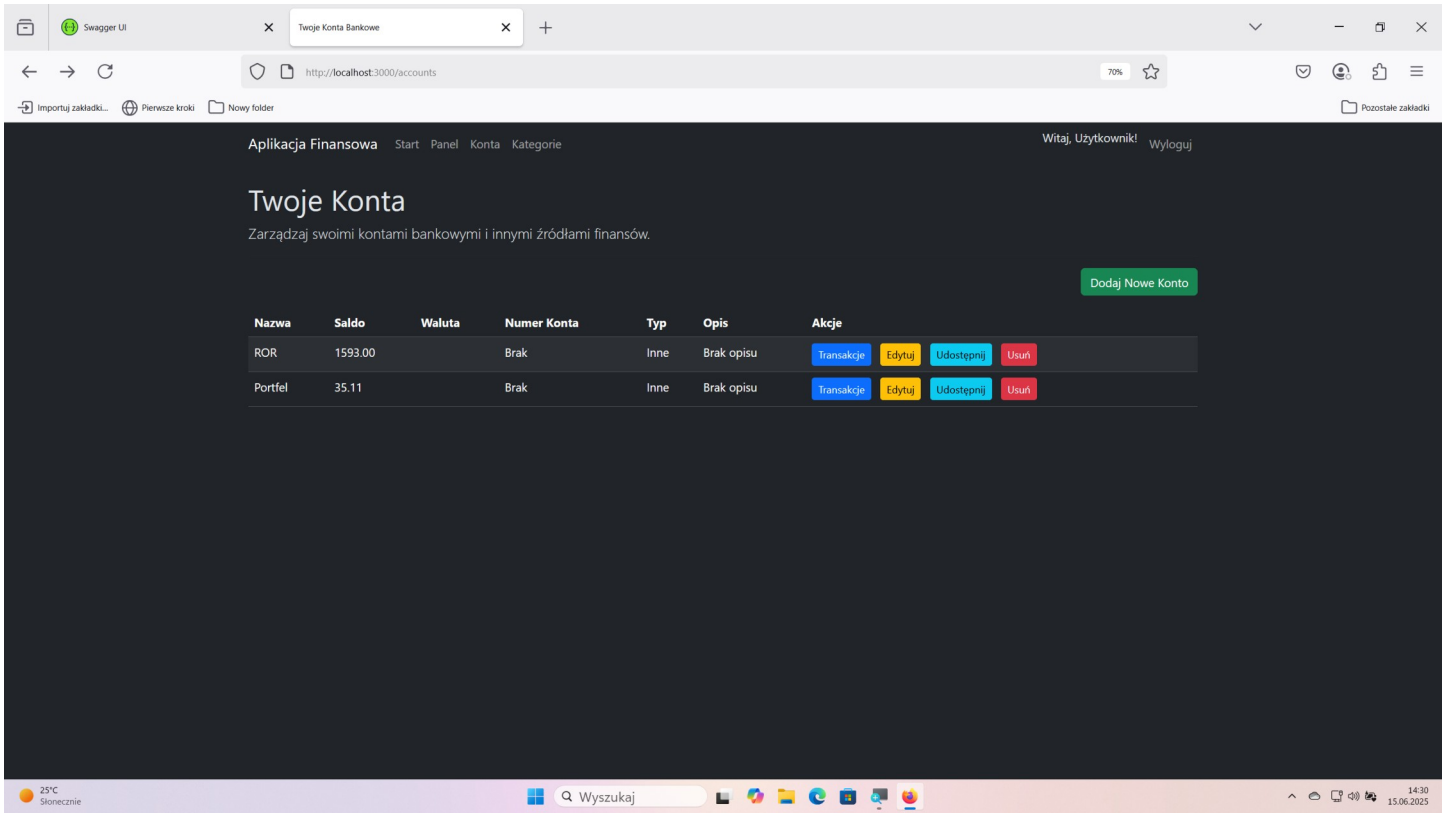
2.3 Interfejs użytkownika

Trzecim kontenerem wchodzącym w skład systemu jest frontend zbudowany w Node.js, oparty o framework Express.js. Jest to lekka warstwa prezentacji, która pełni rolę interfejsu użytkownika aplikacji. Choć Express.js jest typowo kojarzony z tworzeniem API, w tym przypadku został użyty do generowania stron HTML po stronie serwera, w architekturze przypominającej klasyczny wzorzec MVC (Model-View-Controller). Rozwiązanie to łączy prostotę z elastycznością – logika kontrolerów, widoków i danych pozostaje rozdzielona, co ułatwia rozwój i konserwację aplikacji.



Screenshot 5: Frontend otrzymał odpowiedź API, że użytkownik o takim loginie już istnieje

Widoki generowane są z użyciem silnika szablonów EJS, co pozwala dynamicznie renderować zawartość strony na podstawie danych otrzymywanych z API. Serwer Express.js działa więc jako pośrednik między użytkownikiem a API – pobiera dane o kontach, transakcjach czy saldach, a następnie przekazuje je do odpowiednich szablonów, aby wyświetlić je w przejrzysty sposób. Dzięki temu użytkownik końcowy otrzymuje gotowe strony bez konieczności wykonywania samodzielnych zapytań do API – wszystko dzieje się po stronie serwera.



Screenshot 6: Interfejs użytkownika - widok kont

W obecnej wersji frontend korzysta jedynie z części możliwości udostępnianych przez backendowe API – obsługuje logowanie użytkownika, rejestrację, przeglądanie kont i transakcji oraz dodawanie nowych operacji finansowych. Jego modularna konstrukcja pozwala jednak na szybkie rozszerzenie o nowe funkcje, np. zarządzanie uprawnieniami do kont czy dynamiczne przeliczanie walut. W przyszłości interfejs może zostać rozbudowany także o komponenty klienckie (np. Vue.js czy React), które jeszcze bardziej usprawnią interakcję z użytkownikiem i zwiększą responsywność aplikacji.

3 Uruchomienie

3.1 Instrukcja uruchomienia systemu

- Zbuduj aplikację w Visual Studio
  Otwórz całe rozwiązanie w Visual Studio Community (plik \Menedzer_portfela_55208_55209\PersonalFinanceManager\PersonalFinanceManager.sln) Zbuduj je ręcznie (SHIFT+CTRL+B) przed uruchomieniem kontenerów. To konieczne, ponieważ pierwszy kontener (z bazą danych MS SQL) korzysta z plików .sql, które tworzą tabele. Bez wcześniejszego zbudowania projektu baza nie zostanie odpowiednio zainicjalizowana i aplikacja się nie uruchomi.
- Dostosuj ustawienia w appsettings.json kontenera WebAPI. Plik konfiguracyjny appsettings.json w katalogu WebApi zawiera m.in. dane połączeniowe do bazy, ustawienia logowania i klucz JWT. Ważne: W sekcji ExchangeRateApi należy wpisać swój klucz API pobrany z https://app.exchangerate-api.com/, po założeniu darmowego konta. Można również dostosować długość przechowywania danych w pamięci podręcznej (CacheDurationMinutes), aby uniknąć częstych zapytań do zewnętrznego API.

- Uruchom kontenery przez Dockera:
  Użyj polecenia:
  docker-compose up –build
  Dzięki temu system uruchomi wszystkie trzy kontenery: bazę danych, API i frontend, łącząc je w jedną sieć app-network. Upewnij się, że masz zainstalowany Docker Desktop – na komputerach z systemem Windows wymagany jest Docker Desktop – to on umożliwia uruchamianie kontenerów na lokalnej maszynie.

3.2 Dostęp do usług z poziomu komputera gospodarza:

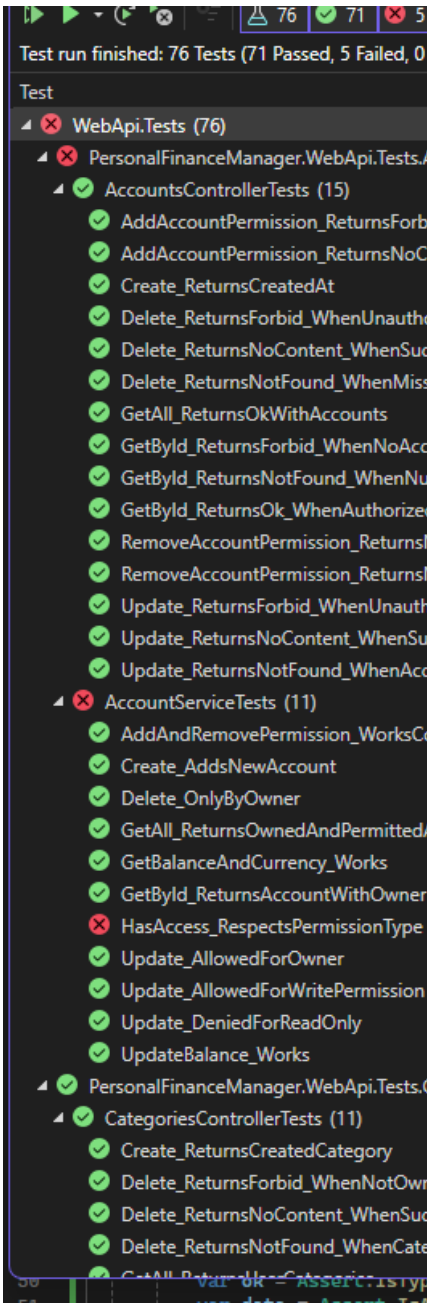
- Dzięki przekierowaniu portów w docker-compose.yml, kontenery są dostępne lokalnie:
  - Baza danych MS SQL: localhost:1433 – możesz podłączyć się z zewnątrz (np. SSMS) i ręcznie przeglądać tabele.
  - WebAPI: dostępne pod: http://localhost:5000. Pod adresem http://localhost:5000/swagger dostępny jest swagger.
  - Frontend: dostępny pod adresem http://localhost:3000

3.3 Dodatkowe uwagi

- Hasła i klucze JWT podane w plikach są tylko przykładowe – przy wdrożeniu należy je zmienić.
- System nie wymaga manualnego wchodzenia do kontenerów po starcie – wszystkie usługi są automatycznie gotowe do działania, o ile konfiguracja została poprawnie ustawiona.

4 Testy

W ramach projektu przeprowadzono dwa główne typy testów mających na celu zapewnienie poprawności działania logiki aplikacji po stronie API. Pierwszym typem były testy kontrolerów, które realizowano z wykorzystaniem mockowanych zależności serwisów. Dzięki temu możliwe było odizolowanie testowanej warstwy od implementacji logiki biznesowej i skupienie się na poprawnym przetwarzaniu żądań HTTP, walidacji danych wejściowych oraz odpowiednim mapowaniu wyników do kodów odpowiedzi.



Screenshot 7: Testy jednostkowe

Drugim typem testów były testy serwisów, które wykonywano z użyciem wbudowanej bazy danych InMemory oraz rzeczywistej implementacji kontekstu bazy danych ApplicationDbContext. Ten typ testów pozwalał na zweryfikowanie integralności i poprawności logiki biznesowej w kontekście interakcji z bazą danych, bez konieczności łączenia się z faktyczną instancją SQL Servera. Dzięki temu możliwe było szybkie uruchamianie testów i łatwe kontrolowanie środowiska testowego. Testy te odegrały istotną rolę w sprawdzeniu spójności operacji na danych oraz zachowania aplikacji w różnych scenariuszach biznesowych.