WitelonBank - Aplikacja Bankowa

Dokumentacja Techniczna REST API

Andrzej Czabajski

9 czerwca 2025

Spis treści

1 Wstęp

1.1 Cel i zakres projektu

Projekt "WitelonBank"zakłada stworzenie kompleksowego systemu bankowości elektronicznej, dostępnego na wielu platformach: web, mobile oraz desktop. Głównym celem systemu jest zapewnienie użytkownikom wygodnego i bezpiecznego narzędzia do zarządzania finansami osobistymi, a administratorom – pełnej kontroli i możliwości monitorowania operacji systemowych.

1.2 Rola i cel API

Kluczowym komponentem architektury systemu jest backend w postaci REST API, który został w całości zaimplementowany w ramach niniejszego projektu. API stanowi kręgosłup systemu, pełniąc rolę warstwy pośredniczącej między aplikacjami klienckimi (frontend) a bazą danych. Odpowiada za całą logikę biznesową, autoryzację, uwierzytelnianie, przetwarzanie transakcji oraz zarządzanie danymi.

1.3 Zakres dokumentacji

Niniejsza dokumentacja techniczna skupia się na opisie architektury, implementacji oraz sposobu uruchomienia modułu REST API. Przedstawiono w niej wykorzystane technologie, strukturę projektu, zastosowane wzorce projektowe oraz szczegółowy opis kluczowych funkcjonalności i mechanizmów bezpieczeństwa.

1.4 Główne założenia projektowe

System został zaprojektowany z myślą o dwóch głównych typach aktorów: standardowym użytkowniku (kliencie banku) oraz administratorze. API realizuje następujące, kluczowe założenia funkcjonalne:

- Zarządzanie kontem osobistym (logowanie, 2FA, reset hasła).
- Wykonywanie operacji bankowych (przelewy, zarządzanie kartami płatniczymi).
- Obsługa inwestycji w kryptowaluty.
- Konfiguracja płatności cyklicznych (zleceń stałych).
- Funkcje administracyjne (monitorowanie transakcji, zarządzanie kontami użytkowników, generowanie raportów).
- Zapewnienie wysokiego poziomu bezpieczeństwa i poufności danych.

2 Opis Funkcjonalny Systemu

System udostępnia szeroki wachlarz funkcjonalności, które są realizowane przez odpowiednie endpointy REST API.

2.1 Aktorzy systemu

Użytkownik (Klient) – główny aktor systemu, posiadający dostęp do funkcji związanych z zarządzaniem własnymi finansami, takich jak:

- Sprawdzanie salda i historii transakcji.
- Wykonywanie przelewów krajowych.
- Zarządzanie kartami płatniczymi (blokowanie, limity).
- Inwestowanie w cyfrowe aktywa (Bitcoin, Ethereum).
- Definiowanie i zarządzanie zleceniami stałymi.
- Zarządzanie listą zaufanych odbiorców.

Administrator – uprzywilejowany użytkownik z dostępem do operacji zarządczych i analitycznych:

- Monitorowanie wszystkich transakcji w systemie.
- Zarządzanie kontami użytkowników (blokowanie, zmiana limitów).
- Generowanie zagregowanych raportów finansowych.
- Podgląd statystyk systemowych.

2.2 Mapowanie funkcjonalności na API

Każda funkcjonalność biznesowa została odwzorowana na konkretny zasób i operacje w REST API, co zapewnia zgodność implementacji ze specyfikacją projektową. Przykładowo:

- Funkcjonalność zarządzania kartami jest realizowana przez endpointy w `KartaController.php`.
- Logika związana z uwierzytelnianiem i 2FA znajduje się w `UzytkownikController.php`.
- Operacje administracyjne, takie jak blokowanie kont, są dostępne pod ścieżkami chronionymi przez middleware, np. `PATCH /api/admin/konta/{idKonta}/block`.

3 Instrukcja uruchomienia i konfiguracji środowiska

Poniższa instrukcja opisuje kroki niezbędne do uruchomienia API na środowisku lokalnym oraz zdalnym.

3.1 Wymagania wstępne

- PHP w wersji 8.2 lub wyższej
- Composer (manager zależności dla PHP)
- Serwer WWW (np. Apache, Nginx)
- Baza danych MySQL
- Git

3.2 Instrukcja uruchomienia lokalnego

1. Sklonuj repozytorium projektu:
   `git clone <https://github.com/Jaktaktonie/apiWitelonBank`
2. Przejdź do katalogu projektu i zainstaluj zależności PHP:
   `composer install`
3. Skopiuj plik konfiguracyjny środowiska:
   `cp .env.example .env`
4. Wygeneruj klucz aplikacji:
   `php artisan key:generate`
5. W pliku `.env` skonfiguruj połączenie z lokalną bazą danych (DB_HOST, DB_PORT, DB_DATABASE, DB_USERNAME, DB_PASSWORD).
6. Uruchom migracje, aby stworzyć strukturę bazy danych, oraz seedery, aby wypełnić ją danymi testowymi:
   `php artisan migrate -seed`
7. Uruchom lokalny serwer deweloperski Laravela:
   `php artisan serve`
   API będzie dostępne pod adresem `http://127.0.0.1:8000`.

3.3 Konfiguracja środowiska zdalnego

Wdrożenie na serwerze produkcyjnym (np. z panelem Apache) wymaga dodatkowych kroków:

- Skonfigurowanie domeny tak, aby jej `DocumentRoot` wskazywał na katalog `/public` projektu.
- Ustawienie w pliku `.env` zmiennych środowiskowych na produkcyjne, w szczególności:
  `APP_ENV=production`
  `APP_DEBUG=false`
- Skonfigurowanie połączenia z produkcyjną bazą danych.

- Zarejestrowanie zadania cyklicznego (cron job) na serwerze, które będzie co minutę uruchamiać komendę Laravela do obsługi zleceń stałych:
  `* * * * * cd /sciezka/do/projektu && php artisan schedule:run » /dev/null 2>&1`
- API dla projektu Witelon Bank znajduje się pod adresem:
  https://witelonapi.host358482.xce.pl/

4 Opis Technologiczny

4.1 Stos technologiczny

- **Backend:** PHP 8.2, Framework Laravel 11.
- **Baza Danych:** MySQL.
- **Serwer WWW:** Apache.
- **Kluczowe pakiety zewnętrzne:**
  - `laravel/sanctum` – do obsługi uwierzytelniania API (tokeny).
  - `darkaonline/l5-swagger` – do automatycznego generowania dokumentacji OpenAPI.
  - `barryvdh/laravel-dompdf` – do generowania plików PDF.

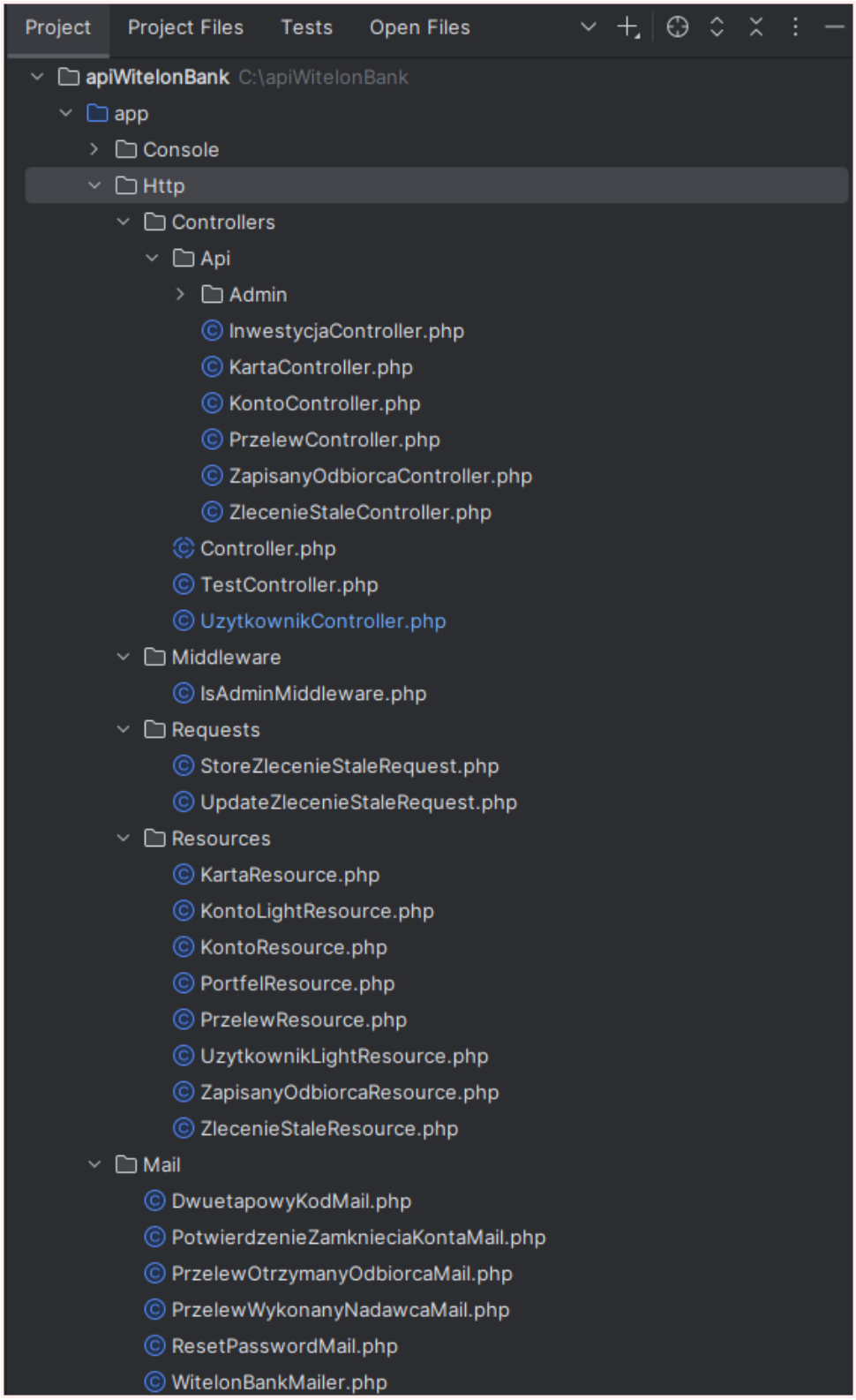
4.2 Architektura API

Aplikacja została zbudowana zgodnie z wzorcem architektonicznym **Model-View-Controller (MVC)**, z wyraźnym podziałem odpowiedzialności. Struktura projektu jest logicznie zorganizowana w katalogi, co ułatwia nawigację i utrzymanie kodu.

API jest zgodne z zasadami **REST**. Wykorzystuje standardowe metody HTTP (GET, POST, PATCH, DELETE), kody statusu HTTP do sygnalizowania wyniku operacji oraz bezstanową komunikację opartą na tokenach uwierzytelniających. Dostęp do endpointów administracyjnych jest dodatkowo chroniony przez dedykowany middleware `IsAdminMiddleware`.

4.3 Struktura bazy danych

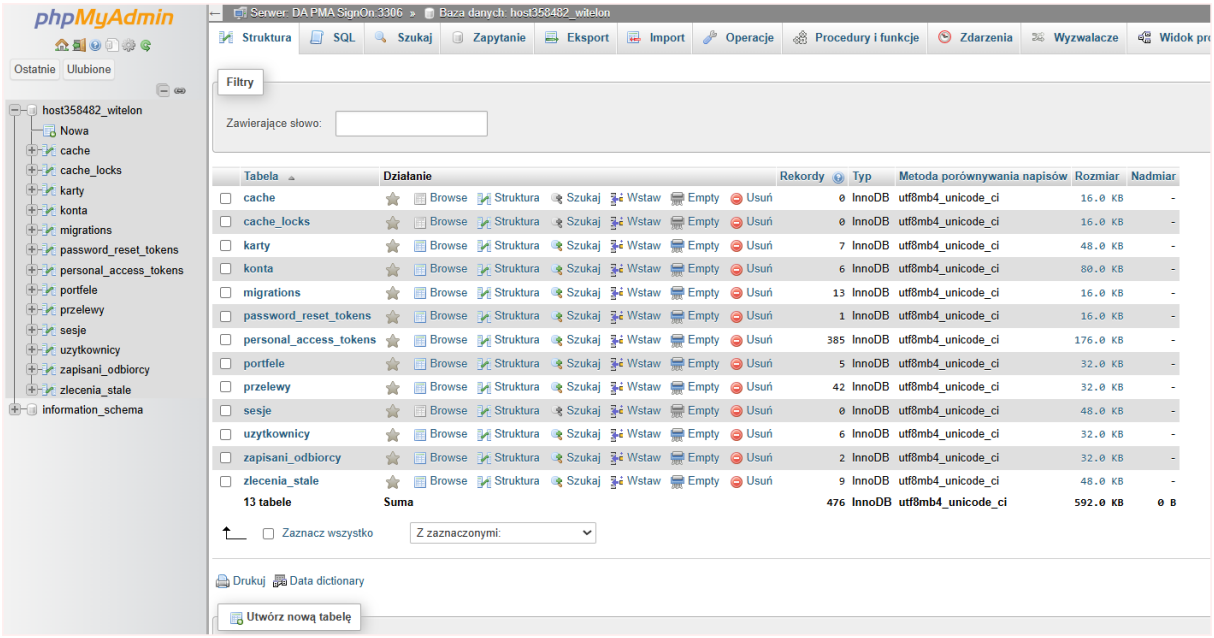
Struktura bazy danych została zdefiniowana za pomocą mechanizmu migracji Laravela, co zapewnia wersjonowanie i łatwość odtworzenia schematu. Za interakcję z tabelami odpowiadają **modele Eloquent ORM**, które definiują również relacje między danymi (np. `hasMany`, `belongsTo`).



Rysunek 1: Struktura katalogów projektu w PHPStorm.

4.4 Dokumentacja endpointów API (Swagger)

Całe API zostało udokumentowane zgodnie ze standardem **OpenAPI 3.0 (Swagger)**. Adnotacje zostały umieszczone bezpośrednio w kodzie kontrolerów, modeli i requestów.



Rysunek 2: Schemat tabel bazy danych w phpMyAdmin.

Dzięki temu dokumentacja jest zawsze aktualna i spójna z implementacją. Interaktywny interfejs, wygenerowany poleceniem `php artisan l5-swagger:generate`, pozwala na przeglądanie i testowanie endpointów bezpośrednio z przeglądarki.

4.5 Zastosowane wzorce projektowe

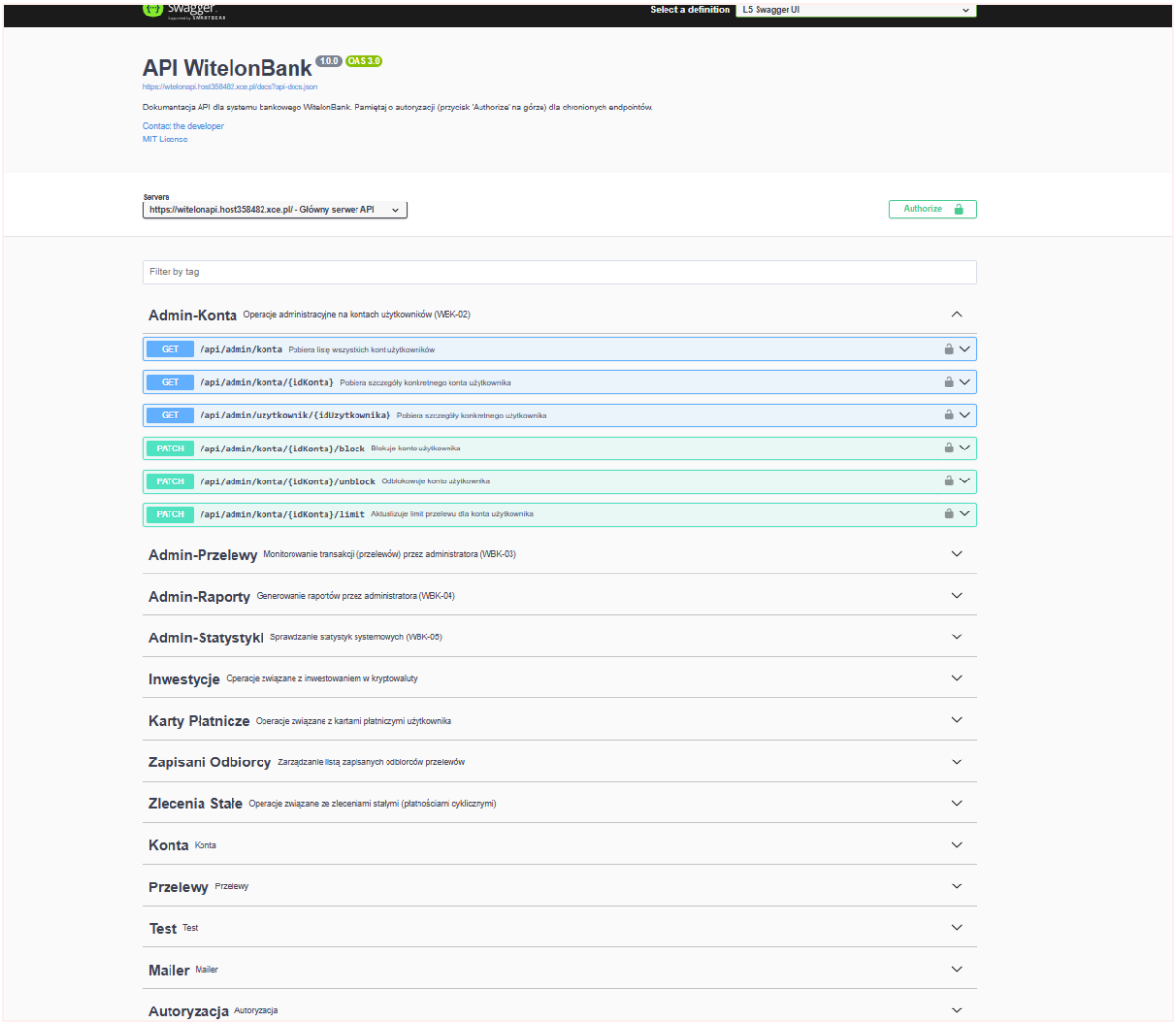
W projekcie świadomie wykorzystano szereg wzorców projektowych i dobrych praktyk w celu zapewnienia czystości, skalowalności i łatwości utrzymania kodu:

API Resource – do transformacji modeli Eloquent na odpowiedzi JSON. Pozwala to na precyzyjne kontrolowanie struktury danych wyjściowych oraz ukrywanie wrażliwych informacji (np. maskowanie numeru karty). Poniżej przykład klasy `KartaResource`.

Form Request – do separacji logiki walidacji od logiki kontrolera. Dedykowane klasy (np. `StoreZlecenieStaleRequest`) zawierają reguły walidacyjne oraz logikę autoryzacji dla danego żądania, co odciąża kontrolery i zwiększa czytelność kodu.

Command – do enkapsulacji logiki, która ma być wykonywana w tle lub z linii poleceń. Wzorzec ten został użyty do implementacji mechanizmu płatności cyklicznych (`ProcessScheduledPayments.php`), który jest uruchamiany przez harmonogram zadań (Scheduler).

Zasada Otwarto-Zamknięta (Open/Closed Principle) – architektura została zaprojektowana tak, aby była otwarta na rozszerzenia, ale zamknięta na modyfikacje. Dobrym przykładem jest moduł inwestycji w kryptowaluty, gdzie dodanie nowego cyfrowego aktywa wymaga jedynie modyfikacji tablicy konfiguracyjnej, bez potrzeby zmiany logiki pobierania cen.



Rysunek 3: Fragment interfejsu użytkownika Swagger UI.

Wstrzykiwanie Zależności (Dependency Injection) – Laravel intensywnie wykorzystuje ten wzorzec do zarządzania zależnościami między klasami, co ułatwia testowanie i zwiększa elastyczność aplikacji.

4.6 Testowanie API

W celu zapewnienia poprawności działania API przeprowadzono **manualne testy funkcjonalne** przy użyciu narzędzia **Postman**. Przygotowano kolekcję zapytań obejmującą wszystkie endpointy, co pozwoliło na weryfikację:

- Poprawności odpowiedzi dla scenariuszy pozytywnych (kody 2xx).
- Obsługi błędów walidacji (kody 422).
- Zabezpieczeń autoryzacji (kody 401, 403).
- Spójności formatu danych zwracanych przez API.

```php
<?php

namespace App\Http\Resources;

use ...

class KartaResource extends JsonResource  7 usages  andrzej *
{
    /**
     * Transform the resource into an array.
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array  andrzej *
    {
        return [
            'id' => $this->id,
            'id_konta' => $this->id_konta,
            // Maskowanie numeru karty - pokazujemy tylko ostatnie 4 cyfry
            'nr_karty_masked' => '**** **** **** ' . substr($this->nr_karty, offset: -4),
            'data_waznosci' => $this->data_waznosci->format('Y-m-d'),
            'zablokowana' => $this->zablokowana,
            'limit_dzienny' => $this->whenNotNull($this->limit_dzienny),
            'typ_karty' => $this->whenNotNull($this->typ_karty),
            'platnosci_internetowe_aktywne' => $this->platnosci_internetowe_aktywne,
            'platnosci_zblizeniowe_aktywne' => $this->platnosci_zblizeniowe_aktywne,
            'created_at' => $this->created_at->toDateTimeString(),
            'updated_at' => $this->updated_at->toDateTimeString(),
            'konto' => new KontoLightResource($this->whenLoaded( relationship: 'konto')),
        ];
    }
}
```

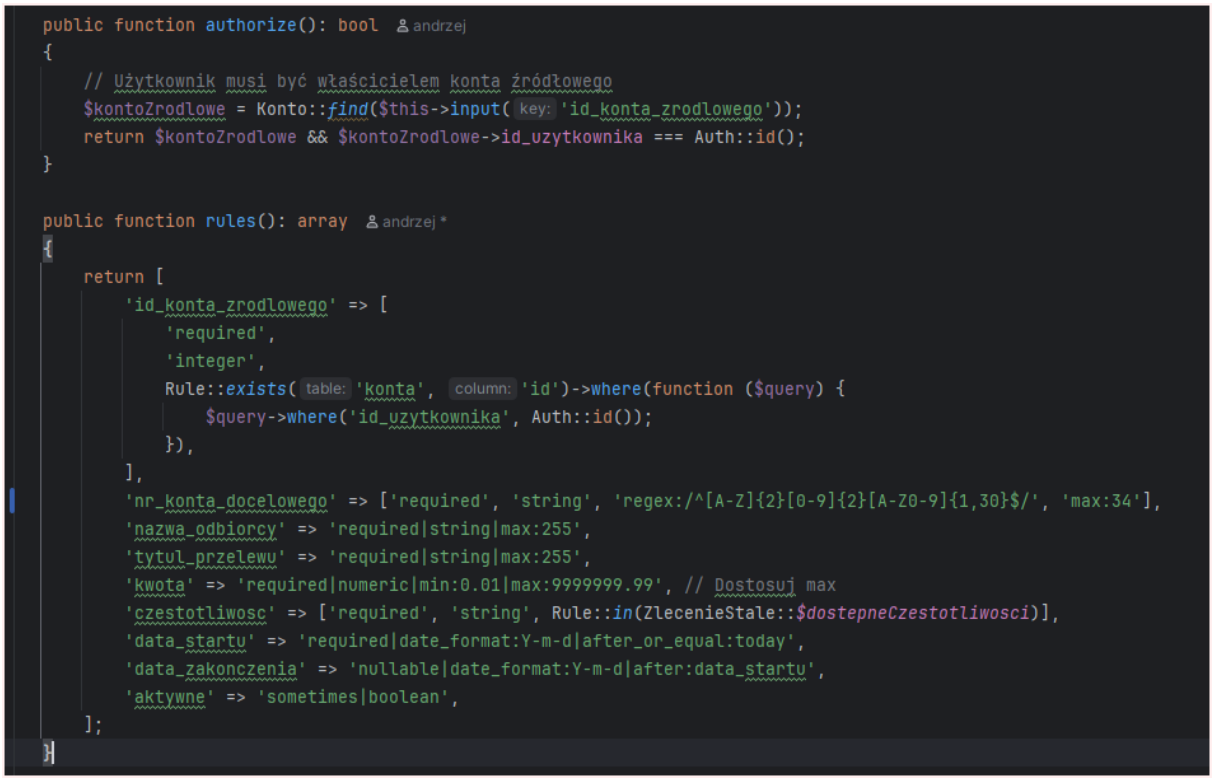
Rysunek 4: Przykład wzorca API Resource w `KartaResource.php`.

Taki proces testowania pozwolił na wczesne wykrycie i naprawę błędów w logice biznesowej i zabezpieczeniach.

5 Wnioski

5.1 Podsumowanie zrealizowanych prac

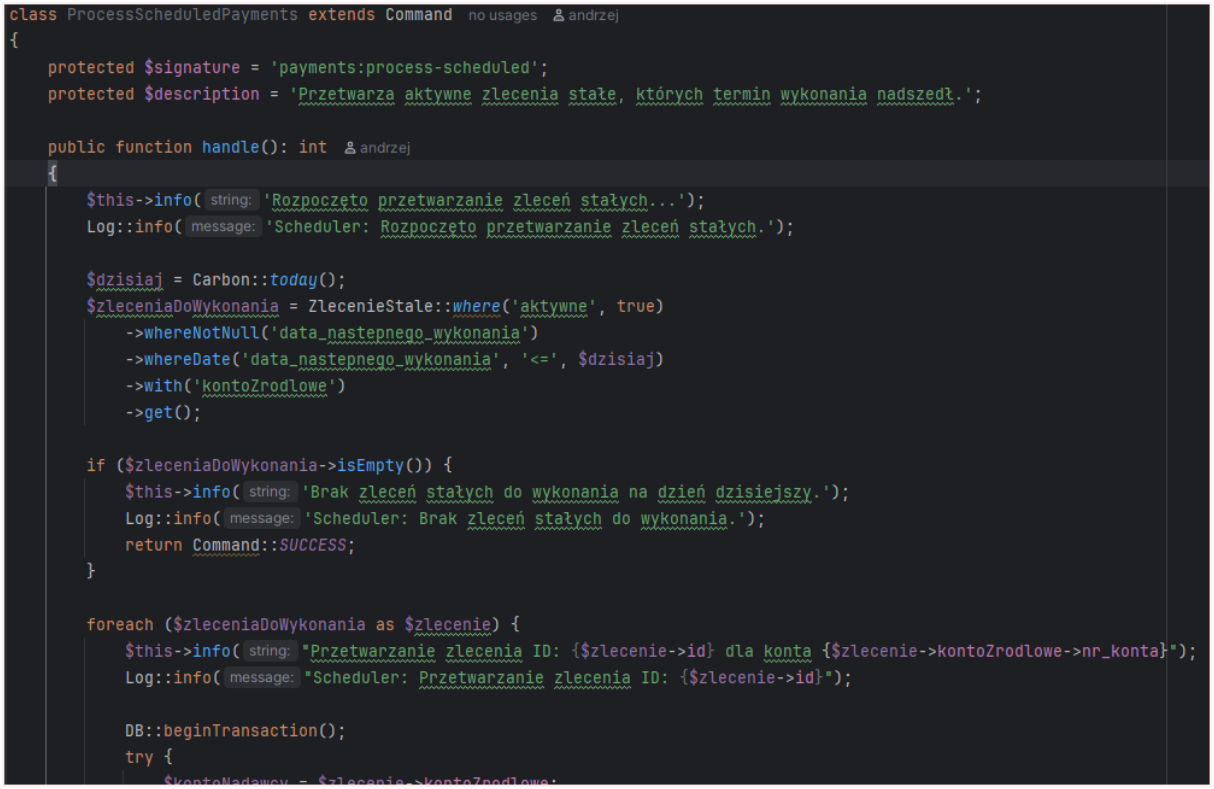
W ramach projektu z powodzeniem zrealizowano kompletny, dobrze zorganizowany i bezpieczny system REST API dla aplikacji bankowej. Aplikacja realizuje wszystkie założone cele funkcjonalne, włączając w to zaawansowane mechanizmy, takie jak płatności cykliczne, uwierzytelnianie dwuskładnikowe oraz integracja z zewnętrznymi serwisami. Zastosowanie frameworka Laravel oraz dobrych praktyk programistycznych pozwoliło na stworzenie skalowalnego i łatwego w utrzymaniu produktu.



```php
public function authorize(): bool  andrzej
{
    // Użytkownik musi być właścicielem konta źródłowego
    $kontoZrodlowe = Konto::find($this->input( key: 'id_konta_zrodlowego'));
    return $kontoZrodlowe && $kontoZrodlowe->id_uzytkownika === Auth::id();
}

public function rules(): array  andrzej *
{
    return [
        'id_konta_zrodlowego' => [
            'required',
            'integer',
            Rule::exists( table: 'konta',  column: 'id')->where(function ($query) {
                $query->where('id_uzytkownika', Auth::id());
            }),
        ],
        'nr_konta_docelowego' => ['required', 'string', 'regex:/^[A-Z]{2}[0-9]{2}[A-Z0-9]{1,30}$/', 'max:34'],
        'nazwa_odbiorcy' => 'required|string|max:255',
        'tytul_przelewu' => 'required|string|max:255',
        'kwota' => 'required|numeric|min:0.01|max:9999999.99', // Dostosuj max
        'czestotliwosc' => ['required', 'string', Rule::in(ZlecenieStale::$dostepneCzestotliwosci)],
        'data_startu' => 'required|date_format:Y-m-d|after_or_equal:today',
        'data_zakonczenia' => 'nullable|date_format:Y-m-d|after:data_startu',
        'aktywne' => 'sometimes|boolean',
    ];
}
```

Rysunek 5: Przykład wzorca Form Request w `StoreZlecenieStaleRequest.php`.



```php
class ProcessScheduledPayments extends Command  no usages  andrzej
{
    protected $signature = 'payments:process-scheduled';
    protected $description = 'Przetwarza aktywne zlecenia stałe, których termin wykonania nadszedł.';

    public function handle(): int  andrzej
    {
        $this->info( string: 'Rozpoczęto przetwarzanie zleceń stałych...');
        Log::info( message: 'Scheduler: Rozpoczęto przetwarzanie zleceń stałych.');

        $dzisiaj = Carbon::today();
        $zleceniaDoWykonania = ZlecenieStale::where('aktywne', true)
            ->whereNotNull('data_nastepnego_wykonania')
            ->whereDate('data_nastepnego_wykonania', '<=', $dzisiaj)
            ->with('kontoZrodlowe')
            ->get();

        if ($zleceniaDoWykonania->isEmpty()) {
            $this->info( string: 'Brak zleceń stałych do wykonania na dzień dzisiejszy.');
            Log::info( message: 'Scheduler: Brak zleceń stałych do wykonania.');
            return Command::SUCCESS;
        }

        foreach ($zleceniaDoWykonania as $zlecenie) {
            $this->info( string: "Przetwarzanie zlecenia ID: {$zlecenie->id} dla konta {$zlecenie->kontoZrodlowe->nr_konta}");
            Log::info( message: "Scheduler: Przetwarzanie zlecenia ID: {$zlecenie->id}");

            DB::beginTransaction();
            try {
                $kontoNadawcy = $zlecenie->kontoZrodlowe;
```

Rysunek 6: Przykład wzorca Command w `ProcessScheduledPayments.php`.

5.2 Napotkane problemy i ich rozwiązania

Największymi wyzwaniami technicznymi w projekcie były:

- **Implementacja płatności cyklicznych:** Problem został rozwiązany poprzez stworzenie dedykowanej komendy Artisan (`ProcessScheduledPayments`) uruchamianej przez harmonogram zadań (Scheduler). Logika wyliczania kolejnych dat wykonania została umieszczona w modelu `ZlecenieStale`, a spójność danych zapewniono dzięki transakcjom bazodanowym.
- **Integracja z zewnętrznym API kryptowalut:** Dynamiczne pobieranie cen wymagało komunikacji z API CoinGecko. Zaimplementowano to przy użyciu fasady `Http` Laravela, dodając mechanizmy obsługi błędów (np. timeout, niedostępność usługi) oraz buforowanie odpowiedzi w celu optymalizacji wydajności.

5.3 Propozycje dalszego rozwoju

Stworzone API stanowi solidną podstawę do dalszego rozwoju. Potencjalne kierunki rozszerzeń to:

- Implementacja testów automatycznych (jednostkowych i funkcjonalnych) w celu zwiększenia niezawodności.
- **Implementacja wielojęzyczności:** Chociaż było to jednym z początkowych założeń, w obecnej wersji API komunikaty o błędach i sukcesie są statyczne (w języku polskim). Dalszy rozwój powinien objąć implementację mechanizmów lokalizacyjnych Laravela (katalog `lang`), aby API mogło zwracać odpowiedzi w języku polskim lub angielskim, w zależności od nagłówka `Accept-Language` w żądaniu klienta.
- Wersjonowanie API w celu zapewnienia kompatybilności wstecznej przy wprowadzaniu zmian.