System do wynajmu pojazdów – Frocar

Sprawozdanie z pracy projektowej

Autorzy:

Mateusz Tęcza – API
Jakub Trznadel – Aplikacja mobilna
Dawid Skowroński – Aplikacja desktopowa
Jakub Kozubek – Aplikacja webowa

Data: 3 czerwca 2025

Spis treści

1 Opis funkcjonalny systemu

FroCar to nowoczesny system wynajmu pojazdów, który umożliwia użytkownikom wypożyczanie samochodów bezpośrednio od innych użytkowników za pośrednictwem aplikacji mobilnej lub strony internetowej oraz administratorom zarządzanie kontami użytkowników, pojazdami udostępnionymi w ofercie oraz monitorowanie i analize działania platformy. System składa się z kilku kluczowych komponentów, z których każdy został zaprojektowany z myślą o intuicyjnej i kompleksowej obsłudze, aby zapewnić wygodne korzystanie zarówno klientom jak i administratorom.

API: Sercem systemu FroCar jest API oparte na C# (.NET), które zapewnia niezawodną komunikację między wszystkimi częściami systemu. Umożliwia ono zarządzanie danymi, uwierzytelnianie użytkowników oraz integrację funkcji, takich jak dodawanie pojazdów, ustalanie cen i dostępności czy przetwarzanie wypożyczeń.

Aplikacja mobilna: Dedykowana użytkownikom (klientom i właścicielom pojazdów) aplikacja mobilna, stworzona w technologii Flutter (Dart), oferuje intuicyjny interfejs do przeglądania dostępnych samochodów w okolicy, rezerwowania pojazdów, zarządzania ogłoszeniami oraz śledzenia statusu wypożyczeń w czasie rzeczywistym.

Strona internetowa: Aplikacja webowa, zbudowana w React i TypeScript, dostarcza użytkownikom i właścicielom pojazdów wygodny dostęp do systemu FroCar przez przeglądarkę. Pozwala na łatwe dodawanie aut, określanie ich dostępności i cen, a także przeglądanie szerokiej gamy samochodów z poziomu komputera.

Aplikacja desktopowa: Panel administracyjny FroCar, stworzony w C# z wykorzystaniem frameworka Avalonia, został zaprojektowany z myślą o administratorach, oferując intuicyjne zarządzanie użytkownikami, pojazdami, ogłoszeniami, wypożyczeniami i recenzjami, a także zaawansowane funkcje, takie jak interaktywna mapa z filtrami i szczegółowe statystyki systemowe, które umożliwiają efektywne i precyzyjne nadzorowanie platformy.

FroCar integruje te elementy, tworząc spójny ekosystem, który łączy funkcjonalność, łatwość obsługi i zrównoważone podejście, promując współdzielenie zasobów w nowoczesny i efektywny sposób.

1.1 Funkcjonalności aplikacji desktopowej

- **Logowanie i rejestracja:** Administrator może zalogować się do systemu. Rejestracja nowych administratorów odbywa się z domyślną rolą użytkownika, a rola administratora jest nadawana przez innego administratora w aplikacji.
- **Reset hasła:** Administrator ma możliwość zresetowania swojego hasła w przypadku potrzeby.
- **Działanie w tle:** Aplikacja minimalizuje się do paska zadań i działa w tle. Obsługuję funkcje powrotu do poprzednio trwającego zadania.
- **Obsługa powiadomień** Podczas działania w tle aplikacja stale nadzoruje obsługę powiadomień oraz dynamicznie wyświetla je w prawym dolnym rogu ekranu.
- **Zarządzanie użytkownikami:** Administrator może wyświetlać listę użytkowników, edytować ich dane oraz usuwać konta.
- **Zarządzanie pojazdami:** Możliwość dodawania, edytowania i usuwania pojazdów w systemie.

- **Lista pojazdów z filtrami:** Administrator może przeglądać listę pojazdów z opcjami filtrowania, co ułatwia zarządzanie flotą.
- **Interaktywna mapa:** System wyświetla interaktywną mapę z rozmieszczeniem pojazdów, wspierającą filtry dla lepszej nawigacji. Po wciśnięciu na mapie na dany pojazd wyświetlane są szczegółowe dane dotyczące pojazdu.
- **Zarządzanie ogłoszeniami:** Administrator może zatwierdzać lub usuwać ogłoszenia użytkowników, z opcją filtrowania.
- **Lista wypożyczeń:** Wyświetlanie aktualnie trwających i zakończonych wypożyczeń z możliwością ich anulowania lub usunięcia.
- **Zarządzanie recenzjami:** Administrator widzi listę recenzji pojazdów wrac z oceną i może je moderować.
- **Statystyki systemowe:** System prezentuje statystyki, które wspierają monitorowanie i analizę działania platformy.
- **Motyw aplikacji:** System obsługuje funkcję zmiany motywu aplikacji (jasny/ciemny). Domyślnie jest ustawiony motyw systemowy.

1.2 Funkcjonalności aplikacji mobilnej i webowej

- **Rejestracja i logowanie**: Użytkownicy mogą tworzyć nowe konta i logować się do systemu. Zaimplementowano również funkcję resetowania hasła.
- **Przeglądanie ofert pojazdów**: Aplikacja wyświetla interaktywną mapę z dostępnymi pojazdami do wynajęcia. Każdy znacznik na mapie reprezentuje pojazd, a po kliknięciu na niego użytkownik może przeglądać szczegółowe informacje o wybranym samochodzie.
- **Filtrowanie i wyszukiwanie pojazdów**: Użytkownicy mają dostęp do zaawansowanych opcji filtrowania, pozwalających na wyszukiwanie pojazdów według marki, liczby miejsc, typu paliwa, ceny, typu nadwozia oraz lokalizacji. Istnieje także możliwość wyczyszczenia wszystkich zastosowanych filtrów.
- **Zarządzanie wypożyczeniami**: Użytkownicy mogą przeglądać listę swoich aktywnych i zakończonych wypożyczeń. Dla zakończonych wypożyczeń dostępna jest opcja dodania recenzji, zawierającej ocenę i opcjonalny komentarz.
- **Powiadomienia systemowe**: Aplikacja wyświetla powiadomienia o istotnych wydarzeniach w systemie, takich jak zmiany statusu wypożyczeń. Ikona powiadomień w pasku nawigacyjnym informuje o liczbie nieprzeczytanych wiadomości.
- **Ustawienia i profil użytkownika**: Użytkownicy mogą przeglądać i edytować dane swojego profilu. Dostępna jest również funkcja zmiany motywu aplikacji (jasny/ciemny).

1.3 Funkcjonalności API

- **Logowanie**: Użytkownicy i administratorzy mogą zalogować się do systemu, korzystając z bezpiecznego mechanizmu autoryzacji.
- **Zarządzanie pojazdami przez administratora**: Administrator ma możliwość dodawania, edytowania oraz usuwania pojazdów w systemie.
- **Zarządzanie ogłoszeniami przez administratora**: Administrator może zatwierdzać lub usuwać ogłoszenia dodane przez użytkowników.
- **Statystyki systemowe**: Administrator ma dostęp do szczegółowych statystyk systemowych, umożliwiających monitorowanie aktywności w systemie.
- **Przeglądanie pojazdów**: Użytkownicy mogą przeglądać listę dostępnych pojazdów w systemie.
- **Filtrowanie pojazdów**: Użytkownicy mogą filtrować pojazdy według marki, ceny, pojemności silnika czy ilości miejsc co ułatwia znalezienie odpowiedniego pojazdu.
- **Lokalizacja pojazdów na mapie**: Użytkownicy mają dostęp do interaktywnej mapy, na której wyświetlane są lokalizacje dostępnych pojazdów.
- **Rejestracja i logowanie użytkownika**: Użytkownicy mogą tworzyć nowe konta oraz logować się do systemu za pomocą bezpiecznych mechanizmów uwierzytelniania.
- **Resetowanie hasła**: Użytkownicy mogą zresetować swoje hasło w przypadku jego utraty, korzystając z procedury odzyskiwania hasła.
- **Wynajem pojazdów**: Użytkownicy mogą rezerwować pojazdy na określony czas, korzystając z intuicyjnego procesu rezerwacji.
- **Powiadomienia o rezerwacjach**: Użytkownicy otrzymują powiadomienia dotyczące statusu swoich rezerwacji, takich jak potwierdzenie, anulowanie czy zakończenie.
- **Historia wynajmów**: Użytkownicy mają dostęp do historii swoich wynajmów, co pozwala na łatwe śledzenie wcześniejszych rezerwacji.
- **Dodawanie pojazdów przez właścicieli**: Właściciele pojazdów mogą dodawać swoje pojazdy do systemu w celu udostępnienia ich do wynajmu.
- **Zarządzanie ogłoszeniami przez właścicieli**: Właściciele mogą edytować oraz usuwać swoje ogłoszenia pojazdów.
- **Ocenianie i recenzowanie pojazdów**: Użytkownicy mogą oceniać i dodawać recenzje do wynajętych pojazdów, co wspiera transparentność i jakość usług.

2 Opis technologiczny

System Frocar został opracowany przy użyciu nowoczesnych technologii, dostosowanych do specyfiki każdej części projektu.

2.1 Technologie

- **API:** Oparte na C# w frameworku .NET, zapewniające stabilną komunikację między komponentami systemu.
- **Aplikacja mobilna:** Stworzona w Flutter (Dart), dedykowana dla klientów i właścicieli pojazdów.
- **Aplikacja desktopowa:** Zbudowana w C# z wykorzystaniem frameworka Avalonia służąca do czynności administracyjnych systemu.
- **Aplikacja webowa:** Opracowana w React z TypeScript, skierowana do klientów i właścicieli pojazdów.

2.2 Specyfika aplikacji desktopowej

Aplikacja desktopowa została zaprojektowana z myślą o administratorach. Wykorzystuje następujące technologie i pakiety:

- **Framework:** Avalonia 11.3.0, nowoczesny framework do tworzenia aplikacji wieloplatformowych.
- **Motyw i style:** Avalonia.Themes.Fluent 11.3.0 dla spójnego i estetycznego interfejsu.
- **Czcionki:** Avalonia.Fonts.Inter 11.3.0 dla czytelnego wyświetlania tekstu.
- **Mapy:** Mapsui 4.1.9 i Mapsui.Avalonia 4.1.9 do obsługi interaktywnej mapy pojazdów.
- **Komunikacja z API:** RestSharp 112.1.0 do wykonywania żądań HTTP.
- **Uwierzytelnianie:** System.IdentityModel.Tokens.Jwt 8.9.0 do obsługi tokenów JWT.

Struktura projektu opiera się na pliku `Project.csproj`, który definiuje zależności, zasoby (np. folder `Assets`) oraz konfigurację, taką jak `net8.0` jako framework docelowy.

2.3 Specyfika aplikacji mobilnej

Aplikacja mobilna została stworzona z myślą o użytkownikach końcowych. Wykorzystuje następujące technologie i pakiety:

- **Framework**: **Flutter SDK** (wersja 3.29.0), co umożliwia tworzenie natywnych aplikacji na platformy Android i iOS z jednej bazy kodu.
- **Język programowania**: **Dart** (wersja: 3.7.0), nowoczesny, zoptymalizowany pod kątem UI język programowania, będący podstawą frameworka Flutter.
- **Zarządzanie Stanem**: **Provider** do efektywnego zarządzania globalnym i lokalnym stanem aplikacji.
- **Mapy**: **Google Maps Flutter** do wyświetlania interaktywnej mapy pojazdów oraz **geolocator** do pobierania aktualnej lokalizacji użytkownika. Geokodowanie adresów realizowane jest za pomocą `nominatim.openstreetmap.org`.

- **Komunikacja z API**: Pakiet **http** do wykonywania żądań HTTP i interakcji z API backendu.
- **Uwierzytelnianie**: **flutter_secure_storage** do bezpiecznego przechowywania tokenów JWT i danych użytkownika oraz **jwt_decoder** do dekodowania tokenów. **Stylizacja**: Wbudowane widżety **Flutter Material Design** dla spójnego i estetycznego interfejsu. Obsługiwane są również motywy jasny i ciemny.

2.4 Specyfika aplikacji webowej

Aplikacja webowa została zaprojektowana z myślą o użytkownikach korzystających z komputerów stacjonarnych i laptopów, zapewniając responsywny interfejs dostosowany do różnych rozmiarów ekranów. Wykorzystuje następujące technologie i pakiety:

- **Framework**: **React** w połączeniu z **Vite** jako narzędziem do szybkiego budowania. Zapewnia to dynamiczny i wydajny interfejs użytkownika, a Vite znacząco skraca czas uruchamiania deweloperskiego i budowania aplikacji.
- **Język programowania**: **TypeScript (TSX)**, czyli nadzbiór JavaScriptu, wzbogacony o typowanie statyczne. Dzięki temu kod jest bardziej przewidywalny, łatwiejszy w utrzymaniu i mniej podatny na błędy, szczególnie w większych projektach.
- **Zarządzanie Stanem**: Zastosowano **React Context API** w połączeniu z hookami `useState` i `useReducer` do efektywnego zarządzania stanem globalnym i lokalnym aplikacji.
- **Komunikacja z API**: Do wykonywania żądań HTTP i interakcji z API backendu wykorzystano bibliotekę **Axios**. Oferuje ona wygodne mechanizmy do obsługi zapytań, odpowiedzi oraz przechwytywania błędów.
- **Uwierzytelnianie**: Bezpieczne uwierzytelnianie użytkowników jest realizowane poprzez przechowywanie tokenów JWT (JSON Web Tokens) w **HTTP-only cookies** przy użyciu pakietu `js-cookie`. Minimalizuje to ryzyko ataków XSS. Dekodowanie tokenów na stronie klienta odbywa się za pomocą biblioteki `jwt-decode`.
- **Mapy**: Do wyświetlania interaktywnej mapy pojazdów wykorzystano bibliotekę **React-Google-Maps** (`@react-google-maps/api`).
- **Komponenty UI i Stylizacja**: Aplikacja korzysta z bibliotek **Bootstrap** i **React-Bootstrap** dla predefiniowanych komponentów UI oraz szybkiego prototypowania. Dodatkowo, obsłużono motywy jasny i ciemny, aby dopasować się do preferencji użytkownika.
- **Routing**: Zarządzanie nawigacją w aplikacji odbywa się za pomocą biblioteki **React Router DOM** (`react-router-dom`), umożliwiającej deklaratywne definiowanie tras i widoków.
- **Walidacja Formularzy**: Walidacja danych wprowadzanych przez użytkownika w formularzach jest realizowana przy użyciu biblioteki **Yup** (`yup`), która ułatwia definiowanie schematów walidacji.
- **Powiadomienia**: Do wyświetlania spersonalizowanych, nietrwałych powiadomień dla użytkownika wykorzystano bibliotekę **React Toastify** (`react-toastify`).

- **Animacje**: Animacje interfejsu użytkownika są implementowane za pomocą biblioteki **Framer Motion** (`framer-motion`), co przyczynia się do płynności i estetyki aplikacji.
- **Ikony**: Zestaw ikon do wzbogacenia interfejsu użytkownika dostarcza biblioteka **React Icons** (`react-icons`).

2.5 Specyfika aplikacji API

Aplikacja API została zaprojektowana jako backend wspierający funkcjonalności systemu wynajmu pojazdów. Wykorzystuje następujące technologie i pakiety:

- **Framework**: **.NET 8.0**, nowoczesna platforma do tworzenia skalowalnych aplikacji serwerowych, zapewniająca wysoką wydajność i bezpieczeństwo.
- **Język programowania**: **C#**
- **Uwierzytelnianie**: **Microsoft.AspNetCore.Authentication.JwtBearer** (wersja 8.0.13) do obsługi tokenów JWT, zapewniających bezpieczne uwierzytelnianie użytkowników i administratorów.
- **Szyfrowanie haseł**: **BCrypt.Net-Next** (wersja 4.0.3) do bezpiecznego hashowania i weryfikacji haseł użytkowników.
- **Baza danych**: **Microsoft.EntityFrameworkCore.SqlServer** (wersja 8.0.13) oraz **Microsoft.EntityFrameworkCore.Design** i **Microsoft.EntityFrameworkCore.Tools** do zarządzania danymi w bazie SQL Server z użyciem Entity Framework Core.
- **Zadania w tle**: **Hangfire** (wersja 1.8.18) z **Hangfire.SqlServer** do zarządzania i wykonywania zadań asynchronicznych, takich jak wysyłanie powiadomień.
- **Serializacja danych**: **Newtonsoft.Json** (wersja 13.0.3) do efektywnego przetwarzania i parsowania danych w formacie JSON.
- **Dokumentacja API**: **Swashbuckle.AspNetCore** (wersja 6.4.0) do generowania interaktywnej dokumentacji API w standardzie OpenAPI (Swagger).

2.6 Testy jednostkowe w aplikacji desktopowej

Aplikacja desktopowa została poddana rygorystycznym testom jednostkowym, aby zapewnić wysoką jakość i niezawodność kodu. Wykonano 164 testy jednostkowe, z czego 152 zakończyły się wynikiem pozytywnym. Daje to wysoki poziom pokrycia kodu, co minimalizuje ryzyko błędów i zapewnia stabilność systemu. Testy obejmowały kluczowe funkcjonalności, takie jak logowanie, zarządzanie użytkownikami, pojazdami i wypożyczeniami, a także interakcje z API. Wyniki testów potwierdzają solidność architektury aplikacji i jej gotowość do pracy w środowisku produkcyjnym.

2.7 Testy w aplikacji mobilnej

Aplikacja mobilna została poddana rygorystycznym testom w celu zapewnienia wysokiej jakości i niezawodności kodu. Zaimplementowano **testy jednostkowe** oraz **testy widżetów**. Testy jednostkowe skupiały się na weryfikacji logiki biznesowej, takiej jak parsowanie danych i działanie algorytmów filtrowania. Testy widżetów zapewniały poprawność renderowania interfejsu użytkownika i jego reakcji na interakcje. Zastosowanie kompleksowego zestawu testów znacząco przyczyniło się do minimalizacji ryzyka błędów i zwiększenia stabilności systemu, co jest kluczowe dla gotowości aplikacji do środowiska produkcyjnego.

2.8 Testy w aplikacji webowej

W projekcie zastosowano **testy end-to-end (E2E)** do weryfikacji funkcjonalności aplikacji z perspektywy użytkownika. Testy te zostały zaimplementowane przy użyciu frameworku **Cypress**, co pozwala na symulowanie interakcji użytkownika z interfejsem graficznym przeglądarki.

- **Cypress**: Wykorzystanie Cypressa umożliwia pisanie niezawodnych i łatwych w utrzymaniu testów E2E. Testy te są uruchamiane w środowisku przeglądarki, co zapewnia wierność symulacji rzeczywistego użytkowania aplikacji.
- **Zakres testów**: Zaimplementowano testy dla kluczowych ścieżek użytkownika, obejmujące między innymi:
  - `AddCarPage.cy.js`: Testy dodawania nowych ofert samochodów.
  - `login.cy.js`: Testy procesu logowania użytkownika.
  - `profilePage.cy.js`: Testy funkcjonalności strony profilu użytkownika.
  - `register.cy.js`: Testy procesu rejestracji nowego konta.
  - `RentalHistoryPage.cy.js`: Testy historii wypożyczeń.
  - `rentalspage.cy.js`: Testy strony z listą dostępnych wypożyczeń.
  - `rentalspagedetails.cy.js`: Testy strony ze szczegółami konkretnego wypożyczenia.
  - `RentCarPage.cy.js`: Testy procesu wypożyczania samochodu.
- **Cel**: Celem tych testów jest zapewnienie, że kluczowe funkcjonalności aplikacji działają poprawnie po każdej zmianie w kodzie, minimalizując ryzyko regresji i poprawiając ogólną stabilność systemu.

2.9 Testy jednostkowe w API

Aplikacja API została poddana kompleksowym testom jednostkowym w celu zapewnienia wysokiej jakości kodu i niezawodności systemu. Zaimplementowano **148 testów jednostkowych** przy użyciu frameworka **xUnit**, które obejmują następujące obszary:

- **Kontrolery**: Testy weryfikują poprawność działania punktów końcowych API, w tym walidację danych wejściowych oraz zwracanie odpowiednich odpowiedzi.

- **Serwisy**: Testy sprawdzają logikę biznesową, taką jak zarządzanie rezerwacjami i powiadomieniami.
- **Modele**: Testy zapewniają poprawność struktur danych i ich walidację, w tym integralność danych w interakcji z bazą danych.

Zastosowanie testów jednostkowych pozwoliło na wczesne wykrywanie błędów, zapewnienie zgodności z wymaganiami funkcjonalnymi oraz zwiększenie stabilności i niezawodności aplikacji API w środowisku produkcyjnym.

3 Wzorce architektoniczne i projektowe

System FroCar wykorzystuje wzorce architektoniczne i projektowe, aby zapewnić modularność, skalowalność i łatwość utrzymania. Poniżej opisano kluczowe wzorce zastosowane w projekcie, ze szczególnym uwzględnieniem ich roli w aplikacjach i usługach.

3.1 Wzorce wykorzystane w aplikacji desktopowej

- **Wzorzec DTO (Data Transfer Object):** Służy do efektywnego przenoszenia danych między warstwami systemu, takimi jak API i aplikacje klienckie. Modele, takie jak UserDto, CarRentalDto czy CarListing, są prostymi strukturami danych z właściwościami, bez złożonej logiki biznesowej. Umożliwiają spójne i uporządkowane przekazywanie informacji, np. danych użytkownika czy szczegółów wypożyczenia, między backendem a aplikacją desktopową, minimalizując zależności i ułatwiając integrację.
- **Wzorzec Fasada:** Dostarcza uproszczony interfejs do złożonych operacji, ukrywając szczegóły implementacji. Klasy serwisowe, takie jak CarService czy UserService, działają jako fasady, upraszczając interakcję z API poprzez ukrywanie szczegółów żądań HTTP, autoryzacji i obsługi błędów. Umożliwia to łatwe wywoływanie funkcji, np. logowania czy zarządzania pojazdami, bez znajomości złożonej logiki backendowej.
- **Wzorzec Obserwator:** Umożliwia reagowanie na zmiany stanu w sposób dynamiczny. Modele, np. CarRentalDto i ReviewDto, dziedziczą po ReactiveObject z biblioteki ReactiveUI, wykorzystując metodę RaiseAndSetIfChanged. Dzięki temu interfejs użytkownika automatycznie odświeża się po zmianie danych, np. statusu wypożyczenia. W NotificationService wzorzec realizuje powiadamianie o nowych notyfikacjach w czasie rzeczywistym za pomocą timera.
- **Wzorzec Adapter:** Dostosowuje dane z jednego formatu do potrzeb systemu. Klasa NominatimResult mapuje odpowiedź JSON z zewnętrznego API Nominatim, przekształcając pola, takie jak "lat" i "lon", na właściwości Latitude i Longitude. Umożliwia to łatwe użycie danych geolokalizacyjnych w aplikacji desktopowej, np. do wyświetlania pozycji pojazdów na interaktywnej mapie, bez konieczności ręcznej transformacji formatu.
- **Wzorzec Kompozyt:** Pozwala traktować pojedyncze obiekty i ich grupy w sposób jednolity. Modele, takie jak CarRentalDto, ReviewDto i CarListing, tworzą

hierarchię danych, zawierając referencje do innych obiektów DTO, np. CarRentalDto obejmuje CarListing i UserDto. Umożliwia to przekazywanie złożonych, powiązanych danych, takich jak informacje o wypożyczeniu wraz z danymi pojazdu i użytkownika, w jednym, spójnym obiekcie.

3.2 Wzorce wykorzystane w aplikacji mobilnej

Podczas tworzenia aplikacji mobilnej skorzystaliśmy z kilku kluczowych wzorców projektowych, które pomogły w utrzymaniu czystości kodu, skalowalności i łatwości zarządzania stanem:

- **Wzorzec Strategia (Strategy Pattern)**: Ten wzorzec został zastosowany do implementacji **filtrowania ofert samochodów**. Pozwala to na elastyczne dodawanie i modyfikowanie kryteriów filtrowania bez zmiany kodu klienckiego. Każde kryterium (np. filtrowanie po marce, cenie, typie paliwa) jest zdefiniowane jako osobna strategia, implementująca wspólny interfejs. Dzięki temu, w przyszłości łatwo będzie dodać nowe opcje filtrowania, minimalizując wpływ na istniejący kod.

- **Wzorzec Budowniczy (Builder Pattern)**: Ten wzorzec został zaimplementowany do **konstruowania złożonych obiektów modeli danych**, takich jak `CarListing`. Klasa `CarListingBuilder` pozwala na tworzenie instancji `CarListing` krok po kroku, ustawiając poszczególne właściwości za pomocą dedykowanych metod (`setId`, `setBrand`, itp.). Dzięki temu proces tworzenia obiektu jest bardziej czytelny i elastyczny, szczególnie w przypadku obiektów z wieloma atrybutami. Metody takie jak `fromJson` oraz `placeholder` wykorzystują ten budowniczy, aby w uporządkowany sposób tworzyć obiekty `CarListing` z danych JSON lub jako wartości domyślne, zapewniając spójność i poprawność konstrukcji.

- **Wzorzec Adapter (Adapter Pattern)**: W pliku `notification_adapter.dart` w folderze `services` znajduje się klasa `MapToNotificationAdapter`, która implementuje interfejs `NotificationAdapter`. Jej zadaniem jest **konwersja listy map na listę obiektów `NotificationModel`**. Jest to klasyczny przykład wzorca Adapter, gdzie jeden interfejs (`List<Map<String, dynamic>>`) jest dostosowywany do innego (`List<NotificationModel>`), aby umożliwić współpracę niekompatybilnych ze sobą struktur danych.

- **Wzorzec Obserwator (Observer Pattern)**: W kontekście zarządzania stanem za pomocą pakietu `Provider`, wzorzec Obserwator jest fundamentalny. Klasy takie jak `NotificationProvider`, `ThemeProvider` czy `UserProvider` dziedziczą po `ChangeNotifier`. Widżety, które "nasłuchują" na zmiany w tych providerach za pomocą `Consumer` lub `Provider.of<T>(context, listen: true)`, reagują na powiadomienia o zmianach stanu wywoływane przez `notifyListeners()`. Dzięki temu interfejs użytkownika automatycznie aktualizuje się, gdy zmieniają się dane w providerach, co zapewnia dynamiczność i spójność wyświetlanych informacji.

3.3 Wzorce wykorzystane w aplikacji webowej

- **Wzorzec Strategia (Strategy Pattern)**: Wzorzec ten został zastosowany do implementacji **filtrowania ofert samochodów**. Umożliwia on dynamiczne wybieranie i stosowanie różnych algorytmów filtrowania bez modyfikacji kodu klienckiego

(ProfilePage.tsx). Każde kryterium filtrowania (np. marka, dostępność, typ paliwa) jest odrębną strategią (funkcją taką jak `filterByBrand`, `filterByAvailability`), zgodną ze wspólnym interfejsem `CarFilterStrategy`. Funkcja `applyFilters` działa jako kontekst, który przyjmuje i kolejno stosuje wybrane strategie. To podejście zapewnia **elastyczność** i **łatwość rozszerzania** o nowe opcje filtrowania w przyszłości, zgodnie z zasadą otwarte/zamknięte.

- **Wzorzec Obserwator (Observer Pattern)**: Klasa `NotificationService` pełni rolę **Obserwowanego (Subject/Publisher)**. Posiada metody `subscribe` i `unsubscribe`, które pozwalają innym komponentom (Obserwatorom/Subskrybentom) rejestrować się i wyrejestrowywać w celu otrzymywania powiadomień. Metoda notifyObservers jest wywoływana po pomyślnym pobraniu nowych powiadomień, rozgłaszając je do wszystkich zarejestrowanych subskrybentów. Dzięki temu, komponenty UI mogą reagować na nowe powiadomienia w czasie rzeczywistym, bez konieczności ciągłego sprawdzania stanu.

- **Wzorzec Singleton (Singleton Pattern)**: Klasa `NotificationService` zaimplementowała ten wzorzec poprzez eksportowanie pojedynczej instancji siebie (`export const notificationService = new NotificationService();`). Gwarantuje to, że w całej aplikacji istnieje tylko jeden obiekt `NotificationService`. Dzięki temu zapewnia globalny i jednolity punkt dostępu do mechanizmów zarządzania powiadomieniami i ich okresowego odpytywania (pollingu), eliminując ryzyko duplikacji danych i zapewniając spójność operacji.

3.4 Wzorce wykorzystane w aplikacji API

- **Wzorzec Obserwator (Observer Pattern)**: Zastosowano w mechanizmie powiadomień systemowych. Serwisy odpowiedzialne za powiadomienia ukazują zmiany statusu rezerwacji (np. zakończenie, anulowanie) i reagują na nie, wysyłając odpowiednie komunikaty do użytkowników. Dzięki temu system dynamicznie informuje użytkowników o istotnych zmianach w czasie rzeczywistym.

- **Wzorzec Strategia (Strategy Pattern)**: Zastosowano w module sortowania pojazdów w kontrolerze `FilterController`. Wzorzec ten umożliwia dynamiczne wybieranie strategii sortowania (np. według ceny, marki, pojemności silnika lub liczby miejsc) poprzez interfejs `ISortStrategy`. Dedykowane klasy, takie jak `PriceSortStrategy`, `BrandSortStrategy`, `EngineSortStrategy`, `SeatsSortStrategy` oraz `DefaultSortStrategy`, implementują ten interfejs, definiując logikę sortowania dla różnych kryteriów. Klasa `SortStrategyContext` zarządza strategiami, umożliwiając łatwe przełączanie między nimi na podstawie parametru `sortBy` przekazanego w żądaniu HTTP. Dzięki temu system jest elastyczny i łatwy do rozszerzenia o nowe kryteria sortowania bez konieczności modyfikacji istniejącego kodu.

- **Wzorzec Fasada (Facade Pattern)**: Zastosowano w celu uproszczenia złożonych operacji w kontrolerach, takich jak `AccountController`, poprzez wykorzystanie serwisów, takich jak `EmailService`, `NotificationService` oraz `RentalService`. Na przykład, metoda `SendEmailAsync` w `EmailService` ukrywa szczegóły konfiguracji SMTP do wysyłania maili resetujących hasło, oferując prosty interfejs dla kontrolerów. Serwisy te działają jak fasada, ukrywając skomplikowaną logikę wewnętrzną

i umożliwiając kontrolerom łatwe wykonywanie złożonych zadań, takich jak wysyłanie powiadomień czy zarządzanie rezerwacjami.

- **Wzorzec Budowniczy (Builder Pattern)**: Zastosowano w celu tworzenia złożonych obiektów, takich jak `CarListing`, w operacjach dodawania i edytowania pojazdów. Wzorzec ten umożliwia konstruowanie obiektów `CarListing` krok po kroku, z walidacją pól, takich jak marka, cena, pojemność silnika czy liczba miejsc, co jest zgodne z komunikatami błędów zdefiniowanymi w `ErrorMessages` (np. `BadRequestRequiredBrand`, `BadRequestEngineCapacity`). Dzięki Builderowi proces tworzenia obiektów jest bardziej czytelny i elastyczny, szczególnie przy obsłudze żądań HTTP z danymi wejściowymi od użytkownika lub właściciela pojazdu. Wzorzec ten minimalizuje ryzyko błędów walidacji i zapewnia spójność danych w systemie.

4 Podział obowiązków i odpowiedzialności w zespole

- **Mateusz Tęcza:** Odpowiedzialny za projektowanie i implementację API w C# (.NET).
- **Jakub Trznadel:** Opracowanie aplikacji mobilnej w Flutter (Dart) dla klientów i właścicieli pojazdów.
- **Dawid Skowroński:** Rozwój aplikacji desktopowej w C# z wykorzystaniem Avalonia dla administratorów systemu.
- **Jakub Kozubek:** Stworzenie aplikacji webowej w React i TypeScript dla klientów i właścicieli pojazdów.

5 Instrukcja lokalnego i zdalnego uruchomienia systemu

5.1 Lokalne uruchomienie aplikacji desktopowej

1. **Wymagania wstępne:**
   - Zainstaluj .NET 8.0 SDK.
   - Zainstaluj środowisko uruchomieniowe Avalonia (dostępne na stronie projektu Avalonia).
2. **Pobieranie kodu:**
   - Sklonuj repozytorium projektu: `git clone https://github.com/dawid-skowronski/FrocarDesktop`.
   - Przejdź do katalogu aplikacji desktopowej: `cd AdminPanel`.
3. **Kompilacja i uruchomienie:**
   - Wykonaj komendę `dotnet restore`, aby pobrać wszystkie zależności.
   - Skompiluj projekt: `dotnet build`.
   - Uruchom aplikację: `dotnet run`.

4. **Konfiguracja:**
    - Upewnij się, że API jest uruchomione i dostępne (np. na `http://localhost:5001`).
    - Skonfiguruj poświadczenia API w pliku konfiguracyjnym aplikacji.

5.2 Lokalne uruchomienie aplikacji webowej

1. **Wymagania wstępne:**
    - Zainstaluj **Node.js** (zalecana wersja LTS).
2. **Pobieranie kodu:**
    - Sklonuj repozytorium projektu:
      `git clone https://github.com/dawid-skowronski/FrocarWeb`.
    - Przejdź do katalogu głównego aplikacji webowej.
3. **Instalacja zależności:**
    - Wykonaj komendę `npm install`, aby pobrać wszystkie zależności projektu.
4. **Kompilacja i uruchomienie:**
    - Uruchom aplikację w trybie deweloperskim: `npm run dev`. Aplikacja będzie dostępna pod wskazanym adresem (zazwyczaj `http://localhost:5173`).
5. **Konfiguracja:**
    - Upewnij się, że backend API jest uruchomiony i dostępny.
    - Skonfiguruj adres URL API oraz klucze (np. klucz Google Maps API) w pliku konfiguracyjnym środowiska (.env).

5.3 Lokalne uruchomienie aplikacji API

1. **Wymagania wstępne:**
    - Zainstaluj **Visual Studio 2022** (zalecana wersja Community, Professional)
2. **Pobieranie kodu:**
    - Sklonuj repozytorium projektu: `https://github.com/dawid-skowronski/FrocarAPI` lub pobierz plik FrogCar.
3. **Wczytanie projektu:**
    - Otwórz plik rozwiązania (`FrogCar.sln`) w Visual Studio 2022.
4. **Konfiguracja:**
    - Upewnij się, że wszystkie zależności NuGet (np. `Microsoft.EntityFrameworkCore.SqlServer`), są poprawnie zainstalowane. Wykonaj jedną z poniższych metod:

- **W konsoli systemowej**: Przejdź do katalogu projektu i wykonaj komendę `dotnet restore` w terminalu, aby pobrać wszystkie pakiety NuGet.
- **W konsoli menedżera pakietów**: W Visual Studio otwórz konsolę menedżera pakietów (*Package Manager Console*) i wykonaj komendę `Restore-Package`.

5. **Migracje bazy danych:**

   - W terminalu w Visual Studio (Package Manager Console) wykonaj komendę `Update-Database`, aby zastosować migracje Entity Framework Core i utworzyć schemat bazy danych.

6. **Kompilacja i uruchomienie:**

   - Skompiluj projekt, wybierając konfigurację *Debug* w Visual Studio.
   - Uruchom aplikację, naciskając `F5` lub klikając przycisk *Start* w Visual Studio. API będzie dostępne pod domyślnym adresem (np. `https://localhost:5001` lub `http://localhost:5000`).

5.4 Zdalne uruchomienie systemu

1. **Serwer API:**

   - Wdróż API na serwerze (np. za pomocą IIS, Docker lub chmury, takiej jak Azure).
   - Upewnij się, że API jest dostępne publicznie, np. pod adresem `https://projekt-tripify.hostingasp.pl/index.html`.

2. **Aplikacja desktopowa:**

   - Skompiluj aplikację do postaci wykonywalnej: `dotnet publish -c Release`.
   - Wdróż wygenerowany plik wykonywalny na maszynach administratorów.
   - Skonfiguruj adres URL API w ustawieniach aplikacji, aby łączyć się ze zdalnym serwerem.

3. **Aplikacja mobilna:**

   - Sklonuj repozytorium i przejdź do katalogu projektu mobilnego.
   - Pobierz zależności: `flutter pub get`.
   - Uruchom aplikację na wybranym urządzeniu/emulatorze: `flutter run`.
   - Aplikacja mobilna jest domyślnie skonfigurowana do komunikacji ze zdalnym backendem

4. **Wymagania sieciowe:**

   - Stabilne połączenie internetowe.
   - Otwarte porty dla API (np. 443 dla HTTPS).

6 Wnioski projektowe

6.1 Mateusz Tęcza

Moja rola w tym projekcie skupiała się na zbudowaniu solidnego backendu oraz zaprojektowaniu i wdrożeniu API przy pomocy C# oraz .Netu. Było to kluczowe doświadczenie, które pozwoliło mi znacząco poszerzyć moją wiedzę i umiejętności. Od początku projektu zdawałem sobie sprawę, że moja praca nad backendem będzie miała wpływ na reszte zespołu, a więc starałem się dostarczyć API w jak najszybszym czasie oraz na jak najlepszym poziomie. Niezwykle sprawna komunikacja w zespole okazała się kluczowa, ponieważ nie było żadnych problemów ani nie wyjaśnionych spraw. Jestem przekonany, że zdobyta wiedza i umiejętności przyda się w dalszej podróży z programowaniem.

6.2 Jakub Trznadel

Ten projekt był dla mnie pierwszym kontaktem zarówno z tworzeniem aplikacji mobilnych, jak i z samym frameworkiem Flutter. Początki były wymagające, wymagały intensywnej nauki od podstaw i mierzenia się z nowymi koncepcjami. Niemniej jednak, z czasem coraz bardziej doceniałem elastyczność i wydajność Fluttera, a proces tworzenia interaktywnego interfejsu i integracji z zewnętrznymi usługami, takimi jak mapy czy API backendu, okazał się niezwykle satysfakcjonujący. Jestem zadowolony z osiągniętych rezultatów i wierzę, że zdobyte doświadczenie przyda się w przyszłości.

6.3 Dawid Skowroński

Praca nad aplikacją desktopową była wymagającym, ale satysfakcjonującym zadaniem. Wykorzystanie Avalonia i wzorca MVVM pozwoliło na stworzenie przejrzystej i skalowalnej architektury. Integracja z API za pomocą RestSharp była efektywna, a użycie Mapsui umożliwiło implementację interaktywnej mapy. Testy jednostkowe potwierdziły niezawodność kluczowych funkcjonalności. Wyzwaniem było zapewnienie spójności interfejsu z motywem FluentTheme, co udało się osiągnąć dzięki odpowiednim pakietom. Projekt uświadomił mi, jak ważne jest dobre planowanie i testowanie w złożonych systemach.

6.4 Jakub Kozubek

Mój pierwszy projekt frontendowy w Reactcie był sporym wyzwaniem. Początki były trudne, zwłaszcza w porównaniu do pisania w czystym JavaScriptcie. Musiałem zrozumieć, co dokładnie się robi i dlaczego, co pochłonęło sporo czasu. Podobnie było z wzorcami projektowymi – początkowo nie widziałem ich zastosowania ani korzyści. Jednak po opanowaniu podstaw Reacta, praca zaczęła sprawiać mi dużą satysfakcję. Ogromnym wsparciem była też znakomita współpraca z Mateuszem Tęczą, naszym backendowcem, dzięki której projekt posuwał się naprzód gładko i sprawnie. Dziś doceniam znaczenie wzorców projektowych, choć wiem, że w przyszłości mogę wykorzystać je jeszcze efektywniej. React, wraz z całym ekosystemem bibliotek i narzędzi, naprawdę upraszcza proces tworzenia aplikacji, sprawiając, że jest on znacznie bardziej intuicyjny i przyjemny.