**Politechnika Krakowska im. Tadeusza Kościuszki**

**Wydział Inżynerii Elektrycznej i Komputerowej**



Kacper Nowak  
 Karol Węgrzyn  
 Sebastian Młodzik

Technologie Obiektowe

Projekt

GoFleetGo

[Cel Projektu 4](#_Toc158482766)

[Zakres Projektu 4](#_Toc158482767)

[Główne Funkcje Produktu 5](#_Toc158482768)

[Diagram Przypadków Użycia 6](#_Toc158482769)

[Scenariusze Użytkownika 7](#_Toc158482770)

[Klient Prywatny 7](#_Toc158482771)

[Schemat Bazy Danych 7](#_Toc158482772)

[Opis klas używanych do komunikacji: 8](#_Toc158482773)

[Schemat przekazywania danych 8](#_Toc158482774)

[Opis klas Repositories 9](#_Toc158482775)

[Company 9](#_Toc158482776)

[Model 10](#_Toc158482777)

[Reservation 12](#_Toc158482778)

[Review 14](#_Toc158482779)

[Ride 15](#_Toc158482780)

[Route 17](#_Toc158482781)

[Subscription 19](#_Toc158482782)

[Transaction 20](#_Toc158482783)

[User 21](#_Toc158482784)

[Vehicle 23](#_Toc158482785)

[TCP 26](#_Toc158482786)

[Server 26](#_Toc158482787)

[Client 27](#_Toc158482788)

[ClientHandler 28](#_Toc158482789)

[UI – Logowanie, Rejestracja, Menu 31](#_Toc158482790)

[Rejestracja 31](#_Toc158482791)

[Sprawdzanie poprawności danych tworzonego konta 31](#_Toc158482792)

[Logowanie 39](#_Toc158482793)

[Weryfikacja danych logowania 40](#_Toc158482794)

[Menu 41](#_Toc158482795)

[Wypożyczenie samochodu 42](#_Toc158482796)

[Widok Poruszania Pojazdem 46](#_Toc158482797)

[Funkcje kończące przejazd 49](#_Toc158482798)

[Aktualizacja lokalizacji i stanu paliwa po zakończonym przejeździe 51](#_Toc158482799)

[Rezerwacja samochodu 52](#_Toc158482800)

[Services 54](#_Toc158482801)

[VehicleService 54](#_Toc158482802)

[RideService 54](#_Toc158482803)

[Wnioski 55](#_Toc158482804)

# Cel Projektu

Celem projektu jest opracowanie kompleksowego oprogramowania wspomagającego procesy wypożyczalni samochodowej, umożliwiającego efektywne zarządzanie flotą pojazdów, rezerwacjami oraz obsługę klientów. Oprogramowanie ma zapewnić łatwą nawigację, zautomatyzować procesy administracyjne, zwiększyć przejrzystość operacyjną oraz poprawić obsługę klienta, przyczyniając się do efektywnego i rentownego funkcjonowania wypożyczalni samochodowej.

# Zakres Projektu

Program ma umożliwić proces wyszukiwania najbliższych pojazdów, rezerwacji i wypożyczania pobliskich samochodów. System geolokalizacji za pomocą, którego będziemy śledzić położenie samochodów w czasie rzeczywistym będzie podstawą działania aplikacji, na jego bazie będziemy optymalizować rozmieszczenie pojazdów oraz obliczać koszty wypożyczenia.

Klienci wypożyczali by pojazd i musieli opłacić kwotę początkową, a następnie ustaloną kwotę za każdy kilometr przebyty w czasie wypożyczenia. Po zakończonym przejeździe podawali by swoją lokalizację a następnie program porównywał by ich pozycję z miejscami wykluczonymi z zakończenia pojazdu(obszar poza granicami miasta oraz mniejsze strefy w centrum miasta). Za parkowanie w strefie wykluczonej na użytkowników będą nakładane kary.

W ramach projektu zostanie utworzona baza danych, która będzie odpowiadać za przechowywanie danych w bezpieczny sposób.

# Główne Funkcje Produktu

* Rejestracja i logowanie,

podstawowa funkcja programu, pozwoli na identyfikację, dostęp klienta do: portfela, historii przejazdów, wypożyczania pojazdów.

* Określanie obecnego zasięgu pojazdu,

na podstawie tego parametru będziemy w stanie określić, czy samochód dalej ma być uznawany jako aktywny. Użytkownik musi być świadomy jaki dystans będzie mógł pokonać przy obecnym stanie paliwa.

* Wypożyczenie samochodu,

główna funkcja programu, udostępnia pojazd dla użytkownika, pobiera początkową opłatę oraz rozpoczyna naliczanie opłaty za przejechane odcinki drogi.

* Zwrot samochodu,

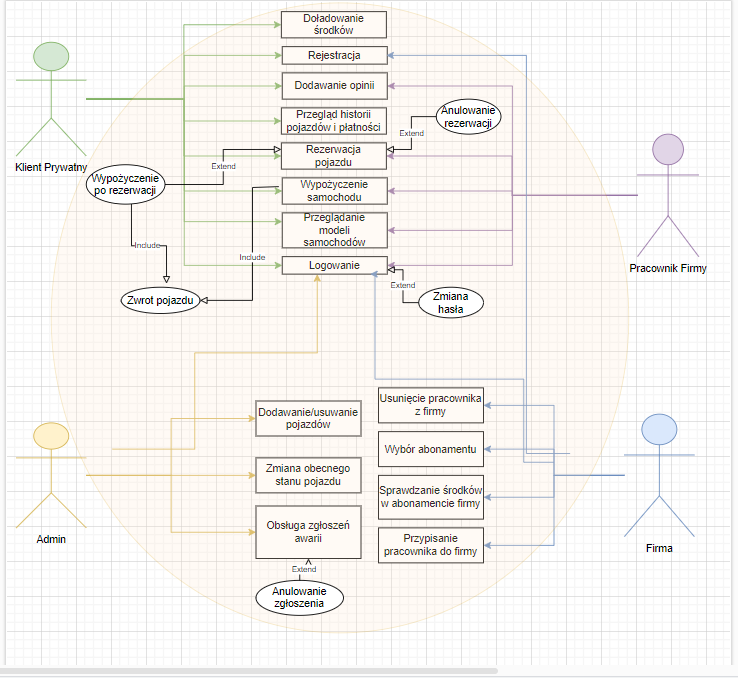
kończy wynajem samochodu, w tym momencie podliczane są wszelkie koszty, określana jest lokalizacja końcowa pojazdu oraz wszelkie zapytania o jakość usługi

* Obliczanie długości trasy, zużycia i kosztów paliwa,

funkcje, które będą określały wszelkie koszty i parametry przejazdu

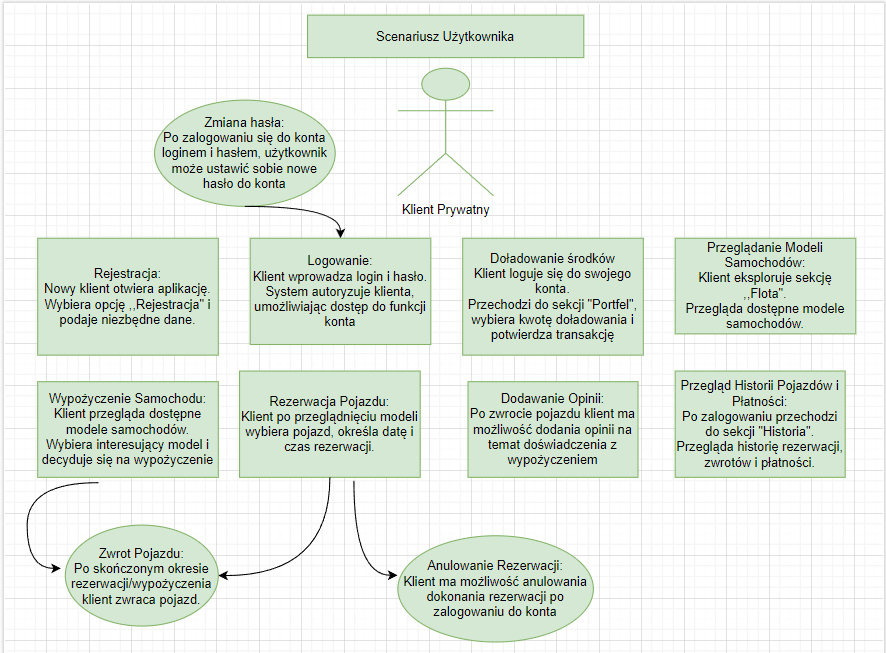
* Zgłaszanie uszkodzeń pojazdu,

w przypadku, gdy klient uszkodzi pojazd lub, gdy pojazd, który będzie chciał wypożyczyć będzie miał widoczne uszkodzenia musi mieć możliwość zgłoszenia

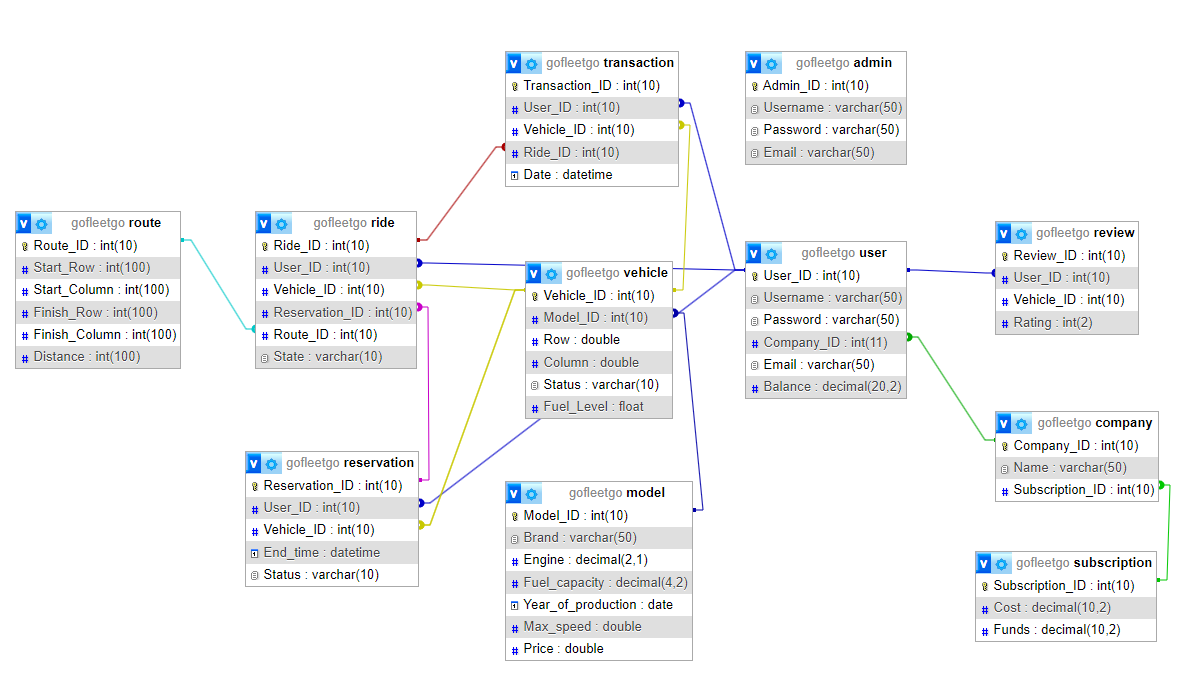
Diagram Przypadków Użycia   


# Scenariusze Użytkownika

## Klient Prywatny



# Schemat Bazy Danych



Komunikacja Klient-Serwer-Baza Danych

## Opis klas używanych do komunikacji:

* **DTO(Data Transfer Object)**  
  Wszystkie klasy w tej paczce dziedziczą po interfejsie Serializable. Pozwala nam to przekształcać je na sekwencje bitów i przesyłać pomiędzy klientem a serwerem i zdeserializować.
* **ClientRequest**   
  klasa w DTO package, która jest wysyłana do serwera.   
  Przekazujemy w niej:
  + **Action**, które zawiera informacje o tym jaką akcje chce wykonać użytkownik,
  + **Data,** dane, które mogą być zserializowane. Zawiera informacje potrzebne do wykonania danej akcji.
  + **PrivateToken,** zawiera informacje i indywidualnym tokenie, za pomocą którego możemy identyfikować użytkownika
* **ServerResponse**zawiera w sobie dane, które serwer zwraca do użytkownika, np.
  + **resultCode**, mówiący o rezultacie danej akcji,
  + **resultMessage**, zawiera komunikat, jeżeli chcemy przekazać użytkownikowi wiadomość,
  + **data**, zawiera dane dla użytkownika, które mogą być zdeserializowane.
* **NetworkClient**Metoda sendRequest() umożliwia nam wysłanie żądania do serwera i odebrania odpowiedzi.
* **ClientHandler**W tej klasie po odebraniu obiektu ClientRequest i odczytaniu go sprawdzamy jaką akcje chciał wykonać użytkownik, a następnie w bloku Switch Case wykonujemy odpowiednie operacje. Na koniec wysylamy odpowiedz do klienta.
* **ConnectionManager**otwiera połączenie z bazą danych. Wykorzystywany we wszystkich klasach służących do komunikacji z bazą danych.

## Schemat przekazywania danych

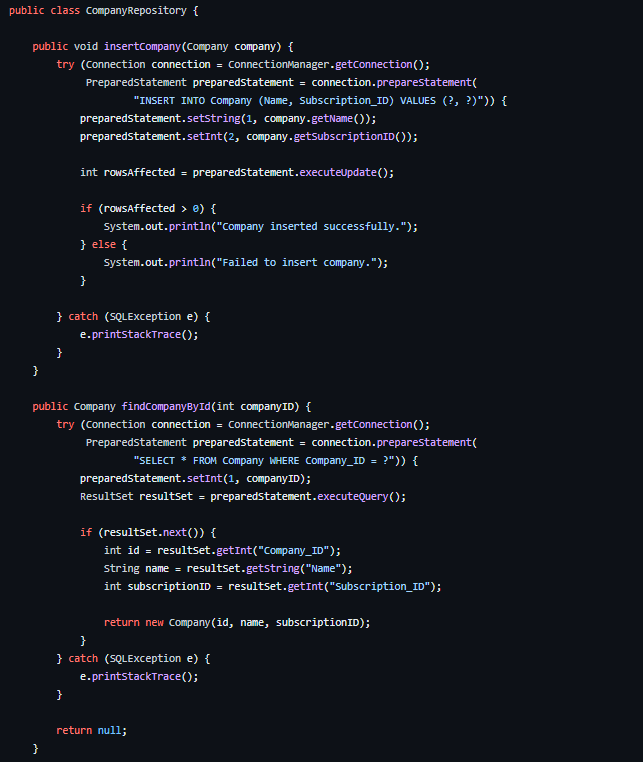
Tworzenie obiektu ClientRequest -> wysłanie obiektu przez sendRequest() -> odebranie obiektu w ClientHandler -> wykonanie akcji (komunikacja z Repository danej klasy lub/oraz odpowiedniego servisu) -> wysłanie odpowiedzi (ServerResponse) z ClientHandler -> sendRequest zwraca odpowiedź od serwera.

# Opis klas Repositories

Package Repositories zawiera klasy które odzwierciedlają tabele utworzone w bazie danych.  
Każda tabela posiada swoją klasę, w której stworzone są metody do komunikacji się serwera z bazą danych.

Każda funkcja korzysta z funkcji getConnection() klasy ConnectionManager, która otwiera połączenie z bazą i zwraca nam wartość connection.

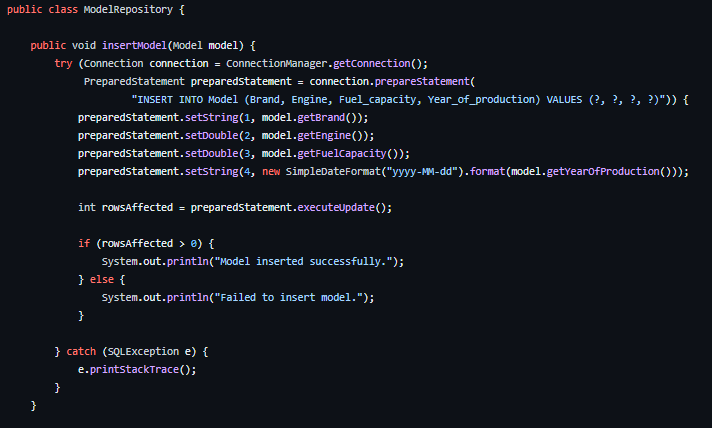
## Company

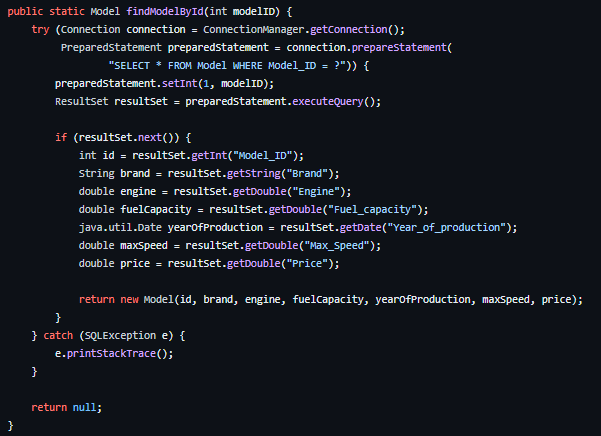


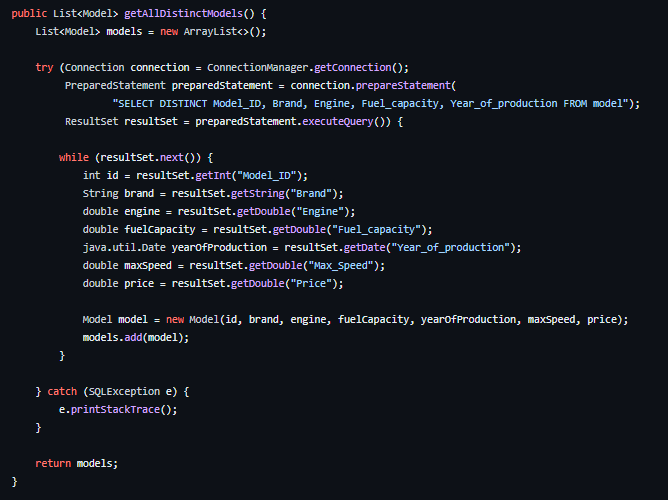
Zawiera dwie metody połączone z Bazą Danych:

* *insertCompany* służąca do wstawiania nowej firmy
* FindCompanyById umożliwiająca wyszukanie firmy po jej ID

## Model



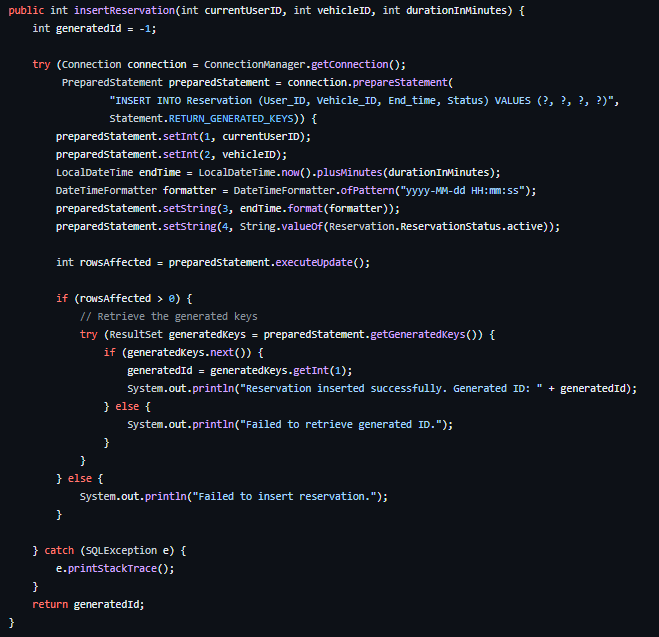
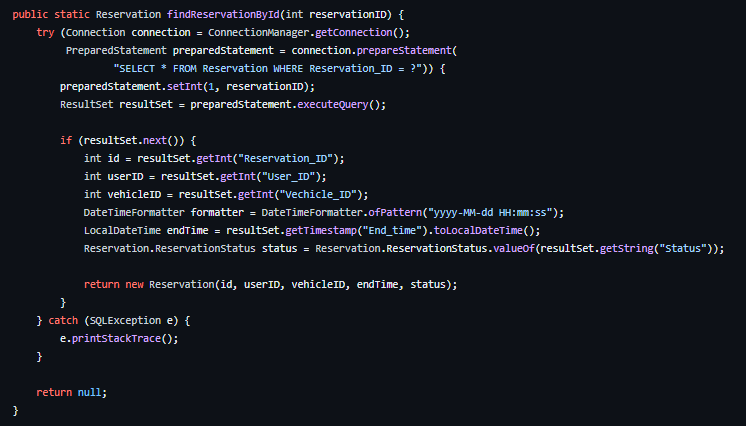
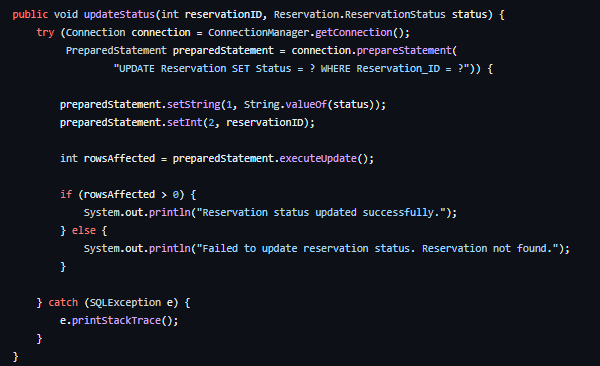




Zawiera metody:

* insertModel do dodawania pojazdów
* FindModelById do wyszukiwania pojazdów na podstawie podanego ID
* GetAllDistinctModels do wyszukiwania modeli bez powtarzania ich

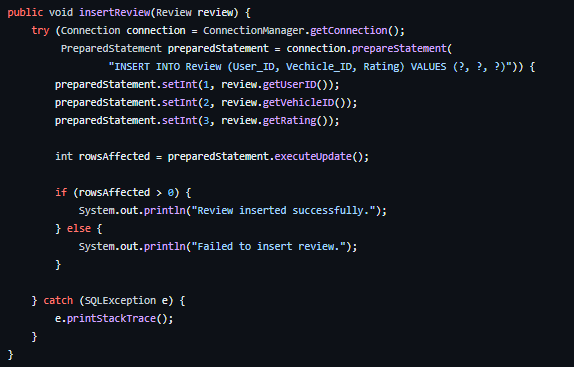
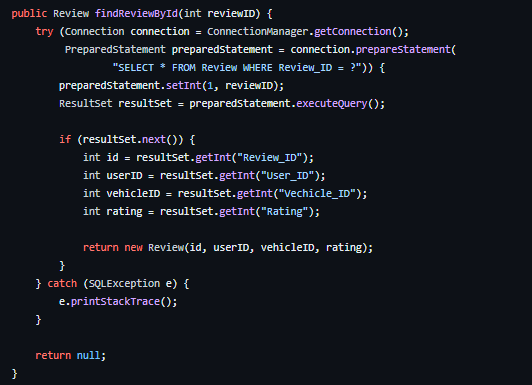
## Reservation

Metody:

* InsertReservation do wstawiania rezerwacji do Bazy
* FindReservationById do wyszukiwania rezerwacji
* UpdateReservationAndLinkedVehicleState odpowiada za aktualizację stanu rezerwacji i powiązanego pojazdu
* UpdateStatus odpowiedzialna za aktualizację statusu rezerwacji w bazie danych

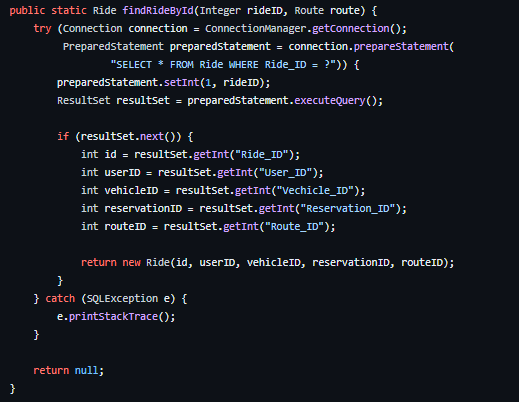
## Review

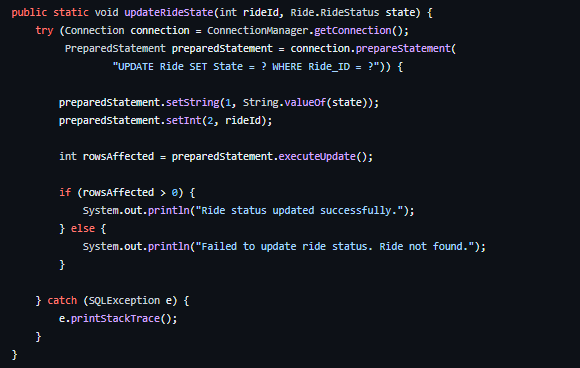
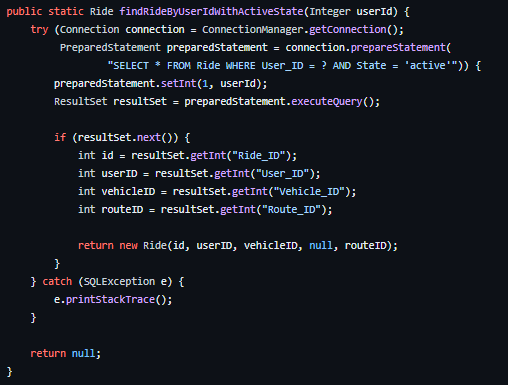
  


Metody:

* InsertReview służąca do dodawania oceny w Bazie Danych
* FindReviewById do wyszukiwania oceny po ID

## Ride

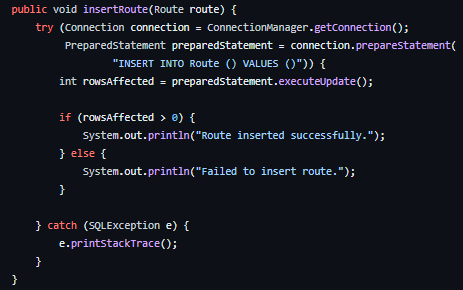
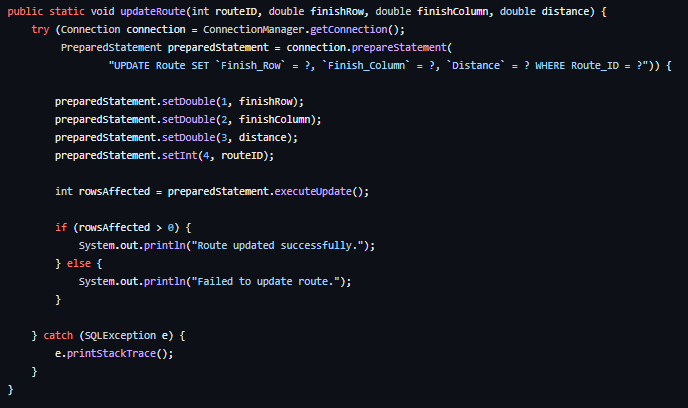
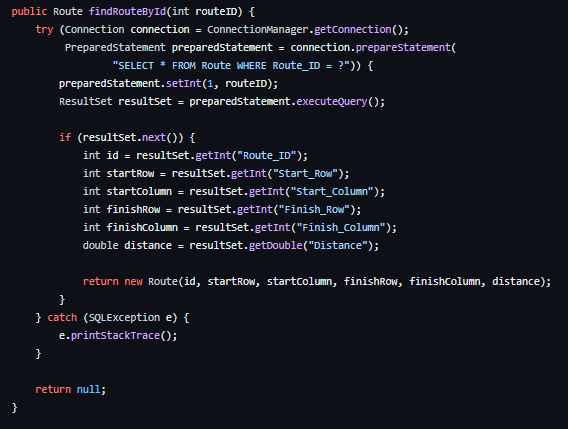
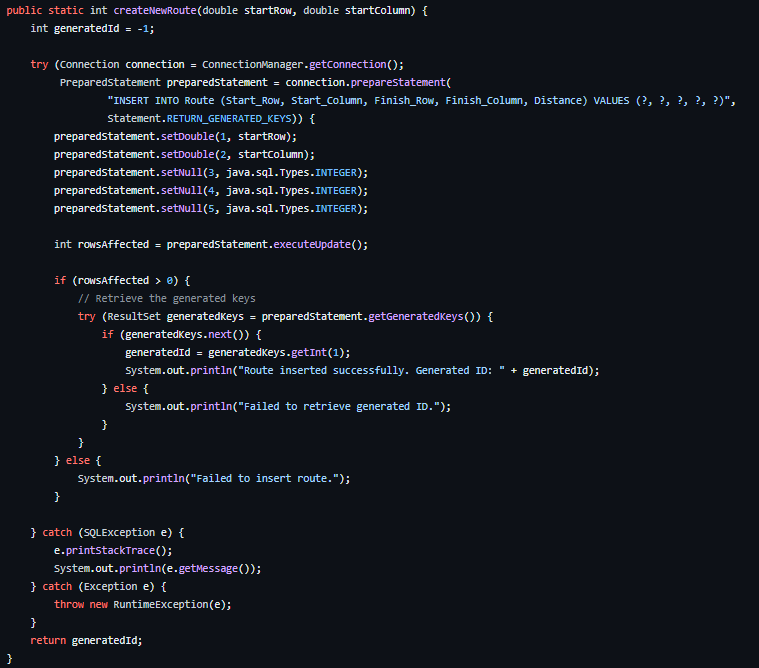
  


Metody:

* InsertRide do wstawiania przejazdu do Bazy
* FindRideById do wyszukiwania pojazdu w Bazie
* UpdateRideState do aktualizacji statusu przejazdu
* FindRideByUserIdWithActiveState do wyszukiwania aktywnych przejazdów

## Route

Metody:

* InsertRoute do aktualizacji trasy w Bazie Danych
* FindRouteById do odnajdywania w Bazie trasy
* UpdateRoute do aktualizacji trasy
* CreateNewRoute do rozpoczynania nowej trasy

## Subscription



Metody:

* InsertSubscription służąca do dodania nowej subskrypcji w Bazie
* FindSubscriptionById do wyszukiwania subskrypcji

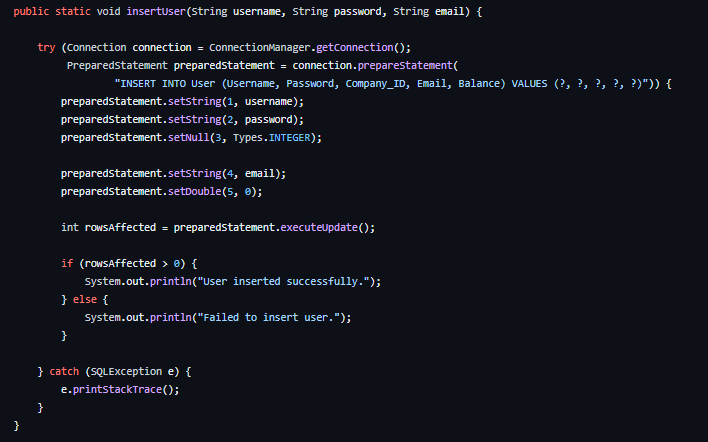
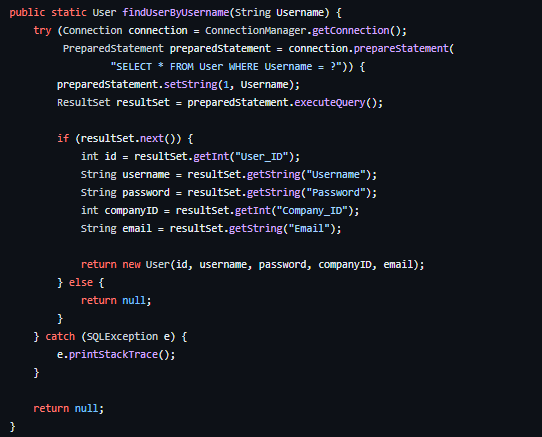
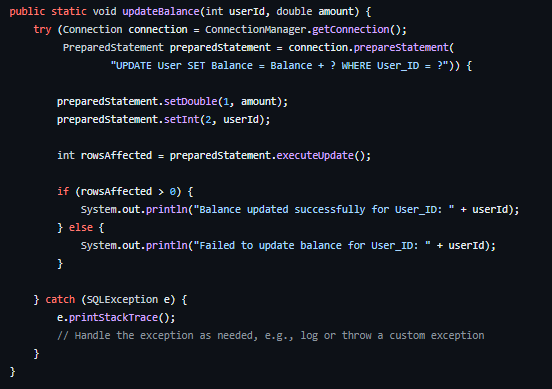
## Transaction



Metody:

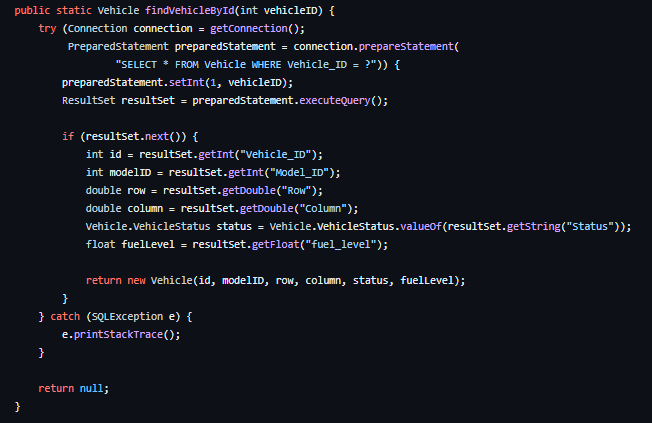
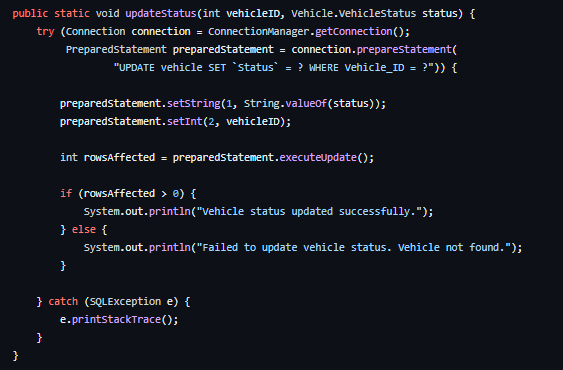
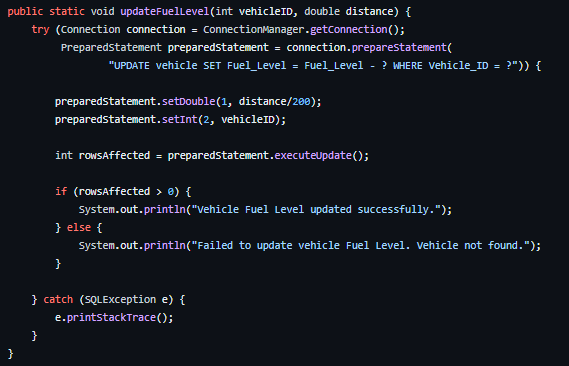
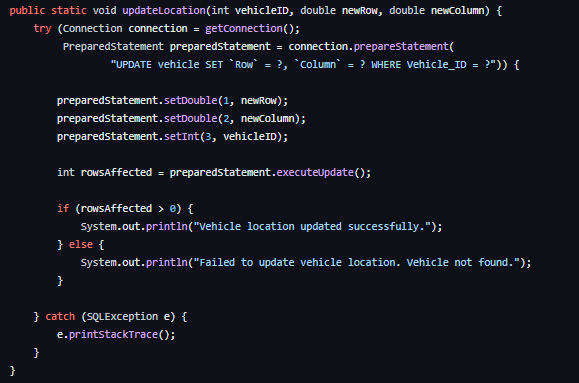
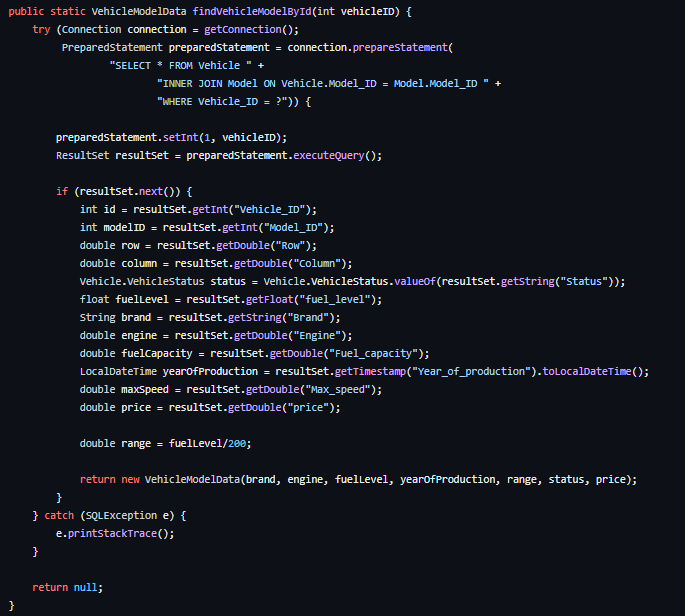
* InsertTransaction do wstawiania transakcji w Bazie Danych
* FindTransactionById do wyszukiwania danej transakcji po ID

## User

  
  
  
  
Metody:

* InsertUser do wstawiania nowego użytkownika do Bazy Danych
* FindUserById do wyszukiwania użytkowników po ich ID
* FindUserByUsername do wyszukiwania użytkowników po ich nazwie
* UpdateBalance do aktualizacji portfela użytkownika

## Vehicle

Metody:

* InsertVehicle do dodania pojazdu do Bazy Danych
* UpdateStatus do aktualizacji statusu dostępności pojazdu
* UpdateFuelLevel do aktualizacji stanu paliwa
* FindVehicleById do wyszukiwania pojazdu

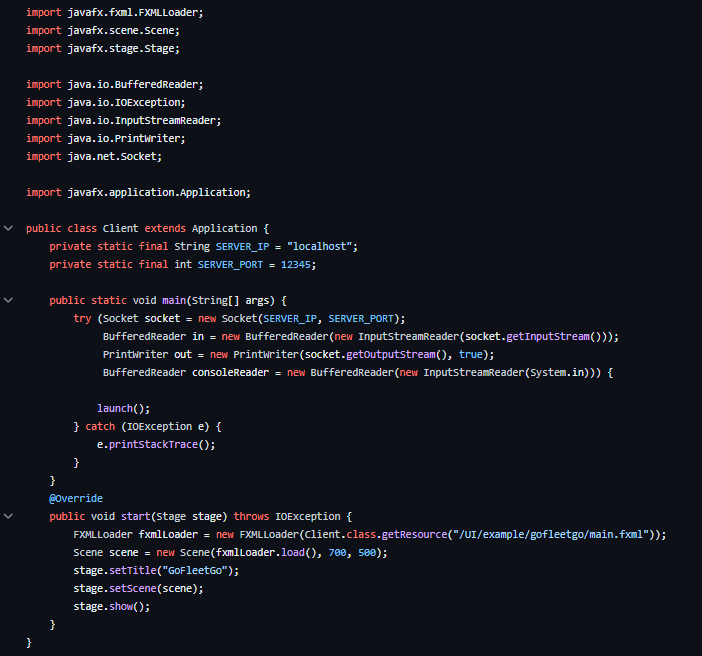
# TCP

## Server



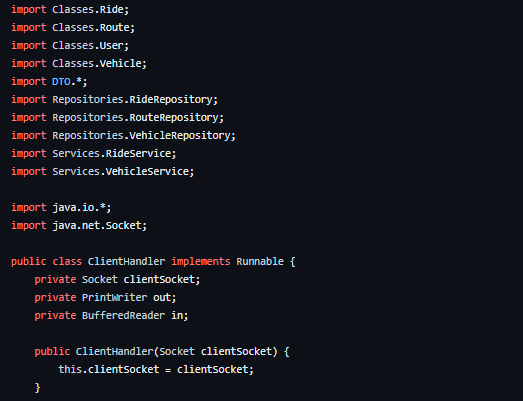
Inicjalizuje serwer na określonym porcie (w tym przypadku 12345).  
Tworzy pulę wątków (*ExecutorService*), która obsługuje klientów.  
Oczekuje na połączenia klientów w nieskończonej pętli.  
Po nawiązaniu połączenia, tworzy nowy wątek (*ClientHandler*) dla danego klienta i przekazuje go do puli wątków.  
Proces ten powtarza się, aby obsługiwać multipleksowanie klientów.

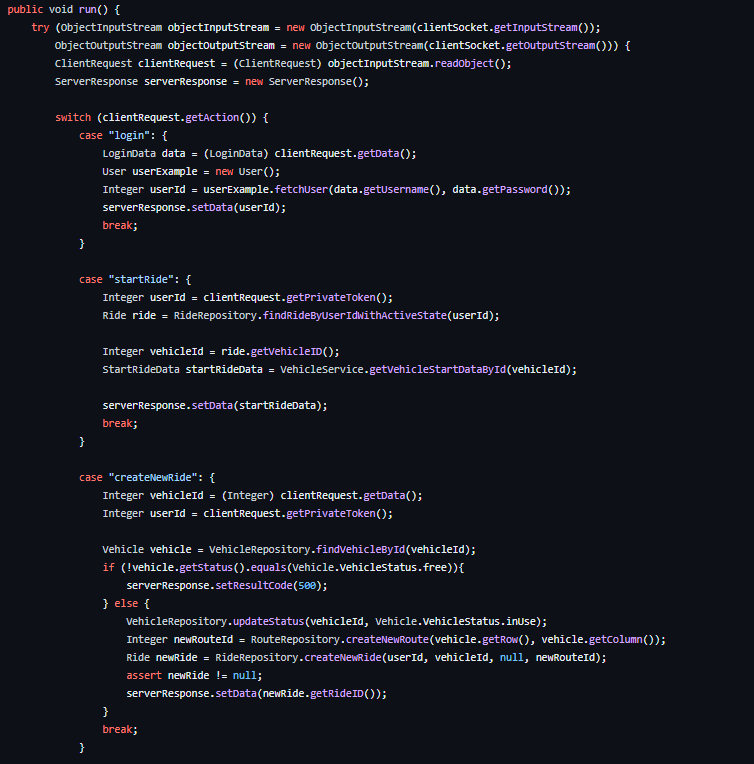
## Client

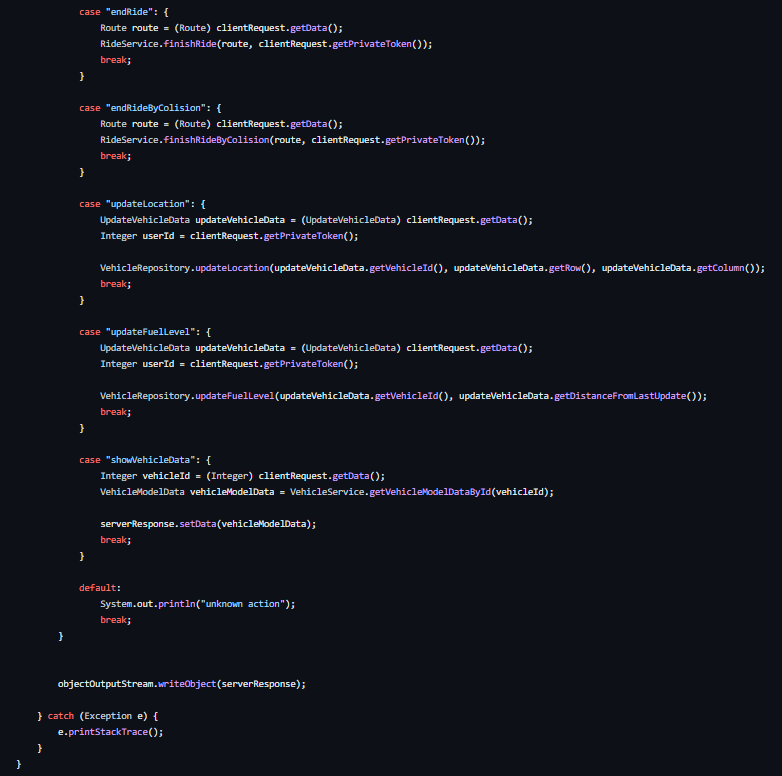


Inicjalizuje klienta, łącząc się z adresem IP (localhost) i portem (12345), na którym działa serwer.  
Tworzy strumienie danych (*BufferedReader* i *PrintWriter*) do komunikacji z serwerem.  
Rozpoczyna interakcję z serwerem, a następnie uruchamia aplikację *JavaFX*, wczytując odpowiedni widok (*main.fxml*).  
Okno interfejsu graficznego jest uruchamiane w oddzielnym wątku za pomocą metody *start*.

## ClientHandler







Metoda run klasy ClientHandler jest nadpisywana, ponieważ klasa implementuje interfejs Runnable. Co oznacza, że każde wywołanie tej klasy jest osobnym wątkiem, przez co obsługuje zapytania od klientów w ramach serwera w sposób wielowątkowy. W zależności od akcji przesłanej przez klienta, podejmuje odpowiednie działania. Poniżej przedstawiam opisy poszczególnych przypadków:

"*login*": Sprawdza dane logowania użytkownika, korzystając z klasy *User* i metody *fetchUser*. Następnie przekazuje identyfikator użytkownika do odpowiedzi serwera.

"*startRide*": Pobiera aktywną trasę użytkownika, korzystając z klasy *RideRepository*, a następnie uzyskuje dane początkowe do rozpoczęcia jazdy dla danego pojazdu.

"*createNewRide*": Tworzy nową trasę (*ride*) dla użytkownika, jeżeli wybrany pojazd jest dostępny. W przypadku niedostępności pojazdu, ustawia kod odpowiedzi serwera na 500 (błąd).

"*endRide*": Kończy trasę (*ride*) dla użytkownika, aktualizując stan pojazdu i trasy. Dane o trasie przesyłane są jako obiekt *Route*.

"*endRideByColision*": Podobna do "*endRide*", ale z dodatkowym uwzględnieniem kolizji. Zakończenie trasy oznacza również wyłączenie pojazdu i naliczenie dodatkowych kosztów związanych z kolizją.

"*updateLocation*": Aktualizuje lokalizację pojazdu na podstawie danych przesłanych przez klienta.

"*updateFuelLevel*": Aktualizuje poziom paliwa pojazdu na podstawie danych przesłanych przez klienta.

"*showVehicleData*": Przeszukuje dane dotyczące modelu pojazdu na podstawie przesłanego identyfikatora pojazdu i przekazuje te dane do odpowiedzi serwera.

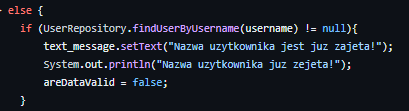
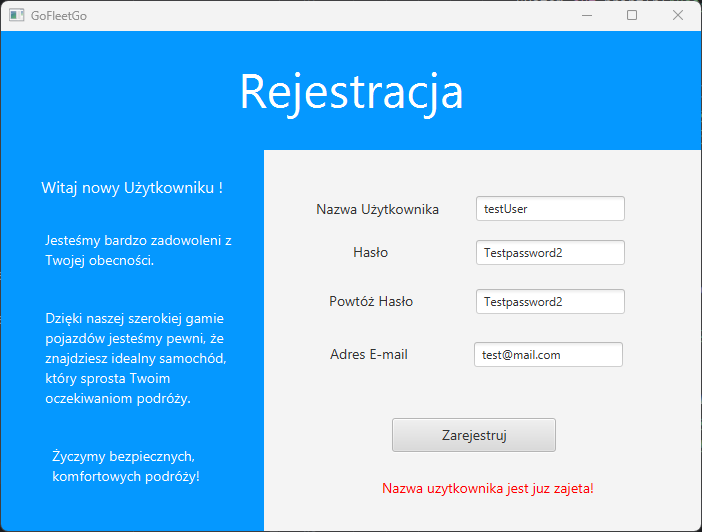
# UI – Logowanie, Rejestracja, Menu

## Rejestracja

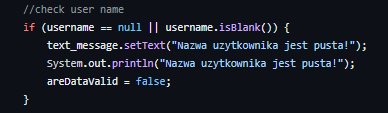
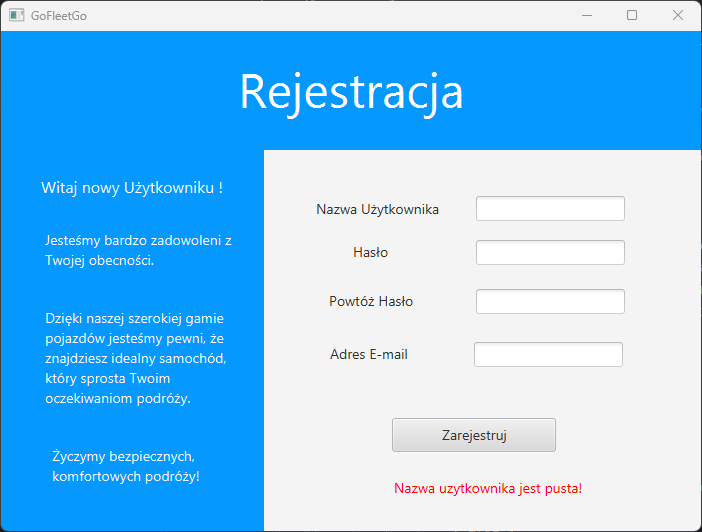
Proces rejestracji w systemie odbywa się w klasie *RegisterController,* a dokładniej mówiąc odpowiada za to metoda *changeToMain*, która zawiera w sobie walidację danych uzupełnianych przez klientów aplikacji. By utworzyć konto trzeba spełnić następujące wymagania.

## Sprawdzanie poprawności danych tworzonego konta

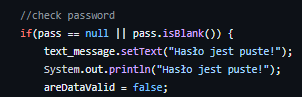
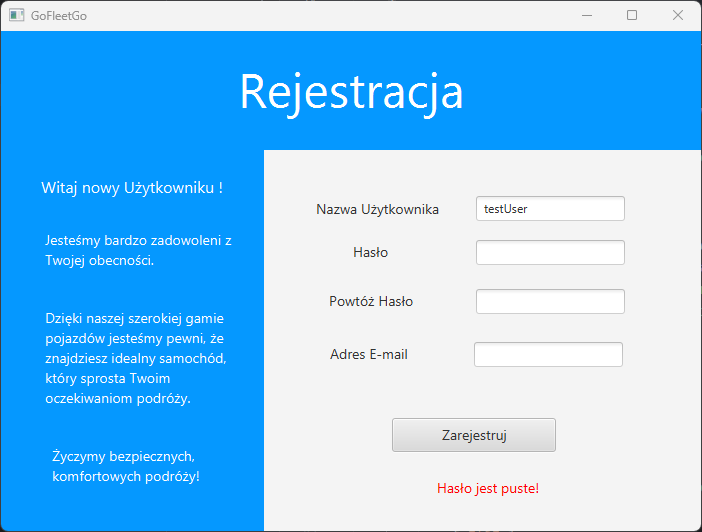
Nazwa nie może znajdować się już w systemie:

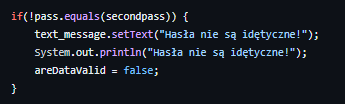
Nazwa nie może być pusta:

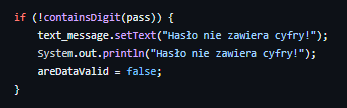
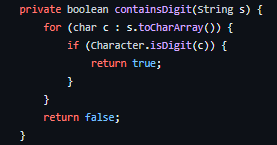
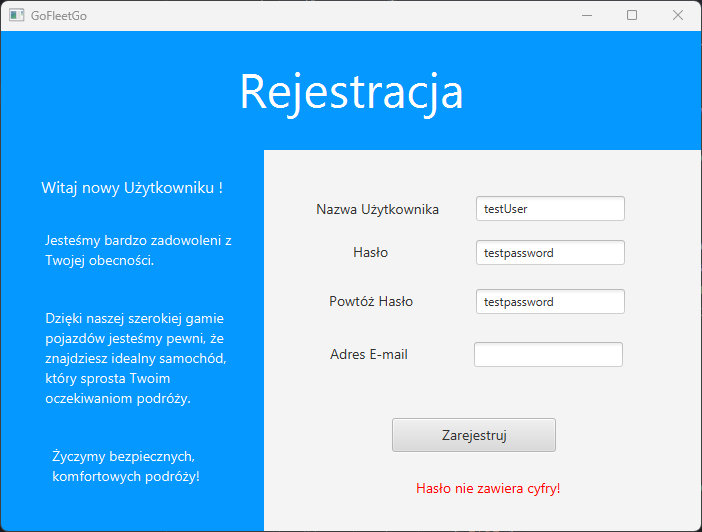
Hasło nie może być puste:

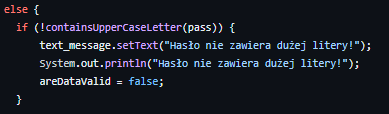
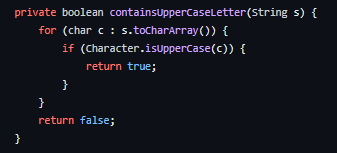
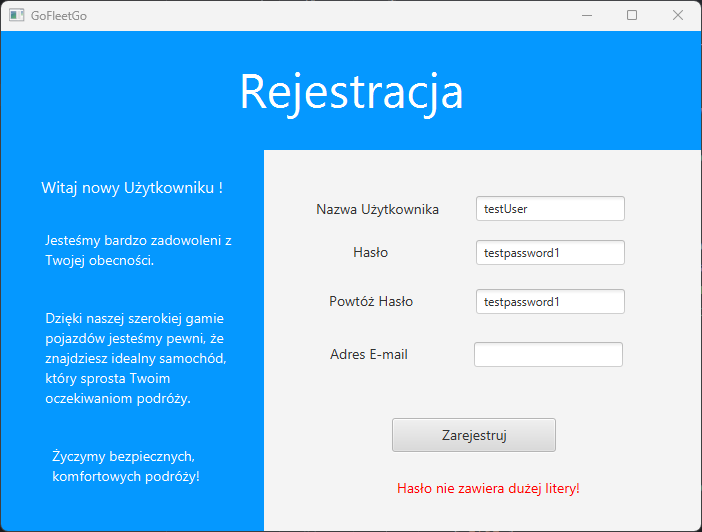
Hasła muszą być zgodne:

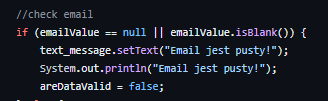
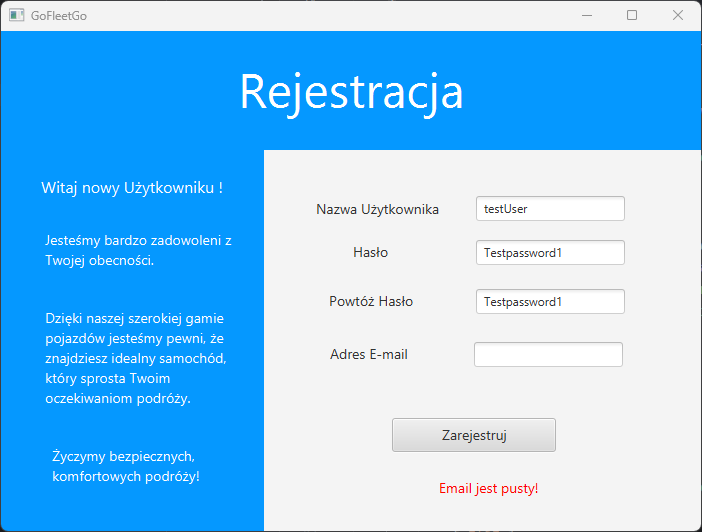

Hasło musi zawierać cyfre:

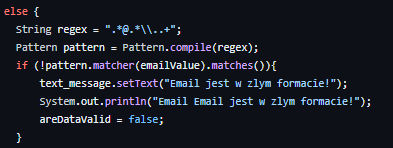
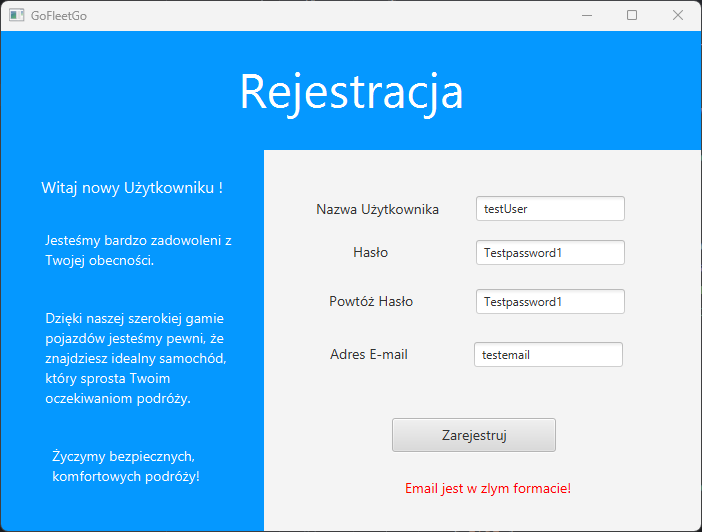
Hasło składa się z dużej litery:

Email jest wymagany:

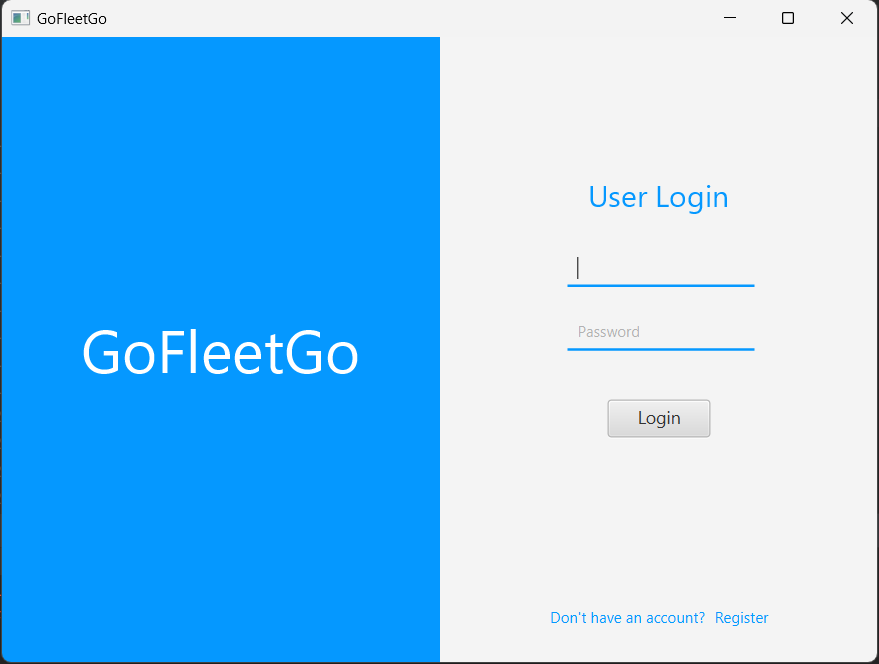
  


Email musi być w odpowiednim formacie (nazwa@domena.tekst):

## Logowanie

W widoku logowania posiadamy pola Login oraz Password, przycisk Login oraz możliwość rejestracji poprzez wciśnięcie słowa Register



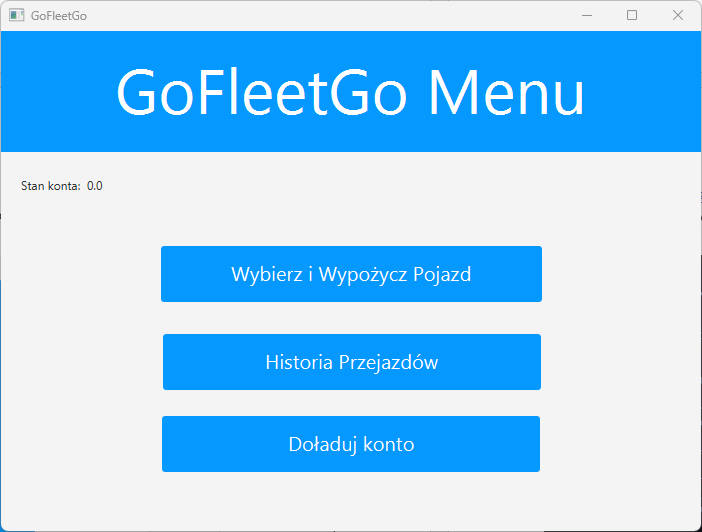
## Weryfikacja danych logowania

Poprawne logowanie do systemu weryfikuje klasa *User*, a dokładnie metoda znajdująca się w niej *fetchUser:*



Metoda *fetchUser* sprawdza, czy podane dane logowania (nazwa użytkownika i hasło) zgadzają się z danymi przechowywanymi w bazie danych MySQL. Po nawiązaniu połączenia z bazą danych, wyszukiwany jest użytkownik o podanej nazwie. Jeśli użytkownik istnieje, sprawdzane jest, czy podane hasło jest zgodne z poprawnym hasłem użytkownika. W przypadku zgodności, użytkownik jest uważany za zalogowanego, ustawiając *isLogged* na *true* i zwracając ID użytkownika. W przeciwnym razie *isLogged* ustawiane jest na *false* i zwracany jest komunikat o niepoprawnych danych logowania.

## Menu



Po pomyślnym zalogowaniu wyświetla się menu programu z możliwością wyboru:

* *Wybierz i Wypożycz Pojazd  
  Które pozwala nam przejść do kolejnego widoku z dostępnymi pojazdami*
* *Historia Przejazdów  
  Niestety ta funkcjonalność nie została jeszcze dokończona*
* *Doładuj konto  
  Przenosi nas do następnego widoku, gdzie możemy doładować swój stan portfela*

# Wypożyczenie samochodu



W tym widoku wyświetlamy wszystkie dostępne w obecnym momencie pojazdy wraz z kompletnymi informacjami na ich temat.

Pojazdy mogą znajdować się w 3 stanach:

* Free – gotowy do wypożyczenia
* InUse – obecnie użytkowany przez innego użytkownika
* Disabled – pojazd wyłączony z użytku

Aby wynająć samochód musimy wybrać dany model i wcisnąć przycisk Rozpocznij Przejazd.  
W tym momencie sprawdzamy, czy użytkownik ma odpowiednią ilość pieniędzy oraz czy stan pojazdu jest odpowiedni.





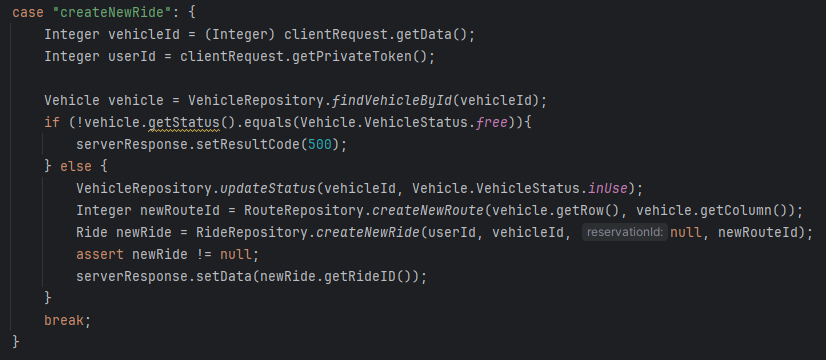


Wciskając przycisk **Rozpocznij Przejazd** uruchamiamy funkcję **sendIdOfCar()**, w której określamy Id wybranego przez użytkownika pojazdu oraz rozpoczynamy proces sprawdzania czy użytkownik może wynająć pojazd.

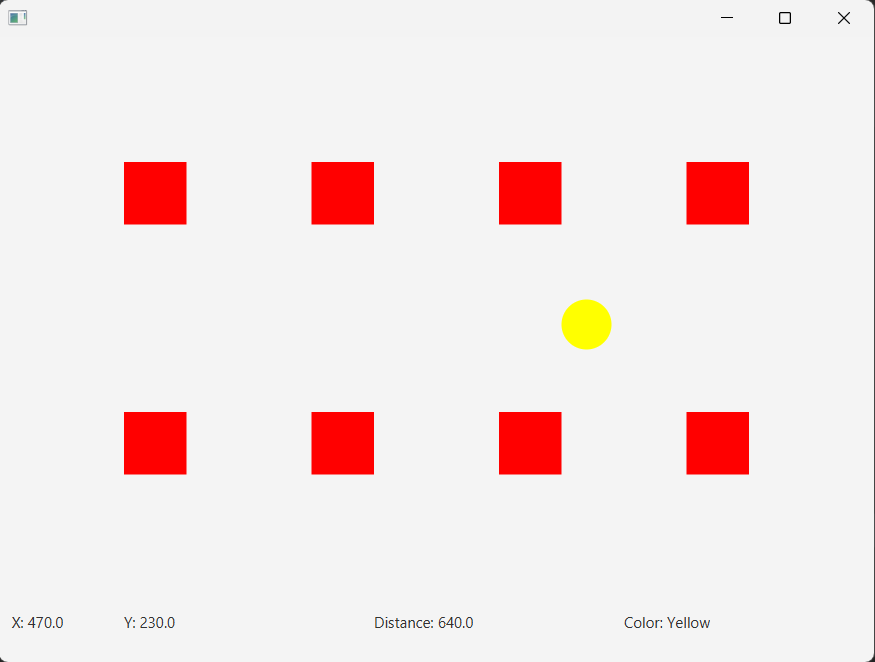
**Sprawdzenie pieniędzy użytkownika**

Wykonujemy akcje „showUserBalance”, aby serwer zwrócił nam ilość pieniędzy jaką posiada obecnie użytkownik.  
Sprawdzamy tą wartość, aby wypisać komunikat błędu lub przejść do sprawdzenia stanu pojazdu.  
  
**Sprawdzanie stanu pojazdu**

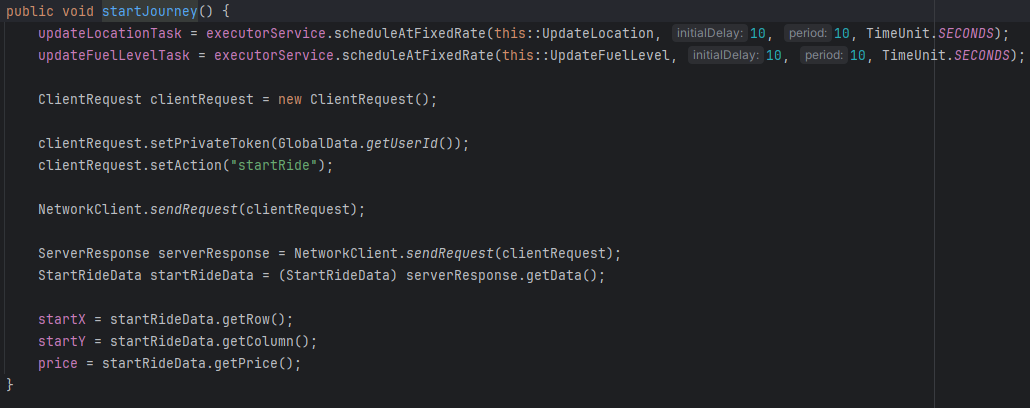
Wykonujemy akcje „createNewRide”. Która tworzy nowy Ride w bazie danych i przenosi nas do następnego widoku, lub jeżeli samochód nie znajduje się w stanie „free” zwraca nam resultCode 500, co oznacza, że akcja się nie udała i wypisuje komunikat błędu.



## Widok Poruszania Pojazdem



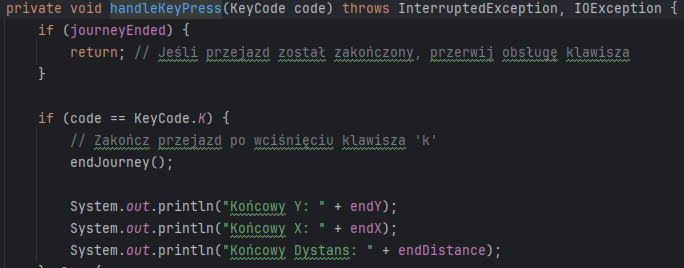
Na początku inicjalizujemy widok oraz wywołujemy funkcję startJourney()

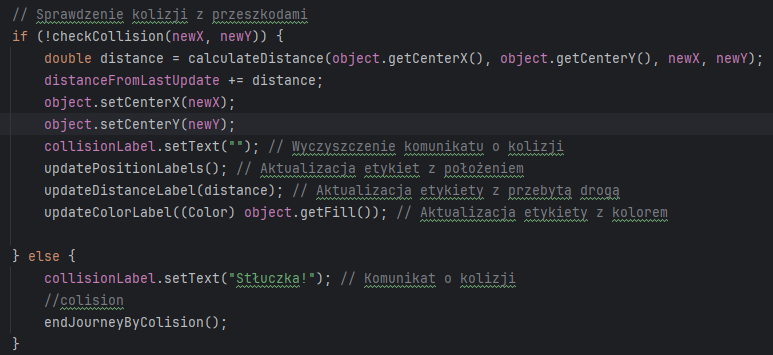


W tej funkcji na początku uruchamiamy funkcje aktualizujące stan paliwa oraz aktualną lokalizację.  
Następnie wysyłamy do serwera akcję startRide z aktualnym Id klienta, serwer zwraca nam aktualną pozycję samochodu oraz cenę za każdy przejechany kilometr.

Za pomocą setOnKeyPressed odczytujemy jakiego przycisku aktualnie używa klient oraz czy nie zakonczyl on przejazdu.  


Przejazd kończymy przyciskiem „K”

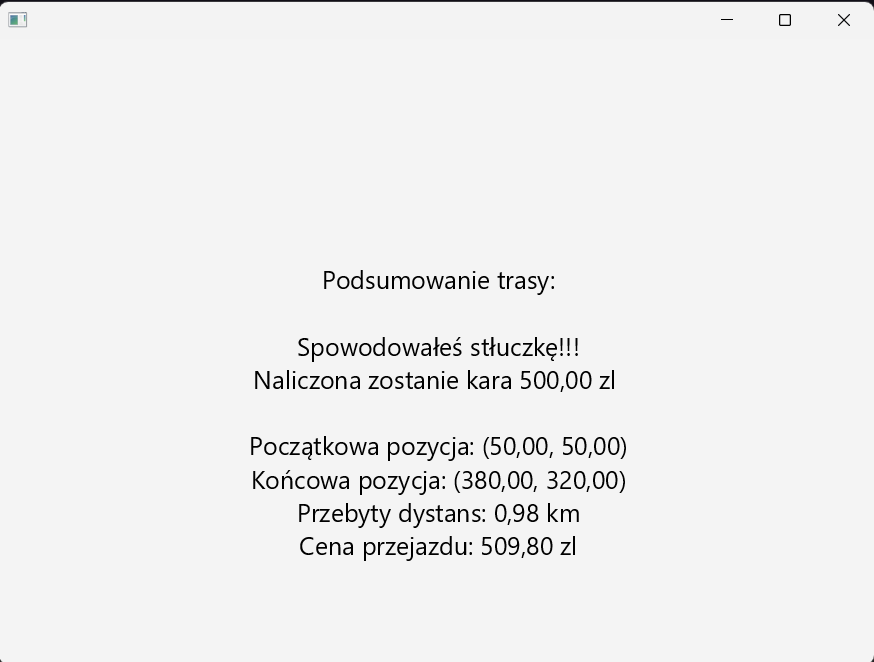




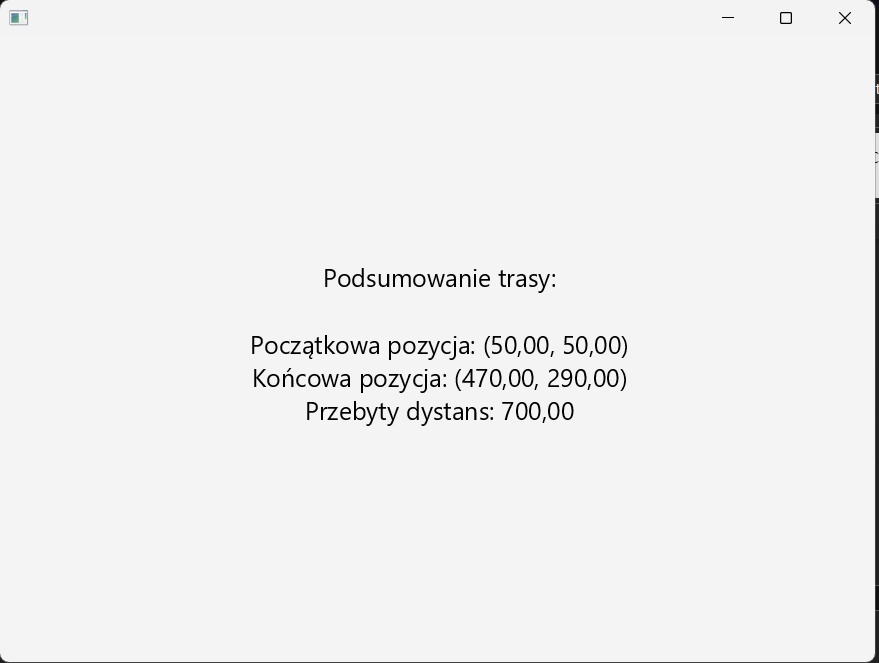
Jeżeli wystąpiła stłuczka kończymy przejazd odpowiednią funkcją, jeżeli nie to obliczamy pokonany dystans od ostatniego odświeżenia, aktualizujemy aktualną pozycję oraz aktualizujemy wyświetlany tekst.

Sterowanie pojazdem odbywa się za pomocą klawiszy strzałek. Czerwone bloki są to budynki znajdujące się na mapie, gdy uderzymy w nie autem pojawia się komunikat o stłuczce:

Przykład ekranu końcowego po stłuczce



Przykład ekranu końcowego po poprawnie zakończonym przejeździe



## Funkcje kończące przejazd

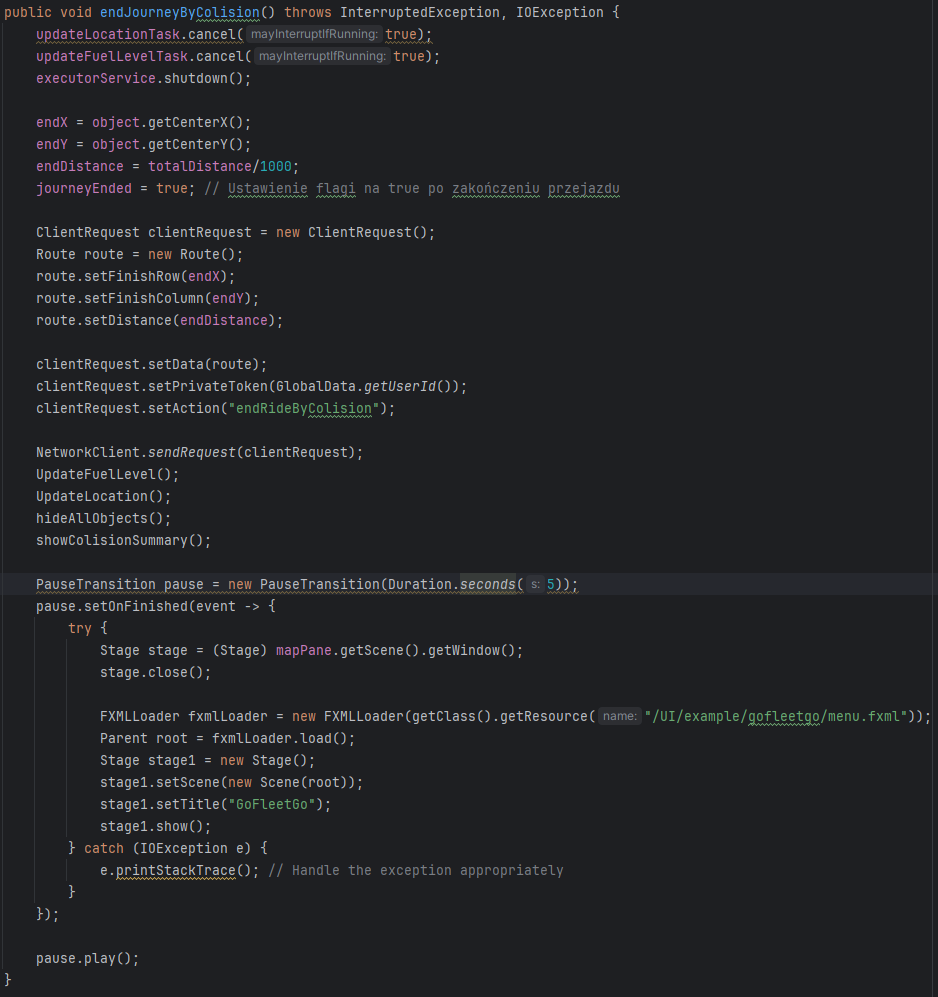
Posiadamy dwa przypadki kończenia przejazdu:

* Kolizja
* Umyślne zakończenie przejazdu

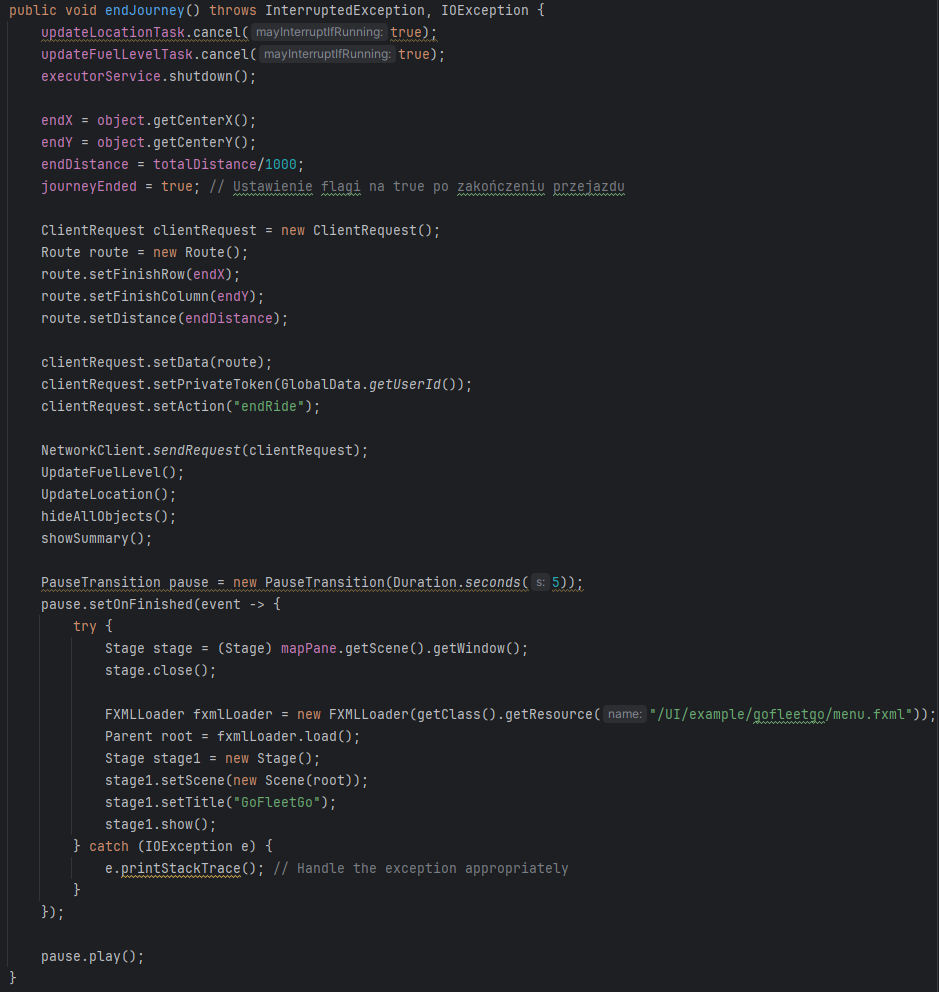
W obu przypadkach schemat jest taki sam. Najpierw zamykamy wątki, które aktualizują stan paliwa i lokalizjację. Odczytujemy najbardziej aktualną pozycję samochodu, obliczamy całkowity dystans, komunikujemy do serwera zakończenie przejazdu i aktualizujemy ostatecznie dane na temat paliwa i lokalizacji oraz wyświetlamy podsumowanie przejazdu.

Jedyną różnicą pomiędzy dwoma przypadkami jest nagłówek akcji który wysyłamy do serwera oraz dodatkowy koszt przy stłuczce.

**Przykład kodu dla kolizji:**



**Przykład kodu dla umyślnego zakończenia przejazdu:**



## Aktualizacja lokalizacji i stanu paliwa po zakończonym przejeździe

  
Metoda *UpdateLocation* jest odpowiedzialna za aktualizację lokalizacji pojazdu w systemie. Tworzy zapytanie do serwera, w którym przesyła dane trasy, ustawiając koordynaty docelowe na podstawie aktualnej lokalizacji obiektu. Następnie wysyła to zapytanie, informując serwer o zmianie lokalizacji pojazdu.

Metoda *UpdateFuelLevel* ma za zadanie aktualizować poziom paliwa pojazdu. Podobnie jak wcześniej, generuje zapytanie do serwera, przekazując dane trasy, ale w tym przypadku odległość przejechaną od ostatniej aktualizacji paliwa. Po wysłaniu zapytania, zeruje wartość *distanceFromLastUpdate*, aby przygotować się do zbierania nowych danych o przejechanym dystansie. W ten sposób serwer otrzymuje informacje o aktualnym stanie paliwa pojazdu.

**Użycie**

Funkcje te są używane podczas kończenia przejazdu, ale także są wywoływane na początku przejazdu.

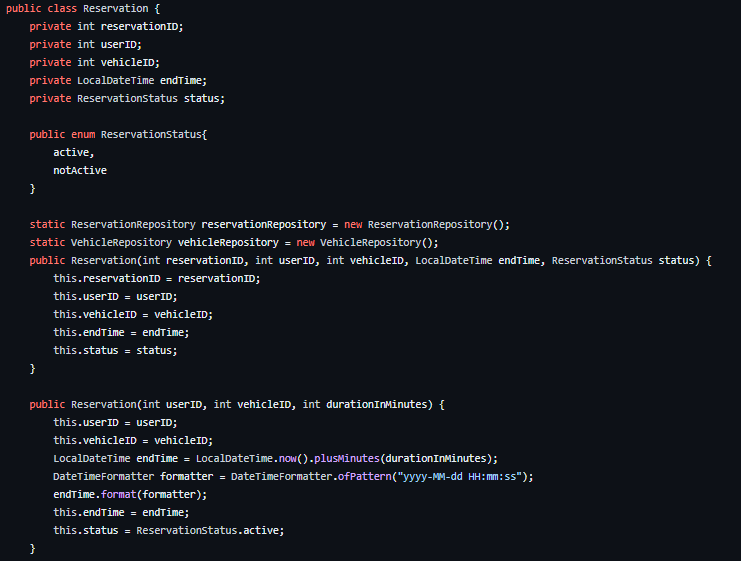
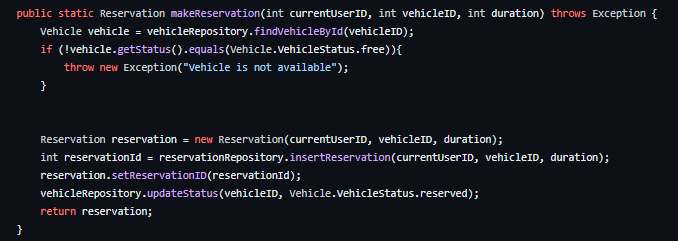


Tworzymy wtedy wątki, które co 10 sekund wywołują dane funkcje, co pozwala nam na bieżąco odświeżać stan paliwa i lokalizację, tak aby informacje w bazie danych były jak najbardziej aktualne.

## Rezerwacja samochodu

Finalnie ta funkcjonalność nie została wprowadzona dla klientów, ale rozpoczęliśmy jej implementacje.

Za rezerwacje samochodu odpowiedzialna jest klasa *Reservation:*

  
  
Przechowuje ona zmienne, które wykorzystuje podczas dokonywania rezerwacji pojazdu.

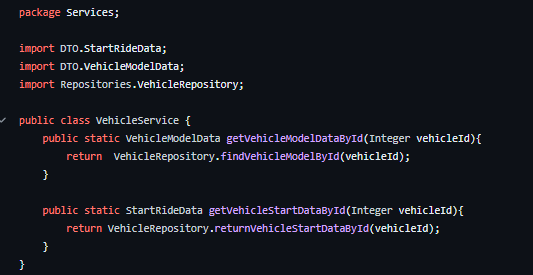
Pierwszy konstruktor służy do tworzenia obiektu rezerwacji na podstawie przekazanych parametrów, takich jak *reservationID, userID, vehicleID, endTime i status*.  
Drugi konstruktor służy do tworzenia rezerwacji na podstawie *userID, vehicleID* i czasu trwania rezerwacji w minutach.

Metoda *makeReservation* umożliwiająca utworzenie nowej rezerwacji w systemie. Przyjmuje trzy parametry: *currentUserID* (ID zalogowanego użytkownika), *vehicleID* (ID pojazdu do zarezerwowania) oraz *duration* (czas trwania rezerwacji w minutach).  
Metoda inicjalizuje proces rezerwacji, sprawdzając dostępność pojazdu za pomocą *findVehicleById* z klasy *VehicleRepository*. Jeśli pojazd nie jest dostępny (czyli nie jest w stanie *free*), generowany jest wyjątek z informacją o niedostępności pojazdu.  
W przypadku dostępności pojazdu, tworzona jest nowa rezerwacja poprzez drugi konstruktor klasy *Reservation*, który automatycznie oblicza czas zakończenia rezerwacji dodając do bieżącego czasu określony czas trwania. Następnie nowa rezerwacja jest dodawana do bazy danych za pomocą metody *insertReservation* z klasy *ReservationRepository,* zwraca nam ona ID nowo utworzonej rezerwacji, dzięki temu możemy sprawdzić ID przy wypożyczeniu samochodu wcześniej zarezerwowanego.



# Services

## VehicleService



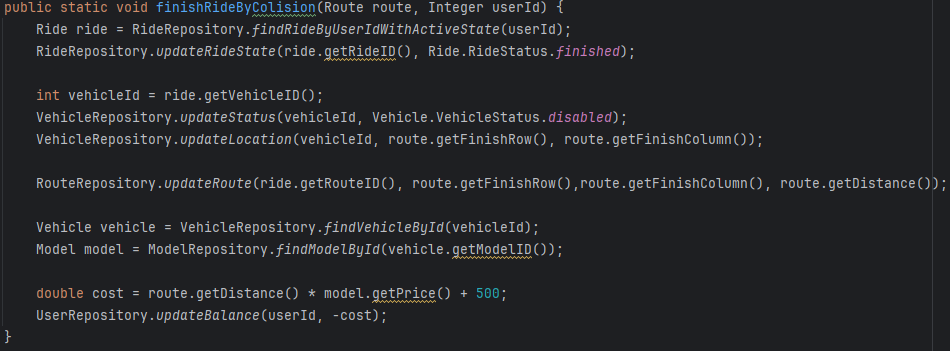
Metoda *getVehicleModelDataById* odpowiada za pobranie danych modelu pojazdu na podstawie jego identyfikatora (vehicleId). Wykorzystuje do tego metodę *findVehicleModelById* z klasy VehicleRepository.

Metoda *getVehicleStartDataById* odpowiada za pobranie danych początkowych dla rozpoczęcia jazdy danym pojazdem na podstawie jego identyfikatora (*vehicleId)*. Wykorzystuje do tego metodę *returnVehicleStartDataById* z klasy *VehicleRepository*.

## RideService



Metoda *finishRide* służy do zakończenia trwającej jazdy. Najpierw pobiera informacje o aktualnie trwającej jeździe użytkownika (o identyfikatorze *userId*), a następnie aktualizuje różne parametry, takie jak stan jazdy, status pojazdu, lokalizacja, poziom paliwa, trasa oraz obciąża konto użytkownika kosztem przejechanego dystansu na podstawie danych o trasie (*Route*). W tym celu korzysta z metod różnych repozytoriów, takich jak *RideRepository, VehicleRepository, RouteRepository, i UserRepository*.



Metoda **finishRideByColision** odpowiada za zakończenie bieżącej jazdy użytkownika w przypadku kolizji. Po zidentyfikowaniu aktywnej jazdy użytkownika, aktualizuje jej status na "finished" i dezaktywuje powiązany pojazd, ustawiając jego status na "disabled". Następnie, na podstawie danych trasy z kolizji, aktualizuje lokalizację pojazdu, trasę i oblicza koszt jazdy, uwzględniając odległość i cenę modelu pojazdu. Warto zauważyć, że do kosztu dodawana jest także dodatkowa opłata za kolizję. Ostatecznie, saldo użytkownika jest aktualizowane o obliczony koszt jazdy, co skutkuje pełnym i spójnym zakończeniem jazdy w przypadku kolizji.

# Wnioski

Najwięcej problemów podczas tworzenia tego projektów spowodowała nam zdecydowanie komunikacja Klient – Serwer i połączenie jej z UI. Był to nasz pierwszy projekt wykorzystujący takie połączenie co przełożyło się na dużą ilość błędów i problemów logicznych.

Największym błędem było złe rozplanowanie pracy, komunikację klient-server zostawiliśmy jako ostatni etap co przełożyło się na duża ilość zmian i poprawek w istniejącym już kodzie.

Ciekawym wyzwaniem był również widok, w którym użytkownik sam steruje swoim pojazdem