Aplikacja do zarządzania flotą pojazdów



Kacper Mazur

Informatyka i ekonometria

131378

Spis treści

1. Wprowadzenie i ogólne założenia projektu
2. Instalacja
3. Interfejs użytkownika
4. Funkcjonalność
5. Bezpieczeństwo
6. Dokumentacja API
7. Przydatne Informacje

**1.Wprowadzenie i ogólne założenia projektu:**

Celem aplikacji jest usprawnienie zarządzania flotą pojazdów w firmach transportowych oraz przedsiębiorstwach posiadających flotę samochodową. Aplikacja umożliwia efektywne zarządzanie danymi dotyczącymi pojazdów, kierowców, przeglądów technicznych, ubezpieczeń oraz informacji o kierowcach. Aplikacja JavaFX połączona z bazą danych umożliwiająca zarządzanie flotą pojazdów w firmie transportowej w celu uproszczenia logistycznego, zabieg ten ma na celu na celu łatwiejsze wyszukiwanie konkretnego pojazdu/pojazdów, kierowców, ubezpieczeń i przeglądów w celu sprawdzenia stanu jego przeglądu technicznego i informacji dodatkowych. Aplikacja została zaprojektowana jako narzędzie dla administratorów floty, którzy mogą za jej pomocą dodawać, usuwać oraz wprowadzać dane w przejrzystym interfejsie graficznym. Kluczowym elementem aplikacji jest jej zdolność do przechowywania informacji w relacyjnej bazie danych, co zapewnia integralność danych oraz możliwość łatwego ich przetwarzania i analizy. Poprzez dostępne w aplikacji pole powiadomień użytkownik na bieżąco informowany jest o nieważnym ubezpieczeniu oraz przeglądzie w celu prostej weryfikacji konieczności odnowienia tych danych. Dodatkowo do aplikacji zaimplementowana zostanie tabela kierowcy, która znacząco otworzy możliwości dalszej rozbudowy projektu.

Wymagania systemowe:

**Urządzenia i platformy** Aplikacja została zaprojektowana do pracy na komputerach stacjonarnych oraz laptopach z zainstalowanym systemem operacyjnym Windows, macOS lub Linux. Została napisana w języku Java, co zapewnia jej przenośność między platformami, pod warunkiem, że urządzenie spełnia wymagania środowiska uruchomieniowego Javy.

**Ograniczenia technologiczne** Technologia Javy wymaga zainstalowanego środowiska Java Runtime Environment (JRE) w wersji 8 lub wyższej. Nie przewiduje się obecnie wsparcia dla urządzeń mobilnych ani przeglądarek internetowych, ponieważ aplikacja została stworzona jako aplikacja desktopowa z wykorzystaniem JavaFX.

**Wymagania sprzętowe i programowe**

* Procesor: Minimum Intel Core i3 lub odpowiednik.
* Pamięć RAM: Minimum 4 GB (zalecane 8 GB dla lepszej wydajności).
* Miejsce na dysku: Minimum 500 MB wolnego miejsca.
* System operacyjny: Windows 10/11, macOS Catalina lub wyższy, dowolna dystrybucja Linuxa z bibliotekami obsługującymi Javę.
* Zainstalowane oprogramowanie:
  + Java Development Kit (JDK) w wersji 8 lub wyższej.
  + MySQL w wersji 8.0 lub wyższej do obsługi bazy danych.
  + IntelliJ IDEA lub inne IDE do ewentualnej modyfikacji kodu źródłowego.

**Opis tabel**

1. **Tabela: kierowcy**
   * **PESEL** (bigint): Klucz główny jednoznacznie identyfikujący kierowcę.
   * **Imię** (varchar): Imię kierowcy.
   * **Nazwisko** (varchar): Nazwisko kierowcy.
   * **DataUrodzenia** (date): Data urodzenia kierowcy, potrzebna do identyfikacji wieku.
   * **NumerPrawoJazdy** (varchar): Numer prawa jazdy, wymagany do weryfikacji uprawnień.
   * **Uprawnienia** (varchar): Kategorie prawa jazdy posiadane przez kierowcę.

Typy danych wybrano z uwagi na ich przeznaczenie – varchar pozwala na przechowywanie tekstu, a bigint zapewnia odpowiednią przestrzeń dla numeru PESEL.

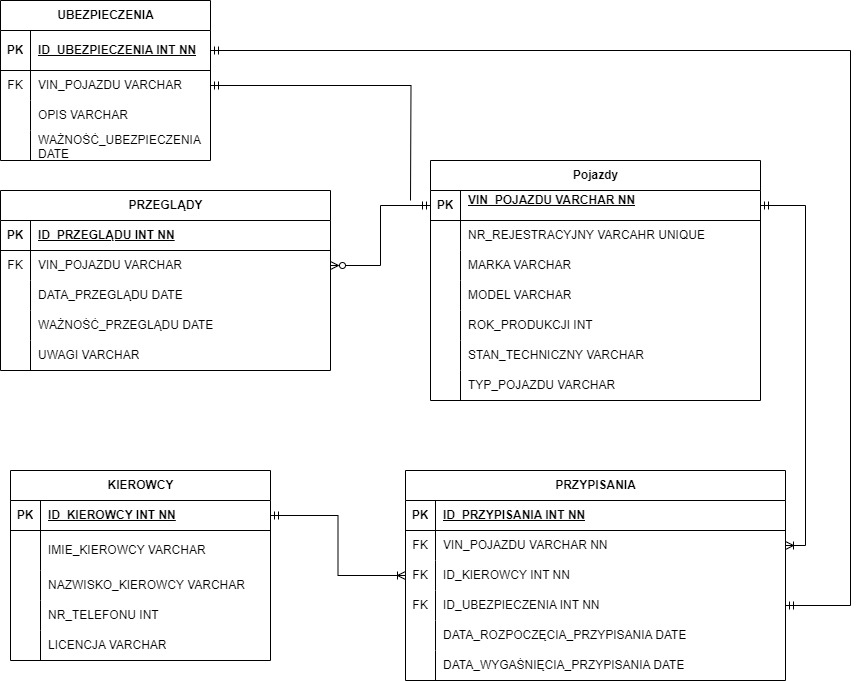
1. **Tabela: pojazdy**
   * **PojazdID** (int): Klucz główny.
   * **NumerVIN** (varchar): Numer identyfikacyjny pojazdu (unikalny).
   * **Marka** (varchar): Marka pojazdu.
   * **Model** (varchar): Model pojazdu.
   * **RokProdukcji** (int): Rok produkcji pojazdu.
   * **NumerRejestracyjny** (varchar): Numer rejestracyjny pojazdu.
   * **StanTechniczny** (varchar): Opis stanu technicznego.
   * **TypPojazdu** (varchar): Typ pojazdu (np. ciągnik siodłowy, samochód osobowy).

Wybrano varchar dla tekstu, int dla wartości liczbowych, co zapewnia optymalną wydajność.

1. **Tabela: przegladtechniczny**
   * **PrzegladID** (int): Klucz główny.
   * **PojazdID** (int): Klucz obcy wskazujący na tabelę pojazdy.
   * **DataPrzegladu** (date): Data wykonania przeglądu technicznego.
   * **WynikPrzegladu** (varchar): Wynik przeglądu (np. pozytywny, negatywny).
   * **Uwagi** (varchar): Dodatkowe informacje dotyczące przeglądu.
2. **Tabela: przypisania**
   * **PrzypisanieID** (int): Klucz główny.
   * **PojazdID** (int): Klucz obcy wskazujący na tabelę pojazdy.
   * **PESEL** (bigint): Klucz obcy wskazujący na tabelę kierowcy.
   * **DataPrzypisania** (date): Data przypisania pojazdu do kierowcy.
3. **Tabela: ubezpieczenia**
   * **UbezpieczenieID** (int): Klucz główny.
   * **PojazdID** (int): Klucz obcy wskazujący na tabelę pojazdy.
   * **RodzajUbezpieczenia** (varchar): Typ ubezpieczenia (np. OC, AC).
   * **DataRozpoczęcia** (date): Data rozpoczęcia ubezpieczenia.
   * **DataZakończenia** (date): Data zakończenia ubezpieczenia.
   * **Koszt** (decimal): Koszt ubezpieczenia.

Tabele zaprojektowano tak, aby wspierały spójność danych oraz łatwość ich przetwarzania w aplikacji.

**Graficzne przedstawienie relacji i tabel projektu.**



**2. Instalacja**

**1. Instrukcja instalacji**

**Niezbędne oprogramowanie**

1. **Java Development Kit (JDK)**
   * Minimalna wersja: 8 lub nowsza
2. **XAMPP Controll Panel lub MySQL Server**
   * Minimalna wersja: 5.7 lub nowsza
3. **IDE (zintegrowane środowisko programistyczne)**
   * Przykład: IntelliJ IDEA (Community lub Ultimate) lub Eclipse IDE
4. **Git (opcjonalne)**
   * Do zarządzania wersjami kodu.

**Kroki instalacji**

1. Pobierz i zainstaluj **JDK**. Upewnij się, że zmienna środowiskowa JAVA\_HOME wskazuje na katalog instalacyjny JDK.
2. Zainstaluj **XAMPP Control Panel** lub inne środowisko do zarządzania bazami danych MySql.
3. Zainstaluj wybrane **IDE** (np. IntelliJ IDEA) i otwórz w nim projekt aplikacji.

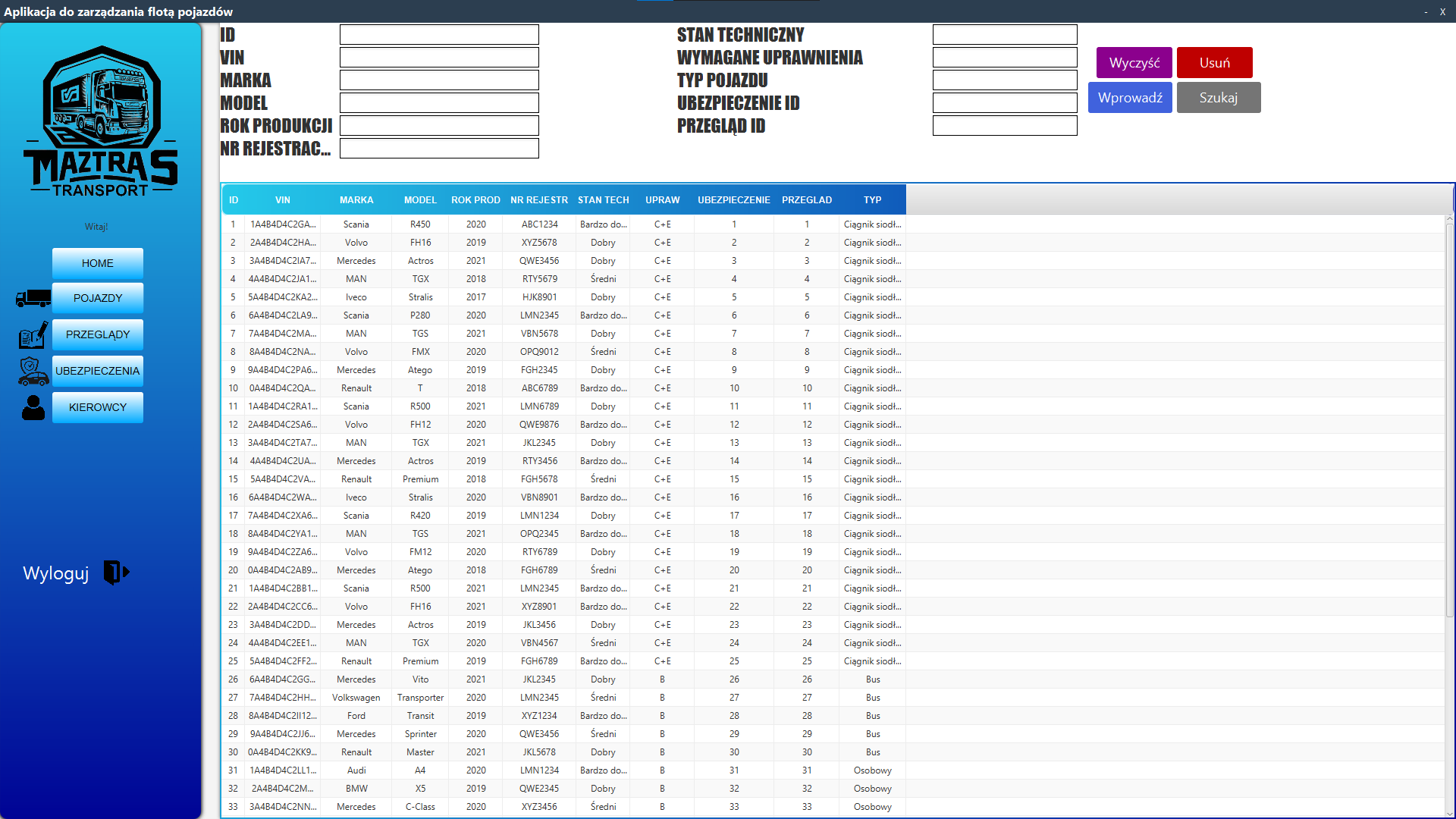
**3.Interfejs Użytkownika**

Wygląd graficzny i podział na okna:

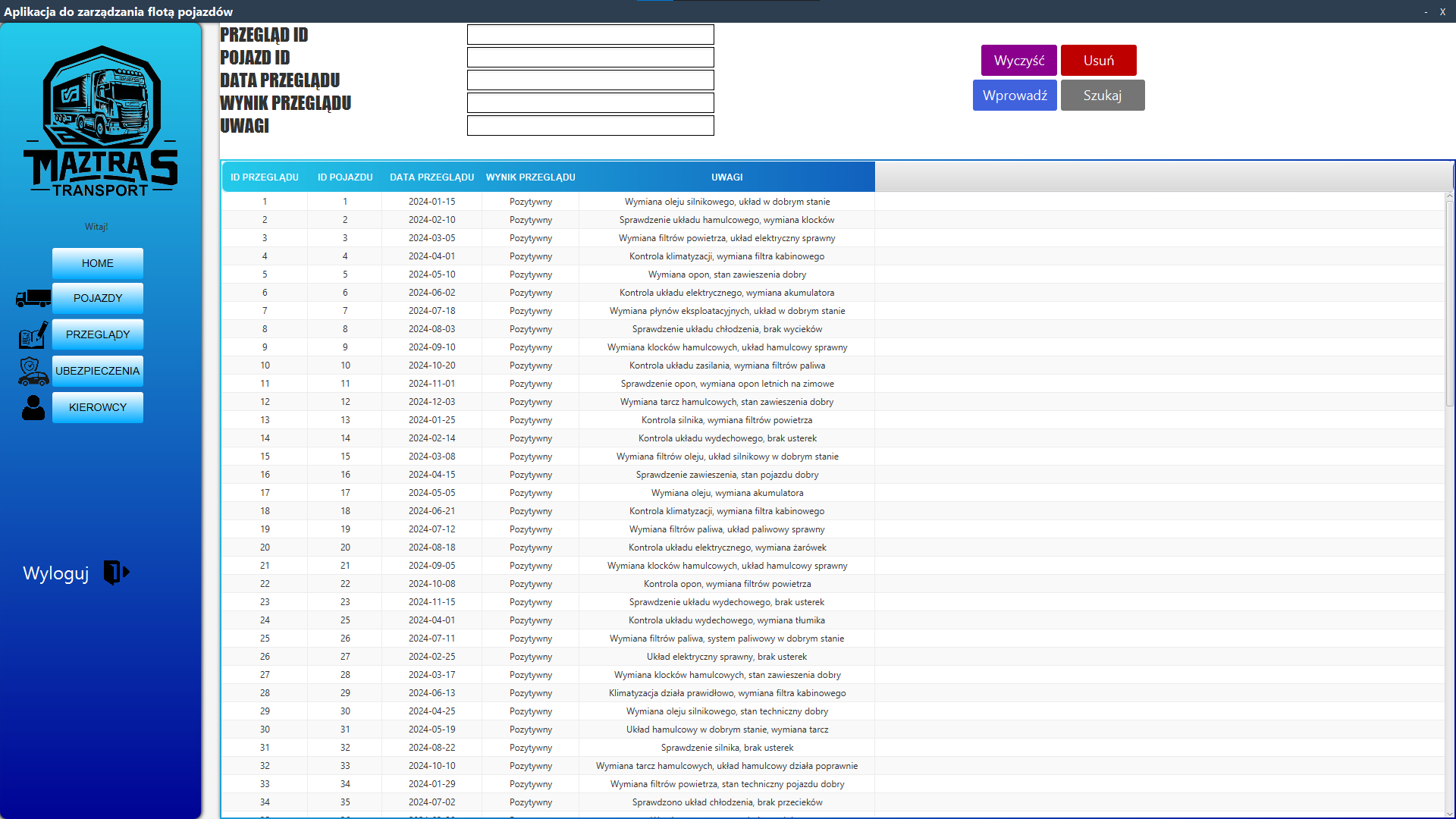
1. Logowanie - początkowe, startowe okno weryfikujące dane wpisane przez użytkownika



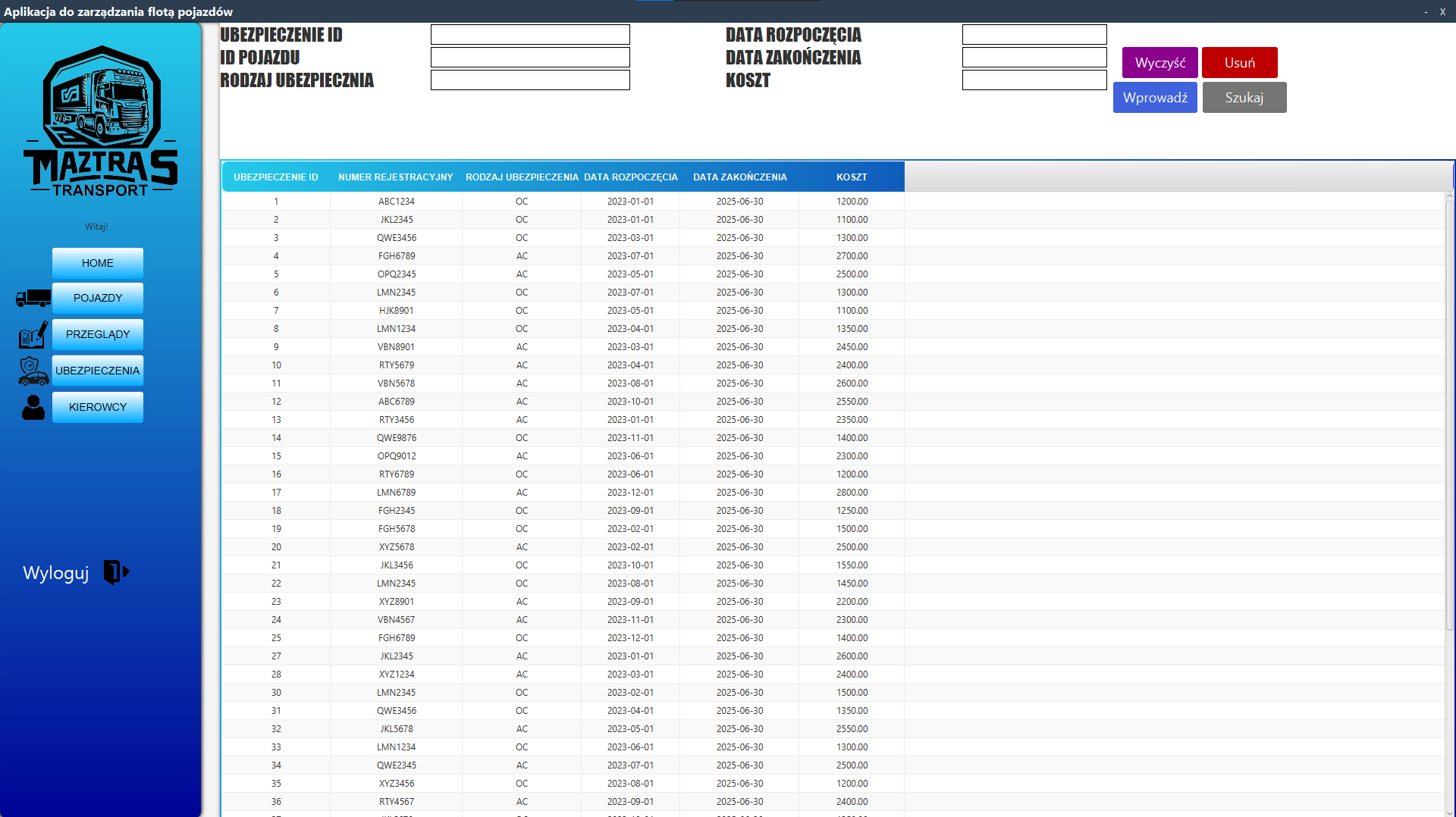
1. Pojazdy - zakładka pozwalająca sprawdzić informacje danym pojeździe bądź grupie pojazdów



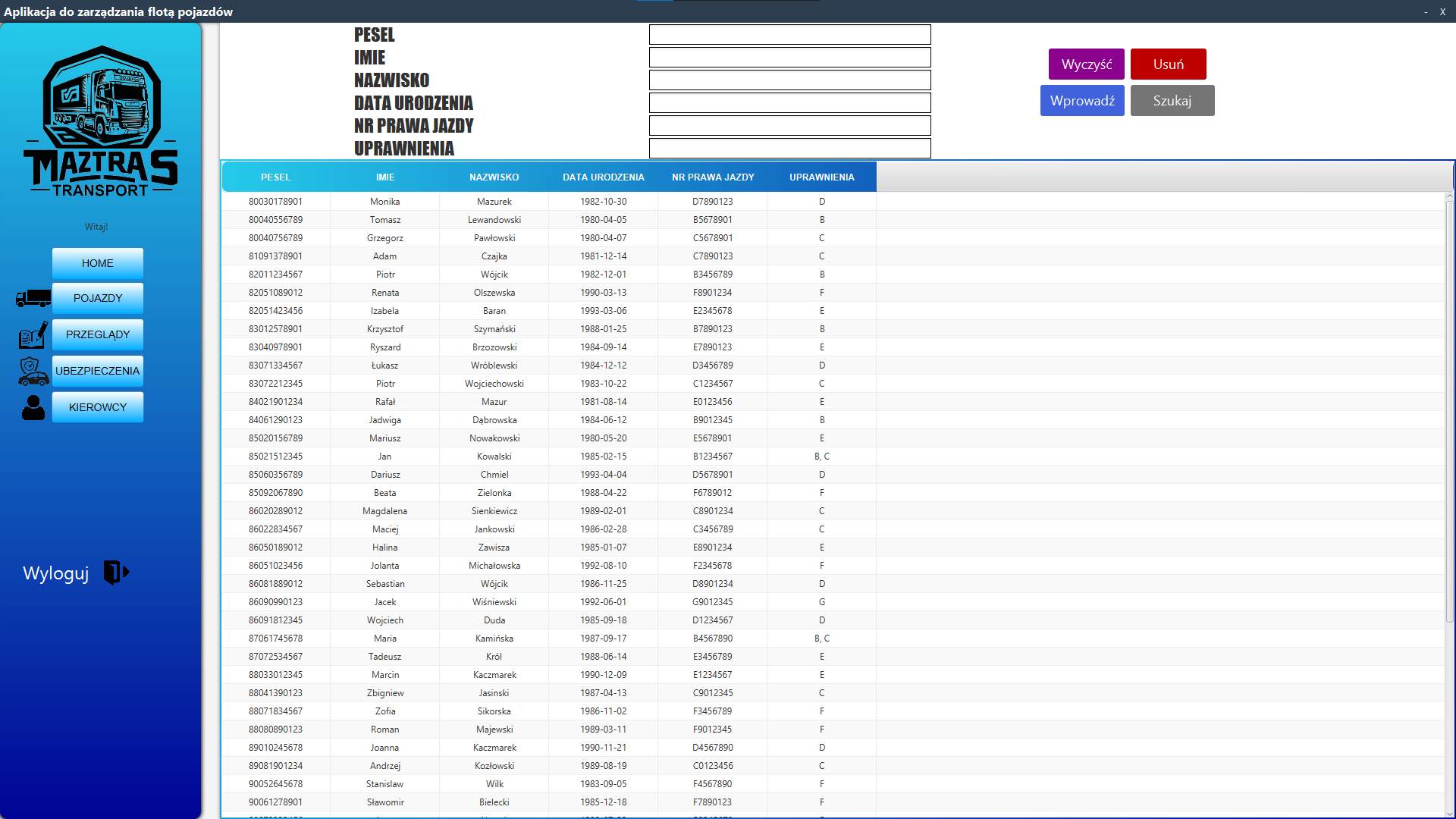
1. Przegląd - zakładka przedstawiać będzie dogłębne informacje o przeglądzie i ubezpieczeniu konkretnego pojazdu bądź grupie pojazdów.



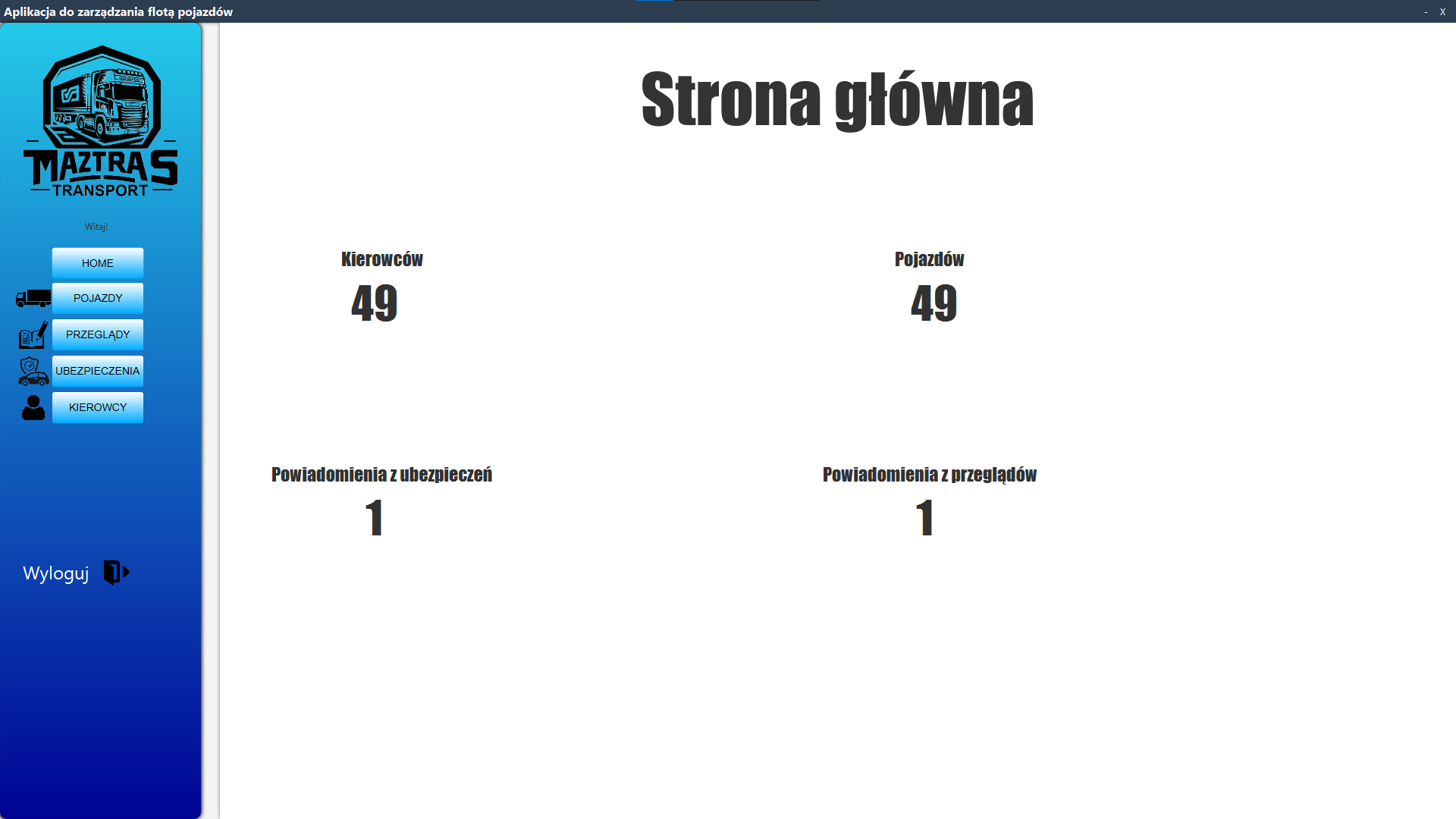
1. Ubezpieczenia- wyświetlenie konkretnych informacji zwrotnych dla użytkownika dotyczących ubezpieczeń.



1. Kierowcy – zakładka przedstawia kierowców i informacje o nich.



1. Powiadomienia – zakładka pozwalająca w prosty sposób określić liczbę zatrudnionych kierowców, bieżącą liczbę pojazdów oraz zasygnalizować nieważne przeglądy i ubezpieczenia w celu weryfikacji i reakcji przez administratora w celu odesłania konkretnego pojazdu na wymagany zabieg kontroli.



**4.Funkcjonalności:**

**a. Lista funkcjonalności**

1. **Zarządzanie pojazdami:**
   * Dodawanie nowych pojazdów do bazy danych.
   * Edycja danych pojazdów, takich jak marka, model, numer rejestracyjny.
   * Przeglądanie historii pojazdu (ubezpieczenia, przeglądy, przypisanie do kierowcy).
2. **Zarządzanie dokumentami:**
   * Dodawanie nowych ubezpieczeń i przeglądów technicznych.
   * Weryfikacja ważności ubezpieczenia i przeglądów.
   * Powiadomienia o zbliżających się terminach ważności.
3. **Zarządzanie kierowcami:**
   * Dodawanie i edycja danych kierowców.
   * Przypisywanie kierowców do pojazdów.

**b. Opis krok po kroku, jak wykonać daną czynność**

**Zarządzenie dodawaniem (na przykładzie dodania nowego pojazdu):**

1. Użytkownik loguje się do systemu jako administrator.
2. W panelu głównym wybiera sekcję „Pojazdy”.
3. W formularzu wprowadza szczegóły pojazdu:
   * VIN
   * Marka
   * Model
   * Rok produkcji
   * Numer rejestracyjny
   * Stan techniczny
   * Wymagane uprawnienia
   * ID ubezpieczenia
   * ID przeglądu
   * Typ pojazdu
4. Kliknięcie „Wprowadź” powoduje aktywowanie procedury sprawdzania danych.
5. System potwierdza dodanie pojazdu i wyświetla nowy wpis na liście pojazdów.

**Zarządzanie ubezpieczeniem pojazdu (dodanie ubezpieczenia):**

1. Użytkownik loguje się do systemu jako administrator.
2. W panelu głównym wybiera sekcję „Ubezpieczenia”.
3. W formularzu wprowadza szczegóły ubezpieczenia:
   * ID ubezpieczenia
   * ID pojazdu
   * Rodzaj ubezpieczenia
   * Data rozpoczęcia.
   * Data zakończenia
   * Koszt.
4. Klika „Wprowadź”.
5. System weryfikuje dane i zapisuje ubezpieczenie.
6. Powiadomienie o dodaniu ubezpieczenia jest wysyłane do użytkownika.

**c. Diagramy przepływu informacji**

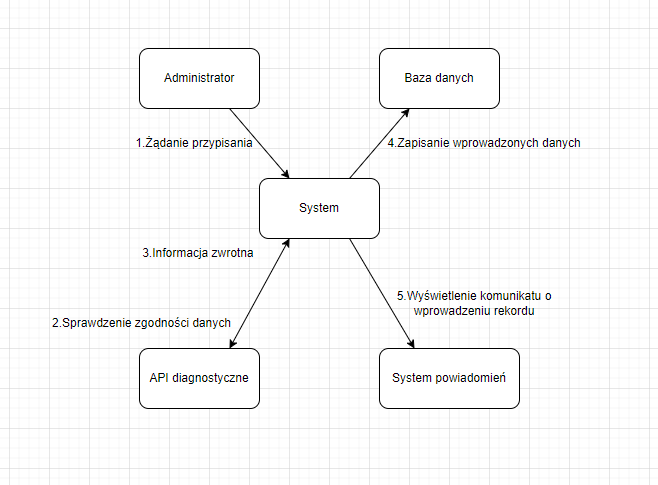
**Przykład: Przypisywanie pojazdu do kierowcy**

1. **Rola administratora**:
   * Administrator przypisuje kierowcę do pojazdu poprzez panel zarządzania.
   * System weryfikuje, czy kierowca ma odpowiednie uprawnienia (np. kategorie prawa jazdy).
   * Po zatwierdzeniu dane są zapisywane w bazie.
2. **Współpraca z systemami zewnętrznymi**:
   * System sprawdza ważność przeglądów technicznych pojazdu przed przypisaniem (np. integracja z API diagnostycznym).
   * Jeśli pojazd jest gotowy do użycia, przypisanie jest rejestrowane.

**Przykładowy diagram przepływu:**

1. Użytkownik (Administrator) -> System -> Baza danych: Dodanie kierowcy.
2. System -> API diagnostyczne: Sprawdzenie wprowadzonych danych.
3. System -> Powiadomienie: Potwierdzenie przypisania.

Analogicznie dla pozostałych danych.



**5.Bezpieczeństwo**

1. Zabezpieczenie przed dostaniem się osoby trzeciej do bazy danych działające na zasadzie wprowadzenia poprawności loginu i hasła do aplikacji.
2. Zabezpieczenia SQL:
3. Tabela pojazdy:

* PojazdID – PrimaryKey,NN,auto\_increment
* NumerVIN-UNIQUE
* UbezpieczenieID-UNIQUE
* PrzeglądID-MUL

1. Tabela kierowcy:

* PESEL- PrimaryKey,NN
* NumerPrawoJazdy-UNIQUE

1. Tabela ubezpieczenia:

* UbezpieczenieID- PrimaryKey,NN
* PojazdID-UNIQUE

1. Tabela przegladtechniczny:

* PrzegladID- PrimaryKey,NN
* PojazdID-MUL

W celu zabezpieczenia spójności danych niemożliwe jest dodanie przeglądu i ubezpieczenia do nieistniejącego pojazdu, zatem kolejność dodawania nowego pojazdu do bazy danych wygląda następująco:

Pojazd->Ubezpieczenie->Przegląd

W przeciwnym razie dodanie nowego rekordu okaże się niemożliwe!

**6.Dokumentacja API**

**1.Klasa DocumentController**

DocumentController to klasa kontrolera JavaFX odpowiedzialna za obsługę ekranu logowania w aplikacji. Klasa umożliwia użytkownikowi logowanie za pomocą nazwy użytkownika i hasła, a w przypadku poprawnego uwierzytelnienia przenosi użytkownika na główny ekran aplikacji (Dashboard.fxml). Zapewnia także możliwość zamknięcia aplikacji.

**Metody:**

**1. close()**

* **Opis**:
  + Zamyka aplikację, kończąc proces działania programu.

**2. loginAdmin()**

* **Opis**:
  + Obsługuje logowanie użytkownika.
  + Weryfikuje, czy pola username i password są wypełnione.
  + Sprawdza dane logowania w tabeli login w bazie danych.
  + Jeśli dane są poprawne:
    - Wyświetla komunikat o pomyślnym logowaniu.
    - Przechodzi na główny ekran aplikacji (Dashboard.fxml).
  + Jeśli dane są niepoprawne:
    - Wyświetla komunikat o błędzie logowania.
* **Funkcjonalność**:
  + Wykorzystuje zapytanie SQL SELECT \* FROM login WHERE username=? AND password=? do weryfikacji danych.
  + Przechowuje nazwę użytkownika w zmiennej getData.username.
  + Ładuje nową scenę (Dashboard.fxml) i ustawia ją jako bieżącą.
* **Obsługa błędów**:
  + Wyświetla komunikaty o błędach:
    - Jeśli pola są puste: „Proszę uzupełnić wszystkie pola!”.
    - Jeśli dane są niepoprawne: „Niepoprawne dane logowania!”.

**3. initialize(URL location, ResourceBundle resources)**

* **Opis**:
  + Metoda wywoływana automatycznie podczas inicjalizacji kontrolera.
  + Aktualnie nie zawiera żadnej logiki, ale może być używana do wstępnej konfiguracji.

**Pola klasy**

1. **main\_form**
   * Typ: AnchorPane
   * Opis: Główny kontener sceny logowania. Używany do przełączania widoku.
2. **username**
   * Typ: TextField
   * Opis: Pole tekstowe, w którym użytkownik wpisuje swoją nazwę użytkownika.
3. **password**
   * Typ: PasswordField
   * Opis: Pole tekstowe do wprowadzania hasła. Ukrywa wprowadzone znaki.
4. **connect**
   * Typ: Connection
   * Opis: Obiekt połączenia z bazą danych.
5. **prepareStatement**
   * Typ: PreparedStatement
   * Opis: Obiekt używany do przygotowania i wykonania zapytań SQL.
6. **result**
   * Typ: ResultSet
   * Opis: Obiekt zawierający wyniki zapytań SQL.
7. **alert**
   * Typ: Alert
   * Opis: Obiekt używany do wyświetlania komunikatów dla użytkownika.

**2.Klasa DashboardController:**

DashboardController jest głównym kontrolerem aplikacji zarządzającej pojazdami, kierowcami, przeglądami technicznymi i ubezpieczeniami. Klasa łączy się z bazą danych w celu obsługi odczytu, zapisu, edycji i usuwania danych.

**A. Metody i Klasy**

**1. Zarządzanie Aplikacją**

* **minimize(ActionEvent event)**
  + Minimalizuje aktualne okno aplikacji.
  + Przykład użycia:
    - Użytkownik naciska przycisk minimalizacji, aplikacja wyłącza tryb pełnego okna i pojawia się na pasku zadań.
* **close(ActionEvent event)**
  + Zamyka aktualne okno aplikacji.
  + Przykład użycia:
    - Użytkownik naciska przycisk przeznaczony do zamknięcia aplikacji, następuje wylogowanie i wyłączenie procedury.
* **logout(ActionEvent event)**
  + Wylogowuje użytkownika po potwierdzeniu w oknie dialogowym.
  + Przykład użycia:
    - Użytkownik naciska przycisk po którym następuje wylogowanie.
* **switchForm(ActionEvent event)**
  + Zmienia widok w aplikacji (Home, Pojazdy, Przeglądy, Ubezpieczenia, Kierowcy).
  + Przykład użycia:
    - Użytkownik naciska przycisk, w zależności od wyboru zmienia scenę w dashboard.

**2. Zarządzanie Pojazdami**

* **CarListData()**
  + **Opis: Pobiera listę pojazdów z bazy danych i zwraca ją jako obserwowalną listę (ObservableList).**
  + **Przykład użycia:**
    - **Funkcja wykorzystywana automatycznie do wczytania danych o pojazdach podczas uruchamiania aplikacji.**
* **ShowCarListData()**
  + **Opis: Wyświetla dane pojazdów w tabeli (TableView) na podstawie listy pobranej z bazy danych.**
  + **Przykład użycia:**
    - **Po wczytaniu danych z bazy aplikacja prezentuje je w tabeli na ekranie.**
* **searchCarData(ActionEvent event)**
  + **Opis: Wyszukuje pojazdy na podstawie kryteriów wprowadzonych w polach tekstowych, a następnie aktualizuje tabelę wyników.**
  + **Przykład użycia:**
    - **Użytkownik wprowadza wybrane dane w pola wyszukiwania i naciska przycisk „szukaj”. Tabela pokazuje pojazdy spełniające kryterium.**
* **deleteCar(ActionEvent event)**
  + **Opis: Usuwa wybrany pojazd z bazy danych po potwierdzeniu przez użytkownika.**
  + **Przykład użycia:**
    - **Użytkownik zaznacza pojazd w tabeli i naciska przycisk "Usuń". Aplikacja usuwa rekord z bazy.**
* **addCarToDatabase(ActionEvent event)**
  + **Opis: Dodaje nowy pojazd do bazy danych na podstawie danych wprowadzonych w polach tekstowych.**
  + **Przykład użycia:**
    - **Użytkownik wypełnia formularz dodawania pojazdu i klika "Wprowadź". Aplikacja zapisuje dane do bazy.**

**3. Zarządzanie Ubezpieczeniami**

* **InsuranceListData()**
  + **Opis: Pobiera listę ubezpieczeń z bazy danych i zwraca ją jako obserwowalną listę.**
  + **Przykład użycia:**
    - **Aplikacja automatycznie wywołuje funkcję przy uruchomieniu, aby wczytać dane o ubezpieczeniach.**
* **ShowInsuranceListData()**
  + **Opis: Wyświetla dane ubezpieczeń w tabeli na podstawie listy pobranej z bazy danych.**
  + **Przykład użycia:**
    - **Ubezpieczenia są prezentowane w tabeli po uruchomieniu aplikacji lub po aktualizacji danych.**
* **searchInsuranceData(ActionEvent event)**
  + **Opis: Wyszukuje ubezpieczenia na podstawie kryteriów wpisanych w pola tekstowe i aktualizuje tabelę wyników.**
  + **Przykład użycia:**
    - **Użytkownik wprowadza wybrane dane, po czym klika przycisk wyszukiwania. Tabela pokazuje pasujące wyniki.**
* **deleteInsurance(ActionEvent event)**
  + **Opis: Usuwa wybrane ubezpieczenie z bazy danych po potwierdzeniu przez użytkownika.**
  + **Przykład użycia:**
    - **Użytkownik zaznacza ubezpieczenie w tabeli i naciska przycisk "Usuń". Rekord zostaje usunięty.**
* **addInsuranceToDatabase(ActionEvent event)**
  + **Opis: Dodaje nowe ubezpieczenie do bazy danych na podstawie danych wprowadzonych w formularzu.**
  + **Przykład użycia:**
    - **Użytkownik wypełnia formularz dodawania ubezpieczenia i klika "Wprowadź". Aplikacja zapisuje dane.**

**4. Zarządzanie Kierowcami**

* **getDriverListData()**
  + **Opis: Pobiera listę kierowców z bazy danych i zwraca ją jako obserwowalną listę.**
  + **Przykład użycia:**
    - **Funkcja uruchamiana automatycznie przy inicjalizacji aplikacji.**
* **ShowDriverListData()**
  + **Opis: Wyświetla listę kierowców w tabeli na podstawie pobranych danych.**
  + **Przykład użycia:**
    - **Kierowcy są prezentowani w tabeli po wczytaniu danych z bazy.**
* **searchDriverData(ActionEvent event)**
  + **Opis: Wyszukuje kierowców na podstawie danych wprowadzonych w pola tekstowe i aktualizuje tabelę wyników.**
  + **Przykład użycia:**
    - **Użytkownik wpisuje nazwisko kierowcy i klika przycisk wyszukiwania. Tabela pokazuje dopasowane rekordy.**
* **deleteDriver(ActionEvent event)**
  + **Opis: Usuwa wybranego kierowcę z bazy danych po potwierdzeniu przez użytkownika.**
  + **Przykład użycia:**
    - **Użytkownik wybiera kierowcę w tabeli i naciska przycisk "Usuń". Rekord zostaje usunięty.**
* **addDriverToDatabase(ActionEvent event)**
  + **Opis: Dodaje nowego kierowcę do bazy danych na podstawie danych wprowadzonych w formularzu.**
  + **Przykład użycia:**
    - **Użytkownik wypełnia formularz kierowcy i klika "Wprowadź". Dane są zapisywane w bazie.**

**5. Zarządzanie Przeglądami**

* **getInspectionListData()**
  + **Opis: Pobiera listę przeglądów technicznych z bazy danych i zwraca ją jako obserwowalną listę.**
  + **Przykład użycia:**
    - **Dane przeglądów są wczytywane automatycznie podczas uruchamiania aplikacji.**
* **ShowInspectionListData()**
  + **Opis: Wyświetla listę przeglądów technicznych w tabeli na podstawie pobranych danych.**
  + **Przykład użycia:**
    - **Przeglądy są prezentowane w tabeli po inicjalizacji aplikacji.**
* **searchInspectionData(ActionEvent event)**
  + **Opis: Wyszukuje przeglądy techniczne na podstawie kryteriów wpisanych w pola tekstowe i aktualizuje tabelę wyników.**
  + **Przykład użycia:**
    - **Użytkownik wprowadza datę przeglądu i klika przycisk wyszukiwania. Tabela wyświetla wyniki.**
* **deleteInspectionRecord(ActionEvent event)**
  + **Opis: Usuwa wybrany przegląd techniczny z bazy danych po potwierdzeniu przez użytkownika.**
  + **Przykład użycia:**
    - **Użytkownik wybiera przegląd w tabeli i naciska przycisk "Usuń". Rekord zostaje usunięty.**
* **addInspectionToDatabase(ActionEvent event)**
  + **Opis: Dodaje nowy przegląd techniczny do bazy danych na podstawie danych wprowadzonych w formularzu.**
  + **Przykład użycia:**
    - **Użytkownik wypełnia formularz przeglądu i klika "Wprowadź". Dane są zapisywane w bazie.**

**6. Statystyki**

* **getDriverCount()**
  + **Opis: Liczy łączną liczbę kierowców w bazie danych.**
  + **Przykład użycia:**
    - **Funkcja uruchamiana automatycznie w celu aktualizacji statystyk.**
* **updateDriverCountLabel()**
  + **Opis: Aktualizuje etykietę wyświetlającą liczbę kierowców na ekranie.**
  + **Przykład użycia:**
    - **Statystyka "Liczba kierowców" na dashboardzie jest automatycznie odświeżana.**
* **getCarsCount()**
  + **Opis: Liczy pojazdy w bazie danych.**
  + **Przykład użycia:**
    - **Funkcja uruchamiana automatycznie w celu aktualizacji statystyk.**
* **updateCarsCountLabel()**
  + **Opis: Aktualizuje etykietę wyświetlającą liczbę pojazdów na ekranie.**
  + **Przykład użycia:**
    - **Statystyka "Liczba pojazdów" na dashboardzie jest automatycznie odświeżana.**
* **getValidInsuranceCount()**
  + **Opis: Liczy liczbę wygasłych ubezpieczeń w bazie danych.**
  + **Przykład użycia:**
    - **Statystyka "Wygasłe ubezpieczenia" jest automatycznie obliczana.**
* **updateInsuranceLabel()**
  + **Opis: Aktualizuje etykietę dotyczącą liczby wygasłych ubezpieczeń.**
  + **Przykład użycia:**
    - **Dashboard automatycznie odświeża liczbę wygasłych ubezpieczeń i wyświetla jako liczbę w powiadomieniach.**
* **getInvalidInspectionCount()**
  + **Opis: Liczy liczbę przeglądów technicznych po terminie w bazie danych.**
  + **Przykład użycia:**
    - **Statystyka "Nieważne przeglądy" jest obliczana automatycznie.**
* **updateInspectionLabel()**
  + **Opis: Aktualizuje etykietę dotyczącą liczby nieważnych przeglądów.**
  + **Przykład użycia:**
    - **Dashboard automatycznie odświeża liczbę przeglądów po terminie i wyświetla jako liczbę powiadomienia.**

**3.Klasa App:**

**A)Metody**

**1. start(Stage stage)**

* **Opis:**
  + **Metoda uruchamiana automatycznie podczas startu aplikacji.**
  + **Inicjalizuje bazę danych, wczytuje główny widok aplikacji (Main.fxml), ustawia scenę oraz uruchamia okno aplikacji.**
* **Funkcjonalność:**
  + **Sprawdza połączenie z bazą danych za pomocą metody checkAndInitializeDatabase().**
  + **Wczytuje widok główny z pliku Main.fxml za pomocą klasy FXMLLoader.**
  + **Ustawia scenę aplikacji w stylu bez dekoracji (StageStyle.UNDECORATED).**
  + **Wyświetla okno aplikacji.**
* **Przykład użycia:**
  + **Metoda jest wywoływana automatycznie przy uruchamianiu aplikacji, np. gdy użytkownik dwukrotnie kliknie ikonę aplikacji.**

**2. checkAndInitializeDatabase()**

* **Opis:**
  + **Sprawdza, czy baza danych jest dostępna, i nawiązuje połączenie.**
  + **W przypadku problemów z połączeniem wypisuje komunikat na konsolę i uniemożliwia uruchomienie aplikacji.**
* **Funkcjonalność:**
  + **Wywołuje metodę database.connectDB() w celu połączenia z bazą danych.**
  + **Zwraca wartość true, jeśli połączenie zakończyło się sukcesem, lub false, jeśli wystąpił błąd.**
* **Przykład użycia:**
  + **Funkcja jest wywoływana podczas startu aplikacji, aby upewnić się, że baza danych działa prawidłowo.**

**3. main(String[] args)**

* **Opis:**
  + **Główny punkt wejścia aplikacji. Uruchamia metodę launch(args) z klasy Application, która rozpoczyna działanie aplikacji JavaFX.**
* **Funkcjonalność:**
  + **Uruchamia aplikację, co powoduje wywołanie metody start(Stage stage).**
* **Przykład użycia:**
  + **Wywoływana automatycznie przez JVM podczas uruchamiania programu, np. w terminalu poprzez komendę java org.example.App.**

**4.Klasa driverData:**

**driverData to klasa modelu reprezentująca dane dotyczące kierowcy. Służy do przechowywania informacji o kierowcy w aplikacji. Dane przechowywane w tej klasie są wykorzystywane w różnych operacjach, takich jak wyświetlanie, edytowanie, usuwanie i wyszukiwanie kierowców.**

**Pola klasy**

1. **pesel**
   * **Typ: String**
   * **Opis: Numer PESEL kierowcy.**
   * **Uwagi: Przechowuje unikalny identyfikator kierowcy w postaci ciągu znaków.**
2. **imie**
   * **Typ: String**
   * **Opis: Imię kierowcy.**
3. **nazwisko**
   * **Typ: String**
   * **Opis: Nazwisko kierowcy.**
4. **dataUrodzenia**
   * **Typ: Date**
   * **Opis: Data urodzenia kierowcy.**
5. **NumerPrawoJazdy**
   * **Typ: String**
   * **Opis: Numer prawa jazdy kierowcy.**
6. **uprawnienia**
   * **Typ: String**
   * **Opis: Typ uprawnień posiadanych przez kierowcę (np. kategoria B, C).**

**Konstruktor**

**driverData(String pesel, String imie, String nazwisko, Date dataUrodzenia, String NumerPrawoJazdy, String uprawnienia)**

* **Opis:**
  + **Inicjalizuje obiekt klasy driverData za pomocą dostarczonych parametrów.**
* **Parametry:**
  + **pesel: Numer PESEL kierowcy.**
  + **imie: Imię kierowcy.**
  + **nazwisko: Nazwisko kierowcy.**
  + **dataUrodzenia: Data urodzenia kierowcy.**
  + **NumerPrawoJazdy: Numer prawa jazdy kierowcy.**
  + **uprawnienia: Typ uprawnień posiadanych przez kierowcę.**

**Adnotacje**

1. **@Getter (z biblioteki Lombok):**
   * **Automatycznie generuje gettery dla wszystkich pól w klasie.**
   * **Nie ma potrzeby ręcznego definiowania metod takich jak getPesel() czy getImie().**

**Metody (wygenerowane przez Lombok)**

1. **getPesel()**
   * **Zwraca wartość pola pesel.**
2. **getImie()**
   * **Zwraca wartość pola imie.**
3. **getNazwisko()**
   * **Zwraca wartość pola nazwisko.**
4. **getDataUrodzenia()**
   * **Zwraca wartość pola dataUrodzenia.**
5. **getNumerPrawoJazdy()**
   * **Zwraca wartość pola NumerPrawoJazdy.**
6. **getUprawnienia()**
   * **Zwraca wartość pola uprawnienia.**

**5.Klasa carData:**

driverData to klasa modelu reprezentująca dane dotyczące kierowcy. Służy do przechowywania informacji o kierowcy w aplikacji. Dane przechowywane w tej klasie są wykorzystywane w różnych operacjach, takich jak wyświetlanie, edytowanie, usuwanie i wyszukiwanie kierowców.

**Pola klasy**

1. **pesel**
   * Typ: String
   * Opis: Numer PESEL kierowcy.
   * Uwagi: Przechowuje unikalny identyfikator kierowcy w postaci ciągu znaków.
2. **imie**
   * Typ: String
   * Opis: Imię kierowcy.
3. **nazwisko**
   * Typ: String
   * Opis: Nazwisko kierowcy.
4. **dataUrodzenia**
   * Typ: Date
   * Opis: Data urodzenia kierowcy.
5. **NumerPrawoJazdy**
   * Typ: String
   * Opis: Numer prawa jazdy kierowcy.
6. **uprawnienia**
   * Typ: String
   * Opis: Typ uprawnień posiadanych przez kierowcę (np. kategoria B, C).

**Konstruktor**

**driverData(String pesel, String imie, String nazwisko, Date dataUrodzenia, String NumerPrawoJazdy, String uprawnienia)**

* **Opis**:
  + Inicjalizuje obiekt klasy driverData za pomocą dostarczonych parametrów.
* **Parametry**:
  + pesel: Numer PESEL kierowcy.
  + imie: Imię kierowcy.
  + nazwisko: Nazwisko kierowcy.
  + dataUrodzenia: Data urodzenia kierowcy.
  + NumerPrawoJazdy: Numer prawa jazdy kierowcy.
  + uprawnienia: Typ uprawnień posiadanych przez kierowcę.

**Adnotacje**

1. **@Getter** (z biblioteki Lombok):
   * Automatycznie generuje gettery dla wszystkich pól w klasie.
   * Nie ma potrzeby ręcznego definiowania metod takich jak getPesel() czy getImie().

**Metody (wygenerowane przez Lombok)**

1. **getPesel()**
   * Zwraca wartość pola pesel.
2. **getImie()**
   * Zwraca wartość pola imie.
3. **getNazwisko()**
   * Zwraca wartość pola nazwisko.
4. **getDataUrodzenia()**
   * Zwraca wartość pola dataUrodzenia.
5. **getNumerPrawoJazdy()**
   * Zwraca wartość pola NumerPrawoJazdy.
6. **getUprawnienia()**
   * Zwraca wartość pola uprawnienia.

**6.Klasa insuranceData:**

insuranceData to klasa modelu reprezentująca dane dotyczące ubezpieczeń pojazdów. Klasa służy do przechowywania informacji o ubezpieczeniach w aplikacji, wykorzystywana jest do operacji takich jak wyświetlanie, dodawanie, edytowanie i usuwanie danych o ubezpieczeniach.

**Pola klasy**

1. **UbezpieczenieID**
   * Typ: Integer
   * Opis: Unikalny identyfikator ubezpieczenia w bazie danych.
2. **NumerRejestracyjny**
   * Typ: String
   * Opis: Numer rejestracyjny pojazdu, do którego przypisane jest ubezpieczenie.
3. **RodzajUbezpieczenia**
   * Typ: String
   * Opis: Rodzaj ubezpieczenia (np. OC, AC, NNW).
4. **DataRozpoczecia**
   * Typ: Date
   * Opis: Data rozpoczęcia ważności ubezpieczenia.
5. **DataZakonczenia**
   * Typ: Date
   * Opis: Data zakończenia ważności ubezpieczenia.
6. **Koszt**
   * Typ: BigDecimal
   * Opis: Koszt ubezpieczenia.

**Konstruktor**

**insuranceData(Integer ubezpieczenieID, String NumerRejestracyjny, String rodzajUbezpieczenia, Date dataRozpoczecia, Date dataZakonczenia, BigDecimal koszt)**

* **Opis**:
  + Konstruktor inicjalizuje obiekt klasy insuranceData za pomocą danych dotyczących ubezpieczenia pojazdu.
* **Parametry**:
  + ubezpieczenieID: Unikalny identyfikator ubezpieczenia.
  + NumerRejestracyjny: Numer rejestracyjny pojazdu.
  + rodzajUbezpieczenia: Typ ubezpieczenia (np. OC, AC, NNW).
  + dataRozpoczecia: Data rozpoczęcia ważności ubezpieczenia.
  + dataZakonczenia: Data zakończenia ważności ubezpieczenia.
  + koszt: Koszt ubezpieczenia.

**Adnotacje**

1. **@Getter** (z biblioteki Lombok):
   * Automatycznie generuje gettery dla wszystkich pól w klasie.
   * Nie ma potrzeby ręcznego definiowania metod takich jak getUbezpieczenieID() czy getNumerRejestracyjny().

**Metody (wygenerowane przez Lombok)**

1. **getUbezpieczenieID()**
   * Zwraca wartość pola UbezpieczenieID.
2. **getNumerRejestracyjny()**
   * Zwraca wartość pola NumerRejestracyjny.
3. **getRodzajUbezpieczenia()**
   * Zwraca wartość pola RodzajUbezpieczenia.
4. **getDataRozpoczecia()**
   * Zwraca wartość pola DataRozpoczecia.
5. **getDataZakonczenia()**
   * Zwraca wartość pola DataZakonczenia.
6. **getKoszt()**
   * Zwraca wartość pola Koszt.

**7.Klasa inspectionData:**

inspectionData to klasa modelu reprezentująca dane dotyczące przeglądów technicznych pojazdów. Klasa służy do przechowywania i zarządzania informacjami o przeglądach w aplikacji, umożliwiając operacje takie jak wyświetlanie, wyszukiwanie czy dodawanie przeglądów.

**Pola klasy**

1. **przegladID**
   * Typ: int
   * Opis: Unikalny identyfikator przeglądu technicznego w bazie danych.
2. **pojazdID**
   * Typ: int
   * Opis: Identyfikator pojazdu, do którego przypisany jest przegląd techniczny.
3. **dataPrzegladu**
   * Typ: Date
   * Opis: Data wykonania przeglądu technicznego.
4. **wynikPrzegladu**
   * Typ: String
   * Opis: Wynik przeglądu technicznego (np. Pozytywny, Negatywny).
5. **uwagi**
   * Typ: String
   * Opis: Dodatkowe uwagi lub komentarze związane z przeglądem.

**Konstruktor**

**inspectionData(int przegladID, int pojazdID, Date dataPrzegladu, String wynikPrzegladu, String uwagi)**

* **Opis**:
  + Konstruktor inicjalizuje obiekt klasy inspectionData za pomocą danych dotyczących przeglądu technicznego.
* **Parametry**:
  + przegladID: Unikalny identyfikator przeglądu.
  + pojazdID: Identyfikator pojazdu.
  + dataPrzegladu: Data wykonania przeglądu.
  + wynikPrzegladu: Wynik przeglądu technicznego.
  + uwagi: Dodatkowe uwagi dotyczące przeglądu.

**Adnotacje**

1. **@Getter** (z biblioteki Lombok):
   * Automatycznie generuje gettery dla wszystkich pól w klasie.
   * Nie ma potrzeby ręcznego definiowania metod takich jak getPrzegladID() czy getPojazdID().

**Metody (wygenerowane przez Lombok)**

1. **getPrzegladID()**
   * Zwraca wartość pola przegladID.
2. **getPojazdID()**
   * Zwraca wartość pola pojazdID.
3. **getDataPrzegladu()**
   * Zwraca wartość pola dataPrzegladu.
4. **getWynikPrzegladu()**
   * Zwraca wartość pola wynikPrzegladu.
5. **getUwagi()**
   * Zwraca wartość pola uwagi.

**8.Klasa database:**

Klasa database jest odpowiedzialna za zarządzanie połączeniem z bazą danych MySQL oraz zapewnienie, że baza danych aplikacji istnieje i jest poprawnie skonfigurowana. Klasa automatycznie tworzy bazę danych, jeśli jej brakuje, i importuje dane z pliku SQL, co pozwala na szybkie uruchomienie aplikacji.

**Metody**

**1. connectDB()**

* **Opis**:
  + Ustanawia połączenie z bazą danych.
  + Sprawdza, czy baza danych zarzadzanie\_flota\_pojazdow istnieje.
  + Jeśli baza danych nie istnieje, automatycznie ją tworzy i importuje dane z pliku SQL (zarzadzanie\_flota\_pojazdow.sql).
* **Funkcjonalność**:
  + Ładuje sterownik MySQL.
  + Łączy się z serwerem MySQL (domyślnie localhost).
  + Sprawdza, czy baza danych zarzadzanie\_flota\_pojazdow istnieje:
    - Jeśli tak: nawiązuje połączenie z istniejącą bazą.
    - Jeśli nie: tworzy nową bazę danych, importuje strukturę i dane z pliku SQL.
* **Zwraca**:
  + Obiekt Connection reprezentujący aktywne połączenie z bazą danych.

**2. importDatabase(Connection connection)**

* **Opis**:
  + Importuje dane do bazy danych z pliku SQL znajdującego się w zasobach projektu (zarzadzanie\_flota\_pojazdow.sql).
* **Funkcjonalność**:
  + Wczytuje plik SQL za pomocą klasy InputStream.
  + Przetwarza i wykonuje kolejne zapytania SQL zawarte w pliku.
  + Ignoruje komentarze (--) i puste linie w pliku SQL.
  + Loguje zapytania SQL oraz potencjalne błędy podczas ich wykonywania.
* **Przykład użycia**:
  + Metoda jest wywoływana automatycznie przez connectDB() w przypadku braku bazy danych.
  + W konsoli wyświetla komunikaty o statusie importu (np. wykonane zapytania SQL).

**Proces działania klasy**

1. **Połączenie z bazą danych**:
   * Klasa próbuje nawiązać połączenie z serwerem MySQL na localhost:3306 przy użyciu użytkownika root i pustego hasła.
2. **Sprawdzanie bazy danych**:
   * Zapytanie SQL weryfikuje, czy baza danych zarzadzanie\_flota\_pojazdow istnieje.
   * Jeśli baza istnieje:
     + Klasa zamyka połączenie z serwerem głównym i otwiera połączenie z tą bazą.
   * Jeśli baza nie istnieje:
     + Klasa tworzy bazę danych i wykonuje metodę importDatabase, aby zaimportować dane i strukturę z pliku SQL.
3. **Import danych**:
   * Plik SQL (zarzadzanie\_flota\_pojazdow.sql) jest wczytywany z zasobów projektu.
   * Wykonywane są wszystkie zapytania SQL zawarte w pliku, a klasa loguje sukcesy lub błędy wykonania.

**8.Przydatne infromacje**

**Ograniczenia aplikacji**

1. **Brak obsługi wielu użytkowników**:
   * Aplikacja obsługuje jedynie jednego użytkownika (administratora), co może być problematyczne w środowisku wieloosobowym.
2. **Brak zaawansowanego systemu uprawnień**:
   * Każdy użytkownik (lub administrator) ma pełny dostęp do wszystkich danych w aplikacji. Brakuje mechanizmów kontrolowania dostępu (np. uprawnienia dla różnych ról).
3. **Brak funkcji eksportu danych**:
   * Aplikacja nie oferuje możliwości eksportu danych (np. do plików CSV, PDF), co ogranicza możliwość współpracy z innymi systemami.
4. **Brak funkcji raportowania**:
   * Aplikacja nie posiada funkcji generowania raportów lub analiz (np. roczny raport przeglądów czy wygasłych ubezpieczeń).
5. **Brak możliwości pracy w trybie online**:
   * Aplikacja wydaje się działać w środowisku lokalnym, co ogranicza jej zastosowanie w większych organizacjach wymagających zdalnego dostępu.

**Zalety aplikacji**

1. **Przejrzysty interfejs użytkownika**:
   * Dashboard umożliwia łatwe zarządzanie danymi pojazdów, kierowców, ubezpieczeń i przeglądów w jednym miejscu.
2. **Prosta obsługa danych**:
   * Aplikacja oferuje intuicyjne funkcje dodawania, edycji, wyszukiwania i usuwania rekordów, co ułatwia zarządzanie bazą danych.
3. **Integracja z bazą danych**:
   * Pełne połączenie z bazą danych umożliwia przechowywanie dużej ilości danych i ich zarządzanie w sposób strukturalny.
4. **Automatyczna inicjalizacja**:
   * Po uruchomieniu aplikacja automatycznie ładuje dane do odpowiednich tabel i aktualizuje statystyki (np. liczba pojazdów, wygasłe ubezpieczenia).
5. **Funkcje wyszukiwania**:
   * Użytkownik może wyszukiwać dane w każdej sekcji (pojazdy, kierowcy, ubezpieczenia, przeglądy) na podstawie wprowadzonych kryteriów.
6. **Walidacja podstawowa**:
   * Aplikacja sprawdza, czy wszystkie wymagane pola zostały wypełnione, co zmniejsza ryzyko wprowadzenia niekompletnych danych.
7. **Obsługa błędów podczas operacji na bazie danych**:
   * W przypadku błędów (np. brak połączenia z bazą), aplikacja wyświetla komunikaty ostrzegawcze, co pomaga w diagnozowaniu problemów.
8. **Kolorowe wskaźniki statusu**:
   * Przeglądy i ubezpieczenia są wyróżniane kolorami w tabeli (np. czerwony kolor dla wygasłych rekordów), co zwiększa czytelność i efektywność pracy.
9. **Prosty system statystyk**:
   * Aplikacja wyświetla statystyki, takie jak liczba pojazdów, kierowców, wygasłych ubezpieczeń czy nieważnych przeglądów, co pozwala użytkownikowi szybko uzyskać kluczowe informacje.
10. **Możliwość rozbudowy**:
    * Kod aplikacji jest napisany w modularny sposób, co pozwala na łatwe dodawanie nowych funkcji i rozszerzeń w przyszłości.