Bartłomiej Jagiełło 254521

Piotr Kołpa 254557

Michał Najwer 254560

Agata Rudzka 242466

**Projekt Programistyczny IoT**

**System do obsługi kart parkingowych**

**Podstawy Internetu Rzeczy laboratorium 2021/2022**

Prowadzący:

dr inż. Krzysztof Chudzik

# 1. Spis treści

[1. Spis treści 2](#_Toc94215151)

[2. Wymagania projektowe 3](#_Toc94215152)

[2.1. Wymagania funkcjonalne 3](#_Toc94215153)

[2.2. Wymagania niefunkcjonalne 3](#_Toc94215154)

[2.3 Użytkownicy systemu 4](#_Toc94215155)

[2.4 Diagram Przypadków Użycia 4](#_Toc94215156)

[3. Schemat architektury systemu 5](#_Toc94215157)

[4. Schemat bazy danych 6](#_Toc94215158)

[5. Opis implementacji i zastosowanych rozwiązań 7](#_Toc94215159)

[5.1. Front-end 7](#_Toc94215160)

[5.2. Back-end 7](#_Toc94215161)

[5.3 Opis komunikacji MQTT 8](#_Toc94215162)

[5.3.1 Ogólny opis komunikacji 8](#_Toc94215163)

[5.3.2 Diagram sekwencji przesyłanych wiadomości 9](#_Toc94215164)

[5.3.3 Zabezpieczenia komunikacji MQTT 9](#_Toc94215165)

[5.3.4 Fragment kodu odpowiedzialny za ustawianie komunikacji MQTT na serwerze 10](#_Toc94215166)

[5.3.5 Obsługa wiadomości na serwerze 12](#_Toc94215167)

[5.3.6 Fragmenty kodu odpowiedzialny za ustawianie komunikacji MQTT na klientach (szlabanów i czytnika): 13](#_Toc94215168)

[5.3.7 Ustawienia brokera 14](#_Toc94215169)

[5.3.8 Przykład komunikacji 15](#_Toc94215170)

[6. Opis implementacji bazy danych 17](#_Toc94215171)

[6.1. Analiza otrzymanych wiadomości przez bazę danych 18](#_Toc94215172)

[6.2. Wjazd na parking 19](#_Toc94215173)

[6.3. Wyjazd z parkingu 21](#_Toc94215174)

[6.4. Zeskanowanie nowej karty 22](#_Toc94215175)

[7. Opis działania i prezentacja interfejsu 23](#_Toc94215176)

[8. Szczegółowy opis wkładu pracy Autorów 26](#_Toc94215177)

[9. Podsumowanie 27](#_Toc94215178)

[10. Literatura 28](#_Toc94215179)

[11. Aneks 29](#_Toc94215180)

# 2. Wymagania projektowe

System obsługi parkingu będzie zajmował się przechowywaniem w bazie danych informacji jakie osoby aktualnie korzystają z danego parkingu, będzie kontrolował szlabany wjazdowe i wyjazdowe. Każda osoba uprawniona do korzystania z parkingu będzie posiadała swoją unikatową kartę RFID przeznaczoną do identyfikacji.

## 2.1. Wymagania funkcjonalne

1. Podnoszenie szlabanu po zeskanowaniu aktywnej karty RFID i opuszczenie po chwili.
2. Monitorowanie stanu zapełnienia parkingu i nie wpuszczanie nowych użytkowników jeśli jest pełny.
3. Monitorowanie wjazdów i wyjazdów z parkingu przy pomocy czytników kart RFID.
4. Blokowanie prób wielokrotnego wjazdu na tą samą kartę bez wyjazdu.
5. Dodawanie kart RFID skojarzonych z konkretnym użytkownikiem poprzez imię i nazwisko.
6. Blokowanie kart RFID.
7. Aktywowanie kart RFID.
8. Zmiana właściciela karty RFID.
9. Przeglądanie listy kart RFID.
10. Wyświetlanie danych karty RFID w tym danych o użytkowniku i historii wjazdów / wyjazdów.
11. Konto administratora odpowiedzialne za zarządzanie systemem kart RFID.
12. Dodawanie nowych szlabanów po identyfikatorze.

## 2.2. Wymagania niefunkcjonalne

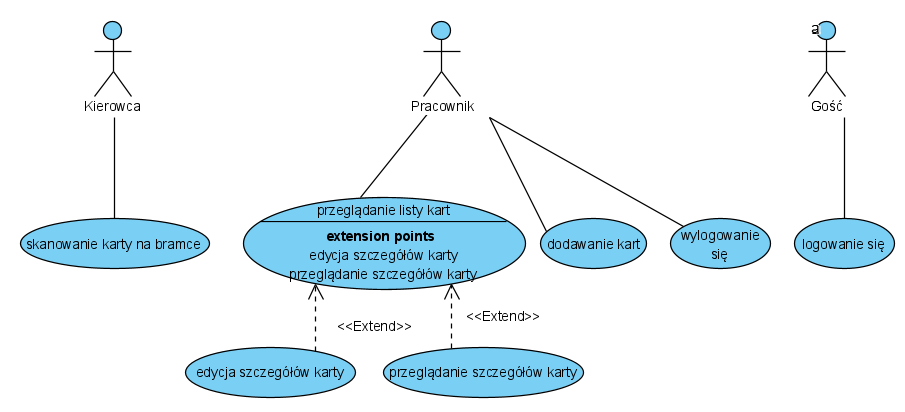
1. Aplikacja webowa działająca na przeglądarkach Google Chrome, Mozilla Firefox, Microsoft Edge, Safari.
2. Aplikacja webowa działająca na systemach windows (11, 10, 8, 7) i linux (przynajmniej ubuntu, debian, redhat).
3. Obsługa 24/7.
4. Możliwość rozszerzenia systemu o następne urządzenia: szlabany wjazdowe, wyjazdowe.
5. Obsługa wielu użytkowników jednocześnie.
6. Zabezpieczenie przed nieautoryzowanym połączeniem poprzez klucze.

## 2.3 Użytkownicy systemu

W niniejszym rozdziale zdefiniowano grupy użytkowników, korzystających z projektowanego systemu:

* Pracownik – użytkownik zalogowany posiadający uprawnienia pozwalające na dodawanie nowych kart do systemu; usuwanie, aktywowanie i blokowanie kart; przypisywanie właściciela do karty oraz przeglądanie danych kart parkingowych.
* Kierowca – użytkownik niezalogowany będący właścicielem karty RFID, którą ma możliwość skanować przy wjeździe i wyjeździe z parkingu.
* Gość – użytkownik niezalogowany, jedyną dostępną dla niego funkcją systemu jest logowanie.

## 2.4 Diagram Przypadków Użycia



Rys. Diagram przypadków użycia

W ramach projektu opracowano diagram przypadków użycia. Przewidziano na nim trzy grupy użytkowników: Gościa, Kierowcę i Pracownika.

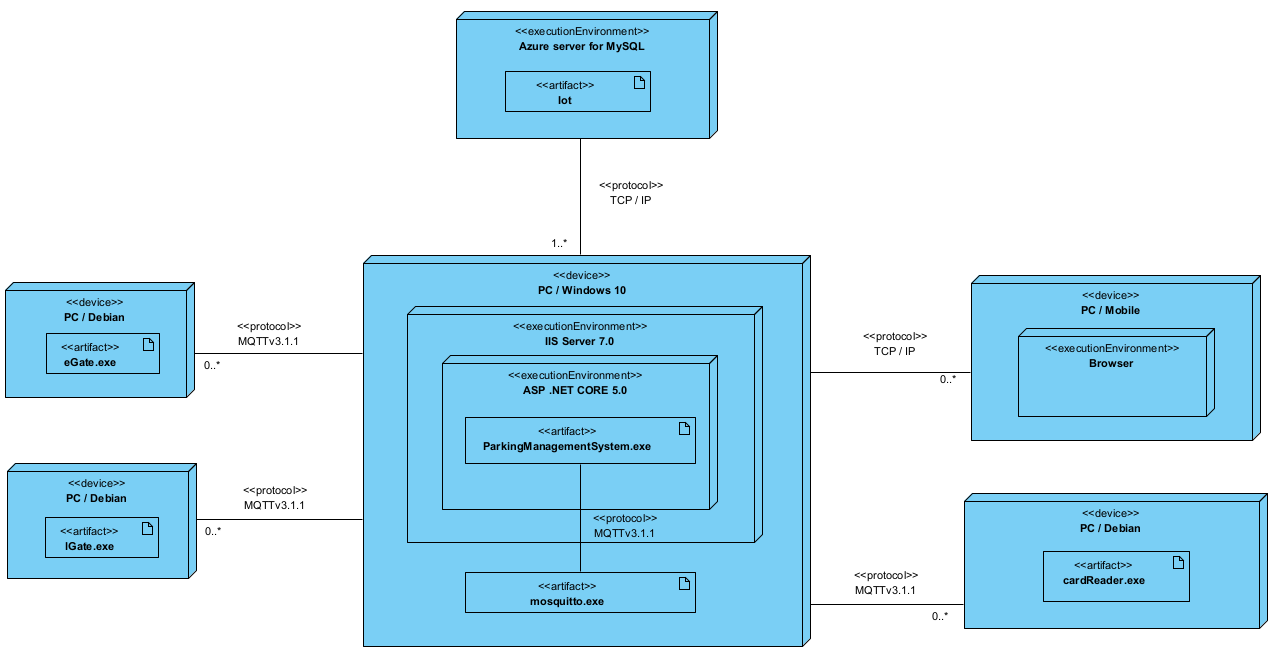
Gość jest niezalogowanym użytkownikiem aplikacji internetowej, ma możliwość zalogowania na konto pracownika jeżeli jest jego posiadaczem.

Kierowca jest użytkownikiem nielogującym się do systemu – ma jedynie możliwość korzystania   
z karty wydanej mu przez pracownika skanując ją w bramce parkingu przy przejeździe przez szlaban.

Pracownik jest użytkownikiem zalogowanym w aplikacji. Ma możliwość przeglądania kart zarejestrowanych w systemie a także edycji ich szczegółów w tym ich blokowania i odblokowywania. Może też skanować nowe karty czytnikiem i dodawać je do systemu przypisując im przy tym w aplikacji internetowej właściciela.

Diagram został wykonany przy użyciu programu Visual Paradigm.

# 3. Schemat architektury systemu

****

Rys.2 Diagram rozmieszczenia UML

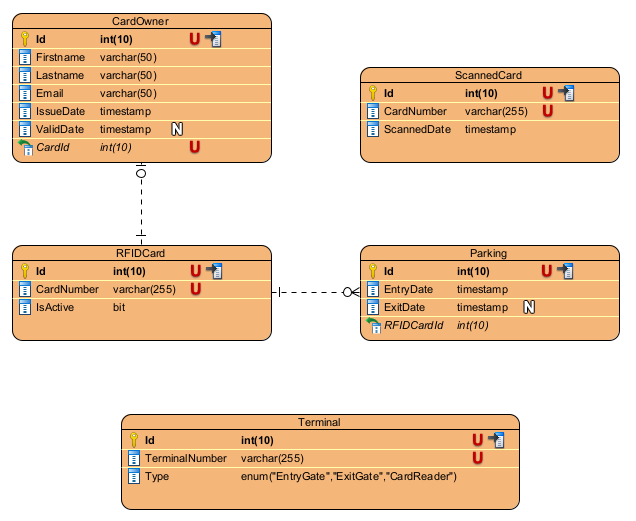
Na powyższym diagramie przedstawiono przykładowy schemat systemu obsługi parkingu. Po lewej stronie diagramu mamy klientów szlabanów odpowiednio wjazdowego i wyjazdowego połączonych z serwerem znajdującym się w centralnej części. Szlabany wyposażone są w czytniki kart RFID i przesyłają dane z odczytanych kart do serwera. Klientów szlabanów może być dowolnie wielu w ramach jednego parkingu. Komunikaty przesyłane są dzięki protokołowi MQTT w wersji 3.1.1.

Serwer składa się brokera Mosquitto, który przekazuje wiadomości do serwera który jest programem w środowisku IIS Server działającym na platformie ASP .NET CORE 5.0. Serwer łączy się z bazą danych MySQL znajdującą się na serwerze w usłudze Microsoft Azure. Każdy serwer ma jedną bazę danych. Jeden serwer obsługuje jeden parking.

Ponadto serwer udostępnia stronę www do zarządzania kartami i uprawnieniami dostępną dla dowolnie wielu użytkowników.

Ostatnim elementem systemu jest czytnik kart służący do dodawania nowych kart uprawnionych do wjazdu na parking połączony z serwerem. Czytników może być dowolnie wiele, wykorzystują ten sam sposób komunikacji jak szlabany.

# 4. Schemat bazy danych



Rys. Schemat bazy danych

W ramach projektu zaprojektowano i zaimplementowano relacyjną bazę danych przechowująca numery kart, oraz ich właścicieli. Umożliwia ona sprawdzenie czy danej karcie nie skończył się jeszcze termin ważności oraz czy nie została zablokowana. W celu bezpieczniejszego dodawania kart do systemu, tymczasowo zeskanowane nowe karty są przechowywane w osobnej tabeli oczekując na zaakceptowanie przez administratora.

Karty RFID mogą występować bez właściciela, lecz wtedy domyślnie są jako nieaktywne i nie da się z nich korzystać. W momencie wjazdu na parking dodaje się rekord do tabeli Parking, wraz z datą wjazdu oraz identyfikatorem karty. W momencie wyjazdu dany rekord jest uzupełniany o datę wyjazdu. W celu lepszej identyfikacji terminali ich numery wraz z funkcją są przechowywane w osobnej tabeli.

# 5. Opis implementacji i zastosowanych rozwiązań

## 5.1. Front-end

Front-end aplikacji został zaimplementowany w technologii MVC (Model-View-Controller) .NET 5.0. Modele oraz kontrolery są napisane w języku C#, natomiast widoki są zaimplementowane z użyciem silnika Razor pages. Za logowanie oraz kontrole dostępu w ramach strony internetowej odpowiedzialny jest pakiet Identity.

## 5.2. Back-end

Do implementacji back-endu zastosowano poniższe technologie:

Klient (czytnik kart):

* Python w wersji 3.8.10 - język programowania wysokiego poziomu umożliwiając korzystanie z dużej ilości bibliotek znacznie usprawniających zadania takie jak generowanie interfejsu użytkownika.
* Protokół MQTT v3.1.1 - prosty protokół transmisji danych oparty o wzorzec publikacja - subskrypcja. Rozwiązanie umożliwia przesył informacji między urządzeniami w ramach zdefiniowanego tematu.
* Biblioteka tkinter - biblioteka języka Python umożliwiająca i usprawniająca tworzenie interfejsu graficznego.
* Biblioteka eclipse paho mqtt - biblioteka kliencka języka Python implementująca protokół MQTT i umożliwiająca komunikację z brokerem.
* Biblioteka ssl - biblioteka kliencka języka Python pozwalająca na ustanowienie zabezpieczonego połączenia sieciowego z wykorzystaniem protokołu TLS i certyfikatów X.509.

Server:

* framework ASP .NET Core 5.0.
* Entity Framework Core.
* baza danych w technologii MySQL działająca w usłudze Azure.
* biblioteka MQTTnet 3.1.1 - biblioteka pozwalająca na wykorzystanie protokołu MQTT w języku C# do komunikacji.

Implementację oraz testy implementacji po stronie czytnika kart wykonano przy użyciu programu Visual Studio Code z zainstalowanym rozszerzeniem Remote-SSH umożliwiającym testowanie kodu na “osobnych” maszynach.

Testy komunikacji MQTT przeprowadzono z wykorzystaniem programu Wireshark w celu sprawdzenia formatu i treści przesyłanych komunikatów oraz systemu wyświetlania komunikatów brokera Mosquitto oraz framework’u ASP .NET.

## 5.3 Opis komunikacji MQTT

### 5.3.1 Ogólny opis komunikacji

Komunikacja odbywa się na porcie 8883 który jest domyślnym portem protokołu MQTT przy użyciu protokołu TLS w wersjach 1.2 oraz 1.3.

Klienci MQTT wysyłają wiadomości ze swoim tematem.

W przypadku bram szlabanów są to odpowiednio:

* gate/e/id\_klienta - dla bramy wjazdowej.
* gate/l/id\_klienta - dla bramy wyjazdowej.

Dla czytników kart temat to:

* reader/id\_klienta.

Bramy szlabanów odczytują wiadomości o tematach:

* gate/e/id\_klienta/r dla bramy wjazdowej.
* gate/l/id\_klienta/r dla bramy wyjazdowej.

Na powyższe dwa tematy serwer odsyła odpowiedź czy szlaban należy podnieść czy nie. W przypadku odmowy podawany jest dokładny powód.

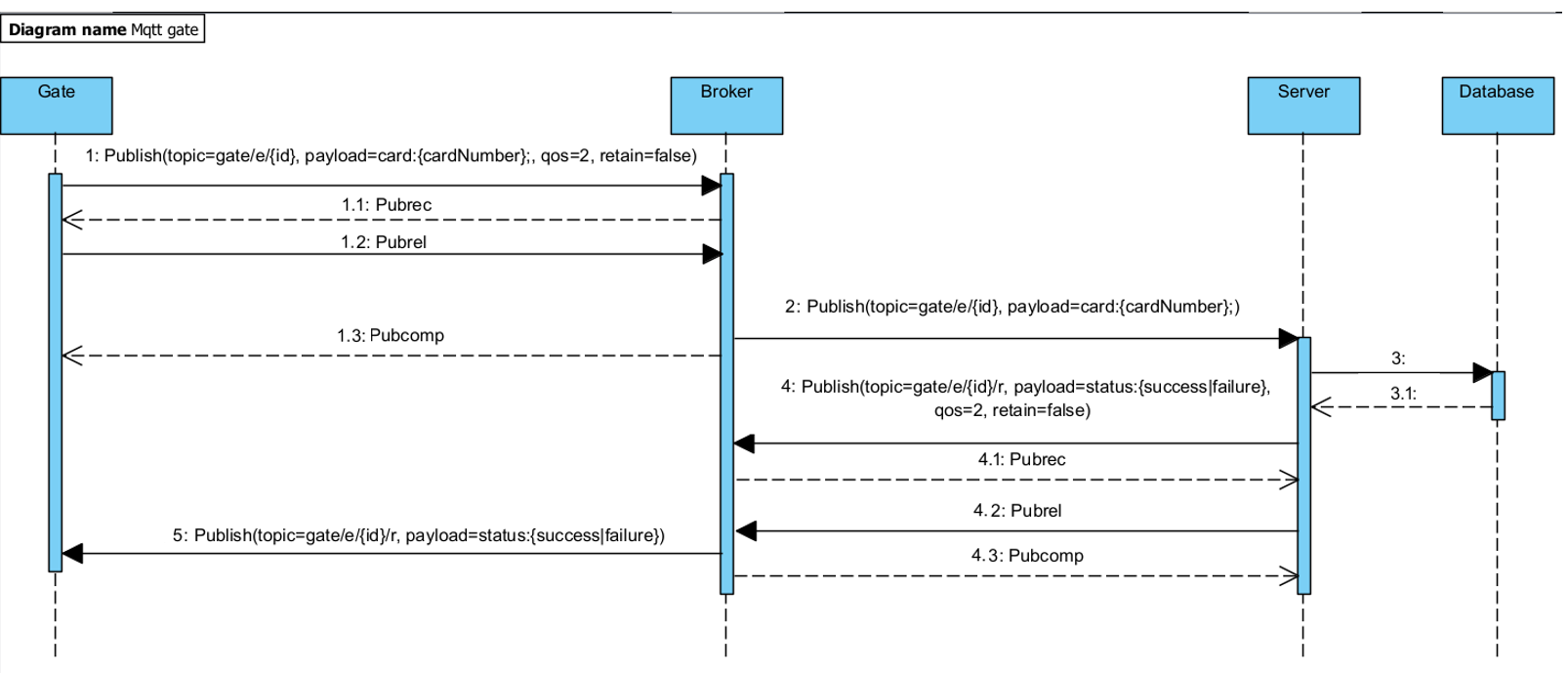
Serwer MQTT odczytuje wszystkie wiadomości wysyłane przez bramy szlabanów i czytniki kart. Następnie po skomunikowaniu z bazą danych i ustaleniu odpowiedzi przesyła odpowiedź na tematy odczytywane przez bramy szlabanów odpowiednio dla bramy wjazdowej i wyjazdowej.

Wszystkie wiadomości wysyłane są z flagą qos=2 co zapewnia, że broker dostarczy każdą wiadomość co najwyżej raz. Ilość przesyłanych danych w systemie nie jest duża, natomiast ważne jest aby każda wiadomość dotarła tylko raz. Inaczej może dojść do sytuacji w których szlabany otrzymają kilkukrotnie polecenie otwarcia po zeskanowaniu tej samej karty.

Użyta jest również flaga retain=false żeby broker nie przesyłał wiadomości jeśli klienci byli rozłączeni. Np. jeśli szlaban utraci połączenie z siecią i odnowi po 5 minutach, nie powinna do niego przyjść wiadomość sprzed 5 minut nakazująca otworzenie.

Serwer również nie musi sprawdzać wiadomości jeśli był rozłączony. Kierowca albo zrezygnował już z wjazdu na parking albo spróbował zeskanować kartę ponownie, nie ma sensu przetwarzać starego żądania. Eliminuje to również problem jak w przypadku szlabanu, gdy wiadomość może przyjść kilka razy.

### 5.3.2 Diagram sekwencji przesyłanych wiadomości



Rys. Diagram sekwencji przesyłanych wiadomości

Diagram dotyczy wiadomości przesyłanych przez szlabany (zarówno wjazdowe jak i wyjazdowe) do serwera. Diagram dla wiadomości przesyłanych przez czytnik kart wygląda podobnie, z tą różnicą, że serwer nie odpowiada wiadomością publish zatem komunikacja kończy się na wiadomości 3.

### 5.3.3 Zabezpieczenia komunikacji MQTT

Całość komunikacji odbywa się przy użyciu protokołu TLS co sprawia, że wiadomości są zaszyfrowane. Klienci szlabanów i czytnika korzystają z wersji TLS 1.3, a serwer z wersji 1.2. Broker pozwala na wersję 1.3 jak i 1.2, decyzja którego protokołu użyć należy do klienta, który się z nim łączy.

Do uwierzytelniania klientów, serwera i brokera używane są certyfikaty X. 509. Każdy klient posiada swój własny certyfikat podpisany przez urząd certyfikacji (CA), broker i serwer posiadają oddzielne certyfikaty. Każda ze stron weryfikuje czy druga strona komunikacji ma certyfikat podpisany przez zaufany urząd certfikacji.

Uwaga: Z powodu pewnych ograniczeń w kodzie języka pythona, aktualnie weryfikacja tożsamości brokera przez klientów jest wyłączona. Jest to spowodowane wyłącznie tym, że do celów projektu certyfikaty są podpisywane lokalnie i serwery CA nie rozpoznają lokalnego CA. Jednakże podpisanie certyfikatu przez znany CA kosztuje dlatego na potrzeby projektu użyty został lokalny CA. Po stronie serwera udało się to obejść poprzez nadpisanie funkcji do weryfikacji tożsamości i podanie jej certyfikatu lokalnego CA. Na brokerze problem nie występuje, ponieważ podczas konfiguracji można podać certyfikat lokalnego CA. Ograniczenie dotyczy systemu Windows, w systemie Linux w dość łatwy sposób można dodać lokalne CA do zaufanych.

Jeśli certyfikaty są prawidłowe następnym etapem uwierzytelniania jest sprawdzenie czy login i hasło podane przez klienta lub serwer zgadzają się z tymi zapisanymi na brokerze w pliku password\_file.

Jeśli powyższe warunki zostaną spełnione może dojść do komunikacji na następujących zasadach:

Serwer może czytać wiadomości wysłane na tematy:

* gate/e/id\_klienta
* gate/l/id\_klienta
* reader/id\_klienta

i wysyłać wiadomości na tematy:

* gate/e/id\_klienta/r
* gate/l/id\_klienta/r

Klienci mogą czytać wiadomości wysłane na tematy:

* gate/e/id\_klienta/r
* gate/l/id\_klienta/r

i wysyłać wiadomości na tematy:

* gate/e/id\_klienta
* gate/l/id\_klienta
* reader/id\_klienta

Klienci mogą czytać tematy teoretycznie nie związane z nimi, jest to spowodowane ograniczeniami acl gdzie bez podawania konkretnej nazwy użytkownika nie można inaczej wydzielić tematów, np. w jakiś sposób grupując użytkowników.

### 5.3.4 Fragment kodu odpowiedzialny za ustawianie komunikacji MQTT na serwerze

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rys. Kod ustawiający komunikację MQTT na serwerze

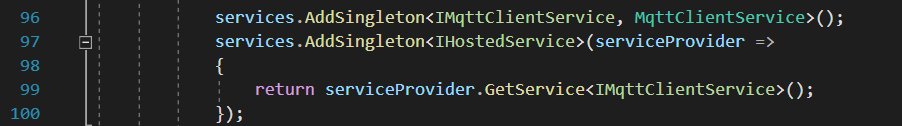
Czynności wykonywane po kolei:

* Ustawiane są nazwa użytkownika, hasła, identyfikator klienta.
* Clean session ustala, że broker ma nie przechowywać wiadomości dla serwera jeśli nie jest on podłączony (żądania obsługiwane są natychmiast albo wcale).
* Keep alive period określa, że co 30 sekund ma być wysyłany komunikat między serwerem i brokerem (jeśli w tym czasie nie są przesyłane inne wiadomości) potwierdzający, że oba są podłączone.
* Ustawiany jest adres brokera i port do komunikacji.
* Następnie ustawiany jest protokół TLS w wersji 1.2 i podawany jest certyfikat używany do uwierzytelniania.
* Dane wczytywane są z pliku konfiguracyjnego appsettings.json, ale nic nie stoi na przeszkodzie aby hasła były wymagane od użytkownika przy starcie serwera.

Obraz zawierający tekst

Opis wygenerowany automatycznie

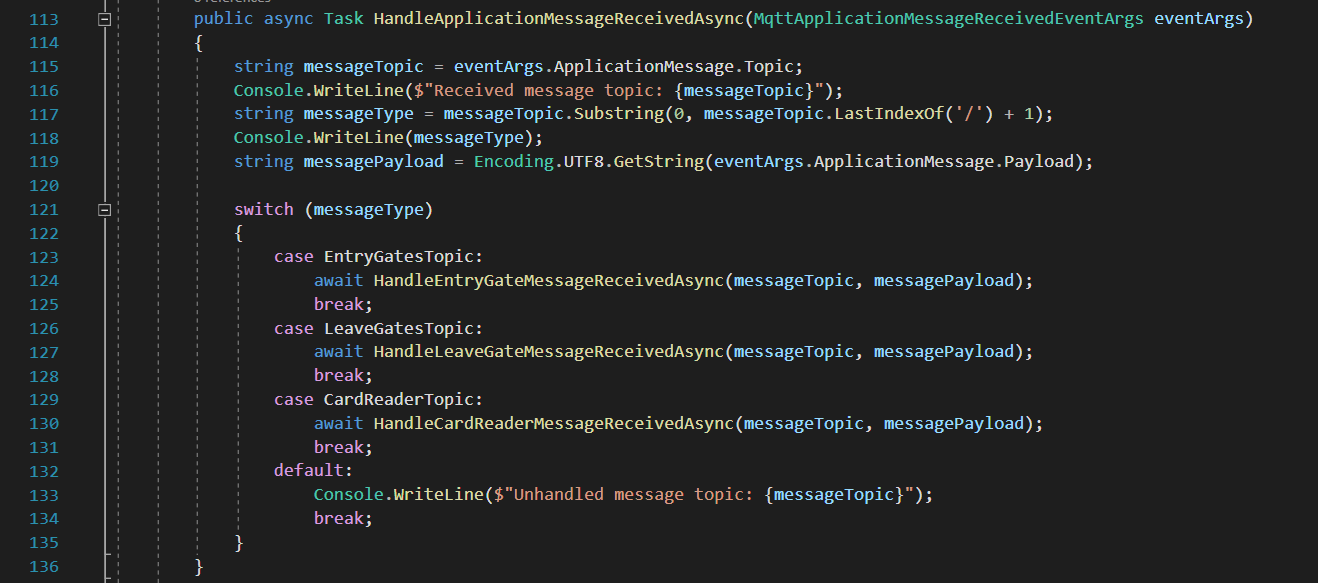
Rys. Plik konfiguracyjny serwera



Rys. Tworzenie usługi klienta MQTT

Tworzenie usługi klienta MQTT polega na stworzeniu jednej instancji klasy MqttClientService która działa od początku uruchomienia serwera do jego zamknięcia. Klasa implementuje interfejs IMqttClientService i jest użyta do stworzenia jednej instancji usługi działającej w tle (usługi IHostedService), która tak jak instancja klienta MQTT działa od początku uruchomienia serwera do jego zamknięcia.

### 5.3.5 Obsługa wiadomości na serwerze



Rys. Obsługa przychodzących wiadomości - ogólnie

Za obsługę wiadomości i innych zdarzeń związanych z MQTT odpowiedzialna jest klasa MqttClientService. Instancja tej klasy po połączeniu z serwerem, które następuje podczas tworzenia, dokonuje subskrybcji na tematy podane w sekcji 5.3.1. Jeśli połączenie zostanie zerwane serwer co 5 sekund ponowi próbę połączenia.

Metoda odpowiedzialna za obsłużenie przychodzących wiadomości odczytuje temat wiadomości, następnie odczytuje temat bez identyfikatora klienta i na jego podstawie wywołuje odpowiednią metodę do obsługi konkretnego typu klienta. Do tej metody przekazuje temat razem z identyfikatorem klienta oraz odczytaną i sformatowaną treść wiadomości.

Obraz zawierający tekst

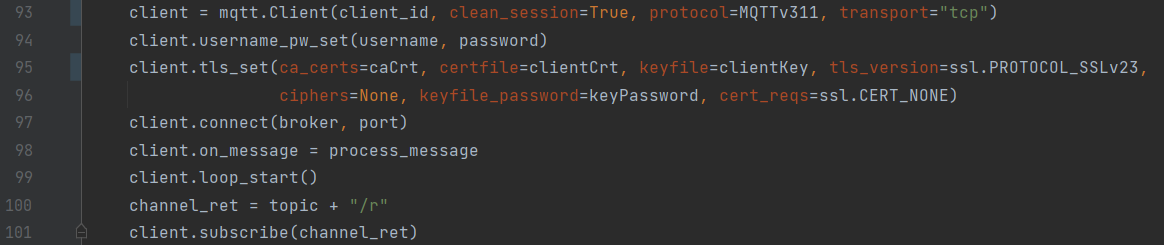
Opis wygenerowany automatycznie

Rys. Obsługa wiadomości od szlabanów wjazdowych

Metoda odpowiedzialna za obsługę szlabanów wjazdowych odczytuje identyfikator szlabanu (z tematu) i numer karty (z wiadomości). Przesyłane dane są w postaci „parametr:wartość;” po średniku mogą być dodane kolejne parametry. Aktualnie używany jest tylko jeden parametr: card.

Następnie wywoływana jest metoda CheckEntry która sprawdza czy szlaban należy otworzyć i odsyła tę informację do klienta szlabanu.

### 5.3.6 Fragmenty kodu odpowiedzialny za ustawianie komunikacji MQTT na klientach (szlabanów i czytnika):



Rys. Kod ustawiający komunikację MQTT u klienta

Dla klientów szlabanów i czytników tworzony jest obiekt klienta MQTT o podanym id. Parametr clean\_session=True ustala, aby po rozłączeniu się klienta żadne wysłane do niego wiadomości nie były zapisywane (podwójne zabezpieczenie, drugie takie ustawienie jest na brokerze). Następnie ustawiana jest wersja protokołu MQTT na 3.1.1 oraz określony zostaje protokół do transportu TCP.

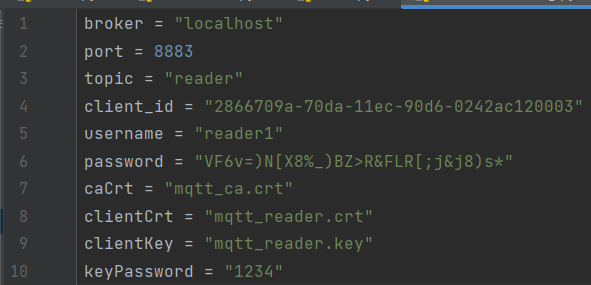
Przed ustawieniami sesji ustawiana jest nazwa użytkownika i hasło.

Konfigurowana jest sesja TLS, w celu poprawnego działania ustawiane są następujące parametry:

* ca\_certs - certyfikat CA który weryfikuje tożsamość brokera.
* certfile - certfikat klienta.
* keyfile - klucz klienta.
* tls\_version - klient będzie korzystał z najlepszej dostępnej wersji TLS (lub SSL) na brokerze, w tym wypadku TLSv1.3.
* ciphers - użycie domyślnych algorytmów do szyfrowania wiadomości..
* keyfile\_password - hasło do pliku z kluczem klienta.
* cert\_reqs - parametr określający czy weryfikować tożsamość brokera, aktualnie nie weryfikowana z powodu podpisywania certyfikatu brokera lokalnie, w środowisku produkcyjnym powinno być ustawione CERT\_REQUIRED.

Następnie następuje połączenie z brokerem o konkretnym adresie ip i porcie. Ustawiona zostaje metoda do obsługi przychodzących wiadomości (on\_message). Uruchomiony zostaje klient i subskrypcja na odpowiednie tematy (dla czytnika kart kod wygląda identycznie, ale nie są wykonywane linijki 101, 100 i 98 z *wyżej*).

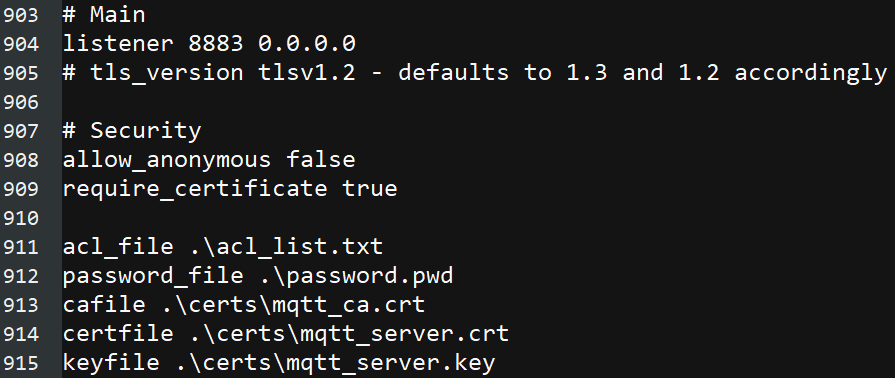
Wszystkie dane klienta, połączenia TLS i brokera pochodzą z pliku konfiguracyjnego klienta.



Rys. Plik konfiguracyjny klienta

W środowisku produkcyjnym wartości client\_id, username i password oraz ścieżki do plików powinny być przechowywane w pamięci tak aby odczytanie ich było niemożliwe.

### 5.3.7 Ustawienia brokera



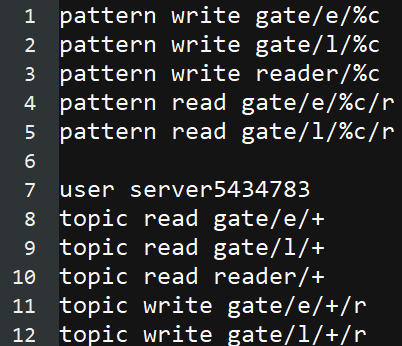
Rys. Ustawienia zabezpieczeń brokera

Aktualnie broker korzysta z portu 8883 na adresie localhost (w środowisku produkcyjnym powinien to być adres maszyny na której uruchomiony będzie broker).

Obsługiwane są połączenia wykorzystujące protokoły TLS w wersji 1.2 lub 1.3. Zabronione jest połączenie klientów bez loginu i hasła, a podane dane muszą być zgodne z tymi przechowywanymi w pliku password\_file podanym brokerowi. Broker wymaga również od klientów prawidłowych certyfikatów, podpisanych przez CA podane brokerowi w pliku cafile.

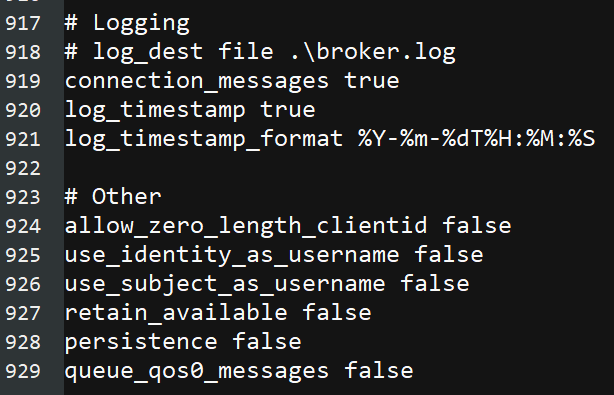
Szczegóły zabezpieczeń:

* listener 8883 0.0.0.0 - ustala uruchomienie brokera na porcie 8883 na adresie localhost.
* tls\_version - można ustawić jedną wymaganą od klientów wersję TLS, jeśli zostawi się tę wartość jako domyślną broker zezwoli na połączenia z wersją 1.2 lub 1.3.
* allow\_anonymous - zezwalanie na połączenia tylko użytkowników z podaną nazwą użytkownika i hasłem.
* require\_certificate - ustawienie wymaga od klientów prawidłowych certyfikatów od klientów. Certyfikaty muszą być podpisane przez CA podane w pliku cafile.
* acl\_file - plik acl ustalający tematy dla połączonych urządzeń niżej.
* password\_file - plik z loginami i (zaszyfrowanymi) hasłami klientów.
* cafile - plik z certyfikatem CA które podpisywało certyfikaty klientów.
* certfile - certyfikat brokera.
* keyfile - klucz prywatny brokera.



Rys. Plik acl określający dostępne tematy

Plik acl ustala jakie tematy mogą być wykorzystywane do komunikacji. W pierwszej sekcji znajdują się tematy dla każdego klienta podłączonego do brokera. W drugiej tematy tylko dla klienta o nazwie server5434783.



Rys. Inne ustawienia brokera

Inne ustawienia zawierają konfigurację formatu wypisywanych wiadomości, ustawienia nazw użytkownika, zapamiętywania wiadomości w bazie danych oraz zapamiętywania i przesyłania wiadomości dla rozłączonych klientów.

### 5.3.8 Przykład komunikacji

Dzięki programowi Wireshark możemy zobaczyć jak wygląda ruch pakietów TCP na porcie 8883 (komunikację MQTT).

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rys. Połączenie serwera z brokerem

Po uruchomieniu brokera i serwera widzimy rozpoczecie komunikacji przy użyciu protokołu TLSv1.2. Następuje wymiana i sprawdzenie certyfikatów oraz ustanowienie połączenia (Handshake). Następnie w application data następuje subskrypcja serwera. Wiadomości jest kilka, dla każdego tematu na który serwer się subskrybuje wysyłana jest osobna wiadomość.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rys. Dane z połączenia serwera z brokerem

Jeśli zajrzymy do wnętrza przesyłanych pakietów okaże się, że dane są nieczytelne i niezrozumiałe. Dostępna jest jedynie informacja o protokole MQTT, protokole TLS oraz wielkości przesyłanych danych.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rys. Połączenie klienta z brokerem

Następnie uruchamiam klienta MQTT (szlaban wjazdowy w tym przypadku) i następuje podobny proces jak dla serwera (inna wersja protokołu).

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rys. Dane przesłane między klientem, a serwerem

Następnie wysyłany jest numer odczytanej karty RFID. Ponieważ w całej komunikacji ustawiony jest QoS 2, oprócz danych wysyłanych jest też dużo informacji zwrotnych o potwierdzeniach.

Jednakże po wersjach protokołu TLS i numerach portów łatwo można zauważyć, że pierwsza wiadomość wysłana z użyciem protokołu TLSv1.2 to wiadomość od brokera do serwera. W zaznaczonym fragmencie dane o wczytanej karcie trafiają na serwer i niemożliwe jest odczytanie treści (treść wiadomości to „card:25425”).

# 6. Opis implementacji bazy danych

Baza danych została wygenerowana za pomocą Entity Framework Core na podstawie modeli utworzonych w ramach wzorca MVC. W celu poprawnego działania wykorzystanego frameworku konieczne jest dołączenie do projektu wymaganych pakietów NuGet.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rys. Menager pakietów NuGet

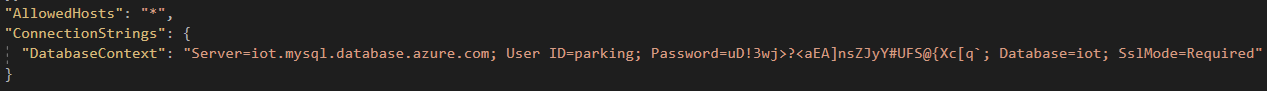
Konfiguracja tego frameworku, przede wszystkim wskazanie połączenia do bazy danych, znajduje się w pliku Startup.cs

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rys. Fragment metody ConfigurateServices odpowiedzialny za skonfigurowanie polaczenia z bazą danych

ConnectionString znajduje się w pliku konfiguracyjnym appsettings.json



Rys.21 ConnectionString w pliku appsettings.json

Komunikacja z bazą danych jest odbywa się z pomocą klasy DatabaseContext, która dziedziczy po klasie IdentityDBContext. Odpowiada ona także za zadeklarowanie kolekcji DbSet<TEntity>, które są używane w celu pobrania danych wybranych tabel z bazy danych. Każda encja z bazy danych ma odpowiadający model. W bazie danych wykorzystywanej przez aplikacje znajdują się tabele:

* RFIDCards
* CardOwners
* Parkings
* ScannedCards
* Terminals

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rys. Klasa DatabaseContext

## 6.1. Analiza otrzymanych wiadomości przez bazę danych

Tak jak zostało wspomniane w dokumentacji MQTT, po odczytaniu identyfikatoru szlabanu i numeru karty, dane te przekazywane są do metod CheckEntry, CheckLeave i CheckCard znajdujących się w klasie DatabaseContext. Metody te analizują dane otrzymanych kart i na podstawie kilku czynników decydują co zostanie przesłane do terminalu. W tym celu został zdefiniowany enum DbResponse zawierający wszystkie możliwe opcje.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rys. Enum DbResponse

Otworzenie szlabanu następuje wyłącznie w sytuacji otrzymania Success. Każda z metod sprawdzających poprawność danych, najpierw analizuje dane terminalu z którego została nadana wiadomość (na przykładzie CheckEntry).

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rys. Metoda CheckEntry

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rys. Metoda CheckTerminal

Za pomocą zapytań LINQ dostępnych dla frameworku .NET zostaje pobrane pierwszy terminal o podanych numerze, jeśli nie zostanie znaleziony żaden terminal, to zostanie zwrócony null, następnie zostaje sprawdzona poprawność tych danych (lub jej brak). W przypadku powodzenia zostaje zwrócony Success. W tym przypadku zostanie następnie wywołana odpowiednia metoda zapisu.

## 6.2. Wjazd na parking

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rys. Metoda SaveEntry

Za pomocą LINQ zostaje pobrana pierwsza karta o podanym numerze, która to jest analizowana przez metodę CheckCard.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rys.27 Metody CheckCard i DeactivateCard

Metoda CheckCard analizuje otrzymaną kartę pod względem poprawności jej danych, oraz przede wszystkim znajduje pierwszy postój na parking dla tej karty, który nie posiada daty wyjazdu (czyli wjazd bez wyjazdu). Jeśli taki parking istnieje to karta nie jest dopuszczona do wjazdu. Za pomocą tej metody aktualizowana jest także aktywność karty, jeżeli z jakiegoś powodu karta jest aktywna, ale mimo to nie posiada właściciela lub dla danego właściciela skończył się termin ważności. W takim przypadku wywoływana jest metoda DeactiveCard, która usuwa właściciela karty oraz ustawia ją na nieaktywną.

Jeśli karta jest prawidłowa zostaje dodany nowy rekord do bazy danych do tabeli Parkings zawierający aktualną datę i godzine (czyli wjazdu), oraz id tej karty. Zostanie też ostatecznie zwrócony Success.

## 6.3. Wyjazd z parkingu

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rys. Metoda SaveLeave

W przypadku wyjazdu pierwsze kroki dzieją się analogicznie do wjazdu. Różnica znajduje się w metodzie SaveLeave gdzie sprawdzamy poprawność danych za pomocą metody CheckParking.

Obraz zawierający tekst, ekran, zrzut ekranu, srebrny

Opis wygenerowany automatycznie

Rys. Metoda CheckParking

SaveParking różni się od metody CheckCard, głównie tym, że pobierane są wszystkie postoje na parkingu które nie mają daty wyjazdu, i tylko w przypadku kiedy lista tych postojów ma jeden element, na kartę zostaje zezwolony wyjazd. W celu uniknięcia problemów administracyjnych, w przypadku kiedy podczas postoju skończy się termin ważności, lub z jakiegoś powodu karta zostanie zablokowana przez administratora, na daną kartę będzie można wyjechać z parkingu (Natomiast nie zostanie dozwolony następny wjazd na ten parking).

## 6.4. Zeskanowanie nowej karty

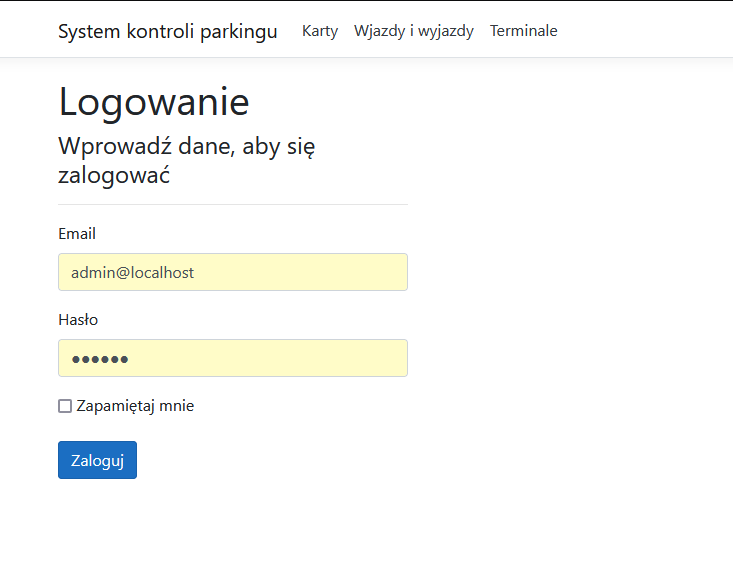
Obraz zawierający tekst

Opis wygenerowany automatycznie

Rys. Metoda SaveCard

W przypadku skanowania karty, po sprawdzeniu terminala tak samo jak w dwóch powyższych przypadkach, kod jest znacznie mniej rozległy, ponieważ sprawdzane jest tylko czy karta o podanym numerze istnieje. Jeśli tak nie jest, karta zostaje dodane do oczekujących na akceptacje administratora i przypisanie jej użytkownika.

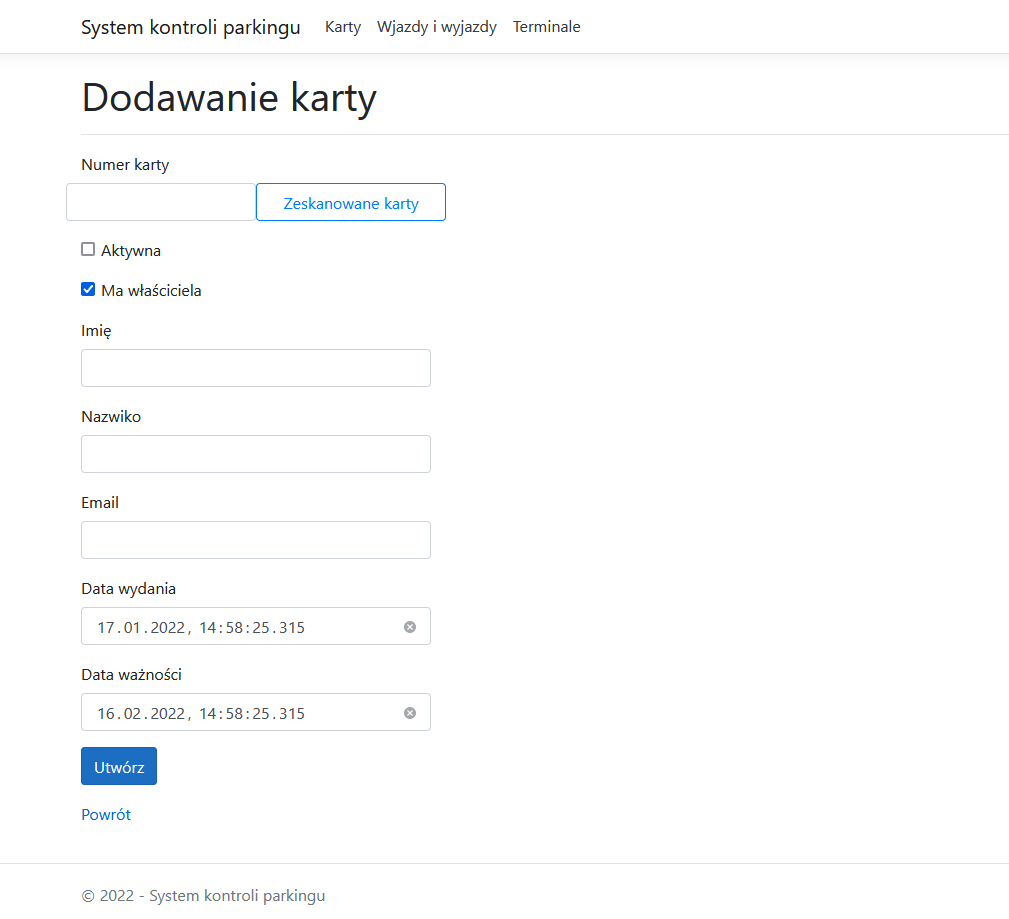
# 7. Opis działania i prezentacja interfejsu



Obraz zawierający tekst

Opis wygenerowany automatycznie

Rys. Ekran główny kart

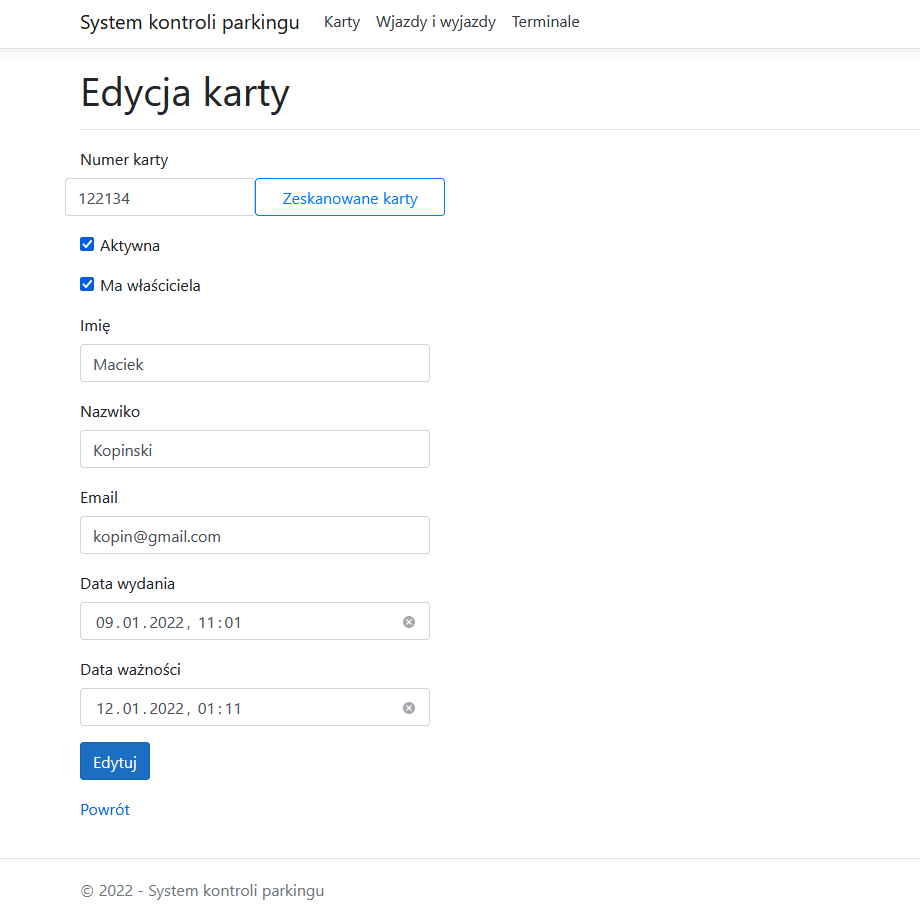


Rys. Dodawanie nowej karty

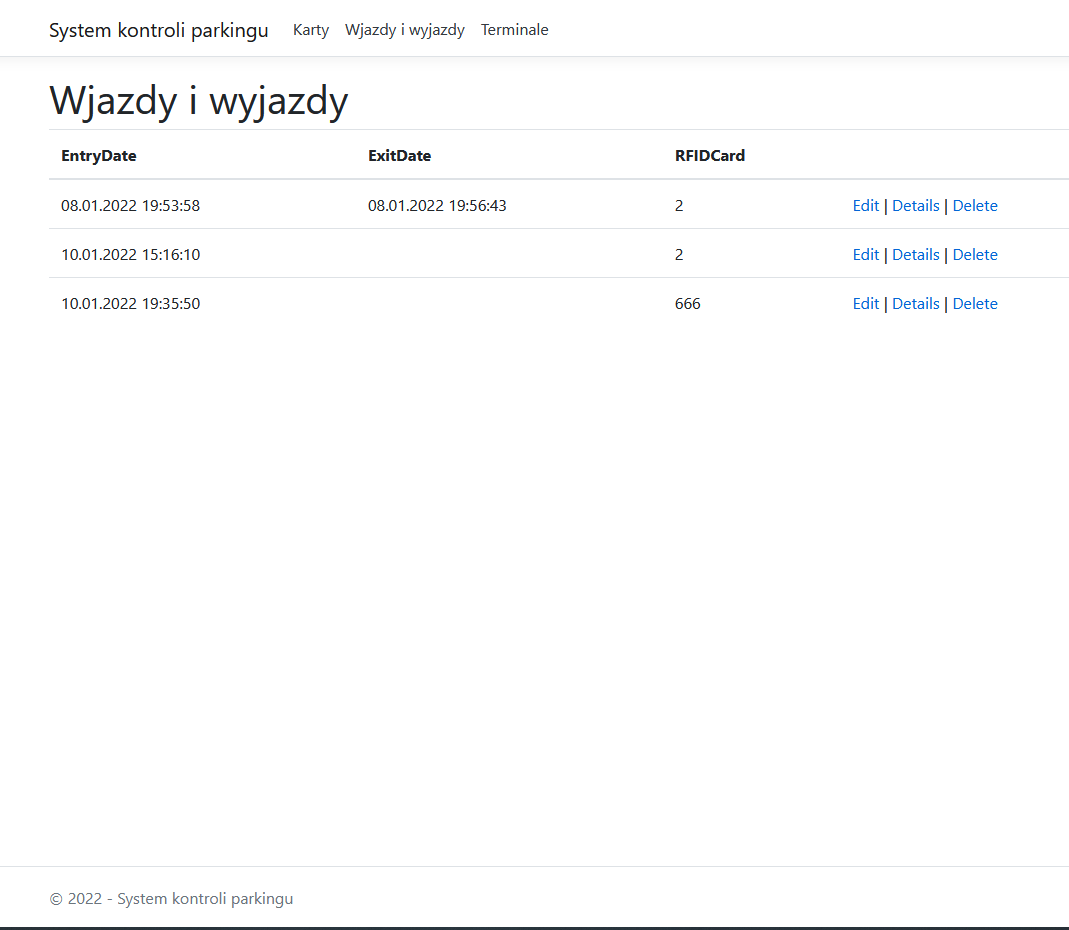
Obraz zawierający tekst

Opis wygenerowany automatycznie

Rys. Wyświetlanie detali karty



Rys. Edycja danych karty



Rys. Ewidencja wjazdów i wyjazdów

Interfejs użytkownika służy do zarządzaniami kartami, ich właścicielami oraz daje możliwość podglądu logów wjazdów i wyjazdów. Użytkownik musi w pierwszej kolejności się zalogować, do tego celu służy ekran „Logowanie”. Zalogowanie się daje dostęp do kolejnych funkcji strony internetowej.

Główny ekran kart pozwala na podgląd wszystkich kart oraz ich użytkowników. Posiada on również możliwość wyszukiwania osób i kart. Z tego widoku użytkownik ma możliwość przejścia do ekranów dodawania nowych kart, edycji, detali lub usuwania karty.

Drugim ekranem jest widok Wjazdów i wyjazdów (rys 22), ekran ten umożliwia analizę logów wjazdów i wyjazdów z parkingu oraz ewentualną edycję wpisów.

Widok Terminale umożliwia administratorowi dodawanie, modyfikacje i usuwanie czytników kart RFID.

# 8. Szczegółowy opis wkładu pracy Autorów

Bartłomiej Jagiełło: - konfiguracja brokera MQTT

- ustalanie formatu wiadomości MQTT 50%

- rejestracja i obsługa serwera mq MQTT w .NET

- obsługa wiadomości po stronie klienta MVC i przekazanie do logiki

- zabezpieczenia protokołu MQTT

- dokumentacja MQTT w Visual Paradigm

- testy integracyjne MQTT

- testy jednostkowe MQTT w .NET i na brokerze

Agata Rudzka: - implementacja obsługi czytnika kart w pythonie (pobieranie wartości,  
 przesył i obsługa informacji zwrotnej)

* dokumentacja przypadków użycia w Visual Paradigm
* testy implementacji
* pisanie dokumentacji, definiowanie wymagań i użytkowników, podsumowanie
* przygotowanie prezentacji systemu w PowerPoincie

Michał Najwer:

* projekt i implementacja interfejsu użytkownika w technologii MVC
* utworzenie serwera bazy danych na platformie Azure
* podłączenie i skonfigurowanie szyfrowania połączenia do aplikacji serwera bazy danych
* dokumentacja opisu działania i prezentacji interfejsu
* utworzenie i zarządzanie repozytorium z kodem projektu na platformie github
* dodanie uwierzytelniania użytkowników na stronie internetowej 50%

Piotr Kołpa:

* projekt bazy danych
* dokumentacja diagramu ERD bazy danych w Visual Paradigm
* implementacja modeli MVC
* implementacja bazy danych za pomocą Entity Framework
* ustalanie formatu wiadomości MQTT 50%
* implementacji analizy otrzymanych wiadomości pod względem poprawności numerów oraz aktywności kart, dat wjazdów i wyjazdów, oraz numerów i funkcji terminali
* implementacja kodów błędów precyzujących dlaczego nie nastąpi otwarcie szlabanu
* dodanie uwierzytelniania użytkowników na stronie internetowej 50%

# 9. Podsumowanie

W ramach projektu zaprojektowano i zaimplementowano system pozwalający na obsługę parkingu z wykorzystaniem urządzeń internetu rzeczy.

Implementacja systemu została zrealizowana w technologiach Python i .NET i spełnia wymagania funkcjonalne opisane w rozdziale drugim niniejszej pracy. Do komunikacji zapewnienia komunikacji między urządzeniami zastosowano protokół MQTT pozwalający na przesyłanie między nimi komunikatów w zdefiniowanych wątkach. Ponadto zabezpieczono komunikację przy użyciu uwierzytelniania z wykorzystaniem certyfikatów przechowywanych na kliencie i brokerze.

Aplikacja internetowa pozwala na rejestrowanie nowych kart i przypisywanie ich do właścicieli, edycję ich danych w tym ich blokowanie i odblokowywanie a także ich usuwanie. Oprócz tego system pozwala na skanowanie kart przy wjeździe i wyjeździe z parkingu zapewniając stosowną odpowiedź a więc pozwalając na wjazd bądź nie. Zostały tym samym spełnione założenia projektowe określone w pierwszych rozdziałach pracy.

W perspektywach rozwoju systemu można wyróżnić:

* poszerzenie systemu na monitorowanie większej liczby parkingów i szlabanów
* wprowadzenie systemu płatności i monitorowania czasu pobytu samochodu kierowcy na parkingu
* wprowadzenie kart „stałego klienta” o specjalnych przywilejach, np.: dłuższym czasie darmowego pobytu na parkingu

# 10. Literatura

1. [Dokumentacja MQTT w .](https://github.com/dotnet/MQTTnet)NET
2. [Zabezpieczanie protokołu MQTT](https://www.hivemq.com/blog/mqtt-security-fundamentals-tls-ssl/)
3. [Ustawianie ACL dla MQTT](https://medium.com/jungletronics/mosquitto-acls-ac062aea3f9)
4. [Dokumentacja mosquitto dla .conf](https://mosquitto.org/man/mosquitto-conf-5.html)
5. [Dokumentacja ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-5.0)
6. [Dokumentacja Entity Framework Core](https://docs.microsoft.com/en-us/ef/core/)
7. [Dokumentacja ASP.NET Identity](https://docs.microsoft.com/en-us/aspnet/identity/)
8. [Dokumentacja LINQ](https://docs.microsoft.com/en-us/dotnet/csharp/linq/)
9. [Dokumentacja Bootstrap](https://getbootstrap.com/docs/5.1/getting-started/introduction/)

## 

# 11. Aneks