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

Spis treści

1. 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.

1.1 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.

1.2 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. Użytkownicy systemu

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

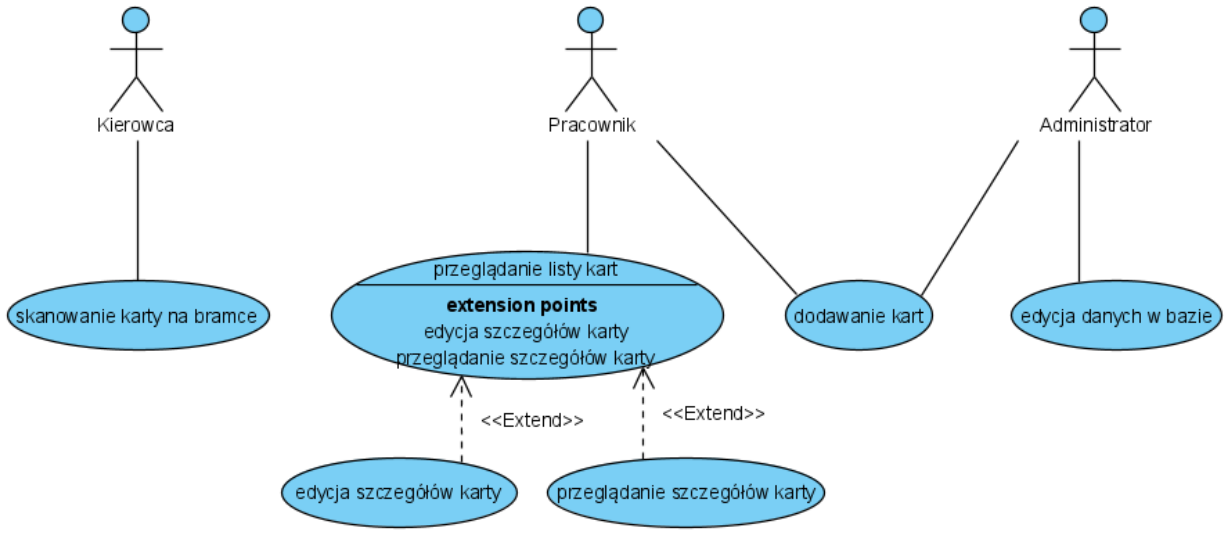
- Pracownik - osoba posiadająca uprawnienia pozwalające na dodawanie nowych kart do systemu; usuwanie, aktywowanie i blokowanie kart; przypisywanie właściciela do karty oraz przeglądanie danych kart parkingowych.

- Kierowca - osoba będąca właścicielem karty RFID, która ma możliwość skanować przy wjeździe i wyjeździe z parkingu.

- Administrator - użytkownik o uprawnieniach pozwalających na zarządzanie kartami RFID i dodawanie nowych szlabanów.

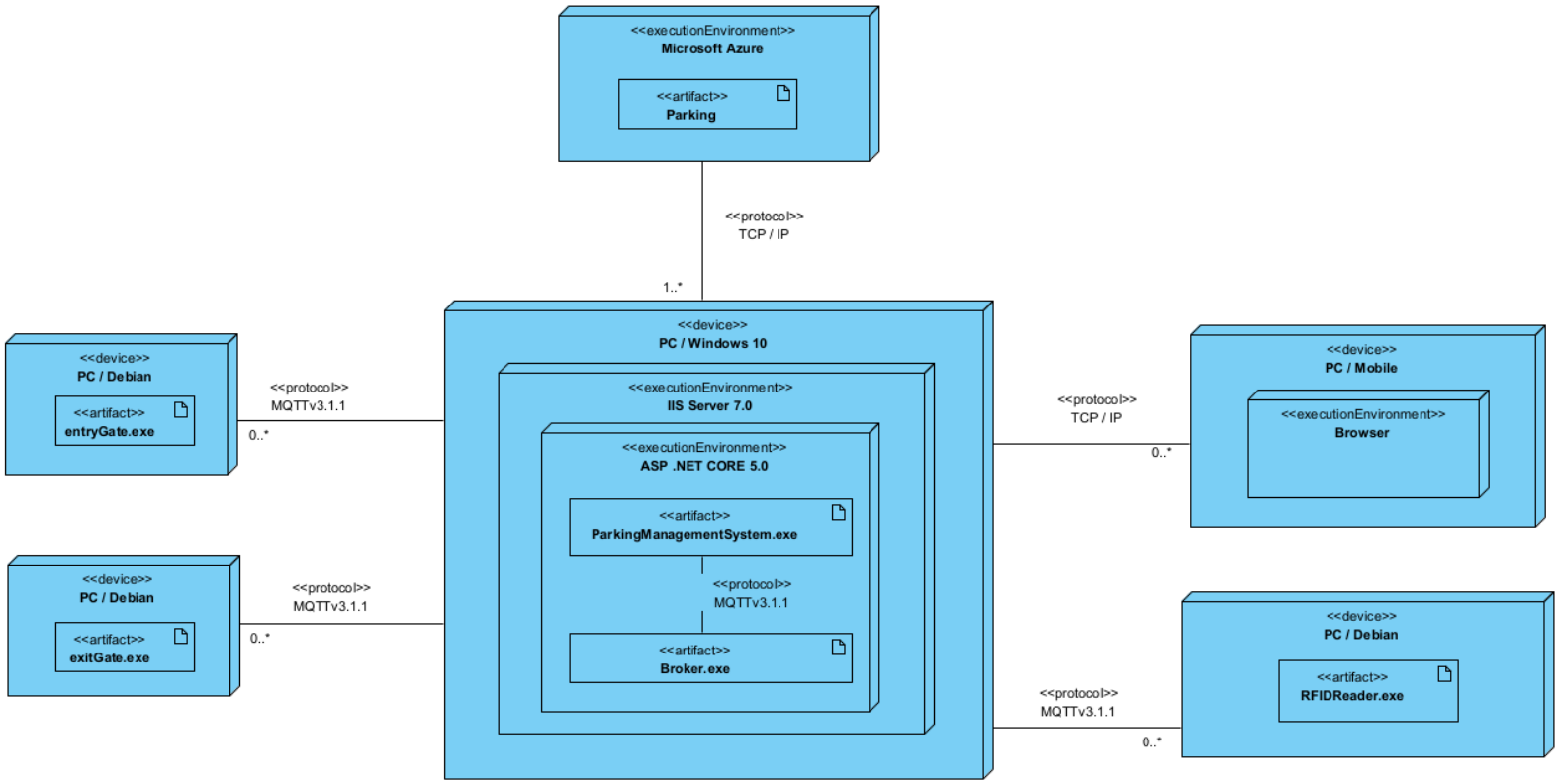
3. Diagram Przypadków Użycia

Na podstawie zdefiniowanych grup użytkowników oraz wymagań funkcjonalnych przygotowano diagram przypadków użycia obrazujący funkcje dostępne dla poszczególnych użytkowników.



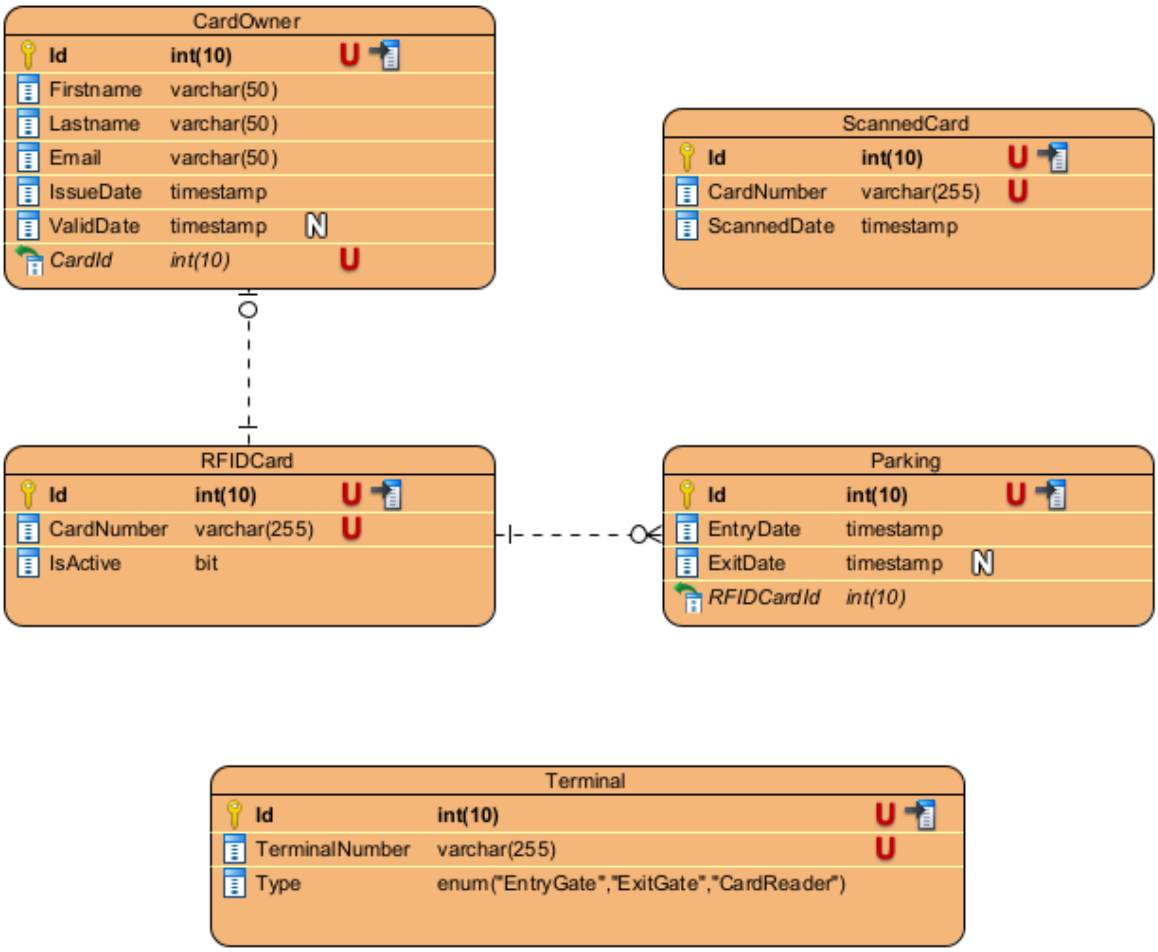
4. Schemat architektury systemu

Architektura opracowanego systemu została zaimplementowana tak jak to przedstawiono na poniższym schemacie. Front-end aplikacji został zaprogramowany w technologii .NET, posiada on dostęp do bazy przechowywanej na platformie Microsoft Azure. Back-end został z kolei przygotowany w językach Python (obsługa czytników) oraz .NET (serwer z brokerem).



5. 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.



6. Opis implementacji i zastosowanych rozwiązań

6.1. Front-end

6.2. 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.

Server:

- framework ASP .NET Core 5.0
- framework Entity Framework Core
- baza danych w technologii MySQL
- 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.

6.2.1 Opis komunikacji mqtt

Komunikacja odbywa się na porcie 8883 który jest domyślnym portem protokołu mqtt przy użyciu protokołu tls.
Klienci mqtt wysyłają wiadomości ze swoim tematem
W przypadku bram szlabanów są to odpowiednio gate/e/id_klienta - dla bramy wjazdowej oraz gate/l/id_klienta - dla bramy wyjazdowej
Bramy subskrybują się na wiadomości o tematach gate/e/id_klienta/r oraz gate/l/id_klienta/r odpowiednio dla bramy wjazdowej i wyjazdowej - na te tematy serwer odsyła odpowiedź czy szlaban należy podnieść czy nie
Czytnik kart wysyła wiadomości z tematem reader/id_klienta

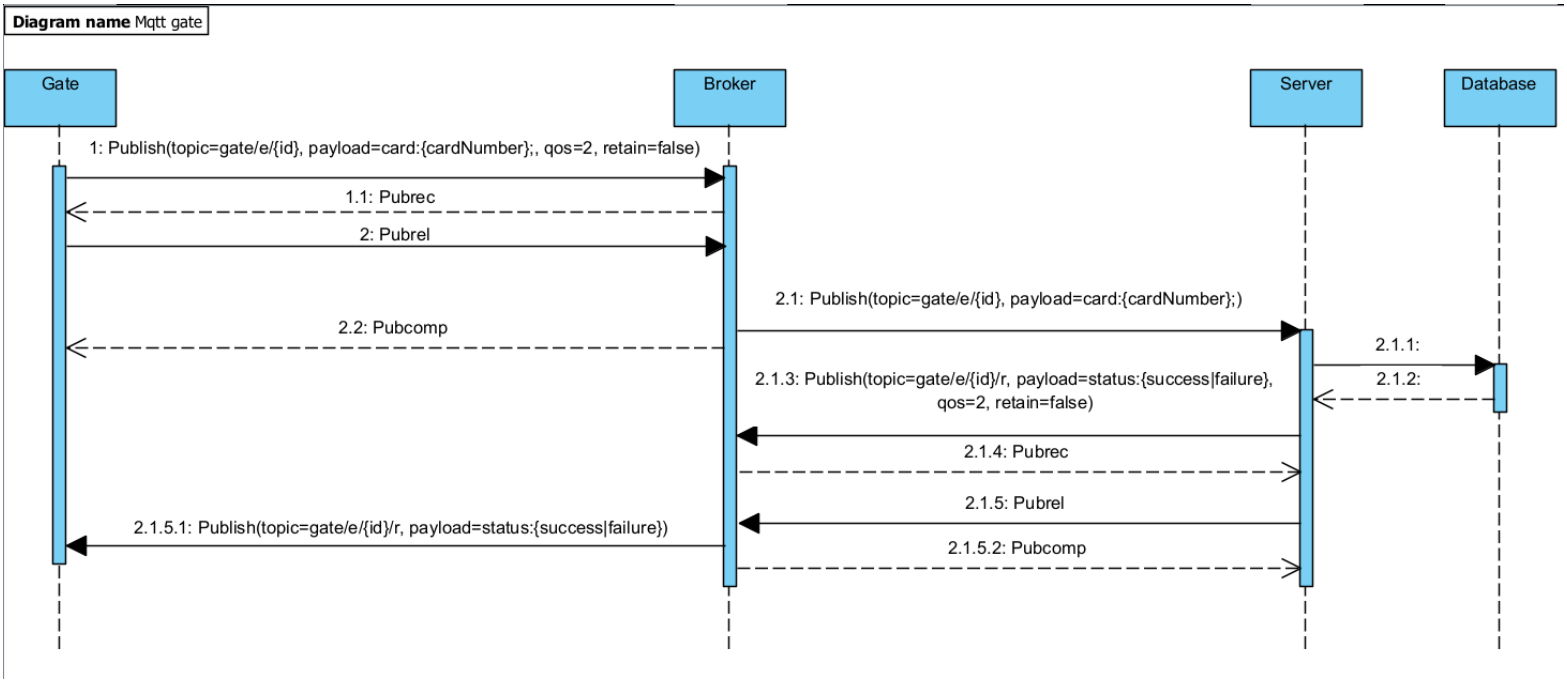
Serwer mqtt subskrybuje się na wszystkie wiadomości z tematami zaczynającymi się od gate/e, gate/l, reader i zawierającymi id klientów.
Następnie po skomunikowaniu z bazą danych i ustaleniu odpowiedzi przesyła odpowiedź na tematy gate/e/id_klienta/r oraz gate/l/id_klienta/r odpowiednio dla bramy wjazdowej 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.
Tak samo serwer nie musi sprawdzać wiadomości jeśli był rozłączony, kierowca albo zrezygnował już z wjazdu na parking albo spróbował ponownie, nie ma sensu przetwarzać starego żądania, eliminuje to też problem jak w przypadku szlabanu.

6.2.1.1 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 2.1.2.



6.2.1.2 Zabezpieczenia komunikacji mqtt

Całość komunikacji odbywa się przy użyciu protokołu tls co sprawia, że wiadomości są zaszyfrowane. Klienci szlabanów i czytnika korzystają z wersji 1.3, serwer z wersji 1.2, brokera pozwala zarówno na wersję 1.3 jak i 1.2. Ponadto do autentykacji klientów i brokera używane są certyfikaty x.509, każdy klient posiada swój własny certyfikat podpisany przez urząd certyfikacji (CA), broker posiada oddzielny certyfikat.

Jeśli certyfikaty są prawidłowe następnym etapem autentykacji jest sprawdzenie czy login i hasło podane przez klienta zgadzają się z tymi zapisanymi na serwerze.
Jeśli powyższe warunki zostaną spełnione może dojść do komunikacji na następujących zasadach (patrz dalej acl):

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

Fragment kodu odpowiedzialny za ustawianie komunikacji mqtt na serwerze

```
64              var optionBuilder = new MqttClientOptionsBuilder();
65              optionBuilder
66                  .WithCredentials(clientSettings.UserName, clientSettings.Password)
67                  .WithClientId(clientSettings.Id)
68                  .WithTcpServer(brokerHostSettings.Host, brokerHostSettings.Port);
69
70              optionBuilder.WithTls(new MqttClientOptionsBuilderTlsParameters()
71              {
72                  UseTls = true,
73                  SslProtocol = System.Security.Authentication.SslProtocols.Tls12,
74                  Certificates = new List<X509Certificate>()
75                  {
76                      _mvcServerCert
77                  },
```

Ustawiane są nazwa użytkownika, hasła, id. Następnie ustawiany jest adres brokera i port do komunikacji. Dane wczytywane z pliku konfiguracyjnego appsettings.json:

```
 9    "BrokerHostSettings": {
10      "Host": "localhost",
11      "Port": 8883
12    },
13    "ClientSettings": {
14      "Id": "server5eb020f043ba8930506acbddserver",
15      "UserName": "server5434783",
16      "Password": "n(n7F8e75~CQVKg8:3Rt`YLDz}d/HR"
17    },
```

Następnie ustawiany jest protokół tls w wersji 1.2 i podawany jest certyfikat używany do autentykacji.

Tworzenie usługi clienta mqtt:

```
96              services.AddSingleton<IMqttClientService, MqttClientService>();
97              services.AddSingleton<IHostedService>(serviceProvider =>
98              {
99                  return serviceProvider.GetService<IMqttClientService>();
100             });
```

6.2.1.3 Obsługa wiadomości na serwerze

Za obsługę wiadomości i innych zdarzeń związanych z mqtt odpowiedzialna jest klasa MqttClientService

Metoda odpowiedzialna za obsłużenie przychodzących wiadomości odczytuje temat wiadomości oraz treść i wywołuje odpowiednią metodę do obsługi klienta:

```
113     public async Task HandleApplicationMessageReceivedAsync(MqttApplicationMessageReceivedEventArgs eventArgs)
114     {
115         string messageTopic = eventArgs.ApplicationMessage.Topic;
116         Console.WriteLine($"Received message topic: {messageTopic}");
117         string messageType = messageTopic.Substring(0, messageTopic.LastIndexOf('/') + 1);
118         Console.WriteLine(messageType);
119         string messagePayload = Encoding.UTF8.GetString(eventArgs.ApplicationMessage.Payload);
120 
121         switch (messageType)
122         {
123             case EntryGatesTopic:
124                 await HandleEntryGateMessageReceivedAsync(messageTopic, messagePayload);
125                 break;
126             case LeaveGatesTopic:
127                 await HandleLeaveGateMessageReceivedAsync(messageTopic, messagePayload);
128                 break;
129             case CardReaderTopic:
130                 await HandleCardReaderMessageReceivedAsync(messageTopic, messagePayload);
131                 break;
132             default:
133                 Console.WriteLine($"Unhandled message topic: {messageTopic}");
134                 break;
135         }
136     }
```

Metoda odpowiedzialna za obsługę szlabanów wjazdowych odczytuje identyfiaktor szlabanu (z tematu) i numer karty (z wiadomości), następnie wywołuje metodę CheckEntry która sprawdza czy szlaban należy otworzyć i odsyła tę informację do klienta szlabanu:

```
        3 references
63      private string GetCardId(string payload, string parameter = "card")
64      {
65          string[] cardParameter = payload.Split(':', ';');
66 
67          return cardParameter[Array.IndexOf(cardParameter, parameter) + 1];
68      }
69 
        1 reference
70      private async Task HandleEntryGateMessageReceivedAsync(string messageTopic, string messagePayload)
71      {
72          string clientId = messageTopic.Substring(messageTopic.LastIndexOf('/'));
73          string cardNumber = GetCardId(messagePayload);
74          Console.WriteLine($"Received card RFID: {cardNumber}");
75          using (var scope = _scopeFactory.CreateScope())
76          {
77              var db = scope.ServiceProvider.GetRequiredService<DatabaseContext>();
78              bool openGate = await db.CheckEntry(clientId, cardNumber);
79              Console.WriteLine($"Open gate: {openGate}");
80              await SendGateResponse(messageTopic, openGate);
81          }
82      }
```

Fragmenty kodu odpowiedzialny za ustawianie komunikacji mqtt na klientach (szlabanów i czytnika):

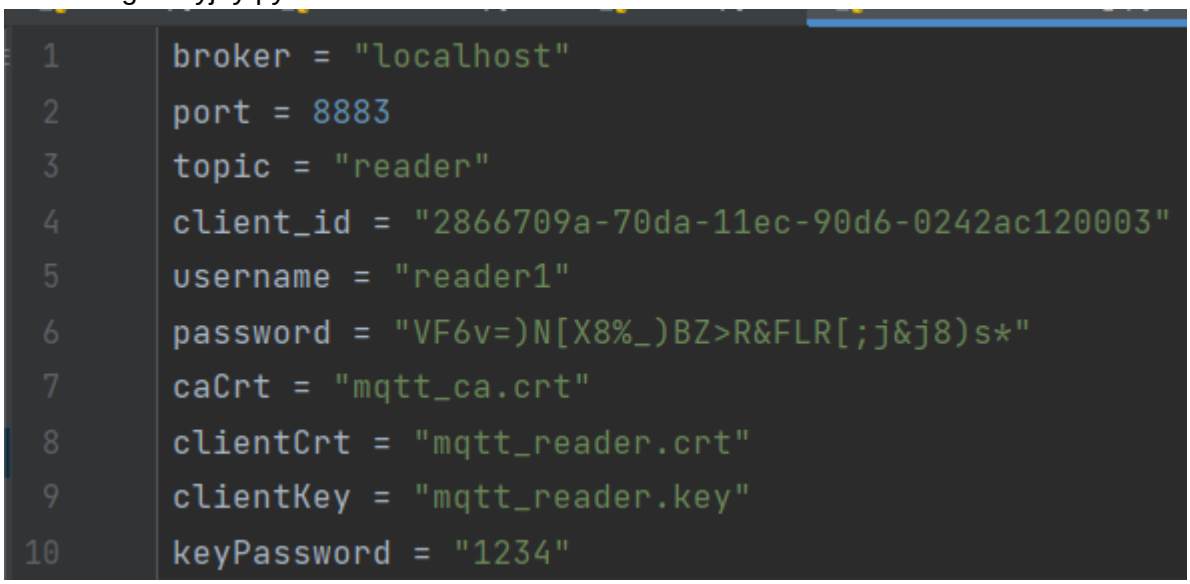
```
93  client = mqtt.Client(client_id, clean_session=True, protocol=MQTTv311, transport="tcp")
94  client.username_pw_set(username, password)
95  client.tls_set(ca_certs=caCrt, certfile=clientCrt, keyfile=clientKey, tls_version=ssl.PROTOCOL_SSLv23,
96                 ciphers=None, keyfile_password=keyPassword, cert_reqs=ssl.CERT_NONE)
97  client.connect(broker, port)
98  client.on_message = process_message
99  client.loop_start()
100 channel_ret = topic + "/r"
101 client.subscribe(channel_ret)
```

Tworzony jest klient o podanym id, nazwie użytkownika i haśle. Ustawiane są wersje protokołu mqtt i protokołu do transportu oraz clean_session=True aby po rozłączeniu się klienta nie były zapisywane żadne wysłane do niego wiadomości (podwójne zabezpieczenie, drugie takie ustawienie jest na brokerze).

W powyższym kodzie przy konfiguracji protokołu TLS ustawiono też 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 1.3)
ciphers - użycie domyślnych algorytmów do szyfrowania wiadomości
keyfile_password - hasło do pliku z kluczem klienta
cert_reqs - czy weryfikować tożsamość brokera, aktualnie nie weryfikowana z powodu podpisywania certyfikatu brokera lokalnie, w środowisku produkcyjnym CERT_REQUIRED

Następnie następuje połączenie z brokerem o adresie ip i porcie wczytanym z pliku, ustawiona zostaje metoda do obsługi przychodzących wiadomości, uruchamiany klient i subskrypcja na odpowiednie tematy (dla czytnika kart nie ma ostatnich 2 linijek i metody on_message)

Plik konfiguracyjny pythona:



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

6.2.1.4 Ustawienia 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 protokoły tls 1.2 i 1.3. Zabronione jest połączenie klientów bez loginu i hasła, a podane muszą być zgodne z tymi przechowywanymi na brokerze.

Acl_file - plik acl (patrz dalej)
Password_file - plik z loginami i (zaszyfrowanymi) hasłami klientów
Broker wymaga też poprawnych certyfikatów od wszystkich swoich klientów podpisanych przez CA o sygnaturze podanej w pliku ustawianym przez cafile.
Certfile i keyfile to odpowiednio certyfikat i klucz prywatny brokera.

```
903 # Main
904 listener 8883 0.0.0.0
905 # tls_version tlsv1.2 - defaults to 1.3 and 1.2 accordingly
906
907 # Security
908 allow_anonymous false
909 require_certificate true
910
911 acl_file .\acl_list.txt
912 password_file .\password.pwd
913 cafile .\certs\mqtt_ca.crt
914 certfile .\certs\mqtt_server.crt
915 keyfile .\certs\mqtt_server.key
```

Acl (opisany przy zabezpieczeniach):

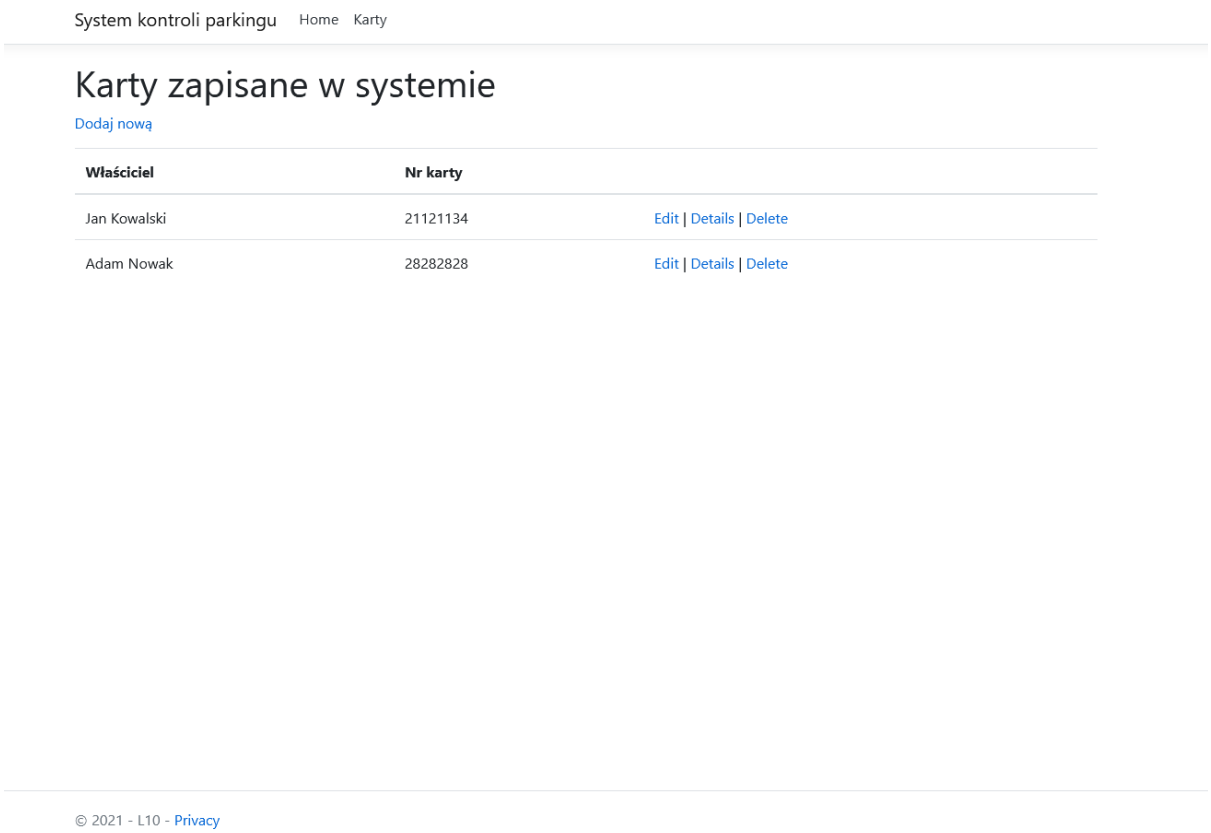
```
 1 pattern write gate/e/%c
 2 pattern write gate/l/%c
 3 pattern write reader/%c
 4 pattern read gate/e/%c/r
 5 pattern read gate/l/%c/r
 6
 7 user server5434783
 8 topic read gate/e/+
 9 topic read gate/l/+
10 topic read reader/+
11 topic write gate/e/+/r
12 topic write gate/l/+/r
```

Inne ustawienia brokera:

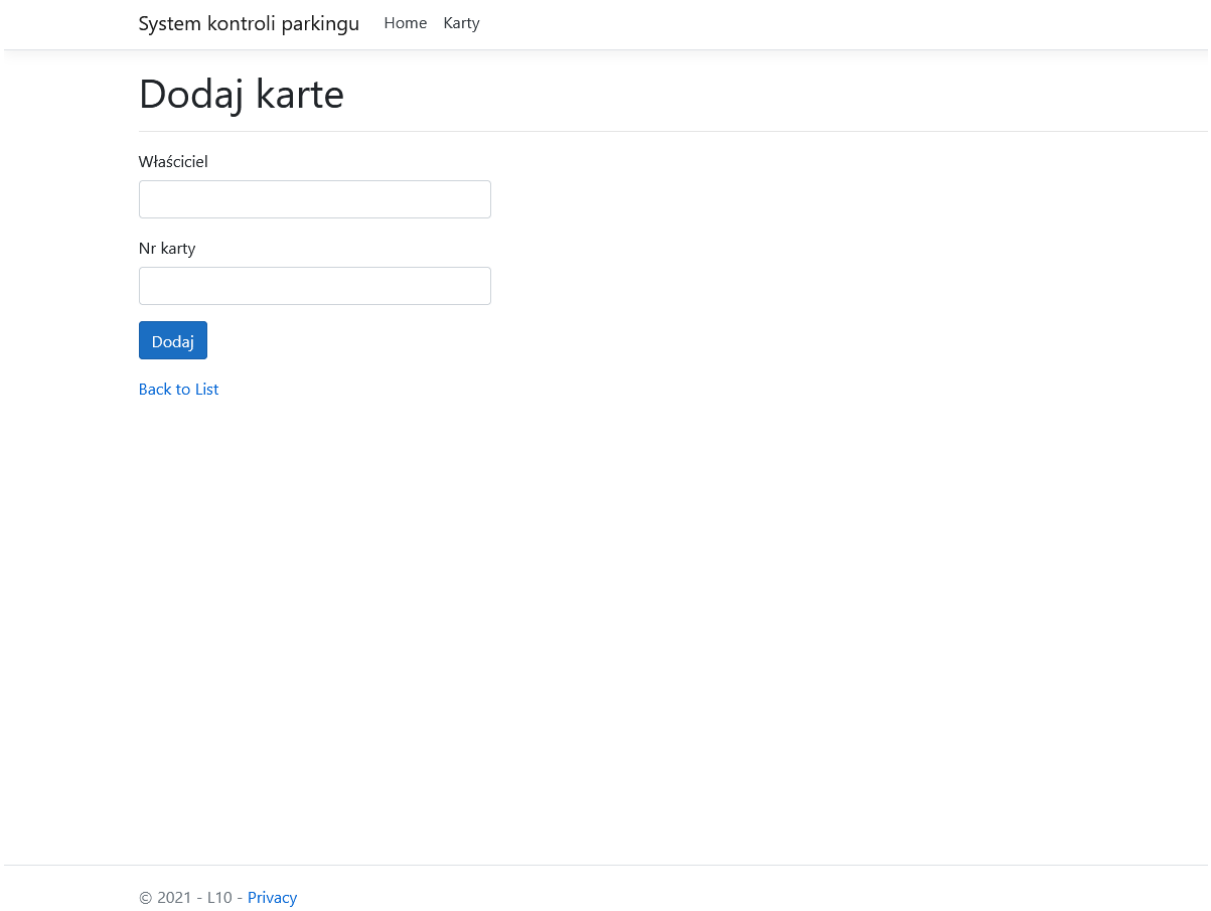
```
# Logging
# log_dest file .\broker.log
connection_messages true
log_timestamp true
log_timestamp_format %Y-%m-%dT%H:%M:%S

# Other
allow_zero_length_clientid false
use_identity_as_username false
use_subject_as_username false
retain_available false
persistence false
queue_qos0_messages false
```

7. Opis działania i prezentacja interfejsu

Widok listy kart, których dane są przechowywane w systemie:



Widok formularza do dodawania nowej karty:



8. Szczegółowy opis wkładu pracy Autorów

Bartłomiej Jagiełło: - konfiguracja brokera mqtt
- ustalanie formatu wiadomości mqtt 50%
- rejestracja i obsługa serwera 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 brokera

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

Michał Najwer:

Piotr Kołpa:
- projekt bazy danych
- dokumentacja diagramu ERD bazy danych w Visual Paradigm
- implementacja modeli MVC
- implementacja bazy danych za pomocą Entity Framework
- wygenerowanie certyfikatów x.509 odpowiedzialnych za zabezpieczenie połączenia między klientem mvc a bazą danych (WIP)
- 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 (WIP)

9. Podsumowanie

10. Literatura

1. Dokumentacja mqqt w .net
2. Zabezpieczanie protokołu mqtt
3. Ustawianie ACL dla mqtt
4. Dokumentacja mosquitto dla .conf

11. Aneks