

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
2. Wymagania projektowe	3
2.1. Wymagania funkcjonalne	3
2.2. Wymagania нефunkcjonalne	3
2.3 Użytkownicy systemu	4
2.4 Diagram Przypadków Użycia	4
3. Schemat architektury systemu	5
4. Schemat bazy danych	6
5. Opis implementacji i zastosowanych rozwiązań	7
5.1. Front-end	7
5.2. Back-end	7
5.3 Opis komunikacji MQTT	8
5.3.1 Ogólny opis komunikacji	8
5.3.2 Diagram sekwencji przesyłanych wiadomości	9
5.3.3 Zabezpieczenia komunikacji MQTT	9
5.3.4 Fragment kodu odpowiedzialny za ustawianie komunikacji MQTT na serwerze	10
5.3.5 Obsługa wiadomości na serwerze	12
5.3.6 Fragmenty kodu odpowiedzialny za ustawianie komunikacji MQTT na klientach (szlabanów i czytelnika):	13
5.3.7 Ustawienia brokera	14
5.3.8 Przykład komunikacji	15
6. Opis implementacji bazy danych	17
Analiza otrzymanych wiadomości przez bazę danych	18
Wjazd na parking	19
Wyjazd z parkingu	21
Zeskanowanie nowej karty	22
7. Opis działania i prezentacja interfejsu	23
8. Szczegółowy opis wkładu pracy Autorów	26
9. Podsumowanie	27
10. Literatura	28
11. Aneks	29

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 zapelnienia 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 нефunkcjonalne

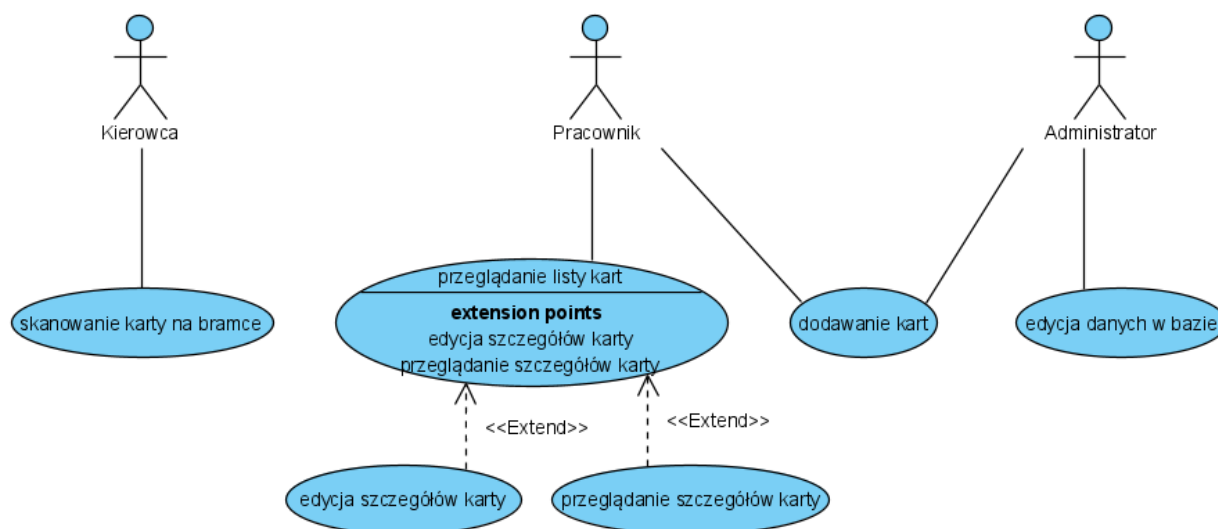
1. Aplikacja webowa działająca na przeglądarkach Google Chrome, Mozilla Firefox, Microsoft Edge, Safari.
2. Aplikacja webowa działająca na systemach windows (11, 10, 8, 7) i linux (przynajmniej ubuntu, debian, redhat).
3. Obsługa 24/7.
4. Możliwość rozszerzenia systemu o następne urządzenia: szlabany 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:

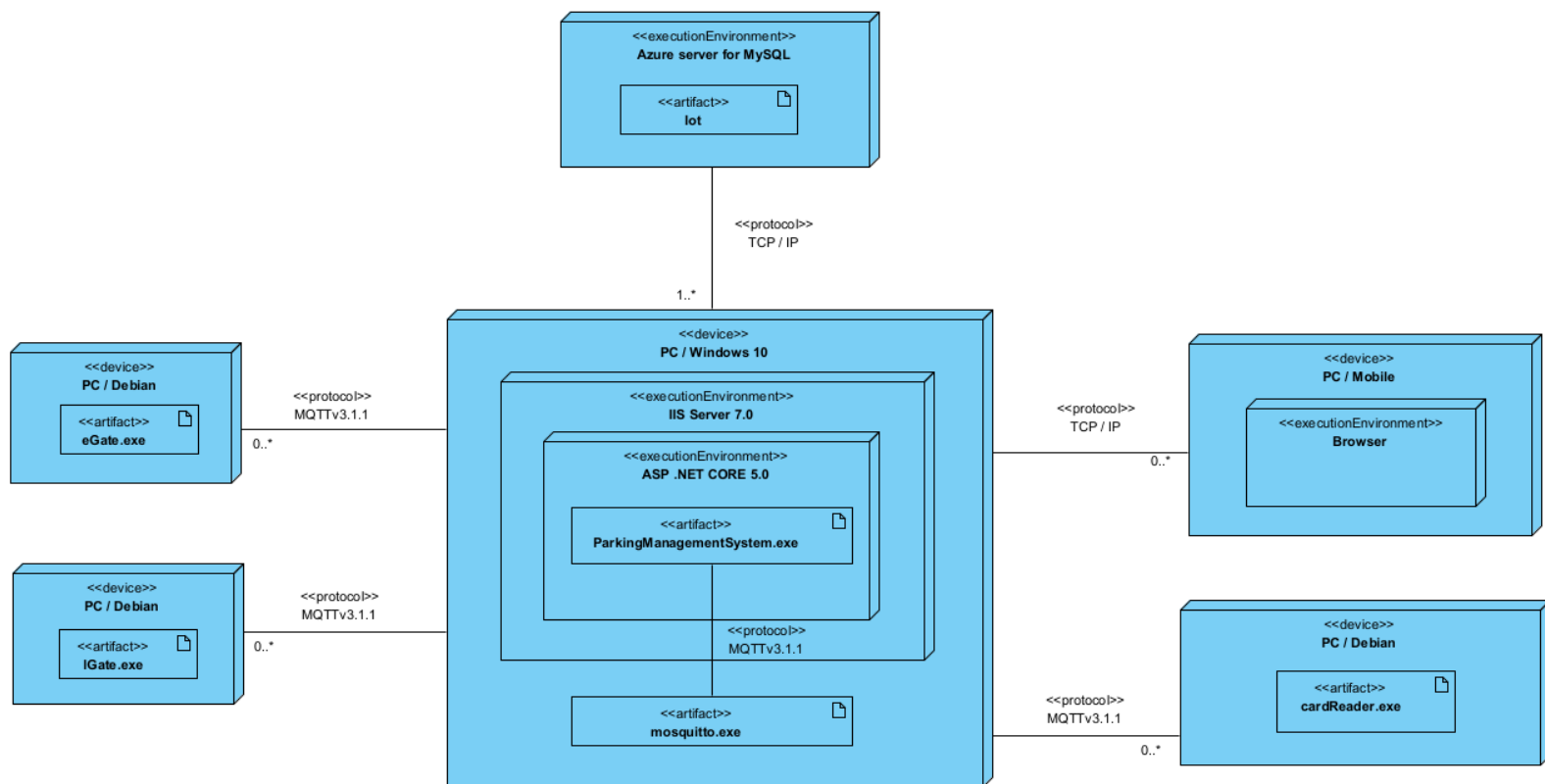
- Pracownik - osoba posiadająca uprawnienia pozwalające na dodawanie nowych kart do systemu; usuwanie, aktywowanie i blokowanie kart; przypisywanie właściciela do karty oraz przeglądanie danych kart parkingowych.
- Kierowca - osoba będąca właścicielem karty RFID, która ma możliwość skanować przy wjeździe i wyjeździe z parkingu.
- Administrator - użytkownik o uprawnieniach pozwalających na zarządzanie kartami RFID i dodawanie nowych szlabanów.

2.4 Diagram Przypadków Użycia



Rys.1 Diagram przypadków użycia

3. Schemat architektury systemu



Rys.2 Diagram rozmieszczenia UML

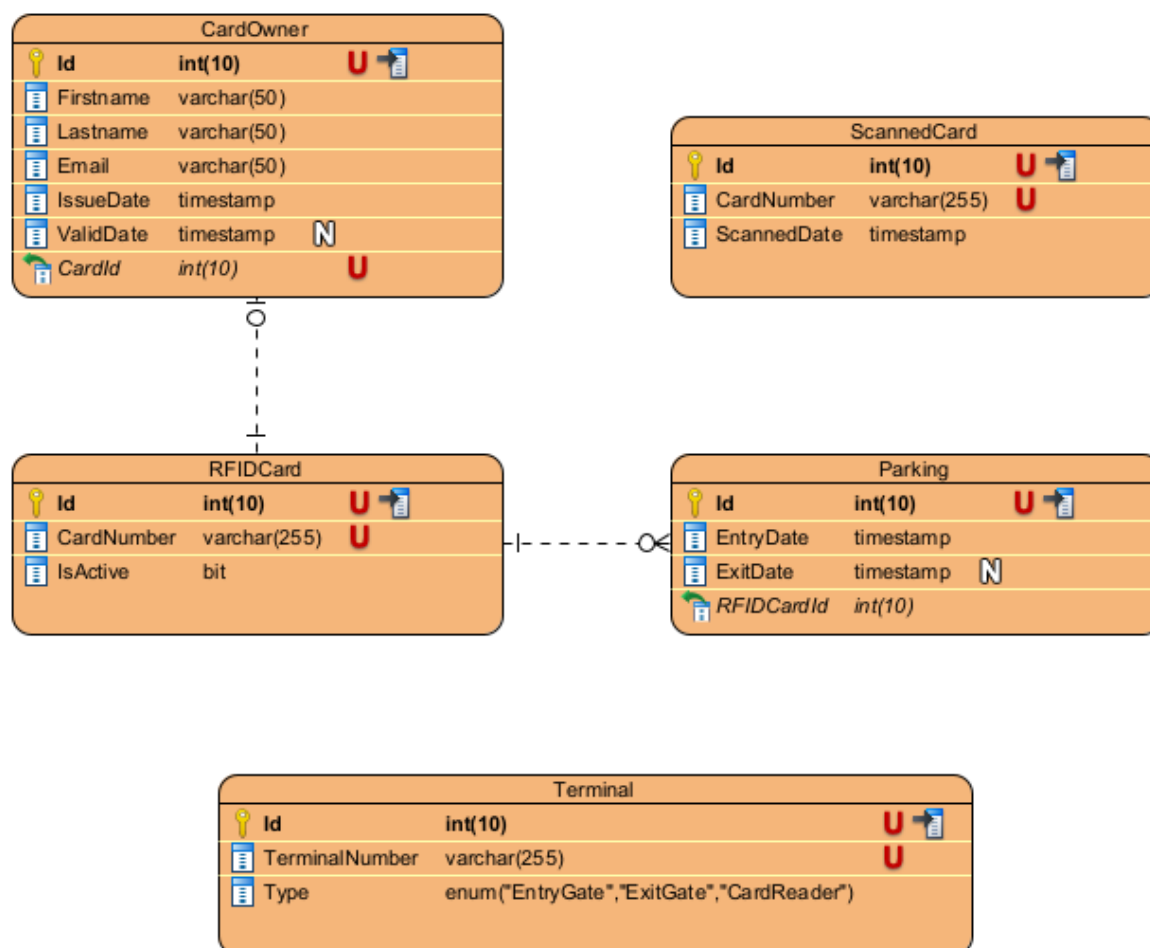
Na powyższym diagramie przedstawiono przykładowy schemat systemu obsługi parkingu. Po lewej stronie diagramu mamy klientów szlabanów odpowiednio wjazdowego i wyjazdowego połączonych z serwerem znajdującym się w centralnej części. Szlabany wyposażone są w czytniki kart RFID i przesyłają dane z odczytanych 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 z 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, wykorzystując ten sam sposób komunikacji jak szlabany.

4. Schemat bazy danych



Rys.3 Schemat bazy danych

W ramach projektu zaprojektowano i zaimplementowano relacyjną bazę danych przechowującą 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 w technologii Razor pages.

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 umożliwiającą 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.
- framework 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.

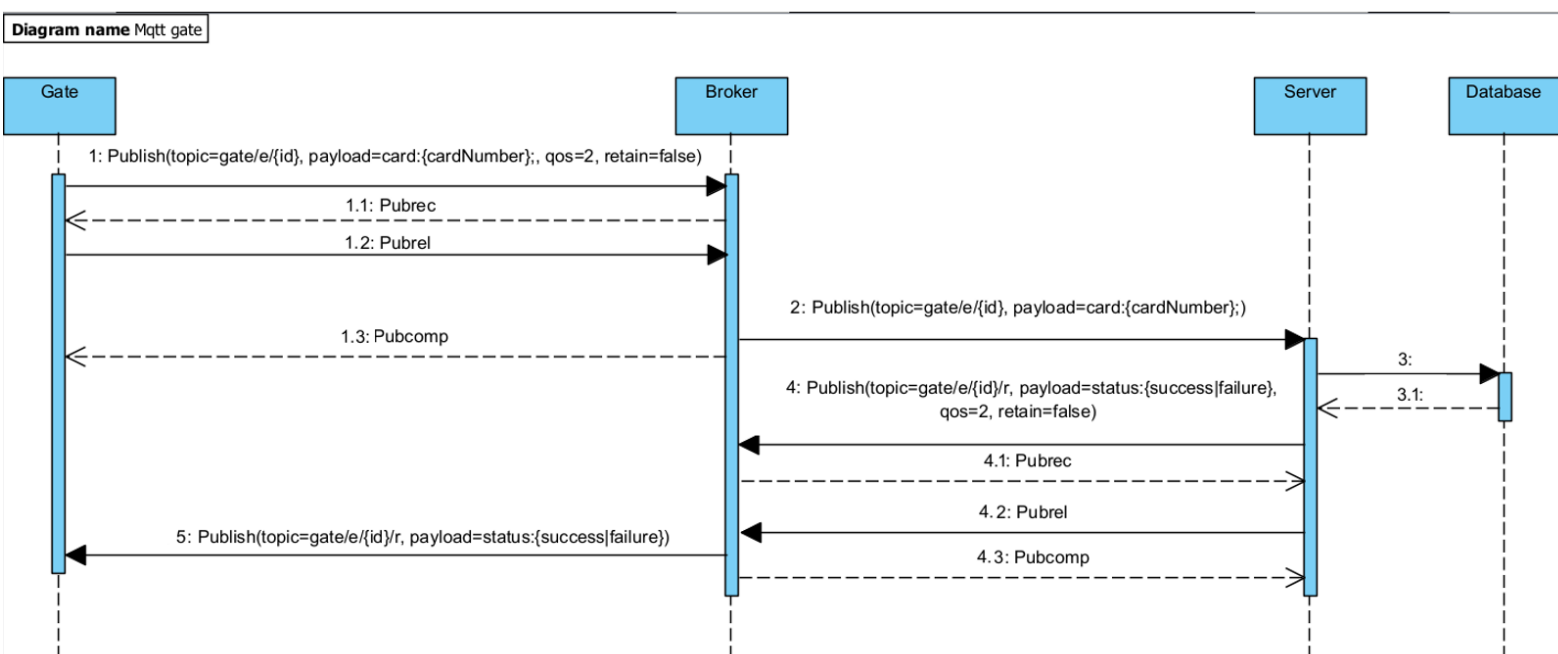
Serwer MQTT odczytuje wszystkie wiadomości wysyłane przez bramy szlabanów i czytniki kart. Następnie po skomunikowaniu z bazą danych i ustaleniu odpowiedzi przesyła odpowiedź na tematy odczytywane przez bramy szlabanów odpowiednio dla bramy wjazdowej i wyjazdowej.

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 przysyłał 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 otwarcie.

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.4 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 certyfikacji.

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

```
61 var optionBuilder = new MqttClientOptionsBuilder();
62 optionBuilder
63     .WithCredentials(clientSettings.UserName, clientSettings.Password)
64     .WithClientId(clientSettings.Id)
65     .WithCleanSession(true)
66     .WithKeepAlivePeriod(new System.TimeSpan(0, 0, 30))
67     .WithTcpServer(brokerHostSettings.Host, brokerHostSettings.Port);
68
69 optionBuilder.WithTls(new MqttClientOptionsBuilderTlsParameters()
70 {
71     UseTls = true,
72     SslProtocol = System.Security.Authentication.SslProtocols.Tls12,
73     Certificates = new List<X509Certificate>()
74     {
75         new X509Certificate2(clientSettings.CertFile, clientSettings.CertPassword)
76     },
77 });
```

Rys.5 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.

```

 9      "BrokerHostSettings": {
10          "Host": "localhost",
11          "Port": 8883,
12          "CaCertFile": "mqtt_ca.crt"
13      },
14      "ClientSettings": {
15          "Id": "server5eb020f043ba8930506acbddserver",
16          "UserName": "server5434783",
17          "Password": "n(n7F8e75~CQVKg8:3Rt`YLDz}d/HR",
18          "CertFile": "client_certificate.pfx",
19          "CertPassword": "1234"
20      },

```

Rys.6 Plik konfiguracyjny serwera

```

services.AddSingleton<IMqttClientService, MqttClientService>();
services.AddSingleton<IHostedService>(serviceProvider =>
{
    return serviceProvider.GetService<IMqttClientService>();
});

```

Rys.7 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

```
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 }
```

Rys.8 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 subskrypcji 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.

```
57 private string GetCardId(string payload, string parameter = "card")
58 {
59     string[] cardParameter = payload.Split(':', ';');
60
61     return cardParameter[Array.IndexOf(cardParameter, parameter) + 1];
62 }
63
64 private async Task HandleEntryGateMessageReceivedAsync(string messageTopic, string messagePayload)
65 {
66     string clientId = messageTopic.Substring(messageTopic.LastIndexOf('/') + 1);
67     string cardNumber = GetCardId(messagePayload);
68     Console.WriteLine($"Received card RFID: {cardNumber}");
69     using (var scope = _scopeFactory.CreateScope())
70     {
71         var db = scope.ServiceProvider.GetRequiredService<DatabaseContext>();
72         DbResponse message = await db.CheckEntry(clientId, cardNumber);
73         Console.WriteLine($"Open gate: {message}");
74         await SendGateResponse(messageTopic, message);
75     }
```

Rys.9 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):

```
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)
```

Rys.10 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` - certyfikat 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.

```
1     broker = "localhost"
2     port = 8883
3     topic = "reader"
4     client_id = "2866709a-70da-11ec-90d6-0242ac120003"
5     username = "reader1"
6     password = "VF6v=)N[X8%_)BZ>R&FLR[;j&j8)s*"
7     caCrt = "mqtt_ca.crt"
8     clientCrt = "mqtt_reader.crt"
9     clientKey = "mqtt_reader.key"
10    keyPassword = "1234"
```

Rys.11 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

```
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
```

Rys. 12 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.

```
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
```

Rys. 13 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.

```

917 # Logging
918 # log_dest file .\broker.log
919 connection_messages true
920 log_timestamp true
921 log_timestamp_format %Y-%m-%dT%H:%M:%S
922
923 # Other
924 allow_zero_length_clientid false
925 use_identity_as_username false
926 use_subject_as_username false
927 retain_available false
928 persistence false
929 queue_qos0_messages false

```

Rys.14 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).

TCP	44	56598 → 8883	[ACK] Seq=1 Ack=1 Win=8192 Len=0
TLSv1.2	215	Client Hello	
TCP	44	8883 → 56598	[ACK] Seq=1 Ack=172 Win=2619648 Len=0
TLSv1.2	2303	Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello Done	
TCP	44	56598 → 8883	[ACK] Seq=172 Ack=2260 Win=5933 Len=0
TLSv1.2	1282	Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Handshake Message	
TCP	44	8883 → 56598	[ACK] Seq=2260 Ack=1410 Win=2618368 Len=0
TLSv1.2	1166	New Session Ticket, Change Cipher Spec, Encrypted Handshake Message	
TCP	44	56598 → 8883	[ACK] Seq=1410 Ack=3382 Win=4811 Len=0
TLSv1.2	170	Application Data	
TCP	44	8883 → 56598	[ACK] Seq=3382 Ack=1536 Win=2618368 Len=0

Rys.15 Połączenie serwera z brokerem

Po uruchomieniu brokera i serwera widzimy rozpoczęcie 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.

6955	437.251170	127.0.0.1	127.0.0.1	TLSv1.2	170 Application Data
6956	437.251195	127.0.0.1	127.0.0.1	TCP	44 8883 → 56598 [ACK] Seq=3382 Ack=1536 Win=2618368 Len=0


```

> Frame 6955: 170 bytes on wire (1360 bits), 170 bytes captured (1360 bits) on interface \Device\NPF_{...} id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 56598, Dst Port: 8883, Seq: 1410, Ack: 3382, Len: 126
▼ Transport Layer Security
  ▼ TLSv1.2 Record Layer: Application Data Protocol: mqtt
    Content Type: Application Data (23)
    Version: TLS 1.2 (0x0303)
    Length: 121
    Encrypted Application Data: 0000000000000000189ebfee83df2784a236a69cd43f02ef65949d5fecfc3429218b53286...
    [Application Data Protocol: mqtt]

```


0000	02 00 00 00 45 00 00 a6	ec c1 40 00 80 06 00 00
0010	7f 00 00 01 7f 00 00 01	dd 16 22 b3 ed bd 25 fb
0020	df 32 0b 87 50 18 12 cb	89 bd 00 00 17 03 03 00
0030	79 00 00 00 00 00 00 00	01 89 eb fe e8 3d f2 78
0040	4a 23 6a 69 cd 43 f0 2e	f6 59 49 d5 fe cf c3 42
0050	92 18 b5 32 86 dd e9 e7	4d 10 e3 1d 6c 24 8e 33
0060	07 28 2c 06 4a dd 09 17	0c de 07 0c ea fa bc 9a
0070	39 2e e6 90 20 a4 ff b8	1e fb 7e db 94 14 cb 5a
0080	0f 79 60 ed 32 d7 0e 23	5c cb ab 9c f5 4a 72 01
0090	44 39 e3 fb 5e ff 31 00	ce de 7a 8e 50 c1 5a c5
00a0	7c 58 cf 43 64 f5 a6 b3	be e4

Rys.16 Dane z połączenia serwera z brokerem

Jeśli zajrzemy 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.


```

TCP      44 56605 → 8883 [ACK] Seq=1 Ack=1 Win=2619648 Len=0
TLsv1.3 561 Client Hello
TCP      44 8883 → 56605 [ACK] Seq=1 Ack=518 Win=2619648 Len=0
TLsv1.3 2470 Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data, Application Data
TCP      44 56605 → 8883 [ACK] Seq=518 Ack=2427 Win=2617344 Len=0
TLsv1.3 2245 Change Cipher Spec, Application Data, Application Data, Application Data
TCP      44 8883 → 56605 [ACK] Seq=2427 Ack=2719 Win=2617344 Len=0
TLsv1.3 160 Application Data
TCP      44 8883 → 56605 [ACK] Seq=2427 Ack=2835 Win=2617344 Len=0

```

Rys.17 Połączenie klienta z brokerem

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

7282	454.502026	127.0.0.1	127.0.0.1	TLsv1.3	125 Application Data
7283	454.502038	127.0.0.1	127.0.0.1	TCP	44 8883 → 56605 [ACK] Seq=4718 Ack=2990 Win=2617088 Len=0
7284	454.502804	127.0.0.1	127.0.0.1	TLsv1.3	70 Application Data
7285	454.502826	127.0.0.1	127.0.0.1	TCP	44 56605 → 8883 [ACK] Seq=2990 Ack=4744 Win=2615040 Len=0
7288	454.507974	127.0.0.1	127.0.0.1	TLsv1.3	70 Application Data
7289	454.508005	127.0.0.1	127.0.0.1	TCP	44 8883 → 56605 [ACK] Seq=4744 Ack=3016 Win=2617088 Len=0
7290	454.508481	127.0.0.1	127.0.0.1	TLsv1.2	130 Application Data
7291	454.508508	127.0.0.1	127.0.0.1	TCP	44 56598 → 8883 [ACK] Seq=1668 Ack=3603 Win=4590 Len=0
7292	454.508599	127.0.0.1	127.0.0.1	TLsv1.3	70 Application Data
7293	454.508611	127.0.0.1	127.0.0.1	TCP	44 56605 → 8883 [ACK] Seq=3016 Ack=4770 Win=2614784 Len=0
7446	463.293257	127.0.0.1	127.0.0.1	TLsv1.3	126 Application Data

> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

> Transmission Control Protocol, Src Port: 8883, Dst Port: 56598, Seq: 3517, Ack: 1668, Len: 86

▼ Transport Layer Security

▼ TLsv1.2 Record Layer: Application Data Protocol: mqtt

Content Type: Application Data (23)

Version: TLS 1.2 (0x0303)

Length: 81

Encrypted Application Data: 2f351d292d82fdb6fee53fc33ea0cd35abb73684e70e912783b00f157419ad497434ddc3...

[Application Data Protocol: mqtt]

```

0000 02 00 00 00 45 00 00 7e ee 03 40 00 80 06 00 00
0010 7f 00 00 01 7f 00 00 01 22 b3 dd 16 df 32 0c 0e
0020 ed bd 26 fd 50 18 27 f3 0e 0b 00 00 03 03 00
0030 51 2f 35 1d 29 2d 82 fd h6 fe e5 3f c3 3e a0 cd
0040 35 ah b7 36 84 e7 0e 91 27 83 h0 0f 15 74 19 ad
0050 49 74 34 dd c3 3f 16 9f 29 53 43 79 2c 7b 29 5h
0060 82 a3 da d7 7b he a5 ef ef 81 e7 4c 46 40 dc e2
0070 ah ha 98 b6 eb 76 29 ac 97 0d 1d 0b 49 e9 d4 c8
0080 e7 8e

```

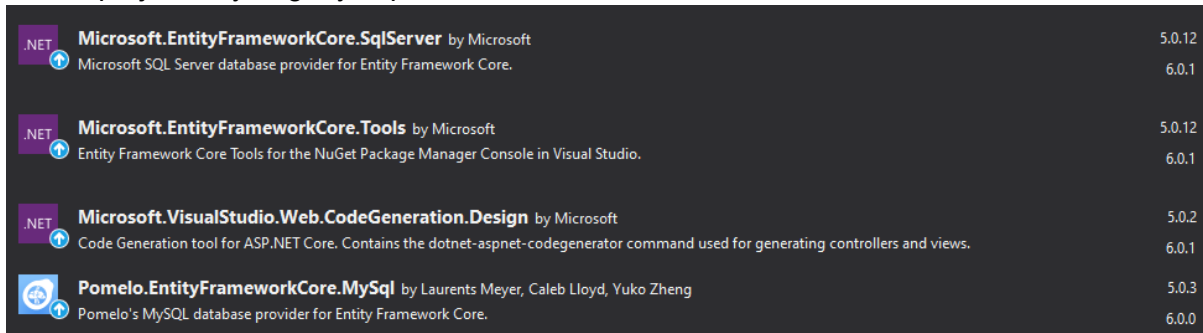
Rys.18 Dane przesłane między klientem, a serwerem





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.



	Microsoft.EntityFrameworkCore.SqlServer by Microsoft	5.0.12
	Microsoft SQL Server database provider for Entity Framework Core.	6.0.1
	Microsoft.EntityFrameworkCore.Tools by Microsoft	5.0.12
	Entity Framework Core Tools for the NuGet Package Manager Console in Visual Studio.	6.0.1
	Microsoft.VisualStudio.Web.CodeGeneration.Design by Microsoft	5.0.2
	Code Generation tool for ASP.NET Core. Contains the dotnet-aspnet-codegenerator command used for generating controllers and views.	6.0.1
	Pomelo.EntityFrameworkCore.MySql by Laurents Meyer, Caleb Lloyd, Yuko Zheng	5.0.3
	Pomelo's MySQL database provider for Entity Framework Core.	6.0.0

Rys.19 Menager pakietów NuGet

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

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<DatabaseContext>(options =>
        options.UseMySQL(
            Configuration.GetConnectionString("DatabaseContext"),
            ServerVersion.AutoDetect(Configuration.GetConnectionString("DatabaseContext"))));
}
```

Rys.20 Fragment metody ConfigureServices odpowiedzialny za skonfigurowanie połączenia z bazą danych

ConnectionString znajduje się w pliku konfiguracyjnym appsettings.json

```
{
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DatabaseContext": "Server=iot.mysql.database.azure.com; User ID=parking; Password=uD13wj?<aEA]nsZJyY#UFS@{Xc[q'; Database=iot; SslMode=Required"
  }
}
```

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 aplikację znajdują się tabele:

- RFIDCards
- CardOwners
- Parkings
- ScannedCards
- Terminals

```

23 references
public class DatabaseContext : IdentityDbContext
{
    0 references
    public DatabaseContext(DbContextOptions<DatabaseContext> options) : base(options)
    {
    }

    0 references
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<RFIDCard>()
            .HasOne(s => s.CardOwner)
            .WithOne(s => s.RFIDCard)
            .HasForeignKey<CardOwner>(s => s.CardId);

        base.OnModelCreating(modelBuilder);
    }

    16 references
    public DbSet<RFIDCard> RFIDCards { get; set; }

    0 references
    public DbSet<CardOwner> CardOwners { get; set; }

    7 references
    public DbSet<Parking> Parkings { get; set; }

    3 references
    public DbSet<ScannedCard> ScannedCards { get; set; }

    8 references
    public DbSet<Terminal> Terminals { get; set; }
}

```

Rys.22 Klasa DatabaseContext

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.

```

public enum DbResponse
{
    Success,
    NotExistingTerminal,
    WrongTerminaltype,
    NotExistingCard,
    DeactivatedCard,
    CardUsedToEntry,
    NotExistingParking,
    CardAlreadyAdded
}

```

Rys.23 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).

```
1 reference
public async Task<DbResponse> CheckEntry(string terminalNumber, string cardNumber)
{
    DbResponse response = CheckTerminal(terminalNumber, TerminalTypes.EntryGate);

    if (response == DbResponse.Success)
        return await SaveEntry(cardNumber);
    else
        return response;
}
```

Rys.24 Metoda CheckEntry

```
3 references
private DbResponse CheckTerminal(string terminalNumber, TerminalTypes type = TerminalTypes.CardReader)
{
    var terminal = Terminals.FirstOrDefault(t => t.TerminalNumber.Equals(terminalNumber));

    if (terminal == null)
        return DbResponse.NotExistingTerminal;
    else if (terminal.Type != type)
        return DbResponse.WrongTerminaltype;
    else
        return DbResponse.Success;
}
```

Rys.25 Metoda CheckTerminal

Za pomocą zapytań LINQ dostępnych dla frameworku .NET zostaje pobrany 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.

Wjazd na parking

```
private async Task<DbResponse> SaveEntry(string cardNumber)
{
    var card = RFIDCards.Include(c => c.Parkings).Include(c => c.CardOwner).FirstOrDefault(c => c.CardNumber.Equals(cardNumber));
    DbResponse response = CheckCard(card);

    if (response == DbResponse.Success)
    {
        Parking parking = new()
        {
            EntryDate = DateTime.Now,
            CardId = card.Id
        };

        Add(parking);
        await SaveChangesAsync();

        return DbResponse.Success;
    }
    else
        return response;
}
```

Rys.26 Metoda SaveEntry

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

```

1 reference
private DbResponse CheckCard(RFIDCard? card)
{
    if (card == null)
        return DbResponse.NotExistingCard;

    var parking = card.Parkings.FirstOrDefault(p => p.ExitDate == null);

    if (parking != null)
        return DbResponse.CardUsedToEntry;
    else
    {
        if (card.IsActive)
            if (card.CardOwner == null || card.CardOwner.ValidDate < DateTime.Now)
                return DeactivateCard(card);
            else
                return DbResponse.Success;
        else
            return DbResponse.DeactivatedCard;
    }
}

1 reference
private DbResponse DeactivateCard(RFIDCard? card)
{
    if (card.CardOwner != null)
        Remove(card.CardOwner);
    card.IsActive = false;
    Update(card);
    SaveChanges();

    return DbResponse.DeactivatedCard;
}

```

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 godzinę (czyli wjazdu), oraz id tej karty. Zostanie też ostatecznie zwrócony Success.

Wyjazd z parkingu

```
1 reference
private async Task<DbResponse> SaveLeave(string cardNumber)
{
    var card = RFIDCards.Include(c => c.Parkings).FirstOrDefault(c => c.CardNumber.Equals(cardNumber));
    DbResponse response = CheckParking(card);

    if (response == DbResponse.Success)
    {
        var parking = card.Parkings.FirstOrDefault(p => p.ExitDate == null);
        parking.ExitDate = DateTime.Now;
        Update(parking);
        await SaveChangesAsync();

        return DbResponse.Success;
    }
    else
        return response;
}
```

Rys.28 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.

```
1 reference
private DbResponse CheckParking(RFIDCard? card)
{
    if (card == null)
        return DbResponse.NotExistingCard;

    var parkings = card.Parkings.Where(p => p.ExitDate == null).ToList();

    if (parkings.Count != 1)
        return DbResponse.NotExistingParking;
    else
        return DbResponse.Success;
}
```

Rys.29 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).

```
1 reference
private async Task<DbResponse> SaveCard(string cardNumber)
{
    var card = RFIDCards.FirstOrDefault(c => c.CardNumber.Equals(cardNumber));

    if (card == null)
    {
        ScannedCard scannedCard = new()
        {
            CardNumber = cardNumber,
            ScanDate = DateTime.Now
        };
        Add(scannedCard);
        await SaveChangesAsync();

        return DbResponse.Success;
    }
    else
        return DbResponse.CardAlreadyAdded;
}
```

Rys.30 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 akceptację administratora i przypisanie jej użytkownika.

7. Opis działania i prezentacja interfejsu

System kontroli parkingu Karty Wjazdy i wyjazdy Terminale

Karty

[Dodaj nową](#)

[Wyszukaj](#) x

Nr karty	Aktywna	Imię	Nazwisko	Email	Data wydania	Data ważności	
122134	Tak	Maciek	Kopinski	kopin@gmail.com	09.01.2022 11:01:00	12.01.2022 01:11:00	Edytuj Detale Usuń
7688	Tak	Piotr	Kolpa	kolpaaa@gmail.com	10.01.2022 19:11:26	09.02.2022 19:11:26	Edytuj Detale Usuń
445322	Tak	Bartek	Jagiello	barti@wp.pl	10.01.2022 19:11:43	09.02.2022 19:11:43	Edytuj Detale Usuń
2	Tak						Edytuj Detale Usuń
696969	Nie						Edytuj Detale Usuń
99898	Tak						Edytuj Detale Usuń
0000	Nie						Edytuj Detale Usuń
666	Tak						Edytuj Detale Usuń

© 2022 - System kontroli parkingu

Rys.31 Ekran główny kart

System kontroli parkingu Karty Wjazdy i wyjazdy Terminale

Dodawanie karty

Numer karty

[Zeskanowane karty](#)

☐ Aktywna
☒ Ma właściciela

Imię

Nazwisko

Email

Data wydania

Data ważności

[Utwórz](#)

[Powrót](#)

© 2022 - System kontroli parkingu

Rys.32 Dodawanie nowej karty

Detale karty

Numer karty	122134
Aktywna	Tak
Imię	Maciek
Nazwisko	Kopinski
Email	kopin@gmail.com
Data wydania	09.01.2022 11:01:00
Data ważności	12.01.2022 01:11:00

[Edytuj](#) | [Powrót](#)

Rys.33 Wyświetlanie detali karty

Edycja karty

Numer karty

122134

[Zeskanowane karty](#)

☒ Aktywna

☒ Ma właściciela

Imię

Maciek

Nazwisko

Kopinski

Email

kopin@gmail.com

Data wydania

09.01.2022, 11:01

Data ważności

12.01.2022, 01:11

[Edytuj](#)

[Powrót](#)

Rys.34 Edycja danych karty

Wjazdy i wyjazdy

EntryDate	ExitDate	RFIDCard	
08.01.2022 19:53:58	08.01.2022 19:56:43	2	Edit Details Delete
10.01.2022 15:16:10		2	Edit Details Delete
10.01.2022 19:35:50		666	Edit Details Delete

Rys.35 Ewidencja wjazdów i wyjazdów

Interfejs użytkownika służy do zarządzania kartami, ich właścicielami oraz daje możliwość podglądu logów wjazdów i wyjazdów. Główny ekran kart (rys 18) 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.

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

Michał Najwer:

- projekt i implementacja interfejsu użytkownika w technologii MVC
- utworzenie, 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

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

9. Podsumowanie

10. Literatura

1. [Dokumentacja MQTT w .NET](#)
2. [Zabezpieczanie protokołu MQTT](#)
3. [Ustawianie ACL dla MQTT](#)
4. [Dokumentacja mosquitto dla .conf](#)
5. [Dokumentacja ASP.NET Core](#)
6. [Dokumentacja Entity Framework Core](#)
7. [Dokumentacja ASP.NET Identity](#)
8. [Dokumentacja LINQ](#)

11. Aneks