

Projekt programistyczny  
System przejazdów pracowniczych w komunikacji miejskiej

Podstawy internetu rzeczy  
Prowadzący laboratorium: dr inż. Krzysztof Chudzik

Kacper Gaudyn, nr indeksu 266873  
Jakub Krąpiec, nr indeksu 266503  
Michał Pesta, nr indeksu 266899  
Michał Trojanowski, nr indeksu 266864

Data ukończenia pracy: 26 stycznia 2024

# Spis treści

<b>1</b>	<b>Wstęp</b>	<b>2</b>
1.1	Problem do rozwiązania . . . . .	2
1.2	Krótki opis implementacji . . . . .	2
<b>2</b>	<b>Wymagania projektowe</b>	<b>3</b>
2.1	Podstawowe wymagania funkcjonalne . . . . .	3
2.1.1	Kierowca . . . . .	3
2.1.2	Pracownik . . . . .	3
2.1.3	Administrator . . . . .	3
2.2	Podstawowe wymagania нефункционалне . . . . .	3
<b>3</b>	<b>Opis architektury systemu</b>	<b>4</b>
3.1	Elementy architektury z opisem . . . . .	4
3.2	Graficzna reprezentacja architektury . . . . .	4
3.3	Baza danych . . . . .	5
3.3.1	Schemat bazy danych . . . . .	5
3.3.2	Scenariusze i ich wpływ na dane . . . . .	5
<b>4</b>	<b>Opis implementacji i zastosowanych rozwiązań</b>	<b>8</b>
4.1	Wykorzystane języki, technologie . . . . .	8
4.2	Najważniejsze funkcje systemu . . . . .	8
4.2.1	Ustalanie trasy przez kierowcę . . . . .	8
4.2.2	Zmiana przystanku przez kierowcę . . . . .	8
4.2.3	Płacenie za przejazd przez pracownika . . . . .	9
4.2.4	Dodanie pieniędzy do konta pracownika przez administratora . . . . .	10
4.3	Implementacja MQTT . . . . .	11
4.3.1	Broker MQTT . . . . .	11
4.3.2	Terminal . . . . .	12
4.4	Implementacja terminala . . . . .	12
4.4.1	Pobieranie danych o kursach i ich przystankach . . . . .	12
<b>5</b>	<b>Opis działania i prezentacja interfejsu</b>	<b>14</b>
5.1	Sposób instalacji i uruchomienia aplikacji . . . . .	14
5.1.1	Zawartość plików . . . . .	14
5.1.2	Instalacja . . . . .	14
5.2	Przedstawienie działania aplikacji . . . . .	15
5.2.1	Panel administracyjny . . . . .	15
5.2.2	Terminal (wersja Raspberry Pi) . . . . .	19
5.2.3	Terminal (wersja testowa) . . . . .	19
<b>6</b>	<b>Wkład pracy Autorów</b>	<b>21</b>
6.1	Pierwszy tydzień (8.01 - 14.01) . . . . .	21
6.2	Drugi tydzień (15.01 - 21.01) . . . . .	21
6.3	Trzeci tydzień (22.01 - 26.01) . . . . .	22
<b>7</b>	<b>Podsumowanie</b>	<b>23</b>
7.1	Stopień zgodności z wymaganiami . . . . .	23
7.1.1	Wymagania funkcjonalne . . . . .	23
7.1.2	Wymagania нефункционалне . . . . .	23
7.2	Napotkane problemy w implementacji . . . . .	23
7.2.1	Uruchomienie systemu na komputerze laboratoryjnym . . . . .	23

# 1 Wstęp

## 1.1 Problem do rozwiązania

W firmie przewozowej komunikacji miejskiej postanowiono ułatwić korzystanie z systemu i połączyć pewne funkcjonalności z Internetem. Do tej pory, pojazdy informację o kolejnych przystankach i trasach komunikowały tylko lokalnie w pojeździe, a pracownicy którzy płacili za przejazd mogli to robić tylko gotówką.

## 1.2 Krótki opis implementacji

Do rozwiązania problemu stworzono system który:

- umożliwia pracownikowi płatność za pomocą karty pracowniczej, którą można zasilać środkami,
- umożliwia kierowcy pojazdu zmianę przystanku na którym się znajduje, jak i trasy po której ma zamiar odbyć kurs,
- umożliwia administratorowi systemu zarządzanie danymi w bazie.

Część systemu która znajduje się w pojeździe jest nazywana terminalem.

Terminal jest obsługiwany przez kierowcę i pracownika.

Terminal posiada wyświetlacz, czytnik kart RFID, pasek LED, dwa przyciski i enkoder.

Do bazy danych przekazywane są dane z terminala (zmiana przystanku bądź trasy, płatność).

## **2 Wymagania projektowe**

### **2.1 Podstawowe wymagania funkcjonalne**

#### **2.1.1 Kierowca**

1. Jako kierowca chcę zmieniać przystanki za pomocą terminalu w trakcie wykonywania kursu
2. Jako kierowca chcę wybierać trasę kursu w terminalu przy rozpoczynaniu jazdy
3. Jako kierowca chcę za pomocą terminalu rozpoczynać jazdę i ją kończyć po przejechaniu trasy
4. Jako kierowca chcę, aby po dotarciu na ostatni przystanek jazda kończyła się automatycznie

#### **2.1.2 Pracownik**

5. Jako pracownik chcę wsiadać na przystanku i płacić za przejazd za pomocą swojej karty pracowniczej
6. Jako pracownik chcę, aby koszt przejazdu był zależny od przejechanych przystanków i był pobierany dopiero po zakończeniu przejazdu (przyłożeniu karty pracowniczej drugi raz)

#### **2.1.3 Administrator**

7. Jako administrator chcę zarządzać<sup>1</sup> wszystkimi kursami
8. Jako administrator chcę zarządzać wszystkimi przystankami
9. Jako administrator chcę zarządzać wszystkimi pracownikami, w szczególności ilością pieniędzy na ich koncie

### **2.2 Podstawowe wymagania нефunkcjonalne**

1. System powinien obsługiwać więcej niż jednego pracownika naraz
2. Operacje wykonywane na terminalu (płatność przez pracownika, zmiana trasy i przystanków przez kierowcę) powinny być jak najszybciej przekazywane do bazy danych

---

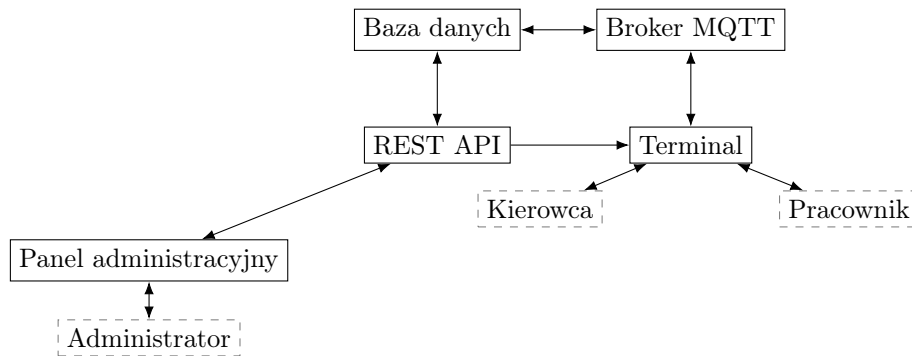
<sup>1</sup>Poprzez zarządzanie rozumiemy operacje: dodawania, odczytu, aktualizowania i usuwania

## 3 Opis architektury systemu

### 3.1 Elementy architektury z opisem

- Baza danych - zawiera wszystkie dane
- REST API - zarządza danymi bazy danych i udostępnia odpowiednie operacje innym podmiotom
- Panel administracyjny - służy administratorom do zarządzania danymi
- Broker MQTT - komunikuje się z terminalami w pojazdach i przekazuje informacje o przejechanych przystankach i płatnościach
- Terminal - znajduje się w każdym pojeździe, umożliwia kierowcy ustalanie trasy i zmianę przystanków, oraz umożliwia pracownikom płacenie za swoje przejazdy
- Użytkownicy
  - Kierowca - kieruje pojazdem i na terminalu może zmieniać trasy, przystanki
  - Pracownik - może wsiadać do pojazdów i płacić w terminalu za przejazd kartą
  - Administrator - zarządza danymi w systemie

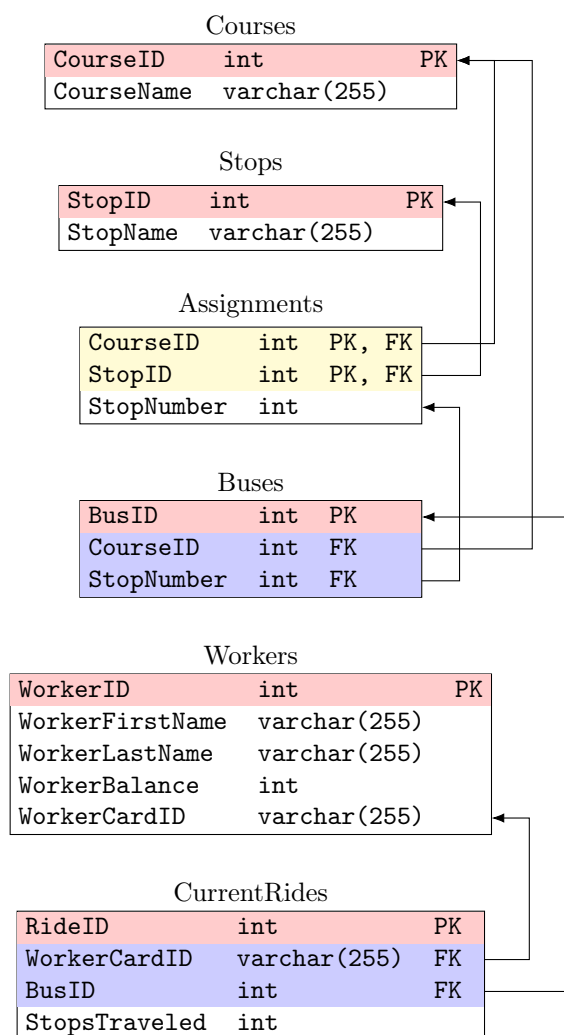
### 3.2 Graficzna reprezentacja architektury



Rysunek 1: Diagram elementów architektury z kierunkami przekazywania danych

### 3.3 Baza danych

#### 3.3.1 Schemat bazy danych



Rysunek 2: Tabele bazy danych z zaznaczonymi relacjami między nimi

#### 3.3.2 Scenariusze i ich wpływ na dane

1. Pracownik wsiada na przystanku, przykłada kartę i po przejechaniu 3 przystanków przykłada ją znowu aby zapłacić i wysiada.  
Zakładamy, że każdy przystanek kosztuje 1, czyli pracownik zapłaci 3.

Workers				
WorkerID	WorkerFirstName	WorkerLastName	WorkerBalance	WorkerCardID
1	Jan	Kowalski	100	ABCD

Rysunek 3: Dane w bazie przed wykonaniem scenariusza

Workers				
WorkerID	WorkerFirstName	WorkerLastName	WorkerBalance	WorkerCardID
1	Jan	Kowalski	97	ABCD

CurrentRides			
RideID	WorkerCardID	BusID	StopsTraveled
⋮			
5	ABCD	1	3
⋮			

Rysunek 4: Dane w bazie po wykonaniu scenariusza

W tabeli **Workers** zmieniła się kolumna **WorkerBalance**, a w tabeli **CurrentRides** został dodany nowy rekord.

- Kierowca pojazdu który nie jest na żadnej trasie, ustala nową trasę o nazwie **MediumLengthCourse**.

Courses	
CourseID	CourseName
⋮	
2	MediumLengthCourse
⋮	

Buses		
BusID	CourseID	StopNumber
1	NULL	NULL

Rysunek 5: Dane w bazie przed wykonaniem scenariusza

Buses		
BusID	CourseID	StopNumber
1	2	1

Rysunek 6: Dane w bazie po wykonaniu scenariusza

- Kierowca pojazdu który jest na trasie **ShortCourse**, zmienia przystanek z **Our Company** na **Amusement Park**.

Courses

CourseID	CourseName
⋮	
3	ShortCourse
⋮	

Stops

StopID	StopName
⋮	
12	Our Company
⋮	
14	Amusement Park
⋮	

Assignments

CourseID	StopID	StopNumber
⋮		
3	12	1
3	14	2
⋮		

Buses

BusID	CourseID	StopNumber
1	3	1

Rysunek 7: Dane w bazie przed wykonaniem scenariusza

Buses

BusID	CourseID	StopNumber
1	3	2

Rysunek 8: Dane w bazie po wykonaniu scenariusza



## 4 Opis implementacji i zastosowanych rozwiązań

### 4.1 Wykorzystane języki, technologie

- Baza danych - SQLite
- REST API - Python, FastAPI
- Panel administracyjny - JavaScript, Svelte
- Broker MQTT - Python
- Terminal - Python

### 4.2 Najważniejsze funkcje systemu

#### 4.2.1 Ustalanie trasy przez kierowcę

Kod 1: Terminal, Lokalizacja: client/client.py

```
1 def begin_route() -> None:
2     global stop_rfid_polling
3     global current_stop_index
4     global routes
5     global route_index
6     current_stop_index = 0
7     GPIO.add_event_detect(buttonGreen, GPIO.FALLING, callback=on_green_button_while_on_route,
8                             bounce_time=500)
9     GPIO.add_event_detect(buttonRed, GPIO.FALLING, callback=on_red_button_while_on_route, bounce_time
10                             =500)
11     client.publish("buses/driver", f"choose_course?bus={bus_id}&course={routes[route_index].name}")
12     draw_stops_screen(routes[route_index], current_stop_index)
13     stop_rfid_polling = False
14     rfid_thread = threading.Thread(target=listen_rfid)
15     rfid_thread.daemon = True
16     rfid_thread.start()
```

Po wybraniu trasy przez kierowcę terminal komunikuje się z brokerem MQTT poprzez temat `buses/driver` i wysyła mu odpowiednie dane, następnie też aktywowana jest możliwość skanowania karty pracownika zaimplementowana w wątku `rfid_thread`. Dodatkowo funkcjonalność przycisków: zielonego i czerwonego zostaje zmieniona, tak aby była odpowiedzialna już za zmianę przystanków, a wyświetlacz pokazuje przystanki.

Kod 2: Broker MQTT, Lokalizacja: backend/mqtt\_server.py

```
1 def choose_course(bus_id: int, course_name: str):
2     course_id = db_management.select('Courses', ['CourseID'], [('CourseName', course_name)])[0][0]
3     db_management.update('Buses', ('CourseID', course_id), ('BusID', bus_id))
4     db_management.update('Buses', ('StopNumber', 1), ('BusID', bus_id))
```

Broker MQTT po otrzymaniu danych od terminala, aktualizuje bazę danych.

#### 4.2.2 Zmiana przystanku przez kierowcę

Kod 3: Terminal, Lokalizacja: client/client.py

```
1 def on_green_button_while_on_route(_) -> None:
2     global current_stop_index
3     global routes
4     global route_index
5     global stop_rfid_polling
6
7     route = routes[route_index]
8     client.publish("buses/driver", f"next_stop?bus={bus_id}")
9     if current_stop_index < len(route.stops) - 1:
10         current_stop_index += 1
11         draw_stops_screen(route, current_stop_index)
12     else:
13         stop_rfid_polling = True
14         GPIO.remove_event_detect(buttonGreen)
15         GPIO.remove_event_detect(buttonRed)
16         route_index = 0
17         current_stop_index = 0
18         select_route()
```

Przy zmianie przystanku przez kierowcę terminal wysyła dane do brokera MQTT, następnie informacja o aktualnym przystanku na wyświetlaczu zostaje zmieniona. Jeżeli następnego przystanku nie ma, to wtedy trasa jest automatycznie zakończona.

Kod 4: Broker MQTT, Lokalizacja: backend/mqtt\_server.py

```
1 def next_stop(bus_id: int):
2     bus_data = db_management.select('Buses', ['BusID', 'CourseID', 'StopNumber'],
3                                     [('BusID', bus_id)])
4     if not bus_data:
5         return NO_SUCH_BUS
6     bus_id, course_id, stop_number = bus_data[0]
7     try:
8         stops = [tup[0] for tup in db_management.select('Assignments', ['StopID',
9                                                         [('CourseID', course_id)])]]
10    except OperationalError:
11        return BUS_NOT_ON_ROUTE
12    new_stop_number = stop_number + 1
13    if new_stop_number == 0 or new_stop_number > len(stops):
14        for attribute_name in ['CourseID', 'StopNumber']:
15            db_management.update('Buses', (attribute_name, 'null'), ('BusID', bus_id))
16        return ROUTE_ENDED
17    else:
18        db_management.update('Buses', ('stopNumber', new_stop_number), ('BusID', bus_id))
19        new_stop_id = db_management.select('Assignments', ['StopID',
20                                                            [('CourseID', course_id), ('stopNumber', new_stop_number)])
21                                          [0][0])
22        new_stop_name = db_management.select('Stops', ['StopName'], [('StopID', new_stop_id)])[0][0]
23        add_stop_to_workers(bus_id)
24        return f'{new_stop_number}-{new_stop_name}'
```

Broker MQTT po otrzymaniu danych od terminala aktualizuje przystanek w bazie danych, jeżeli podany pojazd jest na trasie, następnie aktualizowana jest liczba przejechanych przystanków dla każdego pracownika który znajduje się na trasie pojazdu.

#### 4.2.3 Płacenie za przejazd przez pracownika

Kod 5: Terminal, Lokalizacja: client/client.py

```
1 def on_card_scanned(uid: list[int]) -> None:
2     global last_card_scan_value_time
3     global current_stop_index
4     global routes
5     global route_index
6     uid_int = int(''.join(list(map(lambda e: str(e), uid))))
7     print(f'scanned {uid_int}')
8     (last_value, last_time) = last_card_scan_value_time
9     if last_time is not None and last_time + datetime.timedelta(seconds=7) > datetime.datetime.now():
10        return
11    last_card_scan_value_time = (uid_int, datetime.datetime.now())
12    client.publish("buses/worker", f"use_card?card={uid_int}&bus={bus_id}")
```

Po przyłożeniu karty przez pracownika, terminal wysyła dane do brokera MQTT.

---

```

1 def card_used(card_id: str, bus_id: int):
2     riding_workers = [tup[0] for tup in db_management.select_all('CurrentRides', ['WorkerCardID'])]
3     if int(card_id) in riding_workers:
4         return worker_gets_out(card_id)
5     else:
6         return worker_gets_in(card_id, bus_id)
7
8
9 def worker_gets_in(card_id: str, bus_id: int):
10    if int(card_id) not in [tup[0] for tup in db_management.select_all('Workers', ['WorkerCardID'])]:
11        return 'Invalid card.'
12    try:
13        max_ride_id = max([tup[0] for tup in db_management.select_all('CurrentRides', ['RideID'])])
14    except ValueError:
15        max_ride_id = 0
16    db_management.insert('CurrentRides', (max_ride_id + 1, card_id, bus_id, 0))
17    return f'Card validated succesfully.'
18
19
20 def worker_gets_out(card_id: str):
21    stops_traveled = db_management.select('CurrentRides', ['StopsTraveled'], [( 'WorkerCardID', card_id
22    )])[0][0]
23    db_management.delete('CurrentRides', ('WorkerCardID', card_id))
24    current_balance = db_management.select('Workers', ['WorkerBalance'], [( 'WorkerCardID', card_id )])
25    [0][0]
26    db_management.update('Workers', ('WorkerBalance', current_balance - stops_traveled), ('
27    WorkerCardID', card_id))
28    return f'Balance after ride: {current_balance - stops_traveled}.'

```

---

Broker MQTT po otrzymaniu danych od terminala, sprawdza czy pracownik zaczyna, lub kończy przejazd:

- jeżeli zaczyna przejazd, to po sprawdzeniu czy taki pracownik istnieje w bazie, zostaje dodany rekord o aktualnym przejeździe,
- jeżeli kończy przejazd, to z jego konta zostaje potrącona kwota zależna od przejechanych przystanków, oraz jego przejazd zostaje usunięty z bazy.

#### 4.2.4 Dodanie pieniędzy do konta pracownika przez administratora

---

Kod 7: Panel administracyjny, Lokalizacja: frontend/app/src/routes/employees/+page.svelte

---

```

1 async function editBalance(){
2     if(bonusData.bonus == null) bonusData.bonus = 0.0
3     try{
4         const url = '/addbalance/${bonusData.worker_id}?value=${bonusData.bonus}'
5         const response = await fetch(baseUrl + url, {
6             method: "GET",
7         })
8         if(response.ok){
9             console.log("Form data sent successfully");
10            employees = await getEmployeesData();
11        }else{
12            console.error("error sending form data", response.status);
13        }
14    }catch(e){
15        console.log(e);
16    }
17 }

```

---

Panel administracyjny przekazuje odpowiednie dane do REST API, następnie pobiera zaktualizowane dane o pracownikach.

---

Kod 8: REST API, Lokalizacja: backend/server.py

---

```

1 @app.get("/addbalance/{worker_id}")
2 async def add_balance_endpoint(worker_id: int, value: float):
3     current_balance = db_management.select('Workers', ['WorkerBalance'], [( 'WorkerID', worker_id )])
4     [0][0]
5     db_management.update('Workers', ('WorkerBalance', current_balance + value), ('WorkerID',
6     worker_id))
7     return {'success': f'Balance of worker with ID {worker_id} has changed from {current_balance} to
8     ,
9     f'{current_balance + value}'}

```

---

REST API po otrzymaniu danych, aktualizuje bazę danych.

## 4.3 Implementacja MQTT

### 4.3.1 Broker MQTT

Kod 9: Broker MQTT, Lokalizacja: backend/mqtt\_server.py

```
1 import paho.mqtt.client as mqtt
2 (...)
3
4 broker_ip = "0.0.0.0"
5
6 CLIENT = mqtt.Client()
7
8 (...)
9
10
11 def process_message(client, userdata, message):
12     global CLIENT
13     message_decoded = (str(message.payload.decode("utf-8"))).split(".")[0]
14     print(message_decoded)
15     message_dict = query_string_to_dict(message_decoded)
16     if 'next_stop' in message_dict:
17         next_stop(message_dict['next_stop']['bus'])
18     elif 'choose_course' in message_dict:
19         choose_course(message_dict['choose_course']['bus'], message_dict['choose_course']['course'])
20     elif 'use_card' in message_dict:
21         result_code = card_used(message_dict['use_card']['card'], message_dict['use_card']['bus'])
22         CLIENT.publish('response/success', result_code)
23
24
25 def connect_to_broker():
26     CLIENT.connect(broker_ip)
27     CLIENT.on_message = process_message
28     CLIENT.loop_start()
29     CLIENT.subscribe("buses/#")
30
31
32 def disconnect_from_broker():
33     CLIENT.loop_stop()
34     CLIENT.disconnect()
35
36
37 def run_mqtt_server():
38     connect_to_broker()
39     input()
40
41
42 (...)
43
44
45 if __name__ == "__main__":
46     run_mqtt_server()
```

Broker MQTT subskrybuje każdy temat, który zaczyna się od `buses/`, każda wiadomość która jest odbierana przez broker jest w formacie `query string`, natomiast odpowiedzi są wysyłane tematem `response/success`.

### 4.3.2 Terminal

Kod 10: Terminal, Lokalizacja: client/client.py

```
1  #!/usr/bin/env python3
2
3  import paho.mqtt.client as mqtt
4  (...)
5
6  server_ip = "10.108.33.123"
7  bus_id = 0
8
9  client = mqtt.Client()
10 (...)
11
12
13 def begin_route() -> None:
14     (...)
15     client.publish("buses/driver", f"choose_course?bus={bus_id}&course={routes[route_index].name}")
16     (...)
17
18
19 def on_mqtt_message(client, userdata, message):
20     message_decoded = str(message.payload.decode('utf-8'))
21     print(f'message received: {message_decoded}')
22     draw_message(message_decoded)
23     timer = threading.Timer(7, draw_stops_screen, args=(routes[route_index], current_stop_index))
24     timer.start()
25
26
27 (...)
28
29
30 if __name__ == "__main__":
31     try:
32         disp.Init()
33         disp.clear()
34
35         client.connect(server_ip)
36         client.on_message = on_mqtt_message
37         client.loop_start()
38         client.subscribe("response/#")
39
40         (...)
41
42         while True:
43             _ = input()
44     (...)

```

Terminal subskrybuje wszystkie tematy zaczynające się od `response/`, oraz publikuje dane na różnych tematach zaczynających się od `buses/`. Dane które są odebrane z `response/` są przekazywane do wyświetlacza, który je pokazuje.

## 4.4 Implementacja terminala

### 4.4.1 Pobieranie danych o kursach i ich przystankach

Kod 11: Terminal, Lokalizacja: client/client.py

```
1 def fetch_routes() -> list[Route]:
2     with urllib.request.urlopen(f'http://{server_ip}:55555/courses') as url:
3         data = json.load(url)
4         for route_name in data:
5             stops = map(lambda stop_name: Stop(stop_name), data[route_name])
6             routes.append(Route(route_name, list(stops)))
7     return routes

```

Do pobierania danych o kursach i przystankach terminal wykorzystuje REST API.

```
1 @app.get("/courses")
2 async def courses_endpoint():
3     result = {}
4     courses = db_management.select_all('Courses', ['CourseID', 'courseName'])
5     for course_id, course_name in courses:
6         stops = db_management.select('Assignments', ['StopID', 'StopNumber'], [('CourseID',
7             course_id)])
8         result[course_name] = ['' for _ in range(len(stops))]
9         for stop_id, stop_number in stops:
10             stop_name = db_management.select('Stops', ['StopName'], [('StopID', stop_id)])[0][0]
11             result[course_name][stop_number-1] = stop_name
12     return result
```

---

REST API pobiera z bazy danych kursy, następnie pobiera ich przystanki i je przypisuje do zwracanych danych.

## 5 Opis działania i prezentacja interfejsu

### 5.1 Sposób instalacji i uruchomienia aplikacji

#### 5.1.1 Zawartość plików

- backend/
  - baza danych - pliki: `db_management.py`, `init.sql`, `insert_example.sql`
  - REST API - plik: `server.py`
  - broker MQTT - pliki: `mqtt_server.py`, `mqtt_client_TEST.py`
- frontend/app - panel administracyjny
- client/ - terminal

#### 5.1.2 Instalacja

##### 1. Baza danych

- przejdź do folderu `backend`
- uruchom plik `db_management.py`
- w folderze powinien utworzyć się plik `alpha.db`, zawierający przykładowe dane

##### 2. REST API i broker MQTT

- zainstaluj wymagane pakiety za pomocą komendy `pip install -r requirements.txt`
- upewnij się, że masz zainstalowane i uruchomione **Mosquitto**  
UWAGA: w pliku konfiguracyjnym Mosquitto powinny się znaleźć następujące linijki, aby urządzenia mogły się połączyć:

Kod 13: Dodatek do pliku `mosquitto.conf`

```
1 allow_anonymous true
2 listener 1883 0.0.0.0
```

- uruchom plik `server.py`
- działanie REST API można testować pod adresem: <http://localhost:5555/docs>

##### 3. Panel administracyjny

- upewnij się, że masz zainstalowany **Node.js**
- przejdź do folderu `frontend/app`
- zainstaluj pakiety za pomocą komendy `npm install`
- wprowadź komendę `npm run dev`
- działanie można testować pod adresem: <http://localhost:5173/>

##### 4. Terminal (wersja Raspberry Pi)


- przejdź do folderu `client`
- otwórz plik `client.py` i zmodyfikuj zmienną `server_ip` tak, aby odpowiadała adresowi IP brokera MQTT
- uruchom plik `client.py`

##### 5. Terminal (wersja testowa)

- przejdź do folderu `backend`
- otwórz plik `mqtt_client_TEST.py` i zmodyfikuj zmienną `BROKER_IP` tak, aby odpowiadała adresowi IP brokera MQTT
- uruchom plik `mqtt_client_TEST.py`

## 5.2 Przedstawienie działania aplikacji

### 5.2.1 Panel administracyjny

 Employees Map Routes

**Employees List**


Worker ID	Card ID	First Name	Last Name	Balance
1	25111252110228	John	Smith	500
2	187121523413	Emily	Johnson	700.5

**Add New Employee**

Card ID


First Name

Last Name



**Increase Worker's Balance**

Worker ID



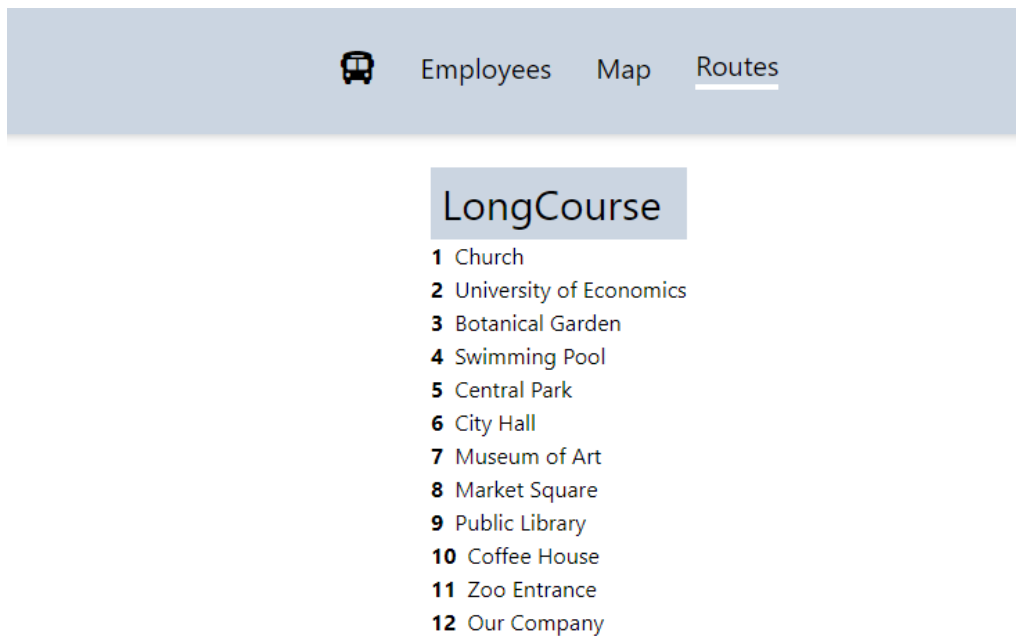
Rysunek 9: Widok pracowników w panelu administracyjnym

Przykładowo dodamy nowego pracownika i dodamy mu pieniądze do stanu konta, wypełniając odpowiednie pola i klikając znak +.

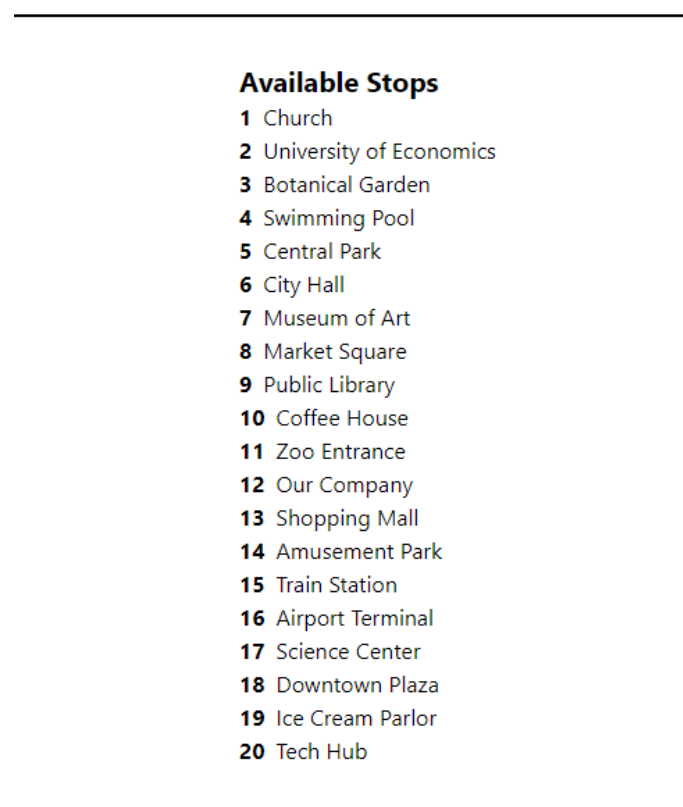
Employees List				
Worker ID	Card ID	First Name	Last Name	Balance
1	25111252110228	John	Smith	500
2	187121523413	Emily	Johnson	700.5
3	123123123	Mark	Davis	200

Rysunek 10: Dodany nowy pracownik





Rysunek 11: Widok przejazdów w panelu administracyjnym, kursy



Rysunek 12: Widok przejazdów w panelu administracyjnym, przystanki

Add New Stop

New Stop Name

+

Add New Course

New Course Name

Stops Ids: 1,2,3...

+

Rysunek 13: Widok przejazdów w panelu administracyjnym, dodawanie nowych danych

Przykładowo dodamy nowy przystanek o nazwie **Test Stop**, oraz kurs o nazwie **TestCourse**, który będzie go zawierał.

TestCourse

1 Test Stop

2 Amusement Park

3 Train Station

Available Stops

1 Church

2 University of Economics

3 Botanical Garden

4 Swimming Pool

5 Central Park

6 City Hall

7 Museum of Art

8 Market Square

9 Public Library

10 Coffee House

11 Zoo Entrance

12 Our Company

13 Shopping Mall

14 Amusement Park

15 Train Station

16 Airport Terminal

17 Science Center

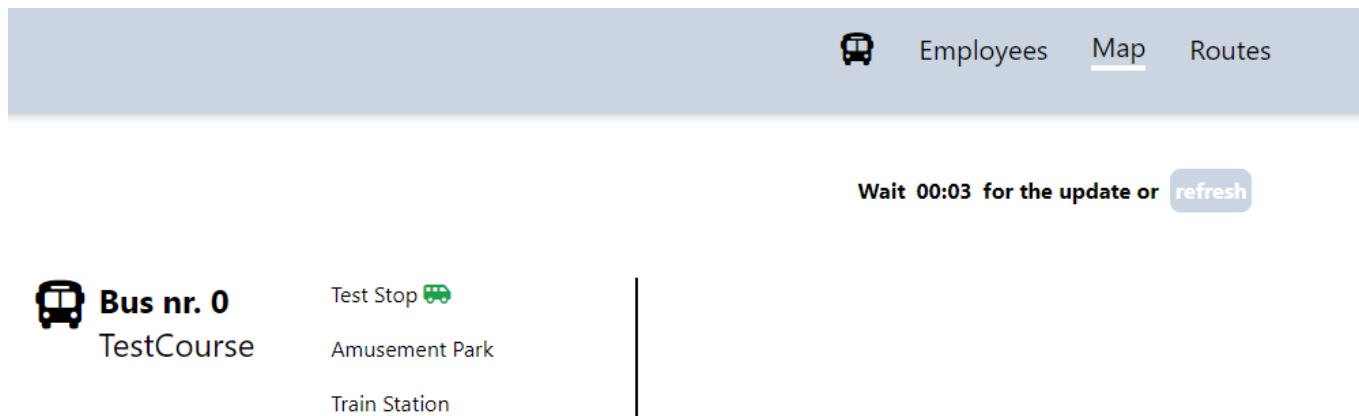
18 Downtown Plaza

19 Ice Cream Parlor

20 Tech Hub

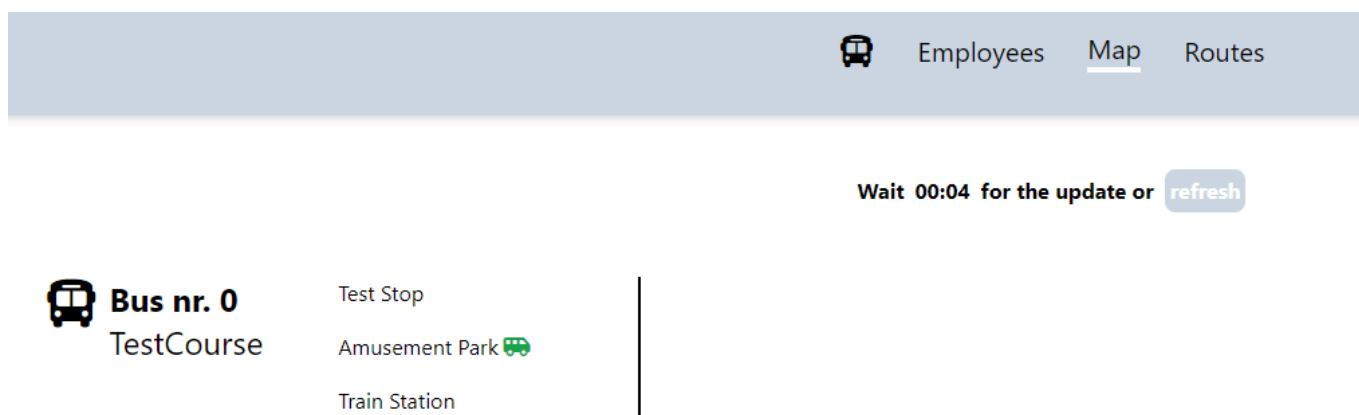
21 Test Stop

Rysunek 14: Dodany nowy kurs zawierający nowy przystanek



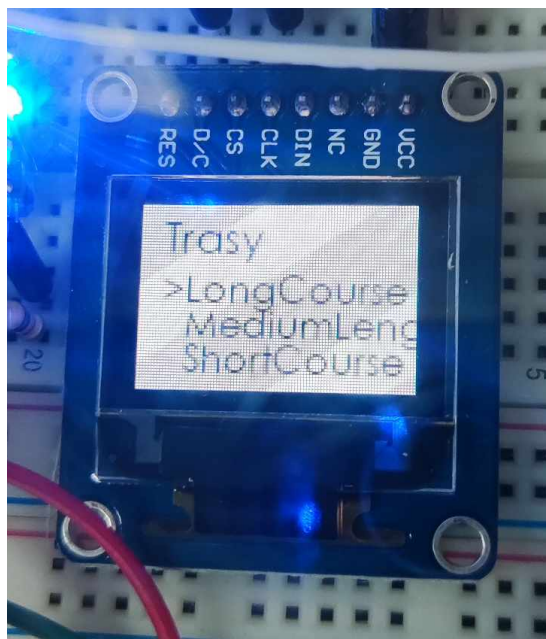
Rysunek 15: Mapa pojazdów odbywających kursy

W terminalu zmieniamy przystanek na którym pojazd się znajduje.

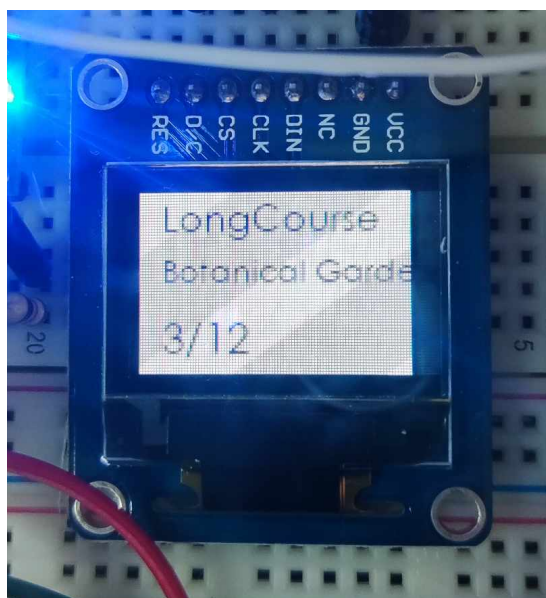


Rysunek 16: Zmiana przystanku widoczna na mapie

### 5.2.2 Terminal (wersja Raspberry Pi)

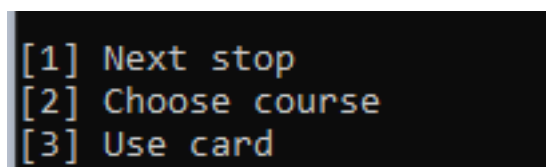


Rysunek 17: Widok wyboru trasy



Rysunek 18: Widok aktualnego przystanku na trasie

### 5.2.3 Terminal (wersja testowa)



Rysunek 19: Widok menu

```
[1] Next stop
[2] Choose course
[3] Use card
2

1. LongCourse
2. MediumLengthCourse
3. ShortCourse
4. TestCourse
4

[1] Next stop
[2] Choose course
[3] Use card
3

[1] Next stop
[2] Choose course
[3] Use card

Card validated succesfully

1

[1] Next stop
[2] Choose course
[3] Use card
1

[1] Next stop
[2] Choose course
[3] Use card
3

[1] Next stop
[2] Choose course
[3] Use card

Balance after ride: 198
```

Rysunek 20: Przykład wyboru kursu, zmiany przystanków i płacenia za przejazd przez pracownika

## 6 Wkład pracy Autorów

### 6.1 Pierwszy tydzień (8.01 - 14.01)

- Kacper Gaudyn
  - struktura bazy danych, tabele: `Courses`, `Stops`, `Assignments`, `Workers`, `CurrentRides`
  - tworzenie pliku bazy danych, podstawowe operacje na danych
- Jakub Krapiec
  - terminal - obsługa przycisków, struktura przystanków i tras
  - dodanie możliwości wyboru trasy i przystanków za pomocą enkodera, obsługa przycisków rozbudowana o nowe funkcjonalności, obsługa RFID
- Michał Pesta
  - utworzenie szkieletu panelu administracyjnego, dodanie paska nawigacji
  - dodanie strony z informacjami o pracownikach
- Michał Trojanowski
  - utworzenie szkieletu raportu, dodanie do raportu wymagań projektowych i opisu architektury systemu
  - terminal - wyświetlanie informacji na wyświetlaczu

### 6.2 Drugi tydzień (15.01 - 21.01)

- Kacper Gaudyn
  - dodanie szkieletu REST API
  - dodanie do bazy danych tabeli `Buses`, pobieranie pracowników i pojazdów w REST API, drobne poprawki
  - dodanie brokera MQTT, dokumentacji do brokera MQTT oraz terminala w wersji testowej
- Jakub Krapiec
  - terminal - poprawki w obsłudze przycisków i enkodera
  - poprawki w strukturze bazy danych, dodano pobieranie kursów w REST API
  - terminal - zmiany wizualne w wyświetlaniu informacji
- Michał Pesta
  - dodanie przykładowych danych do bazy danych, dodana funkcjonalność zmiany przystanków w REST API
  - pobieranie danych z REST API w panelu administratora, dodanie podglądu pojazdów w ruchu
  - dodanie dokumentacji do REST API, drobne poprawki w bazie danych i REST API, dodanie funkcjonalności związanych z pracownikami w REST API
- Michał Trojanowski
  - aktualizacja architektury systemu w raporcie, dodanie do raportu częściowego opisu implementacji, dodanie do raportu wstępu
  - terminal - dodana komunikacja z brokerem MQTT
  - terminal - wyświetlanie informacji od brokera

### 6.3 Trzeci tydzień (22.01 - 26.01)

- Kacper Gaudyn
  - poprawki w testowych danych bazy danych, poprawki w obsłudze identyfikatora karty pracownika w brokerze MQTT
  - drobne poprawki w REST API i dokumentacji
  - dokończenie opisu architektury w raporcie
- Jakub Krąpiec
  - drobne poprawki w raporcie, zmiany w komunikacji z brokerem MQTT w terminalu, pobieranie danych o trasach i przystankach z REST API w terminalu
  - drobne poprawki w terminalu
  - dodanie opisu implementacji w raporcie
- Michał Pesta
  - obsługa przystanków w panelu administracyjnym; komunikacja z REST API, poprawki w ustawieniach CORS
  - drobne poprawki w REST API oraz panelu administracyjnym
  - dodanie opisu działania w raporcie
- Michał Trojanowski
  - obsługa kursów w panelu administracyjnym
  - drobne poprawki w REST API
  - dodanie do raportu: wkładu pracy i podsumowania, zrobienie prezentacji

## 7 Podsumowanie

### 7.1 Stopień zgodności z wymaganiami

#### 7.1.1 Wymagania funkcjonalne

1. **Zrealizowane** - odpowiedni przycisk w terminalu po wybraniu trasy
2. **Zrealizowane** - wybór za pomocą enkodera
3. **Zrealizowane** - realizowane poprzez przyciski
4. **Zrealizowane**
5. **Zrealizowane** - należy przyłożyć kartę przy wsiadaniu i wysiadaniu
6. **Zrealizowane** - koszt wynosi 1
7. **Zrealizowane** - panel administratora
8. **Zrealizowane** - panel administratora
9. **Zrealizowane** - panel administratora

#### 7.1.2 Wymagania niefunkcjonalne

1. **Zrealizowane**
2. **Zrealizowane**

### 7.2 Napotkane problemy w implementacji

#### 7.2.1 Uruchomienie systemu na komputerze laboratoryjnym

Na komputerach laboratoryjnych nie ma zainstalowanego `Node.js`, a użytkownik studenta nie ma uprawnień administratora, więc też nie było możliwości instalacji. Problem dotyczył panelu administracyjnego i został rozwiązany poprzez zmianę środowiska na Raspberry Pi, tam już instalacja była możliwa.