Raport z realizacji projektu programistycznego

System biblioteczny

Autorzy (grupa C):

Dawid Błaszczyk - 280518

Błazej Kowal - 280655

Alina Lenart - 280588

Bartosz Wacławiak - 280462

Prowadzący laboratorium:

dr inż. Krzysztof Chudzik

Data ukończenia pracy:
25.01.2025

Spis treści

1 Wymagania projektowe

1.1 Wymagania funkcjonalne

System biblioteczny IoT musi spełniać następujące wymagania funkcjonalne:

1.1.1 Obsługa kart RFID

W1.1.1.1 System musi umożliwiać odczyt kart RFID za pomocą czytnika MFRC522.

W1.1.1.2 System musi automatycznie wykrywać przyłożenie i zabranie karty RFID.

W1.1.1.3 Odczytany identyfikator karty (UID) musi być konwertowany do formatu szesnastkowego i przesyłany do serwera centralnego.

W1.1.1.4 System musi rozróżniać między kartami klientów biblioteki i kartami przypisanymi do książek.

1.1.2 Zarządzanie bazą danych

W1.1.2.1 System musi przechowywać informacje o klientach (imię, nazwisko, powiązana karta RFID).

W1.1.2.2 System musi przechowywać informacje o książkach (tytuł, autor, powiązana karta RFID).

W1.1.2.3 System musi rejestrować wypożyczenia i zwroty książek z datami operacji.

W1.1.2.4 System musi umożliwiać automatyczne tworzenie nowych rekordów kart przy pierwszym skanowaniu nieznanego UID.

1.1.3 Proces wypożyczania i zwracania

W1.1.3.1 System musi umożliwiać wypożyczenie książki poprzez zeskanowanie karty klienta, a następnie karty książki.

W1.1.3.2 System musi umożliwiać zwrot książki przy pomocy zeskanowania karty klienta, a następnie karty książki.

W1.1.3.3 System musi wyświetlać informacje o aktywnych wypożyczeniach klienta po zeskanowaniu jego karty.

W1.1.3.4 System musi weryfikować poprawność operacji (np. czy książka jest dostępna, czy klient już ją wypożyczył).

1.1.4 Interfejs użytkownika

W1.1.4.1 System musi posiadać webowy interfejs graficzny dostępny przez przeglądarkę.

W1.1.4.2 Interfejs musi umożliwiać przeglądanie listy wszystkich klientów, książek i wypożyczeń.

W1.1.4.3 Interfejs musi umożliwiać ręczne dodawanie, edycję i usuwanie klientów oraz książek.

W1.1.4.4 Interfejs musi wyświetlać informacje o zeskanowanej karcie w czasie rzeczywistym.

W1.1.4.5 Interfejs musi umożliwiać przeprowadzenie pełnego procesu wypożyczania/zwrotu z graficznym przewodnikiem.

1.1.5 Komunikacja MQTT

W1.1.5.1 Terminale RFID (Raspberry Pi) muszą komunikować się z serwerem centralnym przez protokół MQTT.

W1.1.5.2 System musi publikować zdarzenia skanowania kart na topic `raspberry/rfid/scan`.

W1.1.5.3 Serwer musi odpowiadać z danymi o kliencie lub książce na topic `raspberry/rfid/response`.

W1.1.5.4 System musi obsługiwać sterowanie diodami LED przez MQTT (topic `raspberry/led`).

1.1.6 Informacje zwrotne dla użytkownika terminala

W1.1.6.1 System musi sygnalizować gotowość do skanowania zieloną diodą LED.

W1.1.6.2 System musi sygnalizować przetwarzanie karty czerwoną diodą LED.

W1.1.6.3 System musi emitować dźwięk buzzera po pomyślnym odczytaniu karty.

W1.1.6.4 System musi wyświetlać informacje o stanie operacji na wyświetlaczu OLED (oczekiwanie, wykryto kartę, przetwarzanie, dane klienta/książki).

1.2 Wymagania niefunkcjonalne

System musi spełniać następujące wymagania niefunkcjonalne:

1.2.1 Wydajność

W1.2.1.1 Czas odpowiedzi serwera na żądanie API nie powinien przekraczać 500ms w warunkach normalnego obciążenia.

W1.2.1.2 System musi przetwarzać zdarzenia RFID w czasie rzeczywistym (opóźnienie poniżej 1 sekundy od momentu skanowania do wyświetlenia informacji).

W1.2.1.3 Aplikacja webowa musi ładować się w czasie nie dłuższym niż 3 sekundy przy standardowym połączeniu internetowym.

1.2.2 Niezawodność i stabilność

W1.2.2.1 System musi być odporny na tymczasową utratę połączenia z brokerem MQTT i automatycznie wznawiać komunikację.

W1.2.2.2 W przypadku błędu odczytu karty RFID, system musi wyświetlić komunikat o błędzie i umożliwić ponowną próbę.

W1.2.2.3 Baza danych musi zapewniać integralność danych (brak duplikatów wypożyczeń, prawidłowe daty operacji).

1.2.3 Skalowalność

W1.2.3.1 Architektura systemu musi umożliwiać łatwe dodanie kolejnych terminali RFID bez modyfikacji kodu serwera.

W1.2.3.2 Baza danych musi być zaprojektowana w sposób umożliwiający przechowywanie tysięcy rekordów klientów i książek.

1.2.4 Bezpieczeństwo

W1.2.4.1 Dane w bazie danych muszą być zabezpieczone przed nieautoryzowanym dostępem poprzez odpowiednią konfigurację uprawnień.

W1.2.4.2 System musi walidować wszystkie dane wejściowe z API, aby zapobiec nieprawidłowym operacjom.

1.2.5 Środowisko uruchomieniowe i przenośność

W1.2.5.1 Terminal RFID musi działać na platformie Raspberry Pi 4B z systemem operacyjnym Raspberry Pi OS.

W1.2.5.2 Serwer backendowy musi być uruchamialny na systemach Linux, macOS i Windows.

W1.2.5.3 Aplikacja webowa musi być kompatybilna z nowoczesnymi przeglądarkami (Chrome, Firefox, Safari, Edge).

W1.2.5.4 System musi umożliwiać uruchomienie brokera MQTT w sieci lokalnej (np. na Raspberry Pi lub innym serwerze).

W1.2.5.5 Baza danych SQLite musi być przenośna i nie wymagać dodatkowej konfiguracji serwera baz danych.

1.2.6 Technologie i standardy

W1.2.6.1 Backend: Node.js z frameworkiem NestJS, TypeScript, TypeORM.

W1.2.6.2 Frontend: React 19 z TypeScript, React Router, Tailwind CSS, Vite.

W1.2.6.3 Komunikacja IoT: Protokół MQTT z brokerem uruchomionym w sieci lokalnej.

W1.2.6.4 Baza danych: SQLite 3.

W1.2.6.5 Raspberry Pi: Python 3 z bibliotekami paho-mqtt (klient MQTT) i mfrc522 (obsługa czytnika RFID).

W1.2.6.6 Komunikacja w czasie rzeczywistym: WebSockets (Socket.IO) dla aktualizacji interfejsu użytkownika.

1.2.7 Użyteczność

W1.2.7.1 Interfejs graficzny musi być intuicyjny i nie wymagać specjalistycznego przeszkolenia.

W1.2.7.2 System musi dostarczać jasne komunikaty o stanie operacji (powodzenie, błąd, oczekiwanie).

W1.2.7.3 Wyświetlacz OLED na terminalu RFID musi prezentować czytelne informacje o aktualnym stanie systemu.

W1.2.7.4 Kolorowe diody LED muszą jednoznacznie sygnalizować stan systemu (zielony = gotowy, czerwony = przetwarzanie/błąd).

1.2.8 Dokumentacja i kod

W1.2.8.1 Kod źródłowy musi być czytelny i zgodny ze standardami danego języka programowania.

W1.2.8.2 Projekt musi zawierać pliki konfiguracyjne umożliwiające łatwe uruchomienie systemu.

2 Opis architektury systemu

System został zaprojektowany w architekturze wielowarstwowej. Składa się z następujących elementów:

- warstwy klienckiej (aplikacja webowa React),
- warstwy serwerowej (backend NestJS),
- warstwy komunikacyjnej (MQTT, WebSockets),
- warstwy danych (baza danych SQLite),
- warstwy urządzeń IoT (Raspberry Pi z czytnikiem RFID).

2.1 Schemat architektury aplikacji

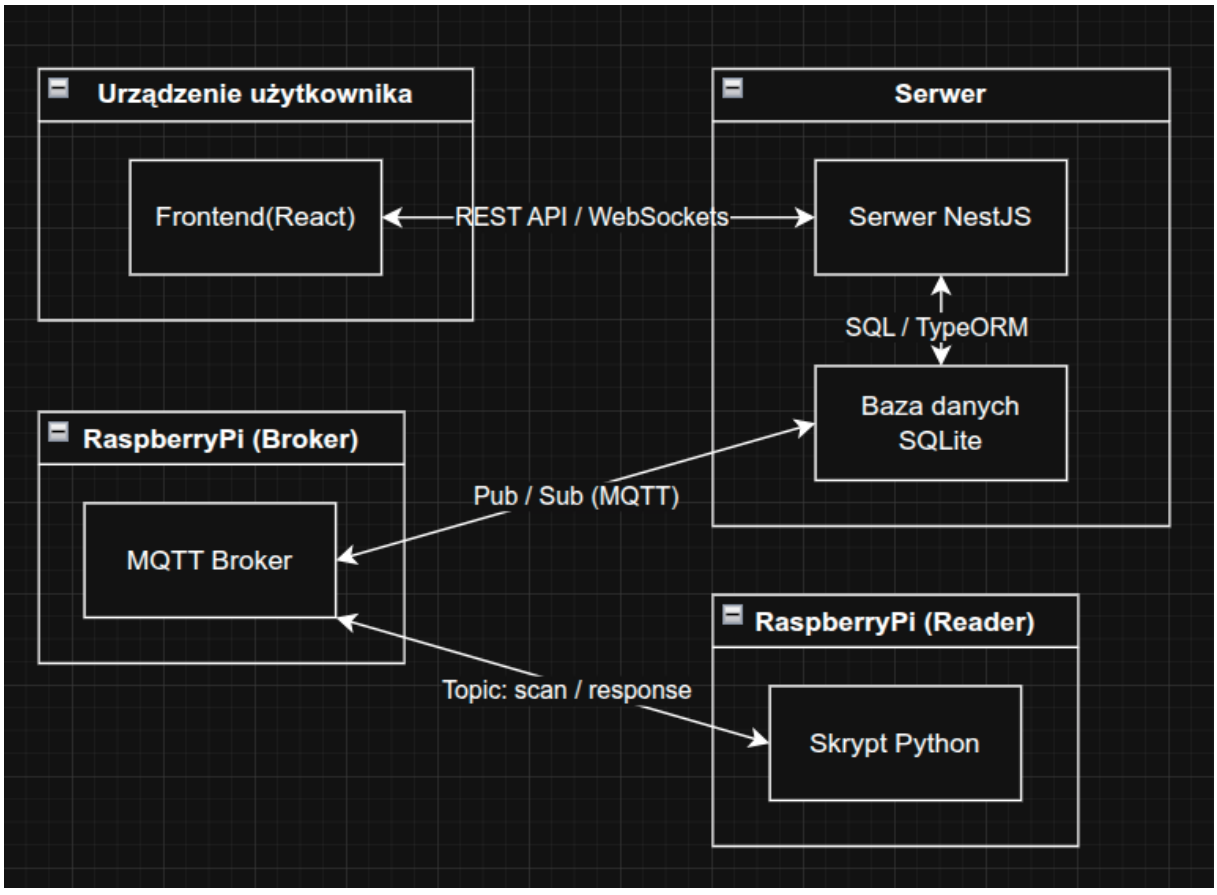
Poniżej przedstawiono schemat architektury systemu z uwzględnieniem architektury sieciowej.

Architektura systemu opiera się na następujących komponentach:

- **Frontend (React)** - aplikacja webowa działająca w przeglądarce, komunikująca się z backendem przez REST API i WebSockets (Socket.IO).
- **Backend (NestJS)** - serwer aplikacyjny obsługujący logikę biznesową, komunikujący się z bazą danych SQLite oraz brokerem MQTT.
- **Broker MQTT** - broker komunikacji MQTT uruchomiony w sieci lokalnej (np. na Raspberry Pi lub osobnym serwerze), pośredniczący w komunikacji między terminalem RFID a backendem.
- **Raspberry Pi** - terminal RFID wyposażony w czytnik MFRC522, diody LED, buzzer i wyświetlacz OLED, komunikujący się z backendem przez protokół MQTT.
- **Baza danych SQLite** - lokalna baza danych przechowująca informacje o klientach, książkach, kartach RFID i wypożyczeniach.

Komunikacja w systemie odbywa się następująco:

- Raspberry Pi publikuje zdarzenia skanowania kart na topic `raspberry/rfid/scan`.
- Backend subskrybuje ten topic, przetwarza dane i odpowiada na topic `raspberry/rfid/response`.
- Backend steruje diodami LED na Raspberry Pi przez topic `raspberry/led`.
- Frontend otrzymuje aktualizacje w czasie rzeczywistym przez WebSockets (Socket.IO).



Rysunek 1: Schemat architektury systemu

3 Opis implementacji i zastosowanych rozwiązań

3.1 Kluczowe elementy implementacji

W tej sekcji opisano najważniejsze fragmenty kodu odpowiedzialne za kluczowe funkcje systemu.

3.1.1 Obsługa odczytu kart RFID

Czytnik RFID na Raspberry Pi wykorzystuje bibliotekę `mfrc522` do komunikacji z modułem MFRC522. Poniżej przedstawiono kluczowy fragment kodu odpowiedzialny za odczyt karty:

Listing 1: Odczyt karty RFID - rfid_reader.py

```
class RFIDReader:
    """rfid reader handler"""

    def __init__(self):
        """initialize rfid reader"""
        self.reader = MFRC522()
        print("Czytnik RFID zainicjalizowany")

    def wait_for_card(self):
        """
        wait for card and return uid

        returns:
            dict: uid_bytes, uid_hex, uid_int, timestamp
            none: if card read failed
        """
        print("\nOczekiwanie na przylozenie karty RFID...")

        card_logged = False
        last_seen_time = 0
        card_data = None

        while True:
            now = time.time()
            (status, tag_type) = self.reader.MFRC522_Request(self.reader.

            if status == self.reader.MI_OK:
                (status_uid, uid) = self.reader.MFRC522_Anticoll()

                if status_uid == self.reader.MI_OK:
                    # update last seen time
                    last_seen_time = now
```

```
            # card detected after absence
            if not card_logged:
                card_logged = True

                # convert uid to different formats
                uid_int = 0
                for i in range(len(uid)):
                    uid_int += uid[i] << (i * 8)

                uid_hex = "".join([f"{x:02X}" for x in uid])
                timestamp = datetime.now().strftime("%Y-%m-%d %H:%

                card_data = {
                    'uid_bytes': list(uid),
                    'uid_hex': uid_hex,
                    'uid_int': uid_int,
                    'timestamp': timestamp
                }

                print("\nKarta wykryta!")
                print(f"   Czas:      {timestamp}")
                print(f"   UID (hex): {uid_hex}")
                print(f"   UID (int): {uid_int}")

                return card_data
```

Funkcja `wait_for_card()` w sposób ciągły monitoruje obecność karty RFID przy użyciu pętli blokującej. W momencie wykrycia tagu, czytnik pobiera jego unikalny identyfikator (UID), a następnie konwertuje go do formatu listy bajtów, ciągu szesnastkowego oraz liczby całkowitej.

3.1.2 Obsługa wypożyczeń

Serwis wypożyczeń w backendzie odpowiada za logikę wypożyczania i zwracania książek:

Listing 2: Serwis wypożyczeń - borrow.service.ts

```
@Injectable()
export class BorrowService {
  async create(bookCardId: string, clientCardId: string) {
```

```
    const book = await this.bookRepo.findOne({
      where: { cardId: bookCardId } });
    if (!book) throw new NotFoundException('Book not found');

    const client = await this.clientRepo.findOne({
      where: { cardId: clientCardId } });
    if (!client) throw new NotFoundException('Client not found');

    const borrows = await this.borrowRepo.find({
      where: { book: { cardId: bookCardId } },
      relations: ['book'] });
    const active = borrows.find(b => b.returnedAt == null);
    if (active) throw new BadRequestException(
      'Book is already borrowed');

    const now = new Date();
    const due = new Date();
    due.setDate(now.getDate() + 21);
    const borrow = this.borrowRepo.create({
      book, client, borrowedAt: now, dueDate: due });
    return this.borrowRepo.save(borrow);
  }

  async returnBook(borrowId: number) {
    const borrow = await this.findOne(borrowId);
    if (!borrow) throw new NotFoundException('Borrow not found');
    if (borrow.returnedAt) throw new BadRequestException(
      'Book already returned');
    borrow.returnedAt = new Date();
    return this.borrowRepo.save(borrow);
  }
}
```

Metoda `create()` weryfikuje dostępność książki przed utworzeniem wypożyczenia. Domyślny okres wypożyczenia wynosi 21 dni. Metoda `returnBook()` oznacza wypożyczenie jako zwrócone poprzez ustawienie pola `returnedAt`.

3.2 Implementacja komunikacji MQTT

Komunikacja MQTT jest kluczowym elementem systemu, umożliwiającym wymianę danych między Raspberry Pi a backendem. Poniżej przedstawiono implementację klienta MQTT po stronie Raspberry Pi:

Listing 3: Klient MQTT - mqtt_client.py

```
class MQTTClient:
    def __init__(self, on_led_change=None, on_response=None):
        self.client = mqtt.Client(client_id=MQTT_CLIENT_ID)
        self.client.on_connect = self._on_connect
        self.client.on_message = self._on_message
        self.connected = False
        self.on_led_change = on_led_change
        self.on_response = on_response

    def _on_connect(self, client, userdata, flags, rc):
        """callback on broker connection"""
        if rc == 0:
            self.connected = True
            print(f"Polaczono z MQTT brokerem: {MQTT_BROKER}:{MQTT_PORT}"
            # subscribe to backend response topics
            self.client.subscribe(MQTT_TOPIC_LED)
            self.client.subscribe(MQTT_TOPIC_RESPONSE)

    def _on_message(self, client, userdata, msg):
        """callback on mqtt message received"""
        topic = msg.topic
        payload = json.loads(msg.payload.decode())

        if topic == MQTT_TOPIC_LED:
            color = payload.get('color', 'unknown')
            if self.on_led_change:
                self.on_led_change(color)
        elif topic == MQTT_TOPIC_RESPONSE:
            if self.on_response:
                self.on_response(payload)

    def publish_scan(self, card_data):
        """publish scanned card info"""
```

```
        message = {
            'uid': card_data['uid_hex'],
            'uid_int': card_data['uid_int'],
            'timestamp': card_data['timestamp']
        }
        payload = json.dumps(message)
        result = self.client.publish(MQTT_TOPIC_SCAN, payload)
        return result.rc == mqtt.MQTT_ERR_SUCCESS
```

Po stronie backendu, serwis MQTT subskrybuje odpowiednie topiki i przetwarza przychodzące wiadomości:

Listing 4: Serwis MQTT - mqtt.service.ts

```
@Injectable()
export class MqttService implements OnModuleInit {
  private client: mqtt.MqttClient;
  private messageHandlers: MessageHandler[] = [];

  onModuleInit() {
    const brokerUrl = process.env.MQTT_BROKER_URL ||
      'mqtt://localhost:1883';
    this.client = mqtt.connect(brokerUrl);

    this.client.on('connect', () => {
      this.logger.log(`Connected to MQTT broker ${brokerUrl}`);
      this.client.subscribe('raspberry/rfid/scan');
      this.client.subscribe('raspberry/rfid/cancel');
    });

    this.client.on('message', (topic, payload) => {
      const message = payload.toString();
      this.messageHandlers.forEach(handler => {
        handler(topic, message);
      });
    });
  }

  publish(topic: string, payload: any) {
    const message = typeof payload === 'string' ?
      payload : JSON.stringify(payload);
```

```
    this.client.publish(topic, message);
  }
}
```

Struktura topiców MQTT:

- `raspberry/rfid/scan` - Raspberry Pi publikuje tutaj dane zeskanowanej karty.
- `raspberry/rfid/response` - Backend odpowiada z danymi o karcie, kliencie lub książce.
- `raspberry/led` - Backend steruje kolorami diod LED na Raspberry Pi.

3.3 Szyfrowanie i uwierzytelnianie

W obecnej wersji systemu komunikacja odbywa się w sieci lokalnej bez szyfrowania. Zastosowano następujące mechanizmy zabezpieczeń:

- **Walidacja danych wejściowych** - Wszystkie dane przychodzące przez API są walidowane za pomocą biblioteki `class-validator` w NestJS. Przykład walidacji:

  Listing 5: Przykład walidacji danych

```
export class CreateClientDto {
  @IsString()
  @IsNotEmpty()
  name: string;

  @IsEmail()
  @IsNotEmpty()
  email: string;
}
```

- **Integralność danych w bazie** - Baza danych SQLite wykorzystuje relacje i ograniczenia zapewniające integralność danych. Każda karta RFID może być przypisana tylko do jednego klienta lub jednej książki.
- **Ochrona przed błędnymi operacjami** - System weryfikuje poprawność operacji przed ich wykonaniem (np. sprawdza, czy książka jest dostępna przed wypożyczeniem, czy klient nie ma już wypożyczonej tej książki).

3.4 Inne istotne rozwiązania

3.4.1 Komunikacja w czasie rzeczywistym przez WebSockets

System wykorzystuje Socket.IO do przekazywania aktualizacji z backendu do frontendu w czasie rzeczywistym. Gdy Raspberry Pi skanuje kartę, backend natychmiast emituje zdarzenie do wszystkich podłączonych klientów webowych:

Listing 6: Gateway WebSocket - events.gateway.ts

```
@WebSocketGateway({ cors: { origin: '*' } })
export class EventsGateway {
  @WebSocketServer()
  server: Server;

  emit(event: string, data: any) {
    this.server.emit(event, data);
  }
}
```

W serwisie RFID, po przetworzeniu skanowania karty, emitowane jest zdarzenie:

```
this.gateway.emit('rfid/scanned', response);
```

3.4.2 Automatyczne tworzenie kart

System automatycznie tworzy nowe rekordy kart w bazie danych podczas rejestracji użytkownika lub książki, jeśli karta o danym UID nie istnieje. Pozwala to na elastyczne zarządzanie kartami bez konieczności ręcznego dodawania każdej karty przed użyciem podczas procesu rejestracji.

3.4.3 Obsługa błędów i timeoutów

System implementuje mechanizmy obsługi błędów:

- Timeout dla operacji skanowania (domyślnie 10 sekund).
- Automatyczne przełączanie LED na zielony po zakończeniu operacji.
- Generyczny komunikat o błędzie wyświetlany na wyświetlaczu OLED.
- Logowanie wszystkich operacji w konsoli dla celów diagnostycznych.

4 Opis działania i prezentacja interfejsu

4.1 Instalacja i uruchomienie aplikacji

4.1.1 Wymagania systemowe

- Node.js w wersji 18 lub nowszej
- Python 3.8 lub nowszy
- Docker i Docker Compose (dla brokera MQTT, jeśli nie będzie on postawiony na Raspberry Pi)
- npm lub yarn
- Raspberry Pi 4B z systemem Raspberry Pi OS (dla terminala RFID)

4.1.2 Instalacja i uruchomienie brokera MQTT

Broker MQTT (Eclipse Mosquitto) można uruchomić na dwa sposoby: przy użyciu kontenera Docker lub bezpośrednio w systemie operacyjnym Raspberry Pi.

Opcja A: Uruchomienie w kontenerze Docker Ta metoda pozwala na szybkie odizolowanie brokera od reszty systemu.

1. Przejdź do katalogu `mqtt/`:

   ```
   cd mqtt
   ```

2. Uruchom kontener Docker:

   ```
   docker-compose up -d
   ```

3. Broker będzie dostępny na porcie 1883 (domyślnie `localhost:1883`).

Opcja B: Uruchomienie w systemie Raspberry Pi OS (według instrukcji) W zestawach laboratoryjnych broker Mosquitto jest już zainstalowany, lecz domyślnie nie uruchamia się przy starcie systemu.

1. **Instalacja (jeśli wymagana):** Jeśli system nie posiada brokera, należy go zainstalować komendą:

```
sudo apt update
sudo apt install mosquitto mosquitto-clients
```

2. **Uruchomienie usługi:** Start brokera przeprowadzamy poleceniem:

```
sudo systemctl start mosquitto.service
```

3. **Weryfikacja:** Sprawdzenie statusu działania usługi:

```
sudo systemctl status mosquitto.service
```

4. **Dostęp zdalny:** Aby umożliwić dostęp z innych urządzeń w sieci izolowanej, należy do pliku `/etc/mosquitto/mosquitto.conf` dopisać:

```
allow_anonymous true
listener 1883 0.0.0.0
```

Następnie zrestartować usługę: `sudo systemctl restart mosquitto.service`.

4.1.3 Instalacja i uruchomienie backendu

1. Przejdź do katalogu `backend_nestjs/`:

```
cd backend_nestjs
```

2. Zainstaluj zależności:

```
npm install
```

3. (Opcjonalnie) Skonfiguruj zmienne środowiskowe w pliku `.env`:

```
PORT=3000
MQTT_BROKER_URL=mqtt://localhost:1883
MQTT_ENABLED=true
DB_DATABASE=database.sqlite
```

4. Uruchom serwer w trybie deweloperskim:

   ```
   npm run start:dev
   ```

5. Serwer będzie dostępny pod adresem `http://localhost:3000`.
6. Baza danych SQLite zostanie automatycznie utworzona przy pierwszym uruchomieniu wraz z przykładowymi danymi.

4.1.4 Instalacja i uruchomienie frontendu

1. Przejdź do katalogu `frontend_react/`:

   ```
   cd frontend_react
   ```

2. Zainstaluj zależności:

   ```
   npm install
   ```

3. Uruchom aplikację w trybie deweloperskim:

   ```
   npm run dev
   ```

4. Aplikacja będzie dostępna pod adresem `http://localhost:5173`.

4.1.5 Instalacja i uruchomienie terminala RFID na Raspberry Pi

1. Skopiuj katalog `raspberry-pi-python/` na Raspberry Pi.
2. Zainstaluj wymagane biblioteki Python:

   ```
   pip3 install -r requirements.txt
   ```

3. Skonfiguruj adres brokera MQTT w pliku `config.py`:

   ```
   MQTT_BROKER = "IP_ADRES_BACKENDU"
   MQTT_PORT = 1883
   ```

4. Uruchom główny skrypt:

   ```
   python3 main.py
   ```

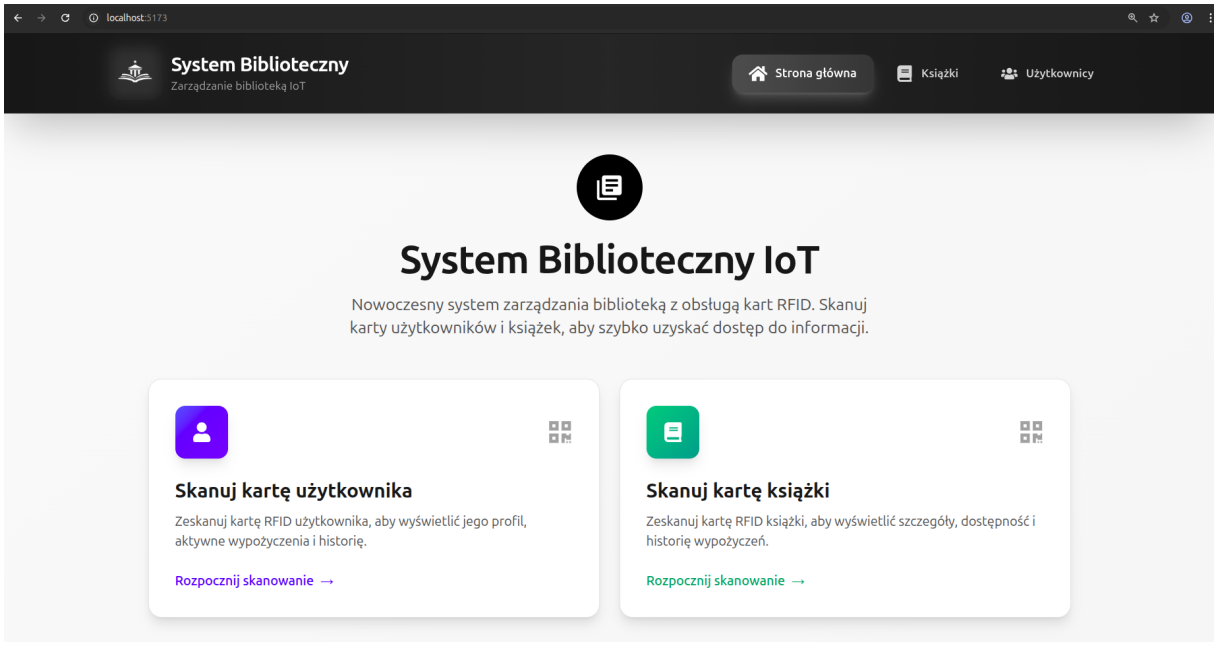
4.1.6 Weryfikacja działania systemu

Po uruchomieniu wszystkich komponentów:

1. Otwórz aplikację webową w przeglądarce (`http://localhost:5173`).
2. Sprawdź, czy backend odpowiada (API dostępne pod `http://localhost:3000`).
3. Sprawdź, czy broker MQTT działa (można użyć narzędzi takich jak MQTT Explorer).
4. Przetestuj skanowanie karty RFID na Raspberry Pi - informacje powinny pojawić się w aplikacji webowej w czasie rzeczywistym.

Aplikacja webowa oferuje następujące funkcjonalności:

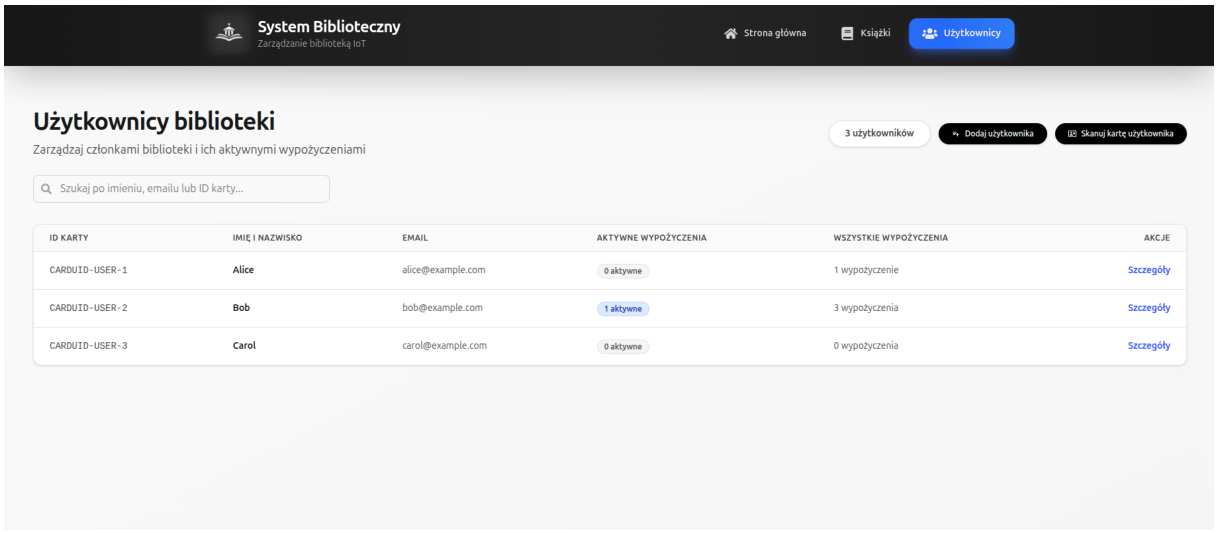
4.1.7 Strona główna



Rysunek 2: Strona główna

Strona główna pełni funkcję centrum nawigacyjnego systemu. Zawiera sekcję hero z tytułem i opisem systemu, dwie główne karty akcji umożliwiające skanowanie kart RFID użytkowników i książek, sekcję szybkiego dostępu z przyciskami prowadzącymi do listy użytkowników i książek, oraz sekcję informacyjną "Jak to działa?"wyjaśniającą proces korzystania z systemu w trzech krokach. Po kliknięciu karty skanowania, wyświetla się dialog z instrukcjami, a system oczekuje na przyłożenie karty RFID do czytnika.

4.1.8 Zarządzanie klientami

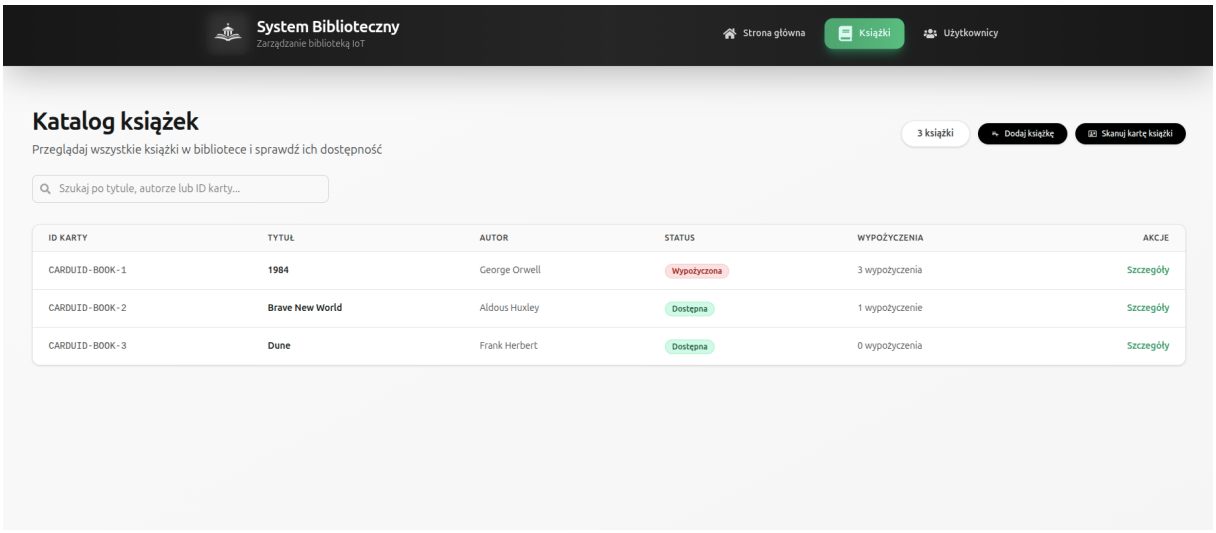


Rysunek 3: Lista klientów

Strona `/clients` umożliwia:

- Przeglądanie listy wszystkich klientów biblioteki.
- Dodawanie nowych klientów (imię, nazwisko, email).
- Edycję danych istniejących klientów.
- Usuwanie klientów.
- Przypisywanie kart RFID do klientów.
- Przeglądanie historii wypożyczeń danego klienta.

4.1.9 Zarządzanie książkami



Rysunek 4: Lista książek

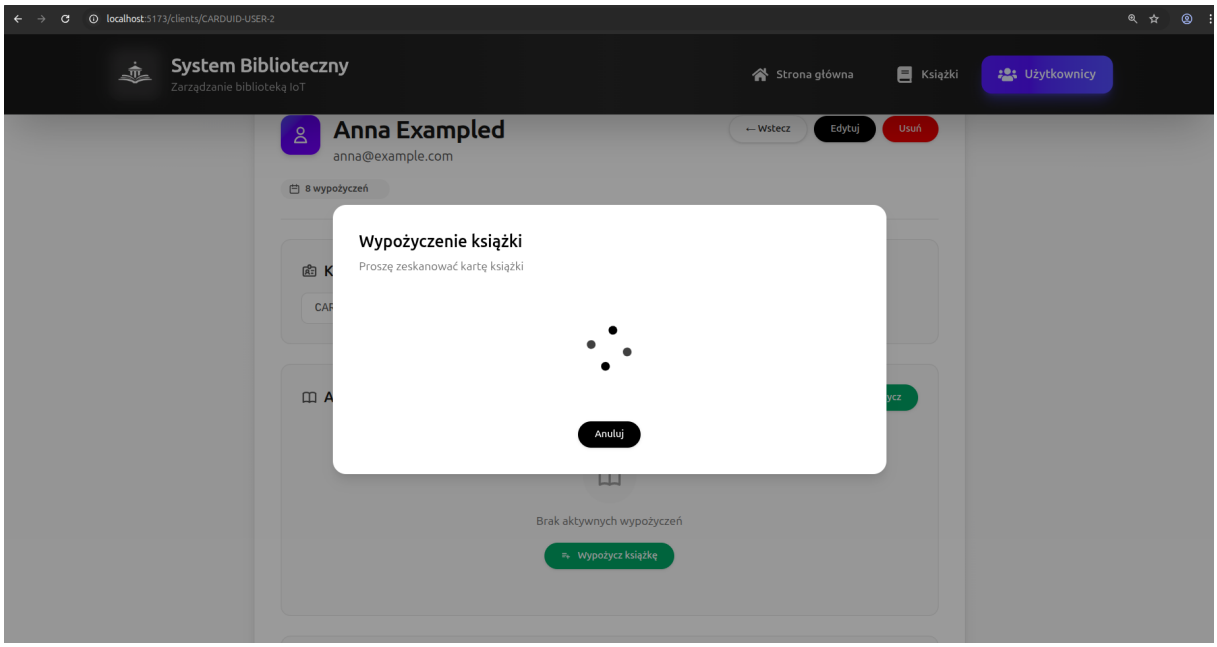
Strona `/books` umożliwia:

- Przeglądanie listy wszystkich książek w bibliotece.
- Dodawanie nowych książek (tytuł, autor).
- Edycję danych istniejących książek.
- Usuwanie książek.
- Przypisywanie kart RFID do książek.
- Sprawdzanie dostępności książek.

4.1.10 Proces wypożyczania i zwracania

System umożliwia wypożyczanie i zwracanie książek na dwa sposoby:

1. **Przez terminal RFID:**
    - Zeskanuj kartę klienta (LED zmieni się na zielony).
    - Zeskanuj kartę książki (system automatycznie utworzy wypożyczenie lub zwróci wypożyczoną książkę).
    - Informacja o wypożyczeniu pojawi się w aplikacji webowej w czasie rzeczywistym.
2. **Przez interfejs webowy:**



Rysunek 5: Wypożyczenie

- Wybierz klienta i książkę z listy.
- Kliknij przycisk "Wypożycz"lub "Zwróć".
- System automatycznie zaktualizuje status wypożyczenia.

4.1.11 Wyświetlacz OLED na terminalu RFID

Terminal RFID wyświetla następujące informacje na wyświetlaczu OLED:

- **Oczekiwanie na kartę** - komunikat zachęcający do przyłożenia karty.
- **Wykryto kartę** - wyświetlenie UID zeskanowanej karty.
- **Przetwarzanie** - informacja o przetwarzaniu danych przez backend.
- **Dane klienta/książki** - wyświetlenie informacji o znalezionym kliencie lub książce.
- **Nowa karta** - informacja o nieznanej karcie.
- **Błąd** - komunikat o błędzie operacji.

5 Opis wkładu pracy Autorów

5.1 Dawid Błaszczyk

- Konfiguracja i uruchomienie środowiska do testowania komunikacji MQTT.

- Testowanie integracji między komponentami systemu.
- Przygotowanie raportu.
- Weryfikacja zgodności z wymaganiami funkcjonalnymi i niefunkcjonalnymi.
- Przygotowanie instrukcji instalacji i uruchomienia.
- Testowanie scenariuszy użycia i identyfikacja błędów.
- Dodanie nowych funkcjonalnośći do Frontendu i Backendu.

5.2 Błazej Kowal

- Implementacja backendu w NestJS (struktura modułów, serwisy, kontrolery).
- Implementacja komunikacji MQTT między Raspberry Pi a backendem.
- Implementacja WebSockets (Socket.IO) dla aktualizacji w czasie rzeczywistym.
- Projekt i implementacja bazy danych SQLite z wykorzystaniem TypeORM.
- Implementacja logiki wypożyczeń i zwrotów książek.

5.3 Alina Lenart

- Implementacja kodu Python dla Raspberry Pi (czytnik RFID, MQTT client).
- Implementacja obsługi wyświetlacza OLED.
- Implementacja kontroli diod LED i buzzera.
- Testowanie i debugowanie komunikacji MQTT.
- Dokumentacja kodu Python i instrukcje instalacji.
- Implementacja symulatora Raspberry Pi do testowania bez fizycznego urządzenia.

5.4 Bartosz Wacławiak

- Implementacja aplikacji webowej w React z TypeScript.
- Implementacja routingu i nawigacji w aplikacji (React Router).
- Implementacja komunikacji z backendem przez REST API.

- Implementacja integracji z WebSockets dla aktualizacji w czasie rzeczywistym.
- Implementacja komponentów do zarządzania klientami, książkami i wypożyczeniami.

6 Podsumowanie

Projekt został zrealizowany zgodnie z założonymi wymaganiami funkcjonalnymi i niefunkcjonalnymi. System biblioteczny IoT spełnia wszystkie podstawowe wymagania:

6.1 Stopień zgodności z wymaganiami

6.1.1 Wymagania funkcjonalne

Wszystkie wymagania funkcjonalne zostały zrealizowane:

- System umożliwia odczyt kart RFID za pomocą czytnika MFRC522.
- System automatycznie wykrywa przyłożenie i zabranie karty RFID.
- Odczytany identyfikator karty jest konwertowany do formatu szesnastkowego i przesyłany do serwera.
- System rozróżnia między kartami klientów i książek.
- System przechowuje informacje o klientach, książkach i wypożyczeniach w bazie danych SQLite.
- System umożliwia wypożyczanie i zwracanie książek przez skanowanie kart.
- System posiada webowy interfejs graficzny z pełną funkcjonalnością zarządzania.
- Komunikacja MQTT działa poprawnie między Raspberry Pi a backendem.
- System sygnalizuje stany operacji za pomocą diod LED, buzzera i wyświetlacza OLED.

6.1.2 Wymagania niefunkcjonalne

Większość wymagań niefunkcjonalnych została spełniona:

- System przetwarza zdarzenia RFID w czasie rzeczywistym (opóźnienie < 1s).
- Aplikacja webowa ładuje się szybko dzięki wykorzystaniu Vite i React.

- System jest odporny na tymczasową utratę połączenia z brokerem MQTT.
- Baza danych zapewnia integralność danych.
- Architektura umożliwia łatwe dodanie kolejnych terminali RFID.
- System działa na wymaganych platformach (Linux, macOS, Windows dla backendu, Raspberry Pi OS dla terminala).
- Komunikacja MQTT odbywa się bez szyfrowania (zgodnie z założeniami dla wersji proof-of-concept w sieci lokalnej).

6.2 Napotkane trudności

W trakcie implementacji napotkano na następujące trudności:

6.2.1 Problemy techniczne

- **Integracja WebSockets z React** - Wymagało to właściwej konfiguracji Socket.IO po stronie klienta i serwera oraz obsługi reconnectów.
- **Konfiguracja CORS** - Wymagana była odpowiednia konfiguracja CORS w backendzie NestJS oraz WebSocket Gateway dla komunikacji między frontendem a backendem.

6.2.2 Ograniczenia

- **Brak fizycznego urządzenia podczas części rozwoju** - Część funkcjonalności była testowana przy użyciu symulatora Raspberry Pi, co wymagało dodatkowego czasu na weryfikację na rzeczywistym urządzeniu.

6.3 Kierunki dalszego rozwoju systemu

Możliwe kierunki rozbudowy i ulepszeń systemu:

6.3.1 Rozbudowa funkcjonalności

- System kar za przetrzymanie książek.
- Historia wypożyczeń z możliwością eksportu do pliku.
- Wyszukiwarka zaawansowana z filtrowaniem po wielu kryteriach.
- System powiadomień email/SMS o terminach zwrotu.

- Obsługa wielu bibliotek w jednym systemie.

6.3.2 Poprawa bezpieczeństwa

- Szyfrowanie komunikacji MQTT (MQTTS z certyfikatami TLS/SSL).

6.3.3 Optymalizacja wydajności

- Migracja z SQLite do PostgreSQL lub MySQL dla większej wydajności i skalowalności.
- Lazy loading komponentów w aplikacji React.

6.3.4 Ulepszenia techniczne

- Implementacja testów jednostkowych i integracyjnych.
- CI/CD pipeline dla automatycznego testowania i wdrażania.
- Dokumentacja API (Swagger/OpenAPI).
- Wsparcie dla wielu języków (i18n).

7 Literatura

- NestJS Documentation. *Official NestJS Documentation*. Dostępne online: `https://docs.nestjs.com/`.
- React Documentation. *React - A JavaScript library for building user interfaces*. Dostępne online: `https://react.dev/`.
- TypeORM Documentation. *TypeORM - Amazing ORM for TypeScript and JavaScript*. Dostępne online: `https://typeorm.io/`.
- MQTT.org. *MQTT - The Standard for IoT Messaging*. Dostępne online: `https://mqtt.org/`.
- Paho MQTT Python Client. *Eclipse Paho MQTT Python Client Library*. Dostępne online: `https://www.eclipse.org/paho/index.php?page=clients/python/docs/index.php`.
- MFRC522 Python Library. *MFRC522-python - A Python library for the MFRC522 RFID reader*. Dostępne online: `https://github.com/mxgxw/MFRC522-python`.

- Socket.IO Documentation. *Socket.IO - Bidirectional and low-latency communication for every platform*. Dostępne online: `https://socket.io/docs/v4/`.
- Tailwind CSS Documentation. *Tailwind CSS - Rapidly build modern websites*. Dostępne online: `https://tailwindcss.com/docs`.
- Vite Documentation. *Vite - Next Generation Frontend Tooling*. Dostępne online: `https://vitejs.dev/`.
- SQLite Documentation. *SQLite - A self-contained, serverless, zero-configuration SQL database engine*. Dostępne online: `https://www.sqlite.org/docs.html`.
- Raspberry Pi Foundation. *Raspberry Pi Documentation*. Dostępne online: `https://www.raspberrypi.com/documentation/`.
- TypeScript Documentation. *TypeScript - JavaScript with syntax for types*. Dostępne online: `https://www.typescriptlang.org/docs/`.

8 Aneks

Kod źródłowy projektu został dołączony w formie elektronicznej jako załącznik. Projekt zawiera następujące komponenty:

- `backend_nestjs/` - Kod źródłowy backendu w NestJS (TypeScript).
- `frontend_react/` - Kod źródłowy aplikacji webowej w React (TypeScript).
- `raspberry-pi-python/` - Kod źródłowy aplikacji dla Raspberry Pi (Python).
- `mqtt/` - Skrypty i konfiguracja pomocnicza do testów komunikacji MQTT.
- `rpi-simulators/` - Symulator Raspberry Pi do testowania bez fizycznego urządzenia.
- `raport/` - Dokumentacja projektu (LaTeX).

Wszystkie pliki konfiguracyjne, zależności i instrukcje instalacji znajdują się w odpowiednich katalogach projektu.