Korepetycje App

Igor Osiakowski, Michał Kruczyńskik, Mikołaj Lewandowski, Jakub Kucharek, Mateusz Ługowski

Spis treści

1 Wprowadzenie

Projekt **Korepetycje App** jest aplikacją służącą do rezerwacji korepetycji przez uczniów i udzielania lekcji w danym terminie jako korepetytor. System pozwala zarejestrować się uczniom i nauczycielom albo zalogować się jako administrator. Uczniowie mogą umawiać lekcje z nauczycielami w ustalonym terminie, na konkretnym poziomie nauczania z danego przedmiotu, a nauczyciele mogą udzielać korepetycji, zarządzać swoim profilem, wystawiać opinie i raporty oraz generować plan zajęć w formacie PDF na kolejny miesiąc. System wspiera również funkcjonalność recenzji nauczycieli, dzięki czemu uczniowie mogą oceniać jakość korepetycji, uczniowie i nauczyciele mają również dostęp do zmiany swoich danych konta. Aplikacja jest wdrożona na platformie Microsoft Azure i wykorzystuje tam w pełni zarządzane usługi: App Service dla frontendu, Azure Database for PostgreSQL dla bazy danych oraz Azure Container Apps dla serwisu e-mail i backendu.

2 Opis aplikacji

Aplikacja stanowi platformę edukacyjną umożliwiającą połączenie uczniów (studentów) z nauczycielami w celu organizowania indywidualnych lekcji online. System zapewnia wsparcie dla rejestracji i logowania użytkowników z podziałem na role: *uczeń*, *nauczyciel* oraz *administrator*. Dzięki funkcjom takim jak planowanie lekcji, wystawianie raportów, dobór poziomu trudności czy filtrowanie po przedmiotach, aplikacja umożliwia kompleksowe zarządzanie procesem nauczania indywidualnego.

Dodatkowo projekt powstał z myślą o temacie **bezpieczeństwa usług chmurowych**, dlatego aplikacja zawiera mechanizmy uwierzytelniania (w tym JWT), rejestracji z weryfikacją e-mailową oraz podział uprawnień dostępu do różnych zasobów systemu. Komunikacja pomiędzy usługami została zaprojektowana z wykorzystaniem architektury mikroserwisowej.

3 Użyte Usługi i Technologie

- **Flask:** Główna platforma zarówno backendu, API jak i frontendu.
- **Flasgger (Swagger):** Automatycznie generuje dokumentację API, umożliwiając interaktywne testowanie endpointów (dostępne pod `/apidocs`).
- **SQLAlchemy:** ORM do zarządzania bazą danych PostgreSQL.
- **Flask-JWT-Extended:** Mechanizm autoryzacji i uwierzytelniania (JWT).
- **APScheduler:** Zarządza zadaniami okresowymi, np. aktualizacją statusów lekcji i czyszczeniem wygasłych kont.
- **Microsoft Azure:** Środowisko produkcyjne, zastąpiło poprzednią konteneryzację w Dockerze:
  - App Service – hostowanie frontendu
  - Azure Database for PostgreSQL – zarządzana baza
  - Azure Container apps - `email_service` wraz z backendem to aplikacje kontenerowe
  - Azure Key Vault – przechowywanie sekretów (SMTP, JWT_SECRET)

Omijanie serwera SMTP W przypadku problemów z serwerem SMTP dostępny jest testowy endpoint `/auth/test/register` z którego możemy skorzystać wchodząc na:

`https://<your-frontend>.azurewebsites.net/auth/test/register`

Tym samym możemy utworzyć konto bez wysyłki wiadomości aktywacyjnej (parametr `is_test=True` wewnątrz usługi). (Plik `auth_api.py` rejestruje ścieżkę testową jako /auth/test/register)

Rejestrowanie się jako admin Jeżeli chcemy zarejestrować się jako admin korzystamy z endpoint `/admin/auth` z którego możemy skorzystać wchodząc na: `https://<your-frontend>.azurewebsites.net/admin/register`

4 Struktura i działanie komponentów projektu

- **Backend (app):** Obsługuje logikę API, uwierzytelnianie, operacje na bazie danych oraz generowanie dokumentacji Swagger. Działa jako wdrożenie kontenerowe w ramach Azure Container Apps i komunikuje się z bazą danych PostgreSQL.
- **Frontend (frontend):** Odpowiada za interfejs użytkownika. Komunikuje się z backendem przez API, umożliwiając rejestrację, logowanie, przeglądanie lekcji, dodawanie recenzji itp.
- **Email Service** (`email_service`): Usługa odpowiedzialna za wysyłkę wiadomości e-mail (potwierdzenia rejestracji) również działa jako wdrożenie kontenerowe.
- **Baza Danych (PostgreSQL):** Przechowuje dane o użytkownikach, lekcjach, recenzjach, raportach oraz innych modelach aplikacji.

4.1 Instalacja (Azure)

Poniżej znajduje się skrócony opis, jak lokalnie przetestować wdrożone na Azure komponenty aplikacji. Pełniejsze instrukcje znajdują się w dedykowanych plikach `README.md` w katalogach każdego serwisu.

1. Frontend (App Service)
   - Otwórz `frontend/README.md` (dedykowana instrukcja).
   - Skonfiguruj zmienne aplikacji: - `BACKEND_URL`, `SECRET_KEY`.
   - Wdróż kontener: `ghcr.io/projbezpieczenstwo/frontend:817e1565ad51cf99b13e3336ead1b8c5a07d714a`.
2. Backend (Container Apps)
   - Otwórz `app/README.md` (dedykowana instrukcja).
   - Ustaw zmienne środowiskowe: `DATABASE_URI`,`SECRET_KEY`,`JWT_SECRET_KEY`,`EMAIL_SERVICE_URI`,`ADMIN_SECRET`.
   - dodaj `ssl_mode = require`
   - Wdróż kontener: `ghcr.io/projbezpieczenstwo/backend:latest`.
3. Email Service (Container Apps)
   - Otwórz `email_service/README.md` (dedykowana instrukcja).
   - Skonfiguruj SMTP: `SMTP_EMAIL`, `SMTP_PASSWORD`, `SMTP_SERVER`, `SMTP_PORT`, `FRONTEND_URI`.
   - Wdróż kontener: `ghcr.io/projbezpieczenstwo/email_service:latest`.
4. Baza danych (Azure Database for PostgreSQL)
   - Otwórz `email_service/README.md` (opis dotyczący bazy danych zawiera się w README dla `email_service`).
   - Utwórz serwer PostgreSQL w Azure (PostgreSQL v17).
   - Zapisz login i hasło – potrzebne do składni `DATABASE_URI`.
5. Testowanie lokalne po wdrożeniu:
   - Frontend: `https://<your-frontend>.azurewebsites.net`
   - Backend API: `https://<your-backend>.azurecontainerapps.io`
   - Email API: `https://<your-email>.azurecontainerapps.io`

5 Działanie i obsługa aplikacji

Po wejściu na stronę główną użytkownik (tzw. *gość*) ma możliwość rejestracji jako uczeń, nauczyciel lub administrator. Proces rejestracji obejmuje podanie danych osobowych, e-maila oraz hasła, a następnie weryfikację adresu e-mail poprzez specjalny link aktywacyjny. Po aktywacji konta można zalogować się do systemu.

Jeżeli chcemy mieć dostęp do konta admina korzystamy z endpointu admin/register: http://localhost:8000/admin/registe
Przy czym sekret to: secret

- **Uczeń** może przeglądać dostępnych nauczycieli, rezerwować lekcje, anulować je, przeglądać historię swoich lekcji, dodawać i usuwać recenzje, edytować własny profil, oraz śledzić historię spotkań.
- **Nauczyciel** może zarządzać swoim kalendarzem dostępności, dodawać raporty do lekcji, generować miesięczny plan zajęć w PDF, przeglądać swoje lekcje i aktualizować profil.
- **Administrator** może wyswietlać, usuwać i tworzyć nowe kody dostępu. Może również edytować credentials.

Aplikacja jak już wcześniej było wspomniane składa się z dwóch głównych komponentów: frontend (interfejs użytkownika oparty o Flask z szablonami HTML) oraz backend (API REST obsługujące logikę aplikacyjną i operacje na bazie danych). Komunikacja z backendem odbywa się poprzez żądania HTTP, a dane przesyłane są w formacie JSON. Uwierzytelnianie odbywa się z wykorzystaniem tokenów JWT, które są przechowywane w sesji użytkownika.

6 Dokumentacja API

Pełna dokumentacja API dostępna jest dzięki Flasgger pod adresem: `http://localhost:5000/apidocs`

Poniżej kilka przykładowych endpointów:

A. Difficulty Levels

- **GET /api/difficulty-levels**
  Opis: Pobiera listę dostępnych poziomów nauczania (np. "Primary School", "Bachelor's").
  Przykładowa odpowiedź:


```
{
  "difficulty_levels": [
    {
      "id": 1,
      "name": "Primary School"
    }
  ]
}
```


Brak parametrów wejściowych; w przypadku błędu zwracany jest kod 500.

B. Lessons

- **GET /api/lesson**
  Opis: Pobiera wszystkie lekcje przypisane do aktualnie zalogowanego użytkownika.
  Przykładowa odpowiedź:


```
{
  "lesson_list": [
    {
      "id": 1,
      "date": "15/01/2025 10:00",
      "price": 60,
      "status": "Scheduled",
      "student_id": 3,
      "subject": "Physics",
```

```
      "teacher_id": 5
    }
  ]
}
```


W przypadku braku lekcji zwracany jest błąd 400.

POST /api/lesson

Opis: Umożliwia uczniowi umówienie się na lekcje.

Dane wejściowe:


```
{
  "date": "15/01/2025 10:00",
  "difficulty_id": 2,
  "subject_id": 3,
  "teacher_id": 1
}
```


Odpowiedzi:

- 201 – Lekcja utworzona pomyślnie.
- 400 – Błąd walidacji.
- 401 – Użytkownik nieautoryzowany.
- 404 – Nie znaleziono nauczyciela, przedmiotu lub poziomu trudności.

C. Reports

- **GET /api/report**
  Opis: Pobiera raporty lekcji dla nauczyciela lub ucznia.
  Przykładowa odpowiedź:


```
{
  "report_list": [
    {
      "comment": "Well done!",
      "date": "15/01/2025 10:00",
      "homework": "Complete exercises 5-10.",
      "progress_rating": 4,
      "student_name": "Dean Bergs",
      "subject": "Biology",
      "teacher_name": "John Doe"
    }
  ]
}
```


W przypadku braku raportów zwracany jest błąd 400.

POST /api/report

Opis: Umożliwia nauczycielowi dodanie raportu do lekcji (komentarz, ocena postępów, zadanie domowe).

Dane wejściowe:


```
{
  "comment": "Good progress made today.",
  "homework": "Complete exercises 5-10.",
  "lesson_id": 10,
  "progress_rating": 4
}
```


Odpowiedzi:

- 201 – Raport utworzony.
- 400 – Błąd walidacji.
- 404 – Lekcja nie znaleziona.
- 500 – Błąd serwera.

D. Subjects

- **GET /api/subjects**
  Opis: Pobiera listę dostępnych przedmiotów (np. Mathematics, Physics).
  Przykładowa odpowiedź:

```
{
  "subjects": [
    {
      "id": 1,
      "name": "Mathematics"
    }
  ]
}
```

W przypadku problemów zwracany jest błąd 500.

E. Teachers

- **GET /api/teacher-list**
  Opis: Pobiera listę nauczycieli z możliwością filtrowania po przedmiocie (`subject`) lub poziomie trudności (`difficulty_id`).
  Parametry zapytania:
  - `subject` – integer, np. 2.
  - `difficulty_id` – integer, np. 1.

  Przykładowa odpowiedź:

```
{
  "teacher_list": [
    {
      "id": 5,
      "name": "John Doe",
      "bio": "Native speaker",
      "difficulty_levels": [3],
      "subjects": [2]
    }
  ]
}
```

W przypadku braku nauczycieli lub nieprawidłowych danych, mogą zostać zwrócone kody 400 lub 404.

PUT /api/teacher-update
Opis: Umożliwia nauczycielowi aktualizację swojego profilu (przedmioty, poziomy trudności, stawka godzinowa).
Dane wejściowe:

```
{
  "difficulty_ids": [1, 2],
  "hourly_rate": 50,
  "subject_ids": [1, 2]
}
```

Odpowiedzi:

- 200 – Aktualizacja pomyślna.
- 400 – Błąd walidacji.
- 401 – Użytkownik nieautoryzowany.
- 404 – Nie znaleziono przedmiotu lub poziomu trudności.

F. Reviews

- **GET /api/teacher-reviews**
  Opis: Pobiera wszystkie recenzje wystawione dla nauczycieli.
  Przykładowa odpowiedź:


```
{
  "reviews": [
    {
      "id": 0,
      "teacher_id": 0,
      "student_id": 0,
      "rating": 0,
      "comment": "string",
      "created_at": "2025-03-30T17:27:58.657Z"
    }
  ]
}
```


W przypadku braku autoryzacji lub błędów serwera, odpowiednie kody odpowiedzi to 401 lub 500.

DELETE /api/teacher-reviews/{teacher_id}
Opis: Umożliwia uczniowi usunięcie recenzji wystawionej dla danego nauczyciela.
Parametr:

- `teacher_id` – integer, identyfikator nauczyciela.

Przykładowa odpowiedź:


```
{
  "message": "Review deleted successfully"
}
```


- **Uwagi:** W przypadku problemów (np. użytkownik nie jest uczniem lub recenzja nie istnieje) mogą zostać zwrócone kody 400 lub 404.

GET /api/teacher-reviews/{teacher_id}
Opis: Pobiera recenzje dla konkretnego nauczyciela.
Parametr:

- `teacher_id` – integer, identyfikator nauczyciela.

POST /api/teacher-reviews/{teacher_id}
Opis: Umożliwia uczniowi dodanie recenzji dla nauczyciela.
Dane wejściowe:


```
{
  "comment": "Great teacher!",
  "rating": 5
}
```


Odpowiedzi:

- 200 – Recenzja dodana.
- 400/401 – Błąd walidacji lub brak autoryzacji.

G. Authentication

- **POST /auth/login**
  Opis: Logowanie użytkownika. System weryfikuje poprawność podanego adresu e-mail i hasła, a następnie zwraca token JWT.
  Dane wejściowe:

```
{
  "email": "john.doe@example.com",
  "password": "password123"
}
```

Przykładowa odpowiedź:

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR...",
  "message": "Login successful.",
  "role": "student"
}
```

W przypadku błędnych danych lub braku pól wejściowych zwracane są kody 400 lub 401.

- **POST /auth/register**
  Opis: Rejestracja nowego użytkownika (ucznia lub nauczyciela). Walidacja obejmuje sprawdzenie poprawności adresu e-mail oraz wymaganych pól. Dla nauczycieli dodatkowo wymagane są dane dotyczące przedmiotów, poziomów trudności oraz stawki godzinowej.
  Dane wejściowe:

```
{
  "difficulty_ids": [1, 2],
  "email": "john.doe@example.com",
  "hourly_rate": 50,
  "name": "John Doe",
  "password": "password123",
  "role": "teacher",
  "subject_ids": [1, 2]
}
```

Odpowiedzi:

- 201 – Rejestracja udana.
- 400 – Błędy walidacji (np. brak wymaganych pól, nieprawidłowy format e-mail, niezgodność ról, email już w użyciu).
- 401/500 – Inne problemy związane z autoryzacją lub błędami serwera.

6.1 Opis wybranych funkcji

Opis funkcji dodatkowych

6.2 Anulowanie lekcji

Endpoint: `PUT /api/lesson/{lesson_id}`
Opis: Umożliwia użytkownikowi (studentowi lub nauczycielowi) anulowanie lekcji maksymalnie na godzinę przed jej rozpoczęciem. **Dane wejściowe:**

```
{ "comment": "Powód odwołania lekcji" }
```

Odpowiedź:

- `200 OK` + `"Lesson successfully canceled"`
- `400 Bad Request` (lekcja nie istnieje lub za późno na anulowanie)

6.3 Naprawa terminu lekcji (rebook)

Endpoint: `PUT /api/lesson/{lesson_id}` (ten sam co anulowanie, tylko inny fragment kodu)
Opis: Poprawiony warunek sprawdzający dopuszczalność zmiany terminu — jeżeli na tę samą godzinę dostępny jest nowy slot, lekcję można przenieść.

6.4 Zachowywanie i reset filtrów na froncie

Filtry w przeglądarce nauczycieli (subject, difficulty) zapisują się w `sessionStorage`, a przyciskiem „Reset" czyści się wszystkie wybrane wartości.

6.5 Ustawienia konta użytkownika

- **GET /auth/user** – zwraca dane aktualnie zalogowanego użytkownika.
- **POST /auth/update** – umożliwia aktualizację swojego profilu (email, password, name, a dla Teacher także bio, hourly_rate, subject_ids, difficulty_level_ids).
- **POST /auth/update/{user_id}** – (rola `admin`) pozwala adminowi zmienić dane dowolnego użytkownika.

6.6 Account settings

Front: przycisk „Account settings" w menu prowadzi do endpointu `/auth/user`, skąd frontend pobiera JSON z polami usera i wypełnia formularz.

7 Bezpieczeństwo

Aplikacja *"Korepetycje App"* została zaprojektowana z myślą o zapewnieniu wysokiego poziomu bezpieczeństwa, szczególnie w kontekście usług chmurowych. W systemie zaimplementowano kilka kluczowych mechanizmów zabezpieczających:

- **Weryfikacja i walidacja danych:** Przed przyjęciem danych rejestracyjnych, aplikacja weryfikuje poprawność formatu adresu e-mail przy użyciu wyrażenia regularnego. Dodatkowo, walidowane są wszystkie niezbędne pola (imię, e-mail, hasło, rola), co zapobiega przesyłaniu niekompletnych lub nieprawidłowych danych.
- **Haszowanie haseł:** Hasła użytkowników są haszowane przy użyciu funkcji z modułu `werkzeug.security`, co gwarantuje, że nawet w przypadku wycieku danych, hasła pozostaną chronione.
- **Mechanizm JWT:** Po udanym logowaniu, użytkownik otrzymuje token JWT (JSON Web Token), który jest wykorzystywany do uwierzytelniania przy dostępie do chronionych zasobów API. Tokeny te są generowane z odpowiednim czasem ważności, co ogranicza ryzyko ich nadużycia.
- **Rejestracja z potwierdzeniem e-mail:** Proces rejestracji wymaga weryfikacji adresu e-mail. Użytkownik, po przesłaniu danych rejestracyjnych, otrzymuje wiadomość e-mail z unikalnym linkiem aktywacyjnym (`auth_key`). Dopiero po kliknięciu w link konto zostaje aktywowane, co minimalizuje ryzyko rejestracji fałszywych użytkowników.
- **Dodatkowe zabezpieczenia dla roli nauczyciela:** Rejestracja jako nauczyciel wymaga podania poprawnego kodu (`teacher_code`), a także weryfikacji podanych identyfikatorów przedmiotów oraz poziomów trudności. To dodatkowe zabezpieczenie pozwala ograniczyć możliwość nieautoryzowanego dostępu do roli nauczyciela.
- **Izolacja usług:** Aplikacja jest uruchamiana w kontenerach Docker, co pozwala na izolację poszczególnych komponentów (backend, frontend, usługa e-mail, baza danych) oraz ogranicza możliwość ataków sieciowych. Dodatkowo, komunikacja między serwisami odbywa się w ramach prywatnych sieci Docker.
- **Monitorowanie i logowanie:** System loguje kluczowe zdarzenia (np. próby rejestracji, logowania, wysyłki e-maili), co umożliwia szybkie wykrycie i reakcję na potencjalne incydenty bezpieczeństwa.

Wszystkie te mechanizmy razem zapewniają, że aplikacja jest zabezpieczona zarówno na poziomie danych użytkowników, jak i komunikacji między usługami, co jest kluczowe w środowisku usług chmurowych.

7.1 Testy bezpieczeństwa

Poniżej opisujemy, jak uruchomić i czego oczekiwać od automatycznych testów weryfikujących powyższe mechanizmy.

7.1.1 Uruchamianie testów

Testy wykonujemy narzędziem `pytest` z poziomu głównego katalogu projektu:

```
pytest --maxfail=1 --disable-warnings -q
```

Parametry:

- `-maxfail=1` – zatrzymuje testy po pierwszej porażce,
- `-disable-warnings` – wycisza ostrzeżenia,
- `-q` – tryb cichy (quiet).

7.1.2 Scenariusze i oczekiwane wyniki

Próba 1: Niepoprawny e-mail przy rejestracji

Żądanie: `POST /auth/register` z `email="invalid"`
Oczekiwanie: `400 Bad Request`, `"Invalid email format."`

```
response = test_client.post('/auth/register', json={
  'name':'X','email':'invalid','password':'p','role':'student'})
assert response.status_code==400
assert response.json['message']=='Invalid email format.'
```

Próba 2: Poprawna rejestracja studenta

Żądanie: `POST /auth/register` z poprawnym e-mailem
Oczekiwanie: `200 OK`, `"Verify your email now!"`

```
response = test_client.post('/auth/register', json={
  'name':'Test','email':'t@mail.com','password':'p','role':'student'})
assert response.status_code==200
assert response.json['message']=='Verify your email now!'
```

Próba 3: Logowanie przed weryfikacją e-mail

Żądanie: `POST /auth/login` dla niepotwierdzonego konta
Oczekiwanie: `401 Unauthorized`, `"Verify your email."`

```
response = test_client.post('/auth/login', json={
  'email':'t@mail.com','password':'p'})
assert response.status_code==401
assert response.json['message']=='Verify your email.'
```

Próba 4: Logowanie po potwierdzeniu e-mail

Żądanie: `GET /auth/confirm/{auth_key}`
Następnie: `POST /auth/login`
Oczekiwanie: `200 OK`, zwrot `access_token`

```
# potwierdzenie:
test_client.get(f'/auth/confirm/{temp.auth_key}')
response = test_client.post('/auth/login', json={...})
assert response.status_code==200
assert 'access_token' in response.json
```

Próba 5: Dostęp z niewłaściwą rolą

Żądanie: `POST /api/calendar` z tokenem studenta
Oczekiwanie: `403 Forbidden`, `"User must be a teacher"`

```
headers={'Authorization':f'Bearer {login_student}'}
response = test_client.post('/api/calendar', headers=headers, json={...})
assert response.status_code==403
assert response.json['message']=='User must be a teacher'
```

8 Pozostałe Testy Bezpieczeństwa i Aplikacji

Aby zapewnić, że wszystkie mechanizmy uwierzytelniania, autoryzacji i walidacji działają zgodnie z oczekiwaniami, w projekcie zaimplementowano zestaw automatycznych testów bezpieczeństwa przy użyciu `pytest`.

8.1 Uruchamianie testów

Testy uruchamiamy z katalogu głównego projektu poleceniem:

```
pytest --maxfail=1 --disable-warnings -q
```

Parametry:

- `-maxfail=1` – zatrzymuje testy po pierwszej porażce,
- `-disable-warnings` – wycisza komunikaty ostrzegawcze,
- `-q` – skrócony (quiet) tryb wyjścia.

8.2 Opis scenariuszy i oczekiwane wyniki

Poniżej kluczowe przypadki testowe sprawdzające bezpieczeństwo logiki autoryzacji i walidacji:

1. **Rejestracja użytkownika – student**
   Test: wysłanie żądania `POST /auth/register` z poprawnymi danymi studenta. *Oczekiwane:* kod `200` i komunikat `"Verify your email now!"`.

   ```
   def test_register_student(test_client):
       response = test_client.post('/auth/register', json={
           'name': 'Test Student',
           'email': 'student@gmail.com',
           'password': 'password123',
           'role': 'student'
       })
       assert response.status_code == 200
       assert response.json['message'] == 'Verify your email now!'
   ```

2. **Rejestracja duplikatu e-maila**
   Test: próba ponownej rejestracji z tym samym adresem e-mail. *Oczekiwane:* kod `400` i komunikat `"Email already in use."`.

   ```
   def test_register_duplicate_email(test_client, setup_users):
       response = test_client.post('/auth/register', json={
           'name': 'Duplicate User',
           'email': 'student@gmail.com',
           'password': 'password123',
   ```

```
        'role': 'student'
    })
    assert response.status_code == 400
    assert response.json['message'] == 'Email already in use.'
```

3. **Logowanie przed weryfikacją e-mail**
   Test: wysłanie `POST /auth/login` dla konta niepotwierdzonego. *Oczekiwane:* kod `401` i komunikat `"Verify your email."`.

```
def test_login_unverified(test_client):
    # zakładamy, że TempUser został utworzony, ale nie potwierdzony
    response = test_client.post('/auth/login', json={
        'email': 'student@gmail.com',
        'password': 'password123'
    })
    assert response.status_code == 401
    assert response.json['message'] == 'Verify your email.'
```

4. **Logowanie z poprawnymi poświadczeniami**
   Test: po potwierdzeniu konta wywołanie `POST /auth/login`. *Oczekiwane:* kod `200` i zwrot tokena JWT.

```
def test_login(test_client):
    # potwierdzenie konta
    temp_user = TempUser.query.filter_by(email='student@gmail.com').first()
    test_client.get(f'/auth/confirm/{temp_user.auth_key}')
    response = test_client.post('/auth/login', json={
        'email': 'student@gmail.com',
        'password': 'password123'
    })
    assert response.status_code == 200
    assert 'access_token' in response.json
```

5. **Dostęp do chronionych zasobów bez tokena**
   Test: żądanie np. `GET /api/teacher-list` bez nagłówka `Authorization`. *Oczekiwane:* kod `401` i komunikat `"Missing Authorization Header"`.

```
def test_get_teacher_list_fail(test_client):
    response = test_client.get('/api/teacher-list')
    assert response.status_code == 401
    assert response.json['msg'] == 'Missing Authorization Header'
```

6. **Dostęp do chronionych zasobów z poprawnym tokenem**
   Test: wysłanie żądania z ważnym tokenem studenta. *Oczekiwane:* kod `200` i struktura odpowiedzi zawierająca np. klucz `'teacher_list'`.

```
def test_get_teacher_list_success(test_client, login_student):
    headers = {'Authorization': f'Bearer {login_student}'}
    response = test_client.get('/api/teacher-list', headers=headers)
    assert response.status_code == 200
    assert 'teacher_list' in response.json
```

7. **Zarządzanie kalendarzem jako nie-teacher**
   Test: wysłanie `POST /api/calendar` z tokenem studenta. *Oczekiwane:* kod `403` i komunikat `"User must be a teacher"`.

```
def test_add_calendar_user_not_teacher(test_client, login_student):
    headers = {'Authorization': f'Bearer {login_student}'}
    response = test_client.post('/api/calendar', headers=headers, json={ ... })
    assert response.status_code == 403
    assert response.json['message'] == 'User must be a teacher'
```

8.3 Implementacja w kodzie

Poniżej fragmenty z pliku `auth_api.py` i dekoratora `jwt\_required()`, które realizują weryfikację:

```
// auth_api.py
@auth.route('/login', methods=['POST'])
def login():
    ...
    if user and user.check_password(password):
        access_token = user.generate_jwt()
        return jsonify({...}), 200
    else:
        return jsonify({'message': 'Invalid email or password.'}), 401

// decorators.py
def jwt_required(role=None):
    def decorator(fn):
        @wraps(fn)
        def wrapper(*args, **kwargs):
            verify_jwt_in_request()           # sprawdza nagłówek Authorization
            user = get_user_by_jwt()
            if role and user.role != role:
                return jsonify({'message': f'User must be a {role}'}), 403
            return fn(*args, **kwargs)
        return wrapper
    return decorator
```

Testy w `test_auth.py` i `test_api.py` bezpośrednio odzwierciedlają te mechanizmy: `assert status_code==401` dla braku nagłówka, `assert status_code==403` dla niewłaściwej roli, `assert status_code==200/400/404` dla walidacji pól i duplikatów.

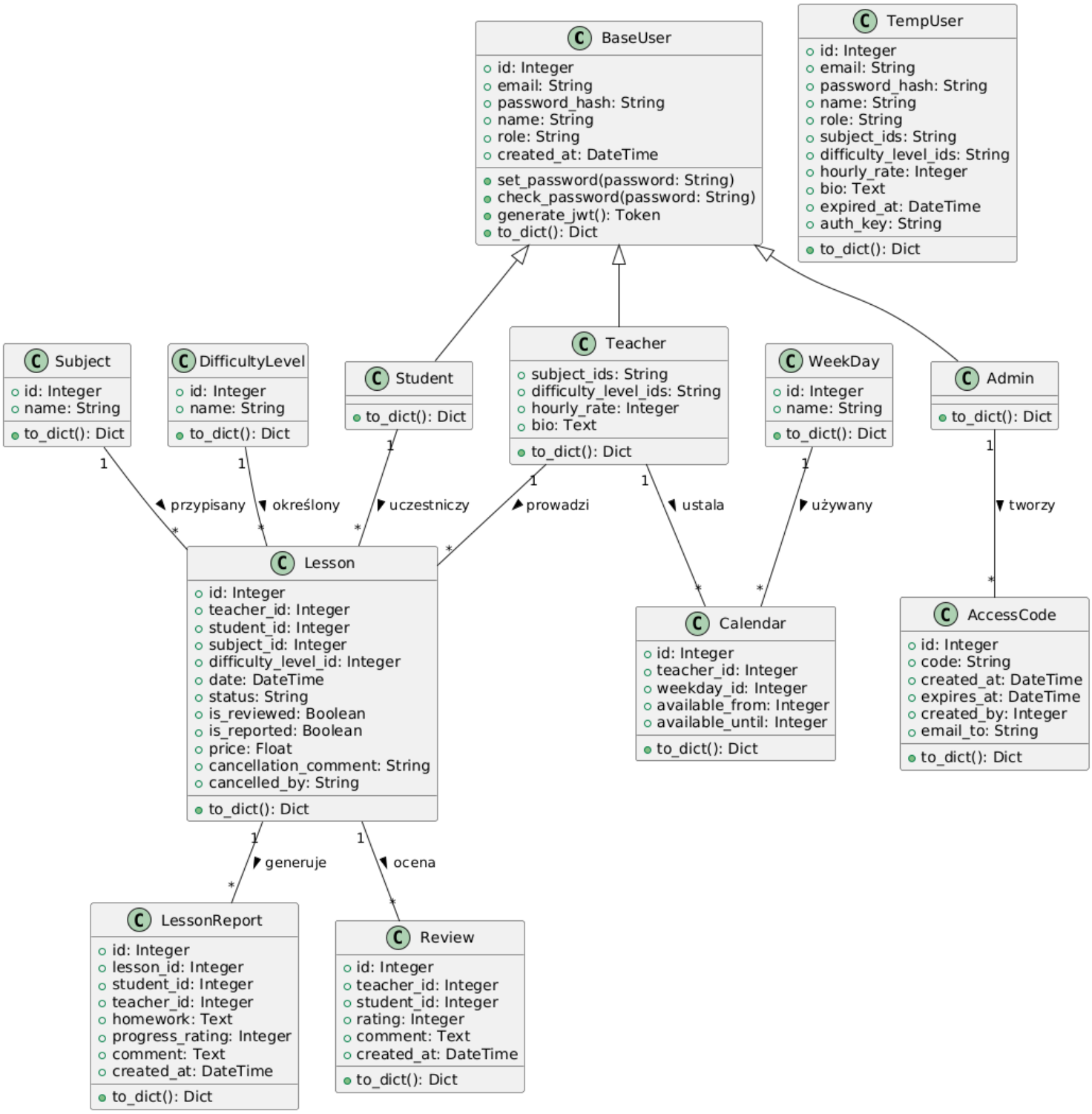
Dzięki temu mamy pełny zestaw testów, które:

- sprawdzają poprawność walidacji danych wejściowych,
- weryfikują mechanizmy haszowania i generowania JWT,
- potwierdzają ochronę zasobów przed nieautoryzowanym dostępem,
- oraz zabezpieczenia specyficzne dla ról (student vs. teacher).

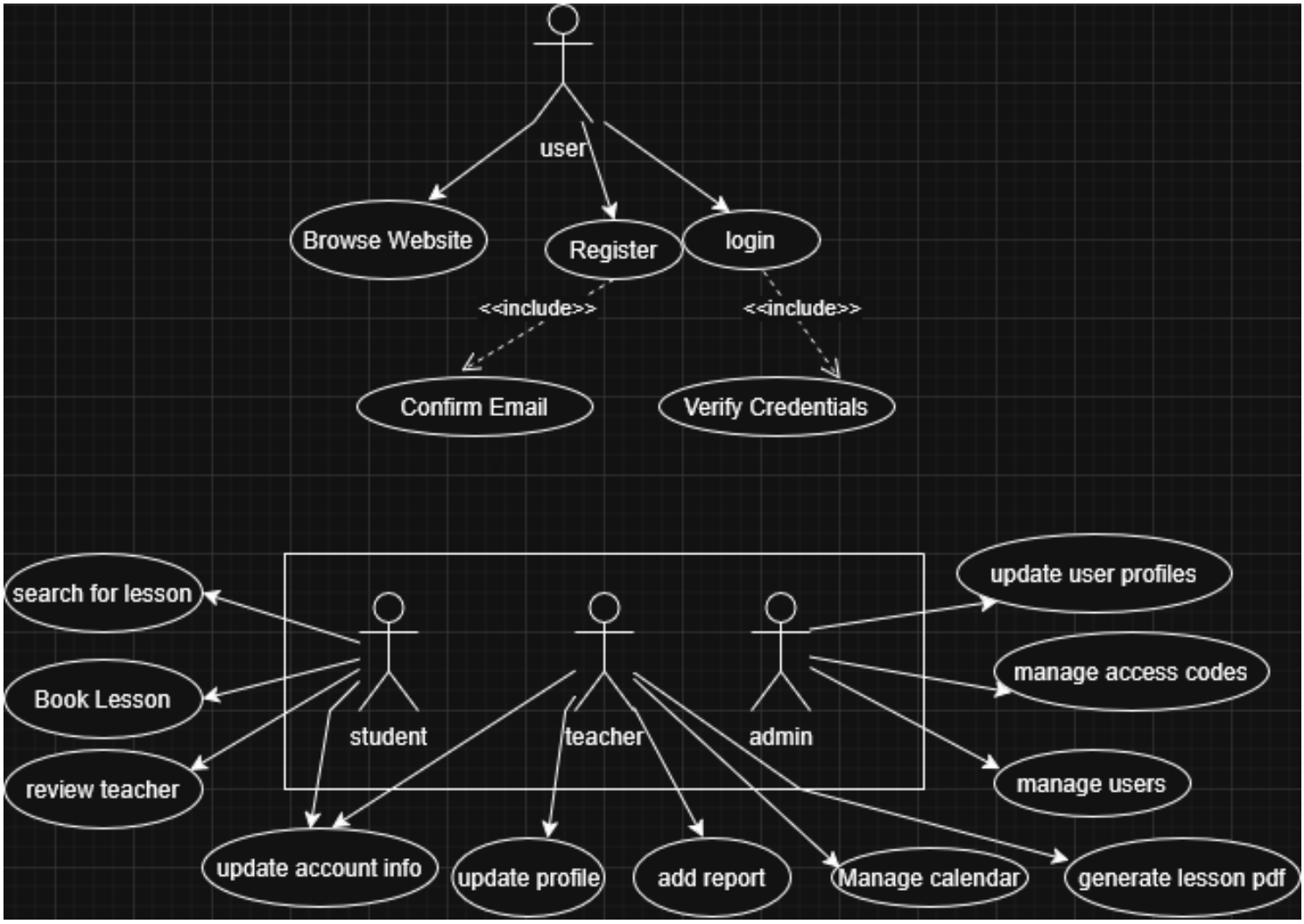
9 Diagramy

Diagramy przedstawiają główne modele używane w aplikacji i ich relacje. Kod do diagramów został napisany w PlantUML. Pliki graficzne znajdują się w katalogu projektu.

9.1 Diagram klas

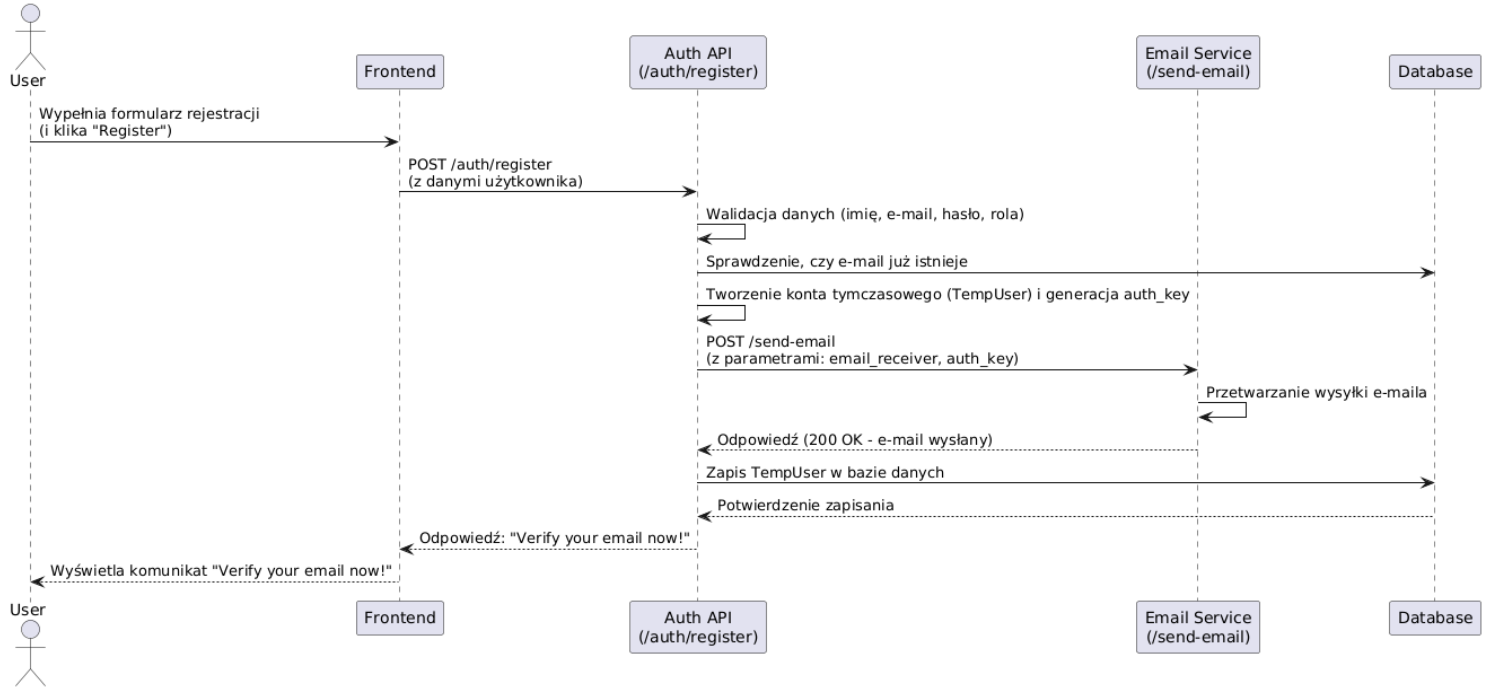


9.2 Diagram przypadków użycia z użytkownikiem

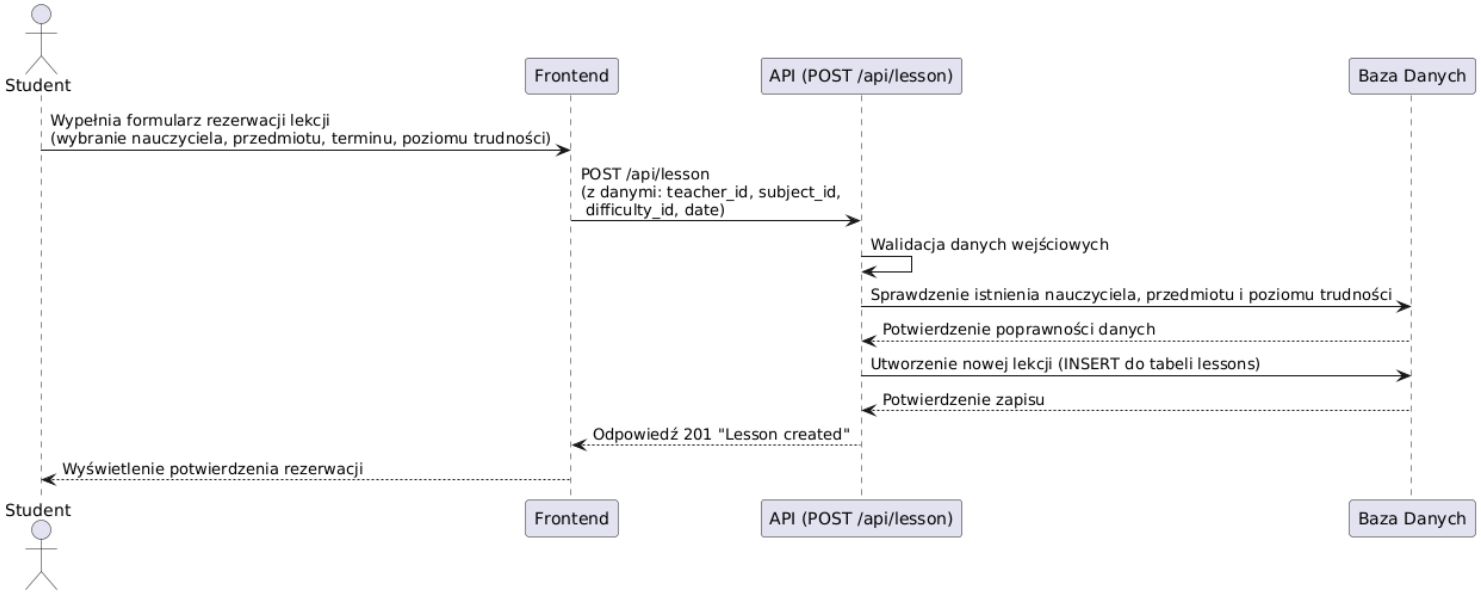


9.3 Diagramy sekwencji

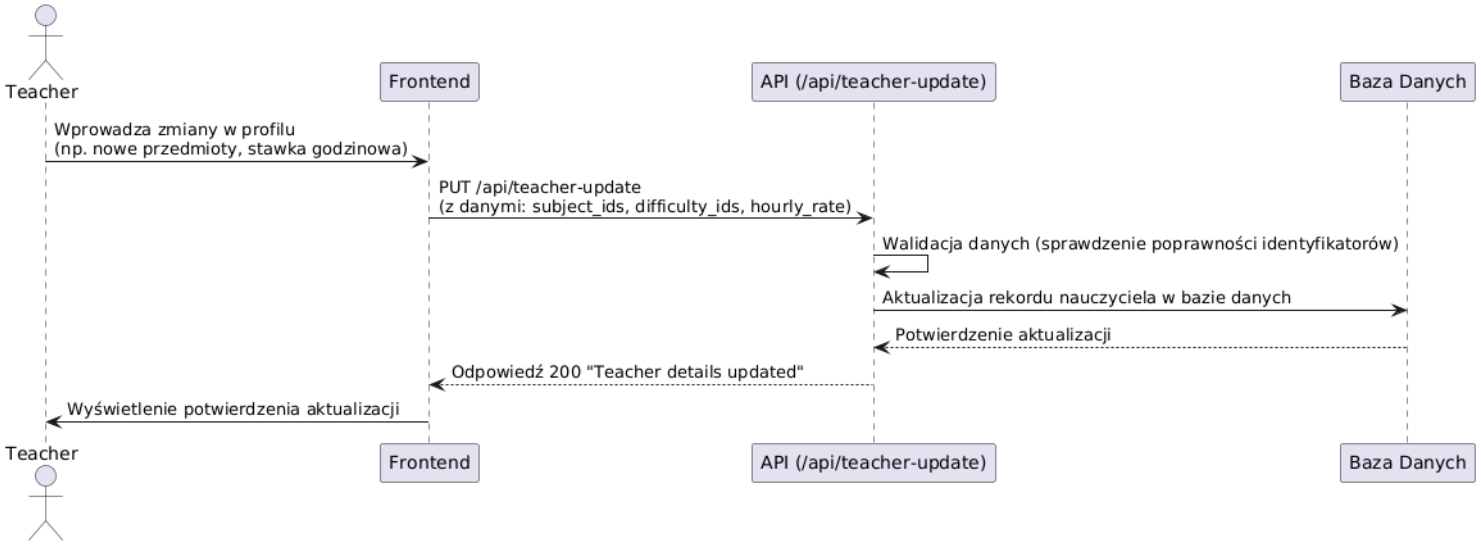
9.3.1 Rejestracja użytkownika



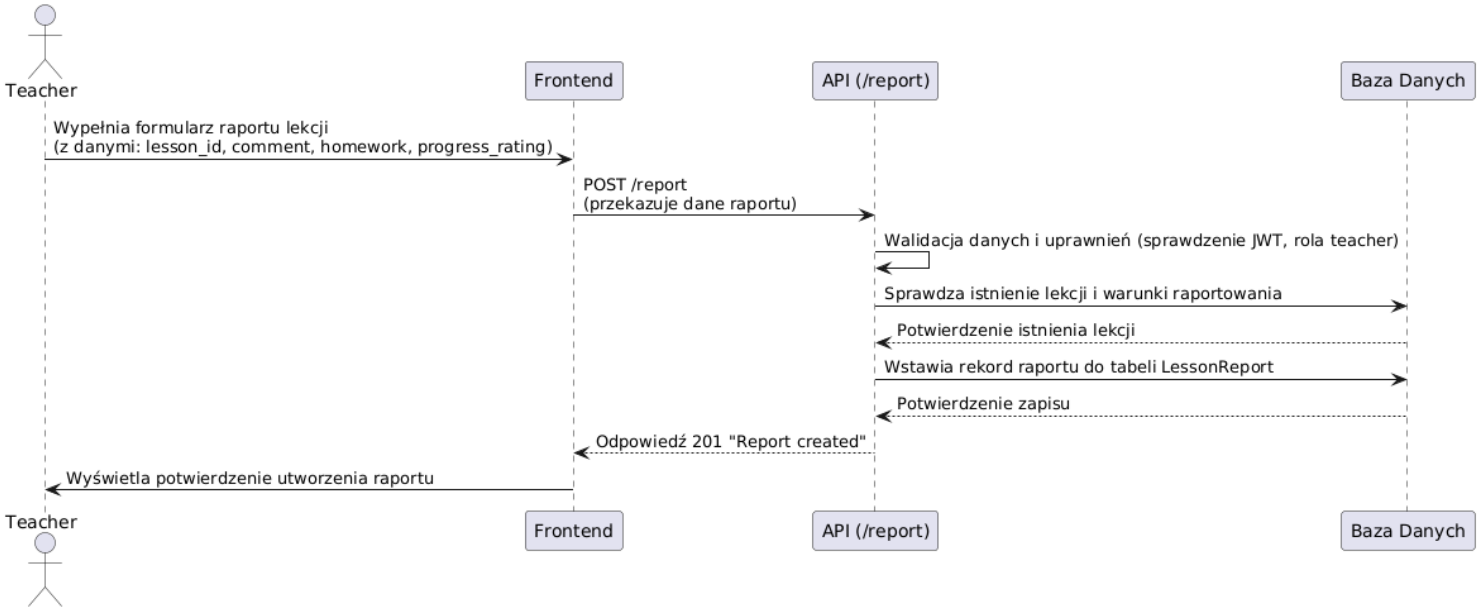
9.3.2 Rezerwacja lekcji przez studenta



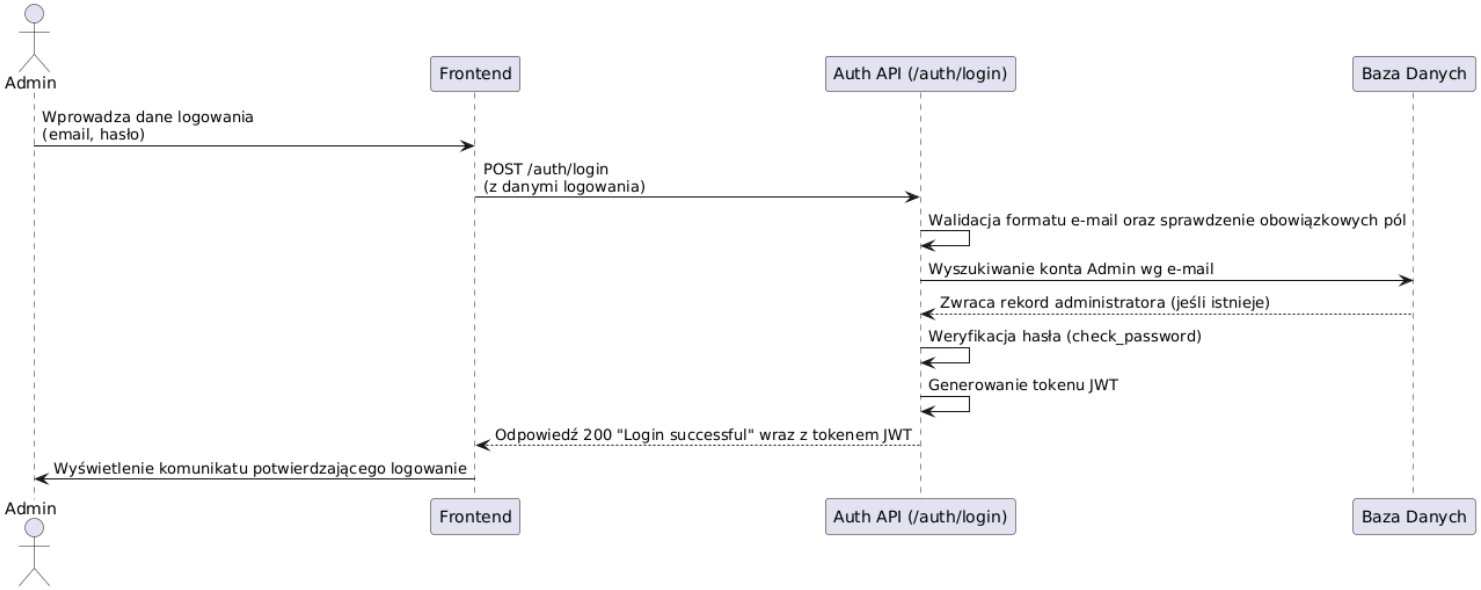
9.3.3 Aktualizacja profilu nauczyciela



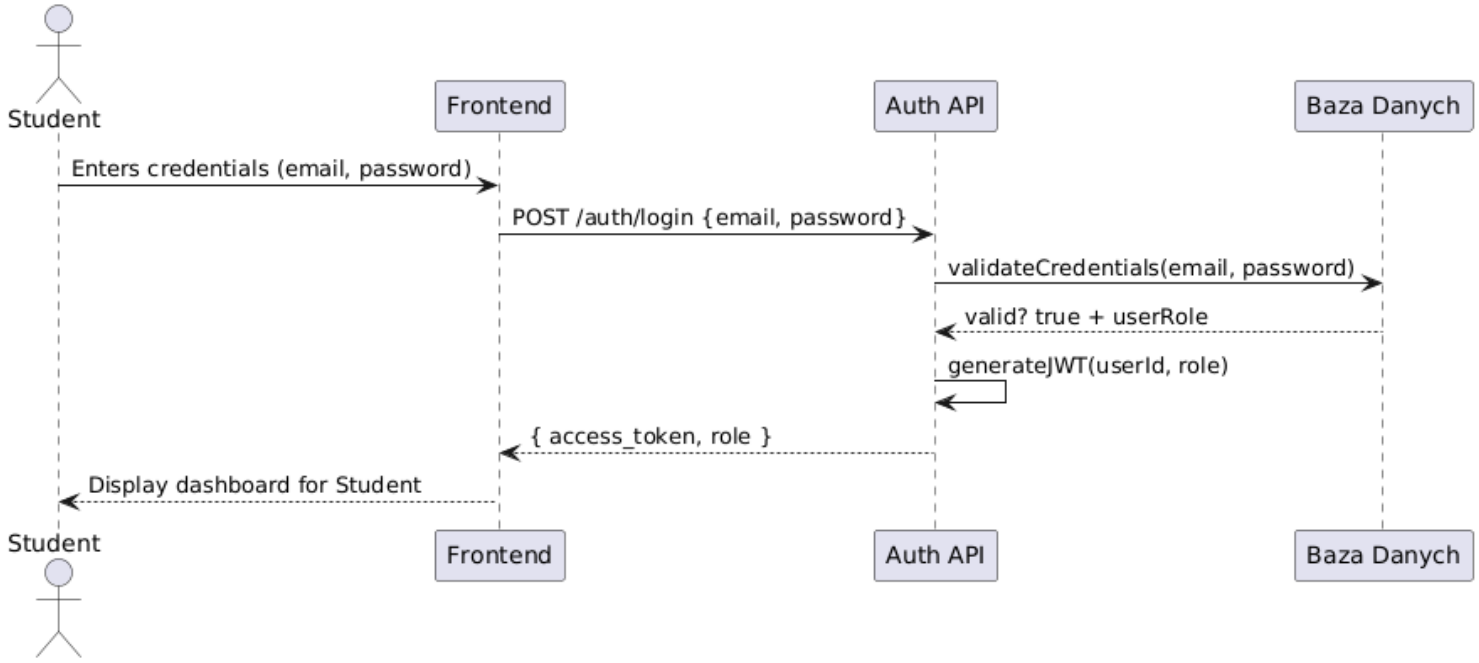
9.3.4 Dodawanie raportu przez nauczyciela



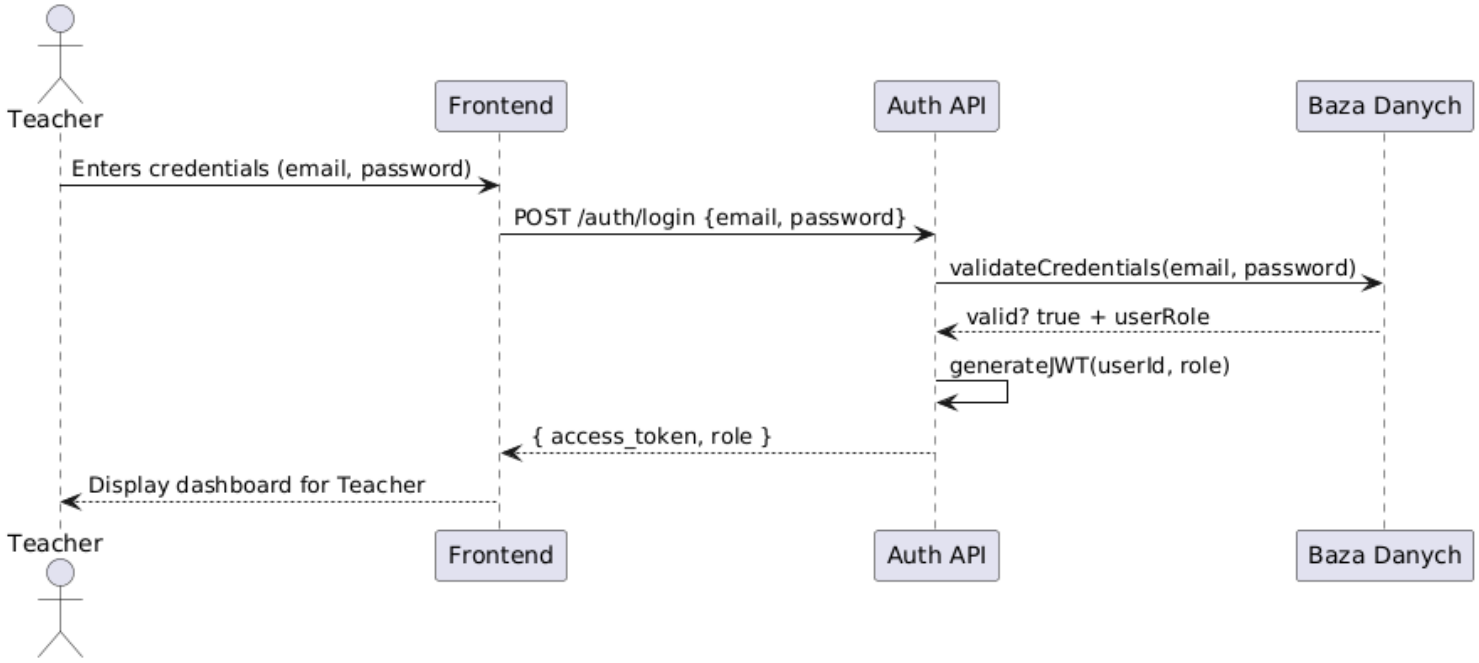
9.3.5 Logowanie administratora



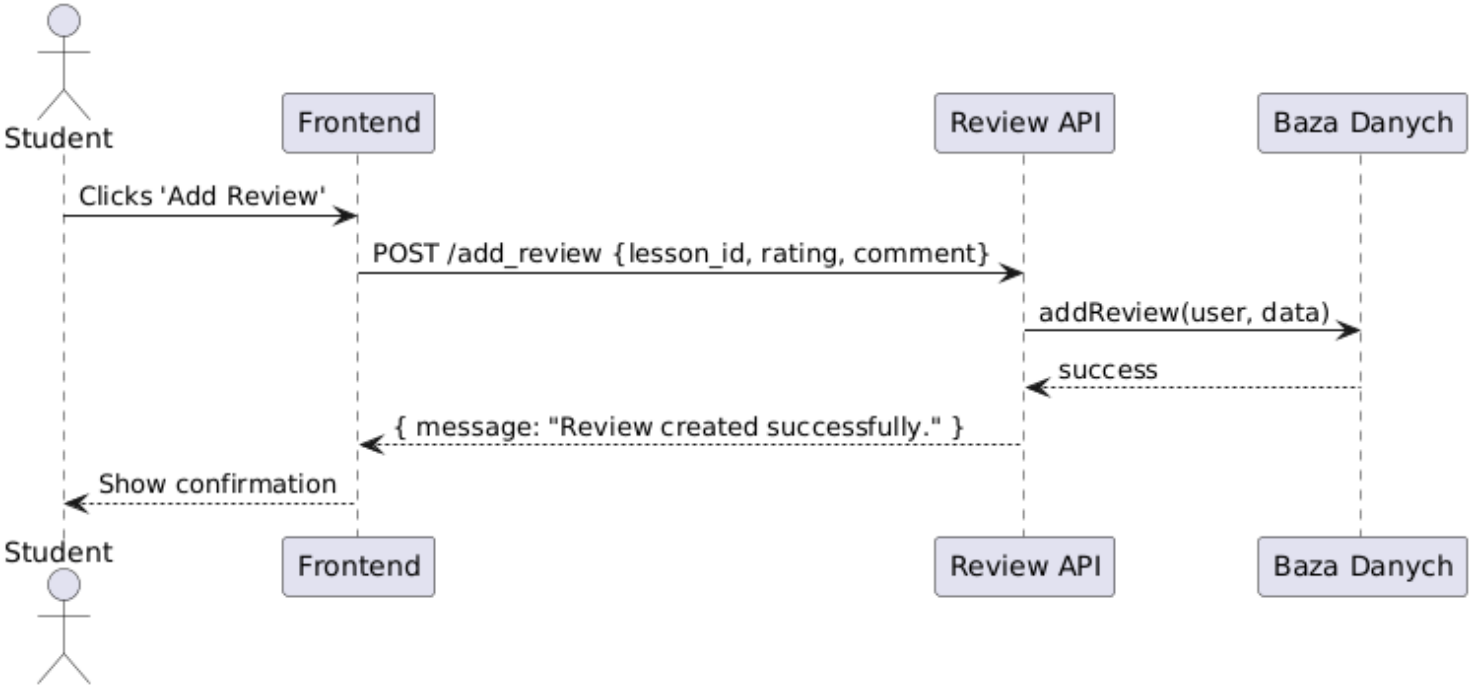
9.3.6 Logowanie studenta



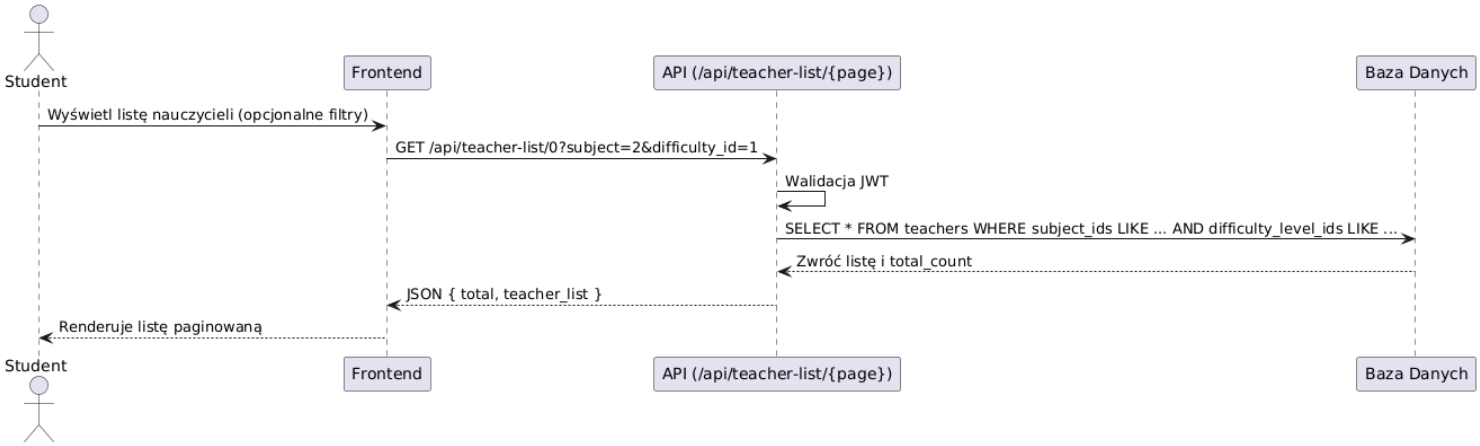
9.3.7 Logowanie nauczyciela



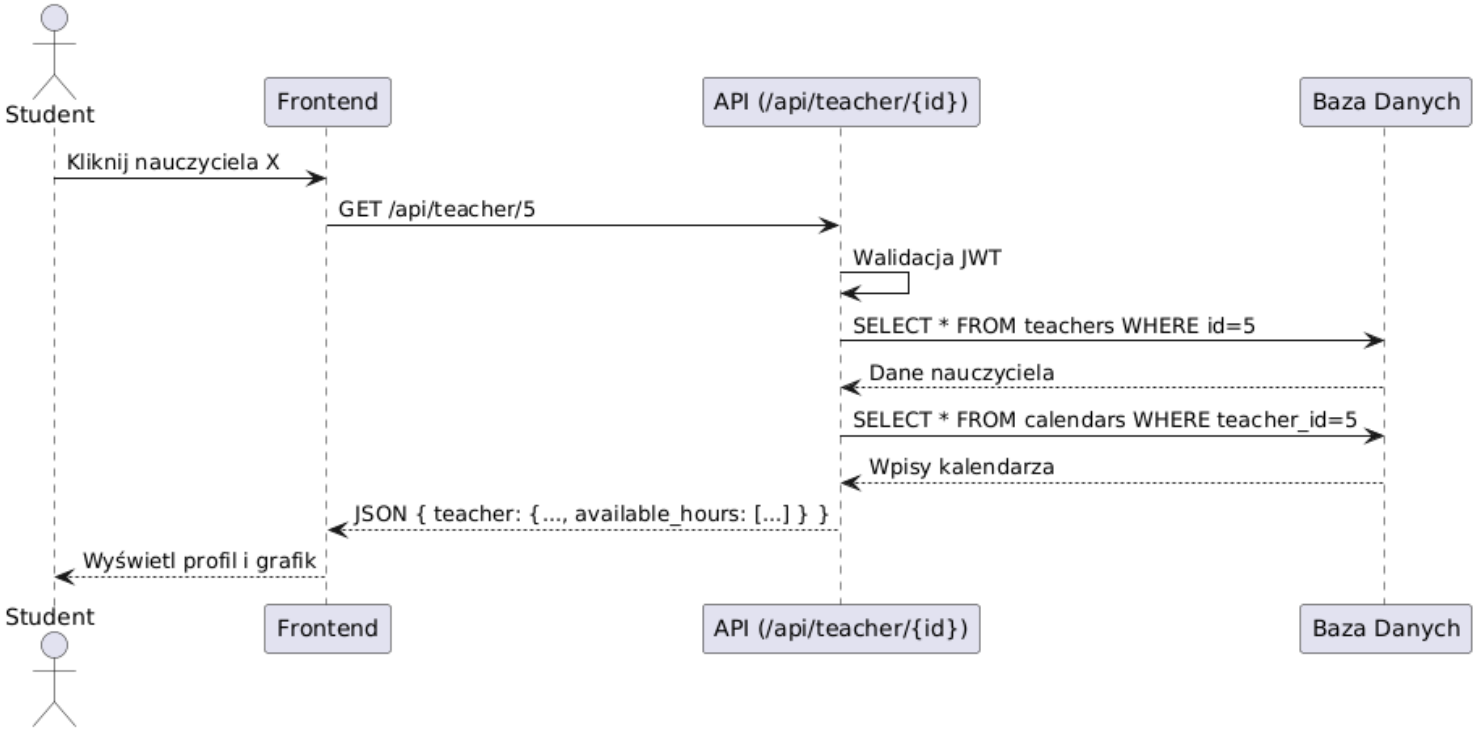
9.3.8 Dodawanie recenzji przez studenta



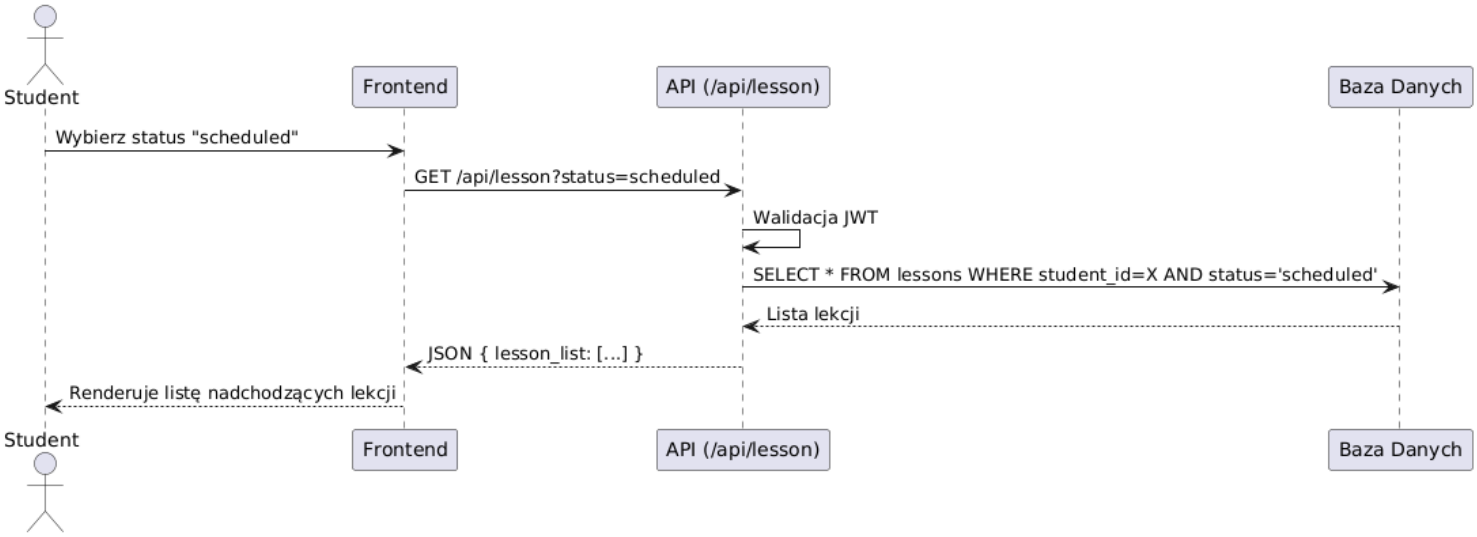
9.3.9 Przeglądanie listy nauczycieli



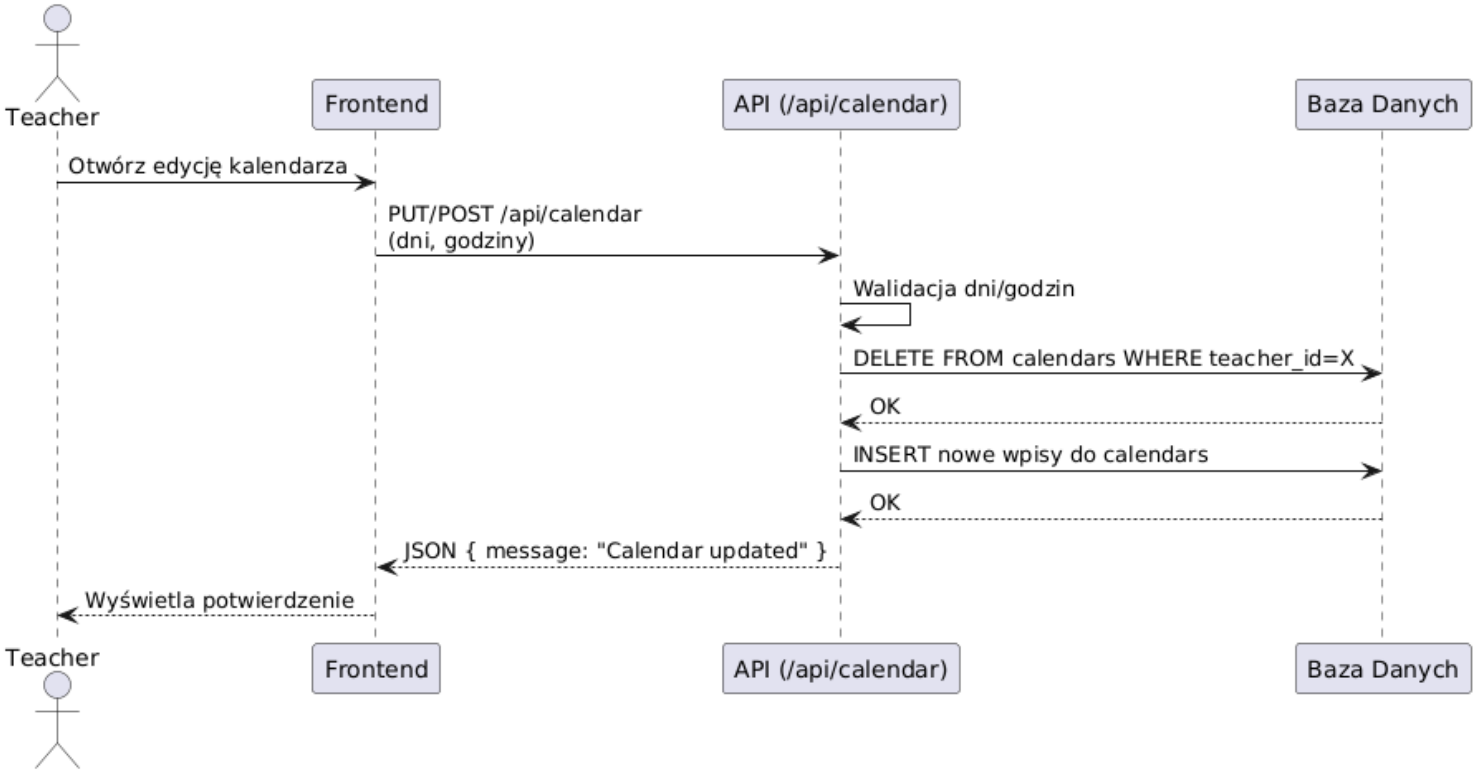
9.3.10 Pobieranie szczegółów nauczyciela



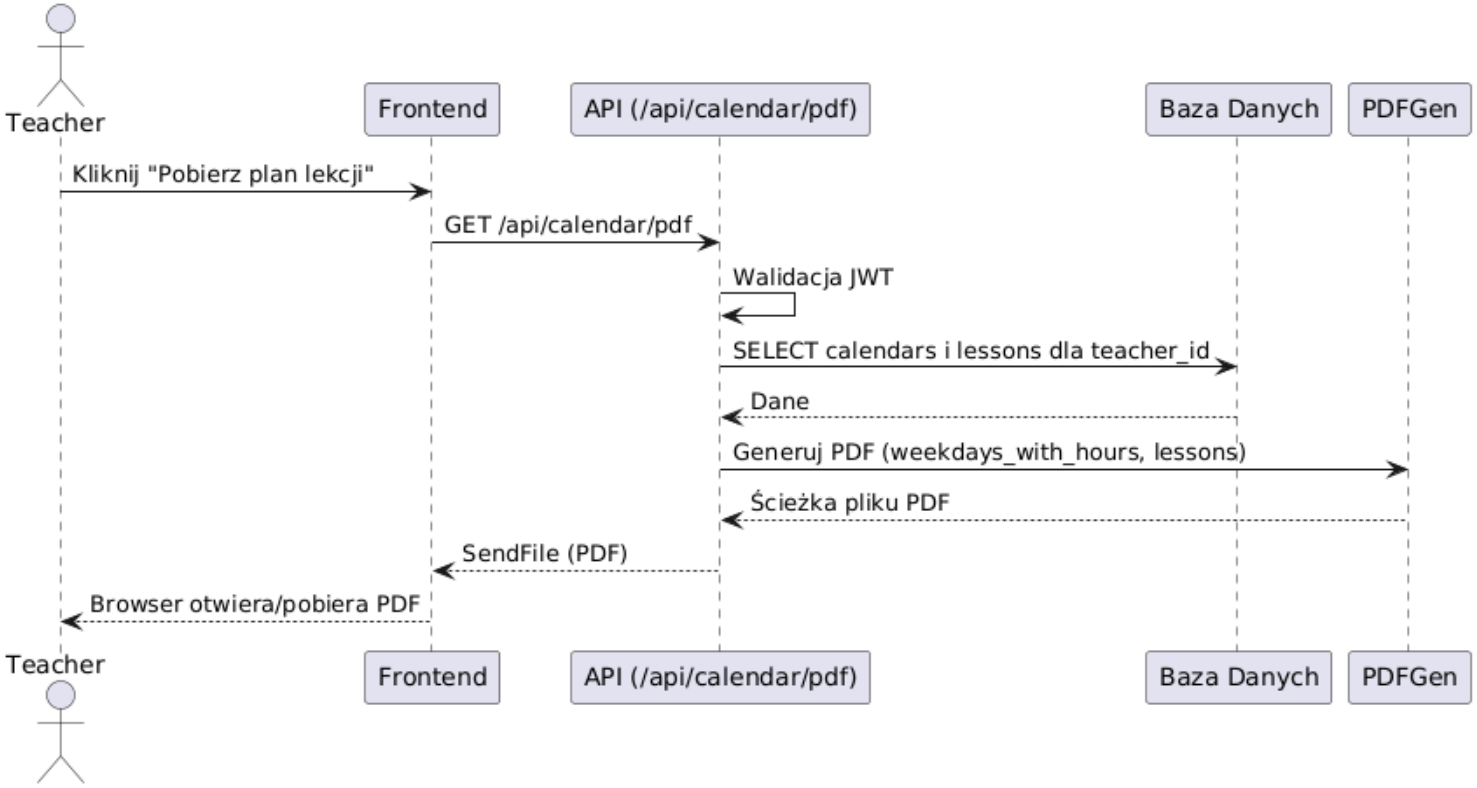
9.3.11 Przeglądanie lekcji ze statusem



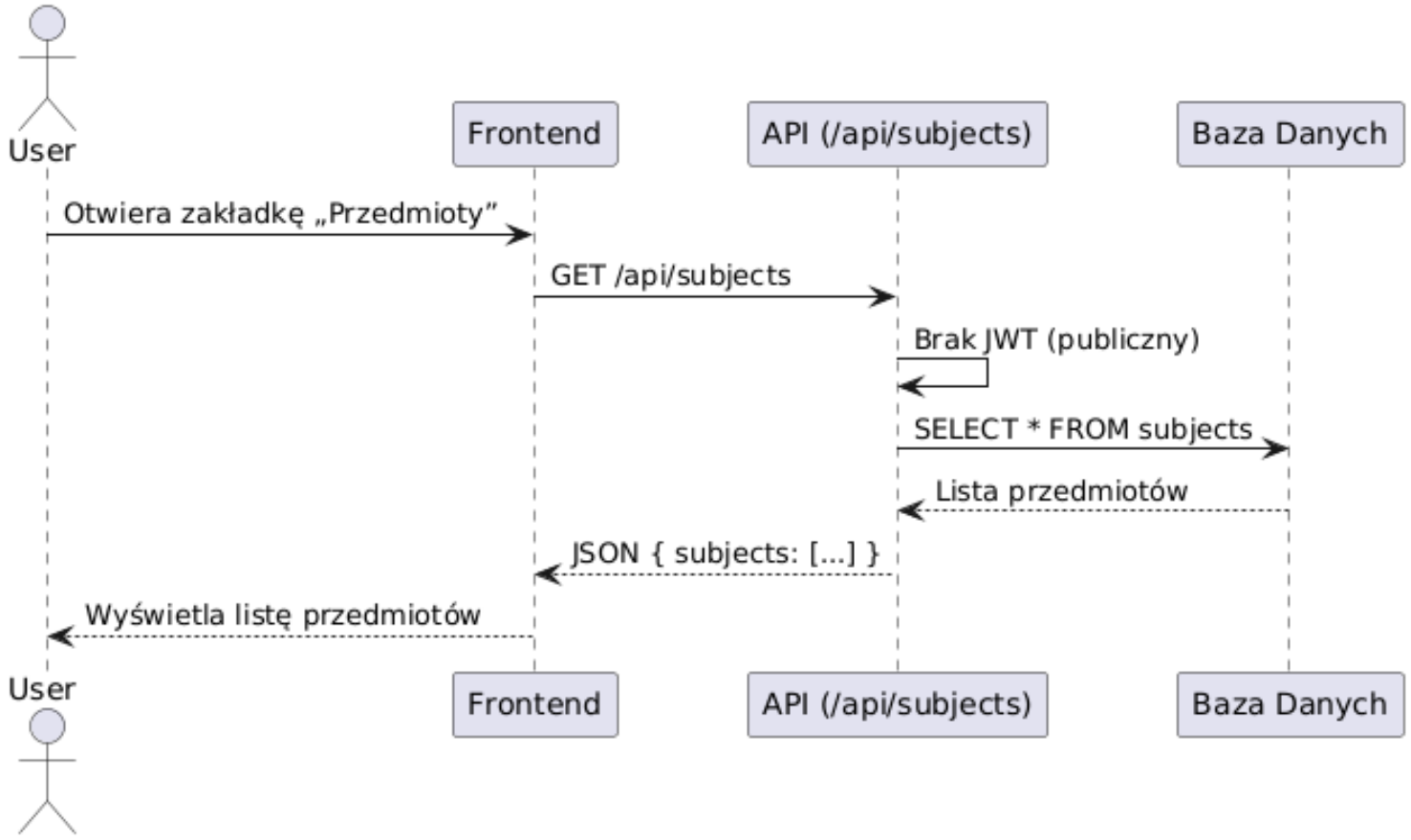
9.3.12 Zarządzanie kalendarzem



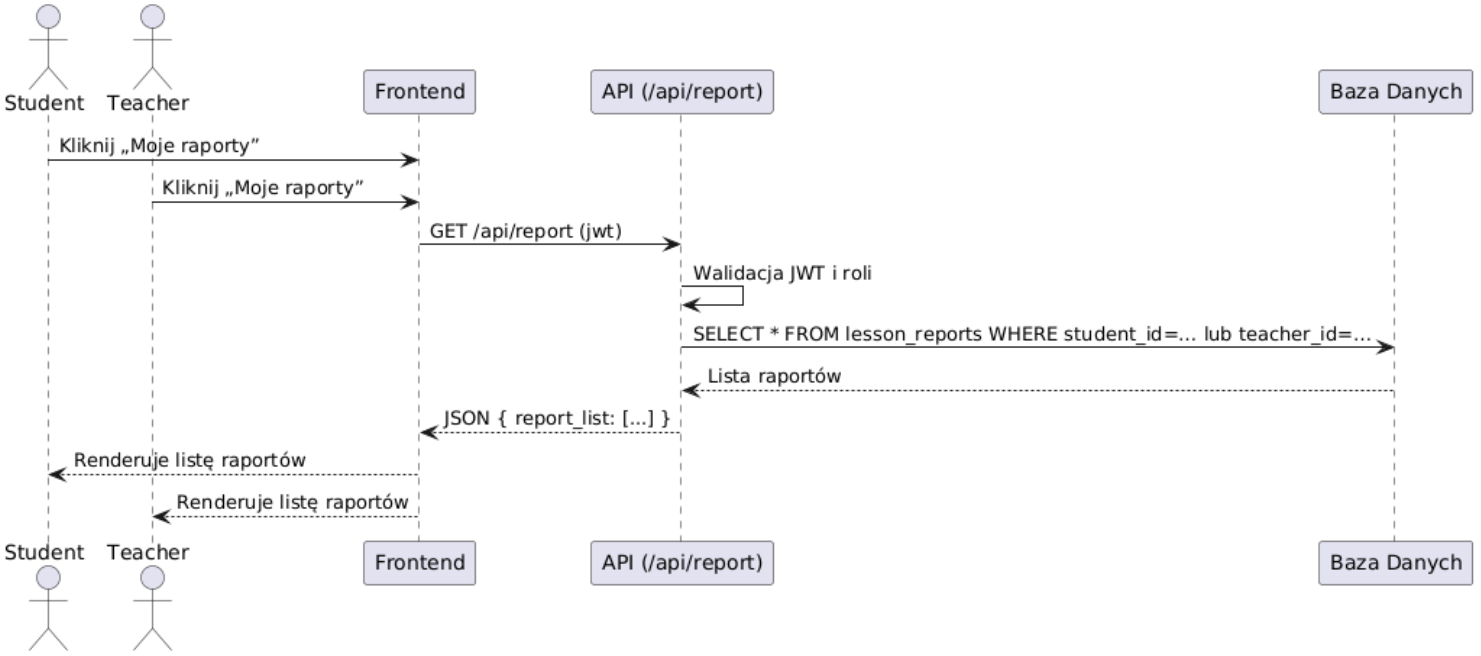
9.3.13 Generowanie planu lekcji PDF



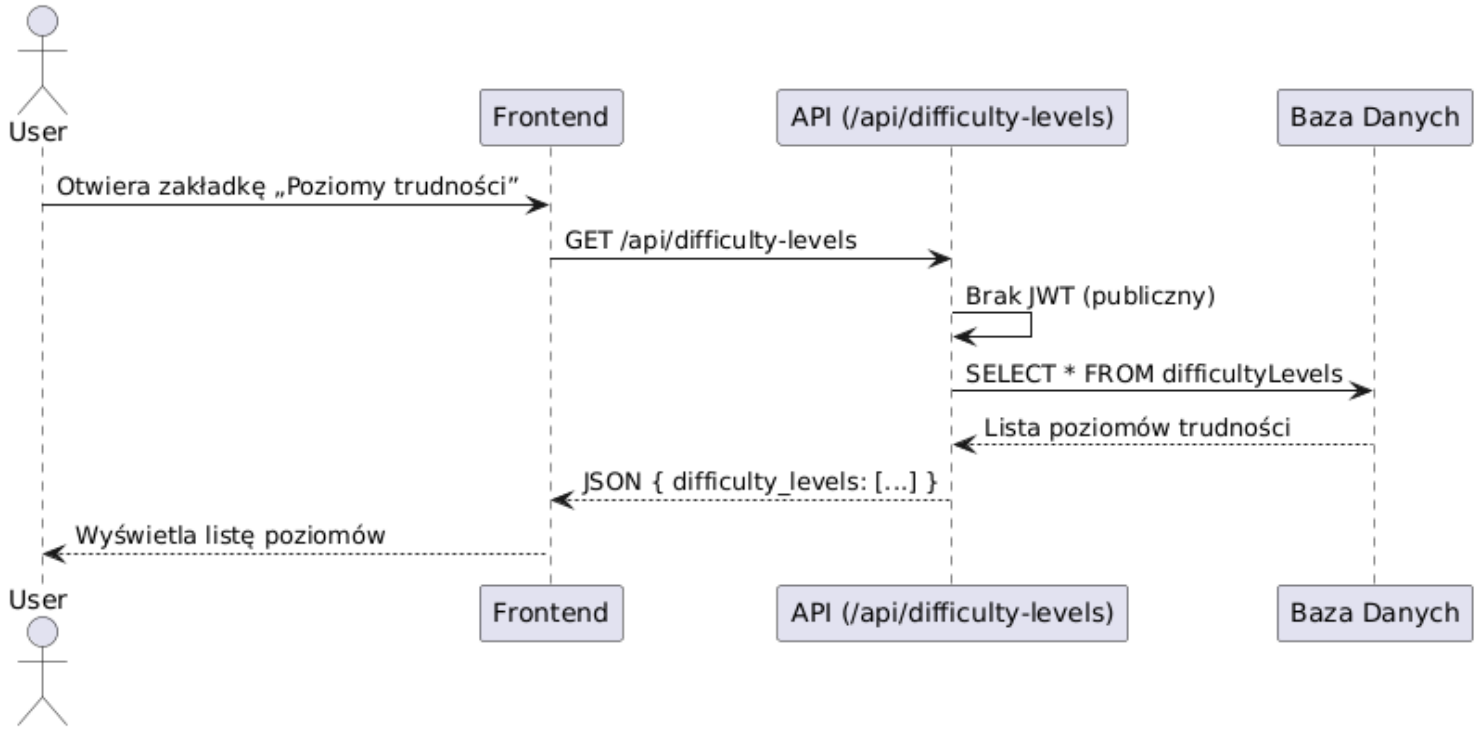
9.3.14 Przeglądanie listy przedmiotów



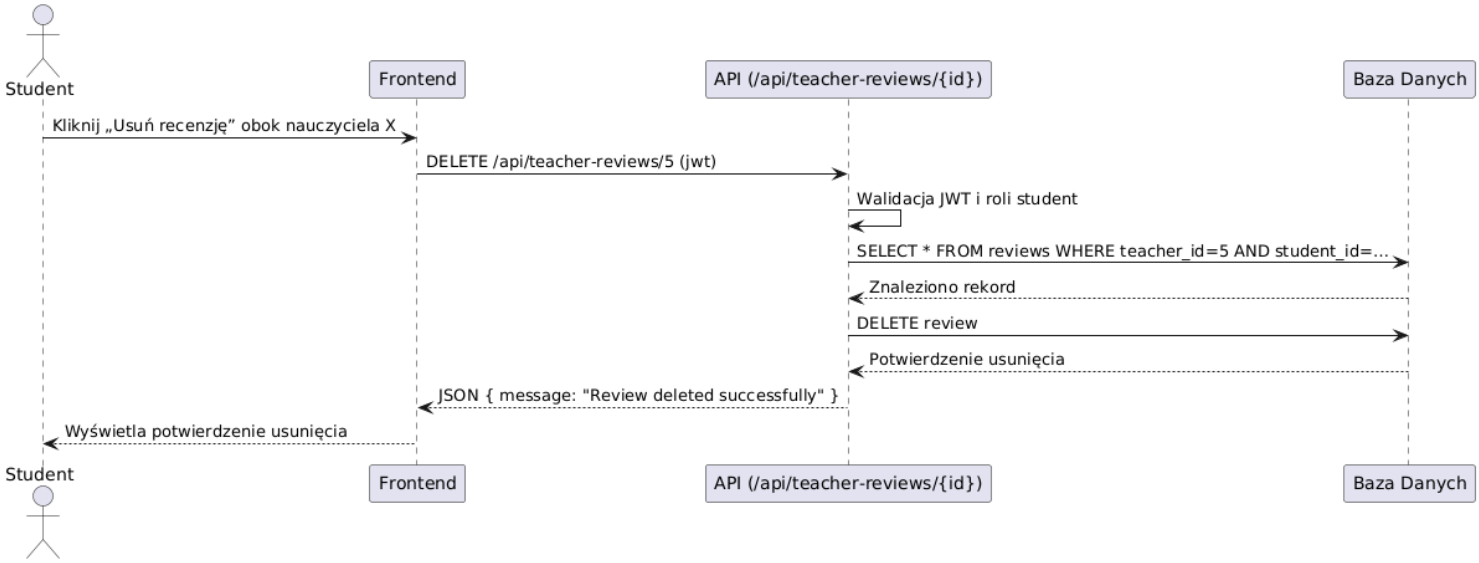
9.3.15 Przeglądanie własnych raportów



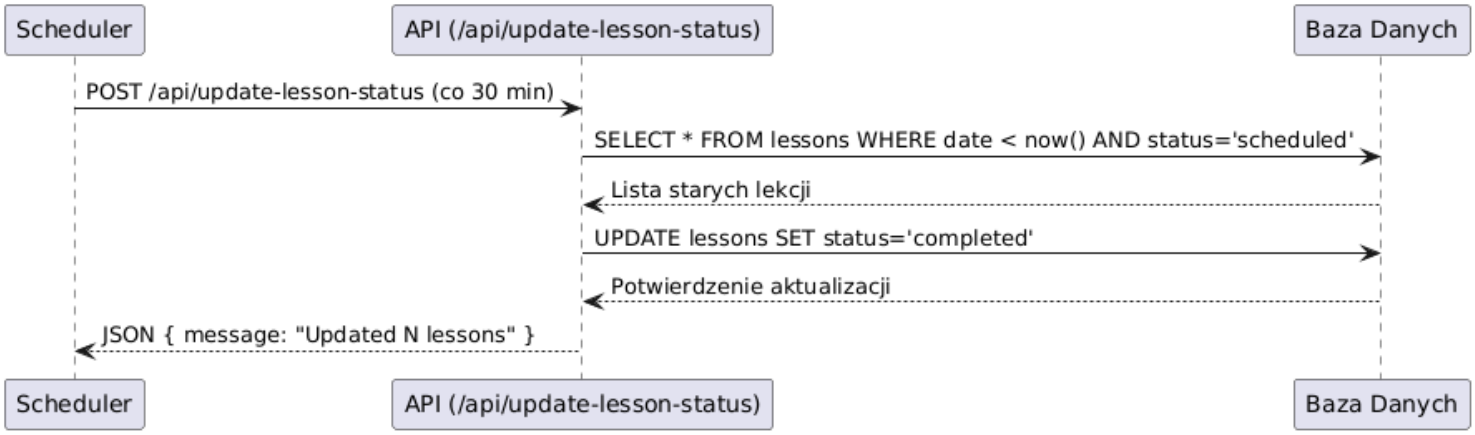
9.3.16 Przeglądanie poziomów trudności



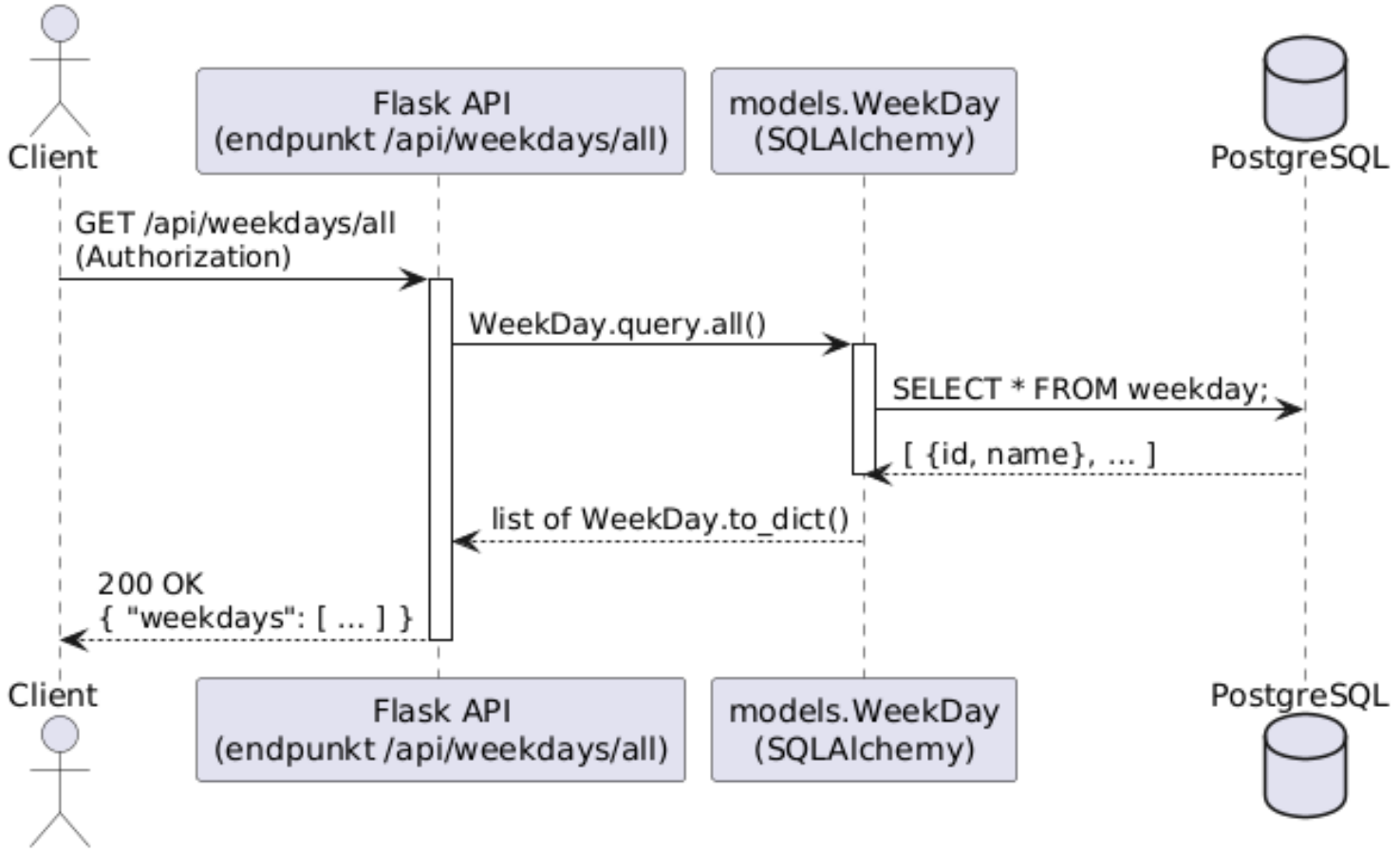
9.3.17 Przeglądanie i usuwanie recenzji



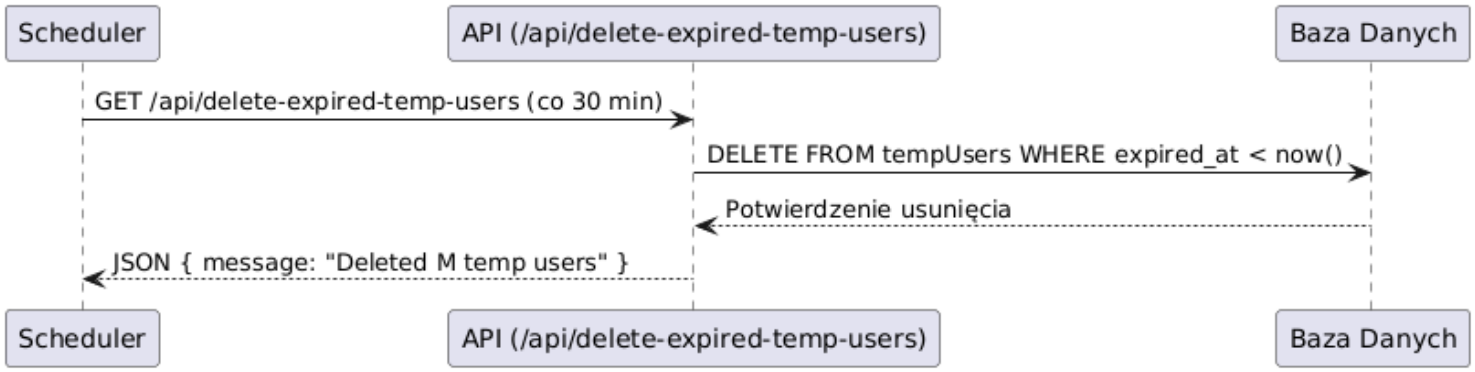
9.3.18 Aktualizacja statusów lekcji w tle



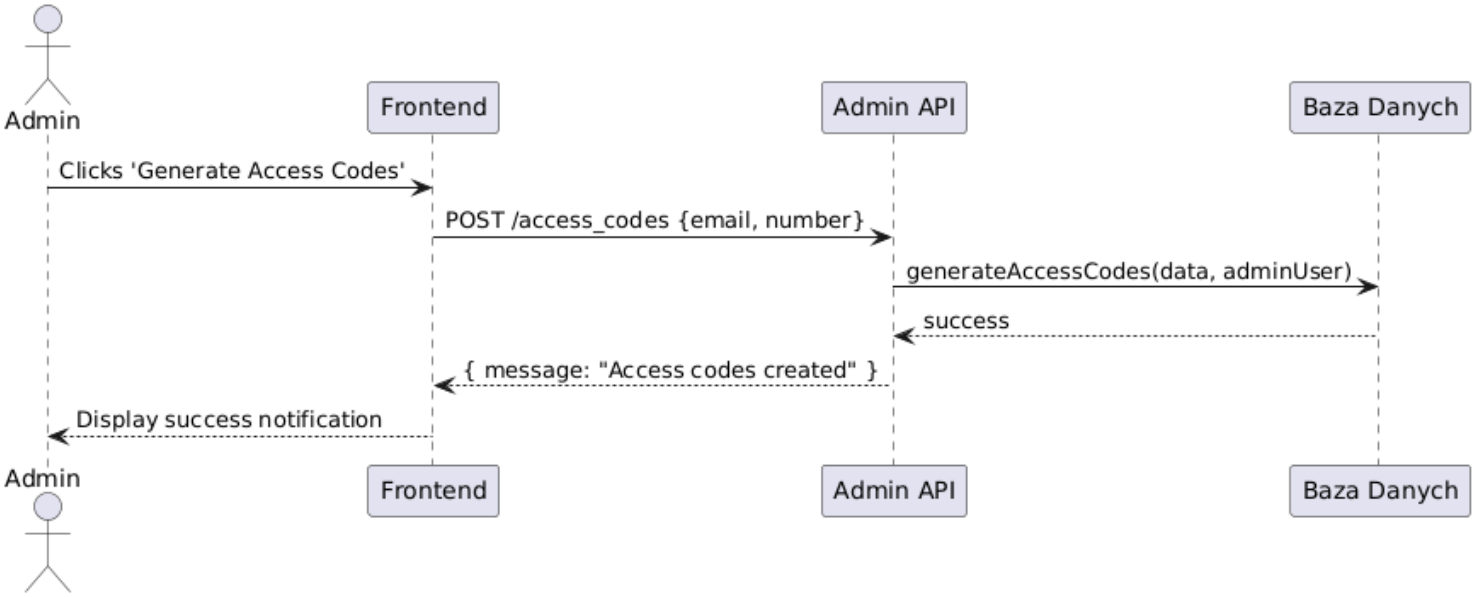
9.3.19 Przeglądanie listy dni tygodnia



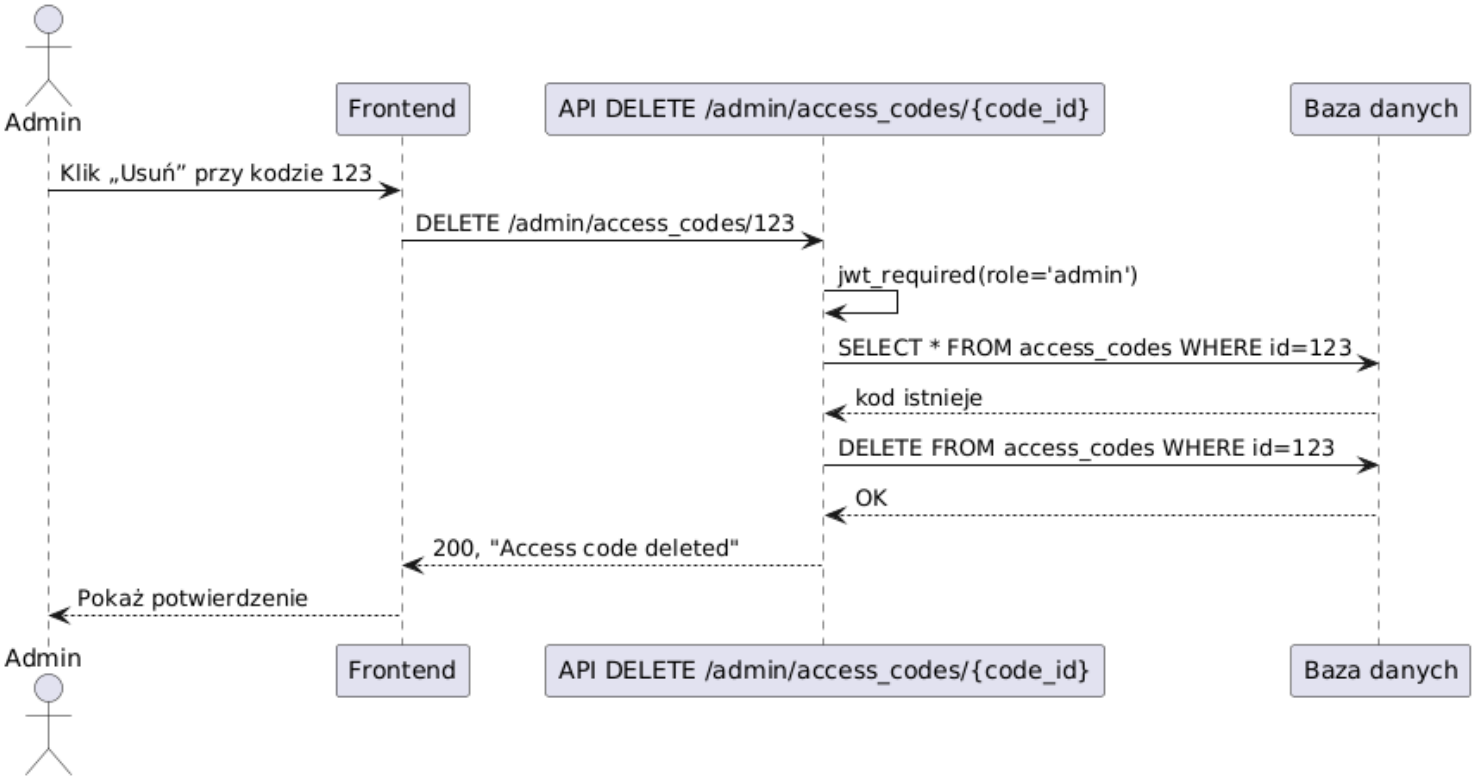
9.3.20 Usuwanie wygasłych kont tymczasowych



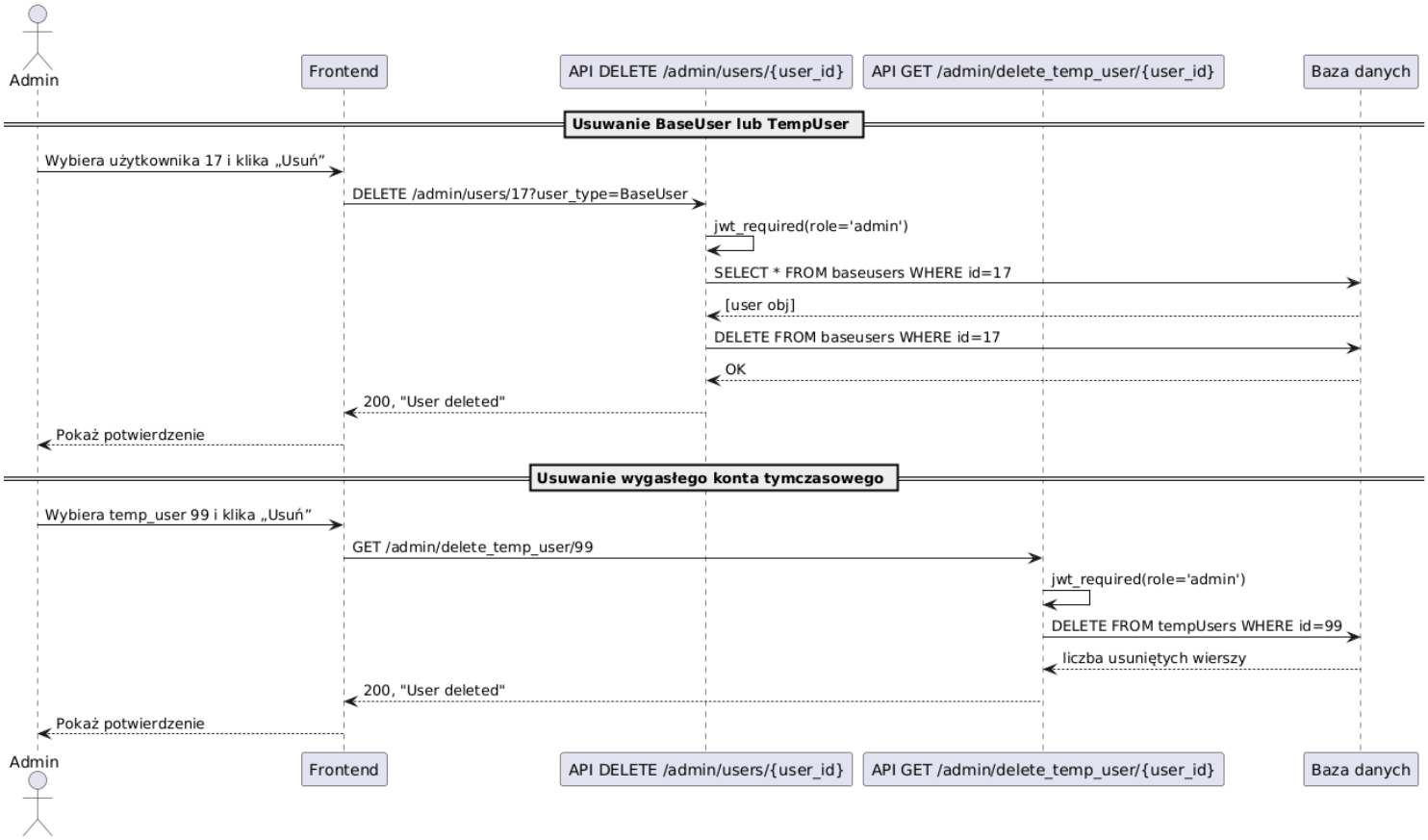
9.3.21 Generowanie kodów dostępu przez admina



9.3.22 Usuwanie pojedynczego kodu dostępu przez Admina

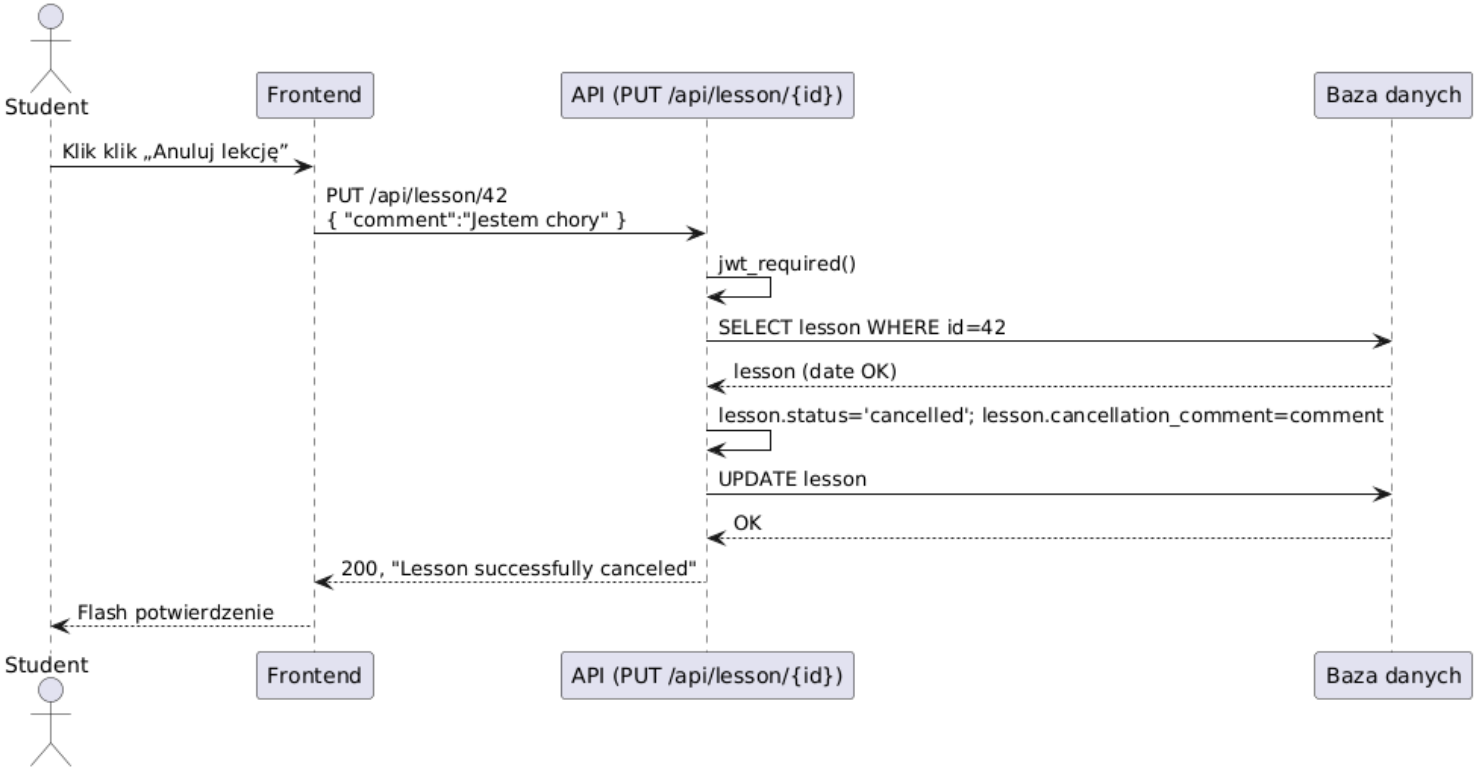


9.3.23 Usuwanie użytkownika lub tymczasowego konta przez Admina

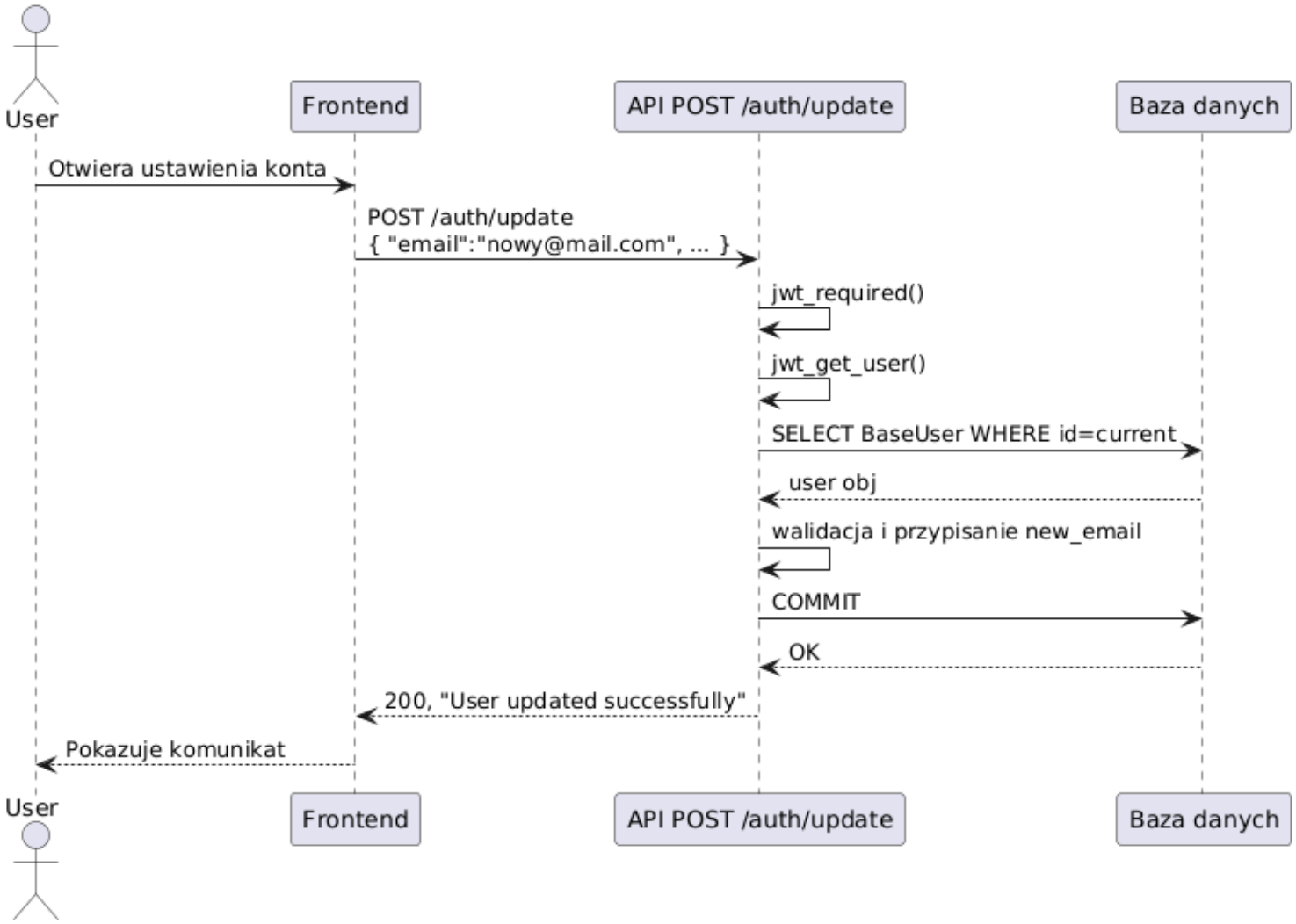


9.4 Diagramy dodatkowych funkcji

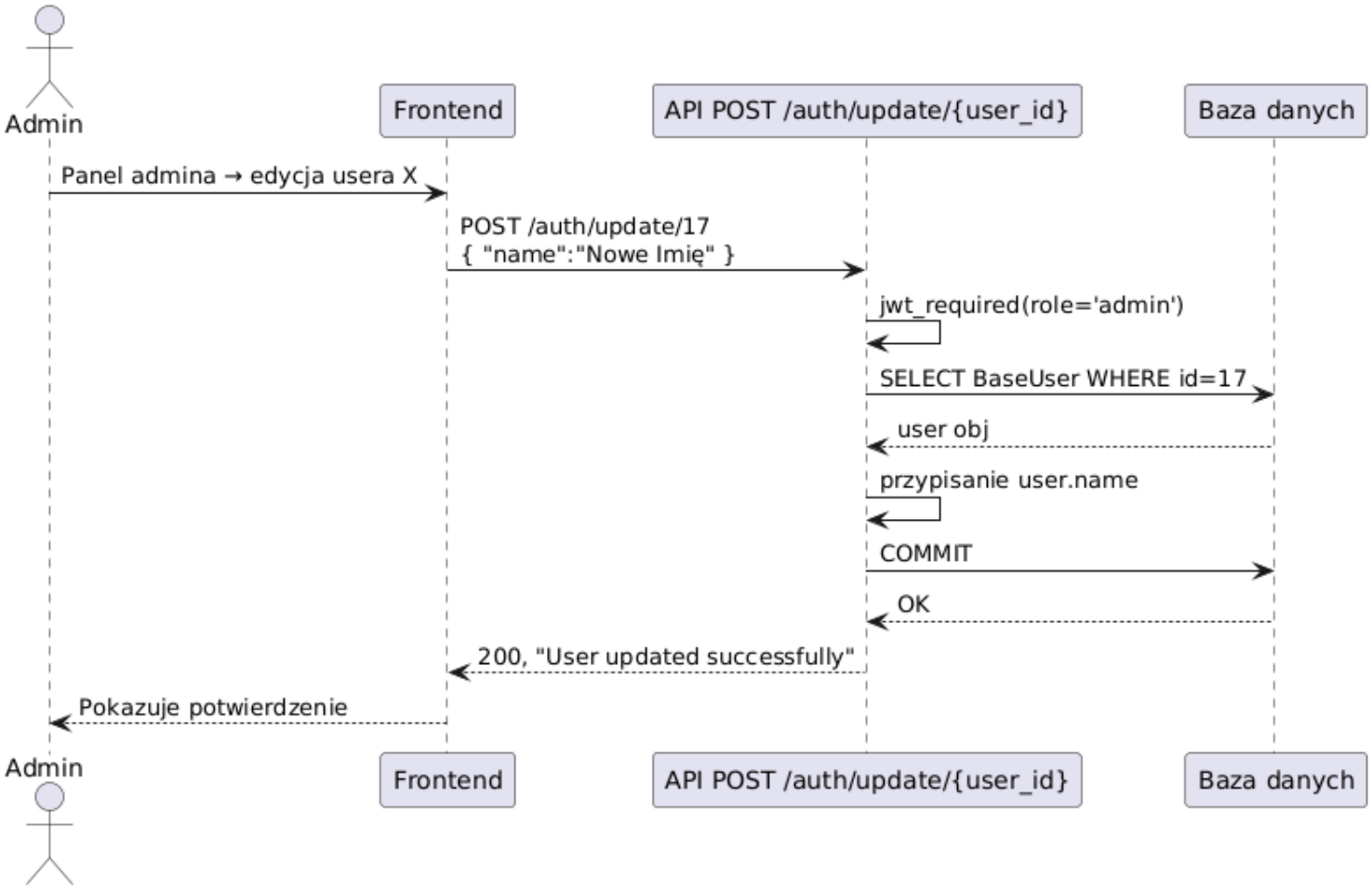
9.4.1 Anulowanie lekcji



9.4.2 Aktualizacja profilu – użytkownik



9.4.3 Aktualizacja profilu przez administratora



9.4.4 Pobieranie ustawień konta