

POLITECHNIKA ŚWIĘTOKRZYSKA

Wydział Elektrotechniki, Automatyki i Informatyki

Marek Supierz

Numer albumu: 092817

Andrzej Mysior

Numer albumu: 092768

25. Projekt prostego systemu IoT w Pythonie z REST API i autoryzacją JWT

Projekt: Bezpieczeństwo systemów IoT

Prowadzący: mgr inż. Bartłomiej Bugara

Katedra Systemów Informatycznych

Kielce 2025

Spis treści

1	Streszczenie	2
2	Wprowadzenie	2
3	Cele i zakres	2
4	Stos technologiczny	3
5	Architektura systemu	5
5.1	Widok logiczny	5
5.2	Diagramy	8
6	Model danych	10
7	Interfejs API	12
7.1	Autoryzacja i uwierzytelnianie	12
7.2	Przykłady wywołań	13
8	Klient GUI (PyQt5)	14
9	Testy jednostkowe	21
10	Instrukcja uruchomienia	23
10.1	Wymagania	23
10.2	Uruchomienie backendu	23
10.3	Uruchomienie GUI	23
11	Możliwe zagrożenia	23
12	Podsumowanie	24

1 Streszczenie

Krótki opis celu projektu, zastosowanej technologii (FastAPI, JWT, PyQt5, SQLite) oraz najważniejszych rezultatów (np. pokrycie testami, działające GUI, symulator telemetry).

2 Wprowadzenie

Rozwój Internetu Rzeczy (IoT) powoduje dynamiczny wzrost liczby urządzeń połączonych z siecią, które wymieniają dane w sposób automatyczny i często bez nadzoru użytkownika. Wraz z tą ewolucją rośnie znaczenie bezpieczeństwa, zarówno po stronie infrastruktury serwerowej, jak i samych urządzeń końcowych. Celem niniejszego projektu było zaprojektowanie oraz implementacja prostego, edukacyjnego systemu IoT, który pozwala zrozumieć podstawowe aspekty projektowania bezpiecznych interfejsów API, autoryzacji użytkowników i urządzeń, a także mechanizmów uwierzytelniania opartych o tokeny JWT.

Projekt został zrealizowany w ramach zajęć *Bezpieczeństwo systemów IoT* i obejmuje kompletny zestaw komponentów: backend REST API zbudowany w technologii FastAPI, prosty klient graficzny w PyQt5, bazę danych SQLite oraz symulator urządzeń generujących dane telemetryczne.

3 Cele i zakres

Głównym celem projektu było stworzenie kompletnego, lecz uproszczonego środowiska IoT, które demonstruje praktyczne zastosowanie zasad bezpieczeństwa w architekturze systemów rozproszonych. System miał umożliwić:

- bezpieczne uwierzytelnianie i autoryzację użytkowników oraz urządzeń z wykorzystaniem tokenów JWT;
- tworzenie i zarządzanie urządzeniami IoT poprzez panel administracyjny;
- odbieranie i przechowywanie danych telemetrycznych wysyłanych przez symulator urządzeń;
- kontrolę dostępu do zasobów na podstawie ról i typów tokenów;
- wizualizację danych oraz konfigurację urządzeń w aplikacji klienckiej (PyQt5).

Zakres projektu obejmował implementację warstwy serwerowej (API, logika biznesowa, bezpieczeństwo, warstwa danych), klienta GUI oraz symulatora urządzeń. Dodatkowo przygotowano zestaw testów automatycznych w frameworku `pytest`, które pozwalają zweryfikować poprawność działania kluczowych komponentów systemu.

Projekt nie skupia się na aspektach sprzętowych IoT, lecz koncentruje się na modelowaniu komunikacji, autoryzacji i zarządzania danymi w bezpiecznym środowisku aplikacyjnym.

4 Stos technologiczny

Stos technologiczny projektu dobrano z naciskiem na prostotę uruchomienia, przejrzystość architektury i możliwość stopniowej rozbudowy. Warstwa serwerowa oparta jest na frameworku FastAPI. Wybrano FastAPI ze względu na wbudowaną obsługę typów i walidacji danych poprzez Pydantic, co umożliwia deklaratywne określenie schematów wejściowych i wyjściowych oraz redukuje liczbę ręcznych walidacji w kodzie. FastAPI automatycznie generuje specyfikację OpenAPI (Swagger / ReDoc), co ułatwia testowanie i dokumentowanie punktów końcowych bez konieczności sięgania po zewnętrzne narzędzia. W porównaniu do Flask, który jest prostym i popularnym mikroframeworkiem, FastAPI dostarcza te funkcjonalności natywnie; Flask wymagałby ręcznego dobierania rozszerzeń do walidacji, dokumentacji i obsługi asynchronicznej, co zwiększa nakład pracy i potencjalne źródła niezgodności. Z tego względu FastAPI uznano za bardziej adekwatne do celów projektu: demonstracji dobrych praktyk i łatwego testowania API.

Do uruchamiania aplikacji zastosowano serwer Uvicorn, który zapewnia wydajną obsługę asynchronicznych połączeń oraz wygodny tryb developerski z automatycznym przeładowaniem kodu, co skraca cykl weryfikacji zmian.

Warstwę przechowywania danych zrealizowano przy użyciu SQLite. Decyzja o zastosowaniu SQLite wynika z pragmatyki środowiska dydaktycznego: brak konieczności instalacji oddzielnego serwera bazy danych, przenośność pojedynczego pliku bazy oraz minimalne wymagania konfiguracyjne upraszczają uruchomienie i dystrybucję projektu. Alternatywne rozwiązania, takie jak PostgreSQL czy MySQL, oferują zaawansowane mechanizmy transakcyjne, lepsze możliwości równoległych zapisów i skalowania, lecz ich wdrożenie wiązałoby się z większą złożonością infrastruktury i podniesieniem progu wejścia dla osób uruchamiających projekt lokalnie. Wybór SQLite nie wyklucza migracji do systemu produkcyjnego; zastosowanie warstwy ORM umożliwia przejście na PostgreSQL.

Dostęp do bazy zrealizowano z wykorzystaniem SQLAlchemy jako warstwy ORM. Zastosowanie ORM pozwala operować na encjach w postaci klas Pythona, co upraszcza modelowanie relacji i logikę przetwarzania danych oraz ogranicza konieczność ręcznego pisania zapytań SQL. SQLAlchemy zapewnia jednocześnie elastyczność i możliwość migracji do innych systemów bazodanowych, a także zmniejsza ryzyko błędów związanych z bezpośrednim konstruowaniem zapytań (np. SQL injection). W ocenie projektu użycie ORM stanowi kompromis pomiędzy wygodą programistyczną a potrzebą zachowania kontroli nad wydajnością w miejscach krytycznych.

Interfejs kliencki zrealizowano w oparciu o PyQt5. Wybrano rozwiązanie desktopowe, aby umożliwić szybkie przygotowanie funkcjonalnego GUI służącego do zarządzania urządzeniami, przeglądania danych telemetrycznych i testowania mechanizmów autoryzacji bez tworzenia i utrzymywania osobnego środowiska frontendowego. Alternatywą byłby frontend webowy (React, Vue itp.), który ułatwia udostępnienie interfejsu wieloużytkownikowego i dostęp przez przeglądarkę, lecz wymagałby oddzielnego stacku technologicznego i dodatkowych narzędzi do budowy i serwowania zasobów. W kontekście projektu dydaktycznego, gdzie priorytetem była prostota wdrożenia oraz demonstracja działania API, PyQt5 okazał się praktycznym wyborem.

Mechanizmy bezpieczeństwa oparto na JSON Web Tokens (JWT) oraz bezpiecznym przechowywaniu haseł. W projekcie zastosowano rozdzielenie tokenów dostępu i tokenów odświeżających oraz odrębne polityki dla tokenów użytkowników i tokenów urządzeń, co umożliwia precyzyjne określenie uprawnień i czasu życia sesji. Hasła przechowywane są w formie hashy przy użyciu sprawdzonych algorytmów (np. bcrypt), a walidacja i filtrowanie danych wejściowych realizowane są na poziomie schematów Pydantic. W architekturze przewidziano również rejestr zdarzeń bezpieczeństwa (audit log) do monitorowania nieautoryzowanych prób dostępu i incydentów, co zwiększa możliwość późniejszej analizy i reakcji.

W zakresie testowania i zapewnienia jakości przyjęto framework pytest do uruchamiania testów jednostkowych i integracyjnych. Testy obejmują krytyczne ścieżki aplikacji, w tym mechanizmy autoryzacji, obsługę endpointów oraz operacje na bazie danych. Dzięki testom możliwe jest wczesne wykrywanie regresji i utrzymanie stabilności implementacji podczas wprowadzania zmian. Zarządzanie zależnościami odbywa się przy użyciu pliku requirements.txt, co ułatwia odtworzenie środowiska uruchomieniowego; standardowa komenda

developerska uruchamia serwer poleceniem `uvicorn app.main:app --reload`.

Podsumowując, przyjęty stos technologiczny stanowi kompromis pomiędzy minimalizmem konfiguracji a gotowością do rozszerzenia. FastAPI i Uvicorn dostarczają nowoczesne i wydajne środowisko serwerowe z natywnym wsparciem dla walidacji i asynchroniczności, SQLite upraszcza uruchomienie projektu w środowisku dydaktycznym, a SQLAlchemy zapewnia warstwę abstrakcji umożliwiającą migrację do baz produkcyjnych. PyQt5 pozwala szybko dostarczyć funkcjonalny klient, zaś mechanizmy JWT i odpowiednie hashowanie haseł zapewniają podstawowe wymogi bezpieczeństwa. Wybór tych technologii umożliwia szybkie prototypowanie i wygodne testowanie, pozostawiając jednocześnie jasną ścieżkę migracji do rozwiązań bardziej zaawansowanych (np. PostgreSQL, webowy frontend, menedżer procesów) bez konieczności przepisania kluczowej logiki aplikacji.

5 Architektura systemu

5.1 Widok logiczny

Architektura systemu została zaprojektowana w sposób warstwowy, z wyraźnym rozdziałem odpowiedzialności pomiędzy warstwą API, warstwą usług (logiką biznesową) oraz warstwą dostępu do danych. Taki podział umożliwia czytelne oddzielenie zadań związanych z obsługą protokołu HTTP i walidacją wejścia od rzeczywistej logiki domenowej oraz operacji na bazie danych, co ułatwia testowanie, rozwój i ewentualną wymianę komponentów.

Warstwa API pełni rolę punktu wejściowego dla żądań zewnętrznych. Została zorganizowana w postaci oddzielnych routerów odpowiadających za konkretne obszary funkcjonalne: `auth`, `devices`, `device_data`, `admin` oraz `health`. Router `auth` obsługuje mechanizmy logowania, wydawania tokenów dostępu i tokenów odświeżających oraz końcówki związane z wylogowaniem i rotacją tokenów. Router `devices` udostępnia operacje CRUD dotyczące urządzeń: tworzenie, modyfikację, pobieranie listy i szczegółów, zaś router `device_data` przyjmuje i udostępnia odczyty telemetryczne (endpoints do przesyłania pojedynczych odczytów i batchy, pobierania historii). Router `admin` grupuje operacje administracyjne wymagające wyższych uprawnień, takie jak nadawanie ról, usuwanie zasobów czy ręczne resetowanie sekretów urządzeń. Router `health` udostępnia sondy stanu aplikacji (`liveness/readiness`), w tym weryfikację połączenia z bazą danych i dostępności zależnych usług.

W warstwie API stosowane są schematy wejściowe i wyjściowe oparte na Pydantic, które oddzielają strukturę danych przyjmowanych przez endpoints od modeli persystencji. Schematy te służą do wstępnej walidacji pól, normalizacji danych oraz do generowania dokumentacji OpenAPI. Odpowiedzi serwisu są filtrowane tak, aby nie ujawniać wrażliwych informacji (np. hashów haseł, sekretów urządzeń), a błędy domenowe mapowane są na odpowiednie kody HTTP za pomocą predefiniowanych wyjątków oraz globalnych handlerów wyjątków.

Warstwa usług zawiera logikę biznesową i działa jako pośrednik pomiędzy warstwą API a warstwą danych. Usługi autoryzacji odpowiadają za weryfikację poświadczeń, generowanie i weryfikację tokenów JWT, obsługę tokenów odświeżających oraz za polityki ograniczające (np. TTL tokenów, rotacja refresh tokenów). Usługi urządzeniowe realizują operacje dotyczące lifecycle urządzenia — walidację schematów profilu urządzenia, tworzenie i rotację sekretów, aktualizację metadanych oraz mechanizmy kontroli uprawnień (np. who may manage a device). Usługi odpowiedzialne za odczyty telemetryczne weryfikują rozmiar i format payloadu, ograniczają częstotliwość zapisu zgodnie z politykami (rate limiting / minimalny odstęp między odczytami) oraz agregują lub normalizują dane przed zapisem. Usługa logowania i audytu zapisuje istotne zdarzenia bezpieczeństwa i operacyjne (nieudane próby logowania, błędne żądania, zmiany w konfiguracji), co umożliwia późniejszą analizę incydentów. Moduł symulatora jako usługa testowa generuje zdarzenia telemetryczne według zdefiniowanych profili i harmonogramów; jego zadania mogą być realizowane jako zadania tła lub zewnętrzny proces korzystający z API.

Warstwa dostępu do danych została zrealizowana przy użyciu SQLAlchemy i obejmuje definicje modeli odpowiadających encjom domenowym: użytkownik, urządzenie, odczyt telemetryczny, tokeny odświeżające oraz rekordy audytu/zdarzeń bezpieczeństwa. Modele te zawierają odpowiednie relacje i indeksy optymalizujące najczęściej wykonywane zapytania (np. indeks po identyfikatorze urządzenia oraz po znaczniku czasu w tabeli odczytów). Dostęp do sesji bazy danych odbywa się za pośrednictwem fabryki sesji (sessionmaker) i jest wstrzykiwany do funkcji obsługujących żądania poprzez dependency injection FastAPI; transakcje są zamykane przez eksplicytny commit/rollback w warstwie usług lub za pomocą kontekstów zapewniających poprawne zwalnianie zasobów. W miejscach krytycznych stosowane są transakcje obejmujące kilka operacji (np. tworzenie urządzenia i jednocześnie tworzenie rekordu audytu), aby zapewnić spójność danych.

Mapowanie pomiędzy schematami Pydantic a modelami SQLAlchemy odbywa się w

dedykowanych funkcjach konwersji lub warstwach repozytoriów. Odpowiedzialność za bezpieczne exposowanie danych spoczywa na warstwie API: zwracane obiekty są serializowane przez Pydantic z odpowiednim odfiltrowaniem pól wrażliwych. Dodatkowo zastosowano centralny mechanizm obsługi błędów i wyjątków domenowych, który przekłada wewnętrzne wyjątki (np. brak uprawnień, nieistniejący zasób, przekroczenie limitu payloadu) na czytelne komunikaty z odpowiednimi statusami HTTP.

Autoryzacja i kontrola dostępu realizowana jest poprzez dependency injection kontrolerów zabezpieczeń w FastAPI, które sprawdzają poprawność tokenu JWT, weryfikują rolę użytkownika oraz, w przypadku urządzeń, sprawdzają zgodność sekretu lub tokenu urządzenia. Operacje administracyjne są dodatkowo zabezpieczone mechanizmami wymuszającymi posiadanie określonych uprawnień. Wszystkie krytyczne akcje modyfikujące stan systemu są rejestrowane w tabeli audytu, co ułatwia późniejszą inspekcję i analizę bezpieczeństwa.

Moduł symulatora oraz obsługa zadań asynchronicznych wykorzystuje mechanizmy zadań tła FastAPI lub zewnętrzne biblioteki HTTP (np. httpx) do wysyłania symulowanych odczytów do punktów końcowych API. Symulator jest konfigurowalny pod względem profilu urządzenia, częstotliwości wysyłania oraz rozkładu wartości telemetrycznych. W środowisku testowym symulator umożliwia generowanie scenariuszy obciążeniowych oraz testowanie polityk ograniczeń i mechanizmów obronnych (np. rate limiting, odrzucanie nadmiernych payloadów).

W obszarze obserwowalności i logowania zaimplementowano strukturalne logowanie zdarzeń aplikacyjnych oraz mechanizm zapisu logów zdarzeń bezpieczeństwa do bazy. Router `health` wykorzystuje te mechanizmy do zgłaszania stanu aplikacji; sondy zdrowia wykonują minimalne zapytanie testowe do bazy danych oraz sprawdzają dostępność kluczowych komponentów, dzięki czemu można odróżnić problemy z infrastrukturą od błędów w logice aplikacji.

W całości architektury przewidziano możliwości rozszerzenia: dzięki separacji warstw możliwa jest wymiana komponentów (np. przejście z SQLite na PostgreSQL, zastąpienie PyQt5 frontendem webowym) bez konieczności przebudowy warstwy biznesowej.

5.2 Diagramy

Diagram komponentów systemu.

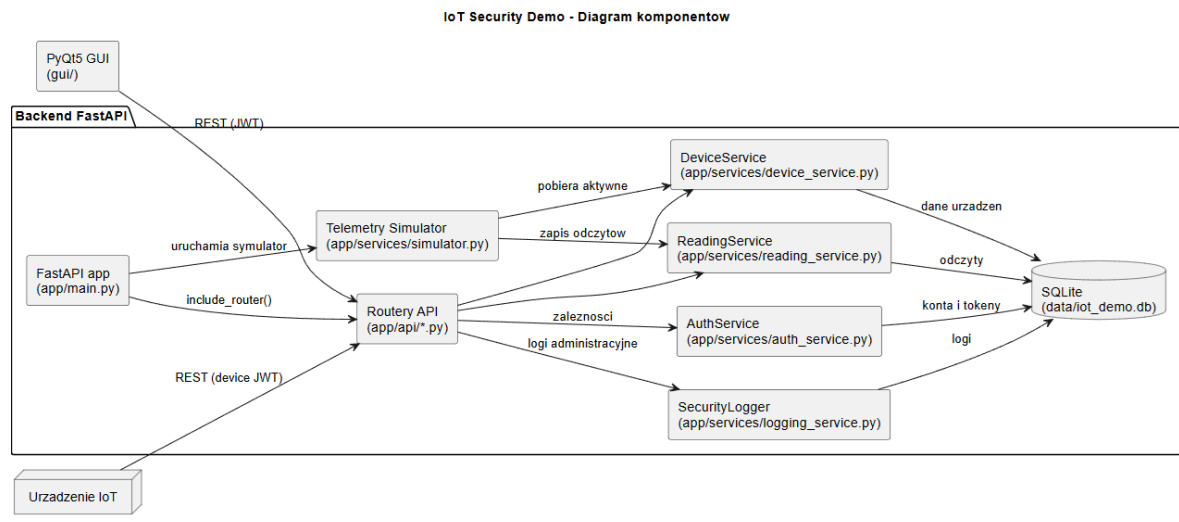
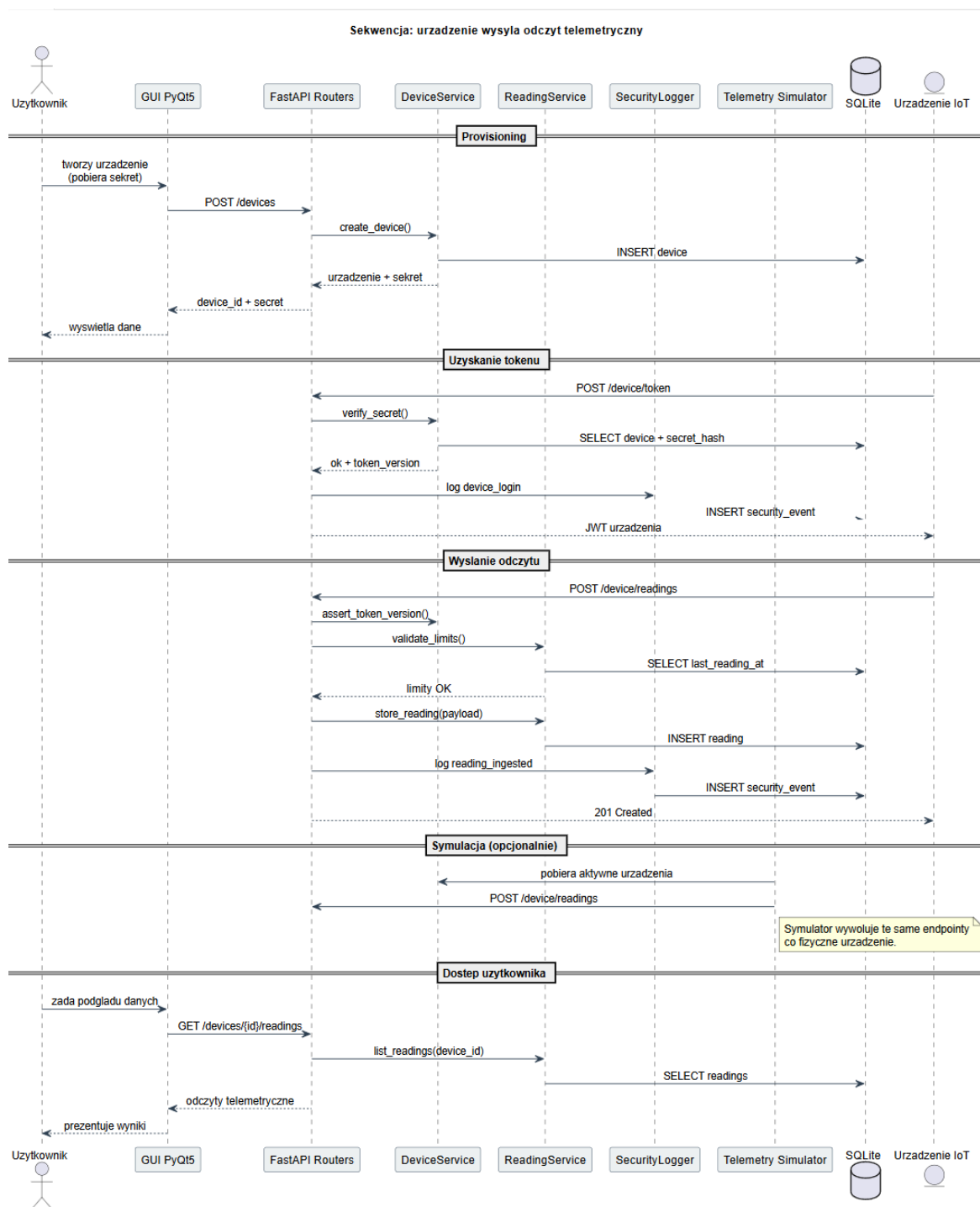


Diagram sekwencji.



6 Model danych

Model danych zaprojektowano z myślą o czytelności, spójności i łatwości rozszerzania. Schemat obejmuje pięć głównych encji: użytkownik, urządzenie, odczyt telemetryczny, token odświeżający oraz zdarzenie bezpieczeństwa (audit). Relacje pomiędzy encjami i indeksy dobrano tak, aby odpowiadały typowym zapytaniom systemu (pobieranie listy urządzeń, filtrowanie odczytów według identyfikatora i przedziału czasowego, weryfikacja ważności tokenów).

Tabela użytkowników przechowuje podstawowe informacje o koncie: identyfikator, adres e-mail, hash hasła, rolę (np. zwykły użytkownik / administrator), znacznik czasu ostatniego logowania oraz ewentualne metadane (np. aktywność, flaga zablokowania). Pole z adresem e-mail posiada ograniczenie unikalności, co pozwala efektywnie walidować duplikaty już na poziomie bazy. Hasła nie są przechowywane w postaci jawnej — zapisywany jest jedynie hash z użyciem bezpiecznego algorytmu (np. bcrypt) oraz saltów, co minimalizuje ryzyko kompromitacji przy wycieku danych.

Tabela urządzeń zawiera informacje identyfikujące sprzęt: unikalny identyfikator urządzenia, nazwa, kategoria/profil, metadane konfiguracyjne oraz hash sekretu urządzenia (sekret nie jest przechowywany jawnie). Dla urządzeń przewidziano pole wskazujące właściciela (relacja do tabeli użytkowników) oraz pola pomocnicze, takie jak status (aktywny/wyłączony), wersja konfiguracji czy timestamp ostatniej komunikacji. W modelu uwzględniono również indeksy po identyfikatorze urządzenia i po kolumnie używanej przy zapytaniach listujących (np. po nazwie lub statusie).

Tabela odczytów telemetrycznych zawiera surowe dane przesyłane przez urządzenia wraz z metadanymi: identyfikator odczytu, identyfikator urządzenia (klucz obcy), znacznik czasu (timestamp), typ/pole payloadu oraz sam payload (najczęściej serializowany JSON). Dla wydajnego pobierania historii odczytów zalecane jest posiadanie indeksu na kolumnie identyfikatora urządzenia oraz na kolumnie znacznika czasu. Ze względu na charakter danych telemetrycznych rozważono politykę przechowywania i archiwizacji — starsze odczyty mogą być okresowo agregowane lub przenoszone do archiwum, jeśli zajdzie potrzeba ograniczenia rozmiaru bazy.

Tabela tokenów odświeżających (refresh token) przechowuje informacje niezbędne do walidacji i rotacji tokenów: unikalne identyfikatory tokenów, powiązanie z użytkownikiem,

data wygenerowania, data wygaśnięcia oraz flaga unieważnienia. W projektach produkcyjnych atrakcyjne jest przechowywanie tylko skrótu tokenu (hash) zamiast tokena w postaci jawnej, co umożliwia weryfikację przedstawionego tokena po zahashowaniu i jednocześnie zmniejsza ryzyko wycieku. Mechanizm rotacji refresh tokenów i ewidencjonowania jwt pozwala na szybkie unieważnienie pojedynczych tokenów oraz na implementację polityk bezpieczeństwa (np. unieważnianie poprzednich tokenów przy wydaniu nowego).

Tabela zdarzeń bezpieczeństwa (audit log) zapisuje ważne incydenty i akcje: nieudane próby logowania, zmiany uprawnień, reset sekretów urządzeń, przekroczenia limitów zapisu telemetrycznego i inne zdarzenia istotne dla analizy bezpieczeństwa. Rekordy audytu zawierają co najmniej: identyfikator zdarzenia, typ zdarzenia, powiązany użytkownik lub urządzenie (opcjonalnie), znacznik czasu oraz dodatkowe dane opisowe. Rejestr audytu powinien być zapisywany w sposób append-only, a dostęp do jego modyfikacji ograniczony do wybranych operacji administracyjnych.

W odniesieniu do SQLite przyjęto następujące założenia. SQLite jest wykorzystywane jako lekka, przenośna baza danych, odpowiednia dla środowiska demonstracyjnego i developerskiego. Infrastruktura oparta na pliku bazy upraszcza uruchomienie i dystrybucję projektu, jednak należy pamiętać o ograniczeniach tego rozwiązania: jednoczesne operacje zapisu są blokowane na poziomie pliku, co przy dużej liczbie równoległych zapisów może powodować spadki wydajności i konflikty. Z tego powodu dla scenariuszy produkcyjnych z intensywnym ruchem odczytów/zapisów rekomendowane jest przejście do systemu takiego jak PostgreSQL. Dzięki zastosowaniu warstwy ORM zmiana silnika bazy danych wymaga jedynie minimalnych zmian konfiguracyjnych.

Inicjalizacja schematu bazy może odbywać się dwojako. W trybie szybkiego prototypowania i testów możliwe jest wygenerowanie wszystkich tabel bezpośrednio przy starcie aplikacji, np. przy użyciu mechanizmu SQLAlchemy: utworzenie silnika (engine) oraz wywołanie `Base.metadata.create_all(engine)`. To rozwiązanie jest wygodne przy pierwszym uruchomieniu, lecz nie obsługuje zarządzania migracjami schematu w czasie, dlatego w projektach rozwijanych dłużej rekomendowane jest użycie narzędzia migracyjnego, np. Alembic. Alembic umożliwia kontrolowane wprowadzanie zmian w strukturze bazy (dodawanie kolumn, indeksów, migracje danych) oraz wersjonowanie schematu, co jest niezbędne przy współpracy zespołowej i przy aktualizacjach środowisk produkcyjnych.

7 Interfejs API

7.1 Autoryzacja i uwierzytelnianie

Mechanizm uwierzytelniania oparto na schemacie „access / refresh” z wykorzystaniem tokenów JWT. Po pomyślnym uwierzytelnieniu użytkownik otrzymuje krótko żyjący token dostępu (`access_token`) oraz dłużej ważny token odświeżający (`refresh_token`). Token dostępu służy do autoryzacji większości żądań aplikacyjnych i powinien zawierać minimalny zestaw informacji niezbędnych do weryfikacji uprawnień (np. identyfikator podmiotu i rolę). Token odświeżający służy wyłącznie do uzyskania nowej pary tokenów i podlega mechanizmowi rotacji — przy wydaniu nowego refresh tokena poprzedni powinien zostać unieważniony.

Dla urządzeń zdefiniowano odrębne poświadczenia (`device_token` lub sekret urządzenia). Polityka dotycząca TTL oraz sposób rotacji sekretów/tokenów urządzeń dobierane są w zależności od możliwości urządzenia (np. czy potrafi bezpiecznie przechowywać sekret) oraz od modelu provisioning’u (jednorazowy sekret przy rejestracji vs. provisioning przez kanał zabezpieczony).

Weryfikacja po stronie serwera obejmuje sprawdzenie podpisu, pól standardowych (`exp`, `iat`) oraz kontekstowych (`iss`, `aud`, `jti`). Wszystkie żądania zawierające wrażliwe dane muszą odbywać się po TLS/HTTPS. Wdrożenie dodatkowych mechanizmów (np. ograniczenie TTL, przechowywanie refresh tokenów jako hashy, audytowanie zdarzeń) znacząco zwiększa odporność systemu na kompromitację poświadczeń.

7.2 Przykłady wywołań

Przykład żądania logowania (request) i typowej odpowiedzi (response):

```
# POST /auth/login
# Content-Type: application/json
```

```
{ "email": "user@example.com", "password": "Secret123!" }
```

```
# HTTP/1.1 200 OK
# Content-Type: application/json
```

```
{
  "access_token": "<eyJhbGciOiJI...>",
  "refresh_token": "<eyJhbGciOiJI...>",
  "token_type": "bearer",
  "expires_in": 900
}
```

Przykład rejestracji urządzenia (skrót):

```
# POST /admin/devices
# Authorization: Bearer <admin_access_token>
```

```
{ "name": "sensor-01", "category": "temperature", "profile": {} }
```

Przykład wysyłki odczytu telemetrycznego przez urządzenie:

```
# POST /device/readings
# Authorization: Bearer <device_token>
```

```
{
  "device_id": "sensor-01",
  "timestamp": "2025-10-17T08:30:00Z",
  "payload": { "t": 21.3 }
}
```

8 Klient GUI (PyQt5)

Aplikacja kliencka stanowi lekki, desktopowy interfejs przeznaczony do zarządzania zasobami systemu IoT oraz do monitorowania i analizy danych telemetrycznych. Interfejs zaprojektowano z naciskiem na czytelność oraz prostotę obsługi — główne zadania to prezentacja listy urządzeń, przegląd historii odczytów, zarządzanie cyklem życia urządzeń oraz wykonywanie operacji administracyjnych przez użytkowników z odpowiednimi uprawnieniami. Całość utrzymana jest w logice zakładkowej (tabs), co ułatwia szybki dostęp do najważniejszych funkcji bez przeładowania ekranu.

Widok główny aplikacji składa się z trzech podstawowych obszarów: lista urządzeń, widok odczytów oraz panel administracyjny. W zakładce urządzeń wyświetlana jest lista wszystkich urządzeń wraz z kluczowymi metadanymi — nazwa, kategoria, status połączenia oraz znacznik ostatniej komunikacji. Elementy listy umożliwiają szybkie sortowanie i filtrowanie (np. po nazwie, statusie lub kategorii) oraz wybór urządzenia w celu uzyskania szczegółów. Dla wybranego urządzenia prezentowany jest panel boczny lub modal z dodatkowymi informacjami (np. pełne metadane, ostatnie odczyty, konfiguracja) oraz z kontrolkami operacyjnymi (rotacja sekretu, edycja, usunięcie — o ile uprawnienia na to pozwalają).

Zakładka z odczytami udostępnia historyczny widok telemetryczny i mechanizmy filtracji. Użytkownik może ograniczyć zakres czasowy (od / do), dobrać sposób agregacji danych (surowe wartości, średnie wg okna czasowego) oraz wybrać format wyświetlania (lista tabelaryczna lub wykresy). Widok tabelaryczny umożliwia szybkie przeglądanie szczegółów każdego odczytu, a wykresy (time series) wspomagają wykrywanie trendów i anomalii. W interfejsie przewidziano możliwość eksportu zaznaczonego zakresu odczytów do pliku CSV oraz opcję pobrania surowych payloadów w formacie JSON.

Funkcjonalność dodawania i konfigurowania urządzeń zaimplementowano w formie kreatora (wizard) lub dialogu formularzowego. Formularz umożliwia uzupełnienie podstawowych pól opisowych (nazwa, kategoria, profil), opcjonalne pola konfiguracyjne oraz podgląd przykładowego payloadu wysyłanego przez urządzenie. Po zatwierdzeniu danych kreator wykonuje żądanie do API i w przypadku powodzenia aktualizuje widok listy. W przypadku błędów walidacji lub niepowodzenia operacji użytkownik otrzymuje czytelny komunikat z opisem problemu i opcją ponowienia akcji.

Operacje krytyczne, takie jak rotacja sekretu urządzenia lub usunięcie zasobu, są po-

twierdzone przez dodatkowe okno dialogowe i — w zależności od polityki bezpieczeństwa systemu — wymagają dodatkowego potwierdzenia tożsamości (np. ponowne podanie hasła lub sprawdzenie roli). Po rotacji sekretu nowy sekret jest wyświetlany w trybie jednorazowym; aplikacja informuje użytkownika o konieczności zapisania sekretu, a w bazie przechowywana jest jedynie jego bezpieczna reprezentacja (hash).

Wszystkie wywołania sieciowe wykonywane są w sposób nieblokujący względem wątku interfejsu użytkownika. Podczas długotrwałych operacji (pobieranie dużej liczby odczytów, eksport) interfejs pokazuje wskaźnik postępu lub spinner, a przy błędach sieciowych wyświetlane są komunikaty z możliwością ponowienia operacji. Mechanizmy retry i timeouty są stosowane po stronie klienta dla poprawy odporności na chwilowe przerwy w łączności.

W zakresie prezentacji i użyteczności przewidziano następujące elementy ergonomiczne: responsywna tabela z możliwością zmiany szerokości kolumn, inteligentne paginowanie w widoku dużych zbiorów danych, możliwość zapamiętywania ostatnio używanych filtrów, kontekstowe menu prawym przyciskiem myszy dla szybkich akcji oraz łatwy dostęp do dokumentacji API lub pomocy w aplikacji. Dla użytkowników administracyjnych dostępne są dodatkowe zakładki i kontrolki (np. przegląd sesji, ręczne unieważnianie tokenów, audyt zdarzeń).

Aspekty bezpieczeństwa w kliencie obejmują bezpieczne przechowywanie poświadczeń (wykorzystanie mechanizmów systemowych przygody do bezpiecznego magazynowania w środowiskach produkcyjnych), transmisję wyłącznie po TLS oraz ukrywanie wrażliwych pól (np. pole sekretu wyświetlane jako „maskowane” do momentu jednoznacznej akcji użytkownika). Interfejs powinien również respektować uprawnienia zwracane przez API — przyciski i funkcje niedostępne dla bieżącego użytkownika są niewidoczne lub dezaktywowane.

Funkcje eksportu obejmują eksport listy urządzeń i historii odczytów do formatu CSV oraz możliwość zapisania wyeksportowanych plików z odpowiednimi nagłówkami i metadany. W obszarze diagnostyki dostępny jest panel z logami lokalnymi lub uproszczonymi komunikatami błędów, które ułatwiają wsparcie administracyjne i debugowanie problemów podczas testów.

Panel logowania – System IoT

?

×

Witaj ponownie 🖐️

Zaloguj się, aby kontynuować pracę w konsoli bezpieczeństwa.

Adres e-mail

Hasło

Zaloguj się

[Nie masz konta? Zarejestruj się](#)

Tokeny dostępu odświeżamy automatycznie w trakcie pracy aplikacji.

Rysunek 1: Panel logowania

Panel logowania – System IoT

Dołącz do systemu

Utwórz konto, aby monitorować i chronić swoje urządzenia IoT.

Adres e-mail

jan.nowak@example.com

Hasło

Wprowadź co najmniej 8 znaków

Utwórz konto i zaloguj

[Masz już konto? Wróć do logowania](#)

Tokeny dostępu odświeżamy automatycznie w trakcie pracy aplikacji.

Rysunek 2: Panel rejestracji użytkownika

Konsola bezpieczeństwa IoT

Zalogowano jako supierz.marek@gmail.com (Użytkownik)

Urządzenia

Urządzenia

Wybierz urządzenie, aby zobaczyć szczegóły.

Kategoria: - Właściciel: -
Utworzono: - Ostatni odczyt: -

Limit 100 od (ISO 8601) do (ISO 8601) **Filtruj** **Eksportuj CSV** **Eksportuj TXT**

Dodaj urządzenie **Odśwież**
Usuń urządzenie **Rotuj sekret**

Legenda kategorii

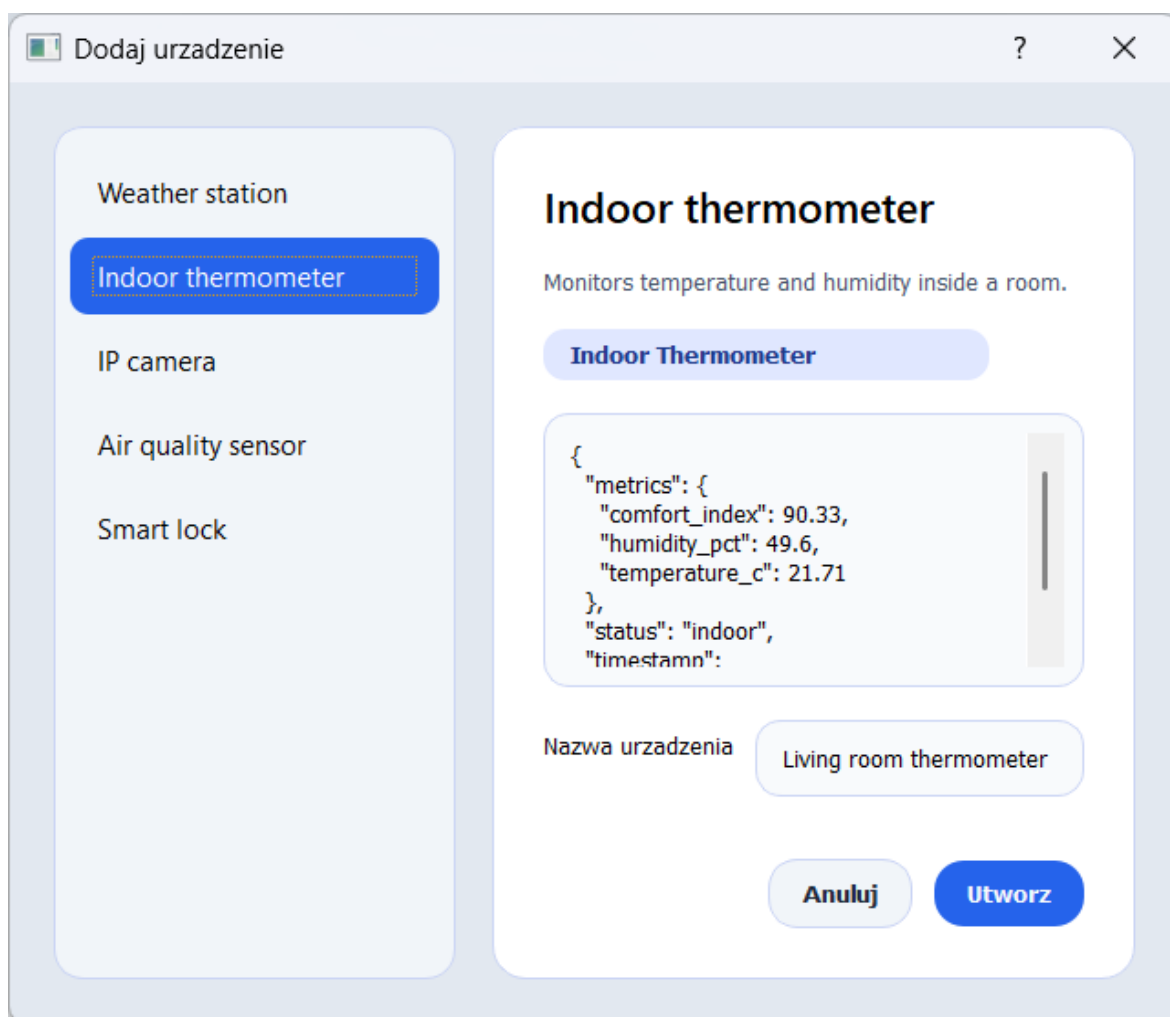
Legenda: wybierz urządzenie, aby zobaczyć opis parametrów kategorii.

Wyloguj

Łącznie odczytów: -

Kolumny stałe: Odebrano (czas zapisu), Znacznik urządzenia (czas z czujnika), opcjonalny Status oraz metryki specyficzne dla kategorii.

Rysunek 3: Główne okno użytkownika



Dodaj urządzenie

Weather station

Indoor thermometer

IP camera

Air quality sensor

Smart lock

Indoor thermometer

Monitors temperature and humidity inside a room.

Indoor Thermometer

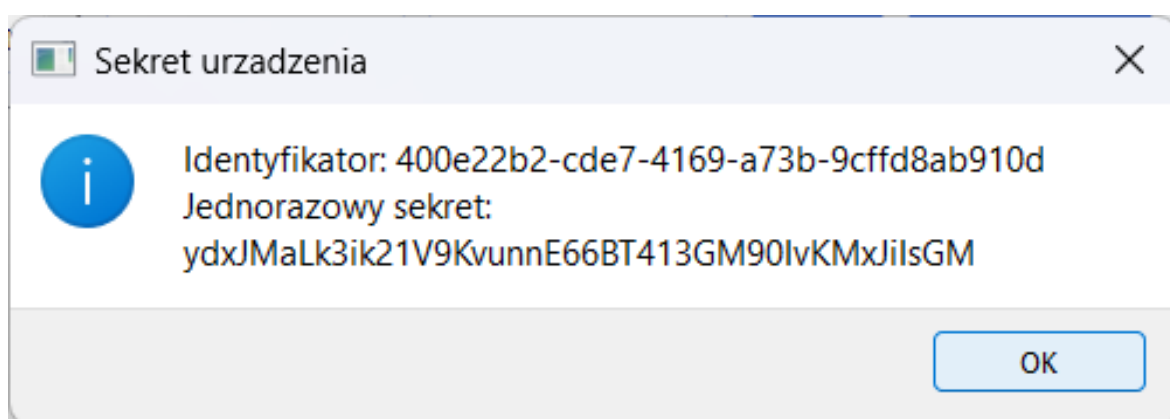
```
{
  "metrics": {
    "comfort_index": 90.33,
    "humidity_pct": 49.6,
    "temperature_c": 21.71
  },
  "status": "indoor",
  "timestamp":
}
```

Nazwa urządzenia

Living room thermometer

Anuluj Utworz

Rysunek 4: Formularz dodawania urządzenia



Sekret urządzenia

i

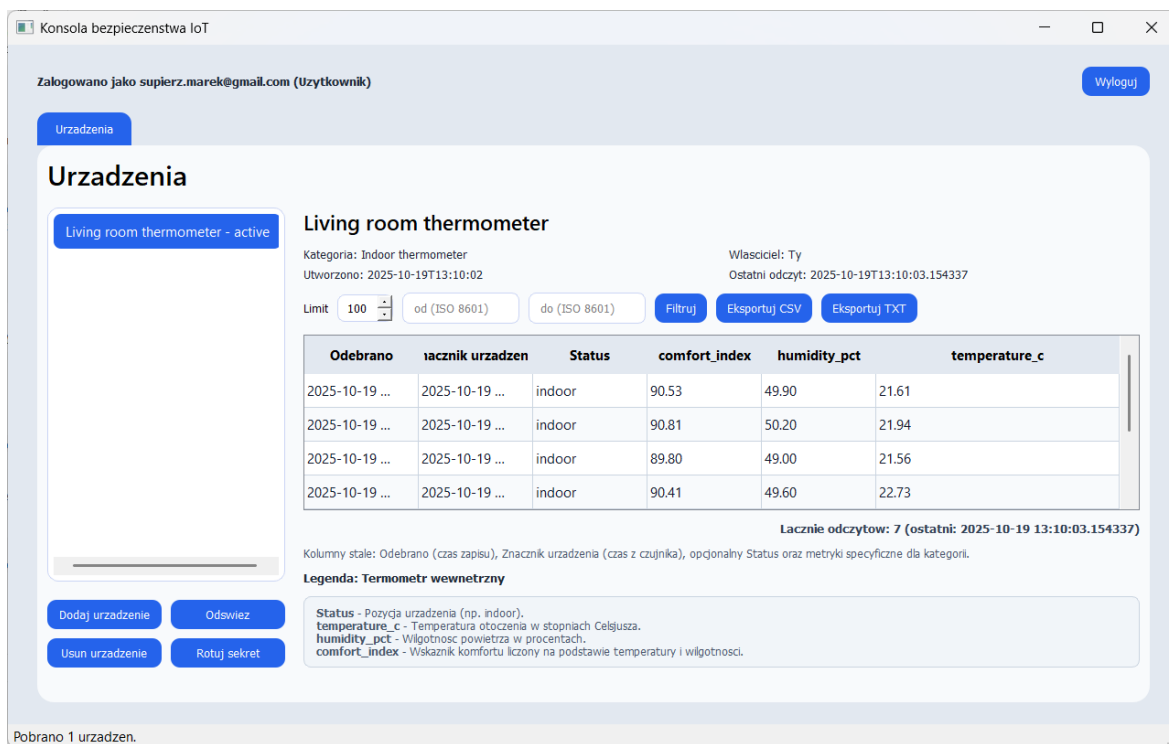
Identyfikator: 400e22b2-cde7-4169-a73b-9cffd8ab910d

Jednorazowy sekret:

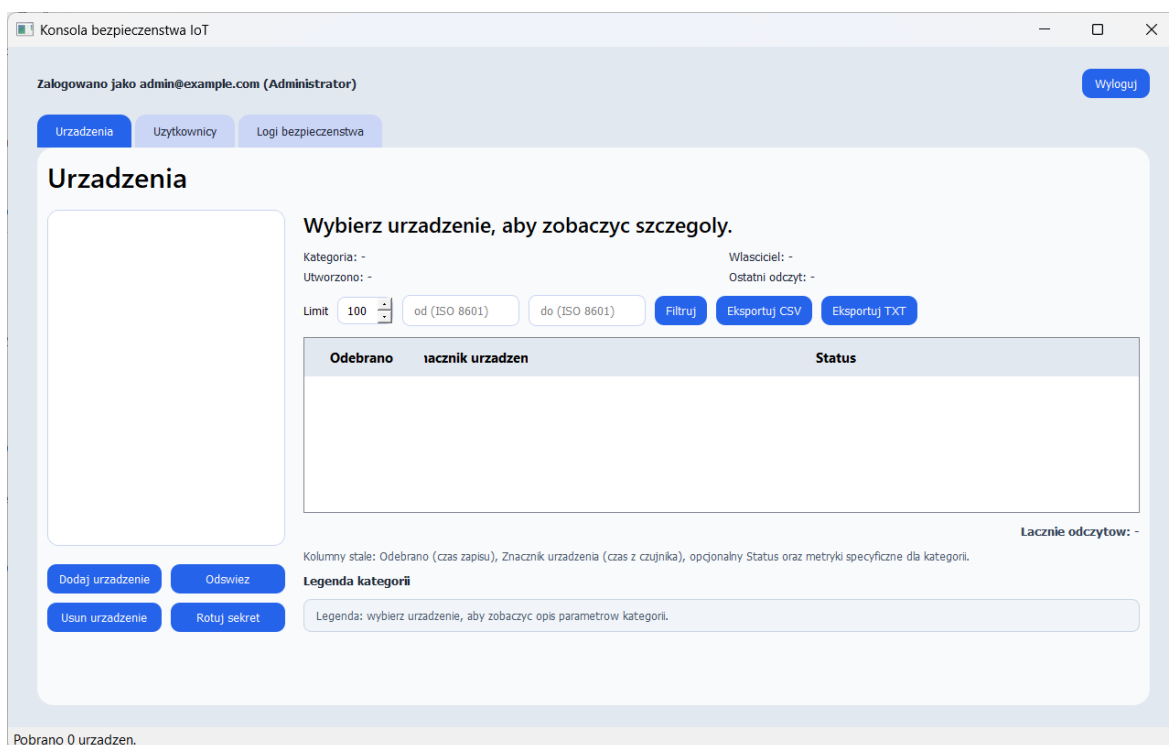
ydxJMaLk3ik21V9KvunnE66BT413GM90lvKMxJilsGM

OK

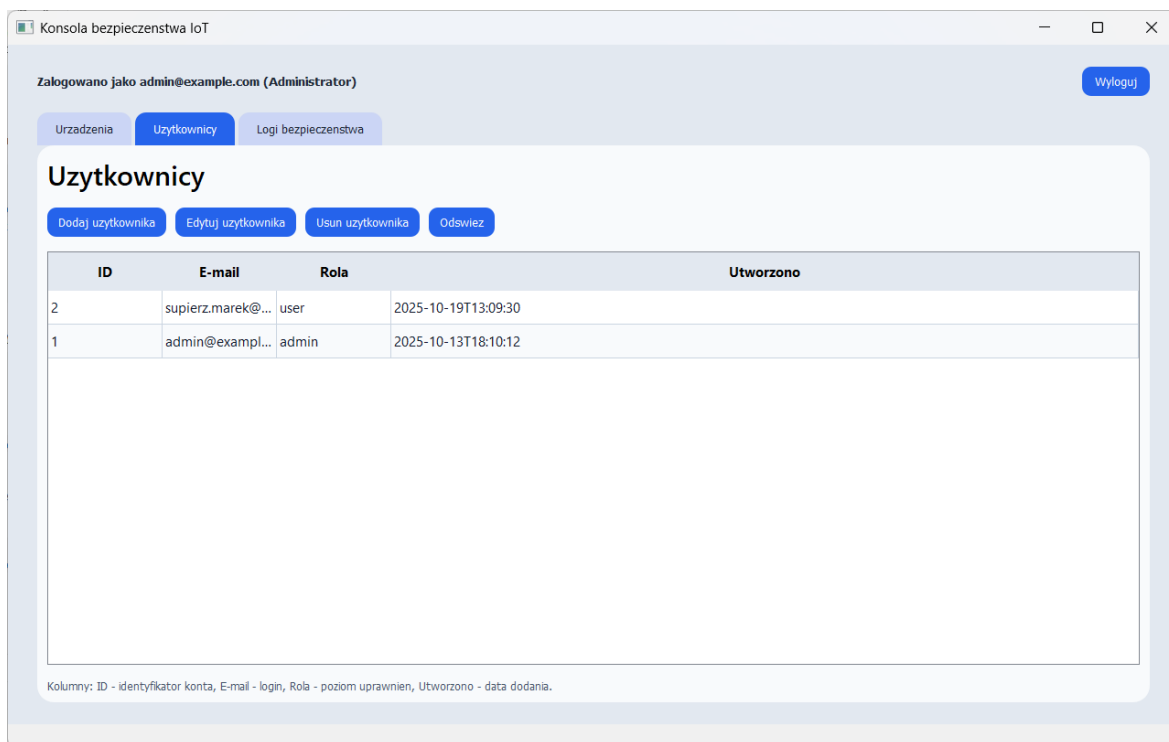
Rysunek 5: Rotacja sekretu urządzenia (widok jednorazowego sekretu)



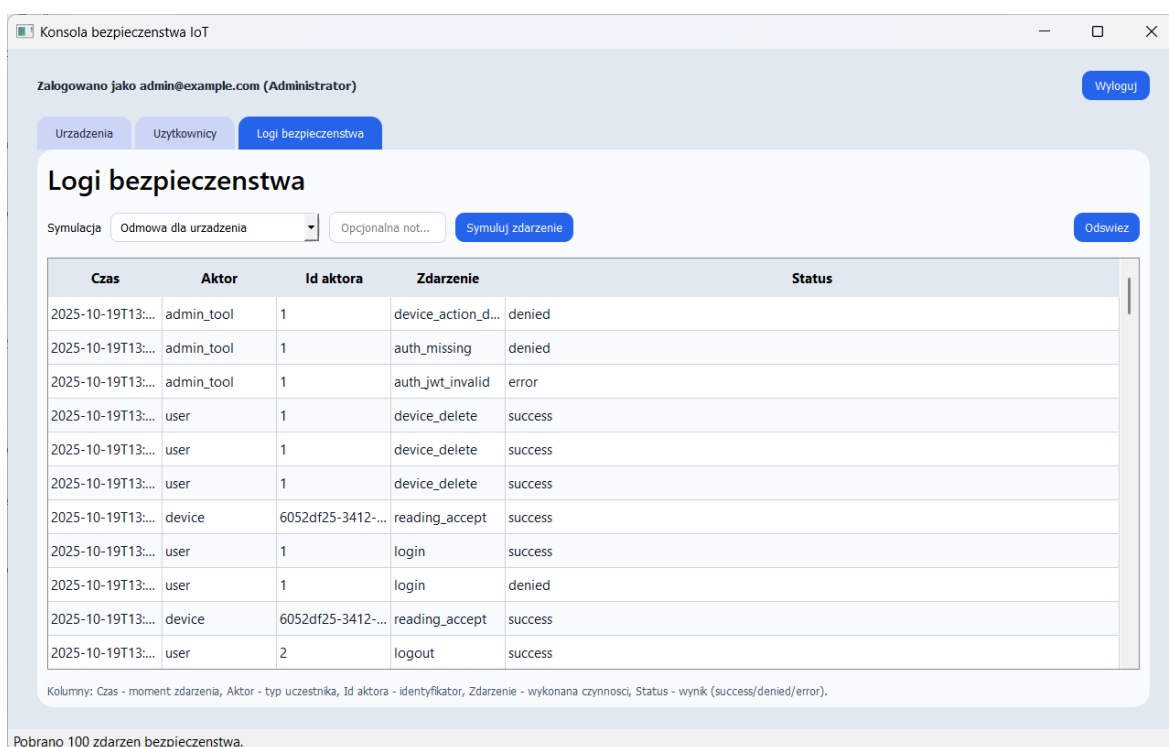
Rysunek 6: Przykładowe dane telemetryczne



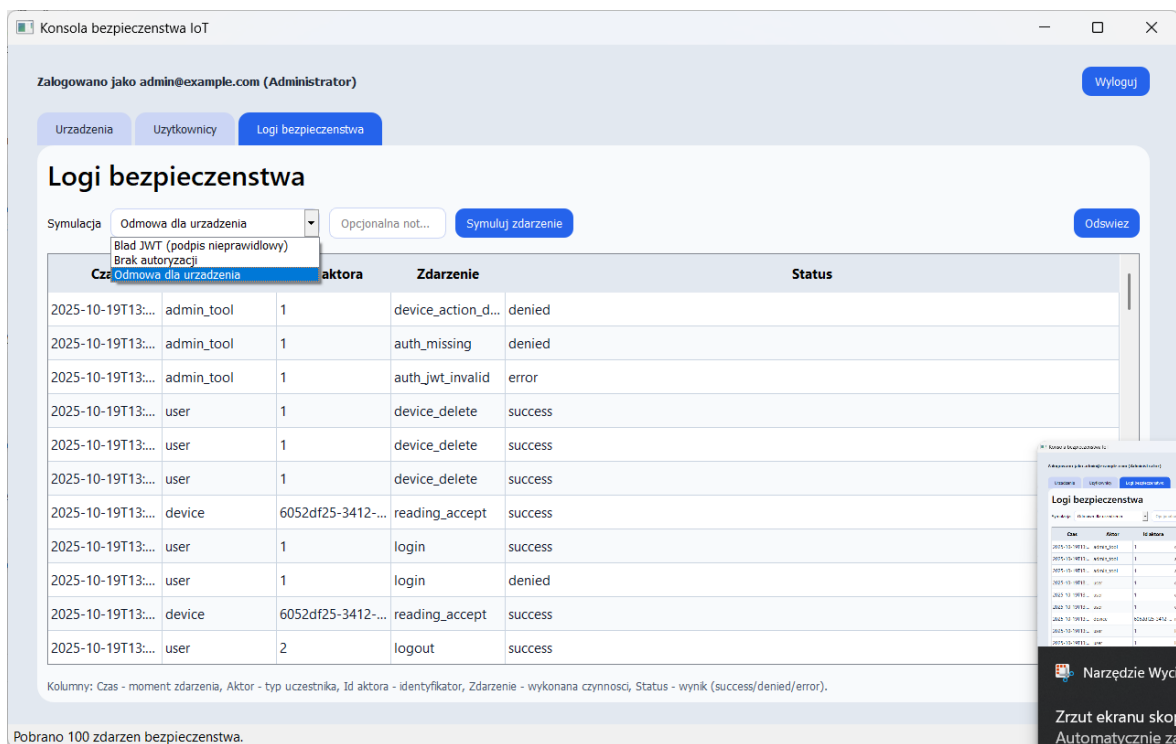
Rysunek 7: Panel administratora



Rysunek 8: Zarządzanie użytkownikami



Rysunek 9: Logi systemowe / rejestr audytu



Rysunek 10: Przykładowe typy testów i scenariusze

9 Testy jednostkowe

Poniżej przedstawiono wyniki ostatniego uruchomienia testów jednostkowych wraz z raportem pokrycia kodu (pytest + pytest-cov). Testy uruchomiono lokalnie w środowisku deweloperskim.

```
# polecenie uruchomienia testów
pytest --cov=app --cov=gui --cov-report=term

# skrócony wynik:
49 passed, 1 warning in 25.21s
TOTAL COVERAGE: 89\%
```

Tabela wyników pokrycia (wybrane moduły)

Moduł	Stmts	Miss	Cover
app/api/dependencies.py	52	10	81%
app/api/routes/devices.py	82	8	90%
app/services/auth_service.py	99	12	88%
app/services/device_service.py	129	21	84%
app/services/reading_service.py	104	6	94%
app/services/simulator.py	71	9	87%
app/services/user_service.py	49	7	86%
gui/api_client.py	205	34	83%
gui/app.py	25	6	76%
gui/main_window.py	777	126	84%
TOTAL	2320	262	89%

Interpretacja wyników

Ogólne pokrycie testami wynosi 89%. Wykonano 49 testów — wszystkie zakończone powodzeniem — oraz odnotowano jedno ostrzeżenie związane z deprecjacją w bibliotece Pydantic.

Pełne pokrycie występuje w prostych modułach schematów i lekkich endpointach API — przykładowo pliki `app/api/routes/auth.py` oraz `app/api/routes/device_data.py` są w całości objęte testami. W warstwie serwisów biznesowych (m.in. `app/services/device_service.py`, `app/services/auth_service.py`, `app/services/simulator.py`) uzyskano dobre, lecz niepełne pokrycie w zakresie 84–88%. Braki dotyczą głównie ścieżek błędów, warunków brzegowych oraz rzadko wykonywanych gałęzi logiki (np. retry, obsługa wyjątków).

Warstwa GUI charakteryzuje się niższym pokryciem: plik `gui/app.py` osiąga około 76%, a `gui/main_window.py` około 84%. Duża liczba niepokrytych linii w `gui/main_window.py` wskazuje na obecność kodu ściśle związanych z interakcją użytkownika i renderowaniem UI, które z natury są trudniejsze do objęcia testami jednostkowymi.

10 Instrukcja uruchomienia

10.1 Wymagania

Python 3.11+

```
C:\> pip install -r requirements.txt
```

10.2 Uruchomienie backendu

```
C:\> uvicorn app.main:app --reload
# Swagger UI: http://127.0.0.1:8000/docs
# Redoc:      http://127.0.0.1:8000/redoc
# Health:     http://127.0.0.1:8000/healthz
```

10.3 Uruchomienie GUI

```
C:\> python -m gui
```

11 Możliwe zagrożenia

- **Kradzież tokenów** — jeśli ktoś zdobędzie token (access lub refresh), może udawać użytkownika. *Co robić:* stosować krótkie TTL dla tokenów dostępu, rotować i przechowywać jedynie hashe refresh tokenów, używać HTTPS, logować podejrzane użycie tokenów.
- **Ataki typu brute-force / credential stuffing** — próby odgadywania haseł lub używania wykradzionych par login/hasło. *Co robić:* blokować konto po kilku nieudanych próbach, stosować rate limiting dla endpointów logowania, wymagać silnych haseł i (gdzie możliwe) MFA.
- **Flood telemetryczny / nadużycie API** — urządzenia wysyłają zbyt dużo danych i przeciążają system. *Co robić:* limitować liczbę żądań i rozmiar payloadu, wymuszać minimalny odstęp między odczytami, monitorować nietypowy ruch i automatycznie odcinać źródła nadużyć.

- **Brak TLS (HTTPS)** — przesyłanie danych niezabezpieczonym kanałem naraża na podsłuch i modyfikację. *Co robić:* wymuszać HTTPS, stosować aktualne certyfikaty, rozważyć HSTS i sprawdzać poprawność certyfikatów po stronie klienta.
- **Błędy przy migracjach bazy danych** — ręczne zmiany schematu mogą prowadzić do utraty danych lub niezgodności. *Co robić:* używać narzędzia do migracji (np. Alembic), testować migracje i robić kopie zapasowe przed zmianami.

12 Podsumowanie

Projekt został zrealizowany zgodnie z założeniami. Powstał backend REST oparty na FastAPI z mechanizmami uwierzytelniania JWT, baza danych SQLite oraz prosty klient desktopowy w PyQt5. Implementacja obejmuje rejestrację i logowanie użytkowników, rejestrację urządzeń, przyjmowanie odczytów telemetrycznych, rotację sekretów oraz rejestr zdarzeń. Kod został przetestowany — wszystkie testy przeszły pomyślnie (49 testów), a raport pokrycia pokazuje około 89% pokrycia. Dokumentacja i przykłady wywołań znajdują się w repozytorium. Podział prac został zrealizowany zgodnie z przyjętym harmonogramem — zadania zostały rozdzielone mniej więcej po połowie między Markiem Supierzem i Andrzejem Mysiorem, a postępy monitorowano zgodnie z planem.