Wyższa Szkoła Bankowa we Wrocławiu

Wydział Finansów I Zarządzania

Instytut Informatyki I Gospodarki Cyfrowej

Informatyka spejcalizacja aplikacje mobilne

Nr indeksu: 81723

Karol Malicki

**“**Projektowanie i implementacja aplikacji mikroserwisowej dla **zarządzania notyfikacjami** z uwierzytelnianiem opartym na JSON Web Token**”**

Praca dyplomowa

napisana pod kierunkiem

Wojciecha Barczyńskiego

w roku akademickim 20224/2025

Wrocław 2024

**Streszczenie**

Celem niniejszej pracy inżynierskiej jest opracowanie systemu bazującego na architekturze mikroserwisów, który wykorzystuje tokeny JWT do zapewnienia bezpiecznej autentykacji oraz zarządzania użytkownikami. W pracy zaprezentowano projekt kilku mikroserwisów, w tym usługi autentykacji, zarządzania użytkownikami, funkcji administracyjnych oraz komponentów wspierających powiadomienia e-mail, logowanie, audyt i migracje baz danych. Ponadto, omówiono sposób integracji tych mikroserwisów z bazami danych relacyjnymi (PostgreSQL) oraz NoSQL (MongoDB), a także zastosowanie narzędzi takich jak Flyway i Vault do zarządzania bazami danych i kluczami w klastrze.

W ramach pracy opracowano rozwiązanie umożliwiające efektywne zarządzanie danymi użytkowników oraz zapewniające bezpieczny dostęp do aplikacji. Projekt obejmuje również wdrożenie procesu autentykacji z użyciem tokenów JWT oraz integrację różnych komponentów systemu, co umożliwia jego łatwą skalowalność i rozbudowę. Praca zawiera również analizę wyzwań związanych z bezpieczeństwem, monitorowaniem systemu oraz integracją mikroserwisów. Wynikiem prac jest działający system oparty na mikroserwisach, który może stanowić fundament dla rozwoju aplikacji webowych i mobilnych wymagających rozdzielonych komponentów i bezpiecznej obsługi użytkowników.

### **1. Wstęp**

1.1 Zakres pracy  
1.2 Cel pracy  
1.3 Metodyka i organizacja pracy

### 2. Podstawy teoretyczne

2.1 Wprowadzenie do architektury mikroserwisowej  
2.2 JSON Web Token (JWT) – teoria i zastosowania  
2.3 Zarządzanie tajemnicami w systemach rozproszonych  
2.4 Wzorce projektowe w architekturze mikroserwisowej  
2.5 Narzędzia wspierające implementację mikroserwisów

### 3. Analiza problemu i założeń projektowych

3.1 Wymagania funkcjonalne systemu  
3.2 Wymagania niefunkcjonalne  
3.3 Przegląd dostępnych rozwiązań i wybór technologii  
3.4 Opis scenariuszy użycia

### 4. Projekt systemu

4.1 Architektura systemu  
4.2 Specyfikacja mikroserwisów

* 4.2.1 **auth-service**: logowanie, uwierzytelnianie, generowanie JWT
* 4.2.2 **user-service**: zarządzanie użytkownikami
* 4.2.3 **admin-service**: funkcje administracyjne
* 4.2.4 **mail-service**: powiadomienia e-mail
* 4.2.5 **logger-service**: logowanie i audyt  
  4.3 Integracja z bazami danych
* 4.3.1 PostgreSQL – dane strukturalne
* 4.3.2 MongoDB – dane niestrukturalne  
  4.4 Mechanizmy komunikacji między mikroserwisami  
  4.5 Wykorzystanie Vault i Google Cloud KMS do zarządzania tajemnicami

### 6. Zarządzanie bezpieczeństwem

6.1 Bezpieczne przechowywanie i dystrybucja kluczy  
6.2 Implementacja uwierzytelniania i autoryzacji z JWT  
6.3 Zabezpieczenia komunikacji między mikroserwisami  
6.4 Monitoring bezpieczeństwa i logowanie incydentów

### 7. Monitorowanie i skalowalność systemu

7.1 Narzędzia monitorujące i ich konfiguracja  
7.2 Skalowanie infrastruktury w GKE  
7.3 Analiza wydajności systemu

### 8. Testowanie i weryfikacja systemu

8.1 Strategie testowania mikroserwisów  
8.2 Testowanie procesów autentykacji  
8.3 Weryfikacja poprawności integracji komponentów  
8.4 Wyniki testów i analiza jakości

### 9. Wnioski i perspektywy rozwoju

9.1 Podsumowanie wyników pracy  
9.2 Możliwości rozwoju systemu  
9.3 Wyzwania i rekomendacje na przyszłość

**Rozdział 1**

Wstęp

W niniejszej pracy inżynierskiej przedstawiono projekt i implementację systemu opartego na mikroserwisach, z wykorzystaniem technologii zapewniających bezpieczną autentykację oraz efektywne zarządzanie użytkownikami w środowisku rozproszonym. W odpowiedzi na dynamiczny rozwój aplikacji internetowych oraz rosnące wymagania dotyczące skalowalności i elastyczności systemów IT, architektura mikroserwisowa stała się kluczowym podejściem w projektowaniu nowoczesnych rozwiązań informatycznych. Celem pracy jest opracowanie systemu, który integruje mikroserwisy z mechanizmem autentykacji opartym na tokenach JWT (JSON Web Token), przy jednoczesnym zapewnieniu bezpieczeństwa i wydajności.

Projekt obejmuje zaprojektowanie oraz implementację zespołu mikroserwisów, współpracujących za pośrednictwem interfejsów API. W skład systemu wchodzą następujące mikroserwisy:

* **auth-service** – odpowiedzialny za autentykację użytkowników oraz generowanie i weryfikację tokenów JWT,
* **user-service** – zajmujący się zarządzaniem kontami użytkowników, przechowywaniem danych profilowych oraz ich edycją,
* **admin-service** – zapewniający funkcje administracyjne, takie jak zarządzanie uprawnieniami, kontrolą dostępu i kontami administratorow,
* **mail-service** – obsługujący powiadomienia e-mail oraz komunikację z użytkownikami,
* **logger-service** – odpowiedzialny za rejestrowanie logów systemowych i monitorowanie działań,
* **flyway** – narzędzie do migracji bazy danych, wspierające wersjonowanie schematów,
* **vault** – serwis zarządzający kluczami i sekretami systemowymi,
* **postgresql** – relacyjna baza danych do przechowywania danych strukturalnych,
* **mongodb** – baza NoSQL do przechowywania danych niestrukturalnych,
* **mailhog** – narzędzie testowe do przechwytywania wiadomości e-mail.

Każdy z mikroserwisów został zaprojektowany w celu realizacji określonych funkcji, co umożliwia modularne podejście do budowy systemu. Praca skupi się na analizie komunikacji między mikroserwisami, implementacji bezpiecznych mechanizmów autentykacji oraz integracji komponentów w celu uzyskania wydajnego i bezpiecznego systemu.

Celem pracy jest szczegółowe przedstawienie procesu projektowania i implementacji architektury mikroserwisowej, z naciskiem na:

1. Implementację mechanizmów autentykacji z użyciem tokenów JWT.
2. Zarządzanie danymi w systemach rozproszonych (relacyjnymi i NoSQL).
3. Monitorowanie i logowanie działań systemowych.
4. Bezpieczne zarządzanie kluczami i sekretami.
5. Realizację migracji baz danych przy użyciu nowoczesnych narzędzi.

Praca składa się z części teoretycznej oraz praktycznej. Celem części teoretycznej jest przybliżenie podstawowych koncepcji związanych z architekturą mikroserwisową, zasadami bezpieczeństwa oraz technologiami zastosowanymi w projekcie. W części praktycznej skoncentrowano się na implementacji poszczególnych mikroserwisów oraz integracji systemu w spójną całość.

### 1.1 Zakres pracy

W ramach niniejszej pracy zaprojektowano i zaimplementowano mikroserwisy, które wspólnie tworzą system rozproszony. Każdy mikroserwis pełni specyficzną rolę, co pozwala na elastyczność i skalowalność systemu. Kluczowe aspekty realizacji obejmują:

* **Projektowanie mikroserwisów** z uwzględnieniem najlepszych praktyk architektonicznych.
* **Bezpieczna autentykacja użytkowników** przy pomocy tokenów JWT.
* **Integracja danych** w bazach relacyjnych (PostgreSQL) i niestrukturalnych (MongoDB).
* **Monitorowanie i logowanie** w celu zapewnienia pełnej obserwowalności systemu.
* **Zarządzanie kluczami i sekretami** w oparciu o nowoczesne narzędzia.

### 1.2 Zawartość pracy

* **Rozdział 2**: Omówienie teoretycznych podstaw mikroserwisów, tokenów JWT oraz zasad bezpieczeństwa.
* **Rozdział 3**: Proces projektowania systemu mikroserwisowego, komunikacja i integracja komponentów.
* **Rozdział 4**: Implementacja mikroserwisu odpowiedzialnego za autentykację.
* **Rozdział 5**: Zarządzanie danymi użytkowników i profilami.
* **Rozdział 6**: Bezpieczne zarządzanie danymi i kluczami w środowisku rozproszonym.
* **Rozdział 7**: Monitorowanie i logowanie w systemie.
* **Rozdział 8**: Wzorce projektowe i praktyki programistyczne w implementacji mikroserwisów.
* **Rozdział 9**: Podsumowanie wyników pracy, wnioski oraz możliwości dalszego rozwoju systemu.

Praca ta stanowi kompleksowe studium dotyczące tworzenia systemów mikroserwisowych, łącząc teorię z praktycznymi rozwiązaniami.

**Rozdział 2**

**Podłoże pracy**

Rozwój technologii informatycznych w ostatnich latach doprowadził do znacznych zmian w projektowaniu i implementacji systemów informatycznych. Rosnące wymagania użytkowników oraz potrzeba skalowalności i elastyczności przyczyniły się do popularyzacji architektury mikroserwisowej. W niniejszym rozdziale przedstawiono kontekst technologiczny i podstawy teoretyczne dla projektowanego systemu opartego na mikroserwisach, który wykorzystuje tokeny JWT oraz narzędzia takie jak Vault i Google Cloud KMS do zarządzania bezpieczeństwem.

**2.1 Mikroserwisy i ich rola w nowoczesnych systemach IT**

Architektura mikroserwisowa stała się dominującym wzorcem w projektowaniu aplikacji o wysokiej złożoności. W porównaniu do monolitycznych rozwiązań, mikroserwisy oferują:

* **Skalowalność:** Możliwość niezależnego skalowania poszczególnych usług w zależności od obciążenia.
* **Elastyczność:** Łatwość wprowadzania nowych funkcji i modyfikacji bez wpływu na inne elementy systemu.
* **Odporność na błędy:** Ograniczenie wpływu awarii jednego mikroserwisu na cały system.
* **Technologiczna różnorodność:** Możliwość wykorzystania różnych technologii i języków programowania do implementacji poszczególnych mikroserwisów.
* **Zwiększona produktywność:** Mniejsze zespoły developerskie mogą skupić się na rozwoju i utrzymaniu pojedynczych mikroserwisów.

Każdy mikroserwis jest odpowiedzialny za konkretny zestaw funkcji i komunikuje się z innymi komponentami za pomocą interfejsów API. W przypadku omawianego systemu, mikroserwisy takie jak auth-service, user-service, admin-service, logger-service czy mailer-service pełnią kluczowe role w zarządzaniu bezpieczeństwem, użytkownikami, monitorowaniem działania systemu oraz wysyłką emaili.

Mikroserwisy charakteryzują się również:

* **Komponentyzacją przez usługi:** Każdy mikroserwis jest niezależnie wymienialny i aktualizowalny, co zwiększa elastyczność systemu. (Microservices Architecture)
* **Organizacją wokół możliwości biznesowych:** Podział na mikroserwisy, takie jak auth-service czy user-service, odzwierciedla konkretne funkcje biznesowe. (Microservices Architecture)
* **Zdecentralizowanym zarządzaniem:** Zespoły mają większą autonomię w wyborze technologii i wdrażaniu zmian. (Microservices Architecture)
* **Zdecentralizowanym zarządzaniem danymi:** Każdy mikroserwis może zarządzać własną bazą danych, co pozwala na optymalizację wyboru technologii przechowywania danych. (Microservices Architecture)

Ten podział na małe, niezależne usługi przynosi szereg korzyści:

* **Zwinność:** Mikroserwisy ułatwiają zarządzanie poprawkami błędów i wydawanie nowych funkcji. Można zaktualizować usługę bez ponownego wdrażania całej aplikacji i wycofać aktualizację, jeśli coś pójdzie nie tak. (Microservices Architecture)
* **Małe, skoncentrowane zespoły:** Mikroserwis powinien być na tyle mały, aby jeden zespół mógł go zbudować, przetestować i wdrożyć. Małe zespoły zwiększają zwinność. (Microservices Architecture)
* **Odpowiednie narzędzie do zadania:** W architekturze mikroserwisowej zespoły mogą wybrać technologię, która najlepiej pasuje do danego mikroserwisu, co pozwala na wykorzystanie różnych stosów technologicznych. (Microservices Architecture)
* **Izolacja błędów:** Luźne powiązanie mikroserwisów zwiększa odporność aplikacji. Jeśli jeden z mikroserwisów stanie się niedostępny, nie wpłynie to na całą aplikację. (Microservices Architecture)
* **Skalowalność:** Mikroserwisy można skalować niezależnie, co pozwala na skalowanie tylko tych usług, które wymagają większych zasobów. (Microservices Architecture)
* **Izolacja danych:** Każdy mikroserwis ma własną bazę danych, co ułatwia aktualizację schematu i minimalizuje ryzyko wpływu na inne usługi. (Microservices Architecture)

**Typy architektur mikroserwisowych:**

* Architektura oparta na API Gateway: API Gateway pełni rolę centralnego punktu wejścia dla wszystkich żądań klientów, kierując je do odpowiednich mikroserwisów.
* Architektura oparta na Message Broker: Mikroserwisy komunikują się ze sobą za pomocą kolejki komunikatów (np. RabbitMQ, Kafka).
* Architektura hybrydowa: Połączenie różnych typów architektur w celu wykorzystania ich zalet.

**Zalety i wady architektury mikroserwisowej:**

Zalety:

* Lepsza skalowalność i elastyczność.
* Zwiększona odporność na awarie.
* Możliwość wykorzystania różnych technologii.
* Łatwiejsze wdrażanie i utrzymanie.

Wady:

* Zwiększona złożoność systemu.
* Wyzwania związane z komunikacją między mikroserwisami.
* Trudności w monitorowaniu i debugowaniu.

**Przykłady zastosowań w rzeczywistych systemach:**

* Netflix – Firma wykorzystuje mikroserwisy do obsługi ogromnej liczby użytkowników, umożliwiając dynamiczne skalowanie i wdrażanie nowych funkcji bez przestojów.
* Amazon – Mikroserwisowa architektura pomaga w zarządzaniu setkami tysięcy transakcji na sekundę, zapewniając wysoką dostępność i elastyczność w obsłudze klientów.
* Uber – System Ubera opiera się na mikroserwisach, co umożliwia szybkie dostosowywanie się do zmian rynkowych oraz integrację z różnymi systemami płatności i mapowania.
* Spotify – Platforma wykorzystuje mikroserwisy do zarządzania odtwarzaniem muzyki, rekomendacjami i interakcjami społecznymi, co pozwala na niezależny rozwój poszczególnych funkcjonalności (Exploring Microservices Architecture: A Comprehensive Guide).

**2.2 Tokeny JWT i ich zastosowanie**

Tokeny JWT (JSON Web Token) są powszechnie stosowane w architekturze mikroserwisowej jako mechanizm autoryzacji i autentykacji. W niniejszym systemie tokeny JWT są wykorzystywane do:

* **Autentykacji:** Potwierdzania tożsamości użytkowników i mikroserwisów.
* **Autoryzacji:** Zarządzania dostępem do zasobów i funkcji systemu na podstawie ról i uprawnień.
* **Bezpiecznej komunikacji:** Zapewnienia integralności i poufności danych przesyłanych pomiędzy mikroserwisami.

Tokeny są podpisywane kluczami kryptograficznymi przechowywanymi w usłudze Vault, co umożliwia ich weryfikację przez dowolny mikroserwis w systemie. Dzięki temu rozwiązaniu unika się konieczności centralizacji logiki autoryzacji.

**Struktura tokenów JWT:**

Token JWT składa się z trzech części oddzielonych kropkami:

1. **Header:** Zawiera informacje o typie tokenu i algorytmie szyfrowania.
2. **Payload:** Zawiera dane użytkownika lub mikroserwisu, takie jak identyfikator, rola, uprawnienia.
3. **Signature:** Podpis cyfrowy, który zapewnia integralność tokenu.

**Algorytmy szyfrowania:**

* HS256 (HMAC with SHA-256)
* RS256 (RSA Signature with SHA-256)
* ES256 (ECDSA Signature with SHA-256)

**Przykłady implementacji w różnych językach programowania:**

* **Go:** https://github.com/golang-jwt/jwt
* **Java:** https://github.com/auth0/java-jwt
* **Python:** https://pyjwt.readthedocs.io

**2.3 Zarządzanie API w mikroserwisach**

Zarządzanie API w architekturze mikroserwisowej jest kluczowe dla zapewnienia spójności, bezpieczeństwa i wydajności. Narzędzia i techniki zarządzania API obejmują:

* **Swagger/OpenAPI:** Tworzenie dokumentacji API i automatyczne generowanie kodu klienckiego.
* **API Gateway:** Centralny punkt wejścia dla wszystkich żądań, który może obsługiwać routing, uwierzytelnianie, autoryzację i caching.
  + **API Gateway:** pełni rolę pośrednika między klientami a mikroserwisami. Odpowiada za:
    - **Routing żądań:** Kierowanie żądań do odpowiednich mikroserwisów na podstawie adresu URL, nagłówków lub innych kryteriów.
    - **Uwierzytelnianie i autoryzacja:** Sprawdzanie tożsamości klienta i weryfikacja jego uprawnień do dostępu do zasobów.
    - **Buforowanie (caching):** Przechowywanie często używanych odpowiedzi, aby zmniejszyć obciążenie mikroserwisów i poprawić wydajność.
    - **Ograniczanie szybkości (rate limiting):** Kontrolowanie liczby żądań, które klient może wysłać w danym okresie, aby zapobiec przeciążeniu systemu.
    - **Transformacja żądań i odpowiedzi:** Modyfikowanie formatu lub zawartości żądań i odpowiedzi, aby dostosować je do potrzeb klienta lub mikroserwisu.
* **Service Mesh:** Warstwa infrastrukturalna, która zarządza komunikacją między mikroserwisami, zapewniając routing, load balancing i monitoring.

W omawianym systemie, dokumentacja API może być generowana za pomocą Swagger/OpenAPI, a API Gateway może być wykorzystany do centralnego zarządzania dostępem do mikroserwisów.

**2.4 Testowanie mikroserwisów**

Testowanie mikroserwisów jest niezbędne do zapewnienia ich jakości i niezawodności. Różne rodzaje testów obejmują:

* **Testy jednostkowe:** Testowanie pojedynczych funkcji lub modułów w obrębie mikroserwisu.
* **Testy integracyjne:** Testowanie interakcji między mikroserwisami.
* **Testy kontraktowe:** Weryfikacja zgodności interfejsów API między mikroserwisami.
* **Testy end-to-end:** Testowanie całego systemu z perspektywy użytkownika.

W omawianym systemie, należy zastosować kombinację różnych typów testów, aby zapewnić jego poprawne działanie.

**2.5 Zarządzanie kluczami i tajemnicami**

W systemie rozproszonym, bezpieczeństwo zależy w dużej mierze od efektywnego zarządzania kluczami kryptograficznymi i innymi tajemnicami (np. hasłami, tokenami dostępowymi). W tym celu wykorzystano narzędzie Vault, które pełni rolę centralnego systemu zarządzania tajemnicami.

* **Vault:** Przechowuje klucze prywatne, dane uwierzytelniające oraz polityki dostępu. Oferuje dynamiczne dostarczanie tajemnic na żądanie mikroserwisów.
* **Google Cloud KMS:** Służy do szyfrowania kluczy Vault, zapewniając ich ochronę na poziomie infrastruktury chmurowej.

Integracja Vault z Google Cloud KMS pozwala na:

* Bezpieczne przechowywanie kluczy unseal, wymaganych do uruchomienia usługi Vault.
* Automatyzację procesów zarządzania tajemnicami w środowisku Kubernetes.

**2.6 Infrastruktura oparta na Kubernetes i Terraform**

Mikroserwisy są wdrażane w klastrze Kubernetes, co zapewnia:

* **Orkiestrację:**
  + Kubernetes, a konkretnie GKE (Google Kubernetes Engine), umożliwia automatyczne zarządzanie cyklem życia kontenerów, ich skalowaniem i rozmieszczaniem. Dzięki deklaratywnemu podejściu Kubernetes, definiujemy pożądany stan aplikacji, a system dba o jego utrzymanie. (Kubernetes Documentation, GKE Documentation)
  + Orkiestracja obejmuje zarządzanie zasobami obliczeniowymi, sieciowymi i pamięcią masową, co jest kluczowe w dynamicznie zmieniającym się środowisku mikroserwisów.
* **Odporność:**
  + Kubernetes zapewnia mechanizmy samonaprawy, które automatycznie restartują uszkodzone kontenery, zastępują awarie węzłów i przywracają usługi do działania. (Kubernetes Documentation)
  + Dzięki rozproszeniu obciążenia i replikacji kontenerów, awaria pojedynczego komponentu nie powoduje przestoju całej aplikacji.

Do konfiguracji infrastruktury wykorzystano narzędzie Terraform, które umożliwia automatyczne tworzenie zasobów w chmurze, takich jak:

* **Klastry GKE (Google Kubernetes Engine):**
  + Terraform pozwala na zdefiniowanie i wdrożenie klastrów GKE za pomocą kodu, co zapewnia spójność i powtarzalność infrastruktury. (Terraform Google Cloud Provider Documentation)
  + Możliwość zarządzania konfiguracją klastra, taką jak liczba węzłów, typy maszyn i ustawienia sieciowe, ułatwia dostosowanie środowiska do potrzeb aplikacji.
* **Instancje Google Cloud KMS:**
  + Terraform umożliwia automatyczne tworzenie i zarządzanie instancjami Google Cloud KMS, co jest kluczowe dla bezpiecznego przechowywania kluczy kryptograficznych. (Terraform Google Cloud Provider Documentation)
  + Dzięki automatyzacji, proces zarządzania kluczami staje się mniej podatny na błędy ludzkie.
* **Sieci i reguły zapory ogniowej:**
  + Terraform pozwala na definiowanie i wdrażanie sieci wirtualnych oraz reguł zapory ogniowej, co zapewnia kontrolę nad ruchem sieciowym w klastrze. (Terraform Google Cloud Provider Documentation)
  + Definiowanie sieci i reguł zapory za pomocą kodu, zapewnia spójność środowiska pomiędzy różnymi etapami wdrażania.

**2.7 Uzasadnienie wyboru technologii**

Decyzja o wykorzystaniu technologii takich jak Vault, Google Cloud KMS, Terraform oraz Kubernetes wynika z ich szerokiego zastosowania w branży IT i dostosowania do wymagań nowoczesnych systemów:

* Bezpieczeństwo: Vault i Google Cloud KMS zapewniają ochronę kluczowych danych.
* Automatyzacja: Terraform redukuje ryzyko błędów ludzkich w procesie wdrażania infrastruktury.
* Skalowalność: Kubernetes umożliwia dynamiczne dostosowanie zasobów do obciążenia.

Ponadto, wybór tych technologii pomaga w radzeniu sobie z wyzwaniami architektury mikroserwisowej:

* **Złożoność:** Kubernetes upraszcza wdrażanie, skalowanie i zarządzanie wieloma mikroserwisami, zmniejszając złożoność operacyjną.
* **Komunikacja:** API Gateway ułatwia komunikację między klientami a mikroserwisami, zapewniając centralny punkt wejścia i obsługując routing, uwierzytelnianie i inne funkcje.
* **Skalowalność i elastyczność:** Kubernetes umożliwia automatyczne skalowanie mikroserwisów w zależności od obciążenia, co zapewnia elastyczność i wydajność systemu.

Dzięki zastosowaniu tych technologii możliwe jest stworzenie systemu, który nie tylko spełnia współczesne wymagania techniczne, ale również jest przygotowany na przyszłe rozszerzenia i zmiany.

### Rozdział 3: Analiza i Specyfikacja Wymagań

#### 3.1 Wymagania Funkcjonalne

Wymagania funkcjonalne definiują, co system powinien robić. W przypadku projektowanego systemu opartego na mikroserwisach, wymagania funkcjonalne obejmują:

* **Autentykacja i autoryzacja:**
  + System powinien umożliwiać użytkownikom rejestrację i logowanie za pomocą adresu e-mail i hasła.
  + System powinien obsługiwać tokeny JWT do autentykacji i autoryzacji użytkowników.
  + System powinien umożliwiać użytkownikom zmianę hasła.
  + System powinien obsługiwać różne role użytkowników (np. administrator, zwykły użytkownik) z różnymi uprawnieniami.
* **Zarządzanie użytkownikami:**
  + Administratorzy powinni móc tworzyć, odczytywać, aktualizować i usuwać konta użytkowników.
  + Użytkownicy powinni móc przeglądać i edytować swoje dane profilowe.
* **Komunikacja między mikroserwisami:**
  + Mikroserwisy powinny komunikować się ze sobą za pomocą gRPC.
  + Mikroserwisy powinny używać synchronicznego protokołu komunikacji.
* **Logowanie:**
  + System powinien rejestrować wszystkie istotne zdarzenia, takie jak logowania użytkowników, błędy systemowe i operacje administracyjne.
  + Logi powinny być przechowywane w centralnej bazie danych.
* **Wysyłanie e-maili:**
  + System powinien wysyłać e-maile do użytkowników w celu weryfikacji konta, resetowania hasła i powiadomień.
  + System powinien obsługiwać szablony e-maili.

#### 3.2 Wymagania Niefunkcjonalne

Wymagania niefunkcjonalne definiują, jak system powinien działać. W przypadku projektowanego systemu, wymagania niefunkcjonalne obejmują:

* **Wydajność:**
  + System powinien obsługiwać określoną liczbę żądań na sekundę.
  + Czas odpowiedzi systemu powinien być krótszy niż określona wartość.
* **Skalowalność:**
  + System powinien być skalowalny w poziomie, aby obsłużyć rosnący ruch.
  + Skalowanie powinno być automatyczne.
* **Bezpieczeństwo:**
  + System powinien chronić dane użytkowników przed nieautoryzowanym dostępem.
  + System powinien być odporny na ataki.
* **Dostępność:**
  + System powinien być dostępny przez określony procent czasu.
  + System powinien być odporny na awarie.
* **Utrzymanie:**
  + System powinien być łatwy w utrzymaniu i aktualizacji.
  + Kod powinien być czytelny i dobrze udokumentowany.

#### 3.3 Diagramy UML

##### Diagram Komponentów:

[Użytkownik] --> [API Gateway]

[API Gateway] --> [Auth Service]

[API Gateway] --> [User Service]

[API Gateway] --> [Admin Service]

[Auth Service] --> [Vault]

[User Service] --> [PostgreSQL]

[Admin Service] --> [PostgreSQL]

[Auth Service] --> [Logger Service]

[User Service] --> [Logger Service]

[Admin Service] --> [Logger Service]

[Mailer Service] --> [SMTP Server]

##### Diagram Sekwencji (Logowanie Użytkownika):

[Użytkownik] -> [Auth Service]: Żądanie logowania (email, hasło)

[Auth Service] -> [User Service]: Walidacja użytkownika (gRPC)

[User Service] -> [PostgreSQL]: Sprawdzenie danych użytkownika

[PostgreSQL] -> [User Service]: Zwrócenie danych użytkownika

[User Service] -> [Auth Service]: Zwrócenie wyniku walidacji

[Auth Service] -> [Vault]: Pobranie klucza do podpisania tokena

[Vault] -> [Auth Service]: Zwrócenie klucza

[Auth Service] -> [Auth Service]: Generowanie tokena JWT

[Auth Service] -> [Użytkownik]: Zwrócenie tokena JWT

#### 3.4 Scenariusze Użycia

##### Scenariusz 1: Rejestracja Użytkownika

1. Użytkownik wypełnia formularz rejestracyjny na stronie internetowej.
2. System wysyła żądanie rejestracji do user-service.
3. User-service waliduje dane i zapisuje nowego użytkownika w bazie danych PostgreSQL.
4. User-service wysyła e-mail weryfikacyjny do użytkownika za pośrednictwem mailer-service.
5. Użytkownik klika link weryfikacyjny w e-mailu, aby aktywować konto.

##### Scenariusz 2: Logowanie Użytkownika

1. Użytkownik wprowadza adres e-mail i hasło na stronie logowania.
2. System wysyła żądanie logowania do auth-service.
3. Auth-service deleguje walidację użytkownika do user-service za pomocą gRPC.
4. User-service sprawdza dane logowania w bazie danych PostgreSQL.
5. Jeśli dane są poprawne, auth-service generuje token JWT i zwraca go użytkownikowi.

##### Scenariusz 3: Zmiana Hasła

1. Użytkownik loguje się do systemu.
2. Użytkownik przechodzi do sekcji zmiany hasła w swoim profilu.
3. Użytkownik wprowadza stare i nowe hasło.
4. System wysyła żądanie zmiany hasła do user-service.
5. User-service waliduje dane i aktualizuje hasło użytkownika w bazie danych.

##### Scenariusz 4: Resetowanie Hasła

1. Użytkownik klika link "Zapomniałem hasła" na stronie logowania.
2. Użytkownik wprowadza swój adres e-mail.
3. System wysyła żądanie resetowania hasła do user-service.
4. User-service generuje token resetowania hasła i zapisuje go w bazie danych.
5. User-service wysyła e-mail z linkiem do resetowania hasła za pośrednictwem mailer-service.
6. Użytkownik klika link w e-mailu i wprowadza nowe hasło.
7. System aktualizuje hasło użytkownika w bazie danych.

##### Scenariusz 5: Dodanie Administratora

1. Administrator loguje się do systemu.
2. Administrator przechodzi do sekcji zarządzania administratorami.
3. Administrator wypełnia formularz z danymi nowego administratora.
4. System wysyła żądanie dodania administratora do admin-service.
5. Admin-service waliduje dane i zapisuje nowego administratora w bazie danych.

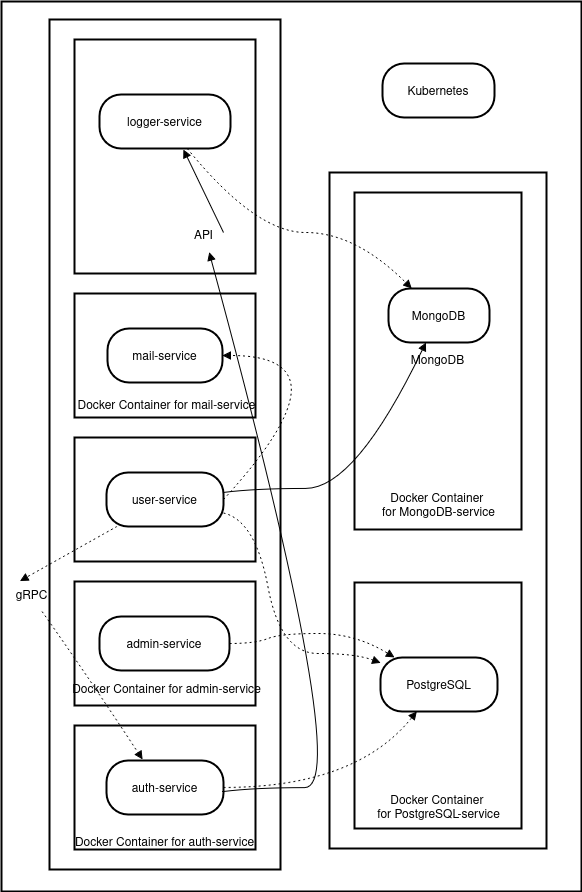
##### Scenariusz 6: Logowanie Administratora

1. Administrator wprowadza adres e-mail i hasło na stronie logowania.
2. System wysyła żądanie logowania do auth-service.
3. Auth-service deleguje walidację administratora do admin-service za pomocą gRPC.
4. Admin-service sprawdza dane logowania w bazie danych.
5. Jeśli dane są poprawne, auth-service generuje token JWT z rolą administratora i zwraca go użytkownikowi.

### 4. Projekt systemu

W tej części pracy szczegółowo opisano projekt systemu, obejmujący jego architekturę, specyfikację mikroserwisów, integrację z bazami danych oraz mechanizmy komunikacji między komponentami. Ponadto, omówione zostały narzędzia do zarządzania tajemnicami, takie jak Vault oraz Google Cloud KMS.

#### 4.1 Architektura systemu

System zaprojektowano w oparciu o architekturę mikroserwisową, co zapewnia wysoką elastyczność i skalowalność. Wszystkie komponenty systemu są autonomiczne i realizują określoną funkcjonalność, co umożliwia łatwe dodawanie nowych funkcji oraz modyfikowanie istniejących bez wpływu na inne części systemu. Architektura mikroserwisowa została wsparta przez technologie kontenerowe (Docker) oraz platformę orkiestracyjną Kubernetes, co zapewnia efektywne zarządzanie i skalowanie aplikacji w środowisku produkcyjnym.

Komunikacja pomiędzy mikroserwisami odbywa się głównie za pomocą protokołu gRPC, który pozwala na szybki i niezawodny transfer danych pomiędzy serwisami. Dodatkowo, w celu zapewnienia wysokiego poziomu bezpieczeństwa, wszystkie usługi są chronione za pomocą standardów uwierzytelniania JWT oraz mechanizmów autoryzacji opartych na rolach użytkowników.

#### 4.2 Specyfikacja mikroserwisów

Projekt systemu obejmuje następujące mikroserwisy, które pełnią określone funkcje w systemie:

##### 4.2.1 auth-service: logowanie, uwierzytelnianie, generowanie JWT

Serwis auth-service odpowiada za proces logowania użytkowników oraz generowanie tokenów JWT, które służą do autentykacji w innych mikroserwisach. Główne funkcje serwisu obejmują:

* **Logowanie użytkownika**: Użytkownik podaje swoje dane uwierzytelniające (np. nazwę użytkownika i hasło). Serwis weryfikuje te dane i, jeżeli są poprawne, generuje token JWT.
* **Dwuskładnikowa autoryzacja**: Dla administratorów serwis obsługuje dodatkowy etap weryfikacji tożsamości, w którym użytkownik musi podać kod wygenerowany przez aplikację mobilną lub otrzymany w wiadomości SMS.
* **Generowanie i walidacja tokenów JWT**: Serwis generuje tokeny JWT, które zawierają informacje o użytkowniku, jego uprawnieniach oraz czasie wygaśnięcia tokenu. Po wygenerowaniu, token jest zwracany użytkownikowi i wykorzystywany do dalszej autoryzacji w systemie.
* **Zarządzanie sesjami**: Serwis obsługuje mechanizm sesji, który umożliwia utrzymanie stanu zalogowanego użytkownika w systemie.

##### 4.2.2 user-service: zarządzanie użytkownikami

Serwis user-service odpowiedzialny jest za zarządzanie danymi użytkowników. W ramach tego serwisu realizowane są następujące funkcje:

* **Rejestracja użytkownika**: Nowi użytkownicy mogą założyć konto, podając dane osobowe oraz preferencje.
* **Zarządzanie profilami użytkowników**: Użytkownicy mogą edytować swoje dane, zmieniać hasła i aktualizować inne informacje.
* **Zarządzanie uprawnieniami**: Serwis obsługuje funkcje przypisywania ról i uprawnień użytkownikom w systemie.

Serwis komunikuje się z bazą danych PostgreSQL, w której przechowywane są dane strukturalne o użytkownikach, takie jak login, hasło (szyfrowane), e-mail oraz role.

##### 4.2.3 admin-service: funkcje administracyjne

Serwis admin-service jest odpowiedzialny za zarządzanie administracyjnymi funkcjami systemu. Jego główne zadania to:

* **Zarządzanie użytkownikami**: Administratorzy mają możliwość tworzenia nowych użytkowników, edytowania ich uprawnień, przypisywania ról oraz usuwania kont.
* **Weryfikacja tożsamości administratorów**: Serwis zapewnia bezpieczny proces logowania i autentykacji administratorów, który obejmuje dwuskładnikową weryfikację.
* **Zarządzanie uprawnieniami**: Serwis umożliwia przypisywanie specjalnych uprawnień do poszczególnych funkcji systemu, takich jak dostęp do wrażliwych danych lub możliwość konfiguracji serwisów.

Serwis komunikuje się zarówno z bazą danych PostgreSQL, jak i z innymi serwisami w systemie za pomocą gRPC.

##### 4.2.4 mail-service: powiadomienia e-mail

Serwis mail-service odpowiedzialny jest za wysyłanie powiadomień e-mail w różnych scenariuszach, takich jak:

* **Potwierdzenie rejestracji**: Użytkownicy otrzymują e-mail z potwierdzeniem utworzenia konta.
* **Przypomnienia o haśle**: Serwis wysyła wiadomości e-mail do użytkowników, którzy zapomnieli swoje hasło, zawierające instrukcje resetowania hasła.
* **Powiadomienia o zmianach**: Użytkownicy są informowani o wszelkich zmianach w systemie, które mogą mieć wpływ na ich dane lub bezpieczeństwo.

Serwis będzie wykorzystywał standardowe biblioteki do obsługi e-maila oraz zapewniał odpowiednią konfigurację dla różnych dostawców poczty.

##### 4.2.5 logger-service: logowanie i audyt

Serwis logger-service zapewnia centralne logowanie i audyt operacji wykonywanych w systemie. Główne zadania serwisu obejmują:

* **Zbieranie logów**: Wszystkie istotne operacje systemowe, takie jak logowanie, zmiany w danych użytkowników, błędy aplikacji, będą zapisywane w centralnym systemie logów.
* **Audyt**: Serwis umożliwia przechowywanie szczegółowych informacji o działaniach użytkowników i administratorów, co pozwala na audytowanie i analizowanie potencjalnych nieprawidłowości w systemie.
* **Bezpieczeństwo**: Logi będą szyfrowane i zabezpieczone przed nieautoryzowanym dostępem, a także zapewniona zostanie możliwość monitorowania i weryfikacji w czasie rzeczywistym.

#### 4.3 Integracja z bazami danych

System integruje się z dwoma głównymi bazami danych: PostgreSQL i MongoDB, które służą do przechowywania danych strukturalnych i niestrukturalnych.

##### 4.3.1 PostgreSQL – dane strukturalne

PostgreSQL będzie wykorzystywane do przechowywania danych strukturalnych, takich jak:

* **Dane użytkowników**: Informacje o użytkownikach, w tym dane kontaktowe, role i uprawnienia.
* **Dane logowania**: Hasła użytkowników (szyfrowane) oraz sesje.
* **Zarządzanie uprawnieniami**: Struktura ról i przypisanych uprawnień w systemie.

PostgreSQL zapewnia wysoką wydajność, integrację z różnymi językami programowania oraz zaawansowane mechanizmy zapytań, które są idealne do przechowywania danych użytkowników oraz zarządzania nimi w sposób wydajny i bezpieczny.

##### 4.3.2 MongoDB – dane niestrukturalne

MongoDB będzie używane do przechowywania danych niestrukturalnych, takich jak:

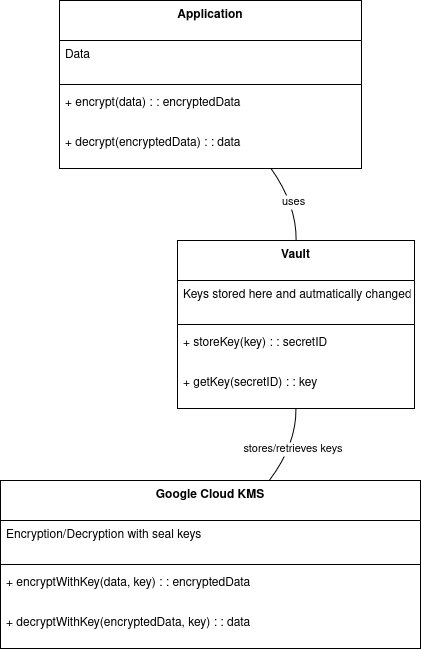
* **Logi operacji**: Zarejestrowane działania użytkowników oraz systemu.
* **Zawartość e-maili**: Dane powiązane z wiadomościami e-mail wysyłanymi przez system.
* **Inne dane o niskiej strukturze**: Dane, które mogą się dynamicznie zmieniać i nie wymagają sztywnej struktury relacyjnej.

MongoDB zapewnia elastyczność w przechowywaniu danych, co jest przydatne w przypadku przechowywania logów, wiadomości e-mail i innych dynamicznych danych.

#### 4.4 Mechanizmy komunikacji między mikroserwisami

Mikroserwisy w systemie komunikują się ze sobą za pomocą gRPC, co zapewnia wysoką wydajność i niskie opóźnienia. Każdy mikroserwis udostępnia swoje API, które jest wykorzystywane przez inne serwisy do wymiany danych. Dodatkowo, komunikacja pomiędzy mikroserwisami jest zabezpieczona za pomocą protokołów SSL/TLS, co zapewnia poufność danych.

#### 4.5 Wykorzystanie Vault i Google Cloud KMS do zarządzania tajemnicami

Aby zapewnić bezpieczeństwo przechowywanych danych i kluczy, system integruje się z narzędziami do zarządzania tajemnicami, takimi jak Vault i Google Cloud KMS. Te narzędzia umożliwiają:

* **Bezpieczne przechowywanie kluczy i haseł**: Vault i Google Cloud KMS przechowują wszystkie tajemnice systemowe, takie jak klucze szyfrujące oraz hasła, w bezpieczny sposób.
* **Zarządzanie dostępem**: System zarządza dostępem do tych tajemnic na podstawie ról i uprawnień, co pozwala na precyzyjne kontrolowanie, które usługi mogą uzyskiwać dostęp do wrażliwych danych.
* **Szyfrowanie danych**: Narzędzia te zapewniają zaawansowane mechanizmy szyfrowania danych zarówno w trakcie przechowywania, jak i przesyłania, co zwiększa poziom bezpieczeństwa systemu.

### 5. Implementacja

W tej części pracy omówiono szczegóły implementacji systemu, obejmujące środowisko deweloperskie, konfigurację infrastruktury, proces wdrażania mikroserwisów na Google Kubernetes Engine (GKE), oraz szczegóły implementacji wybranych mikroserwisów. Dodatkowo, zaprezentowano mechanizmy testowania i debugowania, które zostały wdrożone w celu zapewnienia wysokiej jakości kodu oraz stabilności systemu.

#### 5.1 Opis środowiska deweloperskiego

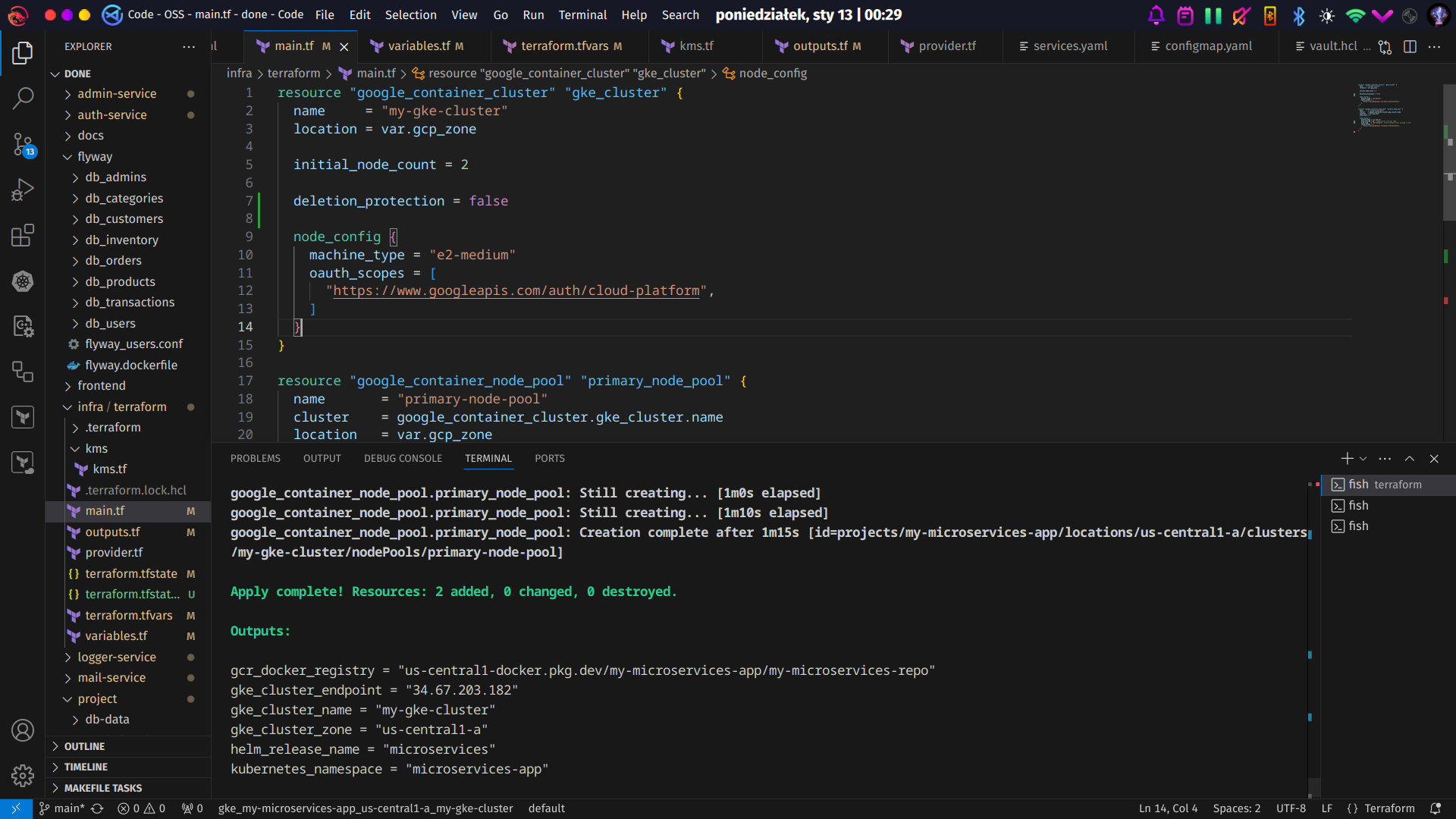
Środowisko deweloperskie dla systemu zostało zaprojektowane z myślą o wykorzystaniu narzędzi i technologii wspierających efektywną pracę zespołową oraz umożliwiających łatwe i szybkie wdrażanie kodu. W projekcie zastosowano następujące technologie:

* **Języki programowania**: Główne mikroserwisy zostały napisane w języku Go, który jest uznawany za szybki i skalowalny, szczególnie w kontekście mikroserwisów. Dodatkowo, dla niektórych mikroserwisów, takich jak mail-service, użyto języka Python, ze względu na jego popularność wśród bibliotek do obsługi e-maili.
* **Konteneryzacja**: Docker został wykorzystany do konteneryzacji wszystkich mikroserwisów. Każdy serwis został zapakowany w osobny kontener, co zapewnia izolację środowiskową oraz umożliwia ich łatwe uruchamianie na różnych maszynach.
* **Orkiestracja**: Do zarządzania kontenerami i wdrażania mikroserwisów na środowisku produkcyjnym użyto Google Kubernetes Engine (GKE). GKE zapewnia automatyczne skalowanie, zarządzanie i monitorowanie mikroserwisów w ramach klastra Kubernetes.
* **Repozytorium kodu**: Wszystkie pliki projektowe są przechowywane w systemie Git, z użyciem platformy GitHub, co umożliwia wersjonowanie kodu oraz łatwą współpracę w zespole deweloperskim.
* **Baza danych**: W projekcie użyto dwóch baz danych: PostgreSQL do przechowywania danych strukturalnych oraz MongoDB do przechowywania danych niestrukturalnych. Obie bazy danych są zarządzane w kontenerach i są dostępne dla odpowiednich mikroserwisów.

## 5.2 Konfiguracja infrastruktury z użyciem Terraform

Do automatyzacji konfiguracji infrastruktury w chmurze oraz zapewnienia powtarzalności i skalowalności środowiska produkcyjnego, wykorzystano narzędzie Terraform. Dzięki niemu możliwe było zdefiniowanie zasobów chmurowych w sposób deklaratywny, a następnie ich automatyczne uruchomienie.

Poniższy zrzut ekranu przedstawia fragment kodu Terraform odpowiedzialny za tworzenie

klastra Kubernetes w Google Kubernetes Engine (GKE):

Jak widać na powyższym screenie, kod Terraform definiuje zasoby takie jak:

* google\_container\_cluster: Tworzy klaster GKE o nazwie "my-gke-cluster" w strefie "us-central1-a". Parametry takie jak initial\_node\_count i machine\_type określają początkową liczbę węzłów w klastrze oraz typ maszyn wirtualnych używanych dla węzłów.
* google\_container\_node\_pool: Definiuje pulę węzłów "primary-node-pool" dla klastra GKE.

Podstawowe elementy infrastruktury skonfigurowane za pomocą Terraform to:

**Google Cloud Platform:**

* **Google Kubernetes Engine (GKE):** Terraform automatycznie tworzy klastry GKE, konfiguruje pulę węzłów, ustawia wymagane zasoby obliczeniowe oraz zapewnia odpowiednią konfigurację sieci. Umożliwia to łatwe i szybkie uruchomienie środowiska Kubernetes w chmurze.
* **Zasoby GCP:** Terraform zarządza również innymi zasobami GCP, takimi jak bazy danych (PostgreSQL i MongoDB), przechowywanie tajemnic (Vault), Load Balancer oraz reguły firewallowe. Dzięki temu cała infrastruktura, nie tylko Kubernetes, jest zarządzana w sposób spójny i zautomatyzowany.

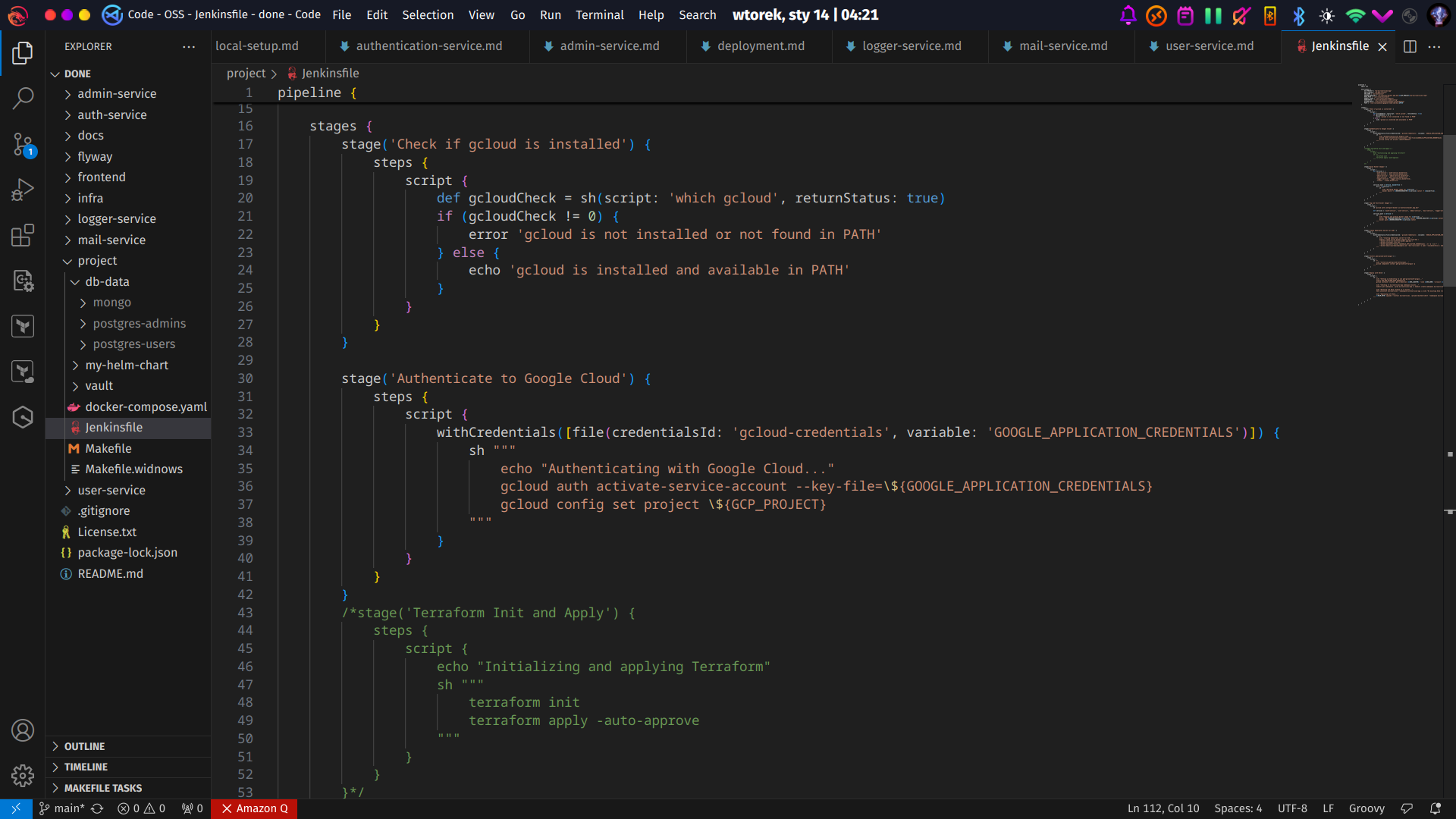
**Kubernetes:**

* **Zasoby Kubernetes:** Terraform może automatyzuowac konfigurację deploymentów, serwisów, podów oraz innych zasobów Kubernetes, takich jak ConfigMaps, Secrets i Persistent Volumes. Umożliwia to precyzyjne zdefiniowanie środowiska uruchomieniowego dla aplikacji i zapewnia spójność konfiguracji w różnych środowiskach (np. developerskim, testowym i produkcyjnym). W niniejszym projekcie, do zarządzania zasobami Kubernetes wykorzystano narzędzie Helm I Jenkins.

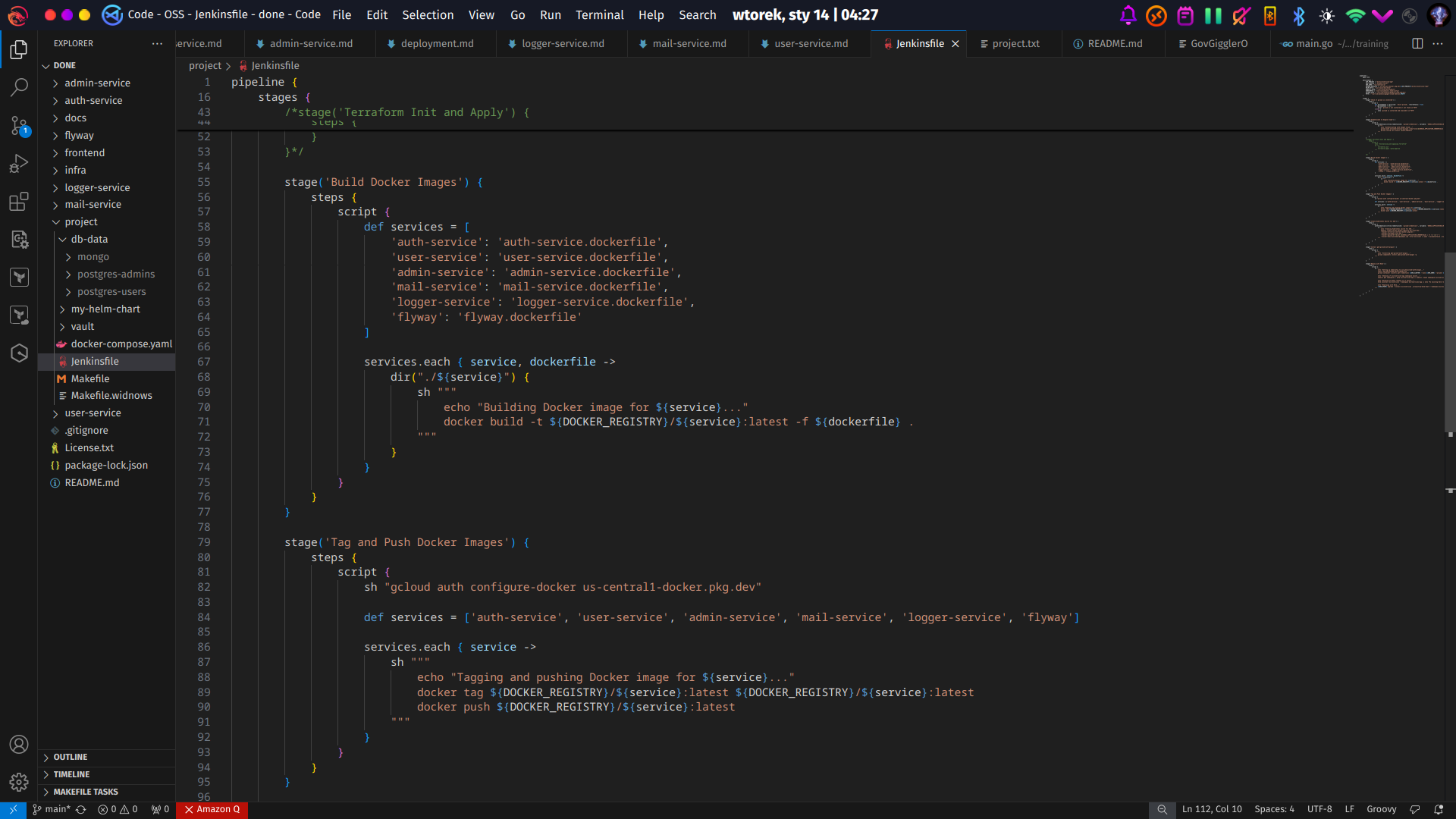
Dzięki temu podejściu infrastruktura jest **skalowalna, powtarzalna i łatwa w zarządzaniu**. Zmiany w konfiguracji mogą być łatwo wprowadzane i wdrażane za pomocą Terraform, bez ryzyka wystąpienia niekompatybilności lub błędów związanych z ręczną konfiguracją. Terraform zapewnia również **wersjonowanie infrastruktury**, co umożliwia śledzenie zmian i przywracanie poprzednich stanów w razie potrzeby.

## 5.3 Proces wdrażania mikroserwisów na GKE

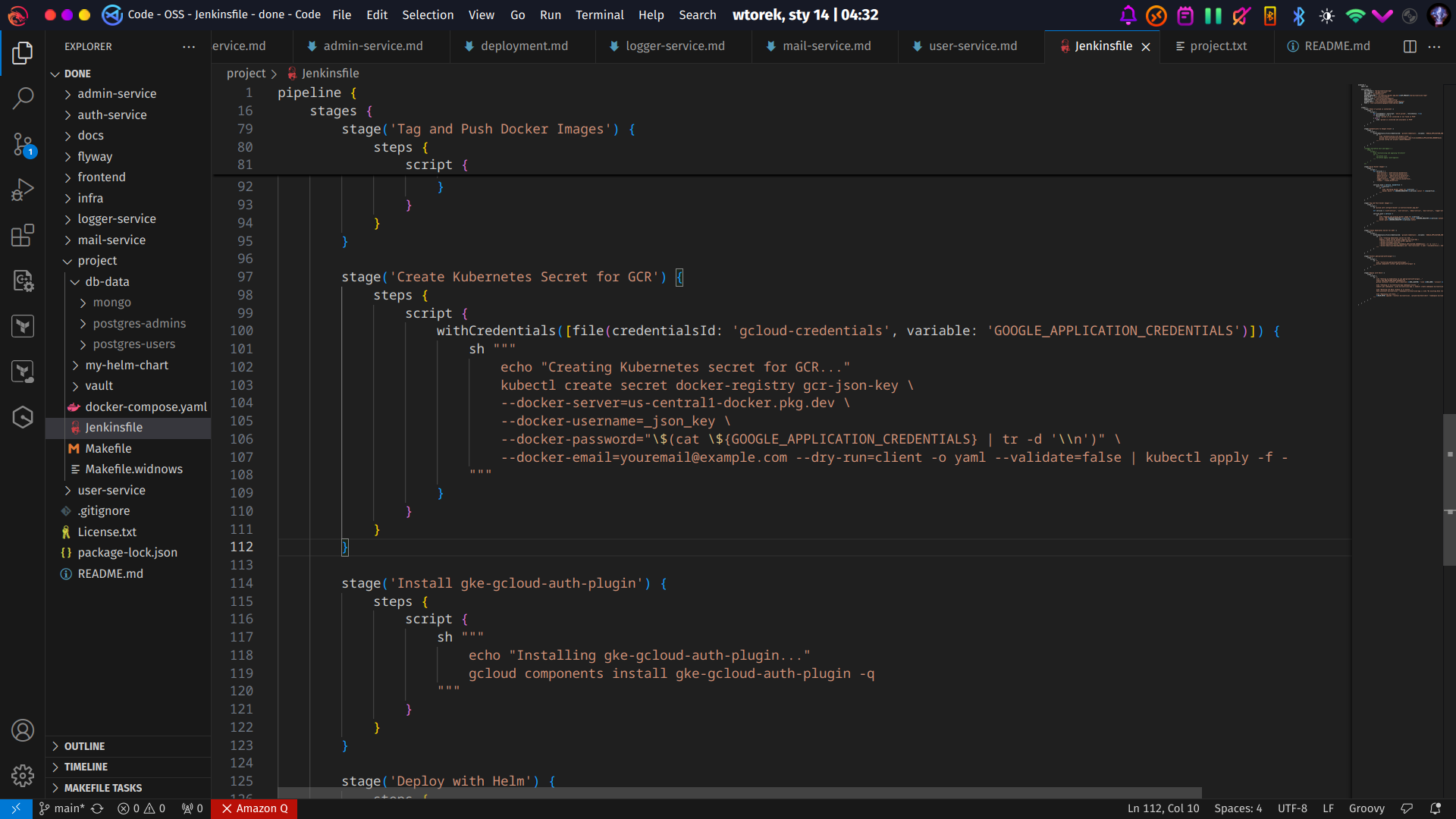
Wdrażanie mikroserwisów na Google Kubernetes Engine (GKE) to wieloetapowy proces zautomatyzowany przy użyciu narzędzi CI/CD, takich jak Jenkins. W niniejszym projekcie wykorzystano Jenkins Pipeline do zdefiniowania i orkiestracji całego procesu wdrażania. Poniżej przedstawiono szczegółowy opis kolejnych etapów pipeline'u, wraz z fragmentami kodu ilustrującymi ich implementację:



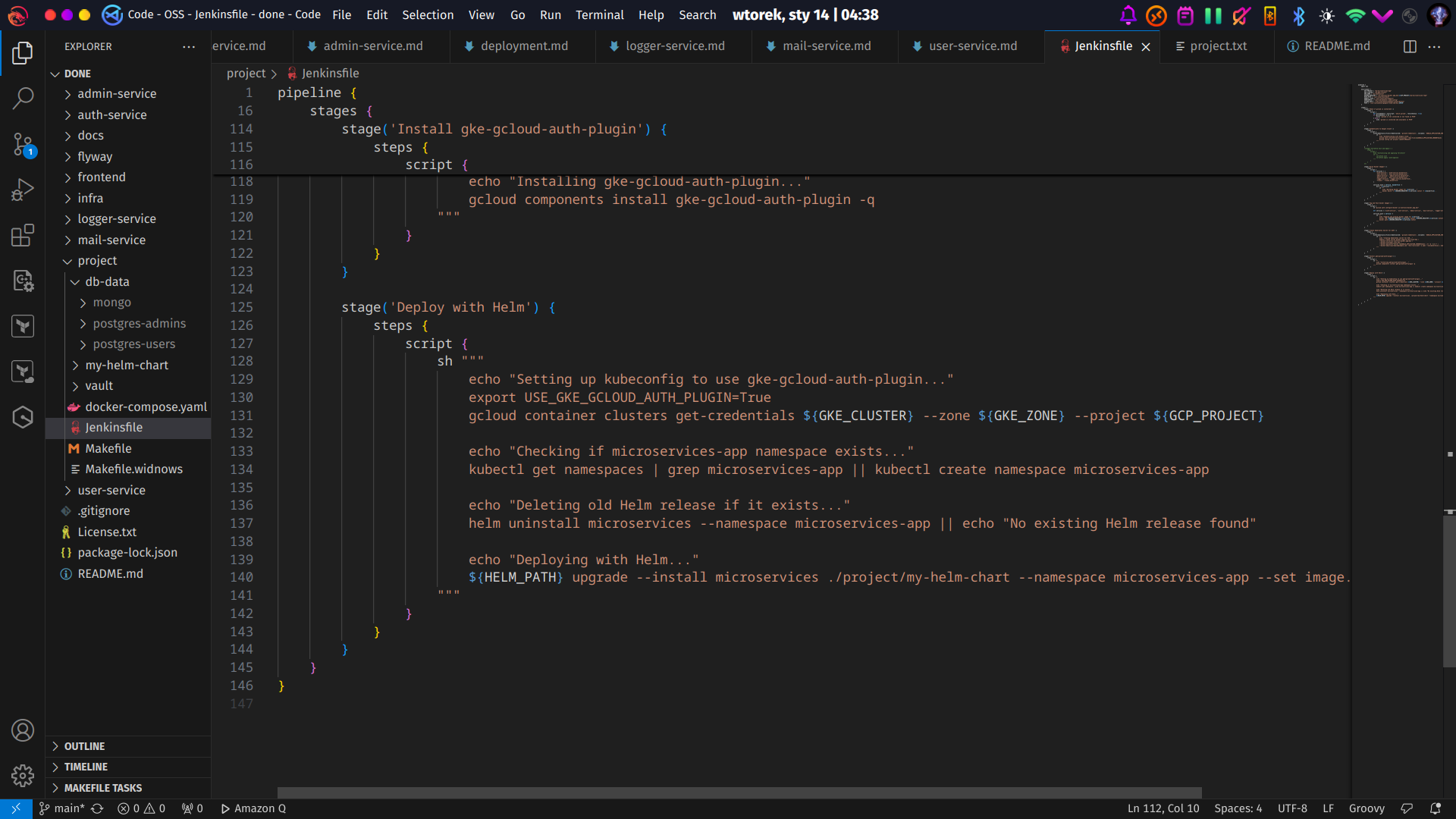
Pipeline rozpoczyna się od sprawdzenia dostępności narzędzia gcloud oraz uwierzytelnienia w Google Cloud Platform. Wykorzystywane są do tego celu zmienne środowiskowe zdefiniowane na początku skryptu, takie jak GCP\_PROJECT, GKE\_CLUSTER i GKE\_ZONE. Uwierzytelnienie odbywa się za pomocą klucza serwisowego przechowywanego w Jenkins Credentials.



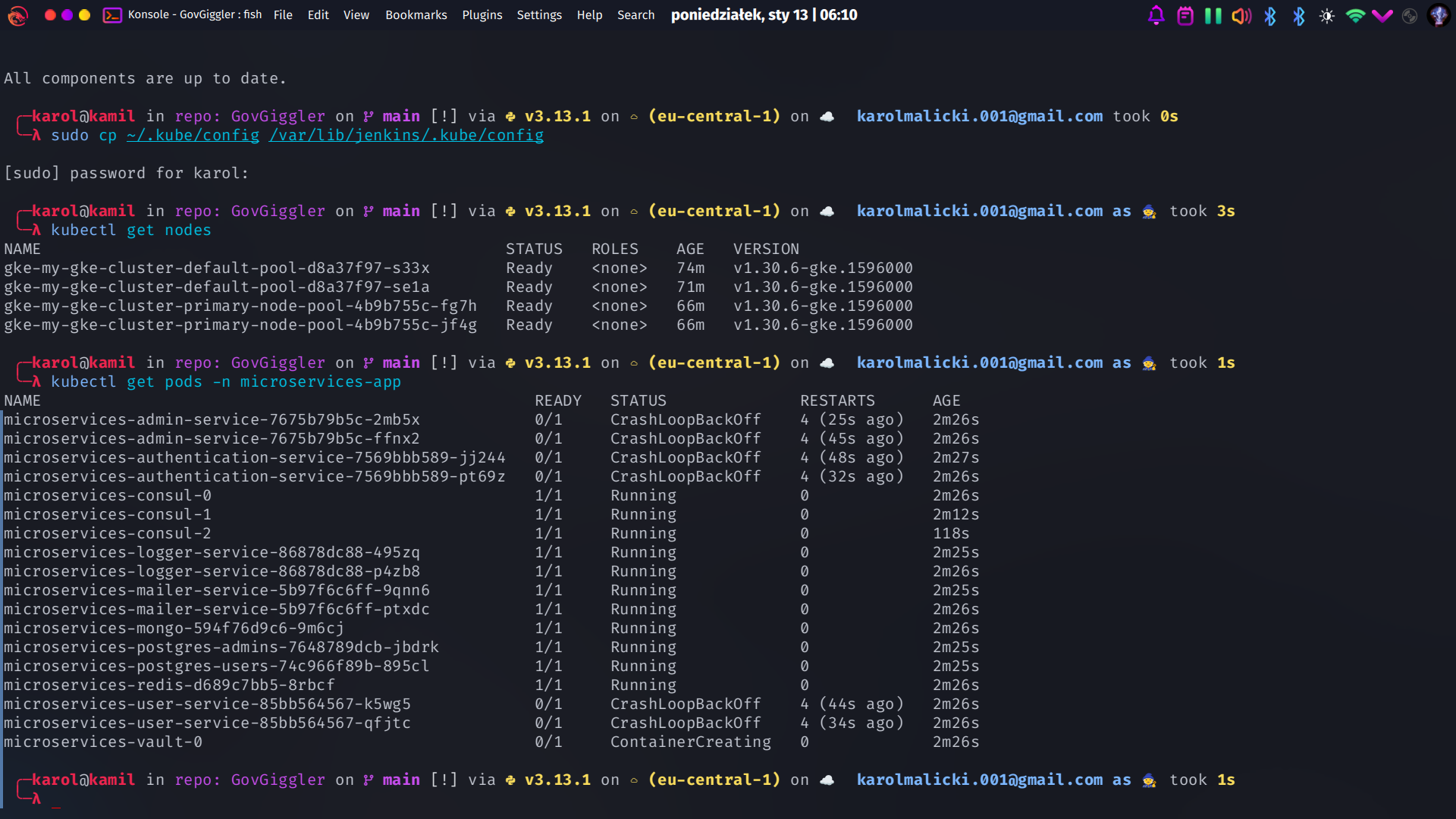
**W kolejnym etapie pipeline'u budowane są obrazy Docker dla każdego mikroserwisu. Definicje usług i odpowiadających im plików Dockerfile są przechowywane w zmiennej** services**. Po zbudowaniu, obrazy są tagowane i przesyłane do Google Container Registry (GCR).**



Przed wdrożeniem aplikacji na klaster Kubernetes, tworzony jest sekret przechowujący dane uwierzytelniające do GCR. Umożliwia to pobieranie obrazów przez pody w klastrze. Dodatkowo instalowany jest plugin gke-gcloud-auth-plugin, który upraszcza uwierzytelnianie w GKE.



Ostatnim etapem pipeline'u jest wdrożenie aplikacji na klaster Kubernetes za pomocą narzędzia Helm. Helm umożliwia zarządzanie aplikacjami w Kubernetesie poprzez definiowanie szablonów (chartów) opisujących zasoby aplikacji. W tym przypadku wykorzystywany jest chart ./project/my-helm-chart, który zawiera definicje deploymentów, serwisów i innych zasobów Kubernetes dla mikroserwisów.

Illustration 1: Sprawdzanie statusu podów w klastrze Kubernetes.

#### 5.4 Szczegóły implementacji wybranych mikroserwisów

##### 5.4.1 auth-service: uwierzytelnianie i zarządzanie tokenami

Mikroserwis auth-service odpowiada za proces logowania, uwierzytelniania oraz zarządzanie tokenami JWT. Implementacja opiera się na frameworku Go, który umożliwia łatwe tworzenie bezpiecznych aplikacji.

1. **Logowanie użytkownika**: Podczas logowania użytkownik podaje dane (login i hasło), które są sprawdzane w bazie danych. Po pomyślnej weryfikacji, generowany jest token JWT, który zawiera informacje o użytkowniku oraz jego uprawnieniach.
2. **Generowanie tokenów**: Token JWT jest podpisywany przy użyciu sekretnego klucza, który jest przechowywany w bezpiecznym magazynie, takim jak Google Cloud KMS lub Vault.
3. **Autoryzacja**: Każde żądanie do innych mikroserwisów zawiera token JWT w nagłówku. Serwisy te weryfikują tokeny i na tej podstawie udzielają dostępu lub odmawiają wykonania operacji.
4. **Bezpieczeństwo**: Zastosowano dwuskładnikowe uwierzytelnianie dla administratorów, które wymaga podania zarówno hasła, jak i jednorazowego kodu z aplikacji mobilnej.

##### 5.4.2 logger-service: mechanizmy audytowe

Mikroserwis logger-service odpowiedzialny jest za logowanie i audyt operacji wykonywanych w systemie. Implementacja loggera została oparta na standardowych bibliotekach Go oraz popularnych rozwiązaniach do logowania, takich jak logrus.

1. **Rejestracja operacji**: Wszystkie istotne operacje, takie jak logowanie użytkowników, zmiany w danych użytkowników oraz działania administratorów, są rejestrowane w centralnym systemie logów.
2. **Formatowanie logów**: Logi są zapisywane w formacie JSON, co umożliwia łatwą obróbkę i analizę danych.
3. **Bezpieczeństwo logów**: Logi zawierają dane o użytkownikach, które są wrażliwe. Z tego powodu logi są szyfrowane i przechowywane w bazie danych MongoDB.
4. **Audyt**: Serwis umożliwia przechowywanie pełnej historii działań, co pozwala na przeprowadzenie audytu i identyfikację potencjalnych problemów lub nieprawidłowości w systemie.

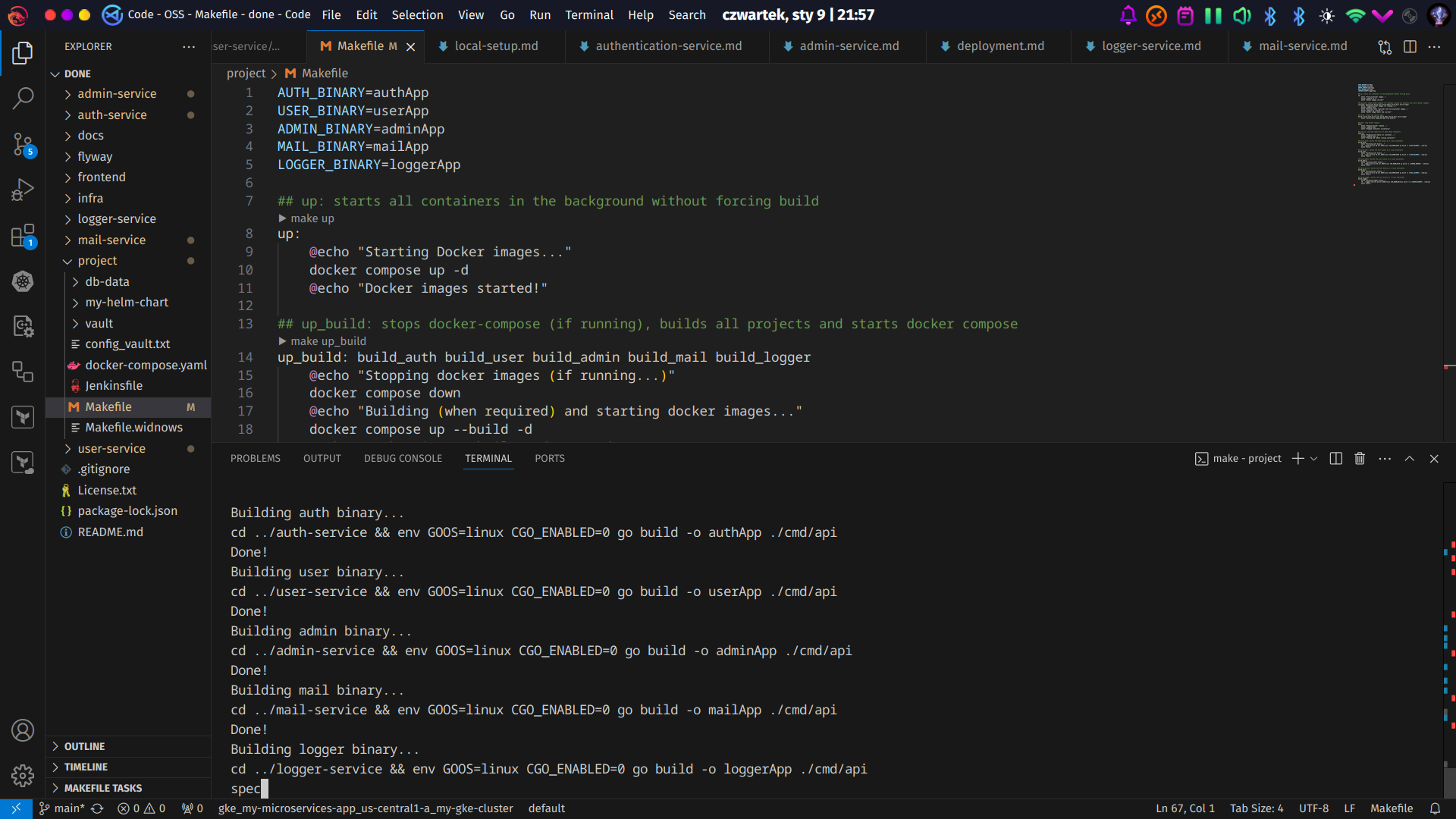
#### 5.5 Mechanizmy testowania i debugowania

Testowanie i debugowanie aplikacji są kluczowe dla zapewnienia niezawodności systemu. W projekcie zastosowano szereg technik i narzędzi w celu zapewnienia jakości kodu:

* **Testy jednostkowe**: Każdy mikroserwis posiada zestaw testów jednostkowych, które sprawdzają podstawowe funkcjonalności serwisów. Testy są napisane w Go, a narz ędzie Go testing jest używane do ich automatycznego uruchamiania.
* **Testy integracyjne**: Testy integracyjne sprawdzają interakcje pomiędzy mikroserwisami oraz z bazą danych. Zapewniają one, że poszczególne komponenty systemu współpracują ze sobą poprawnie.
* **Testowanie obciążeniowe**: Testy obciążeniowe są przeprowadzane na środowisku stagingowym w celu sprawdzenia, jak system radzi sobie z dużą liczbą równoczesnych żądań. Narzędzie Apache Jmeter zostało wykorzystane do symulacji obciążenia.
* **Debugowanie**: W trakcie rozwoju systemu używano narzędzi takich jak go-lint, go run, oraz systemu logów w Kubernetesie do szybkiego wykrywania i naprawiania błędów.

**Zdjęcie 1: Proces budowania i uruchamiania aplikacji z wykorzystaniem Makefile i Docker Compose**

Przedstawiony zrzut ekranu ilustruje proces budowania aplikacji w środowisku developerskim z użyciem narzędzia Makefile. Widoczny jest terminal, w którym wykonywane są komendy budujące poszczególne mikroserwisy aplikacji. Każdy mikroserwis jest kompilowany do osobnego binarium z wykorzystaniem polecenia go build, co jest typowym podejściem w projektach napisanych w języku Go, umożliwiającym niezależne wdrażanie i skalowanie poszczególnych komponentów.

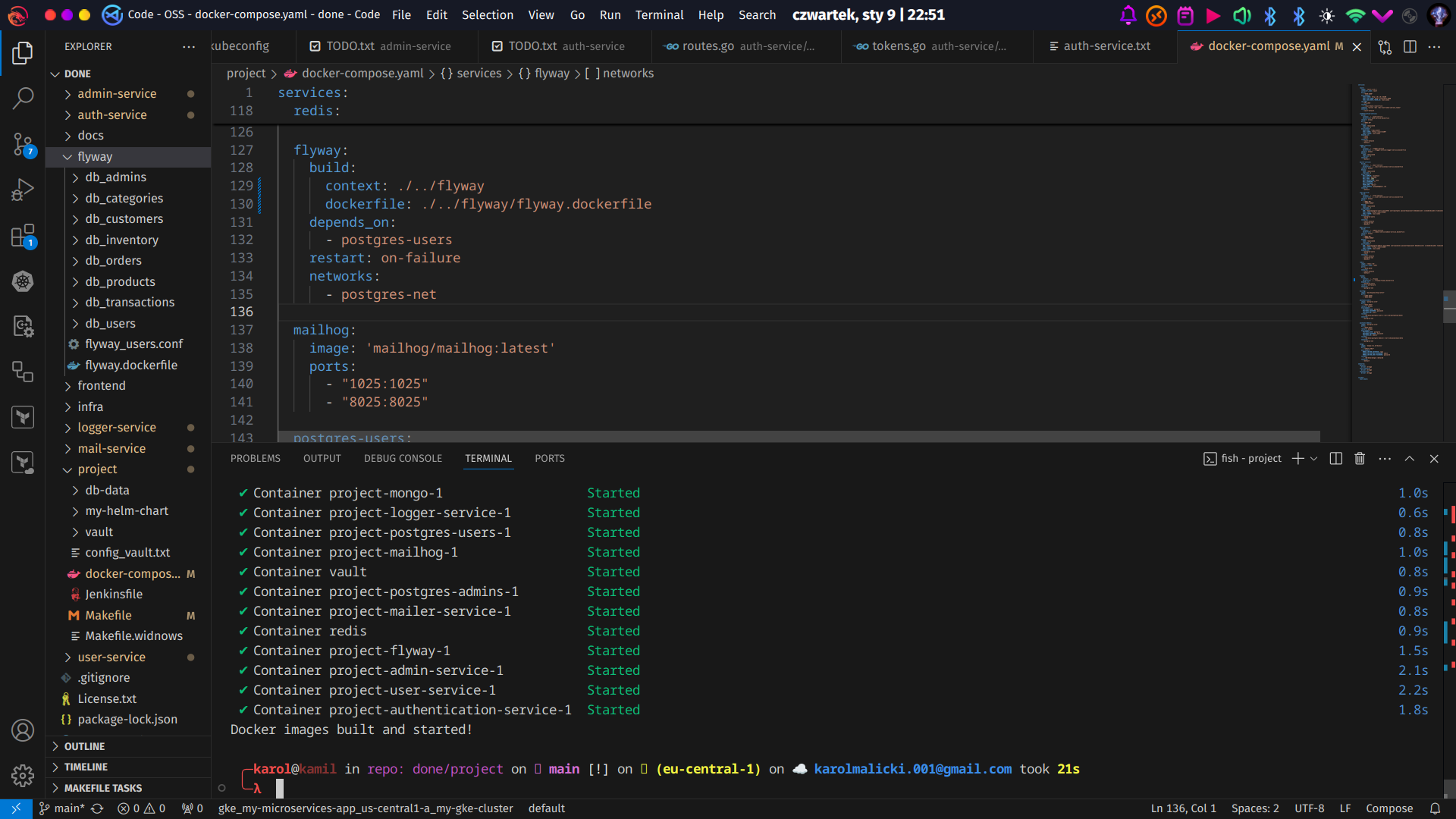
Makefile jest plikiem tekstowym zawierającym instrukcje dla narzędzia make, które automatyzuje proces budowania aplikacji. W prezentowanym przypadku Makefile definiuje kroki kompilacji dla każdego z mikroserwisów: auth-service, user-service, admin-service, mail-service oraz logger-service. Takie podejście pozwala na zautomatyzowanie i standaryzację procesu budowania, co jest szczególnie istotne w przypadku projektów opartych na architekturze mikroserwisowej, gdzie każdy serwis jest budowany i wdrażany niezależnie.

Budowanie każdego mikroserwisu do osobnego binarium jest kluczowe dla architektury mikroserwisowej, ponieważ pozwala to na niezależne wdrażanie i skalowanie poszczególnych komponentów aplikacji, co zwiększa elastyczność i odporność systemu na awarie. Wykorzystanie języka Go do implementacji mikroserwisów jest również istotnym aspektem, ponieważ Go to język programowania charakteryzujący się wysoką wydajnością, prostotą i wbudowanymi mechanizmami współbieżności, co czyni go odpowiednim narzędziem do tworzenia mikroserwisów.

Poza budowaniem, Makefile ułatwia również uruchamianie aplikacji w środowisku developerskim. W tym celu wykorzystuje się narzędzie Docker Compose, które zarządza konteneryzacją poszczególnych mikroserwisów. Plik docker-compose.yml definiuje konfigurację każdego kontenera, w tym obraz Dockera, porty, zależności i zmienne środowiskowe.

Komenda make up\_build w Makefile'u odpowiada za zbudowanie wszystkich mikroserwisów i uruchomienie ich w kontenerach Dockera za pomocą Docker Compose. Proces ten obejmuje następujące kroki:

1. **Zatrzymanie działających kontenerów:** docker compose down
2. **Budowanie obrazów Docker:** docker compose build
3. **Uruchomienie kontenerów w tle:** docker compose up -d

Dzięki zastosowaniu Makefile i Docker Compose, proces developmentu i wdrażania aplikacji opartych na mikroserwisach jest znacznie uproszczony i bardziej efektywny. Automatyzacja i modularność tego procesu przyczyniają się do zwiększenia wydajności pracy i ułatwiają zarządzanie złożonymi systemami.

## 6. Zarządzanie bezpieczeństwem

Bezpieczeństwo jest kluczowym elementem współczesnych systemów informatycznych, szczególnie w kontekście architektury mikroserwisów, która wiąże się z wieloma wyzwaniami związanymi z zarządzaniem danymi, uwierzytelnianiem użytkowników oraz

**6.1 Bezpieczne przechowywanie i dystrybucja kluczy**

Jednym z najistotniejszych elementów systemu zabezpieczeń w nowoczesnych aplikacjach jest zarządzanie kluczami kryptograficznymi. Klucze te są niezbędne do przechowywania danych w sposób bezpieczny, zapewniania integralności komunikacji oraz autoryzacji użytkowników i usług. W kontekście omawianego systemu zarządzania mikroserwisami, kluczowym komponentem odpowiedzialnym za przechowywanie i dystrybucję kluczy jest **Vault**, który działa jako centralne repozytorium dla tajemnic (secrets) i kluczy kryptograficznych.

Vault, wykorzystując Google Cloud KMS (Key Management System), zapewnia bezpieczne przechowywanie kluczy, takich jak klucze do szyfrowania danych, klucze do podpisywania JWT (JSON Web Token) oraz klucze służące do bezpiecznej komunikacji między mikroserwisami. Zastosowanie Google Cloud KMS w połączeniu z Vault pozwala na automatyczne zarządzanie cyklem życia kluczy, ich rotację oraz zapewnienie dostępu tylko do uprawnionych użytkowników lub mikroserwisów.

Wszystkie tajemnice przechowywane w Vault są szyfrowane, a dostęp do nich jest ściśle kontrolowany za pomocą polityk dostępu, które mogą być dostosowane do konkretnych wymagań bezpieczeństwa. Dzięki temu możliwe jest zachowanie pełnej kontroli nad dostępem do wrażliwych danych w systemie oraz zapewnienie, że tylko odpowiednie usługi i użytkownicy mogą uzyskać dostęp do kluczy w celu ich użycia.

Dodatkowo, system oparty na Kubernetes w połączeniu z Vault pozwala na dynamiczne dostarczanie kluczy do mikroserwisów w czasie rzeczywistym, eliminując potrzebę przechowywania kluczy w konfiguracjach serwisów. Dzięki temu ryzyko wycieku kluczy w przypadku kompromitacji poszczególnych serwisów zostaje znacznie zminimalizowane.

**6.2 Implementacja uwierzytelniania i autoryzacji z JWT**

Uwierzytelnianie i autoryzacja użytkowników oraz mikroserwisów to fundament bezpieczeństwa w aplikacjach opartych na architekturze mikroserwisów. W niniejszym projekcie mechanizm ten opiera się na JSON Web Tokenach (JWT), które umożliwiają bezpieczną i efektywną wymianę informacji pomiędzy usługami a użytkownikami.

**6.2.1 Uwierzytelnianie użytkowników i administratorów**

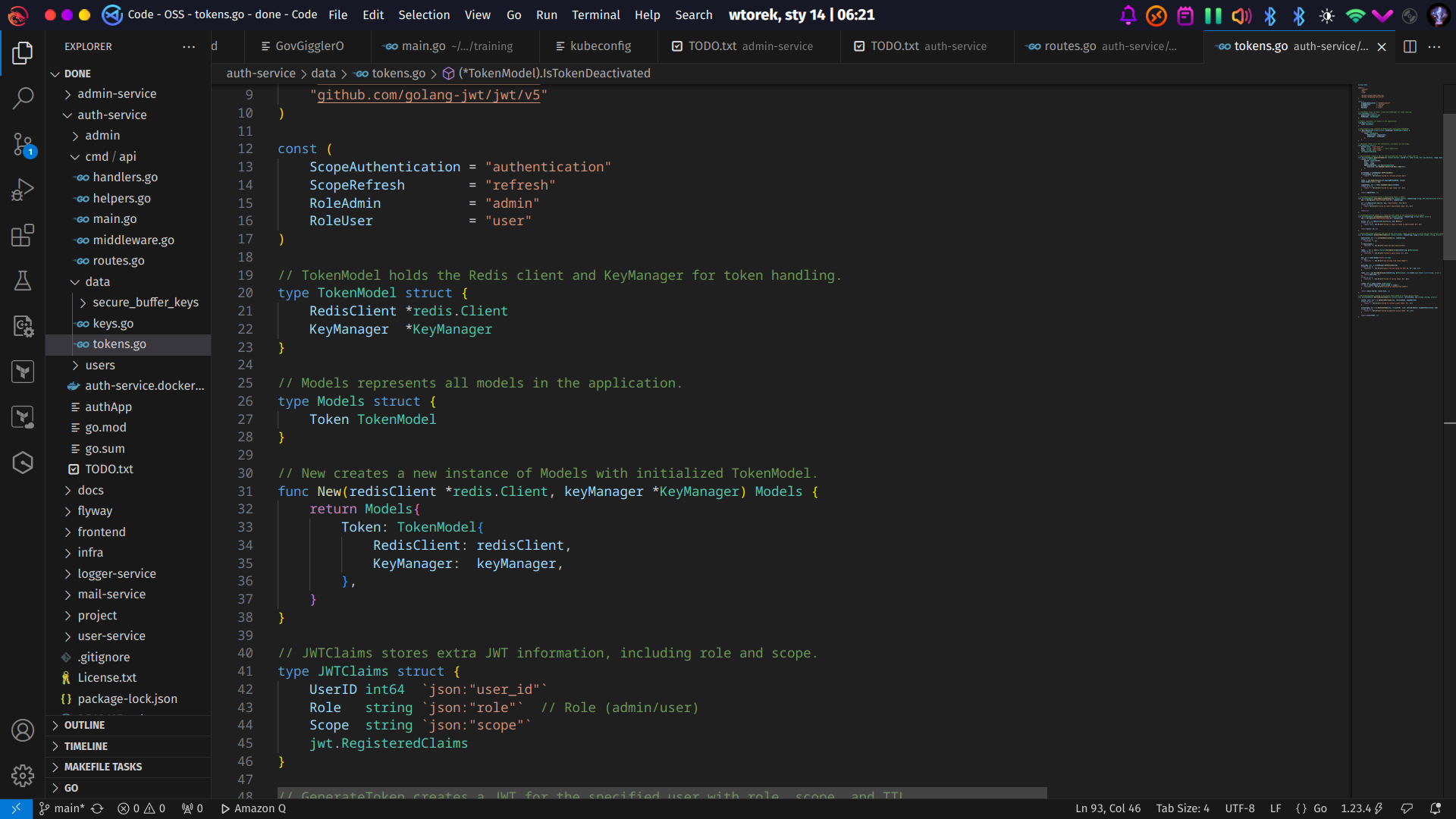
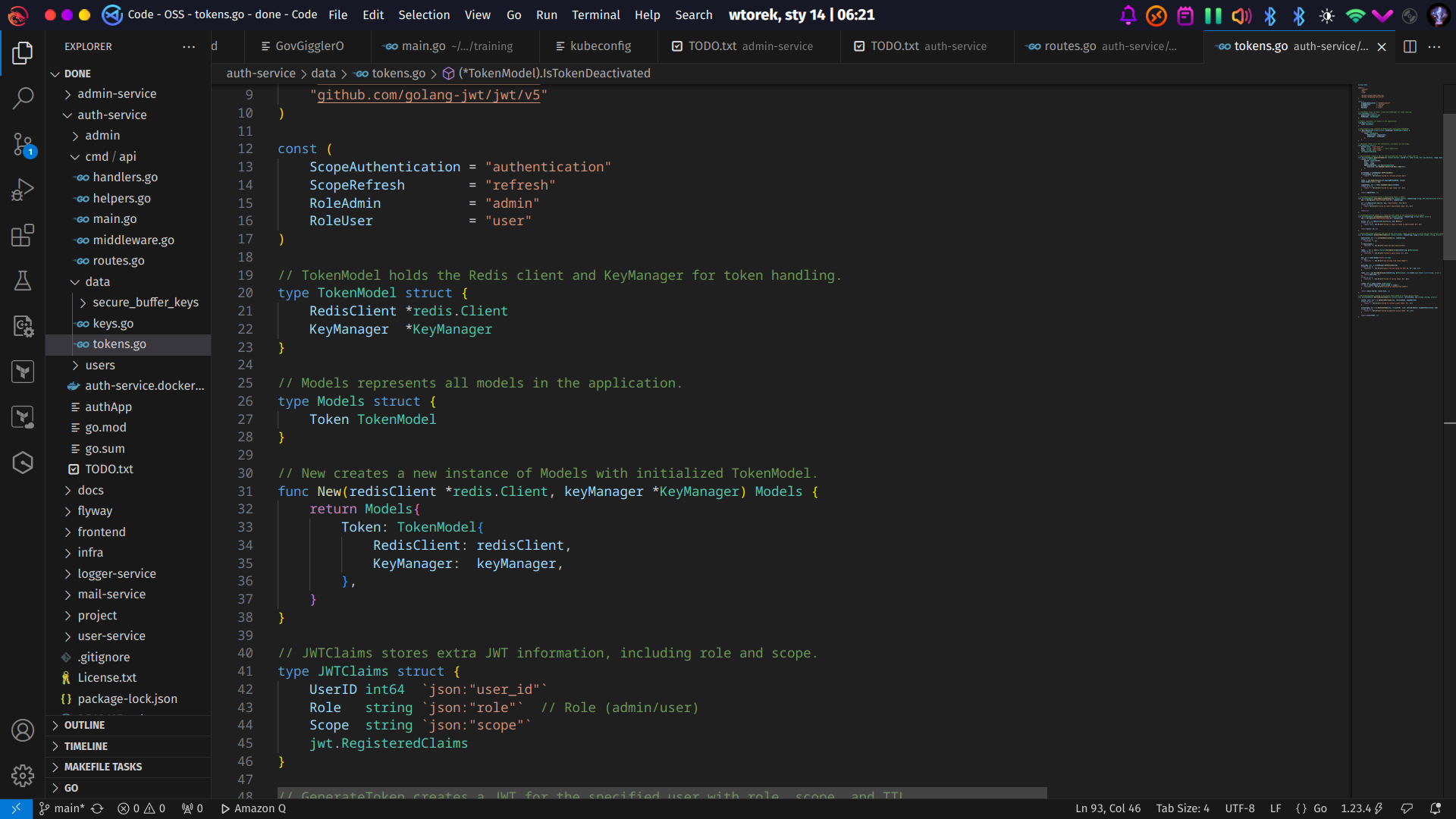
System rozróżnia dwa główne typy użytkowników: zwykłych użytkowników i administratorów, z których każdy ma przypisane odpowiednie uprawnienia. Uwierzytelnianie odbywa się poprzez podanie adresu e-mail i hasła, weryfikowanych w bazie danych. Po pomyślnej weryfikacji użytkownik otrzymuje dwa rodzaje tokenów:

* **Access Token:** Token o krótkim okresie ważności, służący do uwierzytelniania żądań i dostępu do zasobów.
* **Refresh Token:** Token o długim okresie ważności, używany do odświeżania Access Token po jego wygaśnięciu.

Zastosowanie JWT zapewnia decentralizację procesu uwierzytelniania. Każdy mikroserwis może samodzielnie weryfikować tokeny, korzystając z publicznych kluczy, co gwarantuje integralność i autentyczność danych.

**6.2.2 Generowanie tokenów**

Proces generowania tokenów JWT jest kluczowy dla bezpieczeństwa systemu. W projekcie wykorzystano bibliotekę github.com/golang-jwt/jwt/v5 do tworzenia i podpisywania tokenów. Poniższy kod ilustruje proces generowania Access Token i Refresh Token:

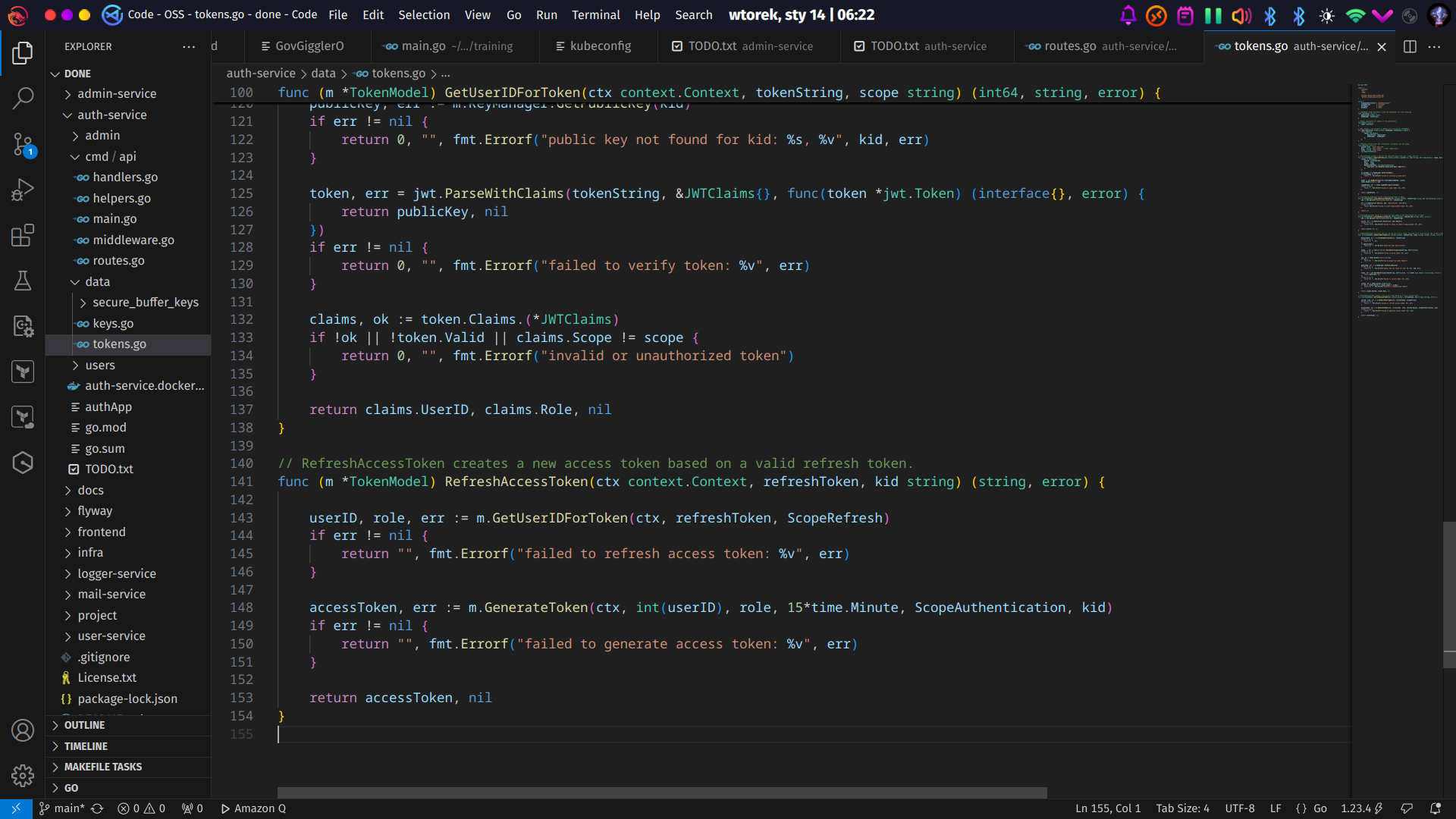


### 

Jak widać, każdy token zawiera informacje o użytkowniku, jego roli i zakresie uprawnień (scope). Dodatkowo, tokeny są podpisywane przy użyciu klucza prywatnego, co uniemożliwia ich modyfikację.

**6.2.3 Odświeżanie tokenów**

Mechanizm odświeżania tokenów pozwala na wydłużenie sesji użytkownika bez konieczności ponownego logowania. Gdy Access Token wygaśnie, klient może użyć Refresh Token do uzyskania nowego Access Token. Poniższy kod przedstawia implementację endpointu odpowiedzialnego za odświeżanie tokenów:

Illustration 2: Implementacja odświeżania tokenów JWT w Go.

Endpoint /token/refresh weryfikuje ważność Refresh Token i generuje nowy Access Token, zapewniając ciągłość dostępu do zasobów.

**6.2.4 Autoryzacja i role użytkowników**

Po uwierzytelnieniu system przechodzi do autoryzacji, przypisując uprawnienia zależnie od roli użytkownika. W projekcie zastosowano mechanizm ról, który definiuje dostęp do zasobów.

Zarówno użytkownicy, jak i administratorzy, mają różne poziomy dostępu. Administratorzy mogą zarządzać kontami, danymi i hasłami, podczas gdy użytkownicy mają ograniczony dostęp do swoich danych i podstawowych funkcji.

### 6.3 Zabezpieczenia komunikacji między mikroserwisami

W architekturze mikroserwisów, w której usługi komunikują się ze sobą przez sieć, jednym z kluczowych elementów zapewniających bezpieczeństwo jest szyfrowana komunikacja między mikroserwisami. W omawianym systemie komunikacja między usługami odbywa się zarówno za pomocą **REST API**, jak i **gRPC**, co umożliwia szybkie i bezpieczne przesyłanie danych.

Aby zapewnić poufność i integralność komunikacji, wszystkie kanały wymiany danych są szyfrowane przy użyciu protokołów TLS/SSL. Usługi takie jak **Authentication Service**, **Logger Service**, **Mailer Service** i inne są odpowiednio zabezpieczone przed nieautoryzowanym dostępem oraz atakami typu „man-in-the-middle”.

Każdy mikroserwis autentykuje i autoryzuje żądania przy użyciu **JWT**, co pozwala na zapewnienie, że tylko uprawnione usługi mają dostęp do wymaganych zasobów. W przypadku, gdy jedno z żądań pochodzi od nieautoryzowanej usługi, odpowiedź serwera zawiera kod błędu 403 lub 401, w zależności od rodzaju naruszenia.

Dodatkowo, system wykorzystuje **Vault** do dynamicznego zarządzania kluczami i certyfikatami SSL, co umożliwia łatwą rotację certyfikatów i minimalizuje ryzyko ich kompromitacji.

**6.4 Monitoring bezpieczeństwa i logowanie incydentów**

Zapewnienie bezpieczeństwa systemu to nie tylko implementacja mechanizmów ochrony danych i szyfrowania, ale również ciągły monitoring i rejestrowanie zdarzeń. Pozwala to na szybką identyfikację i reakcję na potencjalne zagrożenia.

W niniejszym systemie za rejestrowanie zdarzeń odpowiada **Logger-Service**, który zapisuje w bazie danych MongoDB wszelkie operacje na danych i próby nieautoryzowanego dostępu. Struktura logów umożliwia łatwe wyszukiwanie i analizę zdarzeń mogących świadczyć o atakach lub nieprawidłowościach.

W przypadku wykrycia podejrzanych działań logi pozwalają na ustalenie źródła problemu. System generuje również powiadomienia o krytycznych błędach, umożliwiając natychmiastową reakcję administratorów.

Integracja z Vault pozwala na audytowanie dostępu do wrażliwych danych i kontrolowanie, które usługi korzystały z tajemnic w określonych okresach. Logi z tego procesu są zapisywane w dedykowanym rejestrze, co zapewnia pełną kontrolę nad zarządzaniem kluczami i danymi wrażliwymi.

**6.4.1 Logowanie zdarzeń poprzez API**

Logger-Service udostępnia API do rejestrowania zdarzeń z mikroserwisów. Endpoint /log przyjmuje żądania HTTP POST z danymi w formacie JSON, zawierającymi nazwę zdarzenia i dane do zapisania.

Poniższy kod przedstawia implementację funkcji WriteLog w pliku handlers.go, obsługującej endpoint /log:

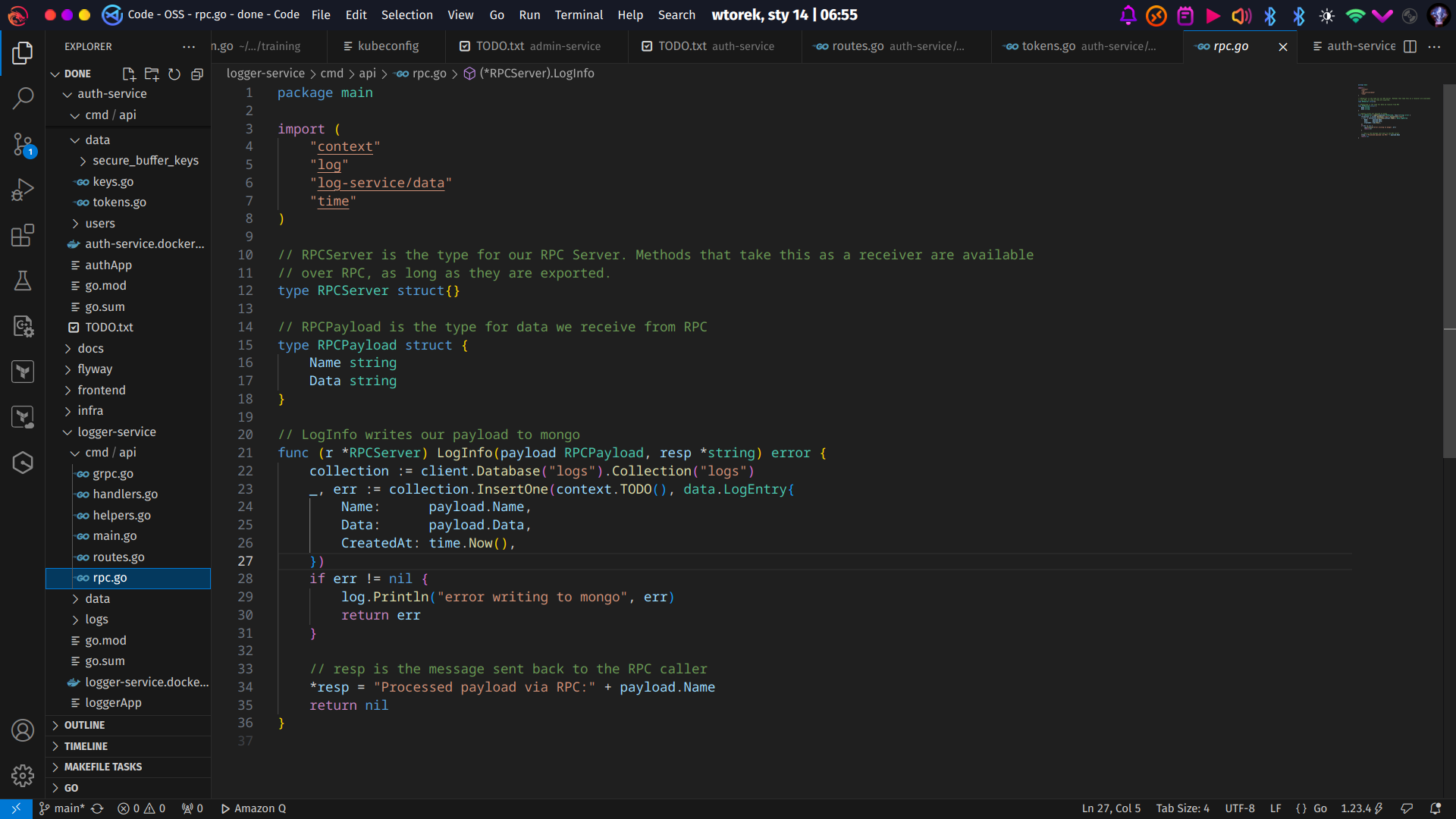


Funkcja WriteLog odczytuje dane z żądania, tworzy wpis w logu i zapisuje go w bazie danych. W przypadku błędu zwracany jest odpowiedni kod odpowiedzi HTTP.

**6.4.2 Logowanie zdarzeń poprzez RPC**

Oprócz API, Logger-Service udostępnia również interfejs RPC (Remote Procedure Call) do rejestrowania zdarzeń. Pozwala to na wydajne i synchroniczne logowanie z mikroserwisów napisanych w różnych językach programowania.

Poniżej przedstawiono implementację funkcji LogRPC w pliku rpc.go, obsługującej żądania RPC:



Funkcja LogRPC przyjmuje jako argument strukturę LogRequest zawierającą dane do zapisania w logu. Następnie tworzony jest wpis w logu i zapisywany w bazie danych. W przypadku sukcesu zwracana jest struktura LogResponse z potwierdzeniem

## 8. Testowanie i weryfikacja systemu

Testowanie oprogramowania jest kluczowym etapem w procesie tworzenia systemów informatycznych, szczególnie w przypadku skomplikowanych rozwiązań opartych na architekturze mikroserwisów. Mikroserwisy wymagają przeprowadzenia różnych rodzajów testów, aby upewnić się, że system działa zgodnie z wymaganiami oraz zapewnia oFunkcja LogRPC przyjmuje jako argument strukturę LogRequest zawierającą dane do zapisania w logu. Następnie tworzony jest wpis w logu i zapisywany w bazie danych. W przypadku sukcesu zwracana jest struktura LogResponse z potwierdzeniemdpowiedni poziom bezpieczeństwa, wydajności i niezawodności. Celem tego rozdziału jest przedstawienie strategii testowania mikroserwisów w projekcie, z naciskiem na procesy uwierzytelniania, weryfikację integracji komponentów oraz analizę wyników testów, co pozwala na ocenę jakości systemu.

### 8.1 Strategie testowania mikroserwisów

Mikroserwisy to małe, niezależne jednostki aplikacji, które są odpowiedzialne za konkretne funkcjonalności w ramach większego systemu. Testowanie takich systemów wymaga zastosowania odpowiednich strategii, które pozwolą na skuteczne sprawdzenie każdego mikroserwisu, a także ich współdziałania w ramach całego systemu. W projekcie, który obejmuje szereg mikroserwisów takich jak **authentication-service**, **logger-service**, **mailer-service** czy **user-service**, zastosowane zostały różne techniki testowania, w tym:

1. **Testowanie jednostkowe (Unit Testing)**  
   Testowanie jednostkowe jest podstawową metodą testowania poszczególnych funkcji i metod w obrębie mikroserwisów. Testy jednostkowe sprawdzają, czy konkretne fragmenty kodu działają poprawnie, izolując je od innych zależności. Dla każdego mikroserwisu napisano testy jednostkowe, które sprawdzają poprawność działania funkcji takich jak generowanie tokenów JWT, walidacja danych użytkownika, proces logowania, a także interakcje z bazami danych (np. MongoDB w przypadku logger-service).

Przykładem testu jednostkowego może być sprawdzenie, czy funkcja generująca token JWT zwraca token o odpowiedniej strukturze, zawierający wymagane dane (np. identyfikator użytkownika) oraz czy token jest poprawnie podpisany i weryfikowalny.

1. **Testowanie integracyjne (Integration Testing)**  
   Testy integracyjne są kluczowe w architekturze mikroserwisów, ponieważ sprawdzają, jak poszczególne mikroserwisy współpracują ze sobą. W tym przypadku, testowanie integracyjne polega na uruchamianiu mikroserwisów w środowisku, które symuluje rzeczywiste warunki produkcyjne, i weryfikowaniu, czy komunikacja między nimi (np. za pomocą HTTP, gRPC lub RPC) działa poprawnie.

W szczególności testowanie integracyjne w projekcie obejmowało sprawdzenie, czy mikroserwis **authentication-service** poprawnie generuje i weryfikuje tokeny, oraz czy **logger-service** prawidłowo zapisuje dane do bazy danych MongoDB. Testowano również, czy wszystkie usługi poprawnie integrują się z bazami danych i zewnętrznymi systemami (np. z systemem pocztowym w **mailer-service**).

1. **Testowanie end-to-end (E2E Testing)**  
   Testowanie end-to-end polega na testowaniu całego procesu biznesowego, który przebiega przez wszystkie mikroserwisy, aby upewnić się, że cała aplikacja działa zgodnie z oczekiwaniami. Testy E2E w tym projekcie obejmowały cały przepływ, począwszy od logowania użytkownika w **authentication-service**, poprzez generowanie i wysyłanie tokenu, aż po zapisanie logów w **logger-service** i ewentualne wysłanie powiadomienia e-mail w **mailer-service**. Testy te pozwalają na weryfikację, czy cała aplikacja jest w stanie obsłużyć typowy scenariusz użytkownika, od logowania po interakcję z innymi mikroserwisami.
2. **Testowanie obciążeniowe (Load Testing)**  
   Mikroserwisy muszą być w stanie obsługiwać dużą liczbę żądań, szczególnie w środowiskach produkcyjnych, gdzie skala użytkowników jest znaczna. W ramach testów obciążeniowych przeprowadzono testy symulujące wysoką liczbę równoczesnych połączeń do różnych mikroserwisów, aby sprawdzić, jak system radzi sobie z dużym ruchem. Testy te pozwoliły na identyfikację potencjalnych wąskich gardeł w systemie i optymalizację niektórych mikroserwisów, takich jak **authentication-service**, który generuje tokeny JWT, oraz **mailer-service**, który wysyła e-maile.
3. **Testowanie bezpieczeństwa (Security Testing)**  
   W związku z tym, że projekt opiera się na wrażliwych danych (np. dane logowania, hasła użytkowników), testowanie bezpieczeństwa jest kluczowe. W ramach testów bezpieczeństwa przeprowadzono testy takie jak ataki typu **SQL Injection** (pomimo używania MongoDB), **Cross-Site Scripting (XSS)**, czy **Cross-Site Request Forgery (CSRF)**. Zostały również przeprowadzone testy weryfikujące poprawność implementacji JWT, sprawdzając, czy tokeny są odpowiednio szyfrowane, podpisane i czy można je skutecznie weryfikować.

### 8.2 Testowanie procesów autentykacji

W kontekście systemu mikroserwisów, procesy uwierzytelniania i autoryzacji są kluczowe dla zapewnienia bezpieczeństwa i poprawności działania aplikacji. Testowanie tych procesów obejmuje różne scenariusze, w których użytkownicy próbują się zalogować, uzyskać dostęp do chronionych zasobów lub wylogować się z aplikacji.

1. **Testowanie procesu logowania**  
   Proces logowania w **authentication-service** jest kluczowym elementem systemu, który pozwala na weryfikację tożsamości użytkownika. Testowanie logowania obejmowało scenariusze poprawne (gdzie użytkownik wprowadza poprawne dane logowania, np. adres e-mail i hasło), jak również testowanie sytuacji niepoprawnych, takich jak błędne dane logowania, brakujące dane, czy zablokowane konto. Każdy z tych scenariuszy był testowany pod kątem odpowiednich odpowiedzi serwera i właściwego działania mechanizmu JWT.
2. **Testowanie procesu rejestracji i resetowania hasła**  
   Kolejnym ważnym procesem w systemie uwierzytelniania jest rejestracja nowego użytkownika oraz procedura resetowania hasła. Testy weryfikowały, czy procesy te są bezpieczne i odporne na ataki. Sprawdzono, czy system odpowiednio waliduje dane wejściowe, czy hasła są przechowywane w sposób bezpieczny (np. w postaci haszowanej), oraz czy użytkownicy otrzymują odpowiednie powiadomienia (np. linki do resetowania hasła).
3. **Testowanie tokenów JWT**  
   Testowanie mechanizmu JWT obejmowało sprawdzenie, czy tokeny są poprawnie generowane, przechowywane i weryfikowane. Testy te uwzględniały m.in. testowanie mechanizmu odświeżania tokenów, aby upewnić się, że system prawidłowo generuje i odświeża tokeny dostępu, oraz czy tokeny wygasają po określonym czasie, uniemożliwiając dostęp do zasobów po ich wygaśnięciu.
4. **Testowanie procesu autoryzacji**  
   W ramach testów autoryzacji weryfikowano, czy tylko odpowiedni użytkownicy mają dostęp do zasobów chronionych w systemie. Testowanie to obejmowało zarówno sprawdzanie, czy różne role użytkowników (np. użytkownik, administrator) mają odpowiednie uprawnienia, jak i testowanie mechanizmu odwoływania dostępu (np. usuwanie tokenów w przypadku wylogowania się lub zmiany uprawnień).

### 8.3 Weryfikacja poprawności integracji komponentów

W systemie opartym na mikroserwisach, weryfikacja poprawności integracji komponentów jest jednym z kluczowych etapów testowania. Integracja poszczególnych mikroserwisów, takich jak **authentication-service**, **logger-service** czy **mailer-service**, wymaga zapewnienia, że wszystkie komponenty współpracują ze sobą poprawnie.

1. **Testowanie komunikacji między mikroserwisami**  
   Testowanie integracji mikroserwisów obejmuje sprawdzenie, czy mikroserwisy mogą wymieniać dane i komunikować się za pomocą protokołów takich jak HTTP, gRPC czy RPC. Testy te weryfikują, czy dane przesyłane między usługami są poprawnie serializowane i deserializowane, a także czy odpowiedzi są zwracane w odpowiednim czasie.
2. **Testowanie baz danych i innych zasobów zewnętrznych**  
   W ramach testów integracyjnych sprawdzano, czy mikroserwisy prawidłowo integrują się z bazami danych (PostgreSQL, MongoDB) oraz z systemami zewnętrznymi (np. SMTP w przypadku **mailer-service**). Testowanie obejmowało weryfikację, czy mikroserwisy mogą poprawnie odczytywać i zapisywać dane w bazach danych, a także czy połączenia z systemami zewnętrznymi są bezpieczne i stabilne.

### 8.4 Wyniki testów i analiza jakości

Po przeprowadzeniu testów, wyniki zostały szczegółowo przeanalizowane, aby ocenić jakość systemu oraz wykryć ewentualne problemy.

1. **Wyniki testów jednostkowych**  
   Wszystkie testy jednostkowe zakończyły się sukcesem, co potwierdza, że poszczególne komponenty mikroserwisów działają poprawnie w izolacji. Testy wykazały również, że funkcje takie jak generowanie tokenów JWT czy walidacja danych użytkownika są realizowane zgodnie z oczekiwaniami.
2. **Wyniki testów integracyjnych i E2E**  
   Testy integracyjne i end-to-end ujawniły pewne problemy z komunikacją między mikroserwisami w przypadku wysokiego obciążenia. W szczególności, **authentication-service** wymagał optymalizacji w zakresie wydajności przy większej liczbie równoczesnych żądań.
3. **Analiza jakości bezpieczeństwa**  
   Testy bezpieczeństwa wykazały, że system jest odporny na ataki takie jak SQL Injection, XSS czy CSRF. Dodatkowo, mechanizm JWT działa zgodnie z wymaganiami, zapewniając odpowiedni poziom bezpieczeństwa.

Wnioski z przeprowadzonych testów umożliwiły optymalizację systemu, poprawę jego wydajności oraz zwiększenie poziomu bezpieczeństwa. Projekt jest gotowy do dalszego rozwoju i przygotowań do wdrożenia w środowisku produkcyjnym.

## 9. Wnioski i perspektywy rozwoju

### 9.1 Podsumowanie wyników pracy

Projekt dotyczący budowy bezpiecznego systemu mikroserwisowego opartego na architekturze z użyciem **Vault**, **Google Cloud KMS**, oraz **Terraform** stanowi istotny krok w kierunku stworzenia zaawansowanego, odpornego na zagrożenia i skalowalnego rozwiązania. Celem tej pracy było zaprojektowanie oraz zaimplementowanie infrastruktury, która zapewnia bezpieczne przechowywanie i dystrybucję kluczy oraz zarządzanie danymi wrażliwymi w kontekście systemów mikroserwisowych. Dzięki zastosowaniu najlepszych praktyk w zakresie zarządzania bezpieczeństwem oraz nowoczesnych narzędzi, takich jak **Vault** oraz **Google Cloud KMS**, system jest w stanie zapewnić wysoki poziom ochrony danych, a jednocześnie umożliwia łatwą integrację i zarządzanie w chmurze.

W ramach realizacji projektu udało się skutecznie wdrożyć mechanizmy uwierzytelniania i autoryzacji za pomocą **JWT**, co stanowi fundament dla kontrolowania dostępu do zasobów w systemie. Wszystkie mikroserwisy zostały zaprojektowane z myślą o wysokiej dostępności i niezawodności, a testy, zarówno jednostkowe, jak i integracyjne, pozwoliły na wychwycenie oraz eliminację potencjalnych problemów z komunikacją i wydajnością.

Zrealizowane rozwiązania umożliwiły stworzenie elastycznego systemu, który może być z łatwością rozwijany i skalowany w zależności od zmieniających się potrzeb. System ten ma potencjał do obsługi różnych scenariuszy biznesowych, w tym aplikacji webowych, systemów bankowych czy rozwiązań e-commerce, które wymagają wyspecjalizowanego zarządzania danymi użytkowników i ich uprawnieniami.

### 9.2 Możliwości rozwoju systemu

Choć projekt spełnia wszystkie podstawowe założenia, to wciąż istnieje szeroki wachlarz możliwości rozwoju i ulepszania systemu. Poniżej przedstawiam kilka potencjalnych kierunków, w których projekt mógłby zostać rozbudowany:

1. **Wdrożenie pełnej obsługi SSL/TLS**  
   Chociaż projekt jest obecnie skonfigurowany z myślą o testowych i deweloperskich środowiskach, jednym z kluczowych elementów, który powinien zostać wprowadzony w przyszłości, jest pełne wsparcie dla komunikacji szyfrowanej przy użyciu **SSL/TLS**. Jest to absolutnie konieczne w kontekście produkcyjnym, aby zapewnić bezpieczeństwo przesyłanych danych, a także spełnić standardy ochrony prywatności (np. RODO).
2. **Integracja z dodatkowymi usługami chmurowymi**  
   Obecnie projekt opiera się na **Google Cloud** jako chmurowym dostawcy, ale istnieje możliwość rozszerzenia systemu o inne platformy, takie jak **AWS** czy **Azure**. Integracja z tymi platformami mogłaby umożliwić szersze wykorzystanie narzędzi chmurowych, takich jak **AWS KMS** czy **Azure Key Vault**, a także dać większą elastyczność w zakresie skalowania infrastruktury.
3. **Rozwój monitoringu i automatyzacji**  
   Zwiększenie zaawansowania w zakresie monitoringu i automatyzacji to kolejny ważny krok. Integracja z narzędziami monitorującymi, takimi jak **Prometheus**, **Grafana** czy **ELK Stack**, mogłaby umożliwić lepsze śledzenie wydajności mikroserwisów oraz szybsze reagowanie na wszelkie problemy związane z dostępnością czy wydajnością. Ponadto, automatyzacja procesów związanych z zarządzaniem infrastrukturą, np. za pomocą **Terraform** w połączeniu z **Ansible** czy **Kubernetes Operators**, mogłaby umożliwić jeszcze łatwiejsze zarządzanie środowiskiem produkcyjnym.
4. **Wsparcie dla więcej baz danych**  
   Obecnie projekt wykorzystuje **PostgreSQL** oraz **MongoDB** jako główne bazy danych, ale w przyszłości warto rozważyć obsługę innych baz danych, takich jak **Cassandra** czy **Elasticsearch**, które mogłyby zwiększyć elastyczność systemu, szczególnie w kontekście aplikacji wymagających przechowywania dużych zbiorów danych w czasie rzeczywistym.
5. **Zwiększenie bezpieczeństwa poprzez Zero Trust Architecture**  
   Przyszłe rozszerzenie systemu mogłoby obejmować implementację **Zero Trust Architecture** (ZTA), która zakłada, że żadne urządzenie ani użytkownik w sieci nie jest automatycznie zaufany. Implementacja ZTA w tym systemie zwiększyłaby bezpieczeństwo, zwłaszcza w przypadku ataków wewnętrznych. Każda komunikacja i każde żądanie musiałoby być dokładnie weryfikowane pod kątem tożsamości użytkownika i autoryzacji.
6. **Integracja z systemami SIEM (Security Information and Event Management)**  
   W przypadku rozszerzenia systemu do produkcji, integracja z systemami **SIEM**, takimi jak **Splunk** czy **ELK Stack**, mogłaby zapewnić dodatkowy poziom monitoringu i wykrywania anomalii w systemie. Analiza logów, wykrywanie nieautoryzowanych prób dostępu oraz analiza incydentów w czasie rzeczywistym pozwoliłyby na szybsze reagowanie na potencjalne zagrożenia.

### 9.3 Wyzwania i rekomendacje na przyszłość

Pomimo licznych sukcesów osiągniętych w trakcie realizacji projektu, napotkano także szereg wyzwań, które będą wymagały uwagi w przyszłości. Oto kilka z nich:

1. **Zarządzanie skalowalnością**  
   Choć system jest skalowalny dzięki architekturze mikroserwisów i kontenerów Docker, to jego wydajność przy bardzo dużej liczbie użytkowników może stwarzać wyzwania, zwłaszcza w kontekście **authentication-service** i **mailer-service**, które muszą obsługiwać dużą liczbę równoczesnych żądań. Wyzwanie to można rozwiązać poprzez wdrożenie strategii skalowania dynamicznego (np. **auto-scaling**) w Kubernetesie oraz optymalizację kodu mikroserwisów.
2. **Bezpieczeństwo przechowywania kluczy**  
   Choć projekt wykorzystuje **Google Cloud KMS** oraz **Vault** do przechowywania kluczy, dalsze poprawki w zakresie zarządzania cyklem życia kluczy (np. rotacja kluczy, audyt i monitorowanie dostępu do kluczy) mogłyby znacznie podnieść bezpieczeństwo systemu. W szczególności warto zadbać o to, by klucze były regularnie rotowane, a dostęp do nich ograniczony do niezbędnych ról i użytkowników.
3. **Kompleksowość konfiguracji w chmurze**  
   Złożoność konfiguracji w chmurowych środowiskach, w tym zarządzanie sekretami, politykami, dostępem oraz integracjami z różnymi usługami, może być wyzwaniem. W tym kontekście warto rozważyć bardziej zautomatyzowane podejście do zarządzania konfiguracjami, np. przez wykorzystanie **GitOps** oraz **Infrastructure as Code (IaC)** w połączeniu z narzędziami do zarządzania konfiguracjami.
4. **Utrzymanie zgodności z przepisami prawa i regulacjami**  
   W przyszłości warto skoncentrować się na dostosowaniu systemu do międzynarodowych regulacji prawnych, takich jak **RODO** (GDPR) w Unii Europejskiej czy **CCPA** w Stanach Zjednoczonych. Będzie to wymagało m.in. wprowadzenia bardziej zaawansowanych mechanizmów anonimizacji danych, audytów dostępu oraz ścisłej kontroli nad danymi wrażliwymi.

### Podsumowanie

Wnioski płynące z realizacji tego projektu wskazują, że architektura mikroserwisowa, połączona z zaawansowanymi rozwiązaniami w zakresie zarządzania bezpieczeństwem, może stanowić fundament nowoczesnych, odpornych na ataki systemów informatycznych. Chociaż projekt już teraz spełnia swoje założenia, wciąż istnieje wiele możliwości rozwoju, które mogą uczynić system bardziej skalowalnym, bezpiecznym i odpornym na przyszłe zagrożenia. Dzięki podejściu opartemu na najlepszych praktykach inżynierii oprogramowania, takich jak **DevOps**, **CI/CD** i **Infrastructure as Code**, system może ewoluować w kierunku jeszcze bardziej zaawansowanych rozwiązań w zakresie zarządzania danymi i ich ochrony.

**Bibliografia**

* Ma, W. (2024). Exploring Microservices Architecture: A Comprehensive Guide. <https://www.devskillbuilder.com/exploring-microservices-architecture-a-comprehensive-guide-fecb5f00be3b>
* Ozkaya, M. (2021). Microservices Architecture. <https://medium.com/design-microservices-architecture-with-patterns/microservices-architecture-2bec9da7d42a>
* Terraform Google Cloud Provider Documentation. <https://registry.terraform.io/providers/hashicorp/google/latest/docs>
* Kubernetes Documentation. <https://kubernetes.io/docs/>
* Google Kubernetes Engine (GKE) Documentation. <https://cloud.google.com/kubernetes-engine/docs>