

**UNIWERSYTET WSB MERITO W GDAŃSKU
WYDZIAŁ INFORMATYKI I NOWYCH TECHNOLOGII**

mgr inż. Grzegorz Konopka
nr albumu 59388

**Projekt i implementacja aplikacji
pobierającej wiadomości z poczty email
i wysyłającej powiadomienia SMS**

Praca inżynierska
na kierunku Informatyka

Praca napisana pod kierunkiem
dr inż. Elżbiety Zamiar

Gdynia 2024

Niniejszą pracę pragnę zadektykować moim Rodzicom **Bogusławie i Ryszardowi**,
którzy zaszczepili we mnie chęć do ciężkiej pracy i samodoskonalenia
oraz **Martynie**,

która znosiła trudny moich działań w procesie zmiany profesji z inżyniera mechanika na
programistę. Bez **Was** ten proces byłby możliwy, ale znacznie rozwlekły w czasie.

Za inspirację, wyrozumiałość oraz pomoc przy realizacji tej pracy pragnę złożyć
serdeczne podziękowania mojemu promotorowi **dr inż. Elżbiecie Zamiar**.

Spis treści

Wstęp.....	7
Rozdział 1. Monolit, mikroserwisy i webserwisy	9
1.1 Podejście w pisaniu aplikacji na przestrzeni lat	9
1.2 Aplikacja w stylu monolitycznym.....	10
1.3 Aplikacja zorientowana na usługi	12
1.4 Opis architektury w modelu REST	14
1.4.1 Metody protokołu HTTP	16
1.4.2 Poziomy dojrzałości architektury REST	17
Rozdział 2. Wykorzystane w projekcie technologie i narzędzia	22
2.1 Język programowania i jego framework	22
2.1.1 Java.....	22
2.1.2 Spring Framework.....	23
2.2 Narzędzia wspomagające wytwarzanie i budowanie aplikacji	26
2.2.1 Zintegrowane środowisko programistyczne – IntelliJ IDEA	26
2.2.2 Apache Kafka i komunikacja asynchroniczna serwisów	27
2.2.3 Maven – automatyzacja budowania aplikacji	29
2.3 Narzędzia wspomagające wdrożenie i monitorowanie stanu aplikacji.....	32
2.3.1 Docker i Docker Compose jako narzędzie usprawniające wdrożenie	32
2.3.2 Monitorowanie stanu aplikacji	34
Rozdział 3. Projekt i realizacja rozwiązania.....	36
3.1 Identyfikacja i analiza wymagań.....	36
3.1.1 Założenia projektu.....	36
3.1.2 Analiza systemowa – podział na serwisy	37
3.1.3 Wspólne API dla serwisów	38
3.1.4 Komunikacja z Gmail API	41
3.1.5 Komunikacja z Twilio API	53
3.2 Mikroserwisy.....	56
3.2.1 Serwis do odczytywania wiadomości email.....	63
3.2.2 Serwis do filtrowania wiadomości email	64

3.2.3	Serwis do tworzenia wiadomości SMS	64
3.2.4	Serwis do wysyłania notyfikacji	65
3.2.5	Klient do podglądu aplikacji	66
3.3	Testowanie	69
3.4	Wdrażanie i monitorowanie aplikacji	71
3.4.1	Konfiguracja uruchamiania aplikacji w środowisku konteneryzowanym	72
3.4.2	Implementacja narzędzi do monitorowania stanu aplikacji	76
	Zakończenie	80
	Spis bibliografii	83
	Spis rysunków i tabel	85
	Załączniki	88
	A – Dokumentacja email-rest-client	88
	B – Dokumentacja email-filtering-service	93
	C – Dokumentacja message-service.....	96
	D – Dokumentacja notification-service.....	105

Wstęp

Obecnie wskutek dużego szumu informacyjnego na pocztach użytkowników, który wynika m.in. z posiadania wielu kont w różnych serwisach, reklam, notyfikacji z serwisów społecznościowych, phisngu, spamu często zdarza się, że ważna wiadomość może umknąć naszej uwadze – na przykład opłacenie faktury czy przypomnienie o terminie płatności ubezpieczenia. Przedstawiony problem mogłaby rozwiązać aplikacja do wysyłania powiadomień SMS, która na podstawie stworzonych filtrów oraz odczytanych wiadomości email tworzyłaby wiadomości, które następnie byłyby wysyłane na określony numer telefonu w postaci przypomnienia SMS. Taka redundancja przypominania o ważnym wydarzeniu przyczyniłaby się do zmniejszenia prawdopodobieństwa niewykonania jakieś istotnej czynności.

Przedmiotem pracy dyplomowej jest projekt i implementacja aplikacji do wysyłania powiadomień SMS przez serwis Twilio (platforma pozwalająca na wykonywanie usług telefonicznych) na podstawie wyfiltrowanych emaili otrzymanych przy pomocy Gmail API (ang. Application Programming Interface). Implementację aplikacji wykonano w architekturze zorientowanej na usługi (mikroserwisy), co w przypadku późniejszego rozwoju aplikacji pozwoli na łatwiejsze dodanie nowych funkcjonalności takich jak np. możliwość obsługi wielu użytkowników, autentykacja i autoryzacja przy pomocy konta Gmail. Dodatkowo zostanie zaimplementowany webowy klient, który pozwoli na ręczne dodawanie powiadomień i podgląd już stworzonych wiadomości, czy odebranych emaili spełniających kryteria. Podczas realizacji projektu wykorzystano nowoczesne podejście w kwestii architektury aplikacji – zastosowano podejście DevOps (ang. Development & Operations). Każdy z mikroserwisów opakowano w kontener dockerowy, co pozwoli na łatwe wdrożenie aplikacji na dowolnej maszynie posiadającej Docker Engine.

Praca dyplomowa składa się w trzech rozdziałach, które zostały podzielone według następującej struktury: rozdział pierwszy – część teoretyczna, rozdział drugi – część narzędziowa, rozdział trzeci – część projektowa i implementacyjna.

W rozdziale pierwszym zostały poruszone takie zagadnienia jak historia rozwoju aplikacji webowych na przestrzeni ostatnich lat. Skupiono się w nim na omówieniu zalet i wad w podejściu do tworzenia aplikacji w architekturze monolitu oraz tworzeniu aplikacji zorientowanych na usługi. Oprócz tego poruszono ważny temat jak styl architektoniczny REST (ang. REpresentational State Transfer), który bardzo dobrze współpracuje z aplikacjami

napisanych w architekturze mikroserwisów. Poruszono również także aspekt ewolucji tego stylu - model dojrzałości Richardsona.

Rozdział drugi natomiast skupia się na narzędziach niezbędnych i wspomagających wytwarzanie oprogramowania. Został on podzielony na trzy podrozdziały:

- w pierwszym skupiono się na opisie wybranego języka programowania i opisie frameworka przyśpieszającego budowanie aplikacji;
- w drugim przedstawiono narzędzia wspomagające wytwarzanie oprogramowania, począwszy od zintegrowanego środowiska programistycznego przez narzędzia wspomagające budowanie aplikacji a kończąc na narzędziu, który umożliwia komunikację między serwisami;
- w ostatnim opisano narzędzia wspomagające proces wdrażania aplikacji na środowisku produkcyjnym przy użyciu takiego narzędzia jak Docker wraz z Docker Compose. Oprócz tego poruszono problem monitorowania stanu aplikacji – agregacja i wyświetlanie logów przy użyciu stosu ELK (Elasticsearch, Logstash, Kibana), a także udostępnianie i wyświetlanie metryk przy pomocy Grafany oraz Prometheusa.

Rozdział trzeci stanowi opis realizacji części praktycznej pracy dyplomowej. Skupiono się w nim na analizie biznesowej rozważanego problemu oraz na implementacji mikroserwisów. Został on podzielony na trzy podrozdziały. Pierwszy z nich porusza aspekty analizy systemowej – analiza wymagań, podział na mikroserwisy ze względu na odpowiedzialność, tworzenie wspólnego API dla mikrosług. W drugim podrozdziale został omówiony każdy z mikroserwisów usługowych i narzędziowych. W trzecim natomiast został poruszony proces wdrażania aplikacji na środowisku skonteneryzowanym przy użyciu Dockera wraz z narzędziami pozwalającymi na monitorowanie stanu aplikacji.

Do wykonania części projektowej konieczne było zapoznanie się z pozycjami literaturowymi obejmującymi takie zagadnienia jak: podstawy programowania w Javie, sztuka pisania dobrego kodu, wzorce projektowe, testowanie, a także frameworki przyśpieszające proces wytwarzania oprogramowania [Horstmann 2016, Horstmann 2017, Beck 2020, Bloch 2018, Freeman & Robson 2022, Walls 2015, Martin 2014, Martin 2017].

Rozdział 1. Monolit, mikroserwisy i webserwisy

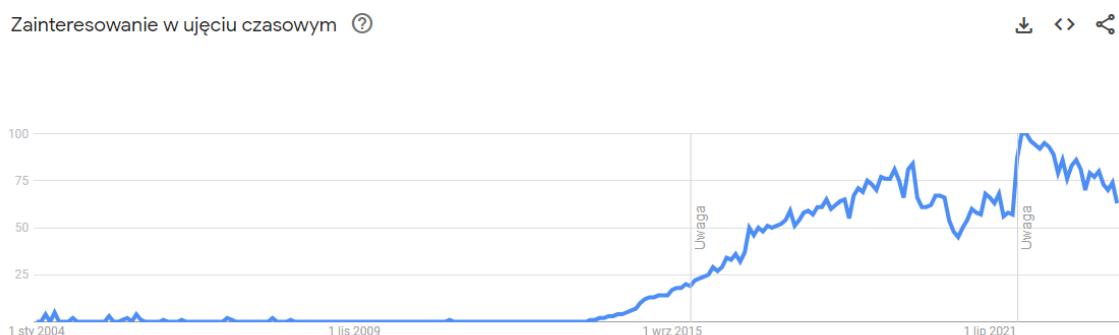
Niniejszy teoretyczny rozdział skupia się na opisaniu architektury aplikacji – jak zmieniała się ona na przestrzeni lat z uwzględnieniem zalet oraz wad każdego z podejść. Dodatkowo w kontekście aplikacji webowych została opisana architektura w modelu REST (ang. REpresentational State Transfer) wraz z omówieniem podstawowych metod HTTP (ang. HyperText Transfer Protocol) oraz model dojrzałości Richardsona.

1.1 Podejście w pisaniu aplikacji na przestrzeni lat

Na potrzeby tego rozdziału zostanie przedstawiona skrócona i uproszczona definicja dwóch rodzajów budowy aplikacji jakie można najczęściej można spotkać w środowisku webowym:

- monolit – pojedyncza aplikacja zawierająca jedną bazę kodu, która definiuje logikę aplikacji, interfejs użytkownika i warstwę dostępu do danych. Utrzymywana jest jako monolityczna jednostka i obsługiwana z centralnej lokalizacji [Martin Fowler, 2023];
- mikroserwisy – aplikacja będąca sumą małych aplikacji współpracujących ze sobą poprzez komunikację np. za pomocą protokołu HTTP (ang. HyperText Transfer Protocol) czy JMS (ang. Java Message Service) [Martin Fowler, 2023].

Rysunek 1. Zainteresowanie frazą "Microservices" na przestrzeni lat



Źródło: opracowanie własne

Wykres powyżej przedstawia zainteresowanie frazą „Microservices” na przestrzeni ostatnich 19 lat. Na jego podstawie można wysnuć błędny wniosek, że architektura aplikacji zorientowanej na usługi (mikroserwisy) została wymyślona około 2013 roku, a zainteresowanie nią zaczęło rosnąć na początku 2014 roku. Nie jest to prawda – architektura mikroserwisów

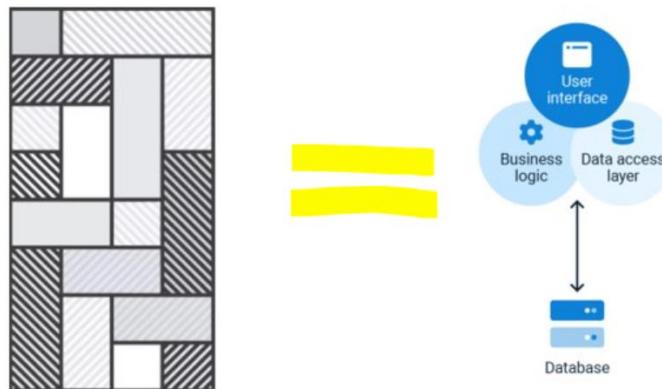
swój początek ma tak naprawdę w książce „*Domain Driven Design*” Erica Evansa w której to zostało poruszone takie zagadnienie jak „*Bounded Context*”. Fraza ta oznacza, że podsystem aplikacji powinien być ograniczony wspólnym kontekstem. Nie powinien on współdzielić danych między innymi kontekstami jak ma to miejsce w przypadku klasycznej aplikacji monolitycznej. Termin Mikroserwisy został po raz pierwszy użyty przez Petera Rodgersa w 2005 roku podczas konferencji Web Services Edge. Można więc uznać Rodgersa za twórcę tego terminu [Evans, 2003].

Trudno jest jednoznacznie zdefiniować przyczynę tak dużego wzrostu zainteresowania architekturą zorientowaną na usługi. Można jedynie doszukiwać się przyczyn na podstawie rozwoju innych technologii i wzrostu złożoności aplikacji. Niewątpliwie ważnym elementem wzrostu zainteresowania tym terminem należy szukać w popularyzacji internetu i aplikacji webowych. Za przyczyną takiego zjawiska stoi niewątpliwie dynamiczny rozwój usług chmurowych takich jak Azure, Google Cloud czy Amazon Web Services. Rozwój ten pozwolił na to, aby aplikacje nie były utrzymywane sprzętowo przez właściciela oprogramowania, ale by były umieszczone u dostawcy usług chmurowych. Na wykresie widać pewien spadek zainteresowania w latach 2020 – 2021. Wynika on z tego, że twórcy oprogramowania zdali sobie sprawę, że nie każda aplikacja nadaje się do takiej architektury. W przypadku małych systemów narzut pracy potrzebny na stworzenie aplikacji działającej w takim podejściu jest zbyt duży – zachodzi tutaj bowiem problem sporych trudności w konfiguracji wspólnej infrastruktury [Gos & Zabierowski, 2020].

1.2 Aplikacja w stylu monolitycznym

Hasłem monolit w inżynierii oprogramowania określa się aplikację jednowarstwową, w której to interfejs użytkownika, warstwa biznesowa, warstwa dostępu do danych jak i sama baza danych znajduje się w jednym ekosystemie. Można ją przedstawić jako jedno repozytorium kodu, a z biznesowego punktu widzenia jest to aplikacja, która wykonuje wymagane operacje i wszystkie kroki pośrednie niezbędne do osiągnięcia celu w ramach jednego systemu. Aplikacja jest samodzielna i niezależna od innych aplikacji komputerowych. Nie oznacza to natomiast, że monolit nie może zostać zaprojektowany w formie różnych modułów, które ze sobą współpracują [Gos & Zabierowski, 2020].

Rysunek 2. Ideowe przedstawienie monolitu



Źródło: opracowanie własne

Aby w pełni opisać aplikację wykonaną jako monolit warto skupić się na wypisaniu i omówieniu najważniejszych zalet oraz wad.

Zalety:

- odpowiednia na start firmy – nie są wymagane duże kompetencje organizacyjne;
- dobra dla małych aplikacji – w przypadku małych aplikacji monolit to odpowiednie podejście bowiem nie występują tutaj problemy architektoniczne jak w przypadku mikroserwisów;
- łatwość wdrożenia – wdrożenie aplikacji na produkcję nie jest skomplikowane, bowiem występuje tutaj tylko jeden artefakt;
- proste monitorowanie stanu aplikacji – przez to, że aplikacja ma strukturę monolitu, logi z aplikacji występują tylko w jednym miejscu, a także cały ruch sieciowy kierowany jest do jednej instancji aplikacji.

Wady:

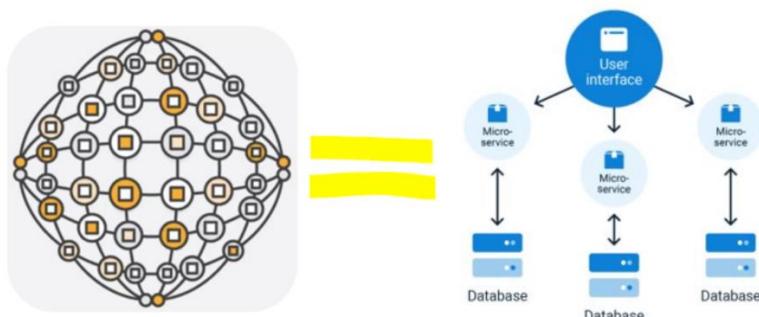
- trudna w skalowaniu – w przypadku, gdy okaże się, że jakaś część systemu ma zbyt duży ruch, konieczne jest skalowanie całej aplikacji co generuje niepotrzebny koszt;
- długi czas budowania – monolity klasy enterprise potrafią budować się wraz z testami integracyjnymi ponad dobę;
- skomplikowana edycja – kod źródłowy aplikacji jest bardzo długi i skomplikowany przez co złożoność cyklometryczną ciągle rośnie;

- trudno rozwijać architekturę – komunikacja z zewnętrznymi systemami jest utrudniona;
- nowe wydania trwają bardzo długo – przez złożoność kodu, mnogość zespołów developerskich i brak ściśle określonej odpowiedzialności oraz skomplikowane testowanie czas ten może wynosić nawet kilka miesięcy;
- trudniej zastosować podejście zwinne – trudno zastosować tutaj podejście MVP (ang. Minimum Valuable Product);
- trudno o innowacyjność – w przypadku pojawienia się nowej technologii nie można migrować tylko części funkcjonalności biznesowej;
- utrudnione dodawanie nowych funkcjonalności – każda nowa funkcjonalność komplikuje i tak już skomplikowany kod.

1.3 Aplikacja zorientowana na usługi

Hasłem mikroserwisy określa się styl tworzenia architektury, który implementuje wzorzec aplikacji zorientowanej na usługi poprzez zbudowanie całego systemu na podstawie małych, luźno połączonych ze sobą serwisów komunikujących się ze sobą przy pomocy protokołów komunikacyjnych takich jak np. HTTP (ang. HyperText Transfer Protocol) czy JMS (ang. Java Message Service). Zgodnie z pierwszym z pryncypów akronimu SOLID (ang. SRP, Open/Closed, Liskov Substitution Principle, Interface Segregation, Dependency Inversion) [Evans, 2003], S reprezentuje SRP (ang. Single Responsibility Principle) – co oznacza, że mikroserwis powinien być odpowiedzialny tylko za jedną funkcję z biznesowego punktu widzenia np. za logowanie użytkownika i powinien być całkowicie niezależny od pozostałych składowych systemu [Gos & Zabierowski, 2020].

Rysunek 3. Ideowe przedstawienie mikroserwisów jako sieć małych aplikacji



Źródło: opracowanie własne

Aby w pełni opisać aplikację wykonanej w architekturze zorientowanej na usługi warto skupić się na wypisaniu i omówieniu najważniejszych zalet oraz wad.

Zalety:

- łatwe w skalowaniu – w przypadku, gdy okaże się, że jeden z mikroserwisów posiada większe niż zakładano obciążenie w łatwy sposób może zostać zeskalowany do np. dwóch instancji;
- krótki czas budowania – kod źródłowy jest niewielki co pozwala na krótkie budowanie, rzędu kilkunastu, kilkudziesięciu minut;
- prosta i klarowna odpowiedzialność – każdy mikroserwis posiada przypisany do siebie zespół developerski, który za niego odpowiada;
- łatwo rozwijać architekturę – łatwo dodać nową funkcjonalność np. autoryzację dwuskładnikową poprzez utworzenie nowego mikroserwisu, który za to odpowiada;
- krótki czas wdrożenia – mikroserwisy często są wdrażane na produkcje raz w tygodniu;
- łatwość zastosowania metodyk zwinnych – dzięki temu, że mikroserwis ma przypisany zespół, łatwiej zastosować podejście Agile;
- duża innowacyjność – w przypadku pojawienia się nowej technologii, można w łatwy sposób zrobić re-implementację istniejącego serwisu;
- łatwe dodawanie nowych funkcjonalności – repozytorium kodu jest niewielkie, zatem łatwo dodać nową funkcjonalność i ją przetestować.

Wady:

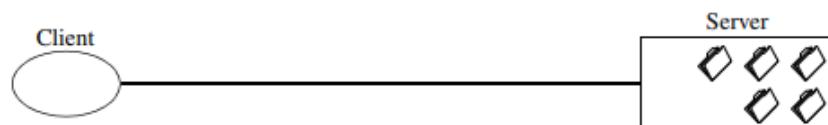
- potrzebna duża ilość specjalistów – ekosystem mikroserwisów wymaga nie tylko developerów oprogramowania, ale również specjalistów od architektury i wdrożeń;
- trudne testowanie – trudno wykonać testowanie E2E (ang. End To End);
- trudność w utrzymaniu architektury – skomplikowana architektura sprawia, że trudno jest nad nią zapanować, bowiem mamy do czynienia z całym ekosystemem aplikacji;
- utrudnione monitorowanie aplikacji – ruch sieciowy odbywa się w wielu miejscach jednocześnie;
- drogie we wdrożeniu – pierwsze wydanie aplikacji wymaga dużego nakładu pracy pod kątem architektonicznym.

1.4 Opis architektury w modelu REST

Temat architektury w modelu REST (ang. Representational State Transfer) to styl tworzenia oprogramowania, który został zaprezentowany po raz pierwszy w rozprawie doktorskiej Roy'a Fieldinga w 2000 roku pod tytułem „Architectural Styles and the Design of Network-based Software Architectures” – co można przetłumaczyć na „Style Architektoniczne i projektowanie aplikacji opartych o sieć internetową”. Należy zaznaczyć, że REST jest stylem, a nie wymaganym standardem. W swojej pracy Roy Fielding nie narzucił konkretnego rozwiązania, ale skupił się na opisie paradigmatów jakimi powinny wyróżniać się aplikacje tak tworzone. Według jego rozwiazań aplikacja napisana w tym stylu powinna wyróżniać się następującymi cechami:

- aplikacja napisana w stylu klient – serwer – warstwa ta powinna rozdzielać problemy związane z interfejsem użytkownika od problemów związanych z przechowywaniem danych. Klient wysyła zapytania, a serwer na nie odpowiada. Zapytanie od klienta musi zawierać komplet informacji koniecznych do wykonania polecenia. Podobnie w przypadku serwera – daje on odpowiedź tylko na wydane zapytanie;

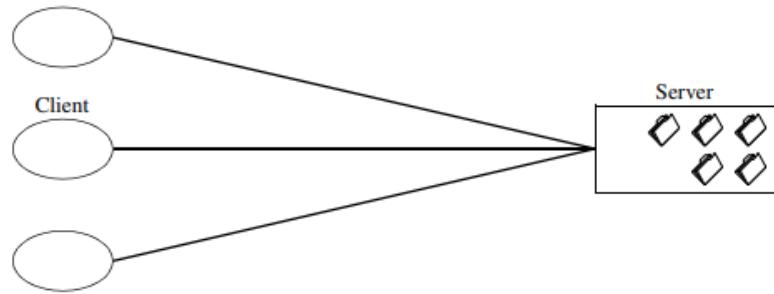
Rysunek 4. Architektura klient - serwer



Źródło: Roy Thomas Fielding, Architectural Styles and the Design of Network-based Software Architectures, University of California 2020, s. 78

- bezstanowość – każde zapytanie od klienta powinno zawierać komplet informacji niezbędnych do wykonania zapytania bowiem serwer nie przechowuje stanu o sesji użytkownika po swojej stronie. W architekturze napisanej w stylu REST nie istnieje takie pojęcie jak stan użytkownika czy sesje. Jeżeli z punktu widzenia bezpieczeństwa aplikacji konieczne jest uwierzytelnione zapytanie, wówczas może ono zostać zweryfikowane przy pomocy dodatkowych informacji pozwalających na zweryfikowanie zapytania np. poprzez token;

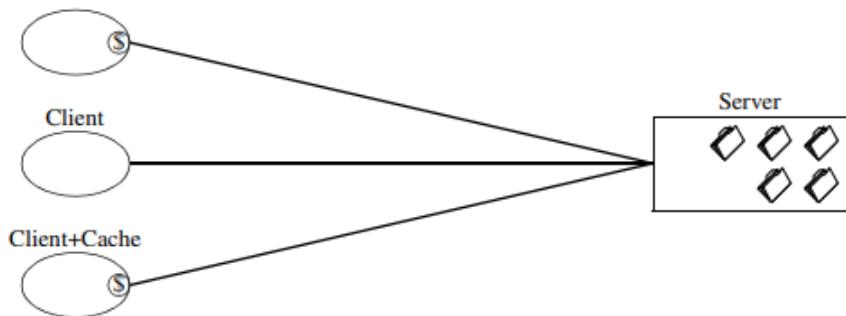
Rysunek 5. Architektura klient-bezstanowy serwer



Źródło: Roy Thomas Fielding, Architectural Styles and the Design of Network-based Software Architectures, University of California 2020, s. 78

- wykorzystanie pamięci podręcznej (ang. cache) – mechanizm ten pozwala na szybsze zwrócenie odpowiedzi od serwera w przypadku, gdy dane zapytanie się powtarza;

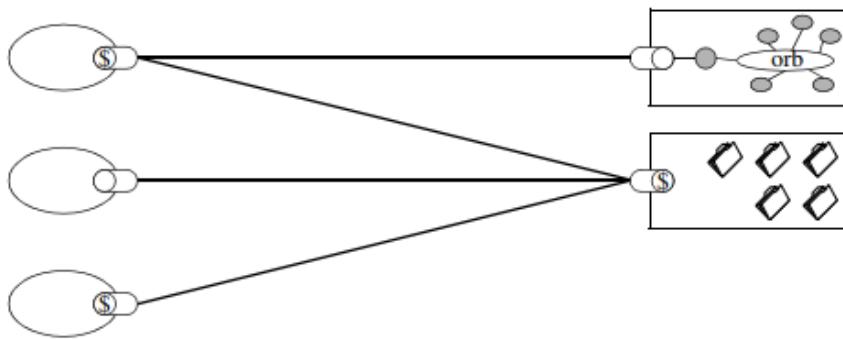
Rysunek 6. Klient-Cache-Stateless-Server



Źródło: Roy Thomas Fielding, Architectural Styles and the Design of Network-based Software Architectures, University of California 2020, s. 80

- jednorodny interfejs – wszystkie zapytania powinny spełniać reguły formatowania określone przez API (ang. Application Programming Interface). Specyfikacja endpointów musi pozwalać na uzyskanie takiej samej odpowiedzi niezależnie od tego jaki klient dane zapytanie wysłał;

Rysunek 7. Uniform-Klient-Cache-Stateless-Server



Źródło: Roy Thomas Fielding, Architectural Styles and the Design of Network-based Software Architectures, University of California 2020, s. 82

- separacja warstw – REST zakłada, że warstwy aplikacji powinny być od siebie odseparowane. Oznacza to, że warstwa dostępu do danych powinna być odseparowana od logiki biznesowej i od warstwy prezentacji. Żadna z warstw nie powinna bezpośrednio oddziaływać na inne warstwy np. warstwa kontrolera nie powinna mieć dostępu do warstwy dostępu do danych. Za manipulację danymi odpowiada warstwa biznesowa, która wykonując odpowiednie zadania ma wpływ na to co zostanie dodane/usunięte w warstwie repozytorium;
- kod na żądanie (opcjonalny paradygmat) – zakłada możliwość udostępniania skryptów wykonywalnych użytkownikom [Fielding, 2000].

1.4.1 Metody protokołu HTTP

W kontekście architektury REST należy wspomnieć o protokole HTTP (ang. HyperText Transfer Protocol). Mimo, że ten styl architektoniczny nie wymusza stosowania tego protokołu – można używać na przykład TCP (ang. Transmission Control Protocol), jednakże przez swoją prostotę i możliwość spełnienia podstawowych założeń tego stylu architektonicznego przy tworzeniu aplikacji webowych stał się standardem.

Jednym z elementów składowych protokołu HTTP są metody – pozwalają one na zdefiniowanie rodzaju akcji jaka ma zostać wykonana na serwerze. Wyróżnia się metody:

- GET – pobiera informację o zasobach z określonego adresu,
- HEAD – podobnie jak GET pobiera informacje o zasobach z tą różnicą jednak, że zwraca tylko nagłówki,

- POST – przesyła żądania i wykonuje akcję w stosunku do określonego zasobu,
- PUT – aktualizacja danych dotyczących zasobu do serwera,
- DELETE – skasowanie zasobu z serwera,
- OPTIONS – prośba o przesłanie informacji na temat dostępnych metod komunikacji dla danego zasobu,
- TRACE – zdalne śledzenie wszelkich innych metod http,
- CONNECT – stworzenie tunelu poprzez serwer proxy,
- PATCH – podobnie jak PUT odpowiada za aktualizację zasobu dostępnego na serwerze z tą różnicą, że aktualizuje tylko część zasobu.

Metody HTTP można podzielić na idempotentne i nieimotentne. Idempotentność oznacza właściwość operacji, która pozwala na ich wielokrotne wywołanie bez zmiany wyniku [Fowler, 2023].

Rysunek 8. Charakterystyka metod HTTP

The chart is titled "IDEMPOTENCE" in large white letters at the top. Below it, a subtitle reads "WHEN PERFORMING AN OPERATION AGAIN GIVES THE SAME RESULT". The chart consists of a table with three columns: "HTTP METHOD", "IDEMPOTENCE", and "SAFETY". The rows represent different HTTP methods: GET, HEAD, PUT, DELETE, POST, and PATCH. The "IDEMPOTENCE" column contains "YES" for GET, HEAD, PUT, and DELETE, and "NO" for POST and PATCH. The "SAFETY" column contains "YES" for GET, HEAD, and PUT, and "NO" for DELETE, POST, and PATCH. The entire chart is set against a dark green background.

HTTP METHOD	IDEMPOTENCE	SAFETY
GET	YES	YES
HEAD	YES	YES
PUT	YES	NO
DELETE	YES	NO
POST	NO	NO
PATCH	NO	NO

NORDICAPIS.COM

Źródło: <https://nordicapis.com/understanding-idempotency-and-safety-in-api-design/>
[dostęp 27.11.2023]

1.4.2 Poziomy dojrzałości architektury REST

W kontekście architektury napisanej w stylu REST należy wspomnieć również o opisie dojrzałości architektury scharakteryzowanej przez Leonarda Richardsona podczas konferencji QCon. Według niego architekturę REST możemy podzielić na cztery poziomy, począwszy

od zerowego, a kończąc na trzecim. Jeżeli aplikacja będzie charakteryzować się obecnością cech występujących w ostatnim z nich, można wtedy ją nazwać RESTful (ang. Representational State Transfer fully).

Poziom 0 – aplikacja nie spełnia założeń zgodnych z architekturą REST. API (ang. Application Programming Interface) nie wykorzystuje potencjału protokołu HTTP (ang. HyperText Transfer Protocol). Implementacja nie opiera się na unikalnych adresach do zasobów URI (ang. Uniform Resource Identifier). Wykorzystuje jedną metodę HTTP, a operacje do wykonania zawierają się w ciele żądania/zapytania. Zarówno proces dodawania zasobu, wyświetlania, usuwania zasobu realizowany jest przez jeden URI (ang. Uniform Resource Identifier) i zawsze zwraca ten sam status odpowiedzi HTTP, niezależnie od rezultatu.

Rysunek 9. Przykład żądania HTTP w przypadku poziomu 0

```
POST /appointmentService HTTP/1.1
[various other headers]

<openSlotRequest date = "2010-01-04" doctor = "mjones"/>
```

Źródło: <https://martinfowler.com/articles/richardsonMaturityModel.html> [dostęp 27.11.2023]

Rysunek 10. Przykład odpowiedzi HTTP w przypadku poziomu 0

```
HTTP/1.1 200 OK
[various headers]

<openSlotList>
  <slot start = "1400" end = "1450">
    <doctor id = "mjones"/>
  </slot>
  <slot start = "1600" end = "1650">
    <doctor id = "mjones"/>
  </slot>
</openSlotList>
```

Źródło: <https://martinfowler.com/articles/richardsonMaturityModel.html> [dostęp 27.11.2023]

Poziom 1 – usługi wykorzystują wiele identyfikatorów URI (ang. Uniform Resource Identifier). Zazwyczaj są to metody HTTP GET i POST. Pierwszy z nich dotyczy otrzymywania informacji o zasobie, drugi natomiast dotyczy dodawania zasobu. Poziom ten zakłada, że każdy zasób na serwerze ma swój indywidualny identyfikator. Często operacje

wykonywane na zasobie są oznaczane poprzez dodanie dodatkowego słowa kluczowego (czasownika) do adresu np: `/book/create`, `/book/3/delete`.

Rysunek 11. Przykład zapytania HTTP o zasób w przypadku poziomu 1

```
POST /slots/1234 HTTP/1.1
[various other headers]

<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

Źródło: <https://martinfowler.com/articles/richardsonMaturityModel.html> [dostęp 27.11.2023]

Rysunek 12. Przykład odpowiedzi HTTP o zasób w przypadku poziomu 1

```
HTTP/1.1 200 OK
[various headers]

<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointment>
```

Źródło: <https://martinfowler.com/articles/richardsonMaturityModel.html> [dostęp 27.11.2023]

Poziom 2 – w powyższym poziomie podobnie jak w przypadku pierwszego, wykorzystywanych jest wiele identyfikatorów URI (ang. Uniform Resource Identifier). Wykorzystywany jest natomiast pełny potencjał protokołu HTTP (ang. HyperText Transfer Protocol). Akcje wykonywane na zasobie/zasobach nie są definiowane poprzez dodawanie rodzaju akcji do adresu URI, lecz poprzez odpowiednią metodę HTTP. Dodatkowo takie API (ang. Application Programming Interface) nie ogranicza się tylko do zwracania statusu 200 HTTP, ale wykorzystywane są również inne, które jawnie opisują typ odpowiedzi od serwera.

Rysunek 13. Przykład zapytania HTTP o zasób w przypadku poziomu 2

```
POST /slots/1234 HTTP/1.1
[various other headers]

<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

Źródło: <https://martinfowler.com/articles/richardsonMaturityModel.html> [dostęp 27.11.2023]

Rysunek 14. Przykład odpowiedzi HTTP o utworzenie zasobu w przypadku poziomu 2

```
HTTP/1.1 201 Created
Location: slots/1234/appointment
[various headers]

<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointment>
```

Źródło: <https://martinfowler.com/articles/richardsonMaturityModel.html> [dostęp 27.11.2023]

Poziom 3 – najwyższy poziom API (ang. Application Programming Interface) w rozumieniu Richardsoна. Aktualnie nie jest zbyt często realizowany ze względu na dodatkową złożoność. Poziom ten określany jest samodokumentującym, bowiem z poziomu zapytania, można otrzymać listę dodatkowych URI (ang. Uniform Resource Identifier), które odnoszą się do danego zasobu i wszystkich pochodnych. Mowa tutaj o spełnieniu wymagania HATEOAS (ang. Hypertext As The Engine Of Application State).

Rysunek 15. Przykład zapytania HTTP o zasób w przypadku poziomu 3

```
POST /slots/1234 HTTP/1.1
[various other headers]

<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

Źródło: <https://martinfowler.com/articles/richardsonMaturityModel.html> [dostęp 27.11.2023]

Rysunek 16. Przykład odpowiedzi HTTP o utworzenie zasobu w przypadku poziomu 3

```
HTTP/1.1 201 Created
Location: http://royalhope.nhs.uk/slots/1234/appointment
[various headers]

<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
  <link rel = "/linkrels/appointment/cancel"
        uri = "/slots/1234/appointment"/>
  <link rel = "/linkrels/appointment/addTest"
        uri = "/slots/1234/appointment/tests"/>
  <link rel = "self"
        uri = "/slots/1234/appointment"/>
  <link rel = "/linkrels/appointment/changeTime"
        uri = "/doctors/mjones/slots?date=20100104&status=open"/>
  <link rel = "/linkrels/appointment/updateContactInfo"
        uri = "/patients/jsmith/contactInfo"/>
  <link rel = "/linkrels/help"
        uri = "/help/appointment"/>
</appointment>
```

Źródło: <https://martinfowler.com/articles/richardsonMaturityModel.html> [dostęp 27.11.2023]

Rozdział 2. Wykorzystane w projekcie technologie i narzędzia

Rozwój technologii informatycznych przekształcił sposób, w jaki tworzone są aplikacje i w jaki sposób się z nich korzysta. W dzisiejszym dynamicznym środowisku cyfrowym, szybkość wytwarzania oprogramowania stała się kluczowym czynnikiem sukcesu. Aby sprostać rosnącym oczekiwaniom użytkowników oraz skrócić czas wprowadzania innowacyjnych rozwiązań na rynek, programiści i zespoły deweloperskie korzystają z różnorodnych narzędzi ułatwiających proces tworzenia aplikacji.

W poniższym rozdziale skupiono się na narzędziach wspomagających tworzenie aplikacji pod kątem realizowanego projektu. Podzielony on został na trzy podrozdziały, w których to odpowiednio:

- skupiono się na języku programowania i jego frameworkach
- skupiono się na procesie wytwarzania i budowania aplikacji,
- poruszono zagadnienie wdrożenia i monitorowania systemu.

2.1 Język programowania i jego frameworki

Rozdział ten zawiera informacje na temat użytego języka programowania i frameworka wspomagającego tworzenie aplikacji.

2.1.1 Java

Java to obiektowy, wielozadaniowy język programowania, który został stworzony przez firmę Sun Microsystems (wykupiony przez Oracle Corporation). Jego główne cechy to przenośność, bezpieczeństwo, obiektowość i wielozadaniowość. Java szeroko stosowana jest w tworzeniu różnych aplikacji, od prostych programów na urządzenia mobilne po zaawansowane systemy rozproszone. Jednak to aplikacje webowe klasy enterprise są najczęściej zbudowane przy pomocy tego języka.

Przenośność wynika z tego, że możliwe jest uruchamianie aplikacji na różnych platformach bez konieczności modyfikacji kodu źródłowego. Jest to możliwe dzięki użyciu wirtualnej maszyny java (JVM ang. Java Virtual Machine), która tłumaczy kod źródłowy na kod bajtowy – postać pośrednią, która jest interpretowana przez JVM na różnych systemach operacyjnych.

Bezpieczeństwo wynika z zastosowania mechanizmów takich jak: automatyczne zarządzanie pamięcią, która utrudnia zjawisko wycieku pamięci.

Obiektowość z kolei pozwala na stworzenie kodu posiadającego strukturę, która odpowiada oczekiwaniom aplikacji. Najważniejsze paradygmaty programowania obiektowego to:

- abstrakcja,
- polimorfizm,
- enkapsulacja,
- dziedziczenie.

Wielozadaniowość zaś oznacza, że aplikacje napisane w Javie mogą być wykorzystywane do różnorakich celów – począwszy od aplikacji terminalowych, a kończąc na aplikacjach desktopowych czy webowych [Java Documentation, 2023].

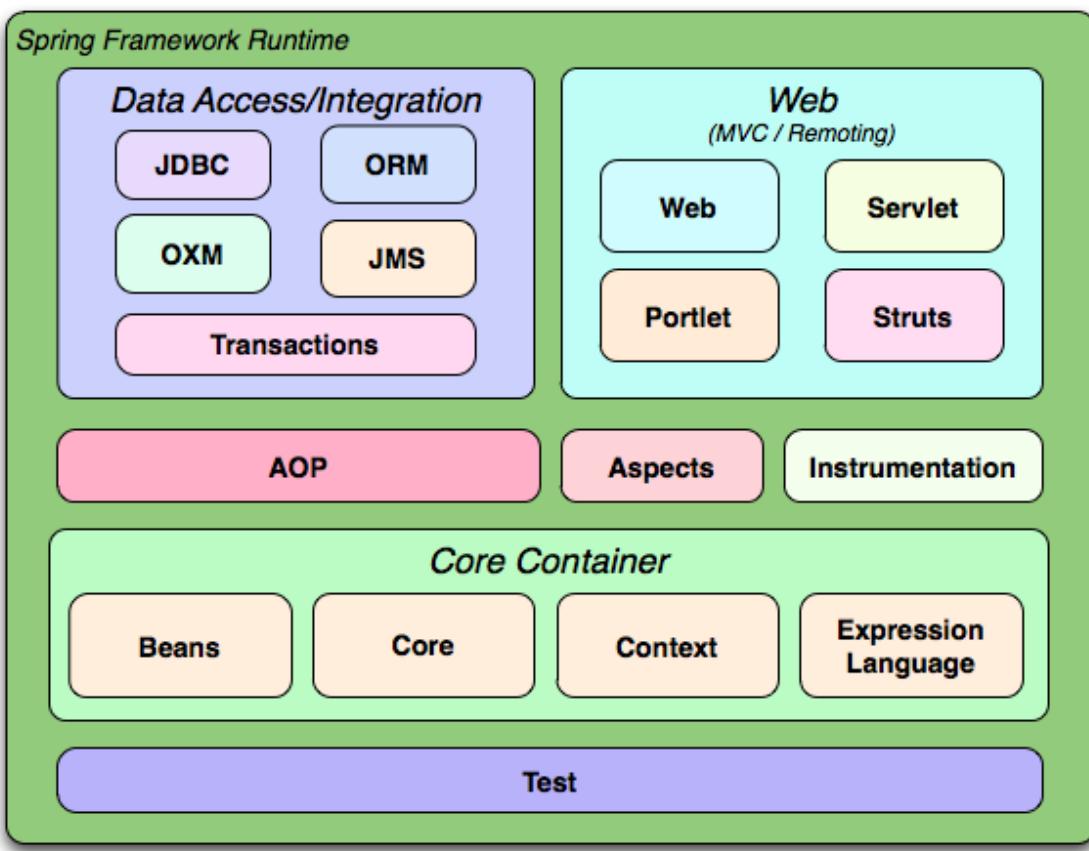
2.1.2 Spring Framework

Spring Framework to kompleksowy, modułowy i bezpieczny framework do tworzenia aplikacji w języku Java i pochodnych. Powstał w celu ułatwienia i przyśpieszenia procesu tworzenia oprogramowania, zwłaszcza aplikacji biznesowych klasy enterprise.

Framework ten powstał w 2003 roku jako odpowiedź na złożoność współczesnych specyfikacji J2EE (ang. Java 2 Enterprise Edition). J2EE wymagał ogromnego narzutu w kwestii tworzenia aplikacji webowych w postaci tworzenia Servletów (klas implementujących rozwiązania HTTP), a także persystencji danych – w tamtym czasie brakowało narzędzi do zarządzania pulą połączeń do bazy danych i narzędzi do mapowania obiektów na ich reprezentację w bazie danych. Chociaż aktualnie specyfikacja J2EE została uproszczona to Spring Framework na stałe zdominował się w ekosystemie aplikacji webowych napisanych w Javie. Głównie z tego powodu, że jest stale udoskonalany i co jakiś czas udostępnia rewolucyjne rozwiązania.

Spring zbudowany jest z modułów co oznacza, że w zależności od potrzeb w aplikacji można używać tylko jego części co czyni to narzędzie niezwykle elastycznym – nie ma potrzeby do małej aplikacji używać wszystkich funkcjonalności sprawiając, że aplikacja spakowana w archiwum *war* lub *jar* posiada wiele niepotrzebnych zależności.

Rysunek 17. Budowa Spring Framework



Źródło: Spring Framework Documentation [dostęp 27.11.2023]

W Springu można wyróżnić pięć głównych modułów:

- Core Container – składa się on z modułów Core, Beans, Context oraz Expression Language. Moduły Core i Beans zapewniają między innymi takie narzędzia jak IoC (ang. Inversion of Control) oraz DI (ang. Dependency Injection) – bardzo istotne z punktu widzenia zarządzania zależnościami. IoC odpowiada za to, że to framework odpowiada za tworzenie obiektów, a nie programista. DI natomiast za to, że Spring sam wstrzykuje odpowiednie zależności. Podmoduł Context odpowiada, za to, aby instancje klasy były odpowiednio ładowane i zarządzane przez framework m.in. poprzez mechanizm proxy. Expression Language dostarcza narzędzi z zakresie wykonywania generycznych zapytań – przydanych m.in. podczas ładowania konfiguracji z plików konfiguracyjnych;
- Data Access/Integration – warstwa dostępu do danych składająca się z modułów JDBC (ang. Java DataBase Connectivity), ORM (ang. Object Relational Mapping), OXM (ang. Object XML), JMS (ang. Java Message Service).

Te moduły zapewniają warstwy integracyjne dla popularnych interfejsów API (ang. Application Programming Interface). Warto jeszcze wspomnieć o module Transaction, który obsługuje zarządzanie transakcjami w przypadku zapisu do bazy danych;

- Web – moduł ten składa się z modułów WWW, Web-Servlet, Web-Struts i Web-Portlet. Zapewniają one podstawowe funkcje integracyjne w środowisku webowym takich jak: komunikacja w stylu REST (ang. Representational State Transfer) czy MVC (ang. Model View Controller);
- AOP (ang. Aspect Oriented Programming) i Instrumentation – zapewniają narzędzia do programowania aspektowego, a także narzędzia do manipulowania kodem bajtowym aplikacji;
- Test – moduł ten dostarcza narzędzi do testowania komponentów Springa za pomocą bibliotek Junit lub TestNG.

Spring Framework pozwala na zarządzanie zależnościami przy użyciu deskryptorów wdrożenia (plików XML, w których zdefiniowane są „beans” i inne zależności) oraz za pomocą adnotacji. Współczesne aplikacje klasy enterprise napisane w tym frameworku wykorzystują drugi sposób. Pierwszy z nich spotykany jest tylko w starych projektach, gdzie narzut pracy potrzebny na migrację do adnotacji byłby zbyt duży. Adnotacji jest bardzo dużo, podstawowe dotyczą głównie tworzenia komponentów:

- @Component – oznacza klasę jako komponent, który ma być zarządzany przez kontener Springa,
- @Autowired – wstrzykuje zależności pomiędzy komponentami w sposób automatyczny,
- @Repository – specjalny rodzaj adnotacji @Component, który dodatkowo wskazuje na to, że dana klasa odpowiada za warstwę pośrednią w kontekście dostępu do danych,
- @Service – specjalny rodzaj adnotacji @Component reprezentujący warstwę biznesową,
- @Controller – komponent będący warstwą obsługującą żądania HTTP (ang. HyperText Transfer Protocol).

Spring Boot z kolei to kolejny framework, który został nadbudowany na podstawie Spring Framework. Ma on na celu jeszcze bardziej uproszczenie procesu tworzenia aplikacji. Zapewnia on bowiem domyślne ustawienia, automatyczną konfigurację, a także wbudowany

serwer aplikacyjny Tomcat. Temu narzędziu przyświeca teza: konwencja ponad konfigurację [Spring Framework Documentation, 2023].

2.2 Narzędzia wspomagające wytwarzanie i budowanie aplikacji

W tym podrozdziale zostaną omówione narzędzia, które wspomagają proces wytwarzania i budowania aplikacji, począwszy od wyboru środowiska developerskiego, poprzez narzędzie używane do komunikacji między nimi, a na narzędziu do budowania aplikacji kończąc.

2.2.1 Zintegrowane środowisko programistyczne – IntelliJ IDEA

Zintegrowane środowisko programistyczne (ang. Integrated Development Environment - IDE) odgrywa kluczową rolę w procesie tworzenia oprogramowania, a wśród tego rodzaju narzędzi wyłania się jedno z najbardziej wyrafinowanych - IntelliJ IDEA. Ta platforma opracowana przez JetBrains stała się nieodłącznym narzędziem dla programistów na całym świecie.

Jednym z kluczowych elementów IntelliJ IDEA jest przyjazny interfejs użytkownika. Dzięki swojej intuicyjnej konstrukcji oraz bogatemu zestawowi narzędzi, zapewnia wygodne środowisko pracy dla programistów. Funkcje takie jak automatyczne uzupełnianie kodu, refaktoryzacja, czy też możliwość natychmiastowej weryfikacji poprawności kodu sprawiają, że praca staje się bardziej efektywna i przyjemna.

IntelliJ IDEA oferuje zaawansowane funkcje uzupełniania kodu. Dzięki mechanizmom sztucznej inteligencji, IDE podpowiada możliwe metody, zmienne oraz inne elementy składniowe, przyspieszając proces pisania kodu z zachowaniem dobrych technik programowania. Dodatkowo, wbudowana analiza statyczna kodu umożliwia wykrywanie potencjalnych błędów jeszcze przed wykonaniem programu, co zdecydowanie ułatwia debugowanie i poprawę kodu, ponieważ nie jest konieczne budowanie aplikacji w celu weryfikacji semantyki.

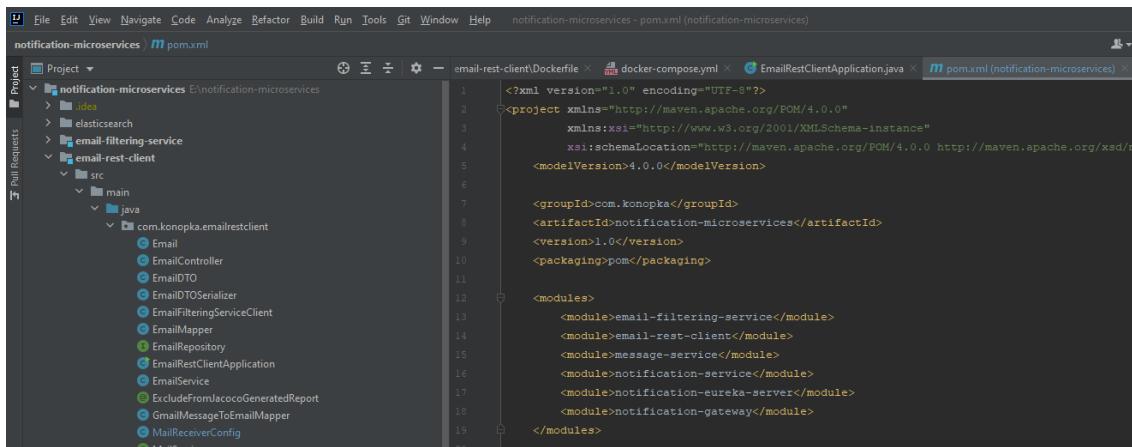
IDEA umożliwia wykonywanie bezpiecznych zmian w strukturze kodu dzięki narzędziom refaktoryzacji. Programiści mogą w łatwy sposób przeprowadzać zmiany w nazwach, strukturze klas czy też ekstrahować powtarzające się fragmenty kodu. Ponadto, IntelliJ IDEA dostarcza narzędzi do zarządzania bazami danych, wspierając integrację z popularnymi systemami.

Jednym z kluczowych atutów IntelliJ IDEA jest wsparcie dla różnorodnych języków programowania i frameworków. Oprócz obsługi języków takich jak Java, Kotlin, JavaScript

czy Python, platforma oferuje szerokie wsparcie dla popularnych frameworków takich jak Spring, Angular czy React, co sprawia, że jest uniwersalnym narzędziem dla programistów pracujących w różnych technologiach.

IntelliJ IDEA stale ewoluje, dostarczając regularnie aktualizacje i ulepszenia. Deweloperzy JetBrains starają się nieustannie dopasowywać narzędzie do zmieniających się potrzeb programistów, wprowadzając nowe funkcje oraz usprawnienia. Ponadto, istnieje aktywna społeczność użytkowników, którzy dzielą się wiedzą, tworząc pluginy oraz udzielając wsparcia na forum, co zwiększa wartość tego narzędzia [IntelliJ Idea Documentation, 2023].

Rysunek 18. Widok ogólny środowiska developerskiego



Źródło: opracowanie własne

2.2.2 Apache Kafka i komunikacja asynchroniczna serwisów

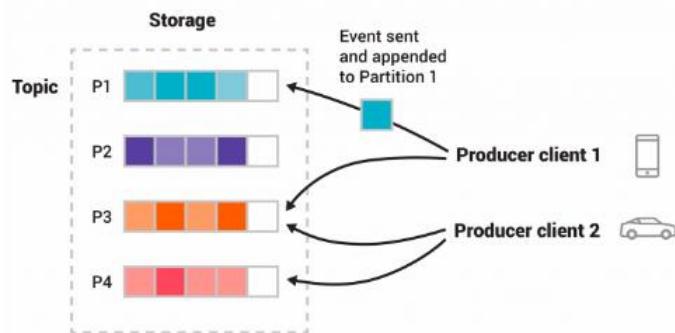
Apache Kafka to otwarte oprogramowanie do przesyłania strumieniowego (ang. streaming) danych, które zostało stworzone w celu obsługi przetwarzania strumieniowego, gromadzenia danych oraz integracji systemów. Jest to rozbudowany system, który umożliwia przesyłanie dużych ilości danych w czasie rzeczywistym pomiędzy różnymi aplikacjami, systemami i komponentami.

W ekosystemie kafki można rozróżnić dwa rodzaje komponentów:

- serwery – instancje oprogramowania, które są odpowiedzialne za obsługę przesyłania strumieniowego danych w ramach klastra. Klastry Kafka składają się z co najmniej jednego brokerów, a zazwyczaj są to multipleksowane instancje uruchomione na różnych maszynach w celu osiągnięcia skalowalności i odporności na awarie;

- klienci – aplikacje lub inne komponenty, które komunikują się z serwerami Kafka, czyli brokerami, w celu przesyłania lub odbierania strumieni danych. W kontekście klientów, można wyróżnić ich dwa rodzaje:
 - producenci – wysyłają dane do brokera,
 - konsumenci – pobierają dane z brokera.

Rysunek 19. Proces wysyłania eventu



Źródło: Apache Kafka Documentation [dostęp 27.11.2023]

Aby zrozumieć ideę działania Apache Kafka ważne jest zdefiniowanie dodatkowo dwóch podstawowych pojęć używanych w tym narzędziu:

- topic – rodzaj strumienia, do którego producenci wysyłają dane, a konsumenci odbierają je. Mogą mieć wiele partycji, co pozwala na równoległe przetwarzanie;
- event – zdarzenie, jednostka informacji, która jest przesyłana pomiędzy producentem a konsumentem. Wiadomość składa się z klucza (ang. key), wartości (ang. value) i metadanych.

Często jako event w rozumieniu aplikacji webowych rozumie się zserializowany obiekt DTO (ang. Data Transfer Object) [Apache Kafka Documentation, 2023].

Rysunek 20. Przykładowy zdeserializowany obiekt klasy MessageDTO

```
{  
    "body": "Body",  
    "emailUuid": "1836bd007ffab57c",  
    "sendDate": "21-09-2022",  
    "status": "SENT"  
}
```

Źródło: opracowanie własne

2.2.3 Maven – automatyzacja budowania aplikacji

Maven to narzędzie stosowane w procesie budowy oprogramowania, znane głównie ze swojej roli w zarządzaniu zależnościami, komplikacji, testowaniu i pakowaniu projektów. Jego głównym celem jest ułatwienie i usprawnienie procesu wytwarzania oprogramowania poprzez standaryzację projektów, zarządzanie zależnościami oraz automatyzację wielu aspektów cyklu życia projektu. Narzędzie to swoje działanie opiera na pliku XML w którym to zawarte są niezbędne dane dotyczące struktury projektu, zależności, konfiguracji oraz lokalnym repozytorium, które pobiera dane na lokalną jednostkę z centralnego repozytorium (maven central).

Rysunek 21. Przykładowy plik POM.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.konopka</groupId>
    <artifactId>notification-microservices</artifactId>
    <version>1.0</version>
    <packaging>pom</packaging>
    <modules>
        <module>email-filtering-service</module>
        <module>email-rest-client</module>
        <module>message-service</module>
        <module>notification-service</module>
        <module>notification-eureka-server</module>
        <module>notification-gateway</module>
    </modules>
    <dependencies>
        <dependency>
            <groupId>com.konopka</groupId>
            <artifactId>email-filtering-service</artifactId>
            <version>1.0</version>
        </dependency>
        <dependency>
            <groupId>com.konopka</groupId>
            <artifactId>email-rest-client</artifactId>
            <version>1.0</version>
        </dependency>
        <dependency>
            <groupId>com.konopka</groupId>
            <artifactId>message-service</artifactId>
            <version>1.0</version>
        </dependency>
        <dependency>
            <groupId>com.konopka</groupId>
            <artifactId>notification-service</artifactId>
            <version>1.0</version>
        </dependency>
    </dependencies>
</project>
```

Źródło: opracowanie własne

W Maven jednostka pracy (ang. goal), wykonuje konkretne zadanie w trakcie cyklu życia projektu. Goale są uruchamiane poprzez wiersz poleceń Mavena lub przez narzędzia IDE (ang. Integrated Development Environment). Operacji możliwych do wykonania jest bardzo wiele, jednakże z punktu widzenia pracy developera najistotniejsze są następujące:

- clean – czyści projekt z plików wykonywalnych,
- validate – sprawdza czy wszystkie dane niezbędne do zbudowania projektu są dostępne,
- compile – kompliluje kod źródłowy programu do kodu bajtowego,
- test – uruchamia testy jednostkowe,
- package – pakuje kod źródłowy do pliku archiwum np. *war*, *jar*,
- verify – uruchamia testy integracyjne, wydajnościowe,

- install – instaluje archiwum w lokalnym repozytorium,
- deploy – wysyła aplikację na serwer artefaktów (jeżeli takowy został skonfigurowany),
- site – generuje stronę www projektu.

Goale wypisane powyżej można pogrupować w trzy grupy:

- grupa odpowiedzialna za czyszczenie projektu: clean,
- grupa odpowiedzialna za budowanie projektu: validate, compile, test, package, verify, install, deploy,
- grupa odpowiedzialna za dokumentację: site.

Warto zauważyć, że jeżeli zostanie wykonana jednostka pracy z grupy np. package wówczas wszystkie etapy poprzedzające ten proces również zostaną wykonane, czyli: validate, compile, test, package. Wynika to z faktu, iż żeby przeprowadzić proces pakowania projektu do archiwum konieczne jest zweryfikowanie czy kod jest poprawny, następnie trzeba go skompilować i wykonać testy jednostkowe [Maven Documentation, 2023].

2.3 Narzędzia wspomagające wdrożenie i monitorowanie stanu aplikacji

W niniejszym podrozdziale zostaną przedstawione narzędzia wspomagające proces wdrożenia oraz monitorowania stanu aplikacji. Wdrożenie aplikacji to nie tylko moment oddania produktu do użytku, lecz również proces, który wymaga ciągłego monitorowania, dostosowywania i optymalizacji. W tym kontekście narzędzia wspierające wdrażanie stają się niezastąpionymi pomocnikami, umożliwiając zarządzanie środowiskiem, automatyzację procesów oraz szybkie reagowanie na ewentualne problemy. Monitorowanie stanu aplikacji to kluczowy element utrzymania wysokiej jakości usług. Narzędzia monitorujące dostarczają informacji na temat wydajności, dostępności i ogólnej kondycji systemu, co pozwala na szybkie identyfikowanie i naprawianie ewentualnych problemów. Wspierają one również proces podejmowania decyzji dotyczących optymalizacji zasobów, skalowania infrastruktury czy planowania przyszłych aktualizacji.

2.3.1 Docker i Docker Compose jako narzędzie usprawniające wdrożenie

Docker jest platformą służącą do konteneryzacji. Można ją uznać za rewolucyjne narzędzie, które zmienia sposób w jaki rozwijane, testowane i wdrażane jest oprogramowanie. Proces tworzenia obrazu to technologia izolacji aplikacji i jej zależności, co pozwala na spakowanie jej w jednostkę zwaną kontenerem. Docker dostarcza narzędzi do tworzenia, uruchamiania i zarządzania tymi kontenerami. Do najważniejszych zalet Dockera należy zaliczyć:

- izolacja – kontenery izolują aplikacje od siebie nawzajem eliminując tym samym konflikty;
- przenośność – konteneryzacja pozwala na to, aby aplikacja mogła być uruchomiona na dowolnym środowisku, które posiada Dockera;
- szybkość – w porównaniu do wirtualnego systemu, na którym mogłyby być uruchomiona aplikacja, kontener uruchamiany jest zazwyczaj w mniej niż 5 minut.

W ekosystemie dockera można wyróżnić jego kluczowe składniki:

- obraz – szablon/wzorzec, który jest używany przez dockera dotworzenia kontenera. Obraz może być przechowywany lokalnie lub w repozytorium. Jest on wersjonowany i niezmienny – co oznacza, że raz utworzony nie może zostać zmieniony;

- kontener – utworzone i odseparowane środowisko dla danej aplikacji. Zostało zbudowane na podstawie obrazu i zawiera tylko dane niezbędne dla uruchomionej aplikacji. Działa jako odrębny proces w systemie Linux (wszystkie kontenery współdzielą jądro systemu);
- dockerfile – deskryptor utworzenia obrazu, w którym zdefiniowane jest w jaki sposób dany obraz ma powstać np. z jakiego systemu operacyjnego ma korzystać;
- docker-compose – narzędzie umożliwiające definiowanie i uruchamianie wielu kontenerów jednocześnie. Pozwala ono na opisanie środowiska za pomocą pliku konfiguracyjnego YML, w którym określona jest konfiguracja, zależności i inne parametry. Używany jest zazwyczaj, gdy wymagane jest stworzenie systemu opartego o architekturę mikroserwisową.

Rysunek 22. Proces tworzenia kontenera



Źródło: <http://dast.webd.pl/podstawy-dockera/> [dostęp 22.11.2023]

W Dockerze komunikujemy się z systemem za pomocą Docker CLI (ang. Command Line Interface) lista najpopularniejszych komend wraz z krótkim opisem znajduje się poniżej:

- docker images – wyświetla listę dostępnych obrazów w lokalnym systemie,
- docker ps – wyświetla listę działających kontenerów,
- docker pull <nazwa obrazu> - pobiera obraz z zewnętrznego repozytorium,
- docker build -t <nazwa obrazu> <ścieżka do pliku Dockerfile> - tworzy obraz na podstawie podanego pliku wdrożenia,
- docker run <nazwa obrazu> - uruchamia nowy kontener na podstawie obrazu,
- docker stop <nazwa kontenera>/<id kontenera> - zatrzymuje działający kontener,
- docker rm <nazwa kontenera>/<id kontenera> - usuwa zatrzymany kontener,
- docker rmi <nazwa obrazu>/<id kontenera> - usuwa obraz na podstawie jego nazwy,
- docker-compose up – uruchamia ekosystem obrazów,

- docker compose down – zatrzymuje i usuwa ekosystem utworzonych kontenerów [Docker Documentation, 2023].

2.3.2 Monitorowanie stanu aplikacji

Prometheus i Grafana to narzędzia stosowane głównie w dziedzinie monitoringu systemów i aplikacji.

Prometheus to system monitorowania i alertowania, który jest zaprojektowany do gromadzenia metryk (takich jak liczba żądań HTTP, zużycie pamięci, obciążenie CPU, czy innych zdefiniowanych przez użytkownika) z różnych komponentów systemu. Prometheus używa modelu danych opartego na czasie, co pozwala na analizę danych w określonym przedziale czasowym. Dodatkowo oferuje mechanizmy alertowania, które umożliwiają reagowanie na potencjalne problemy w czasie rzeczywistym [Prometheus Documentation, 2023].

Grafana to narzędzie do wizualizacji danych, które integruje się z różnymi źródłami danych, w tym z Prometheus. Pozwala użytkownikom tworzyć interaktywne i atrakcyjne wykresy, tabele oraz panele kontrolne na podstawie zebranych metryk. Grafana umożliwia monitorowanie i analizę danych w czasie rzeczywistym, a także dostosowywanie interfejsu użytkownika do indywidualnych potrzeb [Grafana Documentation 2023].

W połączeniu, Prometheus i Grafana tworzą potężne narzędzie do monitorowania, zbierania danych i wizualizacji, które wspomaga zarządzanie i utrzymanie systemów informatycznych.

Rysunek 23. Serwisy wykryte przez Prometheusa, które posiadają metryki

Targets

email-rest-client (1/1 up) <small>show less</small>					
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://email-rest-client:8081/email/api/actuator/prometheus	UP	instance="email-rest-client:8081" job="email-rest-client"	1.605s ago	7.114ms	

filtering-service (1/1 up) <small>show less</small>					
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://email-filtering-service:8082/filtering/api/actuator/prometheus	UP	instance="email-filtering-service:8082" job="filtering-service"	7.102s ago	7.891ms	

message-service (1/1 up) <small>show less</small>					
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://message-service:8083/msg/api/actuator/prometheus	UP	instance="message-service:8083" job="message-service"	7.654s ago	8.734ms	

notification-service (1/1 up) <small>show less</small>					
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://notification-service:8084/notification/api/actuator/prometheus	UP	instance="notification-service:8084" job="notification-service"	8.404s ago	7.312ms	

prometheus (1/1 up) <small>show less</small>					
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://prometheus:8080/metrics	UP	instance="prometheus:8080" job="prometheus"	8.404s ago	7.312ms	

Źródło: opracowanie własne

Rysunek 24. Panel podglądu aplikacji w Grafana



Źródło: <https://grafana.com/oss/grafana/> [dostęp 22.11.2023]

Rozdział 3. Projekt i realizacja rozwiązania

Niniejszy rozdział zawiera opis części praktycznej projektu aplikacji pobierającej wiadomości z poczty email i wysyłającej powiadomienia SMS. Zostały w nim opisane założenia dla aplikacji, mikroserwisy oraz sposób implementacji.

3.1 Identyfikacja i analiza wymagań

W tej części zostały zawarte kroki, które zostały podjęte na samym początku tworzenia aplikacji. Skupiono się tutaj na analizie wymagań.

3.1.1 Założenia projektu

Nadrzędnym celem projektu jest wspomaganie użytkownika w zapamiętywaniu o ważnych wydarzeniach poprzez wysyłanie powiadomień SMS – w podstawowej wersji aplikacji żaden interfejs graficzny nie jest konieczny, ale jego obecność ułatwi zrozumienie działania aplikacji. Pozwoli na ręczne dodawanie wiadomości i dalszy rozwój funkcjonalności poprzez m.in. dodanie formularza do dodawania filtrów w przyszłości. W celu stworzenia aplikacji posiadającej oczekiwane funkcje należy zdefiniować wymagania funkcjonalne i niefunkcjonalne, zarówno obligatoryjne jak i fakultatywne. Dzięki takiemu podejściu explicite można określić czy stworzony produkt spełnia oczekiwania.

Obligatoryjne wymagania funkcjonalne:

- OF-1 – aplikacja działa z pocztą w domenie *gmail.com*,
- OF-2 – odczytywanie wszystkich wiadomości z poczty i zapis w bazie danych,
- OF-3 – filtrowanie odczytywanych wiadomości z poczty i na podstawie spełnionych kryteriów tworzenie wzoru wiadomości SMS i zapis w bazie danych,
- OF-4 – wysyłanie wiadomości SMS przy użyciu serwisu *twilio.com* dostarczającego usługi VOIP (ang. Voice Over Internet Protocol) na 2 dni przed terminem określonym w wygenerowanej wiadomości,
- OF-5 – każdorazowa aktualizacja stanu wysłanej wiadomości,
- OF-6 – aplikacja po wdrożeniu nie wymaga ingerencji użytkownika końcowego,
- OF-7 – monitorowanie stanu aplikacji poprzez agregację logów.

Fakultatywne wymagania funkcjonalne:

- FF-1 – klient webowy pozwalający na przeglądanie pobranych wiadomości email,
- FF-2 – klient webowy pozwalający na przeglądanie statusu stworzonych powiadomień SMS,

- FF-3 – klient webowy pozwalający na ręczne dodawanie wiadomości do wysłania w formie SMS,
- FF-4 – klient webowy pozwalający na ręczne wywołanie akcji wysyłania powiadomień dla niewysłanych wiadomości,
- FF-5 – monitorowanie stanu aplikacji przy pomocy narzędzia do wizualizacji zdarzeń (stan uruchomienia aplikacji i inne parametry).

Obligatoryjne wymagania niefunkcjonalne:

- ONF-1 – skalowalność aplikacji,
- ONF-2 – dostępność serwisu tylko w wewnętrznej sieci.

Fakultatywne wymagania niefunkcjonalne:

- FNF-1 – przenośność aplikacji – powinna być możliwość uruchomienia na dowolnej platformie,
- FNF-2 – niski koszt sprzętowy,
- FNF-3 – możliwość skorzystania z usług chmurowych do wdrożenia aplikacji na produkcję.

3.1.2 Analiza systemowa – podział na serwisy

Postanowiono, że aby umożliwić realizację obligatoryjnych celów funkcjonalnych takich jak skalowalność aplikacji oraz fakultatywnych niefunkcjonalnych takich jak przenośność aplikacji i niski koszt sprzętowy dobrym rozwiązaniem architektonicznym, jeżeli chodzi o projektowany system będą mikroserwisy.

Analizując cechy aplikacji można wyróżnić w niej kilka serwisów, które muszą spełniać określone funkcjonalności by aplikacja działała zgodnie z oczekiwaniami:

- serwis odpowiedzialny za komunikację z Gmail API pobierający wiadomości z poczty email,
- serwis odpowiedzialny za filtrowanie odebranych wiadomości,
- serwis odpowiedzialny za tworzenie na podstawie wyfiltrowanych wiadomości wzorce wiadomości SMS,
- serwis wysyłający powiadomienia SMS.

Na tej podstawie zgodnie z regułą SOLID, a uściślając z pierwszą literą tego akronimu S – Single Responsibility Principle, gdzie każda część systemu powinna mieć jedną odpowiedzialność postanowiono, że zostaną stworzone cztery mikroserwisy funkcjonalne:

- email-rest-client – mikroserwis odpowiedzialny za komunikację z Gmail API i pobieranie wiadomości email,
- email-filtering-service – mikroserwis odpowiedzialny za filtrowanie wiadomości wysłanej przez email-rest-client i dający informację zwrotną czy dany mail spełnia określone kryteria,
- message-service – mikroserwis odpowiedzialny za generowanie wiadomości SMS,
- notification-service – mikroserwis odpowiedzialny za wysyłanie powiadomień SMS.

Aby wymienione wyżej mikroserwisy mogły ze sobą współpracować i aby nie definiować explicite endpointów dla każdego z nich potrzebne będą mikroserwisy narzędziowe, które pozwolą na łatwiejszą komunikację i “wykrywanie się” wzajemnie. Dzięki bibliotekom zawartym w Spring Framework jest to możliwe, zatem zostaną stworzone:

- notification-eureka-server – mikroserwis odpowiedzialny za komunikację między mikroserwisami; dzięki niemu nie będzie konieczne definiowanie adresu IP mikroserwisów – wystarczą unikalne nazwy;
- notification-gateway – mikroserwis pozwalający na komunikację klientowi końcowemu tylko poprzez jeden główny endpoint; dzięki temu klient nie musi być świadomy tego, że ruch jest dystrybuowany na inne endpointy do określonego mikroserwisu.

3.1.3 Wspólne API dla serwisów

W związku z tym, że mikroserwisy muszą mieć wspólne reprezentacje obiektów, tak aby komunikacja między nimi przebiegała bez utraty istotnych informacji z perspektywy serwisu, który te informacje odbiera oraz żeby nie eksponować danych, które są kluczowe dla encji, warto rozpocząć projektowanie od wspólnego API. Jako że docelowa aplikacja nie będzie bardzo rozbudowana postanowiono, iż zamiast tworzenia interfejsów i ich implementacji w każdym z mikroserwisów z osobna zostaną zdefiniowane klasy DTO (ang. Data Transfer Object), które będą nośnikiem informacji podczas komunikacji.

Rysunek 25. Klasa EmailDTO

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class EmailDTO {

    @Schema(description = "From who is an email", name = "from",
             required = true, example = "example@example.com")
    private String from;
    @Schema(description = "Subject", name = "subject",
             required = true, example = "Example subject")
    private String subject;
    @Schema(description = "Body", name = "body",
             required = true, example = "Body")
    private String body;
    @Schema(description = "Sent date", name = "date",
             required = true, example = "Mon, 12 Sep 2022 19:29:39 +0200")
    private String date;
    @Schema(description = "UUID of received email", name = "messageId",
             required = true, example = "18332c00ef8c07fb")
    private String messageId;

}
```

Źródło: opracowanie własne

EmailDTO – klasa odpowiedzialna za transfer informacji po pobraniu wiadomości email z Gmail API. Każda wiadomość email posiada takie informacje jak:

- from – pole określające nadawcę wiadomości,
- subject – temat wiadomości,
- body – treść wiadomości,
- date – data wysłania wiadomości,
- messageId – unikalny identyfikator wiadomości.

Rysunek 26. Klasa FilterDTO

```
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
public class FilterDTO {  
  
    @Schema(description = "Key for a filter", name = "key",  
            required = true, example = "example@example.com")  
    private String major;  
    @Schema(description = "Value for a filter", name = "value",  
            required = true, example = "Invoice")  
    private String val;  
}
```

Źródło: opracowanie własne

FilterDTO – klasa odpowiedzialna za transfer informacji klucz-wartość dla każdego z emaili:

- major – klucz,
- val – wartość.

Rysunek 27. Klasa MessageDTO

```
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
public class MessageDTO {  
  
    @Schema(description = "Body", name = "body",  
             required = true, example = "Body")  
    private String body;  
    @Schema(description = "UUID of received email", name = "emailUuid",  
             required = true, example = "18332c00ef8c07fb")  
    private String emailUuid;  
    @Schema(description = "Deadline", name = "date",  
             required = true, example = "23-02-2023")  
    @JsonFormat(pattern = "dd-MM-yyyy")  
    private LocalDate sendDate;  
    @Schema(description = "Status", name = "date",  
             required = true, example = "NOT_SENT")  
    private Status status;  
  
}
```

Źródło: opracowanie własne

MessageDTO – klasa odpowiedzialna za transfer informacji zawartości utworzonej wiadomości:

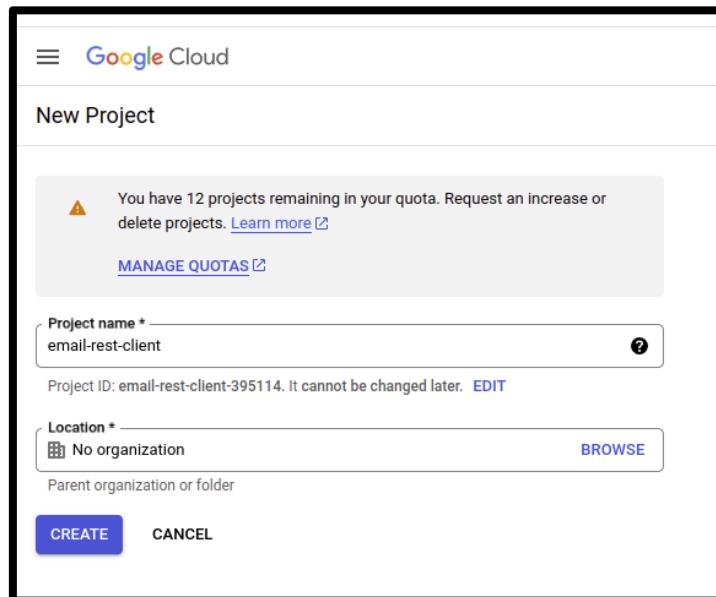
- body – treść wiadomości,
- emailUuid – unikalny identyfikator wiadomości email,
- sendDate – data dla wysłania wiadomości,
- status – status wysłania wiadomości.

3.1.4 Komunikacja z Gmail API

Aby móc korzystać z odczytywania maili przy pomocy interfejsu Gmail API należy po utworzeniu poczty email na tej platformie wykonać szereg operacji w celu otrzymania danych uwierzytelniających, które pozwolą stronie trzeciej (np. aplikacji webowej) na odczytywanie wiadomości.

Proces utworzenia „client_id” i „secret_id” należy zacząć od odwiedzenia strony <https://console.cloud.google.com>, a następnie utworzenia nowego projektu tak jak na zrzucie ekranu poniżej.

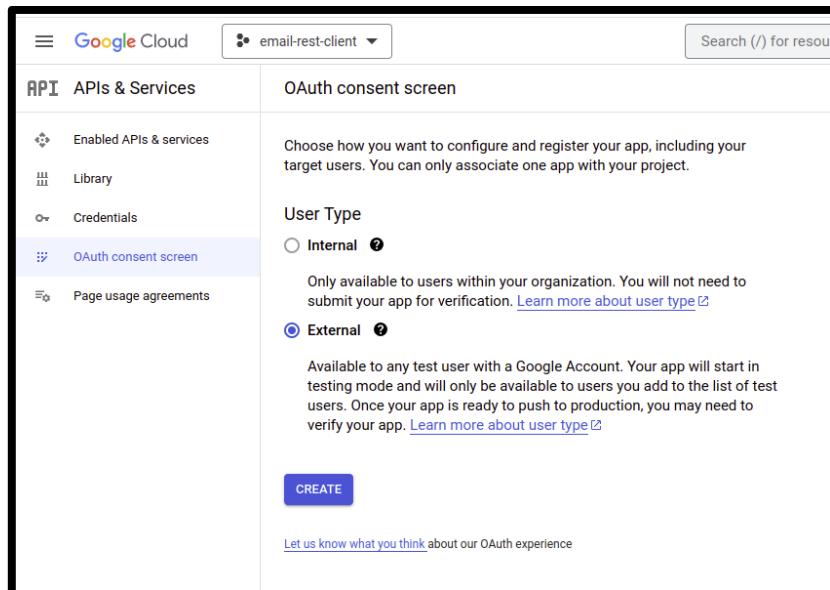
Rysunek 28. Dodawanie nowego projektu



Źródło: opracowanie własne

Następnie należy skonfigurować klienta OAuth. W celu konfiguracji należy otworzyć zakładkę „OAuth consent screen” i udostępnić interfejs każdemu zewnętrznemu użytkownikowi (np. aplikacji).

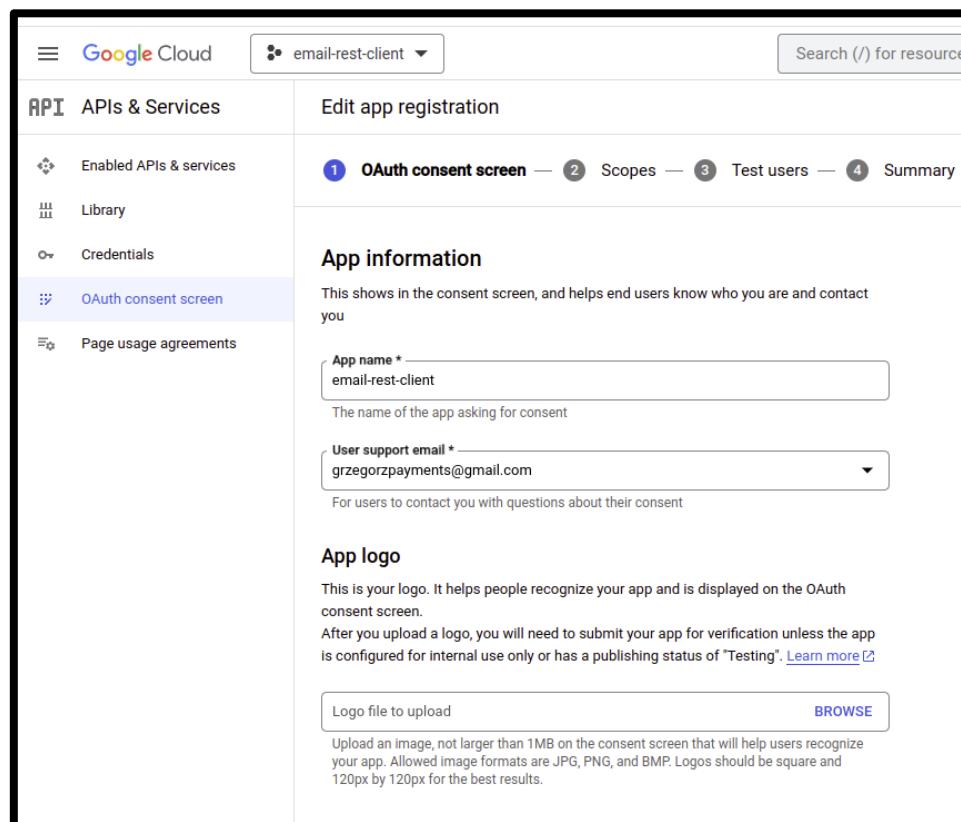
Rysunek 29. Nadawanie uprawień zewn. klientom na autoryzację przy pomocy OAuth



Źródło: opracowanie własne

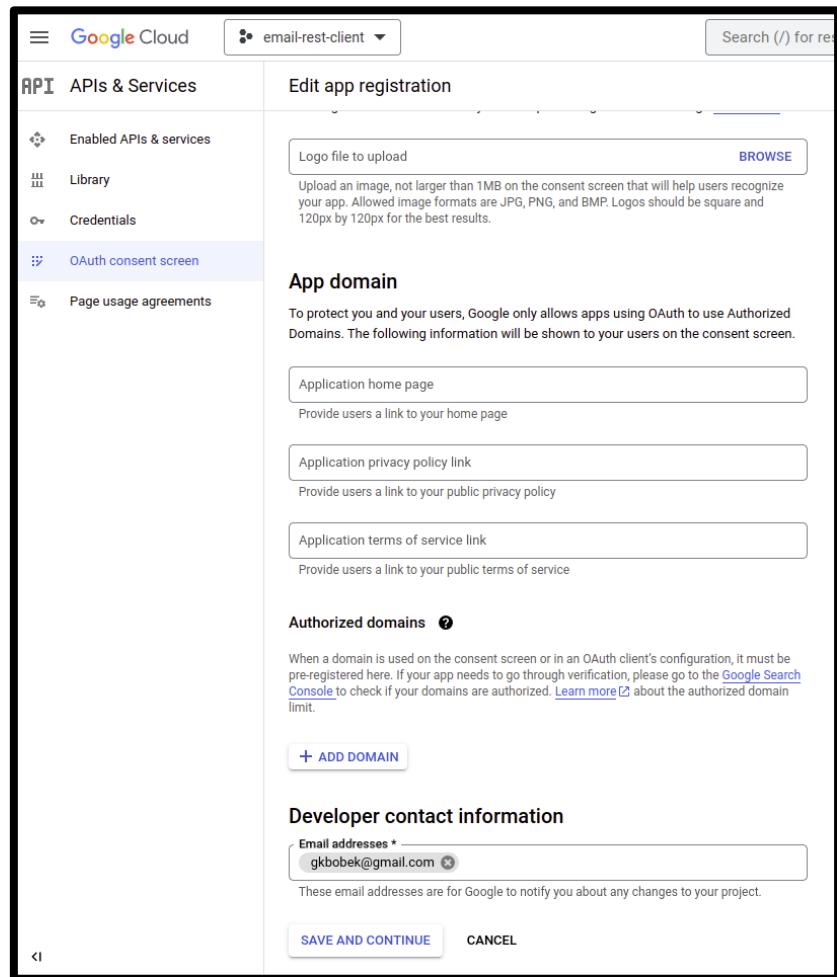
Po utworzeniu nowej konfiguracji dla klienta należy przejść przez proces utworzenia żetonu odświeżania (ang. refresh token) dla aplikacji. Proces ten składa się z 4 etapów. W pierwszym z nich należy potrzebne będzie zdefiniowanie nazwy aplikacji oraz podanie adresu email w domenie gmail.com dla którego planowane jest skorzystanie z OAuth.

Rysunek 30. Etap pierwszy przy uzyskiwaniu refresh token, część I



Źródło: opracowanie własne

Rysunek 31. Etap pierwszy przy uzyskiwaniu refresh tokena, część II



Źródło: opracowanie własne

W kolejnym etapie istnieje możliwość nadania określonych uprawień naszej aplikacji.

W przypadku email-rest-client proces ten można pominąć.

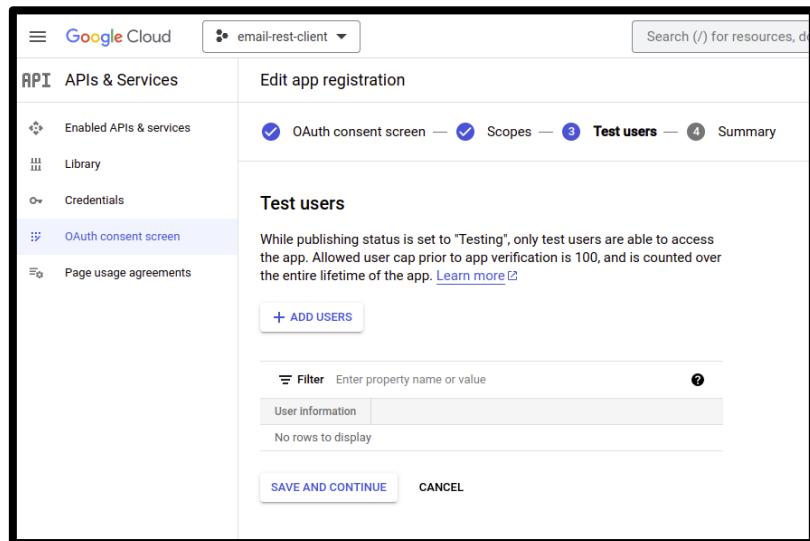
Rysunek 32. Etap drugi, nadawanie uprawień

The screenshot shows the 'Edit app registration' page for an OAuth consent screen. The left sidebar lists 'Enabled APIs & services', 'Library', 'Credentials', and 'OAuth consent screen' (which is selected). The main area has tabs for 'OAuth consent screen' (selected), 'Scopes' (highlighted in blue), 'Test users', and 'Summary'. A note explains that scopes express permissions requested from users. An 'ADD OR REMOVE SCOPES' button is present. Below, sections for 'Your non-sensitive scopes', 'Your sensitive scopes', and 'Your restricted scopes' show tables with no rows. At the bottom are 'SAVE AND CONTINUE' and 'CANCEL' buttons.

Źródło: opracowanie własne

Następny etap to nadanie uprawnień do korzystania z OAuth testowym użytkownikom. Podobnie jak w operacji poprzedniej, nie planuje się udostępnić tokenu tylko testowym użytkownikom, lecz wszystkim którzy taki token będą posiadać.

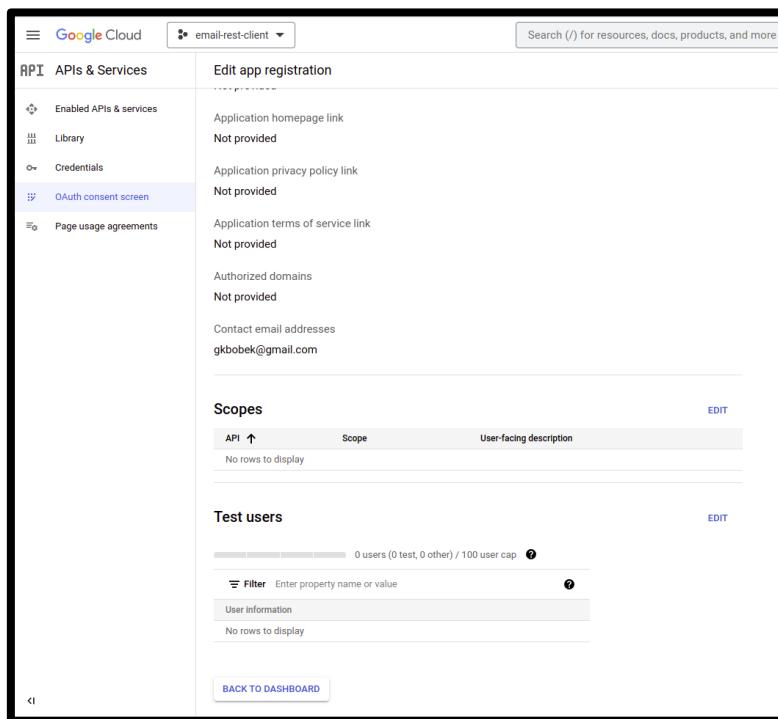
Rysunek 33. Etap trzeci, nadawanie uprawień testowym użytkownikom



Źródło: opracowanie własne

Ostatni etap stanowi podsumowanie, w którym to zostaną zaprezentowane dotychczasowe wybory.

Rysunek 34. Etap czwarty, podsumowanie

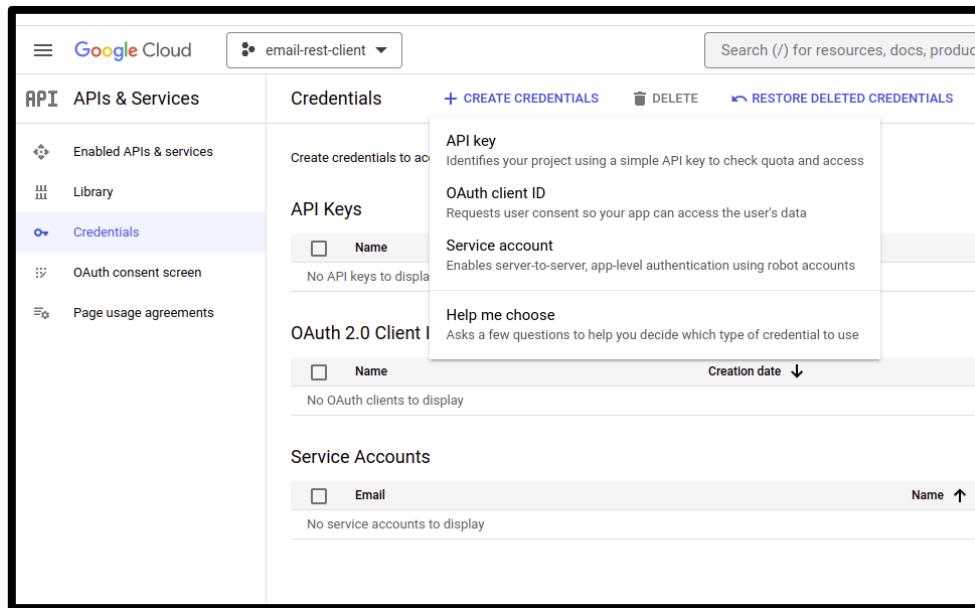


Źródło: opracowanie własne

Po przejściu całego procesu i powrocie do ekranu głównego na platformie <https://console.cloud.google.com> należy utworzyć OAuth Client ID. Aby to zrobić wystarczy

kliknąć w zakładkę „Credentials” dostępną po lewej stronie, a następnie kliknąć „Create credentials” i z menu rozwijalnego wybrać „OAuth Client ID”.

Rysunek 35. Tworzenie OAuth Client ID



Źródło: opracowanie własne

Następnie należy podać typ aplikacji dla jakiej zamierza się utworzyć klucz OAuth, nazwę aplikacji i kliknąć przycisk „Create” tak jak na rysunku poniżej.

Rysunek 36. Tworzenie OAuth Client ID, część II

The screenshot shows the 'Create OAuth client ID' dialog box in the Google Cloud Platform. On the left, there's a sidebar with 'APIs & Services' and 'Credentials' selected. The main area has a title 'Create OAuth client ID'. It includes fields for 'Application type' (set to 'Web application'), 'Name' (set to 'email-rest-client'), and a note about adding authorized domains. Below this are sections for 'Authorized JavaScript origins' (with a note about browser requests) and 'Authorized redirect URIs' (with a note about web server requests). At the bottom are 'CREATE' and 'CANCEL' buttons.

Źródło: opracowanie własne

Wracając do okna „Credentials” będzie można zauważyc, że nowy klucz został utworzony. Można go pobrać. Dzięki informacjom w nim zawartym aplikacja będzie w stanie komunikować się z systemem poczty elektronicznej przy użyciu „client_id” i „secret_id”, który pozwoli na uzyskanie access tokena i refresh tokena.

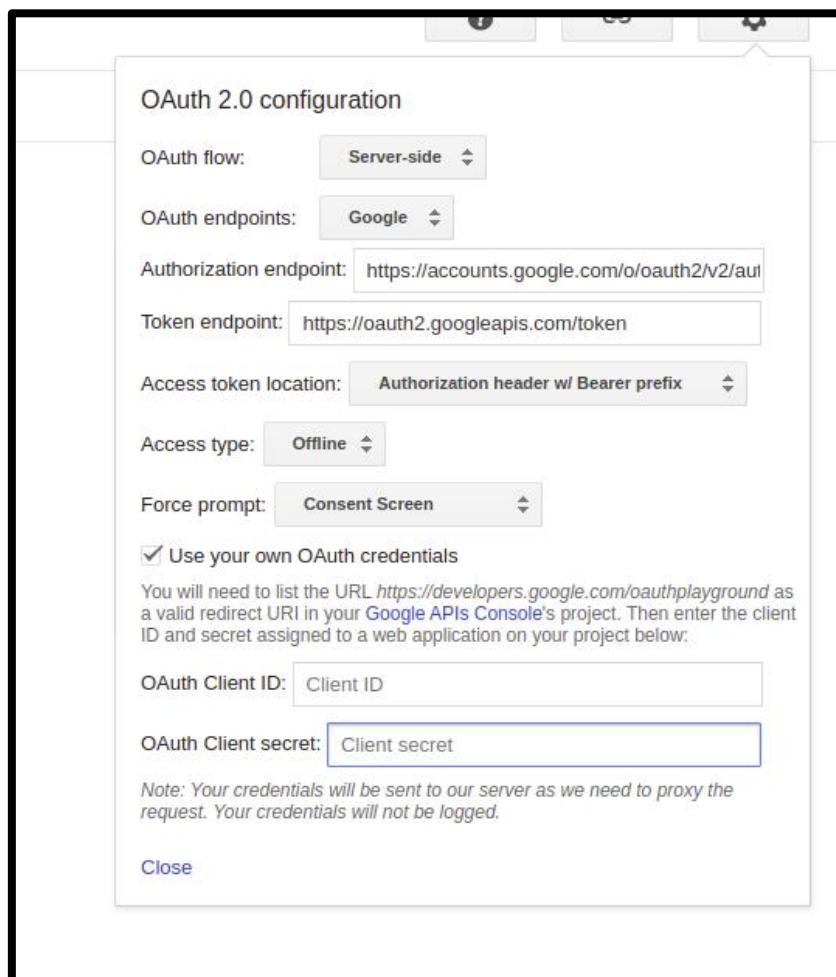
Rysunek 37. Wygenerowany klucz OAuth

OAuth 2.0 Client IDs				
Name	Creation date	Type	Client ID	Actions
email-rest-client	Aug 6, 2023	Web application	794599199998-Fidv...	

Źródło: opracowanie własne

Te dwie pary klucz-wartość będą potrzebne, żeby otrzymać refresh tokena, który jest niezbędny do uwierzytelnienia się w aplikacji. Aby tego dokonać należy odwiedzić stronę <https://developers.google.com/oauthplayground> a następnie wejść w ustawienia i dodać nowo utworzone pary klucz-wartość.

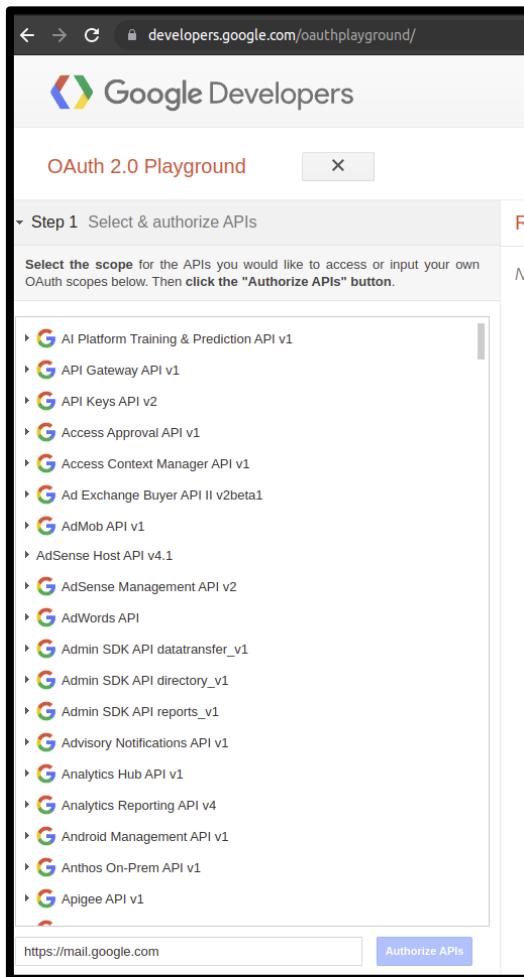
Rysunek 38. Dodawanie pary klucz - wartość



Źródło: opracowanie własne

Następnie trzeba podać dla jakiej usługi chcemy wygenerować refresh tokena. Można to zrobić poprzez znalezienie Gmail API na liście rozwijalnej lub podać bezpośrednio adres do poczty email.

Rysunek 39. Dodawanie usługi Gmail do uzyskania refresh tokena

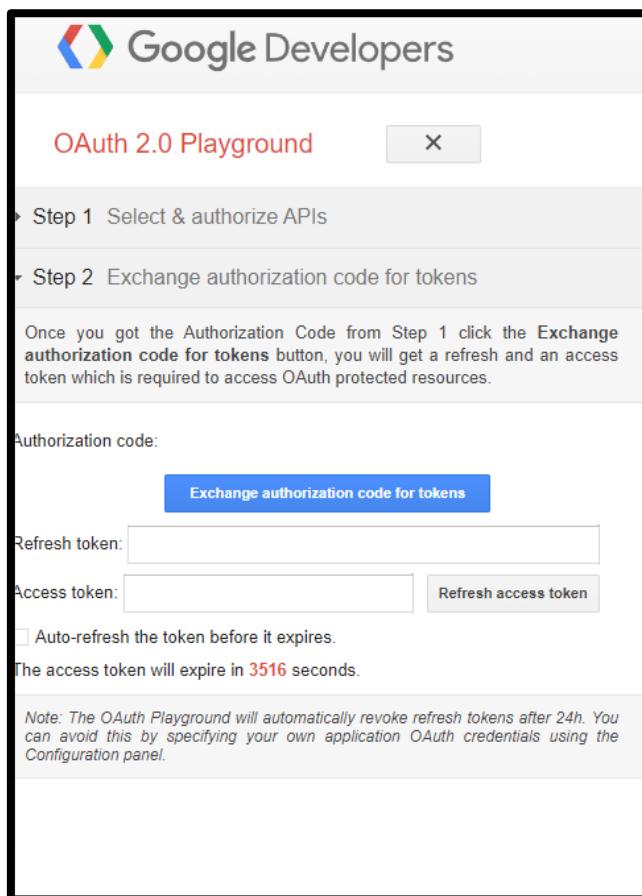


Źródło: opracowanie własne

W kolejnej operacji należy się zalogować na swoje konto poczty, a gdy ta operacja zakończy się sukcesem będzie można uzyskać refresh tokena przy użyciu przycisku „Exchange authorization code for tokens”.

Wykonana operacja jest ostatnią podczas procesu generowania refresh tokena. Aby móc korzystać z niego w aplikacji zaleca się dodanie wartości dla tego klucza do zmiennych środowiskowych systemu operacyjnego oraz zapisanie pliku json z „secret_id” i „client_id” dostępnych w panelu utworzonych aplikacji w konsoli Google'a.

Rysunek 40. Wygenerowany refresh i access token



Źródło: opracowanie własne

Do komunikacji z usługą poczty w domenie gmail.com niezbędne będzie skorzystanie z biblioteki od Google, która pozwala na komunikację z tym serwisem. Aby to zrobić, do projektu należy dodać zależności w pliku pom.xml tak jak na rysunku poniżej.

Rysunek 41. Niezbędne zależności umożliwiające korzystanie z Gmail API

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-mail</artifactId>
</dependency>
<dependency>
    <groupId>com.google.api-client</groupId>
    <artifactId>google-api-client</artifactId>
    <version>1.25.0</version>
</dependency>
<dependency>
    <groupId>com.google.apis</groupId>
    <artifactId>google-api-services-gmail</artifactId>
    <version>v1-rev110-1.25.0</version>
</dependency>
<dependency>
    <groupId>com.google.auth</groupId>
    <artifactId>google-auth-library-oauth2-http</artifactId>
    <version>1.3.0</version>
</dependency>
```

Źródło: opracowanie własne

Zależności te, pozwolą korzystanie z klas udostępnianych przez Gmail do utworzenia sesji użytkownika z serwisem poczty, a także uwierzytelnienie. Rysunek poniżej przedstawia konfigurację klasy odpowiedzialnej za komunikację z serwisem Gmail w mikroserwisie email-rest-client.

Rysunek 42. Konfiguracja klasy odpowiedzialnej za komunikację z Gmail API

```
@Slf4j
@Configuration
public class MailReceiverConfig {

    @Value("${spring.application.name}")
    private String APPLICATION_NAME;

    @Value("${spring.gmail.refresh.token}")
    private String REFRESH_TOKEN;

    private static final GsonFactory JSON_FACTORY = GsonFactory.getDefaultInstance();

    @Bean
    @Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    public Gmail gmail() throws IOException, GeneralSecurityException {
        final InputStreamReader in = new InputStreamReader(Objects.requireNonNull(getClass().getClassLoader()
                .getResourceAsStream("credentials.json")));
        Gmail service;
        GoogleClientSecrets clientSecrets = load(JSON_FACTORY, in);

        Credential authorize = new GoogleCredential.Builder()
                .setTransport(GoogleNetHttpTransport.newTrustedTransport())
                .setJsonFactory(JSON_FACTORY)
                .setClientSecrets(clientSecrets.getDetails().getClientId(), clientSecrets.getDetails()
                        .getClientSecret())
                .build()
                .setRefreshToken(REFRESH_TOKEN);

        final NetHttpTransport HTTP_TRANSPORT = GoogleNetHttpTransport.newTrustedTransport();
        service = new Gmail.Builder(HTTP_TRANSPORT, JSON_FACTORY, authorize)
                .setApplicationName(APPLICATION_NAME).build();
        return service;
    }
}
```

Źródło: opracowanie własne

Odpowiedzialność utworzenia obiektu zostaje dana frameworkowi Spring przy użyciu adnotacji `@Bean`. Dodatkowo cykl życia instancji klasy został określony jako prototyp, co oznacza, że każdorazowa potrzeba skorzystania z obiektu klasy `Gmail` spowoduje utworzenie nowej instancji. Jest to niezbędne, ponieważ do logowania aplikacja będzie korzystać z access tokena, który ma ważność 3600 sekund, a proces generowania odbywa się na samym początku tworzenia instancji przy użyciu refresh tokena – co oznacza, że w innym przypadku po jednej godzinie aplikacja utraciłaby dostęp do odczytywania wiadomości.

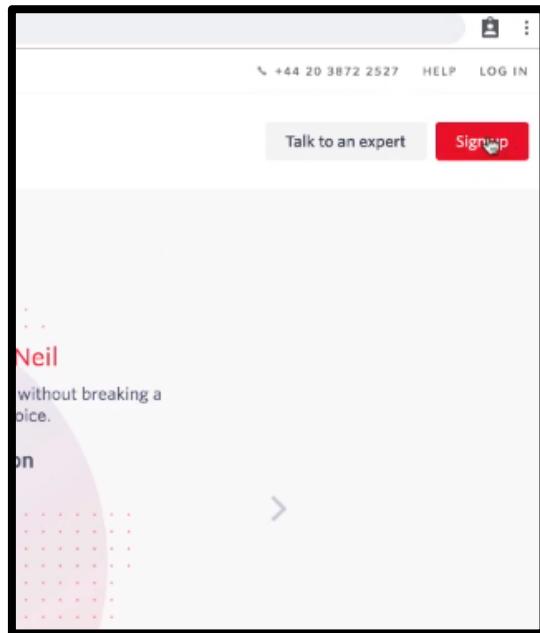
3.1.5 Komunikacja z Twilio API

Twilio to amerykańska firma z siedzibą w San Francisco w Kalifornii, która dostarcza programowalne narzędzia komunikacyjne do nawiązywania i odbierania połączeń telefonicznych, wysyłania i odbierania wiadomości tekstowych oraz wykonywania innych funkcji komunikacyjnych za pomocą interfejsów API usług sieciowych. W swojej ofercie

ma również czasową usługę wysyłania powiadomień SMS, która pozwala na wysłanie blisko 1000 wiadomości. Na potrzeby aplikacji takie konto w zupełności spełnia oczekiwania.

Korzystanie z Twilio API jest podobne jak w przypadku Gmail API. Proces należy rozpocząć od utworzenia konta, a następnie podanie niezbędnych informacji.

Rysunek 43. Pierwszy etap tworzenia konta



Źródło: opracowanie własne

Rysunek 44. Drugi etap tworzenia konta

A screenshot of the Twilio website's sign-up page, specifically the second step. The heading reads 'Get started with a free Twilio account. No credit card required.' Below this are four input fields: 'First Name' and 'Last Name' in separate boxes, followed by 'Email' and 'Choose a password' in adjacent boxes. To the right of the password fields is another box labeled 'Password, again'. At the bottom left is a red 'Get Started' button. Next to it is a small note: 'By clicking the button, you agree to our [legal policies](#).'. At the very bottom, there is a link 'Already have an account? [Login](#)'.

Źródło: opracowanie własne

Następnie należy podać numer telefonu, aby uwierzytelnić nowo utworzone konto.

Rysunek 45. Uwierzytelnianie konta przy pomocy numeru telefonu

Verify Your Identity

The screenshot shows a "Verify Phone Number" interface. At the top, there's a "NUMBER" field with a dropdown menu set to "+44" and a "Phone Number" input field containing "+441622322331". To the right of the input fields is a link "Why verify a phone number?". Below these are two buttons: a blue "Verify" button and a red "Skip" button. A note below the input fields states: "We will contact you at the number above with a verification code". At the bottom, there's a checkbox with the text: "The phone number you provide will be used for authentication when you login to Twilio Console. A Twilio onboarding specialist may also use this number to reach out with free onboarding support. If you do not want to be contacted at this phone number, please check this box." The checkbox is currently unchecked.

Źródło: opracowanie własne

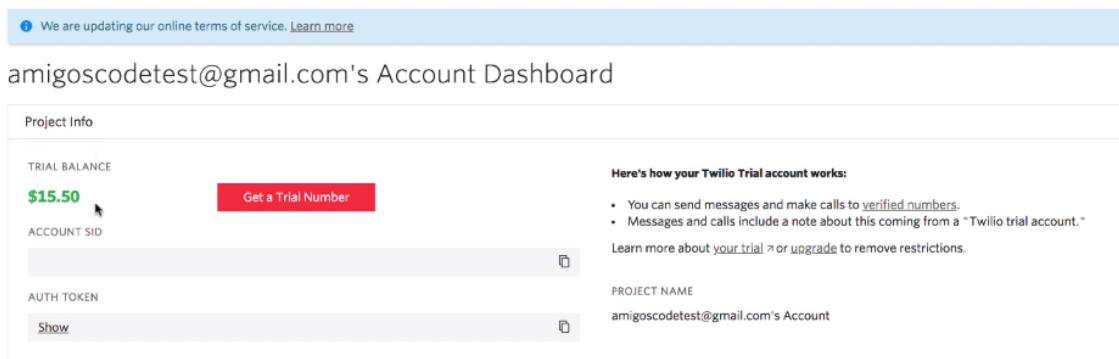
Rysunek 46. Akceptacja wyg. numeru tel. do wysyłania powiadomień SMS



Źródło: opracowanie własne

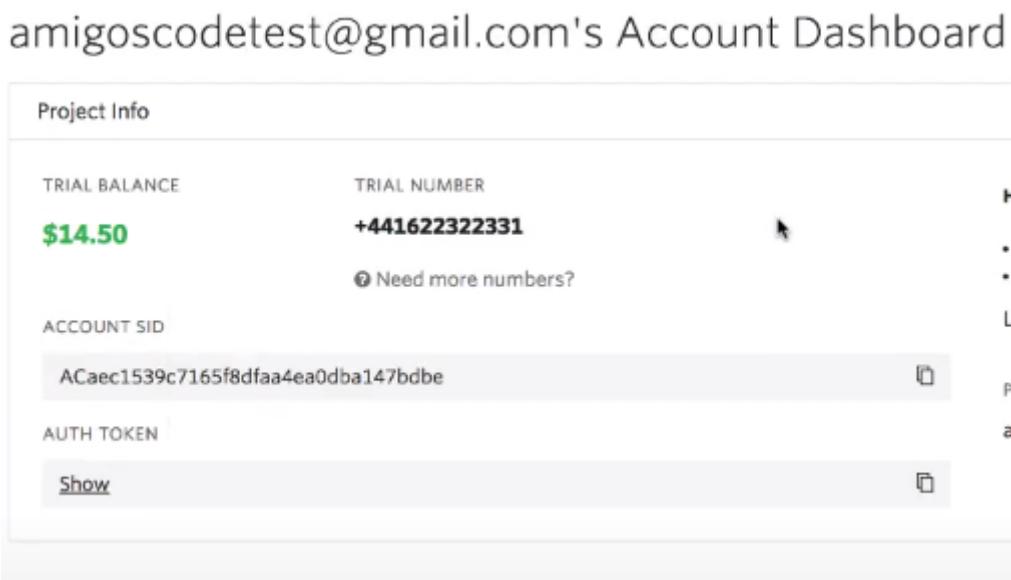
Po akceptacji zaproponowanego numeru telefonu przez system w panelu konsoli zostaną utworzone dane do uwierzytelnienia. Po wykonaniu tego procesu w konsoli będą dostępne „account SID” i „auth token”. Posłużą one jako dane niezbędne do klasy konfiguracyjnej notification-service. Dzięki nim, możliwa będzie komunikacja mikroserwisu z Twilio API.

Rysunek 47. Panel użytkownika



Źródło: opracowanie własne

Rysunek 48. Dane uwierzytelniające



Źródło: opracowanie własne

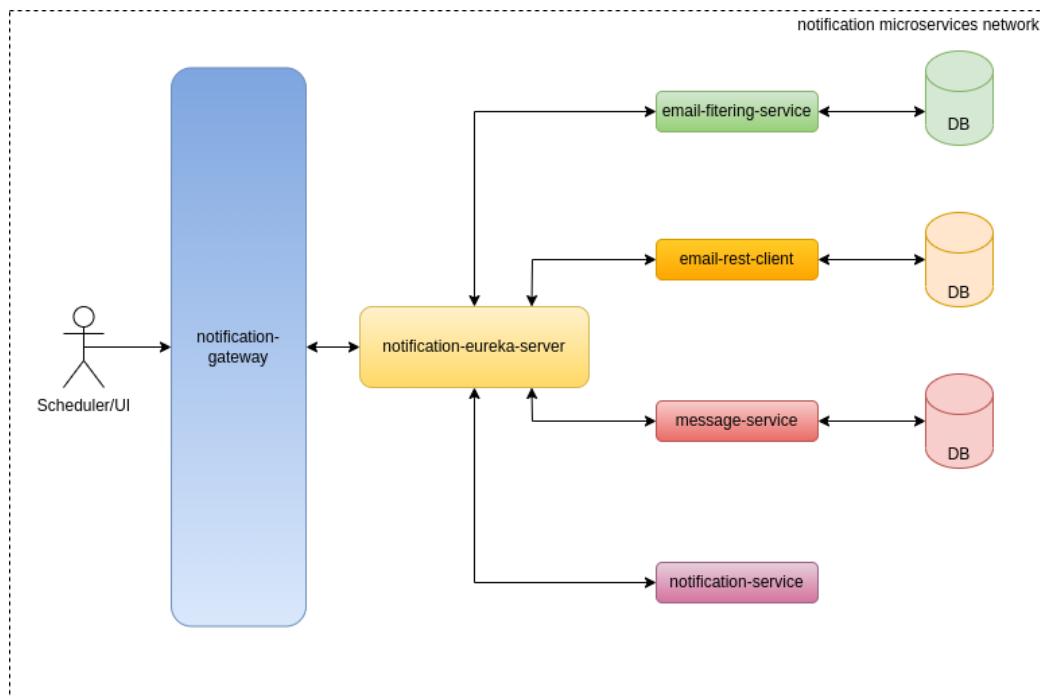
3.2 Mikroserwisy

Schemat ideowy poniżej przedstawia architekturę powstałej aplikacji. Zgodnie z informacjami zawartymi w podrozdziale 3.1.2, w którym to została opisana analiza systemowa rozwiązywanego zagadnienia, mikroserwisy zostały podzielone na:

- narzędziowe - na schemacie widoczne jako notification-gateway oraz notification-eureka-server,
- usługowe - na schemacie oznaczone jako email-filtering-service, email-rest-client, message-service, notification-service.

Wszystkie mikroserwisy usługowe z wyjątkiem notification-service (służący do wysyłania powiadomień SMS) posiadają własną bazę danych. Bazy te nie są zależne od siebie w żaden sposób tj. nie przechowują danych zależnych od siebie zatem nie występuje tutaj popularny problem dla architektury rozproszonej w postaci problemu transakcyjności dla relacyjnych baz danych (ang. ACID – Atomicity, Consistency, Isolation, Durability).

Rysunek 49. Uproszczona architektura aplikacji



Źródło: opracowanie własne

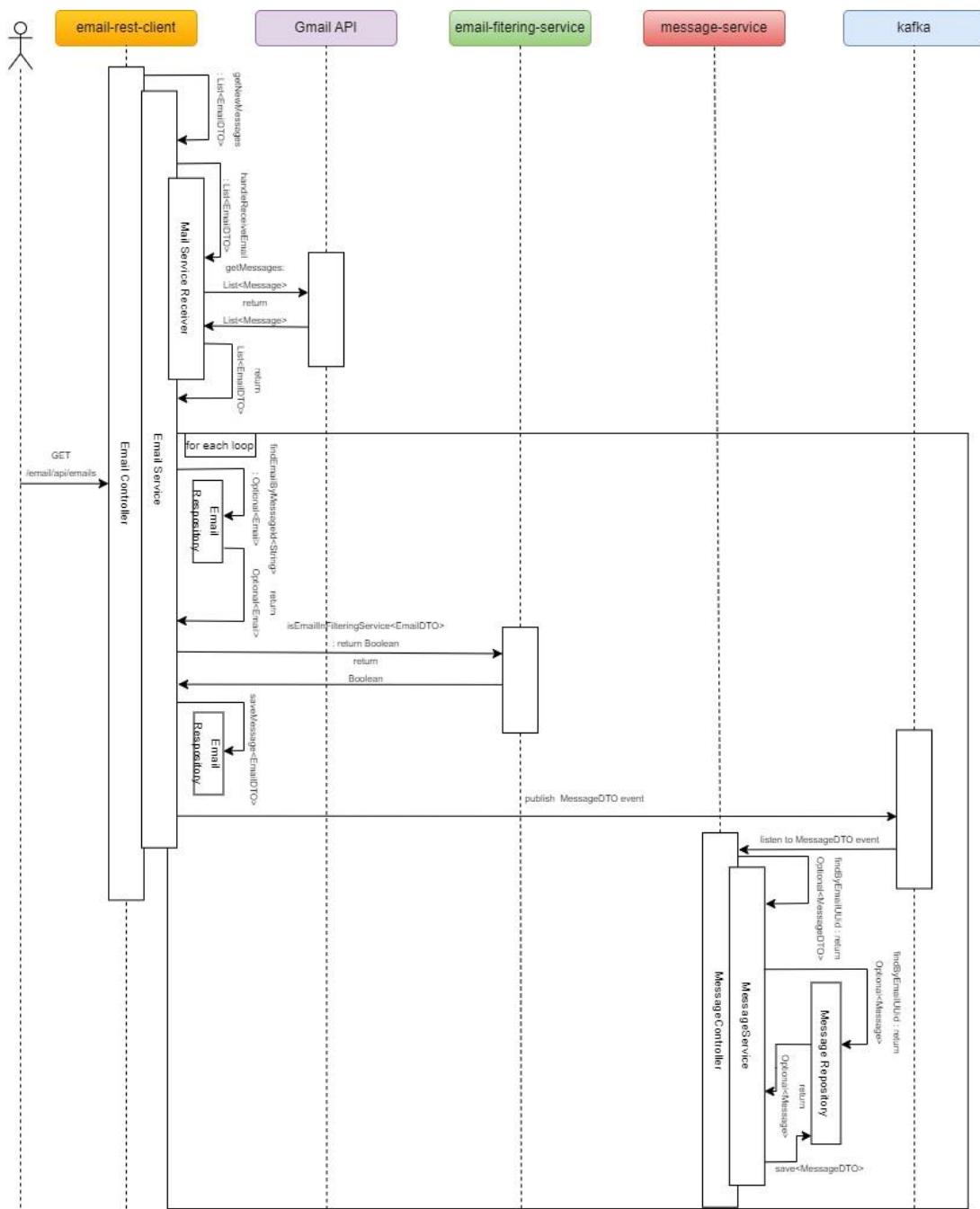
Kolejne diagramy sekwencji przedstawiają cały proces działania aplikacji – od momentu odczytania wiadomości email z poczty w domenie gmail.com do momentu wysłania wiadomości SMS. Odczytywanie wiadomości email jest dostępne w dwóch wariantach. W przypadku uruchamiania aplikacji w trybie developerskim (lokalnie) komunikacja między mikroserwisami występuje tylko w sposób synchroniczny – poprzez wysyłanie żądań HTTP. Dla aplikacji uruchomionej w trybie rozproszonym tj. przy użyciu kontenerów dockerowych występuje dodatkowy serwis (Kafka) odpowiedzialny za komunikację asynchroniczną przy użyciu eventów.

Z diagramów wynika, że interakcja następuje między 5 serwisami – w przypadku komunikacji asynchronicznej. Proces przetwarzania informacji rozpoczyna się przez interakcję użytkownika końcowego z warstwą frontendową aplikacji lub poprzez schedulera

umieszczonego w email-rest-client za pomocą żądania HTTP GET na adres: <http://email-rest-client:8080/email/api/emails>. Żądanie HTTP jest obsługiwane przez warstwę kontrolera email-rest-client, który następnie przekazuje zapytanie do warstwy biznesowej, czyli do Email Service. Ten z kolei przekazuje zapytanie do warstwy pośredniej, czyli do Mail Service Receiver. Ta warstwa aplikacji odpowiada za bezpośrednią komunikację z Gmail API – to w niej znajdują się odpowiednie zależności umożliwiające odpytanie poczty. Serwis Gmail zwraca wszystkie wiadomości email, a następnie są one mapowane na listę obiektów DTO (Data Transfer Object), które zostają przekazane z powrotem do Email Service. Następnie każda z wiadomości jest weryfikowana z warstwą infrastruktury RDBS (ang. Relational DataBase System) w postaci Email Repository. Jeżeli dana wiadomość nie znajduje się w bazie danych następuje zapytanie przy pomocy żądania HTTP GET filtering-service czy dana wiadomość email spełnia kryteria co do autora wiadomości i jej treści. Jeżeli nie spełnia tych kryteriów, to w tym miejscu proces się kończy. Natomiast gdy zostają one spełnione Email Service zapisuje wiadomość email w bazie danych przy ponownym użyciu Email Repository. Po wykonaniu tej operacji EmailDTO jest przekształcany na inny obiekt DTO tj. MessageDTO, który po deserializacji zostaje wysłany do Kafki (część asynchroniczna komunikacji między mikroserwisami). MessageController w message-service nasłuchuje na eventy typu MessageDTO. Gdy taki otrzyma przekazuje zdeserializowany obiekt do Message Service, który następnie sprawdza czy dana wiadomość nie znajduje się w bazie - jeżeli encji o podanym UUID (ang. Universally Unique Identifier) nie ma, kolejną, a zarazem ostatnią czynnością jest zapis do bazy danych.

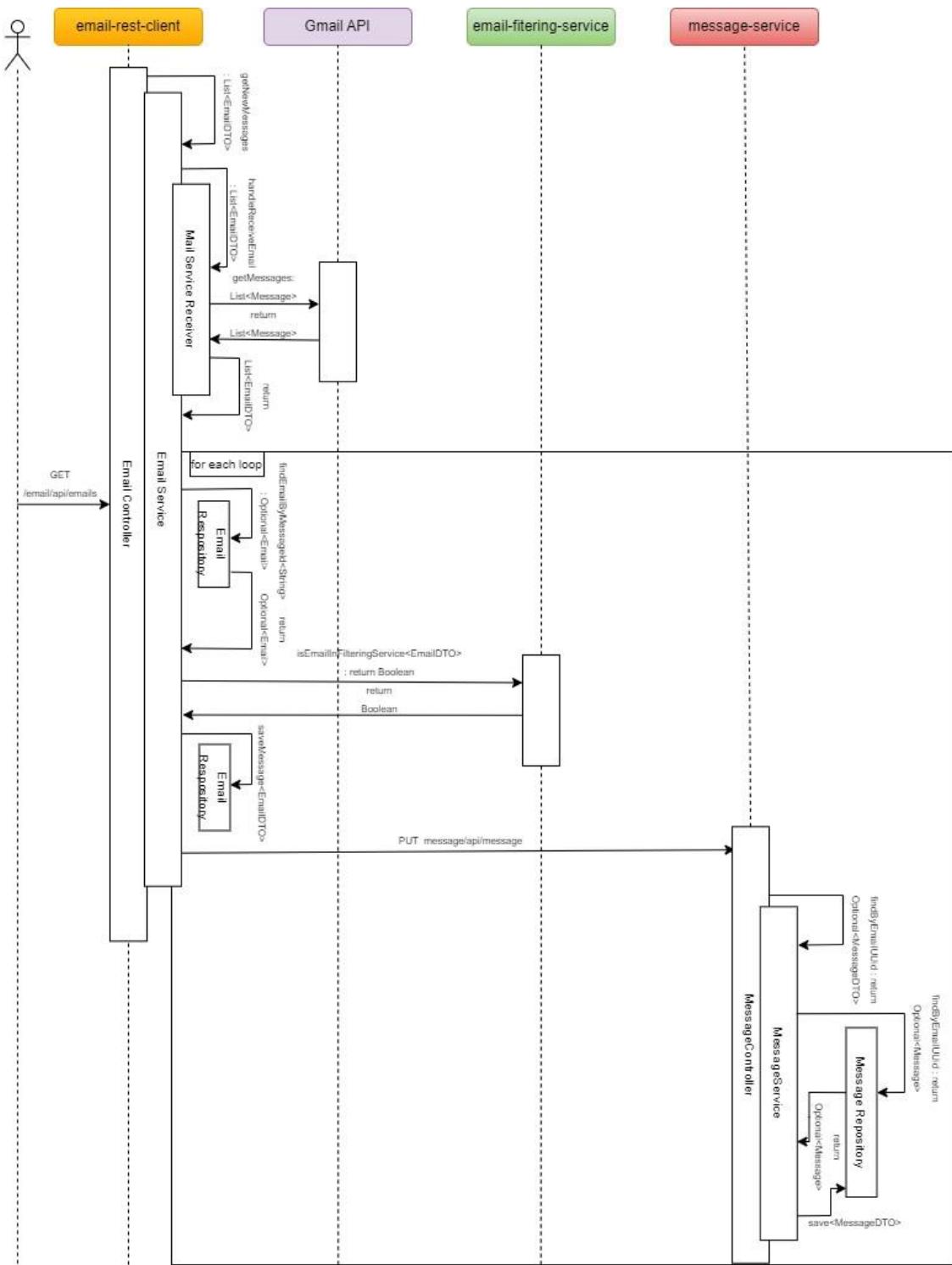
Schemat przedstawiony na rysunku 51 praktycznie nie różni się od tego przedstawionego na rysunku 50 – tutaj proces z punktu widzenia wartości biznesowej przebiega tak samo. Dzięki użyciu polimorfizmu inwokacja metod przebiega w ten sam sposób – różnice stanowią szczegóły implementacyjne. W tym przypadku nie występuje zjawisko wysyłania eventów, ale aktualizacja statusu wysłanej wiadomości przez bezpośrednie żądanie HTTP.

Rysunek 50. Diagram sekw. odczytywania wiadomości email (asynchroniczny)



Źródło: opracowanie własne

Rysunek 51. Diagram sekw. odczytywania wiadomości email (synchroniczny)

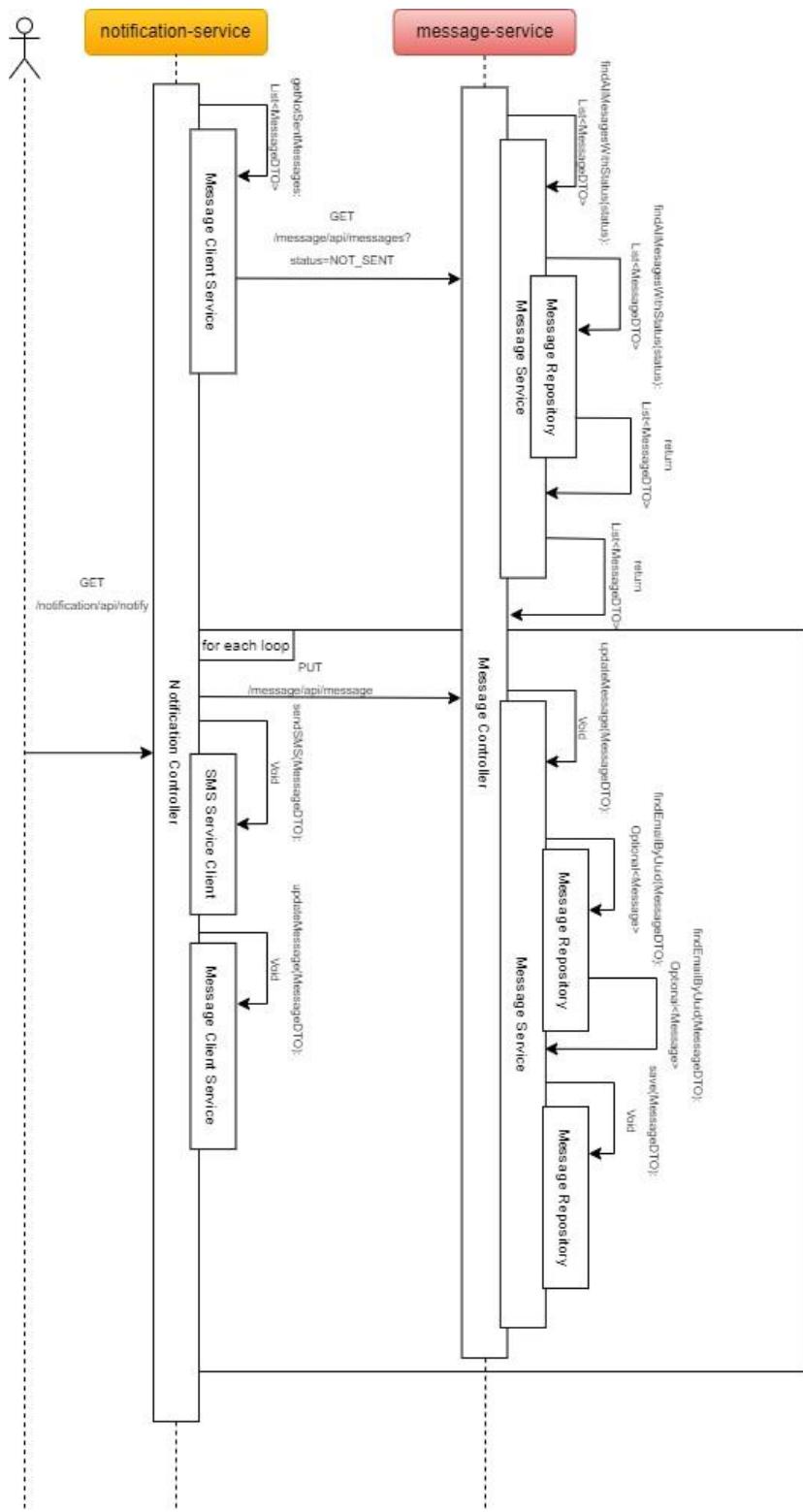


Źródło: opracowanie własne

W przypadku wysyłania powiadomienia SMS, zarówno podczas uruchamiania aplikacji w trybie developerskim jak i rozproszonym cały proces rozpoczyna się od wysłania żądania

HTTP przez warstwę frontendową aplikacji lub schedulera uruchamianego o określonej porze. Następnie warstwa kontrolera w notification-service przy pomocy Message Client Service pobiera listę wiadomości, które posiadają status: nie wysłany. Message Service z kolei wykonuje niezbędne operacje do wykonania żądania HTTP i zwraca odpowiedź do notification-service. Następnie lista pobranych wiadomości SMS, jeżeli spełnia kryteria dot. wykonywalności, następuje żądanie HTTP do klienta zewnętrznego (Twilio API) i wysłanie wiadomości, a następnie aktualizacja jej statusu w serwisie message-service na: wysłany.

Rysunek 52. Diagram wysyłania powiadomień SMS



Źródło: opracowanie własne

3.2.1 Serwis do odczytywania wiadomości email

Serwis jako całość posiada jedną odpowiedzialność, więc nie było potrzeby dzielenia klas na pod-pakiety. Wszystkie klasy znajdują się w pakiecie: *com.konopka*, a ich lista jest dostępna poniżej:

- Email.java – reprezentuje encję w bazie danych,
- Status.java – odzwierciedla status wiadomości: wysłany lub niewysłany,
- EmailMapper.java – klasa narzędziowa odpowiedzialna za przekształcanie wiadomości email z bazy danych na obiekt DTO (ang. Data Transfer Object),
- EmailDTO.java – odpowiada przekształconej wiadomości email na obiekt dostępny w warstwie serwisowej,
- MessageDTO.java – obiekt typu DTO (ang. Data Transfer Object) reprezentujący wiadomość SMS,
- EmailDTOSerializer.java – klasa odpowiedzialna za serializację wiadomości email,
- GmailMessageToEmailMapper.java – klasa narzędziowa odpowiedzialna za przekształcanie wiadomości email otrzymaną przez serwis poczty Gmail na obiekt email używany przez aplikację,
- EmailRepository.java – reprezentuje repozytorium tj. dostęp do bazy danych,
- EmailService.java – warstwa biznesowa aplikacji odpowiedzialna za pobieranie wiadomości email oraz zapis do bazy danych przy pomocy EmailRepository,
- MessageServiceClient.java – klient serwisu message-service w przypadku korzystania z komunikacji synchronicznej (przy pomocy żądań HTTP),
- MessageServiceEventProducer.java – odpowiada za wysyłanie zdeserializowanego obiektu do brokera Kafka,
- EmailFilteringServiceClient.java – klient serwisu email-filtering-service,
- EmailController.java – warstwa kontrolera odpowiedzialna za przyjmowanie żądań HTTP,
- MailServiceReciever.java – klient odpowiedzialny za komunikację z Gmail API (ang. Application Programming Interface),
- MailRecieverConfig.java – klasa konfiguracyjna,
- OpenApiConfiguration.java – klasa konfiguracyjna dla tworzenia dokumentacji,

- EmailRestClientApplication.java – główna klasa stanowiąca wejście do programu i uruchamiająca kontekst aplikacji.

3.2.2 Serwis do filtrowania wiadomości email

Serwis odpowiedzialny za filtrowanie wiadomości jest jednym z mniejszych mikroserwisów w kontekście ekosystemu wysyłania notyfikacji. Wszystkie klasy odpowiedzialne za wykonanie filtrowania znajdują się w pakiecie *com.konopka*:

- EmailDTO.java – obiekt DTO (ang. Data Transfer Object) reprezentujący wiadomość email,
- EmailFilteringServiceApplication.java – główna klasa stanowiąca wejście do programu i uruchamiająca kontekst aplikacji,
- Filter.java – reprezentuje encję w bazie danych, forma filtra to para klucz – wartość,
- FilterDTO.java – przekształcona encja na obiekt DTO (ang. Data Transfer Object),
- FilteringController.java – warstwa kontrolera odpowiedzialna za przyjmowanie żądań HTTP,
- FilterMapper.java – klasa odpowiedzialna za przekształcanie encji na obiekt DTO (ang. Data Transfer Object),
- FilterRepository.java – warstwa dostępu do bazy danych,
- FilterService.java – warstwa biznesowa odpowiedzialna za komunikację z bazą danych i mapowanie,
- OpenApiConfiguration.java – klasa konfiguracyjna dla tworzenia dokumentacji.

3.2.3 Serwis do tworzenia wiadomości SMS

Serwis, który odpowiada za tworzenie wiadomości SMS jest drugim, jeżeli chodzi o wielkość mikroserwisem. Podobnie jak w przypadku pozostałych serwisów w związku z tym, że posiada jedną odpowiedzialność nie występują w nim pod-pakiety. Struktura projektu prezentuje się następująco:

- EmailDTO.java – obiekt DTO (ang. Data Transfer Object) reprezentujący wiadomość email,
- EmailDTOConsumer.java – klasa odpowiedzialna za konsumowanie wiadomości z brokera Kafki,

- EmailDTODeserializer.java – klasa odpowiedzialna za deserializację obiektu z brokera Kafka i utworzenie na jej podstawie obiektu EmailDTO,
- KeyPattern.java – encja reprezentująca rekord w bazie danych,
- KeywordsRepository.java – warstwa dostępu do danych,
- Message.java – encja reprezentująca rekord w bazie danych,
- MessageController.java – warstwa kontrolera odpowiedzialna za przyjmowanie żądań HTTP,
- MessageDTO.java – obiekt typu DTO (ang. Data Transfer Object) reprezentujący wiadomość SMS,
- MessageMapper.java – klasa narzędziowa odpowiedzialna za przekształcanie wiadomości SMS z bazy danych na obiekt DTO (ang. Data Transfer Object),
- MessageParsingService.java – warstwa serwisowa odpowiedzialna za przekształcanie obiektu EmailDTO w MessageDTO przy udziale warstwy dostępu do danych (ang. Data Transfer Object),
- MessageRepository.java – warstwa dostępu do danych,
- MessageService.java – warstwa biznesowa odpowiedzialna za komunikację z warstwą dostępu do danych (odczytywanie i zapis),
- MessageServiceApplication.java – główna klasa stanowiąca wejście do programu i uruchamiająca kontekst aplikacji,
- MessageServiceConfig.java – klasa konfiguracyjna,
- OpenApiConfiguration.java – klasa konfiguracyjna dla dokumentacji,
- Status.java – odzwierciedla status wiadomości: wysłany lub niewysłany,
- Template.java – encja reprezentująca wzorzec wiadomości,
- TemplateRepository.java – warstwa dostępu do danych.

3.2.4 Serwis do wysyłania notyfikacji

Mikroserwis odpowiedzialny za wysyłanie notyfikacji jest najmniejszym mikroserwisem spośród serwisów usługowych. Nie posiada warstwy dostępu do danych. Odpowiedzialny jest jedynie za wysyłanie powiadomienia SMS i aktualizację statusu wysłania wiadomości poprzez komunikację z message-service. Lista klas wraz z krótkim opisem odpowiedzialności znajduje się poniżej:

- MessageClientService.java – klient odpowiedzialny za komunikację z message-service (aktualizacja stanu wysłanej wiadomości),

- MessageDTO.java – obiekt typu DTO (ang. Data Transfer Object) reprezentujący wiadomość SMS,
- NotificationController.java – warstwa kontrolera odpowiedzialna za przyjmowanie żądań HTTP,
- NotificationServiceApplication.java – główna klasa stanowiąca wejście do programu i uruchamiająca kontekst aplikacji,
- NotificationServiceConfig.java – klasa konfiguracyjna,
- OpenApiConfiguration.java – klasa konfiguracyjna do dokumentacji,
- SmsServiceClient.java – klient odpowiedzialny za komunikację z Twilio API (ang. Application Programming Interface),
- Status.java – odzwierciedla status wiadomości: wysłany lub niewysłany.

3.2.5 Klient do podglądu aplikacji

Aplikacja będąca klientem dla wymienionych powyżej mikroserwisów została napisana w framework'u React. W przypadku aplikacji napisanej w tym ekosystemie zamiast klas używa się komponentów. Lista komponentów wraz z krótkim opisem odpowiedzialności znajduje się poniżej:

- Index.js – komponent odpowiedzialny za punkt startowy aplikacji,
- App.js – komponent odpowiedzialny za widok główny aplikacji,
- Header.js – komponent, którego główną funkcją jest wygenerowanie wraz z funkcjonalnościami paska nawigacyjnego,
- HomeContent.js – wyświetlanie zawartości strony na stronie głównej pod paskiem nawigacyjnym,
- MessageList.js – pobieranie i wyświetlanie listy wiadomości SMS otrzymanej od warstwy backendu,
- EmailList.js – pobieranie i wyświetlanie listy wiadomości email otrzymanej od warstwy backendu,
- AddNewMessage.js – komponent pozwalający na wyświetlenie formularza do ręcznego tworzenia wiadomości SMS,
- MessageForm.js – komponent pozwalający na wypełnienie danych i wysłanie ich do warstwy backendu.

Notification-microservices-client jest aplikacją typu CRUD (ang. Create Read Update Delete) z wyłączoną funkcjonalnością usuwania – z założenia raz utworzona wiadomość nie może zostać usunięta, a proces usuwania wiadomości email nie stanowi założeń projektu. Zrzuty ekranu poniżej przedstawiają w obrazowej formie wszystkie funkcjonalności aplikacji.

Rysunek 53. Ekran powitalny wraz z paskiem nawigacyjnym



Źródło: opracowanie własne

Rysunek 54. Ekran wiadomości email (widok ogólny)

ID	From	Body	Sent date
df55a06a-6640-4a3d-9893...	Example Example <example@example.co...	New payment	21-02-2022
df55a06a-6640-4a3d-9893...	Example Example <example@example.org>	New invoice	22-02-2022
df55a06a-6640-4a3d-9893...	Example Example <example@example.biz>	New cancellation	23-02-2022

1-3 of 3 < >

Źródło: opracowanie własne

Rysunek 55. Ekran wiadomości email (widok tabeli)

ID ↑	From	Body	Sent date
df55a06a-6640-4a3d-9893...	Example Example <example@example.co...	New payment	21-02-2022
df55a06a-6640-4a3d-9893...	Example Example <example@example.org>	New invoice	22-02-2022
df55a06a-6640-4a3d-9893...	Example Example <example@example.biz>	New cancellation	23-02-2022

1-3 of 3 < >

Źródło: opracowanie własne

Rysunek 56. Ekran wiadomości SMS (widok ogólny)

ID	Description	Deadline	Status
e976f0d4-22d1-48b...	Message	16-10-2021	NOT_SENT
e976f0d4-22d1-48b...	Message	16-10-2021	NOT_SENT
e976f0d4-22d1-48b...	Message	16-10-2021	SENT
e976f0d4-22d1-48b...	Message	16-10-2021	SENT



Źródło: opracowanie własne

Rysunek 57. Ekran wiadomości SMS (widok tabeli)

ID	Description	Deadline	Status
e976f0d4-22d1-48b...	Message	16-10-2021	NOT_SENT
e976f0d4-22d1-48b...	Message	16-10-2021	NOT_SENT
e976f0d4-22d1-48b...	Message	16-10-2021	SENT
e976f0d4-22d1-48b...	Message	16-10-2021	SENT

Źródło: opracowanie własne

Rysunek 58. Formularz ręcznego dodawania wiadomości SMS

16-10-2021

Add new Message

Body

UUID

Select roles:

NOT_SENT

Deadline

rrrr-mm-dd

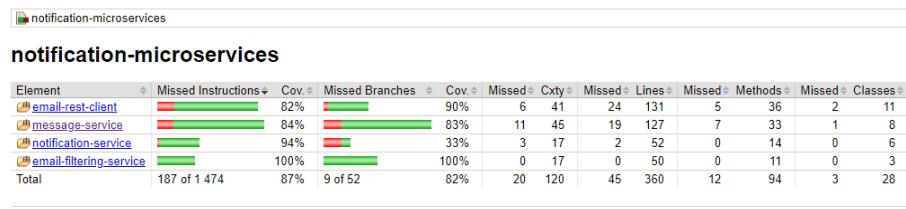
Send

Źródło: opracowanie własne

3.3 Testowanie

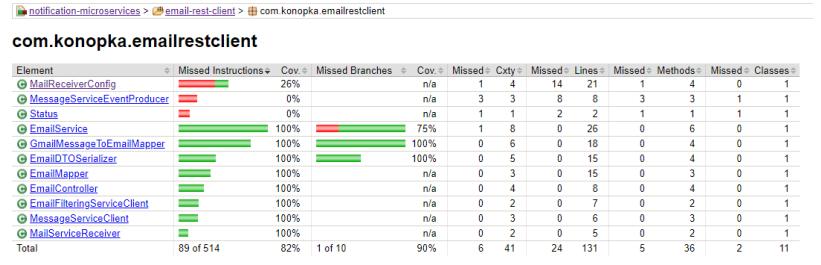
Testowanie aplikacji przeprowadzono w sposób automatyczny oraz manualny. Proces automatyczny obejmował testowanie jednostkowe w którym to skupiono się na badaniu wyłącznie jednej funkcjonalności systemu oraz testy integracyjne w których to zbadano zachowanie poszczególnych warstw aplikacji oraz współpracę między nimi. Do przeprowadzenia testów integracyjnych niezbędne było skorzystanie z bazy danych przechowywanej w pamięci podręcznej oraz stworzenie wydmuszek odpowiednich zewnętrznych mikroserwisów (ang. mock) wraz z oczekiwana odpowiedzią HTTP. Raport z testów został wykonany przy użyciu biblioteki Jacoco.

Rysunek 59. Raport z testów (widok ogólny)



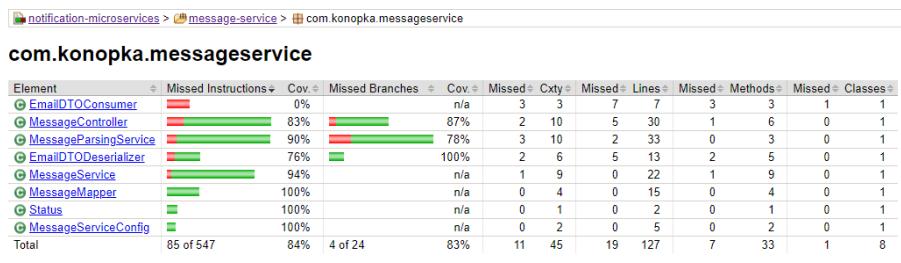
Źródło: opracowanie własne

Rysunek 60. Raport z testów (email-rest-client)



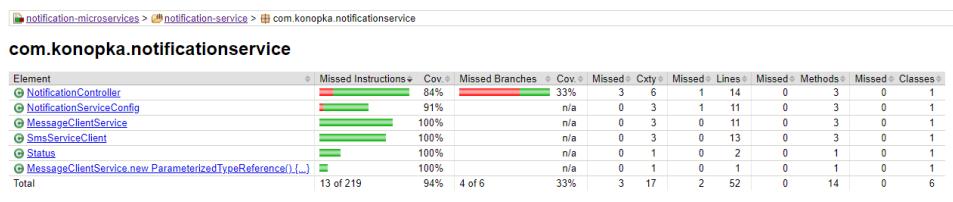
Źródło: opracowanie własne

Rysunek 61. Raport z testów (message-service)



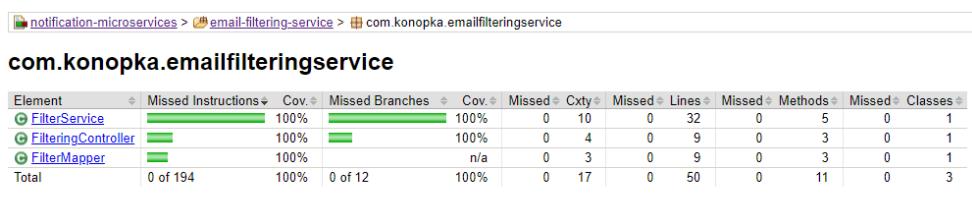
Źródło: opracowanie własne

Rysunek 62. Raport z testów (notification-service)



Źródło: opracowanie własne

Rysunek 63. Raport z testów (email-filtering-service)



Źródło: opracowanie własne

Z testowania wyłączone zostały klasy narzędziowe oraz wygenerowane przy użyciu Lomboka. Ogólne pokrycie testami całości aplikacji wynosi 87%, co jest wynikiem zadowalającym.

Testowanie manualne zostało przeprowadzone przy użyciu aplikacji Postman. Plik z importem kolekcji znajduje się w plikach źródłowych projektu.

Rysunek 64. Widok kolekcji w Postman

The screenshot shows the Postman interface with the following details:

- Left Sidebar (Collection Tree):** Shows the structure of the collection:
 - NOTIFICATION-MICROSERVICES
 - email-filtering-service
 - PUT Check if email fulfill filters req...
 - email-rest-client
 - GET Get Emails
 - message-service
 - GET Get messages
 - GET Get NOT SENT messages
 - GET Get SENT messages
 - POST Save message** (highlighted)
 - POST Update message
 - notification-service
 - GET Send SMS notification
- Top Bar:** New, Import, Overview, POST Save message, +, ...
- Request Details:** POST http://localhost:8080/msg/api/message
- Headers:** Headers (9)
- Body:** (highlighted) JSON
- Body Content:**

```
1  {
2   "from": "example@example.com",
3   "subject": "Example subject",
4   "body": "Body",
5   "date": "Mon, 12 Sep 2022 19:29:39 +0200",
6   "messageId": "18332c00ef8c07fb"
7 }
```
- Other Options:** Params, Authorization, Tests, Settings

Źródło: opracowanie własne

3.4 Wdrażanie i monitorowanie aplikacji

Proces wdrażania i monitorowania aplikacji jest bardzo istotny z punktu widzenia administracji. Jeżeli jest on skomplikowany i co więcej zależny od platformy, na której jest wdrażany wówczas zachodzi ryzyko, że aplikacja będzie zachowywać się w sposób inny niż oczekiwany lub nie będzie możliwa do wdrożenia. Monitorowanie natomiast pełni istotną funkcję diagnostyczną. Łatwość monitorowania aplikacji pozwala na szybsze zauważenie błędów w działaniu, wychwycenie przypadków brzegowych nie pokrytych testami, które sprawiają, że program zachowuje się inaczej niż powinien. Co więcej pozwala na dostrzeżenie słabych punktów w aplikacji (np. konieczność przeskakowania aplikacji, gdyż odpowiedź na zapytanie HTTP jest zbyt długa). W tym celu wykorzystano szereg narzędzi usprawniających ten proces:

- aplikacja uruchamia się w środowisku skonteneryzowanym co sprawia, że nie jest zależna od platformy, na której jest uruchamiana, ponieważ, kontener posiada swój własny obraz systemu operacyjnego,
- wykorzystano szereg narzędzi do monitorowania aplikacji takich jak:
 - stos ELK (ang. Elasticsearch, Logstash, Kibana) do agregacji i prezentacji logów

- Prometheus i Grafana do zbierania istotnych metryk dla serwisu i ich prezentacji (ruch sieciowy, obciążenie bazy danych, czas odpowiedzi na zapytania HTTP itp).

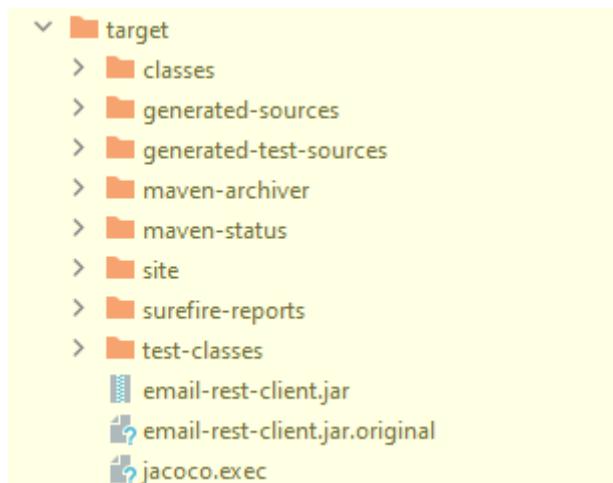
3.4.1 Konfiguracja uruchamiania aplikacji w środowisku konteneryzowanym

Proces uruchamiania aplikacji w środowisku skonteneryzowanym należałoby zacząć od zbudowania wszystkich mikroserwisów backendowych. Aby było to możliwe konieczne jest posiadanie zainstalowanych:

- Javy w wersji 17,
- Mavena w wersji 3.6.3 i wyższych.

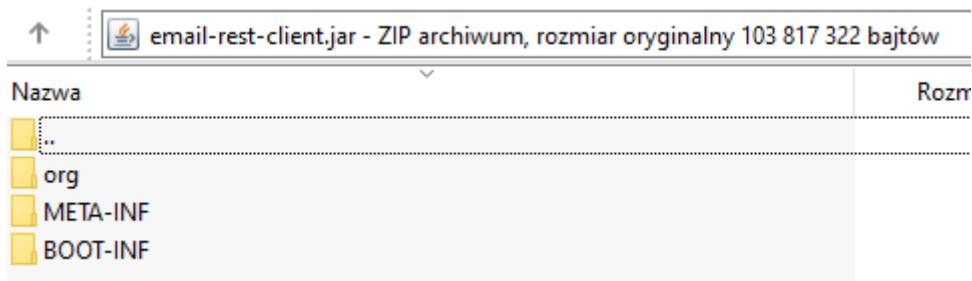
Po zbudowaniu każdego z mikroserwisów napisanych w Javie utworzony zostanie w katalogu */target* plik *jar* (ang. Java Archive) w którym znajdują się wszystkie zależności do projektu a także skompilowane do bytecode'u pliki źródłowe. Do zbudowania projektów można użyć polecenia: *mvn clean verify*.

Rysunek 65. Katalog target wraz z plikami źródłowymi



Źródło: opracowanie własne

Rysunek 66. Przykładowy plik JAR



Źródło: opracowanie własne

Każdy z mikroserwisów do wdrożenia aplikacji w środowisku skonteneryzowanym składa się z plików, które to umożliwiają. Pierwszym z nich jest plik *application.yml* i jego wariacje – *application-dev.yml* i *application-docker.yml*. W pliku tym skonfigurowane są między innymi takie parametry jak:

- port na jakim na zostać wystawiona aplikacja,
- rodzaj bazy danych,
- wczytywanie zmiennych środowiskowych z systemu operacyjnego.

Oprócz tego istotnym z punktu widzenia bezpieczeństwa jest plik *env* w którym to znajdują się definicje wszystkich zmiennych środowiskowych niezbędnych do działania aplikacji i uruchomienia jej kontekstu. Taka konfiguracja pozwala na to, aby zmienne te nie były zdefiniowane w kodzie źródłowym co zwiększa bezpieczeństwo aplikacji, ponieważ bez tego pliku nie jest możliwe jej uruchomienie w kontenerze dockerowym. Jeżeli więc mikroserwis korzysta ze zmiennych środowiskowych niezbędne jest ich dodanie w pliku konfiguracyjnym bądź w inny sposób (np. w pliku Dockerfile – plik ten spełnia formę instrukcji jak utworzyć obraz systemu/aplikacji).

Rysunek 67. Przykładowa zawartość pliku env

```
GMAIL_REFRESH_TOKEN="SAMPLE-GMAIL-REFRESH-TOKEN"  
POSTGRES_URL="jdbc:postgresql://db-email-rest-client:5432/postgres"  
POSTGRES_USERNAME="postgres"  
POSTGRES_PASSWORD="postgres"
```

Źródło: opracowanie własne

Plik Dockerfile jak wspominano powyżej pełni funkcję instrukcji i również jest niezbędny dla utworzenia obrazu systemu, z którego to później będzie możliwe utworzenie kontenera aplikacji.

Rysunek 68. Zawartość pliku Dockerfile

```
FROM openjdk:17-alpine  
  
COPY ./target/email-rest-client.jar /opt/email-rest-client/email-rest-client.jar  
  
WORKDIR /opt/email-rest-client  
  
EXPOSE 8081  
  
ENTRYPOINT ["java", "-jar", "email-rest-client.jar"]
```

Źródło: opracowanie własne

Jak pokazano na rysunku powyżej plik ten to nic innego jak zestaw instrukcji, który na pokazanym przykładzie można zinterpretować następująco:

- krok 1 – skorzystaj z istniejącego obrazu systemu UNIX z wersją Javy 17,
- krok 2 – skopiuj zawartość projektu do innego katalogu w utworzonym obrazie,
- krok 3 – przejdź do folderu, w którym znajduje się plik *jar*,
- krok 4 – wyeksponuj port 8081,
- krok 5 – w terminalu wykonaj polecenie *java -jar email-rest-client.jar*.

W przypadku klienta webowego napisanego w ReactJS proces jest nieznacznie prostszy. Tutaj również jak w przypadku serwisów backendowych niezbędny jest plik wdrożenia Dockerfile. Nie ma natomiast potrzeby tworzenia skompilowanego projektu – dzieje się to wewnątrz kontenera. Różnica ta wynika jedynie z tego, że na potrzeby rozwoju i testowania aplikacji wygodniejsze było budowanie projektu lokalnie. Nic nie stoi natomiast na przeszkodzie by proces ten wyglądał podobnie.

Rysunek 69. Zawartość pliku Dockerfile dla klienta webowego

```
FROM node:16-alpine as builder
WORKDIR /app
COPY package.json .
COPY package-lock.json .
RUN npm install && npm install axios
COPY . .
CMD ["npm", "start"]
```

Źródło: opracowanie własne

Oczywiście samo stworzenie obrazów aplikacji nie wystarcza. Konieczne jest również ich uruchomienie we wspólnym ekosystemie. Taką możliwość zapewnia *plik docker-compose.yml*, który jest spojwem wszystkich utworzonych obrazów. Zasadniczo jego rola sprowadza się do:

- zdefiniowania wspólnej sieci,
- zdefiniowanie wolumenów,
- wskazania kolejności budowania aplikacji,
- wskazania plików konfiguracyjnych.

Aby uruchomić cały ekosystem aplikacji wystarczy wykonać polecenie w terminalu: *docker compose up*. Operacja ta pozwoli na zbudowanie obrazów mikroserwisów poprzez instrukcje zawarte w pliku Dockerfile, a jeżeli korzysta się z zewnętrznych obrazów to zostaną one pobrane. Gdy wszystkie niezbędne obrazy będą utworzone, nastąpi uruchomienie kontenerów i proces można uznać za zakończony.

Rysunek 70. Przykładowy wycinek pliku docker-compose.yml

```
version: '3.7'
services:
  notification-eureka-server:
    container_name: notification-eureka-server
    build:
      context: ./notification-eureka-server
      dockerfile: Dockerfile
    networks:
      - notification-microservices
    ports:
      - "8761:8761"

  email-filtering-service:
    container_name: email-filtering-service
    build:
      context: ./email-filtering-service
      dockerfile: Dockerfile
    networks:
      - notification-microservices
    hostname: email-filtering-service
    env_file:
      - ./email-filtering-service/.env
    environment:
      - eureka.client.service-url.defaultZone=http://notification-eureka-server:8761/eureka/
      - spring.profiles.active=docker
    labels:
      collect_logs_with_filebeat: "true"
      decode_log_event_to_json_object: "true"
    ports:
      - "8082:8082"
    depends_on:
      - notification-eureka-server
      - db-email-filtering-service
    links:
      - db-email-filtering-service

  email-rest-client:
    container_name: email-rest-client
    build:
```

Źródło: opracowanie własne

3.4.2 Implementacja narzędzi do monitorowania stanu aplikacji

W przypadku narzędzi do monitorowania stanu aplikacji w środowisku skonteneryzowanym wykorzystano już dostępne w sieci obrazy, które po odpowiednim skonfigurowaniu w pliku *docker-compose.yml* pozwalają na komunikację między serwisami. Niezbędne oczywiście było utworzenie m.in. metryk, które chciałoby się mierzyć, a także logów w każdej z aplikacji. Użyte narzędzia można podzielić na dwie grupy:

- narzędzia do agregacji logów (Elasticsearch, Filebeat, Logstash i Kibana),

- narzędzia do agregacji metryk (Prometheus i Grafana).

Elasticsearch, Filebeat, Logstash są elementami odpowiedzialnymi za agregację logów, natomiast Kibana za ich prezentację w przyjaznej dla użytkownika formie.

Rysunek 71. Część pliku docker-compose.yml odpowiedzialna za agregację logów

```

elasticsearch:
  image: docker.elastic.co/elasticsearch/elasticsearch:7.2.0
  container_name: elasticsearch
  ports:
    - "9200:9200"
  environment:
    - discovery.type=single-node
    - xpack.security.enabled=false
  networks:
    - notification-microservices
  volumes:
    - ./elasticsearch/data:/var/lib/elasticsearch/data:rw

logstash:
  image: docker.elastic.co/logstash/logstash:7.2.0
  container_name: logstash
  ports:
    - "25826:25826"
    - "5044:5044"
  networks:
    - notification-microservices
  volumes:
    - ./logstash/pipeline:/usr/share/logstash/pipeline:ro
  restart: on-failure
  depends_on:
    - elasticsearch

filebeat:
  image: docker.elastic.co/beats/filebeat:7.2.0
  container_name: filebeat
  environment:
    - strict.perms=false
  networks:
    - notification-microservices
  volumes:
    - ./filebeat/filebeat.yml:/usr/share/filebeat/filebeat.yml:ro
    - /var/lib/docker/containers:/var/lib/docker/containers:ro
    - /var/run/docker.sock:/var/run/docker.sock:ro
    - ./filebeat:/usr/share/filebeat/data:rw
  user: root
  restart: on-failure
  depends_on:
    - logstash

```

Źródło: opracowanie własne

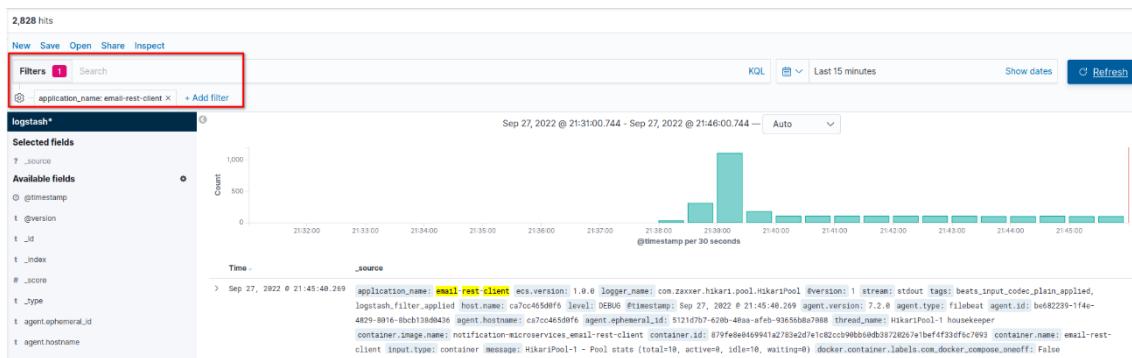
Rysunek 72. Część pliku docker-compose.yml odp. za prezentację logów

```
kibana:
  image: docker.elastic.co/kibana/kibana:7.2.0
  container_name: kibana
  environment:
    - ELASTICSEARCH_HOSTS=http://elasticsearch:9200
  networks:
    - notification-microservices
  ports:
    - "5601:5601"
  restart: on-failure
  depends_on:
    - elasticsearch
```

Źródło: opracowanie własne

Po odpowiednim skonfigurowaniu Kibany w skład, którego wchodzi zdefiniowanie z jakiego agregatora aplikacja ta ma korzystać oraz na podstawie jakiego filtra ma agregować dane – w przypadku stworzonej aplikacji są to odpowiednio Logstash i znacznik czasu (ang. timestamp) uzyska się dostęp do logów aplikacji, co pozwoli na przejrzyste monitorowanie aplikacji.

Rysunek 73. Filtrowanie logów w Kibanie (na przykładzie email-rest-client)



Źródło: opracowanie własne

W przypadku drugiej grupy – Prometheus pełni funkcję agregatora metryk, Grafana natomiast odpowiada za prezentację tych danych. Tutaj podobnie jak w przypadku narzędzi do agregacji logów skorzystano z gotowych obrazów. Dodatkowo dla uproszczenia procesu wdrażania aplikacji przygotowano plik *notification-microservices.json*, który to zawiera wszystkie niezbędne informacje dotyczące panelu obserwatora.

Rysunek 74. Część pliku docker-compose.yml odp. za agregację i prezentowanie metryk

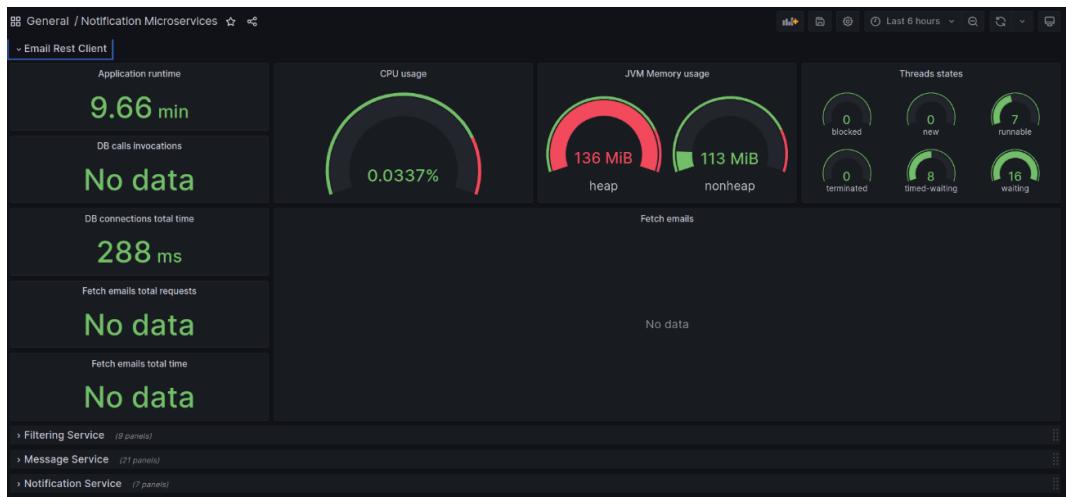
```
prometheus:
  image: prom/prometheus
  container_name: prometheus
  volumes:
    - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
  ports:
    - "9090:9090"
  networks:
    - notification-microservices

grafana:
  image: grafana/grafana
  container_name: grafana
  environment:
    - GF_SECURITY_ADMIN_PASSWORD=root
    - GF_USERS_ALLOW_SIGN_UP=false
  ports:
    - "4000:3000"
  networks:
    - notification-microservices
  depends_on:
    - prometheus
  volumes:
    - ./grafana/provisioning/datasources:/etc/grafana/provisioning/datasources
    - ./grafana/provisioning/dashboards:/etc/grafana/provisioning/dashboards
```

Źródło: opracowanie własne

Po poprawnym zimportowaniu panelu obserwatorskiego i zalogowaniu się do Grafana możliwa jest obserwacja w czasie rzeczywistym stanu aplikacji.

Rysunek 75. Fragment panelu obserwatora w Grafanie



Źródło: opracowanie własne

Zakończenie

Mikroserwisy jako aplikacje klasy enterprise na stałe wpisały się w środowisko twórców oprogramowania. Pozwalają one zespołom developerskim na szybkie dostarczenie MVP (ang. Minimum Valuable Product) i ciągłe rozwijanie aplikacji, przyśpieszając tym samym moment wydania produktu na produkcję. Nowoczesne technologie takie jak Apache Kafka pozwalają na komunikację asynchroniczną, redukując tym samym ryzyko braku responsywności wskutek chwilowej awarii sieci.

Narzędzia takie jak Docker do konteneryzacji wraz z stosem technologicznym do monitorowania pozwalają na szybsze wdrożenie, a także dają niezbędny podgląd na to jak aplikacja się zachowuje – jest to obarczone dużym, dodatkowym nakładem pracy, który sprawia, że współcześnie proces wytwórczy nie ogranicza się tylko do developerów i testerów oprogramowania, ale wymaga holistycznego spojrzenia na cały ekosystem.

Aplikacja zgodnie z założeniem pozwoliła na wysyłanie powiadomień SMS dotyczących ważnych wydarzeń – takich jak opłacenie faktury za energię elektryczną czy terminie płatności za ubezpieczenie. Proces implementacji pozwolił na zapoznanie się z dużą ilością nowoczesnych technologii używanych w przedsiębiorstwach zajmujących się wytwarzaniem oprogramowania. W tabeli umieszczonej poniżej znajduje się zestawienie wszystkich wymagań funkcjonalnych i niefunkcjonalnych wraz z określeniem czy dane kryterium zostało spełnione.

Tabela 1. Mapa wymagań funkcjonalnych i niefunkcjonalnych

Nr wymagania	Czy spełnione?	Opis
OF-1	TAK	aplikacja działa z pocztą w domenie gmail.com
OF-2	TAK	odczytywanie wszystkich wiadomości z poczty i zapis w bazie danych
OF-3	TAK	filtrowanie odczytywanych wiadomości z poczty i na podstawie spełnionych kryteriów tworzenie wzoru wiadomości SMS i zapis w bazie danych
OF-4	TAK	wysyłanie wiadomości SMS przy użyciu serwisu twilio.com dostarczającego usługi VOIP na 2 dni przed terminem określonym w wygenerowanej wiadomości
OF-5	TAK	każdorazowa aktualizacja stanu wysłanej wiadomości

OF-6	TAK	aplikacja po wdrożeniu nie wymaga ingerencji użytkownika końcowego
OF-7	TAK	monitorowanie stanu aplikacji poprzez agregację logów
FF-1	TAK	klient webowy pozwalający na przeglądanie pobranych wiadomości email
FF-2	TAK	klient webowy pozwalający na przeglądanie statusu stworzonych powiadomień SMS
FF-3	TAK	klient webowy pozwalający na ręczne dodawanie wiadomości do wysłania w formie SMS
FF-4	NIE	klient webowy pozwalający na ręczne wywołanie akcji wysyłania powiadomień dla niewysłanych wiadomości
FF-5	TAK	monitorowanie stanu aplikacji przy pomocy narzędzia do wizualizacji zdarzeń (stan uruchomienia aplikacji i inne parametry)
ONF-1	TAK	skalowalność aplikacji
ONF-2	TAK	dostępność serwisu tylko wewnętrznej sieci
FNF-1	TAK	przenośność aplikacji – powinna być możliwość uruchomienia na dowolnej platformie
FNF-2	TAK	niski koszt sprzętowy
FNF-3	NIE	możliwość skorzystania z usług chmurowych do wdrożenia aplikacji na produkcję

Źródło: opracowanie własne

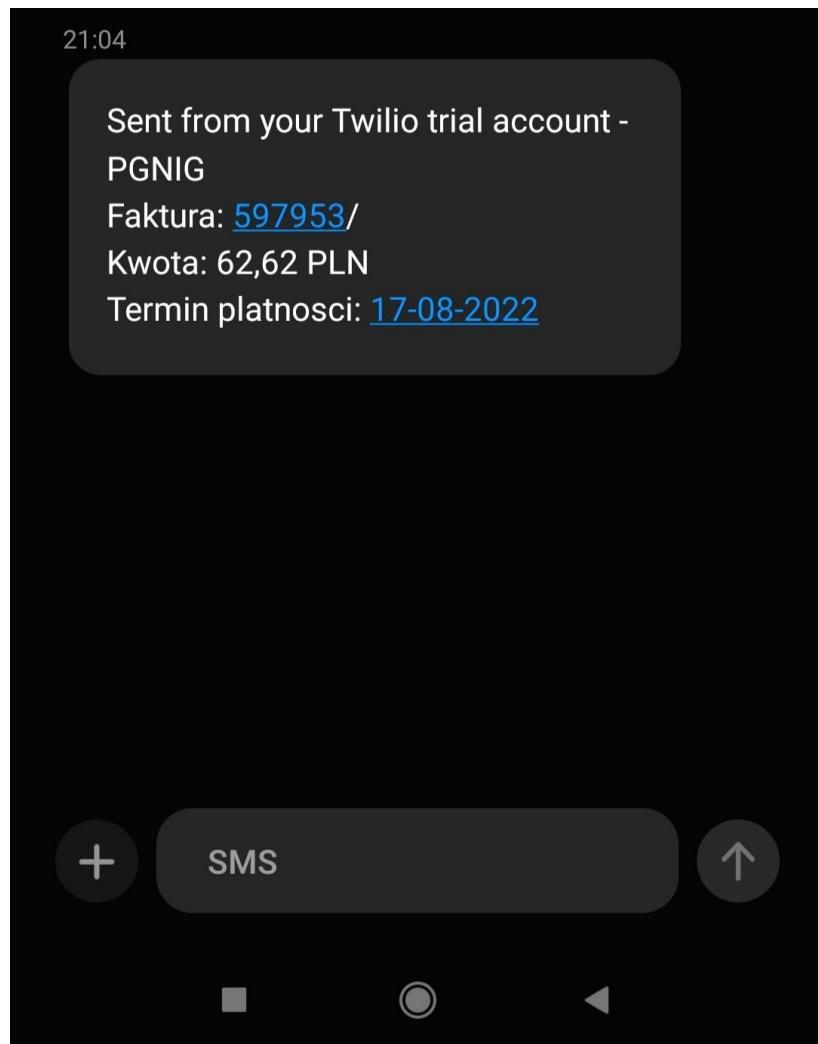
Z tabeli wynika, że większość założeń funkcjonalnych i niefunkcjonalnych zarówno obligatoryjnych jak i fakultatywnych została zrealizowana. Wyjątkiem jest brak spełnienia wymagań FNF-3 oraz FF-4. Pierwsze z nich wymagałoby zastosowania narzędzi do wdrożeń na produkcję takich jak np. Kubernetes. Drugie natomiast nie zostało spełnione, ponieważ aplikacja domyślnie ma działać bezobsługowo i klient webowy jest dodatkiem.

Rozwój w kierunku rozbudowy aplikacji mógłby obejmować takie zagadnienia jak:

- całkowita rezygnacja z komunikacji synchronicznej na rzecz Apache Kafka,

- wdrożenie aplikacji na usługi chmurowe,
- stworzenie nowego mikroserwisu odpowiedzialnego za autoryzacje zapytań,
- możliwość dodawania filtrowania z poziomu klienta webowego,
- możliwość parametryzacji wysyłania powiadomień SMS.

Rysunek 76. Wysłane powiadomienie SMS o płatności za gaz



Źródło: opracowanie własne

Spis bibliografii

Książki i artykuły z czasopism:

1. Kent Beck, *Sztuka tworzenia dobrego kodu*, Wydawnictwo Helion, Gliwice 2020
2. Robert C. Martin, *Mistrz czystego kodu*, Wydawnictwo Helion, Gliwice 2017
3. Robert C. Martin, *Czysty kod – Podręcznik dobrego programisty*, Wydawnictwo Helion, Gliwice 2014
4. Joshua Bloch, *Java Efektywne Programowanie*, Helion, Gliwice 2018
5. Eric Freeman, Elisabeth Robosn, *Wzorce projektowe*, Wydawnictwo Helion, Gliwice 2022
6. Craig Walls, *Spring w akcji, wydanie IV*, Helion, Gliwice 2015
7. Cay S. Horstmann, *Java Podstawy, wydanie X*, Wydawnictwo Helion, Gliwice 2016
8. Cay S. Horstmann, *Java Techniki zaawansowane, wydanie X*, Wydawnictwo Helion, Gliwice 2017
9. Eric Ewans, *Domain Driven Design*, Addison Wesley 2003
10. Roy Thomas Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, University of California 2000
13. Konrad Gos, Wojciech Zabierowski, *The Comparison of Microservice and Monolithic Architecture*, Łódź University of Technology 2020

Źródła elektroniczne:

14. Java Documentation, <https://docs.oracle.com/en/java/javase/17/docs/api/index.html> [dostęp 27.11.2023]
15. IntelliJ Idea Documentation, <https://www.jetbrains.com/help/idea/getting-started.html> [dostęp 27.11.2023]
16. Maven Documentation, <https://maven.apache.org/guides/index.html> [dostęp 27.11.2023]
17. Spring Framework Core,
<https://docs.spring.io/springframework/docs/current/reference/html/core.html#spring-core>, [dostęp 27.11.2023]
18. Docker Documentation, <https://docs.docker.com/> [dostęp 27.11.2023]
19. Apache Kafka Documentation, <https://kafka.apache.org/documentation/> [dostęp 27.11.2023]
20. Prometheus Documentation, <https://prometheus.io/docs/introduction/overview/> [dostęp 27.11.2023]
21. Grafana Documentation, <https://grafana.com/docs/grafana/latest/> [dostęp 27.11.2023]
22. ELK Stack Documentation, <https://www.elastic.co/guide/index.html> [dostęp 27.11.2023]

23. Martin Fowler, Richardson Maturity Model,
<https://martinfowler.com/articles/richardsonMaturityModel.html> [dostęp 27.11.2023]
24. Martin Fowler, Microservices, *<https://martinfowler.com/articles/microservices.html>*
[dostęp 27.11.2023]

Spis rysunków i tabel

Rysunek 1. Zainteresowanie frazą "Microservices" na przestrzeni lat	9
Rysunek 2. Ideowe przedstawienie monolitu.....	11
Rysunek 3. Ideowe przedstawienie mikroserwisów jako sieć małych aplikacji.....	12
Rysunek 4. Architektura klient - serwer.....	14
Rysunek 5. Architektura klient-bezstanowy serwer.....	15
Rysunek 6. Klient-Cache-Stateless-Server.....	15
Rysunek 7. Uniform-Klient-Cache-Stateless-Server	16
Rysunek 8. Charakterystyka metod HTTP.....	17
Rysunek 9. Przykład żądania HTTP w przypadku poziomu 0.....	18
Rysunek 10. Przykład odpowiedzi HTTP w przypadku poziomu 0	18
Rysunek 11. Przykład zapytania HTTP o zasób w przypadku poziomu 1.....	19
Rysunek 12. Przykład odpowiedzi HTTP o zasób w przypadku poziomu 1	19
Rysunek 13. Przykład zapytania HTTP o zasób w przypadku poziomu 2.....	20
Rysunek 14. Przykład odpowiedzi HTTP o utworzenie zasobu w przypadku poziomu 2	20
Rysunek 15. Przykład zapytania HTTP o zasób w przypadku poziomu 3.....	21
Rysunek 16. Przykład odpowiedzi HTTP o utworzenie zasobu w przypadku poziomu 3	21
Rysunek 17. Budowa Spring Framework.....	24
Rysunek 18. Widok ogólny środowiska developerskiego.....	27
Rysunek 19. Proces wysyłania eventu	28
Rysunek 20. Przykładowy zdeserializowany obiekt klasy MessageDTO	29
Rysunek 21. Przykładowy plik POM.xml.....	30
Rysunek 22. Proces tworzenia kontenera.....	33
Rysunek 23. Serwisy wykryte przez Prometheusa, które posiadają metryki	35
Rysunek 24. Panel podglądu aplikacji w Grafana.....	35
Rysunek 25. Klasa EmailDTO	39
Rysunek 26. Klasa FilterDTO	40
Rysunek 27. Klasa MessageDTO.....	41
Rysunek 28. Dodawanie nowego projektu.....	42
Rysunek 29. Nadawanie uprawień zewn. klientom na autoryzację przy pomocy OAuth	42
Rysunek 30. Etap pierwszy przy uzyskiwaniu refresh token, część I.....	43
Rysunek 31. Etap pierwszy przy uzyskiwaniu refresh tokena, część II.....	44
Rysunek 32. Etap drugi, nadawanie uprawień	45

Rysunek 33. Etap trzeci, nadawanie uprawień testowym użytkownikom	46
Rysunek 34. Etap czwarty, podsumowanie.....	46
Rysunek 35. Tworzenie OAuth Client ID	47
Rysunek 36. Tworzenie OAuth Client ID, część II.....	48
Rysunek 37. Wygenerowany klucz OAuth	48
Rysunek 38. Dodawanie pary klucz - wartość	49
Rysunek 39. Dodawanie usługi Gmail do uzyskania refresh tokena	50
Rysunek 40. Wygenerowany refresh i access token	51
Rysunek 41. Niezbędne zależności umożliwiające korzystanie z Gmail API	52
Rysunek 42. Konfiguracja klasy odpowiedzialnej za komunikację z Gmail API.....	53
Rysunek 43. Pierwszy etap tworzenia konta	54
Rysunek 44. Drugi etap tworzenia konta	54
Rysunek 45. Uwierzytelnianie konta przy pomocy numeru telefonu	55
Rysunek 46. Akceptacja wyg. numeru tel. do wysyłania powiadomień SMS	55
Rysunek 47. Panel użytkownika	56
Rysunek 48. Dane uwierzytelniające	56
Rysunek 49. Uproszczona architektura aplikacji	57
Rysunek 50. Diagram sekw. odczytywania wiadomości email (asynchroniczny).....	59
Rysunek 51. Diagram sekw. odczytywania wiadomości email (synchroniczny)	60
Rysunek 52. Diagram wysyłania powiadomień SMS	62
Rysunek 53. Ekran powitalny wraz z paskiem nawigacyjnym	67
Rysunek 54. Ekran wiadomości email (widok ogólny)	67
Rysunek 55. Ekran wiadomości email (widok tabeli).....	67
Rysunek 56. Ekran wiadomości SMS (widok ogólny)	68
Rysunek 57. Ekran wiadomości SMS (widok tabeli).....	68
Rysunek 58. Formularz ręcznego dodawania wiadomości SMS	68
Rysunek 59. Raport z testów (widok ogólny)	69
Rysunek 60. Raport z testów (email-rest-client)	69
Rysunek 61. Raport z testów (message-service)	69
Rysunek 62. Raport z testów (notification-service)	70
Rysunek 63. Raport z testów (email-filtering-service)	70
Rysunek 64. Widok kolekcji w Postman.....	71
Rysunek 65. Katalog target wraz z plikami źródłowymi	72

Rysunek 66. Przykładowy plik JAR.....	73
Rysunek 67. Przykładowa zawartość pliku env	74
Rysunek 68. Zawartość pliku Dockerfile	74
Rysunek 69. Zawartość pliku Dockerfile dla klienta webowego	75
Rysunek 70. Przykładowy wycinek pliku docker-compose.yml	76
Rysunek 71. Część pliku docker-compose.yml odpowiedzialna za agregację logów	77
Rysunek 72. Część pliku docker-compose.yml odp. za prezentację logów	78
Rysunek 73. Filtrowanie logów w Kibanie (na przykładzie email-rest-client).....	78
Rysunek 74. Część pliku docker-compose.yml odp. za agregację i prezentowanie metryk ...	79
Rysunek 75. Fragment panelu obserwatora w Grafanie.....	79
Rysunek 76. Wysłane powiadomienie SMS o płatności za gaz.....	82
Tabela 1. Mapa wymagań funkcjonalnych i niefunkcjonalnych.....	80

Załączniki

A – Dokumentacja email-rest-client

{...} /email/api/api-docs Explore

Fetching email service 1.0 OAS3

/email/api/api-docs

Service for fetching email from Gmail API

Grzegorz Konopka - Website
Send email to Grzegorz Konopka
MIT Licence

Servers
http://localhost:8081/email/api - Generated server url

email-controller

GET	/list-emails	^
Fetch emails from Service		
Parameters	Try it out	^
No parameters		
Responses		
Code	Description	Links
200	OK	No links

Code	Description	Links
	<p>Media type</p> <div style="border: 2px solid green; padding: 2px; display: inline-block;">application/json</div> <p>Controls Accept header.</p> <p>Example Value Schema</p> <pre>[{ "from": "example@example.com", "subject": "Example subject", "body": "Body", "date": "Mon, 12 Sep 2022 19:29:39 +0200", "messageId": "18332c00ef8c07fb" }]</pre>	
400	Bad request	<i>No links</i>
	<p>Media type</p> <div style="border: 2px solid green; padding: 2px; display: inline-block;">application/json</div> <p>Example Value</p> <pre>"Bad request"</pre>	
404	Not found	<i>No links</i>
	<p>Media type</p> <div style="border: 2px solid green; padding: 2px; display: inline-block;">application/json</div> <p>Example Value</p> <pre>"Not found"</pre>	
500	Server error	<i>No links</i>
	<p>Media type</p> <div style="border: 2px solid green; padding: 2px; display: inline-block;">application/json</div>	

Code	Description	Links
	Example Value	
	"Server error"	
GET	/emails	^
Fetch only new emails from Gmail API		
Parameters		Try it out
No parameters		
Responses		
Code	Description	Links
200	OK	No links
	Media type	
	application/json	
	Controls Accept header.	
	Example Value	Schema
	<pre>[{ "from": "example@example.com", "subject": "Example subject", "body": "Body", "date": "Mon, 12 Sep 2022 19:29:39 +0200", "messageId": "18332c00ef8c07fb" }]</pre>	
400	Bad request	No links

Code	Description	Links
	<p>Media type</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;">application/json</div> <p>Example Value</p> <pre>"Bad request"</pre>	
404	<p>Not found</p> <p>Media type</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;">application/json</div> <p>Example Value</p> <pre>"Not found"</pre>	<i>No links</i>
500	<p>Server error</p> <p>Media type</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;">application/json</div> <p>Example Value</p> <pre>"Server error"</pre>	<i>No links</i>

Schemas	^

```
EmailDTO {
    from*           string
                    example: example@example.com
                    From who is an email

    subject*        string
                    example: Example subject
                    Subject

    body*           string
                    example: Body
                    Body

    date*           string
                    example: Mon, 12 Sep 2022 19:29:39 +0200
                    Sent date

    messageId*      string
                    example: 18332c00ef8c07fb
                    UUID of received email

}
```

B – Dokumentacja email-filtering-service

The screenshot shows the API documentation for the Email filtering service. At the top, there is a navigation bar with a green button containing three dots, a path field showing '/filtering/api/api-docs', and a 'Explore' button. Below the navigation bar, the title 'Email filtering service' is displayed, along with its version '1.0' and specification 'OAS3'. A sub-path '/filtering/api/api-docs' is also shown. The main content area contains a brief description: 'Service for filtering email based on regex pattern', the author's name 'Grzegorz Konopka - Website', the contact information 'Send email to Grzegorz Konopka', and the license 'MIT Licence'. In the 'Servers' section, the URL 'http://localhost:8082/filtering/api - Generated server url' is listed. The main endpoint 'filtering-controller' is detailed with a 'PUT /filter' method. It describes the operation as 'Checks if given email match the regex pattern'. The 'Parameters' section indicates 'No parameters'. The 'Request body' is marked as 'required' and has a schema type of 'application/json'. An example value for the request body is provided:

```
{  
  "from": "example@example.com",  
  "subject": "Example subject",  
  "body": "Body",  
  "date": "Mon, 12 Sep 2022 19:29:39 +0200",  
}
```

```
        "messageId": "18332c00ef8c07fb"  
    }  
  
}
```

Responses

Code	Description	Links
200	OK	<i>No links</i>
	<p>Media type</p> <p>application/json</p> <p>Controls Accept header.</p> <p>Example Value Schema</p> <pre>true</pre>	
400	Bad request	<i>No links</i>
	<p>Media type</p> <p>application/json</p> <p>Example Value</p> <pre>"Bad request"</pre>	
404	Not found	<i>No links</i>
	<p>Media type</p> <p>application/json</p> <p>Example Value</p> <pre>"Not found"</pre>	

Code	Description	Links
500	<p>Server error</p> <p>Media type</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;">application/json</div> <p>Example Value</p> <pre>"Server error"</pre>	No links

Schemas		^
EmailDTO	<pre>from* string example: example@example.com From who is an email subject* string example: Example subject Subject body* string example: Body Body date* string example: Mon, 12 Sep 2022 19:29:39 +0200 Sent date messageId* string example: 18332c00ef8c07fb UUID of received email }</pre>	

C – Dokumentacja message-service

The screenshot shows the OpenAPI documentation for the Message service. At the top, there are three buttons: a green one with three dots, a blue one labeled '/msg/api/api-docs', and a grey one labeled 'Explore'. Below this, the title 'Message service' is displayed with a version of '1.0' and a 'OAS3' badge. A sub-path '/msg/api/api-docs' is also shown. The main content area contains a brief description: 'Service for creating message from received email', the author's name 'Grzegorz Konopka - Website', the contact information 'Send email to Grzegorz Konopka', and the license information 'MIT Licence'. Under the 'Servers' section, the URL 'http://localhost:8083/msg/api - Generated server url' is listed. The main endpoint is 'message-controller' with a 'PUT /message' method. The description for this method is 'Update message'. It has no parameters. The request body is required and is of type 'application/json'. An example value for the schema is provided:

```
{  
  "body": "Body",  
  "emailUuid": "18332c00ef8c07fb",  
  "date": "NOT_SENT"  
}
```

Responses

Code	Description	Links
200	OK	<i>No links</i>
	<p>Media type</p> <div style="border: 1px solid green; padding: 2px; display: inline-block;">application/json</div> <p>Controls Accept header.</p> <p>Example Value Schema</p> <pre>{ "body": "Body", "emailUuid": "18332c00ef8c07fb", "date": "NOT_SENT" }</pre>	
400	Bad request	<i>No links</i>
	<p>Media type</p> <div style="border: 1px solid green; padding: 2px; display: inline-block;">application/json</div> <p>Example Value</p> <pre>"Bad request"</pre>	
404	Not found	<i>No links</i>
	<p>Media type</p> <div style="border: 1px solid green; padding: 2px; display: inline-block;">application/json</div> <p>Example Value</p> <pre>"Not found"</pre>	

Code	Description	Links
500	<p>Server error</p> <p>Media type</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;">application/json</div> <p>Example Value</p> <pre>"Server error"</pre>	No links

POST <code>/message</code>	^
Create message	
Parameters	Try it out
No parameters	
Request body <small>required</small>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">application/json</div>
Example Value <small>Schema</small>	
<pre>{ "from": "example@example.com", "subject": "Example subject", "body": "Body", "date": "Mon, 12 Sep 2022 19:29:39 +0200", "messageId": "18332c00ef8c07fb" }</pre>	
Responses	

Code	Description	Links
200	OK	<i>No links</i>
202	Accepted	<i>No links</i>
	<p>Media type</p> <div style="border: 2px solid #8B9E7D; padding: 2px; display: inline-block;">application/json</div> <p>Controls Accept header.</p>	
	<p>Example Value Schema</p> <pre>{ "body": "Body", "emailUuid": "18332c00ef8c07fb", "date": "NOT_SENT" }</pre>	
204	No content	<i>No links</i>
	<p>Media type</p> <div style="border: 2px solid #8B9E7D; padding: 2px; display: inline-block;">application/json</div> <p>Example Value</p> <pre>"No content"</pre>	
400	Bad request	<i>No links</i>
	<p>Media type</p> <div style="border: 2px solid #8B9E7D; padding: 2px; display: inline-block;">application/json</div> <p>Example Value</p> <pre>"Bad request"</pre>	
500	Server error	<i>No links</i>

Code	Description	Links
	<p>Media type</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;">application/json</div>	
	<p>Example Value</p> <pre>"Server error"</pre>	
<hr/>		
POST	/create	^
<hr/>		
Create message		
Parameters		Try it out
No parameters		
Request body <small>required</small>		<div style="border: 1px solid black; padding: 2px; display: inline-block;">application/json</div>
<hr/>		
Example Value <small>Schema</small>		
<pre>{ "body": "Body", "emailUuid": "18332c00ef8c07fb", "date": "NOT_SENT" }</pre>		
<hr/>		
Responses		
Code	Description	Links
201	Created	<i>No links</i>
204	No content	<i>No links</i>

Code	Description	Links
	<p>Media type</p> <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">/*</div> <p>Controls Accept header.</p> <p>Example Value</p> <p>No content</p>	
400	Bad request	<i>No links</i>
	<p>Media type</p> <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">/*</div> <p>Example Value</p> <p>Bad request</p>	
500	Server error	<i>No links</i>
	<p>Media type</p> <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">/*</div> <p>Example Value</p> <p>Server error</p>	
GET /messages		^
<p>Get messages</p>		
Parameters		Try it out

Name	Description	
Message status string (query)	<p><i>Available values : SENT, NOT_SENT</i></p> <p><i>Example : NOT_SENT</i></p> <div style="border: 1px solid black; padding: 2px; display: inline-block;">NOT_SENT</div>	
Responses		
Code	Description	Links
200	OK	<i>No links</i>
	<p>Media type</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;">application/json</div> <p>Controls Accept header.</p>	
	<p>Example Value</p> <pre>[{"body": "Body", "emailUuid": "18332c00ef8c07fb", "date": "NOT_SENT"}]</pre>	
400	Bad request	<i>No links</i>
	<p>Media type</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;">application/json</div> <p>Example Value</p> <pre>"Bad request"</pre>	
404	Not found	<i>No links</i>

Code	Description	Links
	<p>Media type</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;">application/json</div> <p>Example Value</p> <pre>"Not found"</pre>	
500	<p>Server error</p> <p>Media type</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;">application/json</div> <p>Example Value</p> <pre>"Server error"</pre>	No links

Schemas	^
<pre>MessageDTO { body* string example: Body Body emailUuid* string example: 18332c00ef8c07fb UUID of received email date* string example: NOT_SENT Status Enum: [SENT, NOT_SENT] }</pre>	

```
EmailDTO {
    from*           string
                    example: example@example.com
                    From who is an email

    subject*        string
                    example: Example subject
                    Subject

    body*           string
                    example: Body
                    Body

    date*           string
                    example: Mon, 12 Sep 2022 19:29:39 +0200
                    Sent date

    messageId*      string
                    example: 18332c00ef8c07fb
                    UUID of received email

}
```

D – Dokumentacja notification-service

The screenshot shows the API documentation for the notification-service. At the top, there's a navigation bar with a green button containing three dots, a path field showing '/notification/api/api-docs' with a green border, and a 'Explore' button. Below the header, the title 'Notification service' is displayed in large bold letters, followed by '1.0 OAS3'. A sub-header '/notification/api/api-docs' is shown. The main content area contains a brief description: 'Service for sending SMS notifications', author information ('Grzegorz Konopka - Website', 'Send email to Grzegorz Konopka', 'MIT Licence'), and a 'Servers' section with a generated server URL: 'http://localhost:8084/notification/api - Generated server url'. The main content area is titled 'notification-controller' and includes a 'GET /notify' endpoint. The description for this endpoint is: 'Send SMS notification for messages which meets the deadline and has NOT SENT status. Update message status after that operation'. It also includes sections for 'Parameters' (with a 'Try it out' button) and 'Responses'.

1.0 OAS3

/notification/api/api-docs

Notification service

Service for sending SMS notifications

Grzegorz Konopka - Website
Send email to Grzegorz Konopka
MIT Licence

Servers

http://localhost:8084/notification/api - Generated server url

notification-controller

GET /notify

Send SMS notification for messages which meets the deadline and has NOT SENT status. Update message status after that operation

Parameters Try it out

No parameters

Responses

Code	Description	Links
200	<p>OK</p> <p>Media type</p> <div style="border: 1px solid green; padding: 2px; display: inline-block;">application/json</div> <p>Controls Accept header.</p>	<i>No links</i>
400	<p>Bad request</p> <p>Media type</p> <div style="border: 1px solid green; padding: 2px; display: inline-block;">application/json</div> <p>Example Value</p> <p><code>"Bad request"</code></p>	<i>No links</i>
404	<p>Not found</p> <p>Media type</p> <div style="border: 1px solid green; padding: 2px; display: inline-block;">application/json</div> <p>Example Value</p> <p><code>"Not found"</code></p>	<i>No links</i>

Code	Description	Links
500	<p>Server error</p> <p>Media type</p> <p><code>application/json</code></p> <p>Example Value</p> <p><code>"Server error"</code></p>	<i>No links</i>