



# POLITECHNIKA ŁÓDZKA

Wydział Elektrotechniki, Elektroniki,  
Informatyki i Automatyki

Nowoczesne środowiska aplikacji internetowych

Lato 2025/2026

Przygotowali:

Piotr Pleska 263232

Tomasz Kubik 263216

Adam Madej 263219

## Spis treści

Spis treści .....	2
Organizacja mikroservisów i wystawianych endpointów .....	4
User-Service .....	4
Dane techniczne: .....	4
Baza danych .....	4
Endpointy .....	5
Order-Service .....	6
Dane techniczne: .....	6
Baza danych .....	7
Endpointy .....	7
Integracje z zewnętrznymi API (HERE & Google Maps) .....	9
Notification-Service .....	11
Dane techniczne: .....	12
Baza danych .....	12
Endpointy .....	13
Payment-Service .....	13
Dane techniczne: .....	14
Baza danych .....	14
Endpointy .....	14
Security .....	15
Keycloak .....	15
CORS .....	16
Konfiguracja Gatewaya i portów mikroservisów .....	17
Load balancing i Eureka .....	19
Netflix Eureka .....	19
Jaeger .....	20
Algorytmy użyte do optymalizacji tras .....	20
Timefold AI (Constraint Solver) .....	20
Algorytm Zachłanny (Greedy Strategy) .....	21
Metoda Siłowa (Brute Force z Chunkingiem) .....	22
Aplikacja Klientka (Architektura Frontendu) .....	23
Architektura Sterowana Zdarzeniami (Event-Driven Architecture) .....	24

Wzorzec API Composition (Zarządzanie Rozproszonymi Danymi).....	25
Generacja tras dla kurierów przez administratora, główny przepływ systemu BoatDelivery .	25
Cykl życia zdarzeń i komunikacja z Użytkownikiem.....	28
Scenariusz 1: Inicjacja płatności.....	28
Scenariusz 2: Asynchroniczna dystrybucja powiadomień.....	30
Scenariusz 3: Audyt i logi administracyjne .....	31
Pozyskiwanie kluczy API .....	31
SMS API .....	31
UWAGI.....	33
STRIPE .....	33
UWAGI.....	34
HERE API.....	34
Google Maps API .....	36
GMAIL SMPT .....	37
Keycloak Mail SMTP .....	37
Budowanie aplikacji.....	38

# Organizacja mikroservisów i wystawianych endpointów

## *User-Service*

Serwis odpowiedzialny za zarządzanie danymi użytkowników oraz flotą pojazdów. Serwis daje możliwość zapisu użytkownika z mapowaniem danej roli: ADMIN, CUSTOMER, COURIER oraz danych środków transportu do rozwożenia paczek. Rejestrowanie użytkowników zarządzane jest przez serwis Keycloak, jednak jest bezpośrednio synchronizowany z User-Service. Dzięki temu cała autoryzacja jest obsługiwana przez serwis Keycloak (o którym będzie mowa w późniejszej części raportu), a User-Service może obsługiwać działania związane z biznesowym zarządzaniem userów. Oprócz zapisu danych usera, występuje również zarządzanie środkami transportów przypisanych do poszczególnych kurierów (userzy z rolą „COURIER”) poprzez ich tworzenie, edycję, usuwanie przez rolę ADMINa.

Ponadto, User-Service pełni rolę wewnętrznego dostawcy informacji dla innych mikroservisów. Wystawia on wewnętrzne interfejsy API, dzięki którym np. Order-Service (odpowiedzialny za zamówienia) może pobrać listę dostępnych kurierów oraz przypisanych do nich pojazdów, co jest niezbędne do prawidłowego działania algorytmów optymalizujących trasy dostaw. Integracja na linii Keycloak/Order-Service => User-Service realizowana jest za pomocą wewnętrznych webhooków, co gwarantuje spójność i bezpieczeństwo danych.

### **Dane techniczne:**

Framework: Spring Boot (Java)

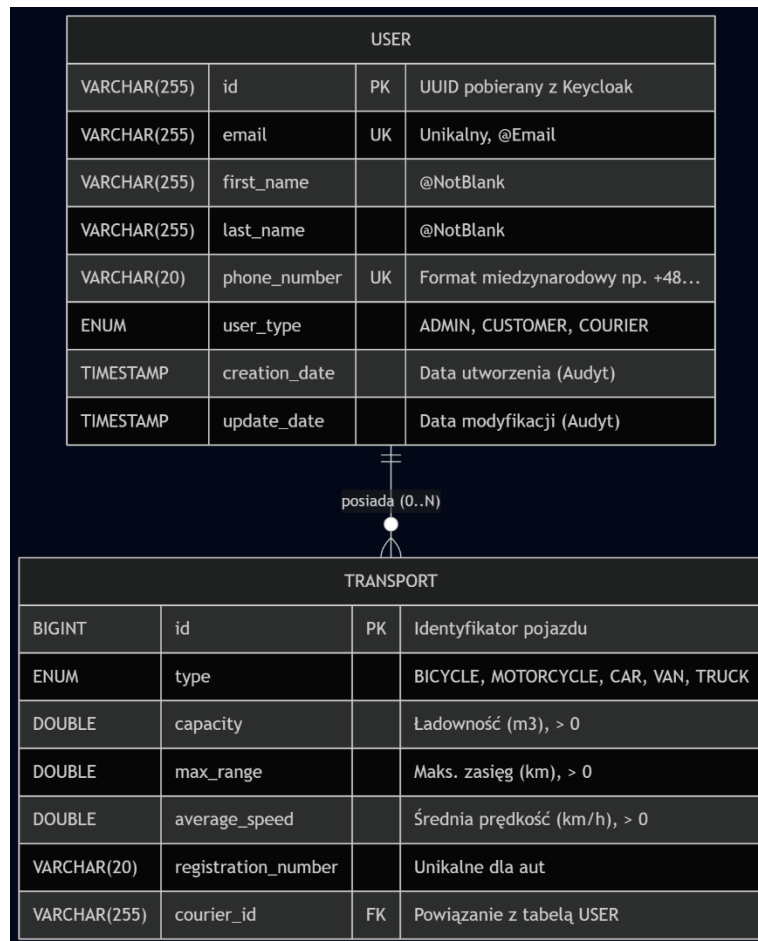
Port: 8081

Baza danych: PostgreSQL (migracje schematów zarządzane przez narzędzie Flyway)

**Zabezpieczenia:** Spring Security (OAuth2 Resource Server z weryfikacją JWT) oraz mechanizm *Pre-Shared Key* dla komunikacji wewnętrznej.

### **Baza danych**

Baza danych user\_service\_db dzieli się na dwie główne encje biznesowe, mapowane za pomocą JPA (Hibernate). Dodatkowo modele dziedziczą z klasy ControlledEntity, która automatycznie zarządza polami audytowymi (creationDate, updateDate, version).



## Endpointy

Kontrolery REST zostały podzielone ze względu na domenę oraz uprawnienia dostępowe.

### Zarządzanie Użytkownikami

METODA	ENDPOINT	UPRAWNIENIA	OPIS AKCJI
GET	/api/user/me	PreAuth(ALL)	Zwraca profil aktualnie zalogowanego użytkownika.
PATCH	/api/user/me	PreAuth(ALL)	Pozwala na częściową aktualizację własnego profilu.
GET	/api/user/my-role	PreAuth(ALL)	Zwraca samą rolę usera.
GET	/api/user	PreAuth(ADMIN)	Pobiera pełną listę wszystkich użytkowników.
GET	/api/user/paginated	PreAuth(ADMIN)	Pobiera listę użytkowników z paginacją.
GET	/api/user/stats/count-by-type	PreAuth(ADMIN)	Zwraca statystyki liczby użytkowników pogrupowane po ich roli.
PUT/PATCH	/api/user/{id}	PreAuth(ADMIN)	Administracyjna (pełna lub częściowa) aktualizacja profilu.
DELETE	/api/user/{id}	PreAuth(ADMIN)	Usunięcie użytkownika z bazy lokalnej i synchronizacja usunięcia do Keycloak.

### Zarządzanie Flotą

METODA	ENDPOINT	WYMAGANE UPRAWNIENIA	OPIS AKCJI
GET	/api/transport	PreAuth(ADMIN)	Zwraca listę wszystkich pojazdów.
GET	/api/transport/{id}	PreAuth(ADMIN, COURIER)	Zwraca szczegóły konkretnego pojazdu – kurier zobaczy tylko swój przypisany pojazd
POST	/api/transport	PreAuth(ADMIN)	Dodanie nowego pojazdu do floty.
PUT	/api/transport/{id}	PreAuth(ADMIN)	Edycja parametrów pojazdu.
DELETE	/api/transport/{id}	PreAuth(ADMIN)	Usunięcie pojazdu z floty.
POST	/api/transport/{id}/assign/{courierId}	PreAuth(ADMIN)	Przypisanie wybranego pojazdu do konkretnego kuriera.
POST	/api/transport/{id}/unassign	PreAuth(ADMIN)	Odebranie pojazdu kurierowi.

### Komunikacja Wewnętrzna (Backend-to-Backend & Webhooki)

METODA	ENDPOINT	ZABEZPIECZENIE	OPIS AKCJI
POST	/api/internal/user/webhook/register	X-Keycloak-Secret	Webhook wyzwalany przez Keycloak w momencie rejestracji. Zapisuje usera w bazie PostgreSQL.
GET	/api/user/internal/couriers	Internal Secret	Interfejs dla Order-Service pobierający listę kurierów i ich aut potrzebnych do wyliczania tras.
GET	/api/user/internal/{id}	Internal Secret	Interfejs dla Order-Service pobierający dane konkretnego użytkownika (np. nadawcy).

## Order-Service

Serwis stanowiący logiczne centrum systemu, odpowiedzialny za zarządzanie cyklem życia zamówień oraz proces wyliczania optymalnych tras kurierskich. W przeciwieństwie do pozostałych modułów, **Order-Service** został zaimplementowany przy użyciu frameworka **Quarkus**, co zapewnia ekstremalnie szybki czas startu i niskie zużycie zasobów, kluczowe przy intensywnych obliczeniach algorytmicznych.

Moduł ten pełni rolę orkiestratora: przyjmuje zamówienia od klientów, komunikuje się z **User-Service** w celu pobrania danych o dostępnej flocie, a następnie wykorzystuje silnik **Timefold AI** (lub algorytmy alternatywne) do wyznaczenia tras. Po opłaceniu zamówienia (informacja z RabbitMQ od **Payment-Service**), zlecenie trafia do puli operacyjnej. Serwis integruje się również z mapami zewnętrznymi, aby zapewnić rzeczywiste czasy przejazdów.

### Dane techniczne:

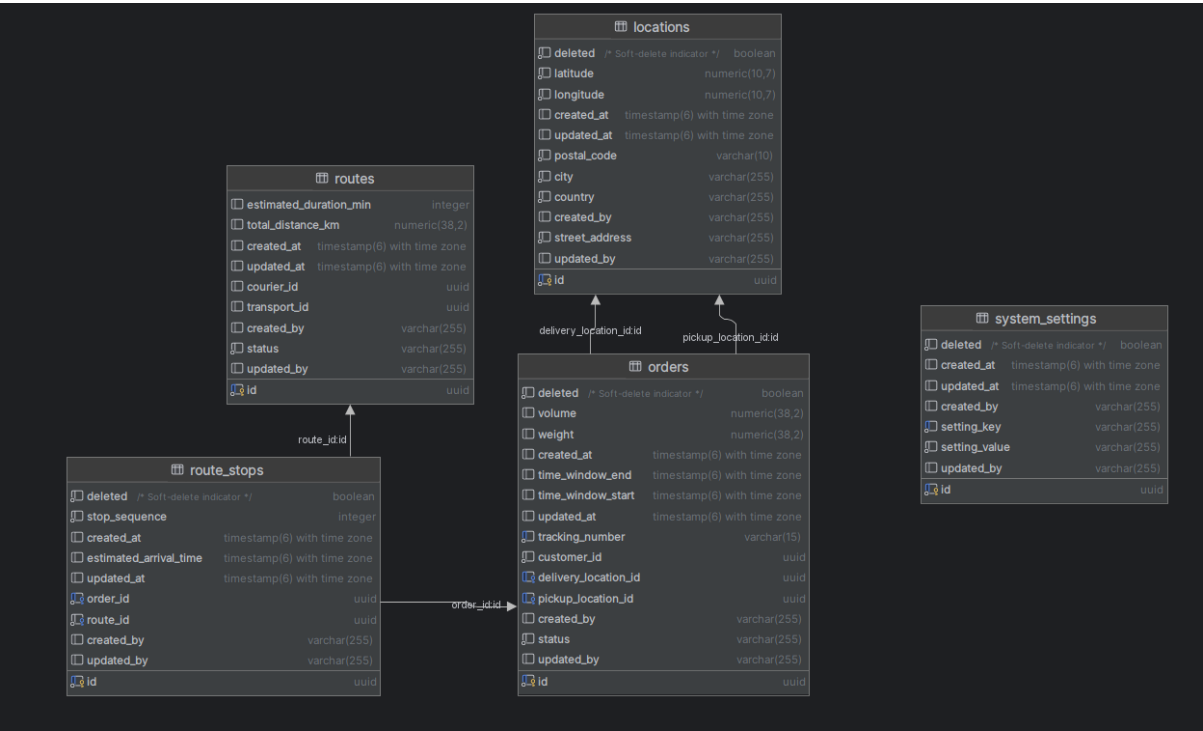
- **Framework:** Quarkus (Java 21)

- **Port:** 8082
- **Baza danych:** PostgreSQL (order\_service\_db)
- **Komunikacja:** REST API, RabbitMQ (Konsumpcja zdarzeń płatności, publikacja zmian statusu paczek).

Baza danych

Baza danych order\_service\_db dzieli się na pięć główne encje biznesowe, mapowane za pomocą JPA (Hibernate). Dodatkowo modele dziedziczą z klasy ControlledEntity, która automatycznie zarządza polami audytowymi (createdAt, updatedAt, createdBy, updatedBy).

Tabela	Opis
Orders	Przechowuje dane o przesyłce (waga, objętość), statusie oraz lokalizacji nadania i odbioru.
Locations	Przechowuje dokładne dane o lokalizacji (wysokość/szerokość geograficzna, adres)
Routes	Definiuje trasę przypisaną do konkretnego kuriera w danym dniu.
RouteStops	Tabela łącząca zamówienia z trasami, określająca kolejność (sequence) odwiedzin punktów.
SystemSettings	Przechowuje globalną konfigurację, np. aktualnie wybrany algorytm routingu.



Endpointy

Zarządzanie Zamówieniami (Order Controller)

Metoda	Endpoint	Uprawnienia	Opis Akcji
POST	/api/orders	CUSTOMER	Składanie nowego zamówienia. Wykonuje geokodowanie adresów w tle (HERE API), generuje unikalny numer listu przewozowego i nadaje status początkowy WAITING_FOR_PAYMENT.
GET	/api/orders	ADMIN	Złożony interfejs pobierający pełną listę wszystkich paczek w systemie. Obsługuje paginację, filtrowanie po konkretnych statusach oraz wyszukiwanie tekstowe (po numerze nadania).
GET	/api/orders/my	CUSTOMER	Zwraca listę zamówień powiązanych wyłącznie z obecnie zalogowanym użytkownikiem (weryfikacja na podstawie pola sub z tokena JWT).
GET	/api/orders/{id}	ADMIN, CUSTOMER, COURIER	Pobiera pełne szczegóły konkretnego zamówienia na podstawie jego unikalnego identyfikatora UUID.
GET	/api/orders/tracking/{trackingNumber}	ADMIN, CUSTOMER, COURIER	Alternatywna metoda pobierania pełnych szczegółów zamówienia za pomocą numeru listu przewozowego (Tracking Number).
GET	/api/orders/tracking/minimalized/{trackingNumber}	ALL (PermitAll)	Publiczny endpoint dla niezalogowanych użytkowników. Służy do śledzenia przesyłki z poziomu strony głównej, zwracając jedynie okrojony i bezpieczny model DTO (bez wrażliwych danych osobowych).
PUT	/api/orders/{id}	ADMIN, CUSTOMER	Pełna aktualizacja danych zamówienia (np. korekta wagi, gabarytów lub danych odbiorcy).
PATCH	/api/orders/{id}/status	ADMIN, COURIER, CUSTOMER	Zmiana logistycznego statusu przesyłki. Mechanizm ten publikuje również



			zdarzenie zmiany do RabbitMQ, wyzwalając powiadomienia e-mail/SMS.
<b>DELETE</b>	/api/orders/{trackingNumber}	ADMIN	Usunięcie (lub archiwizacja) zamówienia z bazy danych.
<b>GET</b>	/api/orders/stats	ADMIN	Zwraca zagregowane dane statystyczne (grupowanie paczek po statusach logistycznych), zasilające wykresy na Dashboardzie w panelu administracyjnym.

## Zarządzanie Trasami i Algorytmami (Routing & Route Controllers)

Metoda	Endpoint	Uprawnienia	Opis Akcji
<b>GET</b>	/api/routing/algorithm	ADMIN	Odczyt aktualnie aktywnej strategii algorytmicznej wyznaczania tras (Timefold AI, Greedy, Brute Force).
<b>POST</b>	/api/routing/optimize	ADMIN	Ręczne wymuszenie natychmiastowego uruchomienia mechanizmu optymalizacji dla wszystkich paczek zgromadzonych na węźle sortowniczym.
<b>GET</b>	/api/courier/route	COURIER	Pobranie w pełni zoptymalizowanej listy przystanków, ułożonej w odpowiedniej kolejności logicznej, dla aktualnie zalogowanego kuriera.

## Integracje z zewnętrznymi API (HERE & Google Maps)



W celu zapewnienia precyzji logistycznej, system korzysta z hybrydowego modelu usług mapowych:

1. **HERE Geocoding API:** Wykorzystywane w momencie składania zamówienia. Zamienia adres tekstowy wpisany przez klienta (np. "Łódzka 97, Łódź") na współrzędne geograficzne (Latitude/Longitude). Wybrano HERE ze względu na korzystny model kosztowy przy dużej liczbie zapytań o geokodowanie.

```
package p.lodz.pl.controller;

import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.QueryParam;
import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;
import p.lodz.pl.dto.maps.HereGeocodeResponse;

@RegisterRestClient(configKey = "here-geocode-api")
public interface HereGeocodeClient {
    @GET
    @Path("/v1/geocode")
    HereGeocodeResponse getGeocode(
        @QueryParam("q") String query,
        @QueryParam("apiKey") String apiKey
    );
}
```

2. **Google Maps Directions API:** Używane na końcowym etapie wizualizacji trasy dla kuriera. Dostarcza precyzyjne dane o natężeniu ruchu drogowego i generuje polilinie wyświetlane na mapie w aplikacji frontendowej.

```
package p.lodz.pl.controller;

import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.QueryParam;
import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;
import p.lodz.pl.dto.maps.DirectionsApiResponse;

@RegisterRestClient(configKey = "google-maps-api")
public interface GoogleMapsClient {
    @GET
    @Path("/maps/api/directions/json")
    DirectionsApiResponse getDirections(
        @QueryParam("origin") String origin,
        @QueryParam("destination") String destination,
        @QueryParam("key") String apiKey
    );
}
```

```
}
```

3. **Optymalizacja kosztów (Wzór Haversine):** Aby uniknąć płatnych zapytań do API Map podczas setek tysięcy iteracji algorytmu Timefold, system wykorzystuje matematyczny wzór Haversine do obliczania odległości w linii prostej na sferze. API Google Maps jest odpytywane dopiero dla finalnie wyznaczonej trasy.

```
private static final int EARTH_RADIUS_METERS = 6371000;

public static String generateTrackingNumber() {
    String randomPart = UUID.randomUUID().toString().substring(0, 8).toUpperCase();
    return "BD-" + randomPart.substring(0, 4) + "-" + randomPart.substring(4, 8);
}

public static int calculateDistance(Location loc1, Location loc2) {
    if (loc1 == null || loc2 == null) return 0;

    double lat1 = loc1.getLatitude().doubleValue();
    double lon1 = loc1.getLongitude().doubleValue();
    double lat2 = loc2.getLatitude().doubleValue();
    double lon2 = loc2.getLongitude().doubleValue();

    double dLat = Math.toRadians(lat2 - lat1);
    double dLon = Math.toRadians(lon2 - lon1);

    double a = Math.sin(dLat / 2) * Math.sin(dLat / 2) +
        Math.cos(Math.toRadians(lat1)) * Math.cos(Math.toRadians(lat2)) *
        Math.sin(dLon / 2) * Math.sin(dLon / 2);

    double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));

    return (int) (EARTH_RADIUS_METERS * c);
}
```

## *Notification-Service*

Serwis odpowiedzialny za scentralizowaną obsługę komunikacji ze wszystkimi uczestnikami procesu: nadawcami, odbiorcami paczek oraz kurierami. Pełni rolę głównego węzła informacyjnego w systemie, obsługując trzy niezależne kanały przekazu: wiadomości e-mail (poprzez serwer SMTP), alerty SMS (integracja z zewnętrznym API) oraz powiadomienia w czasie rzeczywistym na urządzenia użytkowników (wykorzystując technologię SSE – Server-Sent Events).

Architektura serwisu opiera się na modelu asynchronicznym (Event-Driven). Notification-Service nie jest bezpośrednio odpytywany przez inne moduły; zamiast tego aktywnie

nasłuchuje na zdarzenia spływające z brokera RabbitMQ (np. informacje o nowym zleceniu z Order-Service lub o zaksięgowaniu wpłaty z Payment-Service). Takie odseparowanie gwarantuje wysoką odporność na awarie – ewentualne problemy z zewnętrzną bramką SMS nie blokują i nie opóźniają głównych procesów biznesowych, takich jak finalizacja płatności.

Ponadto, serwis jest zoptymalizowany pod kątem wielojęzyczności frontendu (i18n) – zamiast generować gotowe teksty, wysyła do aplikacji w React surowe zdarzenia w formacie JSON, co pozwala na natychmiastowe tłumaczenie komunikatów po stronie przeglądarki. Całość operacji jest ściśle monitorowana, a logi wysyłkowych sukcesów i błędów trafiają do dedykowanej bazy PostgreSQL. Podobnie jak w reszcie systemu, dostęp do historii powiadomień dla administratorów jest wystawiony przez interfejsy API, których autoryzacja zabezpieczona jest tokenami JWT weryfikowanymi przez Keycloak.

### Dane techniczne:

Framework: Spring boot (Java 21)

Port: 8083

Baza danych: PostgreSQL

Komunikacja asynchroniczna: RabbitMQ (publikacja zdarzeń `PaymentEvent` po udanej transakcji)

Bezpieczeństwo i Autoryzacja: OAuth2 / Bearer Token JWT (walidacja kluczy asymetrycznych JWKS z serwera Keycloak dla endpointów administracyjnych)

Integracje Zewnętrzne:

- Zewnętrzne REST API (Bramka SMS).
- Protokół SMTP (JavaMail) do wysyłki wiadomości e-mail

### Baza danych

Schemat bazy danych (notification\_service\_db)

Email logs			
UUID	ID	PK	Unikalny identyfikator (klucz główny)
TIMESTAMP	CREATED_AT		Data i czas utworzenia
TEXT	ERROR_MESSAGE		Treść ewentualnego błędu
TEXT	MESSAGE_CONTENT		Treść powiadomienia
UUID	ORDER_ID		Identyfikator zamówienia
VARCHAR(255)	RECIPIENT_EMAIL		Adres odbiorcy
VARCHAR(255)	STATUS		Status wysyłki powiadomienia
VARCHAR(255)	TRACKING_NUMBER		Numer śledzenia paczki

SMS logs
----------

UUID	ID	PK	Unikalny identyfikator (klucz główny)
TIMESTAMP	CREATED_AT		Data i czas utworzenia
TEXT	ERROR_MESSAGE		Treść ewentualnego błędu
TEXT	MESSAGE_CONTENT		Treść powiadomienia
UUID	ORDER_ID		Identyfikator zamówienia
VARCHAR(255)	RECIPIENT_PHONE		Numer telefonu odbiorcy
VARCHAR(255)	STATUS		Status wysyłki powiadomienia
VARCHAR(255)	TRACKING_NUMBER		Numer śledzenia paczki

## Endpointy

METODA	ENDPOINT	WYMAGANE UPRAWNIENIA	OPIS AKCJI
GET	/api/notifications/stream	Otwarty (PermitAll)	Otwiera strumień SSE. Wypycha asynchronicznie powiadomienia
GET	/api/notifications/logs/email	PreAuth(ADMIN)	Zwraca pełną historię wysłanych powiadomień E-MAIL
GET	/api/notifications/logs/sms	PreAuth(ADMIN)	Zwraca pełną historię wysłanych powiadomień SMS

## Payment-Service

Serwis odpowiedzialny za bezpieczne przetwarzanie transakcji finansowych oraz zarządzanie cyklem życia płatności za przesyłki. Głównym zadaniem modułu jest ścisła integracja z zewnętrznym dostawcą usług płatniczych (Stripe API). Zamiast procesować wrażliwe dane kart kredytowych bezpośrednio w systemie, serwis deleguje ten proces na zewnątrz, generując bezpieczne sesje płatności (Checkout Sessions). Takie podejście zdejmuje z infrastruktury ciężar utrzymania rygorystycznych certyfikatów bezpieczeństwa, przenosząc odpowiedzialność za transakcję na renomowanego operatora.

Proces potwierdzania wpłat zrealizowany jest w pełni asynchronicznie przy użyciu mechanizmu Webhooków. Serwis wystawia dedykowany endpoint, który nasłuchuje zdarzeń płynących bezpośrednio z serwerów Stripe. Każde przychodzące żądanie jest kryptograficznie weryfikowane (walidacja podpisu Webhook Secret), co całkowicie eliminuje ryzyko fałszerstwa transakcji. Po zaksięgowaniu wpłaty, Payment-Service zmienia status zamówienia w dedykowanej bazie PostgreSQL i natychmiast rozgłasza to zdarzenie (publikując PaymentEvent) do brokera RabbitMQ. Dzięki temu powiązane moduły, takie jak Order-Service (do zmiany statusu paczki) czy Notification-Service (do wysyłki faktury), reagują w czasie rzeczywistym bez tworzenia wąskich gardeł w komunikacji sieciowej.

Ponadto, serwis wyposażono w zautomatyzowane procesy utrzymaniowe (Schedulery). Odpowiadają one za cykliczne usuwanie porzuconych, nieopłaconych transakcji, co realizowane jest z wykorzystaniem zoptymalizowanych operacji masowych bezpośrednio na silniku bazy danych, omijając problem przeciążenia pamięci RAM. Dostęp do historii

transakcji (w przyszłości) dla potrzeb panelu administracyjnego odbywałby się poprzez interfejsy REST API, które są ściśle chronione mechanizmem autoryzacji opartym na tokenach JWT z serwera Keycloak.

### Dane techniczne:

Framework: Spring boot (Java 21)

Port: 8084

Baza danych: PostgreSQL

Bezpieczeństwo i Autoryzacja: OAuth2 / Bearer Token JWT (walidacja kluczy asymetrycznych JWKS z serwera Keycloak dla endpointów administracyjnych i wewnętrznych)

- Kryptograficzna weryfikacja podpisów (Stripe Webhook Secret) chroniąca przed fałszowaniem statusów płatności.

Integracje Zewnętrzne: Zewnętrzne REST API operatora płatności (Stripe API - generowanie Checkout Sessions oraz nasłuchiwanie Webhooków).

### Baza danych

Schemat bazy danych (payment\_service\_db)

SMS logs			
UUID	ID	PK	Unikalny identyfikator (klucz główny)
NUMERIC	AMOUNT		Kwota zapłaty
TIMESTAMP	CREATED_AT		Czas stworzenia sesji płatności
VARCHAR(3)	CURRENCY		Waluta płatności
UUID	ORDER_ID		Identyfikator zamówienia
VARCHAR(255)	STATUS		Status płatności
VARCHAR(255)	STRIPE_SESSION_ID		Identyfikator płatności Stripe
VARCHAR(255)	UPDATED_AT		Czas zapłaty

### Endpointy

METODA	ENDPOINT	WYMAGANE UPRAWNIENIA	OPIS AKCJI
POST	/api/payments/create-session	Zalogowany Użytkownik	Przyjmuje dane zamówienia i generuje bezpieczną sesję płatności (Checkout Session) poprzez API Stripe. Zwraca do frontendu wygenerowany adres URL bramki płatniczej
POST	/api/payments/webhook	Otwarty (PermitAll)	Nasłuchuje zdarzeń bezpośrednio z serwerów Stripe (np. o udanej wpłacie).

# Security

## Keycloak

Głównym punktem security całej aplikacji odpowiedzialnym za bezpieczeństwo, uwierzytelnianie oraz autoryzację jest serwer Keycloak. Działa jako niezależny serwis tożsamości implementujący OAuth2 oraz OpenID Connect. Pozwala to na pełną separację danych logowania od logiki biznesowej mikroservisów.

Całość struktury bezpieczeństwa – w tym definicje ról (ADMIN, CUSTOMER, COURIER), ustawienia polityk haseł oraz konfiguracja klienta frontendowego (boat-delivery-frontend) – została utrwalona w pliku konfiguracyjnym realm-export.json. Dzięki temu proces wdrażania systemu jest w pełni zautomatyzowany i powtarzalny w środowiskach kontenerowych (Docker). Plik ten definiuje również Protocol Mappers, które odpowiadają za wstrzykiwanie ról użytkownika do tokena JWT w sekcji realm\_access, co jest kluczowe dla późniejszej autoryzacji w mikroservisach.

System został rozszerzony o dedykowaną logikę biznesową przy użyciu interfejsów SPI (Service Provider Interface):

- **Walidacja danych (UniquePhoneValidator):** Zaimplementowano walidator numeru telefonu, który wymusza na użytkowniku podanie numeru w formacie międzynarodowym (np. +48...) i weryfikuje jego unikalność w skali całego systemu (baza danych Keycloak oraz User-Service) już na etapie wypełniania formularza.
- **Synchronizacja danych (Event Listener SPI):** Aby zapewnić spójność między bazą tożsamości Keycloak, a relacyjną bazą user-service, stworzono plugin UserRegistrationEventListener. Nasłuchuje on zdarzeń typu REGISTER oraz UPDATE\_ROLE. W momencie ich wystąpienia, Keycloak wykonuje bezpieczne połączenie typu **M2M (Machine-to-Machine)** do wewnętrznego endpointu user-service, przesyłając metadane użytkownika. Gwarantuje to, że każdy użytkownik w Keycloaku ma swoje odzwierciedlenie w bazie PostgreSQL, co pozwala na niezależną administrację całej logiki biznesowej.

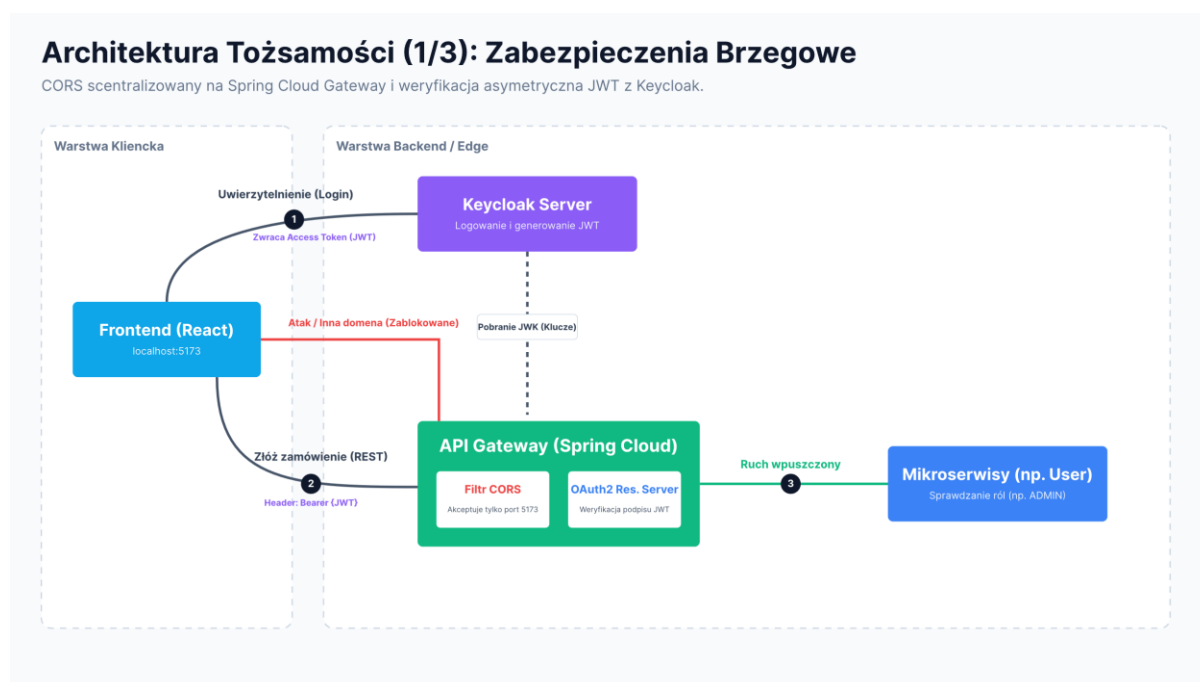
Proces logowania i rejestracji jest podobny do reszty aplikacji dzięki zastosowaniu narzędzia Keycloakify. Pozwoliło ono na zbudowanie customowego motywu (boat-theme) przy użyciu biblioteki React. Dzięki temu formularze dostarczane bezpośrednio przez serwer Keycloak współdzielą te same komponenty UI, co główny frontend aplikacji, eliminując efekt przejścia użytkownika między różnymi systemami podczas autoryzacji.

Po poprawnym zalogowaniu, Keycloak wystawia asymetrycznie podpisany token JWT, który pełni rolę autoryzacji w całym systemie:

1. **Frontend** dołącza go do każdego żądania w nagłówku Authorization: Bearer.
2. **API Gateway** weryfikuje ważność tokena (data wygaśnięcia) oraz jego podpis przy użyciu kluczy publicznych udostępnianych przez Keycloak pod adresem /certs.

3. **Mikroserwisy** wykorzystują zdekodowany token do identyfikacji użytkownika (poprzez pole sub zawierające UUID) oraz do autoryzacji dostępu do konkretnych metod (sprawdzanie ról wewnątrz tokena).

Taka architektura zapewnia, że system jest **bezstanowy (stateless)** – mikroserwisy nie muszą przechowywać sesji użytkownika, ponieważ wszystkie niezbędne informacje o jego uprawnieniach przenoszone są wewnątrz bezpiecznego, podpisanego kryptograficznie tokena.



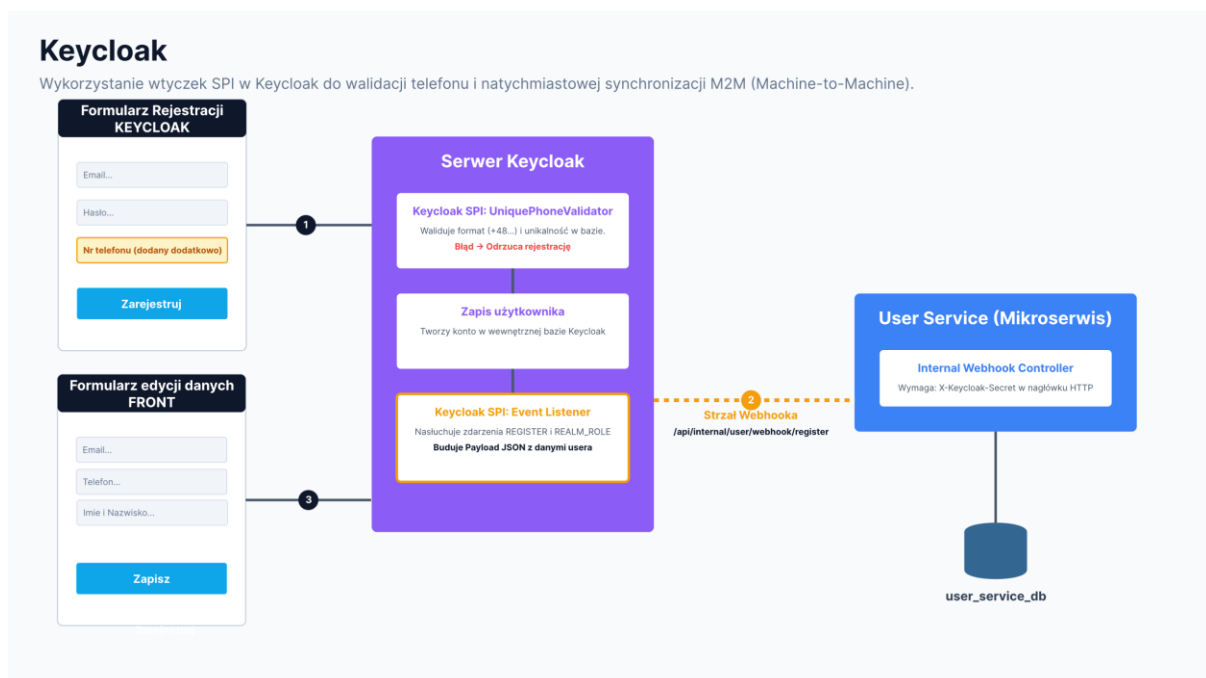
## CORS

Polityka współdzielenia zasobów między mikroserwisami w systemie została zaimplementowana w sposób całkowicie scentralizowany. Zamiast powielać reguły bezpieczeństwa w każdym z mikroserwisów, główną barierą weryfikującą żądania przeglądarkowe jest API Gateway (oparty na Spring Cloud Gateway). Został on skonfigurowany za pomocą filtra `CorsWebFilter` tak, aby akceptować ruch wychodzący wyłącznie z zaufanej domeny aplikacji klienckiej (`http://localhost:5173`). Brama precyzyjnie definiuje dozwolone metody HTTP (m.in. GET, POST, PUT, PATCH, DELETE) oraz zezwala na przesyłanie poświadczeń (`AllowCredentials`), co jest kluczowe dla prawidłowej obsługi sesji i tokenów autoryzacyjnych.

Dzięki takiemu podejściu architektonicznemu docelowe mikroserwisy (takie jak `user-service` czy `order-service`) mają całkowicie wyłączoną obsługę CORS na poziomie własnych konfiguracji Spring Security (`.cors(AbstractHttpConfigurer::disable)`). Rozwiązanie to zapobiega konfliktom podwójnej weryfikacji żądań typu `preflight` (z użyciem metody `OPTIONS`) oraz odciąża logikę biznesową z zadań sieciowych. Jednocześnie wymusza to na aplikacjach webowych bezwzględną komunikację z systemem za pośrednictwem chronionego gatewaya.



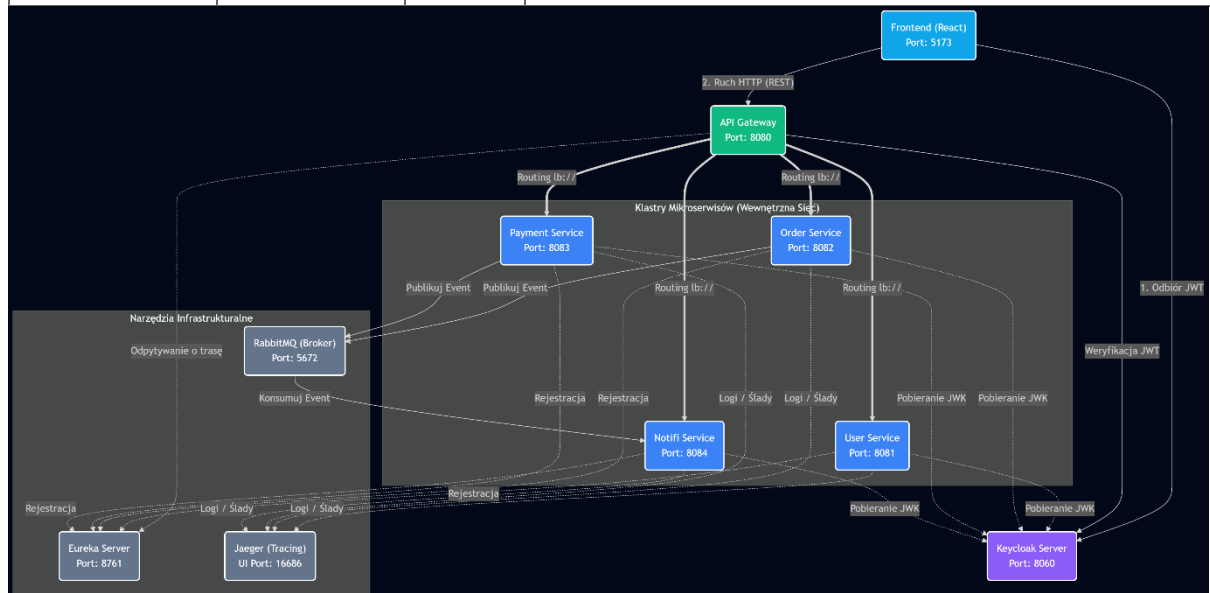
Mimo że API Gateway pełni rolę głównego punktu wejścia do systemu (filtrując zapytania CORS i odrzucając ruch bez podstawowych poświadczeń), architektura bezpieczeństwa opiera się na modelu Zero Trust. Każdy z mikroservisów posiada własną, wbudowaną warstwę ochrony (np. Spring Security lub zabezpieczenia frameworka Quarkus). Mikroservisy nie ufają bezwarunkowo żądaniom przekazywanym przez bramkę. Każde zapytanie trafiające do docelowego serwisu musi zawierać token JWT w nagłówku. Serwis samodzielnie go dekoduje, po raz kolejny weryfikuje jego kryptograficzną sygnaturę z użyciem kluczy publicznych (JWK) pobranych z Keycloak, a następnie ekstrahuje przypisane role (np. ADMIN, COURIER). Dzięki temu bezpośredni dostęp do portów mikroservisów (np. z pominięciem bramki Gateway) przez nieautoryzowanego użytkownika lub atakującego wewnątrz sieci jest całkowicie zablokowany. Dodatkowo, aby zapobiec nieautoryzowanemu wywoływaniu wewnętrznych operacji, komunikacja M2M (Machine-to-Machine) między samymi mikroservisami zabezpieczona jest dedykowanymi sekretami wewnętrznymi (Pre-Shared Keys).



## Konfiguracja Gatewaya i portów mikroservisów

<b>Klient</b>	<b>Frontend (React)</b>	5173	Interfejs użytkownika. Komunikuje się bezpośrednio z API Gateway (8080) oraz serwerem Keycloak (8060) w celu logowania.
<b>Edge / Security</b>	<b>API Gateway</b>	8080	Główna brama systemu. Odbiera ruch z zewnątrz (CORS dozwolony dla 5173), weryfikuje JWT i trasuje zapytania do mikroservisów.

<b>Edge / Security</b>	<b>Keycloak</b>	8060	Serwer tożsamości. Wystawia formularze logowania dla frontendu. Gateway i mikroserwisy odpytują go o klucze publiczne (JWK).
<b>Infrastruktura</b>	<b>Eureka Server</b>	8761	Service Discovery. Rejestrują się w nim API Gateway oraz wszystkie mikroserwisy, aby umożliwić dynamiczny routing (lb://).
<b>Infrastruktura</b>	<b>Jaeger (UI)</b>	16686	Interfejs graficzny do śledzenia logów rozproszonych. Mikroserwisy wysyłają do niego logi w tle – Load Balancing
<b>Mikroserwisy</b>	<b>User Service</b>	8081	Zarządzanie tożsamością i flotą. Rejestruje się w Eurece. Ukryty za Gatewayem (/api/user/**, /api/transport/**).
<b>Mikroserwisy</b>	<b>Order Service</b>	8082	Zarządzanie paczkami i trasami. Rejestruje się w Eurece. Ukryty za Gatewayem (/api/orders/**).
<b>Mikroserwisy</b>	<b>Notifi Service</b>	8083	Powiadomienia (SSE, SMS). Rejestruje się w Eurece. Ukryty za Gatewayem (/api/notifications/**).
<b>Mikroserwisy</b>	<b>Payment Service</b>	8084	Obsługa płatności. Rejestruje się w Eurece. Ukryty za Gatewayem (/api/payments/**).
<b>Bazy / Brokery</b>	<b>RabbitMQ</b>	5672 <i>(15672 UI)</i>	Magistrala zdarzeń. Mikroserwisy (Order, Payment) publikują do niej zdarzenia, a Notifi Service je konsumuje.
<b>Bazy / Brokery</b>	<b>PostgreSQL</b>	5432	Relacyjne bazy danych. W środowisku Docker każdy mikroservis łączy się do własnej, odseparowanej logicznie bazy na tym porcie.



# Load balancing i Eureka

## *Netflix Eureka*

Zastosowanie środowiska kontenerowego (Docker) sprawia, że adresy IP poszczególnych mikroserwisów przydzielane są dynamicznie. Aby wyeliminować konieczność sztywnego wpisywania adresów IP w plikach konfiguracyjnych, wdrożono serwer Netflix Eureka, pełniący rolę centralnego rejestru usług. Mechanizm ten ściśle współpracuje z weryfikacją stanu instancji (Health Checks):

- **Rejestracja:** Każdy uruchamiany mikroserwis (np. user-service, order-service) automatycznie zgłasza do Eureka swoją nazwę, adres IP oraz port.
- **Heartbeats:** Aby utrzymać status UP (Dostępny), usługi wysyłają do Eureka cykliczne sygnały kontrolne (/health).
- **Failover:** W przypadku awarii serwisu lub wyłączenia kontenera, sygnały przestają docierać. Eureka natychmiast zmienia status takiej instancji na DOWN i usuwa ją z rejestru, zapobiegając kierowaniu tam ruchu sieciowego.

Zamiast przechowywać docelowych adresów IP mikroserwisów, w konfiguracji Gatewaya wykorzystano schemat lb:// (np. lb://order-service), który aktywuje mechanizm Client-Side Load Balancing (Spring Cloud LoadBalancer). Dzięki temu, gdy żądanie trafia na bramę, odczytuje ona docelową nazwę usługi, odpytuje lokalny bufor Eureka o listę zdrowych instancji przypisanych do tej nazwy oraz przypadku poziomego skalowania Load Balancer równomiernie rozdziela nadchodzące zapytania. Zapewnia to odporność systemu na awarie pojedynczych węzłów i automatyczną adaptację routingu do aktualnej topologii sieci.

Dodatkowo w architekturze mikroserwisowej pojedyncza akcja (np. złożenie zamówienia) wymaga komunikacji wielu usług (np. Gateway → Order-Service → User-Service → Notifi-Service). Aby umożliwić skuteczne debugowanie i analizę logów przecinających granice sieciowe, zaimplementowano narzędzie Jaeger wspierane przez standard OpenTelemetry / Micrometer.

- **Propagacja kontekstu (Trace ID):** Każde zapytanie trafiające do Gatewaya otrzymuje unikalny globalny identyfikator (Trace ID), który jest wstrzykiwany do nagłówków HTTP i przekazywany przez całą ścieżkę komunikacji.
- **Zakres prac (Span):** Czas spędzony na obsłudze zapytania wewnątrz pojedynczego mikroserwisu tworzy niezależny "Span", powiązany z głównym Trace ID.
- **Wizualizacja i metryki:** Mikroserwisy asynchronicznie przesyłają wygenerowane dane do serwera Jaeger. Przejrzysty interfejs graficzny (port 16686) pozwala administratorom prześledzić dokładny przepływ żądania, czas odpowiedzi poszczególnych usług i natychmiastowo zlokalizować błędy (np. wyjątki rzucane w kodzie) lub wąskie gardła wydajnościowe.

## Jaeger

Dodatkowo w naszej architekturze, aby weryfikować poprawność działania mechanizmów równoważenia obciążenia (Load Balancing), zaimplementowano narzędzie Jaeger wspierane przez standard OpenTelemetry. Dzięki temu można weryfikować:

- Propagacja kontekstu (Trace ID): Każde zapytanie trafiające do Gatewaya otrzymuje unikalny globalny identyfikator (Trace ID), który jest wstrzykiwany do nagłówków HTTP i przekazywany przez całą ścieżkę komunikacji między mikroservisami.
- Identyfikacja instancji (Span): Czas spędzony na obsłudze zapytania wewnątrz pojedynczego mikroservisu tworzy niezależny Span, który niesie ze sobą metadane informujące o tym, która dokładnie fizyczna instancja podjęła się obsługi danego żądania.
- Weryfikacja dystrybucji ruchu: Mikroservisy asynchronicznie przesyłają wygenerowane dane do serwera Jaegera, który głównie jest używany jako narzędzie audytowe dla skalowania horyzontalnego. Pozwala to naocznie udowodnić i przetestować, że w przypadku uruchomienia np. dwóch instancji user-service, kolejne nadchodzące requesty są faktycznie i równomiernie rozdzielane przez API Gateway między wszystkie dostępne instancje.

## Algorytmy użyte do optymalizacji tras

System BoatDelivery wspiera dynamiczną zmianę strategii wyznaczania tras (wzorzec projektowy *Strategy*), co pozwala administratorowi balansować między szybkością obliczeń a ich precyzją.

### Architektura API: Silniki Optymalizacji Tras (VRP)

Trzy niezależne strategie rozwiązywania problemu VRP zaimplementowane w Order Service.

Dostępne Strategie (RoutingStrategy Interface)

TIMEFOLD AI

Timefold Constraint Solver

Domyślny, zaawansowany silnik oparty na sztucznej inteligencji. Równocześnie przetwarza setki kombinacji, operując na precyzyjnym systemie punktów karnych (Hard/Soft Scores). Balansuje optymalny czas, limity ładowności i sprawiedliwe obciążenie kurierów.

GREEDY

Algorytm Zachłanny (Nearest)

Podejście "najbliższego sąsiada". Wybiera zawsze najbliższy możliwy punkt dostawy bez analizy całej trasy. Ekstremalnie szybki mechanizm, idealny dla nieskomplikowanych scenariuszy lub jako awaryjny *fallback* w przypadku braku zasobów.

BRUTE FORCE

Metoda Siłowa z Chunkingiem

Mechanizm sprawdzający absolutnie każdą matematyczną permutację przystanków  $O(n!)$ . Daje 100% gwarancję optymalnej trasy. Limitowany do fragmentów (tzw. chunków) po max. 8 paczek na jedno obliczenie, aby zapobiec paraliżowi obliczeniowemu serwera.

### Timefold AI (Constraint Solver)

Jest to domyślny i najbardziej zaawansowany silnik optymalizacyjny. Wykorzystuje techniki metaheurystyczne do rozwiązywania problemu VRP (*Vehicle Routing Problem*).

- **Reguły twarde (Hard Constraints):** Algorytm pilnuje, aby suma wag paczek na pace nie przekroczyła `maxCargoCapacity` pojazdu.
- **Reguły miękkie (Soft Constraints):** Minimalizacja całkowitego dystansu oraz balansowanie obciążenia (by każdy kurier miał podobną liczbę zleceń).

## Architektura API: Reguły Biznesowe Timefold AI

Modelowanie ograniczeń (`RouteConstraintProvider`) sterujących optymalizacją.

🔧 Hard & Soft Constraints		
<b>HARD LIMIT</b>	<b>Vehicle Capacity Exceeded</b>	<b>Złota zasada logistyki:</b> Algorytm śledzi wagę przesyłek dynamicznie. Uwzględnia, że kurier odbiera nowe paczki i oddaje stare na tej samej trasie. Puła obciążenia auta ani przez sekundę nie może przekroczyć określonej pojemności (np. 1000 kg).
<b>SOFT LIMIT</b>	<b>Minimize Total Distance</b>	Solver otrzymuje surowe kary punktowe (-1 Soft Score) za każdy wyliczony metr dystansu. Wymusza to na silniku szukanie coraz krótszych i ciaśniejszych spłotów trasy względem głównej sortowni logistycznej na Lodowej 97.
<b>SOFT LIMIT</b>	<b>Balance Courier Load</b>	System nakłada potężne kary rosnące wykładniczo (do kwadratu) dla przeładowanych tras. Dzięki temu paczki są równomiernie rozdzielane na dostępną flotę, zapobiegając sytuacji, gdy jeden kurier pracuje 12 godzin, a inny kończy po dwóch.

## Algorytm Zachłanny (Greedy Strategy)

Stosowany do błyskawicznych, uproszczonych obliczeń.

- **Działanie:** System wybiera pierwszy wolny pojazd i przypisuje mu najbliższy dostępny punkt odbioru/dostawy.
- **Zaleta:** Bardzo niska złożoność obliczeniowa.
- **Wada:** Często pomija globalne optimum, co może skutkować dłuższymi trasami końcowymi.

# Strategia Wyznaczania Tras: Algorytm Zachłanny (Greedy)

Szybki mechanizm "Najbliższego Sąsiada" optymalizujący lokalne parametry bez spojrzenia w przyszłość.

## Główne założenia i implementacja (GreedyRoutingStrategy)

### LOGIKA KROKU

#### Najbliższy Sąsiad (Nearest)

Algorytm analizuje obecną pozycję kuriera (startowo: baza) i spośród wszystkich nieprzypisanych paczek wybiera tę, do której dystans w linii prostej (Haversine) jest najkrótszy. Po przydzieleniu, punkt ten staje się nową pozycją kuriera.

### WALIDACJA FLOTY

#### Pojemność i Weryfikacja Auta

System śledzi zmienną `routePeakLoad` (maksymalne historyczne obciążenie auta podczas trasy). Kurierzy bez przypisanego pojazdu (`capacity <= 0`) są w 100% ignorowani. Paczka jest odrzucana dla danego kuriera, jeśli jej waga przekroczyłaby limit ładowności fizycznej.

### FALLBACK

#### Zarządzanie Brakiem Floty

Jeżeli wszystkie pojazdy w systemie zostaną załadowane do 100% pojemności, a w sortowni zostaną paczki, włącza się bezpieczny fallback. Algorytm wymusza nadbagaż, przypisując paczkę do kuriera o najmniejszym obecnym obciążeniu (wymuszenie sprawiedliwości przy przeładunku).

### WADA

#### Brak balansu (Zasada skrajności)

Z natury algorytm nie jest sprawiedliwy. W pierwszej kolejności łąduje najbliższe auto "pod sam korek", zostawiając innych kurierów w bazie. Dodatkowo brak spojrzenia w przyszłość często powoduje powroty i przecinanie się tras pod koniec zmiany.

## Metoda Siłowa (Brute Force z Chunkingiem)

Zapewnia znalezienie matematycznie najlepszej trasy, ale z uwagi na złożoność  $O(n!)$  jest ograniczona.

- **Optymalizacja:** Stosujemy mechanizm **Chunkingu** – dzielimy listę przystanków na mniejsze bloki (max 8-10 punktów), dla których wyliczane są wszystkie możliwe permutacje. Pozwala to na idealne ułożenie trasy lokalnie bez zawieszania serwera.

## Strategia Wyznaczania Tras: Metoda Siłowa (Brute Force)

Matematyczna gwarancja perfekcyjnego, najkrótszego dystansu poprzez sprawdzenie każdej możliwej ścieżki.

### Mechanika i ograniczenia (BruteForceRoutingStrategy)

#### PODZIAŁ PRACY

##### Rozdzielenie "Modulo"

W przeciwieństwie do algorytmu zachłannego, Brute Force musi zapewnić sprawiedliwy podział pracy dla ograniczenia kombinatoryki. Używamy operatora reszty z dzielenia (index % liczba\_kurierów), rozdając paczki po jednej dla każdego dostępnego auta, ignorując początkowo ich lokalizacje na mapie.

#### OBLICZENIA

##### Generowanie Permutacji

Każdy kurier otrzymuje swoją sub-listę paczek. Następnie system używa algorytmu rekurencyjnego (generatePermutations) ze złożonością  $O(n!)$  do sprawdzenia absolutnie każdego możliwego wariantu połączenia punktów i wyłonienia trasy o najmniejszym całkowitym dystansie od i do bazy.

#### KRYTYCZNY BŁĄD

##### Złożoność $O(n!)$ i Hibernate Crash

Wygenerowanie trasy dla zaledwie 15 paczek to **1.3 biliona sprawdzeń**, co sparaliżowałoby infrastrukturę. Dodatkowo, aby uniknąć błędów modyfikacji kolekcji bazy danych w locie ("Detached entity passed to persist"), algorytm pracuje w całości na bezpiecznych, płytkich klonach encji (Mockach).

## Zarządzanie Wydajnością: Mechanizm Chunkingu

Rozwiązanie problemu złożoności czasowej  $O(n!)$  w algorytmie Brute Force.

### Podział zbioru na optymalizowalne klastry (MAX\_CHUNK\_SIZE = 8)

#### Faza 1: Pula Kuriera

##### KURIER\_A

##### Dostawa: 20 Paczek

Próba przeliczenia permutacji dla 20 paczek to ok.  $2.4 \times 10^{18}$  kombinacji. Bez zabezpieczeń, serwer nie ukończyłby zadania przez dekady.

Podział (SubList)

#### Faza 2: Izolowane Obliczenia

##### Chunk 1 (Paczki 1-8)

40 320 kombinacji.  
Wyliczone lokalne optimum w milisekundy.

##### Chunk 2 (Paczki 9-16)

Kolejne odizolowane 40 320 kombinacji.

##### Chunk 3 (Paczki 17-20)

Zaledwie 24 kombinacje dla ostatnich 4 paczek.

Synteza (AddAll)

#### Faza 3: Wynik Końcowy

##### SUKCES OBLICZENIOWY

##### Gotowa Trasa (Złożona)

Łączymy optymalne kawałki z powrotem w jedną trasę. Zamiast kwintyliona operacji, wykonujemy ich łącznie **~80 000**. Uzyskujemy trasę bliską idealowi ze 100% gwarancją stabilności serwera.

## Aplikacja Klientka (Architektura Frontendu)

Warstwa prezentacyjna systemu BoatDelivery została zaprojektowana jako nowoczesna aplikacja typu SPA (Single Page Application). Zrezygnowano z tradycyjnego renderowania po stronie serwera (SSR) na rzecz w pełni rozdzielonej architektury, w której frontend komunikuje się z logiką biznesową wyłącznie za pośrednictwem API Gateway.

**Technologie i optymalizacja:** Projekt oparto na bibliotece **React** w połączeniu z rygorystycznym typowaniem statycznym **TypeScript**, co drastycznie zredukowało liczbę błędów w czasie wykonywania (runtime errors). Jako narzędzie budujące (bundler) wykorzystano **Vite**, które dzięki wykorzystaniu modułów ES (ESM) zapewnia

natychmiastowy czas przeładowania modułów (HMR) podczas developmentu oraz wysoce zoptymalizowaną paczkę wynikową na produkcję.

**Zarządzanie stanem i i18n:** Logika biznesowa po stronie przeglądarki została odseparowana od warstwy widoku za pomocą autorskich, niestandardowych Hooków (np. `useAdminOrders`, `usePayment`). Zapewnia to wysoką reużywalność kodu. Aplikacja natywnie wspiera wielojęzyczność (internacjonalizację) dzięki bibliotece `react-i18next`. Słowniki tłumaczeń (język polski i angielski) ładowane są asynchronicznie, co pozwala na natychmiastową zmianę języka interfejsu bez przeładowywania strony.

**Spójność interfejsu (Keycloakify i Shadcn UI):** Interfejs graficzny zbudowano z wykorzystaniem biblioteki **Tailwind CSS** oraz bezgłowych (headless) komponentów **Shadcn UI**, co gwarantuje pełną responsywność (RWD) oraz natywną obsługę trybów Light/Dark Mode. Kluczowym osiągnięciem architektonicznym na froncie jest wykorzystanie narzędzia **Keycloakify**. Pozwoliło ono na nadpisanie domyślnych, archaicznych stron logowania i rejestracji serwera Keycloak za pomocą tych samych komponentów React oraz klas Tailwind, z których korzysta główna aplikacja. Dzięki temu użytkownik przechodzący do procesu autoryzacji nie odczuwa opuszczenia głównego systemu – interfejs, walidacja i motywy wizualne pozostają w 100% spójne.

## Architektura Sterowana Zdarzeniami (Event-Driven Architecture)

Aby zapewnić wysoką dostępność (High Availability) oraz odporność systemu na awarie (Fault Tolerance), w krytycznych węzłach biznesowych zrezygnowano z synchronicznej komunikacji HTTP/REST na rzecz Architektury Sterowanej Zdarzeniami (EDA). Głównym brokerem wiadomości w systemie jest **RabbitMQ**.

Zastosowanie tego wzorca najlepiej obrazuje proces finalizacji transakcji. Kiedy zewnętrzny operator (Stripe) potwierdza płatność za pomocą Webhooka, `Payment-Service` aktualizuje stan w swojej bazie danych, a następnie publikuje zdarzenie `PaymentEvent` na dedykowaną wymianę (Exchange) w RabbitMQ.

Dzięki temu `Payment-Service` natychmiast kończy swoje zadanie i uwalnia zasoby, nie czekając na odpowiedź innych modułów. Zdarzenie to jest asynchronicznie konsumowane m.in. przez `Notification-Service`, który generuje i wysyła wiadomości e-mail oraz SMS do klienta. Takie rozprzęgnięcie usług (Decoupling) gwarantuje, że nawet w przypadku chwilowej awarii zewnętrznej bramki SMS lub przeciążenia serwera e-mail, proces płatności nie zakończy się błędem. Wiadomości cierpliwie poczekają w kolejce (Queue) i zostaną przetworzone natychmiast po przywróceniu sprawności usług powiadomień.



## Wzorzec API Composition (Zarządzanie Rozproszonymi Danymi)

Podział systemu na mikroserwisy wprowadził wyzwanie w postaci rozproszenia danych relacyjnych. Dane o paczkach (waga, status, trasa) znajdują się w bazie **Order-Service**, podczas gdy wrażliwe dane użytkowników i kurierów (imiona, e-maile, przypisane pojazdy) są wyłączną domeną **User-Service**.

Aby uniknąć problemu obciążania aplikacji frontendowej koniecznością odpytywania wielu serwisów i ręcznego łączenia danych (problem *N+1 Selects*), wdrożono architektoniczny wzorzec **API Composition** po stronie backendu.

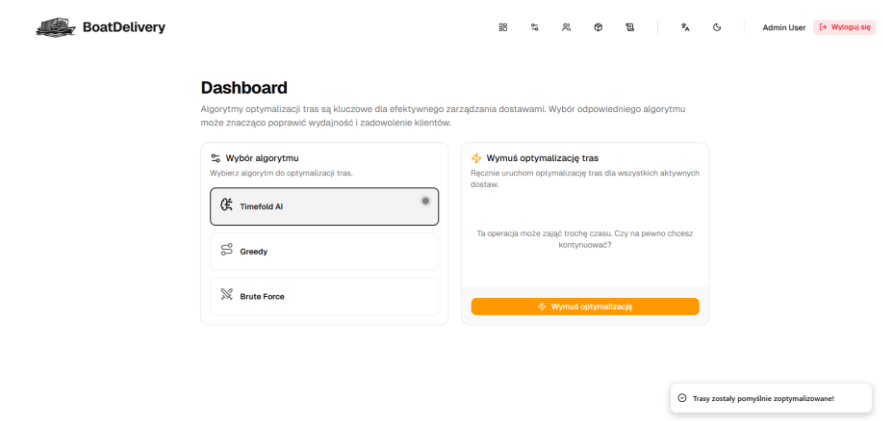
Kiedy administrator żąda wyświetlenia listy zamówień, operacja przebiega następująco:

1. **Order-Service** pobiera paginowaną listę paczek z własnej bazy PostgreSQL.
2. Zanim dane trafią do klienta, **Order-Service** wykonuje błyskawiczne, wewnętrzne zapytanie M2M (Machine-to-Machine) do **User-Service** w celu pobrania danych przypisanych kurierów.
3. Komunikacja ta realizowana jest z użyciem biblioteki **MicroProfile RestClient** i jest bezwzględnie chroniona nagłówkiem **X-Keycloak-Secret** (odrzucającym jakikolwiek ruch z zewnątrz).
4. Pobrane dane użytkowników są mapowane na bezpieczne obiekty DTO (**CourierInfoDTO**) i dynamicznie "wzbogacają" oryginalne encje zamówień.

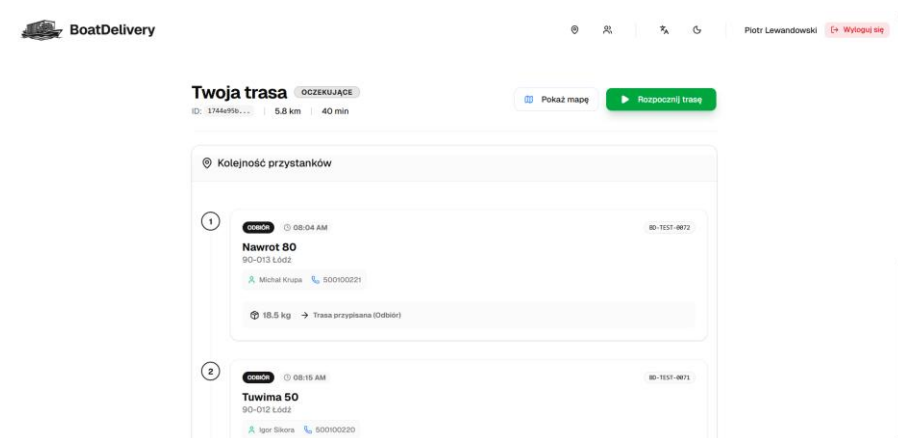
Dzięki temu rozwiązaniu, aplikacja w React otrzymuje w jednym żądaniu HTTP spójny, kompletny i gotowy do wyrenderowania obiekt JSON. Zmniejsza to narzut sieciowy (network overhead), poprawia wydajność interfejsu i ukrywa złożoność architektury mikroservisów przed klientem końcowym.

## Generacja tras dla kurierów przez administratora, główny przepływ systemu BoatDelivery

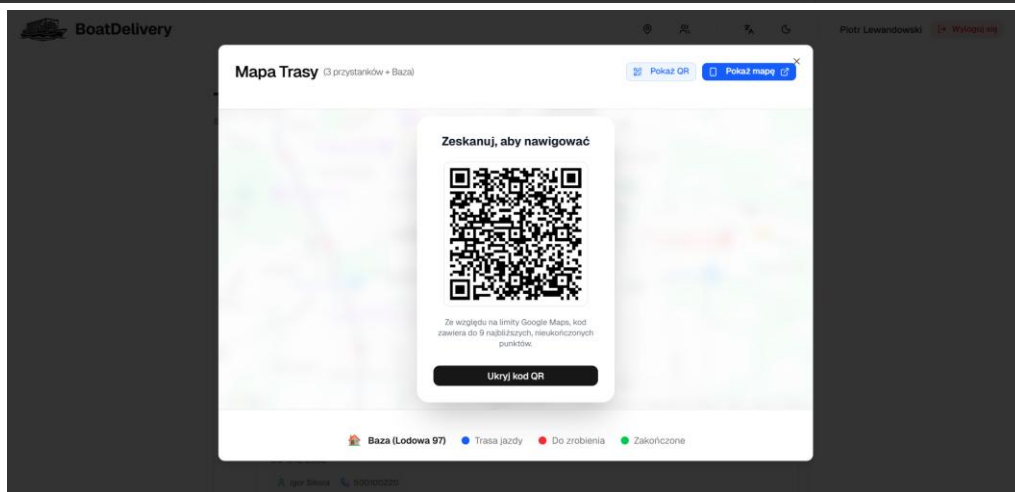
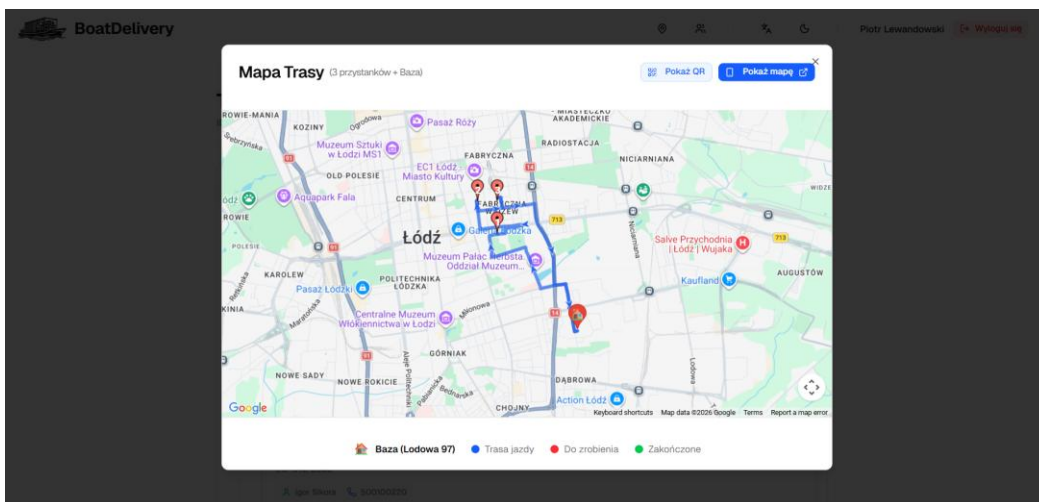
Użytkownik zalogowany z rolą "Administrator" jest w stanie wymusić generację tras. Ma do wyboru jeden z trzech algorytmów.

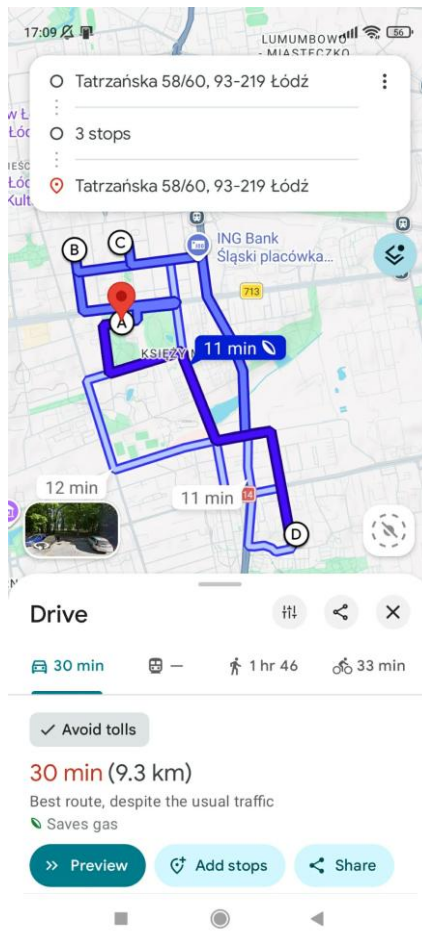


Trasa jest automatycznie przypisywana do kuiera.



Jest on w stanie podejrzeć mapę, zeskanować kod QR, aby mapa trafiła do jego telefonu w aplikacji Google Maps oraz odznaczać zakończone przystanki.





 BoatDelivery

Piotr Lewandowski [Wyloguj się](#)

## Tvoja trasa W TRAKCIE REALIZACJI

ID: 1744e95b... | 5.8 km | 40 min

[Pokaż mapę](#)

[Zakończ trasę](#)

### Kolejność przystanków

1

**ODBIÓR** 08:04 AM

BD-TEST-0072

**Nawrot 80**

90-013 Łódź

[Michał Krupa](#) [500100221](#)

18.5 kg → Zamówienie odebrane od klienta

2

**ODBIÓR** 08:15 AM

BD-TEST-0071

**Tuwima 50**

90-012 Łódź

[Igor Sikora](#) [500100220](#)

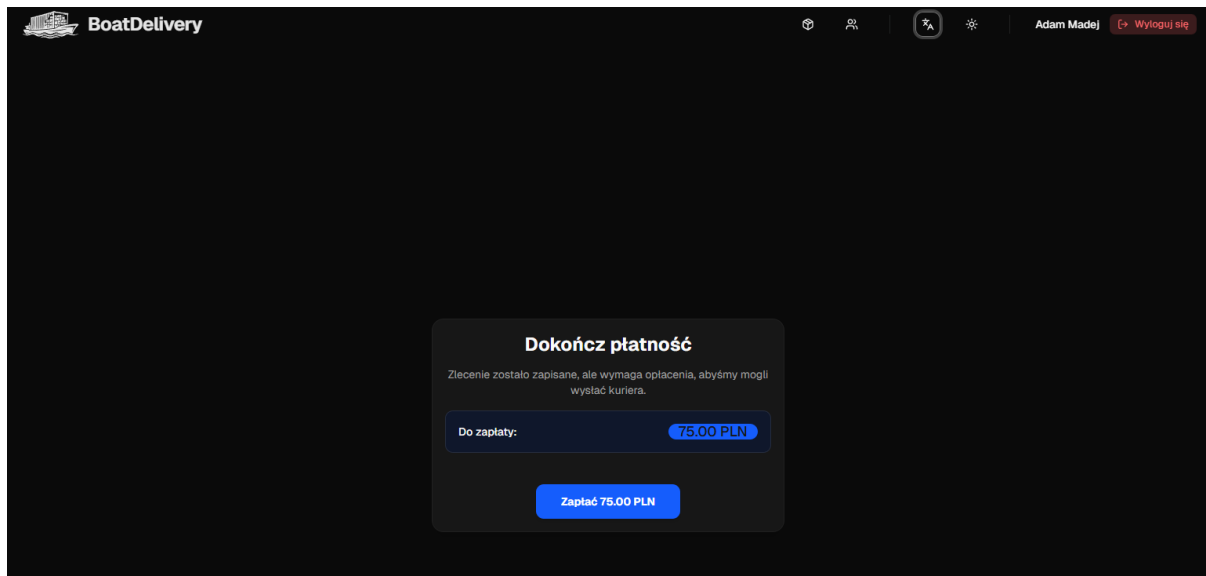
Przystanek oznaczony jako zakończony!

# Cykl życia zdarzeń i komunikacja z Użytkownikiem

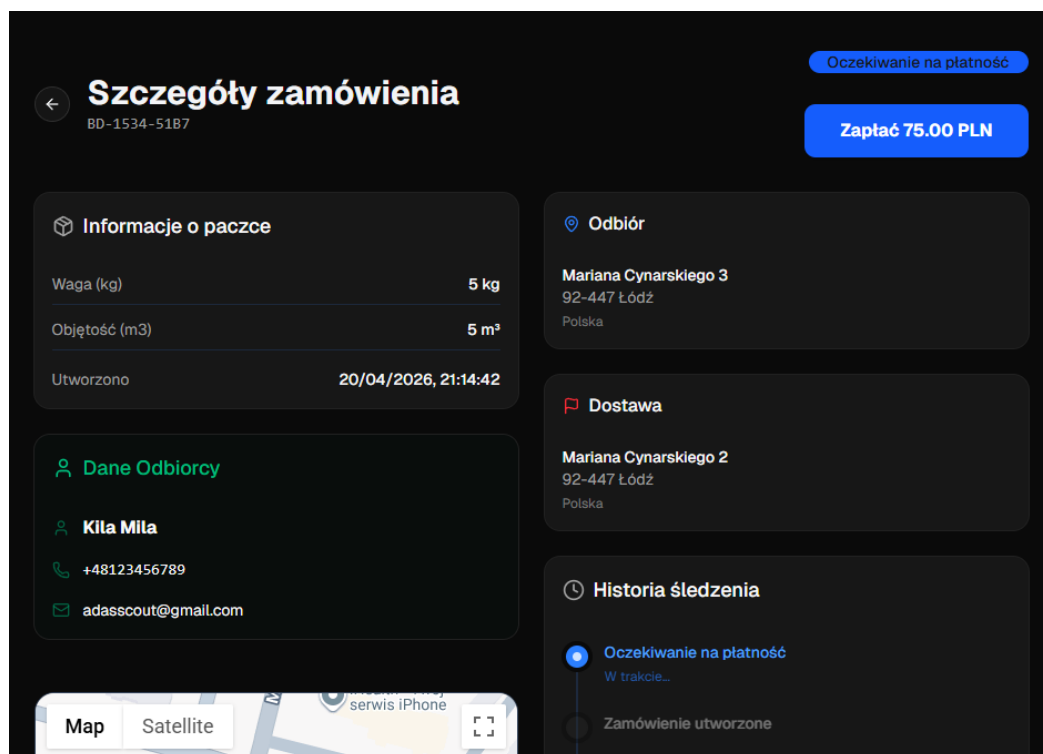
## Scenariusz 1: Inicjacja płatności

Klient, po utworzeniu zamówienia stoi przed najważniejszym krokiem napędzającym BoatDelivery, zapłatą. Musi ona przebiegać idealnie, ale zarazem prosto, aby użytkownik nie miał wątpliwości ILE i ZA CO płaci.

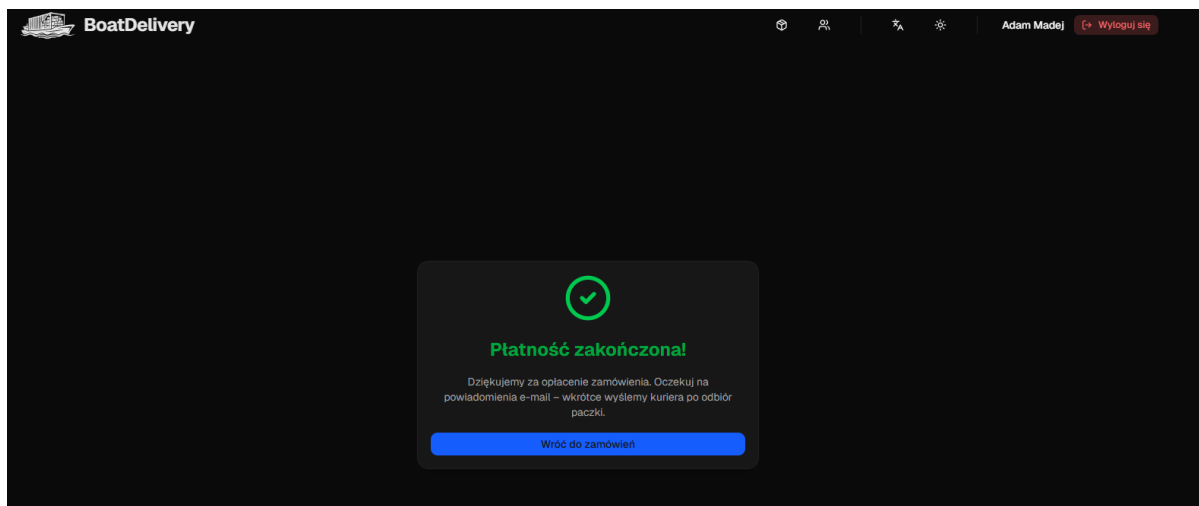
Użytkownik po kliknięciu w przycisk ZAPŁAĆ uruchamia metodę, która generuje bezpieczną sesję płatności i przekierowuje go do bramki Stripe.



Dodatkowo warto nadmienić, że jak każda płatność, może się nie udać. Klient, po kliknięciu w nowo utworzone zamówienia, któremu z różnych powodów nie udało się płatność, ma 24 godziny na opłacenie. Po tym czasie zamówienie zostaje zarchiwizowane w bazie.



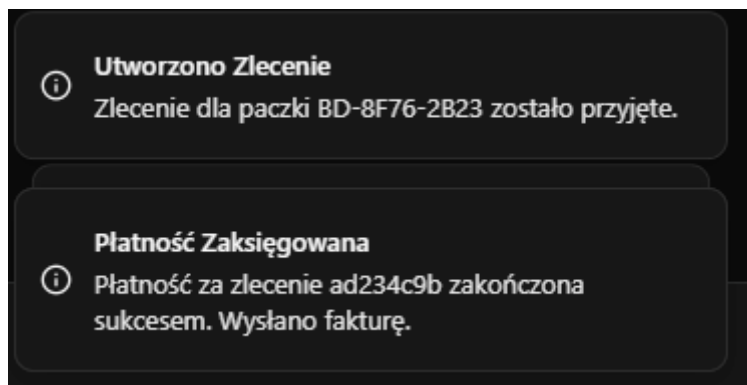
Możliwość ponownej opłaty zamówienia widnieje w szczegółach zamówienia.



Strona wyświetlana po poprawnym zakończeniu zapłaty.

## Scenariusz 2: Asynchroniczna dystrybucja powiadomień

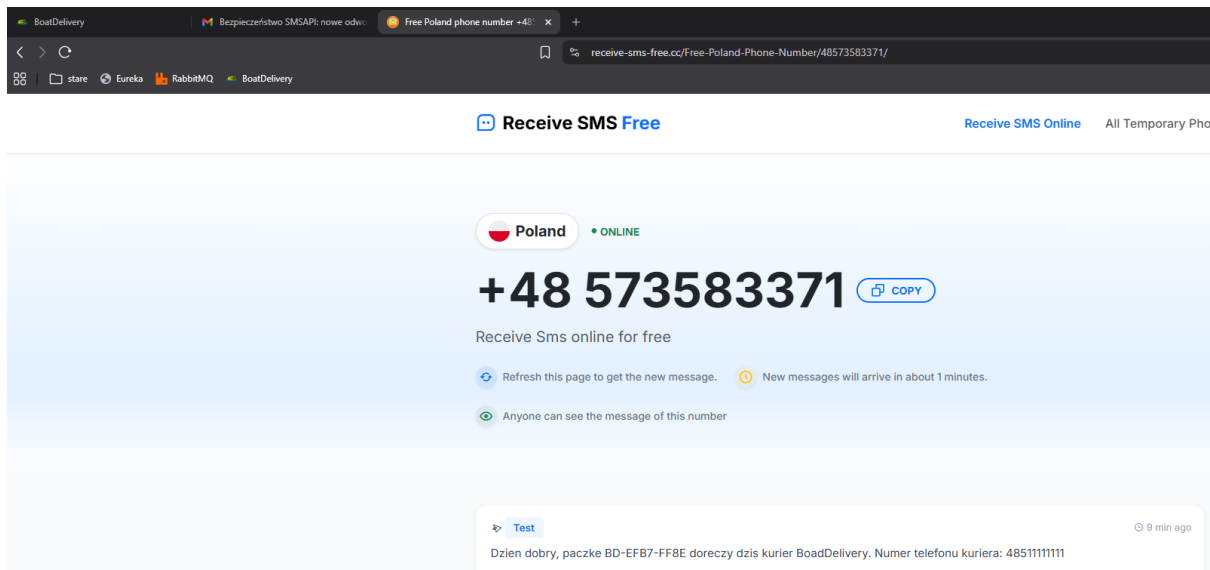
Stripe po weryfikacji karty i obciążeniu jej wysyła asynchroniczny Webhook do Payment-Service. Ten serwis nadaje zdarzenie PaymentEvent do brokera RabbitMQ. Następnie Notification-Service konsumuje tę wiadomość, równolegle uruchamia (zależnie od statusu paczki) trzy niezależne kanały komunikacji, nie blokując wątku głównego (E-MAIL, SSE, SMS)



Powiadomienia jakie wyskakują po opłacie.

kiladotestu	Ktoś nadał do Ciebie paczkę! Nr: BD-8F76-2B23	Cześć Kila, Użytkownik Adam właśnie nadał do Ciebie paczkę nr: BD-8F76-2B23. Poinformujemy Cię, gdy wyr...
kiladotestu	Potwierdzenie płatności i faktura za zlecenie nr: ad234c9b	Cześć, Dziękujemy za opłacenie zlecenia nr ad234c9b. Twoja faktura jest dostępna do pobrania p...
kiladotestu	Zlecenie nadania paczki nr: BD-8F76-2B23	Cześć Adam, Przyjeliśmy Twoje zlecenie nr: BD-8F76-2B23. Oczekuj na przypisanie kuriera, który odbierze paczkę. f...

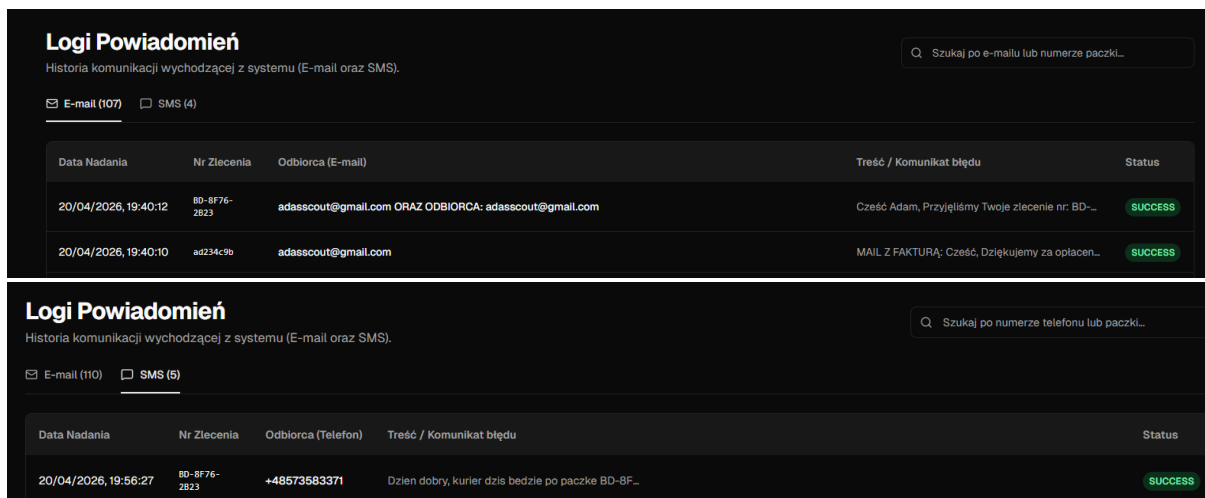
Niebieski – mail do nadawcy paczki (wraz z fakutra), zielony – obiorca.



## Wiadomość SMS

### Scenariusz 3: Audyt i logi administracyjne

Każde powiadomienie (oprócz SSE) jest zapisywane w bazie powiadomień. Pozwala ona weryfikować, czy wiadomość dotarła administratorowi.



Zrzut ekranu panelu administratora (Historia powiadomień)

## Pozyskiwanie kluczy API

Aplikacja używa kilka zewnętrznych serwisów, bez tych kluczy do komunikacji z nimi nie jest możliwe uruchomienie aplikacji.

### SMS API

Program korzysta z polskiego właściciela [BRAMKI SMS](#)

1. Rejestrujemy się na stronie
2. Po udanej rejestracji i logowaniu przechodzimy do:  
<https://ssl.smsapi.pl/react/oauth/manage>
3. Klikamy **Generuj token**
4. Uzupełniamy dane według schematu:

Dodaj Token

Nazwa Tokena

Tvoja Nazwa Tokenu

Data wygaśnięcia tokena

☒ Nigdy

☐ Ustaw datę

☒ Czarna lista

☒ Adresy callback

☒ Kontakty

☒ HLR

☒ MFA

☒ MMS

☒ Profil użytkownika

☒ RCS

☒ Skrócone linki

☒ SMS

☒ Pola nadawcy SMS

☒ Użytkownicy

☒ VMS

ZamknijGeneruj token

5. Wygenerowany token wklejamy do BoatDelivery\notifi-service\.env.docker



```
SPRING_MAIL_HOST=smtp.gmail.c
SPRING_MAIL_PORT=587
SPRING_MAIL_USERNAME=kiladote
SPRING_MAIL_PASSWORD=vkas ama
SMSAPI_TOKEN=TU WKLEJ TOKEN
SPRING_RABBITMQ_HOST=rabbitmq
```

### UWAGI

Przy rejestracji trzeba podać numer telefonu, na który **TYLKO** będzie można wysłać SMS.

### STRIPE

System płatności jest obsługiwany przez popularną platformę [STRIPE](#).

1. Rejestrujemy się na platformie.
2. Po udanej rejestracji, logujemy się.
3. Otwiera nam się panel główny, po prawo mamy dwa klucze API.
4. Kopiujemy SECRET KEY i wklejamy do BoatDelivery\payment-service\.env.docker

```
JWT_JWK_SET_URI=http://keycloak:8060/realms/bc
SPRING_DATASOURCE_URL=jdbc:postgresql://postgr
SPRING_RABBITMQ_HOST=rabbitmq
EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=http://eu
STRIPE_SECRET_KEY=TU WKLEJ SECRET KEY
STRIPE_WEBHOOK_SECRET=TU WKLEJ WEBHOOK SECRET
```

Z uwagi na to, że serwery Stripe nie mają bezpośredniego dostępu do mikrousług uruchomionych w lokalnej sieci wirtualnej (localhost), do testowania płatności asynchronicznych wykorzystujemy skonteneryzowane narzędzie Stripe CLI. Działa ono jako niezależna usługa (kontener) w naszej infrastrukturze Docker Compose, automatycznie tworząc bezpieczny tunel (forwarding) i kierując ruch bezpośrednio do kontenera payment-service.

5. Uruchamiamy stripe-cli-forwarder w dockerze.
6. Dostaniemy informacje o potrzebie autoryzacji:

```
You have not configured API keys yet. Running `stripe login`...
Your pairing code is: best-chaste-feat-talent
This pairing code verifies your authentication with Stripe.
To authenticate with Stripe, please go to: https://dashboard.stripe.com/stripecli/conf
Waiting for confirmation...
```

7. Klikamy w link, który otwiera przeglądarkę. Logujemy się, wpisujemy kod z maila.
8. Po udanej autoryzacji powinniśmy otrzymać webhook secret key:

```
Getting ready...
Ready! You are using Stripe API Version [2026-02-25.clover]. Your webhook signing secret is
whsec_2dadd9683abc0772fd9bd39242b [REDACTED] (^C to quit)
```

9. Kopiujemy ten klucz i wklejamy w to samo miejsce co Secret Key, tylko do linijki STRIPE\_WEBHOOK\_SECRET.

## UWAGI

Do płatności używamy testowej karty:

Numer karty: 4242424242424242

CVC: każde 3 cyfry

Data: każda przyszła data

Warto dodać, że Stripe oferuje więcej testowych kart, zależnych czego potrzebujemy [KARTY](#)



## HERE API


1. Rejestracja <https://platform.here.com/portal/sign-up?step=verify-identity&aid=www.here.com-top-nav-start-building>

2. Niestety trzeba podać kartę już na tym etapie i wybrać subskrypcję :/

### Payment Information

You won't be charged unless you exceed free transaction tiers, as defined on our [pricing page](#).

**Credit card**  
 

**Paypal**  


Card Number

Expiration Date (YYYY)

- Select One -

 / 

- Select One -

CVV

Cardholder Name

Contact Phone Number

Email Address

[Need more info? Contact us](#)

[Do it later](#)

[Save and review details](#)

3. Generujemy klucz


## Find the right tools for your location

Start by registering your app to get credentials. Then

Maps API for JS

**REST APIs**

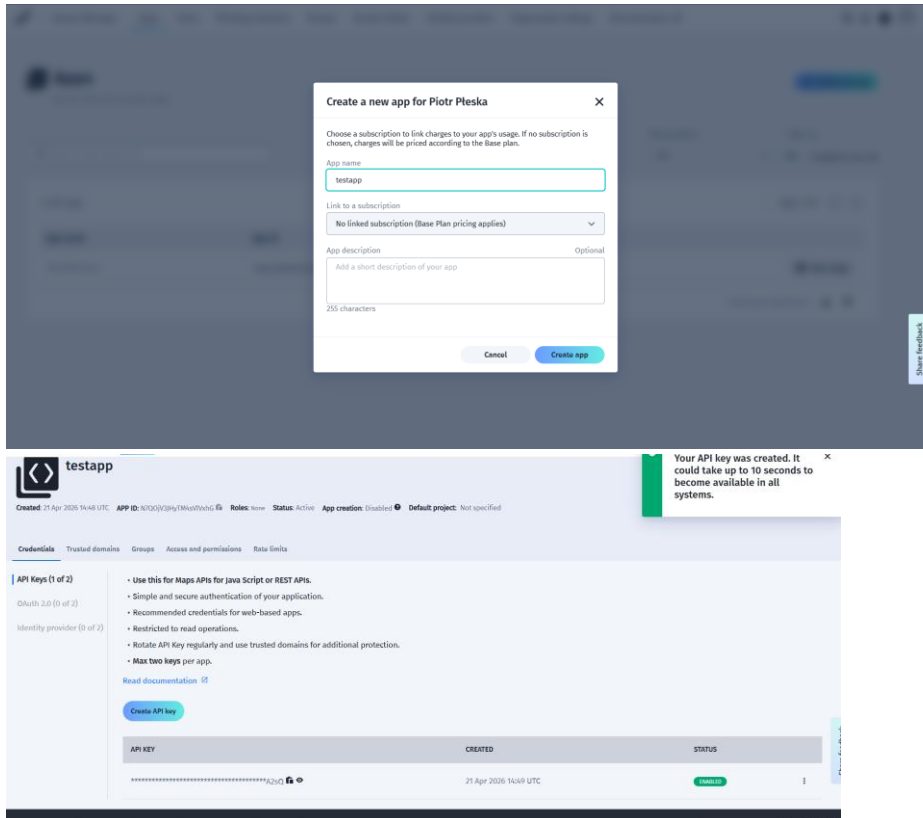
SDKs



### Add and manage apps

Register your app to get your App ID and API key, or manage your credentials. You can do both in the Access Manager.

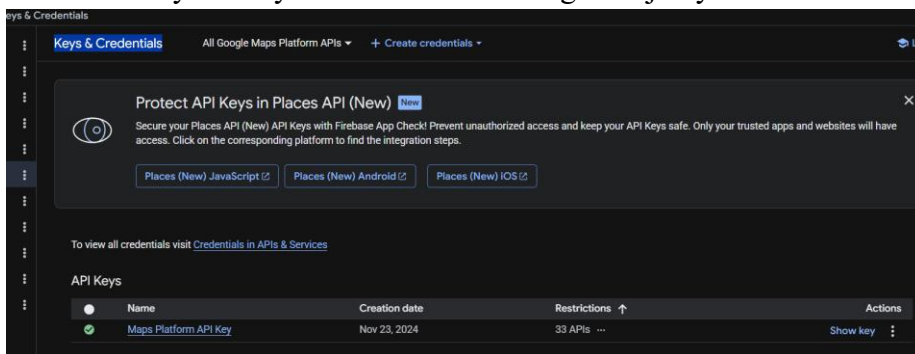
[→ Go to Access Manager](#)



4. Wklejamy w <https://github.com/TomaszKK/BoatDelivery/blob/master/order-service/src/main/resources/application.properties#L55> `here.api.key=`

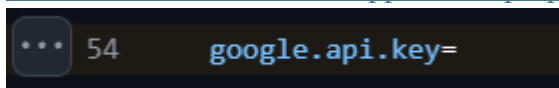
## Google Maps API

1. Rejestracja <https://mapsplatform.google.com/lp/maps-apis/>
2. Dodajemy kartę bankową (polecam Revolut)
3. Włączamy
  - a. Directions API
  - b. Roads API
  - c. Distance Matrix API
4. Przechodzimy do Keys & Credentials oraz generujemy klucz



5. Klucz wklejamy w dwa miejsca:

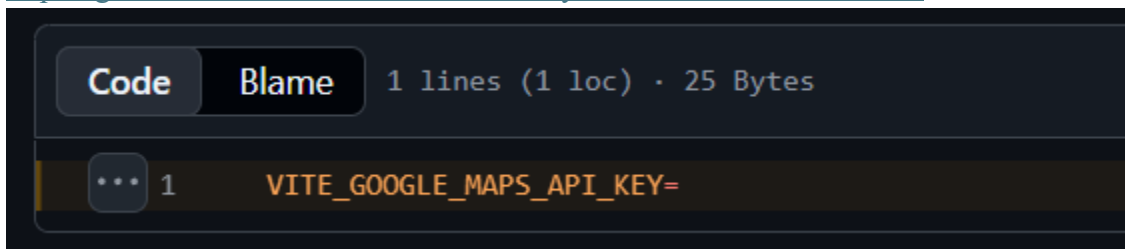
<https://github.com/TomaszKK/BoatDelivery/blob/master/order-service/src/main/resources/application.properties#L54>



```
... 54 google.api.key=
```

oraz

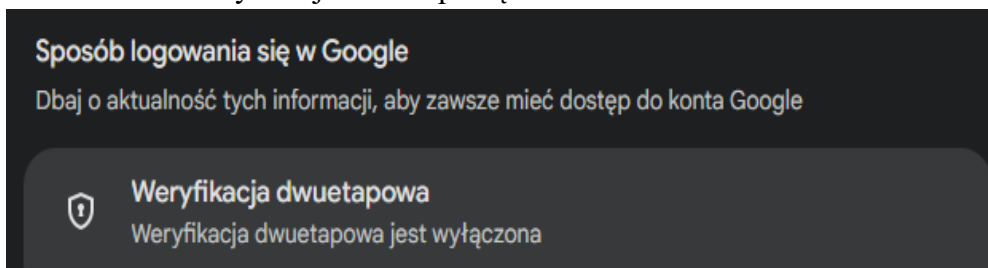
<https://github.com/TomaszKK/BoatDelivery/blob/master/front/.env#L1>



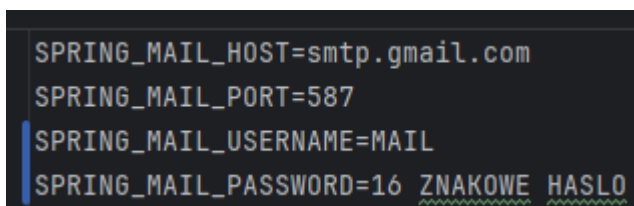
```
Code Blame 1 lines (1 loc) · 25 Bytes
... 1 VITE_GOOGLE_MAPS_API_KEY=
```

## *GMAIL SMPT*

1. Rejestrujemy konto google (najlepiej na gmailu od razu).
2. Przechodzimy do [myaccount.google.com/security](https://myaccount.google.com/security).
3. Trzeba dodać weryfikację dwuetapową:



4. Następnie wchodzimy w Hasła Aplikacji
5. Wpisujemy identyfikator i generujemy 16 znakowy kod.
6. Następnie wchodzi do BoatDelivery\notifi-service\.env.docker i wpisujemy je.



```
SPRING_MAIL_HOST=smtp.gmail.com
SPRING_MAIL_PORT=587
SPRING_MAIL_USERNAME=MAIL
SPRING_MAIL_PASSWORD=16 ZNAKOWE HASLO
```

## *Keycloak Mail SMTP*

Aby ustawić maila do aktywacji kont po rejestracji trzeba zmienić w ustawieniach keycloaka dane do maila z którego były by wysyłane – realm-export.json lub bezpośrednio z poziomu panelu admina Keycloaka

```
"smtpServer": {  
  "allowutf8": "",  
  "replyToDisplayName": "",  
  "debug": "false",  
  "starttls": "true",  
  "auth": "true",  
  "writeTimeout": "10000",  
  "envelopeFrom": "",  
  "ssl": "false",  
  "timeout": "10000",  
  "password": "*****",  
  "port": "587",  
  "replyTo": "",  
  "host": "smtp.gmail.com",  
  "from": "kiladotestu@gmail.com",  
  "fromDisplayName": "Boat Delivery - access",  
  "authType": "basic",  
  "connectionTimeout": "10000",  
  "user": "kiladotestu@gmail.com"
```

## Budowanie aplikacji

Projekt został stworzony w jednym repozytorium, które jest postawione na docker-compose. Dzięki czemu w łatwy sposób można zbudować i uruchomić cały system poprzez komendę:

**docker-compose up -d --build**

W głównym katalogu projektu.