Uniwersytet Rzeszowski

Informatyka, IIr. IIst. Stacjonarne, sem. Letni

Programowanie Aplikacji Webowych Opartych o Mikrousługi

Projekt:

System telemetryczny dla Formuły 1 z obsługą alertów

Grupa Laboratoryjna 1

Maciej Biel (117780)

Spis treści

Wstęp

W niniejszym projekcie zaprojektowano oraz zaimplementowano system służący do **monitorowania danych telemetrycznych** pochodzących z bolidów wyścigowych Formuły 1, a także do **wykrywania i zarządzania alertami** na podstawie zdefiniowanych reguł inżynierskich.

Rysunek 1. Max Verstappen w bolidzie RB21 podczas Grand Prix w Bahrainie 2025.

Projekt został stworzony z myślą o **inżynierach wyścigowych zespołów F1**, gdzie precyzyjne dane telemetryczne i szybka analiza są niezbędne do podejmowania decyzji podczas sesji wyścigowych.

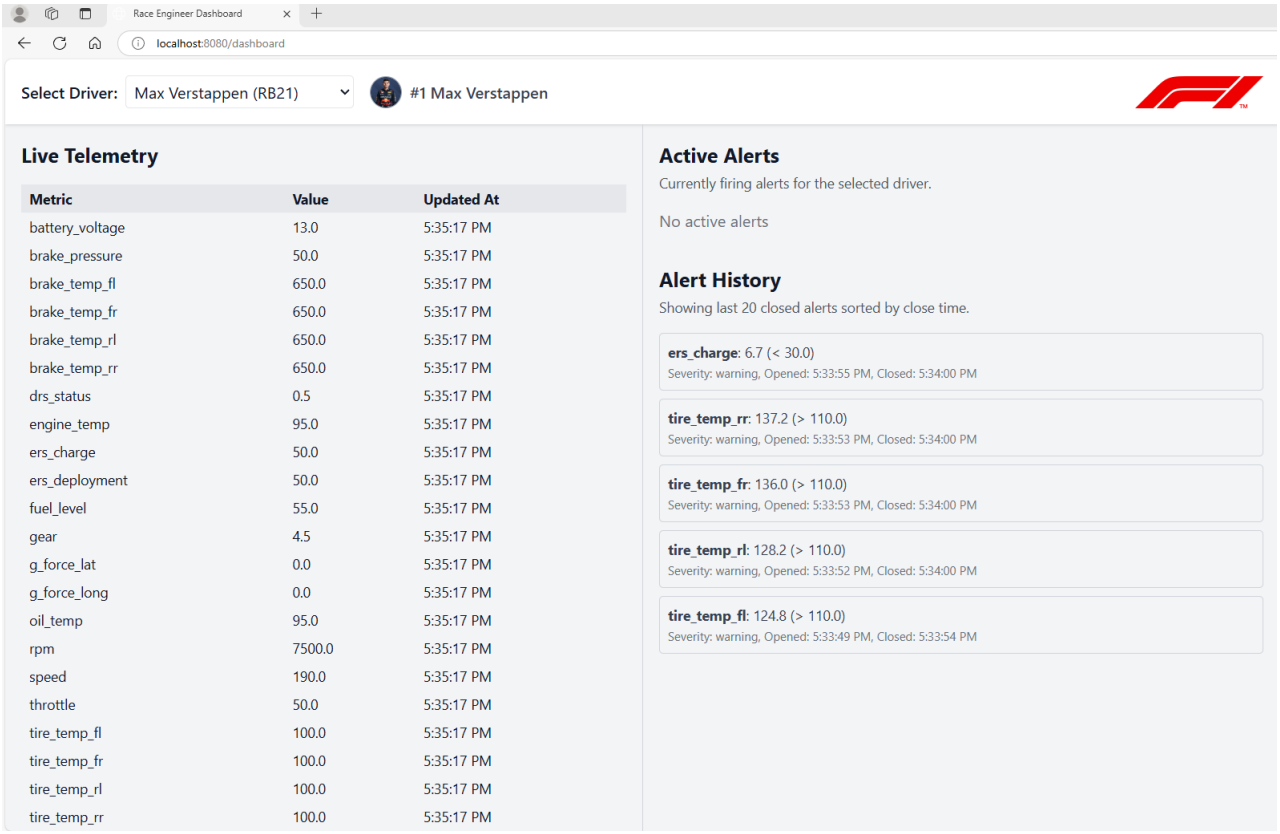
System umożliwia inżynierowi wyścigowemu monitorowanie w czasie rzeczywistym kluczowych parametrów bolidu, **takich jak temperatura opon, ciśnienie hamulców, poziom paliwa, stan systemów ERS/DRS** itp. W przypadku wykrycia wartości przekraczających bezpieczne progi, system generuje alerty, umożliwiając natychmiastowe działania naprawcze lub strategiczne.

Prezentacja systemu

Wyobraźmy sobie, że jesteśmy inżynierem wyścigowym zespołu **Red Bull Racing**, odpowiedzialnym za monitorowanie danych telemetrycznych kierowcy bolidu numer 1, czyli **Maxa Verstappena** podczas trwających mistrzostw (Grand Prix).

Naszym zadaniem jest nie tylko śledzenie na bieżąco parametrów technicznych bolidu, ale także szybka reakcja w przypadku pojawienia się zagrożeń – np. przegrzewających się hamulców lub niskiego poziomu energii ERS.

Do tego zadania wykorzystamy dedykowany **Dashboard**, który w czasie rzeczywistym prezentuje dane napływające z bolidu wybranego obecnie kierowcy.



Rysunek 2. Dashboard z danymi dostępnymi dla inżyniera wyścigowego

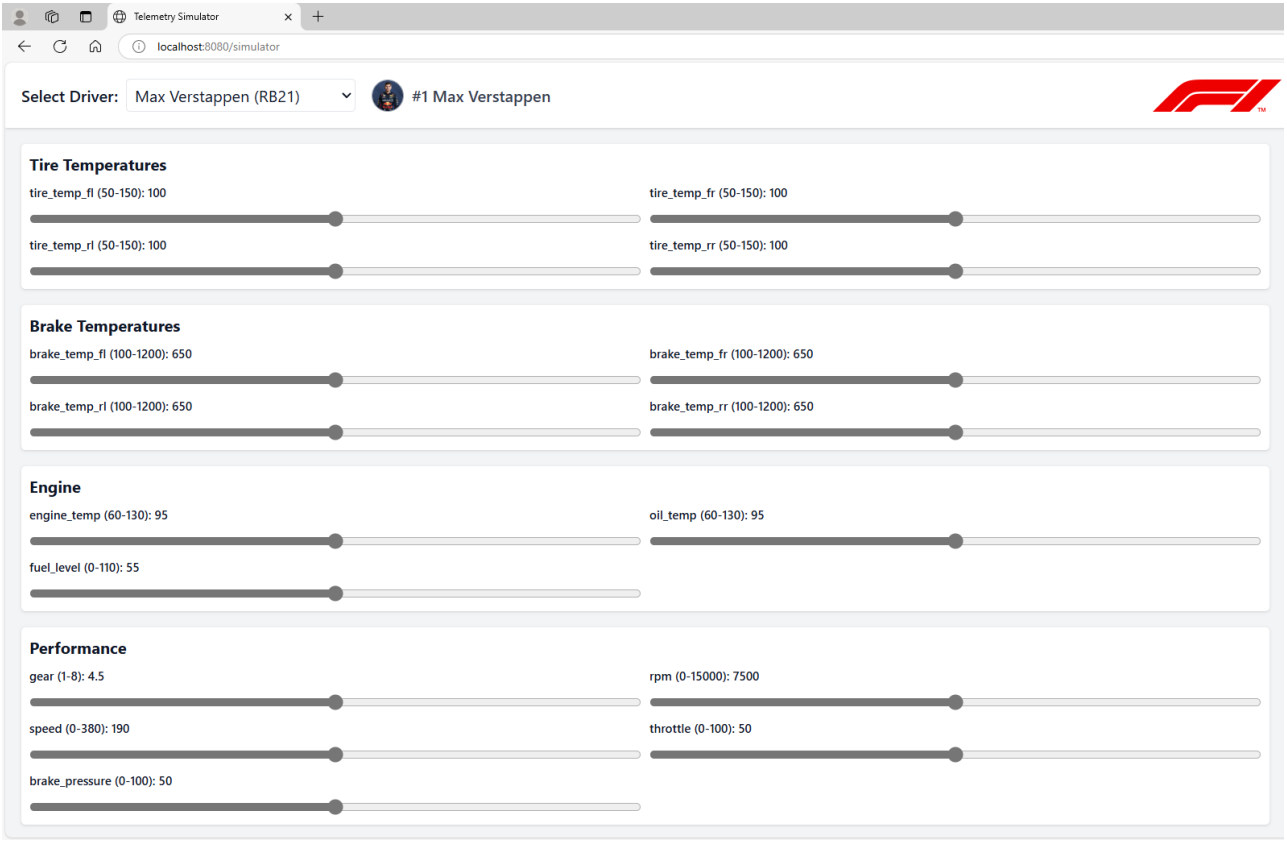
Dashboard umożliwia:

- Wybór kierowcy na torze (np. Max Verstappen).
- Podgląd na żywo wszystkich metryk technicznych.
- Przegląd aktywnych alertów oraz historii ostrzeżeń.

Symulacja jazdy bolidem na torze na żywo

Dane telemetryczne przesyłane do systemu generowane są w osobnej aplikacji symulacyjnej **Simulator**, która imituje bolid poruszający się po torze wyścigowym.

W ramach demonstracji, symulator umożliwia ręczne sterowanie wybranymi metrykami technicznymi pojazdu danego kierowcy, dotyczącymi temperatury opon i hamulców, silnika, wydajności i systemów DRS/ERS, które następnie są automatycznie raportowane do systemu **co 1 sekundę**.

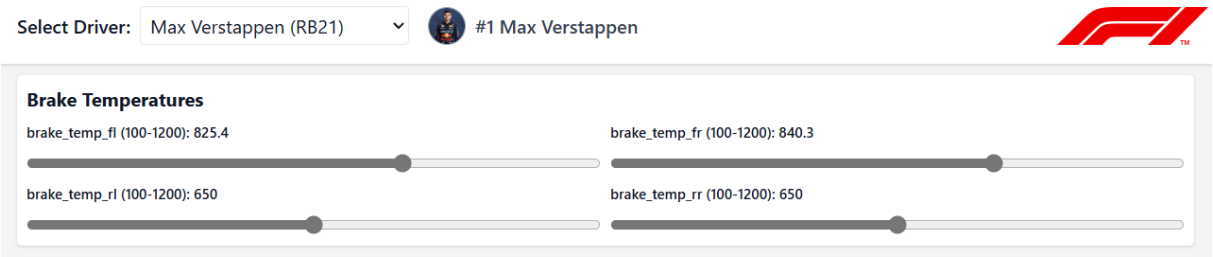


Rysunek 3. Za pomocą suwaków możemy zmieniać na bieżąco raportowane metryki.

Wszystkie dane telemetryczne są reprezentowane jako zestaw etykiet (nazw metryk) wraz z odpowiadającymi im wartościami liczbowymi, które są stale raportowane do systemu.

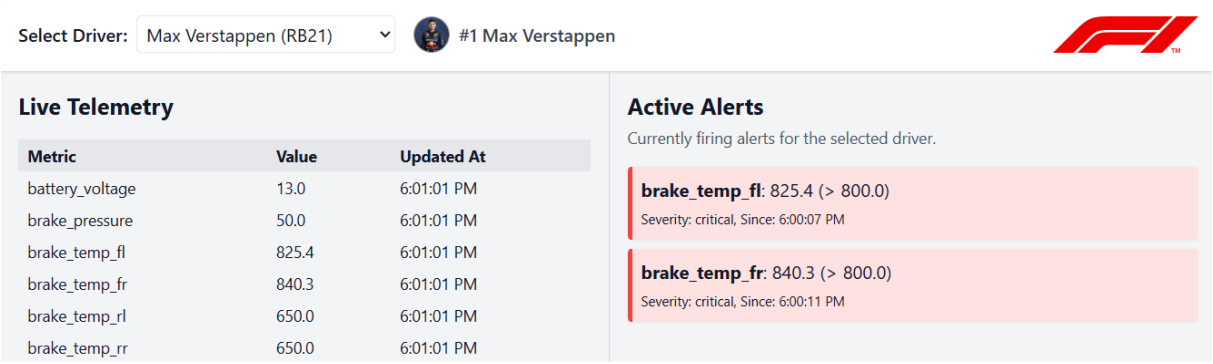
Metryki i Alerty na żywo

Załóżmy, że w trakcie wyścigu wartość temperatury przednich hamulców osiąga **ponad 820°C**, co przekracza bezpieczny próg **800°C**.



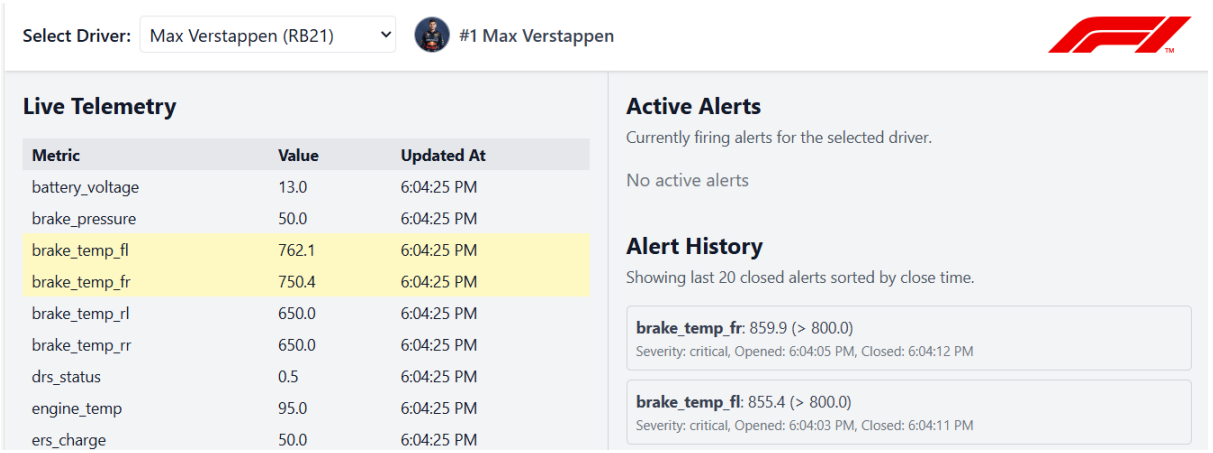
Rysunek 4. W przypadku torów np., w Arabii Saudyjskiej nie jest to rzadka sytuacja.

Od rzazu w **Dashboardzie** pojawiają się **dwa nowe alerty** informujące o przegrzewaniu się przednich hamulców. Inżynier obserwując sytuacje, może szybko podjąć działania, komunikując się radiowo z kierowcą i prosząc o czysty przejazd i zmniejszenie tempa.



Rysunek 5. Dwa krytyczne alerty widoczne są w prawym panelu.

Po krótkiej chwili temperatury spadają poniżej progu, a alert automatycznie znika z listy aktywnych i przenosi się do sekcji ostatnich alertów w historii.

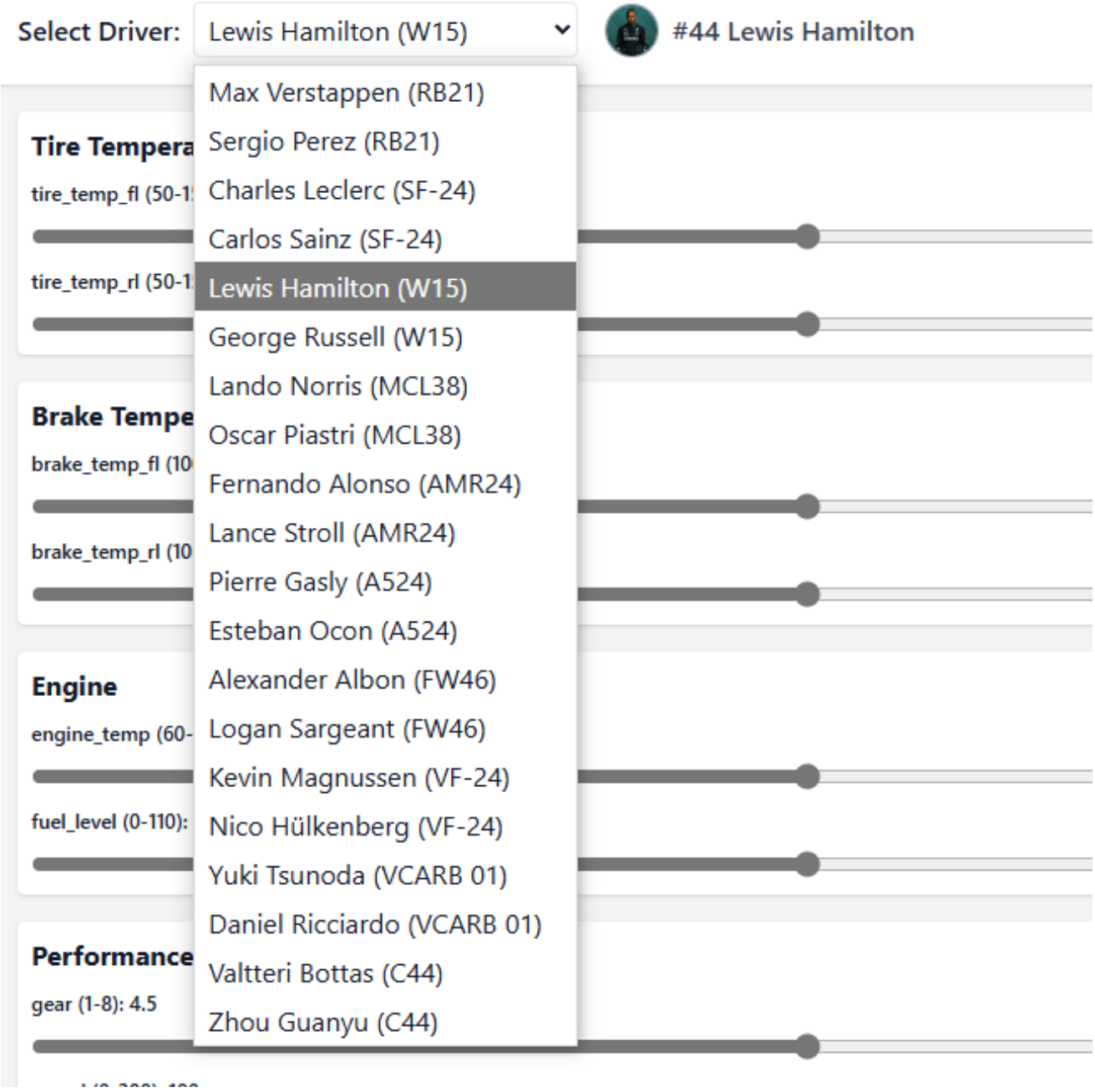


Rysunek 6. Jeżeli metryki ulegają zmianie, są przez sekundę podświetlane na żółto w tabeli.

Kierowcy na torze

Zarówno w **Dashboardzie**, jak i w **Symulatorze** dostępna jest możliwość wyboru jednego z 20 kierowców biorących udział w wyścigu.

Każdy z nich przypisany jest do konkretnego zespołu i bolidu zgodnie z sezonem 2024. System pozwala równolegle symulować jazdę dwóch bolidów, **generując osobne dane** telemetryczne dla każdego z nich i odbierając je na dwóch niezależnych stanowiskach inżynierów wyścigowych.



Rysunek 7. Wybór kierowców jeżdżących obecnie po torze.

Opis działania systemu

System składa się z pięciu głównych komponentów:

- Trzech mikroserwisów backendowych jako API (**Telemetry**, **Rules**, **Alerts**)
- Dwóch aplikacji frontendowych (**Dashboard**, **Simulator**)

Serwisy współpracują w ramach jednej architektury opartej na kontenerach. Całość została zaprojektowana w modelu mikroserwisowym, z użyciem centralnego mechanizmu routingu (**Traefik**) oraz asynchronicznej komunikacji z wykorzystaniem **RabbitMQ**.

Główne komponenty systemu:

- **Simulator** (*frontend*):
    - Prosta aplikacja webowa, która pozwala ręcznie sterować wartościami metryk telemetrycznych (np. temperatura opon, poziom paliwa). Co sekundę przesyła dane do systemu, symulując zachowanie bolidu na torze.
- **Telemetry Service** (*backend*):
    - Przechowuje dane telemetryczne oraz dane samochodów (kierowców). Każda nowa metryka otrzymana z symulatora jest zapisywana w bazie danych, a następnie publikowana jako wiadomość do RabbitMQ.
- **Rules Service** (*backend*):
    - Umożliwia zarządzanie regułami dla metryk telemetrycznych. Subskrybuje dane telemetryczne z RabbitMQ i ocenia je na podstawie wcześniej zdefiniowanych reguł – w przypadku spełnienia warunku, wysyła zdarzenie naruszenia reguły.
- **Alerts Service** (*backend*):
    - Odbiera zdarzenia o naruszeniach reguł z RabbitMQ i na ich podstawie tworzy alerty. Odpowiada za ich stan (aktywny/zamknięty), czas wyzwolenia oraz historię.
- **Dashboard** (*frontend*):
    - Interfejs inżyniera wyścigowego, w którym prezentowane są najnowsze dane telemetryczne oraz aktywne i historyczne alerty. Dane są odświeżane w czasie rzeczywistym.

Uruchomienie aplikacji

Aplikacja została załączona do zadania jako archiwum `f1-telemetry.zip`, zawierające kompletne **monorepozytorium** projektu.

Wymagania wstępne:

- Zainstalowany **Docker** oraz Docker Compose
- Dostęp do portów **8080** (aplikacja) i **8081** (Traefik dashboard)

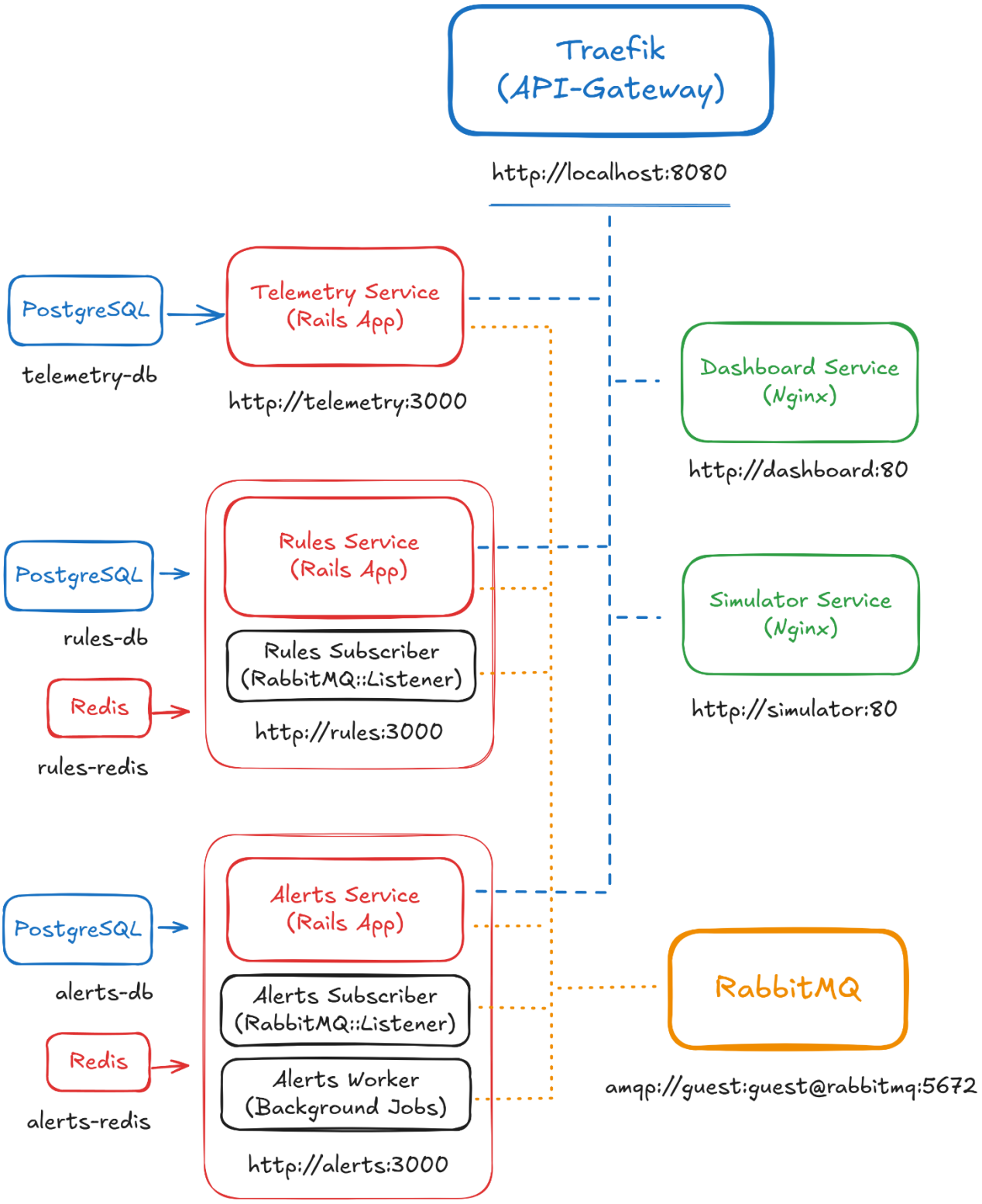
Sposób uruchomienia:

- Rozpakowanie repozytorium i wejście do katalogu projektu: `cd f1-telemetry`
- Uruchomienie wszystkich usług: `docker-compose up --build`
- Pierwsze uruchomienie może potrwać kilka minut ze względu na budowanie obrazów oraz migracje baz danych:
    - Bazy danych w aplikacji zostały w taki sposób zaprojektowane i zamodelowane w `docker-compose.yml`, że każdy restart kontenerów z mikroserwisami backendowymi powoduje **reset danych w aplikacji i ponowne zaseedowanie** minimalnym zestawem danym potrzebnym do działania. Dzięki temu aplikacja jest od razu gotowa do użycia.

Dostęp do aplikacji:

- **Traefik** (API Gateway):
    - `http://localhost:8080` – punkt wejścia do wszystkich usług.
    - `http://localhost:8081` – panel administracyjny Traefika z listą routerów.
- **Frontend**:
    - `http://localhost:8080/simulator` – symulator bolidu.
    - `http://localhost:8080/dashboard` – panel inżyniera wyścigowego.
- **RabbitMQ** Panel (debugowanie):
    - `http://localhost:15672` - (`guest:guest`).

Diagram systemu



Rysunek 8. Diagram systemu przygotowany w narzędziu https://excalidraw.com/

Szczegółowy opis systemu i architektury

Zastosowane technologie

Ruby on Rails

Ruby on Rails to framework stworzony w języku programowania **Ruby**, przeznaczony do szybkiego prototypowania i tworzenia aplikacji webowych, oferujący bogaty zestaw narzędzi, dojrzałe i aktywne community oraz szerokie wsparcie w postaci rozwiniętych bibliotek i gotowych rozwiązań. Framework ten został wykorzystany do budowy **wszystkich mikroserwisów** backendowych.

Dodam od siebie, że **Ruby on Rails**, choć nie jest frameworkiem tak popularnym ani powszechnie stosowanym na tle innych rozwiązań, to został przeze mnie **wybrany świadomie** – wykorzystuję go zawodowo od ponad dwóch lat z sukcesem, co pozwoliło mi sprawnie i przyjemnie zrealizować ten projekt.

Docker & Docker Compose

Technologie konteneryzacji umożliwiające uruchomienie całego systemu w spójnym, izolowanym środowisku. Docker Compose zarządza konfiguracją wszystkich mikroserwisów i komponentów infrastrukturalnych w jednym pliku.

Traefik

Reverse proxy i **API Gateway**, który pełni funkcję routingu żądań HTTP do odpowiednich mikroserwisów oraz automatycznego wykrywania usług
(**Service Discovery**) na podstawie etykiet kontenerów Docker.

RabbitMQ

System kolejkowania wiadomości, umożliwiający asynchroniczną komunikację między mikroserwisami. Zapewnia niezależność i elastyczność w przekazywaniu zdarzeń telemetrycznych, wyników oceny reguł i alertów.

PostgreSQL

Relacyjna baza danych wykorzystywana do trwałego przechowywania danych w każdym mikroserwisie backendowym. Każdy serwis posiada własną, **niezależną instancję** bazy PostgreSQL, co zapewnia izolację danych i zgodność z założeniami architektury mikroserwisowej.

Redis

Baza danych typu key-value, wykorzystywana w Rules Service i Alerts Service do przechowywania danych tymczasowych (cache) oraz wspomagania zadań przetwarzanych cyklicznie (np. wygaszanie alertów).

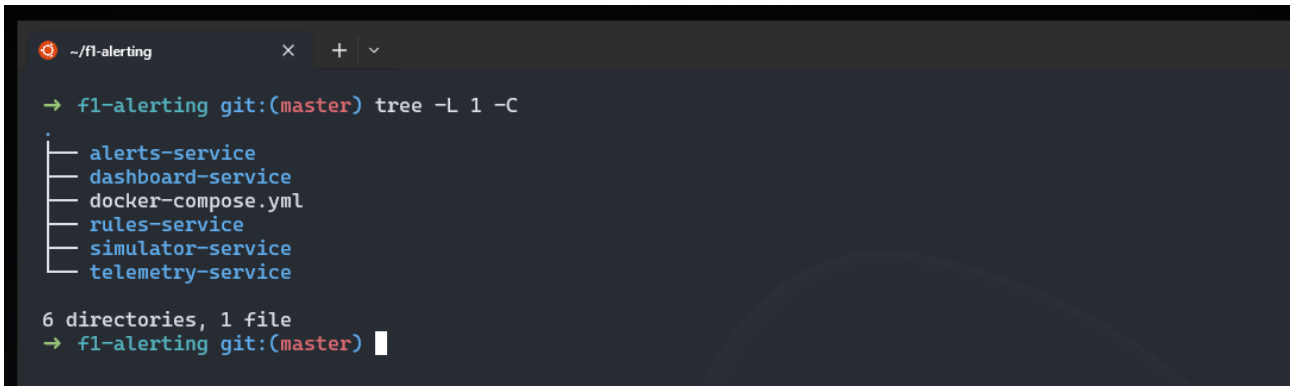
NGINX

Lekki serwer HTTP używany do hostowania statycznych plików aplikacji front-endowych (Dashboard i Simulator) w środowisku Docker.

HTML, Tailwind CSS, JavaScript

Technologie front-endowe wykorzystane do budowy interfejsów użytkownika – Dashboardu oraz Simulatora. Tailwind CSS zapewnia szybkie stylowanie, a JavaScript obsługuje dynamiczne pobieranie i przesyłanie danych przez API.

Omówienie struktury projektu



Rysunek 9. Widok monorepozytorium z poziomu terminala.

Projekt został zorganizowany jako **monorepozytorium**, w którym wszystkie mikroserwisy backendowe, aplikacje frontendowe oraz pliki konfiguracyjne znajdują się w jednym repozytorium i mogą być uruchamiane wspólnie za pomocą Docker Compose.

Zawartość repozytorium:

- **telemetry-service** - Mikroserwis zbierający i zapisujący dane telemetryczne pochodzące z bolidów. Udostępnia API do odczytu i zapisu metryk.
- **rules-service** - Mikroserwis odpowiedzialny za ocenę reguł przypisanych do metryk telemetrycznych. Subskrybuje dane i generuje zdarzenia o przekroczeniu progów określonych w regułach.
- **alerts-service** - Mikroserwis odpowiedzialny za tworzenie i zarządzanie alertami na podstawie zdarzeń otrzymywanych z systemu.
- **dashboard-service** - Aplikacja frontendowa przeznaczona dla inżyniera wyścigowego, umożliwiająca podgląd danych telemetrycznych oraz alertów w czasie rzeczywistym.
- **simulator-service** - Aplikacja frontendowa symulująca jazdę bolidu, pozwalająca ręcznie sterować wartościami metryk i przesyłać je do systemu.
- **docker-compose.yml** – Główny plik konfiguracyjny definiujący wszystkie kontenery potrzebne do uruchomienia systemu, w tym mikroserwisy, bazy danych i serwisy związane z infrastrukturą.

docker-compose.yml – omówienie

Cały system uruchamiany jest za pomocą pliku **docker-compose.yml**, który definiuje usługi infrastrukturalne, mikroserwisy backendowe, aplikacje frontendowe oraz powiązane bazy danych.

Sekcja: Infrastruktura systemu

`gateway` (Traefik)

```
Run Service
gateway:
  image: traefik:v3.0
  command:
    - "--log.level=DEBUG"
    - "--api.insecure=true"
    - "--providers.docker=true"
    - "--providers.docker.exposedbydefault=false"
    - "--entryPoints.web.address=:80"
  ports: [ "8080:80", "8081:8080" ]
  volumes: [ "/var/run/docker.sock:/var/run/docker.sock:ro" ]
```

Rysunek 10

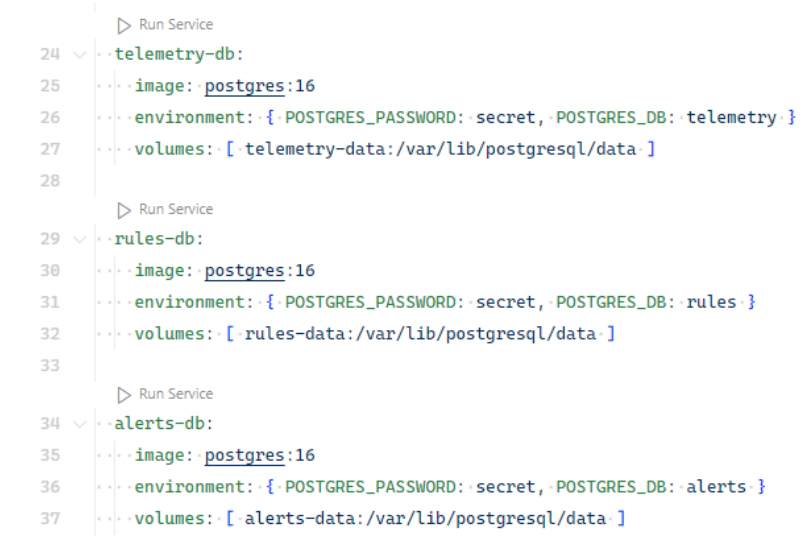
- Pełni funkcję **API Gateway** oraz **Service Discovery**.
- Odbiera żądania HTTP na porcie **8080** i przekazuje je do odpowiednich mikroserwisów lub frontendów na podstawie zdefiniowanych ścieżek URL.
- Konsola administracyjna jest dostępna na porcie **8081.**

`rabbitmq`

```
Run Service
rabbitmq:
  image: rabbitmq:3-management
  ports: [ "15672:15672" ]
  healthcheck:
    test: ["CMD", "rabbitmq-diagnostics", "-q", "status"]
    interval: 3s
    timeout: 10s
    retries: 5
```

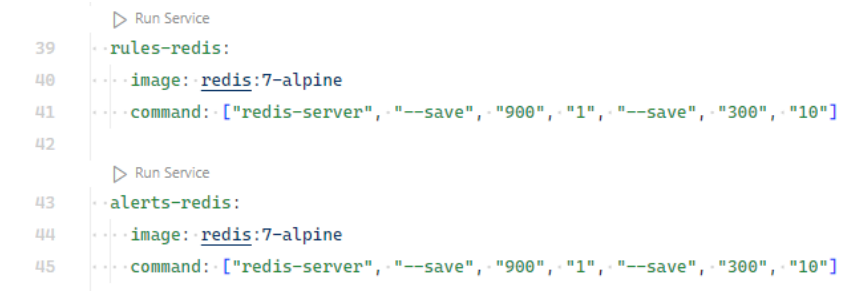
- RabbitMQ, czyli broker wiadomości służący do komunikacji asynchronicznej między mikroserwisami. Umożliwia kolejkowanie i przesyłanie zdarzeń telemetrycznych, wyników reguł i alertów.
- Udostępnia panel zarządzania pod adresem: **localhost:15672**.

`telemetry-db, rules-db, alerts-db`



- Osobne instancje bazy **PostgreSQL**, każda dedykowana odpowiedniemu mikroserwisowi backendowemu.
- Zarządzanie odbywa się za pomocą narzędzia **psql** wewnątrz kontenera.
- Wykorzystano wolumeny służące do trwałego przechowywania danych z PostgreSQL dla każdego mikroserwisu.

`rules-redis, alerts-redis`



- Redis używany do wsparcia działania mikroserwisów (np. cache lub obsługa stanu alertów).
- Zarządzanie odbywa się za pomocą **redis-cli** wewnątrz kontenera.

Sekcja: Mikroserwisy backendowe

`telemetry`

```
Run Service
telemetry:
  build: ./telemetry-service
  depends_on:
    rabbitmq:
      condition: service_healthy
    telemetry-db:
      condition: service_started
  environment:
    DATABASE_URL: postgres://postgres:secret@telemetry-db:5432/telemetry
    RABBIT_URL: amqp://guest:guest@rabbitmq:5672
    RAILS_ENV: development
  volumes: [ ./telemetry-service:/app ]
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.telemetry.rule=PathPrefix(`/telemetry`)"
    - "traefik.http.routers.telemetry.entrypoints=web"
    - "traefik.http.services.telemetry.loadbalancer.server.port=3000"
  stdin_open: true
  tty: true
  command: >
    bash -c "
      bin/rails db:create &&
      bin/rails db:migrate &&
      bin/rails db:schema:load &&
      bin/rails db:seed &&
      rm -f tmp/pids/server.pid &&
      bin/rails s -b '0.0.0.0'
    "
```

- Mikroserwis odpowiedzialny za zbieranie i zapisywanie danych telemetrycznych. Komunikuje się z bazą `telemetry-db` i `RabbitMQ`.
- Udostępniany przez Traefik pod ścieżką `/telemetry`.
- Command przy każdym starcie kontenera resetuje bazę i seeduje przygotowanymi w aplikacji przykładowymi danymi.
- Konfiguracja labels dla **Traefik**:
  - `traefik.enable=true` – aktywuje routing dla serwisu.
  - `traefik.http.routers.telemetry.rule=PathPrefix(/telemetry)` – przekierowuje żądania HTTP zaczynające się od /telemetry.
  - `traefik.http.routers.telemetry.entrypoints=web` – korzysta z punktu wejścia Traefik web (port 80 -> 8080).
  - `traefik.http.services.telemetry.loadbalancer.server.port=3000` – przekazuje ruch do portu 3000 wewnątrz kontenera (Rails).

`rules`

```
Run Service
rules:
  build: ./rules-service
  depends_on:
    rabbitmq:
      condition: service_healthy
    rules-db:
      condition: service_started
    rules-redis:
      condition: service_started
  environment:
    DATABASE_URL: postgres://postgres:secret@rules-db:5432/rules
    RABBIT_URL: amqp://guest:guest@rabbitmq:5672
    REDIS_URL: redis://rules-redis:6379/0
    RAILS_ENV: development
  volumes:
    - ./rules-service:/app
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.strategy-rules.rule=PathPrefix(`/rules`)"
    - "traefik.http.routers.strategy-rules.entrypoints=web"
    - "traefik.http.services.strategy-rules.loadbalancer.server.port=3000"
  stdin_open: true
  tty: true
  command: >
    bash -c "
      bin/rails db:create &&
      bin/rails db:migrate &&
      bin/rails db:schema:load &&
      bin/rails db:seed &&
      rm -f tmp/pids/server.pid &&
      bin/rails s -b '0.0.0.0'
    "
```

- Mikroserwis obsługujący reguły. Ocenia wartości metryk i generuje zdarzenia, gdy reguły zostaną spełnione. Łączy się z bazą `rules-db`, `RabbitMQ` oraz `Redis`.
- Udostępniany przez Traefik pod ścieżką `/rules`.
- Command przy każdym starcie kontenera resetuje bazę i seeduje przygotowanymi w aplikacji przykładowymi danymi.
- Konfiguracja labels dla **Traefik**:
  - `traefik.enable=true` – aktywuje routing dla serwisu.
  - `traefik.http.routers.telemetry.rule=PathPrefix(/rules)` – przekierowuje żądania HTTP zaczynające się od /telemetry.
  - `traefik.http.routers.telemetry.entrypoints=web` – korzysta z punktu wejścia Traefik web (port 80 -> 8080).
  - `traefik.http.services.telemetry.loadbalancer.server.port=3000` – przekazuje ruch do portu 3000 wewnątrz kontenera (Rails).

`rules-subscriber`

- Jest to osobny proces oparty na aplikacji mikroserwisu **`rules`**, uruchamiany w celu asynchronicznego przetwarzania danych telemetrycznych. Nasłuchuje on wiadomości z RabbitMQ, które zawierają nowe odczyty telemetryczne publikowane przez mikroserwis **`telemetry`**.
- Po odebraniu wiadomości, **`rules-subscriber`** ocenia wartości metryk zgodnie z zapisanymi regułami, a w przypadku ich spełnienia, generuje zdarzenia o przekroczeniu progów.
- W definicji usługi zastosowano warunek **`depends_on.rabbitmq.condition: service_healthy`**, który zapewnia, że proces **`rules-subscriber`** zostanie uruchomiony dopiero wtedy, gdy RabbitMQ będzie w pełni gotowy do obsługi komunikacji. W przeciwnym razie, wcześniejsze uruchomienie tego procesu mogłoby skutkować niepowodzeniem połączenia z kolejką, co doprowadziłoby do błędów i konieczności ponownego startu kontenera.

`alerts`

```yaml
alerts:
  build: ./alerts-service
  depends_on:
    rabbitmq:
      condition: service_healthy
    alerts-db:
      condition: service_started
    alerts-redis:
      condition: service_started
  environment:
    DATABASE_URL: postgres://postgres:secret@alerts-db:5432/alerts
    RABBIT_URL: amqp://guest:guest@rabbitmq:5672
    REDIS_URL: redis://alerts-redis:6379/0
    RAILS_ENV: development
  volumes: [ ./alerts-service:/app ]
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.pit-alerts.rule=PathPrefix(`/alerts`)"
    - "traefik.http.routers.pit-alerts.entrypoints=web"
    - "traefik.http.services.pit-alerts.loadbalancer.server.port=3000"
  stdin_open: true
  tty: true
  command: >
    bash -c "
      bin/rails db:create &&
      bin/rails db:migrate &&
      bin/rails db:schema:load &&
      bin/rails db:seed &&
      rm -f tmp/pids/server.pid &&
      bin/rails s -b '0.0.0.0'
    "
```

- Mikroserwis zarządzający alertami, ich tworzeniem, aktualizacją i automatycznym zamykaniem. Łączy się z bazą `alerts-db`, RabbitMQ i Redis.
- Udostępniany przez Traefik pod ścieżką `/alerts`.
- Command przy każdym starcie kontenera resetuje bazę i seeduje przygotowanymi w aplikacji przykładowymi danymi.
- Konfiguracja labels dla **Traefik**:
  - `traefik.enable=true` – aktywuje routing dla serwisu.
  - `traefik.http.routers.telemetry.rule=PathPrefix(/alerts)` – przekierowuje żądania HTTP zaczynające się od /telemetry.
  - `traefik.http.routers.telemetry.entrypoints=web` – korzysta z punktu wejścia Traefik web (port 80 -> 8080).
  - `traefik.http.services.telemetry.loadbalancer.server.port=3000` – przekazuje ruch do portu 3000 wewnątrz kontenera (Rails).

`alerts-subscriber`

```
Run Service
alerts-subscriber:
  build: ./alerts-service
  depends_on:
    rabbitmq:
      condition: service_healthy
    alerts-db:
      condition: service_started
    alerts-redis:
      condition: service_started
  command: bundle exec rake rabbitmq:listen
  environment:
    DATABASE_URL: postgres://postgres:secret@alerts-db:5432/alerts
    RABBIT_URL: amqp://guest:guest@rabbitmq:5672
    REDIS_URL: redis://alerts-redis:6379/0
    RAILS_ENV: development
  volumes: [ ./alerts-service:/app ]
```

- Osobny proces działający na bazie aplikacji z serwisu **`alerts`**, nasłuchujący wiadomości z RabbitMQ o naruszeniach reguł, które skutkują tworzeniem nowych alertów.

`alerts-worker`

```
alerts-worker:
  build: ./alerts-service
  depends_on:
    rabbitmq:
      condition: service_healthy
    alerts-db:
      condition: service_started
    alerts-redis:
      condition: service_started
  command: bundle exec rake alerts:worker
  environment:
    DATABASE_URL: postgres://postgres:secret@alerts-db:5432/alerts
    RABBIT_URL: amqp://guest:guest@rabbitmq:5672
    REDIS_URL: redis://alerts-redis:6379/0
    RAILS_ENV: development
  volumes: [ ./alerts-service:/app ]
```

- Proces działający na bazie aplikacji z serwisu **`alerts`**, uruchamiający skrypt, cały czas w pętli monitorujący otwarte alerty i decydujący o ich zamknięciu w przypadku ustania otrzymywania wiadomości o naruszeniach reguł.

Sekcja: Frontend

`dashboard`

```yaml
dashboard:
  image: nginx:alpine
  volumes:
    - ./dashboard-service:/usr/share/nginx/html:ro
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.dashboard.rule=PathPrefix(`/dashboard`)"
    - "traefik.http.routers.dashboard.middlewares=dashboard-stripprefix"
    - "traefik.http.middlewares.dashboard-stripprefix.stripprefix.prefixes=/dashboard"
    - "traefik.http.routers.dashboard.entrypoints=web"
    - "traefik.http.services.dashboard.loadbalancer.server.port=80"
```

- Aplikacja frontendowa dla inżyniera wyścigowego, wyświetlająca dane telemetryczne oraz alerty.
- Hostowana przez NGINX, dostępna pod ścieżką Trafeik `/dashboard`.
- Konfiguracja labels dla **Traefik**:
    - `traefik.http.routers.telemetry.rule=PathPrefix(/dashboard)` – przekierowuje żądania HTTP zaczynające się od /telemetry.

`simulator`

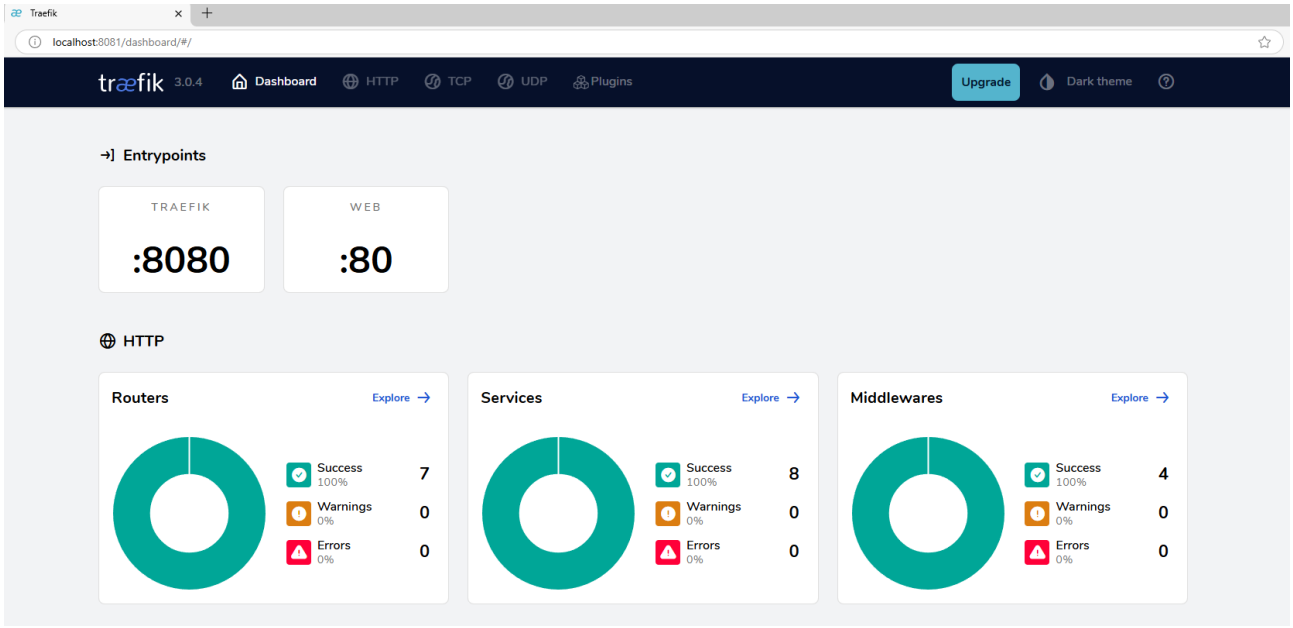
```yaml
simulator:
  image: nginx:alpine
  volumes:
    - ./simulator-service:/usr/share/nginx/html:ro
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.simulator.rule=PathPrefix(`/simulator`)"
    - "traefik.http.routers.simulator.middlewares=simulator-stripprefix"
    - "traefik.http.middlewares.simulator-stripprefix.stripprefix.prefixes=/simulator"
    - "traefik.http.routers.simulator.entrypoints=web"
    - "traefik.http.services.simulator.loadbalancer.server.port=80"
```

- Aplikacja frontendowa symulująca jazdę bolidem. Umożliwia ręczne sterowanie metrykami i przesyłanie ich do systemu.
- Hostowana przez NGINX, dostępna pod ścieżką Trafeik `/simulator`.
- Konfiguracja labels dla **Traefik**:
    - `traefik.http.routers.telemetry.rule=PathPrefix(/simulator)` – przekierowuje żądania HTTP zaczynające się od /telemetry.

Opis serwisów i przepływu danych

Traefik

W projekcie zastosowano Traefik jako **Reverse Proxy**, pełniący jednocześnie rolę **API Gateway** oraz mechanizmu **Service Discovery**. Dzięki integracji z Dockerem, Traefik automatycznie wykrywa kontenery uruchomione w systemie, na podstawie zadeklarowanych **etykiet** (labels) i dynamicznie tworzy do nich routing HTTP.



Rysunek 11. Główny dashboard w panelu administracyjnym Traefik

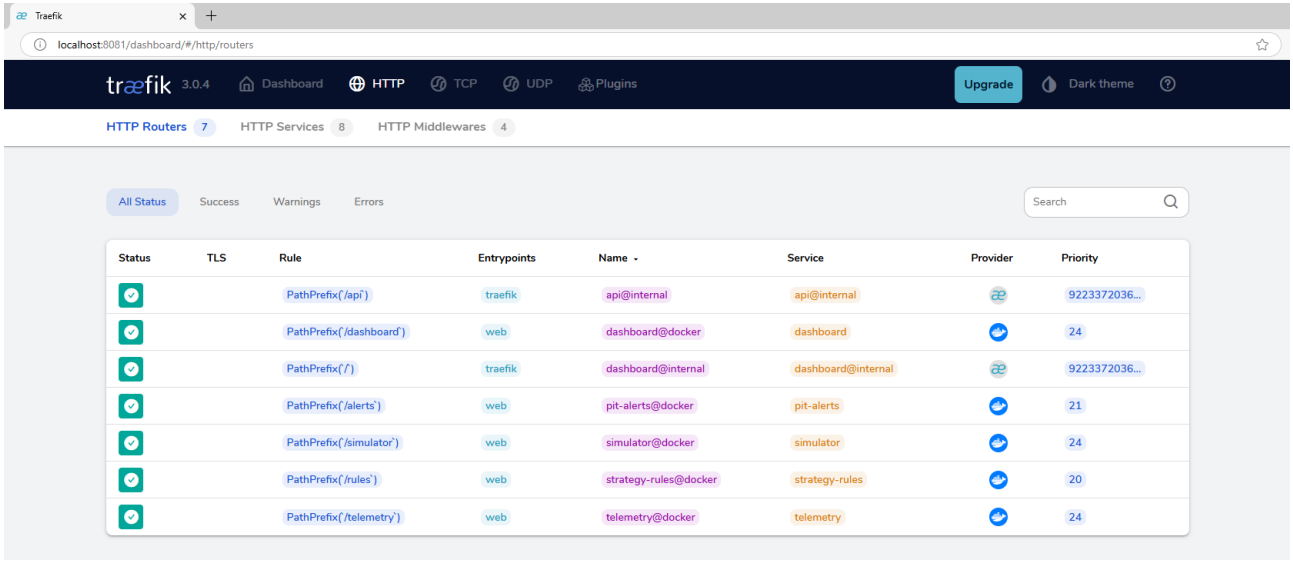
W obecnej konfiguracji Traefik udostępnia dwa adresy:

- `http://localhost:8080` – główny entrypoint, przez który routowane są żądania do wszystkich usług systemu (np. `/telemetry`, `/alerts`, `/dashboard`),
- `http://localhost:8081` – panel administracyjny Traefika, umożliwiający podgląd zarejestrowanych tras, usług i stanu routingu.

Dzięki zastosowaniu Traefika:

- cały system działa w obrębie jednego punktu wejścia,
- nie ma potrzeby znajomości portów poszczególnych mikroserwisów,

Po uruchomieniu systemu, każdy z mikroserwisów i aplikacji frontendowych zostaje **automatycznie wykryty** przez Traefika na podstawie przypisanych etykiet. Trafiają one do zakładki **Routes** w panelu administracyjnym, gdzie można podejrzeć ścieżki URL, reguły routingu oraz docelowe usługi.



Rysunek 12. Zarejestrowane mikroserwisy i ich routing w Traefik

Każdy serwis posiada własny punkt dostępu:

- `/telemetry` - mikroserwis Rails (port 3000)
- `/rules` - mikroserwis Rails (port 3000)
- `/alerts` - mikroserwis Rails (port 3000)
- `/dashboard` - statyczna aplikacja frontendowa na NGINX (port 80)
- `/simulator` - druga aplikacja NGINX (port 80)

Choć wszystkie backendy działają na tym samym porcie 3000, a frontendy na porcie 80, nie dochodzi do konfliktu. Dzieje się tak, ponieważ:

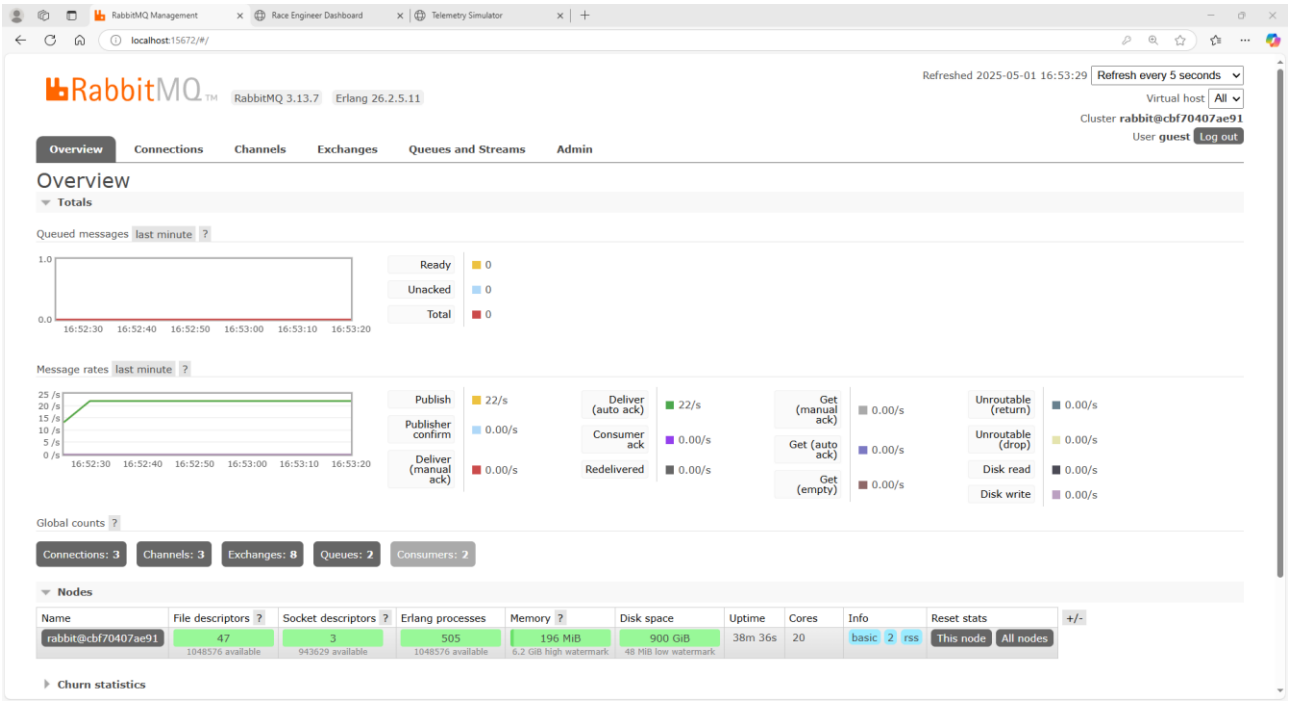
- wszystkie kontenery znajdują się w **wewnętrznej sieci Dockera**, która zapewnia izolację portów między usługami,
- tylko Traefik ma wystawione porty (8080, 8081) na zewnątrz,
- wszystkie pozostałe usługi **nie są dostępne z poziomu hosta**, dzięki czemu mamy pełną kontrolę nad dostępem do serwisów w obrębie systemu.

RabbitMQ

W systemie zastosowano **RabbitMQ** jako mechanizm asynchronicznej komunikacji między mikroserwisami. Zamiast bezpośrednich wywołań HTTP pomiędzy usługami, dane telemetryczne oraz informacje o zdarzeniach przesyłane są jako wiadomości w kolejce.

Po uruchomieniu systemu panel administracyjny RabbitMQ dostępny jest pod adresem `http://localhost:15672`, login i hasło: `guest` / `guest`.

Po zalogowaniu można zaobserwować, że mikroserwisy wymieniają między sobą wiadomości – widoczne są aktywne połączenia, rosnąca liczba wiadomości oraz statystyki (Message rates).

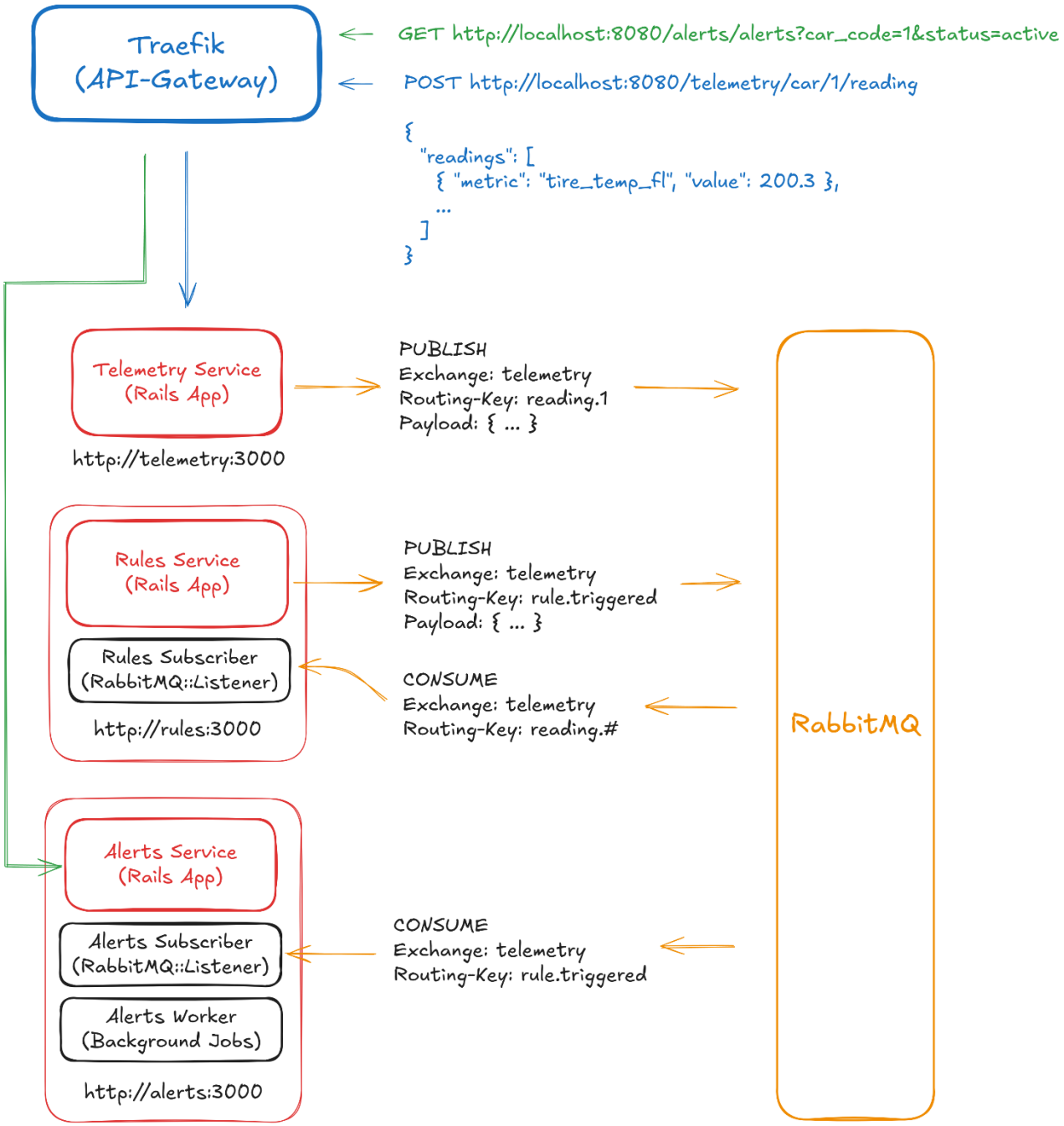


Rysunek 13 Widok dashboardu RabbitMQ

W systemie wykorzystano **Exchange** typu **Topic** o nazwie `telemetry`. Mikroserwisy nie komunikują się bezpośrednio ze sobą, tylko publikują i subskrybują wiadomości za pośrednictwem tego exchange'a.

- **Telemetry Service** publikuje zdarzenia z metrykami jako `reading.<car_code>`, np. `reading.1` dla bolidu Maxa Verstappena.
- **Rules Service** subskrybuje wiadomości o wzorcu `reading.#`, przetwarza dane i może wysłać zdarzenie `rule.triggered`.
- **Alerts Service** subskrybuje `rule.triggered` i tworzy alerty.

Na poniższym rysunku przedstawiono przebieg komunikacji asynchronicznej w systemie, na przykładzie utworzenia **nowej metryki telemetrycznej**. Zaznaczono kolejno publikowane wiadomości, subskrybentów oraz sposób ich przetwarzania w poszczególnych mikroserwisach:



Rysunek 14. Przepływ danych między mikroserwisami za pośrednictwem RabbitMQ.

Telemetry Service

Telemetry Service to jedna z trzech aplikacji backendowych systemu, znajdująca się w katalogu `telemetry-service/`. Jest to samodzielna aplikacja napisana w frameworku Ruby on Rails.

Telemetry Service odpowiada za odbiór, walidację, zapis i publikację danych telemetrycznych generowanych przez symulator bolidu. Przechowuje także informacje o wszystkich dostępnych kierowcach oraz ich samochodach (car_code, nazwisko, zespół, zdjęcie).

Po zapisaniu danych telemetrycznych, każda metryka jest **natychmiast publikowana** jako zdarzenie do **RabbitMQ** w formacie JSON, dzięki czemu inne mikroserwisy mogą je niezależnie przetwarzać (np. Rules Service oceniający reguły).

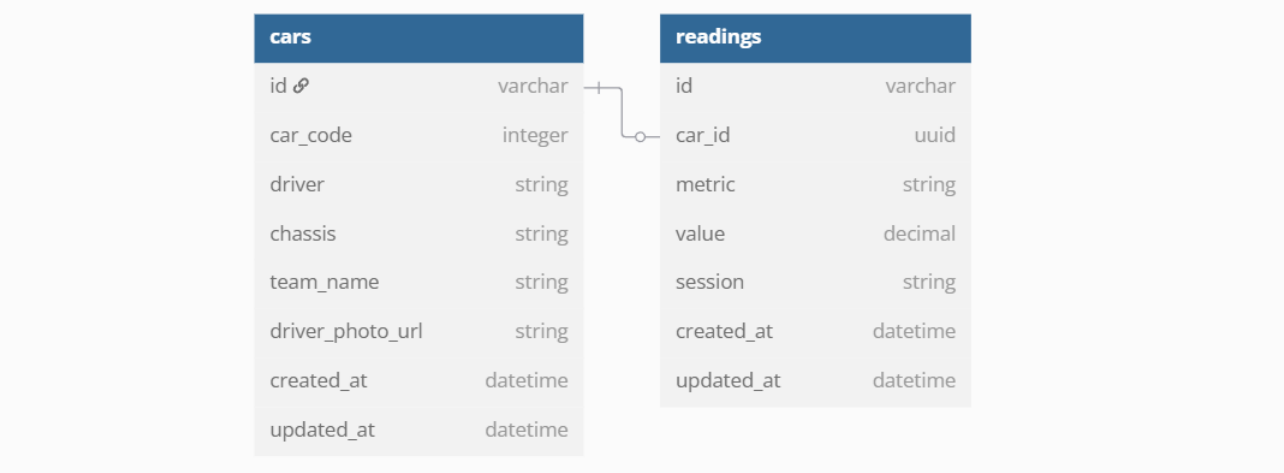
Kluczowe pliki i moduły

- `app/models/car.rb` - Reprezentuje samochód/kierowcę
- `app/models/reading.rb` - Reprezentuje pojedynczą metrykę telemetryczną przypisaną do danego bolidu.
- `app/controllers/cars_controller.rb` - Udostępnia operacje CRUD dla kierowców.
- `app/controllers/readings_controller.rb` - Obsługuje tworzenie i pobieranie danych telemetrycznych; publikuje dane do RabbitMQ.
- `app/services/rabbitmq_publisher.rb` - Singletonowy publisher do RabbitMQ; publikuje zdarzenia telemetryczne do exchange telemetry.
- `db/seeds.rb` - Wprowadza do bazy danych listę kierowców oraz przykładowe metryki.
- `config/routes.rb` - Definiuje wszystkie dostępne endpointy REST API w ramach przestrzeni `/telemetry`.

Model danych

Serwis oparty jest o dwa główne modele:

- **Car** – reprezentuje kierowcę i jego bolid (m.in. car_code, nazwisko, zespół, zdjęcie).
- **Reading** – pojedyncza metryka telemetryczna (np. speed, rpm, fuel_level), przypisana do konkretnego samochodu.



Rysunek 15 Schemat ERD z programu https://dbdiagram.io/

Dostępne endpointy (REST API)

Poniżej przedstawiono tylko najbardziej istotne endpointy, dla każdego zasobu dostępny jest pełny zestaw akcji CRUD zgodny z zasadami tworzenia REST API. Endpointy posiadają namespace (w Railsach scope `/telemetry`):

`/telemetry/up`	GET	Healthcheck dla Traefik
`/telemetry/cars`	GET	Zwraca listę wszystkich kierowców (Car)
`/telemetry/cars/:code`	GET	Zwraca szczegóły jednego kierowcy (po `car_code`)
`/telemetry/cars/:car_code/readings`	GET	Zwraca listę wszystkich odczytów telemetrycznych danego bolidu
`/telemetry/cars/:car_code/readings`	POST	Przyjmuje zestaw nowych metryk telemetrycznych
`/telemetry/cars/:car_code/readings/latest`	GET	Zwraca najnowsze wartości każdej metryki dla wybranego kierowcy

Przykład:

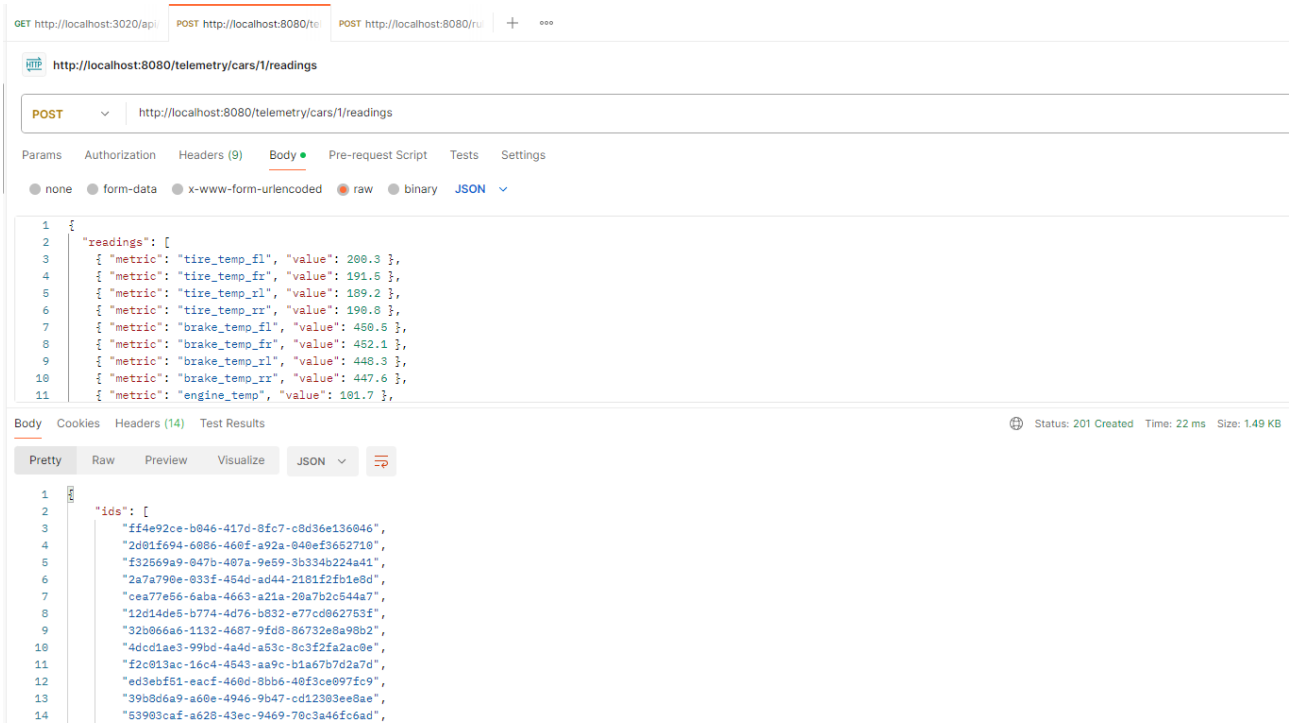
Aby utworzyć **nowy zestaw metryk**, które przesyła bolid, należy wysłać następujące zapytanie na podany poniżej endpoint:

```
POST /telemetry/cars/:car_code/readings

{
  "readings": [
    { "metric": "tire_temp_fl", "value": 200.3 },
    { "metric": "tire_temp_fr", "value": 191.5 },
    { "metric": "tire_temp_rl", "value": 189.2 },
    { "metric": "tire_temp_rr", "value": 190.8 },
    { "metric": "brake_temp_fl", "value": 450.5 },
    { "metric": "brake_temp_fr", "value": 452.1 },
    { "metric": "brake_temp_rl", "value": 448.3 },
    { "metric": "brake_temp_rr", "value": 447.6 },
  ]
}
```

Rysunek 16. Przykładowy payload dla endpointu do tworzenia nowych metryk

Metryki zostaną szybko zapisane (bulk insert w kilka ms) w bazie danych **PostgreSQL** (pod dane typu time-series na większa skalę należałoby użyć dedykowanej do tego rodzaju rekordów bazy danych np. **TimescaleDB**).



Rysunek 17. Sukces zapytania tworzącego zestaw metryk z danej chwili dla danego bolidu.

Publikacja danych do RabbitMQ

Każda metryka po utworzeniu jest osobno publikowana do exchange **telemetry** z routing key w formacie:

```
reading.<car_code>
```

Przykładowy payload dołączany do wiadomości w formacie JSON:

```
{
  "id": "ff4e92ce-b046-417d-8fc7-c8d36e136046",
  "car_code": 1,
  "metric": "brake_temp_fl",
  "value": 820
}
```

Rules Service

Rules Service to druga z aplikacji backendowych systemu, znajdująca się w katalogu `rules-service/`. Jest to niezależna aplikacja stworzona w Ruby on Rails, odpowiedzialna za zarządzanie regułami inżynierskimi oraz ich ocenę na podstawie nadchodzących danych telemetrycznych.

Serwis ten pozwala na tworzenie i edytowanie reguł, które opisują, w jakich sytuacjach należy wygenerować alert (np. „**gdy temperatura hamulca przekroczy 800°C**”). Każda reguła może dotyczyć dowolnej metryki, a także być przypisana do konkretnego kierowcy (`car_scope`), lub obowiązywać globalnie.

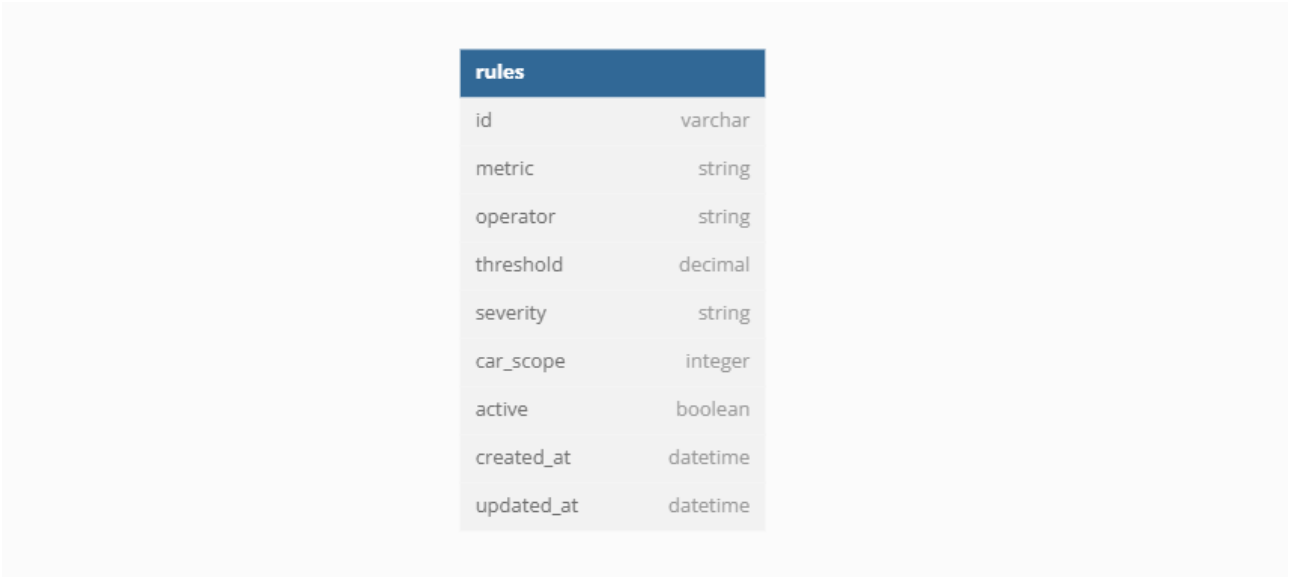
Kluczowe pliki i moduły

- `app/models/rule.rb` - Model bazy danych przechowujący wszystkie reguły (metryka, próg, operator, zakres, aktywność).
- `app/controllers/rules_controller.rb` - REST API do przeglądania, tworzenia i edytowania reguł.
- `app/services/rule_evaluator.rb` - Logika oceny reguł i publikacji zdarzeń `rule.triggered`.
- `app/services/rabbitmq/listener.rb` - Klasa uruchamiana przez worker, inicjująca nasłuchiwanie wiadomości z RabbitMQ.
- `app/services/rabbitmq/subscriber.rb` - Tworzy dynamiczną kolejkę z subskrypcją na `reading.#`, odbiera wiadomości i przekazuje do `RuleEvaluator`.
- `app/services/rabbitmq/publisher.rb` - Klasa pomocnicza do publikowania zdarzeń do RabbitMQ.
- `db/seeds.rb` - Wprowadza przykładowy zestaw reguł do bazy danych (dla bolidu #1 – Max Verstappen).
- `lib/tasks/rabbitmq_listener.rake` - Plik definiujący Rake Task uruchamiay w osobnym kontenerze (`rules-subscriber`). Zadanie to startuje nasłuchiwanie wiadomości z RabbitMQ i deleguje je do logiki przetwarzania (`RuleEvaluator`).

Model danych

Rules Service oparty jest o pojedynczy model bazodanowy:

- **Rule** – reprezentuje jedną regułę inżynierską związaną z daną metryką telemetryczną.



Rysunek 18. Schemat ERD z programu https://dbdiagram.io/

Dostępne endpointy (REST API)

Poniżej przedstawiono wszystkie endpointy dostępne w tym serwisie. Endpointy posiadają namespace (w Railsach scope `/rules`):

`/rules/up`	GET	Healthcheck dla Traefik
`/rules/rules`	GET	Zwraca listę wszystkich reguł w systemie
`/rules/rules/:id`	GET	Zwraca szczegóły pojedynczej reguły
`/rules/rules`	GET	Tworzy nową regułę na podstawie przesłanych danych
`/rules/rules/:id`	PUT/PATCH	Aktualizuje istniejącą regułę
`/rules/rules/:id`	DELETE	Usuwa regułę z systemu

Przetwarzanie danych z RabbitMQ – worker jako osobny proces

W celu przetwarzania danych telemetrycznych przesyłanych przez RabbitMQ, uruchamiany jest osobny proces Dockera – **`rules-subscriber`**, który wykonuje zadanie zdefiniowane w pliku **`lib/tasks/rabbitmq_listener.rake`**.

```
bundle exec rake rabbitmq:listen
```

Zadanie to to tzw. **Rake task** – prosty mechanizm w Ruby on Rails służący do uruchamiania zadań w tle lub skryptów narzędziowych. W tym przypadku uruchamia on klasę **Rabbitmq::Listener**, który inicjalizuje nasłuchiwanie wiadomości i przekazuje je do klasy **RuleEvaluator**. Dzięki takiemu podejściu jesteśmy w osobnym wątku i niezależnie od warstwy HTTP aplikacji nasłuchiwać na zdarzenia z RabbitMQ.

Subskrybent łączy się z **exchange `telemetry`** i nasłuchuje wiadomości z routing key **`reading.#`**, czyli wszystkich metryk wysyłanych przez Telemetry Service. Każda wiadomość przetwarzana jest przez klasę **RuleEvaluator**.

Ocena reguł – logika działania

Po odebraniu wiadomości, klasa **RuleEvaluator** sprawdza wszystkie aktywne reguły (**`Rule.active`**) dla danej metryki. Następnie, dla każdej z nich:

- sprawdza, czy reguła dotyczy danego kierowcy (**`car_scope.nil?`** lub zgodny **`car_code`**),
- porównuje wartość metryki z wartością progową (threshold) na podstawie operatora (**`>`**, **`<`**, **`==`**, **`!=`** itd.),
- jeśli warunek jest spełniony, publikuje nowe zdarzenie **`rule.triggered`** do RabbitMQ.
- nowe zdarzenie może mieć status: **`info`**, **`warning`** lub **`critical`**.

W typowych systemach telemetrycznych, ocena reguł może być wspierana przez modele predykcyjne lub systemy oparte na machine learning – np. prognozujące awarie lub odchylenia. W tym projekcie zastosowano prosty model warunkowy w formie operatora logicznego i wartości progowej.

Publikacja zdarzeń do RabbitMQ

Gdy reguła zostanie spełniona, **Rules Service** wysyła wiadomość JSON z informacją o zdarzeniu naruszenia reguły. Wiadomość ta trafia do exchange telemetry, z routing key:

```
rule.triggered
```

Przykładowy payload wiadomości:

```
{
  "rule_id": "abc123",
  "car_code": 1,
  "metric": "brake_temp_fl",
  "operator": ">",
  "threshold": 800,
  "value": 820.5,
  "severity": "critical",
  "timestamp": "2025-04-30T14:35:20Z"
}
```

Alerts Service

Alerts Service to trzecia aplikacja backendowa w systemie, znajdująca się w katalogu `alerts-service/`. Jest to niezależna aplikacja napisana w frameworku Ruby on Rails, której zadaniem jest tworzenie, aktualizacja i zamykanie alertów na podstawie zdarzeń publikowanych przez Rules Service.

Serwis ten nie tylko tworzy alerty, ale również monitoruje ich aktywność i automatycznie je wygasza po określonym czasie, jeśli dane krytyczne nie są już przesyłane.

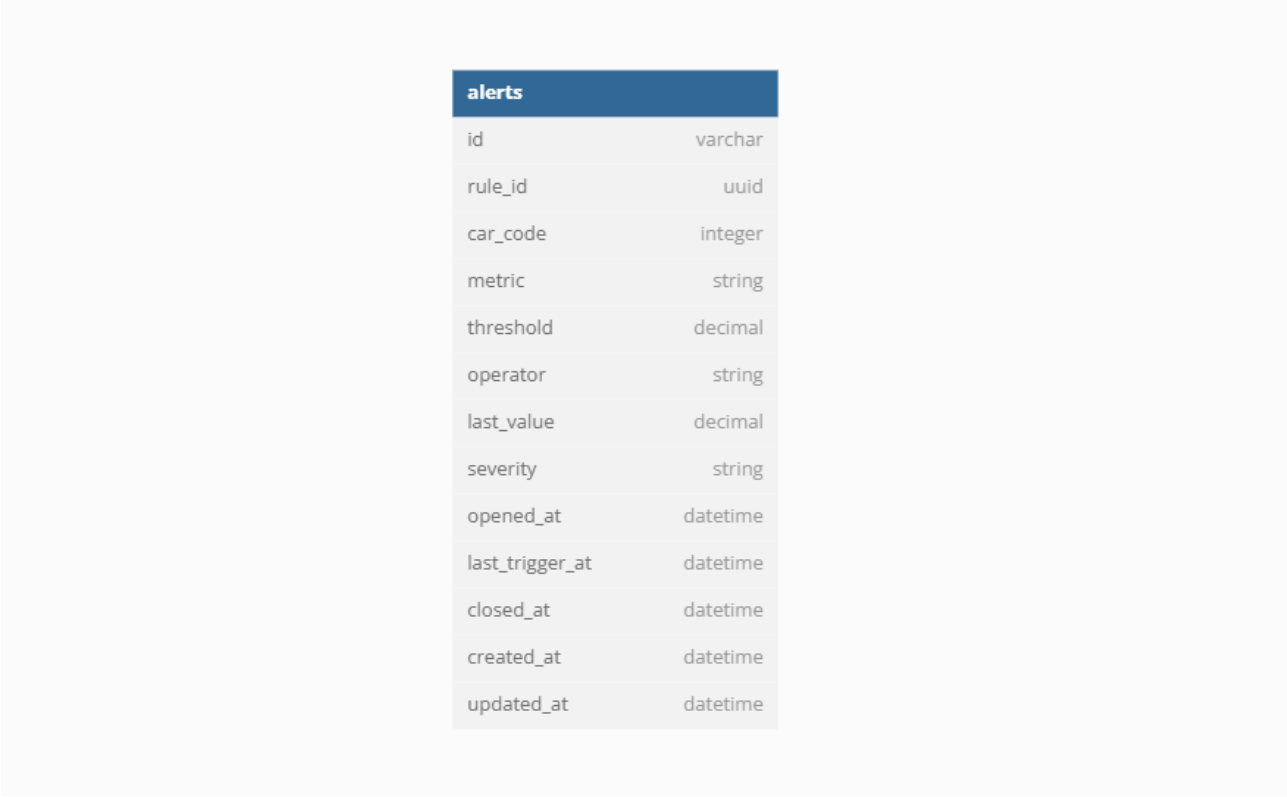
Kluczowe pliki i moduły

- `app/models/alert.rb` - Model opisujący strukturę alertu oraz metody pomocnicze (close!, active, closed).
- `app/controllers/alerts_controller.rb` - Kontroler udostępniający API do pobierania alertów z możliwością filtrowania.
- `app/services/alert_evaluator.rb` - Klasa odpowiedzialna za tworzenie lub aktualizację alertów na podstawie wiadomości z RabbitMQ.
- `lib/tasks/rabbitmq_listener.rake` - Rake task do uruchomienia subskrybenta RabbitMQ (`alerts-subscriber`).
- `lib/tasks/alerts_worker.rake` – Rake task, osobny proces sprawdzający czas wygaśnięcia aktywnych alertów i zamykający je automatycznie.
- `service/rabbitmq/subscriber.rb` - Klasa łącząca się z exchange telemetry i nasłuchująca wiadomości `rule.triggered`.

Model danych

Alerts Service opiera się na jednym modelu:

- **Alert** – reprezentuje pojedynczy alert wyzwolony przez regułę.



Rysunek 19. Schemat ERD z programu https://dbdiagram.io/

Dostępne endpointy (REST API)

Poniżej przedstawiono wszystkie endpointy dostępne w tym serwisie. Endpointy posiadają namespace (w Railsach scope `/alerts`):

`/alerts/up`	GET	Healthcheck dla Traefik
`/alerts/alerts`	GET	Zwraca listę alertów (z opcjonalnymi filtrami)
`/alerts/alerts/:id`	GET	Zwraca szczegóły pojedynczego alertu

Dostępne filtry dla zwracanych alertów z endpointu `/alerts`:

- `car_code=` – filtrowanie po kierowcy,
- `status=active|closed` – status alertu,
- `since=` – od danego czasu (ISO8601),
- `limit=` – maksymalna liczba wyników.

Przetwarzanie danych – odbiór zdarzeń z RabbitMQ

Alerts Service uruchamia osobny proces nasłuchujący RabbitMQ (**`alerts-subscriber`**) poprzez Rake task:

```
bundle exec rake rabbitmq:listen
```

Komponent ten subskrybuje wiadomości z routing key **`rule.triggered`**. Każda odebrana wiadomość przekazywana jest do klasy **AlertEvaluator**, która analizuje jej treść i tworzy nowy alert lub aktualizuje istniejący (np. jeśli ten sam problem trwa dalej).

Przykładowy payload wiadomości z Rules Service:

```
{
  "rule_id": "abc123",
  "car_code": 1,
  "metric": "brake_temp_fl",
  "operator": ">",
  "threshold": 800,
  "value": 820.5,
  "severity": "critical",
  "timestamp": "2025-04-30T14:35:20Z"
}
```

Logika alertów i worker do zamykania

Jeśli dla danego **`rule_id`** i **`car_code`** istnieje już aktywny alert, system aktualizuje jego wartość (**`last_value`**) i czas (**`last_trigger_at`**). W przeciwnym razie tworzony jest nowy rekord w bazie danych z alertem.

Dodatkowo, w ramach osobnego serwisu (**`alerts-worker`**), cyklicznie uruchamiany jest worker za pomocą Rake Task analogicznie jak subscriber (**`rake alerts:worker`**), który sprawdza wszystkie aktywne alerty i zamyka te, które nie zostały odświeżone przez określony czas (konkretnie 3 sekundy). Dzięki temu alerty nie pozostają aktywne w nieskończoność po ustaniu zagrożenia oraz mamy do wglądu historię ostatnich alertów. Jest to taka pseudo wersja crona zrealizowana w aplikacji.

Simulator

Simulator to lekka aplikacja frontendowa składająca się z jednego pliku HTML z dodatkowym kodem JavaScript, znajdująca się w katalogu `simulator-service/`. Udostępniona jest za pomocą kontenera **nginx** i obsługiwana przez Traefik pod adresem `localhost:8080/simulator`.

Aplikacja umożliwia symulację parametrów telemetrycznych dla dowolnego kierowcy F1 wybranego z listy (pełna lista zawodników sezonu 2024 wczytywana z Telemetry Service). Dla każdej metryki dostępny jest suwak, którym można manipulować wartościami (np. temperatura opon, prędkość, poziom paliwa, napięcie baterii).

Co sekundę aplikacja wysyła aktualny zestaw metryk (w formacie JSON) do Telemetry Service, inicjując cały łańcuch przetwarzania danych w systemie.

Wygląd aplikacji został przedstawiony w pierwszy rozdziałach demonstrujących aplikację.

Dashboard

Dashboard to druga aplikacja frontendowa, składająca się z jednego pliku HTML z dodatkowym kodem JavaScript, również serwowana przez **nginx**, zlokalizowana w katalogu `dashboard-service/` i dostępna pod adresem `localhost:8080/dashboard`.

Jest to aplikacja typu live monitoring, zaprojektowana z perspektywy inżyniera wyścigowego. Umożliwia:

- wybór kierowcy (na podstawie listy dostępnej w Telemetry Service),
- obserwację najnowszych wartości wszystkich metryk telemetrycznych (odświeżanie co 1s),
- podgląd aktywnych alertów wraz z poziomem ich krytyczności,
- przegląd historii zamkniętych alertów z ostatnich 24 godzin.

Dashboard komunikuje się bezpośrednio z **Telemetry Service** i **Alerts Service** poprzez REST API, kierując zapytania bezpośrednio na **API Gateway** do **Traefik**.

Wygląd aplikacji został przedstawiony w pierwszy rozdziałach demonstrujących aplikację.

Podsumowanie

Zrealizowany projekt przedstawia kompletny, działający system oparty na **architekturze mikroserwisowej**, umożliwiający odbiór, przetwarzanie i wizualizację danych telemetrycznych z bolidów Formuły 1 w czasie rzeczywistym.

System składa się z trzech usług backendowych, dwóch aplikacji frontendowych oraz infrastruktury pomocniczej (RabbitMQ, Traefik, PostgreSQL), połączonych za pomocą asynchronicznej komunikacji zdarzeniowej.

Stworzony ekosystem pozwala w sposób czytelny i rozdzielony na odpowiedzialności:

- symulować dane telemetryczne,
- oceniać reguły inżynierskie na podstawie metryk,
- generować i zarządzać alertami,
- udostępniać podgląd sytuacji w czasie rzeczywistym inżynierowi wyścigowemu.

W praktyce produkcyjnej analogiczna funkcjonalność alertów i metryk mogłaby zostać zrealizowana za pomocą gotowych narzędzi takich jak **Prometheus** i **Alertmanager**, jednak celem niniejszego projektu nie było odtworzenie istniejącego rozwiązania 1:1, a praktyczna nauka podejścia mikroserwisowego, integracji systemów i komunikacji zdarzeniowej.