Technical Report - **Product specification**

RaceFlow

Course:	IES - Introdução à Engenharia de Software
Date:	Aveiro, 07/10/2024
Students:	113403: José Oliveira 114246: Carolina Prata 113780: João Varela 112733: Hugo Sousa
Project abstract:	RaceFlow is a real-time analytics tool for monitoring Formula 1 cars during races.

Table of contents:

1 Introduction

This first iteration will explore the client's requirements, for the future development of the project.

RaceFlow is a race performance analysis tool for Formula 1 cars. It keeps a constant eye on the key elements of the sport in real time, such as visualization of drivers on track, tire conditions, fuel consumptions, and other relevant data from all drivers, changing dynamically throughout the race. RaceFlow allows the observation and analysis of vehicle conditions in an informative manner because of its intuitive layout, providing deeper insight into team strategies and how these technical variables may influence the final race outcome.

2 Product concept

Vision statement

The vision for RaceFlow is an innovative performance analysis tool, tailor-made for Formula 1 racing. It will offer real-time monitoring of race information: the driver's position on the track, tire condition, fuel consumption, and any other critical measure that may change during the race. It will be targeting teams, analysts, and enthusiasts by providing them with an informative platform on which to monitor vehicle conditions for assessment, which will in turn help make informed strategic decisions capable of influencing race outcomes.

Personas and Scenarios [secção 4.1 + 4.2 neste artigo]

Luís Militão

Luís Militão, 35 years old, is a software engineer living in Lisbon. He is a true Formula 1 enthusiast and rarely misses a race, no matter where he is. His interest goes far beyond just seeing who wins; he loves following the more technical details, like lap times, tire strategies, and the gaps between drivers. For Luís, the most important thing is keeping up with the race in real-time, without

missing any critical information. What frustrates him the most is when apps take too long to update or don't show enough data, like penalties or pit stop strategy changes. He wants everything updated by the second.

Rita Martins

Rita Martins, 28, is a marketing manager in Porto. Although she isn't a big Formula 1 fan, she ends up watching some races because her boyfriend is a fan, and she enjoys watching the events with him. For her, the most important moments are the critical ones, like overtakes or when something unexpected happens in the race, such as an accident or the Safety Car coming onto the track. She doesn't care much about technical details like tire strategies or lap times, preferring a simpler and more visual experience that shows who's winning right away. Rita mainly uses her phone to watch the race when she's out or multitasking and doesn't want to waste time on complicated interfaces or information she doesn't understand.

Pedro Costa

Pedro Costa, 22, is studying Race Engineering at Instituto Superior Técnico in Lisbon. For him, Formula 1 is much more than just watching who finishes first. Pedro is interested in everything related to car performance – from tire temperatures to fuel management and aerodynamics. Every race is an opportunity to learn something new and apply what he's studying. He likes following races on his laptop, where he can see detailed charts and data, and uses his tablet to track real-time car metrics. What bothers

him are apps that don’t show these technical details or take too long to update, as he wants to understand every detail precisely, without any delays.

Product requirements (User stories) agile project management

Real-time Position of Drivers (High Priority):

As a spectator, I want to see the real-time position of every driver during the race, so that I can track their performance throughout the race.

Show Time Gaps (Urgent Priority):

As a spectator, I want to clearly see the time gaps between drivers so that I understand how far apart the drivers are in terms of time and hence might estimate some overtakes.

Fastest Lap Notification (Low Priority):

As a spectator, I want to know at what time a driver achieved the fastest lap of the race, so that at any given time, I will be able to notice who is outshining the others.

Championship Standings (Medium Priority):

As a viewer, I would like to see a table representing the overall standing of drivers and teams in the championship to see at a glance who the leading drivers or teams are.

Detailed View for Each Driver (High Priority):

As a viewer, I would like to click on any driver's name and get race details so that I can get more detailed information about his/her performance.

Real-time Event Notifications (High Priority):

As a viewer, I want to be informed about the main events taking place during any given race, like Safety Cars, penalties, or accidents, in order to keep myself updated about what is regarded as the most important moments.

Weather Forecast Display (Medium Priority):

As a viewer, I'd like to get the weather forecast for the remainder of the race to have an idea of how this might influence strategy decisions made by drivers.

Pitstop Notifications (Low Priority):

As a viewer, I'd like to get notified at each pitstop every driver takes, understand his reason for doing so, to understand how such a decision may influence the rest of the race.

3 Architecture notebook

Key requirements and constraints

Key Requirements and Constraints

The architecture is designed to meet several key requirements and constraints:

Real-time Data Flow:

- The system must provide real-time updates on Formula 1 race data, from both the external Open F1 API and the internal Data Generator. RabbitMQ is used as a message broker to facilitate the flow of internal data, while WebSockets ensure real-time communication between the backend and the frontend.

Data Storage:

- **TimescaleDB** is employed for time-series data storage, essential for monitoring historical data like lap times and performance metrics.

Scalability and Message Brokering:

- RabbitMQ acts as a message broker to handle efficient distribution and processing of internally generated data, ensuring scalability as the volume of internal data grows.

External API Integration:

- The system integrates with the external Open F1 API to retrieve additional data (such as detailed driver information) necessary for entity creation within the system.

The backend uses a dedicated External API Service to fetch and populate this complementary data.

WebSocket Connection:

- WebSockets are used to keep the frontend updated in real-time with data generated internally by the Data Generator. This allows users to have a seamless, low-latency experience when viewing live race data.

Backend Resilience:

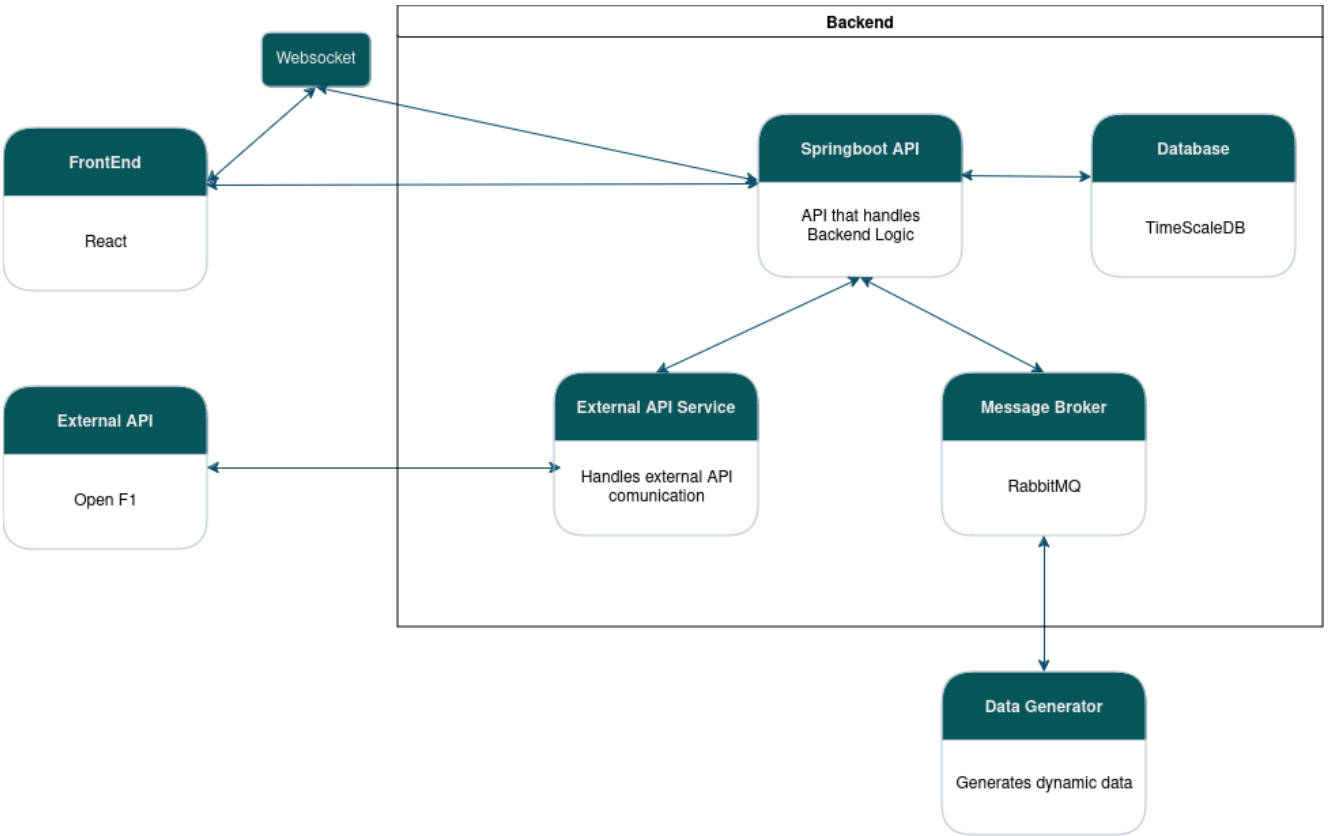
- The Spring Boot backend is responsible for managing all communication with RabbitMQ, TimescaleDB, the frontend, and the external API (for additional data required for entity creation). It must be resilient and capable of handling high data volumes while maintaining quick response times.

Architectural view

The architecture includes the following components and their interactions:

1. **Frontend (React)**:
    - Provides the user interface for visualizing live race data, car statistics, and driver performance.
    - Receives real-time data from the backend via a WebSocket connection for internally generated data (from the Data Generator).
2. **Backend (Spring Boot)**:
    - Serves as the core of the system, responsible for managing all flows of internal and external data.
    - Interacts with:
        - **RabbitMQ** to consume and process internally generated data.
        - **External API Service** to fetch additional data from the Open F1 API for entity creation.
        - **TimescaleDB** to store historical time-series data.
        - **Frontend** via WebSockets to send real-time updates.
3. **External API Service**:
    - A dedicated service within the backend that communicates with the Open F1 API.
    - Responsible for fetching extra data required for entity creation (such as driver details).
    - The data fetched is used to enrich the entities managed by the backend.
4. **Message Broker (RabbitMQ)**:
    - Acts as the intermediary between the internal Data Generator and the

backend.
    - Ensures that internal data is efficiently queued and processed in real-time, facilitating smooth and scalable data handling.
5. **Data Generator**:
    - An internal module that simulates additional data about cars and drivers, complementing the data from the Open F1 API.
    - Sends this data to RabbitMQ, which then routes it to the backend for processing and storage.
6. **Database (TimescaleDB)**:
    - Used for time-series data storage, such as lap times and performance metrics, providing a foundation for historical data analysis.
7. **WebSocket Connection**:
    - Maintains a persistent, low-latency connection between the backend (Spring Boot) and the frontend (React), ensuring that real-time updates from internally generated data are pushed to users without requiring constant HTTP requests.



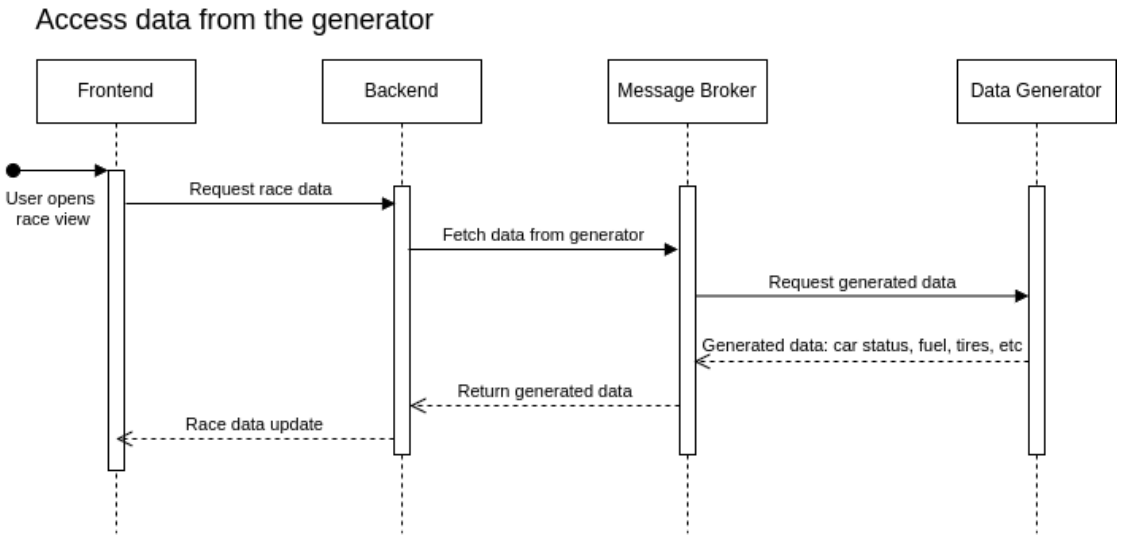
Module interactions

The interactions between these modules follow a well-defined sequence:

1. **Data Ingestion**:
    - The **Data Generator** continuously generates data about the race. This data is sent to the **Message Broker (RabbitMQ)**.
    - **RabbitMQ** processes these data streams and routes them to the **Backend (Spring Boot)**.
    - Simultaneously, the backend (Spring Boot) communicates with the external Open F1 API through the **External API Service** to retrieve additional data, such as driver details, needed for entity creation or race insights.
2. **Backend Processing**:
    - The **Spring Boot** backend consumes the data from **RabbitMQ**, processes it, and stores relevant parts in the **TimeScaleDB** database.
    - For time-sensitive internal data, the backend pushes updates directly to the **Frontend (React)** through **WebSockets**, ensuring real-time data visualization.
3. **Real-time Updates**:
    - The frontend receives real-time updates from the backend via a WebSocket connection, which includes both internally generated data from the Data Generator and any relevant data from the external Open F1 API.
4. **Data Storage**:
    - The backend stores time-series data (e.g., lap times, positions, ...) in **TimeScaleDB**.

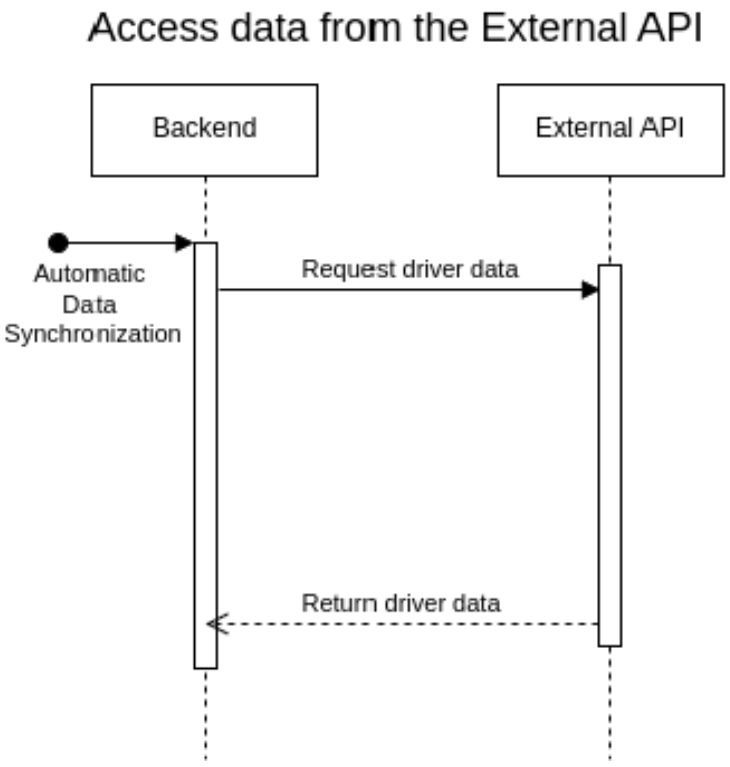
Sequence Diagrams

The sequence diagram titled "Access data from the generator" shows the interaction between the Frontend, Backend, Message Broker, and Data Generator to provide real-time race data to the user. When the user opens the race view, the frontend sends a request to the backend and then it requests data through the message broker. The message broker communicates with the data generator to retrieve said information, such as car status, fuel levels, and tire conditions. This generated data is then sent back through the message broker to the backend, which updates the frontend with the latest race information, ensuring the user receives up-to-date race details effectively.

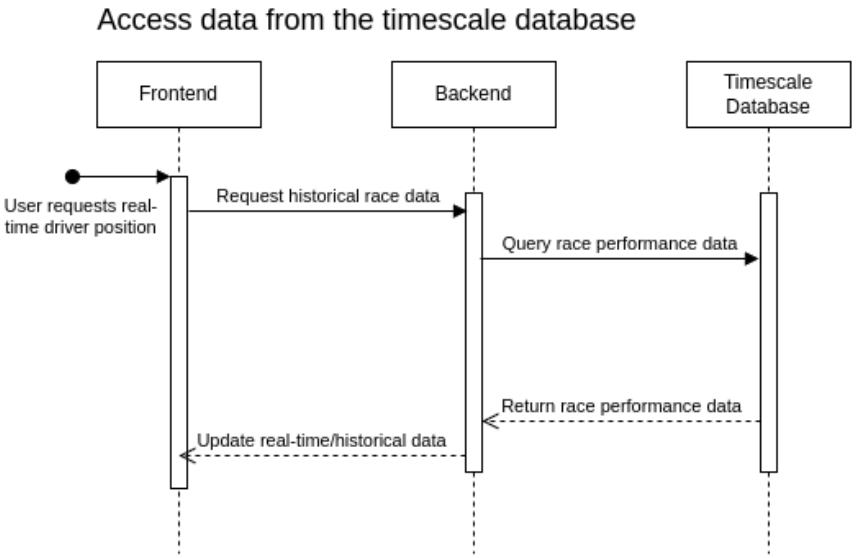


The sequence diagram titled "Access data from the External API" illustrates the automatic data synchronization flow between the **Backend** and an **External API**. However, it aligns with a similar process triggered when a user requests specific information, such as drivers' names or race data, via the frontend.

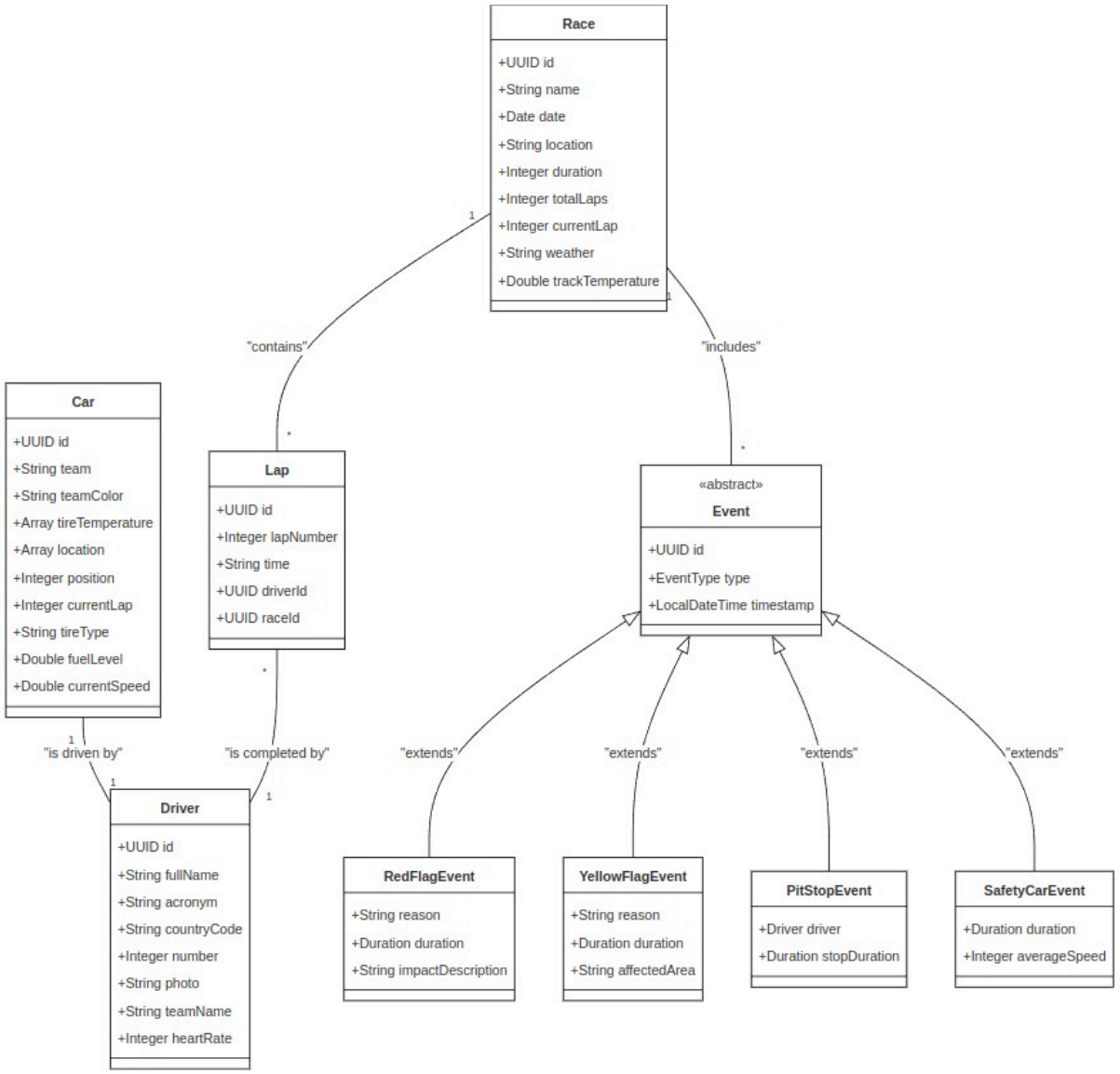
When the user initiates a request from the frontend, this triggers the backend to send a "Request driver data" to the **External API**. The **External API** processes this request and returns the required driver or race data. The backend then relays this data back to the frontend, enabling the user to view the latest information.



The sequence diagram titled "Access data from the timescale database" illustrates the process of obtaining historical race data or real-time driver positions. When the user requests this information from the frontend, it sends a request to the backend. The backend then queries the timescale database for race performance data. Once the database returns the requested data, the backend updates the frontend with the real-time or historical race data, ensuring that users can view accurate race information based on their request.



4 Information perspetive



This UML class diagram outlines the RaceFlow domain, with key classes for **Race**, **Car**, **Driver**, **Lap**, and **Event**. The **Race** class includes attributes like name, date, and location, and contains multiple **Lap** and **Event** entries. The **Car** class represents a Formula 1 car, which is linked one-to-one with a **Driver**. The **Driver** class holds details such as name, team, and heart rate, and is associated with a specific car. The **Lap** class represents a completed lap within a race, connected to both a **Driver** and a **Race**. Finally, the **Event** class is abstract, with subclasses like **RedFlagEvent**, **YellowFlagEvent**, **PitStopEvent**, and **SafetyCarEvent**, each representing different types of race incidents. This structure supports real-time race tracking and analysis by organizing and managing critical race data.

5 References and resources

<document the key components (e.g.: libraries, web services) or key references (e.g.: blog post) used that were really helpful and certainly would help other students pursuing a similar work>