UNIVERSITÀ
DEGLI STUDI
DE L'AQUILA

DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Project Documentation

2023/2024

Course: **Service Oriented Software Engineering**

University of L'Aquila

Prof. *Marco Autili and* ***Doc.*** *Gianluca Filippone*

L'Aquila Smart Road

Adam Bouafia		Matricula: 293137		*adam.bouafia@student.univaq.it*

Summary

Domain

L'Aquila Smart Road Platform is a web-based application designed to detect and track high-speed road violations in real-time.

Built with Spring Boot for the backend and Angular with Bootstrap for the frontend,

it provides efficient traffic management through real-time notifications, comprehensive reporting, and easy access for vehicle owners to view their violation history

Requirements

Functional Requirements

#	Functional Requirement	Priority
1	**Authentication** (login and registration): **FR1.1:** Users must be able to register and create an account. **FR1.2:** Users must be able to log in to the system using their credentials. **FR1.3:** The system must provide different levels of access (e.g., admin, regular user) based on user roles.	**MEDIUM**
2	**Vehicle Management** **FR2.1:** Users must be able to add a new vehicle to the system. **FR2.1.1:** Vehicle information should include registration number, brand, fiscal power, model, and owner details. **FR2.2:** Users must be able to view details of all registered vehicles. **FR2.3:** Users must be able to update the details of an existing vehicle. **FR2.4:** Users must be able to delete a vehicle from the system	**HIGH**
3	**Owner Management** **FR3.1:** Users must be able to add a new vehicle owner. **FR3.1.1:** Owner information should include ID, name, date of birth, and email. **FR3.2:** Users must be able to view details of all vehicle owners. **FR3.3:** Users must be able to update the details of an existing owner. **FR3.4:** Users must be able to delete an owner from the system.	**HIGH**
4	**Monitor Management** **FR4.1:** Users must be able to add a new radar to the system. **FR4.1.1:** Monitor information should include ID, maximum speed limit, and coordinates (longitude and latitude). **FR4.2:** Users must be able to view details of all Monitors. **FR4.3:** Users must be able to update the details of an existing Monitor. **FR4.4:** Users must be able to delete a Monitor from the system.	**HIGH**

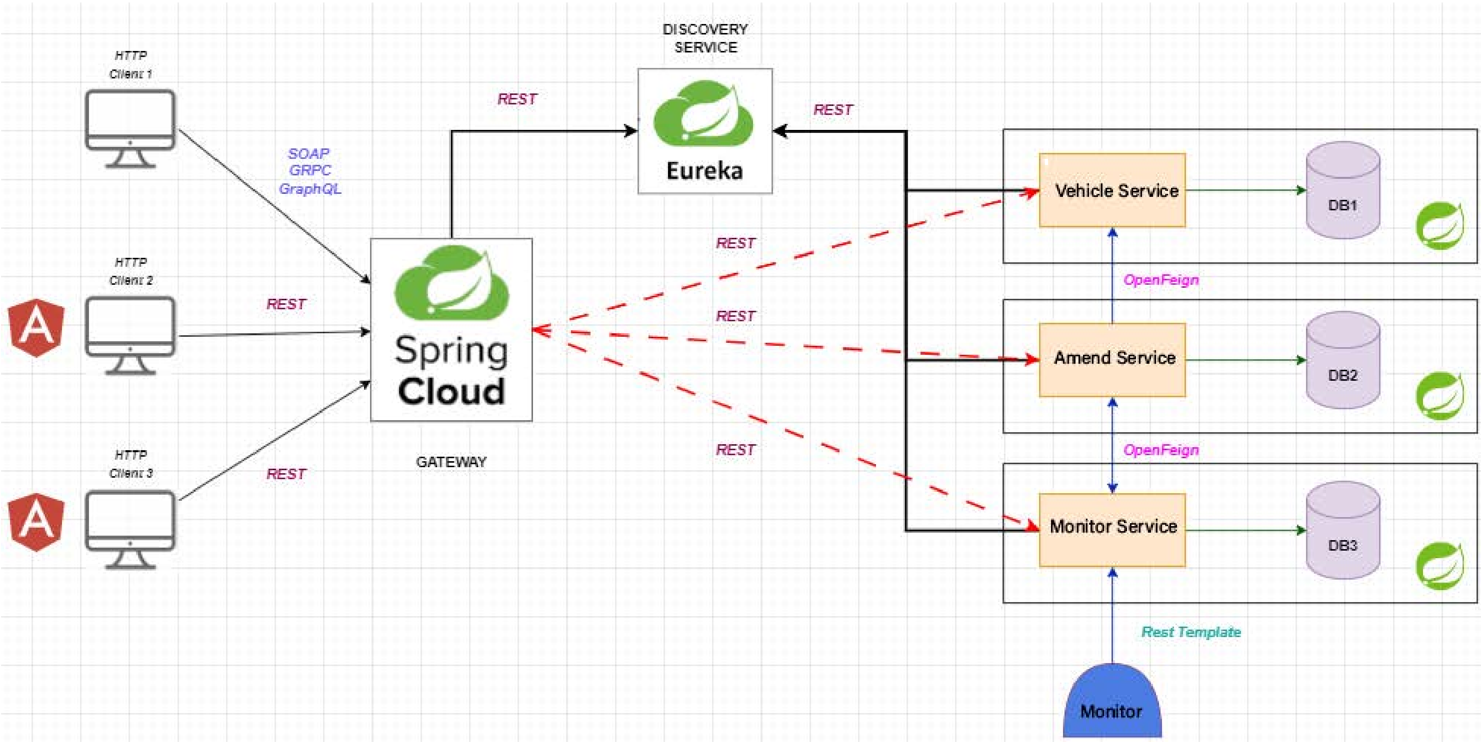
#	Functional Requirement	Priority
	Amends Management **FR5.1:** The system must record speeding violations detected by monitors. **FR5.1.1:** Amend information should include ID, date, radar number, vehicle registration number, vehicle's speed, radar's maximum speed limit, and fine amount. **FR5.2:** Users must be able to view details of all Amends. **FR5.3:** Users must be able download Amends in PDF .	**HIGH**
	Data Generation **FR6.1:** The system must generate reports on monitor violations. **FR6.1.1:** Reports should include statistics such as the number of violations, total fines, etc.	**HIGH**

Non Functional Requirements

- Load balancers on services/prosumers are subject to numerous requests in order to improve performance.

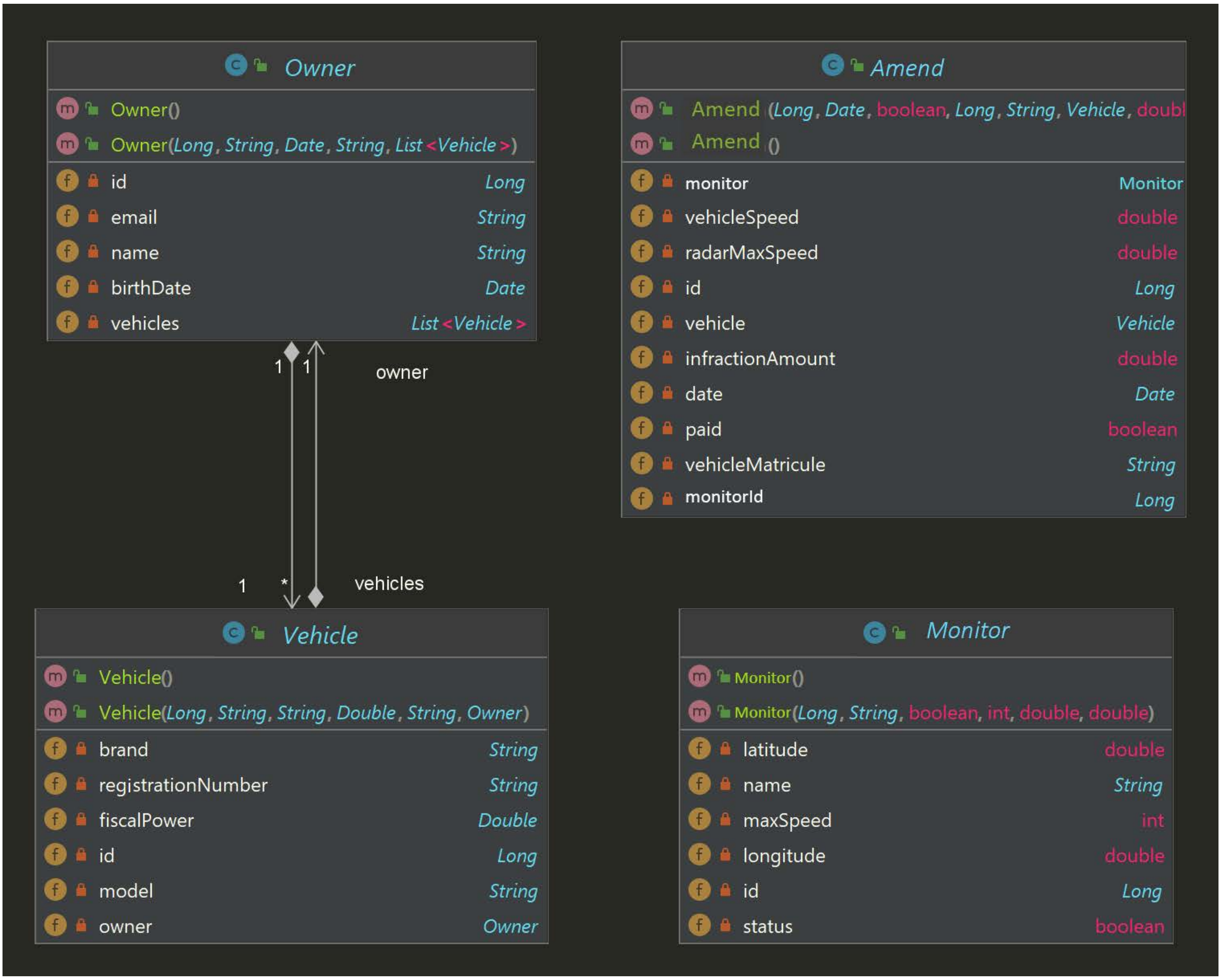
Architecture

Component Diagram



System Architecture - Component Diagram

Class Diagram



Class Diagram

Relationships:

Owner to Vehicle: One-to-many relationship, where an owner can own multiple vehicles.

Vehicle to Owner: Many-to-one relationship, where each vehicle is owned by one owner.

Amend to Monitor: Many-to-one relationship, where multiple amendments can be associated with a single monitor.

Amend to Vehicle: Many-to-one relationship, where multiple amendments can be associated with a single vehicle.

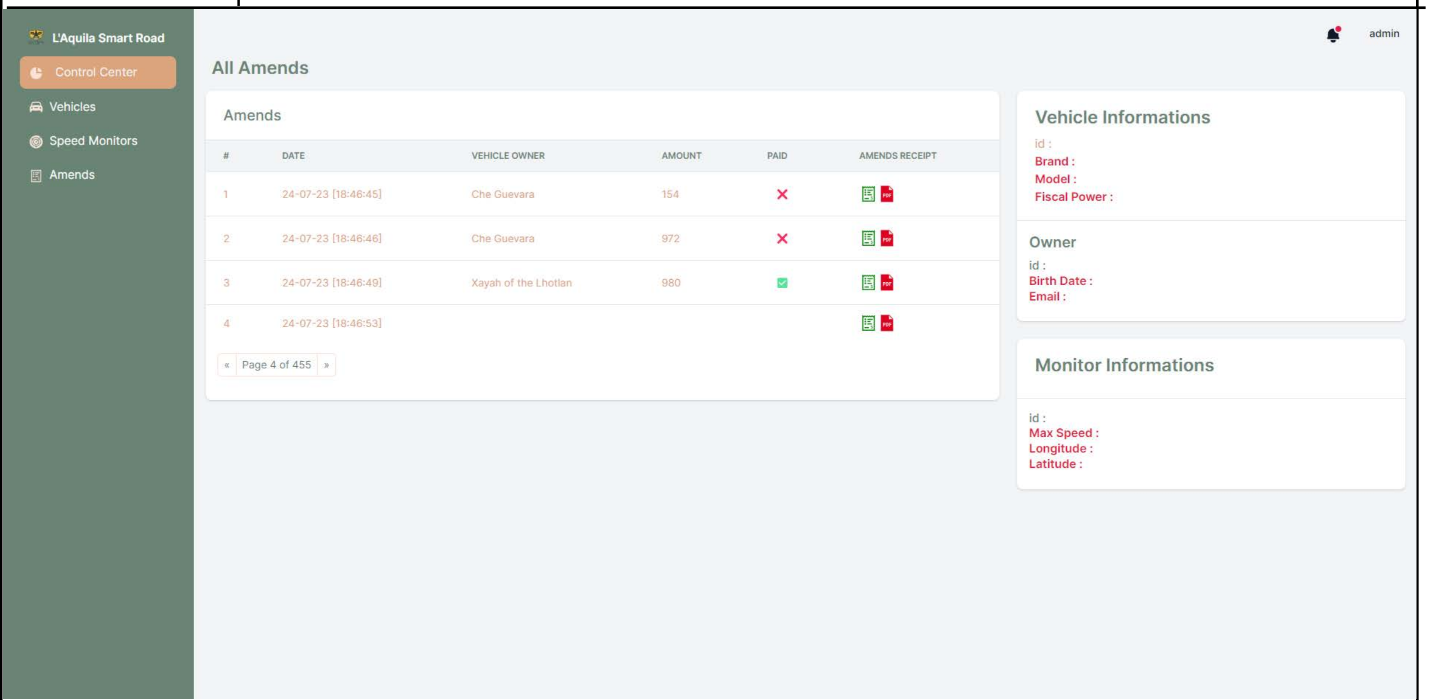
Services Description

The services to be implemented in the system are those shown in the architecture and in this section, they will be described in detail.

Service Name	**Type**	**Protocol**	**Sync / Async**	**Exposed operations**
Amend Service	*Microservice* *Prosumer*	**REST**	Sync	- *Retrieve amend* - *Add new amend* - *Update existing amend* - *Delete amend*
Data Generator Service	*Microservice* *Provider*	**REST**	Sync	- *Generate random Amend Data*
Discovery Service	*Microservice*	**Eureka**	Sync	- *Service registration* - *Service discovery*
Gateway Service	*Gateway*	**REST**	Sync	- *Route requests to appropriate microservices* *API Aggregation*
Monitors Service	*Microservice* *Prosumer*	**gRPC**	Async	- *Generate speed violations* - *Retrieve monitor data* - *Add new monitors* - *Update existing monitors* - *Delete monitors*
Vehicles Service	*Microservice* *Provider*	**REST** **SOAP** **gRPC** **GraphQL**	Sync Async	- *Retrieve vehicle details* - *Add new vehicles* - *Update existing vehicles* - *Delete vehicles* - *Retrieve owner details (GraphQL)* - *Add new owner (gRPC)* - *Update existing owner (gRPC)* - *Delete owner (SOAP)*

Project Demo And Detailed Service Descriptions

Functionality	Description
Amend Service	**Service Type:** Prosumer **Protocol:** REST (synchronous) **Description:** This service handles amendments related to vehicle data. It interacts with both Monitor and Vehicle services to gather necessary data for amendments. **Operations:** **GET /**amend: Retrieves a list of all amendments. **POST /**amend: Creates a new amendment. **PUT /**amend/{id}: Updates an existing amendment. **DELETE /**amend/{id}: Deletes an amendment.



<table>
<tr>
<td>Data Generator Service</td>
<td>

Service Type: Provider

Protocol: REST (synchronous) **Description:** Generates random data for Amends for testing and simulation purposes.

Operations:

POST /generate/data: Generates random new data.

<pre>

Monitor Name : R2
Vehicle RN : 17643a4a-317a-4e50-91b8-36de1ff90326
Vehicle Speed : 153.0

Monitor Name : R4
Vehicle RN : 131c068e-5088-43dd-8ed9-d97df26597f5
Vehicle Speed : 129.0

Monitor Name : R1
Vehicle RN : 546b4693-e39b-4c64-b3dc-2850fa17bbb0
Vehicle Speed : 178.0

Monitor Name : R4
Vehicle RN : 131c068e-5088-43dd-8ed9-d97df26597f5
Vehicle Speed : 217.0

</pre>

</td>
</tr>
<tr>
<td>Monitors Service</td>
<td>

Service Type: Prosumer

Protocol: gRPC (asynchronous) **Description:** Manages monitor data and generates speed violations using gRPC for efficient communication.

Operations:

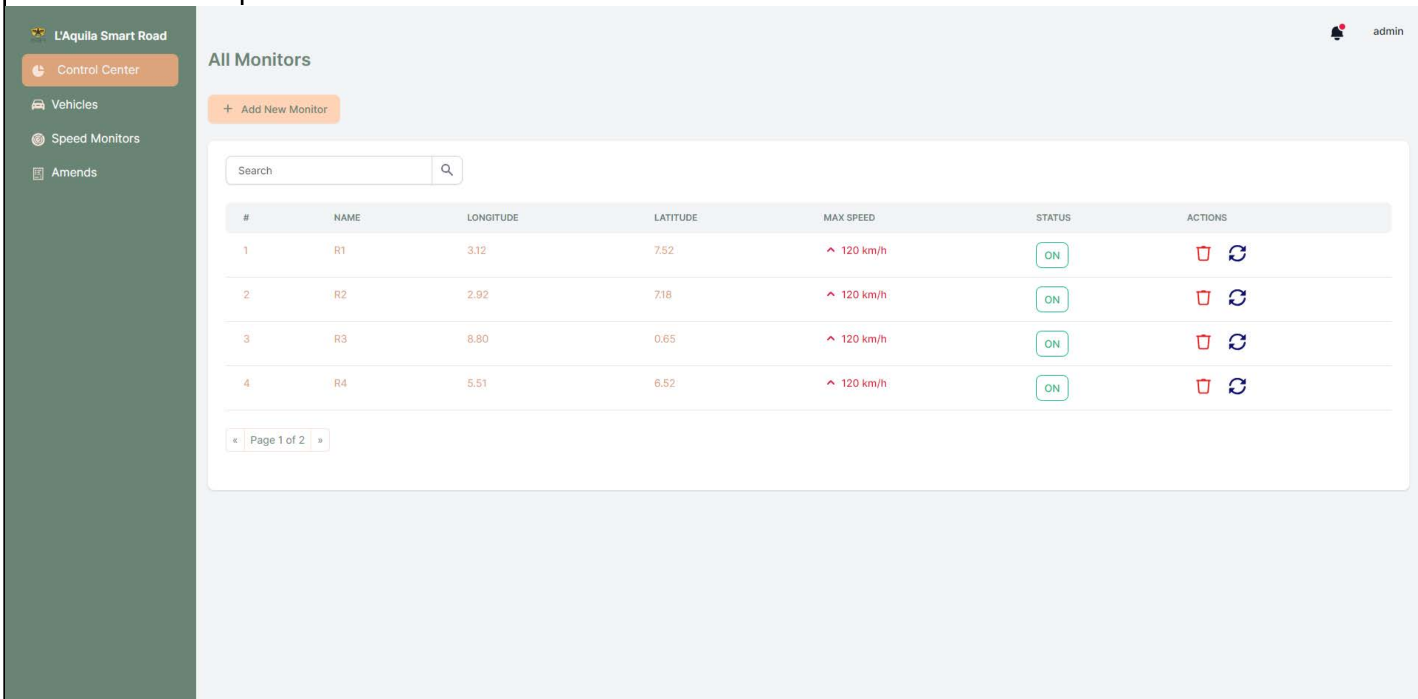
rpc GenerateSpeedViolations (GenerateSpeedViolationRequest) returns (stream SpeedViolation): Generates speed violations asynchronously.

GET /monitors: Retrieves a list of all monitors. **POST /**monitors: Adds a new monitor.

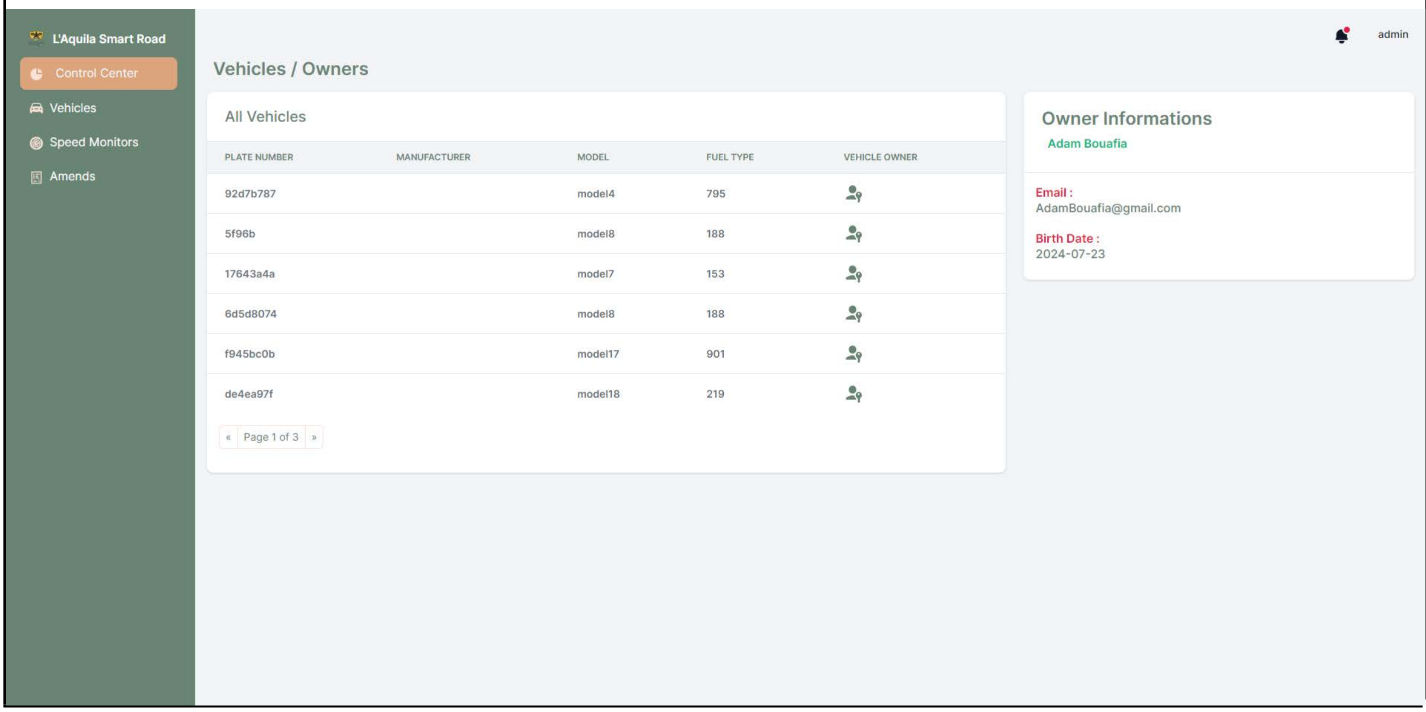
PUT /monitors/{id}: Updates an existing monitor.

DELETE /monitors/{id}: Deletes a monitor.

</td>
</tr>
</table>



Vehicles Service	**Service Type:** Provider **Protocol:** REST, SOAP, gRPC, GraphQL **(both synchronous and asynchronous)** **Description:** Manages vehicle and owner data using multiple protocols to demonstrate various integration methods. Operations: **REST:** **GET /vehicles:** Retrieves a list of all vehicles. **POST /vehicles:** Adds a new vehicle. **PUT /vehicles/ id :** Updates an existing vehicle. **DELETE /vehicles/ id :** Deletes a vehicle. **SOAP:** **getOwnerById:** Retrieves owner details by ID. **getOwners:** Retrieves a list of all owners. **gRPC:** **rpc getOwner(GetOwnerRequest) returns (GetOwnerResponse)**: Retrieves owner details asynchronously. **rpc listOwners(GetAllOwnersRequest) returns (GetAllOwnersResponse):** Lists all owners asynchronously. **rpc saveOwner(SaveOwnerRequest) r**eturns (SaveOwnerResponse): Saves owner details asynchronously. **GraphQL:** query g**etVehicles id, registrationNumber, brand** : Retrieves vehicle details using GraphQL. query **getOwner id, name, birthDate, email** : Retrieves owner details using GraphQL.



Conclusions

Meeting the project requirements

- ### Applying Microservices Architecture

  L'Aquila Smart Road Platform employs a microservices architecture, breaking down the application into multiple, loosely coupled services that perform specific business functions. Each microservice is independently deployable and scalable, which brings numerous benefits over a monolithic architecture.

- ### Benefits of Microservices Architecture

  Scalability: Each microservice can be scaled independently based on its demand. This leads to more efficient resource utilization compared to scaling an entire monolithic application.

  Resilience: Fault isolation ensures that a failure in one microservice does not bring down the entire application, enhancing system reliability.

  Agility: Smaller codebases enable faster development, testing, and deployment cycles. Teams can work on different microservices simultaneously without causing conflicts.

  Technology Diversity: Different microservices can be developed using different technologies that are best suited for their specific requirements.

  Maintainability: Microservices are easier to understand, modify, and extend due to their smaller and focused scope.

- ### Why We Used REST, SOAP, gRPC, and GraphQL

  To demonstrate our comprehensive understanding of API integration and to adhere to the specifications, we implemented four primary API architectures: REST, SOAP, gRPC, and GraphQL.

- **1. REST (Representational State Transfer):**

  Why: REST is widely used and easy to implement. It's suitable for CRUD operations and synchronous communication.
  Application: We used REST for most of our services for straightforward and stateless operations, exposing endpoints via HTTP.
  Enhancements: Swagger/OpenAPI was integrated to provide interactive API documentation, making it easier for developers to understand and consume the APIs.

- **2. SOAP (Simple Object Access Protocol):**

  Why: SOAP is a protocol that offers built-in error handling and supports ACID transactions. It's suitable for secure and reliable message exchanges.
  Application: SOAP was applied using Apache CXF for services that require strong contract and protocol compliance.
  Enhancements: Apache CXF simplifies the development of SOAP services, providing support for WSDL and other web service standards.

- **3. gRPC (Google Remote Procedure Call):**

  Why: gRPC provides high performance, low latency, and supports asynchronous communication. It's suitable for real-time communication and efficient data streaming.
  Application: gRPC was utilized for services that require high efficiency and bi-directional streaming.
  Enhancements: By using gRPC, we leveraged HTTP/2 for performance improvements and used Protocol Buffers for efficient data serialization.

- **4. GraphQL:**

  Why: GraphQL allows clients to request exactly the data they need, minimizing over-fetching and under-fetching of data. It's suitable for complex queries and hierarchical data structures.
  Application: GraphQL was used for services where clients need flexible and precise data queries, particularly for nested or relational data.
  Enhancements: GraphQL controllers were set up to provide query and mutation operations, making data retrieval more efficient and customizable.

- ## Applying Asynchronous Communication

  Asynchronous communication was a key requirement of the project, enabling services to perform tasks without blocking the main execution flow.

  ### gRPC:

  gRPC inherently supports asynchronous communication. For example, the**MonitorServiceGrpc** generates speed violations asynchronously using server streaming, allowing the client to process data as it arrives.

  ### Microservices Parallel Execution:

  Two prosumer services, such as the **MonitorService a**nd **AmendService,** perform their operations in parallel. They gather data concurrently and then synchronize before responding to the client. This ensures faster and more efficient data processing.

Used Technologies

spring boot

gRPC

Apache CXF

SOAP

NGINX

Swagger

Tomcat

Maven™

REST API