**USER AND VEHICLE DATA MANAGEMENT SYSTEM**

Spring Boot is a popular framework within the Java ecosystem that simplifies the development of Java-based applications, especially microservices. Microservices architecture involves breaking down an application into smaller, independently deployable services that work together to fulfil business requirements.

Spring Boot makes it easier to create stand-alone, production-grade Spring-based applications by providing out-of-the-box solutions to many common problems in Java enterprise development. It offers features like auto-configuration, which reduces the need for explicit configuration, and embedded servers, allowing applications to be packaged as executable JAR files.

**Project Overview:**

The User Profile (Microservice 1) is a component, designed to manage user and vehicle information. It serves as a backend service responsible for handling CRUD operations on users and vehicles, storing the data in a MySQL database. This documentation provides a detailed overview of the functionalities, API endpoints, data models, and integration mechanisms of the User profile.

Microservice 2 (Vehicle discovery) requires access to vehicle information stored in Microservice 1's MySQL database. To facilitate this, Microservice 2 initiates requests to Microservice 1, providing the mobile number associated with each vehicle. Microservice 1 processes these requests, retrieves the corresponding vehicle details from the MySQL database, and sends the data back to Microservice 2. Upon receiving the vehicle information, Microservice 2 stores it in its MongoDB database, including the associated mobile number.

The User profile (Microservice 1) plays a crucial role in managing user and vehicle data within the system. Through its APIs and integration mechanisms, it ensures efficient storage, retrieval, and manipulation of user and vehicle information, thereby contributing to the overall functionality and performance of the system.

**Microservice 1- User Profile:**

The User Management System serves as a core component within the system architecture, tasked with managing user and vehicle information efficiently. Its primary purpose is to provide a seamless and secure platform for handling CRUD operations on users and vehicles, ensuring data integrity and reliability.

**Data Models:**

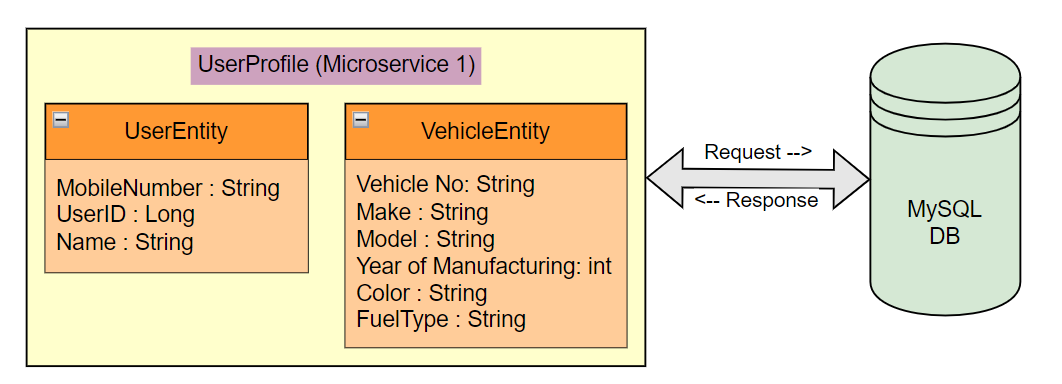
* **User Model:**
  + UserId (Long): Represents the Id of the user.
  + Name (String): Represents the name of the user.
  + Mobile Number (String): Unique identifier for each user.
* **Vehicle Model:**
  + Vehicle Number (String): Unique identifier for each vehicle.
  + Make (String): Manufacturer of the vehicle.
  + Model (String): Model name of the vehicle.
  + Fuel Type (String): Type of fuel used by the vehicle.
  + Color (String): Color of the vehicle.
  + Year of Manufacturing (Integer): Year in which the vehicle was manufactured.

**Functionalities:**

* + Manages user and vehicle information in a MySQL database.
  + Exposes RESTful APIs for CRUD operations on users and vehicles.
  + User data includes name and mobile number, while vehicle data includes vehicle number, make, model, fuel type, color, and year of manufacturing.

**Data Flow Diagram:**

The diagram depicts the interaction between a User Management System (Microservice 1) and an external system (MySQL database).



The arrows indicate the flow of data between the microservice and the database. The label “Request” goes from the microservice to the database, signifying the microservice requesting data from the database. The label “Response” goes from the database to the microservice, indicating the database sending data back to the microservice.

**Microservice 2- Vehicle Discovery:**

Microservice 2 (Vehicle Discovery) requires access to vehicle information stored in Microservice 1's MySQL database. It Provide a comprehensive solution for managing vehicle information alongside user data. It enables CRUD operations on vehicles to support functionalities such as adding, updating, and deleting vehicles. It ensures consistency and accuracy in storing and retrieving vehicle details.

**Data Models:**

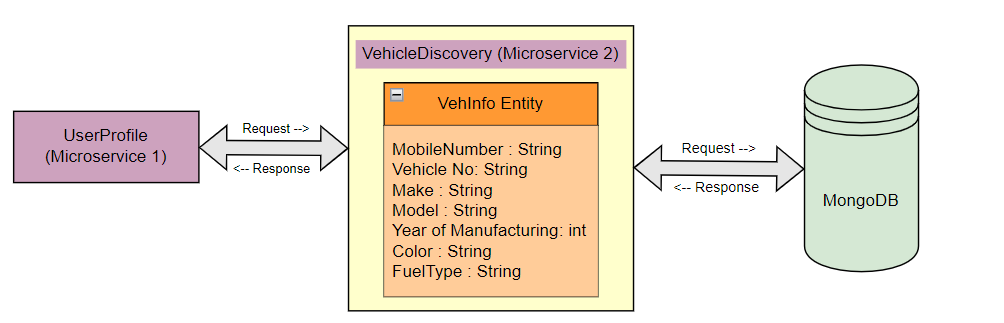
* **Vehicle Model**
* Vehicle number
* Make
* Model
* Fuel type
* Color
* Year of manufacturing
* Associated mobile number (from User Profile System)

**Functionalities:**

* Provides access to vehicle information stored in a MongoDB database.
* Exposes an API dedicated to accessing vehicle details.

**Data Flow Diagram:**

This diagram depicts the illustrates how a microservice named “Vehicle Discovery (Microservice 2)” retrieves information about a vehicle from another microservice named “User Profile (Microservice 1)”.

****

The arrow indicates the flow of data that is requesting to fetch vehicle information from microservices 1 (user profile) and the response provided from microservices 1 stored it in MongoDB database. For Get endpoint, the request is sent to database to retrieves the vehicle information stored in MongoDB database.

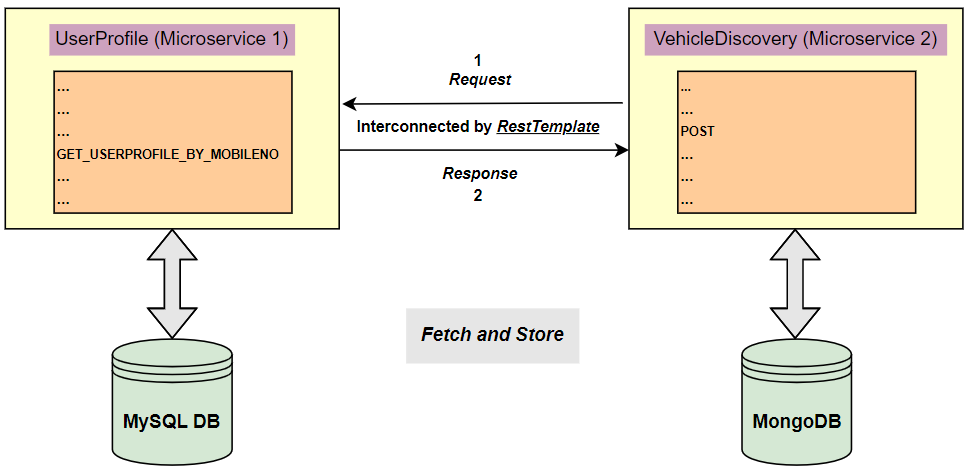
**Interconnection:**

The User profile (Microservice 1) and the Vehicle discovery (Microservice 2) are both connected to MySQL databases for storing data. Microservice 2 additionally utilizes a MongoDB database for its operations.

**Integration:**

To fulfil the requirement of fetching vehicle information from Microservice 1 and storing it in Microservice 2's MongoDB database:

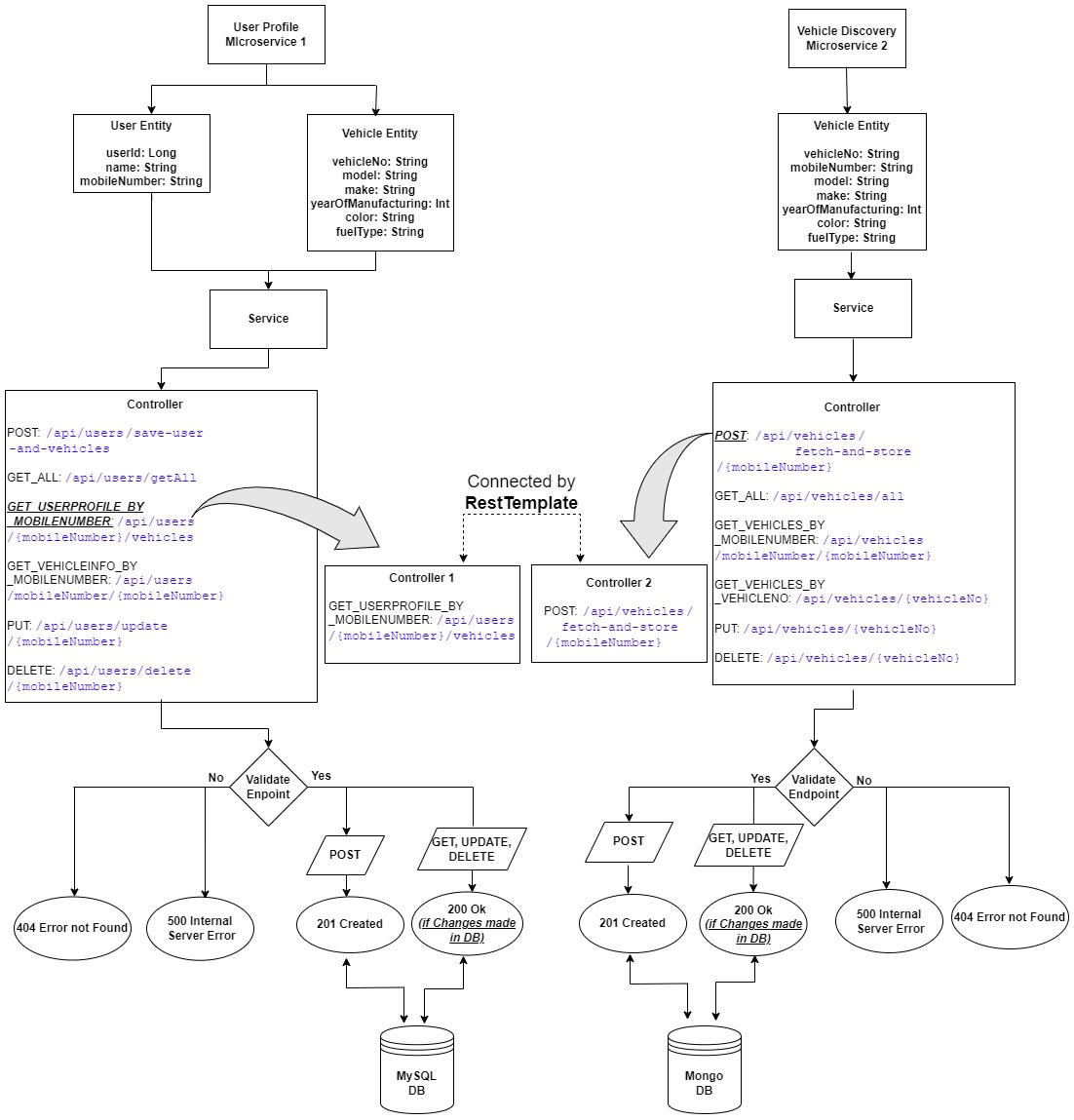
* Microservice 2 fetches vehicle details from Microservice 1 using the mobile number associated with each vehicle.
* Upon retrieval, Microservice 2 saves the fetched vehicle data along with the corresponding mobile number in its MongoDB database.



Sequence of Interactions:

* An arrow labelled "GET\_USERPROFILE\_BY\_MOBILENO" originates from User Profile (Microservice 1) and points to MySQL Database. This signifies that User Profile retrieves user information from the MySQL database using the mobile number.
* An arrow labelled "Response" originates from MySQL Database and points to User Profile (Microservice 1). This indicates that the requested user information is sent back to User Profile.
* An arrow labelled "Request" originates from Vehicle Discovery (Microservice 2) and points to User Profile (Microservice 1). This signifies that Vehicle Discovery sends a request to User Profile to retrieve vehicle information by passing associated mobile number.
* Rest Template is used to make the interservice communication between User Profile (Microservice 1) and Vehicle Discovery (Microservice 2). REST is a architectural style for designing networked applications and Rest Template is a Java class that simplifies making REST calls.
* An arrow labelled "Response" originates from User Profile (Microservice 1) and points to Vehicle Discovery (Microservice 2). This specifies that the HTTP method used in the request is POST, which is typically used to send data to the server.
* An arrow labelled "Fetch and Store" originates from Vehicle Discovery (Microservice 2). This likely represents the internal process of fetching vehicle data from an external source and storing it in the MongoDB database.

**Flow Diagram:**

****

**Technologies Used:**

Here's a list of technologies and frameworks used in the project along with their versions:

1. Spring Boot:
   * Version: 2.5.4
2. Spring Data JPA:
   * Version: 2.5.4
3. Spring Data MongoDB:
   * Version: 2.5.4
4. MySQL Database:
   * Version: 8.0.26
5. MongoDB Database:
   * Version: 5.0.2
6. Java Development Kit (JDK):
   * Version: 11 or higher
7. Apache Maven:
   * Version: 3.8.2
8. Database Drivers (e.g., MySQL Connector/J, MongoDB Java Driver):
   * MySQL Connector/J: Version 8.0.26
   * MongoDB Java Driver: Version 4.3.2
9. Operating System (e.g., Linux, Windows):
   * Windows: Windows 10

10. Integrated Development Environment (IDE):

* + Spring Tool Suite 4

1. Dependency Management (e.g., Apache Maven, Gradle):

* Apache Maven: Version 3.8.2

1. Web Server:

* Apache Tomcat: Version 9.0.53

1. Web Service Protocol (e.g., REST, SOAP):

* REST (Representational State Transfer)

**System Requirements:**

Before setting up the development environment, ensure that your system meets the following requirements:

1. Operating System: Windows 10.
2. Java Development Kit (JDK): Version 11 or higher.
3. Apache Maven: Version 3.8.2 or higher.
4. MySQL Database: Version 8.0.26 or higher.
5. MongoDB Database: Version 5.0.2 or higher.
6. Integrated Development Environment (IDE): Spring tool suite 4.

**Dependencies:**

The project dependencies include:

* Spring Boot: Version 2.5.4
* Spring Data JPA: Version 2.5.4
* Spring Data MongoDB: Version 2.5.4
* MySQL Connector/J: Version 8.0.26
* MongoDB Java Driver: Version 4.3.2

**Fetch Vehicle Data from MySQL and Store it in MongoDB API:**

This Java method retrieves only the vehicle information associated with a given mobile number from a MySQL database and store it into MongoDB.

**@Transactional**

**public** List<VehicleMongo> fetchAndStoreVehicles(String mobileNumber) {

ResponseEntity<List<Vehicle>> responseEntity = restTemplate.exchange(

"http://localhost:8080/api/users/{mobileNumber}/vehicles",

HttpMethod.***GET***,

**null**,

**new** ParameterizedTypeReference<List<Vehicle>>() {},

mobileNumber

);

List<Vehicle> vehicles = responseEntity.getBody();

List<VehicleMongo> vehiclesMongo = vehicles.stream()

.map(v -> {

VehicleMongo vehicleMongo = **new** VehicleMongo();

vehicleMongo.setVehicleNo(v.getVehicleNo());

vehicleMongo.setModel(v.getModel()) vehicleMongo.setYearOfManufacturing(v.getYearOfManufacturing();

vehicleMongo.setMobileNumber(mobileNumber);

vehicleMongo.setMake(v.getMake());

vehicleMongo.setColor(v.getColor());

vehicleMongo.setFuelType(v.getFuelType());

**return** vehicleMongo;

})

.collect(Collectors.*toList*());

vehicleMongoRepository.saveAll(vehiclesMongo);

**return** vehiclesMongo;

}

1. Method Signature:

This method is marked with the @Transactional annotation, implying that the entire method execution will be wrapped in a transactional boundary, ensuring atomicity, consistency, isolation, and durability of the database operations within this method.

The method takes a mobileNumber parameter and returns a List<VehicleMongo>.

1. REST API Call:

This block initiates an HTTP GET request to a RESTful API endpoint using Spring's RestTemplate.

The endpoint URL is dynamic, as it contains a path variable {mobileNumber}.

The response received is of type ResponseEntity<List<Vehicle>>, indicating that the response contains a list of Vehicle objects.

1. Mapping to MongoDB Entities:

The method extracts the list of Vehicle objects from the ResponseEntity.

It then maps each Vehicle object to a corresponding VehicleMongo object, presumably a MongoDB entity.

For each Vehicle, a VehicleMongo object is created and populated with corresponding attributes.

This process essentially transforms Vehicle objects to VehicleMongo objects, possibly to store them in a MongoDB database.

1. Saving to MongoDB:

After the mapping, the method saves the list of VehicleMongo objects to a MongoDB database using a repository (vehicleMongoRepository).

1. Return:

Finally, the method returns the list of VehicleMongo objects that were saved to the MongoDB database.

Overall, this method fetches a list of vehicles associated with a given mobile number from a remote API, transforms them into MongoDB entities, and saves them to a MongoDB database within a transactional context.

**Sample API Request and Response:**

**User Management Microservice**

1. Create User Profile API:

* Request

POST <http://localhost:8080/api/users/save-user-and-vehicles>

Content-Type: application/json

{

    "user": {

        "id": 25,

        "name": "Abinaya",

        "mobileNumber": "1234567890"

    },

    "vehicles": [

        {

            "vehicleNo": "KA01AB2505",

            "make": "Mahindra",

            "model": "Thar",

            "yearOfManufacturing": 2023,

            "color": "black",

            "fuelType": "petrol"

        }

]

}

* Response

[

    {

        "vehicleNo": " KA01AB2505",

        "user": {

            "id": 25,

            "name": " Abinaya ",

            "mobileNumber": "1234567890"

        },

        "model": " Thar ",

        "yearOfManufacturing": 2023,

        "make": " Mahindra ",

        "color": "black",

        "fuelType": "petrol"

    }

]

1. Get All User Profile API:

* Request

GET <http://localhost:8080/api/users/getAll>

Content-Type: application/json

* Response

[

    {

        "vehicleNo": "KA01AB2505",

        "user": {

            "id": 25,

            "name": "Abinaya",

            "mobileNumber": "1234567890"

        },

        "model": "Thar",

        "yearOfManufacturing": 2023,

        "make": "Mahindra",

        "color": "black",

        "fuelType": "petrol"

    },

    {

        "vehicleNo": "TN45BD9505",

        "user": {

            "id": 30,

            "name": "Jai",

            "mobileNumber": "9876543210"

        },

        "model": "Nexon",

        "yearOfManufacturing": 2024,

        "make": "Tata",

        "color": "Grey",

        "fuelType": "petrol"

    }

]

1. Get Specific User Profile By Mobile Number API:

* Request

GET <http://localhost:8080/api/users/1234567890/vehicles>

Content-Type: application/json

* Response

[

    {

        "vehicleNo": " KA01AB2505",

        "user": {

            "id": 25,

            "name": " Abinaya ",

            "mobileNumber": "1234567890"

        },

        "model": " Thar ",

        "yearOfManufacturing": 2023,

        "make": " Mahindra ",

        "color": "black",

        "fuelType": "petrol"

    }]

1. Get only Vehicle Information By Mobile Number API:

* Request

GET <http://localhost:8080/api/users/mobileNumber/1234567890>

Content-Type: application/json

* Response

[

    {

        "vehicleNo": " KA01AB2505",

        "model": "Thar",

        "yearOfManufacturing": 2023,

        "make": "Mahindra",

        "color": "black",

        "fuelType": "petrol"

    }

]

1. Update User Profile By Mobile Number API:

* Request

PUT <http://localhost:8080/api/users/update/1234567890>

Content-Type: application/json

{

    "user": {

        "id": 25,

        "name": "Abinaya Vasudevan",

        "mobileNumber": "1234567890"

    },

    "vehicles": [

        {

            "vehicleNo": "KA01AB2505",

            "make": "Mahindra",

            "model": "Thar",

            "yearOfManufacturing": 2023,

            "color": "black",

            "fuelType": "petrol"

        }

]

}

* Response

{

        "id": 25,

        "name": "Abinaya Vasudevan",

        "mobileNumber": "1234567890"

     }

1. Delete User Profile By Mobile Number API:

* Request

DELETE <http://localhost:8080/api/users/delete/1234567890>

Content-Type: application/json

* Response

User deleted successfully

**Vehicle Discovery System:**

1. Fetch and Store Vehicle Information API:

* Request

POST <http://localhost:8105/api/vehicles/fetch-and-store/1234567890>

Content-Type: application/json

* Response

[

    {

        "vehicleNo": " KA01AB2505",

        "mobileNumber": "1234567890",

        "model": "Thar",

        "make": "Mahindra",

        "color": "black",

        "fuelType": "petrol",

        "yearOfManufacturing": 2023

    }

]

1. Get All Vehicle Information API:

* Request

GET <http://localhost:8105/api/vehicles/all>

Content-Type: application/json

* Response

[

    {

        "vehicleNo": " KA01AB2505",

        "mobileNumber": "1234567890",

        "model": "Thar",

        "make": "Mahindra",

        "color": "black",

        "fuelType": "petrol",

        "yearOfManufacturing": 2023

    },

    {

        "vehicleNo": " TN45BD9505",

        "mobileNumber": "9876543210",

        "model": "Nexon",

        "make": "Tata",

        "color": "black",

        "fuelType": "petrol",

        "yearOfManufacturing": 2024

    }

]

1. Get Vehicle Information along with user mobile number by Mobile Number

* Request

GET <http://localhost:8105/api/vehicles/mobileNumber/1234567890>

Content-Type: application/json

* Response

[

    {

        "vehicleNo": " KA01AB2505",

        "mobileNumber": "1234567890",

        "model": "Thar",

        "make": "Mahindra",

        "color": "black",

        "fuelType": "petrol",

        "yearOfManufacturing": 2023

    }

]

1. Get only Vehicle Information without Mobile Number by Vehicle No API:

* Request

GET <http://localhost:8105/api/vehicles/KA01AB2505>

Content-Type: application/json

* Response

{

    "vehicleNo": " KA01AB2505",

    "model": "Thar",

    "make": "Mahindra",

    "color": "black",

    "fuelType": "petrol",

    "yearOfManufacturing": 2023

}

1. Get only Vehicle No Associated with Mobile Number

* Request

GET <http://localhost:8105/api/vehicles/getVehicleNo/1234567890>

Content-Type: application/json

* Response

[

    {

        "vehicleNo": " KA01AB2505"

    }

]

1. Update Vehicle Information By Mobile Number

* Request

PUT <http://localhost:8105/api/vehicles/1234567890>

Content-Type: application/json

{

    "vehicleNo": " KA01AB2505",

    "mobileNumber": "1234567890",

    "model": "Thar",

    "make": "Mahindra",

    "color": "black",

    "fuelType": "petrol",

    "yearOfManufacturing": 2023

}

* Response

Vehicle updated successfully

1. Delete Vehicle Information by Mobile Number

* Request

DELETE <http://localhost:8105/api/vehicles/1234567890>

Content-Type: application/json

* Response

Vehicle with mobile number 1234567890 has been deleted successfully.

**Error Handling and Status Codes**

Here's a description of each error handling scenario and the corresponding status codes:

1. **Empty Input Exception:**
   * Description: Thrown when the userVehicleRequest is null or either the user or vehicles list is null. Also thrown when any mandatory fields such as name, mobile number, model, make, color, or fuel type are empty or null.
   * Status Code: 400 Bad Request
2. **Duplicate Mobile Number Exception:**
   * Description: Thrown when the mobile number already exists in the database for a different user.
   * Status Code: 400 Bad Request
3. **Invalid Name Exception:**
   * Description: Thrown when the user's name contains invalid characters (non-alphabetic).
   * Status Code: 400 Bad Request
4. **Invalid Mobile Number Digit Exception:**
   * Description: Thrown when the mobile number contains non-numeric characters.
   * Status Code: 400 Bad Request
5. **Invalid Mobile Number Exception:**
   * Description: Thrown when the mobile number is not 10 digits long.
   * Status Code: 400 Bad Request
6. **Invalid Year of Manufacturing Type Exception:**
   * Description: Thrown when the year of manufacturing for a vehicle is not of type Integer.
   * Status Code: 400 Bad Request
7. **Invalid Year of Manufacturing Exception:**
   * Description: Thrown when the year of manufacturing for a vehicle is less than or equal to zero.
   * Status Code: 400 Bad Request
8. **Duplicate Vehicle No Exception:**
   * Description: Thrown when the vehicle number already exists in the database.
   * Status Code: 400 Bad Request
9. **No Such Element Exception:**

* Description: Thrown when no user is found with the provided mobile number in the userRepository. This indicates that there are no vehicles associated with the given mobile number.
* Status Code: 404 Not Found

1. **Invalid Update Exception:**

* Description: Thrown when the ID of the existing user does not match the ID of the user in the request body. This indicates that an attempt was made to update a user with incorrect identification details.
* Status Code: 400 Bad Request

**Database Configuration:**

MongoDB Configuration

server.port=8105

spring.data.mongodb.host=localhost

spring.data.mongodb.port=27017

spring.data.mongodb.database=Micro2

MySQL Configuration

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/micro1

spring.datasource.username=root

spring.datasource.password=Abinaya@255

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=update

spring.main.allow-circular-references=true

1. MongoDB Configuration

* Host: The configuration specifies a local host (localhost) for the MongoDB server. This suggests that MongoDB is running on the same machine as the Spring Boot application.
* Port: The port configured for MongoDB is the default port (27017).
* Database: The application is configured to use a database named Micro2.

2. MySQL Configuration

* Driver Class: The configuration specifies the JDBC driver class for MySQL (com.mysql.cj.jdbc.Driver). This class enables communication between the application and the MySQL database.
* URL: The JDBC URL (jdbc:mysql://localhost:3306/microl) indicates that the MySQL server is also running on the local host (localhost) and uses the default port (3306) for connections. The database name configured here is microl.
* Username: The username for accessing the MySQL database is root.
* Password: The password for the MySQL database user is Abinaya@255.

3. JPA Configuration

* Dialect: The configuration sets the JPA database platform to org.hibernate.dialect.MySQL8Dialect. This line specifies that the application should use the Hibernate dialect for MySQL 8, which ensures compatibility between the application and the MySQL database version.
* Show SQL: The spring.jpa.show-sql property is set to true. This enables logging of SQL statements in the console, which can be helpful for debugging purposes.
* DDL Auto-Update: The property spring.jpa.hibernate.ddl-auto-update is included but commented out. This property specifies how Hibernate should handle the automatic creation, updating, or dropping of database tables in relation to the entity classes. Since it is commented out, the default behavior would likely apply.

4. Additional Configuration

* The configuration includes spring.main.allow-circular-references=true. This property might be used to address potential circular reference issues between beans in the application.

**Conclusion:**

In conclusion, this microservice architecture effectively handles the management of user and vehicle information in a distributed system. By leveraging Spring Boot and associated technologies, it provides a robust and scalable solution for storing, retrieving, and updating data related to users and their vehicles.

The integration between these microservices allows for seamless interaction between user and vehicle data. For example, the **saveUserAndVehicles** API in the User profile allows clients to simultaneously save user and vehicle information, ensuring data consistency and integrity.

Additionally, error handling mechanisms are implemented to handle various validation scenarios and ensure proper handling of exceptions. Proper status codes are returned to clients to convey the outcome of each request accurately.

Overall, this microservice architecture provides a flexible and scalable solution for managing user and vehicle information in a distributed environment. With its modular design and well-defined APIs, it can easily accommodate future enhancements and scale to meet growing demands.