

**Gisma
University
of Applied
Sciences**

Gisma University of Applied Sciences

- - - M604 ADVANCED PROGRAMMING - - -

“Smart Energy Usage Tracking System “

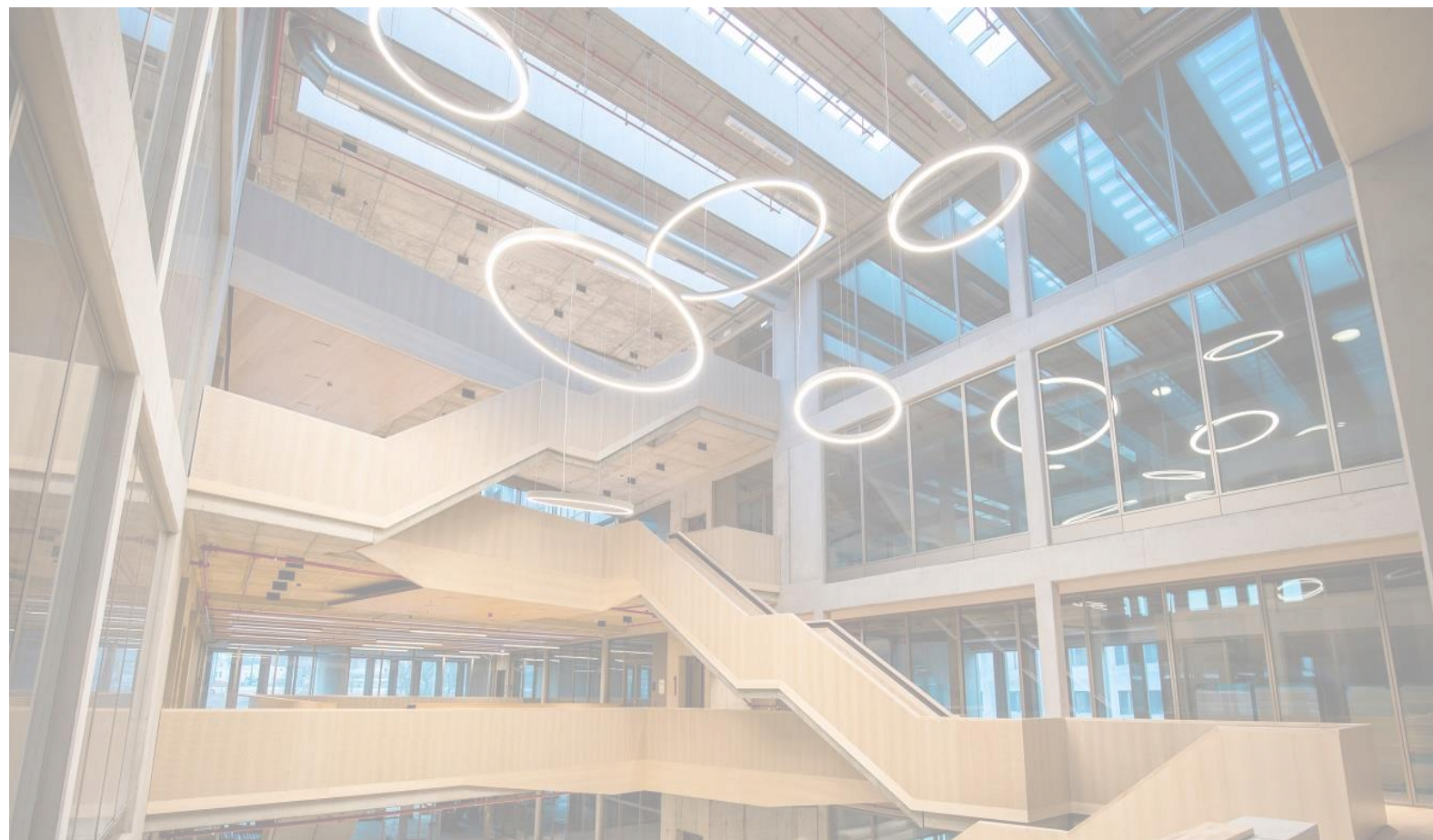
Batuhan Öztürk - GH1031500

Code and Project Report:

<https://github.com/BatuhanOzturk0/SmartEnergyManagementApp>

Project Video:

<https://drive.google.com/file/d/1qXJmPSjkKEiyjU987W3jadSI711a4UGr/view?usp=sharing>



ABSTRACT

Today, technology is developing rapidly and with this development, data is becoming an important resource in every field. Especially in the energy sector, the correct management of the data produced and consumed plays a critical role in both economic and environmental terms. However, the continuous increase of this data makes its management and analysis more complex every day.

Our main motivation when starting this project was to design a system that would allow us to manage data related to energy consumption more easily, regularly and reliably. With this application we developed, we wanted to ensure that the data was stored in a database systematically, not manually, and to establish a cleaner and more stable structure.

We also encountered some difficulties during the project development process. MySQL connection errors, database access, technical problems encountered in API calls slowed down the process from time to time. However, these problems accelerated our learning process and ensured that the project was established on a more solid structure.

This project is a backend application created using the Spring Boot framework. Basic operations such as recording, listing, updating and deleting energy usage data were performed by interacting with the MySQL database. We tested "API endpoints" using the Postman application and wanted to increase the reliability of the system with error handling (Exception Handling) and fallback mechanisms.

The system we want to create at the end of the project allows users to track energy data effectively and understandably. We foresee that this structure can be further strengthened with visual interfaces or sensor connections to be developed in the future.

INTRODUCTION

Nowadays, technology is developing day by day and these developments are evident in every aspect of life. With this technological transformation, the speed of data generation of digital systems has also increased significantly. The energy sector in particular is an area that produces a large amount of data and requires this data to be managed correctly, effectively and securely. Changing consumption habits, the rise of sustainability goals and the need for efficient use of energy resources have made it even more important to collect, store and analyze energy data correctly.

However, some fundamental problems experienced at this point draw attention. The fact that energy usage data is often managed in a scattered, manual or uncontrolled manner causes major problems both at the individual and institutional level. Failure to collect energy data regularly, making it difficult to analyze it and failure to detect potential errors that may harm the system reduces the reliability of the system and creates inefficiency. In addition, the need for systems where users can easily monitor, intervene or report energy data is increasing day by day.

This project was developed to provide solutions to these very problems. My main goal in this project is ; The aim of this project was to design a backend systems where energy consumption data could be stored and managed in a simple, organized and effective manner. This system, developed with Spring Boot, works on a MySQL database and enables the addition, display update and deletion of energy data via REST APIs. In this way , users will be able to easily access and process energy-related data.

My main motivation at the beginning of the project was to facilitate the management of this data , to make processes clean and systematic, and to create a more sustainable software infrastructure, being aware of the importance given to data today. It was aimed that this system would be the basis for more advanced systems in the future, not only technically, but also with its user-friendly and expandable structure.

In this section of my report, I wanted to clearly share why I developed such a system, which problem I was looking for a solution to and for whom this solution could be valuable. The processes experienced throughout the project, the problems encountered and results obtained will be explained in detail in the following sections.

LITERATURE REVIEW

Processing and managing energy consumption data in digital environments has long been an important issue in both the academic world and the industrial field. In many academic studies in this field, it is emphasized that the correct collection, interpretation and reporting of data has a direct impact on the efficiency of energy management. The capabilities of these systems have also increased, especially with the development of big data, the internet of things (IoT) and artificial intelligence technologies.

For example an IEEE study published in 2021 started that IoT-based energy monitoring systems provide energy savings and consumption optimization in home and industrial applications. Similarly, a Springer article from 2022 started that thanks to the transfer of energy data to central servers and its accessibility via API's, users can observe their consumption habits and create behavioral changes.

Applications on the market also provide important examples in this regard. Systems such as Google Nest provide energy efficiency by analyzing the user's heating-cooking behavior. Similarly, some energy companies in Türkiye present users past consumption with graphs and make recommendations via mobile applications. However, most of these systems are either closed source or require infrastructures that are too costly for individual developers to access.

The existence of similar projects developed on universities is also noticeable in the competition between departments. While some projects focused only on the frontend, others worked only with sensor data. However, projects that cover all processes such as a full backend system, data model, API management, error control are fewer in number. At this point, our project aims to set an example for both academic and sectoral applications by establishing a solid structure especially on the backend.

The huge increase in data entry into digital systems in recent years has also increased the needs in this area. When some unsuccessful projects are examined in particular, it is seen that users' trust in the systems is damaged, usually due to reasons such as the data not being processed properly or the system not being reliable. Our project aims to establish a simple but effective structure that allows the data to be received, stored and returned correctly by seeing these deficiencies.

In this sense, this system, inspired by academic studies, works on an open-source backend architecture and provides data flow to the outside world with RESTful APIs. Thanks to the difficulties and solutions experienced during the implementation process, both my learning process was enriched and the robustness of the project increased.

METHODOLOGY

This project was developed to systematically store and manage energy consumption data in a digital environment. The development process of the project was started completely from scratch and structured step by step. Java language and Spring Boot framework were used in the development of the application. MySQL was preferred for database operations. RESTful API architecture was applied throughout the project, and tests were performed with the Postman tool.

In the first phase of the project, a Maven-based Spring Boot Project was created using Spring Initializr. The necessary dependencies (**Spring-boot-starter-web, spring-boot-starter-data-jpa, mysql-connector-java**) were added to the **pom.xml** file and the database connection was configured via the **application.yml** file.

- A MySQL schema named smart_energy_schema_AB was created on the database side and tables named :

Users: Stores user information (ID, name, email, password details)

Devices: Manifests smart energy devices linked to users.

Energy_usage: Energy usage information of devices.

Billing: Recording monthly energy consumption bills.

These tables were defined in this schema. The connections of these tables were structured with foreign key relationships and the energy_usage table was determined as the main table to be used in this project.

- In the backend section, a package structure suitable for the layered architecture called model, repository, service, controller and exception has been created :

Model : An Entity class named EnergyUsage has been defined in the model layer, and database columns have been represented here.

Repository: In the repository layer, the EnergyUsageRepository interface that executes database operations has been created using JPA.

Service: In the service layer, the EnergyUsageService class that manages business logic has been written.

Controller: In the controller layer, REST API endpoints have been defined and made accessible via Postman.

- Each of CRUD (create, read, update, Delete) operations has been implemented step by step :

Create : Data added to the database with POST/api/usage

Read : All data was listed with GET/api/usage, detailed data was retrieved with GET/api/usage/{id}

Update: Data was updated with PUT/api/usage{id}

Delete: Records were deleted with DELETE/api/usage{id}

Error management(Exception Handling) and fallback systems were also included to make the application more stable and user-friendly. By defining the GlobalExceptionHandler class specifically for the project, general and specific errors that may occur in the system(for example: no record found) were presented to the user with meaningful messages. Thanks to this structure, it has become possible to return without the system crashing.

All tests were performed via Postman , a total of 10 data were successfully sent to the system and updateable/deletable structures were tested. The entire structure used in the project was prepared to be simple,understandable and at a new learner level.

USED TECHNOLOGIES

Backend : JAVA 21 + spring boot 3.4

ORM : Spring data JPA

Databases : MySQL(workbench)

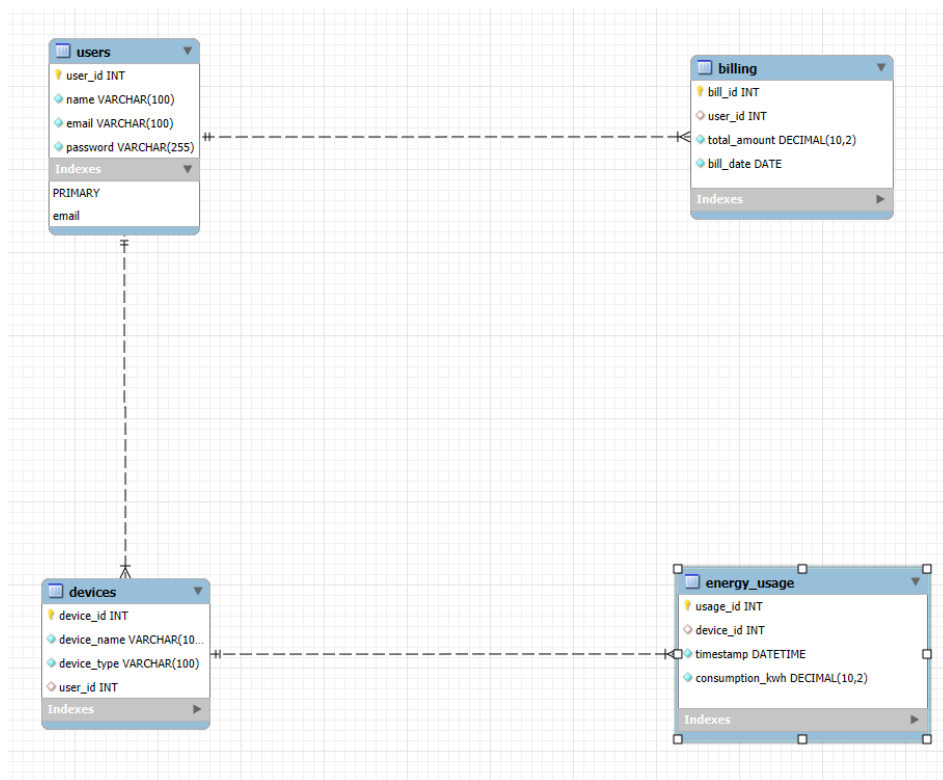
Test: POSTMAN

IDE : IntelliJ IDEA

SYSTEM DESIGN

1- General Architectural Structure

A-SQL DATABASE DESIGN (EER DIAGRAM)



B- Backend (Spring Boot)

In this project, we used the backend part for data processing of the application, execution of business logic and interaction with the database. Also in this project, we used the backend completely in java programming language and created it with Spring Boot framework. Spring Boot provided us with advantages such as fast project setup dependency management and ease of configuration and is widely used in modern web-based applications.

When we created in this project, to control the entire data flow with the backend part and we provided communication with the database by receiving and processing API requests from the outside world.

- java 21

The programming language of the project is Java.

- Spring Boot 3.4

It is used to start and configure the project quickly(Spring-boot-starter-web and spring-boot-starter-data)

- Maven

It is the dependency management system used in the project. Necessary libraries are added and managed automatically via the pom.xml file.

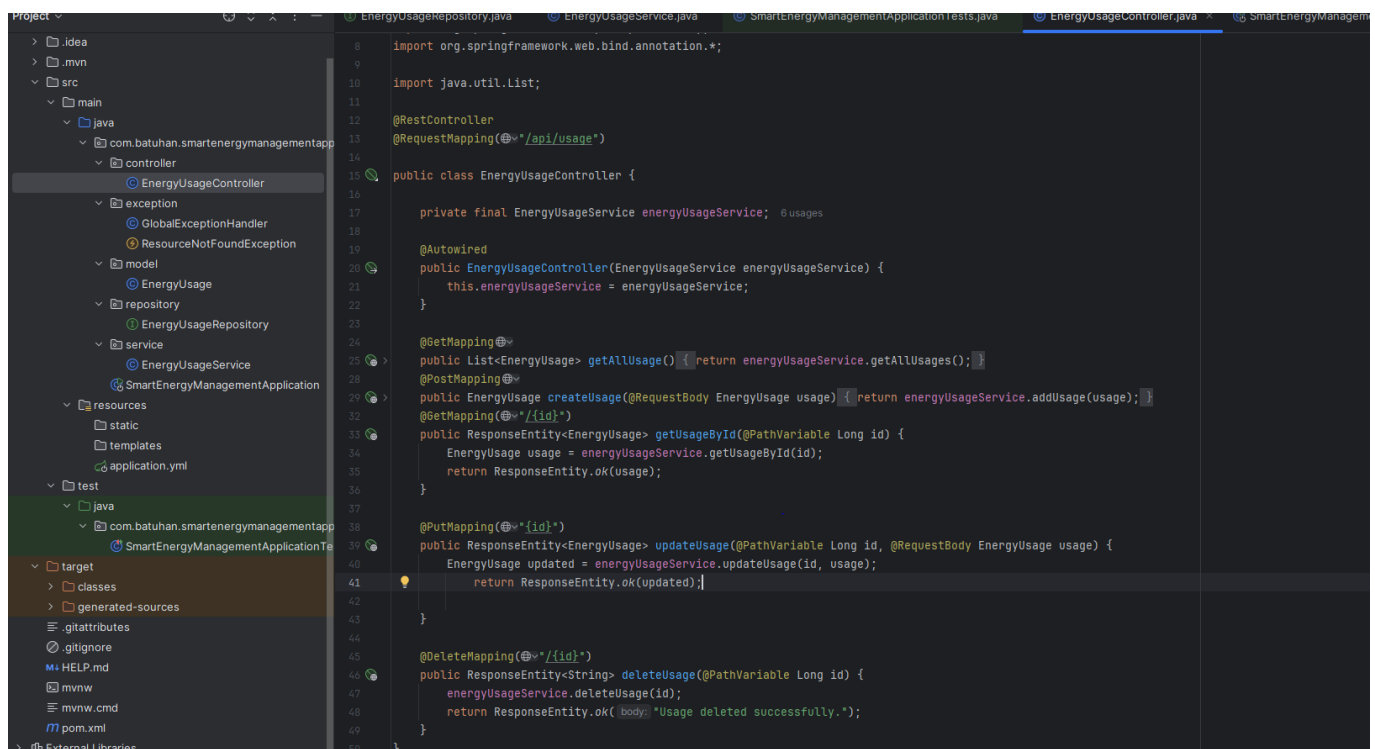
- Spring Data JPA

Simplifies database operations (data insertion , update, deletion, query). Database operations are performed without writing SQL thanks to the repository structure. It uses the ORM (Object Relational Mapping) system; Java classes are connected to database tables

- Exception Handling (Error Management)

A special error catcher system has been established with the `@ControllerAdvice` and `@ExceptionHandler` annotations. When the user makes an incorrect request (for example: when a non-existent ID is called), the system does not crash. Instead, a descriptive error message is returned to the user in JSON format.

C-API (RESTful)



The screenshot shows an IDE with a project structure on the left and the `EnergyUsageController.java` file open in the editor. The project structure includes a `controller` package with `EnergyUsageController`, an `exception` package with `GlobalExceptionHandler` and `ResourceNotFoundException`, a `model` package with `EnergyUsage`, a `repository` package with `EnergyUsageRepository`, and a `service` package with `EnergyUsageService`. The `EnergyUsageController` class is annotated with `@RestController` and `@RequestMapping("/api/usage")`. It contains several methods: `getAllUsage()`, `createUsage()`, `getUsageById()`, `updateUsage()`, and `deleteUsage()`. The `updateUsage()` method has a yellow warning icon next to it, indicating a potential issue with the `updated` variable.

```
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
@RequestMapping("/api/usage")

public class EnergyUsageController {

    private final EnergyUsageService energyUsageService; @ usages

    @Autowired
    public EnergyUsageController(EnergyUsageService energyUsageService) {
        this.energyUsageService = energyUsageService;
    }

    @GetMapping("/")
    public List<EnergyUsage> getAllUsage() { return energyUsageService.getAllUsages(); }

    @PostMapping("/")
    public EnergyUsage createUsage(@RequestBody EnergyUsage usage) { return energyUsageService.addUsage(usage); }

    @GetMapping("/{id}")
    public ResponseEntity<EnergyUsage> getUsageById(@PathVariable Long id) {
        EnergyUsage usage = energyUsageService.getUsageById(id);
        return ResponseEntity.ok(usage);
    }

    @PutMapping("/{id}")
    public ResponseEntity<EnergyUsage> updateUsage(@PathVariable Long id, @RequestBody EnergyUsage usage) {
        EnergyUsage updated = energyUsageService.updateUsage(id, usage);
        return ResponseEntity.ok(updated);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<String> deleteUsage(@PathVariable Long id) {
        energyUsageService.deleteUsage(id);
        return ResponseEntity.ok("Usage deleted successfully.");
    }
}
```

The API is the layer of the project that communicates with the outside world. Users can send data to the database, make queries and perform update and delete operations through this API.

The following http endpoints are defined through the `@RestController` class defined in Spring Boot :

METHOD :

POST /api/usage Adds new energy data

GET /api/usage Brings all data

GET /api/usage/{id} Brings a specific data

PUT /api/usage/{id} Updates specific data

DELETE /api/usage/{id} Deletes specific data

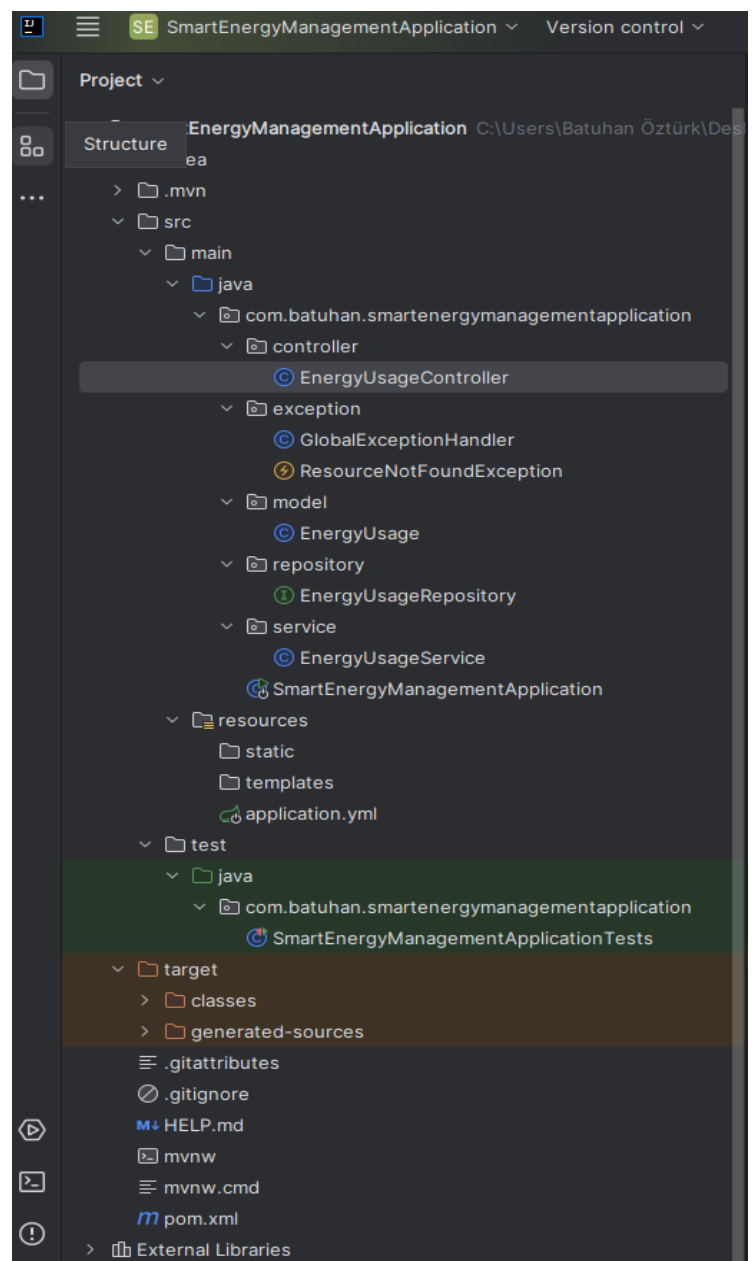
Thanks to these structures, the system has been made open to the outside and can be easily tested. API tests were performed using the Postman tool.

2- How to application Works

- 1- User sends a request (for exmaple POST to add energy data)
- 2- Request comes to controller layer
- 3- Controller => Service => Repository proceeds in order
- 4- Repository executtes database operations
- 5- Results is returned to the user
- 6- If there are any errors, GlobalExceptionHandler comes into play and sends the correct message to the user.

3- Layer structure

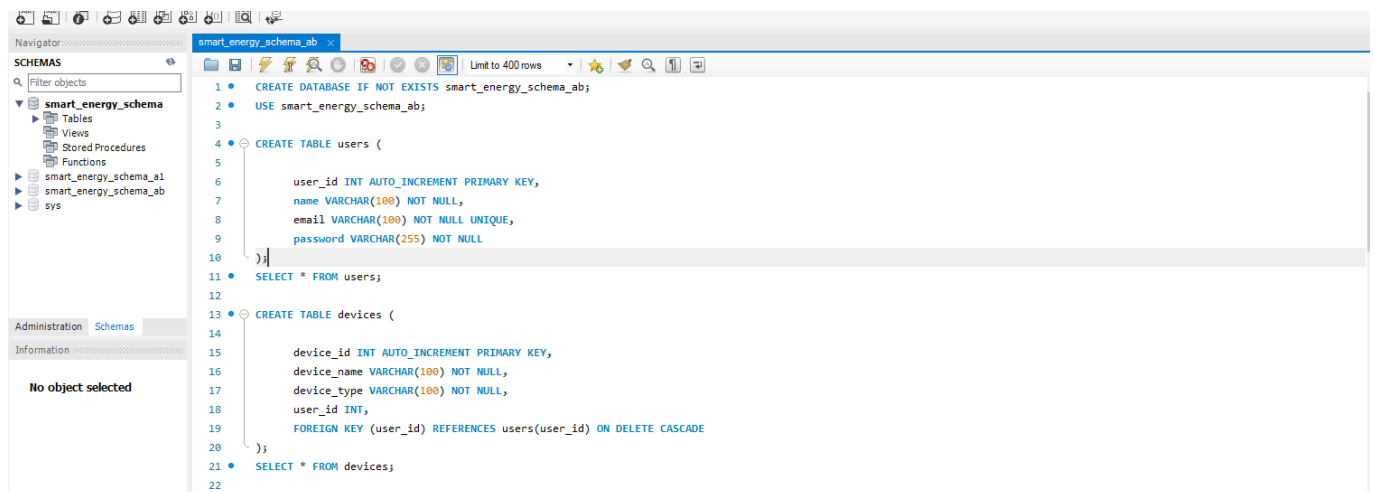
- **controller** => API ends . where requests are received
- **Service** => Business logic. How data is processed.
- **Repository** => Data access. Database queries.



- **Model** => Entity classes. Database tables
- **Exception** => Error handling. Custom errors and exceptions
- **Main class** => Application launcher .Spring Boot entry point.
- **Resources** => Database connection settings are made here. Like applicaton.yml
- **Test** => Used to test whether the written codes work properly.

IMPLEMENTATION

1- SQL PART EXPLAIN



CREATE DATABASE smart energy schema ab : This part creating are in MySQL workbench because we working in this part.

USE smart energy schema ab: Using this code for databases in the current session , all operations performed in this schema.

Create table users: This code created to store user information.

User_id = Each users defined as a primary key that automatically increments.

Name = Users name, should be maximum 100 characters long

Email = users email adress need to unique.

Password = users password, selected 255 characters for security reason.

Create table devices : Table created to store devices for connected to user.

Device_id= For primary key specific to devices.

Device_name = name of the device

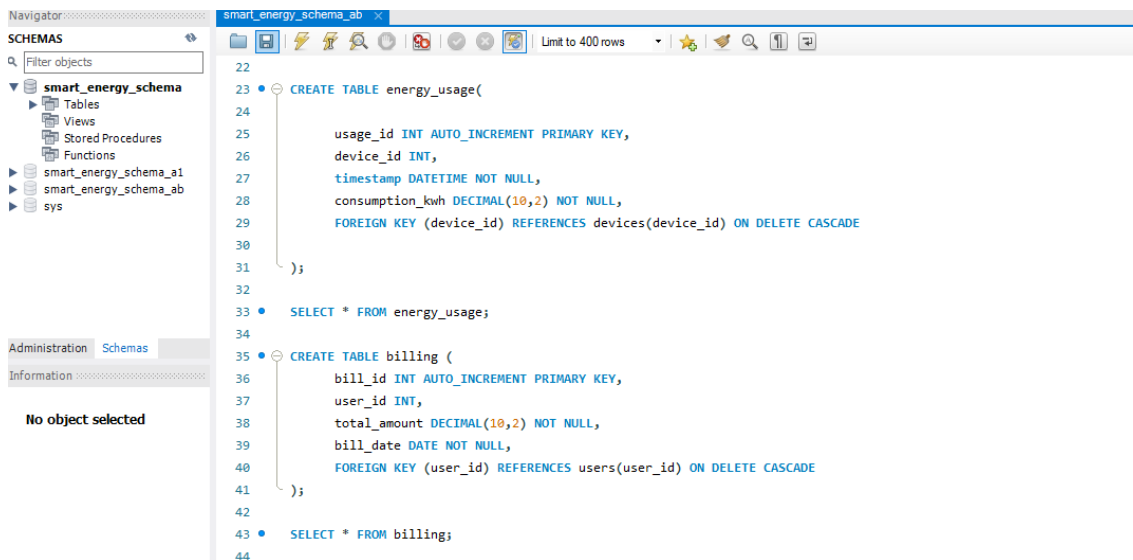
Device_type= type for device like smart sockets, electricity meters.

Foreign key = using connection that user_id is linked to user_id in users table

ON DELETE CASCADE = if any users deleted, all devices connected to user are deleted.

- This model has got using one to many relationship

SELECT * FROM users and SELECT * FROM devices : These queries are using to list data in users and devices table.



The screenshot shows a database management interface with a 'SCHEMAS' panel on the left and a SQL editor on the right. The 'smart_energy_schema_ab' schema is selected in the left panel. The SQL editor contains the following code:

```
22
23 • CREATE TABLE energy_usage(
24     usage_id INT AUTO_INCREMENT PRIMARY KEY,
25     device_id INT,
26     timestamp DATETIME NOT NULL,
27     consumption_kwh DECIMAL(10,2) NOT NULL,
28     FOREIGN KEY (device_id) REFERENCES devices(device_id) ON DELETE CASCADE
29 );
30
31
32
33 • SELECT * FROM energy_usage;
34
35 • CREATE TABLE billing (
36     bill_id INT AUTO_INCREMENT PRIMARY KEY,
37     user_id INT,
38     total_amount DECIMAL(10,2) NOT NULL,
39     bill_date DATE NOT NULL,
40     FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE
41 );
42
43 • SELECT * FROM billing;
44
```

CREATE TABLE energy_usage: Created to storage energy consumption data of devices.

-usage_id = Primary key for each energy usage record.

-device_id = ID of the device usage saving.

-timestamp = Date and time recorded.

-consumption_kwh= Amount of energy consumed.

- Foreign key = Using connection that user_id is linked to user_id in users table

- ON DELETE CASCADE= if any users deleted , all devices connected to user are deleted.

This structure allows us to all record how much energy using each device at certain time.

CREATE TABLE billing: Created to storage bill records and created based users energy consumption.

-bill_id = Primary key each all records:

- user_id= Displaying which users the bill belongs to.
- total_amount= Total energy bill amount that the user must pay for month
- bill_date= Date the bill was created.
- Foreign key = Using connection that user_id is linked to user_id in users table
- ON DELETE CASCADE= if any users deleted , all devices connected to user are deleted.

This structure allows us to track monthly energy bill record for each users.

INSERT INTO = This SQL query add 10 different users to user table . This comment same for all upper code (users,energy_usage,devices,billing)

UPDATE SET WHERE = If we want to update any device,user,energy_usage, billing using ‘Update’ comment .We use ‘SET’ comment for new data and using ‘WHERE’ comment for last data adress.

* * * But we using and creating JAVA and POSTMAN instead of this funtction. * * *

The image shows two screenshots of a SQL IDE window titled 'smart_energy_schema_ab'. The left screenshot displays SQL queries for creating and populating the 'users' and 'devices' tables. The right screenshot displays SQL queries for creating and populating the 'energy_usage' and 'billing' tables.

```

45 • INSERT INTO users (name, email, password) VALUES
46   ('Batuhan Ozturk','batuhan@example.com','14124124'),
47   ('Munevver Ozturk','munevver@example.com','64646341'),
48   ('Harun Ozturk','harun@example.com','12312477'),
49   ('Vehbi Okumus','vehbi@example.com','53678042'),
50   ('Ismet Eren','ismet@example.com','19283640'),
51   ('Semih Tarcan','semih@example.com','82659172'),
52   ('Mehmet Ali','mehmet@example.com','46837482'),
53   ('Samet Akbas','samet@example.com','15235376'),
54   ('Ekin Kaya','ekin@example.com','37590127'),
55   ('Omer Akin','omer@example.com','48362412');
56
57 • UPDATE users SET email = 'batuhan@example.com' WHERE user_id = 1;
58 • DELETE FROM users WHERE user_id = 10;
59
60
61 • INSERT INTO devices (device_name, device_type, user_id) VALUES
62   ('Smart Meter 1','Electricity',1),
63   ('Smart Thermostat','Heating',1),
64   ('Solar Panel 1','Renewable',2),
65   ('Battery Storage','Storage',3),
66   ('EV Charger','Charging',4),
67   ('Smart Fridge','Appliance',5),
68   ('Wind Turbine 1','Renewable',6),
69   ('Smart Bulb 1','Lighting',7),
70   ('Water Heater','Heating',8),
71   ('Security Camera','Surveillance',9);
72
73 • UPDATE devices SET device_name = 'Smart Meter 2' WHERE device_id = 1;
74 • DELETE FROM devices WHERE device_id = 5;
75
76
77
78 • INSERT INTO energy_usage (device_id, timestamp, consumption_kwh) VALUES
79   (1,'2025-02-17 09:36:00',3.5),
80   (1,'2025-02-17 12:12:36',2.1),
81   (2,'2025-02-18 10:35:30',1.8),
82   (3,'2025-02-18 14:45:00',4.2),
83   (4,'2025-02-18 15:10:30',5.3),
84   (5,'2025-02-19 16:30:15',2.9),
85   (6,'2025-02-19 04:40:00',3.7),
86   (7,'2025-02-20 22:20:56',1.3),
87   (8,'2025-02-20 08:00:50',2.8),
88   (9,'2025-02-20 20:30:00',4.9);
89
90 • UPDATE energy_usage SET consumption_kwh = 4.0 WHERE usage_id = 2;
91 • DELETE FROM energy_usage WHERE usage_id = 3;
92
93 • INSERT INTO billing (user_id, total_amount, bill_date) VALUES
94   (1,50.75,'2025-03-01'),
95   (2,60.20,'2025-03-02'),
96   (3,45.10,'2025-03-03'),
97   (4,55.90,'2025-03-04'),
98   (5,70.30,'2025-03-05'),
99   (6,48.60,'2025-03-06'),
100  (7,65.80,'2025-03-07'),
101  (8,52.40,'2025-03-08'),
102  (9,58.90,'2025-03-09'),
103  (10,62.10,'2025-03-10');
104
105 • UPDATE billing SET total_amount = 75.50 WHERE bill_id = 3;
106 • DELETE FROM billing WHERE bill_id = 2;
107
108

```

2-Initializing and Configuring Spring Boot Project (pom.xml + application.yml)

```
m pom.xml (SmartEnergyManagementApplication) x application.yml EnergyUsage.java
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.ap
4   <modelVersion>4.0.0</modelVersion>
5   <parent>
6     <groupId>org.springframework.boot</groupId>
7     <artifactId>spring-boot-starter-parent</artifactId>
8     <version>3.4.4</version>
9     <relativePath/> <!-- lookup parent from repository -->
10  </parent>
11  <groupId>com.batuhan</groupId>
12  <artifactId>SmartEnergyManagementApplication</artifactId>
13  <version>0.0.1-SNAPSHOT</version>
14  <name>SmartEnergyManagementApplication</name>
15  <description>SmartEnergyManagementApplication</description>
16  <url/>
17  <licenses>
18    <license/>
19  </licenses>
20  <developers>
21    <developer/>
22  </developers>
23  <scm>
24    <connection/>
25    <developerConnection/>
26    <tag/>
27    <url/>
28  </scm>
29  <properties>
30    <java.version>21</java.version>
31  </properties>
32  <dependencies> Add Starters...
33    <dependency>
34      <groupId>org.springframework.boot</groupId>
35      <artifactId>spring-boot-starter-data-jpa</artifactId>
36    </dependency>
37    <dependency>
38      <groupId>org.springframework.boot</groupId>
39      <artifactId>spring-boot-starter-web</artifactId>
40    </dependency>
41
```

```
40    </dependency>
41
42    <dependency>
43      <groupId>com.mysql</groupId>
44      <artifactId>mysql-connector-j</artifactId>
45      <scope>runtime</scope>
46    </dependency>
47    <dependency>
48      <groupId>org.springframework.boot</groupId>
49      <artifactId>spring-boot-starter-test</artifactId>
50      <scope>test</scope>
51    </dependency>
52    <dependency>
53      <groupId>mysql</groupId>
54      <artifactId>mysql-connector-java</artifactId>
55      <version>8.0.33</version>
56    </dependency>
57  </dependencies>
58
59  <build>
60    <plugins>
61      <plugin>
62        <groupId>org.springframework.boot</groupId>
63        <artifactId>spring-boot-maven-plugin</artifactId>
64      </plugin>
65    </plugins>
66  </build>
67
68 </project>
69
```

- Maven Configuration and pom.xml File

The pom.xml file is one of the fundamental building parts of my project; it controls project dependencies and customizations. I found the Java version of the program I should run, which libraries I should download and which setups would be active using this file.

Firstly, using Maven, I developed a Spring Boot project; so, the basis of the project setup was established in the part on spring-boot-starter-parenting.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.4.4</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

These lines helped me to reduce issues like version incompatibility in the project and get the most current tools available Spring Boot.

- Spring Data JPA (for database operations)

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
```

Thanks to this library, I was able to connect my java classes directly to database tables. Thus, I had the opportunity to perform object-based operations with data without SQL queries.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

-Spring Web (For creating APIs)

This dependency enabled the necessary structures to be activated to create a REST API. Thanks to this, I was able to create my Controller class

```

52     <dependency>
53         <groupId>mysql</groupId>
54         <artifactId>mysql-connector-java</artifactId>
55         <version>8.0.33</version>
56     </dependency>

```

-MySQL Connector

I used this library so that my application could connect to the MySQL database. Without this library , my application could not communicate with the database.

```

47     <dependency>
48         <groupId>org.springframework.boot</groupId>
49         <artifactId>spring-boot-starter-test</artifactId>
50         <scope>test</scope>
51     </dependency>

```

-Test Dependency

This library was used to test possible errors while developing the project. The test class in the test folder was created thanks to this structure.

```

29     <properties>
30         <java.version>21</java.version>
31     </properties>

```

-Java Version

I preferred Java 21 version in my project because it is both compatible with spring boot 3.4 version and supports more modern features.

- Application.yml (DATABASE CONNECTION)

```
m pom.xml (SmartEnergyManagementApplication) application.yml x EnergyUsage.java
1  server :
2    port: 8080
3
4  spring:
5    datasource:
6      url: jdbc:mysql://localhost:3306/smart_energy_schema_ab
7      username: root
8      password: spring123
9      driver-class-name: com.mysql.cj.jdbc.Driver
10
11
12    jpa:
13      hibernate:
14        ddl-auto: update
15      properties:
16        hibernate:
17          dialect: org.hibernate.dialect.MySQL8Dialect
18          format_sql: true
19          show_sql: true
20
21    main:
22      web-application-type: servlet
23
```

I used the configuration file named application.yml so that my application can communicate with the database. Thanks to this file, I can connect to the MySQL database with my Spring Boot application and perform the necessary operations automatically.

```
m pom.xml (SmartEnergyManagementApplication) applica
1  server :
2    port: 8080
```

This line determines which port my application will run on. The 8080 port is the default port of Spring Boot and was used in this project as well.

```
4  spring:
5    datasource:
6      url: jdbc:mysql://localhost:3306/smart_energy_schema_ab
7      username: root
8      password: spring123
9      driver-class-name: com.mysql.cj.jdbc.Driver
10
```

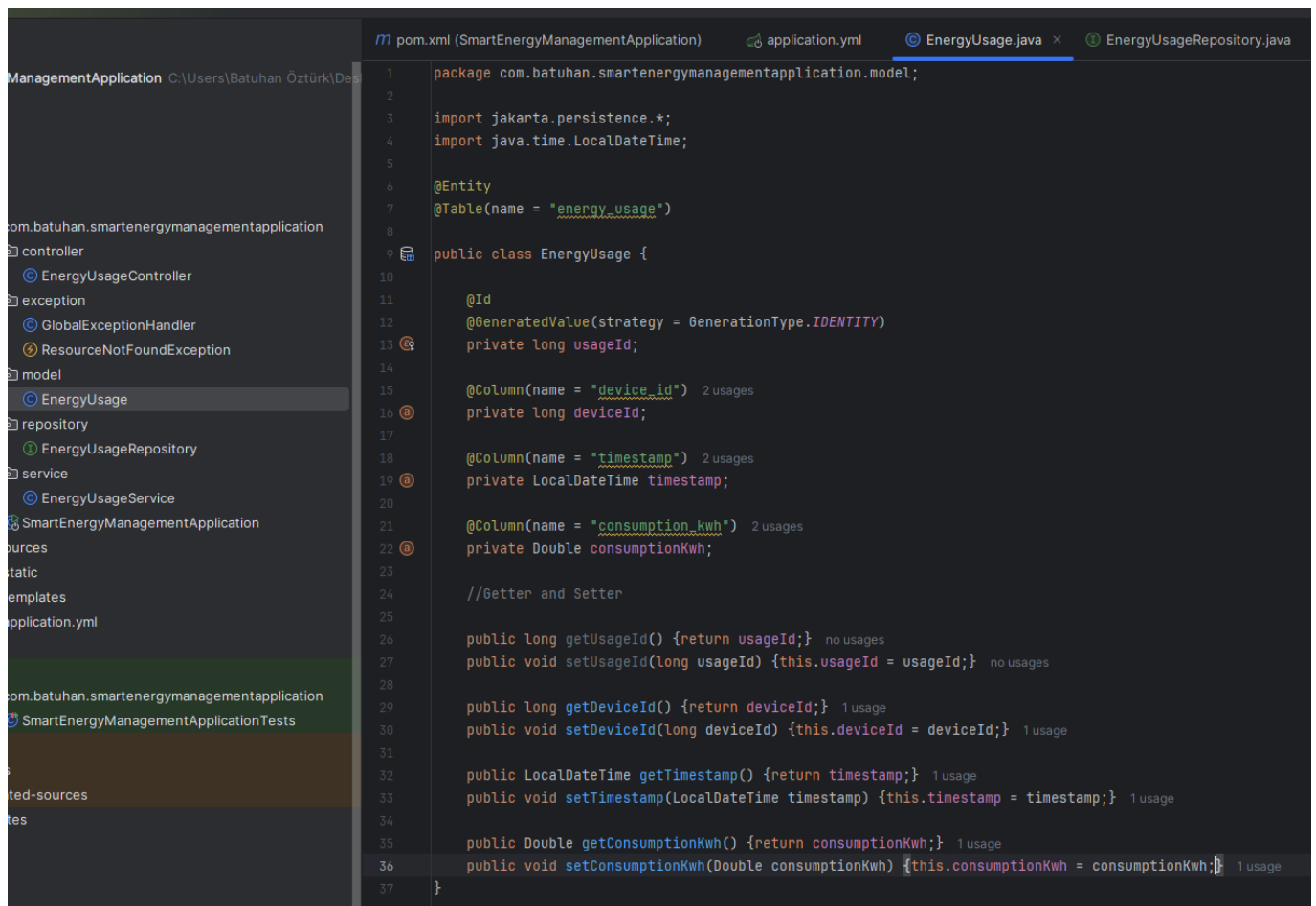
In this section, information such as which database my application will connect to , username and password are defined.

url: The address of the database the application will connect to. Here , I connected to the smart_energy_schema_ab database localhost.

Username&password : The username and password used to connect to MySQL.

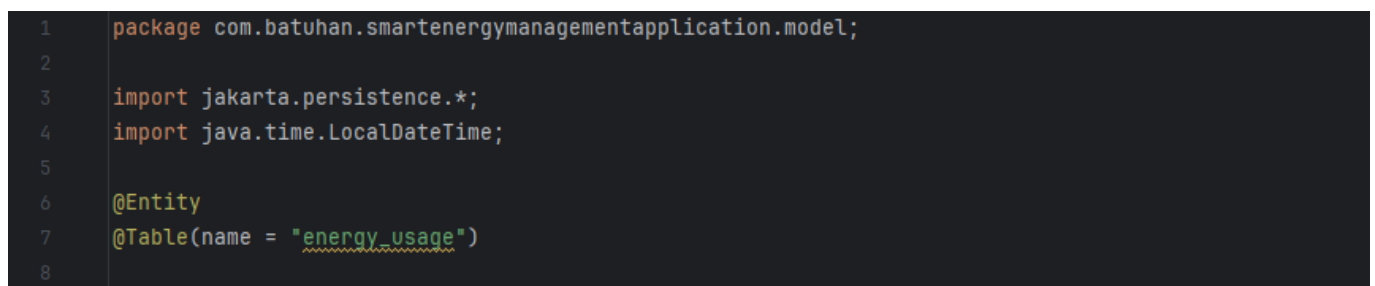
Driver-class-name: The class of the JDBC driver required to connect to The MySQL database.

3- MODEL (ENTITY) PART



```
1 package com.batuhan.smartenergymanagementapplication.model;
2
3 import jakarta.persistence.*;
4 import java.time.LocalDateTime;
5
6 @Entity
7 @Table(name = "energy_usage")
8
9 public class EnergyUsage {
10
11     @Id
12     @GeneratedValue(strategy = GenerationType.IDENTITY)
13     private long usageId;
14
15     @Column(name = "device_id") 2 usages
16     private long deviceId;
17
18     @Column(name = "timestamp") 2 usages
19     private LocalDateTime timestamp;
20
21     @Column(name = "consumption_kwh") 2 usages
22     private Double consumptionKwh;
23
24     //Getter and Setter
25
26     public long getUsageId() {return usageId;} no usages
27     public void setUsageId(long usageId) {this.usageId = usageId;} no usages
28
29     public long getDeviceId() {return deviceId;} 1 usage
30     public void setDeviceId(long deviceId) {this.deviceId = deviceId;} 1 usage
31
32     public LocalDateTime getTimestamp() {return timestamp;} 1 usage
33     public void setTimestamp(LocalDateTime timestamp) {this.timestamp = timestamp;} 1 usage
34
35     public Double getConsumptionKwh() {return consumptionKwh;} 1 usage
36     public void setConsumptionKwh(Double consumptionKwh) {this.consumptionKwh = consumptionKwh;} 1 usage
37 }
38
```

In the project , we needed a Java class that represents the energy_usage table in the database. For this, I created a class called EnergyUsage in the model foled. This class acts as a bridge that allows both data retrieval and saving.



```
1 package com.batuhan.smartenergymanagementapplication.model;
2
3 import jakarta.persistence.*;
4 import java.time.LocalDateTime;
5
6 @Entity
7 @Table(name = "energy_usage")
8
```

Entity: I used it to indicate that this class corresponds to a database table.

Table: This class provides a one-to-one match with the energy_usage table in MySQL.

```
10
11     @Id
12     @GeneratedValue(strategy = GenerationType.IDENTITY)
13     private long usageId;
14
15     @Column(name = "device_id") 2 usages
16     private long deviceId;
17
18     @Column(name = "timestamp") 2 usages
19     private LocalDateTime timestamp;
20
21     @Column(name = "consumption_kwh") 2 usages
22     private Double consumptionKwh;
23
```

```
//Getter and Setter

public long getUsageId() {return usageId;} no usages
public void setUsageId(long usageId) {this.usageId = usageId;} no usages

public long getDeviceId() {return deviceId;} 1 usage
public void setDeviceId(long deviceId) {this.deviceId = deviceId;} 1 usage

public LocalDateTime getTimestamp() {return timestamp;} 1 usage
public void setTimestamp(LocalDateTime timestamp) {this.timestamp = timestamp;} 1 usage

public Double getConsumptionKwh() {return consumptionKwh;} 1 usage
public void setConsumptionKwh(Double consumptionKwh) {this.consumptionKwh = consumptionKwh;} 1 usage
}
```

I wrote get and set methods for each variable to access and change the data from outside. This allows Spring to easily perform operations on this project.

4-REPOSITORY LAYER

```

1 package com.batuhan.smartenergymanagementapplication.repository;
2
3 import com.batuhan.smartenergymanagementapplication.model.EnergyUsage;
4 import org.springframework.data.jpa.repository.JpaRepository;
5 import org.springframework.stereotype.Repository;
6 @Repository 3 usages
7
8
9 public interface EnergyUsageRepository extends JpaRepository<EnergyUsage, Long> {
10
11 }
12

```

At this stage , I created a folder named repository to access the database. I added an interface named EnergyUsageRepository that will work directly with our entity class EnergyUsage.

```

public interface EnergyUsageRepository extends JpaRepository<EnergyUsage, Long> {
}

```

This annotation allows this class to be recognized as a repository by Spring. In other words, this structure is used for data access (CRUD Operations).

**** What is JPA REPOSITORY ? ****

JpaRepository is a ready interface provided by Spring Data JPA. It already contains the basic methods for database operations:

findAll() = Gets all data

findById() = gets data with a specific ID

save (EnergyUsage usage) = Adds new data or updates existing data

deleteById(long id) = deletes data with a specific ID .

5- SERVICE LAYER

```
1 package com.batuhan.smartenergymanagementapplication.service;
2
3 import com.batuhan.smartenergymanagementapplication.exception.ResourceNotFoundException;
4 import com.batuhan.smartenergymanagementapplication.model.EnergyUsage;
5 import com.batuhan.smartenergymanagementapplication.repository.EnergyUsageRepository;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.stereotype.Service;
8
9 import java.util.List;
10 import java.util.Optional;
11 @Service
12
13 public class EnergyUsageService {
14
15     private final EnergyUsageRepository energyUsageRepository;
16
17     @Autowired
18     public EnergyUsageService(EnergyUsageRepository energyUsageRepository) {
19         this.energyUsageRepository = energyUsageRepository;
20     }
21
22     public List<EnergyUsage> getAllUsages() { return energyUsageRepository.findAll(); }
23
24     public EnergyUsage addUsage(EnergyUsage usage) { return energyUsageRepository.save(usage); }
25
26     public EnergyUsage getUsageById(Long id) {
27         Optional<EnergyUsage> optionalUsage = energyUsageRepository.findById(id);
28         if (optionalUsage.isPresent()) {
29             return optionalUsage.get();
30         } else {
31             throw new ResourceNotFoundException("Usage not found with this id: " + id);
32         }
33     }
34
35     public void deleteUsage(Long id) {
36         Optional<EnergyUsage> optionalUsage = energyUsageRepository.findById(id);
37         if (optionalUsage.isPresent()) {
38             EnergyUsage usage = optionalUsage.get();
39             energyUsageRepository.delete(usage);
40         } else {
41             throw new ResourceNotFoundException("Usage not found with this id: " + id);
42         }
43     }
44
45     public EnergyUsage updateUsage(Long id, EnergyUsage updatedUsage) {
46         Optional<EnergyUsage> optionalUsage = energyUsageRepository.findById(id);
47         if (optionalUsage.isPresent()) {
48             EnergyUsage existing = optionalUsage.get();
49             existing.setDeviceId(updatedUsage.getDeviceId());
50             existing.setTimestamp(updatedUsage.getTimestamp());
51             existing.setConsumptionKwh(updatedUsage.getConsumptionKwh());
52             return energyUsageRepository.save(existing);
53         } else {
54             throw new ResourceNotFoundException("Usage not found with this id: " + id);
55         }
56     }
57 }
```

In this file , I wrote the codes that manage the business logic of database transactions.In other words, I controlled the transactions between the requests from the user and the repository layer here.

```
@Service
public class EnergyUsageService {
```

@Service : This annotation tells Spring that this class is a service layer.

public class EnergyUsageService: It is name of the service class. It contains all the business logic.

```
private final EnergyUsageRepository energyUsageRepository;

@Autowired
public EnergyUsageService(EnergyUsageRepository energyUsageRepository) {
    this.energyUsageRepository = energyUsageRepository;
}
```

EnergyUsageRepository: This is the layer that connects to the database.

@Autowired: Spring automatically injects this repository into this class. Thanks to this structure, the service class works through the repository, not directly with the database. This provides a modular and easily testable structure.

```
> public List<EnergyUsage> getAllUsages() { return energyUsageRepository.findAll(); }
```

findAll(): gets all energy usage data in the database. When called from outside, for example from Potsman all records are returned as JSON.

```
24 > public EnergyUsage addUsage(EnergyUsage usage) { return energyUsageRepository.save(usage); }
27
```

Save(usage): Receives the new data and saves it to the database. Return the recorded object.

```
8 public EnergyUsage getUsageById(Long id) { 1 usage
9     return energyUsageRepository.findById(id)
0         .orElseThrow(() -> new ResourceNotFoundException("Usage not found with this id: " + id));
1 }
```

findById(id): Fetches the data with the specified ID. If there is no data Optional.empty() returns. The result is returned with the notFound() message in the controller.

```
31 }
32 public void deleteUsage(Long id) { 1 usage
33     EnergyUsage usage = energyUsageRepository.findById(id).orElseThrow(() -> new ResourceNotFoundException("Usage not found with this id: " + id));
34
35     energyUsageRepository.delete(usage);
36 }
```

In the first line, data is searched by ID. If there is no data, a special error is thrown (ResourceNotFoundException). If there is, the deletion is performed.

```
7 @ public EnergyUsage updateUsage(Long id, EnergyUsage updatedUsage) { 1 usage
8     EnergyUsage existing = energyUsageRepository.findById(id).orElseThrow(() -> new ResourceNotFoundException("Usage not found with this id: " + id));
9
10     existing.setDeviceId(updatedUsage.getDeviceId());
11     existing.setTimestamp(updatedUsage.getTimestamp());
12     existing.setConsumptionKwh(updatedUsage.getConsumptionKwh());
13
14     return energyUsageRepository.save(existing);
15 }
16
```

6- CONTROLLER LAYER

```
EnergyUsageService.java SmartEnergyManagementApplicationTests.java EnergyUsageController.java SmartEnergyManagementApplicati
1 package com.batuhan.smartenergymanagementapplication.controller;
2
3 import com.batuhan.smartenergymanagementapplication.exception.ResourceNotFoundException;
4 import com.batuhan.smartenergymanagementapplication.model.EnergyUsage;
5 import com.batuhan.smartenergymanagementapplication.service.EnergyUsageService;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.http.ResponseEntity;
8 import org.springframework.web.bind.annotation.*;
9
10 import java.util.List;
11
12 @RestController
13 @RequestMapping("/api/usage")
14
15 public class EnergyUsageController {
16
17     private final EnergyUsageService energyUsageService;
18
19     @Autowired
20     public EnergyUsageController(EnergyUsageService energyUsageService) {
21         this.energyUsageService = energyUsageService;
22     }
23
24     @GetMapping
25     public List<EnergyUsage> getAllUsage() { return energyUsageService.getAllUsages(); }
26
27     @PostMapping
28     public EnergyUsage createUsage(@RequestBody EnergyUsage usage) { return energyUsageService.addUsage(usage); }
29
30     @GetMapping("/{id}")
31     public ResponseEntity<EnergyUsage> getUsageById(@PathVariable Long id) {
32         EnergyUsage usage = energyUsageService.getUsageById(id);
33         return ResponseEntity.ok(usage);
34     }
35
36     @PutMapping("/{id}")
37     public ResponseEntity<EnergyUsage> updateUsage(@PathVariable Long id, @RequestBody EnergyUsage usage) {
38         EnergyUsage updated = energyUsageService.updateUsage(id, usage);
39         return ResponseEntity.ok(updated);
40     }
41
42     @DeleteMapping("/{id}")
43     public ResponseEntity<String> deleteUsage(@PathVariable Long id) {
44         energyUsageService.deleteUsage(id);
45         return ResponseEntity.ok("Usage deleted successfully.");
46     }
47 }
```

This class receives the http requests made by the user, directs the necessary operations to the service layer and returns the response. In other words, users can communicate with the application from outside thanks to this class

```
@RestController
@RequestMapping("/api/usage")

public class EnergyUsageController {
```

@RestController : Informs Spring that this class is a REST API. JSON responses are returned.

@RequestMapping("/api/usage"): We start all methods in this class from /api/usage address.

```

    private final EnergyUsageService energyUsageService; // 6 usages

    @Autowired
    public EnergyUsageController(EnergyUsageService energyUsageService) {
        this.energyUsageService = energyUsageService;
    }

```

energyUsageService : We used the business logic layer with this object

@Autowired: Spring automatically creates and injects this object

```

    @GetMapping
    public List<EnergyUsage> getAllUsage() { return energyUsageService.getAllUsages(); }

```

This method fetches all records in the database. It works when a GET/API/USAGE request is sent from the browser or Postman.

energyUsageService.getAllUsages() is called and data is returned.

```

    @PostMapping
    public EnergyUsage createUsage(@RequestBody EnergyUsage usage) { return energyUsageService.addUsage(usage); }

```

@PostMapping: Used to add new data.

@RequestBody: Converts the JSON data coming from the user to the EnergyUsage object in this step. We save it to the database with the energyUsageService.addUsages command.

```

    @GetMapping("/{id}")
    public ResponseEntity<EnergyUsage> getUsageById(@PathVariable Long id) {
        EnergyUsage usage = energyUsageService.getUsageById(id);
        return ResponseEntity.ok(usage);
    }

```

@PathVariable Long id: Receives the {id} parameter coming from the URL.

If there is a record corresponding to this ID in the database, it will be returned, otherwise an error will be thrown

```

    @PutMapping("/{id}")
    public ResponseEntity<EnergyUsage> updateUsage(@PathVariable Long id, @RequestBody EnergyUsage usage) {
        EnergyUsage updated = energyUsageService.updateUsage(id, usage);
        return ResponseEntity.ok(updated);
    }

```

@PutMapping("/{id}") : we updated the data belonging to the specific ID

@RequestBody: we received the updated data from the user. We updated the record in the database with new information with updateUsage()

```

    }
    @DeleteMapping("/{id}")
    public ResponseEntity<String> deleteUsage(@PathVariable Long id) {
        energyUsageService.deleteUsage(id);
        return ResponseEntity.ok(body: "Usage deleted successfully.");
    }
}

```

@DeleteMapping : we used it to delete data with a specific ID. If the deletion process is successful a success message is returned.

7-EXCEPTION HANDLING

```

1 package com.batuhan.smartenergymanagementapplication.exception;
2 import org.springframework.http.HttpStatus;
3 import org.springframework.http.ResponseEntity;
4 import org.springframework.web.bind.annotation.ControllerAdvice;
5 import org.springframework.web.bind.annotation.ExceptionHandler;
6
7 import java.time.LocalDateTime;
8 import java.util.HashMap;
9 import java.util.Map;
10
11 @ControllerAdvice
12
13 public class GlobalExceptionHandler {
14
15     @ExceptionHandler(Exception.class)
16     public ResponseEntity<Map<String,Object>> handleGeneralException(Exception ex) {
17         Map<String,Object> errorResponse = new HashMap<>();
18         errorResponse.put("timestamp", LocalDateTime.now());
19         errorResponse.put("message", ex.getMessage());
20         errorResponse.put("status", HttpStatus.INTERNAL_SERVER_ERROR);
21         return new ResponseEntity<>(errorResponse, HttpStatus.INTERNAL_SERVER_ERROR);
22     }
23
24
25     @ExceptionHandler(ResourceNotFoundException.class)
26     public ResponseEntity<Map<String,Object>> handleResourceNotFoundException(ResourceNotFoundException ex) {
27         Map<String,Object> errorResponse = new HashMap<>();
28         errorResponse.put("timestamp", LocalDateTime.now());
29         errorResponse.put("message", ex.getMessage());
30         errorResponse.put("status", HttpStatus.NOT_FOUND);
31
32         return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);
33     }
34 }
35

```

```

1 package com.batuhan.smartenergymanagementapplication.exception;
2
3 public class ResourceNotFoundException extends RuntimeException { 7 usages
4     public ResourceNotFoundException(String message) { 3 usages
5         super(message);
6     }
7 }
8

```


It is very important to control possible errors that may be encountered while developing the project and to ensure that the application can give proper and explanatory feedback to the user. For this reason, I integrated the exception handling mechanism into my project.

```
@ControllerAdvice
public class GlobalExceptionHandler {
```

@ControllerAdvice: This annotation allows us to define a common error handler for all controllers throughout the application. This class is global error handling place of the application.

```
@ExceptionHandler(Exception.class)
public ResponseEntity<Map<String, Object>> handleGeneralException(Exception ex) {
```

This method runs when an unexpected error occurs.

Exception.class : Covers all exception types.

```
    Map<String, Object> errorResponse = new HashMap<>();
    errorResponse.put("timestamp", LocalDateTime.now());
    errorResponse.put("message", ex.getMessage());
    errorResponse.put("status", HttpStatus.INTERNAL_SERVER_ERROR);
    return new ResponseEntity<>(errorResponse, HttpStatus.INTERNAL_SERVER_ERROR);
}
```

ResponseEntity : Allows us to return an http response. We provide detailed information in the Map

```
@ExceptionHandler(ResourceNotFoundException.class)
public ResponseEntity<Map<String, Object>> handleResourceNotFoundException(ResourceNotFoundException ex) {
    Map<String, Object> errorResponse = new HashMap<>();
    errorResponse.put("timestamp", LocalDateTime.now());
    errorResponse.put("message", ex.getMessage());
    errorResponse.put("status", HttpStatus.NOT_FOUND);

    return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);
}
```

We inform the user when the error occurred (timestamp), what the error is (message) and what http status code it will return (status). It returns details with the code 500 Internal Server Error.


```

1 package com.batuhan.smartenergymanagementapplication.exception;
2
3 public class ResourceNotFoundException extends RuntimeException { 7 usages
4     public ResourceNotFoundException(String message) { 3 usages
5         super(message);
6     }
7 }
8

```

This is a custom exception class. If no data is found (for example , in query by id) this error is thrown. When the exception is thrown, the error message is transferred to the parent class(RuntimeException)with super(message).If the requested data is not found in the database,this block is executed. A 404 not found error is returned to the user.

8-TESTS

The screenshot shows an IDE with the following components:

- Project Explorer:** Shows the project structure with folders like 'service', 'resources', 'test', and 'target'. The file 'SmartEnergyManagementApplicationTests.java' is selected under the 'test' folder.
- Code Editor:** Displays the content of 'SmartEnergyManagementApplicationTests.java':


```

1 package com.batuhan.smartenergymanagementapplication;
2
3 import org.junit.jupiter.api.Test;
4 import org.springframework.boot.test.context.SpringBootTest;
5
6 @SpringBootTest
7 class SmartEnergyManagementApplicationTests {
8
9     @Test
10    void contextLoads() {
11    }
12
13 }
14

```
- Run Console:** Shows the execution logs of the application. The logs indicate that the application started successfully on port 8080. Key log entries include:
 - INFO 14676 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
 - WARN 14676 --- [main] JpaBaseConfiguration\$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during application startup.
 - INFO 14676 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
 - INFO 14676 --- [main] c.b.s.SmartEnergyManagementApplication : Started SmartEnergyManagementApplication in 5.523 seconds (process running for 6.395)
 - INFO 14676 --- [nio-8080-exec-1] o.a.c.c.f.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
 - INFO 14676 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
 - INFO 14676 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms

@SpringBootTest => It allows us to run the Spring application in a test environment by loading entire context. In this way services, controllers and repositories are also loaded.

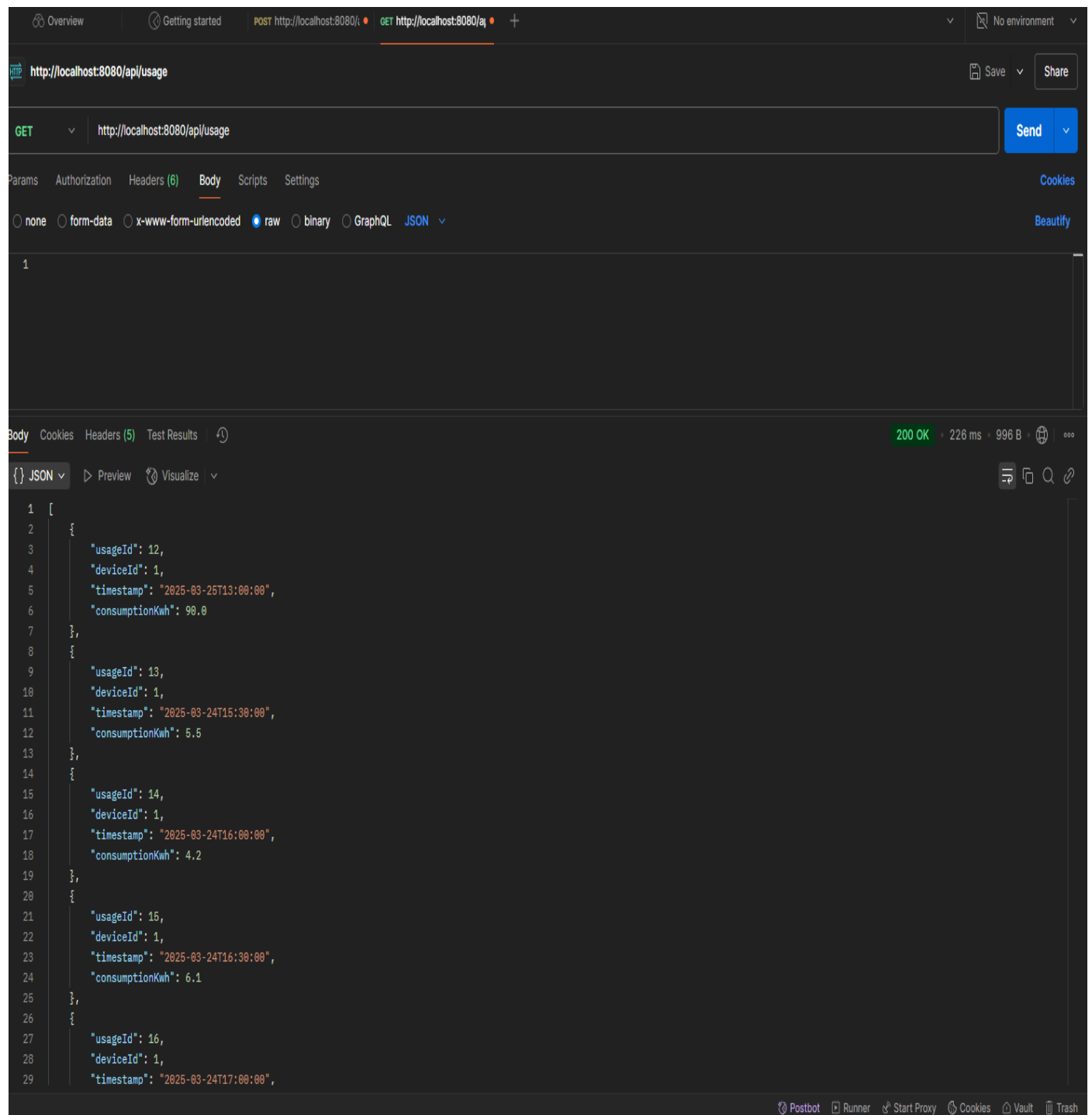
@Test annotation => It is a test marker that comes with Junit5. This method is run as a test.

contextLoads() method => It tests whether the application context is loaded successfully. This test is actually a general health check that shows that the application can be started properly.

RESULTS

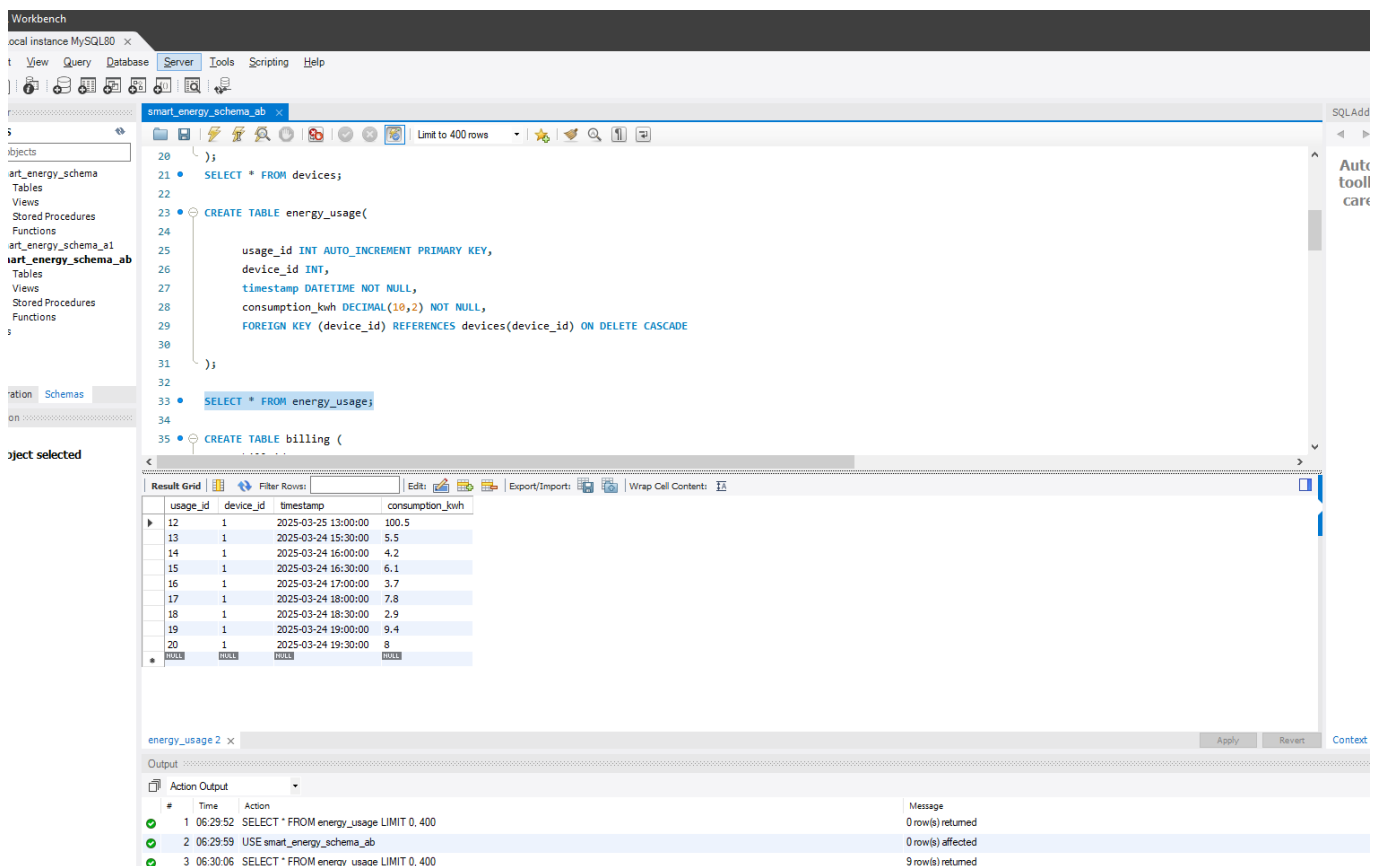
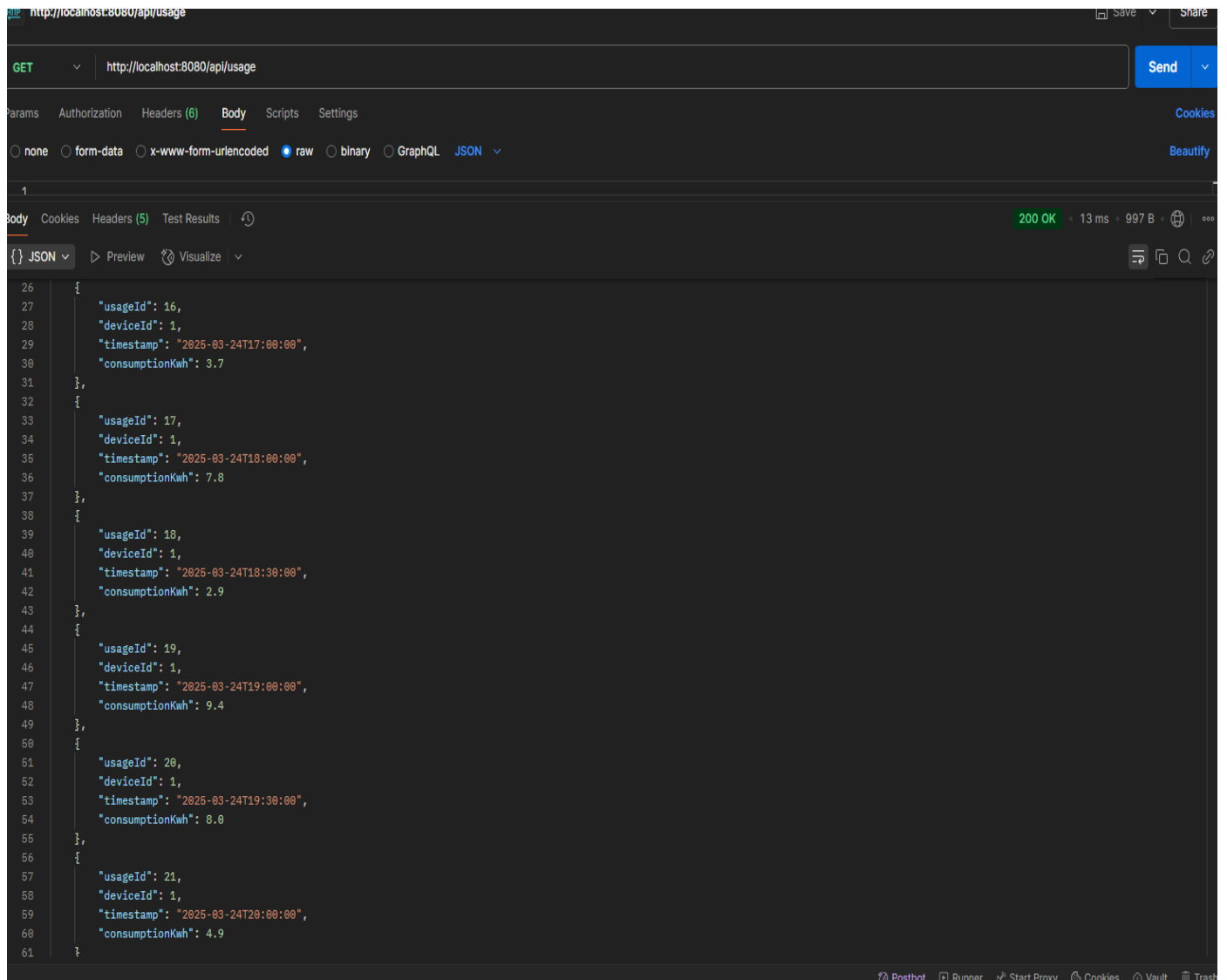
GET

Operation was tested successfully(listing).Now we can pull all EnergyUsage data registered for the user via Postman. After the application was successfully launched, Postman was used to list all energy consumption records with a GET request. The following sample data was obtained as a result of the request made to <http://localhost:8080/api/usage>:

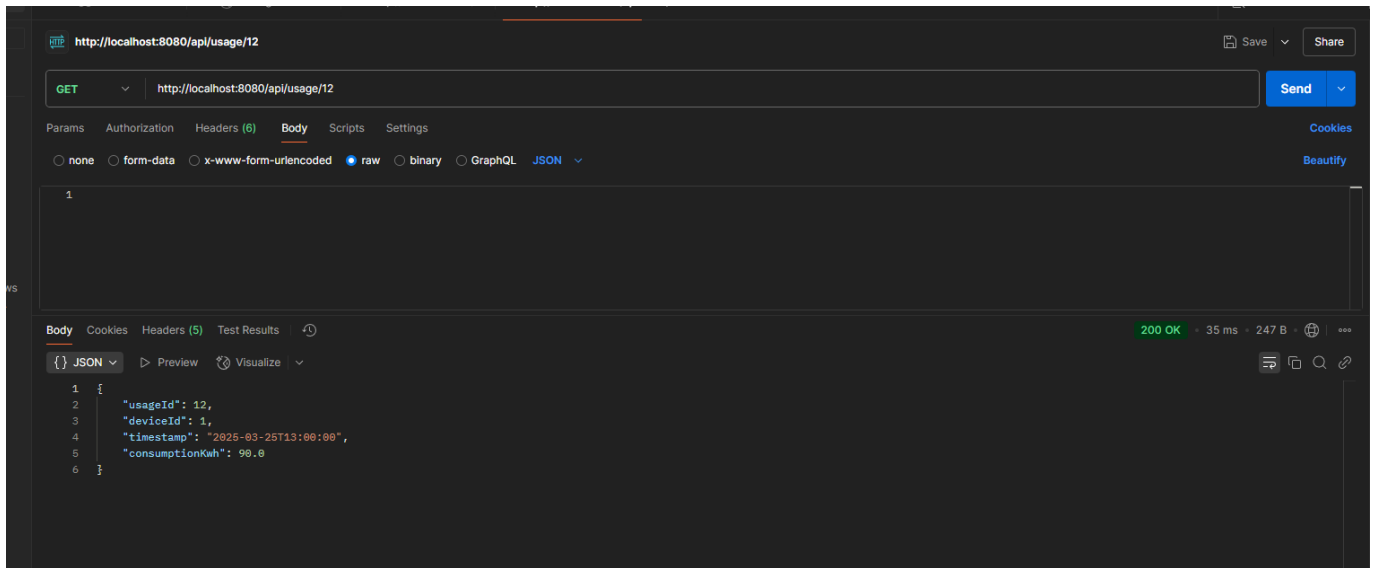


The screenshot shows the Postman interface with a GET request to `http://localhost:8080/api/usage`. The response is a 200 OK status with a 226 ms response time and 996 B of data. The response body is a JSON array of 5 energy usage records.

```
1 [
2   {
3     "usageId": 12,
4     "deviceId": 1,
5     "timestamp": "2025-03-25T13:00:00",
6     "consumptionKwh": 90.0
7   },
8   {
9     "usageId": 13,
10    "deviceId": 1,
11    "timestamp": "2025-03-24T15:30:00",
12    "consumptionKwh": 5.5
13  },
14  {
15    "usageId": 14,
16    "deviceId": 1,
17    "timestamp": "2025-03-24T16:00:00",
18    "consumptionKwh": 4.2
19  },
20  {
21    "usageId": 15,
22    "deviceId": 1,
23    "timestamp": "2025-03-24T16:30:00",
24    "consumptionKwh": 6.1
25  },
26  {
27    "usageId": 16,
28    "deviceId": 1,
29    "timestamp": "2025-03-24T17:00:00",
```



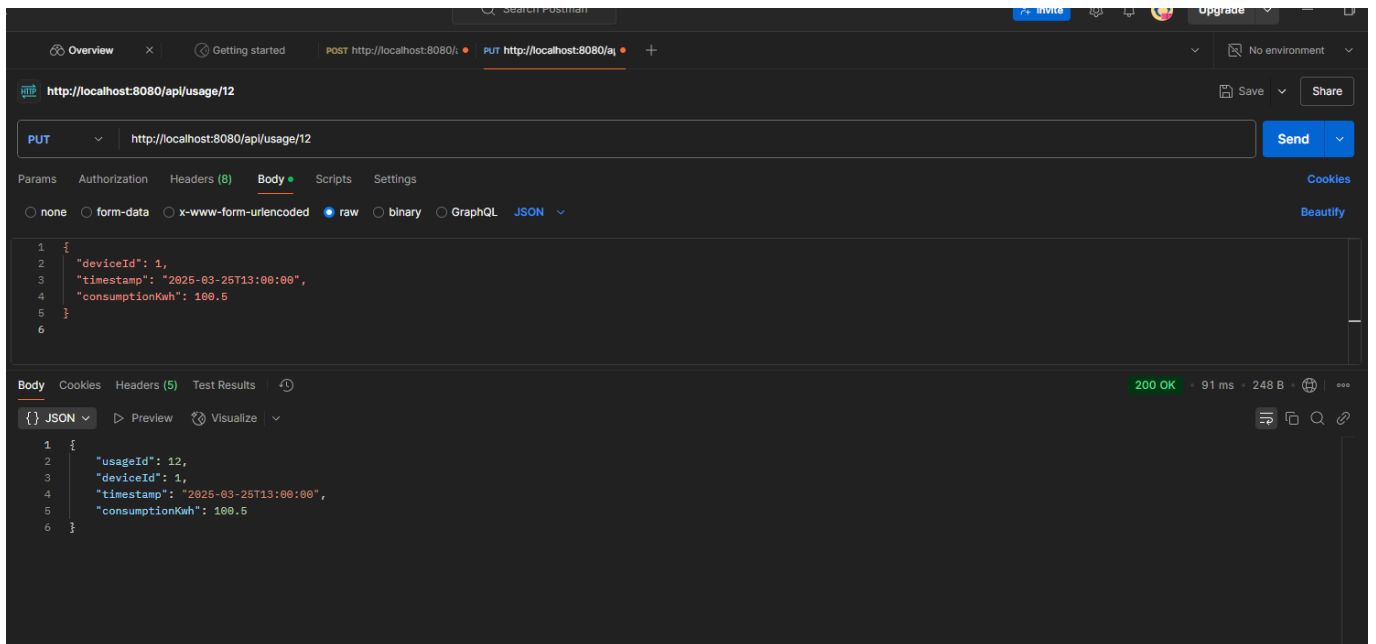
GETBYID (api/usage/{id})



Another important function of the application is that we can pull data according to a specific record ID. So you can find an example in the image above. For example, for a data with usage id : 12

With the GET/api/usage/12 request, we reached the record with usageId value 12 and this allowed us to directly view a specific energy consumption record for users. It is a very useful feature especially in detailed analysis or single data operations.

PUT(UPDATE)



We use PUT request to change/update a record in the database. In this example, we changed the energy consumption record's data from 90.3 to 100 , which is usageId :12. We updated it with a new timestamp and new consumption amount.

Layer Description

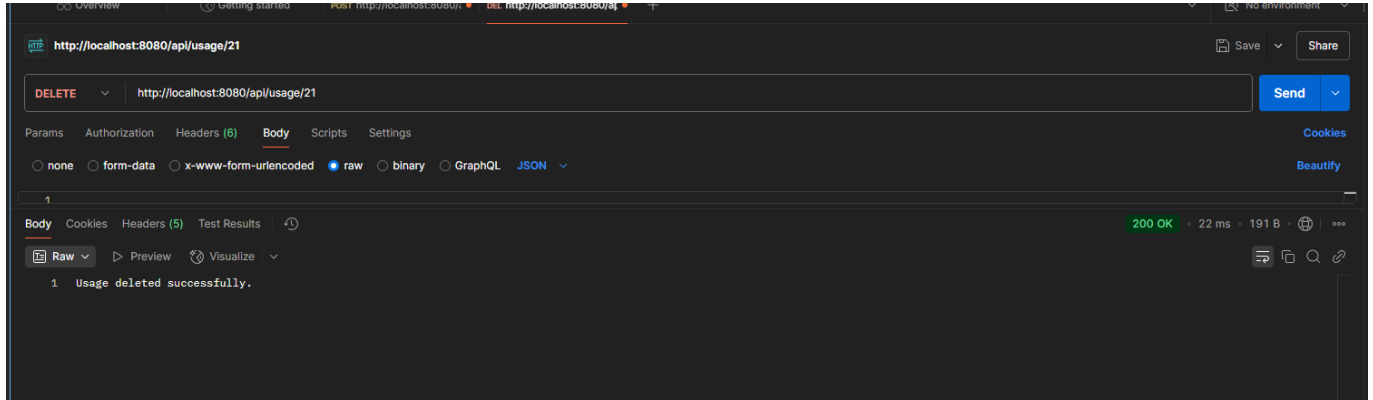
Controller = Received the API request (@PutMapping)

Service = Executed the update logic

Repository = Found the data with JPA, updated it with the new value and saved it

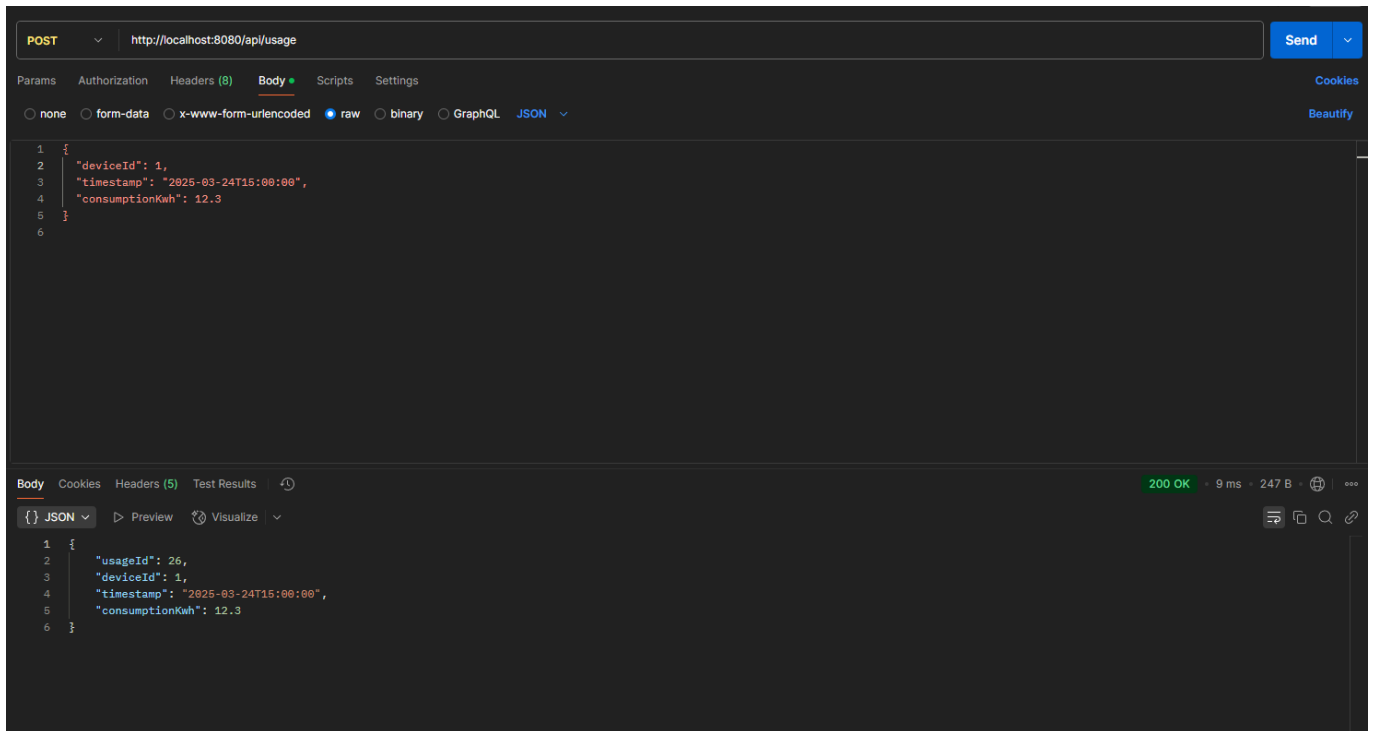
Database = Updated the relevant row in the energy_usage table.

DELETE



Another operation I tested in the project was data deletion . I used the DELETE request to delete the record with the usageld value of 21. The request was completed successfullt and I saw that the data was removed from the system. Also ,the system returned a ‘200 OK’ response in this operation. This showed that the deletion operation worked properly.

POST



One of the first things I tested while developing the project was adding data. To do this, I created a new energy usage data in the system by sending a POST request via the Postman tool. The system returned a 200 OK response and showed the added data in detail.

CHALLENGES AND SOLUTIONS

While developing this project, I encountered some difficulties , both technically and in terms of configuration. However , I managed to solve each of them with detailed research and trials. Below , I explain the main problems I encountered during this process and how I solved them .

1-Foreign Key Error Problem

When I first created the project, I encountered an error like this when trying to add data :

“ message” : “ Can not add or update a child row : a foreign key constraint fails”.

The energy_usage table in the database was connected to the devices table via the device_id field. However , the deviceId value I sent with the POST operation was an ID that was not defined in the devices table.

Solution :

I immediately checked the devices table and entered a suitable data there, such as deviceId : 1 . Thus, the referenced data was created and the POST operation worked successfully.

2- Empty data update (put)

During data update (PUT operation), I was getting an error when the body content I sent was missing or left empty.

Solution:

In the service layer, I first created a structure that checked the existence of the data to be updated

```
EnergyUsage existing = energyUsageRepository.findById(id)
```

```
.orElseThrow(() -> new ResourceNotFoundException("Usage not found with this id: " + id));
```

Thus, if the user sends an incorrect ID, the system gives a nice error message and a readable response is returned instead of the error.

3- Incorrect GET (DELETE) after record deletion

After deleting a record, when I wanted to perform a GET operation with the same ID , the system exploded.

Solution:

I created an exception handling mechanism. With the ResourceNotFoundException class, I return a special error message if the data can not be found. This also informs the user:

```
new ResourceNotFoundException("Usage not found with this id: " + id);
```

4- Not Understanding General Errors

Some errors were seen as 500 Internal Server Error in the system and the error messages were incomprehensible.

Solution:

I created a global exception handler and created a structure that provides detailed information for each error. For example, I handled all errors in the system through the following structure.

```
@ExceptionHandler(Exception.class)
```

```
public ResponseEntity<Map<String, Object>> handleGeneralException(Exception ex) { ... }
```

This way, the error I encounter in Postman is shown in a detailed and explanatory manner. Debugging has become much easier.

CONCLUSION

While doing this project, I developed a RESTful web service in order to manage energy consumption data effectively and to provide users with easy access to this data. While creating the backend architecture using Spring Boot, I performed data storage operations with the MySQL database. The system creates, updates, lists and deletes energy consumption records by receiving data from the user. These CRUD operations both meet the basic needs of the system and increase the flexibility of the software.

With the error management (Exception Handling) I applied in the project, possible errors were communicated to the user in a simple and understandable way. In addition, I performed the test stages of the application with Postman and verified that all API operations worked correctly.

As a result of this work, I think I have created a simple but functional system for users who want to monitor energy consumption. The system I developed meets basic data processing needs and forms a basis for more advanced systems.

FUTURE WORK

I am thinking of adding some improvements to this project in the future :

- For example, we can add systems such as user login and authentication to our system and make the system multi-user and diverse.

- We can show the devices and data of each user, and in addition, if we develop methods such as user-specific data filtering, we can provide easier access to data.

- A frontend interface can be prepared where we present data to users through graphics.
- We can add analysis and recommendation systems for energy consumption, and offer artificial intelligence-supported recommendations to the user.
- Increasing the reliability of our system by integrating more advanced logging and performance monitoring systems can also be a goal for us.