**Power-Grid-System-Documentation**

**Project Overview:**

- \*\*API Gateway (`api-gateway`)\*\*: Routes requests to different services based on path.

- \*\*Service Registry (`service-registry`)\*\*: Uses Eureka to manage service registration and discovery.

- \*\*Sensor Data Service (`sensor-data-service`)\*\*: Handles the sensor data.

- \*\*Real-Time Monitoring Service (`real-time-monitoring-service`)\*\*: Provides real-time monitoring functionality.

- \*\*Load Forecasting Service (`load-forecasting-service`)\*\*: Provides load forecasting functionality.

**Step-by-Step Breakdown of Your Project:**

1. Service Registry (`service-registry`):

This service uses Eureka for service discovery. You've already configured it in `application.properties` to disable the registry and client fetching (`eureka.client.fetch-registry=false` and `eureka.client.register-with-eureka=false`), because this service only needs to act as the registry itself.

2. API Gateway (`api-gateway`):

The API Gateway routes incoming requests to the corresponding service based on the path, using Spring Cloud Gateway and Eureka for service discovery. Your `application.yml` is set up correctly to route requests to the `SENSOR-DATA-SERVICE` and `LOAD-FORECASTING-SERVICE`.

3. Load Forecasting Service (`load-forecasting-service`):

Your `load-forecasting-service` has a simple controller with a `/load-forecast` endpoint, and it's configured correctly to register itself with Eureka.

4. Sensor Data Service and Real-Time Monitoring Service:

Since you mentioned you've already created similar services, I’ll assume they're set up similarly to the `load-forecasting-service`.

**Project Structure Recap:**

```

powergrid-system/

├── api-gateway/ # API Gateway

│ └── src/

│ └── application.yml

├── sensor-data-service/ # Sensor Data Service

│ └── src/

│ └── application.properties

├── real-time-monitoring-service/ # Real-Time Monitoring Service

│ └── src/

│ └── application.properties

├── load-forecasting-service/ # Load Forecasting Service

│ └── src/

│ └── application.properties

└── service-registry/ # Eureka Server

└── src/

└── application.properties

```

**Step-by-Step Breakdown of the Power Grid Micro service Architecture**

**Step 1: Create a Spring Boot Project for Each Service**

Create a Spring Boot project for each of the services you mentioned: `api-gateway`, `sensor-data-service`, `real-time-monitoring-service`, `load-forecasting-service`, and `service-registry`.

**1. \*\*Create the `api-gateway` project\*\*:**

- Use [Spring Initializr](https://start.spring.io/) to generate the base project for the API Gateway:

- Dependencies: Spring Web, Spring Cloud Gateway, Eureka Discovery Client

- Update the `pom.xml` to include the necessary dependencies for Spring Cloud Gateway and Eureka:

```xml

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-gateway</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

```

**2. \*\*Create the `service-registry` project\*\* (Eureka Server):**

- Use Spring Initializr to generate a simple Spring Boot project with the Eureka Server dependency.

- Dependencies: Eureka Server

- Update `application.yml` to configure it as an Eureka Server:

```yaml

spring:

application:

name: service-registry

server:

port: 8761

eureka:

server:

enable-self-preservation: false

```

**3. Create the other services (`sensor-data-service`, `real-time-monitoring-service`, and `load-forecasting-service`):**

- Use Spring Initializr to generate a simple Spring Boot project for each service.

- Dependencies: Spring Web, Eureka Discovery Client

- Update `application.yml` for each service to register them with Eureka:

```yaml

spring:

application:

name: service-name # Replace with service name, e.g., sensor-data-service

server:

port: 8082 # Change port for each service

eureka:

client:

service-url:

defaultZone: http://localhost:8761/eureka/

register-with-eureka: true

fetch-registry: true

```

---

**Step 2: Setting Up the Services**

**Service Registry (`service-registry`)**

You’ve already configured the Eureka Server (Service Registry). The `application.yml` should look like this:

```yaml

spring.application.name=service-registry

server.port=8761

eureka.client.fetch-registry=false

eureka.client.register-with-eureka=false

```

This ensures that the Eureka Server doesn’t try to register itself with Eureka.

**API Gateway (`api-gateway`)**

The `application.yml` in the API Gateway is configured for routing to other services:

```yaml

server:

port: 8081

spring:

application:

name: api-gateway

cloud:

gateway:

routes:

- id: SENSOR-DATA-SERVICE

uri: lb://SENSOR-DATA-SERVICE # Load-balanced URI

predicates:

- Path=/sensor-data/\*\*

- id: LOAD-FORECASTING-SERVICE

uri: lb://LOAD-FORECASTING-SERVICE

predicates:

- Path=/load-forecast/\*\*

eureka:

client:

register-with-eureka: true

fetch-registry: true

service-url:

defaultZone: http://localhost:8761/eureka/

```

**Load Forecasting Service (`load-forecasting-service`)**

Your `load-forecasting-service` has a controller with a `/load-forecast` endpoint. The `application.yml` is configured to register with Eureka and the controller looks like this:

```application.properties

spring.application.name=load-forecasting-service

server.port=8083

eureka.client.service-url.defaultZone=http://localhost:8761/eureka/

eureka.client.register-with-eureka=true

eureka.client.fetch-registry=true

```

```java

package com.load\_forecasting.controller;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/load-forecast")

public class HomeController {

@GetMapping

public ResponseEntity<?> test() {

return ResponseEntity.ok("load-forecasting-service is working");

}

}

```

**Sensor Data Service (`sensor-data-service`)**

Similarly, the `sensor-data-service` would have a similar `application.yml` and controller setup to handle sensor data. For example:

```application.properties

spring.application.name=sensor-data-service

server.port=8082

eureka.client.service-url.defaultZone=http://localhost:8761/eureka/

eureka.client.register-with-eureka=true

eureka.client.fetch-registry=true

```

```java

package com.sensordata.controller;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/sensor-data")

public class SensorDataController {

@GetMapping

public ResponseEntity<?> getSensorData() {

return ResponseEntity.ok("sensor-data-service is running");

}

}

```

**Step 3: Running the System**

**1. Start the Service Registry (Eureka Server):**

- Run the `service-registry` first to initialize Eureka.

```bash

cd service-registry

mvn spring-boot:run

```

**2. Start the Other Services:**

Now, run the services in any order after the Eureka server is up.

```bash

cd api-gateway

mvn spring-boot:run

```

```bash

cd sensor-data-service

mvn spring-boot:run

```

```bash

cd load-forecasting-service

mvn spring-boot:run

```

**3. Access the Services:**

You can now access the services via the API Gateway at `localhost:8081`:

- \*\*Sensor Data Service\*\*: `http://localhost:8081/sensor-data`

- \*\*Load Forecasting Service\*\*: `http://localhost:8081/load-forecast`

**4. Verify Service Registration with Eureka:**

- Visit `http://localhost:8761/` to see the Eureka dashboard and verify that all the services are registered correctly.

**Step 4: Enhancing the Project (Optional)**

- Security: You can secure the services and the API Gateway using Spring Security or OAuth.

- Database Integration: If needed, you can integrate each service with a database to persist data.

- Scaling: Add more microservices or scale existing ones as needed.

- Logging: Set up centralized logging using tools like ELK Stack (Elasticsearch, Logstash, Kibana) or Spring Cloud Sleuth and Zipkin for distributed tracing.

**Conclusion:**

This guide should now provide you with the complete set of steps to:

1. Set up each service (API Gateway, Service Registry, and individual services).

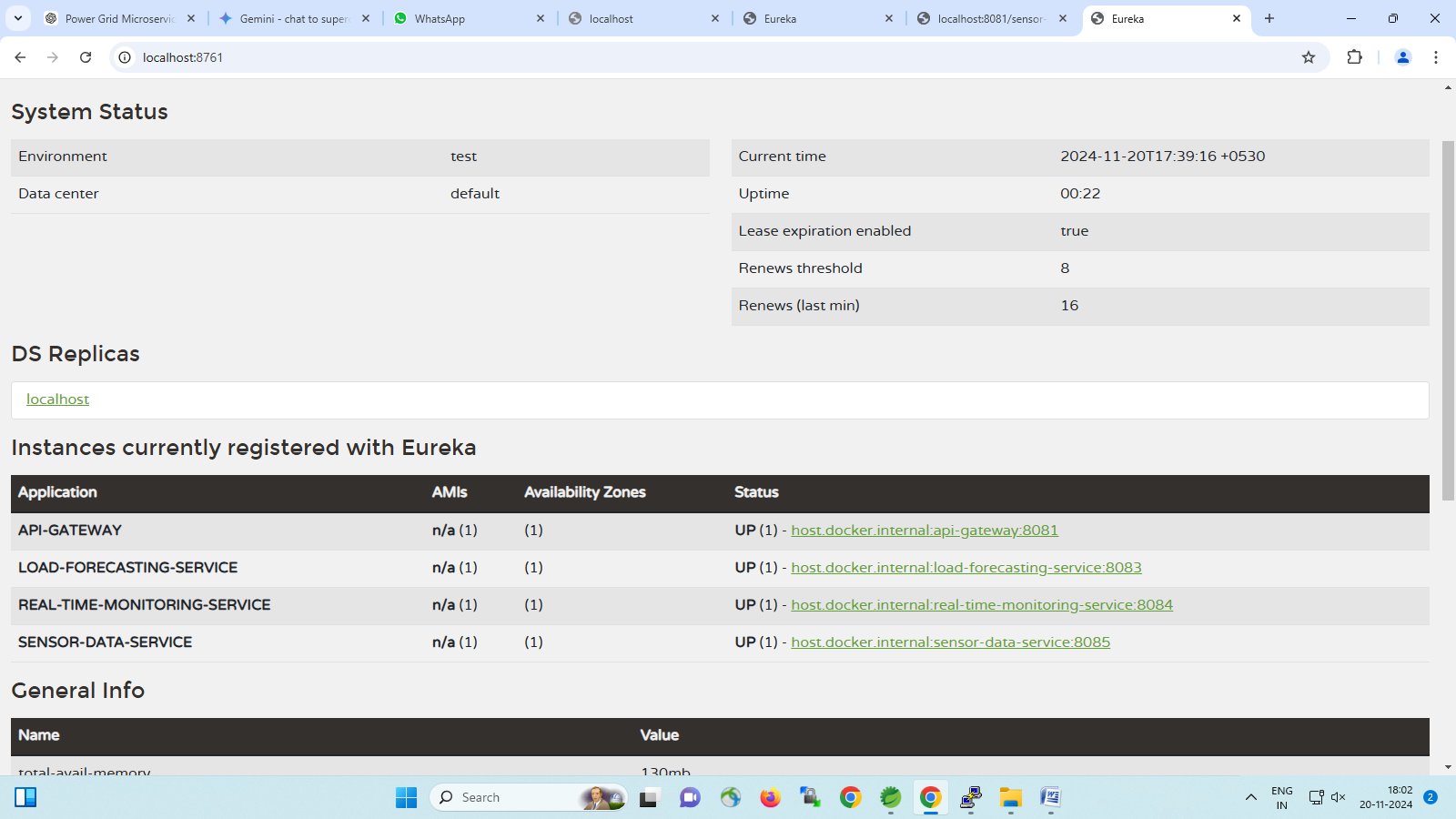
2. Configure the services to register with Eureka.

3. Route requests via the API Gateway.

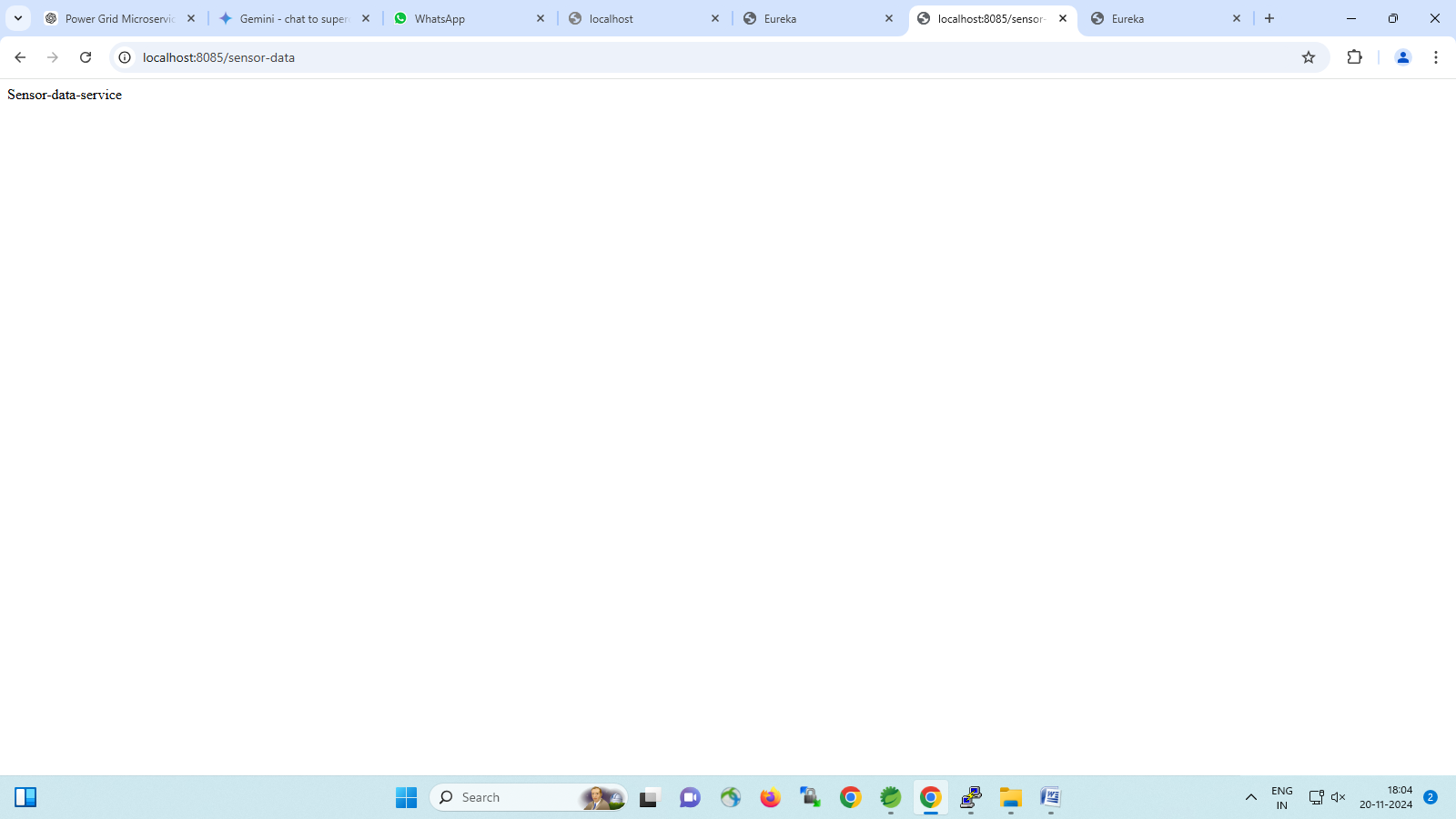
4. Run the system and access services via Eureka and the API Gateway.

**Some Screenshots:**

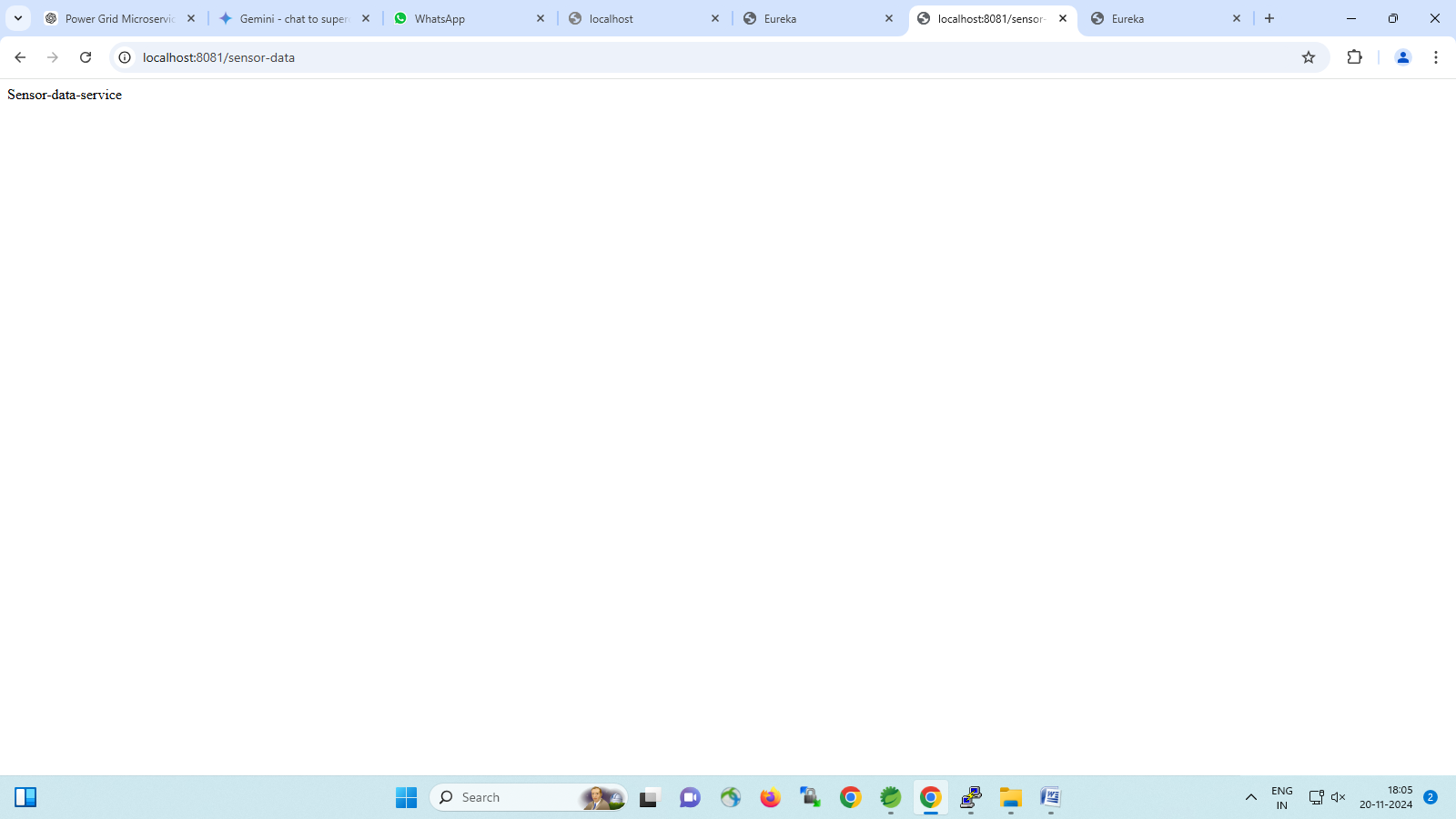
1. Eureka dashboard: All the services are registered correctly.



1. The API is running on its local server at port 8085.



1. The API is running on its API Gateway server at port 8081.



1. The API is running on its Eureka server at host.docker.internal with port 8081.

