Gateway, Monitoring and User Access Control

Cloud and Service Oriented Computing

Group 4: API Gateway and Monitoring	
Students	
David Mata	up202003557@edu.fe.up.pt
Felippe Silva	up202000091@edu.fe.up.pt
Nuno Ferreira	up202000059@edu.fe.up.pt
Pedro Almeida	up202003029@edu.fe.up.pt

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Master in Software Engineering

Teacher: Jorge Barbosa

Índice

1. Introduction

The main purpose of the development of this software is to develop a microservices-based information system for city mobility. To achieve this, each group of Cloud and Service-Oriented Computing will be responsible for the implementation of a part of the system and its respective services.

In the below image it can be found a diagram representing the architecture of the gateway and monitoring solution. In our architecture, we have 3 services, the Gateway, the Health Checking and the Log Aggregation plus n other services that represent the services that are to be developed by the other groups. The gateway performs authentication and authorization on the users and redirects them based on their role to the corresponding service; the health checking service which has the single responsibility of invoking the /health endpoint on the other services to maintain a status over time of the system's state; finally, the log aggregation service which is responsible for storing all the services logs and present them to the user once required.

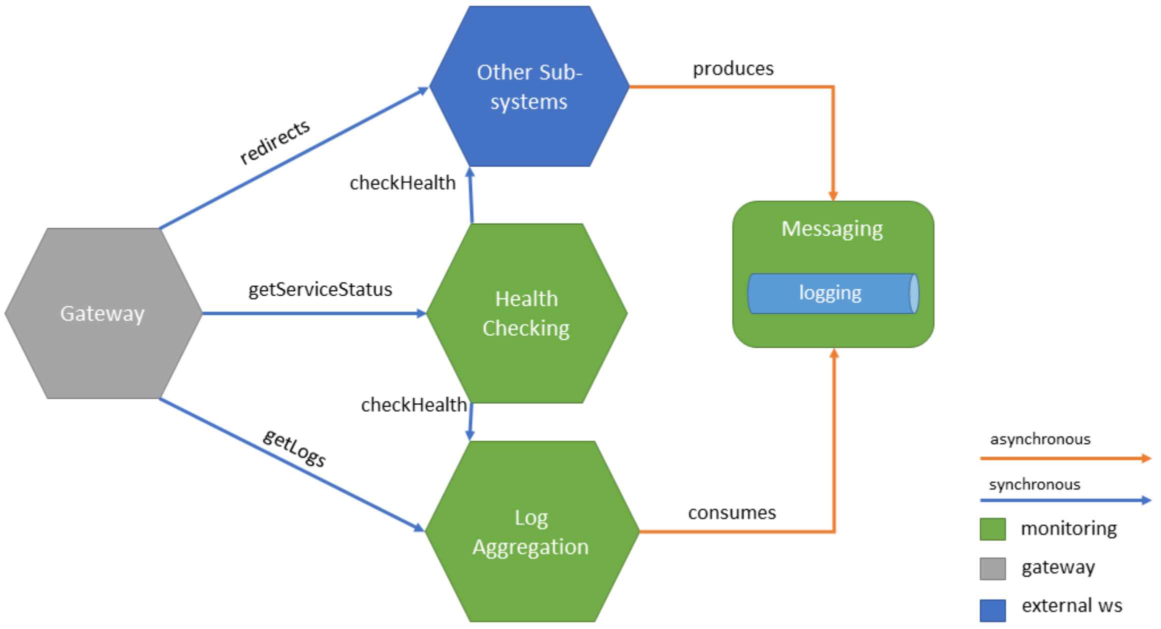


Figure 1 - Architecture of the Gateway and Monitoring solution.

In the below image, it is represented the domain model. It is possible to verify that a Service is represented by a name, IP, port and ID. Also, each Service will have different Endpoints. Each Service will also have the possibility to make Logs that are represented by a message, timestamp, MessageType and OperationType. Each User is also represented by a

ID, an email and one or more tokens. Each user can access different Services at the same time and each Service can be accessed by different Users at the same time.

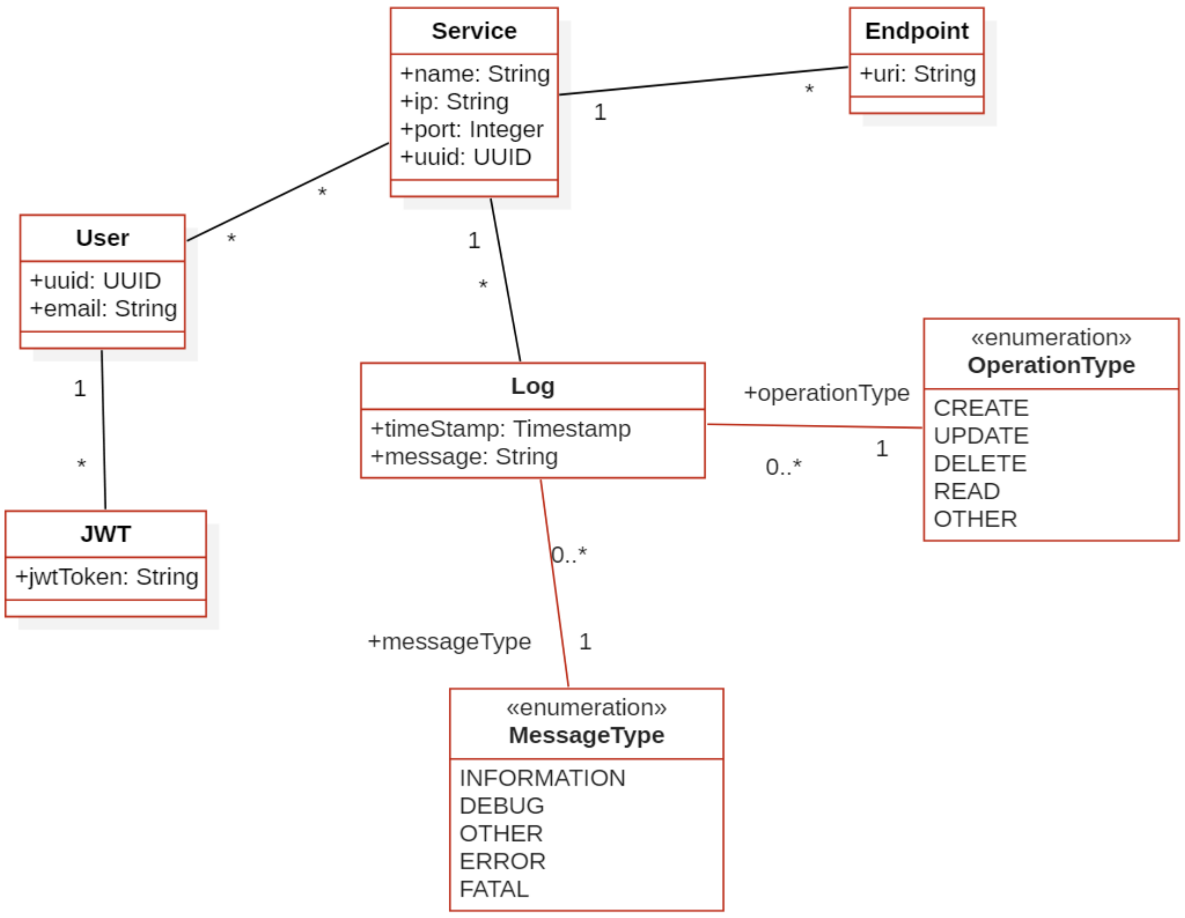


Figure 2 - Class diagram representing the domain model.

2. Gateway

Since the system has different microservices providing data there is the need to provide a single interface to the multiple APIs. To avoid invoking multiple services and combining results, an API gateway that stands between the different microservices and the client can provide a single entry point to the APIs.

With a monolithic approach arises some problems when different clients have distinct requirements. Not all clients need all the returning data, probably leading to a decrease in performance. An API gateway can provide different versions of endpoints tackling this problem.

Basically, this gateway encapsulates the underlying system and decouples from the client. Although the API gateway has the capability to encapsulate different functionalities, in the context of this project, this report will focus on the encapsulation of some functionalities. The main functionalities are logging/tracing (monitoring), routing, and user control access (authorization and authentication).

2.1. Services

The gateway allows that the authentication/authorization doesn't have to be implemented in every microservices. The API gateway ensures that the endpoints are secure to only be accessible to authorized clients. In essence, it controls the access to the microservices hiding them from the client, encapsulating the authorization and authentication functionalities.

One of the key functions of an API gateway is request routing. The API gateway implements some API operations by routing requests to the corresponding service. When it receives a request on a specific endpoint, the API gateway knows which service that request needs to be routed to.

Beyond the request routing, different microservices can provide different communication protocols. This can be a problem, especially if the client communicates directly to the microservices without an intermediate layer. So with the API gateway, a single protocol is defined so that the client has to adapt to only one communication protocol. All of these functionalities are represented in the below image.

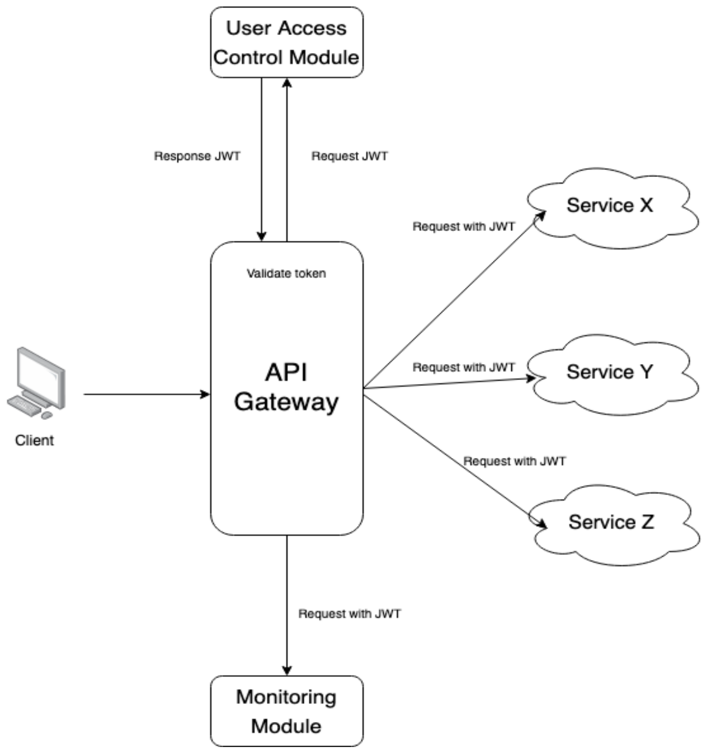


Figure 3 - Services representation of the API Gateway system.

2.2. Service API's and Collaborations

In the context of this project, the API gateway to be integrated that will serve as the main doorway to the system will be composed of the functionalities addressed in the above sections. Those functionalities and respective operations will be described in the below section and table.

The API Gateway is deployed in the cloud, more specifically on Heroku, a Platform as a Service (PaaS). The project has been configured to be automatically deployed to the Heroku Cloud Application Platform every time a change is pushed to the GitHub repository (to the main branch). The following API specifications are available at https://cosn-api-gateway.herokuapp.com/swagger.

OpenAPI: https://cosn-api-gateway.herokuapp.com/swaggerv				
Functions	Operations	Method	Endpoint	Collaborations
Routing	getMyWorkflows()	GET	/getMyWorkflows	MAS service

	runWorkflow()	POST	/runWorkflow	MAS service
	getWorkflowById()	GET	/getWorkflowById	MAS service
	runVisualization()	POST	/runVisualization	Visualiz-ation service
	getAllCities()	GET	/getAllCities	City service
	getSensorByTypeStatic()	GET	/getSensorByType/static	City service
	getSensorValuesStatic()	GET	/getSensorValue/static	City service
	getSensorTypeMobile()	GET	/getSensorType/mobile	City service
	getSensorCityMobile()	GET	/getSensorCity/mobile	City service
	getAllDomainModels()	GET	/getAllDomainModels	Domain Model service
	getSensorsValues()	GET	/getSensorsValues	Real System service
	getAlgorithms()	GET	/getAlgorithms	Algorith ms service
	getAlgorithmsName()	GET	/getAlgorithmsName	Algorith ms service
	getAlgorithmById()	GET	/getAlgorithmById	Algorith ms service
User Access Control	register()	POST		none
	login()	POST		none

	getRole()	GET	/role/{roleId}	none
	getRoles()	GET	/roles	none
	changeUserRole()	POST	/changeUserRole	none
	getUserFromToken()	POST	/getUserFromToken	none

Table 1 - API Gateway services mapped to their operations, collaborations and methods.

As a gateway, we will have to provide endpoints some of which are meant to be used to call some of the underlying services. All routing endpoints are protected with token verification. The token retrieved from login authentication is sent in the header and validated before being forwarded to the service. For each endpoint access, there is also a role check. For endpoints interfacing with the Domain Model Service, Algorithms Service and City Service, the user must have the role of a researcher or administrator to access them. For access to the other endpoints, there is no role filter, but any role authorisation can be added at any moment.

- **getMyWorkflows**()
    - This endpoint returns all workflows from a user, using the header token sent;
    - Request Header: AuthToken (authentication token);
- **runWorkflow**()
    - This endpoint executes a workflow, passing as request body the workflow ID, domain model ID and a description;
    - Request Header: AuthToken (authentication token);
    - Request Body: worflowId, domainModelId and description;
- **getWorkflowById**()
    - The following endpoint returns the workflow by the ID sent in the headers request;
    - Request Header: AuthToken (authentication token) and ID (workflow ID);
- **getAllCities**()
    - This endpoint returns all cities available in the city service;
    - Request Header: AuthToken (authentication token);
- **getSensorByTypeStatic**()
    - This endpoint returns all static sensors in a city, by type and radius;
    - Request Header: AuthToken (authentication token);
    - Query Param: cityID, type, radius;

- **getSensorValuesStatic**()
    - This endpoint returns all static sensors in a city, by radius;
    - Request Header: AuthToken (authentication token);
    - Query Param: cityID, radius;
- **getSensorTypeMobile**()
    - This endpoint returns all mobile sensors in a city, by type;
    - Request Header: AuthToken (authentication token);
    - Query Param: cityID, type;
- **getSensorCityMobile**()
    - This endpoint returns all mobile sensors in a city;
    - Request Header: AuthToken (authentication token);
    - Query Param: cityID;
- **getAllDomainModels**()
    - This endpoint returns all available domain models containing the network topology description;
    - Request Header: AuthToken (authentication token);
- **getSensorsValues**()
    - This endpoint returns the sensor values of a defined sensor by time interval;
    - Request Header: AuthToken (authentication token);
    - Query Param: sensorsIds, startTime, endTime;
- **getAlgorithms**()
    - This endpoint returns all available algorithms;
    - Request Header: AuthToken (authentication token);
- **getAlgorithmById**()
    - This endpoint returns an algorithm by ID;
    - Request Header: AuthToken (authentication token);
- **getAlgorithmsName**()
    - This endpoint returns only the names of all available algorithms;
    - Request Header: AuthToken (authentication token) and Id (algorithm ID);

Meanwhile, we will also have endpoints destined for the user access control system, these endpoints are meant to be used for registration and login purposes, the standard used for this system is explained further below.

The endpoints for these however are as follows:

- **register**()
    - This endpoint is important for registration purposes, it ensures the desired user information is gathered and stored on the database.
- **login**()
    - This endpoint works as a way to log in and validate user information with the users already created, and at the end, a jwt token is generated and returned.

- **getRole**()
    - This endpoint is made for getting information about a specific role like the name.
- **getRoles**()
    - This endpoint works as a way to get all available roles.
- **changeUserRole**()
    - This endpoint works as a way to change a users role, this can only be done by a user with the admin role.
- **getUserFromToken**()
    - This endpoint receives a jwt token and delivers its user and user information along with his role.

When it comes to User Access Control we will be using the JSON Web Token open standard, these web tokens will be used for authorization, as once a user is logged in, each subsequent request will include this token. The choice to use JWT also aids in the implementation of Single Sign-On (SSO) capabilities as it is widely used and will also provide a way for securely exchanging information, as JWTs is signed with secret or private keys.

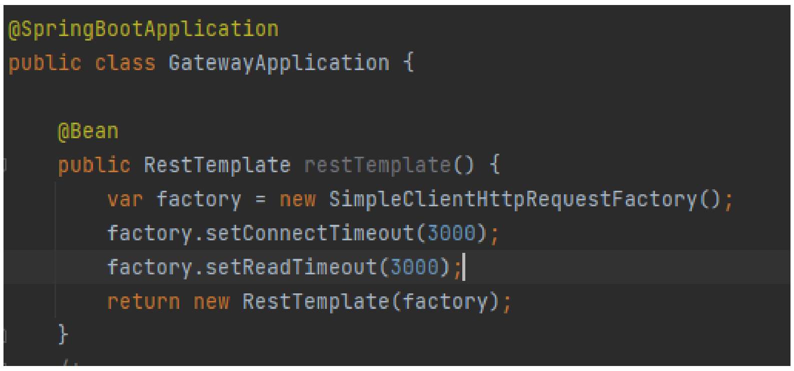
As for how it will work, a user will provide the gateway with the user information needed to create a web token, that token is then stored in the gateway, all communication requests sent by the user to the gateway needs to have this token within the request so we can direct the request to the underlying services, these services will then get information if needed on who the user is that is trying to execute a given procedure.

As explained above, the gateway is meant to act as a one protocol stop for all users, avoiding the myriad of protocols that can be used for the underlying services. For that communication protocol, we picked the “Representational State Transfer” also known as REST. This protocol was created in the 2000s as part of a dissertation by Roy Fielding and is now widely known and used for communication between different services or different parties.

Additionally, to describe the entire REST API the developed gateway will adopt Swagger in order to become more readable and easier to use across all possible clients. Swagger is an API description format built around OpenAPI that can be written in YAML or JSON. This format allows the specification of the available endpoints and corresponding HTTP methods, required parameters or authentication methods and other possibilities. Therefore this type of specification gives a clear insight into how the API responds to parameters. In addition, this API specification will benefit the API documentation and the interaction with the API gateway.

2.3 Implemented Patterns

In the case of API Gateway, when it comes to resilience patterns, the pattern that was implemented is the Timeout pattern. In this component, the service routing is handled by the RestTemplate library. The RestTemplate, which allows synchronous clients to make HTTP requests, has a default timeout. If the request timed out, Spring throws a ResourceAccessException. The underlying exception under this instance is java.net.SocketTimeoutException with the message 'Read timed out'. RestTemplate has an infinite timeout by default. Therefore, it has been configured to set a connection timeout of 3000 milliseconds and a read timeout of also 3000 milliseconds.



```
@SpringBootApplication
public class GatewayApplication {

    @Bean
    public RestTemplate restTemplate() {
        var factory = new SimpleClientHttpRequestFactory();
        factory.setConnectTimeout(3000);
        factory.setReadTimeout(3000);
        return new RestTemplate(factory);
    }
```

Fig. 4 - RestTemplate timeout configuration.

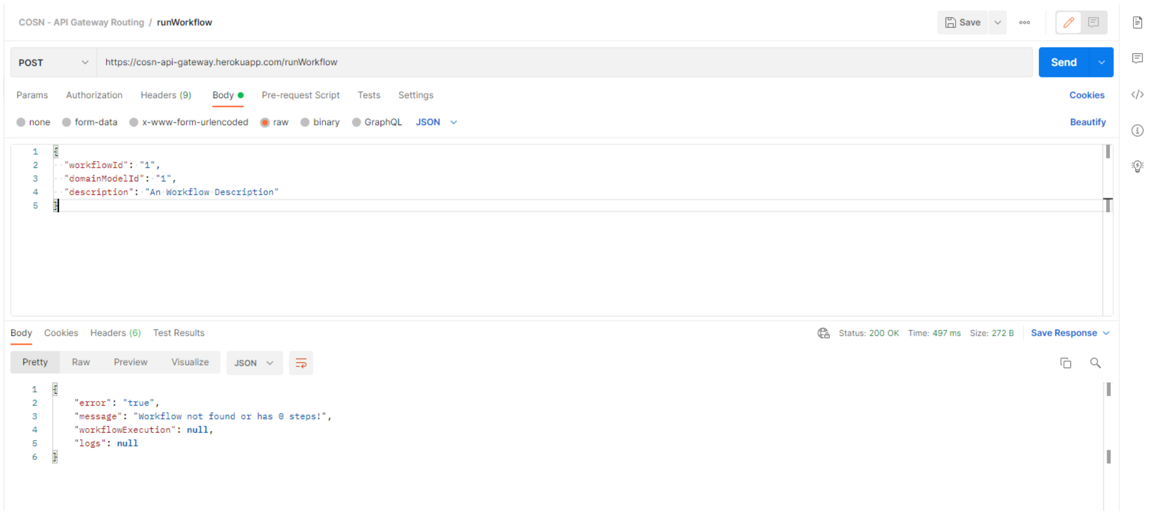
2.4 Security

When it comes to security, the gateway services are hosted on a server with an SSL implementation to guarantee the encryption of all the communications.

To guarantee that certain services are only accessible for authenticated users and that those users have the required authorizations, it was implemented the JWT pattern. To stop users from reusing these access tokens illegally, a validity duration of eight hours was assigned to each of the generated tokens. The token is encrypted with the help of a key to store service data, user data and the token's expiration date.

2.5. Tests

As for the tests performed with Postman, in this section are some test examples. All the remaining requests performed by API Gateway with Postman are attached in this document.



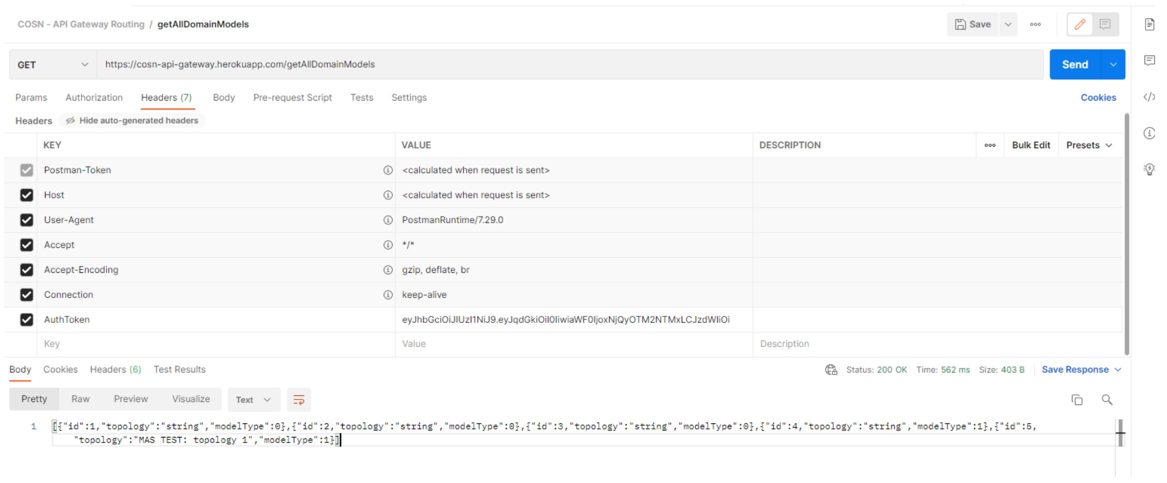
https://cosn-api-gateway.herokuapp.com/runWorkflow

```
{
  "workflowId": "1",
  "domainModelId": "1",
  "description": "An Workflow Description"
}
```

Status: 200 OK Time: 497 ms Size: 272 B

```
{
    "error": "true",
    "message": "Workflow not found or has 0 steps!",
    "workflowExecution": null,
    "logs": null
}
```

Fig. 5 - Postman test example with a POST request to /runWorkflow



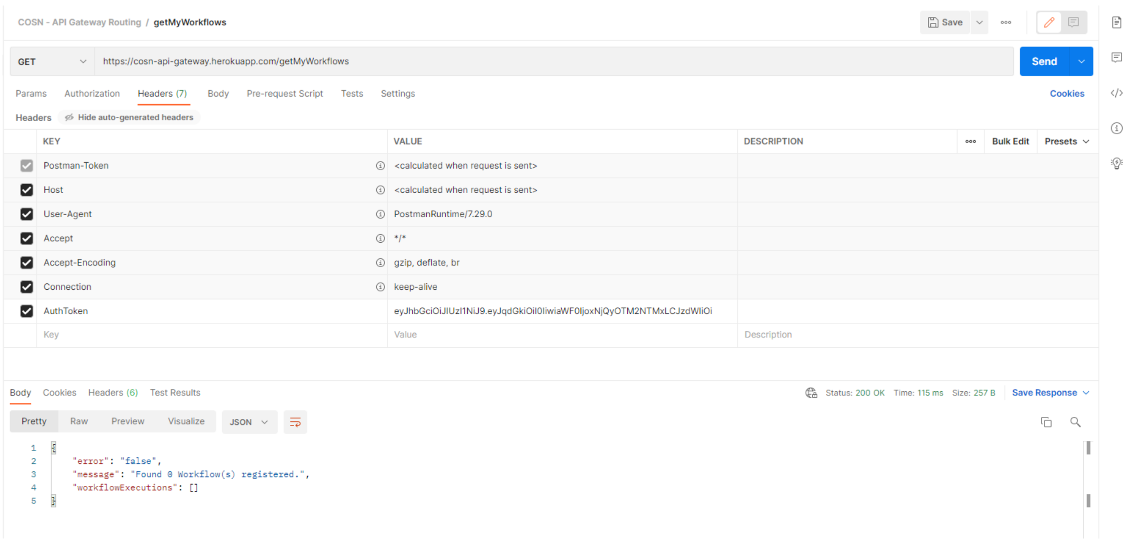
https://cosn-api-gateway.herokuapp.com/getAllDomainModels

KEY	VALUE	DESCRIPTION
✓ Postman-Token	<calculated when request is sent>	
✓ Host	<calculated when request is sent>	
✓ User-Agent	PostmanRuntime/7.29.0	
✓ Accept	*/*	
✓ Accept-Encoding	gzip, deflate, br	
✓ Connection	keep-alive	
✓ AuthToken	eyJhbGciOiJIUzI1NiJ9.eyJqdGkiOiI0IiwiaWF0IjoxNjQyOTM2NTMxLCJzdWIiOi	
Key	Value	Description

Status: 200 OK Time: 562 ms Size: 403 B

```
[{"id":1,"topology":"string","modelType":0},{"id":2,"topology":"string","modelType":0},{"id":3,"topology":"string","modelType":0},{"id":4,"topology":"string","modelType":1},{"id":5,
  "topology":"MAS TEST: topology 1","modelType":1}]
```

Fig. 6 - Postman test example with a GET request to /getAllDomainModels



COSN - API Gateway Routing / getMyWorkflows

GET https://cosn-api-gateway.herokuapp.com/getMyWorkflows

Headers (7)

	KEY	VALUE	DESCRIPTION
✓	Postman-Token	<calculated when request is sent>	
✓	Host	<calculated when request is sent>	
✓	User-Agent	PostmanRuntime/7.29.0	
✓	Accept	*/*	
✓	Accept-Encoding	gzip, deflate, br	
✓	Connection	keep-alive	
✓	AuthToken	eyJhbGciOiJIUzI1NiJ9.eyJqdGkiOiI0IiwiaWF0IjoxNjQyOTM2NTMxLCJzdWIiOi	
	Key	Value	Description

Status: 200 OK Time: 115 ms Size: 257 B

```
{
    "error": "false",
    "message": "Found 0 Workflow(s) registered.",
    "workflowExecutions": []
}
```

Fig. 7 - Postman test example with a GET request to /getMyWorkflow

3. Monitoring

Every microservice solution has to have at least a monitoring portion of the system so that the developers can debug possible problems that have occurred and they can check if the system is working normally.

For the decomposition process of the monitoring subsystem, we used a *Decomposition by Capability* (Richardson, 2018) approach. After further analysis of the application requirements, we noticed that the monitoring aspect of the system is composed of two different capabilities, represented in the below diagram.

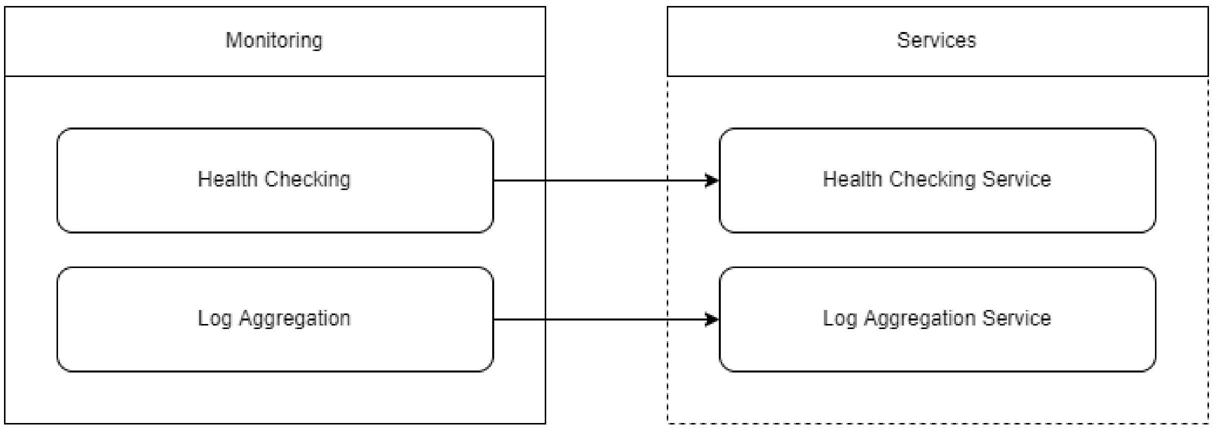


Figure 8 - The capabilities and services identification of the monitoring system.

The first of these capabilities is the *Health Checking* capability. This capability is responsible for permitting the registration and deregistration of the different services and based on the information provided by them perform periodic health check requests on those services; the capability is also responsible for providing data for users to query the current status of all registered services.

The second capability is the *Log Aggregation* capability and it is the most loosely coupled of the two since it makes extensive use of Apache Kafka; the single responsibility of this capability is the recording of the logs from the entire application, which are received through asynchronous messaging and can be queried by the user.

Each one of these capabilities will correspond to its respective service, since their operations are not interrelated in their core, the work they perform is largely independent and they do not require the same data in both services. The only thing that is needed to take into account is that the *Log Aggregation Service* may need to periodically check what are the currently registered services in the *Health Checking Service*.

3.1. Log Aggregation Service

After the subdivision done in the previous report, we ended up also with the Log Aggregation service. The *Log Aggregation* Service is the most loosely coupled of the two since it makes extensive use of Apache Kafka; the single responsibility of this capability is the recording of the logs from the entire application, which are received through asynchronous messaging and can be queried by the user.

3.1.1. Service Specification

OpenAPI: https://cosn-log-aggregation.herokuapp.com/api			
Operation	Method	Endpoint	Collaborations
getLogs()	GET	/logs	none
checkHealth()	GET	/health	none

Table 2 - The monitoring services mapped to their operations and collaborations.

- **getLogs()**
  - A service that accepts filter parameters and returns the logs that were gathered with the message broker.
  - The Parameters are optional, due to not being necessary to filter the logs, and can are specified as follows:
    - startTime: The lower limit of the timestamp filtering, this time is a Long that represents milliseconds;
    - endTime: The upper limit of the timestamp filtering, this time is a Long that represents milliseconds;
    - serviceName: The name of the service, the user can provide this if he wants only logs of a certain service.
- **checkHealth()**
  - A service that returns a status 200 response code for the Health Checking Service. This service does not return any kind of response body it only returns a 200 status.

3.1.2. Service Architecture

This service follows a Hexagonal Architecture, where we have two inbound adapters and one outbound adapter; this service makes use of NodeJS and its packages environment as its base. The first inbound adapter is the REST adapter which is created using the Express.JS library; the second inbound adapter is the Kafka adapter which is created using the KafkaJS npm module. The only outbound adapter is the database adapter which is created using Sequelize and the Postgres database connector package that can be found in the Node Package Manager (NPM).

This service uses PostgreSQL as its data store, which is a relational database and its data model consists of a single table where all the logging records are inserted. The table is made of 6 different fields, this table also represents the message format that the Kafka message must contain:

- **id**: This is an integer that increments as records are inserted, it is the primary key of the table and the sole index.
- **messageType**: This is an enumeration that can be one of five different values:
    - **INFORMATION**: To be used when the message that is sent to Kafka is that of a successful operation;
    - **DEBUG**: To be used when the message is that of an application which is being debugged;
    - **ERROR**: To be used when an exception occurs that doesn't impair the service's normal functioning;
    - **FATAL**: To be used when a fatal exception occurs that impairs the service's normal functioning;
    - **OTHER**: For other kinds of messages;
- **operationType**: This is an enumeration that can be one of five different values which represent the normal CRUD operations:
    - **CREATE**
    - **UPDATE**
    - **DELETE**
    - **READ**
    - **OTHER**
- **message**: A string where the user can serialize any message and register it;
- **timestamp**: Reception time of the asynchronous message.

3.1.3. Implemented Patterns

This service is already an **observability pattern** by itself, the **Log Aggregation Pattern;** this service is responsible for **recording the logs of the entire system** and presenting them to the users. Another observability pattern that was implemented was the **Health Checking Pattern**; this service has implemented a **/health endpoint** for the Health Checking Service to call and update the current system status of this service, the service returns a status 200 if the system is ok.

```
app.get('/health', (req, res) => res.status(200).send())
```

Code 1 - Code excerpt from the index.js file in the Log Aggregation Service

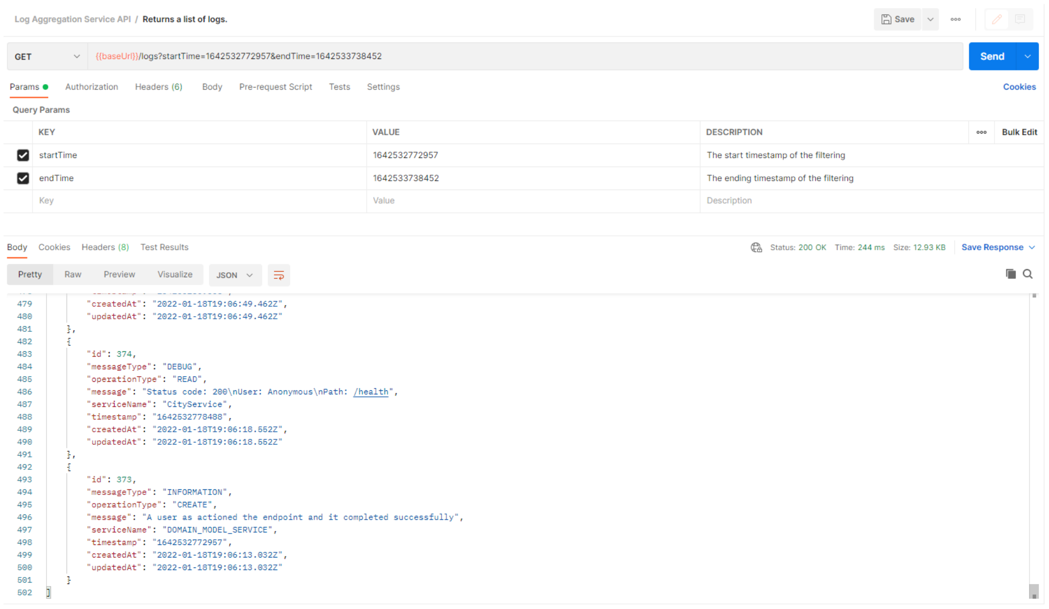
On the domain of **resilience patterns** there was implemented a single pattern on this service, the Fail Fast pattern that consists in validating the parameter inputs in order for it to fail sooner than later effectively turning the service more responsive, this pattern was implemented in the Kafka message consumer.

```
function writeLog(data) {
    return new Promise((resolve, reject) => {
        //Validate data
        if (typeof data.operationType === "undefined"
            || data.operationType === null) {
            reject("The operation type must not be ommited");
        }

        if (typeof data.messageType === "undefined"
            || data.messageType === null) {
            reject("The message type must not be ommited");
        }

        if (typeof data.timestamp === "undefined"
            || data.messageType === null) {
            reject("There must be a time stamp in the log");
        }
        /*
           More Operations that involve writing the log
        */
```

Code 2 - Code excerpt from the controller.js file in the Log Aggregation Service

3.1.4. Tests

The logging service only has 2 endpoints. the /logs and /health. The first 2 images represent various uses of the /logs endpoint, logging with timestamps and logging with the service name. The last image contains a test done on the /health endpoint which has responded with status 200.



GET {{baseUrl}}/logs?startTime=1642532772957&endTime=1642533738452

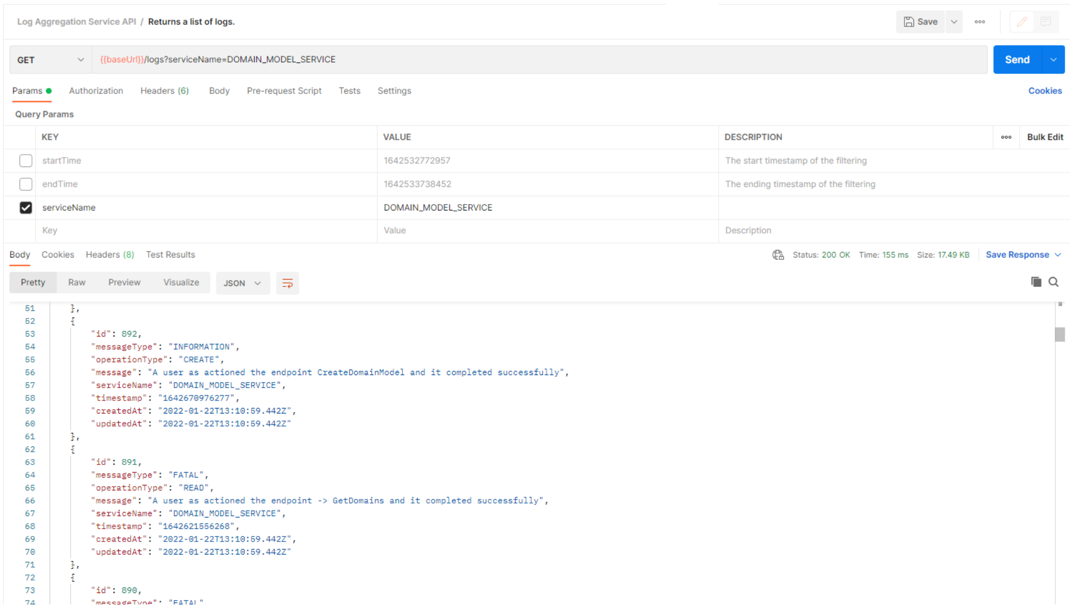
Query Params

	KEY	VALUE	DESCRIPTION
✓	startTime	1642532772957	The start timestamp of the filtering
✓	endTime	1642533738452	The ending timestamp of the filtering
	Key	Value	Description

Status: 200 OK Time: 244 ms Size: 12.93 KB

```
        "createdAt": "2022-01-18T19:06:49.462Z",
        "updatedAt": "2022-01-18T19:06:49.462Z"
    },
    {
        "id": 374,
        "messageType": "DEBUG",
        "operationType": "READ",
        "message": "Status code: 200\nUser: Anonymous\nPath: /health",
        "serviceName": "CityService",
        "timestamp": "1642532778488",
        "createdAt": "2022-01-18T19:06:18.552Z",
        "updatedAt": "2022-01-18T19:06:18.552Z"
    },
    {
        "id": 373,
        "messageType": "INFORMATION",
        "operationType": "CREATE",
        "message": "A user as actioned the endpoint and it completed successfully",
        "serviceName": "DOMAIN_MODEL_SERVICE",
        "timestamp": "1642532772957",
        "createdAt": "2022-01-18T19:06:13.032Z",
        "updatedAt": "2022-01-18T19:06:13.032Z"
    }
]
```

Figure 9 - Testing the /logs endpoint with the startTime and endTime parameters.



GET {{baseUrl}}/logs?serviceName=DOMAIN_MODEL_SERVICE

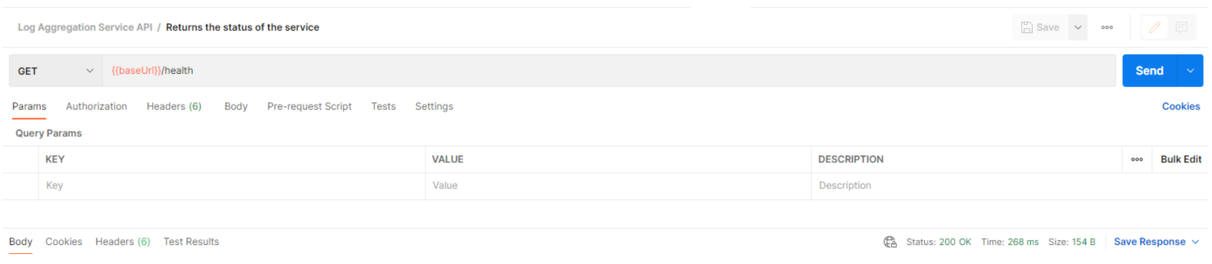
Query Params

	KEY	VALUE	DESCRIPTION
	startTime	1642532772957	The start timestamp of the filtering
	endTime	1642533738452	The ending timestamp of the filtering
✓	serviceName	DOMAIN_MODEL_SERVICE	
	Key	Value	Description

Status: 200 OK Time: 155 ms Size: 17.49 KB

```
    },
    {
        "id": 892,
        "messageType": "INFORMATION",
        "operationType": "CREATE",
        "message": "A user as actioned the endpoint CreateDomainModel and it completed successfully",
        "serviceName": "DOMAIN_MODEL_SERVICE",
        "timestamp": "1642670976277",
        "createdAt": "2022-01-22T13:10:59.442Z",
        "updatedAt": "2022-01-22T13:10:59.442Z"
    },
    {
        "id": 891,
        "messageType": "FATAL",
        "operationType": "READ",
        "message": "A user as actioned the endpoint -> GetDomains and it completed successfully",
        "serviceName": "DOMAIN_MODEL_SERVICE",
        "timestamp": "1642621556268",
        "createdAt": "2022-01-22T13:10:59.442Z",
        "updatedAt": "2022-01-22T13:10:59.442Z"
    },
    {
        "id": 890,
        "messageType": "FATAL"
```

Figure 10 - Testing the /logs endpoint with the service name parameters.


Log Aggregation Service API / Returns the status of the service

GET {{baseUrl}}/health

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Status: 200 OK Time: 268 ms Size: 154 B

Figure 11 - Testing the /health endpoint.

3.2. Health Check Service

As already seen in the previous section, the monitoring subsystem is composed of two different services, *Health Checking* and *Log Aggregation*. These services are loosely coupled with a possible minor collaboration between them.

3.2.1. Service Specification

The Health Check Services API and its collaborations are described in the following table. It was used OpenAPI to specify all the diferente services.

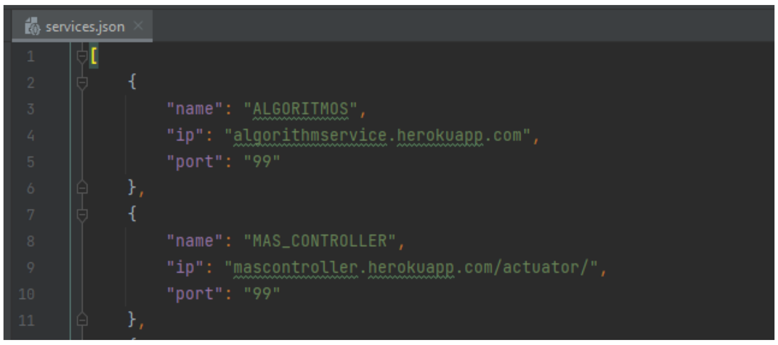
OpenAPI: https://cosn-health-checking.herokuapp.com/swagger-ui/index.html			
Operation	Method	Endpoint	Collaborations
getServicesStatus()	GET	/services	all webservice checkHealth()
updateServiceInfo()	PUT	/services	none
deleteService()	DELETE	/services	none
registerService()	POST	/registerService	none
getAllStatus()	GET	/logs	none
deleteStatus()	DELETE	/logs	none
getStatusByService()	POST	/logsByService	none
getStatusByService Timestamp()	POST	/logsByServiceTimestamp	none

Table 3 - The Health Check Service mapped to its operations and collaborations.

In these subsystems there are no specific collaborations besides the *checkHealth* endpoint that needs to be present in each one of the system's microservices; this endpoint must return the response code 200 to ascertain that the service is working in normal conditions. The description of all available services in the monitoring are stated below:

- **getServicesStatus()**
    - Service that return the current state of all services and their health logs.
- **updateServiceInfo()**
    - Service that updates the information on one of the registered services. This service receives the service ID, name, IP and Port. This service also returns if the service was updated successfully or not.
- **deleteService()**
    - Service that deletes one of the registered services. This service receives the service ID to be deleted. This service also returns if the register was done successfully or not.
- **registerService()**
    - Service that registers a service in order to be able to monitor it. This service receives the service name, the IP and the Port that the service is currently on. This service also returns if the register was done successfully or not.
- **getAllStatus()**
    - Service that gets all health logs from all the services.
- **deleteStatus()**
    - Service that deletes a status log. This service needs to be provided the ID from the log. The service also returns a message according to the success or not of the operation.
- **getStatusByService()**
    - Service that returns all the health logs from a certain provided service (the ID of the service needs to be provided).
- **getStatusByService Timestamp()**
    - Service that returns all the health logs from a certain provided service from an initial timestamp to an end timestamp (the ID of the service, initial timestamp and end timestamp need to be provided).

Alternatively to using the registerService(), it can also be used a configuration file in the service to load all information from others services. This way, it is not necessary for each service to register when starting up, as its information can be manually inserted in the configuration file from the Health Check Service.



services.json

```
[
    {
        "name": "ALGORITMOS",
        "ip": "algorithmservice.herokuapp.com",
        "port": "99"
    },
    {
        "name": "MAS_CONTROLLER",
        "ip": "mascontroller.herokuapp.com/actuator/",
        "port": "99"
    },
```

Figure 12 - Configuration file

3.2.2. Service Architecture

This service uses a layered architecture composed of four elements, the presentation layer, the business layer, the persistence layer and the database layer. The presentation layer is responsible for handling HTTP requests and converting the JSON data into an object and vice-versa. The business layer is responsible for handling business logic and rules and it consists of the service classes. The persistence layer is responsible for all of the storage logic, i.e. fetching objects and translating them into database rows. In the database layer, operations for creating, retrieving, updating and deleting (CRUD) are performed. The workflow of the service is in the figure below.

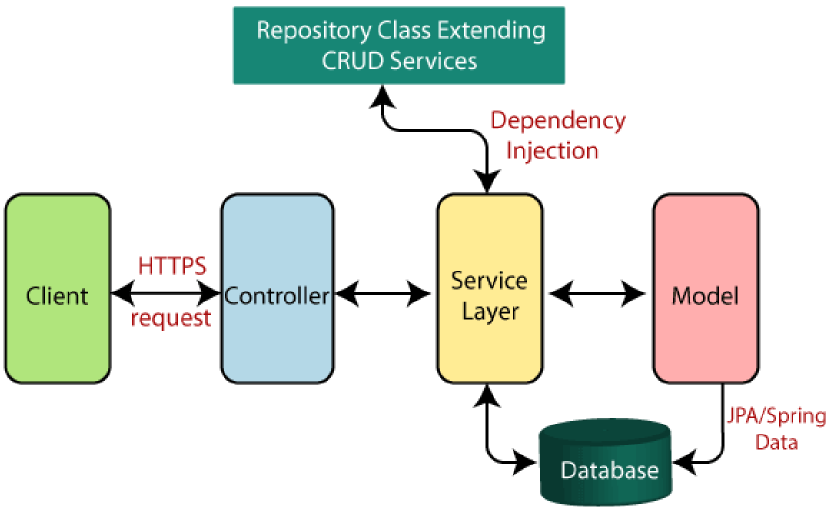


Figure 13 - Service workflow

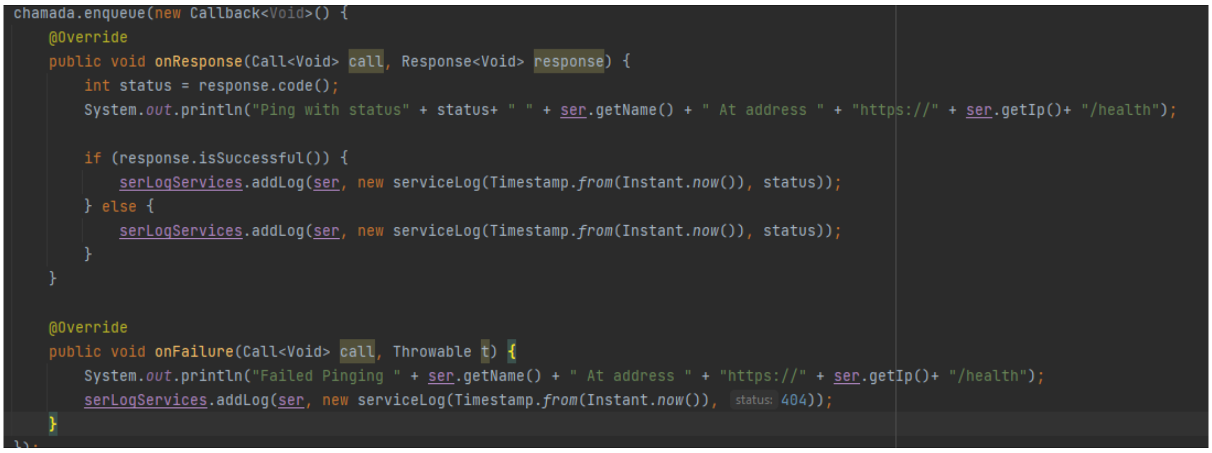
This service uses PostgreSQL as its data store, which is a relational database and its data model consists of two tables. The first table is the service that stores all the services information (ID, service name, service ip, service url and service logs). The other table stores all the information relative to the logs (ID, log timestamp and log status).

3.1.3. Implemented Patterns

This service itself implements an Observability pattern called the Health Check pattern. This pattern tackles the problem of how to detect if an instance of a service is being able or not to respond to requests. This service makes a request to all registered services every 30 seconds to the respective service endpoint /health.

This created another problem, if the Health Check Service did not receive a response, it would be left waiting forever. This was resolved by using the resilence pattern of timeout. In our implementation, we used the library Retrofit 2, which facilitates this implementation.

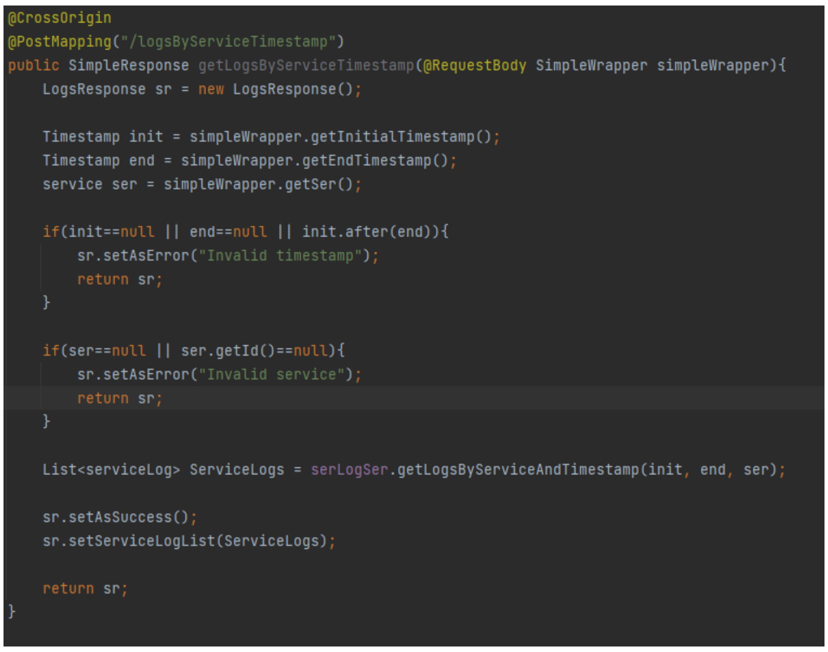
The timeout occurs after 10 seconds without any response. The implementation can be seen in the below image.



```java
chamada.enqueue(new Callback<Void>() {
    @Override
    public void onResponse(Call<Void> call, Response<Void> response) {
        int status = response.code();
        System.out.println("Ping with status" + status+ " " + ser.getName() + " At address " + "https://" + ser.getIp()+ "/health");

        if (response.isSuccessful()) {
            serLogServices.addLog(ser, new serviceLog(Timestamp.from(Instant.now()), status));
        } else {
            serLogServices.addLog(ser, new serviceLog(Timestamp.from(Instant.now()), status));
        }
    }

    @Override
    public void onFailure(Call<Void> call, Throwable t) {
        System.out.println("Failed Pinging " + ser.getName() + " At address " + "https://" + ser.getIp()+ "/health");
        serLogServices.addLog(ser, new serviceLog(Timestamp.from(Instant.now()), status: 404));
    }
```

Figure 14 - Retrofit implementation

In order to make the Health Check Service more responsive, it was also implemented the resilence pattern of fail fast. The fail-fast pattern states that if a service has to fail, it should do it in the fastest way possible. For each endpoint, all the verifications are made to make the endpoint return an error, if at all, in the fastest way possible. An example can be seen in the below image.

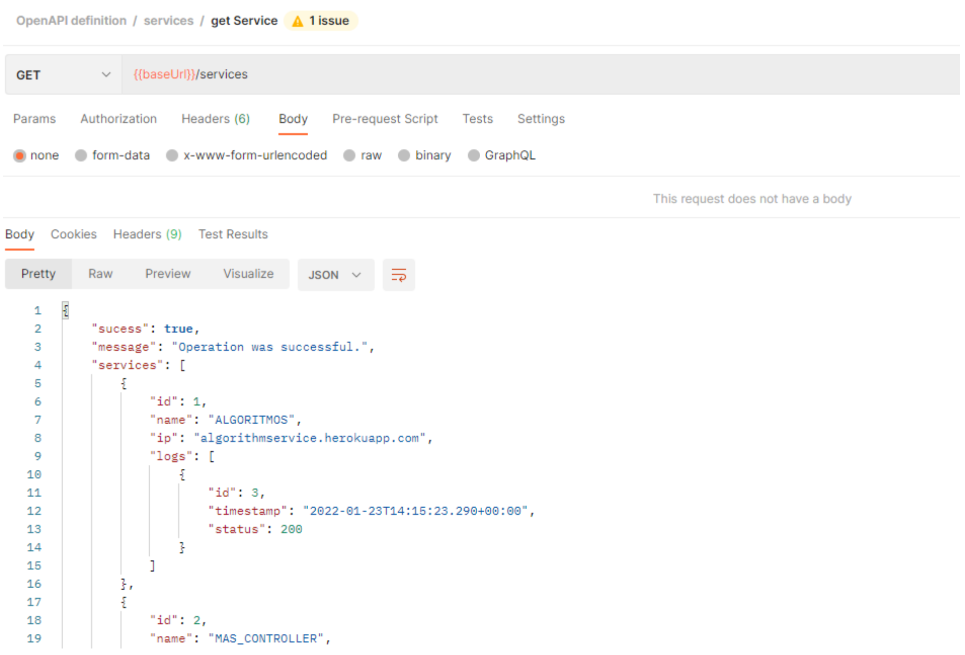


```java
@CrossOrigin
@PostMapping("/logsByServiceTimestamp")
public SimpleResponse getLogsByServiceTimestamp(@RequestBody SimpleWrapper simpleWrapper){
    LogsResponse sr = new LogsResponse();

    Timestamp init = simpleWrapper.getInitialTimestamp();
    Timestamp end = simpleWrapper.getEndTimestamp();
    service ser = simpleWrapper.getSer();

    if(init==null || end==null || init.after(end)){
        sr.setAsError("Invalid timestamp");
        return sr;
    }

    if(ser==null || ser.getId()==null){
        sr.setAsError("Invalid service");
        return sr;
    }

    List<serviceLog> ServiceLogs = serLogSer.getLogsByServiceAndTimestamp(init, end, ser);

    sr.setAsSuccess();
    sr.setServiceLogList(ServiceLogs);

    return sr;
}
```

Figure 15 - Fail fast pattern example

3.1.4. Tests

All the endpoid were tested using Postman. Some examples of the conducted tests can be seen in the below figures.



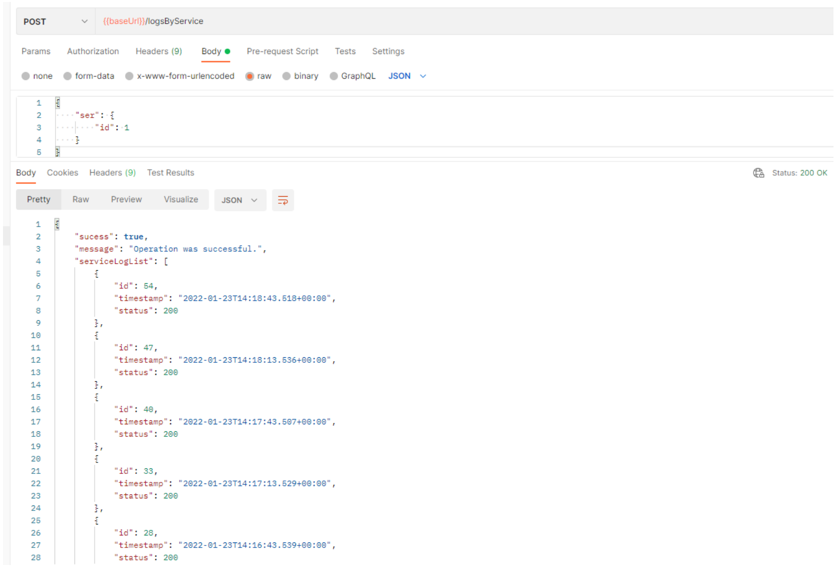
```
GET {{baseUrl}}/services
```

[● none] form-data x-www-form-urlencoded raw binary GraphQL

This request does not have a body

```
{
    "sucess": true,
    "message": "Operation was successful.",
    "services": [
        {
            "id": 1,
            "name": "ALGORITMOS",
            "ip": "algorithmservice.herokuapp.com",
            "logs": [
                {
                    "id": 3,
                    "timestamp": "2022-01-23T14:15:23.290+00:00",
                    "status": 200
                }
            ]
        },
        {
            "id": 2,
            "name": "MAS_CONTROLLER",
```

Figure 16 - Testing the /services endpoint (getServicesStatus()).



```
POST {{baseUrl}}/logsByService
```

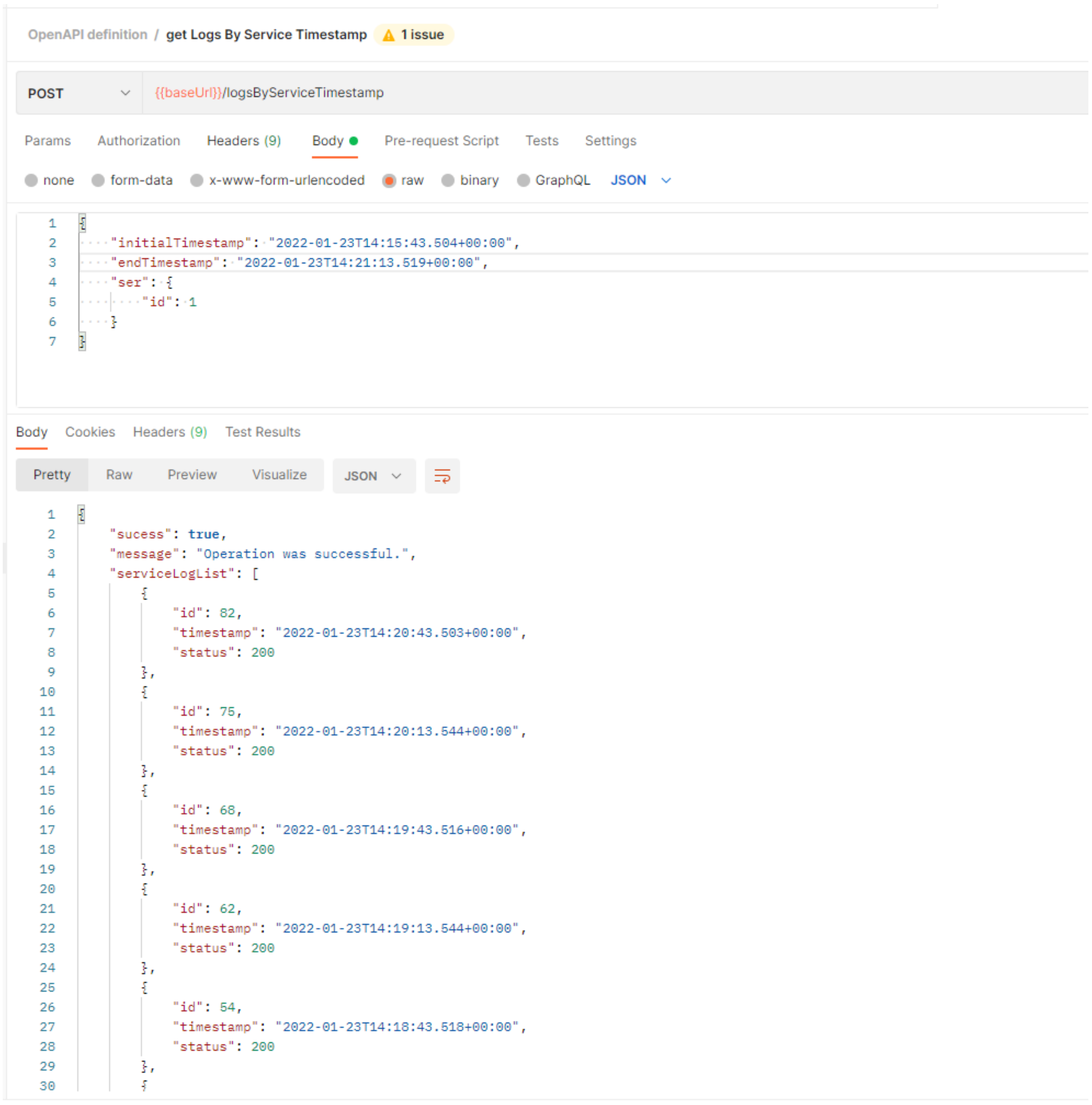
none form-data x-www-form-urlencoded [● raw] binary GraphQL JSON

```
{
    "ser": {
        "id": 1
    }
}
```

Status: 200 OK

```
{
    "sucess": true,
    "message": "Operation was successful.",
    "serviceLogList": [
        {
            "id": 54,
            "timestamp": "2022-01-23T14:18:43.518+00:00",
            "status": 200
        },
        {
            "id": 47,
            "timestamp": "2022-01-23T14:18:13.536+00:00",
            "status": 200
        },
        {
            "id": 40,
            "timestamp": "2022-01-23T14:17:43.507+00:00",
            "status": 200
        },
        {
            "id": 33,
            "timestamp": "2022-01-23T14:17:13.529+00:00",
            "status": 200
        },
        {
            "id": 28,
            "timestamp": "2022-01-23T14:16:43.539+00:00",
            "status": 200
```

Figure 17 - Testing the /logsByService endpoint with and ID=1.



POST {{baseUrl}}/logsByServiceTimestamp

```
{
    "initialTimestamp": "2022-01-23T14:15:43.504+00:00",
    "endTimestamp": "2022-01-23T14:21:13.519+00:00",
    "ser": {
        "id": 1
    }
}
```

```
{
    "sucess": true,
    "message": "Operation was successful.",
    "serviceLogList": [
        {
            "id": 82,
            "timestamp": "2022-01-23T14:20:43.503+00:00",
            "status": 200
        },
        {
            "id": 75,
            "timestamp": "2022-01-23T14:20:13.544+00:00",
            "status": 200
        },
        {
            "id": 68,
            "timestamp": "2022-01-23T14:19:43.516+00:00",
            "status": 200
        },
        {
            "id": 62,
            "timestamp": "2022-01-23T14:19:13.544+00:00",
            "status": 200
        },
        {
            "id": 54,
            "timestamp": "2022-01-23T14:18:43.518+00:00",
            "status": 200
        },
        {
```

Figure 18 - Testing the /logsByServiceTimestamp endpoint with and ID=1, initialTimestamp = “2022-01-23T14:15:43.504+00:00” and endTimestamp = “2022-01-23T14:21:13.519+00:00”.

4. Conclusions

In this project it was possible to successfully implement a microservices oriented application. Our group implemented three different services considering the fact of having three main and different responsibilities, the routing of requests (Gateway), the log aggregation (LogAgregationService) and the monitoring (MonitoringService).

It was also possible to connect and integrate our services with other services from other groups. For that integration it was used OpenAPI specification, for each group to know the endpoints that we had.

We also used different observability and resilience patterns according to our needs and also implemented security aspects in the system.

5. References

About Swagger Specification | Documentation. (n.d.). Swagger. Retrieved December 17, 2021, from https://swagger.io/docs/specification/about/

API gateway pattern. (n.d.). Microservices.io. Retrieved December 17, 2021, from https://microservices.io/patterns/apigateway.html

Calçado, P., & Shah, B. (2020, July 4). *Microservices Design - API Gateway Pattern | by Bibek Shah*. Dev Genius. Retrieved December 17, 2021, from https://blog.devgenius.io/microservices-design-api-gateway-pattern-980e8d02bdd5

Home - OpenAPI Documentation. (n.d.). GitHub Pages. Retrieved December 17, 2021, from https://oai.github.io/Documentation/

Representational state transfer. (n.d.). Wikipedia. Retrieved December 17, 2021, from https://en.wikipedia.org/wiki/Representational_state_transfer

Richardson, C. (2018). *Microservices Patterns: With Examples in Java*. Manning.

Wang, X. (2021, March 6). *How to choose the right API Gateway for your platform: Comparison of Kong, Tyk, Apigee, and alternatives*. Moesif. Retrieved December 17, 2021, from https://www.moesif.com/blog/technical/api-gateways/How-to-Choose-The-Right-API-Gateway-For-Your-Platform-Comparison-Of-Kong-Tyk-Apigee-And-Alternatives/

What is REST — A Simple Explanation for Beginners, Part 1: Introduction. (2017, September 5). Medium. Retrieved December 17, 2021, from https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-1-introduction-b4a072f8740f

Spring Boot Architecture - javatpoint. (n.d.). Javatpoint. Retrieved January 23, 2022, from https://www.javatpoint.com/spring-boot-architecture