
Università di Catania

Department of Maths and Computer Science
University of Catania

Ingegneria dei Sistemi Distribuiti e Laboratorio
Exam Project

---

# NetWatch • Utilities Health Monitor

---

Author:
Antonio Scardace

Submitted to:
Prof. E. Tramontana
Prof. A. Fornaia

Jan-Feb 2024

# Contents

# 1 Abstract

The project aims to be a health checker. It proposes to check in real time the health of each observed utility. The utility can be a Docker Container, a Machine, or a Network Device in your local network. If a utility changes its state (i.e. goes offline or comes back online), the monitor notifies the utility referent by Email, Slack Message, or Telegram message in a public channel.

In this report, I present an old version of this project and a new version of it. The first one was designed as a monolith with fewer features; the latter one, on the contrary, got a full re-design and has been re-implemented.

# 2 The Original and Old Version

The original version of the project had the goals described above. It has been designed to be a health checker and to alert the respective referents. Unfortunately, it has a lot of design and implementation problems including:

- It is a **monolith**, and we know that monolithic applications can be challenging to scale horizontally. Scaling the entire application involves replicating the entire codebase, even if only a specific module requires additional resources. Also, a failure in one module can potentially impact the whole application. There is limited fault isolation, making it harder to contain and manage failures.
- There are no **authentication** and **authorization**, so that everyone can interact with the database and run a query. This can lead to data breaches, data manipulation, or unauthorized retrieval of sensitive information. Many regulatory standards (e.g. GDPR) require organizations to implement robust authentication and authorization measures to protect sensitive data. Non-compliance with these standards can result in legal consequences.
- There is neither an **API gateway** nor a **frontend** for the user to interact with. The only way to insert, update, delete, or read data is by SQL query on the database.
- It was chosen not to use Object-Relational Mapping (**ORM**) and to manually write a MySQL class, which is very time-consuming and makes the code less readable. On the contrary, ORM frameworks provide a level of abstraction that promotes database portability, allowing easier migration between different DBMSs. Also, ORM frameworks typically handle parameterization automatically, reducing the risk of security vulnerabilities.
- **Docker** is not incorporated in the development strategy, so, differences in operating systems, dependencies, and configurations can lead to "it works on my machine" issues. Moreover, scaling applications without containerization can be challenging.
- The **Logging** is not included.

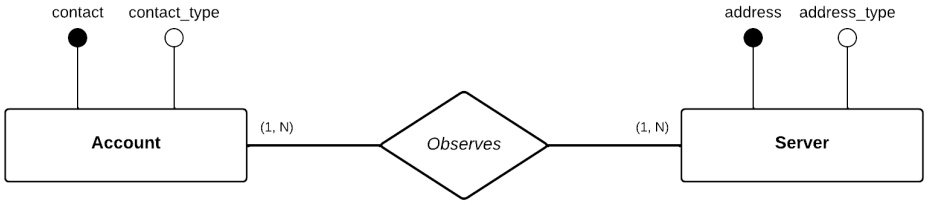


Figure 1: The original project database E-R model.

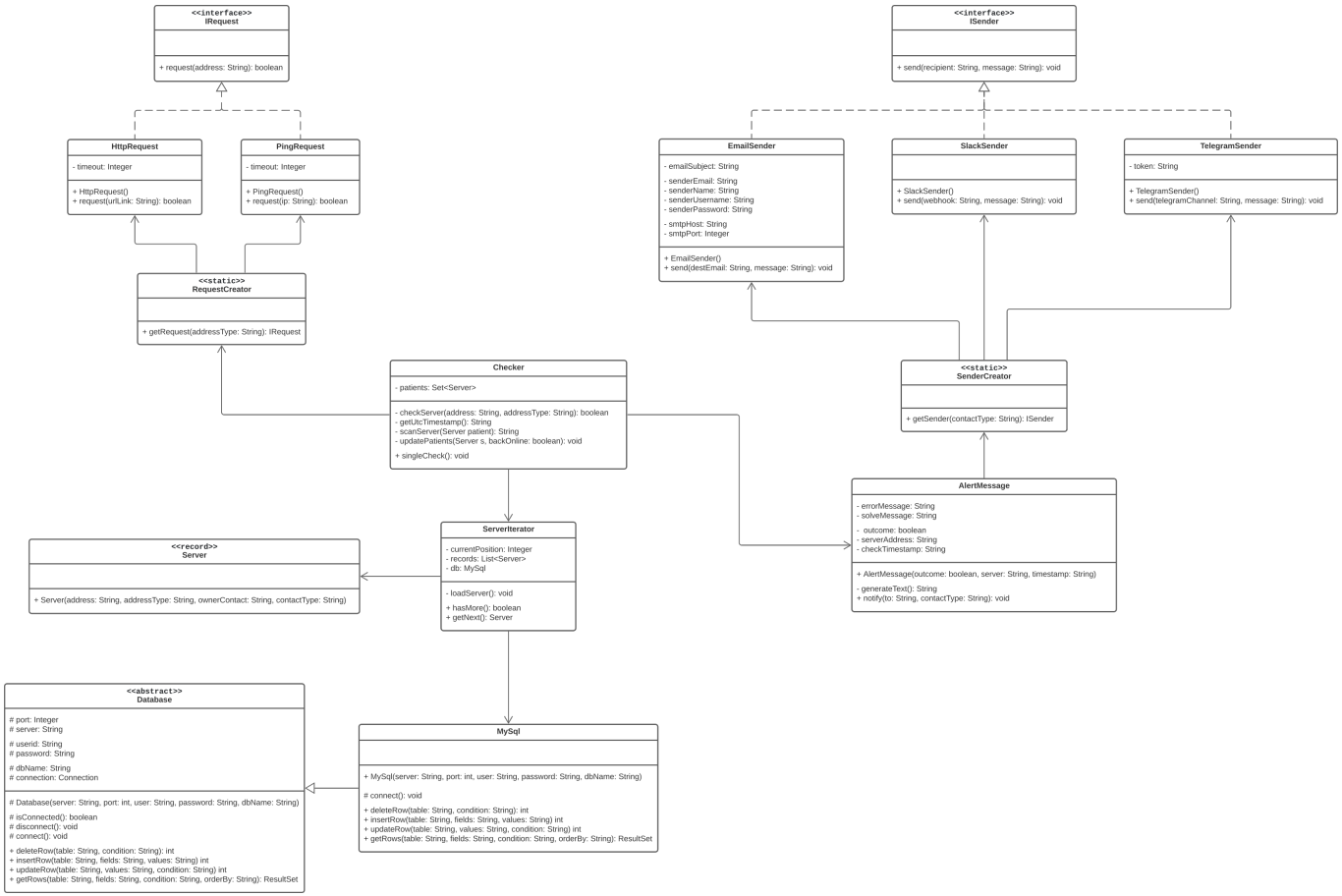


Figure 2: The original monolith project UML.

# 3 New Architecture

The new version of the project has the same goals and outcomes as its older version but has made significant improvements over the design and implementation of its older version. Specifically, these enhancements contribute to a more maintainable, scalable, and secure system:

- The project has been divided into **microservices**, addressing concerns related to horizontal scalability, code modularity, and team separation of concerns. This architecture allows for more flexible and scalable development and deployment.
- An **authentication** and **authorization** mechanism has been implemented. This ensures that users can only interact with the data for which they are authorized, enhancing security and data integrity.
- Users are restricted to interacting only with the **frontend** and the **API Gateway**. This limitation can enhance security and control over user access, preventing unauthorized access to backend components and data.
- Each component has been containerized using **Docker**, providing benefits in terms of portability and scalability. Docker containers encapsulate dependencies and streamline deployment across different environments. The entire project is contained within a **Docker Network**, improving isolation and simplifying management.
- The new version introduces event **logging**.

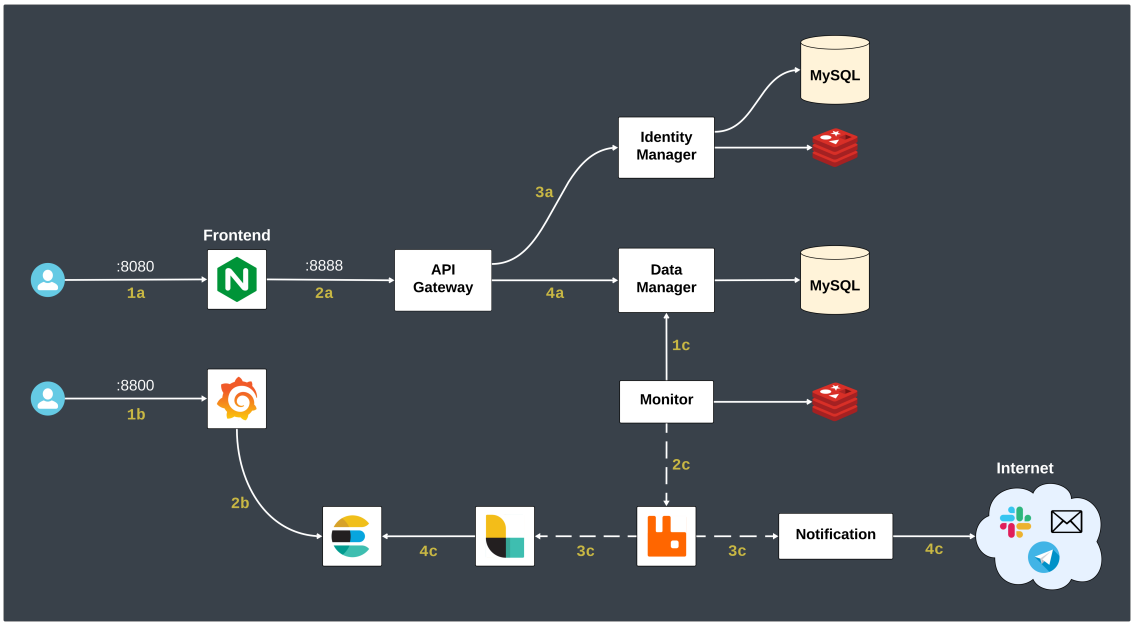


Figure 3: The architecture of the new version of the project.

## 3.1 Data Manager

This logical component is tasked with the management of data related to the contacts of referents, utilities, and their observations. Each referent can observe N (where $N \geq 0$) utilities that pertain to their environment. The microservice exposes **REST APIs**, enabling users to perform operations such as inserting, updating, deleting, and retrieving data.

The microservice is implemented in **Java**, leveraging the **Spring Boot** framework, while data are stored in a **MySQL** database. Spring Boot utilizes the **Hibernate** ORM for seamless communication with the database. Access to the microservice APIs is restricted, and users can only interact with them through the API Gateway [Chapter 3.2] after undergoing authentication and authorization processes.

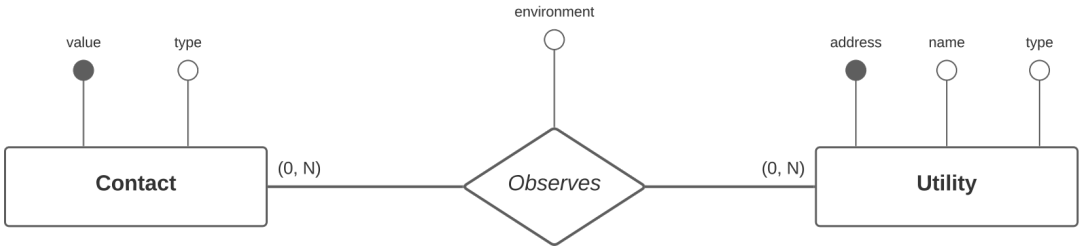


Figure 4: The E-R model of the database of the component.

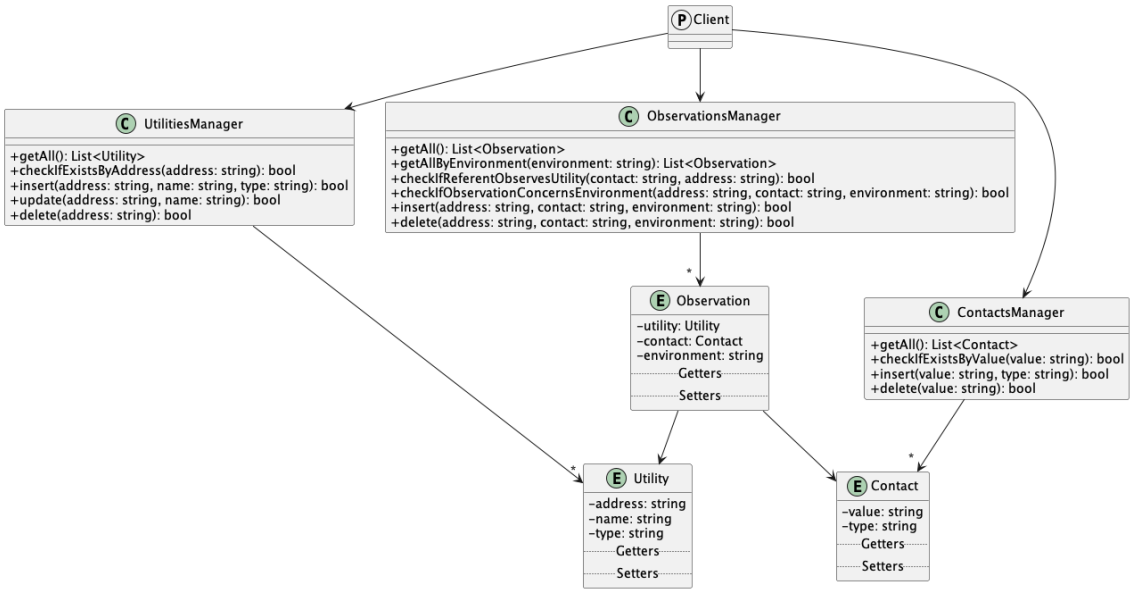


Figure 5: The UML of the data manager microservice.

## 3.2 Identity Manager

This logical component handles the management of data related to authentication and authorization processes. Each user is assigned a specific role, and each role can have N (where $N \geq 0$) permissions for data access. The microservice exposes **REST APIs**, enabling administrators to perform operations such as inserting, updating, deleting, and retrieving data. Non-admin users are limited to accessing only the sign-in and sign-up APIs.

The microservice is implemented in **Java** using the **Spring Boot** framework, and data is stored in a **MySQL** database. It utilizes the Hibernate **ORM** for seamless communication with the database.

Access to its APIs is restricted, allowing users to interact with them solely through the API Gateway [Chapter 3.2] after completing authentication and authorization processes (except for sign-in and sign-up endpoints). The chosen authentication strategy involves using a 32-character long **API Key**. On the user sign-in, a token valid for a short duration (4 days) is generated, and its **UUID** is stored and returned to the user. To maintain authentication and authorization during each HTTP request, excluding those to sign-in and sign-up endpoints, users are required to include it (X-API-KEY: XXXXX) in the **Header section**.

Regarding token storage, two options are considered: using a local data structure (e.g. HashMap) or utilizing a **Redis Cache**. The latter is chosen due to several advantages, including scalability, concurrent access support, and built-in token expiration features.

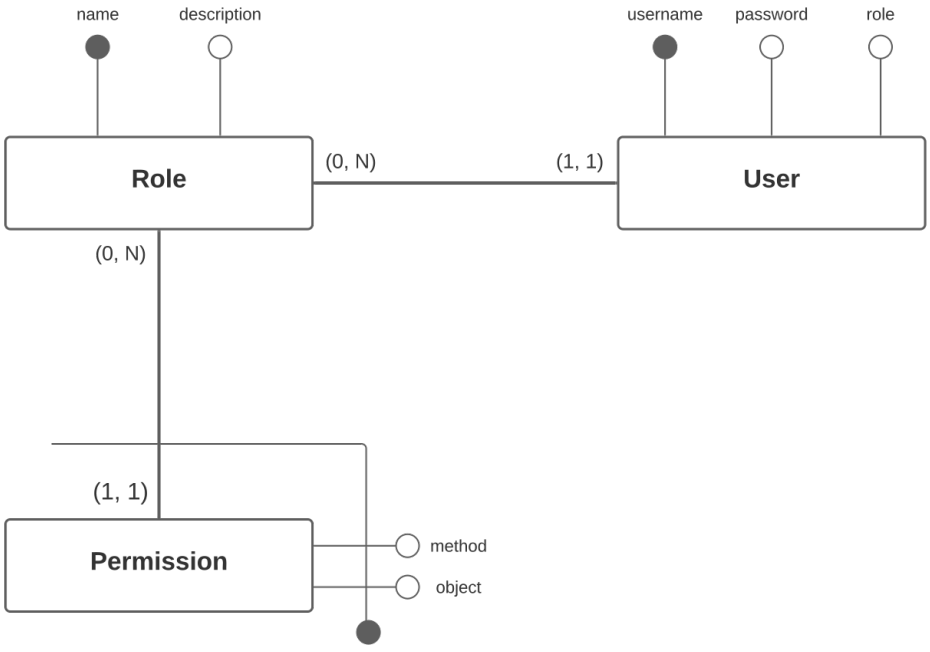


Figure 6: The E-R model of the database of the component.

This microservice is based on the **Authenticator Design Pattern** for the authentication service and implements the **Role-Based Access Control Design Pattern** for the authorization service.

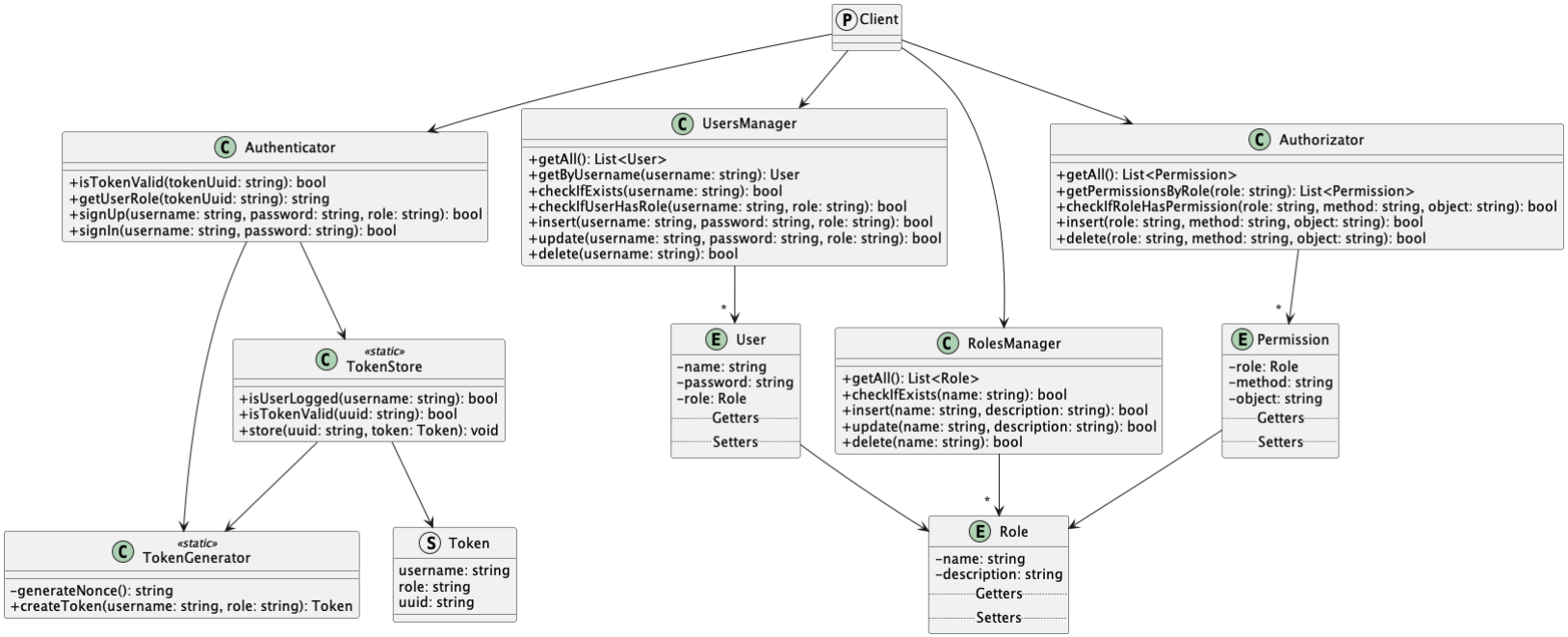


Figure 7: The UML of the identity manager microservice.

## 3.3 API Gateway

This microservice is responsible for managing all incoming HTTP requests **mapping** them to APIs exposed by internal microservices, and handling user authentication and authorization. Additionally, for security purposes, it incorporates a **Rate Limiter**.

This microservice is built using **Java** with the **Spring Boot** framework. The API Gateway is managed with the **Netflix Zuul** service, which facilitates the mapping of external APIs to internal APIs and applies two distinct **filters** to incoming requests: one is for logging, while the other is dedicated to authentication and authorization, leveraging the API Key mechanism. Additionally, it includes rate limiting using the **Guava** library to prevent abuse of the service.

### 3.3.1 APIs accessible by anyone

**POST** /authentication/signup

Registers a new user

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| user | String | Yes | Username |
| psw | String | Yes | Password |
| role | String | Yes | User's Role |

**POST** /authentication/signin

Checks if the credentials are valid and returns the token UUID

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| user | String | Yes | Username |
| psw | String | Yes | Password |

### 3.3.2 APIs accessible only by admins

**GET** /users/

Gets all registered users

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| - | - | - | - |

**GET** /users/check

Checks if a user is registered by their username

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| user | String | Yes | Username |

**PUT** /users/

Updates a specific user's fields

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| user | String | Yes | Username |
| psw | String | Yes | New Password |
| role | String | Yes | Name of the New Role |

**DELETE** /users/

Deletes a user by their username

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| user | String | Yes | Username |

**GET** /roles/

Gets all registered roles

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| - | - | - | - |

**POST** /roles/

Inserts a new role

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| name | String | Yes | Role Name |
| descr | String | Yes | Role Description |

**PUT** /roles/

Updates a specific role description

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| name | String | Yes | Role Name |
| descr | String | Yes | New Role Description |

**DELETE** /roles/

Deletes a role by its name

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| name | String | Yes | Role Name |

**GET** /authorization/

Gets all permissions

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| - | - | - | - |

**GET** /authorization/filter

Gets all permissions associated to a role

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| role | String | Yes | Role Name |

**GET** /authorization/check

Checks if a role has a specific permission

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| role | String | Yes | Role Name |
| method | String | Yes | Authorized Method for the object |
| object | String | Yes | Authorized Object |

**POST** /authorization/

Inserts a new permission

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| role | String | Yes | Role Name |
| method | String | Yes | Authorized Method for the object |
| object | String | Yes | Authorized Object |

**DELETE** /authorization/

Deletes a permission

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| role | String | Yes | Role Name |
| method | String | Yes | Authorized Method for the object |
| object | String | Yes | Authorized Object |

**GET** /contacts/

Gets all contacts

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| - | - | - | - |

**GET** /contacts/check

Checks if a contact is registered by its value

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| value | String | Yes | Contact Value |

**POST** /contacts/

Inserts a new contact

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| value | String | Yes | Contact Value |
| type | String | Yes | Contact Type |

**DELETE** /contacts/

Deletes a contact

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| value | String | Yes | Contact Value |

**GET** /utilities/

Gets all utilities

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| - | - | - | - |

**GET** /utilities/check

Checks if a utility is registered by its address

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| address | String | Yes | Utility Address |

**POST** /utilities/

Inserts a new utility

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| address | String | Yes | Utility Address |
| name | String | Yes | Utility Name |
| type | String | Yes | Utility Address Type |

**PUT** /utilities/

Updates a specific utility name

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| address | String | Yes | Utility Address |
| name | String | Yes | New Utility Name |

**DELETE** /utilities/

Deletes a utility

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| address | String | Yes | Utility Address |

**GET** /observations/

Gets all associations

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| - | - | - | - |

**GET** /observations/check

Checks if an association is registered

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| address | String | Yes | Utility Address |
| contact | String | Yes | Referent Contact Value |

### 3.3.3 APIs accessible exclusively by users with a designated role

**GET** /observations/filter/**:env**

Gets all associations of your environment (role)

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| - | - | - | - |

**POST** /observations/**:env**

Inserts a new association in your environment (role)

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| address | String | Yes | Utility Address |
| contact | String | Yes | Referent Contact Value |

**DELETE** /observations/**:env**

Deletes an association if present in your environment (role)

| Parameter | Type | Is Required | Description |
|---|---|---|---|
| address | String | Yes | Utility Address |
| contact | String | Yes | Referent Contact Value |

## 3.4 Monitor

This logical component is responsible for utility monitoring. The microservice does not expose any APIs and is not accessible to users. It executes a **cron job** periodically, specifically every second. This job involves fetching all registered observations from the Data Manager microservice via API using Data Transfer Objects (**DTO**s), and **asynchronously** checking their health. During this health check, four scenarios may occur:

1. If the utility remains online, the system remains unchanged, and no action is taken.
2. If the utility remains offline, the system remains unchanged, and no action is taken.
3. If the utility transitions from offline to online since the last check, a resolution notification message is prepared and sent to a **RabbitMQ Exchange**.
4. If the utility transitions from online to offline since the last check, an error notification message is prepared and sent to a **RabbitMQ Exchange**.

The microservice is implemented in **Java**, leveraging the **Spring Boot** framework, while data about offline utilities are cached in a **Redis** database. Additionally, the HTTP connection between this microservice and the Data Manager microservice is a **persistent** connection enabled by the **Keep-Alive** flag, with each connection remaining alive for 20 seconds.

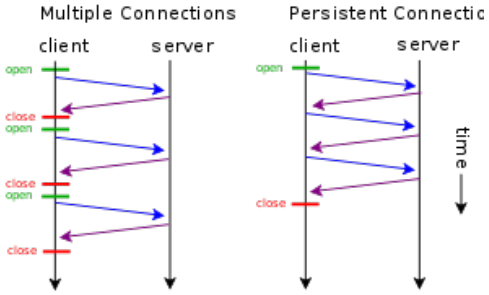


Figure 8: Visual explanation of Keep-Alive advantage.

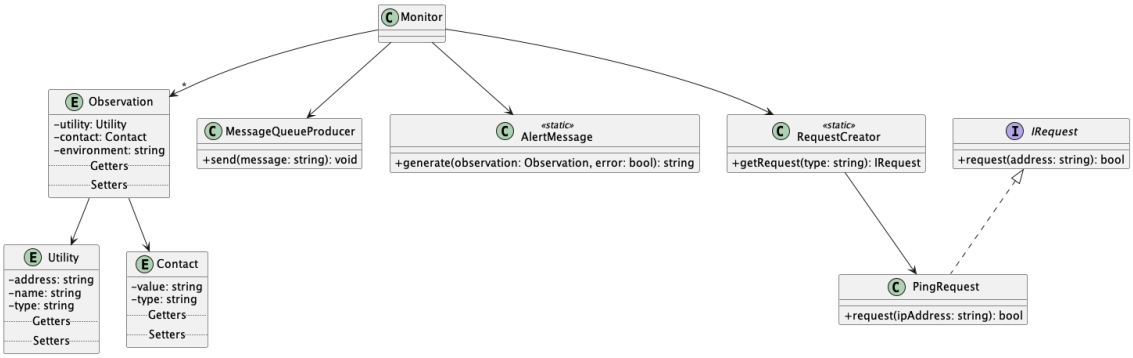


Figure 9: The UML of the monitor microservice.

This microservice implements the **Factory-Method Design Pattern** for the request model. This design pattern enables easy addition of new methods to check a utility if needed in the future.

## 3.5 Notification

This microservice stands out as it operates independently from the project's context, making it highly reusable across various applications. This flexibility is made possible by its reliance on a **message-driven architecture**. By decoupling the notification functionality from project-specific details, this microservice can be seamlessly integrated into diverse environments, and that's why messages are processed within the Monitor microservice.

The microservice is implemented in **Java**, leveraging the **Spring Boot** framework. It implements the **Factory-Method Design Pattern** for the sender model, allowing for easy addition of new notification methods in the future.

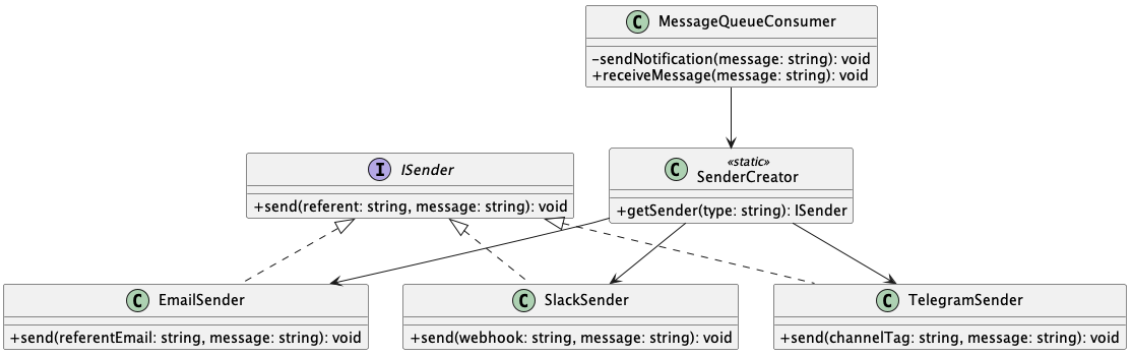


Figure 10: The UML of the notification microservice.

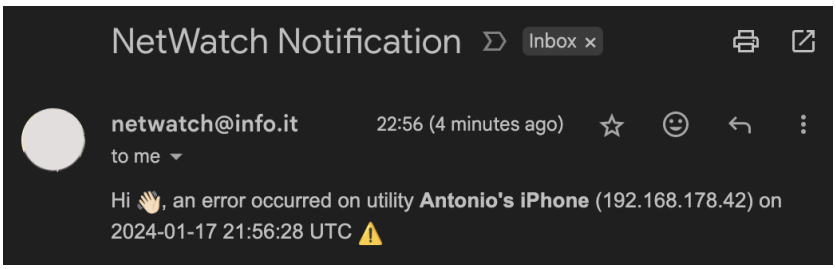


Figure 11: Example of Email alert.

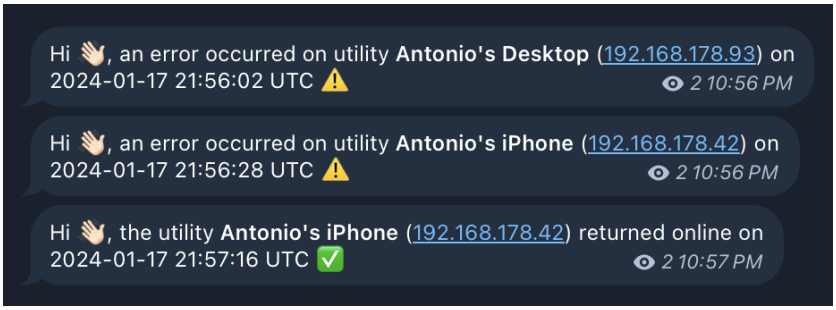


Figure 12: Example of Telegram alerts.

## 3.6 Logging Stack

I have a logging stack consisting of **Logstash**, **ElasticSearch**, and **Grafana**. I only log messages generated by the Monitor microservice when a utility changes its status, and anything else. Logstash receives these logs by reading messages from the **RabbitMQ Exchange**, where the Monitor microservice sends notifications, similar to the Notification microservice. With Grafana, I create visualizations based on these logs.

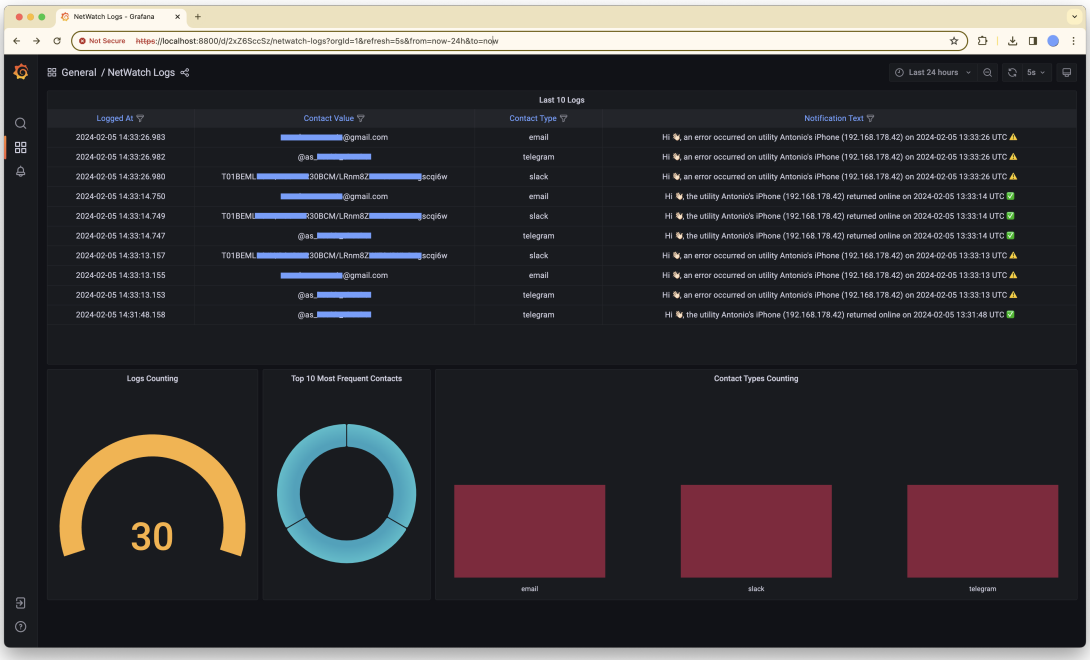


Figure 13: Demo of Grafana.

# 4 Sequence Flows

As depicted in Figure 15, the service involves various interaction flows: three are user-dependent, while one represents continuous internal workflow.

## 4.1 User Flow

The user can have three different flows of interactions:

1. The user interacts with the frontend (**1a**) which provides a set of operations. These involve sending and retrieving data from the backend just via the API Gateway (**1b**). Except for sign-up and sign-in requests, when a request is made, the API Gateway first authenticates and authorizes the request using the API Key (**2a**). If the user is authenticated and authorized to proceed, the request is then redirected to the respective endpoint in the backend (**3a** or **3b**, depending on the nature of the request).

2. The user interacts directly with the API Gateway (**2a**) sending HTTP requests. Except for sign-up and sign-in requests, when a request is made, the API Gateway first authenticates and authorizes the request using the API Key (**2a**). If the user is authenticated and authorized to proceed, the request is then redirected to the respective endpoint in the backend (**3a** or **4a**, depending on the nature of the request).

3. The user interacts directly with the Grafana dashboard (**1b**) to view logging charts.

## 4.2 Internal Flow

Periodically (i.e. every second), the Monitor microservice requests (**1c**) the list of observations from the Data Manager via a persistent HTTP connection (using the keep-alive flag). Asynchronously, it checks the health of each utility. If necessary, it creates an alert message and sends it to the RabbitMQ Exchange (**2c**), which implements the **Publish-Subscribe Design Pattern**.
Each consumer has its queue connected to the RabbitMQ Exchange, and every time a message arrives at it (**2c**), each connected consumer receives the messages (**3c**). Specifically, here we have two consumers: the Notification microservice and Logstash. After consuming a message, the Notification microservice forwards it to the referent contact (**4c**), while Logstash sends it to Elasticsearch **4c**, from which data will be extracted by Grafana (**2b**) upon user request.

# 5 Conclusions

As I conclude this report, I would like to express my gratitude for the invaluable opportunity to delve into the intricacies of software development. Throughout this journey, I have had the privilege to learn and implement various concepts, including Spring Boot, Hibernate, and Redis. These experiences have not only expanded my technical skills but have also provided me with a deeper understanding of modern software development practices. I am particularly pleased with the opportunity to apply the theoretical knowledge acquired during lectures to real-world scenarios.

## 5.1 Future Works

To further enhance the project, I consider implementing the following improvements:

- Adding new supported **platforms** for notifications receiving.
- Adding **Jaeger** for logging service performance.
- Incorporating a **Circuit Breaker** in the API Gateway.
- Implementing the **Frontend**.

## 5.2 Benchmarks

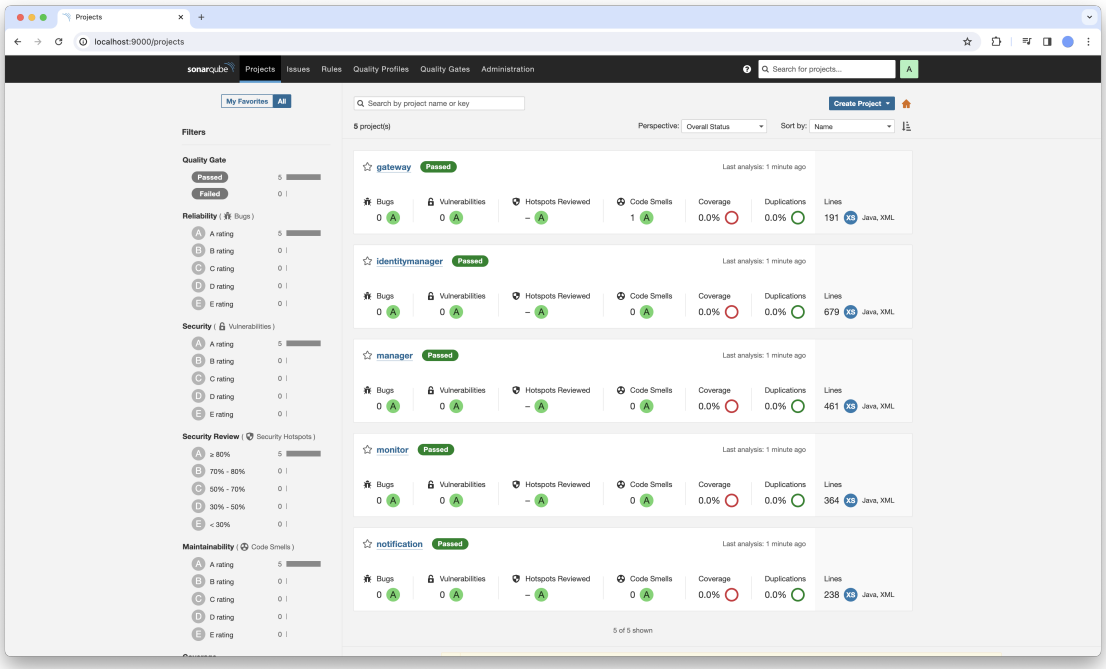


Figure 14: SonarQube benchmark.

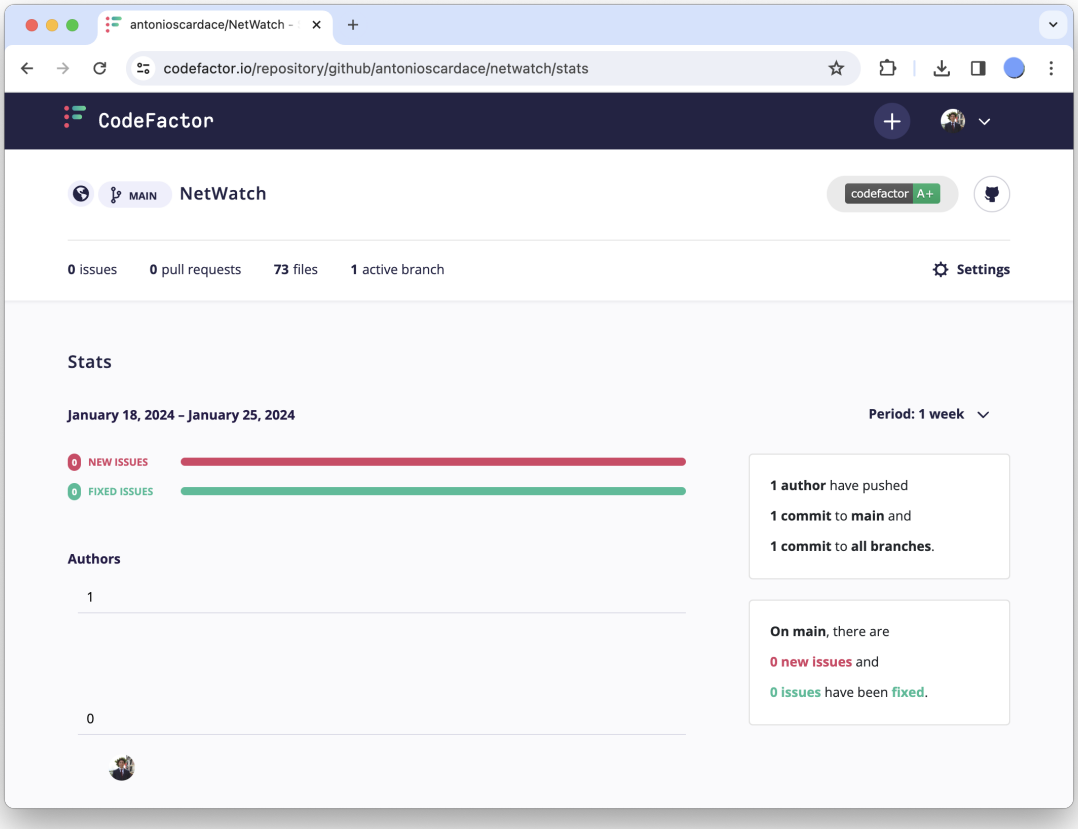


Figure 15: CodeFactor benchmark.