LABORATORY PROJECT

CLOUD COMPUTING

IMPLEMENTATION OF A FAAS

AUTHORS:

MANUEL SARMIENTO TENDERO
KEVIN OMAR PROCEL SALINAS
ISABELL KRISTIN HELLING
CATALIN MARIAN COCIU

PROFESSOR:
JOSE MANUEL BERNABEU AUBÁN

COURSE: 2024/2025

Cloud Computing: FAAS

1. Introduction

The purpose of this project is to design and implement a Function as a Service (FaaS) system, enabling users to register and execute functions on demand through a RESTful API. The system ensures secure access via user registration and authentication while leveraging modern technologies such as APISIX for reverse proxy, NATS for message queuing and key-value storage, and containerized workers for function execution. The implementation includes a microservices architecture, a Docker-based deployment, and a complete documentation of the system's functionality and configuration.

2. Microservices architecture description

The microservices architecture of our FaaS consists of several interconnected nodes that ensure the service is efficient and scalable. Among these, we have NATS, which serves as the messaging system; MongoDB, the database; APISIX, the application interface; and etcd, used for storing key-value configurations. Each of these components plays a key role in the system's operation, contributing to its robustness, flexibility, and ease of maintenance.

○ 2.1. Microservices

NATS

NATS is a high-performance, lightweight, and low-latency messaging system that enables communication between different components in a distributed architecture. It allows microservices to exchange messages without requiring a direct connection, acting as a messaging intermediary and facilitating asynchronous communication.

In this FaaS, it is used because of its simple and lightweight API and its ability to scale horizontally. NATS is the microservice that enables communication between the worker and the API server.

MongoDB

MongoDB is a highly scalable and flexible NoSQL database used in our architecture to efficiently store information. Unlike traditional relational databases that use tables and rows, MongoDB organizes data in BSON (Binary JSON) documents, allowing it to handle semi-structured or unstructured data with ease.

This database was chosen because it also supports horizontal scaling, and its BSON document structure enables efficient and flexible data storage and querying.

In our architecture, MongoDB plays an important role by storing critical system information, such as user data and the functions associated with each user.

API-SERVER

The API server is the central component that exposes the application programming interfaces (APIs) of our system to clients. Its primary function is to manage external requests and coordinate business logic by interacting with other microservices, such as the MongoDB database and the NATS messaging system.

Key features of the API server:

- **Business logic access interface**: The API server provides endpoints that allow external clients to interact with the FaaS. These endpoints support operations like creating users or functions, retrieving data, or executing functions.
- **Communication with other microservices**: The API server interacts with essential components of the architecture, such as MongoDB for accessing and storing data, NATS for publishing or receiving messages, and APISIX for managing and directing traffic to the appropriate APIs. This enables the API server to coordinate and consolidate business logic and request distribution across the architecture.
- **Scalability and flexibility**: As part of a microservices architecture, the API server is designed to scale independently. This means multiple instances of the API server can run simultaneously, handling different requests efficiently.

In summary, the API server acts as the entry point for requests coming from users and coordinates interactions between the various microservices in the system to provide the requested functionalities.

APISIX

APISIX is a high-performance API Gateway that manages the routing of requests to the system's internal microservices. Its main role is to receive incoming client requests and direct them to the appropriate microservices, facilitating traffic management, security, and API optimization.

APISIX was chosen for its following features:

- **Request routing**: It directs incoming requests to the appropriate microservices using reverse proxy, with support for dynamic routing and load balancing.
- **Traffic management**: APISIX allows traffic control through features like rate limiting, circuit breaking, and load balancing.
- **Security**: It provides authentication and authorization (API keys, JWT, OAuth), SSL/TLS encryption, and protection against attacks (e.g., SQL Injection, XSS, DoS).
- **Extensibility**: It can be customized through plugins, adding functionalities such as logging, monitoring, caching, and more.
- **Distributed configuration**: APISIX uses etcd to manage centralized, distributed configuration, ensuring consistency across all nodes.

In the FaaS, APISIX serves as the single entry point for all external requests. When a client sends a request, APISIX receives it and, based on configuration rules, decides which microservice or set of microservices should handle it. Additionally, APISIX ensures security, traffic control, and monitoring, efficiently managing traffic to the backend microservices. Security is enforced using JWT tokens. In this distributed system, APISIX is configured to work with etcd and has specific ports open to handle HTTP and HTTPS traffic, as well as API traffic.

etcd

etcd is a distributed, highly available key-value storage system primarily used to manage configurations and maintain the state of distributed applications. In the FaaS, etcd plays a fundamental role in the distributed configuration management of the system's services.

etcd was chosen for its following advantages:

- **Key-value storage**: It uses a key-value storage format to store configurations and essential metadata for distributed applications.
- **Strong consistency**: It ensures all nodes share a consistent view of the system state using the Raft consensus algorithm.
- **High availability**: It replicates data across multiple nodes, ensuring data is always accessible, even if some nodes fail.
- **RESTful API**: etcd exposes a RESTful API that allows other services to interact with the stored data easily.

In this project, etcd is primarily used for APISIX's distributed configuration. This API Gateway requires a reliable place to store its configuration, such as API routes, security policies, authentication, and other metadata. etcd stores and distributes APISIX's configuration centrally and synchronously across all nodes in the infrastructure. When APISIX's configuration is modified, these changes are reflected in etcd, and all APISIX nodes querying etcd automatically receive the updates without manual intervention.

etcd is essential in this microservices architecture as it provides consistency, high availability, and synchronization among all services. It also enables greater flexibility and scalability.

WORKER

Finally, we have the **worker**, which is the microservice responsible for executing the assigned functions. This component locates the containers containing the function to be performed, executes the corresponding operation, and then returns the result as a **string**.

This microservice communicates with the API server using **NATS** as the messaging system, enabling asynchronous and efficient interaction between the two.

○ 2.2. Their degree of replication

Regarding the degree of replication, almost all microservices have a single node by default, but as mentioned in the previous point, they could be manually scaled horizontally. The only

microservice that is replicated is MongoDB, the database, as the container used is already a cluster by design.

○ 2.3. Their elasticity (variation with load)

General Elasticity of the Architecture:

1. **Horizontal Scalability:** The architecture allows adding new instances of microservices, as previously explained with the API server, worker, or APISIX, without needing to change the configuration of existing services. This provides elasticity to the system.

2. **Load Balancing:** Components like APISIX can handle multiple instances of backend services, acting as a single entry point and distributing requests among the available instances. This facilitates elasticity by managing traffic spikes efficiently.

3. **Microservices Independence:** The microservices are independent, meaning they can be scaled individually based on the specific demands of each. For example:
    a. if the number of function execution requests increases, more instances of the worker can be added.
    b. If API requests increase, more instances of the API server can be deployed.

4. **Asynchronous Messaging with NATS:** Thanks to NATS, messages can be managed asynchronously, allowing consuming services (like the worker) to process messages at their own pace. This adds elasticity since workers can be scaled based on workload.

5. **Persistence with MongoDB:** As MongoDB is configured as a cluster, it can handle large volumes of data and scale horizontally to adapt to greater workloads.

The elasticity of this architecture is moderate in its current state. Microservices can be scaled horizontally, but the scaling process is manual rather than dynamic. However, with technologies like NATS, APISIX, and the MongoDB cluster, the architecture is well-prepared to become highly elastic.

3. A description of the system as a docker-compose file

The docker-compose file consists of the following services:

Services Overview:

1. **NATS**
    - **Image**: `nats:latest`

- **Ports**:
  - `4222`: For NATS clients to connect.
  - `8222`: For NATS API monitoring.
- **Environment**: Stores the JetStream data in `/data/nats-js`.
- **Volumes**: Data is persisted to `./nats-data`.
- **Command**: Enables JetStream storage and verbose logging (`-js --store_dir /data/nats-js -m 8222 -DV`).

Role: Acts as a message broker to facilitate communication between the **worker** and the api-server.

---

2. **Worker**
   - **Build Context**: `./worker/`
   - **Ports**:
     - `3001`: Exposes the worker service.
   - **Environment**: Loaded from `./worker/.env.pro`.
   - **Volumes**: Mounts the Docker socket (`/var/run/docker.sock`) to enable interaction with the Docker daemon.
   - **Depends On**: Starts after `nats` is ready.

   Role:

   - Subscribes to messages from **NATS** to execute registered functions.
   - Sends back responses to NATS for processing by the api-server.

---

3. **Mongo**
   - **Image**: `mongodb/mongodb-atlas-local`
   - **Ports**:
     - `27017`: MongoDB's default client port.

   Role:

   - Provides a database backend for storing user data, functions and executions.

---

4. **API Server**
   - **Build Context**: `./api-server/`
   - **Environment**: Loaded from `./api-server/.env.pro`.
   - **Depends On**:
     - Starts after `apisix`, `mongo`, and `nats`.

   Role:

- Manages user creation, function registration, and function execution requests.

---

5. **APISIX**
   - **Image**: `apache/apisix`
   - **Ports**:
     - `9180`: For admin API access.
     - `9080`: For HTTP traffic.
     - `9091`: For metrics.
     - `9443`: For HTTPS traffic.
   - **Volumes**: Configuration file is mapped to `./apisix_conf/master/config.yaml`.
   - **Environment**: Includes `ADMIN_KEY` for API authentication.
   - **Depends On**: Starts after `etcd`.

   Role:

   - Serves as the API Gateway for routing requests to appropriate services.
   - Provides JWT-based authentication for securing endpoints.

---

6. **Etcd**

   - **Image**: `bitnami/etcd:3.4.9`
   - **Ports**:
     - `2379`: For client communication.
   - **Environment**:
     - Configured to allow unauthenticated access and enable v2 API.
   - **Volumes**: Persists data to `../etcd/etcd_data`.

   Role: Acts as a configuration store for **APISIX**.

---

7. **APISIX-Init**
   - **Image**: `curlimages/curl:latest`
   - **Depends On**: Starts after `apisix` and `etcd`.
   - **Volumes**: Includes a script `configure_apisix.sh` to set up routes for **API Server**.

   Role: Initializes **APISIX** by configuring routes for the **API Server** using its admin API.

4. System described

4.1. How the workers do work

Overview

Workers in the FaaS system execute user-defined functions by interacting with the NATS messaging system and Docker runtime. Each worker subscribes to a task queue (functions) and processes tasks asynchronously, ensuring efficient and isolated function execution.

Message Handling

1. **Task Subscription**:
    - Workers listen to the functions queue (configurable via FUNCTION_DISPATCHING_QUEUE in .env.pro) using the FunctionExecutionRequestedListener class.
    - Messages of type FunctionExecutionRequestedEvent are received, containing:
        - executionId: A unique ID for the execution.
        - func: The function details, including the Docker image to use.
2. **Result Publishing**:
    - After task execution, the worker emits a FunctionExecutionCompletedEvent containing the execution result to a response topic (execution.<executionId>).

Execution Process

The worker's execution process includes:

1. **Ensuring Image Availability**:
    - The ExecuteFunctionService checks if the required Docker image exists locally:
        - If unavailable, it pulls the image from Docker Hub.
        - Local availability is determined by inspecting RepoTags and IDs of existing images.
2. **Container Creation and Execution**:
    - A Docker container is created using the specified image.
    - The container is started, and logs are streamed from stdout and stderr.
3. **Log Handling**:
    - Logs are cleaned using the cleanDockerLogs method to remove unwanted characters and formatting artifacts.
4. **Resource Cleanup**:
    - The container is stopped and removed after execution, ensuring no lingering resource usage.

Error Handling

1. **Execution Errors**:

    - Errors during Docker operations (e.g., pulling images or container creation) are caught and included in the logs returned to the client.
2. **NATS Message Handling**:
    - Durable subscriptions (worker-durable) and acknowledgment settings ensure no task is lost, even if a worker fails.

4.2. How workers are scaled

Manual Scaling with Docker Compose

The workers are designed to scale horizontally by adding or removing instances. Each instance operates independently, connecting to the NATS server and subscribing to the functions queue.

Steps to Scale:

1. **Update Worker Instances**:
    - Use the docker-compose command to scale the worker service:

    docker-compose up --scale worker=<number_of_instances>

For example, to scale to 3 worker instances:

    docker-compose up --scale worker=3

2. **Verify Scaling**:
    - Use docker ps to check running instances.
    - Confirm new workers have connected to NATS via logs.
3. **Automatic Subscription**:
    - New workers automatically join the durable consumer group (worker-group) and start processing tasks.

Load Balancing

The NATS JetStream ensures tasks are distributed across active workers:

- Messages are delivered to workers based on availability.
- Each task is processed exactly once, ensuring reliable execution without duplication.

Limitations

- **Manual Scaling**:
    - The current implementation requires administrators to manually scale workers using Docker Compose commands.
- **No Dynamic Response to Load**:
    - Worker scaling does not adjust automatically based on queue length or workload.

Potential Improvements - Kubernetes Integration:

The system could be deployed on Kubernetes and use the Horizontal Pod Autoscaling (HPA) to dynamically scale workers based on resource usage or custom metrics.

4.3. How the whole system should be configured to adapt to changes in load

Docker Compose, compared to other container orchestrators, offers a basic feature set and lacks built-in support for horizontal or vertical autoscaling. To implement autoscaling, the system should be deployed using a more advanced orchestrator such as Kubernetes, either locally or in a cloud provider like AWS. Kubernetes allows configuring Horizontal Pod Autoscaling (HPA) based on resource metrics (e.g., CPU, memory) of the pods, enabling dynamic scaling based on workload demands.

Alternatively, autoscaling can be implemented manually by configuring the api-server and worker to report their resource metrics to a monitoring system like Prometheus. A custom autoscaler service can then poll these metrics and decide when to send scale-up or scale-down commands to the Docker daemon managing the containers.

For database scaling, the most straightforward approach is to use a managed MongoDB solution, such as MongoDB Atlas, and delegate scaling responsibilities to the cloud provider. The mongodb-atlas-local image used in the current Docker Compose file already operates as a cluster, which limits its scalability options within this setup. However, another viable approach is leveraging a KEDA scaler for MongoDB to enable dynamic scaling.

Similarly, for NATS, the simplest approach is to utilize a managed service and offload scaling responsibilities to a provider. If this option is not feasible due to cost constraints, a viable alternative is using one of the existing KEDA scalers specifically designed for NATS to achieve automated scaling.

4.4. How access to external services on the part of functions should be handled in your implementations

In our implementation, access to external services is structured as follows:

1. **Docker Hub:** The worker nodes interact indirectly with Docker Hub through the Docker daemon (`docker.sock`). When a function is executed, the worker checks if the required image (provided by the user during function registration) is already available locally. If not, the daemon automatically pulls the image from Docker Hub. This process is handled by the `node-docker-api` library, which communicates with the Docker daemon to manage containers.

2. **NATS:** The communication between the API server and workers is managed through NATS. Execution requests are published to the `functions` queue, and workers subscribed to this queue receive the payload, process the request, and execute the

function. Results are then published back to a specific subtopic in the `execution.<executionId>` queue. This is implemented using the `@nestjs-plugins/nestjs-nats-jetstream-transport` library.

3. **APISIX:** APISIX acts as a reverse proxy and authentication gateway. It secures API endpoints using JWT tokens and routes requests to the upstream API server (`api-server:3000`). APISIX is configured to handle incoming requests on port `9080` and manages authentication and routing rules via its Admin API on port `9180`.

4.4.1. What handicaps/virtues presents.

- Handicaps:
    1. **Docker Hub dependency:** If the requested image is not cached locally, the Docker daemon needs to pull the image, which can introduce delays or trigger rate limits if Docker Hub is accessed frequently.

    2. **NATS scaling complexity:** Managing the number of workers and subscriptions efficiently is critical to avoid overloading the message queue.

    3. **APISIX maintenance:** Configuring JWT authentication, managing routes, and keeping the APISIX proxy secure requires careful planning and periodic maintenance.

- Virtues:
    1. **Docker daemon abstraction:** By relying on the Docker daemon, the worker does not directly interact with Docker Hub, simplifying image management and ensuring compatibility with local caching.

    2. **NATS scalability:** NATS enables a decoupled communication architecture, allowing workers to handle function execution asynchronously and scale based on demand.

    3. **APISIX security and flexibility:** The reverse proxy provides centralized control over authentication, routing, and load balancing, improving system reliability and security.

4.4.2. What additional data ought to be included in the encoded strings

- **JWT Token**: Include the user's JWT token for authentication through APISIX.
- **Function Metadata**: Image name (e.g., `user/image:tag`) to identify the container that needs to be pulled and executed. User ID to associate the request with the registered user.
- **Queue Details:** Topic/subtopic names for publishing the execution results (e.g., `execution.<executionId>`).

4.4.3. Any other important matter

1. **Worker-Docker Communication:** Workers communicate with the Docker daemon via a socket (`docker.sock`) using the `node-docker-api` library. This allows the workers to manage containers, including pulling images, starting containers, retrieving logs from `stdout`, and cleaning up resources by stopping and deleting containers after execution.

2. **APISIX Configuration:** APISIX is deployed in traditional mode, combining the `data_plane` and `control_plane`. It uses the `jwt-auth`plugin to validate user requests and routes them to the API server. Rules like CORS are globally configured to support cross-origin requests.

3. **NATS Robustness:** NATS topics are separated into `functions` (execution requests) and `execution.<executionId>` (results). This decoupled design allows the system to process multiple requests in parallel while ensuring results are directed back to the appropriate client.

4. **Caching and Rate Limiting:** To reduce latency and avoid hitting Docker Hub rate limits, Docker images are cached locally. If an image is frequently used, it remains available on the worker's machine without needing to pull it repeatedly.

5. **Resource Management:** Workers manage the container lifecycle to optimize resource usage. Containers are stopped and removed after execution, ensuring no unnecessary resources are consumed.

6. **Logging and Monitoring:** Each execution request is logged with a unique identifier, allowing detailed tracing across Docker, NATS, and APISIX. Logs from container execution (`stdout` and `stderr`) are captured and included in the result.

5. Comparison with other proposals for open source FaaS systems (OpenFaaS, OpenFunction)

In this section, it’s going to compare the project architecture with Open Source FaaS Solutions like OpenFaaS and OpenFunction

1. General Architecture

The FaaS in this project uses APISIX as the API Gateway, MongoDB for storage, and NATS for asynchronous communication. Service orchestration is manual and directly depends on `docker-compose`. In comparison, OpenFaaS is Kubernetes-based and provides a function controller, an API Gateway, and tools like Prometheus for metrics. Its approach is more integrated with Kubernetes, enabling greater automatic orchestration. On the other hand,

OpenFunction is also Kubernetes-based but integrates modern technologies like Knative and Dapr, which are designed to execute serverless functions with an event-driven focus.

2. Elasticity and Scalability

The FaaS in this project has moderate elasticity, as it allows horizontal scaling of microservices by manually adding new instances in the `docker-compose.yml` file. Components like APISIX facilitate load balancing, but there is no integrated dynamic autoscaling system. MongoDB, being configured as a cluster, provides elasticity in storage.

In comparison, OpenFaaS and OpenFunction have significantly higher elasticity due to their integration with Kubernetes. Both solutions support dynamic autoscaling based on performance metrics. Additionally, OpenFunction supports scale-to-zero, where functions are automatically deactivated when there is no traffic, saving resources.

3. Functionalities

In the project's distributed system, NATS provides support for asynchronous tasks, which is ideal for managing events between services like the API server and workers. Functions are executed within Docker containers, and the ecosystem is customized, lacking additional tools like integrated monitoring or deployment templates.

In contrast, OpenFaaS offers pre-configured templates for functions and has a broader ecosystem with native support for Prometheus and Grafana. OpenFunction, designed for modern distributed systems, supports more complex events and multiple programming languages, in addition to integrating advanced technologies like Knative and Dapr for smooth communication and advanced observability.

4. Ease of Use

The implemented FaaS is simple to use and deploy. With `docker-compose`, it is easy to configure services and quickly deploy them in small or controlled environments. However, scaling and configuring new services are manual processes, which can be a limitation in more complex scenarios. In contrast, OpenFaaS has a moderate learning curve since it requires familiarity with Kubernetes, but it provides tools like `faas-cli` to simplify deployment. OpenFunction, on the other hand, is more complex due to its reliance on Knative, Dapr, and Kubernetes, but it offers greater flexibility and support for advanced distributed systems.

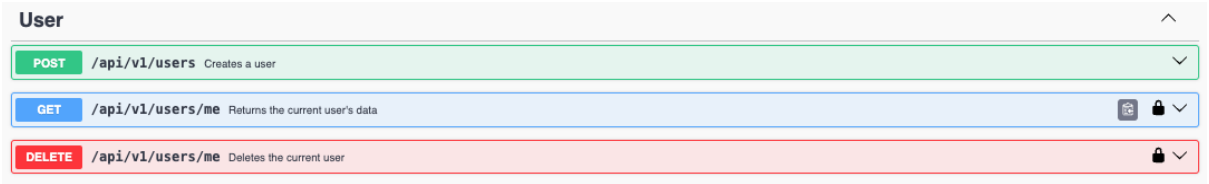
5. Use Cases

The FaaS in this project iis ideal for small, controlled environments or custom projects where simplicity and manual control are preferred. However, it lacks advanced tools for autoscaling and monitoring, making it less suitable for production environments with high workloads or distributed systems. OpenFaaS is a more mature solution, ideal for production environments that require dynamic scaling and support for fast, container-based functions. OpenFunction is targeted at complex and distributed systems that require advanced serverless functionalities and sophisticated event management.

6. API Documentation

Version: api/v1, table of contents:

- **User**



  - Create user:
    - Endpoint: POST /api/v1/users
    - Description: Creates a new user in the system
    - Request Body:

```
{
  "email": "user@example.com",
  "password": "password123"
}
```

    - Response Code: 201 - User created successfully
    - Response Body:

```
{
  "message": "User created successfully."
}
```

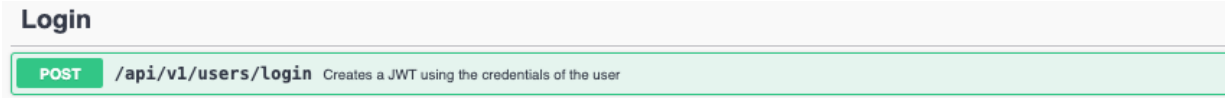
  - Get current user info:
    - Endpoint: GET /api/v1/users/me
    - Description: Retrieves the current user information. The user ID is extracted automatically from the JWT token and it is used to search the information.
    - Response Code: 200 - User Information retrieved successfully
    - Response Body:

```
{
  "userId": "6793b2a097db3ebefc24a16f",
  "email": "user@example.com"
}
```

  - Delete user
    - Endpoint: DELETE /api/v1/users/me
    - Description: Deletes the user from the database and the consumer of apisix, to make the JWT token created invalid.
    - Response Code: 200 - User email deleted successfully
    - Response Body:

```
{
  "message": "User email user@example.com deleted successfully."
}
```

- **Login**



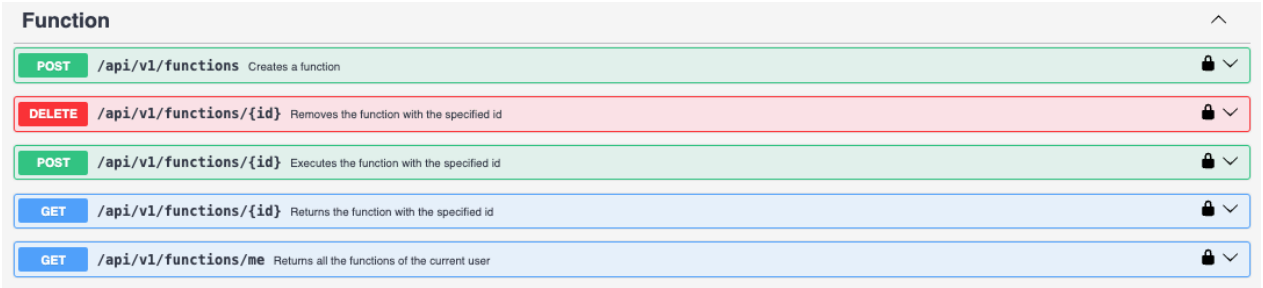
  - User Login:
    - Endpoint: POST /api/v1/users/login
    - Description: Logs in a user and generates a JWT token.
    - Request Body:

```
{
  "email": "user@example.com",
  "password": "password123"
}
```

    - Response Code: 201 - User logged in successfully
    - Response Body:

```
{
  "message": "User logged successfully.",
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJrZXkiOiJ1c2VyMUBleGFtcGxlLmNvbSIsI
OjE3Mzc2NzUyNzUsImV4cCI6MTczNzc2MTY3NX0.FGzFnA10-iTRgQu2o1ElRfP7FqJbd8FghvLkw2T3z1o"
}
```

- **Function**



  - Create Function:
    - Endpoint: POST /api/v1/functions
    - Description: Creates a new function associated with the authenticated user. The authorization header with the JWT token is added automatically thanks to swagger
    - Request Body:

```
{
  "image": "hello-word"
}
```

    - Response Code: 201 - Function created successfully
    - Response Body:

```
{
  "message": "Function created successfully."
}
```

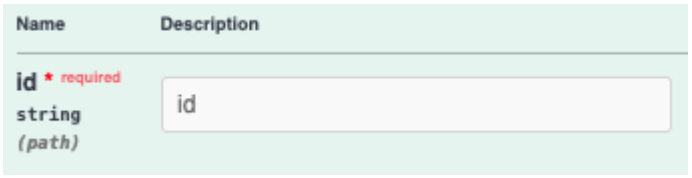
  - Delete Function
    - DELETE /api/v1/functions/{id}
    - Description: Deletes a function by specifying its ID. The function must be associated with the authenticated user.

- Parameter: ID of the function

- Response Code: 201 - Function deleted successfully:
- Response Body:

```
{
  "message": "Function deleted successfully."
}
```

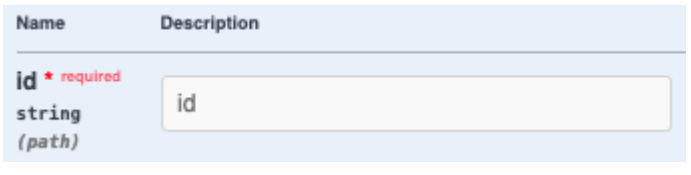
- Execute Function
  - POST /api/v1/functions/execute/{id}
  - Description: Executes a function by specifying its ID. The function must be associated with the authenticated user.
  - Parameter: ID of the function



  - Response Code: 201 - Function executed successfully (e.g nats)
  - Response Body:

```
:[1] 2025/01/23 23:57:05.666318 [INF] Starting nats-server
9[1] 2025/01/23 23:57:05.666436 [INF]   Version:  2.10.24
;[1] 2025/01/23 23:57:05.666437 [INF]   Git:      [1d6f7ea]
<[1] 2025/01/23 23:57:05.666438 [INF]   Cluster:  my_cluster
j[1] 2025/01/23 23:57:05.666439 [INF]   Name:     NCAEF5ESBOLV6NGUCEXSDQJXFWOAQWIN6J246II5YE5LULMUD3C6D2DW
j[1] 2025/01/23 23:57:05.666440 [INF]   ID:       NCAEF5ESBOLV6NGUCEXSDQJXFWOAQWIN6J246II5YE5LULMUD3C6D2DW
P[1] 2025/01/23 23:57:05.666447 [INF] Using configuration file: nats-server.conf
K[1] 2025/01/23 23:57:05.666816 [INF] Starting http monitor on 0.0.0.0:8222
V[1] 2025/01/23 23:57:05.666956 [INF] Listening for client connections on 0.0.0.0:4222
5[1] 2025/01/23 23:57:05.667150 [INF] Server is ready
@[1] 2025/01/23 23:57:05.667225 [INF] Cluster name is my_cluster
U[1] 2025/01/23 23:57:05.667329 [INF] Listening for route connections on 0.0.0.0:6222
```

- Get Function By its ID:
  - GET /api/v1/functions/{id}
  - Description: Returns the function by specifying its ID. The function must be associated with the authenticated user.
  - Parameter: ID of the function



  - Response Code: 201 - Function returned successfully
  - Response Body:

```
{
  "functionId": "string",
  "image": "string"
}
```

- Get Functions of the current user:
  - GET /api/v1/functions/me
  - Description: Returns all the functions of the current user. The user must be authenticated.
  - Response Code: 200
  - Response Body:


```
[
  {
    "id": "6793b2cf97db3ebefc24a170",
    "image": "hello-word"
  },
  {
    "id": "6793ba5d97db3ebefc24a174",
    "image": "nats"
  }
]
```


- **JWT authentication:**
  - To authenticate, first log in to obtain a JWT token. Then, click the button below and enter the token in the "Value" field.