Function as a Service provider via git repositories

Miquel Viel Vàzquez

Oscar Roselló Ibàñez

Alexandre Moncho Rubio

Abstract — **Function-as-a-Service (FaaS) is a revolutionary paradigm, redefining how applications are developed, deployed, and scaled. FaaS represents a serverless computing model where developers can focus solely on writing code without the burden of managing underlying infrastructure. This revolutionary approach enables efficient, scalable, and cost-effective solutions, unlocking new possibilities for application architecture. On the next lines we explain the project and how to deploy and test it. In addition, we describe how this git public projects must be built to use it on the FaaS.**

Keywords - FaaS; deploy; developers; git.

I. Introduction

In an era where an increasing number of tools are available and adapting infrastructure to a working environment becomes more challenging, it is becoming increasingly important to provide solutions that assist developers in focusing on the code and alleviate the arduous task of configuring these infrastructures. FaaS, as a subset of serverless computing, epitomizes this shift by enabling developers to break down applications into modular functions that can be executed in response to events, significantly reducing the complexity associated with infrastructure management. The objective of this paper is to address the implementation, deployment, and operational testing of the developed FaaS, as well as to illustrate how it is deployed and utilized.

II. Methodology

During this section, the various tools used in the project's development will be presented.

A. Typescript (node)

TypeScript is a programming language developed and maintained by Microsoft. It is a superset of JavaScript, which means that any valid JavaScript code is also valid TypeScript code. TypeScript adds static typing to JavaScript, introducing a type system that allows developers to define and enforce data types for variables.

We will use Typescript as the programming language to develop the project because of its static typing, the enhanced code quality, its object-oriented programming features and its community support.

B. NATS

NATS, which stands for "Nano Message Bus" or "NATS.io," is an open-source messaging system designed for building distributed and scalable systems. It provides lightweight and high-performance communication between services or components in a network. NATS was originally developed by Derek Collison and is now part of the Cloud Native Computing Foundation (CNCF).

We will use NATS as the communication middleware to develop the project because of it is lightweight and fast, it publish-subscribe and queue groups models, the scalability, the security and because it is cloud-native.

C. Docker composes.

Docker Compose is a tool for defining and running multi-container Docker applications. It allows you to define a multi-container Docker application in a single file, typically named docker-compose.yml. This file includes the configuration for each service (container) in your application, such as image, ports, volumes, and environment variables.

We will use docker compose as our deployment tool because of defining and managing multi-container applications, its simplified deployment, its isolation and portability, the environment configuration, and the service scaling.

D. Git (GitHub)

Leveraging Git's branching capabilities, we were able to work on features and fixes in isolated environments, ensuring that changes could be merged into the main codebase without disruption. Its conflict resolution tools proved invaluable in handling discrepancies that arose when multiple developers edited the same files concurrently, while integration with code review systems facilitated continuous improvement and knowledge sharing among team members. Furthermore, Git seamlessly integrated with our CI/CD pipelines, automating testing and deployment processes, thereby ensuring that our program maintained ambitious standards of reliability and stability throughout its development lifecycle.

Git served as the backbone of our program's development process, offering a robust version control system that facilitated seamless collaboration and efficient code management.

III. Development

For the development of FaaS, the components have been divided into different microservices that will perform various tasks necessary for the operation of the service.

A. Frontend

The frontend microservice operates as an API gateway within the architecture, encapsulating a Node.js runtime environment within a container. It specifically manages user requests, ensuring access is granted solely upon presentation of a valid bearer token acquired via OAuth.

Key components of this microservice include seamless integration with OAuth for user authentication and authorization, along with a suite of API endpoints. These endpoints facilitate various operations, such as job submission, status checks, result retrieval, and viewing of owned jobs.

Integral to its functionality is the validation of bearer tokens, enforced through rigorous examination of request headers. Requests lacking a valid token are promptly rejected, safeguarding the system against unauthorized access.

For ease of deployment and maintenance, the microservice is containerized using Docker, fostering consistency and portability across diverse environments. Furthermore, its tight integration within the architecture allows for seamless communication with other microservices and external systems.

B. NATS

The NATS microservice aims to serve as the communication middleware among the other microservices. This microservice will be deployed as a cluster of three servers that will provide different queues through which the other microservices communicate.

For communication, the pub/sub pattern will be implemented, with the workers belonging to the same queue group so that only one subscribes to the jobs. Additionally, three queues will be implemented:

Job Queue: The frontend will publish various jobs to be done along with their information, and the workers will subscribe to this as Workers group, ensuring that each task is assigned to only one worker.

Results Queue: The workers will publish the status and results of different jobs, and the frontend will subscribe to receive them.

Observer Queue: Pending

C. Worker

The worker microservice aims to serve as the function executer. It runs a git repository and returns the results. Worker communicates with frontend via Nats using Job queue to receive a function and Results Queue to send the results. After sending the results, workers can get another job to run it.

There are 3 types of git public repositories which can be executed by the worker:

Python, worker install all dependencies. After that, we ran the python main script. Finally, when the python script ends, we clear all python dependencies.

Rust, we ran a cargo project and received the logs to send it to the Frontend. Rust implements all dependencies with "cargo run", compile and run it. All the modules required are installed on a folder inside the git folder.

NodeJS. Scripts with Node (or compiled TypeScript, so its Node) can be executed on the worker. Firstly, we install all dependencies on the npm project. When all dependences are finished, we run the "npm start" script. All the modules required are installed on a folder inside the git folder.

At the end, all git folders are deleted (Rust and Node required modules with it) and, finally, the worker is ready for other functions to execute.

D. Observer

The observer returns some statistics:

The average of request per minute

The average jobs completed per minute.

The average time to a worker executes a function per minute.

IV. Deployment

The deployment encompasses several interconnected services orchestrated using Docker Compose version 3.5. At its core is the NATS Cluster, leveraging NATS image to establish a distributed cluster comprising three nodes: Nats, nats-1, and nats-2. Each node is intricately configured to actively participate in the cluster, facilitating seamless communication across the system.

The Worker Service forms a crucial component, deploying two replicas to enable concurrent execution of tasks. These worker instances are designed to interact closely with the NATS cluster, leveraging it for efficient message passing and coordination, thereby enhancing system responsiveness and scalability.

In parallel, the Frontend service is deployed and exposed on port 3000, providing a user-friendly interface for interaction. This service acts as a bridge between users and the underlying system, seamlessly managing job submissions and presenting results. It establishes communication channels with both the NATS cluster and the Worker Service, ensuring smooth operation and real-time updates.

The Observer service plays a pivotal role in system monitoring and observability, continuously tracking system activities and performance metrics. Integrated with the NATS network, it gains access to real-time data, enabling proactive monitoring and swift response to potential issues.

Furthermore, all services are interconnected via the NATS network, fostering efficient data exchange and collaboration. This network architecture promotes seamless communication and coordination among services, facilitating streamlined operations and enhancing overall system efficiency.

This deployment architecture underscores scalability, resilience, and modularity, facilitating the efficient execution of distributed tasks within the system while ensuring optimal performance and reliability.

V. TEST

To test the application, we created 3 git repositories with 3 different programs in different languages. The repositories are:
https://github.com/mvievaz/PI-Test-for-FaaS : with python test.
https://github.com/mvievaz/Golden-Ratio-for-Faa : with rust test
https://github.com/mvievaz/Taylor-series-for-FaaS : with NodeJS test

VI. USAGE GUIDE

Frontend

To use this FaaS, we first need to authenticate the user via OAuth. To do that, we will use the endpoint /OAuth/authorize, to enter in the google OAuth section. After the google login, we will receive the bearer token used to authenticate in every job Endpoint of the API.

Jobs inside have 4 possible statuses: “pending”, “working”, “error”, “finish”.

Pending: Job waiting to be assigned to a worker service

Working: Job executing in a worker service

Error: Job aborted execution

Finish: Job finished correctly

We utilized the URL localhost:3000 to access all endpoints of our API.

A I. ENDPOINTS OF THE FRONEND

ENDPOINTS	***Method***	**Headers**		**Body**	**Description**
		Authorization	***Content-type***	***JSON type***	
/job/send-job	POST	Bearer token	application/json	Mandatory: “URL” field, with the URL provided of the git repository, Optional: “name” field, with a name assigned to the job	This endpoint with the correct structure will send the job to the frontend

/job/job-status/	GET	Bearer token	application/json	Mandatory: “jobID” field, with the id of the job for finding the status	This endpoint with the correct structure will return the status of the job requested
/job/result/	GET	Bearer token	application/json	Mandatory: “jobID” field, with the id of the job for finding the result	This endpoint with the correct structure will return the result of the job requested
/job/list-jobs	GET	Bearer token	application/json		This endpoint with the correct structure will return a list of the jobs owned
/observer/stats	GET	Bearer token	application/json	Mandatory: “secret” field, with the secret password of the admin The secret is: **adminSecret1234**	This endpoint with the correct structure will return a list of the stats of the observer.
/oauth/authorize	GET				This endpoint is used to authenticate the user via google oauth
/oauth/callback	GET				This endpoint is used to get the callback of the oauth, not useful for the user.

A 1. API usage table

In a curl example of sending a job to the frontend (First, you need to use the /oauth/callback endpoint to get the bearer token):

```
curl --location 'http://localhost:3000/job/send-job' \
--header 'Content-Type: application/json' \
--header 'Authorization: Bearer {{YOUR JWT BEARER TOKEN HERE}' \
--data '{
    "URL":"https://github.com/mvievaz/PI-Test-for-FaaS.git",
    "name": "Job1"
}'
```

Worker

In our architecture, we support the execution of various programming languages within our worker services. However, for these services to operate effectively, the associated Git repository must include a document detailing the execution context. This document will be **faas-manifest.json** with the following fields:

- “language”: Field mandatory with 3 options: “python”, “rust”, “nodeJS”
- “arg”: Field optional to pass parameters to the main function, works on python and rust.
- “requirementsFile”: Field mandatory with the pip packages to download. Works on python only. If there is no requirement needed, the field must be equal to “” (empty string)
- “mainFile”: Field mandatory when using python language to specify the python file to execute.

When the git repository is structured for Node.js, it must include a package.json file containing dependencies and a preconfigured npm start command to ensure proper functionality.

VII. Expansions:

Possible expansions include elaborating on job details, such as adding properties like maximum assigned work time or specific execution requirements. Another enhancement could involve implementing a user storage microservice, moving away from the current dictionary-based approach to a dedicated microservice with a NoSQL database for scalability. Additionally, diversifying the types of jobs supported by the worker could be beneficial. Currently, the worker handles jobs programmed in Python, Node.js, or Rust, but expanding language support could broaden its capabilities. Furthermore, we could use Kubernetes to orchestrate our containers. Additionally, Kubernetes could be employed for container orchestration in our infrastructure. Add environment variables to hardcoded values.

VIII. Conclusions

In conclusion, function-as-a-Service (FaaS) represents a paradigm shift in application development, deployment, and scalability. By abstracting away infrastructure management, FaaS allows developers to focus solely on writing code, resulting in efficient, scalable, and cost-effective solutions. In this paper, we detailed the development, functionality, and deployment of a FaaS service leveraging Git repositories.

The development process involved utilizing TypeScript for its static typing and object-oriented features, NATS as the lightweight and scalable communication middleware, Docker Compose for simplified multi-container application deployment, and Git for version control and collaboration. These tools were instrumental in ensuring code quality, seamless communication, and streamlined deployment.

The FaaS service comprises several microservices, each serving a specific function within the architecture. The frontend service acts as the API gateway, managing user requests and enforcing authentication via OAuth bearer tokens. Meanwhile, the NATS service facilitates communication between microservices, employing pub/sub patterns and queue groups for efficient message passing. The worker service executes tasks in response to user petitions, leveraging Git repositories to define execution contexts.

Deployment involves orchestrating multiple interconnected services using Docker Compose. The NATS cluster, worker service replicas, frontend service, and observer service are deployed and interconnected via the NATS network, ensuring seamless communication and coordination. This deployment architecture emphasizes scalability, resilience, and modularity, facilitating the efficient execution of distributed tasks.

In conclusion, the development, functionality, and deployment of the FaaS service underscore the transformative potential of serverless computing. By abstracting away infrastructure concerns and leveraging Git repositories for code management, our FaaS service offers developers a streamlined approach to building and deploying scalable applications, ushering in a new era of software development.

Bibliographic References

Node.js. (n.d.). https://nodejs.org/

JavaScript with syntax for types. (n.d.). https://www.typescriptlang.org/

NATS.io. (n.d.). https://nats.io/
Docker: Accelerated Container Application Development. (2024, January 23). Docker. https://www.docker.com/
GitHub: Let’s build from here. (n.d.). GitHub. https://github.com/