Cloud Computing final project: creation of a cloud-native microservices application

Project “Tre Acque”

Team “Tre Acque”

Emanuele Petriglia (mat. 888435) Vincenzo Corso (mat. 880965)

1 February 2023

Table of contents

1. Introduction

What is Tre Acque

Tre Acque is a crowd-sourced public database about drinking fountains located on the national territory.

The application allows users to add or remove a fountain with a name, search fountains in a selected map area, give a vote in a five star-based scale, see the average rating, and keep updated via email for added or removed fountains.

Figure: Tre Acque logo

Who is Tre Acque

As written in the title page, we are:

- Emanuele Petriglia (`e.petriglia@campus.unimib.it`) (mat. 888435)
- Vincenzo Corso (`v.corso3@campus.unimib.it`) (mat. 880965)

There is also the collaboration of Gabriele Chiodi (`g.chiodi5@campus.unimib.it`) (mat. 844648) for one microservice.

Repository and license

The project source code is tracked with a **git repository** hosted on the public GitLab instance:

`https://gitlab.com/tre-acque/tre-acque`

Figure: GitLab logo

We choose the **GNU Affero General Public License** version 3 (GNU AGPL) for our project, because it is a free, copyleft license suited for applications that run over a network, like Tre Acque.

Figure: GNU AGPL v3 logo

Interaction with the application

There are two ways to interact with the application:

1. Use the Web browser with a graphical UI;
2. Use `cURL` via command line.

More information about the interaction can be read on the `README.adoc` file in the repository's root directory.

2. Application backend view

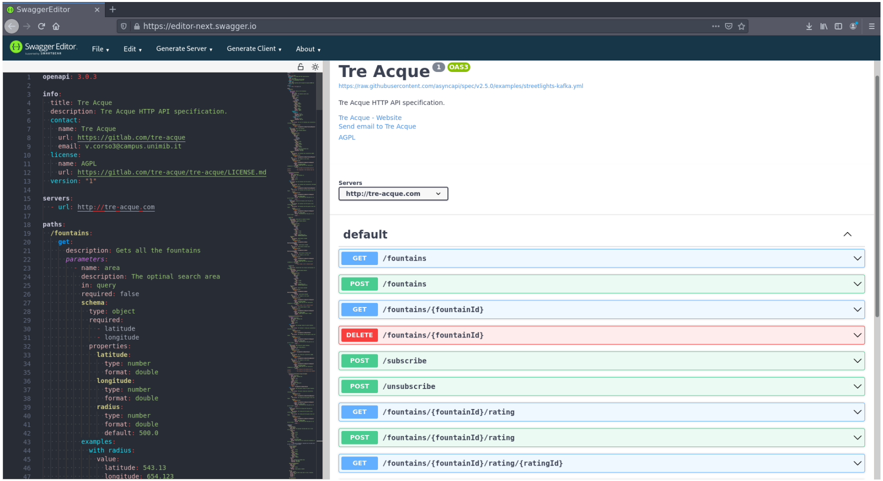
HTTP API

First, we designed an HTTP REST API for the application. The documentation is written in **Open API Specification** version 3, a standard, programming language-agnostic interface description for HTTP API.

The documentation can be found in `doc/api.yaml` file. This is a text file in YAML format that can be visually rendered with tools like **Swagger UI** (see next slide).

Figure: Open API logo

HTTP API: An example with Swagger UI



```
openapi: 3.0.3

info:
  title: Tre Acque
  description: Tre Acque HTTP API specification.
  contact:
    name: Tre Acque
    url: https://gitlab.com/tre-acque
    email: v.corso3@campus.unimib.it
  license:
    name: AGPL
    url: https://gitlab.com/tre-acque/tre-acque/LICENSE.md
  version: "1"

servers:
  - url: http://tre-acque.com

paths:
  /fountains:
    get:
      description: Gets all the fountains
      parameters:
        - name: area
          description: The optinal search area
          in: query
          required: false
          schema:
            type: object
            required:
              - latitude
              - longitude
            properties:
              latitude:
                type: number
                format: double
              longitude:
                type: number
                format: double
              radius:
                type: number
                format: double
                default: 500.0
          examples:
            with radius:
              value:
                latitude: 543.13
                longitude: 654.123
```

Tre Acque 1 OAS3

https://raw.githubusercontent.com/asyncapi/spec/v2.5.0/examples/streetlights-kafka.yml

Tre Acque HTTP API specification.

Tre Acque - Website
Send email to Tre Acque
AGPL

Servers
http://tre-acque.com

default

GET	/fountains
POST	/fountains
GET	/fountains/{fountainId}
DELETE	/fountains/{fountainId}
POST	/subscribe
POST	/unsubscribe
GET	/fountains/{fountainId}/rating
POST	/fountains/{fountainId}/rating
GET	/fountains/{fountainId}/rating/{ratingId}

Figure: Open API logo

Architecture Overview

At high level the architecture is composed by four services (fountain, notification, rating, frontend). The user interacts with the system only through the API Gateway. The backend services exchange messages using Kafka as message broker.

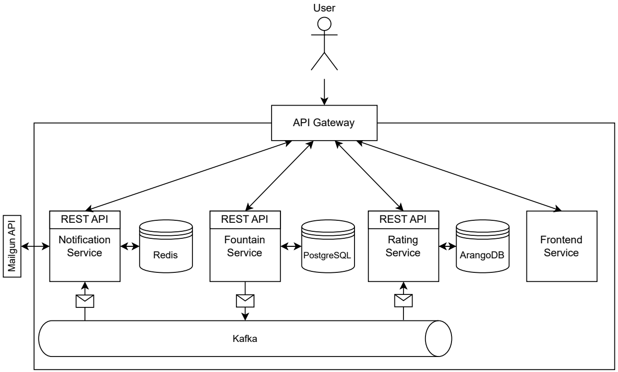


Figure: Tre Acque architecture

API Gateway

- The user sends requests to the gateway through HTTP.
- The gateway forward them to the right service.

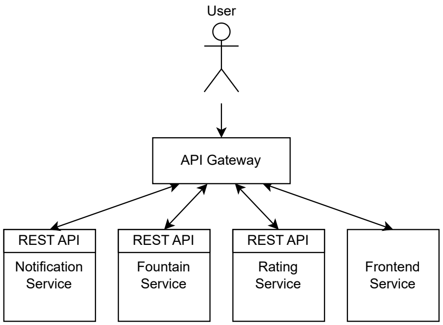


Figure: API Gateway

Fountain Service

- Is written in Java, using the Quarkus framework.
- Has three main responsibilities: adding, deleting and searching fountains (also in a certain area).
- Data are persisted in a PostgreSQL instance with a PostGIS extension installed to support geoqueries.
- When a new fountain is added or deleted, an event is sent to Kafka.

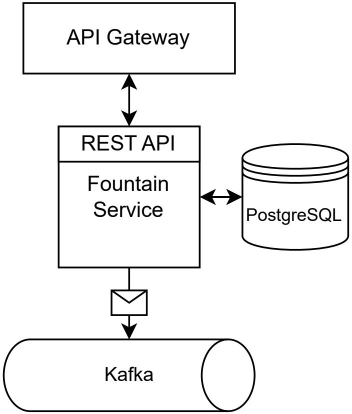


Figure: Rating Service

Rating Service

- Is written in Go.
- Has two main responsibilities: saving a user rating and calculating the average rating of a fountain.
- Data are persisted in an ArangoDB instance. Each document represents a fountain and collects its ratings.
- When a new fountain is added or deleted, the corresponding event is received and processed.

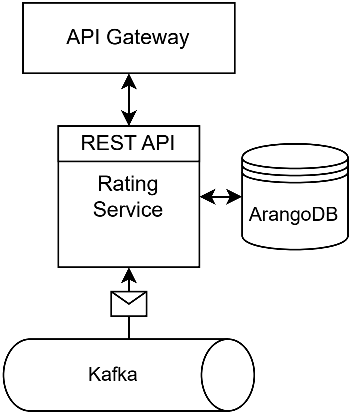


Figure: Rating Service

Notification Service

- Is written in Javascript, using NodeJS.
- Has two main responsibilities: subscribing/unsubscribing users to fountain events and sending emails through an external service (Mailgun).
- For the purpose of this demo, emails can be sent only to authorized recipients (i.e. the team members).
- Data are saved in a persistent Redis instance (AOF strategy). In particular there are two sets (one for each event) containing the user emails.
- When a new fountain is added or deleted, the corresponding event is received and processed.

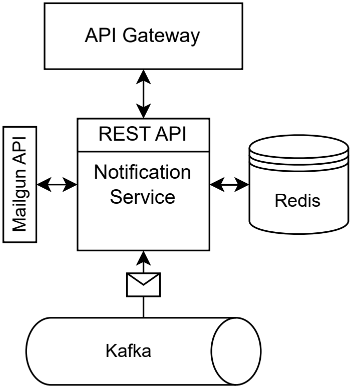


Figure: Notification Service

Kafka

- We chose Kafka as message broker because:
  - supports event replay (a must-have feature for microservices)
  - gives guarantees about message ordering (ensure data consistency)
- From version 2.8 there is the intent to replace Zookeeper with a consensus protocol called KRaft. From version 3.3 this mode is now production-ready.
- We used the KRaft mode (i.e. without zookeeper) to make it easier deploying Kafka.

Figure: Kafka Logo

Frontend Service

- This service returns a static html page containing the frontend app that runs on the client browser (client-side rendering).
- The app delivered is written with React.
- To show the map we used another external service called Mapbox.

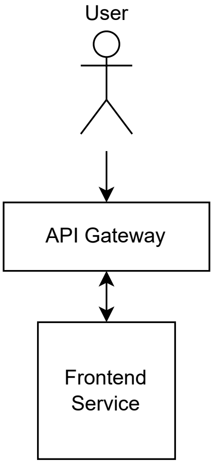


Figure: Frontend Service

Microservices Patterns

In summary:

- each service uses its own db (**Database-per-service Pattern**).
- the fountain service notifies other services sending an event through the message broker (**Messaging Pattern**).
- Users have a single point of access (**API Gateway Pattern**).

In addition:

- backend services offers three endpoints to verify if the instance is started, ready and live (**Health Check Pattern**).
- fountain service uses quarkus extensions to avoid reinventing the wheel and to simplify handling of the cross-cutting concern such as health checking and messaging (**Microservices Chassis Pattern**).

3. Development build and deploy view

Dockerfiles

Each microservice is **containerized**: we written a Dockerfile that specifies how to build the image.

We build the images in a **two phase stages**: one to compile the microservice source code and one to copy only the necessary files to run it. This is to minimize the image size.

- The only exception is notification service, because there is nothing to compile.

The dockerfiles can be found on the root directory of each microservice (as example `rating-service/Dockerfile`).

Figure: Docker logo

Docker Compose

For a local development environment, we wrote a `docker-compose.yaml` file to start the whole application through the **Docker Compose** tool. Make sure to set the correct environment variables for Mapbox and Mailgun.

This enables us to try and test small changes faster than deploy a whole Kubernetes cluster. And, because all services defined in the compose file expose the ports to the network host, we can test single microservices without rebuild the entire image.

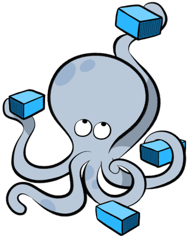


Figure: Docker Compose logo

Kubernetes

We also have a local development **Kubernetes** cluster on a single node, thanks to **minikube**. We wrote all Kubernetes manifest files to start up the application, they can be found on the `k8s` root directory.

Under this directory there are **two types of manifests**: one for development environment and one for production environment. There are some minor, but important, differences, like secrets and images tag for microservices.

A local cluster enables us to try the deploy without involving the production deploy process.

The in-depth report of the Kubernetes cluster we built is done on the next section.

Figure: Minikube logo

Skaffold

To deploy the development Kubernetes cluster, we use **Skaffold**, a tool that handles the building, pushing and deploying an application. We use it **only for development**, not for production.

Why? Because this tool has a development mode that builds automatically the microservices images, and continuous watches and applies (until is stopped) the manifests on the local cluster.

We remember, like for Docker Compose, to set up the correct values for Mailgun and Mapbox secrets. See also the `README.adoc` file for more information.

Figure: Skaffold logo

4. Production build and deploy view

An overview

For the build and deployment of the application, we use the **GitOps workflow**, supported by **GitLab**'s software and service (the repository, the CI/CD runner and the agent).

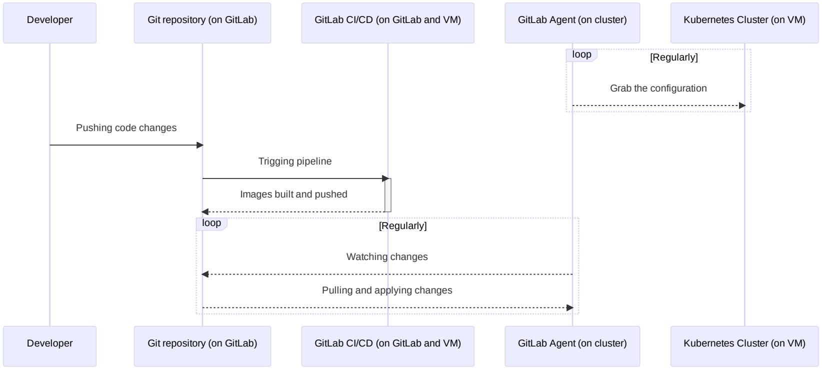

Developer
Git repository (on GitLab)
GitLab CI/CD (on GitLab and VM)
GitLab Agent (on cluster)
Kubernetes Cluster (on VM)
loop
[Regularly]
Grab the configuration
Pushing code changes
Trigging pipeline
Images built and pushed
loop
[Regularly]
Watching changes
Pulling and applying changes
Developer
Git repository (on GitLab)
GitLab CI/CD (on GitLab and VM)
GitLab Agent (on cluster)
Kubernetes Cluster (on VM)

Figure: Main actors and interactions in the GitOps deployment.

4. Production build and deploy view

GitLab CI/CD pipeline

GitLab CI/CD pipeline

The GitLab CI/CD pipeline is triggered on every commit pushed on the git repository. It builds the container images for each microservice and pushes them to our GitLab container registry.

The process of building and pushing is done with **Buildah** instead of Docker, because it has a very specific focus: build and push container images without involving a full container runtime (and a daemon like Docker).

Figure: Buildah logo.

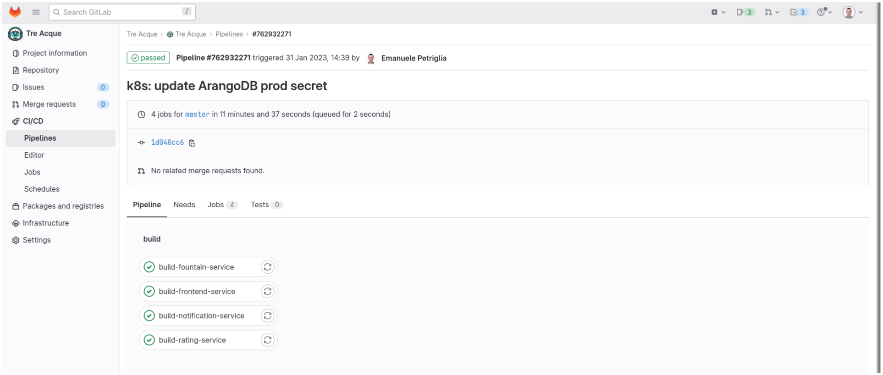
The self-hosted GitLab Runner

We do not use the public GitLab runners, because we have a limited set of minutes on the free tier. So we used the Azure virtual machine provided from the course laboratory to set up and run a GitLab runner.

The runner acts a **pull-based method**: it regularly asks the server if there are jobs to run.

There is only one problem: the GitLab runner and the Kubernetes cluster are on the same virtual machine, so they cannot be activated at the same time due to limited hardware resources.

A pipeline example (pt. 1)


Tre Acque › Tre Acque › Pipelines › #762932271

passed Pipeline #762932271 triggered 31 Jan 2023, 14:39 by Emanuele Petriglia

k8s: update ArangoDB prod secret

4 jobs for `master` in 11 minutes and 37 seconds (queued for 2 seconds)

`1d040cc6`

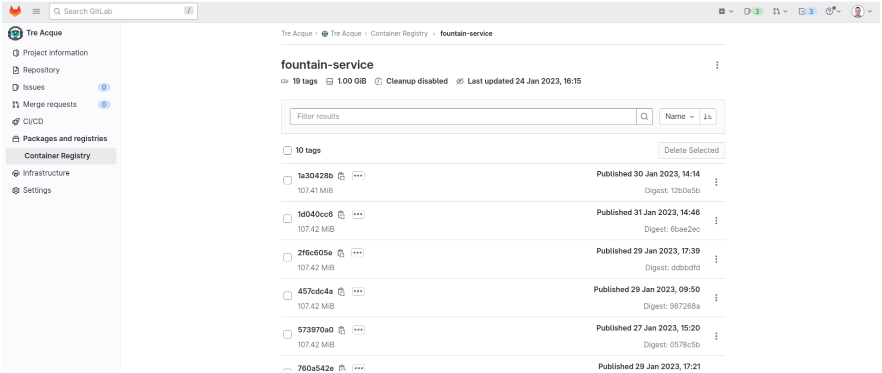
No related merge requests found.

build

- build-fountain-service
- build-frontend-service
- build-notification-service
- build-rating-service

Figure: The screenshot shows a single pipeline with four jobs, one for each microservice. Each job is pulled and executed by the self-hosted runner.

A pipeline example (pt. 2)



fountain-service

19 tags · 1.00 GiB · Cleanup disabled · Last updated 24 Jan 2023, 16:15

10 tags

1a30428b 107.41 MiB	Published 30 Jan 2023, 14:14 Digest: 12b0e5b
1d040cc6 107.42 MiB	Published 31 Jan 2023, 14:46 Digest: 6bae2ec
2f6c605e 107.42 MiB	Published 29 Jan 2023, 17:39 Digest: ddbbdfd
457cdc4a 107.42 MiB	Published 29 Jan 2023, 09:50 Digest: 987268a
573970a0 107.42 MiB	Published 27 Jan 2023, 15:20 Digest: 0578c5b
760a542e	Published 29 Jan 2023, 17:21

Figure: The screenshot shows the container registry for a single microservice. Each image is tagged with the short commit hash.

4. Production build and deploy view

The Kubernetes Cluster

An overview of Kubernetes objects

Under `k8s/prod` folder there are the Kubernetes manifests for production environment. There are subfolders for each object type:

- **configmaps**: configuration files (actually only for PostGIS);
- **deployment**: deployment objects for each microservice;
- **ingress**: a single ingress object that manages external access to the services in the cluster (acts as gateway to microservices);
- **pvc**: persistent volume claims for the stateful sets;
- **secret**: secrets objects (see next slide);
- **services**: services objects for each deployment and statefulset.
- **statefulset**: stateful sets objects for each database and Kafka.

We decided to limit resources for Java applications, so we have two cases: `fountain-service` and `kafka` pods.

The agent for secret objects

We use **Sealed Secrets** controller and application to manage secrets.

Under `secret` folder there are *Sealed Secret* objects, they contains an encrypted version of the original secret object. When these objects are pulled and applied on the cluster, the controller decrypts them, so they are available to the pods. The secret key is stored on the cluster, while the public key can be used with the paired application to create these objects.

The GitLab agent for GitOps

The deployment is done thought the **GitOps workflow**. We configured and installed the GitLab agent operator on the cluster. It reads the configuration from a linked repository, watches and pulls manifests declared in them, and applies to the cluster.

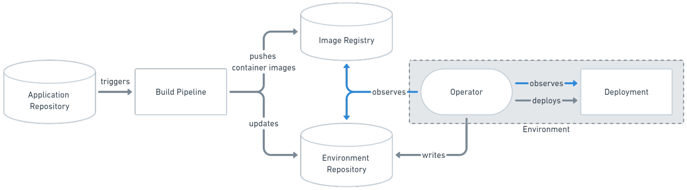
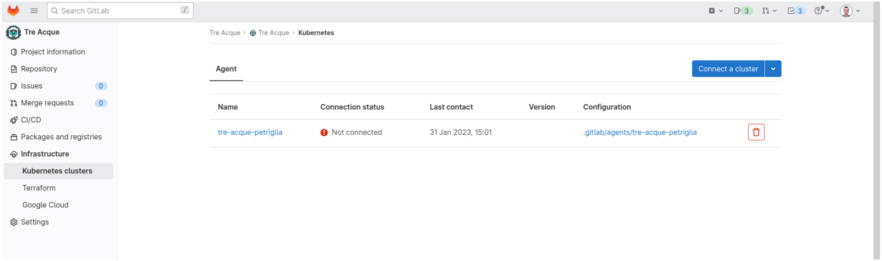


Figure: The GitOps pull-based deployment. Taken from `https://www.gitops.tech/`

The configuration is stored in `.gitlab/agents/AGENTNAME`, where `AGENTNAME` is the name of the agent, in this case `tre-acque-petriglia`.

The link between the agent and the repository



Name	Connection status	Last contact	Version	Configuration
tre-acque-petriglia	Not connected	31 Jan 2023, 15:01		.gitlab/agents/tre-acque-petriglia

Figure: In this page we can see the connected agents and we can also add more agents. The current, and only, cluster is powered off in this screenshot.

5. Conclusion

The result

The application works and can be deployed on a Kubernetes cluster. We think we satisfied all requirements imposed for the project assignment.

We learnt lot of things doing this projects, exploiting every piece of experience and concepts learn throughout the course, university and personal project.

There are lots of improvements to do and there is lot to study. We proposed some next steps in the following last slides.

Future works (pt. 1)

- **Transactional Outbox Pattern**: for now, persisting an entity and sending an event won't be done atomically. We should use a CDC Service (Debezium) to ensure data consistency between services.
- **API Composition Pattern**: for now, to get the fountain details and its average rating, two different requests must be made. Composing these two APIs in the gateway will improve the communication.
- **Continous Monitoring**: for now the application backend and microservices are not monitored, logs and metrics are not gathered and analysed. We should evaluate a solution (Prometheus, Zabbix...).

Future works (pt. 2)

- **Skaffold**: we use this tool in development environment, to build images and deploy a local test cluster. We should evaluate if keep this tool, and if yes, explore it in more detail.
- **Secrets**: for now secrets are not rotated and the private and public keys are managed manually. We should evaluate something more automatized (HashiCorp Vault).
- **Helm**: we have duplicated manifest for development and production environments. We should evaluate Helm to get parametrized manifests and automatic cluster set-up.

Future works (pt. 3)

- **Podman with CRI-O**: currently the default minikube's driver is Docker. We should explore the Podman driver with the CRI-O container runtime.
- **Kubernetes repository**: We had some inconveniences having the Kubernetes manifests and the microservices source code on the same repository, especially with the pipeline activation. We should separate the two things in two different repositories.
- **GitLab Runner**
  - **Configure cache**: currently the custom runner runs each job without the cache enabled. We should configure the runner to enable it, to speed up the images build process.
  - **Exploit runner API**: the runner pulls available jobs from a GitLab server through a simple HTTP API. We can try to implement an alternative (and lightweight) server that implements these API and that supplies jobs without to have a full GitLab instance.

Thank for you attention!

Slides made with LaTeX, beamer package and custom template.