Service design and engineering
University of Trento a.y 2023/2024
Cereser Lorenzo, 247442

Final project report

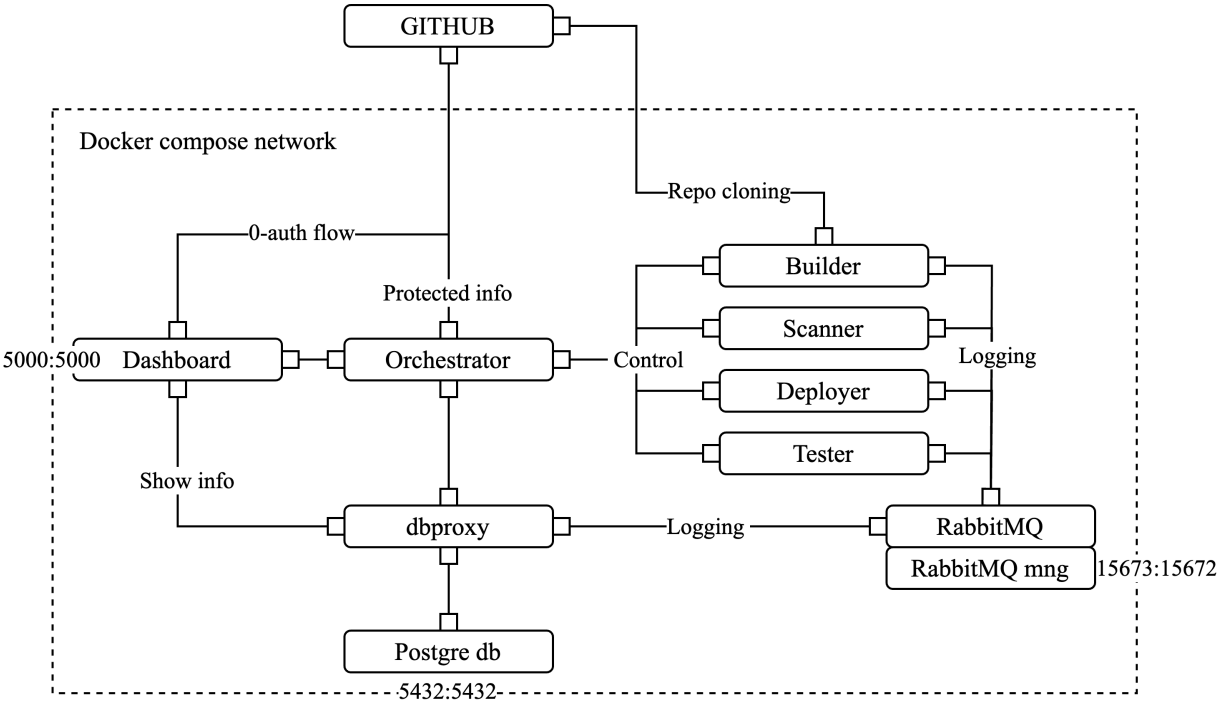
Continuum - a rudimental CI/CD Platform

`https://github.com/lcereser6/SDE-project-Pycontinuum`

1 Introduction

The main goal of this project and its implementation was to create a platform that offered some basic Continuous Integration and Continuous Deployment features. The result achieved is satisfactory in many ways, even though it lacks depth in the development of components out of the scope of this course. The name of the platform is Continuum, it offers the possibility of choosing a repository on your GitHub (containing a Dockerfile), building the image, scanning it for vulnerabilities, running it as a container and testing it. The whole platform is accessed through an interactive front end and it is secured by OAuth 2.0 authentication. The services are all deployed on Docker, orchestrated through Docker-compose and communicate between each other almost exclusively using REST APIs.

2 Architecture



The diagram illustrates a simplified representation of the architecture of the platform. There is only one single point of access to the entire application, being the dashboard. The service running the dashboard also exposes out of the Docker-compose network a port (5000), that allows users to access the application. In addition to that two more ports are exposed for debugging/and monitoring purposes, the first one being the RabbitMQ management plugin web interface, that allows for queue monitoring or debugging, the second for direct access to a DBMS using a client (Datagrip, DBeaver, etc.).
Below a brief description of the various services:

- Dashboard : Point of entrance of the application, handles frontend, data visualization and Oauth interactions.
- Orchestrator : Coordinates the many pahses of the application, managing the other services/container.
- RabbitMQ : Distributed queue used for storing logs in the database
- DBProxy : Proxy for database, action triggers are inserted immediately in the database while logs are instead batched and sent once in a while.
- Postge DB : Open source lightweight database
- Builder : Service that builds the image based on the repository given. Access to repository data is granted through Oauth token.
- Scanner : Service that scan the latest image built on a repository and highlights CVEs (based on Grype)
- Deployer : Service that runs the image as a container
- Tester : Service that tests the container just created (mocked for language agnosticism reasons).
- GitHub APIs (EXTERNAL SERVICE): used to retrieve repositories data and Oauth login.

3 Implementation and Data Models

The implementation was done using python 3.8, and on the main development machine the project was enclosed in a virtual environment allowing for easier dependency management. The repository is structured following a monorepo approach, with a folder for each of the services. Each service owns a requirement file, a Dockerfile and some implementation files. All of the communicating services rely on Flask as web server, some of them follow a basic implementation on a single file, other exploit the flask blueprints and the flask plugin features allowing for code modularity.
Some of the services are also equipped with classes to interact with RabbitMQ queue. The implementation choice of using a distributed queue instead of relying again on REST calls, was done to avoid excessive load on the DBproxy, the latter instead of having to process a write request for each log line, spawns a thread that receives queue messages, and every 15 messages or every 3 seconds writes on the Database, easily managing burst and draught of messages effectively.

The main images hosting the container are the base images containing python, each Dockerfile then manages its own dependencies using pipreqs, a python package that scans the repository in which your scripts are located and generates the requirement file for it, taking also care of possible dependencies conflicts along the way. DBproxy service dependencies were manually handled due to problems with the name of some of the dependencies needed for the Database SDK (binary version needed instead of standard one).
Sensitive variables (API keys, usernames and passwords) are handled through heavy use of environment variables, since it is not possible to securely transmit such data, no demo of the application is provided, but a demonstration will be done at the moment of the examination in class.

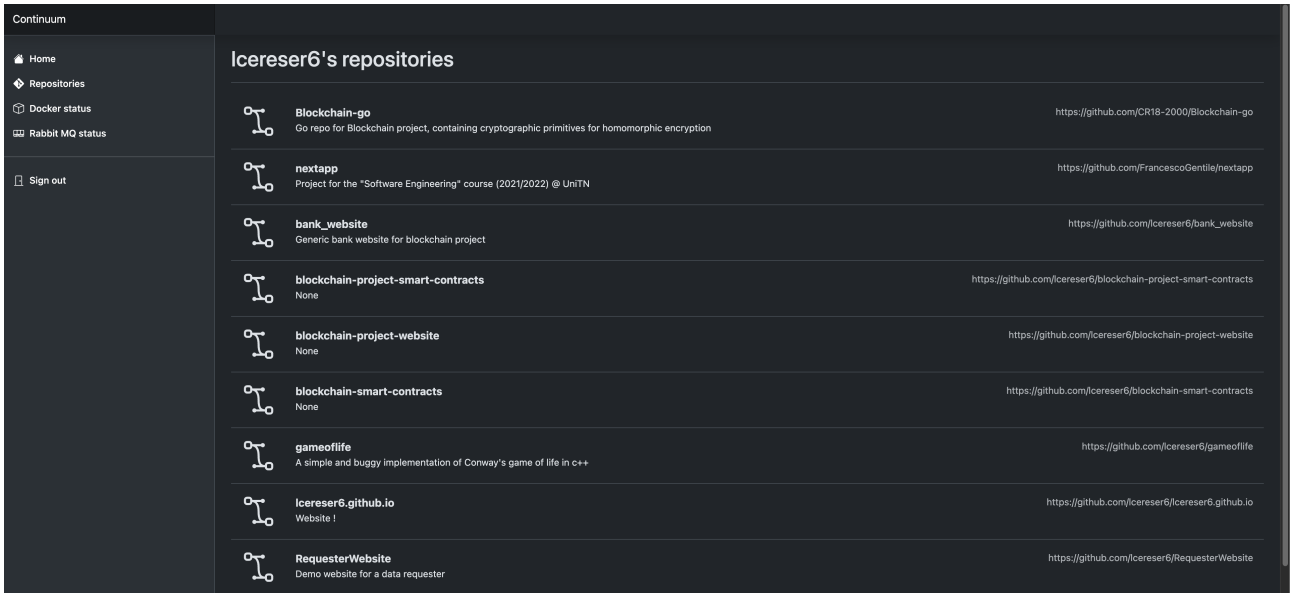


Figure 1: Repository list

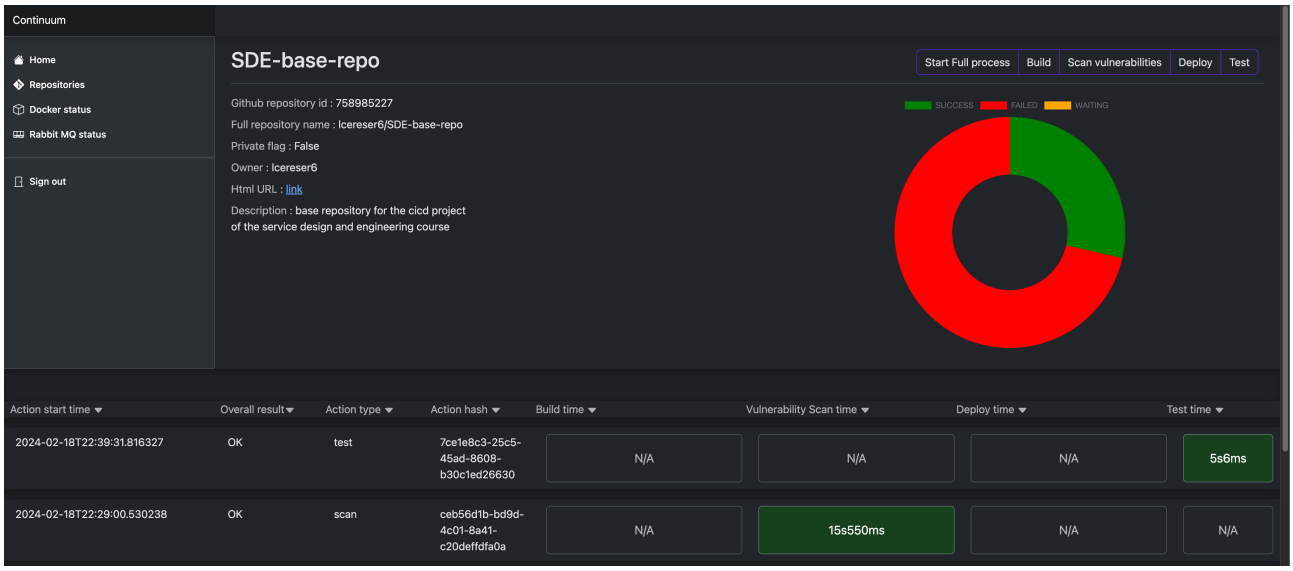


Figure 2: Repository details