Assignment 2: Report

Initial Deployment of ChatHub

28th of April 2024

Authors

103154	João Fonseca
103183	Diogo Paiva
103600	Guilherme Antunes

Professor

Prof. Dr. João Paulo Barraca

Management of Computation Infrastructures - 2023/24

MSc in Informatics Engineering

University of Aveiro

Table of contents

Table of figures

1 Introduction

1.1 Purpose of the report

The purpose of the report is to document the work developed by our group, for the second assignment of the Management of Computation Infrastructures course (2023/24), where we were proposed an initial deployment of the product we chose to operate, ChatHub, our rebranding of TailChat.

1.2 Technologies used

We used Docker to build custom images of the components, and used the DETI registry to store them. To deploy the entire infrastructure – static website and ChatHub components - we used K3s [1], a lightweight distribution of Kubernetes, along with Longhorn [2], a cloud native distributed block storage for Kubernetes, all provided to us by the course professor.

1.3 Code repository

We used a GitHub repository [3] to store the specification of the Docker images and K3s deployments, as well as some scripts to automate the processes of building the Docker images, and deploying the infrastructure. There are README files with some indications on how to use the contents of the repository.

2 Description of the cluster operation

2.1 Diagram

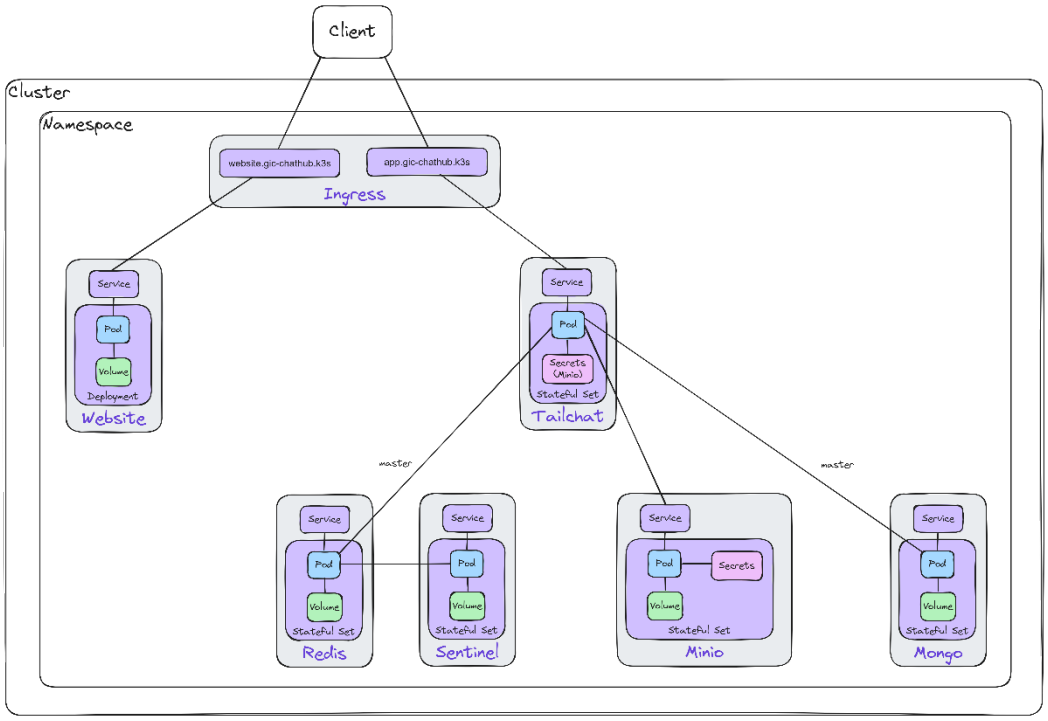

Figure 1. Deployment Diagram on the K3s cluster.

In Figure 1 we can observe the configuration of our deployment on the K3s cluster. The ingress routes traffic for the static website and the application. The application has a connection to the Minio service, and connections to the Redis and Mongo master instance pods.

2.2 Deployment strategy

Since we were working in a shared environment, we defined the tag “gic-chathub”, to help us distinguish our work from others and to avoid conflicts. Following this practice, the Docker images we pushed to the DETI registry were tagged using the prefix “registry.deti/gic-chathub”, and in the K3s cluster we deployed everything under a “gic-chathub” namespace created for the effect. To more easily interact with the K3s cluster, we used Lens [4], a graphical user interface for Kubernetes.

2.2.1 Ingress

The ingress uses Traefik to route requests to our deployment. We defined two different rules, that correspond to the static website and to the ChatHub application itself.

- Rule for host website.gic-chathub.k3s: routes traffic with prefix path / to the static website, to the service port 80.
- Rule for host app.gic-chathub.k3s: routes traffic with prefix path / to the ChatHub application, to the service port 11000.

To allow the translation of these domains into the IP address of the cluster, without using an external DNS server, we added the entry

193.136.82.36 website.gic-chathub.k3s app.gic-chathub.k3s

to our **`/etc/hosts`** file (Ubuntu) or to **`C:/Windows/System32/drivers/etc/hosts`** (Windows). You need to be connected the university's network to be able to access the cluster (either through eduroam in the institution, or using the Check Point Mobile VPN).

2.2.2 Static website

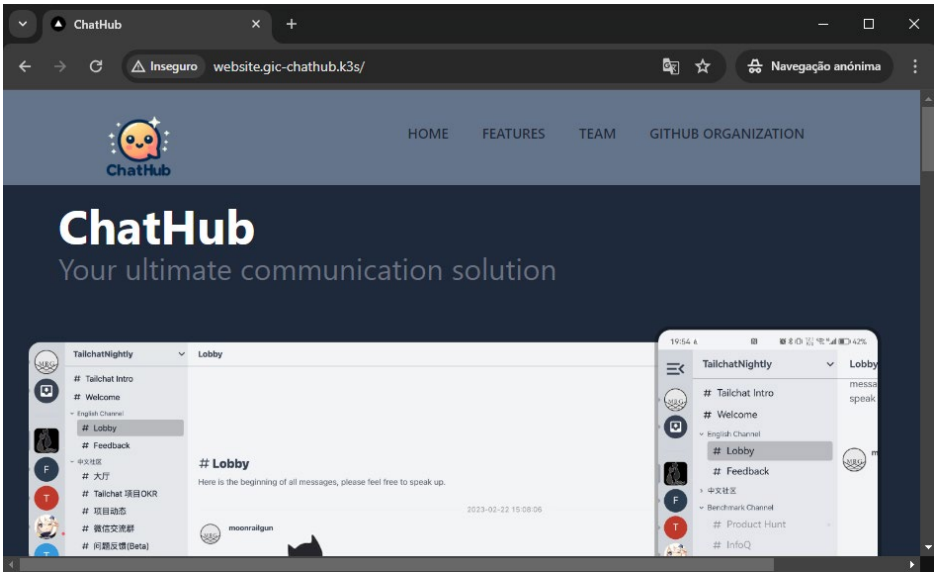

Figure 2. The static website landing page.

The static website was developed by us to present the product we would be deploying. It is accessible through the Traefik ingress on http://website.gic-chathub.k3s, which redirects traffic to the website service on port 80.

The website deployment manages a single instance of our custom website Docker image (registry.deti/gic-chathub/website), that is based on an Alpine Linux image.

To be able to serve static content, we mounted a Persistent Volume Claim (static-pvc) with 10 MiB to the **`/usr/share/nginx/html`** directory inside the container. Our NGINX image has the default configuration, which specifies the **`/usr/share/nginx/html`** directory as the root for serving content [5]. Therefore, all we had to do was build the website's Next.js project, and copy those static files to the NGINX root directory through the website pod.

2.2.3 Minio

Minio in the case of our application serves as a distributed file service, meaning that any multimedia content, such as images or videos, are stored in a bucket.

To build the image of Minio, we are using Alpine Linux as the base. Then, we download Minio from https://dl.min.io/server/minio/release/linux-amd64/minio, give the necessary permissions to the file, expose the server port and the port for the consol, and finally set the working directory.

For the deployment in the cluster, first we set a secret with the user and password for the Minio server. Then, we create a headless service to control the domain within which Stateful Sets are created. After that, we create the Stateful Set with our custom built image and create the service, so that we can access the Stateful Set internally. When we create the Stateful Set, we also mount a Volume Claim Template [6].

2.2.4 Mongo

In the case of our application, the Mongo database serves as the persistent database.

To build the image of Mongo we used Alpine Linux, but an older version than the image we used for the Minio image. This is because Mongo has been removed from Alpine Linux packages due to SPL license. [7] We create the directory for the data, and give ownership of this data to the Mongo user and group. Lastly, we create a volume and expose the ports.

For the deployment, first we set a Service Account, a Cluster Role, and connect the two with a Cluster Role Binding. This will be used for our headless service, that Mongo will utilize when creating DNS association of the replica sets. Next, we create the headless service. Lastly, we set up the Stateful Set in which we also mount a Volume Claim Template, and use the previous created image of Mongo. After all of this, we must do some manual configuration to set up Mongo's replication host. [8]

2.2.5 Redis

Redis serves as the data cache and message transport service for our application.

To build the image of Redis we are also using Alpine Linux, however we had to use the most recent version. This is due to some of our configurations only working with Redis version 7, not Redis 5. Then we add Redis to the image and start the server.

To deploy, we create a Config Map for the configuration of Redis nodes, and then another for the user ACL (list of users that are created). After that, we create the Stateful Sets for Redis nodes and Sentinel nodes, and attach a Volume Claim Template to each. Lastly, we create the headless services for both Redis and Sentinel nodes. [9]

2.2.6 TailChat

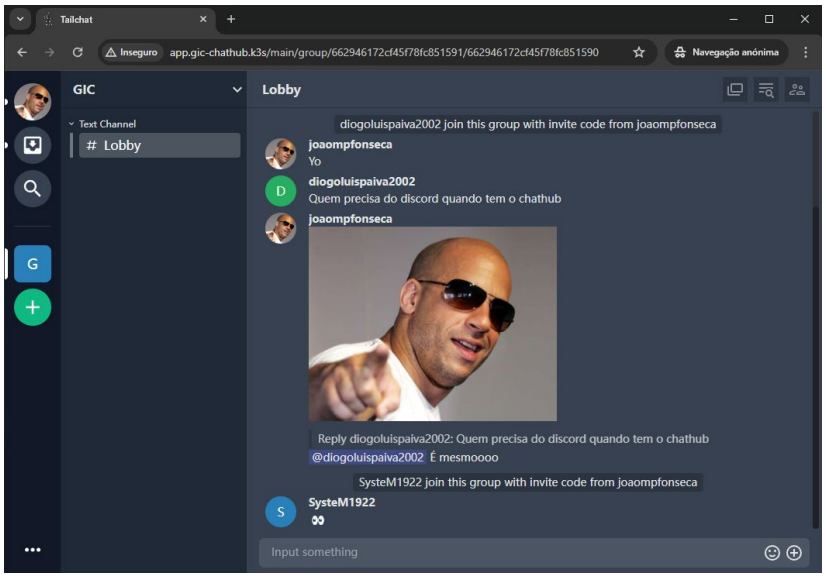

Figure 3. Group text channel in the deployed ChatHub application.

TailChat acts as the point of interaction with the application, and it's responsible for the core services of the system, including the plugin service. Ideally, it would also be responsible for deploying Open API and Admin services into production; however, it was not possible to implement this, due to issues we faced regarding the image build process.

Therefore, due to the build issues, we are utilizing the image "moonrailgun/tailchat", which contains the above mentioned services. We did not assign any volume to this Stateful Set, and as recommended by the developers, we set minimum and maximum resource limits for CPU and memory usage.

The TailChat service connects to other services through internal cluster connections, using Redis as transporter and data cache, Mongo as the database, and Minio as the object storage system. Connections to Mongo and Redis are made directly to the master pod, while a connection to Minio is made directly to the service. This decision is explained by the current inability to automatically re-establish the connection to the new master when the previous one goes down. To establish the connection to Minio, we also need to specify the access data to the service, namely the user and password defined during the deployment of the Minio service.

Finally, we map the path `/health` to provide information about the node's status, and port 11000 as the access port to be covered by Traefik.

2.3 Bottlenecks

In the case of the website, that just serves static information about ChatHub, there isn't a high risk of a bottleneck. Our only NGINX based instance should be more than enough to handle the traffic going to the static website, since the only interaction users can have with it is fetching the page.

In the case of the databases if there are multiple reads and writes at the same time, there can be a bottleneck, since with Redis and Mongo, we are performing direct connections to the master pods (due to not being able to write in the slaves). We look forward to working around this issue in the next delivery.

3 Analysis of strategies and configurations

As a way to optimize the deployment of ChatHub, we defined some strategies and configurations for each of the involved services. In the following table, we mention decisions we took and the respective justifications.

Decision	**Justification**
Storing static website contents in a Persistent Volume Claim (PVC).	Keeps static files from outside the container, therefore making the Docker image smaller. Updating the contents of the static website only require changes in the PVC, are instantly reflected on new requests to the website, and don't imply any downtime.
Storing the credentials for Minio in a secret in the deployment.	Kubernetes secrets are designed to store sensitive information, such as passwords, OAuth tokens, and ssh keys. They are encrypted at rest and can be decrypted only by the Kubernetes API server, ensuring that sensitive data is not exposed in plain text.
Not using operators in Mongo (doing a manual configuration instead).	Operators involve all the cluster and so it can alter some configurations of the other groups. Also, manual configuration allows for greater customization and control over the Mongo deployment.
Using Sentinels for Redis.	Sentinels provide a high availability (HA) setup for Redis. They monitor the health of the Redis master and slave instances, ensuring that the system remains operational even if the master node fails.

4 Evaluation of the cluster operation

As a way to validate that our first basic deployment was working as intended, we performed some manual testing of the static website and of the application itself. In terms of the static website, the only test we performed was seeing if we could access it – we can confirm it works by observing Figure 2.

When it comes to the application, we also tested the core functionality – chat – by creating a group, inviting members to it, and exchanging a few messages, including multimedia content like images. The result can be seen on Figure 3. However, we also wanted to check out how the components behind the application were interacting within the K3s cluster. Therefore, we performed some in-depth testing of the core components.

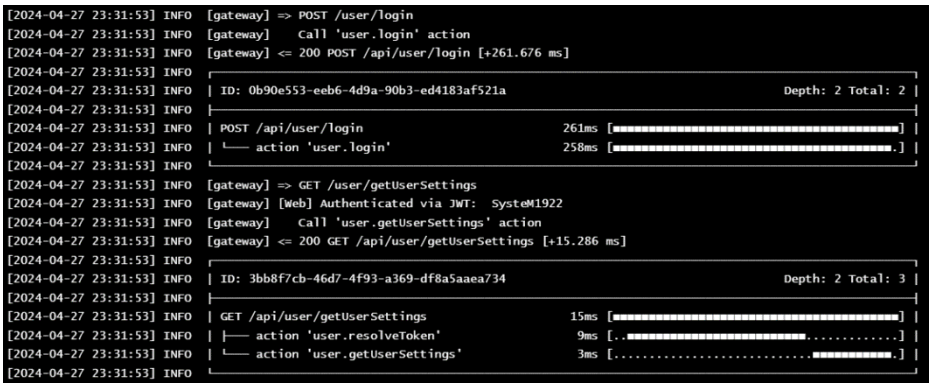

Figure 4. TailChat login action logs.

In Figure 4, we are able to see what happens when a user performs a login to the application. The login request is processed, and then the user settings are retrieved.

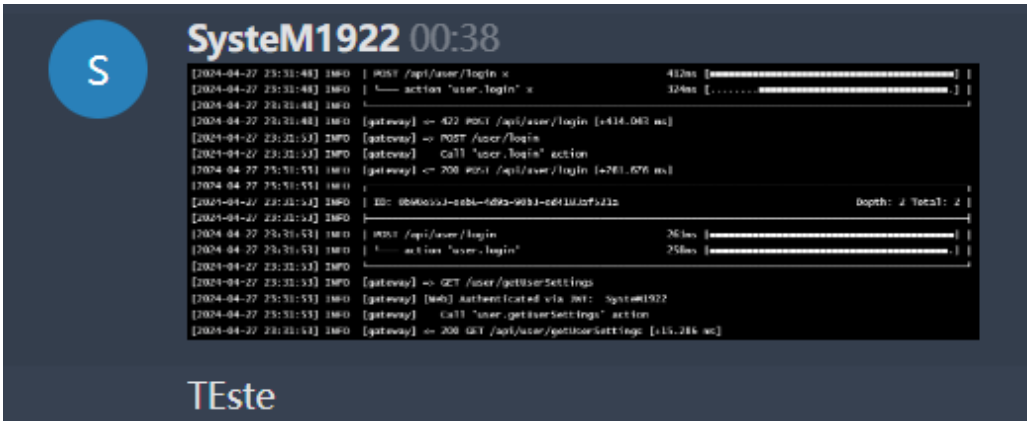

Figure 5. Sending an image and text on a text channel.

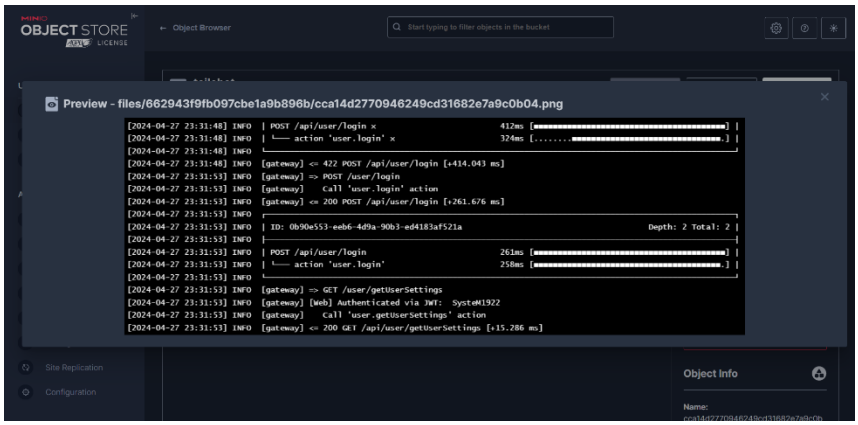

Figure 6. Image stored in Minio.

Figure 7. Text stored in Mongo.

We checked out how images and text were stored, in Minio and Mongo respectively, by sending a message with an image and another with some text (Figure 5). Using Minio's console [10], we were able to access the objects stored in Minio. We were able to find the image we sent on the message (Figure 6). As for the text message, we checked out Mongo to find the object associated with it. By reading the fields, we can see that the message contains information such as its author, the group and text channel it was sent on, mentions or reactions it might have, and when it was created and uploaded to Mongo (Figure 7).

Figure 8. Redis functioning as a transporter and data cache.

In Figure 8, we can see how Redis handles the task of transporting messages and performing caching of the interchanged data.

5 Conclusion

In this assignment, we were able to deploy an existing application using the Docker technology and a K3s cluster. We learned the importance of building our own Docker images, with the smallest number of external dependencies, to reduce the probability of future security compromises, and also trying to keep the image size small and highly decoupled from static contents, by the use of volumes attached to some directory inside them.

Regarding the deployment on the K3s cluster, the use of YAML files to define the infrastructure allowed for an organized and quick deployment process. Whenever there was a change that needed to be done, we only needed to apply the deployment again for it to take effect. In terms of operation, the fact that running pods are automatically restarted in case of failure contributes for the availability of the system. This is also noticed when a deployment is updated, running pods are slowly replaced by newer versions.

The use of Lens was really convenient, since it allowed for an easy debugging and testing of the deployment process in its multiple stages. Having quick access to pods and logs revealed to be essential for a successful deployment of our system.

Setting up replicas for Redis and Minio revealed to be more challenging than anticipated, however we are satisfied with the end result. We look forward to improve their configurations in the next delivery.

References

[1] “K3s,” [Online]. Available: https://k3s.io/.

[2] “Longhorn,” [Online]. Available: https://longhorn.io/.

[3] G. Antunes, D. Paiva and J. Fonseca, “GitHub Repository - Deployment 1,” [Online]. Available: https://github.com/GIC-Assignment-ChatHub/deployment1/.

[4] Mirantis, “Lens,” [Online]. Available: https://k8slens.dev/.

[5] R. Nelson and A. F. Garcia, “Deploying NGINX and NGINX Plus with Docker,” NGINX, [Online]. Available: https://www.nginx.com/blog/deploying-nginx-nginx-plus-docker-2/.

[6] “Kubernetes Examples,” [Online]. Available: https://github.com/kubernetes/examples/tree/master/staging/storage/minio.

[7] “Unix & Linux,” [Online]. Available: https://unix.stackexchange.com/questions/568530/installing-mongodb-on-alpine-3-9.

[8] “Medium,” [Online]. Available: https://medium.com/geekculture/installing-mongodb-on-kubernetes-with-replica-sets-and-no-mongodb-operator-ed8d7f3bb2d1.

[9] “Freedium,” [Online]. Available: https://freedium.cfd/https://medium.com/faun/redis-high-availability-with-sentinel-on-kubernetes-k8s-a1d67842e0ce.

[10] Minio, “Minio Console,” [Online]. Available: https://github.com/minio/console.