Report 1 - Initial Deployment

Gestão de Infraestruturas de Computação

2023/2024

Filipe André Seabra Gonçalves, 98083
João Pedro Saraiva Borges, 98155
Vicente Samuel Gonçalves Costa, 98515

Universidade De Aveiro

Index

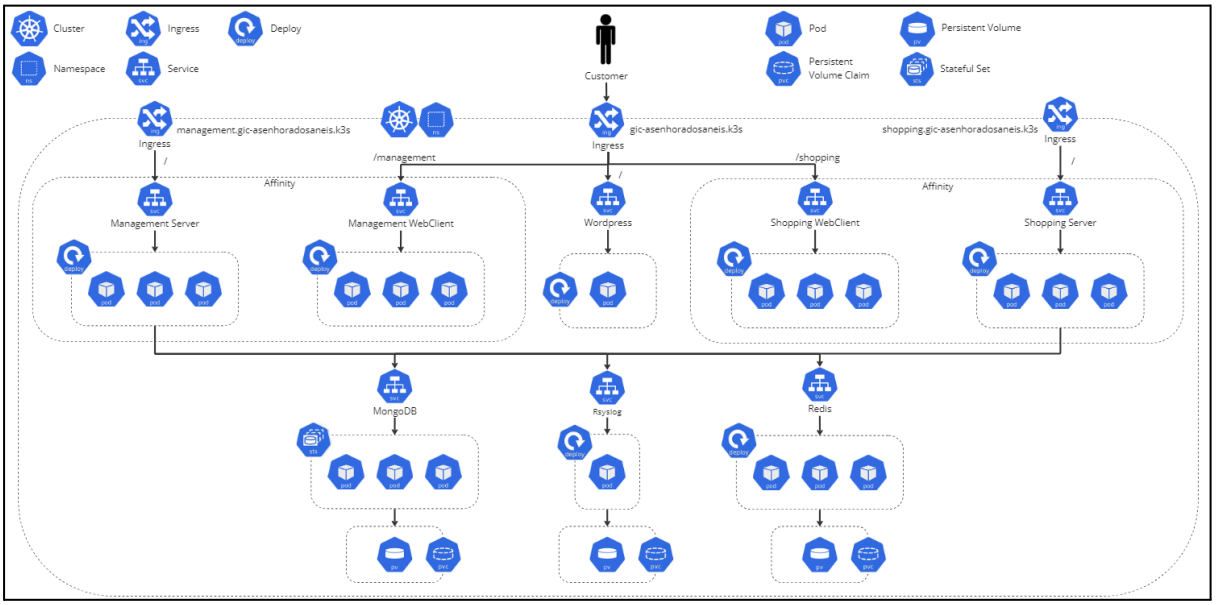
1. Contextualization

Our work, **A Senhora dos Anéis**, is a reliable e-commerce solution, based on an open source and self-hosted project called **Veniqa**, composed of a stack with frameworks such as **Redis**, **MongoDB**, **Vue.js**, **Node.js**, and many others, with scalable and flexible resources.

Our objective is to compose a cluster of deployed *kubernetes* services that can be managed, monitored and available almost all the year around, with low downtime, auto scalable and auto deployable.

2. Cluster Operation

Initially, our *kubernetes* cluster would take this form:



Each service will be explained in more detail in "**3. Configurations and Strategies**", however the most important parts are the following:

- Inside our **cluster** we have our **namespace**, configured with the name **gic-asenhoradosaneis**, which englobes all the services of this deployment
- The customer will access our services by requesting to the **Ingress** controllers, which will later transfer them to the service they want to access
    - There are only five services which the customer can access via **ingress**, the management's server and webclient, shopping's server and webclient, and our wordpress website
    - Each website can be accessed via an **Ingress** controller with *http* hostname "**gic-asenhoradosaneis.k3s**"
    - The management's server can be accessed via an **Ingress** controller with *http* hostname "**management.gic-asenhoradosaneis.k3s**"
    - The shopping' server can be accessed via an **Ingress** controller with *http* hostname "**shopping.gic-asenhoradosaneis.k3s**"
- Each server is also dependent on both databases and our logging service
- Each database and logging service are connected to a **persistent volume claim** which consequently contains a **persistent volume**

As we can see, replication is a fundamental part for having our services continuously available. As such we decided to have **three replica sets** delivering the same service, in every service that seemed fit, so that even if one of the accessing points is not working, another one will be.

Ensuring an odd number of replicas minimizes the likelihood of two replicas being accessed simultaneously by the same customer. This setup ensures that there's always a *master* to manage request distribution in services with a *master-slave* relationship. Additionally, the odd number of votes ensures that consensus is always reached.

2.1. Deployment Strategy

From the many strategies invented, we believe that applying **rolling deployment** would be the most optimized and safe option.

As the name suggests, this strategy slowly replaces the pods of the previous versions with the pods of the new version of the application, one at a time, in a rolling fashion, allowing the old pod versions to work while the new ones are down for updating. A rolling deployment typically waits for new pods to become ready via a readiness check before scaling down the old components.

Although errors can be propagated while implementing a new version, we can still correct them individually and not have downtime in the services, as one of the two gateways would be serving all the requests made by the user while the other would be fixed, which makes it easier to rollback the versions in the updated pods.

The **blue-green deployment** also seems fit for our case. This strategy involves running two versions of the same application at the same time, and whenever a new version passes all the necessary checks, and moving the traffic from the older version (the green version) to the newer version (the blue version). This makes it easier for the developers to deploy and test the services completely and give the safe and tested gateways to the users. Any rollbacks necessary would always be fast and reliable, without impacting the main production deployment.

However, this strategy will not be implemented as the amount of traffic this platform will have will not be big enough to compensate for the higher resources cost and human power.

2.2. Bottlenecks

There are many bottlenecks that we will eventually have to face in our project. The most troublesome is the scalability of the load balancer. In certain seasons the selling of the product might get more attention, either because of publicity done in broad social networks or because of promotions done in specific periods like black friday, summer sales, etc.. Our structure is not strongly made for huge amounts of visits everyday, as the lowest views expected are around 10 views to the site every day.

Another challenge we're currently facing is that our **Redis** and **MongoDB** deployments rely on Docker Hub images. This introduces a dependency within our project's infrastructure, posing a potential point of failure that can't be resolved by merely adding more replicas or resources.

We are also currently working with applications made using **Node.js** when it was in their “early” versions, which makes deployment nowadays more complicated and difficult as some packages and functionalities may be deprecated. Our management and shopping servers demanded some refactoring as the version they should be running would be **Node.js 11**, but because of some packages we had to refactor them to **Node.js 14**. Each refactoring of the main open source code is a risk to any functionality that might be lost or corrupted.

3. Configurations and Strategies

The deployment configurations were mainly configured in the *deployment* folder, which can be accessed here.

Each service, or group of similar services, belongs to a singular deployment file, as it is easier to use the **rolling deployment** strategy when updating a service. It also makes it easier to organize and understand the deployment files and the deployment process.

We also developed a script for building all docker images, pushing them to the class registry and deploying them to our *kubernetes* namespace. It also comes equipped with an option to delete all the docker volumes, which can be useful when trying to have a fresh build - accessed by using the flag “-d 1”.

We will walk through all the services, deployments, stateful sets, persistent volumes and secrets, and explain the motives behind their creation and configuration.

3.1. Redis

Redis is used as a cache database for the management and shopping servers. As **Redis** is a stateful application, the service was also established with the **Stateful Set** property, which means that the instance requires stable, unique identifiers (maintaining the network identity as “hostname” in our configuration) and persistent storage, crucial for data durability and consistency.

It’s being deployed with **three** replicas as it won’t have a single point of failure. It mounts one volume, a **Redis** persistent data storage, which saves all the cached data, and one secret for basic redis configuration.

If the **Redis** ends up failing or having an error, it has a restart policy to always restart when it happens. There is also an established templated container for easier replication, isolation and portability.

The persistent volume claim (**PVC**) should have an access mode of read/write once, which permits only one deployment at a time to access it and do the operations; it is also specified that the storage class is *longhorn*. However, because we don’t have high availability yet, with our **Redis** service not having a *master-slave* relationship, we had to change the access mode to read/write many.

This **PVC** makes the data persistent even if the **Redis** service fails or is turned off.

3.2. MongoDB

MongoDB has the purpose of saving persistent data all over the system, like shopping items, users carts, login sessions, etc..

As **MongoDB** is a stateful application, the service was also established with the **Stateful Set** property, and, just as **Redis**, will have **three** replicas and a template container for isolation, consistency and repeatability in pod creation. The changes made to the template will reflect in all **MongoDB** pods because of the **Stateful Set**, ensuring uniformity and simplifying management.

There is also a **PVC** connected to it to make data persistent with an access mode of read/write many, as **MongoDB** still doesn’t have a *master-slave* relationship.

3.3. Traefik and Middleware

The **Traefik** is used as a *reverse proxy* to redirect requests to the correct service, serving as our **Ingress** controller. Based on the path of a request and their *hostname*, a different service will be redirected to the user.

We used different annotations, which provide additional configuration to the controller, which we list them below:

- Allowing both *http* and *https* request to the **Ingress**
- Always redirecting *http* traffic to *https*
- The redirection is always permanent
- Preserve the original hostname in the redirected request headers

We decided to have three **Ingress** controllers as the management and shopping servers need one controller each for the other services to make their requests. Every **Ingress** controller is the same, with some differences in hostname, but the website **Ingress** controller also has a new annotation:

- Associate a middleware to modify the request by removing certain prefixes from the URI and deliver it to the correct destiny by matching it against the request URI

Some rules are also defined for the different *http* paths when a request is made by the user - for the static website we decided to use an exact pathtype, so only requests to "/" would redirect to the website and for the rest of the paths, we use a prefix pathtype, so that the websites can also access their static files in a subpath, or the servers can access their different functionalities.

3.4. Wordpress

Our static class website was created using **Wordpress**, which is a web content management system, where a user can create a new website, edit and style it and deploy it in a wordpress domain. This service was deployed by simply creating a **cherrypy** python application that returns the *html* file of the website.

3.5. Shopping WebClient

This will be the main interface for the user, it will contain the **Veniqa** store webpage as a template which was done with **Node.js** and **Vue.js**.

The service's name is **shopping-webclient** and has an image pull policy that dictates that the latest version in the registry is always used above the others. There are also some restrictions in the resources to the storage and the CPU:

- Only using 1/1000th of a CPU core
- Not surpassing 100 mb of storage

The service will listen to the port 80, which is connected to the redirect from the **Ingress** controller, and will be forwarded to the port 8080, which is the port inside the pod where the service is executing.

The dockerfile contains a base image derived from version 14 of **Node.js**, an installation and setup for the *http server* package, a working directory definition, the dependencies file and the volumes folder for data persistence, it also builds the application

and runs it. It also copies assets, one by one, into the /var/static folder, and finally runs the *http server*.

This service also has an affinity rule with the **shopping-server** service, which means that wherever a **shopping-server** pod is, a **shopping-webclient** pod also is, minimizing the time of a request between pods.

3.6. Management WebClient

This will be the main interface for the manager, which can check how the operations are going and change some elements on the shopping interface, it is also part of the **Veniqa** template and was done with **Node.js** and **Vue.js**.

It is highly similar to the shopping webclient, the only things that differ from them are the identification, labeling this service with the name **management-webclient**.

This service also has an affinity rule with the **management-server** service.

3.7. Shopping Server

This will be the server in which the **shopping-webclient** will make their requests for security, data, etc.

This service is named **shopping-server**, it will have **three** replicas, and will listen to port 80, which is connected to the redirect from the **Ingress** controller, and will be forwarded to port 3000, which is the port inside the pod where the service is executing.

Within its configurations, this service will mount a secret to provide sensitive configuration data securely, containing information regarding database connection, database port and host, replication mode, service mode (deployment or local), etc., which can be accessed by the secret **shopping-server-secret**.

The dockerfile contains a base image derived from version 8 of **Node.js**, an installation of dependencies, containing libraries for handling *http* requests (*axios*), database connections (**mongoose** and **Redis**), session management (*express-session*), authentication (*bcrypt-nodejs* for securely hashing passwords before storing), email sending (*@sendgrid/mail*) and logging (*winston*).

This service also has an anti affinity to the **shopping-server** service, which means that two **shopping-server** pods will never stay together.

3.8. Management Server

This will be the server in which the **management-webclient** will make their requests for security, data, etc.

Just as before with the webclients, the servers are also really similar, so it basically has the same configuration, it only changes in questions of name, accessible ports, and secret name and values.

This service also has an anti affinity to the **management-server** service.

3.9. Rsyslog

Rsyslog makes logs out of every operation made inside of the *kubernetes* cluster.

Its service enables other components within the cluster to send logs to the syslog server. This service listens on port 5000 and redirects traffic to port 5140 of the **Rsyslog** pods. It also has a persistent volume claim with storage of 50 MB so that even if the cluster is resetted, the data will persist, it has a readwriteonce access mode so that each node writes on it one at a time.

3.10. Secrets

Inside the */deployment/secrets* folder, there is a script file created by us to establish all the secrets for the services. This script utilizes *kubectl* commands to create *kubernetes* secrets.

We define all the secrets for the **shopping-server** and **management-server**, including the database connections strings, environment variables and authentication credentials, and also define some secrets needed to run the **Redis** service.

4. Evaluation of the Cluster Operations

In terms of applications functionality, most of the services inside of this cluster are correctly working with no problems whatsoever, be it in accessing web pages, or connecting to the databases.

However, the servers currently have a bug that when a request has been made, the server returns *502 Bad Gateway*. The origin of the problem is within the server's **Node.js** application. The project works fine locally, but after a very long time debugging we still don't have an evident solution.

Our cluster has a reasonable resource utilization, with average *cpu* cores usage around 0.02 in most of the services, with the **MongoDB Stateful Set** using 0.010 on average, for each replica. In terms of memory, the webclients only occupy around 20 to 27 Mib, whereas the servers occupy almost 850 Mib each replica.

The **persistent volumes** are only 50 Mib and 30 Mib.

We also tried to scale the performance by changing the workload, which resulted in a more than average consumption, but still similar to the values given before.

Even if a service fails or is killed, the cluster will not have downtime and will keep on running, even if slower.

Overall, we believe that in theory, the implementation we made is optimal, reliable and secure so that it becomes difficult to be able to have downtime in our cluster. However, because we couldn't test all the functionalities, mostly the servers requests, we can't be sure that it will be as said.

Finally, to use our hostname in our services, we had to add the three hostnames from our **Ingress** controllers in the */etc/hosts* file in our system:

```
193.136.82.36      gic-asenhoradosaneis.k3s
193.136.82.36      management.gic-asenhoradosaneis.k3s
193.136.82.36      shopping.gic-asenhoradosaneis.k3s
```

5. Future Work

As for the next steps in our deployment, making our services Highly Available (**HA**), which consists of making the **MongoDB Stateful Set** replicas into a *master-slaves* relationship, where only the *master* can write into the **PVC** and the slaves can only read from the **PVC**, and creating a cluster of replicas from our **Redis** pods.

We also intend to make our servers work and finally test the requests and the system as a whole.

Finally, We aim to resolve the dependencies in the Docker images of MongoDB and Redis by creating our own custom Docker image.

6. References

Top 5 Most-Used Deployment Strategies
Advanced Deployment Strategies - Deployments | Developer Guide | OpenShift Dedicated 3
Deployment Strategies - Deployments | Developer Guide | OpenShift Dedicated 3