DailyENews

1st Pedro Amaral
DETI
U. Aveiro
Aveiro, Portugal
NMEC: 93283

2nd Pedro Tavares
DETI
U. Aveiro
Aveiro, Portugal
NMEC: 93103

6th Vasco Sousa
DETI
U. Aveiro
Aveiro, Portugal
NMEC: 93049

***Abstract*—Before a platform can go online there is always a process to determine what the project's architecture should be and what steps should be taken to present users with a stable and flexible platform that can handle most scenarios in the most effective way possible.**

***Index Terms*—deployment, scalability, management**

I. Introduction

For projects to be online first comes the process deployment which comes with a variety of decisions, from the architecture to follow, to scalability, to a solid catastrophe recovery plan. To do so, we must think ahead about the possible bottlenecks present and how to prevent those bottlenecks from causing problems or at least reduce those problems from occurring. For us to get a feel for this process we took advantage of the already existing project JARR (Just Another RSS Reader).

II. Product Description

DailyENews is a platform that allows users to gather news from different selected sources, through their RSS feeds based on JARR. Besides gathering data from different sources, it is also possible to group different sources through the use of categories and add articles of interest to a favorites section, so that the article is more easily accessible. A user selects which sources it wants to fetch from and according to those sources, the user's feed is built accordingly. To allow users to perform these operations there is underlying architecture with a few structured key components as shown in Figure 1.

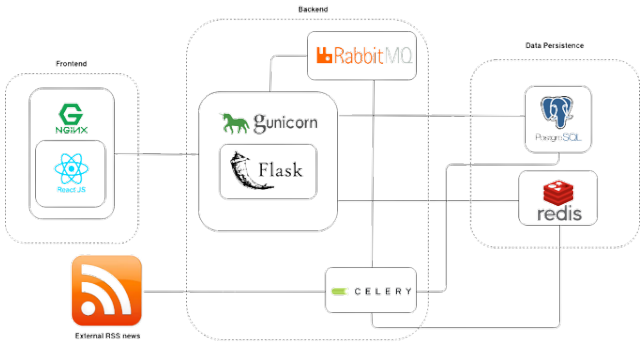

Fig. 1. Project architecture.

III. Component Description

When it comes to the components, the ones that make up the project are the following:

- React: React is the framework used to display the contents to the user.
- NGINX: Platform used to serve the static content which corresponds to the react application build and the promotional website in our project.
- Flask is responsible for the project's API which connects the frontend to the rest of our application handling communication with the persistence layer, data processing, performing requests, and returning the information relative to such requests.
- Gunicorn allows to balance the workload among different instances. This is done by creating multiple copies of the component that requires higher processing power. In our case, gunicorn manages the number of flask instances running simultaneously.
- Celery is a worker. Its job is to fetch all the news from the external RSS sources added by the users and store them in the database.
- RabbitMQ works as a message-broker whose job is to send the tasks from the flask server to the celery workers.
- PostgresSQL is our database and therefore the component that stores data permanently.
- Redis is used to store metrics about the utilization of the entire application.

IV. Redundancy Deployment Strategy

When it comes to redundancy we opted to implement replicas to every component of the architecture. The frontend and the API have 2 replicas each, which allows the application to answer requests even if one of the replicas crashes. Additionally, the worker has 4 replicas so besides redundancy we have more replicas to deal with those requests. When it comes to the database, Redis and Rabbitmq, the replication cannot be done simply by having 2 replicas, since these are stateful. The replication for these components must have odd replicas to reach a consensus. This is mandatory because if the configuration has an even number of replicas the decision on the current state may conflict and when it does, a vote of n to n cannot be decided and the different replicas will keep different data.

V. Scalability

Usually, scalability refers to the ability of a system to handle unpredictable and higher workloads with uniform performance. Usually, this performance will depend on the CPU, RAM, disk, and memory load. To provide a smooth experience to the users, the group considers it mandatory to explore how the state-of-the-art industry scales its applications. After some research, the group found that nowadays, solutions that take advantage of artificial intelligence are widely used. The algorithm is trained with users' usage data and CPU, Disk, RAM, and network load. Using artificial intelligence to solve this problem is out of the scope of the subject, so, another more conventional method is needed.

Factor	Processor	Memory	Disk Space	Network
Total Users			X	
Active Users		X		
Concurrent Users	X	X		X
Total Workers	X			X

The behavior of user usage of the application is going to be very similar for all due to the few functionalities provided. So, the group decided to analyze the resources' usage of 1 concurrent and multiply by the total number of concurrent users. Furthermore, users tend to read news in the morning before work and before going to the bed, so during this time frames the application will have a higher workload. Some kind of time-based scaling will probably help. Instead of scaling resources vertically, to handle more load, using Kubernetes, the application can be scaled horizontally. In other words, instead of upgrading the CPU of one machine or adding more ram, we add a new machine and balance the load between the existing ones. As Table 1 shows, each factor impacts each resource differently, so it's wise to make a 1-by-1 analysis to understand how it's possible to scale more efficiently.

- The total number of users represents the number of people that used the application at least once. The main impact of this factor is in the disk space because more rows are going to fill in the database. To mitigate potential problems due to lack of disk space, it's important to have enough disk space for at least twice the database size. In a more advanced scenario, solutions like sharding could be necessary.
- The number of active users represents the number of people that are using the application but are not doing any requests at the current moment. The main impact of this factor is in the RAM space because objects representing the session or other temporary memory are still cached on memory. To mitigate potential problems related to low RAM available, cache mechanisms should adopt at least one cache invalidation policy.
- The total number of concurrent users represents the number of people that are using the application and doing requests at the current moment. The main impact of this factor is in the CPU usage alongside RAM and network. To mitigate potential problems related to this, it's wise to create multiple servers, and load balances the traffic between those. In this case, this project uses Kubernetes, so the traffic is going to be balanced between pods. The number of pods can be increased or decreased on demand.
- The total number of works users represents the instances of the application that is used to fetch the news from the RSS feeds provided by each user. This part of the software is the most CPU-intensive one, so it's wise to have independent and isolated servers that perform the requests. Using this strategy, potential problems related to CPU usage won't affect application performance. Each server will be capable of running one worker and each worker is only capable of fetching 1 RSS feed concurrently.

VI. Dockerfiles Configuration

To deploy our application, a docker image was needed for every component. The Postgres, Rabbitmq, and Redis images are fetched from docker hub according to the original project (JARR) necessities. For the components-specific of our application (client, server, and worker), a set of Dockerfiles was used to generate those images.

- **client** - Used to build the application's frontend docker image. It initially uses the NodeJS image to build the React application and then copies the built project into an NGINX alpine image along with the promotional website.
- **pythonbase** - Used to build an intermediary image with the common parts of the server and worker docker image. It uses a python 3.9 alpine image as a base and installs all the system dependencies required alongside the python packages. It also adds a new user to run the application reducing the risks compared to running using root.
- **server** - Used to build the application's backend docker image. It uses the python base image described before as a base, adds the gunicorn configurations, and is set to run a script that creates the database tables if they don't exist and then runs the flask application using gunicorn.
- **worker** - Used to build the application's worker docker image. It also uses the pythonbase image described before as a base but then it is only set to run a celery worker.

It is worth noting that the original project (JARR) already included Dockerfiles but these would result in images containing several unused dependencies increasing their size considerably and the website was also served using a NodeJS tool more suitable for the development phase. In our images, these problems were fixed by using NGINX to serve the website instead of the previous tool and making all of the necessary modifications to build them from significantly smaller base images (python:3.9-alpine and nginx:alpine).

VII. Kubernetes Configuration

Each component of the application contains the Kubernetes configuration separated into Services, Config Maps, and Deployments. Furthermore, there is also a Persistent Volume Claim and the Traefik Ingress configurations. Services are an abstract way to expose an application running on a set of Pods as a network service. Pods have their own IP addresses and a

single DNS name for a set of Pods, and it's possible to load-balance across them. Config Maps are an API object used to store non-confidential data in key-value pairs. Pods can consume Config Maps as environment variables, command-line arguments, or as configuration files in a volume. This allows the decoupling between environment-specific configuration and the container images. Finally, the Deployment provides a declarative way to modify the state of the Pods and Replica Sets.

- **client-cluster-service.yaml** - Represents the set of Pods that serve the frontend application.
- **client-config-map.yaml** - Contains the Nginx configuration for the frontend.
- **client-deployment.yaml** - Contains the frontend deployment. It makes use of the custom client docker image referred to in the section above. This file also specifies the port that the client is running on, 80 in this case, and the number of replicas. Finally, the restart policy is also specified as "always" alongside the pod anti-affinity that tells the scheduler to avoid placing multiple replicas in a single node.
- **postgres-cluster-service.yaml** - Represents the set of Pods that run the database engine of the application, in this case, it's PostgresSQL.
- **postgres-deployment.yaml** - Contains the database deployment. This makes use of the official PostgresSQL docker image. This file also specifies the port that the database is running on, in this case, 5432, and the number of replicas. Finally, the restart policy is also specified as "always". Furthermore, this also mounts the Persistent Volume named "Postgres-pvc" to have persistent disk space available to store the database contents.
- **postgres-pvc.yaml** - Represents a piece of storage in the cluster that has been provisioned using Longhorn. In this case, it requests 100Mi and names the volume as "postgres-pvc".
- **rabbitmq-cluster-service.yaml** - Represents the set of Pods that run the queue system of the application, in this case, it's RabbitMQ.
- **rabbitmq-deployment.yaml** - Contains the queue deployment. This makes use of the official RabbitMQ docker image. This file also specifies the port that the queue is running on, in this case, 5672, and the number of replicas. Finally, the restart policy is also specified as "always".
- **redis-cluster-service.yaml** - Represents the set of Pods that run the caching engine of the application, in this case, it's Redis.
- **redis-deployment.yaml** - Contains the caching deployment. This makes use of the official Redis docker image. This file also specifies the port that the caching is running on, in this case, 6379, and the number of replicas. Finally, the restart policy is also specified as "always". Furthermore, this also mounts the Persistent Volume named "redis-pvc" to have persistent disk space available to store data.
- **redis-pvc.yaml** - Represents a piece of storage in the cluster that has been provisioned using Longhorn. In this case, it requests 100Mi and names the volume as "redis-pvc".
- **server-cluster-service.yaml** - Represents the set of Pods that run the server of the application, in this case, it's a Flask application run using gunicorn.
- **server-config-map.yaml** - Contains the DNS configuration for the server and the worker. It will be set to overwrite the "/etc/resolv.conf" file in all of their instances.
- **server-deployment.yaml** - Contains the server deployment. It makes use of the custom server docker image referred to in the section above. This file also specifies the port that the server is running on, in this case, 8000, and the number of replicas. Furthermore, the restart policy is specified as "always" alongside the pod anti-affinity. Finally, the server-config-map is injected to configure the DNS and the gic2-server-secret is injected to configure the connections to the other components and other configurable parameters as needed.
- **worker-deployment.yaml** - Contains the worker deployment. It makes use of the custom worker docker image referred to in the section above. This file also specifies the port that the worker is running on, in this case, 8001, and the number of replicas. Furthermore, the restart policy is specified as "always" alongside the pod anti-affinity. Finally, the server-config-map is injected to configure the DNS and the gic2-server-secret is injected to configure the connections to the other components and other configurable parameters as needed.
- **traefik-ingress.yaml** - Represents the application ingress provider configuration (Traefik). This manages external access to the services in a cluster, in this case using HTTP. Besides that, the ingress may provide load balancing, SSL, and name-based virtual hosting. In this case, the group decided to create two different rules, one for the frontend (app-gic2.k3) and another one for the api (api-gic2.k3s). Using two rules is mandatory otherwise some request paths would conflict.

VIII. Issues During Implementation

When it comes to our implementation we faced some issues:

- When it comes to Redis, we faced an error regarding the transition to Redis Sentinel. The reason behind the issue is that the Flask API implemented was not suitable for Redis Sentinel, since it would not always send requests to the master node. Instead, the API would send requests to any replica at random. This is a problem because Redis' default configuration for slaves sets them to be read-only. After potentially fixing this part by altering the server code we faced a second problem with the implementation of Redis Sentinel. The issue was with the Prometheus library which, to get statistics would, just like the API, make requests to slaves instead of the

master. A second approach was attempted which made use of Redis in cluster mode but the issues remained, it would also be necessary to alter the application code and the code of the library. This issue is still ongoing so in the Kubernetes cluster there is still only one replica of Redis but all configuration files developed for both of the referred attempts are available in the delivered files.

- When it came to Postgres and RabbitMQ there was relatively less documentation on how to make replicas with these components. Although there were attempts to implement this replication, a problem would always be found and no solution on the internet ended up fixing it. As with Redis, the files developed are also available in the delivered files.
- DNS issues. Initially, the application was unable to fetch results from the external RSS sources and therefore we would need to make some configurations about the DNS. After some research, we concluded that configuring the dnsPolicy and dnsConfig fields of the server and the worker deployments would be the best solution. We observed that if the dnsPolicy was set to "ClusterFirst" it would make it impossible to communicate with external sources and if the dnsPolicy was set to "Default" it would be the other way around independently of what we would put in the dnsConfig parameters. Eventually copying the "resolv.conf" file from a pod running with "ClusterFirst" into the pods running with "Default" worked and given the description of these policies [2] we believe that this allows inheriting the host nameserver's configuration while also using the cluster DNS to handle internal address resolution.

IX. Deployment Demo

The following images represent a demo with the most relevant sections and actions from the platform. The demo starts by showing the promotional page(Fig. 2). Then, comes the login (Fig. 3). When inside the platform, there is a landing page (Fig. 4) where it is possible to see the feeds according to category, add new feeds to a category (Fig. 6), select a news article from a feed (Fig. 7), favorite articles (Fig. 8), check favorites (Fig. 9)

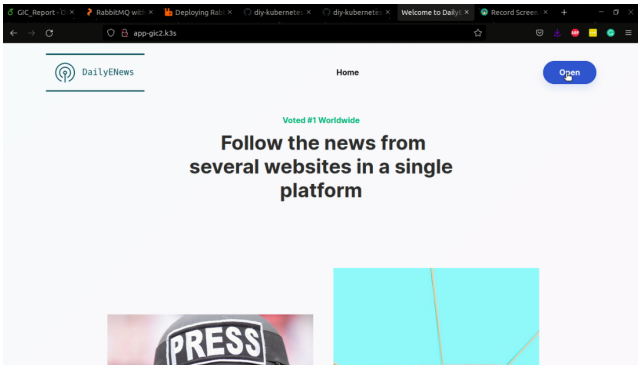

Fig. 2. Promotional website.

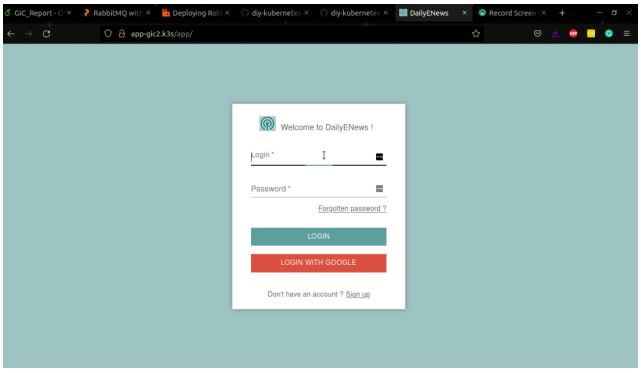

Fig. 3. Login Page.

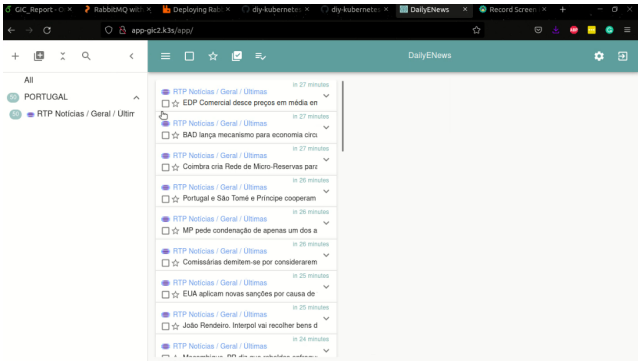

Fig. 4. Landing page.

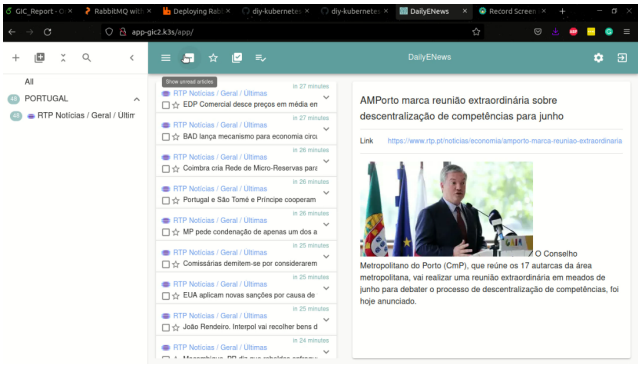

Fig. 5. Unread article section.

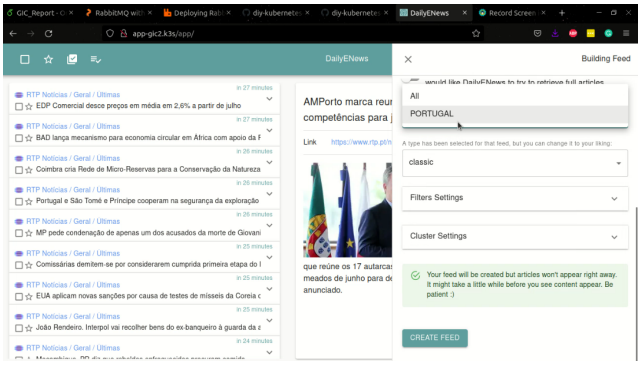

Fig. 6. Adding a feed to a category.

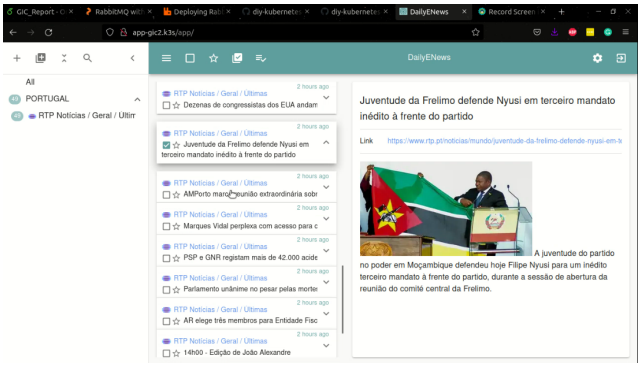

Fig. 7. Landing page with expanded news article.

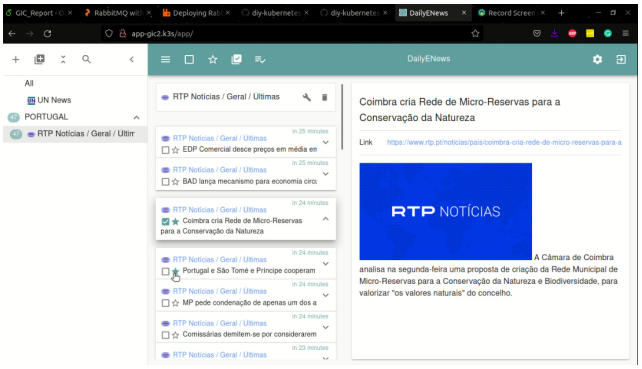

Fig. 8. Article was added to favorites.

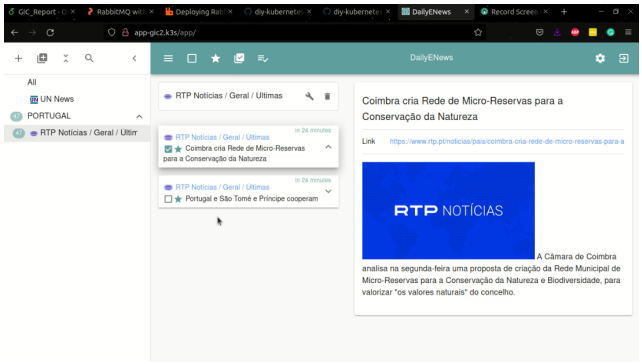

Fig. 9. Favorite news articles section.

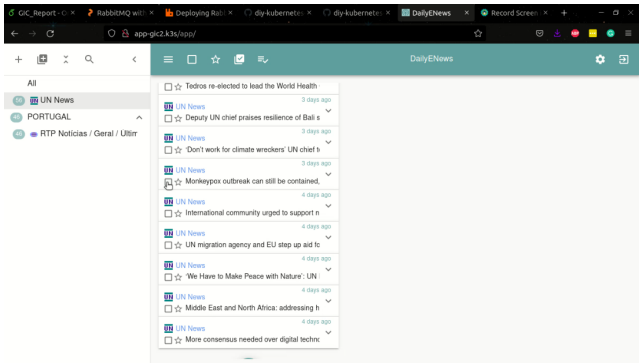

Fig. 10. Feed was added and got news.

X. Conclusion

To summarize, when it comes to the deployment we managed to set up replication for the stateless components of the architecture. The stateful components were not replicated because they are more complex and there were some major issues with one of the components which led to a big waste of time, not only that but most documentation found for these components were not suitable for our implementation. Even with these complications, we managed to deliver the product. It is not as complete as expected but it is a working product and the main goals were achieved.

References

[1] Deploying RabbitMQ to Kubernetes: What's Involved? — RabbitMQ - Blog. (2020). RabbitMQ Blog. https://blog.rabbitmq.com/posts/2020/08/deploying-rabbitmq-to-kubernetes-whats-involved/

[2] DNS for Services and Pods. (2022, May 27). Kubernetes. https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/

[3] Shanmugam, B., Singh, K., Harsh, K., Harsh, K. (2022). Deploying Redis Cluster on Kubernetes — Tutorial and Examples. ContainIQ. https://www.containiq.com/post/deploy-redis-cluster-on-kubernetes

[4] de Antoni, F. (2022, March 30). Redis Cluster on Kubernetes - Geek Culture. Medium. https://medium.com/geekculture/redis-cluster-on-kubernetes-c9839f1c14b6

[5] StackSoft Inc. (2019). StackSoft. https://stacksoft.io/blog/postgres-statefulset/