Drupal on Kubernetes

GIC

Universidade de Aveiro

Ricardo Antunes (98275), Miguel Matos (103341), Filipe Antão (103470)

Contents

Chapter 1

Introduction

In this report, we present the product we chose to use as the basis for the project, as well as the deployment strategy used to deploy it on a Kubernetes cluster, shared by the whole class.

This first delivery, along with this report, consists of the following artifacts:

- Source Code Repo: `github.com/mankings/GIC-PROJECT`
- Project Deployment: `uareviews.k3s` and `landingpage.uareviews.k3s` on the class' cluster

The source code repository is organized in the following manner:

- `config` folder contains configuration files for the containers
- `deploy` folder contains Dockerfiles and Kubernetes declarative configurations
- `reports` folder contains written reports regarding the project
- `scripts` folder contains deployment and update scripts
- `src` folder contains the source code of the applications

Chapter 2

UA Reviews

UAReviews consists of a review application for courses on the University of Aveiro. Thus, its primary user base will be alumni and professors. User activity will probably peak around the times where alumni need to apply for classes, looking for reviews, or when grades are released and reviews are written.

The application is based on Drupal, which is an open-source content management system (CMS). The primary technologies underlying Drupal include a web server, such as Apache or Nginx, PHP, the scripting language in which it is written, and a relational database, used for storing content and settings. It is built upon the LAMP stack (Linux, Apache, MySQL, PHP), which is a popular set of open-source software used to create and host web applications.

Drupal's architecture relies on modularity to expand it's functionality. We used two of these modules to allow for better caching using Redis and enable logging using Rsyslog.

2.1 Traditional Monolithic Deployment

Traditionally, Drupal is deployed in a monolithic architecture where all its components, including the database, file storage, and web server, are installed and run on a single machine or instance. This setup simplifies management and maintenance but can pose challenges as the demands on the system grow, such as increased traffic or content volume, which may lead to performance bottlenecks.

This traditional deployment model often requires significant resources for scaling, as enhancing performance typically involves upgrading the underlying hardware or the hosting environment. Furthermore, the monolithic nature means that the entire system can be impacted by a single point of failure, affecting availability and reliability.

In the following sections, we will describe our approach to deploying a Drupal based application on a Kubernetes cluster and explore how transitioning to a service-oriented architecture can address some of the limitations of the

traditional monolithic deployment.

2.2 Service Oriented Approach

In order to deploy our application on a Kubernetes, we need to containerize its components to fit into the cloud native approach of Kubernetes. Thus, we split the application into the following components:

- PHP-FPM + Apache for the Drupal service
- MariaDB as the database service
- Rsyslog as the logging service
- Redis as the caching service
- Nginx as the load balancing service

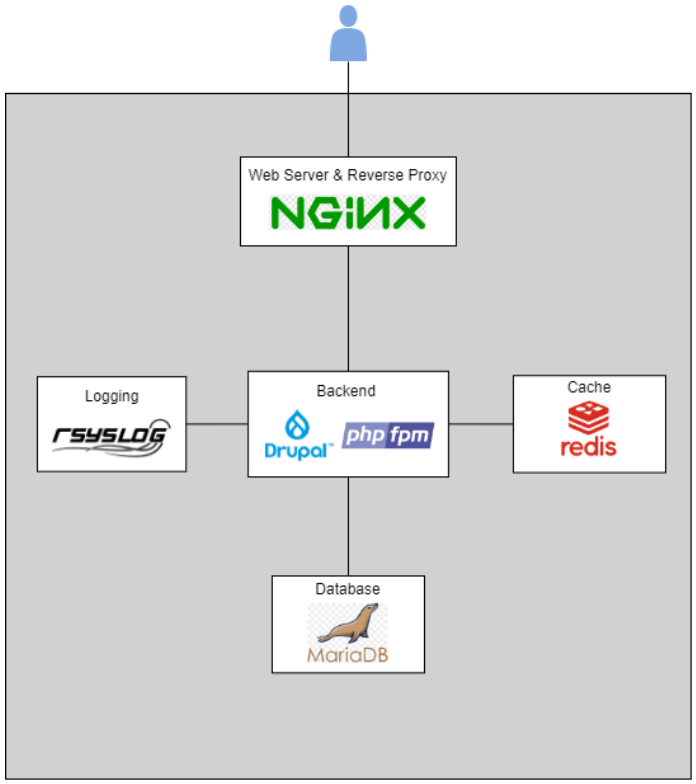

Figure 2.1: System's architecture

Each of this components can be mapped to a Docker container. As for the assignment requirements, we built our own containers based on an Alpine Linux Docker image, versioned 3.19. We installed all packages either using *apk* or *wget*. Regarding container configuration, these following paragraphs contain the changes made to ease the deployment into a cloud native environment.

2.2.1 Drupal

The Drupal container has a PHP-FPM 8.2 server and an Apache2 instance to server the files that PHP generates. On image build, it download the source code of Drupal and sets up Apache to serve it. We also modified the default Drupal settings to read some variables from the environment, such as database connection settings, so that we can later define them in Kubernetes configurations. This container also has an Rsyslog agent inside it so that it can send logs to the logging server, as well as PhpRedis so that Drupal can use Redis as a cache.

2.2.2 MariaDB

For the MariaDB container, we initially had an image made by us, based on alpine:3.19. However, we could not get it to expose the server to outside clients, despite obtaining pod connectivity through pings. Thus, we used the latest official MariaDB docker image from Docker Hub. However, we still have our own Dockerfile and configurations in the GitHub repository.

2.2.3 Redis, Nginx and Rsyslog

For these three images, we install the required packages through *apk* and mounted a default configuration. Later customization to these configurations is done in later stages of deployment, such as passing secrets and other settings that are more prone to change.

Chapter 3

Kubernetes Manifests

Using the Docker images described in the previous chapter, we then had to deploy our application in a Kubernetes cluster.

3.1 Landing Page

3.1.1 ConfigMap

Stores the *nginx.conf* the will be mounted to serve the pages' static files.

3.1.2 PersistentVolumeClaim

Used to store the static files that are served using this web server.

3.1.3 Deployment

The Landing Page consists of a deployment of an Nginx instance serving some static files, which are mounted as a volume. It also mounts a configuration to know how to serve these files.

3.1.4 Service

Exposes the Nginx Landing Page deployment to be accessed by an Ingress rule.

3.2 Drupal

3.2.1 ConfigMap

Stores configuration data for the Drupal application, including MySQL and Redis connection details.

3.2.2 Deployment

Defines the deployment configuration for the Drupal application and specifies container image, environment variables sourced from ConfigMap and Secret, ports, and volume mounts for persistent data storage.

3.2.3 PersistentVolumeClaim

Requests storage for persistent data storage needed by the Drupal application and defines access mode.

3.2.4 Secret

Stores sensitive information like database credentials for the Drupal application and encodes it in base64 format to maintain secrecy.

3.2.5 Service

Exposes the Drupal application to external traffic and routes traffic to the Drupal Deployment based on the selector.

3.3 MariaDb

3.3.1 PersistentVolumeClaim

Requests storage for persistent data storage needed by the MariaDb database and defines access mode.

3.3.2 Secret

Stores sensitive information for the MariaDB database, including user credentials and database name, encoded in base64 format.

3.3.3 Service

Exposes the MariaDB service on port 3306 and the selector ensures that it targets pods labeled with "app: mariadb". Also, ClusterIP is set to None, indicating that this service is not exposed externally.

3.3.4 StatefulSet

Deploys and manages a StatefulSet for MariaDB. The selector specifies pods labeled with "app: mariadb". It defines a template for pods, including labels and containers configuration. Also, containers run MariaDB image, mount persistent volume for data storage, set environment variables using secrets for database configuration, and defines a volumeClaimTemplate for dynamic provisioning of PersistentVolumeClaims.

3.4 Nginx

3.4.1 ConfigMap

Stores the Nginx configuration file (nginx.conf) for the uareviews namespace.

3.4.2 Deployment

Deploys the Nginx container with the specified image and configuration, mounts the Nginx configuration file from the ConfigMap as a volume, and specifies labels and ports for the Nginx container.

3.4.3 Service

Exposes the Nginx deployment internally within the uareviews namespace, maps port 80 of the service to port 80 of the Nginx container, and selects pods labeled with app: nginx to route traffic to.

3.5 Redis

3.5.1 Deployment

Deploys the Redis container with the specified image, sets the number of replicas to 1, specifies labels for the Redis pod, defines environment variables, sourced from a secret named redis-secret.

3.5.2 Service

Exposes the Redis deployment internally within the uareviews namespace, maps port 6379 of the service to port 6379 of the Redis container, and routes traffic to pods labeled with app: redis.

3.6 Rsyslog

3.6.1 Deployment

Deploys the rsyslog container with the specified image, sets the number of replicas to 1, and specifies labels for the rsyslog pod.

3.6.2 Service

Exposes the rsyslog deployment internally, maps port 50514 of the service to port 50514 of the rsyslog container, and routes traffic to pods labeled with app: rsyslog.

3.7 Ingress

3.7.1 Landing Page

Configures the Ingress resource to route traffic to the Landing Page service, sets annotations for Traefik Ingress Controller, defines rules to handle incoming HTTP requests based on the specified host (`landingpage.uareviews.k3s`) and path (/), and routes requests to the Landing Page service (landingpage-svc) on port 80.

3.7.2 Main Ingress

Much like the Landing Page Ingress rule, defines a rule to access the Nginx load balancer in front of Drupal on the host `uareviews.k3s`.

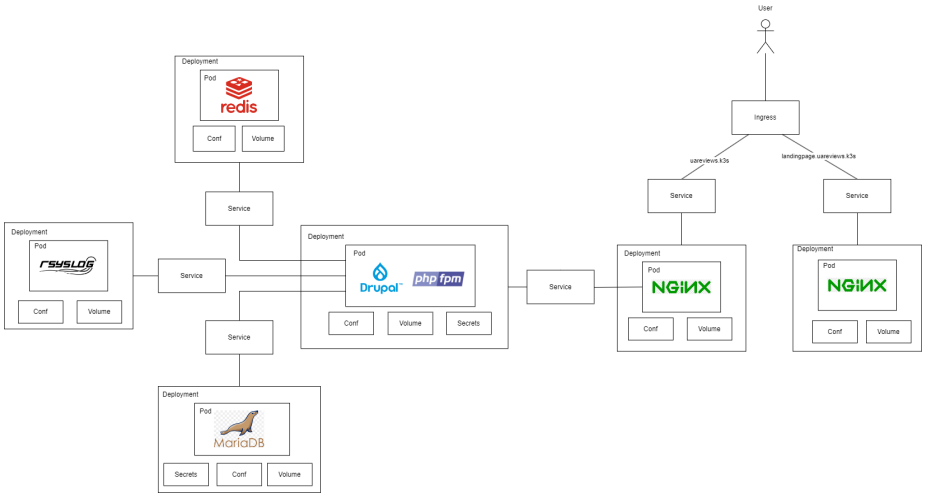

Figure 3.1: Kubernetes architecture

Chapter 4

Deployment and Update Process

To ease the deployment and update process, we made some scripts that help with these processes. These are located in the *scripts* folder of the source code repository, and are meant to be ran at the root of the repository.

Each module has its own set of four scripts, and there is also a set of four wrapper scripts that run the same script for all the modules at the same time. The scripts are organized as follows:

- apply-[module].sh - applies all Kubernetes configuration files related to the module on the cluster
- build-[module].sh - builds ands tags/versions all container images related to the module
- push-[module].sh - pushes container images related to the module to registry.deti
- clean-[module].sh - cleans all resources related to the module from the cluster

Chapter 5

High Availability

High availability (HA) is a crucial aspect of modern software deployment, ensuring that applications remain accessible and operational even in the face of failures or increased load. The primary goal of HA is to minimize downtime and maintain continuous service by using redundancy and fault tolerance. This is particularly important for applications deployed on cloud-native platforms like Kubernetes, where various components can fail independently.

5.1 Healthchecks

Health checks are an essential part of container management in Kubernetes. Essentially, they ensure that the containers within a pod are healthy, meaning they are functioning as expected, allowing Kubernetes to take appropriate actions in case of failures or malfunctions. To perform health checks, Kubernetes uses probes, which are mechanisms that enable Kubernetes to verify the health status of the containers.

There are three types of probes in Kubernetes:

- Liveness probe: determines if a container is operating. If it does not operate, the kubelet shuts down and restarts the container.
- Readiness probe: determines if the application that runs in a container is ready to accept requests. If it is ready, Kubernetes allows matching services to send traffic to the pod. If it is not ready, the endpoints controller removes this pod from all matching services.
- Startup probe: determines if the application that runs in a container has started. If it has started, Kubernetes allows other probes to start functioning. Otherwise, the kubelet shuts down and restarts the container.

In our project, we implemented healthChecks for both mariaDB and Drupal, using liveness probes and readiness probes to ensure the reliability and availability of the services.

5.1.1 mariaDB

For mariaDB, we have configure both liveness and readiness probes. These probes use a simple command to login on the database. This allows us to verify the connectivity and operational state of the database.

```
livenessProbe:
  exec:
    command:
    - sh
    - -c
    - "mariadb -u $MYSQL_USER -p$MYSQL_PASSWORD -e 'SELECT 1'"
  initialDelaySeconds: 30
  periodSeconds: 10
readinessProbe:
  exec:
    command:
    - sh
    - -c
    - "mariadb -u $MYSQL_USER -p$MYSQL_PASSWORD -e 'SELECT 1'"
  initialDelaySeconds: 5
  periodSeconds: 10
```

Figure 5.1: HealthChecks on mariaDB

5.1.2 Drupal

```
livenessProbe:
  exec:
    command:
    - wget
    - --spider
    - -q
    - http://uareviews.k3s/
  initialDelaySeconds: 50
  periodSeconds: 20
```

Figure 5.2: HealthChecks on Drupal

For Drupal, we have configured readiness probes. This probe makes an HTTP command to the specified endpoint. If the response has a 200 status

code, it means the pod is ready to receive traffic. This is important since drupal container might be running but, since it has to configure the database and other modules, it might not be ready to receive requests.

5.2 Autoscaling

With an increase in the number of requests made to our platform, the pods may crash due to elevated resource consumption. So it's important to ensure that there are always a number of pods running that matches the amount of resources being used at a given time. We have two options to make this happen: Vertical Scalability and Horizontal Scalability. Vertical scalability is updating the resources requested by the pod according to the needs of that pod. Horizontal scalability would be increasing the number of pods running that service. For our solution we used Horizontal scaling. For this we created a Kubernetes Object called Horizontal Pod Autoscaler.

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: drupal-hpa
  namespace: uareviews
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: drupal-dep
  minReplicas: 1
  maxReplicas: 3
  targetCPUUtilizationPercentage: 70
```

Figure 5.3: Horizontal Pod Autoscaler

As seen in Figure 5.1, every 15 seconds the controller manager queries the resource utilization of the pods managed by the Drupal deployment. If the resource utilization surpasses the 70% threshold, it creates another pod. Also, when traffic diminishes, the pods are downscaled, thus managing resources consumed.

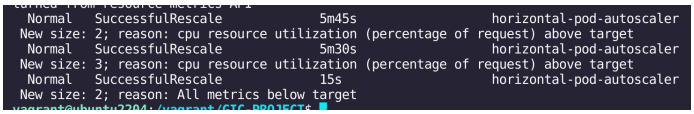

Figure 5.4: Example of AutoScaling

5.3 MariaDB Replication

In order to ensure redundancy and disaster recovery, our database is replicated. It follows a master-slave replication architecture where the Read operations are executed on the slaves and the Write Operations are executed only in the Master. This is possible because, as data is being written in the Master, it propagates the operations to the slaves, ensuring eventual consistency (data will be consistent in all nodes, even if it takes longer due to the duration of the transactions).

The slaves can also act as backups of the database in the case of a disaster, since slaves can be deployed in different nodes in different regions, thus providing geo redundancy.

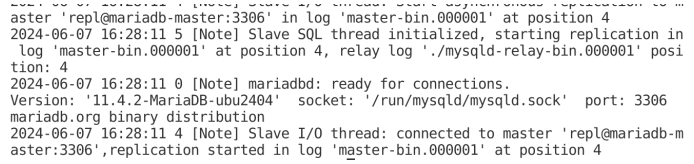

Figure 5.5: Slaves connected to Master and ready to replicate

5.4 Redis High Availability

Redis natively supports an high availability setup by using its Sentinel feature. In essence, the master instances will be the ones responsible for writing, while the slaves will replicate the masters' data. Sentinels will watch the master instance for failovers, and promote the slave with the most up to date data to a new master. Besides that, they are also responsible for providing the slaves with the current masters' IP address, so that they know whose data to replicate. The following diagram depicts the solution's arquitecture:

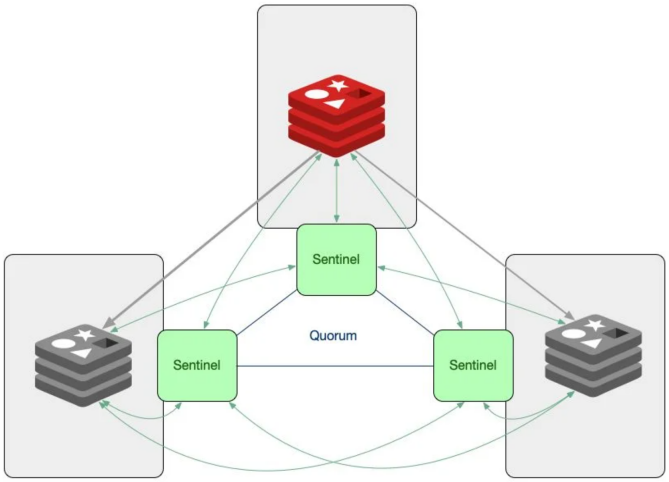

Figure 5.6: Redis Sentinel Setup

```
1:C 07 Jun 2024 16:44:57.529 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
1:C 07 Jun 2024 16:44:57.529 # Redis version=6.2.6, bits=64, commit=00000000, modi
fied=0, pid=1, just started
1:C 07 Jun 2024 16:44:57.529 # Configuration loaded
1:S 07 Jun 2024 16:44:57.530 * monotonic clock: POSIX clock_gettime
1:S 07 Jun 2024 16:44:57.531 * Running mode=standalone, port=6379.
1:S 07 Jun 2024 16:44:57.532 # Server initialized
1:S 07 Jun 2024 16:44:57.532 * Ready to accept connections
1:S 07 Jun 2024 16:44:57.532 * Connecting to MASTER redis-svc:6379
1:S 07 Jun 2024 16:44:57.558 * MASTER <-> REPLICA sync started
1:S 07 Jun 2024 16:44:57.558 * Non blocking connect for SYNC fired the event.
1:S 07 Jun 2024 16:44:57.558 * Master replied to PING, replication can continue...
1:S 07 Jun 2024 16:44:57.558 * Partial resynchronization not possible (no cached m
aster)
1:S 07 Jun 2024 16:44:57.559 * Full resync from master: 63db4efb40415fb08344a75d57
d9ed243bdafa61:0
1:S 07 Jun 2024 16:44:57.616 * MASTER <-> REPLICA sync: receiving 175 bytes from m
aster to disk
1:S 07 Jun 2024 16:44:57.616 * MASTER <-> REPLICA sync: Flushing old data
1:S 07 Jun 2024 16:44:57.616 * MASTER <-> REPLICA sync: Loading DB in memory
1:S 07 Jun 2024 16:44:57.617 * Loading RDB produced by version 6.2.6
1:S 07 Jun 2024 16:44:57.617 * RDB age 0 seconds
1:S 07 Jun 2024 16:44:57.617 * RDB memory usage when created 1.83 Mb
1:S 07 Jun 2024 16:44:57.617 # Done loading RDB, keys loaded: 0, keys expired: 0.
1:S 07 Jun 2024 16:44:57.617 * MASTER <-> REPLICA sync: Finished with success
1:S 07 Jun 2024 16:44:57.618 * Background append only file rewriting started by pi
d 11
1:S 07 Jun 2024 16:44:57.640 * AOF rewrite child asks to stop sending diffs.
11:C 07 Jun 2024 16:44:57.640 * Parent agreed to stop sending diffs. Finalizing AO
F...
11:C 07 Jun 2024 16:44:57.640 * Concatenating 0.00 MB of AOF diff received from pa
rent.
11:C 07 Jun 2024 16:44:57.641 * SYNC append only file rewrite performed
11:C 07 Jun 2024 16:44:57.641 * AOF rewrite: 0 MB of memory used by copy-on-write
1:S 07 Jun 2024 16:44:57.659 * Background AOF rewrite terminated with success
1:S 07 Jun 2024 16:44:57.659 * Residual parent diff successfully flushed to the re
written AOF (0.00 MB)
1:S 07 Jun 2024 16:44:57.659 * Background AOF rewrite finished successfully
```

Figure 5.7: Redis Slave watching Master

Chapter 6

Monitoring and Observability

Since we didn't have the source code for Drupal, we couldn't integrate Observability in our platform, neither with Auto instrumentation with the OpenTelemetry SDK nor with other exporters such as Prometheus. Nonetheless, if this problem was solved and we could export metrics and traces, we know what to do. This next section will explain our course of action if we had integrated OpenTelemetry into our Drupal application.

6.0.1 Collector

The OpenTelemetry Collector receives the metrics and traces exported by Drupal. It then has the capability to process them and export each type to a specific backend that is supported.

6.0.2 Metrics

The metrics we would collect are the following:

- Number of Website Requests: is useful to analyze trends and find the time of the day where our platform is being used the most.
- Number of Failed Login Attempts: is useful to detect possible brute force attacks.
- Resource Consumption: is useful to troubleshoot performance issues that may occur in our platform.

These metrics would be collected by the OpenTelemetry Collector and exported to a specific backend such as Prometheus. Then, using Grafana, we could create a dashboard in order to visualize the metrics.

6.0.3 Traces

Traces are useful in troubleshooting performance issues that may occur in our platform. The traces would be received by the OpenTelemetry Collector and exported to a specific backend such as Jaeger.

6.0.4 Logs

Logs from Drupal are sent to Rsyslog.

Chapter 7

Discussion

7.1 Future Work

There are some areas of this work that need to be worked on. One of them is the Observability and Monitoring of our platform. Since we couldn't integrate OpenTelemetry with Drupal, that would be something to work on in the future.

The pods DNS did not allow us to download Drupal modules from a deployed pod, neither did it allow to check for updates automatically. Thus, we had to resort to an approach where we placed the module files inside the Docker image at build time, which is not optimal for a Drupal application.

Also, the database replication was not working with Drupal, we don't know why. For some reason, Drupal was not responding when using the MariaDb master-slave architecture. So in the future, we would work on it.