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

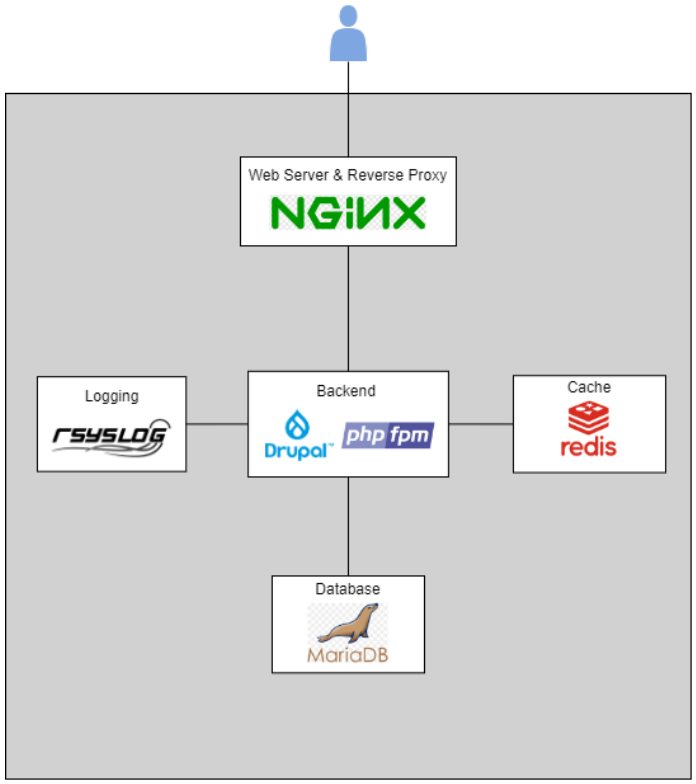

Figure 2.1: System's architecture

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 Deployment

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`.

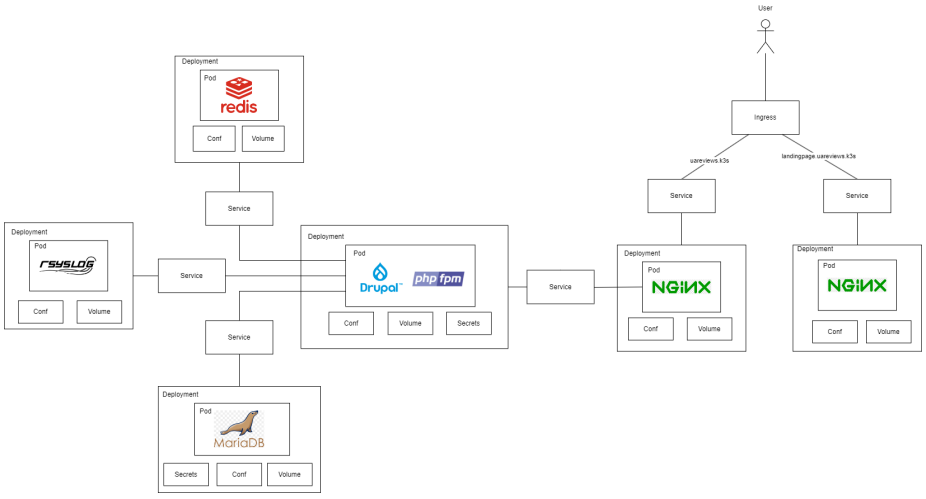

Figure 3.1: Kubernetes architecture

Chapter 4

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.

- apply.sh - applies all Kubernetes configuration files related to the project
- build.sh - builds ands tags/versions all container images related to the project
- push.sh - pushes container images related to the project to registry.deti
- clean.sh - cleans all resources related to the project

Chapter 5

Discussion

5.1 Possible Bottlenecks

To help identify possible bottlenecks, we can take a look at how each individual micro-service can be affected by the current data flow.

The PHP service inside the Drupal container is responsible for generating the files served to the client, so an increased amount of requests with static resource allocation can constrain the service.

The database service suffers from the same type of problem. It also has a need to keep the data stored across multiple replicas consistent, so that the same query to different instances of MariaDB returns always returns the same data.

Redis, has a cache, does not crucially need to persist its data, since content is stored on the database. However, pod crashes will lead to a cache wipe, meaning that the later requests will not benefit from the cache's better query response times. Thus, high availability can be obtained by replicating the in-memory cache, but that is out of scope of this first assignment.