

CONTAINERIZATION AND SCALING OF A PHP APPLICATION

Docker & Kubernetes

Abstract

Author(s) Juntunen, Joni	Type of publication Bachelor's thesis	Published Fall 2019
	Number of pages 41	
Title of publication Containerization and scaling of a PHP application Docker & Kubernetes		
Name of Degree Bachelor of Software Engineering		
Abstract <p>In this project, a monolithic PHP application was containerized with Docker and orchestrated using Kubernetes. The PHP application was built using the Symfony framework. An additional requirement was serving each client organization with their own copy of the application accessed via different DNS names.</p> <p>Choices for the PHP engine and web server were PHP-FPM and Nginx. These were separated to multiple containers to maintain the single process per container principle. PHP application source was added to the PHP-FPM container. The PHP container was built using multistage Dockerfile with a separate build stage.</p> <p>Kubernetes was used to orchestrate the pods consisting of two containers, Nginx and PHP-FPM. Environment variables were used to customize each pod for each client organization. Kubernetes was built on virtual machines as a bare metal solution.</p> <p>To handle the network connections to Kubernetes, a Nginx proxy was used between the WAN network and the Kubernetes network. All traffic to the nodes in the Kubernetes network must pass through the proxy, separating the cluster from public access. Traffic to the proxy was secured with an SSL certificate.</p> <p>Kubernetes was found to be suitable platform for a software as a service product. Running containers inside Kubernetes required definition of multiple different layers and services which increased the complexity of the backend administration.</p>		
Keywords Docker, Kubernetes, containers, orchestration, DevOps, scaling, PHP		

Tiivistelmä

Tekijä(t) Juntunen, Joni	Julkaisun laji Opinnäytetyö, AMK Sivumäärä 41	Valmistumisaika Syksy 2019
Työn nimi PHP-sovelluksen kontitus ja skaalaus Docker & Kubernetes		
Tutkinto Tieto- ja viestintäteknikan insinööri (AMK)		
Tiivistelmä <p>Opinnäytetyössä tehtiin ohjelmistokontitus PHP-ohjelmalle käyttäen Docker-kontteja ja Kubernetes-orkestrointiteknologiaa. PHP-ohjelma oli luotu käyttäen Symfony-ohjelmistokehystä. Lisävaatimuksena oli tarve tarjota jokaiselle asiakkaalle oma kopio ohjelmasta perustuen DNS-verkkonimeen. Työ tehtiin yritykselle IT-palvelut Joni Juntunen.</p> <p>PHP-tulkiksi valittiin PHP-FPM-moottori ja web-palvelimeksi Nginx. Nämä laitettiin erillisiin kontteihin, yksi sovellus konttia kohti periaatteen mukaisesti. Ohjelmiston lähdekoodi tallennettiin mukaan PHP-FPM-konttiin ja kontin kuvaustiedosto muodostettiin monivaiheisella Dockerfile-määrittelyllä.</p> <p>Kubernetesilla hallittiin näistä konteista muodostettuja kapseleita (Pod). Yhteen kapseliin laitettiin kaksi konttia, Nginx-kontti ja PHP-FPM-kontit. Ympäristömuuttujilla kapseleissa olevat kontit muokattiin asiakaskohtaisilla asetuksilla. Kubernetes asennettiin virtuaalipalvelimille.</p> <p>Ulkoisia yhteyksiä varten Kubernetes-klusterin eteen asetettiin Nginx-välityspalvelin. Ulkoinen pääsy klusteriin eristettiin tämän välityspalvelimen taakse. Yhteydet välityspalvelimeen salattiin SSL-salauksella.</p> <p>Kubernetes soveltui alustaksi Software as a Service muotoisen palvelun tuottamiseen. Konttien ajaminen Kubernetesissä vaati useiden eri Kubernetes-palveluiden asetusten määrittelyitä, mikä taas lisäsi palvelun ylläpidon monimutkaisuutta.</p>		
Asiasanat Docker, Kubernetes, Ohjelmistokontit, Orkestrointi, DevOps, Skaalautuminen, PHP		

CONTENTS

1	INTRODUCTION	1
2	THE APPLICATION	2
3	TECHNOLOGIES	3
3.1	PHP	3
3.2	Symfony framework	3
3.3	Nginx	4
3.4	Software containers	5
3.5	Scalability	6
3.6	Kubernetes	8
3.6.1	Pods	8
3.6.2	Services.....	9
3.6.3	Volumes	10
3.6.4	Deployments.....	10
3.6.5	Ingresses.....	11
3.6.6	ConfigMaps and Secrets	11
4	DOCKER	12
4.1	Docker engine	12
4.2	Docker images.....	13
4.3	Dockerfile and dockerignore	14
4.4	Docker container	15
4.5	Docker registry	16
4.6	Docker volumes and storage drivers.....	16
5	ARCHITECTURE.....	17
5.1	Load balancing	18
5.2	Scaling.....	18
5.3	Availability	19
5.4	Data persistence.....	19
5.5	Deployment	20
5.5.1	Continuous Integration.....	20
5.5.2	Continuous Delivery.....	21
6	CONFIGURATIONS	22
6.1	Private image registry	22
6.2	Nginx proxy.....	25

6.3	Kubernetes cluster.....	26
7	SUMMARY	38
	REFERENCES	39

1 INTRODUCTION

Scalability and high availability are important in today's web applications and they go mostly hand in hand with each other. High availability is especially important if the application supports everyday work and is accessed multiple times during the day.

Software containers are a way to package and deliver applications. A PHP web application has unique requirements for running inside containers. Containers suit well for scaling because of their undetermined lifetime. Containers can be terminated and started as required.

This thesis work was done through a company called IT-palvelut Joni Juntunen. The company is a one-man's business that provides outsourced IT services and software development services for client organizations. The company was established in fall 2018.

The PHP application is produced for a new startup using the application as their primary product. The Application's primary functions are project and work management using mobile devices and collection of work hours for salaries and invoices.

The purpose of this thesis work is to containerize the application using Docker and then orchestrate and manage this application in Kubernetes. Design and implementation of the surrounding architecture are also part of the work but not the focus of this thesis.

2 THE APPLICATION

The application is a single-page PHP-based web application distributed with software as a service model. Frontend is custom JavaScript using jQuery with some elements from React JavaScript framework. Mainly navigation and main application component using a state. Views are built on server-side using Twig templates and are accessed via Ajax requests. Views are responsive and designed as mobile first. The CSS framework used is Bootstrap 3.

The current backend is Nginx webserver with MariaDB database. Each client organization has their own copy of the application source code configured via environment variables defined in the configuration file. Databases are running on the same hosts as the application. Client instances are separated with domain names.

The current architecture shown in Figure 1 does not scale except for adding more servers and spreading client instances to different machines. A Single client organization is limited to only one server for their instance.

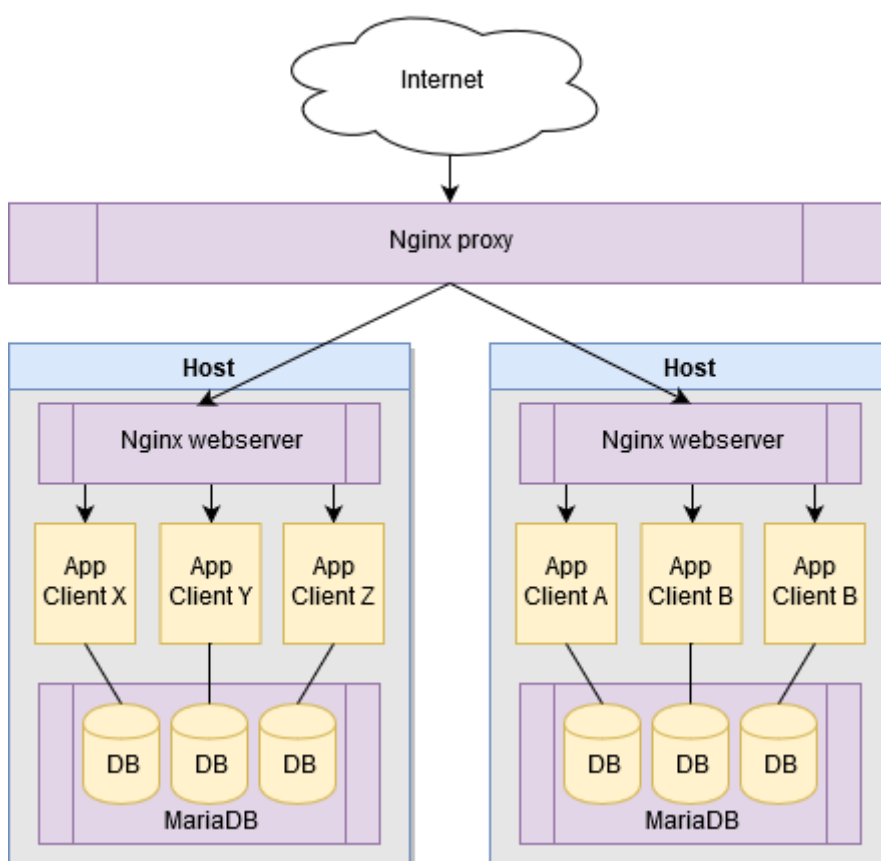


Figure 1. Current architecture

3 TECHNOLOGIES

3.1 PHP

PHP is a scripting language that is compiled during execution. The Web server hosting PHP content uses Common Gateway Interface (CGI) to call the PHP interpreter to execute the scripts. The Nginx web server supports PHP FastCGI Process Manager (PHP-FPM), which uses process pools to handle the requests. (Nginx documentation 2019.)

PHP version 7 introduced an improved Zend engine and performance boost to opcode caching. Opcode caches are used to store and share precompiled scripts to skip the compilation step. These upgrades greatly improved the response times compared to older versions of PHP. (Gavalda 2019; PHP documentation 2019.)

PHP version 7 also introduced the possibility of type hinting and strong typing. PHP is a loosely typed language and variables do not have a type. Type hinting can be used to define types for function arguments and function returns. An exception is thrown when the variable type does not match the hinted type (strict mode) or when the variable cannot be directly converted to the required type (default mode). (PHP documentation 2019.)

PHP has a package manager named Composer for handling the dependencies of the application. Composer uses Packagist as primary repository. Dependencies are stored in the vendor directory and must be delivered with the application. Composer uses the composer.json file to store the information about dependencies. (Composer documentation 2019.)

3.2 Symfony framework

The Symfony framework is an open source full stack framework for PHP language. It scales from minified APIs to full web applications. The framework consists of multiple independent components that all have specific tasks, for example Logger for logs and HTTP Foundation for receiving HTTP requests and creating responses. Some popular PHP projects use Symfony components in their core, like the Drupal 8 content management system. (Symfony SAS 2019.)

The framework uses model-view-controller (MVC) architecture to separate different layers responsible for handling the requests. Models are entities defining domain level objects and their relationships. Models' state is stored to the database using the Doctrine object

relational mapper (ORM). The Doctrine abstracts the database layer and automatically generates SQL queries. (Salehi 2016, 2; Symfony SAS. 2019.)

Controllers handle the requests and change the state of the models. Best practices in Symfony recommend use of thin controllers. Thin controllers are kept as light as possible and all domain level code is executed in separate services called by the controller. The controller returns the response. (Salehi 2016, 32-34; Symfony SAS. 2019.)

Views are the responses created using templates. Models are attached to the views by the controller and templates define the appearance of the views. The default templating engine in Symfony is Twig, which has its own templating language. (Salehi 2016, 23-24; Symfony SAS. 2019.)

Symfony supports environment variables and config files as a source of configuration definitions. Environment variables override config files. Basic definitions are the running environment `APP_ENV` and `DATABASE_URL`. By default, allowed values for `APP_ENV` are `dev`, `prod` and `test`. Creating custom levels is possible. `Dev` is for development use and it disables caches, enables Symfony profiler and displays verbose error messages. The `prod` level enables caches and disables all development features. The `test` level is for automated testing. (Symfony SAS. 2019.)

3.3 Nginx

Nginx is an open source HTTP server and reverse proxy. It uses asynchronous event-based architecture instead of threads and forking to handle client requests. Nginx supports signal hook for reloading the configuration without restarting the service.

Nginx supports FastCGI applications by passing the request to the FastCGI process. Configuration of FastCGI for PHP requires passing the requested script file as a proxy call parameter. The destination can use Linux sockets or TCP ports.

```
nginx-fastcgi-php.conf ●
1  server {
2      location / {
3          fastcgi_pass localhost:9000;
4          fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
5      }
6  }
7  |
```

Image 1. Nginx FastCGI PHP example

The example in Image 1 presents the settings for a proxy which redirects all requests to localhost port 9000 (default port of php-fpm). Row 4 sets the parameter for the script file to be executed. Nginx's internal variables start with dollar \$-character. In the above example \$document_root is the root directory for the request. \$fastcgi_script_name is the path part of the Uniform Request Identifier (URI). It does not include the requested server name or the query parameters. (Nginx documentation 2019.)

3.4 Software containers

Containers are packages that include all the components and dependencies of a software application. Containers differ from virtual machines by having a shared operating system kernel. This has the benefit of requiring less resources and having less virtualization overhead, but they are less secure than virtual machines. Containers separate the application from the infrastructure. Containers have the benefit of being ephemeral instances of applications and services. (Rouse 2018; Docker Inc 2019d; Docker Inc 2019c.)

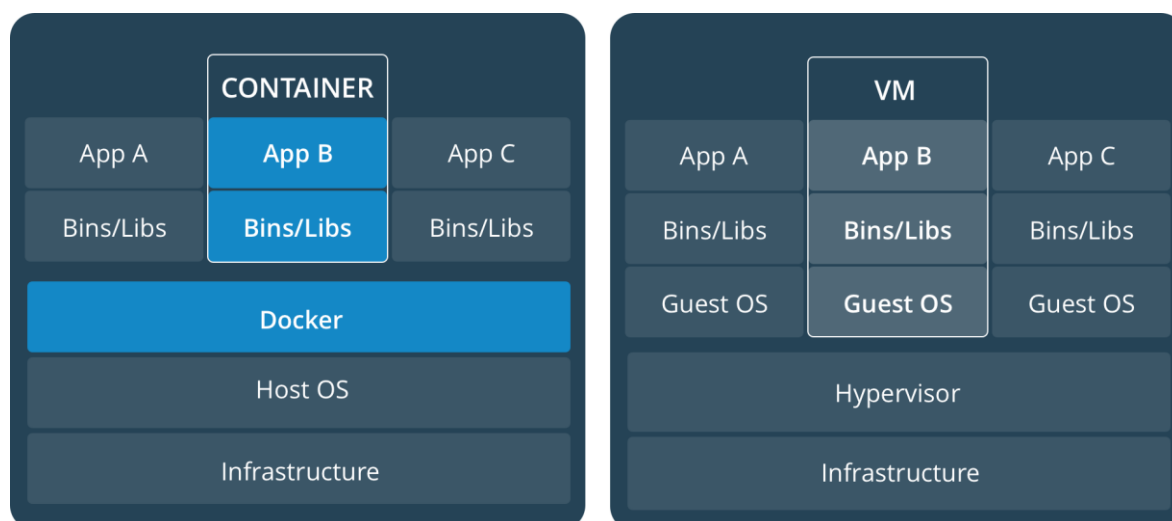


Figure 2. Container versus virtual machine (Docker Inc. 2019)

Figure 2 illustrates the primary difference between containers and virtual machines. A single host operating system with a container hosting platform (Docker) can serve multiple containers that are separated from each other and include all the dependencies of the application. In virtual machine-based systems, additional guest operating systems are required to separate applications from each other. The most flexible use of resources is achieved by using a combination of both technologies: hosting container platforms on multiple virtual machines and orchestrating containers on top of them using technologies like Kubernetes.

History of container technology

The history of container technology began with the development of chroot in version 7 of Unix in 1979. Chroot enabled isolating application's access to only specified directory and its subdirectories. Such a chrooted application cannot access files of other isolated applications running on the same host. (Mell 2018.)

In 1999 first container like isolation of resources was introduced in FreeBSD operating system's jail function. It used chroot implementation with hardened confinement. (Haff 2013.) 2001 the VServer project was born with Virtual Private Servers that separated user-space environment into distinct units. Its biggest weakness was that it required a Linux kernel with specialized patch to function properly. (Red Hat Blog 2015.)

Beginning from year 2002 the Linux namespaces were being developed. Namespaces are abstraction on top of system resources that allows the processes within the same namespace have their own isolated instance of system resources. Resources include things like cgroup root directory, network devices, mount points and hostnames. (Kerrisk 2010, 607.)

In 2003 Google introduced Borg, a container cluster management system. At that time, it relied on isolation technologies of Linux operating system and as such did not have process level isolation for resources. A single process could reserve all available resources and starve other processes. In 2004 Google begun a development of a process container technology known as control groups (cgroups). Cgroups allow organizing processes in to groups whose access to resources, like memory and CPU time, can then be limited. In 2008 cgroups were merged to Linux kernel and Linux container (LXC) technology was developed by IBM. (Mell 2018.)

Docker started as an internal tool for dotCloud company and in 2013 it was released as open source. 2014 Docker Incorporated released 1.0 version of their container technology and 2015 they donated the technology to Open Container Initiative. (Mell 2018.) Also, in 2014 Google announced Kubernetes container orchestration technology that is traced from Borg and in 2015 version 1.0 was released and Google gave the technology to Cloud Native Computing Foundation. (Red Hat Blog 2015; Krochmalski 2016, 6; Mell 2018.)

3.5 Scalability

Scalability is the ability to expand or reduce systems resources and have linear effect in its performance. Node is used to describe a single unit of processing. Performance is measured by how fast node completes a certain computing task. Scalability measures the

trend of performance with increasing load. In web-based applications performance is measured by response time to a request from an end user. (Abd-El-Barr 2005, 63; Liu 2009, 1-4.)

Scaling can be horizontal or vertical and it is measured over multiple dimensions:

- **Performance & efficiency**
Number of processors and their core speed. Efficiency is the balance between the size of the computational task and the number of processors and their interoperation overhead.
- **Size / Load**
Maximum number of processors/hosts that the system can accommodate before there is a negative impact on the performance because of internal communications or because of physical limitations.
- **Application**
The ability of an application to increase its performance by adding more processors for the application to utilize. Distributed systems can have multiple copies of the same application serving requests and increasing number of hosts has the same effect.
- **Generation**
The ability to upgrade the underlying hardware or systems without changes to the application code. Today's virtualization platforms provide abstractions so that hardware can be changed or upgraded without the virtual hosts noticing.
- **Heterogeneity**
The ability of a system to scale using hardware and software components from different vendors.

(Abd-El-Barr 2005, 66-67.)

Horizontal scaling is achieved by adding more nodes. For example, in web-based application by increasing the number of hosts serving the requests. Vertical scaling is about adding more resource to a node. For example, increasing number of processors on a node. Virtualization systems today can handle both scenarios by changing resources available to a virtual host (vertical) and the number of hosts (horizontal).

3.6 Kubernetes

The Kubernetes is an open-source container orchestration tool. It was originally developed by Google and was an update from Borg, Google's inhouse container orchestration software. Development of Kubernetes is now being overseen by the Cloud Native Computing Foundation, that is a sub-foundation of Linux foundation and it has members like Google, Amazon Web Services, Microsoft, IBM, Intel, VMware and RedHat. (Eldridge I. 2018; Cloud Native Computing Foundation 2019.)

Kubernetes consists of several components (Figure 3): Kubernetes master, Kubernetes nodes, ETCD and Flannel an overlay network. ETCD is a key-value store for the clusters data. Overlay network provides a network to the pods to communicate between each other. In addition to these there are abstractions called Kubernetes Objects which include pods, services, volumes and namespaces. Additional higher-levels abstractions called Controllers are built upon these Kubernetes objects. Kubernetes Control Plane is a term that covers all the components that can change the state of a Kubernetes cluster. (Saito, H. Hsu, C. & Lee, C. 2016, 1-7; The Kubernetes Authors 2019b.)

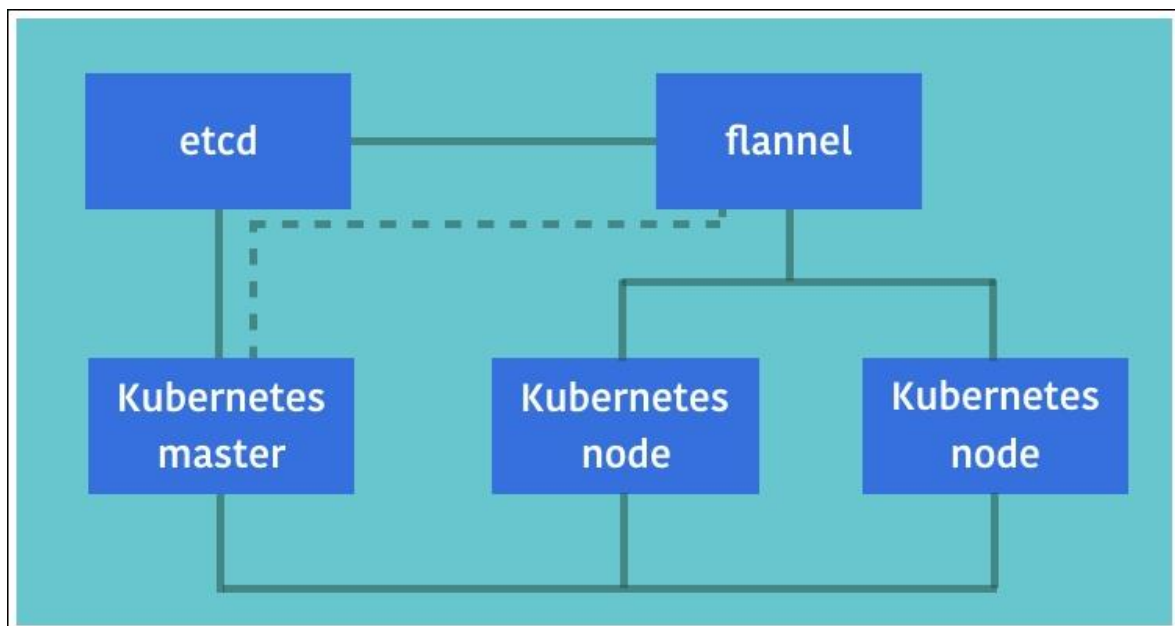


Figure 3. Kubernetes architecture (Saito et al. 2016)

3.6.1 Pods

The pod is the smallest deployment unit in the Kubernetes. Pod is a group of one or more containers, and they are guaranteed to be co-located on the same node. Containers inside the same pod share the localhost network and can directly communicate with each

other. The pods are isolated from each other using Linux namespaces. (Saito et al. 2016, 61; Eldridge I. 2018.)

Replication of the pods between the nodes and recovery of the crashed pods is automated through replication controller. Replication controller will automatically assign crashed pods to healthy nodes and keep the configured number of pods continuously running. (Saito et al. 2016, 67; Eldridge I. 2018.)

3.6.2 Services

The service is a layer that routes network requests to pods. Service has its own IP address and network name inside the cluster. Containers can access the services using service names. There are three different mapping types for the services, each having different availability and visibility:

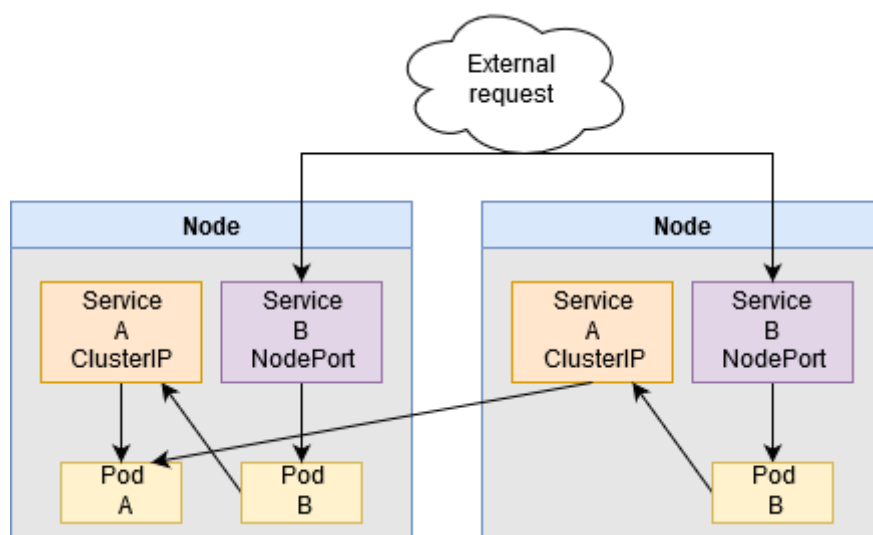


Figure 4. Example of service types ClusterIP and NodePort

- **ClusterIP**
Service A in the example Figure 4. Default service type. ClusterIP service is only available inside the cluster. It cannot be accessed from the outside of the cluster and it does not reserve a port on the nodes.
- **NodePort**
Service B in the example Figure 4. NodePort reserves a network port on each node using range 30000-32767 (default). Traffic to each port is then routed to the service and can be used to access the service from outside of the cluster.
- **LoadBalancer**
LoadBalancer is a Kubernetes cloud service provider specific implementation of mapping a public IP addresses to the nodes and NodePorts. There exists load

balancer for non-cloud Kubernetes called MetalLB that is under active development.

(Kubernetes Authors 2019; Saito et al. 2016, 77-83.)

The services use selector configurations to create endpoints for the matching pods. The pods define labels for metadata and these labels are then searched by the service selector definition. Each pod matching the selector is then mapped as an endpoint for the service. Services without selector definition do not create endpoints automatically. The endpoints are also used to map network locations outside of the cluster for an access through a service or as an access point between different namespaces inside the cluster. (Kubernetes Authors 2019.)

3.6.3 Volumes

Because containers are ephemeral, changes to the files inside containers are lost at termination. Volumes allow persistent storage of the files. The volumes are mounted on the pods and keep their state even if the pods are removed, exception being the emptyDir type.

The volume type sets requirements for the storage provider. For example, emptyDir type, that resets when the pod is removed, is provided by the node, but nfs type requires an NFS server to provide volumes for mounting. The Kubernetes provides an abstraction for storage provider and consumer through PersistentVolume objects. ConfigMaps are also mounted to the pods as volumes. (Saito et al. 2016, 87-88; Eldridge I. 2018.)

3.6.4 Deployments

The deployments are Kubernetes controllers, an abstraction layer for Kubernetes Replicasets that handles the creation and replication of pods using deployment definition. Pods defined without a Replicaset are not handled automatically by the Kubernetes and in case of a node failure does not migrate the pods to a healthy node. (Kubernetes Authors 2019.)

The deployments are configured using YAML -formatted files. The configurations consist of two parts. A header part which contains the configuration file definitions and metadata. And spec part which defines the Replicaset options and a template. The template inside the spec part defines the pod configuration. The template includes definitions like affinity, volumes, containers, configMaps etc. (Kubernetes Authors 2019.)

3.6.5 Ingresses

Kubernetes Ingress is an API to map HTTP and HTTPS requests to services inside the cluster. The Ingress services are exposed as NodePorts on each node and a proxy is required to route the external network traffic to the cluster. Cloud service providers use the LoadBalancers to route the traffic and do not require an external proxy. The Ingresses can route the request based on a target hostname and path. (Kubernetes Authors 2019.)

The Ingress requires an ingress controller to function. Using ingress moves the management of the hosts and the routing rules away from the external proxy and to the cluster level, closer to the actual endpoints. The external proxy is only responsible for the routing of the request to the cluster nodes and securing the external communications to the proxy. (Kubernetes Authors 2019.)

3.6.6 ConfigMaps and Secrets

The ConfigMaps and the Secrets are configuration management and storage solutions provided by the Kubernetes. They store values as a key-value pairs. The ConfigMaps can store the configurations for the applications running inside the containers. The ConfigMaps separate the configurations from the container images and centralize the configuration management. Both are mapped as volumes to the pods. The containers inside the pods can then mount them as files or directories. (Kubernetes Authors 2019.)

The Secrets are like ConfigMaps but are meant to be used to store credentials and other sensitive data. Secrets are not stored in a more secure way compared to the ConfigMaps. The difference is in applying changes to a deployment object: Secrets are reapplied always while ConfigMaps are not. (Kubernetes Authors 2019.)

4 DOCKER

The Docker is a software container technology. Docker containers are software containers that use the Docker's libcontainer instead of the Linux containers. Docker ecosystem is a collection of multiple tools (Figure 5) which provide services such as creating and sharing container images, and running and hosting containers. The Docker also provides an APIs to manage the images and the containers. (Krochmalski 2016, 7.)

Docker Desktop is a free development environment for creating containerized applications. It supports Windows and OS X. It integrates Docker Engine, Docker Compose and includes access to the Docker Hub. (Docker Inc. 2019a.)

4.1 Docker engine

The Docker Engine is a container hosting platform. Docker Engine uses Docker Images that are the base for the containers. The images include all the dependencies of a containerized application. Docker Images support the sharing of dependencies through layers. (Docker Inc. 2019b; AeonLearning 2017.)

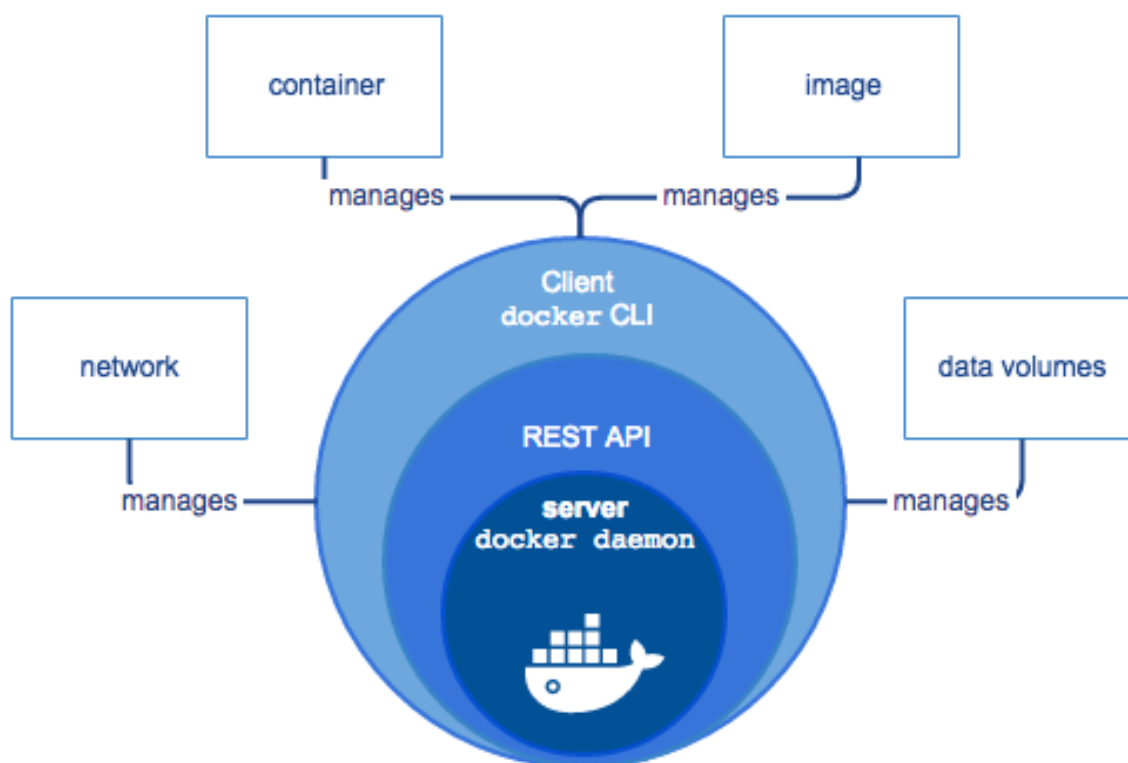


Figure 5. Docker Engine (Docker Inc. 2019c)

The Docker Engine has two different licenses. The Community edition that is free and the Enterprise edition that has certified support and certified Docker Image Registries available. The Community edition of the Docker Engine is only available for the Linux. Enterprise Edition also supports the Windows and Windows Server 2016. The Windows server includes licenses for the Docker Enterprise Edition Basic and is available for no additional costs. (Docker Inc. 2019b.)

The Docker Engine consists of a server process called *dockerd* and a command line interface *docker*. They communicate over a REST application programming interface using Unix sockets or network interfaces. (Docker Inc. 2019c.)

4.2 Docker images

The Docker images are read-only templates that contain the definitions for the Docker containers. The images can be based on other images and usually are. The images are created by defining a Dockerfile that is a text only configuration file with predefined syntax for the instructions. When changing the Dockerfile and rebuilding the image, rebuilds are only needed for the layers that were affected by the change. (Krochmalski 2016, 45-46; Docker Inc. 2019c.)

The layers are readonly except when a container is initialized from an image, then a read and write layer is added on top of the image layers. This layer is only accessible from inside of the initialized container. All readonly layers are shared between multiple containers created from the same Image. The topmost layer is used to store the state of that individual container. Each layer in the image is a set of differences compared to the layer directly below it. (Krochmalski 2016, 47-48; Docker Inc. 2019.)

The layers are shared between the images. Each layer has a unique sha256 digest as an identification, generated from the contents of the layer. Each layer is stored only once and can be part of multiple images. While building the containers the Docker generates intermediate layers and checks the build cache for the existing layers on the host. The layers found in the cache are not rebuilt during the build process to optimize the build time. (Brown 2016; Docker Inc. 2019.)

Modifying the files inherited from the underlying layers inside the running container moves the files to the topmost read and write layer and stores the changed files there. The containers making a large number of changes to the underlying file system grow because of this even if no new files are written by the container. (Docker Inc 2019.)

4.3 Dockerfile and dockerignore

The Dockerfile contains the instructions for the docker build command on how the image will be constructed. Each instruction creates a new layer to the final image. The Docker uses a build context to access the host files during the build process. The build context root is the location of the Dockerfile.

Dockerfile uses an instruction followed by an argument format with a #-character denoting the comment lines. The file is executed from the top to bottom in order. The instructions are not case-sensitive, but the preferred practice is to uppercase instructions to separate them from the arguments. Dockerfile begins with a FROM instruction defining the base image, the word scratch is reserved to denote creation of an image without a base image.

A screenshot of a code editor showing a Dockerfile named 'Dockerfile.dockerfile'. The file contains 12 lines of instructions. Line 1: FROM php:7.3-fpm-alpine. Line 2: (empty). Line 3: RUN mv "\$PHP_INI_DIR/php.ini-development" "\$PHP_INI_DIR/php.ini". Line 4: (empty). Line 5: COPY . /app. Line 6: (empty). Line 7: ENV APP_ENV=dev. Line 8: (empty). Line 9: EXPOSE 8000. Line 10: (empty). Line 11: CMD "php /app/bin/console server:run". Line 12: (empty).

```
1 FROM php:7.3-fpm-alpine
2
3 RUN mv "$PHP_INI_DIR/php.ini-development" "$PHP_INI_DIR/php.ini"
4
5 COPY . /app
6
7 ENV APP_ENV=dev
8
9 EXPOSE 8000
10
11 CMD "php /app/bin/console server:run"
12
```

Image 2. Dockerfile example

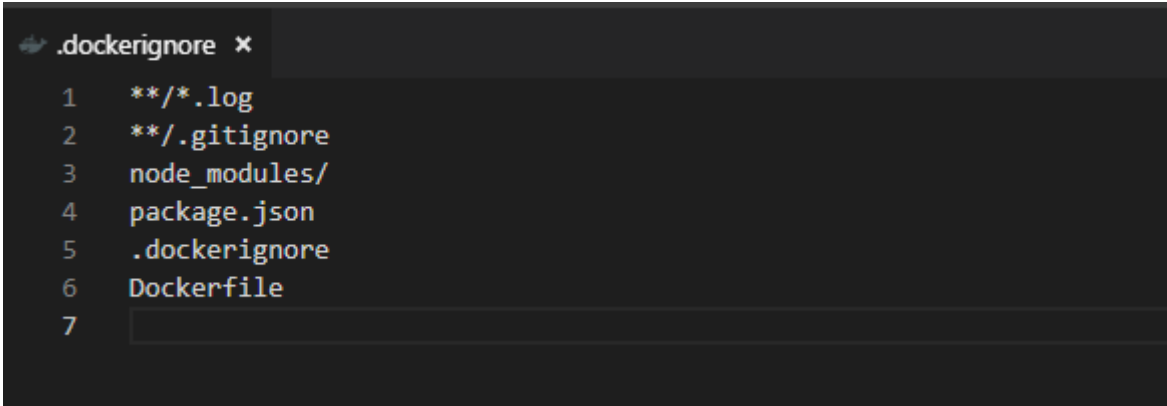
In the example Dockerfile (Image 2) the base image is PHP-FPM image using Alpine Linux distribution as its core. RUN command at row 3 executes a move command inside the image to enable the development configuration. On row 5, COPY instruction copies the contents of the build context directory from the host to the image in directory named app.

The ENV instruction on row 7 sets an environment variable that is available when the container is run. The APP_ENV instructs the Symfony framework to use the development mode.

The EXPOSE instruction informs that port 8000 is to be published on the host machine. The CMD is the command executed when the container is started. This command will instruct the Symfony framework to start the internal PHP web server.

The Dockerignore file is used to exclude files and directories from the build context. There are two practices in defining the excluded files and directories. The first is to directly define the files and directories to be excluded. This has a risk of leaking sensitive data or credentials to the final image. The other is to first define exclusion of everything and then including files and directories required by the final image.

The Dockerignore file does not work in a multistage Dockerfile when copying contents from a stage to another. It only operates on ADD and COPY commands when referencing the contents in the build context.



```
.dockerignore x
1  **/*.log
2  **/.gitignore
3  node_modules/
4  package.json
5  .dockerignore
6  Dockerfile
7
```

Image 3. Dockerignore example

Dockerignore file in the example (Image 3) shows the syntax of excluding the log and gitignore files in any directory of the source, the double asterisk denotes the any depth level. Also excluded are the node_modules directory at root, / forward slash is added for readability to denote that a directory is excluded. Also excluded are the Dockerfile and dockerignore files themselves. (Docker Inc. 2019.)

4.4 Docker container

The Docker container is a runnable instance of a Docker Image. The containers are created from the images and can have additional container specific instructions, for example a network configuration and a mounted storage. Multiple instances of containers can be created from a single image. (Docker Inc. 2019c.)

The container runs only as long as the command defined by the Dockerfile instruction is running. Containers use Linux namespaces to isolate the container from the host operating system and other containers. (Docker Inc. 2019.)

4.5 Docker registry

The Docker Registry is a repository technology to store and distribute the Docker Images. The images are distributed to either public or private repositories. The Docker Hub is a public registry that anyone can use. The Docker's default configuration is to use Docker Hub for the Docker Images. (Docker Inc. 2019c; AeonLearning 2017.)

Docker Incorporated provides a container image to setup a private registry. The private registry also requires a creation of an SSL certificates to secure the connection between the hosts and the registry. The registry also supports multiple different authentication methods. (Docker Inc. 2019.)

4.6 Docker volumes and storage drivers

The volumes are mounted directories or files from the host system. The data written by a container to the mounted volume is persisted even if the container is deleted from the system. Volumes can be shared between multiple containers. Access to the volumes bypasses the storage driver and instead uses direct host access. (Docker Inc. 2019.)

The storage drivers handle reads and writes of the Docker images. The drivers work on top of Linux filesystems and are dependent on the kernel support. Windows and Mac have their own versions of the storage drivers that cannot be changed. The drivers abstract the access to the image layers stored inside different directories. (Docker Inc. 2019.)

The default storage driver is the OverlayFS (overlay2) from Docker version 18.09.0 onwards. Version 18.06 and older use AUFS (a union file system) as default. The Overlay2 is more performant compared to AUFS. (Docker Inc. 2019.)

Selection of the storage drivers is also dependent on the kernel version and the image storage location's filesystem format. The driver selection in order of preference for the filesystem format is: Btrfs, ZFS, Overlay2, AUFS, Overlay, Devicemapper and VFS. Supported formats for Overlay and AUFS are XFS or Ext4. (Docker Inc. 2019; Docker Inc. 2019e.)

5 ARCHITECTURE

The new architecture is built on multiple virtual machines provided by a third-party hosting service. Important properties for virtual machines are a secure and fast network connection between hosts and the possibility to vertically scale the virtual machines.

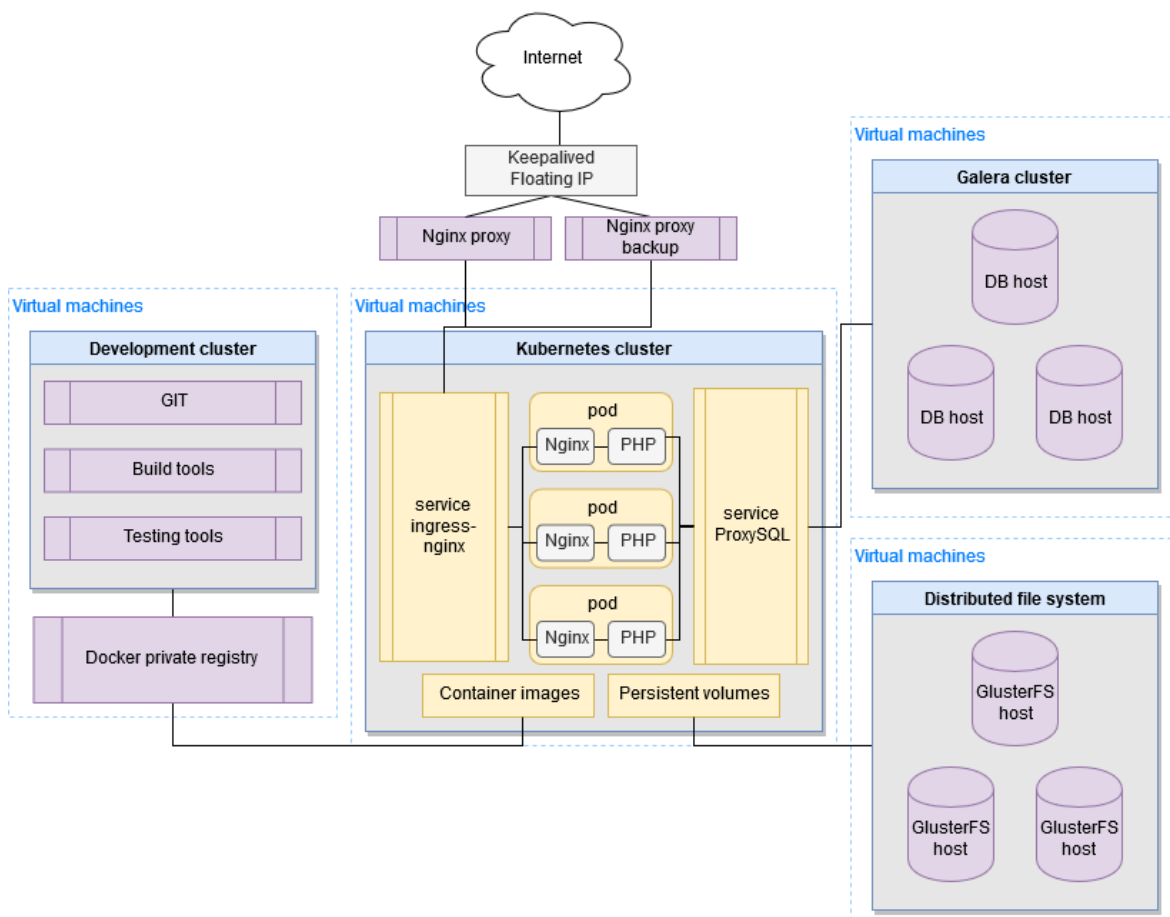


Figure 6. Architecture

The architecture in Figure 6 is divided into multiple clusters:

1. Kubernetes cluster providing the application pods. Minimum of 2 pods per client, which must be distributed to different nodes.
2. Galera cluster providing databases for applications running on the Kubernetes cluster.
3. Distributed file system using GlusterFS as high availability binary storage for data not stored in the database.
4. Testing and development cluster.

The Kubernetes cluster runs multiple containers of the same base application, customized for the client using environment variables. The application is a PHP language based monolithic stateless web application. The application container is built upon a PHP-FPM image and the container images are loaded from a private image registry.

5.1 Load balancing

Access to the Kubernetes cluster is handled via a Nginx proxy with a passive backup in case the main proxy has a failure. This also requires support for a floating IP from the datacenter provider. The Nginx proxy routes the traffic to all worker nodes in the Kubernetes cluster, balancing requests to nodes with the least amount of traffic. Every worker node has an ingress controller for routing the external traffic to their target services inside the cluster. Traffic from the services to the pods is handled by the kube-proxy component that by default uses a round-robin load balancing algorithm.

Pod distribution inside the cluster requires configuration to balance multiple replicas of the same pod between different nodes. By default, the kube-scheduler component chooses the nodes during a spin-up, preferring the nodes with most computational resources available, and tries to distribute them to multiple nodes. There are configuration rules PodAffinity and PodAntiAffinity that can enforce distribution of pods between multiple nodes even in case of a node failure. Normally during node failure, all the running pods from the failed node are distributed to healthy pods after a grace period. After the node recovers, the pods are not distributed again and in the long run this will cause imbalance inside the cluster.

ProxySQL is configured to distribute reads between multiple nodes inside the Galera cluster but writes are targeted to a single node to avoid a deadlock state. Another writer node is selected from all other nodes in case the writer node has a failure.

5.2 Scaling

Databases are scaled by adding more nodes to the Galera cluster. Adding nodes is simplified by having a virtual machine template with required software preinstalled. The database joins the cluster automatically by inserting the correct configuration through virtual machine initialization script. Additional configuration is done to the ProxySQL by adding the new server to the list of available servers.

Kubernetes is horizontally scaled by adding more worker nodes to the cluster. Worker nodes are created from the virtual machine template and joined to Kubernetes by running

the `kubectl join` command. After this the Kubernetes master redistributes current pods between all the nodes.

The Application and service containers are scaled inside the Kubernetes cluster by issuing commands to the Kubernetes scheduler and changing the number of replicas. The number of application containers is based on the size of the client organization.

5.3 Availability

High availability is achieved by having no single point of failure inside the architecture. Every component should have at least another failover component and preferably the ability to balance the load between components.

The Galera cluster is fail-safe between all nodes, and a single surviving node in case of failure can still serve requests and act as a point of recovery for resurrected nodes. In case of total failure where all database nodes go down, recovery is made by determining the node with the latest changes and using that node as a base. This is a catastrophic failure and there would be some downtime.

ProxySQL runs as a service inside the Kubernetes cluster, and load balancing and failovers are handled by Kubernetes. Configuration is handled via Kubernetes ConfigMap and the single endpoint address is pointed for the service with ClusterIP, which all application pods can then utilize as their database host.

Application pods are configured to prioritize distribution to multiple nodes through use of PodAffinity rules providing protection from node failures. Routing for them is provided by Kubernetes services configured as ClusterIPs. Service discovery for new replicas of the pods is handled by Kubernetes through Kubernetes service Selector and pod MatchSelector definitions. These matches allow Kubernetes to automatically generate new endpoints as the number of pod replicas changes.

5.4 Data persistence

Database data is persisted inside each node of the Galera cluster. Each node is a virtual machine and they are connected to each other via virtual network and distributed to multiple datacenters for high availability.

Binary files uploaded by the clients are stored inside GlusterFS mounts provided to application containers as Kubernetes volumes. GlusterFS is distributed to multiple datacenters.

Application sources and container images are stored inside the development cluster. The Source codes are stored in a private Git server and images are stored in a private image repository.

5.5 Deployment

Deployment pipeline is semi-automated. Provision of new clients is handled by internal admin tools that initialize the database and update the ProxySQL cluster, reserve disk space from the binary storage and generates Kubernetes deployment, ingress and service configurations. The API for the DNS management is provided by third party and the SSL certificates are provided by Let's Encrypt.

The private registry for the Docker images is hosted on the development cluster using a Docker registry with Nginx reverse proxy. The images are built using multistage Dockerfiles that have PHP-FPM base image and a build stage for the Composer dependency installation and the WebPack, building the frontend JavaScript code. And a cleanup stage where all the files and directories not required by the running container are stripped from the image. Dockerfile versions are stored inside a separate configuration repository on private Git server.

The application image does not include any client configurations and are stateless. All the client configurations that are required for operation are made in the Kubernetes using configMaps. These include for example the database connection configuration, location of the binary file storage, JWT-certificates and configurations etc. All of these are defined as container environment variables.

5.5.1 Continuous Integration

Code changes are tested locally before being pushed to the Git repository. The integration testing tools are provided by the Symfony framework that uses PHPUnit testing environment as a base. The tests make web-client requests to the application and then asserts the responses. Assertions check for valid responses, changes in the database state and performance of the request.

For testing mock database is generated and rollback of the transactions are used to speed up the tests. The transaction rollback restores the state of database, this is preferred alternative to recreating the database between each test. Transaction rollback ability is provided by the Doctrine ORM.

The responses are asserted for a valid HTTP status codes and a content. Symfony mock client has a crawler for content that can also follow links and mock form submissions. The database changes are validated by querying the state of tables changed by the request. The performance is measured through profiler provided by Symfony. Most important metrics are the response time, database query count and query time.

5.5.2 Continuous Delivery

New features and updates are integrated as soon as all tests have passed, and the feature is mature enough. Updates are delivered via container image registry. The application image is updated, and Kubernetes is triggered to update all the containers.

Database migrations cause minor slowdown by limiting containers which serve the client organization to one. After the migration has succeeded container replica count is restored.

6 CONFIGURATIONS

The configuration is a file-based system where images are described in Dockerfiles and the Kubernetes cluster is defined through yaml-formatted files. All files are stored in a Git repository for version control and documentation of the changes. In the Kubernetes configuration the focus is on separating client organization variables from the container images and routing the external requests to the right pods using Ingress and Kubernetes services.

6.1 Private image registry

A private image registry is used to store the Docker images of the application. The primary reason to use a private registry is to prevent outside access to the source code. If the application image was uploaded to a public registry, anyone could download the image and access the code.

Installation package requirements consist of:

- docker
- docker-compose
- nginx
- apache2-utils
- SSL certificate

Nginx is configured as proxy using an SSL certificate signed with a self-signed CA root certificate. The CA root certificate is distributed to every node in the Kubernetes cluster and added to a trusted certificates storage. Authentication is handled with HTTP basic authentication.

```

reverse-proxy.cnf x
1  server {
2      listen 443;
3      ssl on;
4      ssl_certificate /data/certs/registry.crt;
5      ssl_certificate_key /data/certs/registry.key;
6
7      location / {
8          # Block old docker client
9          if ($http_user_agent ~ "^(docker\/1\.3|4|5(?:!\. [0-9]-dev))|Go ).*$") {
10             return 404;
11         }
12         proxy_pass http://localhost:5000;
13         proxy_set_header Host $http_host;
14         proxy_set_header X-Real-IP $remote_addr;
15         proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
16         proxy_set_header X-Forwarded-Proto $scheme;
17         proxy_read_timeout 900;
18     }
19 }
20

```

Image 4. Nginx reverse proxy

In the Nginx configuration (Image 4), row 9 has regex filter for User-Agent HTTP header. This rule is used to prevent communication from the older, incompatible version of the Docker engine. A private registry requires Docker 1.6.0 or higher version (Docker Inc. 2019).

```

docker-compose.yml ●
1  version: '3'
2
3  services:
4      registry:
5          image: registry:2
6          ports:
7              - "5000:5000"
8          environment:
9              REGISTRY_AUTH: htpasswd
10             REGISTRY_AUTH_HTPASSWD_REALM: Registry
11             REGISTRY_AUTH_HTPASSWD_PATH: /auth/registry.password
12             REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY: /data
13          volumes:
14              - ./registry_auth:/auth
15              - ./registry_data:/data
16          restart: always
17

```

Image 5. Docker Compose configuration

The Docker Compose configuration (Image 5) uses the environment variables, rows 9 through 12, to configure the behavior of the registry. Rows beginning with `REGISTRY_AUTH` are used to configure the authentication type and location of the security file. `REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY` is the root path for the binaries of the docker images.

```
! registry-secret.yml x
1  apiVersion: v1
2  kind: Secret
3  metadata:
4    name: registry-key
5    namespace: example
6  data:
7    .dockerconfigjson:
8      ewoJImF1dGhzIjogewoJCSJyZWdpc3RyeS5leGFtcGxlLmNvbSI6IHsKCQkKJImF1dGgiOiAiWlho
9      aGJYQnNaUW89IgoJXCX0KCX0sCgkiSHR0cEhlYWRLcnMiOiB7CgkKJlVzZXItQWdlbnQiOiAiRG9j
10     a2VyLUNsaWVudC8xO54wMy4yIChsakW51eCkiCgl9Cn0K
11  type: kubernetes.io/dockerconfigjson
```

Image 6. Kubernetes private registry secret

Use of a private registry in Kubernetes requires creation of a secret for Kubernetes (Image 6). The secret can then be referenced in deployment configurations. For Docker registries, the `dockerconfigjson` field must contain base64 encoded contents of a `config.json` file (Image 7). The `type` field defines the type of the secret. Kubernetes requires this information to decode the contents correctly.

```
{ } config.json x
1  {
2    "auths": {
3      "registry.example.com": {
4        "username": "ZXhnbXBsZQo="
5      }
6    },
7    "HttpHeaders": {
8      "User-Agent": "Docker-Client/19.03.2 (linux)"
9    }
10 }
11
```

Image 7. Contents of `dockerconfigjson`

In config.json (Image 7) the password on row 4 is not encrypted. It is a base64 encoded plain password. The format is a username password key-value pair. The hostname for the registry is used in reference to the image in the deployments. That way Kubernetes can match the secrets and image registries.

6.2 Nginx proxy

Nginx proxy is the connection layer for the clients. The proxy routes and balances the traffic between nodes in the cluster. The proxy also has the SSL certificates to secure the client connections to the proxy. After the proxy, all internal requests are not encrypted.

```
proxy.cnf x
1  upstream backend {
2      least_conn;
3      server 192.168.203.195:31918;
4      server 192.168.203.196:31918;
5  }
6
7  server {
8      location / {
9          proxy_pass http://backend;
10         proxy_set_header Host $host;
11         proxy_set_header X-Forwarded-For $remote_addr;
12     }
13
14     listen 443 ssl default_server;
15     ssl_certificate /etc/letsencrypt/live/example.com/fullchain.pem;
16     ssl_certificate_key /etc/letsencrypt/live/example.com/privkey.pem;
17     include /etc/letsencrypt/options-ssl-nginx.conf;
18     ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem;
19 }
20
21 server {
22     listen 80;
23     return 301 https://$host$request_uri;
24 }
25
```

Image 8. Nginx proxy configuration

In the Nginx configuration (Image 8) row 1 is the block, which defines the backend servers. Row 2 has the load balancing algorithm definition “least_conn;” This algorithm routes traffic to a server with the least active connections and presumably the lightest workload.

Rows 3 and 4 are the IP addresses of the worker nodes in the Kubernetes cluster. Port number is the ingress-nginx service’s NodePort number. Image 9 shows the kubectl

command to query the services in the ingress-nginx namespace. The ports column on row 3 displays the port mappings in internal:external format. Internal port 80 is mapped to external port 31918.

```

1  joni@master:/data/k8s$ kubectl get service -n ingress-nginx
2  NAME          TYPE          CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
3  ingress-nginx NodePort      10.97.247.229 <none>         80:31918/TCP,443:32042/TCP 24d
4

```

Image 9. Ingress service

From row 8 onwards in Image 8 is the location block which routes all the traffic to the backend nodes and sets Host and X-Forwarded-For headers. Host name is the requested DNS name, and this is required to route requests to correct pods in the Kubernetes ingress. X-Forwarded-For is the client IP address the request originated from.

Rows 14 through 18 define the SSL certificate configuration. Row 21 server block defines the redirection of HTTP connections to HTTPS.

6.3 Kubernetes cluster

Kubernetes cluster consists of multiple Ubuntu 18.04 LTS based nodes with following requirements:

- swap disabled
- iptables rule
- kubelet
- docker

Figure 7 shows the route taken by an external request to reach the pods inside the cluster. The requests are routed to the Ingress service which has specific rules to choose the destination service or pod. The Ingress to application service route is configured via an API extension that has a spec field for the Nginx specific rules.

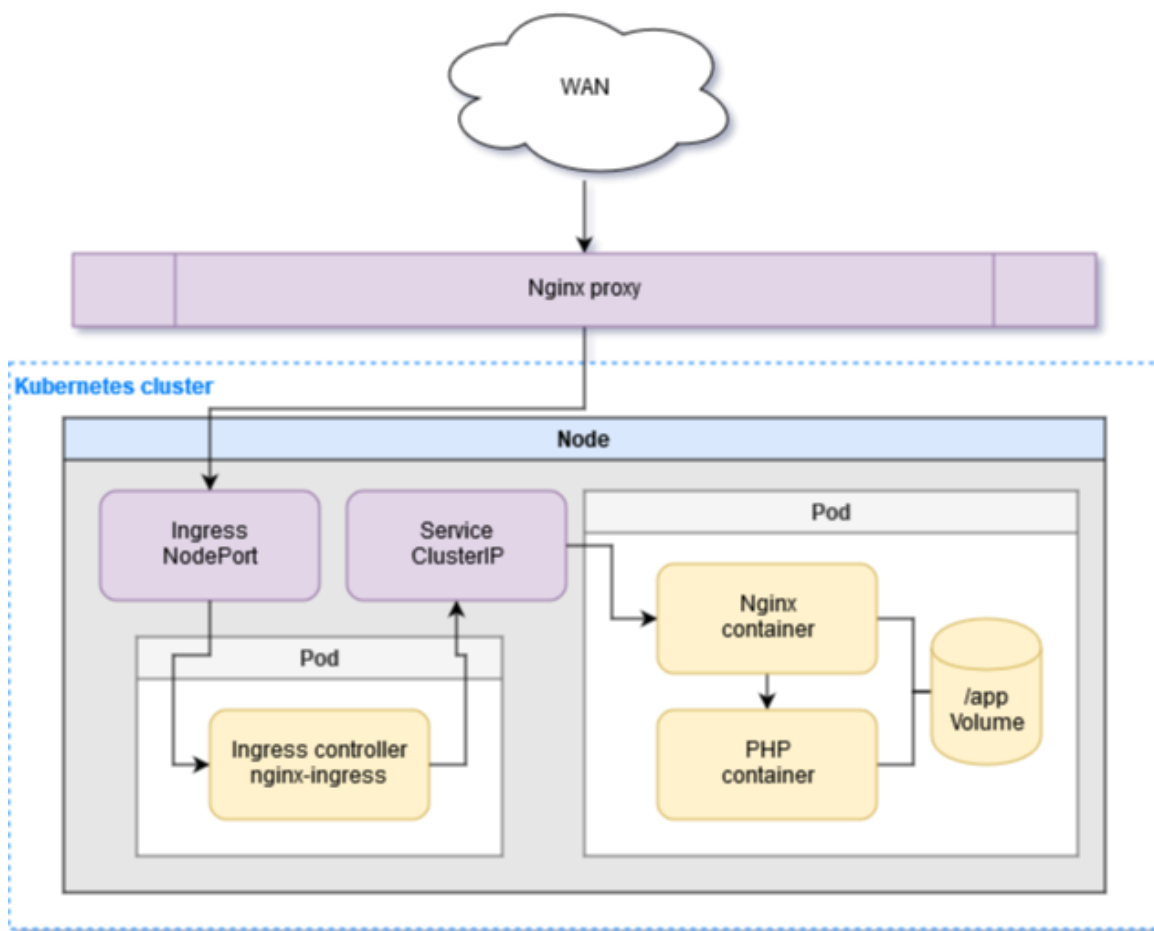


Figure 7. External request

```
! ingress-example.com.yml x
1  apiVersion: extensions/v1beta1
2  kind: Ingress
3  metadata:
4    name: ingress-example.com
5    namespace: exmaple
6    annotations:
7      ingress.kubernetes.io/ingress.class: "nginx"
8  spec:
9    rules:
10   - host: example.com
11     http:
12       paths:
13         - path: /
14           backend:
15             serviceName: service-example
16             servicePort: 80
17
```

Image 10. Application ingress configuration

The application ingress configuration (Image 10) requires an annotation "ingress.kubernetes.io/ingress.class" to set the type of the ingress to which the rules

inside the spec field are applied. Rows 9 to 16 define the rules to route the requests with hostname example.com to the Kubernetes service named service-example using port 80. Row 10 is the target hostname of the request. Rows 11, 12 and 14 are Nginx specific fields. Row 13 defines the path, the forward slash '/'-character denotes all paths. Rows 15 and 16 define the service name and the port.

```
! example-service.yml x
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: service-example
5    namespace: example
6  spec:
7    selector:
8      app: example
9    ports:
10     - protocol: TCP
11       port: 80
12
```

Image 11. Application service configuration

The application service (Image 11) is a layer that routes the requests to the pods inside the cluster. the pods that have metadata labels which match the key-value in the service's selector field (row 8). The endpoints are automatically created for each pod matched via the selector and the requests are balanced between multiple endpoints with round-robin algorithm.

The PHP application pod consists of two containers with a shared data volume:

- Nginx container as a reverse proxy for the PHP-FPM container and as a static content host.
- PHP-FPM container which executes the application scripts.

```

! example-deployment.yml ✕
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: example-app
5    namespace: example
6  spec:
7    replicas: 2
8    selector:
9      matchLabels:
10     app: example

```

Image 12. Deployment metadata

Image 12 shows the basic metadata for a Kubernetes Deployment. Row 7 defines the number of pod replicas this deployment will have and row 8 defines the selector used to find the pods which belong to this deployment.

```

! example-deployment.yml ✕
11  template:
12    metadata:
13      labels:
14        app: example
15    spec:
16      affinity:
17        podAntiAffinity:
18          preferredDuringSchedulingIgnoredDuringExecution:
19            - weight: 100
20              podAffinityTerm:
21                topologyKey: kubernetes.io/hostname
22                labelSelector:
23                  matchExpressions:
24                    - key: app
25                      operator: In
26                      values:
27                        - example
28      imagePullSecrets:
29        - name: registry-key

```

Image 13. Deployment template metadata

Deployments have a template that defines the structure of the pods managed by this deployment. Important part of the template metadata is to have at least one label (row 14) which matches the selector matchLabels definition of the deployment spec (Image 12, row 10).

Row 17 in Image 13 has a podAntiAffinity rule to distribute the pods managed by this deployment to different nodes. If the node has a failure and the pods are redistributed across healthy nodes, the affinity rules are used to distribute the pods to different nodes and keep the cluster balanced. If no affinity rules are defined all the pods in the service might end up running on a single node.

Row 28 in Image 13 defines the Kubernetes secret for accessing the private registry. This secrets' configuration is in Image 6. Multiple ImagePullSecrets can be configured.

```
! example-deployment.yml x
30     volumes:
31     - name: app-root
32       emptyDir: {}
33     - name: config-volume-nginx
34       configMap:
35         name: config-nginx-example
36     - name: config-volume-php
37       configMap:
38         name: config-php-fpm-example
39     - name: secret-volume-jwt
40       secret:
41         secretName: jwt-example.com
```

Image 14. Deployment template volumes

Volumes available for the containers are configured in Image 14. Row 31 configures an empty volume that will be used to share the application files between the both containers. Rows 33, 36 and 41 define the volumes using configMaps and secret. These are then defined in Image 15 as mounts on the containers.

```

! example-deployment.yml x
42     containers:
43     - image: example-registry/example-app:0.1.1
44       imagePullPolicy: IfNotPresent
45       name: php-fpm
46       resources:
47         requests:
48           memory: "256Mi"
49           cpu: "100m"
50         limits:
51           memory: "640Mi"
52           cpu: "500m"
53       volumeMounts:
54       - name: app-root
55         mountPath: /app
56       - name: config-volume-php
57         mountPath: /usr/local/etc/php/conf.d/app.ini
58         subPath: app.ini
59       - name: config-volume-php
60         mountPath: /usr/local/etc/php/conf.d/symfony.ini
61         subPath: symfony.ini
62       - name: secret-volume-jwt
63         mountPath: /jwt
64       lifecycle:
65         postStart:
66           exec:
67             command: ["/bin/bash", "-c", "/lifecycle/postStart.sh"]
68       envFrom:
69       - configMapRef:
70         name: env-example.com

```

Image 15. Deployment template containers

The application container configuration (Image 15) uses the application image from the private registry. The version number is tagged to the image during the build stage. ImagePullPolicy IfNotPresent checks the nodes image storage before downloading the image from the registry.

Memory and CPU time are limited based on the current hardware and approximated number of concurrent requests per customer. The application's average memory consumption per request is about 80 Megabytes of memory and one container can serve about eight concurrent connections at any given time before reaching the memory limit.

Shared volume for the Nginx container is mounted to path /app. Two different PHP configuration files are mounted from the PHP configMap volume. Both are mapped to PHP conf.d directory. Mount on row 62 is the JSON web tokens volume containing the client specific encryption certificates. Application locates these certificates using the environment variables configured in the Image 21 rows 12 and 13.

The lifecycle field on row 64 to 67 defines the script file to be executed on the container initialization. The poststart.sh script (Image 18) is included to the container image during

the image build stage. Rows 68 to 70 defines the configMap from where to load the environment variables.

```
! example-deployment.yml x
71     - image: nginx:1.17.4-alpine
72       name: nginx
73       resources:
74         requests:
75           memory: "64Mi"
76           cpu: "100m"
77         limits:
78           memory: "128Mi"
79           cpu: "500m"
80       volumeMounts:
81         - name: app-root
82           mountPath: /app
83         - name: config-volume-nginx
84           mountPath: /etc/nginx/conf.d/example.conf
85           subPath: example.conf
86
```

Image 16. Nginx container definition

The Nginx container configuration (Image 16) for the reverse proxy is very basic. The memory and CPU time requirements and limits are low because these proxies will only serve requests coming from a single client organization and all requests are shared between multiple replicas.

Volume mounts are the shared volume between the Nginx container and the application container, mounted to /app. And the Nginx configMap which maps the custom Nginx configuration to the Nginx conf.d directory.

```
kubectl -n example create secret generic jwt-example.com
--from-file=private=private.pem --from-file=public=public.pem
```

Image 17. Command to create secret

The Kubernetes Secret containing the private and public keys for the JSON web tokens is generated with the command in Image 17. The n flag sets the namespace to example. Commands create, secret, and generic are instructions for the kubelet. Jwt-example.com is the name of the new secret. --from-file defines the key-value pair for the secret. First is the key followed by a value. Example creates two key-values pairs:

- private: private.pem contents encoded with base64.

- public: public.pem contents encoded with base64.

```
postStart.sh x
1  #!/bin/bash
2  cp -r /src/. /app
3  mkdir /app/var
4
5  php /app/bin/console assets:install
6  php /app/bin/console cache:clear
7
8  chown www-data:www-data -R /app
9  chmod 777 -R /app/var/log
10  chmod 777 -R /app/var/cache
11
```

Image 18. postStart.sh script

The script in Image 18 is the post start script for the application container. The script is executed after the container has initialized and is used to finalize the initialization of the container image. In row 2 the script copies the application source code to the shared volume. This shared volume is accessed by both the application container (PHP-FPM) which executes the PHP scripts and the Nginx container serving static content. Additional initializations from row 5 onwards are instructions for the Symfony framework to install static assets, initialize the internal cache and set folder permissions.

The container is in initializing state until the post start script has completed. Errors in the post start script will prevent the container from entering the running state. Lifecycle scripts should be kept as light as possible.

```
! example-app-nginx.yml x
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: config-nginx-example
5    namespace: example
6  data:
7    example.conf: |
8      server {
9        listen 80 default_server;
10       server_name _;
11
12       root /app/public;
13       location / {
14         try_files $uri /index.php$is_args$args;
15       }
16
17       location ~ ^/index\.php(/|$) {
18         fastcgi_pass localhost:9000;
19         fastcgi_split_path_info ^(.+\.(php|\.*)$);
20         include fastcgi_params;
21
22         fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;
23         fastcgi_param DOCUMENT_ROOT $realpath_root;
24         internal;
25       }
26
27       location ~ \.php$ {
28         return 404;
29       }
30     }
31
```

Image 19. Nginx configMap

The Nginx configMap (Image 19) configures the Nginx to function as a reverse proxy for the PHP application. The Nginx also serves static contents such as CSS and JavaScript files. Rows 17 to 25 has the reverse proxy configuration which redirects the requests to localhost:9000 address. Because the container that provides the PHP-FPM is running inside the same Kubernetes pod, they can access each other through the localhost name and a port number.

```
! example-app-php.yml ●
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: config-php-fpm-example
5    namespace: example
6  data:
7    symfony.ini: |
8      opcache.memory_consumption=256
9      opcache.max_accelerated_files=20000
10     opcache.validate_timestamps=0
11     realpath_cache_size=4096K
12     realpath_cache_ttl=600
13    app.ini: |
14     upload_max_filesize=32M
15     post_max_size=33M
16     date.timezone=Europe/Helsinki
17
```

Image 20. PHP configMap

ConfigMap for the PHP settings (Image 20) is split to two separate configurations. Symfony.ini contains all the optimizations for the Symfony framework. And app.ini has customer specific settings. In this case are specified a maximum file upload size and a time zone for the application.

```
! example-app-env.yml ●
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: env-example.com
5    namespace: example
6  data:
7    APP_ENV: "prod"
8    APP_SECRET: "3aae5956c070a665bc3f7feabe23cddb"
9    DATABASE_URL: "mysql://example:example@service-proxysql:6033/example"
10   ORGANIZATION: "Ab Example Oy"
11   FILE_UPLOAD_ROOT: "/files"
12   JWT_SECRET_KEY: "/jwt/private"
13   JWT_PUBLIC_KEY: "/jwt/public"
14   JWT_PASSPHRASE: "example"
15   JWT_TTL: "3600"
16   JWT_HTTPS_ONLY: "true"
17
```

Image 21. Application environment configMap

Client-based variables and configurations are mapped to clients' containers via ConfigMaps. Image 21 presents an example client configMap for the application (PHP) container.

Rows 7 to 9 define environment variables for the Symfony framework: the running level of the application, hash for security functions and database connection URL. Database connection describes database type followed with credentials, server and database name. On row 9 the service-proxysql is the ProxySQL Kubernetes service which provides the database access. Rows 10 and 11 have variables used by the application and rows 12 to 16 have variables for the JSON authentication library.

```

1  FROM composer:1.9.0 AS build
2
3  RUN apk add --no-cache yarn
4
5  COPY ./src /src
6
7  WORKDIR /src
8
9  RUN composer install --no-dev \
10     && composer dump-autoload --no-dev --optimize --classmap-authoritative \
11     && yarn install \
12     && yarn encore production
13
14  # clean source
15  RUN rm -rf \
16     assets \
17     .git \
18     .idea \
19     tests \
20     node_modules
21  RUN rm -f \
22     .env.test \
23     .gitignore \
24     package.json \
25     phpunit.xml.dist \
26     README.md \
27     webpack.config.js \
28     yarn.lock
29
30  # App image
31  FROM php:7.2-fpm-alpine
32
33  RUN set -xe \
34     && apk add --no-cache bash icu-dev \
35     && docker-php-ext-configure opcache --enable-opcache \
36     && docker-php-ext-install pdo pdo_mysql intl opcache \
37     && mv "$PHP_INI_DIR/php.ini-production" "$PHP_INI_DIR/php.ini"
38
39  COPY --from=build /src /src
40  COPY ./lifecycle /lifecycle
41
42  EXPOSE 9000
43

```

Image 22. Application Dockerfile

The application image does not include any client specific configurations or variables. The image has only base code for the application to function and all customizations are made through environment variables or mounted config maps.

Dockerfile for the application (Image 22) is multistage to separate the build stage from the final runnable image. The Build stage is an image of Composer package manager from the public Docker registry. Build stage executes the Composer to install all the external PHP library requirements of the application.

The build stage also installs Yarn, a JavaScript library package manager, to install all modules required by the application's frontend JavaScript. Yarn also executes Symfony framework's Encore WebPack wrapper, which builds and transpiles the ES6 JavaScript code to a more widely supported ES5 code.

After the build stage has completed, all unnecessary files are removed from the directory used as the base for the application. This is done in order to reduce the size of the final image. In Image 22 the removal of directories and files are separated for readability.

The application image is based on Linux running the Alpine distribution, with PHP FastCGI Process Manager and its requirements pre-installed. The Alpine distribution is selected because of its small size. The Php-fpm Docker image also includes hooks for customizing the PHP installation. In Image 22, rows 33 to 37 are used to install and enable PHP-modules required by the application.

Row 39 has the command to copy the built PHP application from the build stage to the final application image. EXPOSE 9000 opens the PHP FastCGI Process Manager's port to handle requests from the Nginx container.

7 SUMMARY

The purpose of this project was to containerize and orchestrate a PHP application using Docker and Kubernetes. Kubernetes has many abstractions to simplify the management of containers. Deployments abstract the pod lifecycle management and Kubernetes services abstract the pod networking.

Container images should be kept as small as possible for fast delivery and to save space on the image registry server. Hard version numbering is preferred instead of using the latest tag, to avoid unwanted upgrades and to allow the possibility of rollback.

The main challenges with Kubernetes came from routing the external traffic to the right pods and calculating and selecting the memory limits for the PHP application pods. For memory limits, overprovisioning is preferable. Implementing automatic horizontal scaling of the pods is also a possibility in the future.

The next steps for the development of the platform are a centralized logging and monitoring solution, automation of customer instance creation and distribution of clusters over multiple different datacenter locations.

Replacement of the Nginx reverse proxy container inside the PHP application pod is also a possibility in the future. For example, Roadrunner or Nginx Unit are promising technologies to make a PHP application handle a large number of traffic by making the application's front memory persistent and offloading requests to child processes similarly to how NodeJS operates.

REFERENCES

- Abd-El-Barr, M. & El-Rewini, H. 2005. Advanced Computer Architecture and Parallel Processing. John Wiley & Sons Inc.
- AeonLearning Pvt. Ltd. 2017. What is Docker Container [referenced 17 Feb 2019]. Available: <https://acadgild.com/blog/what-is-docker-container-an-introduction>
- Brown, N. 2016. Explaining Docker Image IDs [referenced 16 Nov 2019]. Available: <https://windsock.io/explaining-docker-image-ids/>
- Cloud Native Computing Foundation. Members [referenced 9 Mar 2019]. Available: <https://www.cncf.io/about/members/>
- Composer documentation. Composer documentation [referenced 16 Nov 2019]. Available: <https://getcomposer.org/doc/>
- Docker Inc. 2019. Docker documentation [referenced 15 Nov 2019]. Available: <https://docs.docker.com>
- Docker Inc. 2019a. Docker Desktop [referenced 17 Feb 2019]. Available: <https://www.docker.com/products/docker-desktop>
- Docker Inc. 2019b. Docker Engine [referenced 17 Feb 2019]. Available: <https://www.docker.com/products/docker-engine>
- Docker Inc. 2019c. Docker overview [referenced 17 Feb 2019]. Available: <https://docs.docker.com/engine/docker-overview>
- Docker Inc. 2019d. What is a Container [referenced 3 Feb 2019]. Available: <https://www.docker.com/resources/what-container>
- Docker Inc. 2019e. Docker source code: Linux driver [referenced 15 Nov 2019]. Available: https://github.com/docker/docker-ce/blob/19.03/components/engine/daemon/graphdriver/driver_linux.go
- Eldridge I. 2018. What is Container Orchestration. New Relic [referenced 9 Mar 2019]. Available: <https://blog.newrelic.com/engineering/container-orchestration-explained/>
- Gavalda, M. 2019. The Definitive PHP 5.6, 7.0, 7.1, 7.2 & 7.3 Benchmarks [referenced 26 Nov 2019]. Available: <https://kinsta.com/blog/php-benchmarks/>
- Haff, G. 2013. What are containers and how did they come about? [referenced 16 Feb 2019]. Available: <http://bitmason.blogspot.com/2013/09/what-are-containers-anyway.html>

- Liu, H. 2009. Software Performance and Scalability: A Quantitative Approach. Wiley-Blackwell.
- Kerrisk, M. 2010. The Linux programming interface: a Linux and Unix system programming handbook. No Starch Press.
- Krochmalski, J. 2016. Developing with Docker. Packt Publishing.
- Mell, E. 2018. Dive into the decades-long history of container technology. TechTarget [referenced 3 Feb 2019]. Available: <https://searchitoperations.techtarget.com/feature/Dive-into-the-decades-long-history-of-container-technology>
- Nginx documentation. Nginx documentation [referenced 16 Nov 2019]. Available: <https://nginx.org/en/docs/>
- PHP documentation. PHP documentation [referenced 16 Nov 2019]. Available: <https://www.php.net/manual/en/>
- Red Hat Blog. 2015. The History of Containers [referenced 16 Feb 2019]. Available: <https://rhelblog.redhat.com/2015/08/28/the-history-of-containers/>
- Rouse, M. 2018. Container (containerization or container-based virtualization). TechTarget [referenced 3 Feb 2019]. Available: <https://searchitoperations.techtarget.com/definition/container-containerization-or-container-based-virtualization>
- Salehi, S. 2016. Mastering Symfony. Packt Publishing.
- Saito, H., Hsu, C. & Lee, C. 2016. Kubernetes Cookbook. Packt Publishing.
- Symfony SAS. 2019. What is Symfony? [referenced 24 Nov 2019]. Available: <https://symfony.com/what-is-symfony>
- The Kubernetes Authors. 2019. Kubernetes Documentation [referenced 17 Nov 2019]. Available: <https://kubernetes.io/docs/home/>
- The Kubernetes Authors. 2019a. Learn Kubernetes Basics [referenced 9 Mar 2019]. Available: <https://kubernetes.io/docs/tutorials/kubernetes-basics/>
- The Kubernetes Authors. 2019b. Kubernetes Concepts [referenced 9 Mar 2019]. Available: <https://kubernetes.io/docs/concepts/>
- The Kubernetes Authors. 2019c. Cluster Networking [referenced 9 Mar 2019]. Available: <https://kubernetes.io/docs/concepts/cluster-administration/networking/>

The Kubernetes Authors. 2019d. Pod Overview [referenced 9 Mar 20219]. Available: <https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/>