Tampere University of Applied Sciences

Legacy Server Migration

Elias Ylänen

BACHELOR'S THESIS
April 2019

Degree Programme in Business Information Systems
Web Services

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Tietojenkäsittely
Web-palvelut

ELIAS YLÄNEN:
Vanhentuneen serverin päivitys

Opinnäytetyö 33 sivua, joista liitteitä 8 sivu
Huhtikuu 2019

Tämän opinnäytetyön tavoitteena oli siirtää vanhentuneella Red Hat 6 -serverillä sijaitsevat asiakkaan web-palvelut uudelle Ubuntu-pohjaiselle palvelimelle, samalla palveluiden taustalla olevaa arkkitehtuuria modernisoiden ja sovellusten käyttöönottoa yhdenmukaistaen.

Uuden serverin rakentamisessa hyödynnettiin Dokku-nimistä työkalua, jonka avulla suuri osa palvelimen tarpeellisesta konfiguraatiosta ja ohjelmistosta voidaan automatisoida ja keskittää yhden työkalun alaiseksi. Projektin lopputavoitteena oli saattaa mahdollisimman moni olemassaolevista palveluista hyödyntämään Dokkun tarjoamaa kehitysputkea, Git-pohjaisesta käyttöönotosta Docker-pohjaisten, omiksi kokonaisuuksiksi eristettyjen konttien hyödyntämiseen sovellusten ajossa.

Kaiken kaikkiaan projekti kesti noin neljä kuukautta, syyskuusta 2018 tammikuuhun 2019, sisältäen vaadittujen palveluiden asennuksen uudelle palvelimelle, siirrettävien sovellusten asettamisen Docker-kontteihin, kriittisten riippuvuuksien päivittämisen ja aiemmin globaalien ohjelmistojen eristämisen vain niitä hyödyntävien konttien sisään. Palveluiden kehitys–käyttöönotto -putken yksinkertaistamisen ohella projektin merkittävin tulos on yli 100 korjattua tietoturvahaavoittuvuutta.

Tämä opinnäytetyö on kirjoitettu yhteenvetona päivitysprosessista ja tuomaan asiaan vihkiytymättömille ymmärrystä sekä DevOpsin tilasta vuonna 2019 että mitä vaaditaan toimivan palvelimen pystyttämiseen.

Asiasanat: devops, docker, dokku, kontit, palvelin

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Business Information Systems
Web Services

ELIAS YLÄNEN:
Legacy Server Migration

Bachelor's thesis 33 pages, appendices 8 page
April 2019

The aim of this thesis was to migrate a client's services running on an old Red Hat server to a new Ubuntu-based server, simultaneously modernizing the architecture supporting many of the consumer-facing services deployed onto the server and unifying the deployment process of the services to utilize a single pipeline.

To build the new server, a tool called Dokku was used. Dokku is described as a self-hosted PaaS service, which in practice translates to Dokku automating and aggregating most of the configuration and software required to run a functioning server under a single toolkit. The aim of the migration process was to have as many existing services as possible migrated to the Dokku pipeline, including Git-based deploys and running every service inside an isolated Docker container.

All in all, migrating the server lasted for about four months, including installing the required software onto the new server, Dockerizing almost every application that was to be deployed onto the new server, updating critical dependencies, and scoping previously global functionality to be available only to the containers that utilize them. Along with the streamlined development-deployment pipeline, the most significant result of the project is the over 100 fixed vulnerabilities.

This thesis is written as a summary of the migration process and in order to offer insight about the state of DevOps in the year 2019 to people with little to no knowledge of what it takes to build and configure a functioning server.

Key words: containers, devops, docker, dokku, server

CONTENTS

1 INTRODUCTION 7
2 THE ORIGINAL SERVER 8
2.1 Structure 8
2.2 Issues 9
2.2.1 Deployment 9
2.2.2 Security 10
3 SOLUTION 12
3.1 Dokku 13
3.1.1 Deployment 14
3.1.2 Domains 15
3.1.3 Storage 15
3.1.4 Plugins 17
3.1.5 Customizing Nginx 19
3.2 Service-specific tools and cronjobs 20
3.3 Static files 21
4 CONCLUSIONS 23
REFERENCES 25
APPENDICES 26
Appendix 1. List of Dokku plugins 26
Official Plugins (Beta) 26
Community plugins 27
Appendix 2. Dokku - Customizing the Nginx configuration 31
Customizing the Nginx cofiguration 31

GLOSSARY

Apache	A web server software that controls directing requests, executing code and access control within the server. Also referred to as httpd
AWS	Amazon Web Services. Cloud solution suite provided by Amazon
CI	Continuous integration. A practice for speeding up software development and deployment by introducing automated testing, building and possibly even deploying of pushed code (Fowler, Continuous Integration, 2006)
Container	An operating-system-level virtualized environment, that contains the application code and the underlying software required to run the software in an isolated environment on the host machine (Kasireddy, A Beginner-Friendly Introduction to Containers, VMs and Docker, 2016)
Cronjob	An automated task that's run periodically on the system. Ideal for backups and other routine tasks
DevOps	A combination of philosophies that strive to make application delivery more performant and adaptable by, for example, having the same engineer(s) be responsible for the entire application lifecycle from development to deployment (Amazon, What Is Devops?)
Docker	A container management platform

Dockerizing	Wrapping a service inside a Docker container
Docker-compose	A tool for orchestrating building, deployment and inter-linking of multiple Docker containers
Dokku	A tool to centralize and automate controlling Nginx, Docker, and application deployment on the server (Dokku, Getting Started with Dokku)
Drupal	A content management system (CMS)
Environment variables	Shell session-wide values used by application code during build or runtime. An easier alternative to hard-coding e.g. URLs to services used by the application
EOL	End-of-life. When the vendor has stopped supporting the version or product
Fabric	A Python library used as an abstraction for executing shell scripts over SSH (Forcier, Fabric documentation, 2018)
Nginx	A server software alternative to Apache
Quay.io	A private container repository
Sendmail	Software used to send emails from the server via PHP
Vagrant	An application for controlling virtual machines
WordPress	The most popular CMS in the world

1 INTRODUCTION

This thesis has been made as part of an effort to migrate an undisclosed client's public web services to a new, modern server architecture while also unifying the development – deployment cycle of the client's main consumer portal and all services connected to the site.

The main point of the thesis is to describe the server migration process, and due to that focus not much attention is given to the actual services. All the reader needs to know is that the servers consist of the main corporate website of the client and a handful of smaller services that function either independently or are linked to the main website via embedded HTML iframes.

The aim of this thesis is to describe and analyze the tools and methodologies used during the project in a way that makes the process understandable to a reader who might not have intimate knowledge about the server migration process. By reading this thesis, the reader should gain familiarity with the tools and technical terms utilized in modernizing the underlying systems required by the more visible, user-facing services in the modern web landscape, and an understanding of the general architecture of a functioning web server.

Due to the nature of the project, references to the client, actual directory structures and some names have been replaced with placeholders.

2 THE ORIGINAL SERVER

2.1 Structure

Due to development of many of the services predating the inception of containers and the original development team being unable to take advantage of the latest DevOps methods, the deployment state of the server was in a disarray at the time of starting this thesis. While a handful of the newer applications had been Dockerized, older software was either deployed as just static files, or in the case of the main web portal, as a non-isolated Drupal site. Services like these had no isolation and were dependent on the globally installed, usually outdated, services, both of which can become critical vulnerabilities as time goes by.

The upper-level routing solution on the server was Apache – supplemented with local `.htaccess` files when required – which directed the HTTP requests to the service corresponding with the request URL. To generalize, apart from the Drupal-run main website, most of the complementary services were split into separate front and backend services, with the frontend being a directory with static files, and the backend being a Dockerized service with its own internal server software to handle application-specific routing.

As mentioned in chapter one, traffic to these services was routed through the main website and proxied or redirected to the correct location, meaning that a HTTP request to a service's frontend was made via URL `client.fi/service-frontend` and similarly the backend call was made to `client.fi/service-backend`. This way the systems do not need to be linked to each other, thus circumventing the lack of Docker-compose support on the server.

In addition, a few of the services have databases connected to them. The complementary services had their databases Dockerized due to them being more recent, but the main Drupal service was connected to the server-wide MySQL database server installation.

A few services also had certain tooling added to the servers Crontab to be run periodically. These tasks included daily backups of business-critical data and transforming material uploaded by the customer to a consumable format for certain backend services.

2.2 Issues

2.2.1 Deployment

Prior to this project, many of the client's public services were running on a near-end-of-life Red Hat Enterprise Linux 6.9 (henceforth abbreviated to RHEL 6) server, which made developing and deploying software onto the server more difficult due to insufficient support for more modern tools, such as advanced container management.

The original development team had opted to use shell commands given to the server through SSH using Python's Fabric library, for which dedicated service-specific Fabfiles were created and distributed in the projects' respective Github repositories. Using these files, the applications could be deployed onto the server in a relatively straightforward manner. However, these Fabfiles could be several hundred lines long and only a handful of people fully understood what they did and how they worked.

With Dockerized services, the process also included building the Docker images and uploading them to Quay.io in the CI pipeline, where during deployment the Fabfiles would have the servers download and start the images. This pipeline proved rather fragile with builds occasionally failing due to errors with the Quay.io integration, if not all that complicated from a developer's point of view, since most of the required configuration was abstracted behind the Fabfiles and executable shell scripts. The Docker-based Fabfiles also contained an option to deploy the services using Docker-compose but this could never be taken advantage of during the lifetime of the original server.

2.2.2 Security

More pressingly, due to the server version nearing its end-of-life stage, security vulnerabilities had started cropping up, thus making the continued usage of the server a serious security liability. Pictured below is the lifecycle of RHEL version up to version 6.

Version	General Availability	End of Full Support	End of Maintenance Support 1	End of Maintenance Support 2 (Product retirement)	End of Extended Life-cycle Support	End of Extended Life Phase	Last Minor Release
3	October 23, 2003	July 20, 2006	June 30, 2007	October 31, 2010	January 30, 2014	January 30, 2014	
4	February 14, 2005	March 31, 2009	February 16, 2011	February 29, 2012	March 31, 2017	Ongoing	4.9
5	March 15, 2007	January 8, 2013	January 31, 2014	March 31, 2017	November 30, 2020	Ongoing	5.11
6	November 10, 2010	May 10, 2016	May 10, 2017	November 30, 2020	June 30, 2024	Ongoing	6.10

PICTURE 1. Red Hat Enterprise Linux lifecycle up to version 6 (Red Hat, Red Hat Enterprise Linux Life Cycle)

This issue was compounded by the server also having EOL versions of virtually every critical software installed, all of which were installed to server-wide namespace instead of isolated containers. Listed below (picture 2) are the installed and the latest versions of the software. It is worth noting that since some of the tools have changed their versioning scheme, the version numbers do not necessarily tell the entire truth. Hence, the most critical information are the release and EOL dates.

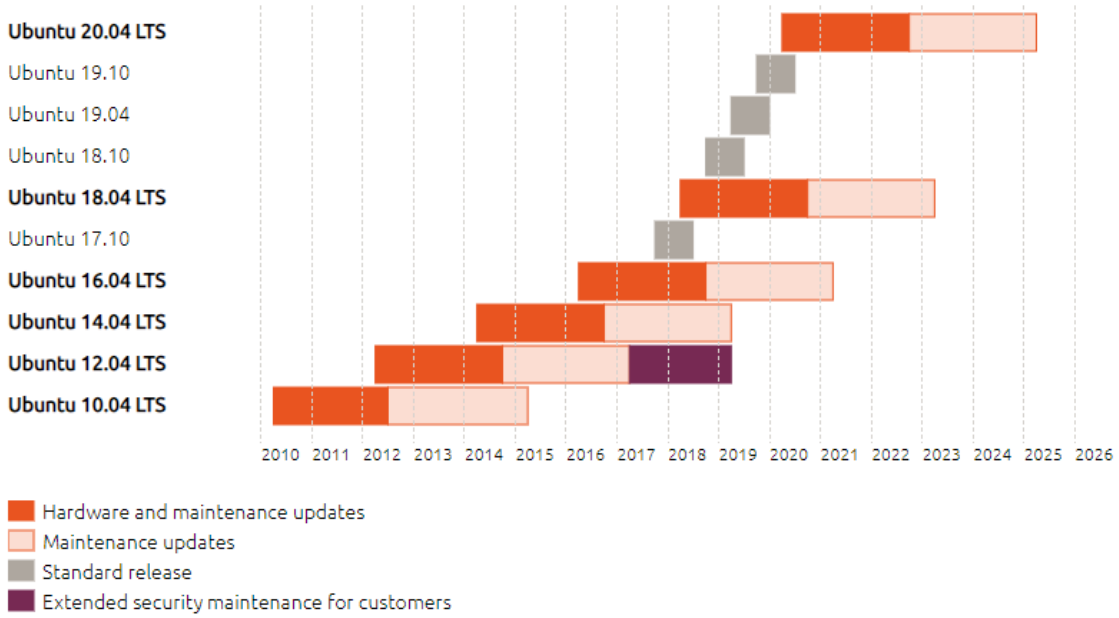
```
Apache
       Installed version: 2.2.15, released 2017-09-19, EOL 2018-01-01
       Latest version: 2.4.38, released 2019-01-22

Docker
       Installed version: 1.7.1, released 2015-06-16, EOL 2017
       Latest version: 18.09.4, released 2019-03-28

MySQL
       Installed version: 5.1.73, released 2013-12-03, EOL 2013-12-31
       Latest version: 8.0.15, released 2019-02-01

Node.js
       Installed version: 0.10.48, released 2016-10-18, EOL 2016-10-31
       Latest version: 11.11.0, released 2019-03-06

PHP
       Installed version: 5.3.3, released 2010-07-22, EOL 2014-08-18
       Latest version: 7.3, released 2018-12-06
```

PICTURE 2. Installed versions, release dates, EOL dates and the latest versions of the critical software on the server

The server was subject to routine monthly maintenance by the server provider, but due to most software vendors not making their latest versions available to operating systems no longer supported, automated system updates were not able to upgrade the installed software to more recent versions, thus the issue never getting automatically improved.

3 SOLUTION

In September of 2018 Futurice was commissioned to start a project to migrate all the services to a newly created server. In place of the old RHEL 6 server new Ubuntu 18.04 LTS based server, which is being supported by Canonical, the developers of Ubuntu, until the second quarter of 2023, was provided to the developers. The LTS version was chosen instead of the more recent 18.10 version due to a longer support lifecycle, as illustrated below (picture 3).



PICTURE 3. Ubuntu release cycle (Canonical, The Ubuntu lifecycle and release cadence)

The service management was also retooled to be more reliable and straightforward. Despite the initial willingness to utilize Docker-compose on the new server, this was dismissed in favour of Dokku, a “self-hosted PaaS” service that combines the deployment and control of Docker containers deployed onto the server with a bundled Nginx installation and abstracts most of the required configuration behind simple command line commands, all of which would have been required to be configured manually via native methods had Docker-compose been used.

The following chapters detail the most commonly used Dokku commands, explaining how they were used and how they work behind the scenes.

3.1 Dokku

Dokku exposes the most often used Docker and Nginx commands and configurations via its own command line commands, thus automating the more manual parts of building a functioning server on top of the two services. Below are listed the Dokku commands used in this project and their short explanations, which will be further elaborated upon in the coming chapters.

```
$ dokku help
Usage: dokku [--quiet|--trace|--rm-container|--rm|--force] COMMAND <app>
       [command-specific-options]

Commands:
  apps
       Manage Dokku apps
  config
       Manages global and app-specific config vars
  domains
       Manage vhost domains used by the Dokku proxy
  enter
       Connect to a specific app container
  logs
       Output app logs
  proxy
       Manage the proxy used by dokku on a per app
  ps
       List processes running in app container(s)
  run
       Run a command in a new container using the current application image
  ssh-keys
       Manage public ssh keys that are allowed to connect to Dokku
  storage
       Mount local volume / directories inside containers

Community plugin commands:
  mysql
       Plugin for managing MySQL services
  postgres
       Plugin for managing Postgres services
  redis
       Plugin for managing Redis services
```

PICTURE 4. Dokku commands used during the project

3.1.1 Deployment

Instead of complicated SSH commands and third party container repositories, all deployment to a Dokku-run server requires is the deploying machine's SSH key added to the target machine using the `dokku ssh-keys:add` command, Git, and the `dokku apps:create <app-name>` command. Adding the target server to local SSH configuration is recommended (picture 5).

```
~/.ssh/config template:
  Host <server name>
    HostName <ip address>
    User <username on the server>
    Port <port number used by SSH to connect to the server>


Commands on the target server:
  ssh-keys:add <name> [/path/to/private/key/uploaded/from/local/computer]
  dokku apps:create <app name>


On the local machine, in the project directory root:
  git remote add <remote name> dokku@<server name>:<app name>
  git push <remote name> master
```

PICTURE 5. Commands required to deploy an app to Dokku

A valid Dokku application requires either a Dockerfile, which services in the context of this project all use, or a Procfile. Procfiles are files used by Heroku – an application hosting PaaS that Dokku emulates on a single-server-wide scope – that tell the system what services to start during the service deployment and how to start them. A Procfile is defined in the format:

```
<process type>: <command>
```

After deployment the services still end up as Docker containers, so from the viewpoint of the deployed product there is no difference which deploy method is used. The Dockerfile method is the more common one, and since these files can be used also locally during development, only Dockerfile-based deploys were used during this project.

3.1.2 Domains

Adding domains to applications on the server is also handled by Dokku via the `dokku domains` command. Domains added this way are written to the application's Nginx configuration residing in `/home/dokku/<app name>/nginx.conf`, as pictured below in picture 6.

```
$ dokku domains:report client-website
  Domains app vhosts: client.fi client.com www.client.fi www.client.com

$ sudo cat /home/dokku/client-website/nginx.conf
  server {
    ...
    server_name  client.fi client.com www.client.fi www.client.com;
    ...
  }
```

PICTURE 6. Example of the relation between Dokku and Nginx configs

When making HTTP requests to the server, Nginx goes through the configured domains and tries to match the request URL to a domain (Igor Sysoev & Brian Mercer, Server Names). If a domain is found, the server directs the traffic to that service. Otherwise, the server uses the configured default service, or missing that, the first service alphabetically.

3.1.3 Storage

Adding sensitive configuration and data that could not be tracked in Git or pushed to Github, due to security concerns, to the services was done by taking advantage of Docker volumes, where a directory on the host system is mounted onto the container, where it behaves like a native directory. This can be achieved in Docker via the -v flag during container startup, but Dokku offers an automated solution to this in the `dokku storage` command, an example of which is shown in picture 7.

```
// Display the Docker volume flags configured by dokku storage
$ dokku storage:report client-website
Storage deploy mounts: -v /host/location/shared/client-website:/shared
Storage run mounts: -v /host/location/shared/client-website:/shared

// Show all files and directories in the container's /shared directory
$ docker exec -it client-website.web.1 ls -al /shared
drwxrwxr-x  7  root 1013        114  Feb 11  09:51  .
drwxr-xr-x  1  root root        75   Mar 19  17:37  ..
-rw-r--r--  1  root root        53   Feb 11  09:40  .env
-rw-rw-r--  1  root <host-user> 467  Jan  2  07:55  sites.php
...
// Rest of the directory content redacted for security reasons

// Show the same files and directories on the host
$ ls -al /location/on/host/client-website
drwxrwxr-x  7  root <host-user> 114  Feb 11  09:51  .
drwxrwxr-x  11 root <host-user> 205  Mar 11  07:00  ..
-rw-r--r--  1  root root        53   Feb 11  09:40  .env
-rw-rw-r--  1  root <host-user> 467  Jan  2  07:55  sites.php
...
// Rest of the directory content redacted for security reasons
```

PICTURE 7. An example of Dokku storage in action and an indication how Dokku app commands are just an abstraction on top Docker

For every service requiring a mounted volume a dedicated directory was created onto the host filesystem, which in turn is mounted to the corresponding container every time the container is built or started. During the container startup an entry-point script creates symbolic links for the required files or directories into locations where the services expect them to reside. An example of this is in picture 8, where all site-specific configuration and files are mounted onto the `/shared` directory in the container. In the case of the service described above, this data is linked to the `/var/www/html/site` directory during build time (picture 8), which is the default location that the site's Drupal installation uses for these files and directories.

```
RUN /bin/bash -l -c "ln -s /shared/sites.php ./sites/"

RUN /bin/bash -l -c "ln -s /shared/site1/files ./sites/site1/"
RUN /bin/bash -l -c "ln -s /shared/site1/settings.php ./sites/site1/"

RUN /bin/bash -l -c "ln -s /shared/site2/files ./sites/site2/"
RUN /bin/bash -l -c "ln -s /shared/site2/settings.php ./sites/site2/"
```

PICTURE 8. Dockerfile commands run during the container build process to create soft links to the content mounted on `/shared`

The storage option is also used to insert environment variables into the containers via `.env` files which are sourced during the container startup. Similar functionality could also be achieved with the `dokku config:set [--encoded] [--no-restart] (<app>|--global) KEY1=VALUE1 [KEY2=VALUE2 ...]` command, but was decided against to keep the experience closer to local development environments, where environment variables are also usually given by sourcing application-specific .env files, and to make editing said variables more straightforward.

In the future the `dokku config` command might have to be also used, since the .env files only work if the environment variables are needed during runtime. Applications that need to be built before being usable, like many contemporary front-end applications, need to have access to these variables during build time, which is usually done in conjunction with building the container. Such situation did not arise during the initial migration project, but development of existing and new services is ongoing.

3.1.4 Plugins

Dokku has a wide library of plugins, including controllers to various databases, application redirection, and so on. The complete list of the official plugins can be found at the Dokku Github repository (appendix 1). For this project, only MySQL, PostgreSQL and Redis plugins were used.

In line with Dokku mainly working as an abstraction layer on top of regular containers, Dokku database plugins are also just Docker containers running a database server instance. Due to these services not generally being contactable from outside the server, Dokku does not generate Nginx configurations to them and the plugin controller adds some commands to ease the handling of these services, but otherwise they do not really differ from any other Dokku-managed application. Below in picture 9 are listed the MySQL plugin commands used in this project and their short descriptions.

```
$ dokku mysql:help

  mysql:app-links <app>
       list all MySQL service links for a given app
  mysql:connect <service>
       connect to the service via the mysql connection tool
  mysql:create <service> [--create-flags...]
       create a MySQL service
  mysql:enter <service>
       enter or run a command in a running MySQL service container
  mysql:export <service>
       export a dump of the MySQL service database
  mysql:import <service>
       import a dump into the MySQL service database
  mysql:link <service> <app> [--link-flags...]
       link the MySQL service to the app
  mysql:list
       list all MySQL services
  mysql:unlink <service> <app>
       unlink the MySQL service from the app
```

PICTURE 9. List of the most used Dokku MySQL plugin commands. Each database plugin has the same commands

Docker containers can be linked together to ease communication across different containers with a shared context. Dokku database plugins expose this functionality with the `dokku mysql:link` command. Using this command, a database container can be linked to another container in a way that the target container gets an environment variable `$DATABASE_URL` that contains a direct URL to the linked database. The database container's name and network hostname are also added to the container's hosts file so that it can be referred to directly in connection URLs, etc. with only the container name. This can be seen in picture 10 which

has the relevant environment variable and the hosts file line pictured when inside the main website container.

```
# echo $DATABASE_URL
mysql://<username>:<pasword>@dokku-mysql-client-website:3306/client-website

# cat /etc/hosts
...
172.17.0.4 dokku-mysql-client-website 78c2a54d0b1c dokku.mysql.client-website
...
```

PICTURE 10. Linking data inside a container. Username and password have been redacted for security reasons

Docker allows connecting any kind of container to each other, but due it rarely being used in production servers, Dokku does not have a dedicated command for it outside of the database plugins. A more modern method of connecting containers to each other, Docker network, is available as a `dokku network` command, but that is outside the scope of this thesis.

3.1.5 Customizing Nginx

Nginx management has been pretty much automated by Dokku, and for simple servers the developer might never need to manually touch the Nginx configuration files. However, should such a situation arise, Dokku offers two methods for extending the Nginx installation (appendix 2).

The more involved option is creating a custom configuration template file for an application that Dokku user. This method was not used during this project and will not be further detailed in this thesis.

The simpler option is to create new `.conf` files for Nginx to include in the global configuration. Nginx has an `include` keyword that can be used to split the service's configuration to smaller, more manageable chunks. Using this functionality Dokku is able to have each service's specific configuration in its own file at `/home/dokku/<app name>/nginx.conf`; Nginx build the final configuration from these chunks each time the service is started.

Taking advantage of this, each service managed by Dokku also has an option for extending the configuration with user-defined files by including every `.conf` file in the `nginx.conf.d/` subdirectory of each service's `/home/dokku/<app name>/` directory.

These configuration files can inlude anything allowed inside Nginx config's server block, including rerouting or proxying traffic to different services or addresses, access management, configuring the maximum allowed file upload size of the server, etc.

3.2 Service-specific tools and cronjobs

As on the old server, the new server also includes several cronjobs. Simple tasks like backup creation and deleting old backups did not drastically change, so those could be copied from the old server with little to no alterations required.

However, many of the "worker" tasks required by the embedded services had to be reworked due to the new architecture. On the old server these tasks were usually just directories within the deployed applications containing a Dockerfile that were built manually and run periodically, thus making changes to these workers getting deployed dependent on deployment of the entire application and manual work from the deployer. This was deemed unsustainable on the new server, since the renewed toolkit allowed for a much leaner development experience.

The application-specific workers were separated into their respective Git repositories that could be developed and deployed separately from the main services. Deploying these services was rewritten to utilize Dokku, but since these workers were only used periodically, a configuration change to disable starting up the applications after building them was made via the `dokku config:set <app name> DOKKU_SKIP_DEPLOY=true` env variable resulting in the images being built on the server, but not allowing them to be run only when needed using the `dokku run` command.

Managing the workers with Dokku also made taking advantage of the usual Dokku commands, such as `dokku storage`, possible. This enabled the worker commands to be condensed into just simple one-line commands that could be added to the server's Crontab. All the Docker-specific logic and filesystem routing could then be handled inside the container's Dockerfile or entrypoint command. Dokku also allows the started containers to be deleted once their task is finished with the `--rm` command, instead of keeping the now useless container running in the background. An example of all these features in practice can be seen below, in picture 11.

```
docker run -i --rm -v /host/directory:/data quay.io/client/service-tools
/data/materials/source_file.txt > /host/directory/service-name/output-
file.json 2> /host/directory/service-name/stderr.log

dokku --rm run service-name-tools npm run parse:area
```

PICTURE 11. The same cronjob as defined on the old and on the new server, respectively

3.3 Static files

Not all services from the old server were Dockerized, usually due to the simplicity of the service and the actual development of the services already having come to an end. These kinds of services were copied directly from the old server and given domain names in Nginx. Due to these services also getting routed through main website, all that was needed was an addition of a custom Nginx configuration file to the main website's `nginx.conf.d/` directory, that instructed Nginx to serve the static files relating to the requested URL.

In the case of both Dockerized and static services existing on the same server, the traffic to Docker containers could proxied to a local port defined as the point of entry between the host and the container, while traffic to the static files was routed directly to the containing directory and the HTML content was served as a response, as seen in picture 12.

```
location /dockerized-service/ {
  proxy_pass http://localhost:<port defined for the app>/;
}

location /static-service/ {
  alias /host/directory/service-name/;
  autoindex on;
}
```

PICTURE 12. Nginx configurations for rerouting traffic to different services

4 CONCLUSIONS

After being in development for close to four months, all domains associated with the relevant services were moved to point to the new server on the 22nd of January 2019. The old server was left intact but not reachable via any URL for a period of time to allow for undoing the migration in case of critical errors in the services' availability, and for reference material both for this thesis, and should the need arise to further configure the new server. The migration process has made developing applications running on the server more straightforward, allowed the entire development team to be able to configure the server if needed, and fixed over 100 vulnerabilities.

Due to ongoing development of most of the services deployed onto the server, the requirements and agreed upon best practices are constantly evolving; few examples being the possible requirement to utilize the `dokku config` command in the future and Dockerizing every service deployed onto the server. Thence, the project cannot be described as completely ready. However, the portion covered in this thesis has been deemed successfully completed by both Futurice and the client, and development has entered its lifecycle management stage.

The advent of container-based application development has brought DevOps and server-related development, both of which have a reputation of being somewhat tedious and difficult aspects of software development, closed to the mainstream. The average developer no longer needs to remember various server configuration directives, Unix commands or commands for several different applications; almost everything can nowadays be achieved with just one tool, like Dokku in this case.

There is an argument to be made for hosting everything on a cloud platform like AWS or PaaS like Heroku, which abstract the server configuration even further, sometimes even freeing the developer entirely from thinking about anything else but the application itself. However, with abstraction a bit of control over the underlying system is also lost, and this is not always ideal. Even if deploying all of the services to Heroku might have been ideal from the developers' viewpoint,

uploading business-critical services to someone else's servers and data centres with no actual control over the server hardware – or even software – is usually not the most tempting idea from the businesses' perspective.

Due to this reason, experiencing how far building hand-made servers has come in the last few years is absolutely pleasing. In a time when everything has more or less to do with computers and software, every aspect of this industry should be as approachable as possible to allow new talent to flourish. This has been the driving idea in software development for quite a while, but finally the same idea has caught on in DevOps circles, after years of confrontational back-and-forth about Unix neckbeards and Ikea developers. One hopes that the current trend keeps gaining momentum so that some day migrating a legacy server will not be a project worthy of an entire thesis.

REFERENCES

Amazon. N.d. What is DevOps? Read 30.03.2019. https://aws.amazon.com/devops/what-is-devops/

Canonical. N.d. The Ubuntu lifecycle and release cadence. Read 31.03.2019. https://www.ubuntu.com/about/release-cycle

Dokku documentation. N.d. Getting Started with Dokku. Read 30.03.2019. http://dokku.viewdocs.io/dokku/getting-started/installation/#what-is-dokku

Forcier, J. 2018. Fabric documentation. Published 26.07.2018. Read 30.03.2019. http://www.fabfile.org/

Fowler, M. 2016. Continuous Integration. Published 01.05.2006. Read 15.04.2019. https://martinfowler.com/articles/continuousIntegration.html

Kasireddy, P. 2016. A Beginner-Friendly Introduction to Containers, VMs and Docker. Published 04.03.2016. Read 30.03.2019. https://medium.freecodecamp.org/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b

Red Hat. N.d. Red Hat Enterprise Linux Life Cycle. Read 30.03.2019. https://access.redhat.com/support/policy/updates/errata

Sysoev, I & Mercer, B. N.d. Server names. Read 31.03.2019. https://nginx.org/en/docs/http/server_names.html

APPENDICES

Appendix 1. List of Dokku plugins

https://github.com/dokku/dokku/blob/master/docs/community/plugins.md

Official Plugins (Beta)

The following plugins are available and provided by Dokku maintainers. Where noted, these plugins should be considered beta software and may not have been used as thoroughly as community plugins. Please file issues against their respective issue trackers.

Plugin	Author	Compatibility
CouchDB (beta)	dokku	0.4.0+
Elasticsearch (beta)	dokku	0.4.0+
Grafana/Graphite/Statsd (beta)	dokku	0.4.0+
MariaDB (beta)	dokku	0.4.0+
Memcached (beta)	dokku	0.4.0+
Mongo (beta)	dokku	0.4.0+
MySQL (beta)	dokku	0.4.0+
Nats (beta)	dokku	0.4.0+
Postgres (beta)	dokku	0.4.0+
RabbitMQ (beta)	dokku	0.4.0+
Redis (beta)	dokku	0.4.0+
RethinkDB (beta)	dokku	0.4.0+
Copy Files to Image	dokku	0.4.0+
HTTP Auth (beta)	dokku	0.4.0+
Let's Encrypt (beta)	dokku	0.4.0+
Maintenance mode (beta)	dokku	0.4.0+
Redirect (beta)	dokku	0.4.0+

Community plugins

Warning: The following plugins have been supplied by our community and may not have been tested by Dokku maintainers.

Datastores

Relational

Plugin	Author	Compatibility
MariaDB	Kloadut	0.3.x
MariaDB (single container)	ohardy	0.3.x
MariaDB (single container)	krisrang	0.3.26+
PostgreSQL	jlachowski	0.3.x
PostgreSQL (single container)	ohardy	0.3.x
PostgreSQL (single container)	Flink	0.3.26+

Caching

Plugin	Author	Compatibility
Nginx Cache	Aluxian	0.5.0+
Redis (single container)	ohardy	0.3.x
Varnish	Zene-dith	Varnish cache between nginx and applica-tion with base configuration

Queuing

Plugin	Author	Compatibility
RabbitMQ	jlachowski	0.3.x
RabbitMQ (single container)	jlachowski	0.3.x
ElasticMQ (SQS compatible)	cu12	0.5.0+
VerneMQ (MQTT Broker)	mrname	0.4.0+

Other

Plugin	Author	Compatibility
etcd	basgys	0.4.x
FakeSNS	cu12	0.5.0+
InfluxDB	basgys	0.4.x
RethinkDB	stuartpb	0.3.x
Headless Chrome	lazyatom	0.8.1+

Plugins Implementing New Dokku Functionality

Plugin	Author	Compatibility
App name as env	cjblomqvist	0.3.x
Docker Direct	josegonzalez	0.4.0+
Dokku Clone	crisward	0.4.0+
Dokku Copy App Config Files	josegonzalez	0.4.0+
Dockerfile custom path	mimischi	0.8.0+
Dokku Registry[1]	agco-adm	0.4.0+
Dokku Require[2]	crisward	0.4.0+
Global Certificates	josegonzalez	0.5.0+
Graduate (Environment Management)	Benjamin-Dobell	0.4.0+
Haproxy tcp load balancer	256dpi	0.4.0+
Hostname	michaelshobbs	0.4.0+
HTTP Auth Secure Apps	matto1990	0.4.0+
Monit (Health Checks)	mbreit	0.8.0+
Nuke Containers	josegonzalez	0.4.0+
Open App Ports	josegonzalez	0.3.x
Proctype Filter	michaelshobbs	0.4.0+
robots.txt	candlewaster	0.4.x

Plugin	Author	Compatibility
SSH Deployment Keys[3]	cedricziel	0.4.0+
SSH Hostkeys[4]	cedricziel	0.3.x
Application build hook	fteychene	0.4.0+
Post Deploy Script	baikunz	0.4.0+

[1] On Heroku similar functionality is offered by the heroku-labs pipeline feature, which allows you to promote builds across multiple environments (staging -> production)

[2] Extends app.json support to include creating volumes and creating / linking databases on push

[3] Adds the possibility to add SSH deployment keys to receive private hosted packages

[4] Adds the ability to add custom hosts to the containers known_hosts file to be able to ssh them easily (useful with deployment keys)

Other Plugins

Plugin	Author	Compatibility
Airbrake deploy	Flink	0.4.0+
APT	F4-Group	0.4.0+
Bower install	alexanderbeletsky	0.3.x
Bower/Grunt	thrashr888	0.3.x
Bower/Gulp	gdi2290	0.3.x
Bower/Gulp	jagandecapri	0.3.x
Builders: bower, compass, gulp, grunt	ignlg	0.4.0+
Chef cookbook	nickcharlton	
Docker auto persist volumes	Flink	0.4.0+
Hostname	michaelshobbs	0.4.0+
Limit (Resource management)	sarendsen	0.9.0+

Plugin	Author	Compatibi-lity
Logspout	michaelshobbs	0.4.0+
Syslog	michaelshobbs	0.10.4+
Long Timeout	investtools	0.4.0+
Monit	cjblomqvist	0.3.x
Monorepo	iamale	0.4.0+
Node	ademuk	0.3.x
Node	pnegahdar	0.3.x
Rollbar	iloveitaly	0.5.0+
Slack Notifications	ribot	0.4.0+
Telegram Notifications	m0rth1um	0.4.0+
Tor	michaelshobbs	0.4.0+
User ACL	Maciej Łebkowski	0.4.0+
Webhooks	nickstenning	0.3.x
Wkhtmltopdf	mbriskar	0.4.0+
Dokku Wordpress	dokku-community	0.4.0+
Access	mainto	0.4.0+

Appendix 2. Dokku - Customizing the Nginx configuration

http://dokku.viewdocs.io/dokku/configuration/nginx/

Customizing the Nginx cofiguration

Dokku uses a templating library by the name of sigil to generate nginx configuration for each app. You may also provide a custom template for your application as follows:
Copy the following example template to a file named `nginx.conf.sigil` and either:
If using a buildpack application, you **must** check it into the root of your app repo.
`ADD` it to your dockerfile `WORKDIR`
if your dockerfile has no `WORKDIR`, `ADD` it to the `/app`folder
When using a custom `nginx.conf.sigil` file, depending upon your application configuration, you *may* be exposing the file externally. As this file is extracted before the container is run, you can, safely delete it in a custom `entrypoint.sh`configured in a Dockerfile `ENTRYPOINT`.
The default template is available here, and can be used as a guide for your own, custom `nginx.conf.sigil` file. Please refer to the appropriate template file version for your Dokku version.

Available template variables

```
{{ .APP }}                      Application name
{{ .APP_SSL_PATH }}             Path to SSL certificate and key
{{ .DOKKU_ROOT }}               Global Dokku root directory (ex: app dir would
                                be `{{ .DOKKU_ROOT }}/{{ .APP }}`)
{{ .DOKKU_APP_LISTENERS }}      List of IP:PORT pairs of app containers
{{ .PROXY_PORT }}               Non-SSL nginx listener port (same as
                                `DOKKU_PROXY_PORT` config var)
{{ .PROXY_SSL_PORT }}           SSL nginx listener port (same as
                                `DOKKU_PROXY_SSL_PORT` config var)
{{ .NOSSL_SERVER_NAME }}        List of non-SSL VHOSTS
{{ .PROXY_PORT_MAP }}           List of port mappings (same as
                                `DOKKU_PROXY_PORT_MAP` config var)
{{ .PROXY_UPSTREAM_PORTS }}     List of configured upstream ports (derived
                                from `DOKKU_PROXY_PORT_MAP` config var)
{{ .RAW_TCP_PORTS }}            List of exposed tcp ports as defined by Dock-
                                erfile `EXPOSE` directive (**Dockerfile apps
                                only**)
{{ .SSL_INUSE }}                Boolean set when an app is SSL-enabled
{{ .SSL_SERVER_NAME }}          List of SSL VHOSTS
```

Note: Application config variables are available for use in custom templates. To do so, use the form of `{{ var "FOO" }}` to access a variable named `FOO`.

Customizing via configuration files included by the default templates

The default nginx.conf template will include everything from your apps `nginx.conf.d/` subdirectory in the main `server {}` block (see above):

```
include {{ .DOKKU_ROOT }}/{{ .APP }}/nginx.conf.d/*.conf;
```

That means you can put additional configuration in separate files, for example to limit the uploaded body size to 50 megabytes, do

```
mkdir /home/dokku/node-js-app/nginx.conf.d/
echo 'client_max_body_size 50m;' > /home/dokku/node-js-app/nginx.conf.d/up-
      load.conf
chown dokku:dokku /home/dokku/node-js-app/nginx.conf.d/upload.conf
service nginx reload
```

The example above uses additional configuration files directly on the Dokku host. Unlike the `nginx.conf.sigil` file, these additional files will not be copied over from your application repo, and thus need to be placed in the `/home/dokku/node-js-app/nginx.conf.d/` directory manually.

For PHP Buildpack users, you will also need to provide a `Procfile` and an accompanying `nginx.conf` file to customize the nginx config *within* the container. The following are example contents for your `Procfile`

```
web: vendor/bin/heroku-php-nginx -C nginx.conf -i php.ini php/
```

Your `nginx.conf` file - not to be confused with Dokku's `nginx.conf.sigil` - would also need to be configured as shown in this example:

```
client_max_body_size 50m;
location / {
    index index.php;
    try_files $uri $uri/ /index.php$is_args$args;
}
```

Please adjust the `Procfile` and `nginx.conf` file as appropriate.