jamk.fi

Implementing a Fault-tolerant and Highly Scalable Content Delivery Service

Aki Muhonen

Bachelor's thesis
September 2020
Information and communications technology
Degree Programme in Network Engineering

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s)	Type of publication	Date
Muhonen Aki	Bachelor's thesis	September 2020
		Language of publication: English
	Number of pages: 43	Permission for web publication: x

Title of publication

Implementing a Fault-tolerant and Highly Scalable Content Delivery Service

Degree programme

Information and Communications Technology

Supervisor(s)

Rantonen, Mika

Kotikoski, Sampo

Assigned by

Kansaneläkelaitos

Abstract

Kansaneläkelaitos (KELA) ensures the social security of Finnish people. KELA has a vast IT-department which produces services for the citizens to use on the Internet. KELA was in need of a new service which shares content for applications.

The objective was to create up to date fault-tolerant and highly scalable content delivery system, which shares static content for applications, so it wouldn't need to be done manually, and was ready to be implemented into production. The service was designed on the basis of an older service which was not ready for production or automatized.

The service included one backend server working as content database, two frontend servers for delivery of the content, working as HTTP servers and a load balancer. The service was fully automated using SaltStack, because SaltStack was widely used in KELA, and made easy to use for the administrators and content users.

The service worked as it was supposed to. It delivered the content from backend to frontends and from frontend to the application. Also, the service was fault-tolerant. It was tested to build the service again if the servers were taken down and emptied.

The service is a simple in the way it works. The hard part was automatizing it using SaltStack, and the service was a success. There was not much to do, or think for the further development. Some small things that are not mandatory for the service. It has been thought from start to finish.

Keywords/tags (subjects)

SaltStack, Apache, SSH, HTTP

Miscellaneous (Confidential information)

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Muhonen Aki	Opinnäytetyö, AMK	Syyskuu 2020
	Sivumäärä 43	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty: x

Työn nimi

Implementing a Fault-tolerant and Highly Scalable Content Delivery Service

Tutkinto-ohjelma

Tieto- ja viestintätekniikka

Työn ohjaaja(t)

Rantonen, Mika

Kotikoski, Sampo

Toimeksiantaja(t)

Kansaneläkelaitos

Tiivistelmä

Kansaneläkelaitos (KELA) varmistaa suomalaisten sosiaaliturvan. KELA:lla on laaja IT-osasto, joka tuottaa palveluja kansalaisille internetissä käytettäväksi. KELA tarvitsi uutta palvelua, joka jakaa sisältöä sovelluksille.

Tavoitteena oli luoda ajan tasalla oleva vikasietoinen ja skaalautuva sisällönjakelupalvelu, joka jakaa staattista sisältöä sovelluksille, jotta sitä ei tarvitse tehdä manuaalisesti, ja oli valmis käyttöönottoon tuotannossa. Palvelu suunniteltiin vanhemman palvelun pohjalta, joka ei ollut valmis tuotantoon tai automatisoitu.

Palvelu sisälsi yhden taustapalvelimen, joka toimi sisällön tietokantana, kaksi edustapalvelinta sisällön jakeluun, jotka toimivat HTTP palvelimina ja kuormanjakajan. Palvelu automatisoitiin täysin SaltStackilla, koska SaltStack oli laajassa käytössä KELA:ssa, ja se oli helppokäyttöinen järjestelmänvalvojille ja sisällön käyttäjille.

Palvelu toimi niin kuin piti. Se toimitti sisällön taustapalvelimesta edustapalvelimiin, ja edustapalvelimilta sovellukselle. Palvelu oli myös vikasietoinen. Palvelun rakentaminen testattiin uudelleen, kun palvelimet olivat alhaalla ja tyhjennetty.

Toiminta tavaltaan palvelu oli yksinkertainen. Vaikein osuus oli automatisoida se käyttäen SaltStack:ia, ja palvelu oli onnistunut. Jatkokehityksen kannalta ei ollut paljon tekemistä tai ajateltavaa. Joitakin pieniä asioita, jotka eivät ole pakollisia palvelulle. Se on ajateltu alusta loppuun.

Avainsanat (asiasanat)

SaltStack, Apache, SSH, HTTP

Muut tiedot (salassa pidettävät liitteet)

Contents

Figures

Tables

Acronyms

ACL	Access Control Lists
CSS	Cascading Style Sheets
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
KELA	Kansaneläkelaitos
PID	Process Identification
RSA	Rivest-Shamir-Adleman
SFTP	SSH File Transfer Protocol
SLS	Salt State File
SSH	Secure Shell
URL	Uniform Resource Locator
YAML	YAML Ain’t Markup Language
YUM	Yellowdog Updater Modified

1 Introduction

1.1 Kansaneläkelaitos

Kansaneläkelaitos, also known as KELA is organization which works under parliament of Finland. KELAs objective is to take care of social security of Finnish people living in Finland and in foreign countries. Social security that KELA takes care of includes things like financial support, health insurance, pension, student allowance, housing benefit and so on. (Elämässä mukana – muutoksissa tukena, 2018.) Clients can use KELAs calculators by themselves to find out if they can get includes from KELA and then apply for includes via internet, phone, service point or mail. (Näin KELA palvelee, 2019.) KELA has many different departments and in the end of 2018 KELA had 7 732 employees. (Kansaneläkelaitos Toimintakertomus ja tilinpäätös, 2018.)

1.2 Project assignment

Due to today's rapid development of technology, automation is needed more than ever in server administration. In this thesis, I will create and execute a solution for certain need in automation. Project assignment I got is to create a fault-tolerant and highly scalable content delivery system, which shares static content for applications. In other words, applications request static content from the service, and it does not need to be added manually. This saves many hours of work when it is automated with service instead of always manually including the content. Especially now, since new applications are made continuously, and therefore the service needs to be highly scalable. Service is meant to take a place in production, and services in production need to be always working or it is not productive for business. Therefore, the service needs to be also fault-tolerant and always running.

1.3 Objectives

The objective is to create a service which is highly scalable, fault-tolerant and secure. Service will be running in production and it needs to have testing environment as well, so basically, need is to create two services, one for production and one for testing.

Testing environment is made because if something changes, or needs to be updated, it can first be tested in this environment. If there was only production environment and something new does not work, it could make some websites not work properly.

As mentioned before, today there is huge need for automation, so management of the service must be automated, and this will create with software called SaltStack. Service will need firewalls and load balancing too so everything can run smoothly, and it will remain secure, and of course it needs to be up and running all times since it is going to production, so alerts and status pages are mandatory in case something breaks. To make this service happen, the background info comes from colleagues, web pages and from KELAs own documentation on how these technologies are used in KELA. For example, there is a good documentation in KELAs intranet how SaltStack is used and how it can be used in KELA, which is used on this thesis to automate this service.

1.4 Research methods

There is a service like this in KELA, which is basically version 0.1 of this service, it is hard coded and not documented or automatized at all. This new service has been on KELAs to do list a long time, and this was an excellent opportunity to make this service come true and write a bachelor's thesis of it at the same time. This old service is running on Nginx and it uses same software as this one. In this service there is some things that are same as in this already existing service, so some configurations can be retrieved from this old service, but these configurations need to be programmed with Jinja so it can be automatized using SaltStack. Therefore, this research is qualitative, because the goal is to research how to create a better service than the last one using these tools.

SaltStack is chosen to be automation tool for this service, because it is in wide use at KELA already and it is well documented, which will make creating this service easier. For Jinja programming, Apache configuration and automatizing in general, help of internet documentation and existing documentation in KELA, which is one of qualitative research methods (Qualitative Research: Definition, Types, Methods and Examples. 2020.), also knowledge of colleagues will be used.

2 Theory

2.1 Access control lists

With ACL permissions can be given for users and groups, permissions are read, write and execute. With read permissions groups and users can open and read files, write permissions files can be modified, and with execute permissions if the file is a program, it can be executed. (POSIX Access Control Lists on Linux, n.d.) ACL will be used in this thesis because, it is the most modern tool for changing user rights or giving them. ACL allows changing owners and default groups instead of using two or more different Unix-programs, for example *chmod* for user rights and *chown* for owners. Also, SaltStack includes own salt state for ACL, which will be used in this thesis. This means that instead of using *cmd run* in salt states, ACL state modules can be used instead. (Salt.States.Linux_ACL, 2020.)

2.2 Apache HTTP Server

Apache HTTP Server is a software which allows to turn computer into web server, this means that content can be shared using HTTP protocol. In other words, files that are in the webserver can be fetched or inspected with browsers or remote machines via URLs. (Getting started, 2020.) Apache is free and opensource software, which makes it a great option to choose for webserver. This means that anyone can take part in developing the apache and modify it to match one's own needs. (About Apache, 2020.) In this service, the content will be shared using Apache servers.

Apache is very flexible web server, it can be modified to match anyone's needs, this can be done with modules. Modules are like extra settings for configuration, for example if needed, access to the content can only be allowed from certain hosts, this can be done with module called *mod_access_compat*. (Apache Module mod_access_compat. 2020.)

2.3 HTTP/HTTPS

Hypertext Transfer Protocol, also known as HTTP, is used to send information and content between devices over networks, like documents, images etc. HTTP have been used since 1990, which was version 0.9, the version 1.1 which I am using in this service came out 1999. (RFC 2616, 1999.)

When a client wants to get content using HTTP, the client sends HTTP request message to the HTTP server. After this, the server responds to the client with HTTP response, as seen on Figure 1, which includes the data that the client asked with the request message. This is how the HTTP protocol works in the general sense. HTTP is a stateless protocol, which means that the connection between the client and server is not maintained throughout the process, connection is being enabled briefly when HTTP messages are sent. (RFC 2616, 1999.)

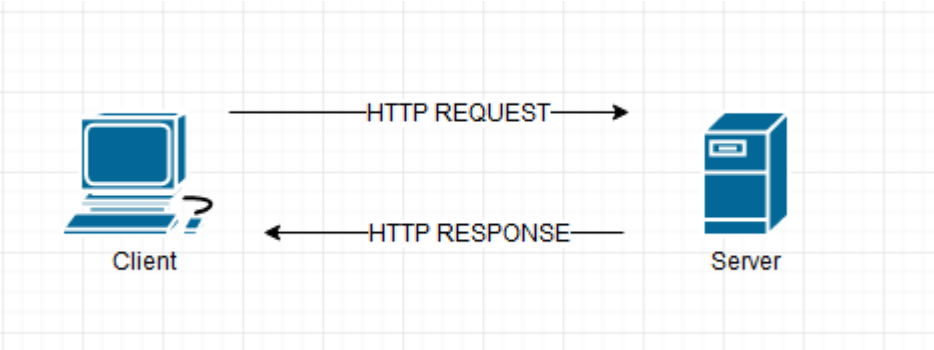

Figure 1 HTTP messages

Difference between HTTP and HTTPS is HTTPS being secured, letter *s* in HTTPS means secure. This denotes that the data which is being sent in HTTPS messages is unreadable while it is traveling through the network. In HTTP messages, the data is plain text when it is traveling through network and anyone who captures those packets can read the data as seen on Figure 2. (RFC 2660, 1999.) HTTP will be used in this service and HTTPS outside the service.

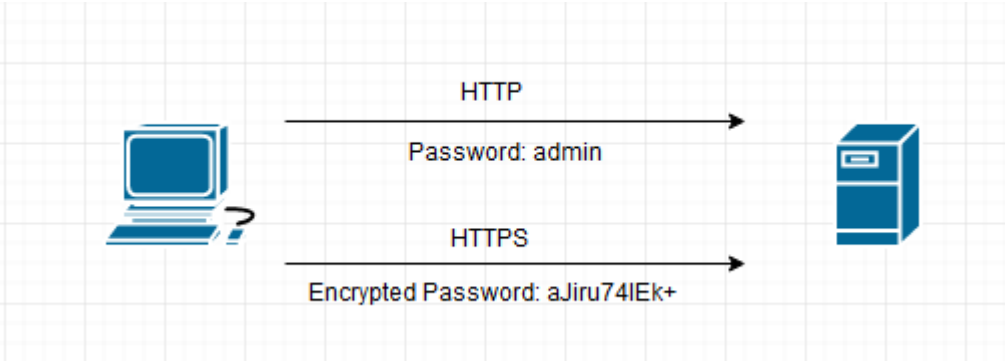

Figure 2 HTTP vs HTTPS

When client sends request to the server, server responds with a certain status code. These status codes can be categorized as seen in table 1 and every code have different meaning. (RFC 2616, 1999.)

Table 1 HTTP status codes

Code	Description
1xx	Informational
2xx	Successful
3xx	Redirection
4xx	Client Error
5xx	Server Error

2.4 Jinja

Jinja templates are used to make configuration files. Actual configuration files are made from these templates, and the templates can include programming, Jinja uses Python for this. (jinja, 2017) For example, Apache configuration has differences based on which operating system is used. With Jinja templates and some if/else statements the configuration file is built from same the template exactly for the right operating system. For this service, all the automatization work will be done with Jinja, which makes this job a lot easier. This is because, SaltStack itself has chosen to use Jinja as default templating language in SLS files. Jinja has very simple syntax and it is very similar to YAML syntax and its trademark is the *{% %}* brackets. (Understanding Jinja. 2020.)

2.5 Logrotate

Logrotate is used to manage logfiles automatically based on the configuration. With Logrotate, old and new logfiles can be moved, removed, mailed or compressed, without needing doing it manually. Some programs use Logrotate by default, one of these programs is Lsyncd. (logrotate(8) – Linux man page, n.d.) In this service, Logrotate is a perfect tool for rotating the log files, because the service will be used for many years without pauses and it writes logs nonstop. Logrotate is in the Linux servers by default, so it isn't a program that needs to be installed manually which is good thing, because it saves work in automatization. With Logrotate, useless log files get removed and it saves a lot of disk space in servers. Also, Logrotate will be configured in a way where it compresses the log files to save even more space than they would without compressing.

2.6 Rsync and lsyncd

Rsync is an application which is used to copy files, Rsync looks up for differences between files in destination and source. This way Rsync saves network capacity needed by sending only the changes in files through network and not the whole file. Rsync is mainly used for file mirroring and backups. (Rsync(1) – Linux man page, n.d.)

Lsyncd is live synchronizing program, which will automatically update files in destination server, when they are modified or added in source server. It uses Rsync and SSH to do this and this way manually updating or sending files between source and destination is not needed. (Lsyncd(1) – Linux man page, n.d.)

2.7 SaltStack

SaltStack, also known as salt, is open source software used for automation and configuration management. Salt uses pull and push executions, in a nutshell, it pulls configuration from repositories and pushes them into servers. SaltStack uses master-slave type setup, in other words, master gives the slave, Saltstack calls these slaves minions, the commands to execute and the minion executes them. This way thousands of servers can be configured simultaneously. (SaltStack, 2018.)

In SaltStack, there is a module called Salt Mine, this module will be used in this service to store and get SSH key. This module is used to collect data from Salt Minion and store it on the Master, this way the data can be used by any Minion. (The salt mine, 2020.)

All Salt State files, also known as SLS files, needs to be built in YAML syntax. This is because YAML is very simple syntax and the creators of SaltStack wants these files to be “Stupidly Simple”. (How do I use Salt States, 2020.) This YAML syntax is presented in figure 3.

```
1.

Key: Value

2.

Key:
  Value

3.

First_Key:
  Second_Key: Value_of_Second_key

4.

Dictionary:
  - list_value_one
  - list_value_two
  - ...
```

Figure 3 YAML syntax

This is all there is for the YAML syntax, it can be used in many ways. In figure 3, numbers 1. and 2. is basically the same thing, but it presented in different way. Number 3. shows that you can put keys in keys, in other words the keys can be nested. Number 4. presents that you can also create a list, meaning that you can create a dictionary that has multiple values. (Understanding YAML, 2020.) Basically, it is all about keys and values, key can be a setting in configuration and values is a value for that setting. For example, OpenSSH client needs to be installed to a server, it can be installed with the SLS file like seen in the figure 4.

```
openssh-client:
  pkg.installed
```

Figure 4 Install openssh with sls

In figure 4, *openssh-client* is the key and *pkg.installed* is the value of that key, and this way OpenSSH client gets installed on the server. Dictionary can be used for example giving user rights to a directory, where dictionary is the name of that folder and the values are usernames that can have access to that folder. (How do I use Salt States, 2020.) All the configuration files that can be seen in this service, is built with mixing YAML and Jinja, these can be seen in the implementation part of the thesis and these are only used for Salt States. Salt Pillar files are only YAML and not Jinja, those files include only pure data, these files don't basically execute anything.

2.8 Secure Shell

Secure shell, SSH for short, is a program which is used to log into remote machines safely. SSH was developed in 1995 by university student, who was tired of username and password sniffers. SSH is still today the most common and safest protocol used by most of the world's webservers and Linux computers to connect remote machines safely. There is also protocol called SSH File Transfer Protocol, SFTP for short. This protocol was created to transfer data with usage of SSH to make it safest way. (SSH (Secure Shell), 2020.) With SSH, it is possible to execute commands in remote machines over network. (ssh(1) – Linux man page, 2013.) In this thesis, SSH is used between backend and frontend for secure distribution of the content. This is because SSH is easy to use, secure and it can be configured in Rsyncd configuration.

3 Planning

3.1 Introduction

As seen in figure 5, content-delivery system will have four servers in it, one server is the database, this server will be called backend in this thesis, which have the content. Content is basically Cascading Style Sheets; these CSS files define what web browser pages looks like. In other words, they are configuration files for HTML pages.

Next two servers are web servers; these servers deliver the content for the applications that uses these CSS files and these servers will be called frontend in this thesis. Last server is load balancer, which handles the requests for the content.

There will be two implementations of the same service, one for testing environment and one for production environment. Implementations will be made the same way, therefore only one introduction for this service is enough.

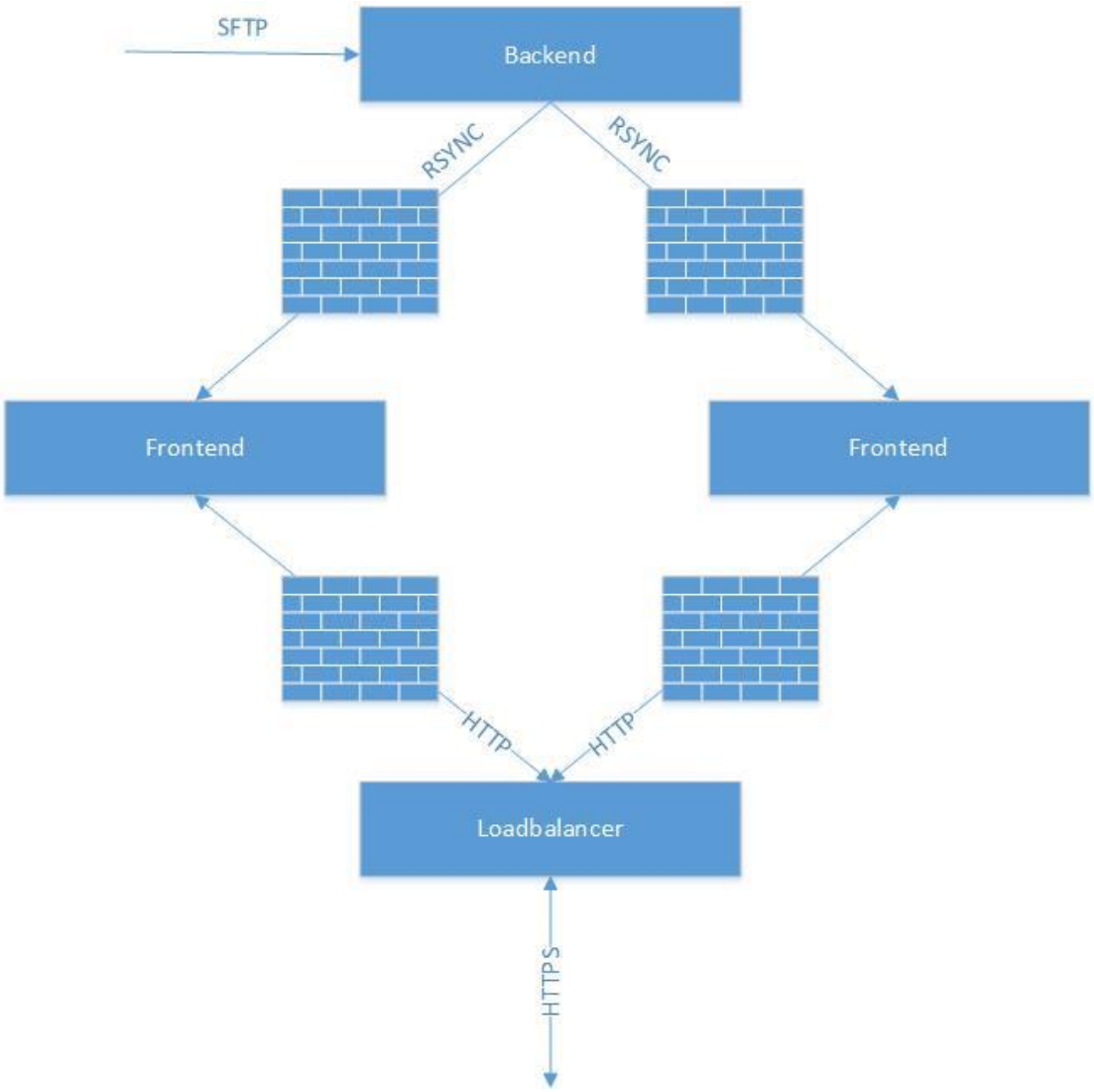

Figure 5 Static content delivery service

3.2 Backend

Backends' role is going to be the database for the content, therefore it needs to store the content, and push the content automatically to the frontends, where it can be fetched. It needs to be made this way, instead of content being fetched straight from the backend because the service is needed to be reliable. If there was only backend, it would be too heavy for it to do everything. Therefore, it would not be reliable, and it could not be called service, it would be a content delivery server. For example, it would not be secure if the server went down, then there would be no way to get the content. The frontend role will be discussed later in more detail in section 3.3.

Backend will deliver content using Rsync and Lsyncd, and Lsyncd needs to know what to do and when, that can be managed by doing a configuration file for it. Lsyncd needs to use SSH -connection for the sake of security. using key authentication with the SSH is mandatory, because there is no one typing the password for the connection, so the SSH keys can be created in backend, and send the public part of the key to the frontend. In addition, there is a need to create all the directories and put correct user rights to them. Log files need to be rotated, so the server doesn't fill up from them. It is enough to see log files from past seven days, so Logrotated -program is a good choice for this, and this needs configuration file too.

3.3 Frontend

Frontends role is to deliver the static content to load balancer, that backend stores and delivers to frontend every 2 seconds. There are two frontend servers, just in case if one of those two goes down for some reason, this way the other frontend can still deliver content.

Content delivery is going to need HTTP server application and for this, choice will be apache. Some might wonder why Apache over Nginx, since this is only static content. Answer to this is that apache is easier to configure in authors opinion, and with Apaches addons, any extra work is not necessary. For example, status page for Apache can easily be added by writing a few lines in Apache's configuration file. Hardening the Apache is also required, so it will be as secure as it can be, this means that only modules that are needed will be added to the service, nothing extra. Also,

only the content that needs to be delivered will go through, everything else is blocked. Required modules can be seen in table 2.

Table 2 Apache modules

Module	Description
Mod_access_compat	Allow or deny access to files and directories.
Mod_authz_core	Allow or deny authenticated users access portions of web site.
Mod_dir	Access certain file or directory via URI.
Mod_headers	Allows HTTP header modifying.
Mod_log_config	Makes custom apache logging possible.
Mod_mime	Allows to set type of the content.
Mod_mpm_prefork	Allows each server process answer incoming requests.
Mod_status	Makes usage of server status page possible.
Mod_systemd	Makes system integration possible for apache service.
Mod_unixd	Makes using user and group rights possible.

So, configuration file for Apache is needed, and just like in backend, Logrotated -program is needed here too, also making a user which will be using the Lsyncd -program between the backend and frontends, because someone, in this case the user needs to deliver the content.

3.4 Load balancer

Implementation of the load balancer is not made by author; it is made by different branch of KELA. However, there are few requirements for the load balancer which

are needed for the service to perform as wanted. These requirements are explained in this section.

Load balancer will be balancing the load equally between both frontend servers and it will be watching the Apache status page in both frontends with algorithm. Algorithm is shown in figure 6, first if statement watches the HTTP responds it gets from the status page. If the HTTP GET request gets response with code 200, this means everything is working as intended and the node can receive packets. If the status code is something else than 200, this means there is something wrong in frontend server and system administrator needs to fix it.

Second if statement in figure 6 is made for planned maintenance, for example if something needs to be updated or fixed in the service. Then empty file called “on” in the folder maintenance must be created, this way the HTTP GET request gets response from maintenance/on with code 200 and this means that the node is set to receive no traffic.

```
FOR NODE IN NODES
  IF http://NODE/system/lbstatus != HTTP 200
    NODE RECEIVES NO TRAFFIC

  IF http://NODE/maintenance/on == HTTP 200
    NODE RECEIVES NO TRAFFIC
```

Figure 6 Algorithm in pseudo code

3.5 SaltStack automation

Service administration will be automated with SaltStack, this means that the servers doesn’t need to be touched when something needs to be updated, this will be done using SaltStack and Jinja templates. This system is also securing that the service will be always running, for example, if the frontends and backend is accidentally wiped clean, this system takes care that everything can be installed on the servers again automatically and everything is running as it should be.

The service will also include state, which is called *highstate*. Highstate will roll on every server, that is included with highstate, once in an hour and the state will make

sure everything is running as it should. This is also monitoring tool, because if the highstate fails, it will send email that the highstate has failed in the service and needs to be fixed. Anything can fail the highstate, from typo in configuration to deleted configuration file. Also, when the highstate is ran, it will update modified configuration files and this way its automation tool itself.

3.6 Salt pillar files

First things that are needed is the pillar files for frontend and backend, which role is to store all the attributes that are needed, and they will be looking like in figure 7 and 8.

```
apache:
  directory: '/some/directory'
  log:
    directory: '/some/directory/logs'
    level: 'error'
  pid_file: '/var/run/apache.pid'
  port: 8000

static:
  content:
    delivery:
      backends:
        - backend.server.1
      directory: '/some/directory'
```

Figure 7 Frontend pillar -file

In figure 7, is the frontend pillar file. Attributes that are in pillar file, are things that can sometime in the future, and that's why they are chosen in this pillar file. When something needs to be changed in frontends, they can be changed in the pillar file, and they will change in the server too, and the frontend servers never needs to be touched.

```
lsync:
  log_directory: '/some/directory'
  pid_file: '/var/run/lsyncd/lsyncd.pid'
  ssh_public_key: 'ssh-rsa AAAF24akaWAkr24...'

static:
  content:
    delivery:
      directory: '/some/directory'
      frontends:
        - frontend.server.1
        - frontend.server.2
      directories:
        'contentdirectory1'
          - ACL_group1
          - ACL_group2
        'contentdirectory2'
          - ACL_group1
          - ACL_group2
```

Figure 8 Backend pillar -file

In figure 8, is the backend pillar file, same as in frontend pillar file, the attributes are things that can be changed, but there are few differences in frontend and backend in the *static:content:delivery* section. Directories section in backend pillar file are the subdirectories of the content directory, which are the source directories of the replicated content. These directories are given the access rights to the right groups who needs to use them and with Lsyncd they are sent to the frontends and they will keep the user rights, therefore this isn't needed to put on the frontend pillar file. The *lsync:ssh_public_key* attribute is just a placeholder for future, it is not implemented in the server, SSH keys will be used in different way.

3.7 SSH keys

SSH keys for this service will be made in the backend Salt State file and the public part of the key will be sent to the salt stacks mine module, where frontend will be picking the public key from and placing it for static user to use. This way the static user can get the data from the backend to frontends for delivery.

3.8 Salt state files

Next things that are needed are the state files, which role is to install and configure everything in the servers. In frontend and backend, configuration files will be made with Jinja templates which the state file will put in place with right settings. In backend the software's which needed own configuration files were Logrotated and Lsyncd and in frontends the software was Apache and Logrotate.

Salt State for backend needs to do following things. Take pillar attributes to the state files which were introduced in figure 8, put correct user rights to files and folders in server, create SSH keys for the static user which will be delivering content from backend to frontend with Lsyncd, install Lsyncd and Rsync to the server and make sure they are running for content delivery, put the Lsyncd and Logrotate configuration files in right place with correct settings, create the SSH key and update in the SaltStack mine module for static user.

Salt State for frontends needs to do following things. Take pillar attributes to the state files which were introduced in figure 7, put correct user rights to files and folders in servers, make sure apache is installed, put the Apache and Logrotate configuration files in right place in the server with correct settings, put correct user rights for the static user, get the SSH key from mine module and place it for static user and make sure the apache is running on the servers.

3.9 Monitoring

Service monitoring is created by different branch of KELA, but it is shown in this chapter how it is done.

Httpd process in port 8000 is monitored in frontend servers by ping, and if the port 8000 doesn't give any answers, email will be sent, so someone can fix whatever problem the service might have, message in the email looks like this: *httpd-process is not responding on node: frontend.server.1 , timestamp: 12.12.2019 17:00:32 ,status FAIL ,port: 8000*.

In backend servers, Lsyncd process will be monitored in process table. If Lsyncd process is not in the process table, email will be sent again, and it would look like this: *lsycnd-process is missing from backend.server.1:LZ*.

3.10 Information security

Organization big as KELA, and when thinking of services that KELA offers, security is a must. KELA basically has information of every citizen of Finland, and this information falling into the wrong hands would be a disaster. KELA has strict rules and laws it must follow when there is personal data in question.

For example, logging, everything must be audited and the logs need to be kept, stored and back upped from 2 to 5 years, so they can be used to track down any kind of abusive use of any service or attacks towards them. Also, usage of the logs must be logged, so no one can modify anything without it being seen. (Katakri, 2015, 46.)

As said previously in this thesis, the content will be basically only CSS files, and there will be no personal data of any kind included in this service. Therefore, security part of this service is not that important in this service. For example, if someone who isn't supposed to, in some way got hands on the content, the person couldn't cause any harm to anyone or anything. Even though this said, security is always important in any service, and this service will have it too. For example, in this service Apache is hardened, and certain access groups are needed for using the service.

4 Implementation

4.1 Backend

Backend is the database of this server, which will include all the content and where the content will be added. Technical side of the backend is explained and built in this chapter. This includes configuring Lsyncd and Logrotate, also these configurations will be automatized completely.

4.1.1 Salt pillar

As mentioned in SaltStack theory part, pillar files are only for storing keys and values. In backend value for Lsyncd log directory is needed, so it can be used in Lsyncd configuration file. This is set in pillar file, because if it needs changing, it can be made without effort. Same goes for Lsyncd PID file, it is next value that is stored in pillar file.

Those were the values for the Lsyncd -program, next up are values for the backend service itself. First is value for the content directory, this is important because it is used in places such as Lsyncd configuration file and backends salt state file. Next up is list of frontend servers, which is needed for the Lsyncd configuration file, so Lsyncd can authorize access via SSH to right servers, where content is shared.

Last things that need to be stored in pillar file is a list of subdirectories in the content directory and a list of access groups that can read, write or execute in those subdirectories. This is made this way, so it is easy to add, remove or edit subdirectories or access groups. Now the pillar file looks like in demonstration figure 9 and now values can be retrieved for salt state files.

```
lsync:
  log_directory: '/path/to/log_directory'
  pid_file: '/path/to/pid_file.pid'

static_content_delivery:
  content_directory: '/path/to/directory'
  list_of_frontend_servers:
    - frontend.server.1
    - frontend.server.2
  sub_directories:
    'directory1':
      - access.group.1
      - access.group.2
    'directory2':
      - access.group.1
```

Figure 9 Salt pillar demonstration picture

4.1.2 Salt state

First things needed in salt state file are variables for every value that is put in the pillar file. This is because pillar file values will be used in many places and to use them, certain module is running every time, which gets the value, that is needed from a key in a pillar file. This module is called *pillar.get*. (salt.modules.pillar.get, 2020.) Instead of writing the *pillar.get* command repeatedly, it can be put in a variable just once, and the variable can be used repeatedly, this can be done by mixing some programming in the salt state file. Setting the variable is demonstrated in figure 10.

```
{% set content_directory = salt['pillar.get']('static_content_delivery:content_directory', '/path/to/directory') %}
```

Figure 10 Setting variable

First thing done in figure 10 is deciding what is the name of this variable, in this case it is *content_directory*. After this the *pillar.get* module is used, which is the *salt['pillar.get']* part in figure 10, this command gets the value for the key *static_content_delivery:content_directory*, which is */path/to/directory*, this was set in the pillar file part and can be seen in figure 9. */path/to/directory* part in figure 10, is default value for the variable *lsyncd_content_directory*, this is set because if there was no value set in the pillar file for the key *static_content_delivery:content_directory*, then this default value would be used and this way the whole automation for this service wouldn't fail because of the missing value.

After all variables are set, packages need to be installed. This is done by *pkg.installed* module. (salt.states.pkg.installed, 2020.) This module is used by giving it a package name that needs to be installed, and the automation does the rest. It gets the package from YUM repository and installs it, this is demonstrated in figure 11.

```
rsync:
  pkg.installed

lsyncd:
  pkg.installed
```

Figure 11 Backends packages

The only packages that are used in this services backend are Rsync and Lsyncd.

Now when installing packages are automatized, next thing is to automatize creating directories and putting correct user rights to them. This can be done with SaltStack module called *file.directory*. (salt.states.file.directory, 2020.) Let's demonstrate this with the content directory. First, path of the directory must be given, which is */path/to/directory*, but this directory is set in variables so it should be used instead. After this, the directory needs owner user, ownership group and user rights, which need to be put in octal numbers. This is demonstrated in figure 12.

```
{{ content_directory }}:
 file.directory:
    - group: root
    - mode: 0755
    - user: root
    - makedirs: true
```

Figure 12 Directory creation

Owner and ownership group is root, because no one is supposed to remove or edit the directory, only people with root user rights can do it if necessary and usually those are the system administrators. *mode: 0755* means that if user is set to get rights to the directory, then that user can read, write or execute in that directory. And if user group is set to have rights to the directory, then members of that group can read and execute in the directory. Everyone else doesn't have any rights to the directory. (chmod(1) – Linux man page, 2010.) Line *makedirs: true* is there because if directories before the content directory does not exist, in this case directories *path* and *to*, then automatization will create those directories first and the automatization does not fail.

Automatization of subdirectories of the content directory is little bit trickier, because those directories needs to be created from a list. This can be done with programming. First, subdirectories need to be put in variable, so code will be simpler, and this is demonstrated in figure 13.

```
{% set content_subdirectories = salt['pillar.get']('static_content_delivery:sub_directories', {}) %}
```

Figure 13 Variable for subdirectories

Few things that is different compared to variable of content directory are, the path where the values are got from and default value. There is no default value, because there isn't only one directory, there is multiple directories in a dictionary. This means that folders are keys and they have a value, which is in this case name or names of access groups, so one folder can have more than one value. Now to the automatization part which is demonstrated in figure 14.

```
{% for (dir, groups) in content_subdirectories.items() %}
{{ content_directory }}/{{ dir }}:
  file.directory:
    - group: root
    - mode: 0755
    - user: root
{{ content_directory }}/{{ dir }}-acl:
  cmd.run:
    - name: |
        {% for group in groups %}
        setfacl -Rm g:{{ group }}:rwx {{ content_directory }}/{{ dir }}
        setfacl -Rm d:g:{{ group }}:rwx {{ content_directory }}/{{ dir }}
        {% endfor %}
{% endfor %}
```

Figure 14 Automatization of subdirectories

There is a for loop that goes through all items from subdirectories in pillar file. Then it makes subdirectory for every key found in the subdirectories, for example first subdirectory would be */path/to/directory/directory1*. After making this directory, the loop would give access right to this directory for every value found in the pillar file from the key, for example, users in *access.group.1* and *access.group.2* would get read, write and execute rights via ACL to this subdirectory. Then it would do the same again for *directory2*, first create directory and give it user rights. After every subdirectory is looped through, it stops and then move on in state file. This way, creating new subdirectories is very easy, administrators just must add name of the subdirectory and access groups to pillar file, and subdirectory will be created, and user rights will be added automatically to the server.

Now when directories are created, next up is creating few files that needs to be in the backend server. These files are configuration file for Lsyncd and Logrotate, which

will be Jinja templates as well as is this state file. This way configuration files can be automated as well, and they will do only things they are meant to do. This can be done with SaltStack module called *file.managed*. (salt.states.file.managed, 2020.) Using this module is demonstrated in figure 15.

```
/etc/lsyncd.conf:
  file.managed:
    - context:
        content_directory: {{ content_directory }}
        frontends: {{ apache_frontends }}
        log_directory: {{ lsyncd_log_directory }}
        pid_file: {{ lsyncd_pid_file }}
    - source: salt://{{ slspath }}/lsyncd.conf
    - user: root
    - group: root
    - mode: 0644
    - template: jinja
```

Figure 15 Creating files

As seen in figure 15, first there is */etc/lsyncd.conf:,* which is the path and the name of the file that needs to be created on the server. Next in the *context:* section is the variables which needs to be used in the Lsyncd configuration file, which were created in the beginning of this Salt State file. *source:* part is the path of the *lsyncd.conf* template file, which is copied in the server and the *{{ slspath }}* means that it is in the same directory as salt state file which is now being created. Last thing this salt state needs to do, after everything else is done, is to check that Lsyncd is running, this is done with *service.running* module (salt.states.service.running, 2020.), which is shown in figure 16.

```
lsyncd-running:
  service.running:
    - enable: True
    - name: lsyncd
```

Figure 16 lsyncd service running

Now that backend salt state file is done, configuration templates need to be done next.

4.1.3 Lsyncd configuration

In previous chapter the automation was created, which creates the lsyncd configuration file from a jinja template. Now the template needs to be created, that includes right configuration for our backend server. This configuration needs to create the link between frontend and backend server, where the content is being shared. In this chapter, the configuration is being explained step by step. The configuration file must be build using LUA scripting language. First settings need to be put in the configuration file. (The Configuration File, N.d.) In these, path to PID file and log file are only things needed, this is demonstrated in figure 17.

```
settings {
  logfile = "{{ log_directory }}/lsyncd.log",
  pidfile = "{{ pid_file }}",
}
```

Figure 17 lsyncd settings

In here the variables can be used, which were set in the pillar file. Using these, like in figure 17, will set logfile to */path/to/log_directory/lsyncd.log* and PID file to */path/to/pidfile.pid*. After settings, list of frontend servers needs to be created. This can be done using jinja templating, since the list of frontend servers can be found in pillar file again, this is demonstrated in figure 18.

```
targetlist = {
  {% for target in frontends -%}
    "{{ target }}",
  {% endfor -%}
}
```

Figure 18 list of frontends

In here, variable called target list is created, which includes list of frontend servers, that are found in the pillar file. There is a for loop that goes through the frontends and puts one item from the list on its own row. It is done this way, so if frontend server gets removed, edited, or added, there is no need to edit the Lsyncd configuration file. Instead, the pillar file needs to be changed and then it changes the Lsyncd

configuration file as well. This way, system administrator does not need to know how the Lsyncd configuration file works. Now that target list is done, next is loop which configures Rsync for every frontend, (Config Layer 4: Default Config, N.d.) this is demonstrated in figure 19.

```
for _, server in ipairs( targetlist ) do
  sync {
    default.rsyncssh,
    source = "{{ content_directory }}",
    host = server,
    targetdir = "{{ content_directory }}",
    delay = 2,
    rsync = { _extra = { "--omit-dir-times",
                         "-e", "ssh -l static -i /path/to/content_directory/pki/rsa -o StrictHostKeyChecking=no" }
    }
  }

end
```

Figure 19 rsync settings

In figure 19, first there is a for loop that puts same sync settings for every frontend server. After this comes sync settings, first there is setting called *default.rsyncssh*, which means that Lsyncd uses Rsync SSH to share the content. *Source* is the directory where the content is in backend server and *targetdir* is where the content is put in the frontend server, *host* is the address of the frontend server and *delay* is time in seconds that syncing content takes between backend and frontend. Last there is *extra* settings for Rsync. *—omit-dir-times* means that Rsync does not preserve times when directories were edited. *-e* specifies remote shell usage (rsync(1) – Linux man page, N.d.) and the last command *ssh -l static -I /path/to/content_directory/pki/rsa –o StrictHostKeyChecking=no*, sets SSH to login with user *static* using identity file, which is in this case private RSA key which has path */path/to/content_directory/pki/rsa*. *-o StrictHostKeyChecking=no* means that when user *static* logs for the first time to destination server, the destination server adds static user to list called known hosts. (ssh(1) – Linux man page, 2013.) This way, when static user wants to connect to the destination server, which is in this case the backend, the server will let static user to connect automatically without checking the key every time. (ssh_config(5) – Linux man page, 2013.) This is done this way, because if *StrictHostKeyChecking* didn't have value of *no*, every time static user wants to connect to the backend, it would ask a question where you have to type *yes* or *no*, and because everything is automated, there is no one answering to that question and the machine can't do that

by itself. This might be considered as big vulnerability in the service, but in this service, there is no classified content, so it does not matter that much.

4.1.4 Logrotate configuration

Last for the backend, Logrotate configuration needs to be done, because worst case scenario the backend servers disk space fills up with logfiles and content does not fit in the content directory anymore and that happening would be unfortunate. Logrotate configuration is demonstrated in figure 20.

```
{{ log_directory }}/lsyncd.log {
  compress
  copytruncate
  createolddir
  daily
  dateext
  missingok
  olddir {{ log_directory }}/rotated
  rotate 7
}
```

Figure 20 logrotate configuration

Compress means that old log files will be compressed using Gzip, so they won't fill the disk space so much. *Copytruncate* cuts original log file, instead of moving old logfile and creating a new one to replace it. *Createolddir* creates directory for the old logfiles where they are moved. *Daily* makes sure that log files get rotated daily. *Dateext* adds year, month and day to log files name, this way it's easier to see what which days log it is. *Missingok* means that, if the log file is missing, it gets skipped without error message. *Olddir* sets path to the directory where old logfiles are moved and stored. *Rotate* is a count how many times logfile is rotated before being removed, in this case it means its rotated 7 times, which means that log file is being stored for 7 days before being removed. (logrotate(8) – Linux man page, N.d.)

4.2 Frontend

Now it's frontends turn to be configured. Configuring frontend goes same way as backend, difference is that in frontend, Apache needs to be configured, in backend

Lsyncd was configured. Frontends Logrotate configuration is done the same way as in backend; therefore, it is not explained again in this section.

4.2.1 Salt pillar

First pillar file needs to be configured, because keys and values are needed for automation. Keys and values are needed for the Apache, so let's start with that. It's good to have Apache directory in the pillar file because if it needs to be changed, it can be done with the pillar file. For the same reason, it's good to have apache log directory and log level in the pillar file. Location to apache PID file and the port that apache listens to is also needed in the pillar file. Again, if one of these needs to be changed, it can be done in the pillar file instead of searching the value in every configuration file. Now the pillar file can be seen in figure 21.

```
apache:
  directory: '/path/to/apache_directory'
  log:
    directory: '/path/to/apache/logs_directory'
    level: 'error'
  pid_file: '/path/to/apache_pidfile.pid'
  port: 8000
```

Figure 21 Frontend pillar file

Log level is set to error, because if there are any errors in Apache, these needs to be logged, so it can be fixed. Port is set to 8000, because the default HTTP port is 80, so the port 8000 is easy to remember.

Now that Apache keys and values are set in the pillar file, next some frontend default paths and values are good to have in the pillar file. These keys and values are the content directory in the frontend and list of backend servers. It's good to have backend servers in the pillar file even if they are not used anywhere, this can help with seeing what servers the backend servers are and these doesn't need to be searched somewhere and if someday more than one backend server is needed, it's easy to add to the list in the pillar file. These can be seen in figure 22.

```
static_content_delivery:
  backends:
    - backend.server.1
  directory: '/path/to/content_directory'
```

Figure 22 Frontend pillar file

4.2.2 Salt state

Making frontends salt state file, goes same way as it did with backend. There are few things that are different in frontend. For instance, instead of Lsyncd now Apache needs to be installed and configured. There are also few extra things that needs to be made in the process. One of these things is shown in figure 23.

```
{% if grains['os_family']|lower == 'redhat' %}
httpd:
  pkg.installed
{% else %}
apache2:
  pkg.installed
{% endif %}
```

Figure 23 Installing apache

Apache has different package names in different operating system. So, if statement is needed here. If the operating system is Redhat, package named *httpd* is installed, which is Apache. If operating is something else than Redhat, for example Ubuntu, then package called *apache2* is installed. This way, the service can be installed on any operating system and there is no need to think what operating system the service needs to be installed on, this helps the automation, because the less you need to think on installing the service, the better. This way anyone can install the service using SaltStack commands, this is demonstrated in the results section.

Next, some changes are needed in the default *httpd* system configuration file. In this file it must be made sure, that Apache uses our configuration file, which will be built from template like the Lsyncd configuration file. Also, System needs to know that Apache is installed in different location and not in default location. This is made this

way, because it is easier to then find all directories and files, when they are in the same location.

Frontends salt state file also contains similarities that were configured in the backend, to help the automation in the service creation. These things are automatically creating directories and files, also putting correct user rights to them. Also, there is creating the configuration files with templates, such as the Apache configuration file.

4.2.3 Apache configuration

Apache configuration is straight forward, there is not much in behalf of automation. There are few settings that depends on an operating system used, for example if operating system is Redhat, then Apache user is *apache* and group is *apache* and if it's something else, then user is *www-data* and group is *www-data*. This can be seen in figure 24.

```
{% if grains.os_family|lower == 'redhat' %}
User apache
Group apache
{% else %}
User www-data
Group www-data
{% endif %}
```

Figure 24 Setting user and group

Also, few load modules are operating system dependent, for example if operating system is Redhat, then it needs to be set in configuration file, that load modules called *log_config*, *systemd* and *unixd* needs to be set manually on the configuration file, if operating system is something else, these modules will be loaded automatically. These settings can be seen in figure 25.

```
{% if grains.os_family|lower == 'redhat' %}
LoadModule log_config_module modules/mod_log_config.so
LoadModule systemd_module modules/mod_systemd.so
LoadModule unixd_module modules/mod_unixd.so
{% endif %}
```

Figure 25 Operating system dependant loadmodules

Apache is hardened, in other words, hardening means that Apache is made as secure as possible. This means that Apache is configured not to list directories, when the website is entered. This way it is secure, and the site user needs to know exactly what to look for and the user does not see all the content that is in the server. As said before, it also has only the modules that are necessary. Also, the Apache status page can only be accessed from KELAs intranet.

4.2.4 SSH with static user

Connection and content sharing are automated with using a user called *static* and SSH connection. It is demonstrated in this chapter, how this works. To make this happen, first creating the SSH keys are needed. This is done in the backends Salt State file and is demonstrated in figure 26.

```
key-generate:
  cmd.run:
    - name: ssh-keygen -b 2048 -t rsa -f /path/to/key_file
    - unless: test -f /path/to/key_file
```

Figure 26 Creating SSH keys

Creating the SSH keys are done with the salt module called *cmd.run*. (Salt.States.Cmd, 2020.) *Cmd.run* will execute a command *ssh-keygen -b 2048 -t rsa -f /path/to/key_file*, which creates RSA keys that has 2048 bits in it and the keys are created in the location that is seen in the figure 26. (SSH-KEYGEN (1), N.d.) For this service this key is strong enough, because there is nothing classified anyway in the service. *Unless* part in the figure 26, means that it tests if the key already exists in the location, this means the module won't create a new RSA key, if there is one already created. Now that the SSH keys are created, public key needs to be put in the mine module, so frontend can retrieve it. This is demonstrated in the figure 27.

```
mine_functions:
  scdn_backend_public_key:
    mine_function: cmd.shell
    cmd: test -f /path/to/key_file.pub && cat /path/to/key_file.pub || echo ''
```

Figure 27 Putting key to mine module

In the figure 27, mine function tests if the public key exists and if it does, it will be printed in the module. When the key is printed, frontend can fetch the key using mine module, this is shown in figure 28.

```
[User@Frontend.server.1 ~]$ sudo salt-call mine.get backend.server.1 scdn_backend_public_key
local:
    ----------
    backend.server.1:
        ssh-rsa AAAAB3NzaC1... root@backend.server.1
```

Figure 28 Fetching public key

As seen in the figure 28, the public key from the backend server can be fetched. Now that this is tested, fetching the public key for *static* user needs to be done and it is done in the frontend salt state server. This is demonstrated in figure 29.

```
{% if (ssh_pub_key is string and
       ssh_pub_key|length > 40 and
       ssh_pub_key.split()|length > 1) %}
   {% set type = ssh_pub_key.split()[0] %}
   {% set key = ssh_pub_key.split()[1] %}
static-rsa:
  ssh_auth.present:
    - user: static
    - enc: {{ type }}
    - name: {{ key }}
{% endif %}
```

Figure 29 Fetching pub key for static

It is not shown in the figure 29, but *ssh_pub_key* is variable for *mine_data.get*, which gets the public key from the mine. First before it can be placed for *static* user to use, some sanity checks need to be done so it can be confirmed that the key really works and is correct. After sanity checks, the key needs to be split, as seen in the figure 29, first part of the mine get print is the encryption type, and the second part is the key. Therefore, it needs to be split, so the correct value goes to correct key.

Now that the key is created and placed in correct place with automation, the static user needs to be created. This is demonstrated in figure 30.

```
accounts:
  static:
    fullname: "Static Content Delivery User"
    home: "/home/static_user"
    uid: 12345
    gid: 12345
    createhome: True
```

Figure 30 Creating static user

This user called *static* is being used by the frontend to retrieve content from the backend and it is fully automatized. In figure 30, the user is created, and it will have own home folder, user ID and group ID.

5 Results

After the configuration files are made, next up is testing the service. First these configuration files need to be put for server to use. This will happen by putting the file paths to the server's salt pillar file. in KELA, each server, even the empty ones have their own salt pillar files, where the server can be modified through SaltStack. These files include user rights, roles, SSH rights etc. So, these salt state files need to be put in the roles, because the servers are empty and they have their own roles, like backend and frontend.

After roles are put, the service can be executed. This will happen with command *Sudo salt-call state.apply role* (salt.modules.state.apply, 2020.), this command is custom made, it will read role part in the server's salt pillar file and executes all salt state files that are found in the roles section of this pillar file. For example, when this command is executed in the frontend server, it will install Apache, create all the directories that is needed, configure Apache and Logrotate etc.

In figure 31 can be seen that the automation and salt mine usage work just like it is supposed to. The key is created, placed in the mine and the frontend can fetch it from there and put it to use for the static user.

```
----------
          ID: static-rsa
    Function: ssh_auth.present
        Name: AAAAB3NzaC1.......................
      Result: True
     Comment: The authorized host key AAAAB3NzaC1
     Started: 13:17:16.627106
    Duration: 2.9 ms
     Changes:
----------
```

Figure 31 SSH key for static user

Running the commands in frontend and backend were success, everything worked as supposed to. These results can be seen in figure 32 for backend and in figure 33 for frontend.

```
----------
          ID: lsyncd-running
    Function: service.running
        Name: lsyncd
      Result: True
     Comment: The service lsyncd is already running
     Started: 12:33:43.381383
    Duration: 34.166 ms
     Changes:

Summary for local
-------------
Succeeded: 25 (changed=5)
Failed:     0
-------------
Total states run:     25
Total run time:   16.495 s
```

Figure 32 Role run for backend

As can be seen, there were no failures for running the role command in backend server, and the service Lsyncd is running. This means it is configured right and there were no errors which would not start the Lsyncd service.

```
----------
          ID: httpd-running
    Function: service.running
        Name: httpd
      Result: True
     Comment: The service httpd is already running
     Started: 13:17:16.669045
    Duration: 29.54 ms
     Changes:

Summary for local
-------------
Succeeded: 28 (changed=7)
Failed:     0
-------------
Total states run:     28
Total run time:    1.621 s
```

Figure 33 Role run for frontend

Same goes for frontend, running the command was success, and there are no failures either. This means that now the service is up and running, also the content should be moving from backend to frontend and from frontend to the requester. This can be confirmed by looking at Apache service status, if there is any traffic. This can be seen in figure 34.

```
Redirecting to /bin/systemctl status httpd.service
* httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service; enabled; vendor preset: disabled)
   Active: active (running) since Mon 2019-12-09 23:15:02 EET; 1 day 14h ago
     Docs: man:httpd(8)
           man:apachectl(8)
 Main PID: 1182 (httpd)
   Status: "Total requests: 382566; Current requests/sec: 3.31e+03; Current traffic:  48MB/sec"
```

Figure 34 Apache service status

Figure 34 shows, that there is traffic and requests for the content are coming non-stop. This means that the service is working as intended.

The service was built in steps, this means that the *state.apply* -command was pretty much spammed on the server a lot, after any fail or mistake in the configuration. Once the command went through without any errors and everything was working, the service was stated as complete and fully functional. Service was also wiped entirely clean, meaning the backend server and both fronted servers was removed from

all applications, software and all directories were removed and the user that was made for this service. After this the command was ran and it was made sure that the whole service was built with the single *state.apply* command on empty servers.

6 Conclusion and discussion

Before making this service, KELA had similar one in the testing branch. This service was hard coded, meaning there was no automatization of any kind. There also were not any kind of documentation of the service, for example where to find the servers, what is installed on the servers, locations of configuration files or directories et cetera. There was only a manual of how to add content to the service, this was the foundation and starting point of this thesis project and it was not production ready.

Objective of this thesis was to create this new service, that is fully automated using SaltStack, is production ready and fully documented in KELAs intranet. To achieve this objective there were steps that are not documented on this thesis, these included ordering servers, user groups, networks and firewall openings, et cetera from different branches of KELA. These objectives were successfully achieved and were not documented because these things are needed to keep as company secrets.

What comes to objective of creating this service and automatizing it was a success. There was nothing that the service needed and was not successfully created. This was achieved with the help of a colleague, internet pages and KELAs documentation. The service itself is simple, hardest part was creating the automation using Jinja programming and thinking of how to make it happen. Creating this service took approximately 2 to 3 months of constant work, it was worked on full time until it was ready when completing work internship for school in KELA. Few times there were hard times in creating this service, when planning how to program something in a way it need to work, but with help of a good work supervisor or more like a mentor in KELA who was pushing forward time after time by giving tips on the programming part and on everything else.

The service was very interesting to create, there was some experience of Apache configuration beforehand and little experience of Python programming. But while

making this service happen, there was a lot of learning about Apache and other programs that were used on this service. Most was learned from jinja programming and SaltStack itself, the SaltStack product and how it is used in KELA became very familiar, because it was used a lot in debugging.

There is not much to do for further development for this service, maybe there could be a server for the log files, where the log files were rotated, so the logs would not need to be removed to save some space on the servers. In addition, there was speaking of using Samba file server for the content users, so it would be easier to edit, add or remove the content just by opening the content folder on the windows machine, but it was denied because this way was not as safe as using SSH connection.

The hardest part of this whole thesis project was to write down the thesis, the service was done in a few months, but writing it all down took over a year. It was hard to plan on how this all should be documented and there were some problems with different things and desperate times as well. But there really isn't anything that would be done differently with the service if it had to be built again. However, maybe there are some things that could be done differently if this thesis had to be written this thesis again. In conclusion, this project has been at times demanding and difficult, however, rewarding when implementing the fault-tolerant and highly scalable content delivery service was done successfully for KELA.

References

About Apache. 2020. About page in apache home page. Accessed on 28.8.2020. Retrieved from https://httpd.apache.org/ABOUT_APACHE.html

Apache Module mod_access_compat. 2020. mod_access_compat manual page. Accessed on 28.8.2020. Retrieved from https://httpd.apache.org/docs/2.4/mod/mod_access_compat.html

Brown, P. Kaluza, J. Troan, E. N.d. logrotate(8) – Linux man page. Accessed on 16.8.2019 and 1.3.2020. Retrieved from https://linux.die.net/man/8/logrotate

Campbell, A. Beck, B. Friedl, M. Provos, N. Raadt, T. Song, D. 2013. ssh(1) – Linux man page. Accessed on 13.7.2020. Retrieved from https://linux.die.net/man/1/ssh

Campbell, A. Beck, B. Friedl, M. Provos, N. Raadt, T. Song, D. 2013. ssh_config(5) – Linux man page. Accessed on 28.2.2020. Retrieved from https://linux.die.net/man/5/ssh_config

Campbell, A. Beck, B. Friedl, M. Provos, N. Raadt, T. Song, D. N.d. SSH-KEYGEN(1). Accessed on 16.3.2020. Retrieved from https://linux.die.net/man/5/ssh_config

Config Layer 4: Default Config. N.d. lsyncd configuration manual page. Accessed on 26.2.2020. Retrieved from https://axkibe.github.io/lsyncd/manual/config/layer4/

Elämässä mukana – muutoksissa tukena. 2018. Article on KELA. Accessed on 24.5.2019. Retrieved from https://www.kela.fi/kela-lyhyesti.

Fielding, R. Gettys, J. Mogul, J. Frystyk, H. Masinter, L. Leach, P. Berners-Lee, T. 1999. Hypertext Transfer Protocol. Accessed on 7.6.2019. Retrieved from https://tools.ietf.org/html/rfc2616

Getting started. 2020. Apache manual page. Accessed on 24.8.2020. Retrieved from http://httpd.apache.org/docs/2.4/getting-started.html

Gruenbacher, A. 2003. Posix Access Control Lists on Linux. Accessed on 9.8.2019. Retrieved from https://www.usenix.org/legacy/events/usenix03/tech/freenix03/full_papers/gruenbacher/gruenbacher_html/main.html

How do I use Salt States?. 2020. Saltstack tutorial page. Accessed on 28.8.2020. Retrieved from https://docs.saltstack.com/en/latest/topics/tutorials/starting_states.html

Jinja. 2017. Homepage of jinja website. Accessed on 16.9.2019. Retrieved from https://jinja.palletsprojects.com/en/2.10.x/

Kansaneläkelaitos Toimintakertomus ja tilinpäätös 2018. 2018. Report of KELA 2018. Accessed on 24.5.2019. Retrieved from https://www.kela.fi/documents/10180/23661953/Tilinp%C3%A4%C3%A4t%C3%B6s+ja+toimintakertomus+2018.pdf/706185ba-8a0a-45f3-afe2-205cd0d76d92

Katakri. 2015. Tietoturvallisuuden auditointityökalu viranomaisille. Accessed on 17.9.2020. Retrieved from

https://www.defmin.fi/files/3165/Katakri_2015_Tietoturvallisuuden_auditointityokalu_viranomaisille.pdf

Kittenberger, A. N.d. lsyncd(1) – Linux man page. Accessed on 9.8.2019. Retrieved from https://linux.die.net/man/1/lsyncd

MacKenzie, D. Meyering, J. 2010. chmod(1) – Linux man page. Accessed on 13.2.2020. Retrieved from https://linux.die.net/man/1/chmod

Näin Kela palvelee. 2019. Article on Kela. Accessed on 5.2.2020. Retrieved from https://www.kela.fi/nain-kela-palvelee

Qualitative Research: Definition, Types, Methods and Examples. blog page on questionpro. Accessed on 18.9.2020. Retrieved from https://www.questionpro.com/blog/qualitative-research-methods/

Rescrola, E. Schiffman, A. 1999. The Secure Hypertext Trasnfer Protocol. Accessed on 7.6.2019. Retrieved from https://tools.ietf.org/html/rfc2660

Rouse, M. 2018. Saltstack. Article on TechTarget. Accessed on 28.8.2020. Retrieved from https://searchitoperations.techtarget.com/definition/SaltStack

SSH (Secure Shell). 2020. Secure Shell page. Accessed on 19.8.2020. Retrieved from https://www.ssh.com/ssh/

salt.modules.pillar.get. 2020. Salt module pillar.get manual page. Accessed on 12.2.2020. Retrieved from https://docs.saltstack.com/en/master/ref/modules/all/salt.modules.pillar.html#salt.modules.pillar.get

salt.modules.state.apply. 2020. Salt module state.apply manual page. accessed on 13.7.2020. Retrieved from https://docs.saltstack.com/en/master/ref/modules/all/salt.modules.state.html#salt.modules.state.apply_

Salt.States.Cmd. 2020. Salt state cmd manual page. Accessed on 16.3.2020. Retrieved from https://docs.saltstack.com/en/latest/ref/states/all/salt.states.cmd.html

salt.states.file.directory. 2020. Salt state file.directory manual page. Accessed on 13.2.2020. Retrieved from https://docs.saltstack.com/en/latest/ref/states/all/salt.states.file.html#salt.states.file.directory

salt.states.file.managed. 2020. Salt state file.managed manual page. Accessed on 26.2.2020. Retrieved from https://docs.saltstack.com/en/master/ref/states/all/salt.states.file.html#salt.states.file.managed

Salt.States.Linux_ACL. 2020. Salt state Linux ACL manual page. Accessed on 18.8.2020. Retrieved from https://docs.saltstack.com/en/latest/ref/states/all/salt.states.linux_acl.html

salt.states.pkg.installed. 2020. Salt state pkg.installed manual page. Accessed on 12.2.2020. Retrieved from

https://docs.saltstack.com/en/latest/ref/states/all/salt.states.pkg.html#salt.states.pkg.installed

salt.states.service.running. 2020. salt state service.running manual page. Accessed on 13.2.2020. Retrieved from https://docs.saltstack.com/en/latest/ref/states/all/salt.states.service.html#salt.states.service.running

The Configuration File. N.d. lsyncd configuration manual. Accessed on 26.2.2020. retrieved from https://axkibe.github.io/lsyncd/manual/config/file/

The Salt Mine. N.d. SaltStack manual page. Accessed on 6.9.2020. Retrieved from https://docs.saltstack.com/en/3000/topics/mine/index.html

Tridgell, A. Mackerras, P. N.d. rsync(1) – Linux man page. Accessed on 7.6.2019 and 28.2.2020. Retrieved from https://linux.die.net/man/1/rsync

Understanding Jinja. 2020. Salststack tutorial page. Accessed on 30.8.2020. Retrieved from https://docs.saltstack.com/en/latest/topics/jinja/index.html

Understanding YAML. 2020. SaltStack tutorial page. Accessed on 28.8.2020. Retrieved from https://docs.saltstack.com/en/master/topics/yaml/index.html