Presentation

Richard Zetterberg <richard.zetterberg@gmail.com>

Contents

1 About me **2**

2 Showcase of large project **3**

2.1 Installations 4
2.2 User interface 6
2.3 Signage devices 7
2.4 Content management system 9
2.5 App management system 10
2.6 Infrastructure 11
2.6.1 Early architecture 11
2.6.2 Late architecture 12

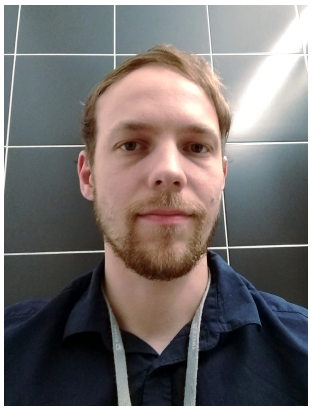
3 Showcase of medium project **14**

3.1 Problem 16
3.2 Solution 17

Abstract

This document contains a presentation about me and show cases of 2 of the projects I have worked on during the last 2 years.

1 About me



I'm a person who is constantly improving myself and my skills. I always try to step outside of my comfort zone to avoid being stuck in a single way of thinking. My strengths are:

- Abstract thinking
- Learning new things quickly
- Being thorough

I'm a very self-driven person, I get motivated by being trusted with important tasks, but collaboration also motivates me.

Besides working with software professionally, I have personal projects that I work on my spare time. I like to share my projects with others and I'm a proponent of FOSS (*free and open-source software*). Besides working with software, I also like writing about software.

My projects https://github.com/rzetterberg

My website https://rzetterberg.github.io

2 Showcase of large project

From 2015 to 2017 I worked as a solution architect and developer the startup Nordic Digital Solutions. The company was an all-inclusive digital signage supplier. My role was to plan, develop and maintain a digital signage system from scratch.

There were another developer besides me that started working at the company after this system had been rolled out in production. That developer was responsible for developing the HTML5-apps for the devices and handling support cases.

When a client signed up for a 3 year contract, the company delivered and installed the monitors and signage devices at the clients location using a sub-contractor. Then the client would get access to a content management system where they could login and control what and how the devices would show their content.

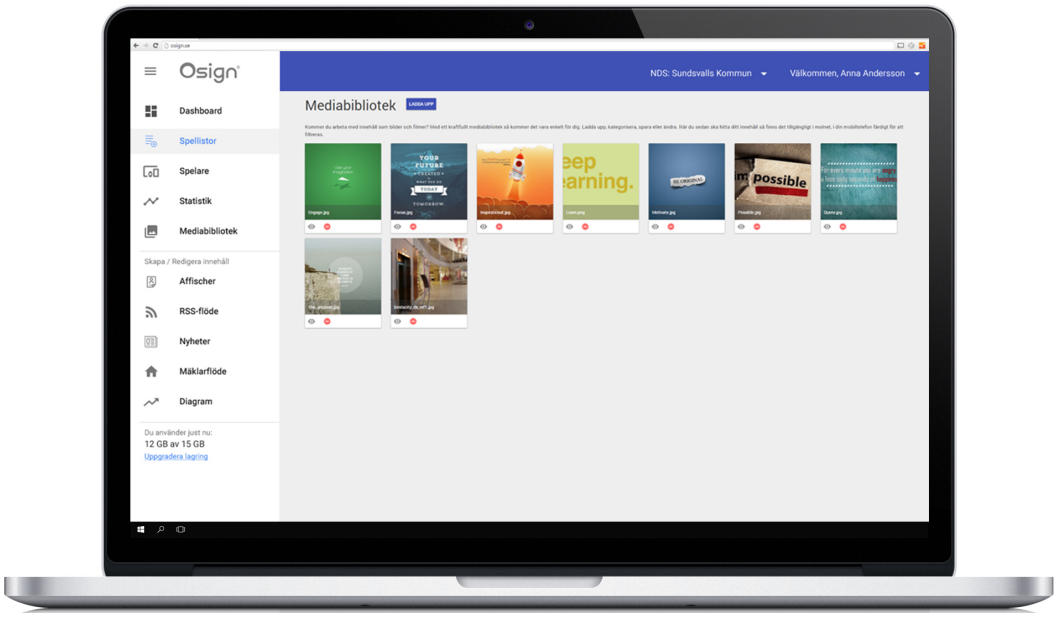
The whole system consisted of different services running in the cloud and all the signage devices using these services.

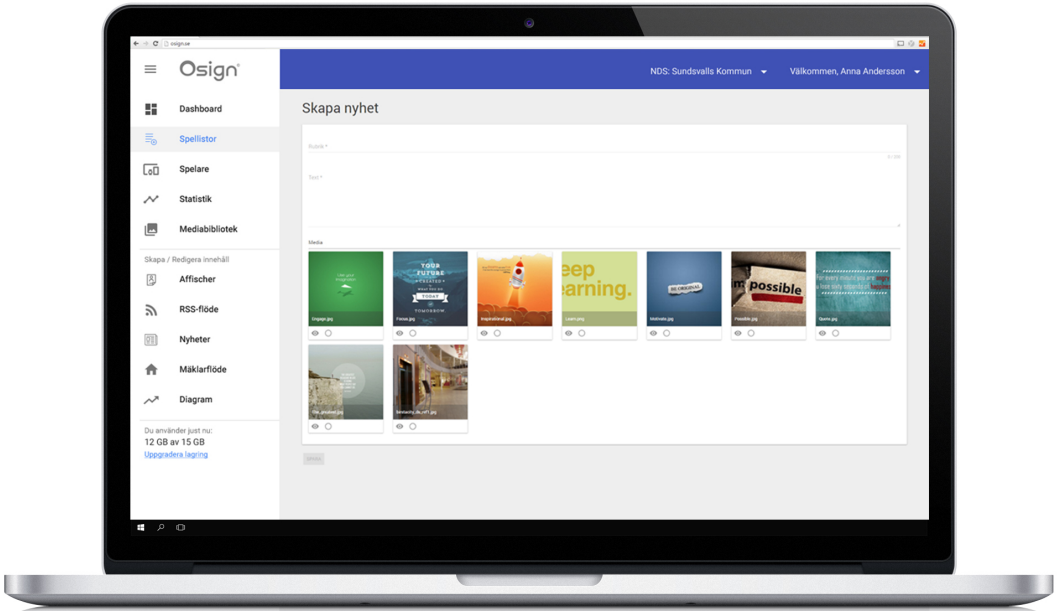
2.1 Installations

#HookYourLook
Shop
everything
DISPUTED
KING
TRAINERS

Södermalm-Centrum
Fantastisk sekelskifteslägenhet med balkong
ACCEPTERAT PRIS
1 600 000 kr
ADRESS
Storgatan 1
RUM
3 rok
BOAREA
82,5 kvm
AVGIFT
4.245 kr
TYP
BOSTADSRÄTT
6
SEP
Visning
KL: 17:00 - 18:00
Välkommen på visning
Ellen Ingman
Fastighetsmäklare
070-782 00 45
BJURI
Är du intresserad a
Kom in till oss

2.2 User interface





2.3 Signage devices

The devices used in this system were off-the-shelf Intel NUCs that was running a custom Debian Jessie distribution. Displaying content on these devices was done by running specialized HTML5-apps in a web browser like program.

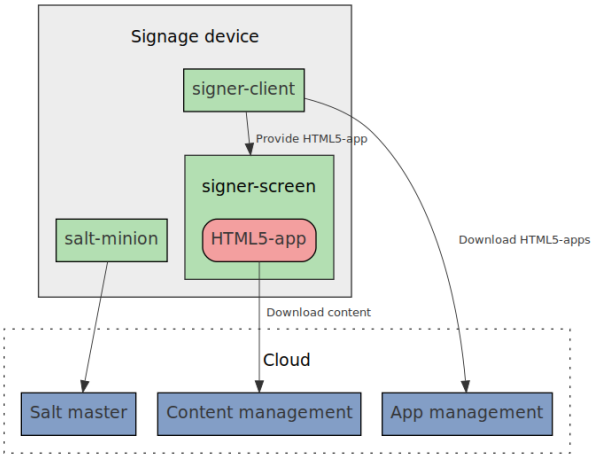


Figure 1: Device architecture

To do this the device had two services running locally:

- `signer-screen`
- `signer-client`

`signer-screen` was a program written in C++, based on Chromium Embedded Framework that would run the HTML5-apps in fullscreen. The HTML5-apps were responsible for downloading and displaying the actual content from a remote content management system. Content was retrieved over HTTPS from a REST API that the content management system provided.

`signer-client` was a program written in Haskell that had 3 main responsibilities:

- Download HTML5-apps from a remote service
- Supply `signer-screen` with the updated HTML5-apps
- Expose a web based admin interface on the local network

The HTML5-apps were retrieved over HTTPS from a REST API that the app distribution system provided. The apps were stored locally on disk and `signer-client` communicated with `signer-screen` locally over a UNIX socket (ZeroMQ request/response) to notify when a new app had been downloaded.

To allow technicians to access the web based admin interface on the local network, all devices used Avahi to broadcast their service across the local network. The technicians could access the devices using their serial number (that was written on the devices) as a `.local` domain name.

To configure and manage the devices SaltStack was used. Using Saltstack we could easily roll out updates across all devices, perform checks on which devices were online, if they had a screen connected, etc. We had about 250 devices in production that was managed by a single `salt-master` instance.

2.4 Content management system

The CMS was Django based and backed by PostgreSQL as persistent data store. At the beginning of the project the built-in admin interface was used, but we slowly moved away from that to use a SPA (single-page application) written in Javascript using the reactive framework Mithril.

The CMS therefor had 2 APIs:

- External API for the signage devices
- Internal API for the admin SPA

To reduce time writing boilerplate code Django REST framework was used in conjunction with Swagger. The APIs were first defined and documented in Swagger, then end-to-end tests were derived automatically from the Swagger file, and finally the APIs were implemented with DRF.

In the interface clients had different types of content they could create and manage:

- News articles
- Posters
- RSS feeds
- Product presentations

When creating content types the client could choose to use images and videos from their media library. For example, when creating a news article they could choose to use an image for the article, or even a video.

These content types were meant to be shown fullscreen, one at a time, like presentation slides. To control how that would happen, the client created playlists where each they could control order and how long each view/slide/page would be shown. These playlists could then be published on the devices.

The CMS had a hierarchy that allowed large companies to use the system. Each company could have one or many subsidiaries that had content separated from each other. Then users were given access to one or many subsidaries of the company, so that they could control which users were allow to change content of which subsidiary.

2.5 App management system

The app management system was written in Haskell and was based on the web framework Yesod, also backed by PostgreSQL as persistent data store. This system provided an admin interface and a REST API.

The admin interface was used by employees to handle apps and screen orientation of all the devices. The devices would then connect to the REST API over HTTPS at a set interval to retrieve the apps and the screen orientation.

2.6 Infrastructure

Amazon Web Services was used as cloud provider for all the services. At the beginning of the project a single AWS account was used and all resources were setup manually using the AWS web console. At the end of the project there were 2 separate AWS accounts (test and prod) under 1 AWS organisation that were setup automatically using Terraform.

Route53 Used to manage DNS

EC2 Used to run all services

RDS Used to host the PostgreSQL instances

S3 Used to store backups and logs

All EC2 instances were configured and managed by SaltStack.

2.6.1 Early architecture

At the beginning of the project all services used a monolithic design where the application, nginx and PostgreSQL was run on the same EC2 instance and the domain name pointed directly towards the Elastic IP of that instance.

The security group of the EC2 instances allowed HTTPS access publically and SSL termination was handled by nginx on each EC2 instance.

Deployment of the application changes was done on the fly using SaltStack, which worked fine 95% of the time. Since the domain name pointed directly to the EC2 instance that hosted the service application, the service would be down when that 5% deployment failure happened.

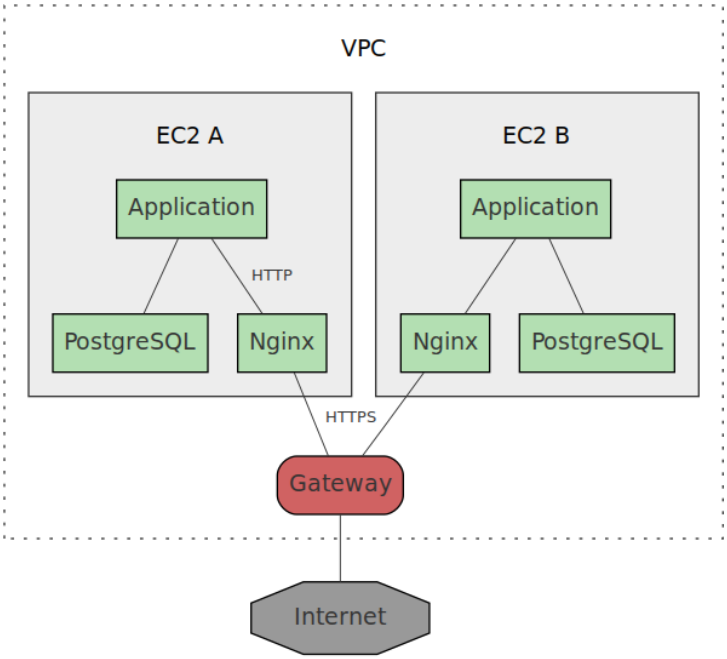


Figure 2: Early architecture

2.6.2 Late architecture

At the end of the project the design changed towards a more microservice oriented design, where the EC2 instance would only host the application, with a nginx EC2 instance in front and PostgreSQL hosted separately on RDS.

The security group of the EC2 application instances were changed to only allow HTTP traffic directly from the nginx EC2 instance. The security group for the nginx EC2 instance allowed HTTPS access publically and handled all SSL termination.

The deployment proceedure was changed to use a Blue/green deployment approach, which meant we could reduce the downtime dramatically when failures happened.

This approach also meant that we could easily increase the availability of the whole system by introducing more EC2 application instances, but also introduce more EC2 proxy instances and configure Route 53 to use them as failovers.

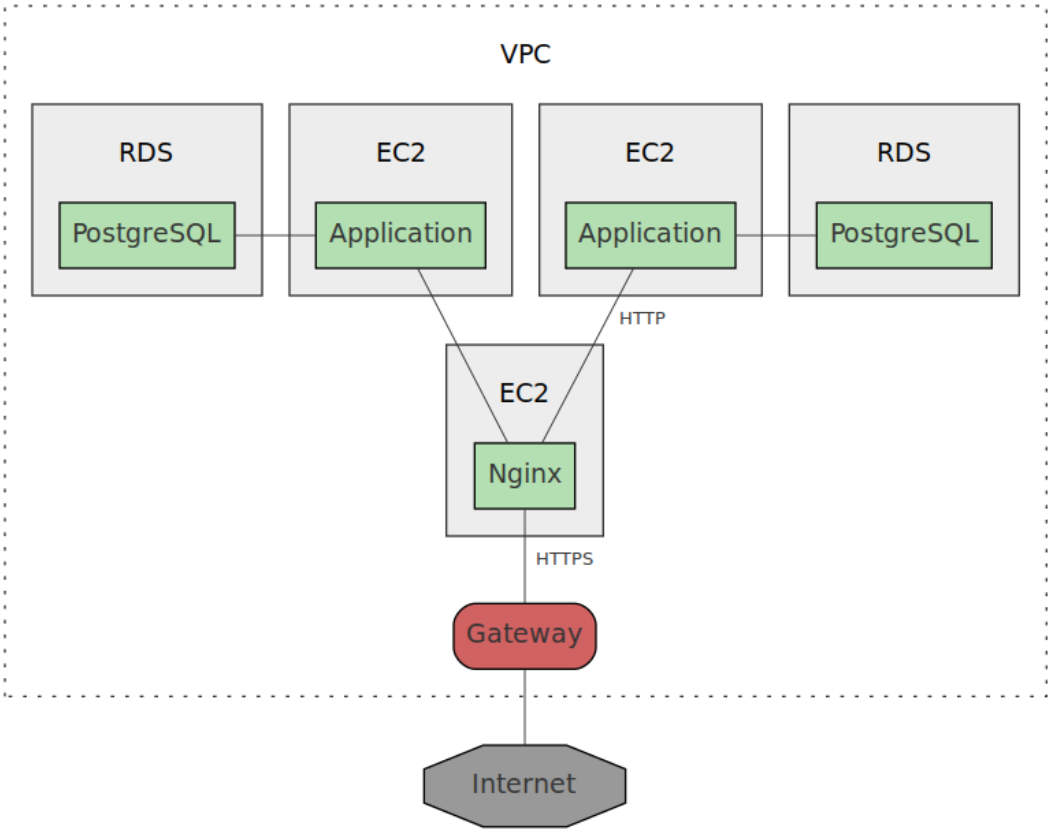


Figure 3: Late architecture

3 Showcase of medium project

During 2017 I worked as a solution architect and developer at Sweco Position. The work group I was a part of had developed a mobile application called PIANO. This application allowed users to do inventory management tied to geographical locations.

Figure 4: Example of geotechnical survey in the app

This application had many different use cases, for example:

- Geotechnical engineers used it to keep track of where ground surveys had been carried out
- Building engineers used to it keep track of roof inspections carried out on gas stations

The reason why this application could be used for so many different things was because it allowed the users to create their own inventory forms. So if the user wanted to keep inventory of products in a store, they could create a form that would contain relevant fields to products, such as product number, product name, price, etc. To store this data CouchDB was used as persistent data store.

Access to these CouchDB-instances were controlled by JSON Web Tokens. Users would authenticate themselves at the authentication service that would provide the app with a JWT that would be use when accessing the CouchDB-instances.

To separate data between different customers, multiple CouchDB-instances were used. A user could have access to *one or more* instances. The JWT contained this information as a list of CouchDB-instances. The way this was setup server side was that each CouchDB-instance was running in a Docker container with a single Nginx proxy infront of all of them. This was running on a single Azure VM.

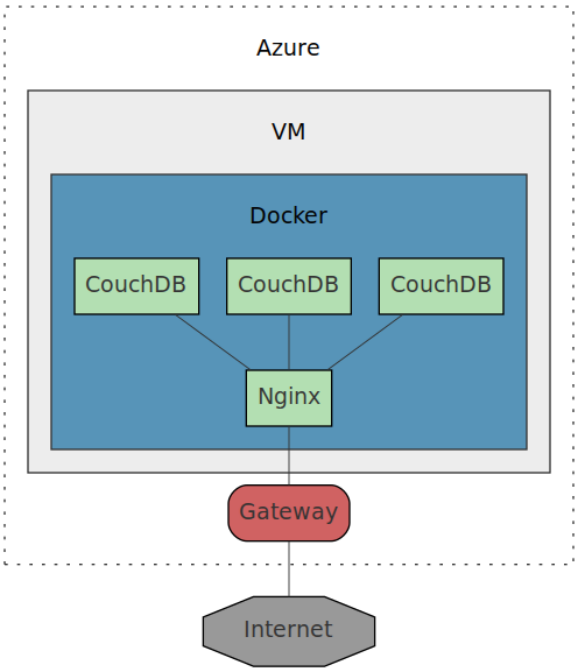


Figure 5: CouchDB and Nginx architecture

3.1 Problem

Each CouchDB-instance had a unique label that was using in the URL. So if you wanted to access CouchDB-instance **test** you'd use a path prefix of `/db/test/`.

The problem with this, was the way it was setup. To add a new CouchDB-instance you'd have to manually add a Nginx config file with the label and IP hard coded inside a `location` block that would catch the path prefix:

```
location /db/test {
  add_header 'Access-Control-Allow-Origin' "$http_origin";
  add_header 'Access-Control-Allow-Credentials' 'true';
  add_header 'Access-Control-Allow-Methods' 'GET, POST, PUT, DELETE, OPTI
  add_header 'Access-Control-Allow-Headers' 'Accept,Authorization,Cache-C

  access_by_lua '
    local jwt = require("nginx-jwt")
    jwt.auth({
      db=function (val) return jwt.table_contains(val, "test") end
    })
  ';

  proxy_pass http://172.12.20.1;
}
```

The proxy CouchDB proxy configuration in each block pointed to the actual IP of the Docker container running the CouchDB-instance. Which meant that if the VM was restarted you'd have to:

- Start all Docker containers
- Restore data from backups
- Check the IP of the containers
- Change each location configuration file with the right IP

At the time there were around 15 CouchDB-instances, which meant that it would take a while to do.

3.2 Solution

My role in this project was to develop new features for the application and improve how the infrastructure was operated. To improve operations I first looked at how I could remove the need to manually create location configurations.

Since the Nginx distribution used was OpenResty I knew that this could be handled better with Lua. At the time Lua was used to verify that the JWT token was correct and that it contained the right database label. Everywhere the database label was mentioned, it was hard coded.

Instead of a separate location block for each database, I created a single location block for all requests against `/db/...`. Inside that block I added the following Lua code:

```
local match, err = ngx.re.match(
    ngx.var.uri,
    "^/db/(?<instance>[a-zA-Z0-9_-]+)/(?<path>.*)",
    "o"
)

if match == nil and err then
    ngx.log(ngx.ERR, "failed to match path: ", err)
    ngx.exit(ngx.HTTP_INTERNAL_SERVER_ERROR)
    return
elseif match == nil then
    ngx.status = ngx.HTTP_NOT_FOUND
    ngx.say('{"error":"not_found","reason":"missing"}')
    return
end
```

This way I could check that the instance label was correct and extract the actual value. Then the JWT verification could be called with that value:

```
local jwt = require("nginx-jwt")
jwt.auth({
      db=function (val) return jwt.table_contains(val, match["instance"])
})
```

To solve the problem with using a hard coded IP address for the Docker containers and starting them manually I used docker-compose. Using docker-compose I could declaratively define each CouchDB-instance and the Nginx container in a single file, and take advantage of the fact that containers could resolve the IP of each other by their name.

So after declaring all CouchDB-instances in the docker-compose file and making sure Nginx used Dockers builtin resolver, I could build the URL in Lua dynamically:

```
local url_parts = {
   "http://",
   match["instance"],
   ":5984/",
   match["path"]
}

if not (ngx.var.args == nil) then
   table.insert(url_parts, "?")
   table.insert(url_parts, ngx.var.args)
end

ngx.var.proxy_url = table.concat(url_parts, "")
```

And then use that URL in the location block:

```
location /db {
  set $proxy_url "";

  access_by_lua_file '/nginx/conf/lua/handle_access.lua';

  add_header 'Access-Control-Allow-Origin' "$http_origin";
  add_header 'Access-Control-Allow-Credentials' 'true';
  add_header 'Access-Control-Allow-Methods' 'GET, POST, PUT, DELETE, OPTI
  add_header 'Access-Control-Allow-Headers' 'Accept,Authorization,Cache-C

  proxy_pass $proxy_url;
}
```

Another benefit of using docker-compose was that developers could easily setup the whole infrastructure locally easily, something which they couldn't do before.

My plan from the beginning was to setup a Kubernetes cluster to get a even more robust solution, however I wasn't given enough time, so I settled with this solution using docker-compose.