HOWEST

THESIS

Automated deployment and performance analysis of a white-label web application

Author:
Niek CANDAELE

Supervisor:
Thomas CLAUWAERT

April 25, 2021

"Funny or thought provoking quote goes here"

Someone, somewhere

HOWEST

Abstract

Bachelor of applied computer science

Automated deployment and performance analysis of a white-label web application

by Niek CANDAELE

A case study and practical implementation of a white-labeled web application. Starting with an existing application, proceeding with analysis of the current implementation and problems, investigating potential solutions and finally implementing them

Acknowledgements

Thanks to Thomas Clauwaert, Serge Morel, en de rest . . . :)

Contents

List of Figures

List of Tables

List of Abbreviations

AWS	**A**mazon **W**eb **S**ervices
CMS	**C**ontent **M**anagement **S**ystem
IaC	**I**nfrastructure **a**s **C**ode
LCP	**L**argest **C**ontentful **P**aint
FID	**F**irst **I**nput **D**elay
CLS	**C**umulative **L**ayout **S**hift
CDN	**C**ontent **D**istribution **N**etwork
CDK	**C**loud **D**evelopment **K**it

For/Dedicated to/To my…

Chapter 1

Introduction

1.1 Original state

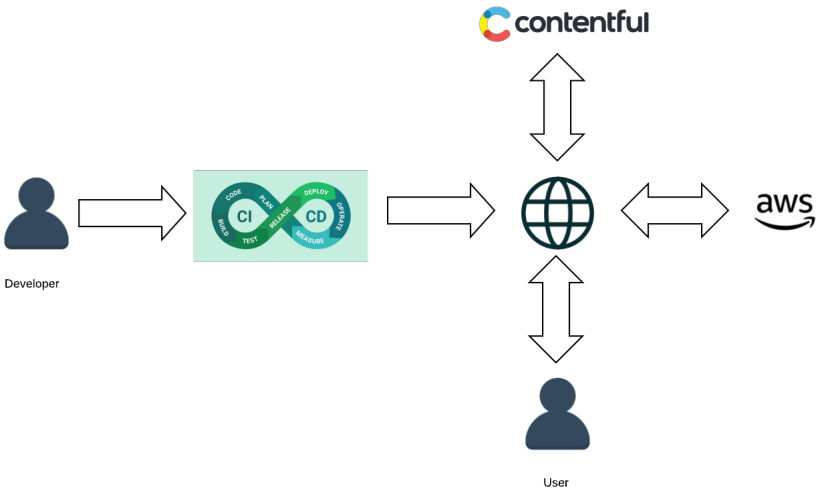

FIGURE 1.1: The original flow of the web application

The inital state of the process we will investigate is a fairly typical web application. Developers write code which gets built and deployed by a CI/CD pipeline. The result are static files which are served by a webserver.

Users who visit the site receive these static files, after which the application will perform API requests to a CMS which returns the necessary assets. The application will also send API requests to a backend (hosted on AWS) for completing business logic.

1.2 Stampix

Stampix is a startup based in Belgium. It's main clients are businesses, businesses can purchase 'prints' with Stampix which they can then use for marketing/loyalty campaigns. The exact purpose always depends on the business, a concrete example would be "Sign up for our newsletter and receive 5 free photos!". So the business is responsible for distributing their purchased prints to the users.

Users who have received these prints can then visit the web application to select photos, crop/rotate/resize/... as needed and ultimately send their selection to Stampix servers. At this point, the responsibility of the web application ends and the

backend processes take over. The photos are sent to printing partners for printing and distribution.

Stampix has many clients and since the primary use case is using the print for marketing purposes, the web application should be appropriately branded with the clients assets.

Chapter 2

Defining the problem

2.1 Performance

One of the main bottlenecks with the current approach is the requests to the CMS. When a user visits the site, they first have to load the basic static files from the server. This will only load a skeleton of the application though. Depending on what domain name the user is visiting (companyX.stampix.com or companyY.stampix.com), the application will send requests to the CMS to load appropriate images and texts.

This whole process results in a long time before the user can actually start using the application. We aim to create a solution that will make this significantly faster.

2.2 Modularity

Stampix has many clients and every client has individual needs. One client might want to block any NSFW pictures while another doesn't mind these types of pictures and instead wants different functionality. This means the web application must remain modular enough to support these different "add-ons". This problem can be solved by writing code that supports this and the exact implementation is outside the scope of this thesis. However, it is an important point and we must make sure the solution we create can support this.

2.3 Deployment

To support different assets for different clients, the current implementation loads the assets during runtime. If we move loading assets from runtime to build time with a static site generator, we will need to develop a methodology for deploying the application.

Chapter 3

Theory

3.1 Static site generators

Static site generators take your app and build it before serving to users. This means users receive plain HTML files. This moves a large computational burden from run time to build time which results in significantly faster load times for users. Furthermore, this approach allows for more aggressive and efficient caching.

3.1.1 Nextjs

Nextjs[1] is a React framework. Not explicitly a static site generator but has support for it. Nextjs has support for a ton of interesting features,

- Image optimization
- Hybrid static site generation and server side rendering
- Internationalization
- Typescript support

// TODO: Fill this out more once the PoC is created

3.1.2 Gatsby

Gatsby[2] is a static site generator at heart. It is one of the most popular frameworks around

- Static site generation leads to better performance
- Image optimization
- Large plugin ecosystem

3.2 Cypress

Cypress[3] is a tool made to run automated end-to-end tests in a browser.

We will use Cypress for it's simple API to control browsers programmatically. It will allow us to define the performance benchmarks as code and easily run that on the different PoC applications.

[1] https://nextjs.org/
[2] https://www.gatsbyjs.com/
[3] https://www.cypress.io/

3.3 Strapi

Strapi[4] is an open-source CMS. While Stampix uses a different CMS, we chose Strapi because it is easy to run locally. Because we are running the CMS locally, it is much easier to create a configuration that we can reuse across different deployments.

3.4 Docker

Docker[5] is a tool to run and manage containers. We will 'Dockerize' all of the components of our benchmarking pipeline: the PoC applications, Strapi and even Cypress. By using containers for this, we can create a pipeline that is platform independent. You, the reader, can run the benchmark on your own computer and you will see similar results we got in this paper.

3.5 Methodology

To make an educated decision on what framework provides the best performance, we will create a benchmarking pipeline. We will create small, proof of concept applications in the above mentioned frameworks which try to replicate real world behavior as close as possible.

Each PoC application must at least be able to load some assets from the CMS and perform an API request to the backend.

3.6 Metrics

Now that we have defined how we will create and run this performance benchmark, we can define some key metrics.

3.6.1 Largest contentful paint

> Largest Contentful Paint (LCP) is an important, user-centric metric for measuring perceived load speed because it marks the point in the page load timeline when the page's main content has likely loaded—a fast LCP helps reassure the user that the page is useful. Walton, n.d.

LCP is an important metric for us. In the original application, this would happen after the request to the CMS. That means LCP will be rather late for the average use. By using static site generators like we will we can make sure all of the content we want to show the user is served from a very fast CDN.

3.6.2 First input delay

https://web.dev/fid/

3.6.3 Cumulative layout shift

https://web.dev/cls/

[4] https://strapi.io/
[5] https://www.docker.com/

3.6.4 Bundle size

When a user visits the site, one of the main reasons they have to wait for a site to load is network connectivity. If a user has a faster network connection, they can download the static files of the website faster. This is a part of the system where Stampix has very little control, it is impossible to give the users a better network connection. However, what we can do, is make sure that the user has to download the smallest amount of data as possible. To achieve this, we can use a technique called minification to take our front end code and make it smaller.

Code written by developers might look something like:

```
function doSomething(parameterName) {
    const aVariableWithALongName = 'foo'
    return aVariableWithALongName + parameterName + 'bar'
}
```

After minification, the code will look something like:

```
function a(o){return"foo"+o+"bar"}
```

Over the entire codebase, shortening the code like this will result in a significantly smaller download size for the user.

Chapter 4

Performance results

4.1 Research results

// TODO: Here we will explain what results came out of our performance research

Chapter 5

Deployment

5.1 Technologies

There are a lot of IaC solutions out there. Ansible, Terraform, Pulumi, ... They all have their own positives and negatives. Choosing the right tool for the job is not easy

Some of the main factors when choosing were

- Integration with AWS
- Easily scriptable

Since Stampix currently runs all their infrastructure on AWS, it makes sense to choose a tool that has good integration with AWS. We will be deploying multiple static site builds onto AWS, this means there is some logic involved for each build. This is why the solution we choose should be easily scriptable: we would rather write the infrastructure as code and not as a declarative language (JSON or YML).

5.2 AWS CDK

> The AWS Cloud Development Kit (AWS CDK) is an open source software development framework to define your cloud application resources using familiar programming languages. AWS, n.d.

I chose AWS CDK because

- it is made by AWS, so the integration with AWS will be great
- it allows us to write infrastructure as javascript code

Being able to use Javascript is a big bonus since Javascript is the main language used at Stampix. Everyone in the team knows it well.

5.3 Configuration

As mentioned earlier, every static site needs a separate configuration. This configuration will determine what client the build is for, allowing us to fetch the right assets from the CMS. Furthermore, this configuration will also allow us to enable or disable certain modules.

In the proof of concept, this configuration is a static JSON document (see the file packages/infra/config.json). This JSON document does not have to be a static file

though. It could come from the CMS or even a custom application. The goal is to make sure that non-technical people can create deployments. This means that they should not be editing straight JSON files. Instead, the person creating the deployment should be guided through a form in order to create this configuration. To make this experience as smooth as possible, I recommend creating a custom application to create these configs. However, I consider this outside the scope of this paper.

5.4 Build and deployment flow

In order to deploy websites, we need to use a domain name. We will use AWS' managed DNS service, Route 53 [1]

The first thing that happens is the configuration JSON is retrieved and read. Based on the contents, several builds of the frontend application are started.

The built files get stored inside S3 [2]. Every campaign gets a separate bucket.

The script will also provision TLS certificates via AWS. Finally, a CloudFront [3] distribution is set up. This will make sure the static files get served when someone visits our sites.

[1] https://aws.amazon.com/route53/
[2] https://aws.amazon.com/s3/
[3] https://aws.amazon.com/cloudfront/

Appendix A

Performance reports

A.1 Initial performance

Lighthouse scores? Some trace info? Other performance indicators?

Bibliography

AWS (n.d.). *AWS CDK*. URL: https://aws.amazon.com/cdk/.
Walton, Philip (n.d.). *Web Vitals*. URL: https://web.dev/vitals/.