HOWEST

THESIS

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

Author:
Niek CANDAELE

Supervisor:
Thomas CLAUWAERT

March 21, 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

Abstract iii

Acknowledgements v

1 Intro 1
- 1.1 How it used to be . . . 1
- 1.2 Solutions . . . 1
- 1.3 Requirements . . . 1
  - 1.3.1 Infrastructure as code . . . 1
  - 1.3.2 Asset managment . . . 2
  - 1.3.3 Deployment configuration . . . 2
  - 1.3.4 Deployment stages . . . 2
  - 1.3.5 Modularity . . . 2
    - NSFW check . . . 3
    - Thank you module . . . 3

2 Research 5
- 2.1 Static site generators . . . 5
  - 2.1.1 Nextjs . . . 5
  - 2.1.2 Gatsby . . . 5
  - 2.1.3 Umi.js . . . 5
- 2.2 Deployment configuration . . . 5
  - 2.2.1 Store inside CMS . . . 5
  - 2.2.2 Storybook . . . 6
- 2.3 Automated testing . . . 6
  - 2.3.1 Selenium . . . 6
  - 2.3.2 Cypress . . . 6
  - 2.3.3 Protractor . . . 6
  - 2.3.4 Playwright . . . 6
- 2.4 Deployment . . . 6
  - 2.4.1 Ansible . . . 7
  - 2.4.2 Terraform . . . 7
  - 2.4.3 Serverless . . . 7
  - 2.4.4 AWS Cloud Development Kit . . . 7
  - 2.4.5 Cloudformation . . . 7

3 Performance 9
- 3.1 Approach . . . 9
- 3.2 Results . . . 9

A Performance reports 11
- A.1 Initial performance . . . 11

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

For/Dedicated to/To my…

Chapter 1

Intro

1.1 How it used to be

Stampix is a startup that prints photos. Stampix' customers are companies, these companies buy printcodes which they can then distribute to their users in context of marketing or loyalty campaigns. Every client gets their own branded web application.

This involves:

- Storing all brand-related content in a CMS (Contentful)
- Pulling in all that content during app runtime
- Deployments for new clients require a lot of manual configuration / dev work
- There's no automated tests, which can cause broken deployments if not careful

This had a few problems which I will explain in detail later ...

1.2 Solutions

Following are the methods used to improve this workflow. Each method will probably get it's own detailed chapter later?

- Using a static site generator to build web app and assets during build time
- Automated testing (Selenium-like / snapshots / unit)
- Deploying each built application to AWS

1.3 Requirements

1.3.1 Infrastructure as code

A big pain point right now is that it takes a lot of manual (development) work to create new deployments. We can solve this by automating the process, however it's not as simple as just building the frontend assets and uploading them to the cloud.

The operations and sales teams must be able to create deployments on their own and they must be able to control certain aspects of the final product.

1.3.2 Asset managment

Brand images, texts, ...

This is currently kept in a CMS. CMS' are made for this, which automatically gives us a lot of functionality.

An additional pain point here is that the structure of the data is still manually managed. Stampix has asked to see if there is some sort of IaC solution possible for this.

1.3.3 Deployment configuration

All the following is currently stored inside the CMS but define a configuration of the deployment.

- Domain name (company.stampix.com).
- Campaign ID (corresponds to an ID in the database)
- Demo mode (No real codes needed but nothing gets printed either)
- googleAnalyticsTrackingCode
- facebookPixelId

Ideally, we can keep track of these variables separately from the regular assets. While the distinction is hard to make right now, some of these values are needed outside the statically generated app. For example, we'll need to domain name to set up DNS.

This should be a (semi) structured document which can be easily edited by non-development teams and finally sent to our IaC solution to create deployments.

1.3.4 Deployment stages

There should be a clear distinction between deployments in production, staging or test. A common occurence right now is that the sales team will create demo's to use during their pitches. This has the risk that they change some configuration which breaks production apps. These demo's also get made, pitched and then promptly forgotten, never to be used again.

- Production: deployments that are live, in actual use.
- Staging: deployments before going to production. Final checks happen here
- Test: sales demos, development builds, ... anything else

1.3.5 Modularity

The web app must support 'plugins'. These plugins can literally be anything. They can include extra logic in the backend, extra pages in the frontend or a combination of both.

TODO: How will this be configured? Could be a multi-select in the deployment configuration

NSFW check

This plugin checks every order for photos that has **N**ot **S**afe **F**or **W**ork content. If an order includes content like this, it gets rejected.

Thank you module

After creating an order, the user is presented with a small form that asks if they would like to say thank you. The user can select an emoji to reflect their feelings. This module is particularly useful for creating metrics at the end of a campaign

Chapter 2

Research

2.1 Static site generators

Static site generators take your app and build in before serving to users. This means users receive plain HTML files. This moves a large computation 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.

2.1.1 Nextjs

Nextjs[1] is a React framework. Not explicitly a static site generator but has support for it

2.1.2 Gatsby

Gatsby[2] is a static site generator at heart. It might be harder to do runtime stuff with Gatsby

2.1.3 Umi.js

2.2 Deployment configuration

2.2.1 Store inside CMS

This is a low-effort solution. The idea is to create a deployment schema config with well-defined properties. We use this schema as a content model inside the CMS.

Positive:

- Non-development teams get a nice interface to change things while still using a controlled schema
- No need to create a custom frontend for this
- Very flexible, easy to add or change properties

Negative:

- Very flexible, easy to create broken configurations if we do not properly validate.
- The CMS becomes an even more important part of the technology stack. I consider this a negative because currently Stampix uses a proprietary, closed source service.

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

2.2.2 Storybook

Stampix is using Storybook[3] to create a component library for the mobile application. With Storybook, the operations team can change some parameters and then export a configuration. This is nice because Storybook provides a sort of sandbox to play around and design a deployment, completely separated from any cloud resources.

Problem with this is that Storybook focuses mostly on styling. We need more than just styling.

2.3 Automated testing

We create lots of different webapps, a different one for each client. Currently, we have no automated tests. This should change

Automated testing gives developers confidence their changes did not break anything. It can spot bugs before the code is even released

A lot of the following tools do very similar things, they emulate or run a browser and we can instruct it to do certain actions on our site after which we can assert if the application is behaving correctly.

// TODO: Since most of these tools are/do basically the same thing, we should def focus on the differences between them.

2.3.1 Selenium

Selenium[4] is an established project

2.3.2 Cypress

Cypress[5] is pretty new and 'cool'

2.3.3 Protractor

2.3.4 Playwright

Playwright[6] is

2.4 Deployment

We will be deploying lots of apps for different customers. It's important that we can do this as rapidly and effortlessly as possible. This means we should adopt infrastructure as code as much as possible. There's a number of tools available, I will talk about some of the big players.

[3] https://storybook.js.org/
[4] https://www.selenium.dev/
[5] https://www.cypress.io/
[6] https://playwright.dev/

2.4.1 Ansible

Ansible[7] is an agentless automation tool. Ansible connects via SSH to a number of hosts you specify and executes tasks on those hosts. It offers a range of powerful tools and a strong ecosystem. Ansible focuses mostly on configuration management.

While there are a few community and official AWS-related plugins available, I don't think Ansible is a good tool for this project. Ansible is built around SSH connections and our system will have (practically) no SSH-capable machines.

2.4.2 Terraform

Terraform [8] is a infrastructure as code tool. In contrast to Ansible, Terraform focuses mostly on infrastructure tasks like creating virtual machines or allocating cloud resources.

2.4.3 Serverless

2.4.4 AWS Cloud Development Kit

2.4.5 Cloudformation

[7] https://www.ansible.com/
[8] https://www.terraform.io/

Chapter 3

Performance

3.1 Approach

Based on the information we gathered in the previous chapter, we will create several proof of concept apps with these technologies. We will also create a standardized performance benchmark using Cypress. Cypress can run the tests on multiple browsers.

The proof of concept apps must replicate the real behavior as close as possible.

- Load images from the CMS
- Perform API requests to a backend
- ... ? // TODO: what else does it need to do? Dont go too far with the PoC apps tho

The tests will be ran using Docker containers. // TODO: explain how it works :) // There's a bunch of moving parts (CMS with database + benchmark + each PoC app)

To accurately gauge performance of each app, we will take into account several factors.

// TODO: Should formalize these and explain what each metric means?

- Web vitals https://web.dev/vitals/
- Build time (aka how long does 'npm run build' take?)
- Lighthouse scores https://web.dev/performance-scoring/
- //TODO: Any more?

We should also try to run these tests locally vs in a deployed environment. Network latency (or other factors) may play a big role!

3.2 Results

// TODO: Tables with a bunch of numbers go in the appendix, // What visualizations can we create to put in here?

// TODO: Which is fastest? Probably have to weigh different factors together. I doubt one technology will be the clear winner here

// TODO: We should also explain why metrics differ per technology. For example, I expect static site generators to have a better Largest Contentful Paint than a app that loads CMS assets during runtime

Appendix A

Performance reports

A.1 Initial performance

Lighthouse scores? Some trace info? Other performance indicators?