



Integration of server side rendering framework with content management system

Jaber Askari

Bachelor's thesis

May 2023

Information and Communication Technologies (ICT)

Software engineering

Askari, Jaber

Integration of server-side rendering framework with content management system

Jyväskylä: JAMK University of Applied Sciences, May 2023, 37 pages

Information and Communication Technologies (ICT). Bachelor's thesis

Permission for open access publication: Yes

Language of publication: English

Abstract

As modern businesses are trying to be more accessible and easily findable through the internet, search engines are becoming more important. Therefore, websites are going toward becoming more and more optimized based on search engine criteria. This has caused the industry to shift towards Search Engine Optimization (SEO). SEO has become a crucial part of businesses. therefore, the aim was to address this issue by providing an efficient architecture that offers a highly optimized website for search engines. To achieve this goal some technologies that offer the best SEO has been researched. Gatsby framework offers server-side rendering (SSR) which is great for SEO. Gatsby is also used generally with a more static site which again static contents have the best SEO. To provide this static content to the Gatsby a headless Content Management System was required (CMS). Contentful is a headless CMS that was researched. To present the final solution the integration of these 2 technologies was crucial, Gatsby as a SSR framework and Contentful as a CMS.

The result was Viinimaa's architecture (viinimaa.fi) which uses the researched technologies to create a fast, responsive, and highly optimized website for search engines. This architecture can be used as a base template for similar sites integrating Gatsby and Contentful together.

Keywords/tags (subjects)

Server-side rendering, Gatsby, Next js, content management systems, React, GraphQL, Contentful

Miscellaneous (Confidential information)

Askari, Jaber

Palvelinpuolen renderöintikehyksen integrointi sisällönhallintajärjestelmään

Jyväskylä: Jyväskylän ammattikorkeakoulu. Toukokuu 2023, 37 sivua

Tieto- ja viestintätekniikan tutkinto-ohjelma. Opinnäytetyö AMK.

Verkkojulkaisulupa myönnetty: Kyllä

Julkaisun kieli: englanti

Tiivistelmä

Hakukoneista on tulossa yhä tärkeämpiä moderneille yrityksille, jotka haluavat olla helpommin saatavilla ja löydettävissä internetissä. Siksi verkkosivustoja optimoidaan yhä enemmän hakukoneiden kriteerien perusteella. Tämä on saanut alan siirtymään kohti hakukoneoptimointia (Search Engine Optimization, SEO). SEO:sta on tullut erittäin tärkeä yrityksille. Siksi tavoitteena oli vastata tarpeeseen tarjoamalla tehokas arkkitehtuuri, joka tarjoaa erittäin optimoidun verkkosivuston hakukoneille.

Tämän tavoitteen saavuttamiseksi on tutkittu joitakin tekniikoita, jotka tarjoavat parhaan hakukoneoptimoinnin. Gatsby tarjoaa palvelinpuolen renderöinnin (server-side rendering, SSR), joka sopii erinomaisesti hakukoneoptimointiin. Gatsbyä käytetään myös yleisesti staattisemman sivuston kanssa, joka on paras hakukoneoptimoinnin kannalta. Tämän staattisen sisällön tarjoamiseksi Gatsbylle vaadittiin sisällönhallintajärjestelmä (Content Management System, CMS). Contentful on headless sisällönhallintajärjestelmä, jota tutkittiin. Lopulliseen ratkaisuun näiden kahden teknologian integrointi oli ratkaisevan tärkeää, Gatsby SSR-kehyksenä ja Contentful CMS-järjestelmänä.

Tuloksena syntyi Viinimaan arkkitehtuuri (viinimaa.fi), joka luo tutkituilla teknologioilla nopean, responsiivisen ja hakukoneoptimoidun sivuston. Tätä arkkitehtuuria voidaan käyttää mallina samankaltaisille sivustoille, joissa käytetään Gatsbya ja Contentfulia.

Avainsanat (asiasanat)

Palvelinpuolen renderöinti, SSR, Gatsby, Next.js, sisällönhallintajärjestelmä, React, GraphQL, Contentful

Muut tiedot (Salassa pidettävät liitteet)

Contents

1	Introduction	3
1.1	Overview	3
1.2	Company's background.....	3
1.3	Objectives of the thesis.....	4
2	Server Side Rendering (SSR)	4
2.1	What is SSR.....	4
2.2	Why SSR?.....	6
2.3	Server Side Rendering vs. Static Site Generation.....	7
2.4	Server Side Rendering vs. Client-Side Rendering.....	8
3	Server Side Rendering frameworks.....	8
3.1	Next.js.....	8
3.1.1	React	8
3.2	Gatsby.....	11
3.2.1	How does Gatsby work?	11
3.2.2	GraphQL in Gatsby	13
3.2.3	GraphiQL tool.....	14
3.2.4	Page queries.....	15
3.2.5	Static queries	17
3.2.6	Page creation in Gatsby	18
3.2.7	Plugins.....	21
4	Content management systems (CMS)	23
4.1	What is a CMS?.....	23
4.2	Traditional CMS.....	23
4.3	Headless CMS.....	24
4.4	Contentful	25
4.4.1	Contentful content APIs.....	26
4.5	WordPress.....	27
5	Project.....	31
5.1	Overview	31
5.2	Technologies	31
5.3	Architecture.....	32
5.3.1	Gatsby Cloud	33
5.3.2	Contentful and Cloudinary.....	33

5.3.3	Adobe Commerce	34
5.4	Integration.....	34
5.4.1	Gatsby and Contentful integration	34
5.4.2	Gatsby Cloud and Contentful.....	34
5.4.3	Gatsby and Adobe Commerce integration	36
5.5	Results	37
6	Conclusion	37
	References	39

Figures

Figure 1.	Server-Side Rendering (Verma 2021).....	6
Figure 2:	Most popular technology skills in the JavaScript tech stack worldwide in 2021 (Mts.)	9
Figure 3:	Gatsby life cycle	13
Figure 4:	Querying with GraphQL (Attardi 2020, Chapter 2: Gatsby Crash Course)	14
Figure 5:	GraphQL tool	15
Figure 6:	Page component and page query example	16
Figure 7:	Page component output in browser	17
Figure 8:	Static query example	18
Figure 9:	Creating pages dynamically	20
Figure 10:	Example of Gatsby plugins	22
Figure 11:	Traditional CMS (Attardi 2020, Chapter 1: Introduction to Netlify CMS).....	24
Figure 12:	Headless CMS (Attardi 2020, Chapter 1: Introduction to Netlify CMS).....	25
Figure 13:	Page structure in WordPress (Pearce 2011, 226-227).....	29
Figure 14:	Post structure in WordPress (Pearce 2011, 226-227.).....	30
Figure 15:	Viinimaa's architecture	32
Figure 16:	Gatsby cloud site settings	35
Figure 17:	Contentful webhook	36

1 Introduction

1.1 Overview

As in today's business every company is trying to be more accessible and easily findable through internet, search engines are playing a great role. Therefore, websites are going toward becoming more and more optimized based on search engines criteria. This has caused the industry to shift into Search Engine Optimization (SEO) a lot more in order to reach more customer and users through internet and search engines. SEO has become a crucial part of businesses and this thesis aimed to address this necessity with providing the fundamental technical knowledge of the related subjects and presenting an efficient architecture that offers a modern, fast, responsive and highly optimized website for search engines. This architecture has been used in *viinimaa.fi* which is a sub brand of Anora group. Viinimaa is mostly a static website with over 2500 pages. It is massive website that its performance, user experience, modernity and SEO are vital for the owners.

1.2 Company's background

Solteq is a Finnish information technology company founded in 1982 that works in various fields. Among many other products and services Solteq offers eCommerce, ERP solutions, product information management, digital marketing, store solutions and application development. Solteq is a growing company with around 676 employees and revenue of 68.4 MEUR in 2022. It has grown to become an international company that works in 6 different countries, Finland, Sweden, Norway, Poland, Denmark, and UK with 14 offices. (Company N.D.)

The project example that has been used in this thesis is a real-life website from Solteq's biggest customers, Anora group. "Anora is a leading wine and spirits brand house in the Nordic region and a global industry forerunner in sustainability" (About us N.D). Anora group has used Gatsby framework (a server-side rendering framework wrapped around React) and Contentful content management systems for their 2 websites; *viinimaa.fi* for Finland market and *folkofolk.se* for Sweden market.

1.3 Objectives of the thesis

The goal of this thesis was to research some of server-side rendering frameworks and their advantages and disadvantages. Also, a description about some of content management systems and their features and differences has been held. After the initial research, a real-life project example using the researched technologies has been presented. In this project the focus was on 2 main parts: architecture and integration. In the project's architecture part an overall perspective of the project, technologies that has been used, their usage and relations, data flow, content management, and the project's deployment has been drawn and explained. In the integration part, the connection, relation, practices of connecting all the technologies together, and how to create an efficient infrastructure by integrating all those technologies has been explained. The outcome of this thesis was a research-based project's architecture for creating a static website using Contentful content management system and Gatsby framework for small to enterprise organizations.

In his thesis it has been tried to answer to the following questions:

1. What is server side rendering and its frameworks?
2. Why is server side rendering needed?
3. What is content management systems?

2 Server Side Rendering (SSR)

2.1 What is SSR

In recent years the Server Side Rendering (SSR) technology has been gaining more attention and trending. It is very important for an IT specialist to be up to date and familiar with the new technologies and trends in the industry. SSR is one of the technologies that eventually an IT specialist will encounter with.

To comprehend SSR frameworks better, it is necessary here to clarify exactly what is meant by SSR and why it is needed.

In the beginning of the internet servers were responsible for serving the static web pages to the clients. The web pages were not intractable or dynamic at all and there was no possibility to change a content or a text but to transit from one page to another. Then some languages such as PHP, Java, Python and Ruby came along that brought the possibility to deliver HTML to client by using the templates. This method was called server-generated pages. (Konshin 2018, 13-15.)

When JavaScript came into play it brought the possibility to create dynamic web pages that clients could interact with. In this case, the servers were responsible only for sending the data to the client, in our case browsers, and it was not involved in templating the data. This separation of client side and server side has increased by the shift towards REST APIs. The industry moved away from server-generated methods to completely client side rendered HTML templates. (Konshin 2018, 13-15.)

To decrease the loading time of a JavaScript page's data in client side there was a possibility to take advantage of the server-generated method to match our need. Therefore, the SSR development method came to play. The term Server-side Rendering is used here to refer to a development method in web-based applications that allows to load and render the initial data, serialize the outcome data and HTML on the servers and then send it as an HTML string baked with data and elements to the client side. On the client side then JavaScript can take over and final elements will be rendered to users. This approach can optimize the rendering time by caching the elements, components, or entire pages in the server and sending them to the client. This is a very efficient way when a page needs to be rendered multiple times in the same state requested by a single or even multiple users. This strategy has the potential to decrease the load on serves due to caching possibilities. (Konshin 2018, 13-15.)

In practice it means that when a page is requested by a client, the server will bake the page with initial data and HTML, then cache it on its local storage. Later when the same or other client clients request the same page in the same state, there is no need to recalculate the page but to send the already cached data of the requested page to the client.

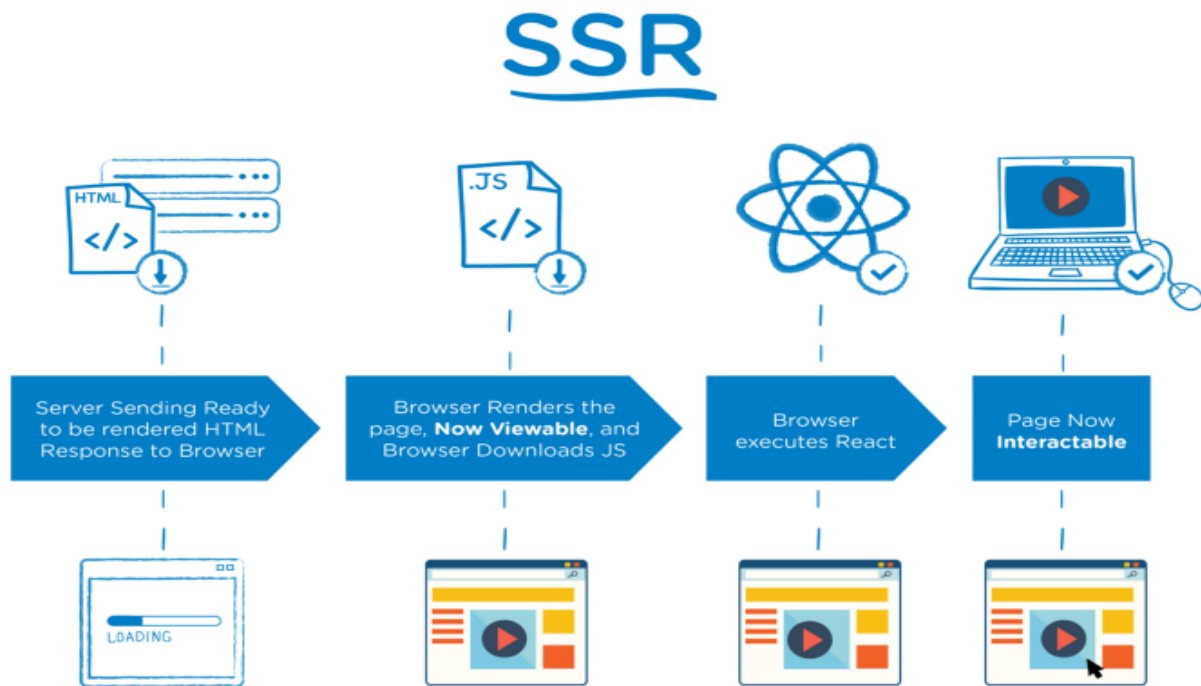


Figure 1. Server-Side Rendering (Verma 2021)

When creating the cache data of a page or component in server, it is critical to give it a time-to-live (TTL) (Mts.). "Time-to-live (TTL) is a value for the period of time that a packet, or data, should exist on a computer or network before being discarded." (Zola & Burke 2021)

It is important to give the cache data a TTL because if the state of the page had changed and the cached data of the page has not been expired, the server will send an invalid page from cached data with the old state to the client. Hence it is critical to regenerate the page's or component's cache data when its state changes.

2.2 Why SSR?

As explained briefly in previous chapter, primarily the benefit of SSR is its fast time of loading and rendering the content of websites for the clients. This advantage of SSR is more visible for slow internet and connection, solo devices, and content heavy pages. Since the content will be rendered in server the client device does not need to have a fast internet connection and much of processing power to download and process all the data and then to render them for the user. Also,

the server-side has much wider bandwidth, faster connection to database and better processing power than the client side. Since the initial data fetching is done in server-side, the rendering process will take much less time than it would have taken for the client. This will overall improve the user experience, accessibility, and web vital metrics. (Server-Side Rendering N.D.)

The other benefit of the SSR is better Search Engine Optimization (SEO) that the search engine crawlers can find and read the fully rendered pages. This will improve the quality and quantity of the website's traffic from a search engine drastically. (Server-Side Rendering N.D.)

It is important to understand the disadvantages of SSR. When this technology is being used it is critical to realize that it is not only about positive points and gains but there are some trade-offs to consider. (Server-Side Rendering N.D.)

- There are limitations in development process. For example, the Browser-specific code can only run in a specific life cycle of the page; external libraries will require unique modification and development to be able to use them in a server rendered page. (Server-Side Rendering N.D.)
- Unlike a static website that can be deployed in a file in any sever and will be functional, a server side rendered page needs a sever that can run Node.js. (Server-Side Rendering N.D.)
- SSR websites will add more load to the server. Specially if it is a high traffic website, the server must perform many CPU-intensive calculation for rendering a whole app in Node.js. So it is important to implement caching technologies to balance out the server load. (Server-Side Rendering N.D.)

2.3 Server Side Rendering vs. Static Site Generation

To better understand the meaning of the SSR and its advantages and to avoid misunderstanding the SSR with Static Site Generation (SSG) a comparison of both SSR and SSG is needed. Static is the key word in SSG method and in practice it means that whole page is static and does not change its state after build. This method is also called pre-rendering. This is another popular method for creating fast website. If the pages requested by all clients are the same, the server will only render the page once during build process and responds to all the requests with that same page. In pre-rendering practice, the server creates the pages as static HTML string or file and usually saves it as cache data. (Server-Side Rendering N.D.)

SSG provides great performance and time-to-content similar to SSR. Also, it is cheaper and has simpler deployment process. All the static HTML pages are generated during build time, and they will be provided to the client as a static website. If the data of the site changes a new build process triggers and the new static files with the updated data will be generated. (Server-Side Rendering N.D.)

2.4 Server Side Rendering vs. Client-Side Rendering

The main difference between Server-Side Rendering versus Client-Side Rendering (CSR) is they how are being offered to the users. In CSR the user's browser is responsible for downloading and loading the website's JavaScript and then executes the React and the server is only response to the request of the browser/client with requested data.

In the following, the most popular server-side rendering frameworks has been explained.

3 Server Side Rendering frameworks

3.1 Next.js

In short Next.js is a React framework that provides SSR. To clarify this statement, we need to know what React is and what a framework means.

A framework in information technology means a foundation that a software or website can be built on. It is usually associated with a programming language and offers different features and functionalities such as routing, data fetching, integration, and many others to facilitate and accelerate the development time of a software. A framework removes the necessity of building a software from scratch and gives a reliable structure to build on. It generally offers a more secure code base, better testing, and scalable software among other features. (What is Framework N.D.)

3.1.1 React

In the recent years React has gained a good reputation and trend in IT industry. It has been the most popular technology skill in JavaScript stack in 2021, as it has more than of 26% of market

share followed by Typescript and ES6. (Most popular technology skills in the JavaScript tech stack worldwide in 2021 N.D.)

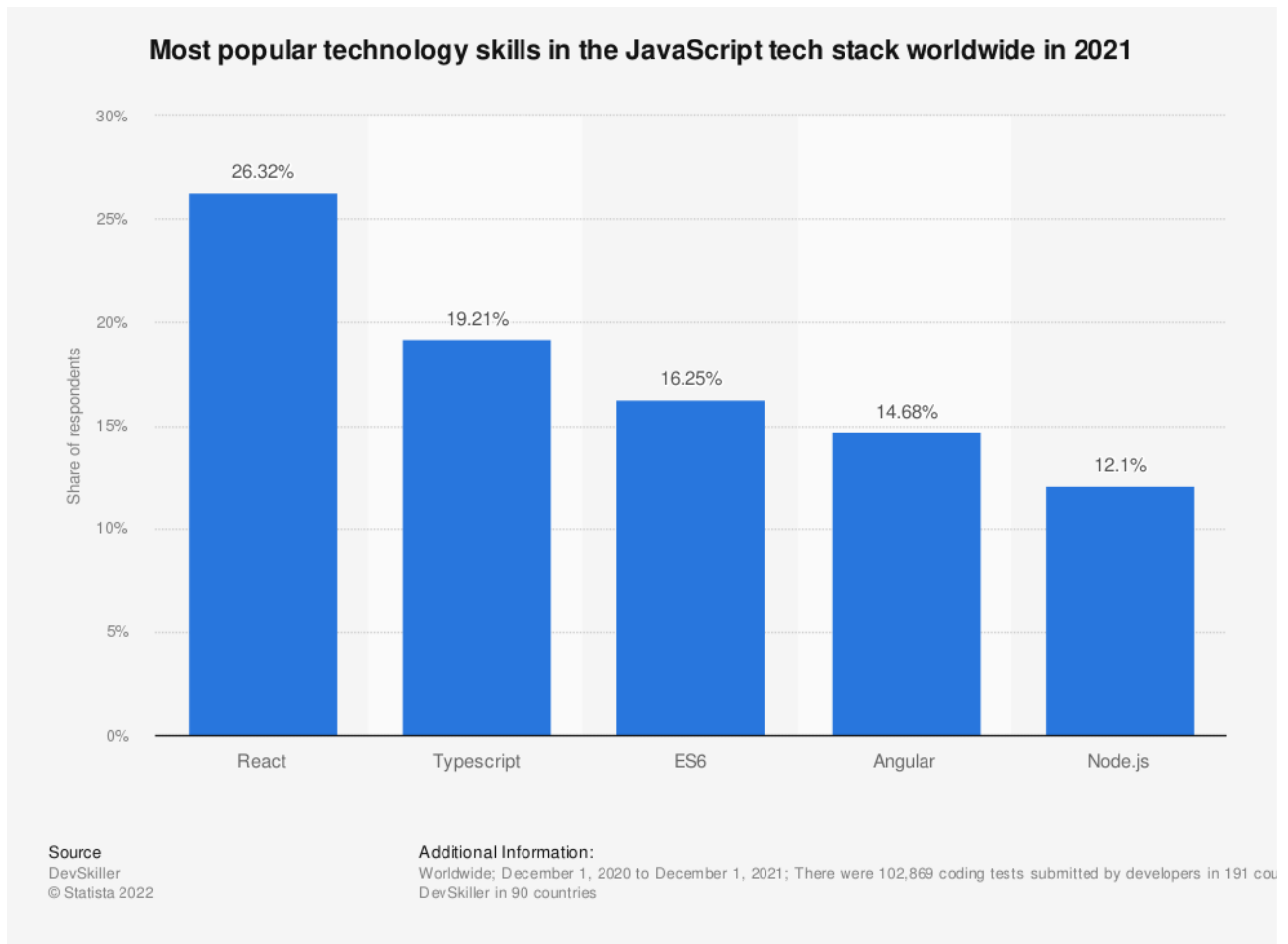


Figure 2: Most popular technology skills in the JavaScript tech stack worldwide in 2021 (Mts.)

React is JavaScript library developed by Facebook for building user interfaces. React is not a framework like Angular or Gatsby that provides ready-made solutions and functionalities. It is a library that gives more possibility and flexibility for development but also it requires more responsibility and more work unlike a framework. React uses virtual DOM unlike other libraries and frameworks, that will be batched into the actual DOM. It typically creates a Single Page Application (SPA). It means that the site is just one page which its contents are dynamic, and changes based on user input.

The other very important aspect of React is that it uses JSX. Simply put, JSX is a combination of HTML and JavaScript together in one file, which gives React a very simple and flexible developing

experience. For creating the virtual DOM React provides a high-level API, but using this API is a time consuming and hideous process. For this reason, an intermediate format such as JSX is being used for creating complex DOM structures. JSX is one of the most popular formats that Facebook uses in React. (Vepsäläinen 2016, 32-35.)

By using Node, React can render on server too. For this reason, most popular SSR frameworks such as Gatsby and Next.js are built around it. Also, it can be used to create native applications for mobile devices using React Native which targets mobile devices rather than the browser.

Next.js is currently the most popular React-based SSR framework because of its dynamic characteristics and its flexibility. Generally, we can divide the SSR frameworks into 2 main categories.

1. Dynamic solutions, such as Electrone, Next.js and After
2. Static solutions such as React Static, Gatsby

The dynamic solution such as Next.js allows the developers to create a more dynamic software and add any dynamic logic that generates the HTML based on client's input and request. These characteristics of the Next.js makes it more flexible and desirable as most the web-based applications are dynamic and Next.js offers a great out of the box solution for this type of applications. But as any other solution it has some weaknesses. One of its biggest drawbacks is that it requires a live server that generates the HTML based on each client's request. This server is better to be a cluster of servers to reduce the workload and redundancy. Also, they should be highly available and monitored consciously for any defects and issues that might appear. Because Next.js is a dynamic solution it will heavily rely on the servers. (Koshin 2018, 15-19)

The static solution such as Gatsby is another SSR framework. The key point here is static that the HTML are static and usually do not change greatly after the first build on the server. This type of static solutions can be served on the services such as Nginx, Apache which are examples of static servers. In this technology the HTML will be pre-backed with initial state on those servers (Nginx, Apache) and then can be served to clients. (Koshin 2018, 15-19)

This subject is discussed more under the Gatsby title which was one of the main focal points of this thesis.

3.2 Gatsby

Same as Next.js, Gatsby is a SSR framework based on React. The Only difference is that Gatsby is mostly a static site generator, but unlike the word *static* suggests, Gatsby can also have dynamic features and functionalities on the client side such as interactive component, data fetching and calling APIs. As it was discussed in React section, React creates a Single Page Application (SPA) but Gatsby creates multiple pages with the help of page templates. Each page template is similar to SPA that contains React components and can be used to create multiple pages with the same layout by looping the data into the template. (Attardi 2020, Chapter 2: Gatsby Crash Course)

Gatsby has its own automatic URL pathing determined by the directory and file structure of the project. It will create pages with the file name without the `.js` extension if the files are located in the `src/pages` directory. For example, a file with located in the `src/pages/home.js` will create URL as `/home`. It is not mandatory to use the Gatsby's automatic URL creation. Gatsby gives the possibility to create custom page URLs by defining them in `gatsby-node.js` file, which is discussed more in the next chapters. (Attardi 2020, Chapter 2: Gatsby Crash Course)

3.2.1 How does Gatsby work?

Gatsby has four main methodologies in its usage.

1. React: for creating the user interface and page templates.
2. Data source: it can be markdown file or any content management systems like Contentful and WordPress.
3. GraphQL: for querying data and feeding them to the React components.
4. Server-side rendering: During the build process that renders and bakes the pages on server and then send them to the client as ready to be shown HTML

When building a Gatsby site, the first thing that Gatsby does is fetching all the required data with the help of plugins or its APIs from data sources such as any CMSs, markdown document

and databases. Next it combines the fetched data into GraphQL Schema and creates a snapshot of all the data that is needed in its local storage. This will create a data layer with the exact shape of the data that has been asked for. This feature removes the necessity to request and fetch the data over the network because it is already generated during build time as a GraphQL schema. This locally generated data schema can be accessed with GraphQL queries and be fed into the page templates and React components.

The next step is building and compiling the data, assets, styles and React components together to generate the HTML pages. To generate the final HTML Gatsby uses a Node.js process in the background to bake and compile everything together to be ready for rendering in browsers. This process is called SSR as it was explained in the previous chapters. In this stage Gatsby create the entire site at once, therefore when the site is deployed, it does not need any servers to handle the data or create the pages because the pages have already been generated and compiled by Gatsby as static HTML that can be served to client as it is. (Overview of the Gatsby Build Process N.D.)

After the build process the generated pages can be deployed on the web servers for customer to use. As a Gatsby site is a full React application, everything that is possible in React is possible in a Gatsby site too (Overview of the Gatsby Build Process N.D). For example, creating dynamic components and fetching data in real time on the browser can be done in Gatsby sites as well. But it should be kept in mind that one of the biggest features of the Gatsby is its static characteristics that makes it very fast. Therefore, using too many dynamic components and fetching data extensively on the client the static aspect of a Gatsby site will get undermined, and it will lose its performance advantages.

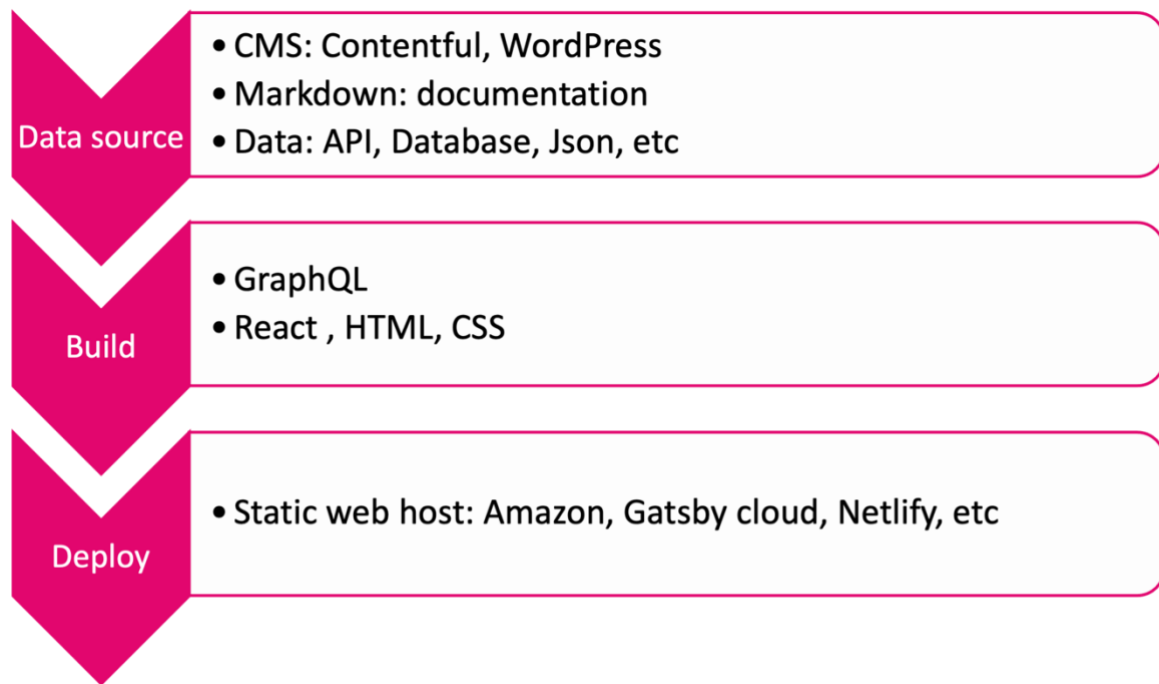


Figure 3: Gatsby life cycle

Since GraphQL is a major part of Gatsby, it is worth the effort to discuss it more.

3.2.2 GraphQL in Gatsby

GraphQL is a query language for APIs that gives the clients the possibility to request for the exact data and fields that is required. Unlike a classic Rest API, in GraphQL you only ask for the fields that you need and nothing more will be sent to you. This makes the responses of GraphQL queries very predictable and the applications that use it are faster with less errors as it is the application that decides what data to receive instead of the server. The other positive point of GraphQL is that in a single query it returns all the resources with the same reference. It means, that if a client asks for a user data it can return multiple user's data or only a single user in one query. This feature of GraphQL is not possible in Rest APIs and a separate end point URLs should be created to support the same functionality. (A query language for your API N.D.)

Figure 4 is an example of querying a data structure with GraphQL. As it is visible the full data of a user contains *firstName* and *email* fields too, but they are not returned in the result because the query did not request for those fields. Hence, we will get only the data and the fields that we have requested for in the initial query.

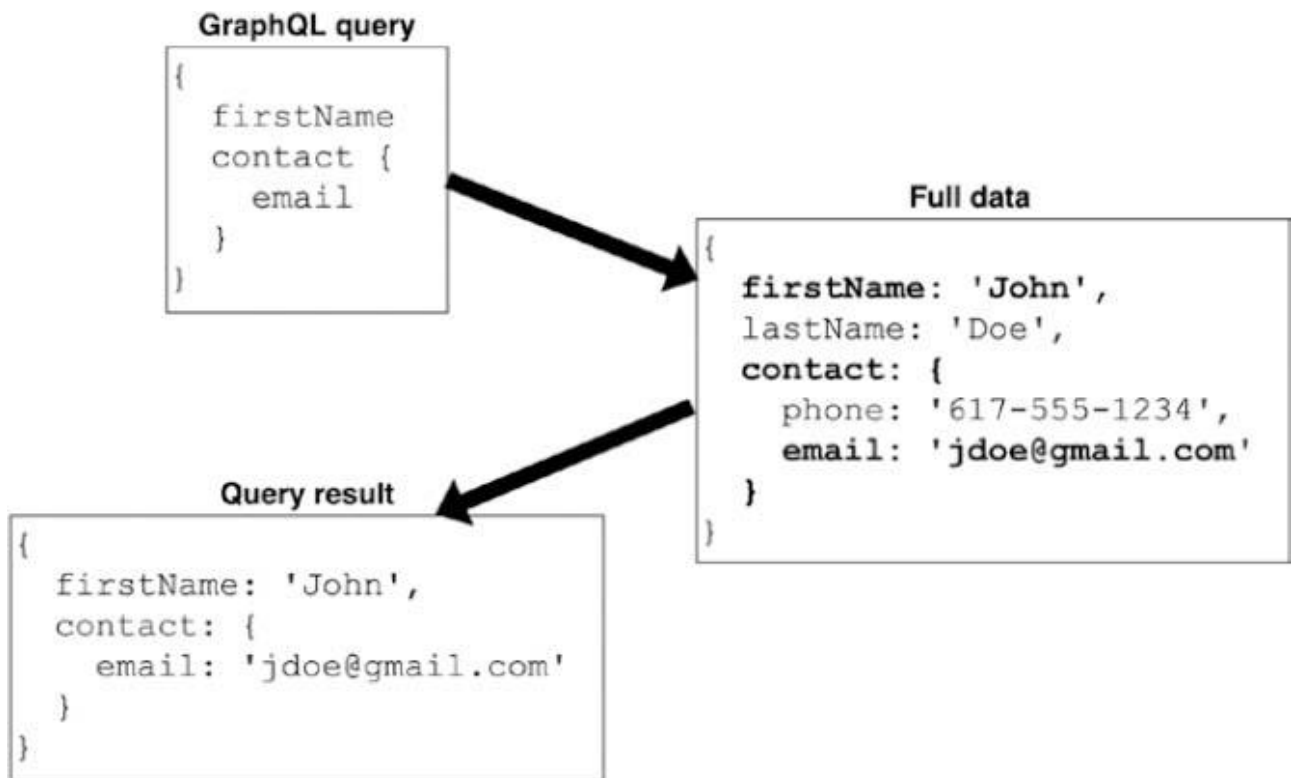


Figure 4: Querying with GraphQL (Attardi 2020, Chapter 2: Gatsby Crash Course)

GraphQL queries can be created in Gatsby using *graphql* tag. It is used to define template queries that later Gatsby will run them during build time: `graphql`Your query here``. It is important to mention here that these queries will only run during build time and not in runtime. Also using variables in GraphQL queries is not support. In the build time Gatsby gathers these tagged templates and generates the GraphQL query which there returned response will have the same structure as the queries. These retuned data later will be feed to page templates and components to generate the HTML pages. (Attardi 2020, Chapter 2: Gatsby Crash Course.)

3.2.3 GraphiQL tool

Gatsby offers a very useful tool for creating GraphQL queries called GraphiQL. It is a powerful GraphQL integrated development environment (IDE), which has syntax highlighting, autocompletion and interactive query creation to mention a few.

[http://localhost:8000/ __graphql](http://localhost:8000/__graphql) is the address that this tool can be found at.

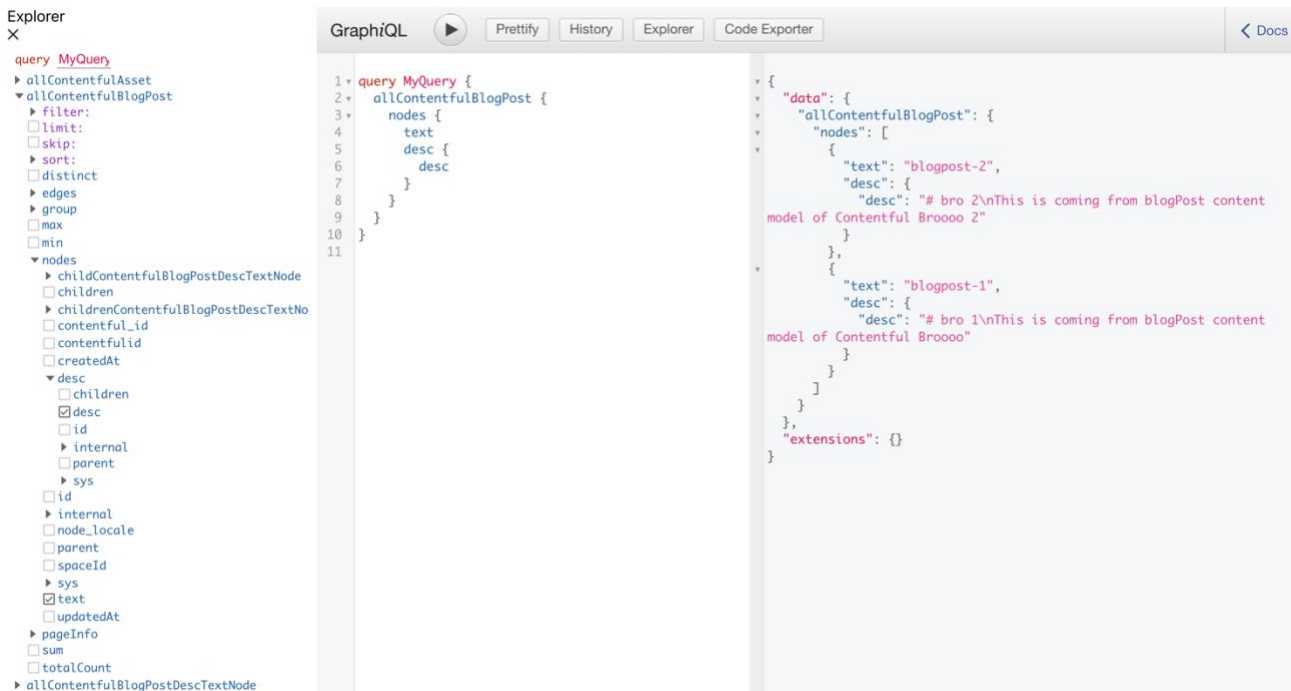


Figure 5: GraphQL tool

Figure 5 displays the interface of GraphQL tool. It is generally consisted of 3 columns. The first column from left is Explorer that shows the entire GraphQL schema data, content types and its fields. It can be used to generate queries interactively by clicking the checkboxes. Here it is also possible to filter and sort the returned data conveniently by using GraphQL tool's interactive features. The next column is the query column which shows the generated query. Here the query can be edited and modified too. The third column on the right is the results column that the returned data of the queries will be displayed in this section. Lastly there is the Docs button on the right top side of the GraphQL tool. Clicking on this button will open a slider that can be used to check for documentation about the content's types and schema elements and fields.

Queries can be divided into 2 types in Gatsby, page queries and static queries. static queries are called component queries too.

3.2.4 Page queries

Page queries are used on the higher level page components. Page components are the high level React components that Gatsby creates the final pages from them by passing the page queries' results as a data props (*props.data*) to them. Page components are like templates that Gatsby can

use them to create pages dynamically. To get the right data to the pages components, variables can also be passed to these page queries. These types of queries can be declared by the help of Gatsby's *graphql* tag and exporting a constant called *query*, but each page component file can have only on query. These queries should be placed in the same file as the page components that later during the build the query's returned data will be passed to the page component on the same file and its results can be access inside the page component's *data prop*. (Querying Data in Pages with GraphQL N.D.)

In Figure 6 there is an example of a page component and page query. This page component (HomePage) should be placed in */src/pages/homePage.js*, then Gatsby by default will create a page from the file and its name will be used as its URL (*localhost:8000/homepage*).

```
import * as React from "react";
import { graphql } from "gatsby";

const HomePage = ({ data }) => {
  return (
    <div>
      <h1>Next line is BlogPost's description: </h1>
      {data.contentfulBlogPost.desc.desc}
    </div>
  );
};

export const query = graphql`
  query HomePageQuery {
    contentfulBlogPost {
      desc {
        desc
      }
    }
  }
`;
```

Figure 6: Page component and page query example

The output of the above page component can be accessed at *localhost:8000/homepage* in which the result of the page query will be rendered on the browser (Figure 7).



Next line is BlogPost's description:

bro 2 This is coming from blogPost content model of Contentful Broooo 2

Figure 7: Page component output in browser

3.2.5 Static queries

Static queries (component query) are low level queries that can be used anywhere in a component. For creating a static query Gatsby provides *useStaticQuery* that uses the React hooks to query for the data inside a component during build time. Because it is a React hook all the rules of the hooks apply to it too. Also, as the static name suggests these types of queries cannot be dynamic and unlike the page queries, can not contain any variables in them. The other limitations of the static queries are that they should be located under */src* directory, and each file can have only one single *useStaticQuery* instance. (Querying Data in Components with the useStaticQuery Hook N.D.)

In figure 8 there is an example of a static query that shows the usage to *useStaticQuery* and its implementation of in an React component. It should be noted that only one static query can be used in a file.

```

src > components > JS header.js > ...
1
2  ∨ import React from "react";
3    import { useStaticQuery, graphql } from "gatsby";
4
5  ∨ export default function Header() {
6  ∨    const data = useStaticQuery(graphql`
7      query HeaderQuery {
8        site {
9          siteMetadata {
10             title
11          }
12        }
13      }
14    `);
15
16  ∨    return (
17  ∨      <header>
18        <h1>{data.site.siteMetadata.title}</h1>
19      </header>
20    );
21  }
22

```

Figure 8: Static query example

Gatsby's *StaticQuery* component is the other way to create static queries. But this component is deprecated in Gatsby version 5. Therefore, it has not been discussed here.

3.2.6 Page creation in Gatsby

As it was discussed in the previous chapters Gatsby will automatically generate pages from files that are located in the `/src/pages/` directory and the file name will be as the URL of generated page. Imagine if there are hundreds of the recipes and each of them needs to have a separate page on the site. Creating these manually and hard coding them will be extremity time consuming and repetitive specially if they have the same structure and interface. To avoid this issue, it is better to create pages programmatically and use one single page component to generate multiple pages. To achieve this Gatsby provides some functions to generate multiple pages dynamically from a page template component. For example, a recipe page template component that can be used to create multiple pages with the same interface and structure but with different recipe

information. It is also possible to customize this page creation process totally, for example change the location of the page component files, create a custom URL and routing for the pages, add or remove the page's data, modify, or create new data types and fields in the GraphQL schema and many more customizations can be achieved.

All the mentioned customizations can be achieved with a file called *gatsby-node.js* which should be located under the root directory of the project. Gatsby provides many useful functions such as *onCreateNode* and *createPages* that can be used in the *Gatsby-node.js* file. These functions will run on different stages of build time. *onCreateNode* function will run every time the Gatsby creates a node in GraphQL schema, hence it can be used to add or modify a specific node. (Attardi 2020, Chapter 2: Gatsby Crash Course.)

The *createPages* function can be used to dynamically to generate pages and because it is called only when the GraphQL schema is ready, it is possible to run queries and get the necessary data and use them to create pages dynamically. Some helper tools such as *action* and *graphql* that can be passed as arguments to *createPages* function are available that can be taken in use. The *action* object contains a function called *createPage*, which should be called for each page with three key value data as follow:

1. path: the page's desired URL, in which the generated page can be found at.
2. component: the address of the page component which will function as page template.
3. context: the page's context, where custom data can be added. All the data passed in context will be available for the page component template and it will be exposed as GraphQL variable. We can use this in the page component to filter and query data based on page's ID or path. (Attardi 2020, Chapter 2: Gatsby Crash Course.)

```

1
2
3 exports.createPages = async ({ actions: { createPage }, graphql }) => {
4   const results = await graphql(`
5     {
6       allRecipe {
7         edges {
8           node {
9             id
10            title
11          }
12        }
13      }
14    }
15  `);
16
17  results.data.allRecipe.edges.forEach((edge) => {
18    const recipe = edge.node;
19
20    createPage({
21      path: `/recipes/${recipe.title}/`,
22      component: require.resolve("../src/templates/RecipePageTemplate.js"),
23      context: {
24        title: recipe.title,
25        id: recipe.id,
26      },
27    });
28  });
29 };
30

```

Figure 9: Creating pages dynamically

In the above figure 9 the usage of the *createPages* function is demonstrated. The *graphql* API is called inside the function to query for all the Recipe data which will return a list of all the Recipe data as an array. Then the returned value of the query is looped through and for each node of the result the *createPage* function is called and path, component, and context are passed to it to create a recipe page from each node. The data in context then can be used as variable in the page component, in this case *RecipePageTemplate.js*, to query the page's data and use these variables as filters to get the corresponding recipe data.

In this example only one page type, Recipe page, has been used. But if it is required to have other page types for example article page, then the corresponding GraphQL query for all the article pages can be added to the same *graphql* string. Then same as the recipe page creation the returned data for the article pages should be looped through and the *createPage* function should be called for each article node data.

3.2.7 Plugins

Plugins are a group of code, functionalities and additions that can be installed into a Gatsby project codebase to utilize it to our advantage. Plugins are an important part of Gatsby workflow, without it, Gatsby does not offer many advantages compared to a traditional React project. The real advantage of a Gatsby project is in utilizing its plugins. Plugins can be found in Gatsby Plugin Library and can be installed with yarn or npm package managers. It is not mandatory to always use the already existing plugins, it is also possible to create a custom plugin of your own that has been tailored to the project's need perfectly. Gatsby's plugins can be categorized into many different types but the 2 most common and important of them are transformer plugins and source plugins. (Attardi 2020, Chapter 2: Gatsby Crash Course.)

Transformer plugins

Gatsby's transformer plugins are a type of plugin that can transform data from one format to another during the build process of a Gatsby site. More specifically, a Gatsby transformer plugin is responsible for processing content that is sourced from external data sources (such as markdown files or JSON data) and transforming that content into a format that can be used by Gatsby's GraphQL data layer. This transformation step is necessary because Gatsby's GraphQL layer requires a specific format for data in order to be queried and used in the creation of pages and components. (Attardi 2020, Chapter 2: Gatsby Crash Course.)

Gatsby provides a number of transformer plugins out of the box, including transformers for markdown files, JSON data, and images. Additionally, third-party transformer plugins can be installed and used in a Gatsby site to handle other data formats or custom data processing requirements. Overall, transformer plugins are an important part of Gatsby's data layer, and are essential for sourcing and transforming data into a format that can be easily consumed by a Gatsby site's pages and components.

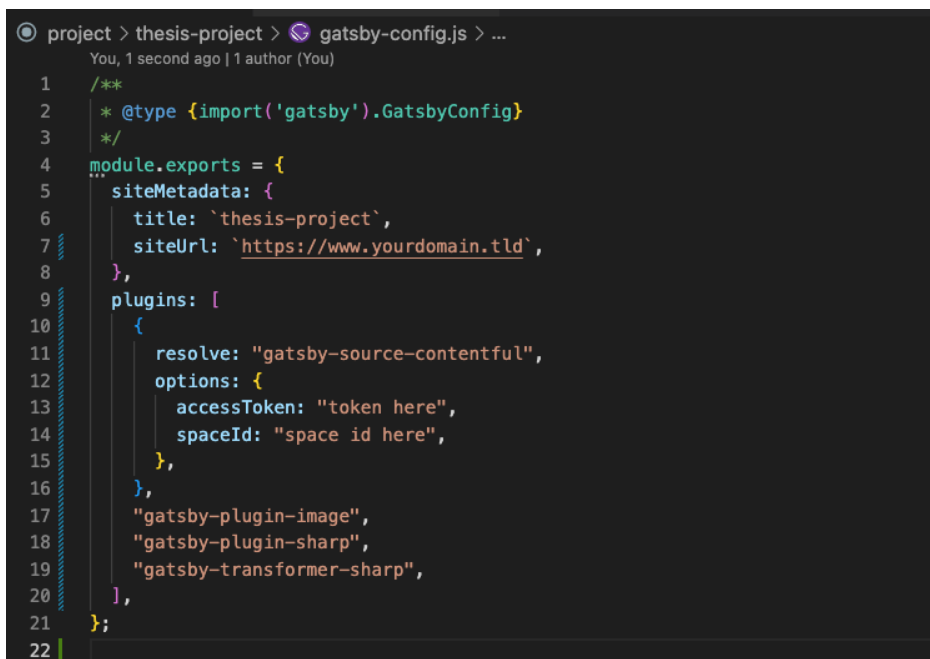
Source Plugins

These types of plugins are used to create more data into Gatsby GraphQL schema from different sources. Without the source plugins the GraphQL schema is quite empty. Source plugins pull data from a variety of sources such as APIs, content management systems (CMS), databases,

Markdown files, CSV files, and more. Then these data can be used to create static pages and websites. Each source plugin is designed to handle a specific data source and can be configured with options to customize how the data is fetched and transformed. For example, the Gatsby source plugin for WordPress allows Gatsby to pull data from a WordPress site, while the Gatsby source plugin for Contentful allows Gatsby to pull data from the Contentful CMS. (Attardi 2020, Chapter 2: Gatsby Crash Course.)

Using source plugins is an important feature of Gatsby's data layer, that allows developers to easily integrate data from various sources into their Gatsby projects. By utilizing source plugins, Gatsby projects can be built with up-to-date and dynamic data without sacrificing the benefits of static site generation. As if the source data changes, the change will trigger a project rebuild, which will cause the plugin to fetch the data again and the GraphQL schema and the static data of the site will get updated with the new data.

As stated before, plugins can be found at Gatsby's plugin library and installed in with yarn and *npm* package managers. Most of the plugins in the library have documentation about how to use and utilize the plugin in a project. Generally, it is required to add the plugin and its options into *plugin* array of the *gatsby-config.js* file located in the root directory of the project.



```
1  /**
2   * @type {import('gatsby').GatsbyConfig}
3   */
4  module.exports = {
5    siteMetadata: {
6      title: `thesis-project`,
7      siteUrl: `https://www.yourdomain.tld`,
8    },
9    plugins: [
10     {
11       resolve: "gatsby-source-contentful",
12       options: {
13         accessToken: "token here",
14         spaceId: "space id here",
15       },
16     },
17     "gatsby-plugin-image",
18     "gatsby-plugin-sharp",
19     "gatsby-transformer-sharp",
20   ],
21 };
22
```

Figure 10: Example of Gatsby plugins

4 Content management systems (CMS)

4.1 What is a CMS?

A Content Management System (CMS) is a software application that helps companies to create, manage and publish their digital content. It is a single location where all the data such as blogs and articles contents are located. With the CMS it is easier to directly create content inside a browser and not to worry about writing them in source code or as a markup language. This removes the necessity to have the developer every time the content of a site needs to be updated. (Attardi 2020, Chapter 1: Introduction to Netlify CMS.)

For example, in a traditional static blog post site every time a new blog needs to be published or modified a developer is needed to add the new content to the source code or as markup language. This is a very time consuming and expensive process with multiple steps in between to achieve the goal. And if the same content needs to be published to other sites as well, the same tedious process needs to be done to each site in order to update the same content to all of them. This issue can be overcome by utilizing a content management system for managing the content of all the sites in a centralized location. A CMS eliminates the need to update the code to publish a new blog post, hence no developer is needed. A content manager without any technical skills simply adds the new content to the CMS only once from a browser in a user-friendly environment and all the sites will be updated with the new content. This aspect of CMS makes them extremely inexpensive, simple, and efficient.

4.2 Traditional CMS

In a traditional CMS generally the content creation and content display take place in a single web application. A content manager uses an interface to log in into the system and creates the content which will be save in a database. Then a user of the site will use the same system to view the content fetched from the database. (Attardi 2020, Chapter 1: Introduction to Netlify CMS.)

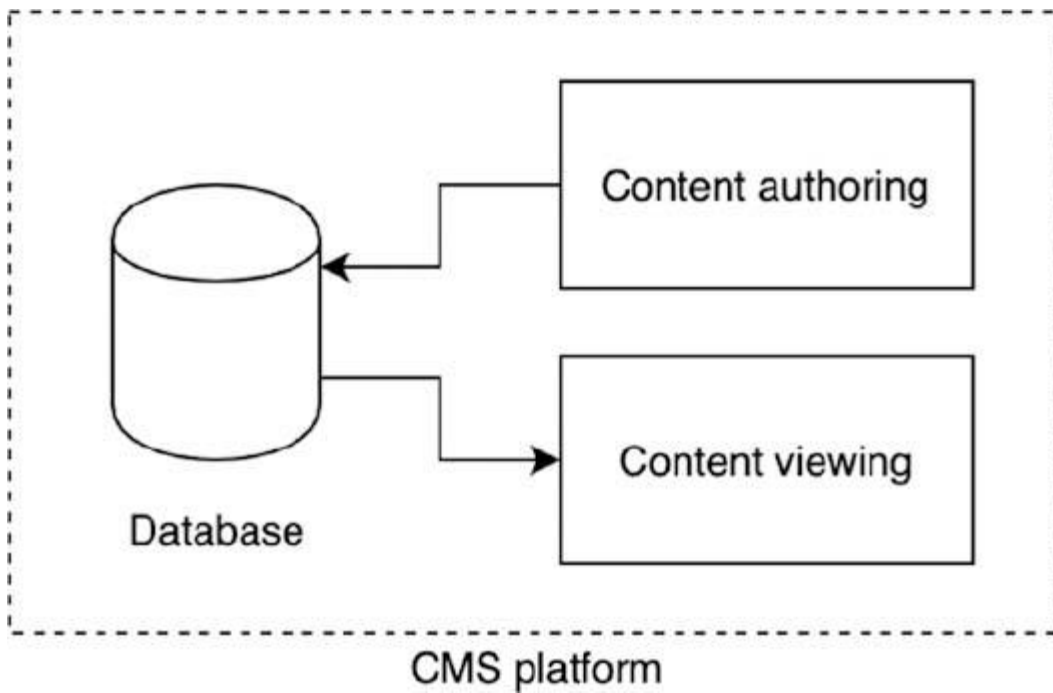


Figure 11: Traditional CMS (Attardi 2020, Chapter 1: Introduction to Netlify CMS)

4.3 Headless CMS

In a headless CMS unlike the traditional CMS the content creation and content display take place on different platforms. In this case content can be provided to the front end via some type of APIs. One of the great advantages of a headless CMS is that the content can be sent to multiple frontend application at the same time with the usage of the APIs. In a headless CMS the content section and its display are totally decoupled from each other which creates the possibility to present the content with more flexibility. Contentful and Netlify are examples of a headless CMS. (Attardi 2020, Chapter 1: Introduction to Netlify CMS)

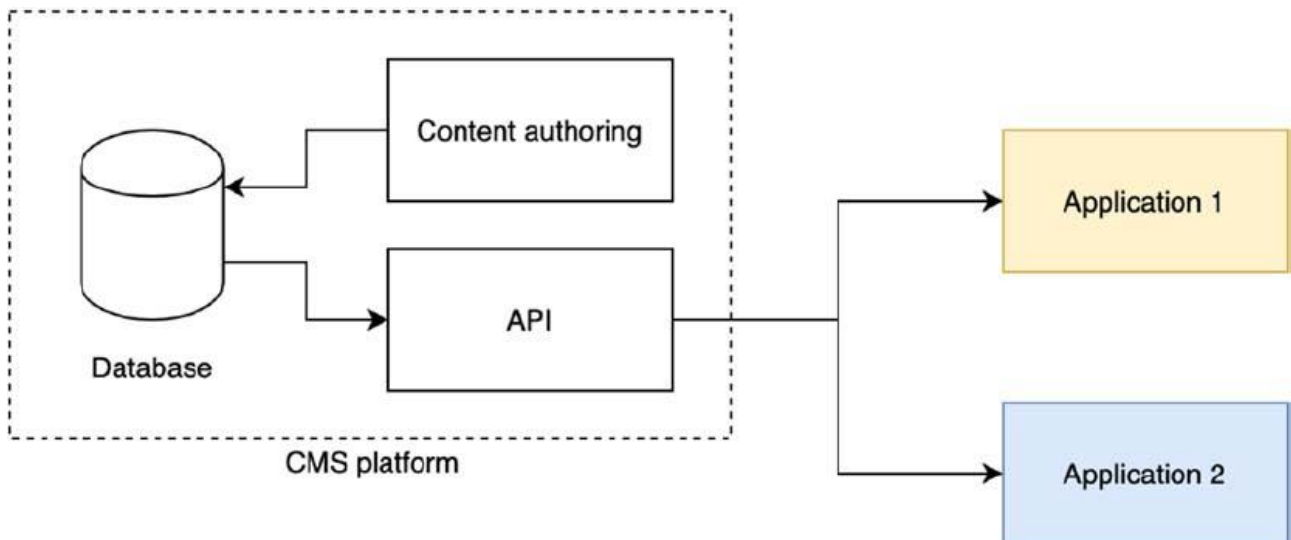


Figure 12: Headless CMS (Attardi 2020, Chapter 1: Introduction to Netlify CMS)

In the next chapters Word Press and Contentful has been discussed more.

4.4 Contentful

Contentful is a headless CMS owned by Microsoft with a web-based content editor application that offers many useful features. Contents such as text, images and videos can be uploaded to Contentful and be delivered to the front-end through its APIs. The different between Contentful and other CMSs is that Contentful is not page-based. It means that the front-end application adapts the content structures not the other way around as is usual for other CMSs. This features of Contentful is called content modeling. (Introduction to Contentful N.D.)

Content modeling simply put is the overall structure and organization of contents. With content modeling it is possible define what type of information a content can contain. It is like an outline for the content. Under content model there are different content types and each content types can have multiple fields. (Content modeling basics N.D.)

For example, an article page in a website can have its own individual content type in Contentful. The title of the article page should have its own text field and a reference field to Author content type for the author of the article. By adding the Author as a reference to the Article page content type, it can be reused in other articles that has been written by the same author. This way by

defining a field as a reference to another content type we can reuse the same content in other places and avoid content duplication and redundancy.

It is vital to consider the needs and requirements of all parties involved in the service when designing the content models and content types. It should be future proof and the future development and feature should be considered. (Content modeling basics N.D.)

4.4.1 Contentful content APIs

For creating or modifying content in Contentful there are 2 methods. The first approach is with the help of its web application and graphical user interface. The Contentful web application provides a comprehensive graphical user interface and a user-friendly environment that makes the usage of the service much easier. For these reasons a content creator will quickly be able to use the service and start creating and modifying the contents without any extensive technical knowledge.

The second method is with the usage of the Contentful APIs. Contentful provides Rest and GraphQL APIs, that by utilizing them we can start modifying, updating, creating, deleting, importing and exporting the content. Generally, Contentful has 5 types of APIs that each of them are explained in the following:

Content Delivery API (CDA)

This is a public read-only API for retrieving data and content from Contentful which is available at *cdn.contentful.com*. The CDA API provides the data to the apps, mobile applications and websites as JSON and the media data such as images and videos are provided as files. CDA API is accessible on a global network which works efficiently. It responds to the API calls from the closest server to the user with the required data. This makes the data delay shorter and improves the user experience of the application. (Contentful content APIs N.D.)

Content Management API (CMA)

The CMA API unlike the CDA requires logging in as Contentful user and authentication. This is a read-write API used for managing the content which is available at *api.contentful.com*. Its main purpose is to export or import content automatically through code, integrating the Contentful will

other system on a backend, or building a custom tool for managing the data. (Contentful content APIs N.D.)

Content Preview API

Content Preview API is similar to the Content Delivery API, but it is used for previewing the content before releasing them to the productions which is available at preview.contentful.com. This is used mainly by content creators, authors, and content managers to preview and test the content before release as if it is published. Content Preview API can be distinguished from the Content Delivery API by a different access token. The only different between them is that the Content Preview API returns the contents that are in *Draft* and *Published* status, but the CDA returns only the contented that are in *Published* status. (Contentful content APIs N.D.)

Images API

Images API is used to deliver the images stored in Contentful and manipulate them based on the application's need. This API can be accessed at images.ctfassets.net, that with the help of it we can crop, resize, change background, or convert them into other formats. (Contentful content APIs N.D.)

GraphQL API

It is a content API that provides each space of the Contentful as GraphQL schema based on existing content type on the space. The schema gets updated every time there is a change in the content types. The GraphQL API can be accessed at graphql.contentful.com. (Contentful content APIs N.D.)

4.5 WordPress

WordPress is one of if not the most popular blogging platform. It was developed by a company called Automatic in 2005. It is an open-source platform used mostly for blogging. But during years

it has evolved to become a framework used for multiple different purposes. Also, WordPress's v3.0 has given it the full capability of becoming a Content Management System. One the reasons the WordPress has become so famous is because of its active contributors. There are thousands of free plugins and themes that are available to utilize. This gives the WordPress a wide range of possibilities that each site will have its own unique view and features. (Pearce 2011, 225-226.)

The core technologies that WordPress is built with is PHP and MySQL. It is important to understand that WordPress unlike Contentful is not a headless CMS but a traditional CMS which both content creation, content storing, and content viewing happen inside one platform.

Posts and pages are the fundamental content types that WordPress stores. Posts are generally the building blocks of a blog that are time-stamped and tagged or categorized. On the other hand, by default pages are generally the static part of a site which are not tagged or categorized and the time-stamp is no not that important because of its static nature. (Pearce 2011, 226-227.)

One of the biggest differences between the post and pages is that pages are arranged hierarchically but and post are categorized and tagged instead. For example, the pages can have sub-pages and are better to be used for a website that most of its contents are static and hierarchically navigable. Of course, it can contain posts as well but presenting the blog posts are the primary goal of the service. (Pearce 2011, 226-227.)

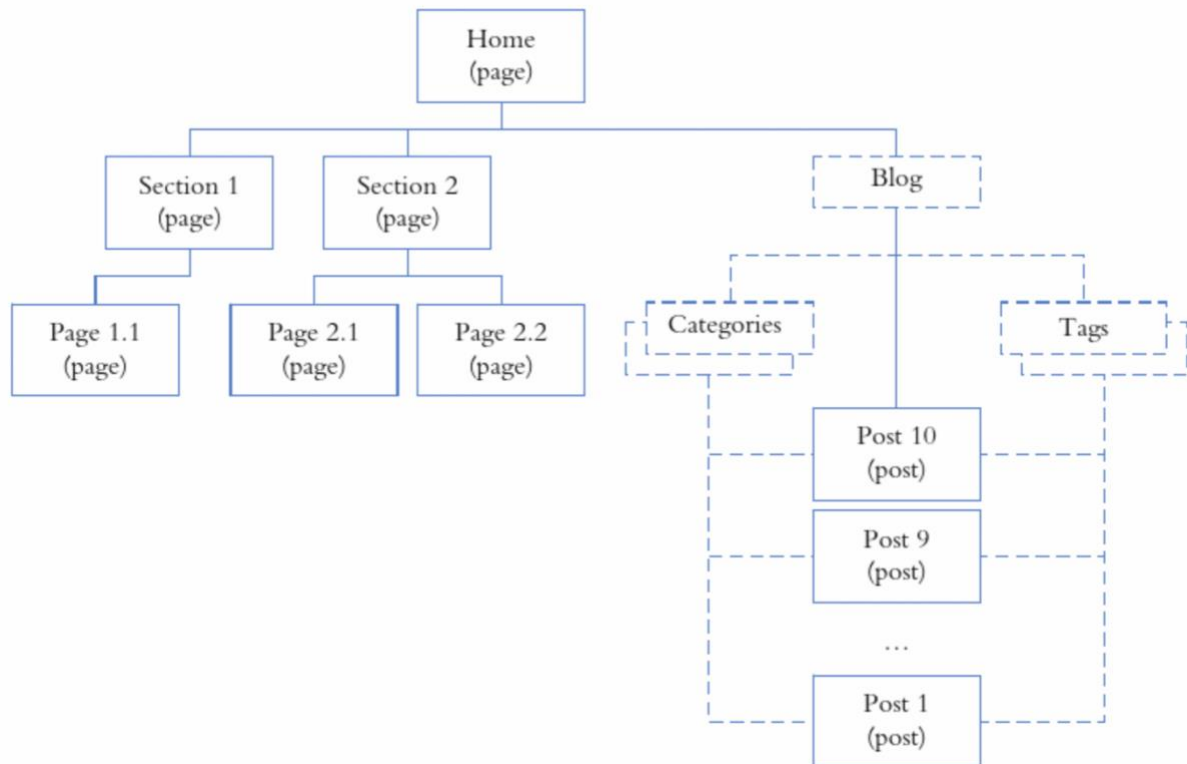


Figure 13: Page structure in WordPress (Pearce 2011, 226-227)

Posts are used when the classic blog methodology is required, and main goal of the service is to provide the blog post to the users as in in Figure 11. The implementation, modification and management of both posts and pages are extremely similar but it is critical to understand their difference and the primary purpose of the site when deciding the which section of the site should be as categorized posts and which as hierarchically arranged pages. (Pearce 2011, 226-227.)

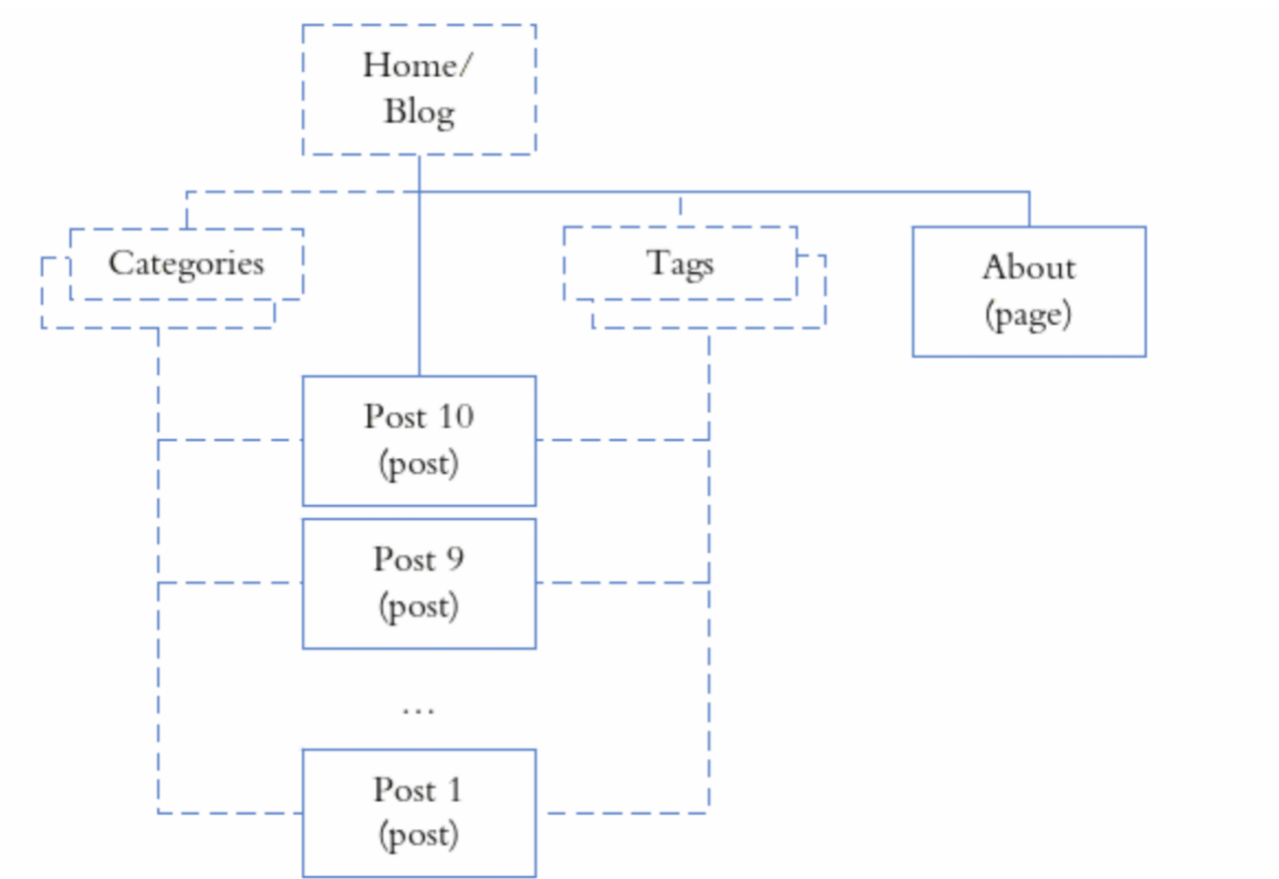


Figure 14: Post structure in WordPress (Pearce 2011, 226-227.)

Links and media files are the other type of data entries that WordPress stores. Media files such as images can be attached to a specific post or page and can be shared among multiple pages. Images can also be given a specific size that are resized when uploading to the media repository. This feature will avoid breaking the site's layout if an image is too big. The other very important aspect of WordPress is its theme API and plugins. Theme API allows developers to create custom and unique theme for the site. Plugins are used to extent or modify the functionality of a WordPress powered site. (Pearce 2011, 228-229.)

Plugins and themes of the WordPress, themes, and plugins are out of scoop of CMSs, therefore they have only been shortly mentioned here.

5 Project

5.1 Overview

In this section Viinimaa (viinimaa.fi) website was selected as an example project of the previously researched subjects. Viinimaa is one of two flagship websites of Anora group company. It is mainly a static web site with hundreds of articles and recipes that promotes the alcoholic and non-alcoholic drinks and presents a lifestyle around its products. Viinimaa is an enormous site with around 2000 pages that has many different technologies in its usage. It is worth mentioning here that the goal of this section is to present the overall Viinimaa's architecture, different blocks and how they work together seamlessly, not presenting its code base.

5.2 Technologies

Viinimaa's core technologies are Gatsby framework and Contentful as its CMS. Of course, it is not only these 2 but a few more technologies and systems that are listed below. Some of these technologies are out of scope of this thesis therefore they have not been elaborated more.

- Gatsby framework: used for building the site and front end.
- Contentful: as its Content Management Systems for storing all the content except media files and product information.
- Adobe Commerce: an e-commerce platform written in PHP that was called previously Magento. Viinimaa uses its GraphQL API for product information and product search.
- Cloudinary: a media platform for storing, manipulating, managing, and delivering images and videos to websites and apps.
- Elastic search: used for site's search functionalities
- Azure: used for hosting the website
- Gatsby Cloud: used for Building the site, automatic build triggers and web hooks
- GitHub: source code's version control
- Google Analytics: site analytics and events

5.3 Architecture

In the project's architecture section an overall perspective of the project, and the relations between the previously explained technologies, their usage, data flow, content management, and the project's deployment has been discussed.

In the figure 15 the map of the Viinimaa website is visible.

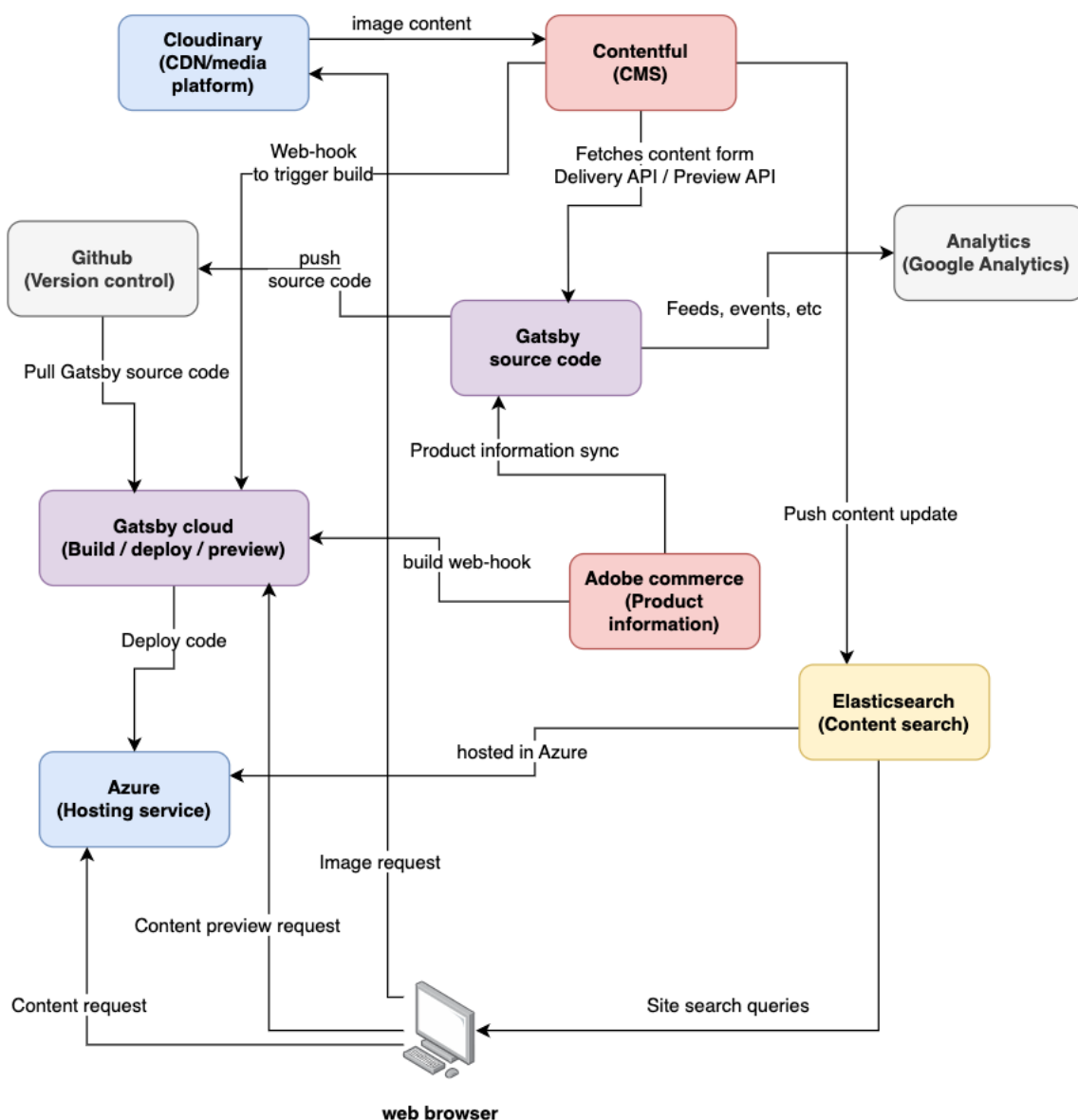


Figure 15: Viinimaa's architecture

As the relations of each part of the project is visible in the above figure, a more clarification of these relations is required.

5.3.1 Gatsby Cloud

When a change happens in Gatsby's project source code, the code changes get pushed to GitHub manually. Then Gatsby Cloud gets notified automatically through the configured pipeline that the source code has been updated. This will trigger a rebuild process of the site on Gatsby Cloud that will update the site with most recent changes. During the build process the first step is data sourcing. Gatsby fetches the content from Contentful through its delivery APIs and syncs the product data from Adobe Commerce, then generates a local schema of all the content. A successful build generates the Gatsby static code which is then deployed to Azure for hosting. Then Azure service remove the previous static code from the server and deploys the updated version instead, which makes the latest changes accessible to the end users. During the build process in the Gatsby Cloud if build fails, the site will not get deployed to Azure until the next successful build. This feature is very handy that removes the possibility of sites going down because of an error.

Gatsby Cloud also supports site caching. This means that if a content is updated in Contentful and a new build is triggered, Gatsby could only rebuilds the pages in which the updated content has been used and the rest of the site will be used from the cache files generated during previous build. Therefore, these types of builds will take shorter time and consumes less resources.

5.3.2 Contentful and Cloudinary

As it has been explained in the before, Contentful is the CMS of the Viinimaa site. Contentful is also directly connected to Cloudinary. Since images are stored in Cloudinary, Contentful needs to be connected to it. This connection has been achieved by installing the Cloudinary plugins to the Contentful which allows the content creator to add images from Cloudinary to the contents. This image information then will be provided to Gatsby when it is fetching the content.

Cloudinary has many useful features such as image and video transformation and manipulation. It also utilizes the caching which makes the image delivery to the end user much faster.

Contentful is also connected to the Gatsby Cloud through webhooks. This connection is needed to notify Gatsby Cloud to trigger a new site build when there is a change in the content of the site. After the new build site is updated with the latest changes in Contentful.

5.3.3 Adobe Commerce

Adobe Commerce is used only for product information and product search. Product information is fetched during build alongside with other data source and the local schema get generated. For search purposes the Adobe Commerce provides a live API that returns the product information based on the end user's search term. Adobe Commerce and Gatsby Cloud are connected same as Contentful through a webhook. When a product is created or modified in the Adobe Commerce, Gatsby Cloud gets notified through the webhook which will trigger a new build to update the live site with the latest changes.

5.4 Integration

5.4.1 Gatsby and Contentful integration

To integrate Gatsby and Contentful together the official *gatsby-source-contentful* plugin has been used. *Gatsby-source-contentful* is a useful plugin and easy to set up. The only thing that is required to set this plugin is Contentful space id and access token. This information can be easily found from Contentful *setting/API keys* menu. The access token and the space id should be added to the *gatsby-config.js* file of the project as it is shown in the figure 10.

5.4.2 Gatsby Cloud and Contentful

To trigger a new build when a content is created or updated in Contentful, a webhook needs to be set up. To set up the webhook first we need the Gatsby Cloud webhook's URL. To access this URL, we need to login into the Gatsby Cloud service and select the correct workspace and site from the opened list. Under the *Site Settings* tab (check figure 16) the URL can be found from the *Webhook* section. This URL needs to be added to Contentful webhook.

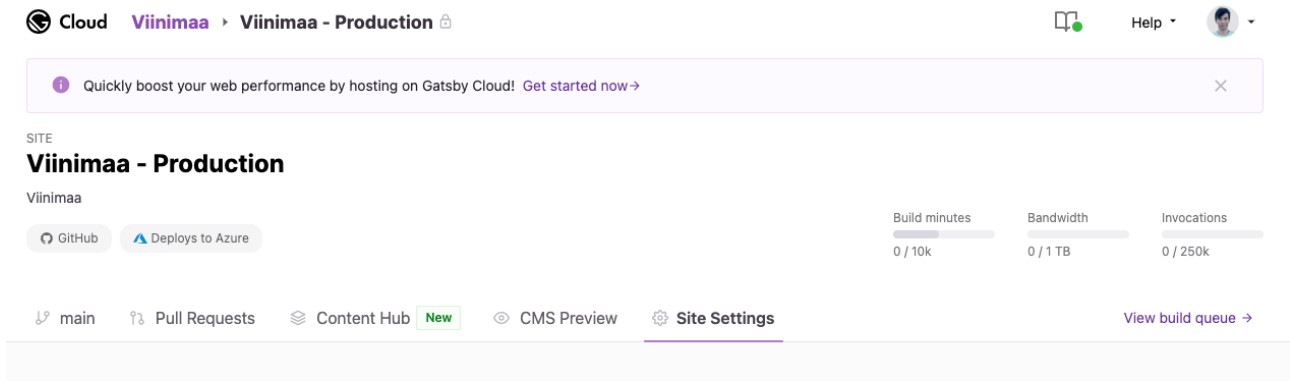


Figure 16: Gatsby Cloud site settings

Next, a Contentful webhook needs to be created. To create a new webhook, we need to login into the Contentful and after going to the correct space, select master environment. Webhooks are only available in master environment. From the settings tab select webhook and then click on the Add Webhook button. A window opens that the Gatsby Cloud's webhook URL needs be pasted in the URL field of the page (check figure 17). Here you can define other settings such as webhook's environment, type a title and on what circumstances the webhook should notify the Gatsby Cloud for a new build. Finish the other settings and save. Now the webhooks are set up and every time there is a change in the Contentful a new build will be triggered in the Gatsby Cloud.

Webhook: Unnamed
Active
Save

Details

Name (required)

URL (required)

POST

☒ Active
Webhook calls will be performed for the configured events.

Triggers

Specify for what kind of events this webhook should be triggered.

☒ Trigger for all events
☐ Select specific triggering events

Filters

This webhook will trigger only for entities matching the filters defined below.

Environment ID (sys.environment.sys.id)

equals

master

Remove

[+ Add filter](#)

Headers

[+ Add custom header](#)
[+ Add secret header](#)
[+ Add HTTP Basic Auth header](#)

Content type

application/vnd.contentful.management.v1+json

Select one of allowed MIME types to be used as the value of the Content-Type header. Any custom Content-Type header will be ignored.

Content length

☐ Automatically compute the value of the Content-Length header

If this option is selected, the byte size of the final request body will be computed and used as the value of the Content-Length header.

Payload

You can customize the webhook payload to match the format expected by the service your webhook calls.
[View documentation](#)

☒ Use default payload
☐ Customize the webhook payload

DOCUMENTATION

- [Intro to webhooks](#)
- [Webhook management API reference](#)

WEBHOOK URL REQUIREMENTS

Please note that webhook calls will not be performed against the following URLs:

- Private IPs (10.x, 192.x, etc.)
- Localhost
- Hostnames without a top-level domain
- URLs that resolve to localhost or redirects

WEBHOOK IP SOURCES

If you need to restrict access to your webhook endpoint based on an IP visit [AWS reference page](#) to obtain information about IP ranges we use to deliver webhook calls.

Figure 17: Contentful webhook

5.4.3 Gatsby and Adobe Commerce integration

Product data are fetched from Adobe Commerce during build with the help of a custom plugin that had been created only for this purpose. The data are fetched through an API that creates the necessary schemas and content types. Since all the product data are coming from the Adobe Commerce, it is required to rebuild the site if there is a change in product data source in order to have an up-to-date site. There for the Gatsby Cloud and Adobe Commerce are connected to each

other through a webhook that triggers a new build in Gatsby Cloud when a product is created or updated.

5.5 Results

The result of the project was a real-life enterprise organization's architecture that has the potential to create a modern and blazing fast static website with great SEO. The Anora group has spent hundreds of thousands of euros to create their infrastructure using this architecture to achieve their business goals. therefore, this architecture and integration can be used by small to enterprise organizations as a base template to create their own website to achieve their business goals without any expenses. This will help especially small to medium organizations to benefit from it. This architecture can be used for similar sites as Viinimaa that has the same types of requirements that most of the site is static, and SEO is important. For example, a news website, a product information website, an article website, a brand's advertisement website, or an organization's information website.

The reason why Gatsby has been chosen over Next js as the frontend of the project, is because the Viinimaa.fi is mostly consisted of static contents and pages that does not require user's input or interaction. Therefore, Gatsby was the best option as static SSR framework compared to Next js which is used when the site's dynamic aspect is greater than its static aspect. Also, Contentful was integrated with Gatsby because of its modern and headless behavior. It is also very fast, future-proof and great for enterprises with lots of data such as Viinimaa. The reason why In Viinimaa WordPress was not used is because, it is not headless as Contentful, it is a quite old system and tends to perform slow. WordPress is a great option for small to medium companies but in Viinimaa's case that the site has more 2500 pages with a huge number of images and medias, serenely WordPress was not a perfect option as Anora needed something reliable, modern, fast and future-proof.

6 Conclusion

The goal of this thesis was to research the most popular SSR frameworks and CMSs and present a real-life project architecture that can be used to a create service which is highly optimized for search engines. One of the reasons why the SSR frameworks are gaining more popularity is

because of their advantages that they provide in SEO and performance. After researching the SSR and the related frameworks such as Gatsby and Next.js, it is quite clear that utilizing these technologies are crucial if a company wants to stay on the top of the results in search engine, reach more customers, be more accessible and have better performance. As in today's business every company is trying to be more accessible and easily findable through internet, site's performance and search engines are playing a great role.

Based on the researched technologies, it is clear why in the Viinimaa project the Gatsby and Contentful has been used. In Viinimaa the goal was to create a website with technologies that are modern, fast, responsive, and optimized for SEO. Also, since most of the site is static content, such as articles, product data, and recipes, Gatsby was the best option that satisfied all the Viinimaa's requirement.

In the end, the goals that had been set for this thesis have been achieved and the objectives and questions of the thesis have been answered. SSR is a technology that is growing in popularity because of the benefits that it provides. Because in SSR the pages are built on the server the search engine crawlers can easily access the content of the pages and present them to the users based on their search term. Therefore, SSR offers much better SEO and performance, since the pages are already built on the server and client (browser) has less computation to do. On the other hand, in the client side rendering, the contents are being rendered on the client side, search engine crawlers have no understanding of the site's content until it is rendered on the client. For this reason, the client side rendering sites have less visibility and worse SEO compared to SSR. Two of the most popular SSR frameworks are Gatsby and Next.js. Gatsby is used for more static sites and Next.js for more dynamic sites. In the project example viinimaa.fi the Gatsby framework has been used because most of the site is static pages with its contents coming from Contentful CMS. From the CMS point of view, the headless CMSs mythology is gaining more attention due to its simplicity and usability. Headless CMS, such as Contentful, has separated the content from the frontend. For a headless CMS is not important in what technologies and how the data are being used and usually it is the frontend that adapts to the content not the other way around.

References

Attardi, J. 2020. Using Gatsby and Netlify CMS: Build Blazing Fast JAMstack Apps Using Gatsby and Netlify CMS. Apress publishing. Chapter 1: Introduction to Netlify CMS, Chapter 2: Gatsby Crash course

About us. Anora group website. Published N.D. Cited 31.03.2022. <https://anora.com/en/about-us>

A query language for your API. GraphQL org website. Published N.D. Cited 12.01.2023. <https://graphql.org/>

Company. Solteq Oyj website. Published N.D. Cited 12.05.2023. <https://www.solteq.com/en/company>

Contentful content APIs. Contentful website. Published N.D. Cited 03.05.2023. <https://www.contentful.com/developers/docs/concepts/apis/>

Content modeling basics. Contentful website. Published N.D. Cited 15.04.2023. <https://www.contentful.com/help/content-modelling-basics/>

Introduction to Contentful. Contentful website. Published N.D. Cited 15.04.2023. <https://www.contentful.com/help/contentful-overview/>

Konshin, K. 2018. Next.js Quick Start Guide : Server-Side Rendering Done Right. Packt Publishing. 13-15, 15-19

Most popular technology skills in the JavaScript tech stack worldwide in 2021. Statista website. Published N.D- Cited 08.01.2023. <https://www-statista-com.ezproxy.jamk.fi:2443/statistics/1292313/popular-technologies-in-the-javascript-tech-stack/>

Overview of the Gatsby Build Process. Gatsby website. Published N.D. Cited 12.01.2023. <https://www.gatsbyjs.com/docs/conceptual/overview-of-the-gatsby-build-process/>

Pearce, J. 2011. Professional Mobile Web Development with WordPress, Joomla and Drupal. John Wiley & Sons, Incorporated. 225-228.

Querying Data in Components with the useStaticQuery Hook. Gatsby website. Published N.D. Cited 22.01.2023. <https://www.gatsbyjs.com/docs/how-to/querying-data/use-static-query/>

Querying Data in Pages with GraphQL. Gatsby website. Published N.D. Cited 22.01.2023. <https://www.gatsbyjs.com/docs/how-to/querying-data/page-query/>

Server-Side Rendering (SSR). Vue.js website. Published N.D. Cited 31.03.2022. <https://vuejs.org/guide/scaling-up/ssr.html>

Vepsäläinen, J. 2016. SURVIVEJS webpack and React from apprentice to master. Leanpub publishing. 32-35

Verma, A. 2021. A deep dive into Server-Side Rendering in JavaScript. Towards Dev website. Cited 10.04.2022. <https://towardsdev.com/server-side-rendering-srr-in-javascript-a1b7298f0d04>

What is Framework. Code Academy website. Published 23.09.2021. Cited 07.01.2023. <https://www.codecademy.com/resources/blog/what-is-a-framework/>

Zola, A & Burke, J. 2021. Time-to-live(TTL). TechTarget website. Cited 10.04.2022. <https://www.techtarget.com/searchnetworking/definition/time-to-live>

