Haaga-Helia
University of Applied Sciences

Headless WordPress development with React using Gatsby.js

Juha Stenroos

Bachelor's Thesis
Degree Programme in ICT
2019

Haaga-Helia
University of Applied Sciences

Abstract

9.11.2019

Author(s) Juha Stenroos	
Degree programme **ICT**	
Report/thesis title Headless WordPress development with React using Gatsby.js	**Number of pages and appendix pages** **30 + 20**

Tämän opinnäyte on opaskirja ohjelmistokehittäjälle, joka jo kenties tuntee JavaScriptiä ja PHP:tä ja on mahdollisesti työskennellyt WordPressin parissa. Se on kuitenkin pyritty kirjoittamaan niin, että käsitteet avautuvat myös sellaiselle lukijalle, jolla ei ole aiempaa kokemusta web-kehittäjänä.

Tarkoitus on tarjota tietoa siitä, miten hyödyntää JavaScript viitekehystä kuten Gatsbya, verkkosivustojen käyttöliittymän rakentamisessa, kuitenkin käyttäen WordPressiä sisällön hallintaan.

Gatsby on suosittuun JavaScript kirjasto Reactiin perustuva ohjelmistoviitekehys, joka hyödyntää monia uusimpia verkon teknologioita, kuten GraphQl kyselykieltä. Tämän kaltaiset viitekehykset edustavat niin sanottua Single Page Application (SPA) lähestymistapaa. Muista vastaavista viitekehyksistä poiketen Gatsby sivustot piirretään kuitenkin serverillä etukäteen ja tarjoillaan käyttäjälle staattisina tiedostoina. Tämä mahdollistaa tavallista paremman suorituskyvyn. Lisäksi hakukoneiden on helppo indeksoida staattisiksi generoitu HTML sisältö. Tällä on etua sivuston löydettävyyden kannalta, mikä on tyypillisesti ollut SPA sovellusten Akilleen kantapää.

On monia hyviä syitä, miksi näin haluttaisiin tehdä. Internet on muuttunut siitä ajasta, kun WordPress julkaistiin. Se on jakautunut www lisäksi älypuhelimilla käytettäviin natiivisovelluksiin, sekä moniin muihin laitteisiin. Samaa sisältöä saatetaan tarvita muissakin kanavissa kuin verkkosivustolla ja näin ollen tiukasti verkkosivuston rakenteeseen sidottu monoliittinen ratkaisu ei välttämättä ole enää paras. Ratkaisu tähän on erottaa sisällönhallinta ja sisällön esittämiseen käytetty käyttöliittymäkerros. Tästä käytetään termiä ’headless’. Gatsby ja monet muut sen kaltaiset JavaScript viitekehykset ovat kevyitä ja moderneja ratkaisuja rakentaa verkkosivuston käyttöliittymä ja kun sivuston sisällön rakenne suunnitellaan verkkosivuston sijaan headless lähtökohdasta voidaan dataa hyödyntää muuallakin.

Keywords
JavaScript, React.js, Gatsby.js, WordPress, SPA, GraphQl

Table of contents

1 Introduction

WordPress is the most popular Content Management System in the World. It covers roughly one third of the top 1 million web sites and has 50% market share in Content Management Systems (BuiltWith Pty Ltd 2019).

WordPress started its journey as a blogging platform and over the years it has been developed as a full featured Content Management System. As recently as 2016 WordPress included REST API to give other applications easy access to content in WordPress database (WordPress Foundation 2016).

This gives developers ability to use WordPress as a headless CMS, meaning that you don't need to use WordPress presentation layer, also known as frontend, but you can use any kind of frontend application to present the data (ContentStack 2018).

One very popular JavaScript frontend library/framework these days is React, developed by Facebook. It is the same frontend Facebook.com uses. Other big websites using React as their frontend view layer include Airbnb, Netflix, Dropbox, BBC, Flipboard, PayPal, Reddit, Salesforce, Squarespace, Tesla, WhatsApp and Uber just to name a few (Coder Academy 2016). It is used by over 600 000 websites (SimilarTech Ltd. 2019).

Since React is a library that is lacking some features of full-fledged framework (Developer 2016), there are frameworks build on top of React. One of these is Gatsby.js. As it is described in Gatsby.js website "Gatsby is a free and open source framework based on React that helps developers build blazing fast websites and apps" (Gatsby Inc. 2019 A).

Gatsby generates static html pages from application for initial load and then continues as a React Single Page Application. Gatsby uses Progressive Web App principles to fetch only things you need first and then load rest of the application in a background later. It also does image optimization and uses Facebook developed query language GraphQl. These features make websites using Gatsby really fast. (Gatsby Inc. 2019 A).

This thesis is all about writing a guide on how to use Gatsby.js together with WordPress CMS.

2 Web technologies

2.1 Content Management Systems

2.1.1 History

History of Content Management in websites starts with early 90's with static webpages where webmaster would directly edit html files and upload them to a server via FTP. Mid 90's when web technology evolved content started to become more dynamic and first monolithic Content Management Systems (CMS) were born. Monolithic CMS means that one software includes everything that is needed to edit and publish content on web. Vignette was published 1995 and is commonly credited to come up with the term CMS. In following year many enterprise CMSs were published, which some, like EPiServer, still exist today. (ContentStack 2018)

In beginning of the 2000's open source CMS start to appear. PHP based CMSs like WordPress, Drupal and Joomla appear and offered a free alternative to enterprise CMSs. WordPress become the most popular by focusing on blog content delivery and allowing third party developers to extend and customize CMS with plugins and themes. (ContentStack 2018)

In 2007 when iPhone was introduced it started the smartphone revolution. Mobile devices started to effect how web content was consumed and how it needed to be delivered. This possessed a problem for monolithic CMSs which were designed to deliver web content for laptops and desktops. First reaction to this was to build separate mobile versions of websites. In 2010 Ethan Marcotte invented a term responsive design, which meant that website layouts would be adaptive to screen size. This is commonly used, industry standard practice today. Also, there was native mobile applications. All this meant that there was a need to decouple content management from display layer. Term Headless refers to CMS without a presentation layer where content is delivered via API to separate frontend application that then presents it. (ContentStack 2018)

2.1.2 Current CMS market

Currently WordPress is by far the most popular CMS. It has nearly 60% CMS market share and it's running on nearly 30% of all websites. Drupal and Joomla come second with market shares of 6.6% and 4.6%. (Mening 2018)

2.2 WordPress

2.2.1 Basics

WordPress is an open source CMS licensed under General Public License (GPLv2 or later) (WordPress Foundation 2018 B). It includes features like user management, media management, publishing tools, theming system, WordPress API (Application Programming Interface) to extend it with plugins, REST API to use it as an application framework and custom content types. WordPress is built with PHP and is storing data to relational database, either MySQL or MariaDB. (WordPress Foundation 2018 A).

One of the main concepts of WordPress API are hooks. Hooks are the way for plugin (or theme functionality) to change WordPress core functionality. Hooks works so that when certain event in WordPress happens (for example page load, or post edit), WordPress calls certain hook and you can use this hook to run a function. Hooks are divided to Actions and Filters. Actions can be used to run a certain PHP function in certain event, but these functions don't need to return anything. Filters can be used to run functions that WordPress passes data through during certain events. These functions take data in as a parameter and return modified version of that data. (WordPress Foundation 2017)

2.2.1 REST API

The Representational state transfer (REST) is a subset of www and it is based on HTTP protocol. Idea of REST is to provide uniform interface semantics to create, read, update and delete data (CRUD). This is an alternative for applications specific interfaces. REST is also stateless, which means that meaning of message does not depend on the state of the conversation. (W3C 2004)

The REST API in WordPress provides developer access to data in WordPress database remotely by sending and receiving JSON (JavaScript Object Notation) objects. This gives developer a change to create, read, update and delete WordPress data from other applications that are not build with WordPress, like JavaScript Web Applications or native mobile apps. (WordPress foundation 2018 E)

First key concept in WordPress REST API are routes and endpoints. Route is path you can access via HTTP call and endpoint is a HTTP method (such as get, post, put or delete) mapped to that route. (WordPress foundation 2018 E)

Second key concept is a request. When you make a HTTP request to registered REST route WordPress will automatically create a request object. Data that is specified in request will determine what answer you will receive back. (WordPress foundation 2018 E)

Third key concept is a response. Response object is a data you get back when you make a request. It can include requested data or error messages. (WordPress foundation 2018 E)

Fourth key concept is a schema which refers to data structure in the endpoint. Each endpoint can have slightly or significantly different structure of data it can return. Schema will determine all possible properties endpoint can return and all the possible parameters it can receive. (WordPress foundation 2018 E)

Fifth and last key concept of WordPress REST API are the Controller Classes. Controller Classes allow you to manage and register endpoints and routes. They also handle requests, utilize schema and generate responses. (WordPress foundation 2018 E)

2.2.2 Native post types and custom post types

WordPress calls different types of content with a general term post type. There are default post types and custom post types. (WordPress foundation 2018 C)

Default post types are native to the WordPress, in other words the post types that are shipped with standard WordPress installation. These default post types are posts (articles type content), pages (static pages), attachment (references to uploaded files), revisions (past versions of posts and pages), navigation menu items, custom css (user created custom css rules saved to database by customizer feature), changesets and user data requests (same as autosave, but for customizer options). (WordPress foundation 2018 C)

Custom post types are content types developer can create using register_post_type() function. Developer can pass numerous arguments for this function to configure exactly the kind of post type that is required for the purpose. (WordPress foundation 2018 C)

2.2.3 Custom fields and Advanced Custom Fields

By default, WordPress post has a certain fields like title, post-name (slug), content etc. Often you will need more fields to keep your data usable and clean. Especially when doing headless WordPress development dividing your content to more structural pieces of data is essential. WordPress has a concept that allows developers to create custom fields which are then stored to the post_meta table in database. (WordPress foundation 2018 D)

Most WordPress developers prefer to use popular Advanced Custom Fields (ACF) plugin to create and manage custom fields. It is basically an industry standard.

What ACF does it gives a pleasant user interface to handle post meta in WordPress admin. Developer can choose which fields to show, in which posts, how they should look like and what they should save to the database. There is also a paid pro version with even more features and control over the data (Advanced Custom Fields 2019)

2.3 JavaScript

JavaScript is determined by Mozilla as "a lightweight, interpreted, or just-in-time compiled programming language, prototype-based, multi-paradigm, dynamic language, supporting object-oriented, imperative, and declarative (e.g. functional programming) styles". It is traditionally known as a scripting language for web browsers, but in these days, it can be used as a server-side programming language or even as a language to create desktop software using runtime environments like Node.js. (Mozilla 2019)

JavaScript is only programming language that browsers can interpret, since HTML and CSS are not programming languages, they are documents and stylesheets. (Wikipedia 2019).

Standards for JavaScript language are called ECMAScript. Current sixth major version is officially called ECMAScript 2015, but it is more commonly known as ECMAScript 6 or just ES6. (Mozilla 2019)

2.4 React.js

2.4.1 Basics

React.js is a JavaScript library, developed by Facebook, for creating user interfaces. React allows developer to create declarative views to each state of the application and efficiently update the view when data is changed. (Facebook Inc 2019 A)

React is component based, which allows you to track state of each component separately and re-use components in multiple views. React component logic is purely written in JavaScript. There are no html templates so there is no need to touch dom. This makes tracking state of your data flow more efficient and less sensitive to errors. (Facebook Inc 2019 A)

React is usually rendered in browser using React DOM, but you can also render it server side using Node.js (as we will do later with Gatsby) or use it to make native mobile applications with React native. (Facebook Inc 2019)

2.4.1 Functional components

Most simple way to create a React component is a JavaScript function that accepts a single argument named 'props' which is an object of data you pass to component. Functional component also returns React element. If props change functional component will rerender. (Facebook Inc 2019)

Example of functional component see Appendix 1 (Facebook Inc 2019):

2.4.2 Props

Props stand for properties and they are the primary way in React to pass data from one component to another. This data flow is one way, from parent to child component. (Facebook Inc 2019 B)

When you render a component as a JSX element you can give it any attributes you want. When React sees user defined component has attributes it makes them object named props. This object is available as an argument to your component. So, if in your parent component you would pass child component attribute name, it would be accessible from child component in props.name. (Facebook Inc 2019 B)

Example of giving an attributes to React component in parent component see appendix 2 (Facebook Inc 2019 B).

Example of using property name in child component see appendix 3 (Facebook Inc 2019 B).

Functional components are also commonly known as dumb components. They just render whatever is passed to them. You can add logic for the rendering, but there is no state management or lifecycle hooks. (Facebook Inc 2019 B)

2.4.3 Class components, state and lifecycle methods

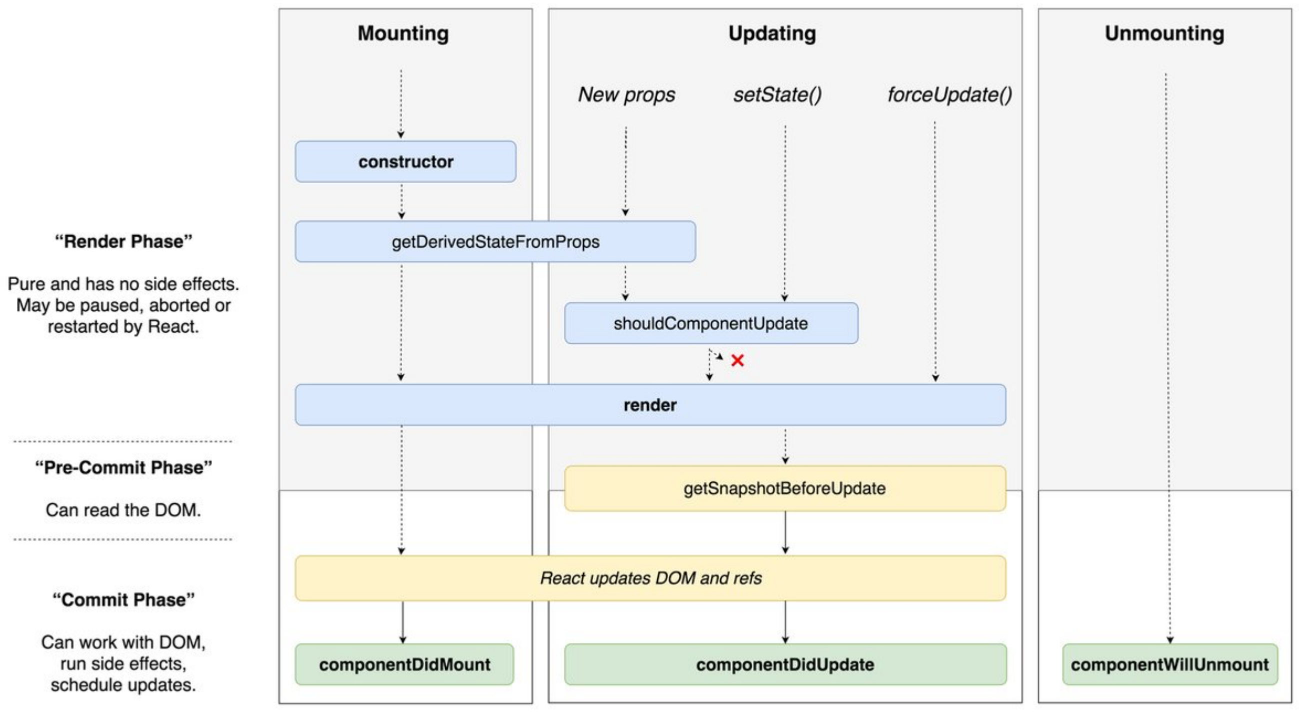
Another way to create React component are JavaScript ES6 classes. Class components have more features than functional components. They have component state and lifecycle methods. They are created so that they will extend React.component class and they have separate render method (also a lifecycle method). (Facebook Inc 2019 B)

Example of creating a class component see appendix 4.

To create a component state, you add class constructor. For constructor you should always pass props and call super function. Inside constructor you can determine your initial state. Whenever the state changes component will update. You can update state with 'setState()' method. State is immutable so you can't mutate it, you always set a new one. (Facebook Inc 2019 B)

Lifecycle functions are used to run certain pieces of code at the certain points of components lifecycle. Lifecycle hooks are special React methods that allow you to update the UI and application state. For example, when component is mounted, updated and when it is unmounted (Do 2018).

Following diagram shows when different lifecycle methods will run (Do 2018).



Here is an example how to use state and lifecycle to create a simple clock component. In this component you will first set your state to have a date property. Initial value is the moment component is loaded. Then you use setInterval JavaScript method inside componentDidMount (meaning this happens when everything is loaded). You set interval to be one second and run your own tick method. Then you will create the tick method that sets a new state every time it's run and set the date object to include that moment. Then you render your JSX and in where you display your state and use toLocaleTimeString method that returns only the time portion of this date. In the end you will also make sure your functions stop running after your component is unmounted by using clearInterval method inside componentWillUnmount lifecycle hook. (Facebook Inc 2019 B)

Example of using class to create a clock component see appendix 5. (Facebook Inc 2019 B)

2.4.4 JSX

All web pages are based on three different languages browser understands. These are html, css, and JavaScript. Special thing with React is it adds a new concept to this mix called JSX. JSX is basically way to write HTML like markup inside JavaScript. (Chinnatmambi 2017, loc 382)

Web browser does not know what JSX is, but it will work either by letting React JavaScript library to convert it on the fly (unefficient and only used during development), or run a production build using React build tools. In both cases JSX is converted to plain html, css and JavaScript browser understands. (Chinnatmambi 2017, loc 388)

JSX rendered by passing it to ReactDom.render method as an argument. The second argument is the DOM element you wish your application to be rendered. (Chinnatmambi 2017, loc 457-471)

2.4.5 Virtual DOM

DOM stands for Document Object Model. DOM is a tree like data structure that represents user interface of web application and changes in application state change DOM. Individual elements in HTML are nodes in DOM tree. When application state changes these nodes are updated. After representation of DOM tree is changed whole new tree needs to be rendered and painted. This re-render part of the process and doing this frequently can decrease application performance significantly. (Hamedani 2018)

Virtual DOM is a virtual representation of the real DOM. How it works is that when state of applications changes, a new virtual DOM is created compared to previous virtual DOM. This process is called diffing. Then application calculates what is the most efficient way to update real DOM. So, what really happens is that application only renders updates to nodes that have changed instead of re-rendering entire DOM tree. This is significantly more efficient than standard way of updating DOM. (Hamedani 2018)

In React every UI piece is a component with state. React watches any changes to these states and updates virtual DOM accordingly, by comparing new and previous state. Once React knows which DOM objects have been changed it only updates those objects to real DOM. (Hamedani 2018)

2.5 Gatsby.js

2.5.1 What is Gatsby

Gatsby.js is an open source JavaScript framework, based on React. Main focus of Gatsby.js is to give developers tools to create fastest possible websites and web apps. (Gatsby Inc. 2019 A)

2.5.2 Basics

First main feature of Gatsby is that pages are prebuilt as a 'static' website and then applications continues its life as a React application. To make sure only the data needed is loaded Gatsby uses GraphQl to load the data from source. Gatsby only loads critical HTML, CSS, data and JavaScript so website is as fast as possible. Once page is loaded Gatsby prefetches resources for linked pages so clicking around the site feels fast. (Gatsby Inc. 2019 B)

2.5.3 What is included?

Gatsby includes out of the box following features (Gatsby Inc. 2019 B):

- Static site generator
- Offline access
- Prefetching linked pages
- Page caching
- No extraneous code fetching
- Progressive image loading
- Responsive image loading
- Inlining of the critical CSS
- Font self-hosting
- Serverless

- Export as code
- Hot reload content
- Hot reload code
- Componentization
- One-way data binding
- Declarative API data queries (GraphQl)
- Declarative UI
- Asset pipelines
- CSS Extensions (SaSS)
- Advanced JavaScript syntax
- React component ecosystem

2.5.4 Gatsby plugins

Gatsby plugins are Node.js packages that use Gatsby Application Programming Interfaces (API). Plugins can for example add data sources, transform data to other formats and add third party services. Gatsby.org has a plugin library that includes many ready-made plugins made by either core Gatsby team or third parties. (Gatsby Inc. 2019 C)

To install plugin for Gatsby project, developers use node package manager (NPM) on their UNIX terminal and run command npm install. Then in gatsby-congif.js file developer adds plugin to plugins array. Here plugins can also take options. (Gatsby Inc. 2019 D)

Example of plugin options see appendix 6

If third party functionality that is required to add via npm package does not require connection to Gatsby APIs and it follows general JavaScript or React.js patterns there is no need to have Gatsby plugin. General rule is that if you would use package without Gatsby you can use it with Gatsby without making it a plugin. Idea of the plugin system is to give third party packages integration to core Gatsby APIs. (Gatsby Inc. 2019 E)

2.5.5 Gatsby-Source-WordPress

Source plugins are special type of plugins that work within Gatsby's data system. As the name suggest they source data from different locations, either local or remote. Data is then turned into what Gatsby calls nodes and node fields. Node fields represent single piece of data inside one node. Then these nodes can be accessed via GraphQl query. (Gatsby Inc. 2019 F)

Gatsby-source-wordpress plugin is developed and maintained by Gatsby core team. It can pull data either from self-hosted WordPress sites or WordPress.com. It includes OAuth authentication to WordPress.com API. Plugin also allows to query responsive images. (Gatsby Inc. 2019 G)

Plugin supports all entities that are available on WordPress REST API such as posts, pages, tags, categories, media, types, users, statuses, taxonomies, site metadata and custom posts types. Also Advanced Custom Fields (ACF) entities and Polylang and WPML language information are supported as well as other post meta that you have registered to REST API. (Gatsby Inc. 2019 G)

Plugin allows developer to choose which routes to fetch. By default, it fetches all endpoints of wp-json. (Gatsby Inc. 2019 G)

Plugin allows developer to fetch data with GraphQl using following pattern. If manufacturer is not specified endpoint is considered as a WordPress core endpoint. (Gatsby Inc. 2019 G)

Example of GraphQl query using Gatsby-source-wordpress see appendix 7.

2.5.6 Pages

Pages in Gatsby are objects created by createPage function. Page object has following properties: path, matchPath, jsonName, component, componentChunkName, internalComponentName, context and updatedAt. (Gatsby Inc. 2019 H)

Redux Pages namespace contains a map from page path to page object. This is updated automatically when createPage function is called as is components Redux namespace. Latter is a map of file which includes a React component. Also, when createPage runs onCreatePage API is executed. This allows plugins to hook on this event and do actions. (Gatsby Inc. 2019 H)

Pages can be created automatically by creating a React component under folder src/pages. When React components are added to this folder createPage function is automatically executed for those components and paths for routing are derived from folder structure and file names. (Gatsby Inc. 2019 H)

Pages can be created also programmatically which we go through in next chapter.

2.5.7 Templates and how to create pages programmatically

Gatsby-source-wordpress will provide you with slug for all posts and other entities, so all developer needs to do is to create page calling createPage function. This is done in Gatsby-node.js file typically by first running query for the data source and then calling createPage for each found node and setting a template file to be used as component. (Gatsby Inc. 2019 K)

Creating a page programmatically example see appendix 8.

Note that actual data fields for individual posts are queried in template file. In Gatsby-node.js file you only query fields needed for programmatical creation like slugs, id, and whatever meta developer happen to need for sites page creation logic. For example, there could be template meta field in WordPress to that would determine here which template in Gatsby is chosen. (Gatsby Inc. 2019 K)

2.5.8 Routing

When you create pages you also automatically create routes. This is done via createPage functions properties and saved to Redux pages namespace. When you create pages by creating them to src/pages, folder names are set to parents in path and file names are set as a slugs. When you create pages programmatically in gatsby-node.js you set path logic to path property of page object while running createPage function. (Gatsby Inc. 2019 M)

Example of createPage function see Appendix 9.

To link between routes in your site developers can use Gatsby-link built in library. Besides hooking to internal routing logic of Gatsby it also prefetches data for linked pages. (Gatsby Inc. 2019 L)

Example of using Gatsby-link see Appendix 10.

2.6 Continuous Deployment and Application Release automation.

"Continuous Deployment is a software development practice in which every code change goes through the entire pipeline and is put into production, automatically, resulting in many production deployments every day" (Electric Cloud 2019).

Usually this is achieved via automation that listen changes in master branch of versions control and then hooks to some kind of CI/CD automation system. So, deployment of application is automated. This is called Application Release Automation (ARA). (Electric Cloud 2019)

ARA includes four key functions. (Gartner 2019)

- Deployment of data, application code and artifacts
- Deployment of specific configurations for each environment
- Process workflow design for automating tasks, deployment steps, and people
- Environment modeling and/or provisioning binaries

Since Gatsby builds static sites it is kind of necessary to set up this kind of pipeline so that when content is updated in WordPress it is also updated in Gatsby site. Typically, Continuous Deployment is triggered only when code changes, but in this case, we would also want to trigger it when data changes.

2.7 GraphQl

GraphQl is a Query language for APIs and a runtime environment to fulfill those data request. We will focus the query language side in this thesis since, we won't be creating our own GraphQl server. GraphQl is based on idea that you can describe the API exactly the data you want, and you will get just that and nothing more. This makes it more efficient than REST. (Facebook Inc. 2019 C)

To query data from GraphQl API you define a query that kind of resembles a JSON object and when you send that query to GraphQl server you will receive back JSON object. It works by asking for specific fields inside an object. So, this means that query and result really resemble each other. (Facebook Inc. 2019 E)

Example of GraphQl query and query result see appendix 11.

In GraphQl you can also pass arguments to fields. You could for example specify you only want an item with certain ID, or you want a field to return height in centimeters or foots. (Facebook Inc. 2019 C)

GraphQl also has a feature called fragments. This allows you to reuse set of fields in more complicated queries. You can also use variables in fragments and in queries anywhere. (Facebook Inc. 2019 E)

Newer feature of GraphQl is directives. These can be used to include or skip field if given argument is true or false. (Facebook Inc. 2019 D)

So, what is the big difference between GraphQl and REST? Both fetch data using HTTP Get requests. Both receive data usually in JSON format. In REST type of the resource and way you fetch it are coupled and in GraphQl they are separate. In GraphQl you can affect what in what kind of form you get your result in query. You can also choose what specific fields you want. This gives much more flexibility in client end of application. In REST shape and size of a quarriable object is determined in server. (Stubailo 2017)

Big difference is also endpoints. GraphQl you typically have one endpoint and you determine in your query what you want from it. In REST you would have a multiple related endpoints (basically list of resources available) and you would query all of them separately. (Stubailo 2017)

3 How to build site with WordPress and Gatsby.js

3.1 Introduction

This is a short guide on how to create websites using WordPress as a backend and Gatsby.js as a frontend.

3.2 Set up WordPress as a headless CMS

If you are just planning to query native post types with native fields, you don't need any configuration. But in most cases, you want to have some extra fields and maybe some extra content types. You might also want to close some endpoints, like the user endpoint.

To close an endpoint, you need to use 'rest_endpoints' filter hook. First you will call 'add_filter', pass it a hook mentioned and a callback function. Then you check if endpoint you want to disable is set and disable it using unset function. You could also use regular expressions here.

Example of closing user endpoints in WordPress REST API: see appendix 12.

If you wish to add new content type to WordPress and add it to REST API, all you need to do is create content type and determine in arguments 'show_in_rest' to be true.

To register a post type in WordPress you use 'init' action hook and pass it a callback function. There is a lot of arguments available for 'register_post_type' function. More details on available arguments can be found in WordPress documentation. One special type of arguments are labels which determine how things are called in your backend. After you have wrote all the arguments you call 'register_post_type' function. This function requires two parameters: name of your post type as a string and your arguments as an array.

Example of registering a post type in WordPress and including it to REST API: see Appendix 13.

To add more fields, you could just register meta fields in code, but most practical and industry standard way is to use Advanced Custom Fields plugin (ACF). Using ACF happens in graphical user interface (commonly known as GUI) and fields can be synced between environments via generated json files. From a point of view of headless development, it's important to note that ACF fields are not automatically part of WordPress REST API. You need to add another plugin called ACF to REST API. This plugin does not require any configuration. Activation of this plugin will expose ACF fields to all REST endpoints.

One last setup we need to do to get our WordPress ready for headless use is to deal with featured images. By default, REST API returns only featured image ID with your post, so you would need to do multiple queries in your front end to get image URL. This is a problem you should solve at your backend. You can add the image URL to your post fields in REST API by calling 'register_rest_field' function. As parameters it takes posts types you want to add this field, name you want to give the field in REST API and array of arguments, where you need to specify your callback function to get content for that field. In your callback you use 'wp_get_attachment_image_src'-function to find your image url. Action hook to call is 'rest_api_init'. (Nguyen 2018.)

Example how to make WordPress REST API return featured image URL see Appendix 14.

3.3 Set up Gatsby.js

3.3.1 Install CLI

Gatsby Command Line Interface (CLI) is a tool to quickly create Gatsby powered projects from basic boilerplate. To install it you need to make sure you have Node.js and node package manager (npm) installed. NPM comes with Node.js installation package, so you only need to install Node.js. Then simply run ‘npm i -g gatsby-cli’ in your terminal and Gatsby CLI will be installed to your computer globally.

3.3.2 Create a new Gatsby project with gatsby-source-wordpress

Once you have CLI installed then you can create a new project by typing ‘gatsby new your-project-name’ to your terminal. Then you can go to your project folder by typing ‘cd your-project-name’ and install source plugin by typing ‘npm install --save gatsby-source-wordpress’. For source plugin to work you need to resolve and configure it in gatsby config file.

3.3.3 Gatsby config

Gatsby config is the main configuration file of your site. This is where you configure all the plugins you want to use at your site. Most important of these are your source plugins, since this is where you determine the routes you are fetching from the backend. In this case source plugin is gatsby-source-wordpress.

This file is read by node.js runtime, so you have to use node module export for your configuration object.

On ‘siteMetadata’ object you can insert basic information about your site such as name and description.

On plugins array you can choose and configure your plugins. Remember that all these plugins also need to be required as a npm packages at your package.json file.

First plugin you should always have is gatsby-plugin-react-helmet. This is a Gatsby optimized version of react-helmet library. It is Search Engine Optimization (SEO) tool, to set all meta data. It is hard to think reason why you would not want to use it. This plugin does not require any additional configuration in this file. Actual metadata object is given as a prop to Helmet component which you will add to your JSX files.

Gatsby-transformer-sharp and gatsby-plugin-sharp are image optimization tools that allow you to query different image sizes. This is useful with supporting sources like WordPress. No configuration is needed for these plugins.

Gatsby-plugin manifest is related to progressive web apps. It provides necessary metadata to save your web application to home screen of iOS or Android device. You should change the data in configuration to match your sites information.

Most important plugin to configure is gatsby-source-wordpress plugin. In options you need to determine 'baseUrl' of your WordPress site, without protocol or trailing slashes. Then you need to choose which protocol your WordPress site uses (http or https).

Then unless you are using WordPress.com set 'hostingWPCOM' to false. Set 'useACF' to true, since you will probably need Advanced Custom Fields.

Set 'verboseOutput' to true to get more informative error messages during development.

Then choose which routes you want to include from your WordPress REST API. If you want to fetch everything just remove this option. To fetch posts choose '**/*/*/posts' and with same logic pages and custom post types (last part being post type slug).

If you want to use SaSS for styling just add gatsby-plugin-sass plugin.

Example of recommended starter Gatsby config file see Appendix 15.

3.3.4 Gatsby node

In gatsby-node.js file you can create pages programmatically. This is useful for example if you want to display your WordPress posts. I talk about posts in this chapter but in example code there is also query for pages, just to show that logic is identical, no matter what content type you are querying.

First in gatsby-node.js there is few basic libraries to require. Those should include bluebird, path and slash. Leave these as they are.

Then create a postsQuery as a JavaScript variable (const). Logic of querying is following. To use gatsby-source-wordpress start queryname with camelcased phrase 'allWordPress'. Then add post type name, in this case 'Post' with uppercase first letter. If you are querying custom post type add 'Wp' between allWordPress and post type name.

Inside your query object first level is edges. This means your individual posts. Then inside each of your edge there is a node (post object) and node has fields. For this simple gatsby-node file we only need id and slug. Fields to display you will query in template. In this file you just programmatically create nodes and set them a template file and routing logic.

Then you need to assign a function for createPages object. This function will take parameters of graphql and boundactioncreators.

From boundActionCreators you will get a property createPage (function) to same name variable by deconstruring it.

Then you should create a new promise and pass our previously created postsQuery to graphql function. Then check for errors and if there are no errors, choose a path to our template file to be used with these edges (posts) and save it to variable. Then for each edge call createPage function and pass an object as a parameter. Inside the object set a logic for path (routing). In this case it will be slash posts, slash and post slug from our query. To do this use template literal (notice backticks, not quotes). Then set a component these posts should use (set this to the path to template defined earlier). Then to give post a context define id to match id in WordPress. Then call resolve to return a promise object.

Example of gatsby-node file see: Appendix 16.

3.3.5 Package.json

After doing previous steps your package.json file should look like in example code. All you need to add is gatsby-plugin-sass by typing ‘npm install --save gatsby-plugin-sass’. This is just bare essentials for gatsby to run with WordPress, while using best practices like react-helmet. You can add more packages when you need them.

Example of package.json file see Appendix 17.

3.3.6 Gatsby local development

Once your project is installed you can run local development server with command ‘gatsby develop’. This will start development server with ‘hot reload’ (site updates every time you update your code). You can view your site in localhost gate 8000 (http://localhost:8000/).

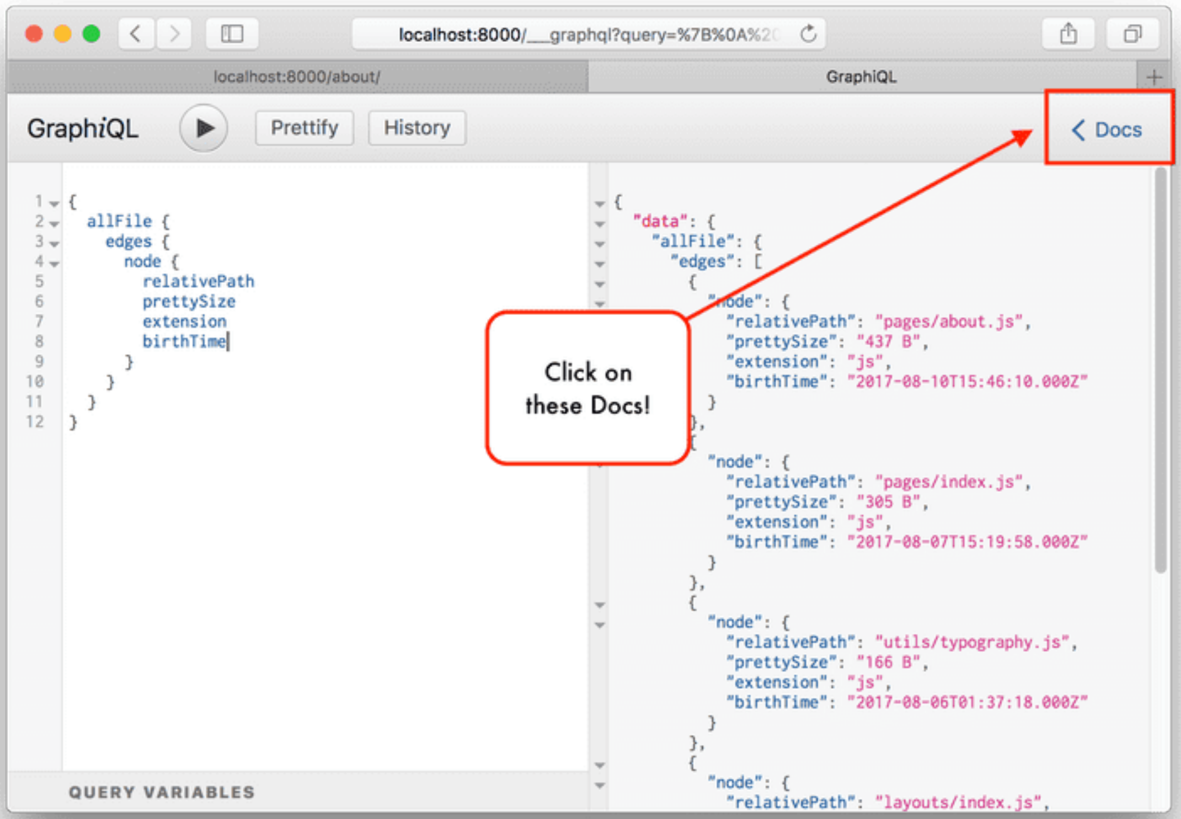
If you update your content in WordPress you need to restart your local server to get the updates, since when you run ‘gatsby develop’ gatsby will fetch all the data and it won’t do it again unless you restart the server. This does not mean that you could not update your queries. Gatsby will fetch all the data from your chosen endpoints with REST, so as long as the data was available there when you started the server, you can query it with GraphQl.

Note also that since development environment does not do server-side rendering, but runs on webpack, some browser specific code that works in development environment may not work when you run build for production. This happens only if you have code in your project that is not compatible with server-side rendering. So, I would recommend testing run for ‘gatsby build’, always before committing new code to master branch. It is good to know that if your code fails it will fail in build and never actually end up in production to cause runtime errors.

3.3.7 GraphiQl – built in tool for testing your queries

To test your queries before adding them to your code and to browse what data your endpoint has and in what format you can use handy tool shipped with Gatsby called GraphiQl explorer. This is not created by Gatsby, but it is included in core package of Gatsby.

To use it go to address http://localhost:8000/___graphql. Then write your query to left and press enter, and your result comes to right. On right corner by clicking the button ‘docs’ you can also click and browse the structure of your endpoint.



```
{
  allFile {
    edges {
      node {
        relativePath
        prettySize
        extension
        birthTime
      }
    }
  }
}
```

```
{
  "data": {
    "allFile": {
      "edges": [
        {
          "node": {
            "relativePath": "pages/about.js",
            "prettySize": "437 B",
            "extension": "js",
            "birthTime": "2017-08-10T15:46:10.000Z"
          }
        },
        {
          "node": {
            "relativePath": "pages/index.js",
            "prettySize": "305 B",
            "extension": "js",
            "birthTime": "2017-08-07T15:19:58.000Z"
          }
        },
        {
          "node": {
            "relativePath": "utils/typography.js",
            "prettySize": "166 B",
            "extension": "js",
            "birthTime": "2017-08-06T01:37:18.000Z"
          }
        },
        {
          "node": {
            "relativePath": "layouts/index.js",
```

(Gatsby 2019 O)

3.3.8 Build your first components

Before building your first page it is smart to first do few basic components. We can start with doing a heading component and a content component. These will be purely functional components that just render the content given to them as a property (prop).

This is probably simplest component you can imagine, a heading. Only thing this component does is to render a string from prop named heading inside a h2 tag. Create a folder inside a components folder, called ‘heading’. Inside that folder create an index.js file and style.scss file. Using index.js is a React naming convention and it will allow you to only refer to your folder and React will find your component inside the folder, when you import it.

In index.js file, first import dependencies. You can use prop-types to give our component type check and also a default value. If you want to use this, import the prop-types library. Then import React itself and lastly stylesheet for general heading styles.

Then define component as a function and assign it to constant ‘Heading’. Remember that React components should always have a capitalized first letter in their name. This tells React that they are not HTML elements but components. Pass a heading prop from our props to this functional component, by using curly braces. If you don’t use curly braces, the object you will get is full props object with all the props. When you use curly braces only heading is deconstructed from props object.

Our function is an arrow function that returns h2 element with class name of articleHeading and string of text inside the element from heading prop. Note that in JSX, CSS class needs to be defined with ‘className’ keyword, since ‘class’ is reserved word in JavaScript.

Then set up a prop-type check to make sure heading prop has to always be a string and assign a default value of empty string to it. Then export component as a default export (this means that by importing without curly braces to specify what to import from this file, it will result this function to be imported).

Then write some styles to style.scss and you can see that they will work too, since we have gatsby-plugin-sass in our package.json and it will compile SCSS automatically and use it as long as it is imported to any rendered component.

Example heading component and related styles see: Appendix 18.

Next component to build is almost as simple but with a little React specific functionality. This is going to be content component that renders the HTML WordPress returns to us. It is otherwise the same as previous component except the naming and the fact that the content we receive from WordPress will be HTML instead of plain text and all the HTML tags have to be interpreted. For this we will use dom element attribute called 'dangerouslySetInnerHTML' that has been built in to React. We will set this as an attribute of div element and pass it an object that includes another object where we set '__html' to be our content prop value.

Example content component see: Appendix 19.

Next component to create is date parser component. First import React as a dependency. Then create a function that takes props as a parameter.

Then take date property from props object and use it to create a date object. Then use 'getDate' JavaScript method to get day from it, 'getMonth' to get month, and 'getFullYear' to get year.

Then we return span element that includes day, month and year in desired format and finally export our function as a default export.

Example 'dateParser' component see: Appendix 20.

3.3.9 Build your first page

This example is a simple frontpage that displays the content and headline of a specific WordPress page with a slug 'frontpage'. For this to work you have to have this page in WordPress. Start by creating an index.js file to pages folder. Also create a style.scss file to the same folder.

First import dependencies using ES6 import syntax. General best practice is to put first native packages, then third party packages and then your own components, styles and helper functions.

From native packages you will need to import React itself and GraphQl from Gatsby. Import also Ramda.js as a third part helper library. You have two components you created earlier: Heading component and Content component. Also import stylesheet file.

Then at the bottom of file create a GraphQl query. Export it as constant named pageQuery (this name of the variable does not need to be unique). Then define query as a frontpageQuery. Note that this name of the query, needs to be unique. To get pages we need to call allWordpressPage as gatsby-source-wordpress syntax showed earlier. Pass an argument filter to query and filter it to return only pages which have a value of the field slug to equal 'frontpage'.

Here is a list of GraphQl operators (Gatsby Inc. 2019 N):

- eq: is equal
- ne: is not equal
- regex: is for regular expression, must match the given pattern
- in: is in array
- nin: is not in array
- gt: is greater than
- gte: is greater than or equal
- lt: is less than
- lte: is less than or equal

Then if you pass filter ‘eq: frontpage’ query will filter to return only edges that match this request. Include following fields inside the nodes: title, content and slug. This will return array with one object.

Then create functional component. Between imports and query place a function IndexPage, pass it a data object as a parameter and export it as a default export.

Inside this function get page from our array using Ramda.js function ‘head’. This will return first item of array in secure way from path data.post.edges. Then we take node object from the resulting object and assign it to a variable page.

Then return what we want to render. Remember that anything you return in React needs to be inside single wrapper element. In this case use div and give it a class name of jstFrontpage. Inside this div pass title to Heading component that will render it and content to Content component that will render them.

Now if you save and go to view localhost:8000/ this content should be rendered.

Then you can write some styles to style.scss and make sure that they will work too.

Example page and related styles see: Appendix 21.

3.3.10 Build your first template

To build your first template let’s assume you have a content type in WordPress called cv and you have created pages representing these cv posts in your gatsby node. Let’s also assume you have layout component to wrap all pages and a SEO component that comes with gatsby starter when you create one in CLI. You also have your DateParser component, Heading and Content components and a stylesheet.

First thing to do is to import dependencies like React, graphql and your components. Then create a function since there is no need for class and pass data from props to it. Data includes a result of your graphql query. Then you get post from data object. Before rendering anything, you should write your query.

This query could be named for example currectCvQuery. This could be anything. You should also pass id we determined in gatsby-node to it and define its type of string. Then use gatsby-source-wordpress keyword wordpresssWpCv which refers to custom WordPress post type that has slug cv. Then ask it to match post id with id of our parameter id. Then ask fields needed.

Then to return what you want to render let’s first wrap whole thing to a div element, since like mentioned earlier any React component must be wrapped inside single element or JSX won’t work. Then use SEO component by passing it post title as a title and post excerpt as a description. Then wrap visible part of render inside layout component. Then render the data you want to render. For date use your Dateparser component, since otherwise it won’t render in any reasonable format.

Finally export the template function.

Example template see: Appendix 22.

3.3.11 Gatsby build

To build your static site ready for production you can just run 'gatsby build' command on your terminal. This will run all queries on Node.js and create your site as a set of static pages and highly optimized assets in process called compiling. You can upload this static bundle to any static hosting platform, and it will work, without needing node.js or anything else, just pure HTML, CSS and JavaScript. Data is saved as JSON files for dynamic use after initial load. Every page gets their own HTML and JSON files.

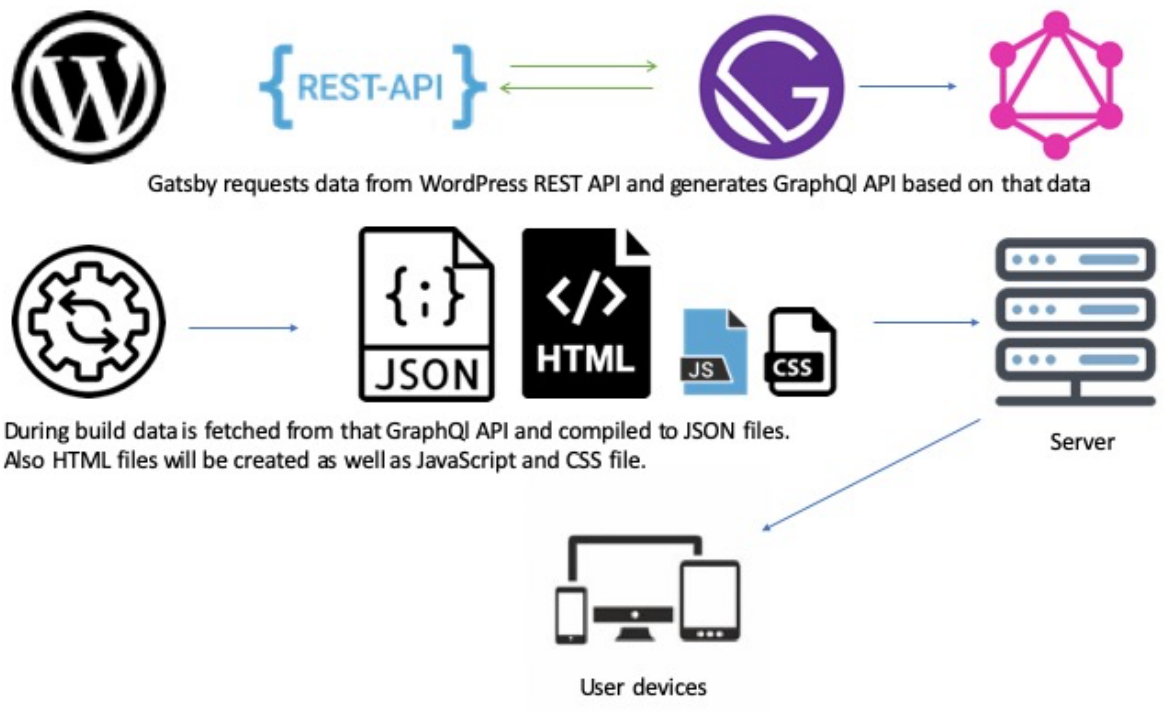

REST-API
Gatsby requests data from WordPress REST API and generates GraphQl API based on that data
JSON
HTML
JS
CSS
During build data is fetched from that GraphQl API and compiled to JSON files.
Also HTML files will be created as well as JavaScript and CSS file.
Server
User devices

Figure 1: Dataflow from WordPress to compiled site

In real life you probably want to automate this build and set up some continuous deployment which I will go through later, since otherwise your site does not update when your content updates. Good to remember that the command is still same even if it is automated and manual builds are good way to test your build does not give any errors, before merging new code to production branch.

3.3.12 How to query from WordPress using GraphQl and gatsby-source-wordpress

How gatsby-source-wordpress works is that on built it will first fetch everything on our endpoint using REST. Then it will generate internal GraphQl API based on that data. Then it will go through your queries and gather the data from that internal GraphQl API, so your build only ends up with data we asked for and nothing else.

During the development when you start your development server it will do the first part the same way and fetch all data and keep it as an object. Then when you change your queries during the development you always have the full data at your disposal and queries are done live. But if the data on the WordPress end changes, you need to restart local server because call to REST is only done when server starts.

First example is a page query. To start query you will first need to call graphql and put our query inside backticks. Then use keyword 'query' and give your query a unique name. Then choose a name of an object to save the result in. In the example it will be called 'post'. Then assign objects value to be whatever our query returns. Actual query starts by calling gatsby-source-wordpress with a keyword 'allWordpress' and right after that continuing with camelCase the post types name. If it is a post then you simply type 'Post', but in this example it is a custom post type campaign, so you need to add keyword 'Wp' between before the actual post type slug. So, we will type 'allWordpressWpCampaign'.

Then you will filter what pages you want. You should use slug field as a filter and ask it to equal 'frontpage'. Then ask response to return as edges, which is an array of nodes. From each node query title, content and slug. If you would also like some ACF fields these are in their own 'acf' object, providing that you have installed AFC to REST API plugin on your WordPress. Then inside this object you just specify which fields you want by using field slug.

Example page query from WordPress using Gatsby source WordPress see: Appendix 23.

That was an example about normal page query, but how to write a query for programmatically created pages? If you remember earlier, we created our posts programmatically in gatsby-node.js file. There we assign routing logic for them. If route matches the pattern `posts/${edge.node.slug}`, meaning that if slug of WordPress post is same as last part of our url path get that post. We also assign this node a context in form of an id. We also assign a template file for this node. So, when this node is loaded according to this, we need to query the post itself in template file. You could name this query currentCampaigns and tell it to except for context variable id, which should be a string. Then use keyword 'wordpressPost', not 'allWordpressPost', since you already loop through that in gatsby-node.js. This expression will expect you to tell which post you want. Determine that you want id to equal our id in context variable. Then simply specify the fields you want.

Example current post query from WordPress using gatsby-source-wordpress see: Appendix 24.

3.3.13 Using external libraries

You can use any Node package manager (npm) libraries that work with React, as long as they are compatible with server-side rendering (SSR). Those packages that are not compatible with SSR might work on your development, but when you do your actual build and things are rendered server side you will get an error. Typical problem causing this might be that package calls some DOM object without checking if it exists and of course when you render in server, you won't have DOM.

You can browse packages in npmjs.com and then install one by typing npm install --save package-name to our terminal in project root folder. For example, 'npm install –save react-helmet'.

3.4 Set up hosting with continuous deployment

3.4.3 GitHub

For continuous deployment to work you should have your code in some Git version control service, that offers access for third party apps. We will use GitHub, since it is the most common one, but also GitLab and BitBucket will work the same way.

3.4.4 Set up hosting and continuous deployment on Netlify

Netlify has buildt in continuous deployment and it is really easy to set up.

First you need to create a Netlify account. Go to www.netlify.com and simply press sign up button and sign in with your GitHub account.

Then add app on control panel and authorize Netlify to access the GitHub repository of your Gatsby site. Choose deployment from master branch and deployment command to be gatsby build.

Then press deploy site for initial deployment. You now have hosting and continuous deployment done halfway. Your site will now automatically build when master branch is updated.

What you still really need is a deployment when WordPress content update. Go to deployment settings and choose continuous deployment. Go to webhooks and add new hook. You can name it for example WordPress content update.

Go to your WordPress code and create new file netlify.php in your inc folder and include that file at your functions.php file so it will be run.

In netlify.php file add the code in example. This code will run function called save_any_post_type, whenever WordPress hook save_post is run. Basically, every time any content is saved. Then inside function you can use WordPress built in http function wp_remote_post and give it Netlify hook (one you just created) URL as a first parameter, and as a second parameter some basic http request options (array). These options are http method which should be post, timeout so built is not restart too often accidently, http version, blocking set to true and headers determining that will send and receive in JSON format. Lastly do error login if something goes wrong.

Now if you deploy your WordPress changes and try to change any post it should start a new build on Netlify.

Example of save-post build hook (Netlify version) see: Appendix 25.

3.4.5 Using Travis CI to set up continuous deployment on other hosting providers

I have published this chapter before submitting this thesis at my blog: http://juha.blog.

In this example I will host my site in Firebase hosting. First, you need a Travis account. If you have a non-commercial project with public GitHub repo you can use free Travis.org. For commercial projects with closed GitHub repos you should use paid Travis.com. Here are the steps to authorize the account in GitHub for both cases.

- Sign up with your GitHub account and give Travis access to your repositories
- Activate Travis for repository you want to use

Since Travis CI is first and foremost meant for testing it tries to run tests first and won't let you do the build if it fails. So, first if you don't have tests, you need to do a little change in our package.json file. Gatsby ships with test script that throws an error and complains you don't have any tests. To make this pass in Travis you need to make our test script look like this (so it gives warning, not error). Also add deploy script with firebase deploy to your scripts.

Example of scripts in package.json file see Appendix 26.

Then the most important part. Add your '.travis.yml' file. This is almost all your Travis configuration. In Travis UI you basically just add your Firebase auth token, which you are referring to in our file. Add this to the root of your repo.

Example includes simple working boilerplate about travis.yml. You can add more options and scripts according to your needs. Most of the options are build environment configs but part that is interesting is at the end. You can specify before build scripts (in this case install firebase tools and gatsby), actual test and build scripts and after successful build after scripts (in this case deploy build to firebase).

Example of .travis.yml file see: Appendix 27.

Then create Firebase token and do a test commit. To make a Firebase token you have to have Firebase tools installed. Creating a token is simple. Just run this command. Type 'firebase login:ci' to your terminal. It forces you to log in to your Firebase account and then gives you a token. Then you go to Travis web UI and put in as an env variable with the name FIREBASE_TOKEN.

Then do a commit and push it to your master branch and check in Travis CI web interface is the build starting.

WordPress is full of webhooks, so this part is not difficult. Just add this script in example to you functions.php and edit it to point to your project. This hooks to save_post hook so every time any post is saved this will run. First, you get your post type using get_post_type function. Then check is this one of the post types you want to launch our request. If not, kill the script there. Then if it is do post request. I suggest using WP built in wp_remote_post function every time you need to send external http request in WordPress, since it's makes your code easy to read, edit and understand.

Note that in our http request URL has one odd detail. Between repository owner and repository name there needs to be %2F instead of /. Next chapter will tell how to get Travis token in code.

Example of save-post build hook on WordPress (Travis version) see: Appendix 28.

As you noticed reading the code you will need Travis access token to send request. You will get it like this. Put –com or –org depending are you using open source .org or commercial .com version. Remember you need to have Travis CLI tools installed.

Type to your terminal: 'travis token –com' and' travis token –com'.

4 Pros and cons of headless WordPress development with Gatsby.js

4.1 Pros

First clear benefit of this approach is performance. Having a static, prerendered site is the most efficient way to render a webpage. Also, fact that Gatsby uses GraphQl to only have the minimum amount of data needed gives some extra efficiency for the time after the initial load.

Second benefit of generating static pages is that it is easy for search engines to read. Everything it prerendered and indexable. This is a clear benefit specifically compared to other headless approaches where rendering pages with JavaScript is a problem in terms of search engine optimization (SEO).

Third benefit is relative easiness and speed of development process. I'm now comparing site building compared to other headless approaches. Main benefit here is that Gatsby has one unified, easy to understand way of fetching data regardless of source. This makes development rather easy since you can focus on your actual site and let Gatsby do most of the heavy lifting.

Fourth benefit is that you can host your Gatsby site anywhere since it's just serving static files. This saves hosting expenses. Most of the cases you can get working hosting for free, even if your site would be rather big. Since your WordPress is only called during build process and not at all during the user session, it can be placed to rather affordable hosting solution too.

4.2 Cons

First downside of this approach is that since site needs to be built every time when content changes from scratch it sets limitations how frequently your site can update. It can take up to ten minutes to run gatsby build if your site has lot of data. For a site that updates frequently this can be an issue.

Second downside is that delay. You have to wait several minutes to see changes in your content to go live.

Third downside is that unlike in traditional WordPress site, you don't have any kind of preview. When you update your content, you can't see beforehand how it's going to look in live site. In my professional life this has been the biggest issue content creators have found with Gatsby.

4.3 Conclusions

Like with any tech this is not a perfect solution. One should always consider the project specifications and goals when choosing a technology. If the emphasis is on scalability, performance, speed of development, long lifecycle and ability to make changes fast this is a good approach. If emphasis is to have as much flexibility and tools for non-programmer content creators as possible or content is updated in second or minute basis, this is wrong approach. Example of this kind of site could be a discussion site or some kind of social media site. For these standard React with Redux is better and WordPress is probably not the backend you want either.

4.4 Lessons learned

When I started this thesis, I was not expecting to learn lot of new things, since this tech stack is rather familiar for me. I work mainly with React, Gatsby and WordPress daily. I have also made couple of sites with this particular combination of WordPress and Gatsby and countless others with different combinations including one of these.

I have always struggled to explain technical information for other people. This is evident with client interactions, presentations about some new technology and training new developers. All of this is part of my work so developing this aspect of my skills is essential. I think process of writing this thesis has least teach me something about communicating technical information to people who are not already expert in this particular field. For example, all the things I should explain about related topics before I jump into the main topic.

Second lesson learned came from need to deep dive how my tools work under the hood. I have previously known how to utilize Gatsby or React, but this thesis has forced me to deep dive to documentations and really understand how they work, what they do in the background and why.

References

BuiltWith Pty Ltd 2019. CMS Usage Distribution in the Top 1 Million Sites. URL: https://trends.builtwith.com/cms. Accessed: 22 March 2019

WordPress Foundation 2016. REST API Merge Proposal, Part 2: Content API. URL: https://make.wordpress.org/core/2016/10/08/rest-api-merge-proposal-part-2-content-api/. Accessed: 22 March 2019

WordPress Foundation 2018 A. Features. URL: https://wordpress.org/about/features/. Accessed: 3 May 2019

WordPress Foundation 2018 B. License. URL: https://wordpress.org/about/license/. Accessed: 3 May 2019

WordPress Foundation 2017. Plugin API. URL: https://codex.wordpress.org/Plugin_API/. Accessed: 3 May 2019

WordPress Foundation 2018 C. Post types. URL: https://codex.wordpress.org/Post_Types. Accessed: 10 May 2019

WordPress Foundation 2018 D. Custom fields. URL: https://codex.wordpress.org/Custom_Fields. Accessed: 10 May 2019

Advanced Custom Fields 2019. Getting started with Advanced Custom Fields. URL: https://www.advancedcustomfields.com/resources/getting-started-with-acf/. Accessed: 10 May 2019

Facebook Inc 2019 A. React JS. URL: https://reactjs.org/. Accessed: 25 May 2019.

Facebook Inc 2019 B. Components and props. URL: https://reactjs.org/docs/components-and-props.html. Accessed: 25 May 2019.

Nancy Do 2018. Understanding React 16.4 Component Lifecycle Methods. URL: https://medium.com/@nancydo7/understanding-react-16-4-component-lifecycle-methods-e376710e5157. Accessed: 25 May 2019.

ContentStack 2018. A History of Content Management Systems and the Rise of the Headless CMS. URL: https://www.contentstack.com/blog/all-about-headless/content-management-systems-history-and-headless-cms. Accessed: 22 March 2019

Coder Academy 2016. Top 32 Sites Built With ReactJS. URL: https://medium.com/@coderacademy/32-sites-built-with-reactjs-172e3a4bed81. Accessed: 22 March 2019

SimilarTech Ltd 2019. Market share & web usage statistics React JS. URL: https://www.similartech.com/technologies/react-js. Accessed: 22 March 2019

Developer 2016. Is React library or a framework?. URL: https://developer.com/is-reactjs-library-or-a-framework-a14786f681a0. Accessed: 22 March 2019

Gatsby Inc. 2019 A. Gatsby.js. URL: https://www.gatsbyjs.org/. Accessed: 22 March 2019

Gatsby Inc. 2019 B. Features. URL: https://www.gatsbyjs.org/features. Accessed: 22 March 2019

Gatsby Inc. 2019 C. Docs: Plugins. URL: https://www.gatsbyjs.org/docs/plugins/ Accessed: 9 July 2019

Gatsby Inc. 2019 D. Docs: What is a plugin. URL: https://www.gatsbyjs.org/docs/what-is-a-plugin/ Accessed: 9 July 2019

Gatsby Inc. 2019 E. Docs: Using a plugin in your site. URL: https://www.gatsbyjs.org/docs/using-a-plugin-in-your-site/ Accessed: 9 July 2019

Gatsby Inc. 2019 F. Creating a source plugin. URL: https://www.gatsbyjs.org/docs/creating-a-source-plugin/ Accessed: 9 July 2019

Gatsby Inc. 2019 G. gatsby-source-wordpress. URL: https://www.gatsbyjs.org/packages/gatsby-source-wordpress/ Accessed: 9 July 2019

Gatsby Inc. 2019 H. Page creation. URL: https://www.gatsbyjs.org/docs/page-creation/ Accessed: 10 July 2019

Gatsby Inc. 2019 J. Gatsby internal terminology. URL: https://www.gatsbyjs.org/docs/gatsby-internals-terminology/Accessed: 10 July 2019

Gatsby Inc. 2019 K. Programmatically create pages. URL: https://www.gatsbyjs.org/tutorial/part-seven/
Accessed: 10 July 2019

Gatsby Inc. 2019 L. Get to know Gatsby building components. URL: https://www.gatsbyjs.org/tutorial/part-one/ Accessed: 10 July 2019

Gatsby Inc. 2019 M. Routing. URL: https://www.gatsbyjs.org/docs/routing/ Accessed: 10 July 2019

Gatsby Inc. 2019 N. GraphQl reference. URL: https://www.gatsbyjs.org/docs/graphql-reference/ Accessed: 17 July 2019

Dave Nguyen 2018. How to get Featured Image from WordPress REST API. URL https://medium.com/@dalenguyen/how-to-get-featured-image-from-wordpress-rest-api-5e023b9896c6. Accessed 29 March 2019.

W3 Consortium 2004. 3.1.3 Relationship to the World Wide Web and REST Architectures. URL https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest. Accessed 9 April 2019.

WordPress Foundation 2018 E. REST API Handbook. URL: https://developer.wordpress.org/rest-api/. Accessed: 6 April 2019

Robert Mening 2018. Popular CMS by market share. URL https://websitesetup.org/popular-cms/. Accessed 3 May 2019.

Pearson Education Inc, Learning React (Kindle version), Kirupa Chinnatmambi 2017

Mosh Hamedani 2018. React virtual DOM explained in simple English. URL https://programmingwithmosh.com/react/react-virtual-dom-explained/
. Accessed 8 July 2019.

Electric Cloud 2019. What is continuous deployment. URL: https://electric-cloud.com/resources/continuous-delivery-101/continuous-deployment/. Accessed: 11 July 2019

Gartner 2019. Application Release Automation. URL: https://www.gartner.com/it-glossary/application-release-automation-ara/. Accessed: 11 July 2019

Sashko Stubailo 2017. GraphQl vs. Rest. URL: https://blog.apollographql.com/graphql-vs-rest-5d425123e34b. Accessed 14 July 2019

Facebook Inc. 2019 C. A query language for your API. URL: https://graphql.org/. Accessed 14 July 2019

Facebook Inc. 2019 D. Introduction to GraphQl. URL: https://graphql.org/learn/. Accessed 14 July 2019

Facebook Inc. 2019 E. Queries. URL: https://graphql.org/learn/queries. Accessed 16 July 2019

Mozilla 2019. JavaScript. URL:
https://developer.mozilla.org/en-US/docs/Web/JavaScript. Accessed: 1 August 2019

Wikipedia 2019. JavaScript. URL: https://en.wikipedia.org/wiki/JavaScript. Accessed: 1 August 2019

Gatsby Inc 2019 O. Introducing GraphiQl. URLhttps://www.gatsbyjs.org/docs/introducing-graphiql/. Accessed: 1 August 2019

Appendices

Appendix 1. Example of functional component (Facebook 2019).

```
function HelloName(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Appendix 2. Example of giving an attributes to React component in parent component (Facebook Inc 2019).

```
return <Welcome name="Sara" />
```

Appendix 3. Example of using property name in child component (Facebook Inc 2019).

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>
}
```

Appendix 4. Example of creating a class component (Facebook Inc 2019).

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Appenxid 5. Example of using class to create a clock component (Facebook Inc 2019).

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Appendix 6. Example of Gatsby plugin options (Gatsby Inc 2019).

```
module.exports = {
  plugins: [
    // Shortcut for adding plugins without options.
    "gatsby-plugin-react-helmet",
    {
      // Standard plugin with options example
      resolve: `gatsby-source-filesystem`,
      options: {
        path: `${__dirname}/src/data/`,
        name: "data",
      },
    },
  ],
}
```

Appendix 7. Example of using GraphQl query with Gatsby-source-wordpress

```
{
  allWordpress${Manufacturer}${Endpoint} {
    edges {
      node {
        id
       type
        // Put your fields here
      }
    }
  }
}
```

Appendix 8. Example of creating a page programmatically in Gatsby

```
exports.createPages = ({ graphql, boundActionCreators }) => {
    const { createPage } = boundActionCreators;

    return new Promise((resolve, reject) => {

                graphql(postsQuery)
                    .then(result => {
                        if (result.errors) {
                            console.log(result.errors);
                            reject(result.errors);
                        }
                        const postTemplate = path.resolve("./src/tem-
plates/post.js");

                      result.data.allWordpressPost.edges.forEach((edge)
=> {
                            createPage({
                                path: `/post/${edge.node.slug}/`,
                                component: slash(postTemplate),
                                context: {
                                    id: edge.node.id,
                                },
                            });
                        });
                        resolve();
                    });

    })

};
```

Appendix 9. Example of using createPage function in Gatsby

```
createPage({
                                path: `/post/${edge.node.slug}/`,
                                component: slash(postTemplate),
                                context: {
                                    id: edge.node.id,
                                },
                            });
```

Appendix 10. Example of using Gatsby-link

```
import { Link } from 'gatsby'
import React from 'react'


const NavComponent = () => (
  <nav>
    <Link
    to={`/`}
    className={this.props.current === '1' ? `current` : null}
    aria-label="Etusivu"
    >
    Etusivu
    </Link>

</nav> );
```

Appendix 11. Example of GraphQl query and query result

```
{
  hero {
    name
    # Queries can have comments!
    friends {
      name
    }
  }
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        }
      ]
    }
  }
}
```

Appendix 12. Disable endpoints, example code, closing all user endpoints.

```
add_filter( 'rest_endpoints', function( $endpoints ){
    if ( isset( $endpoints['/wp/v2/users'] ) ) {
        unset( $endpoints['/wp/v2/users'] );
    }
    if ( isset( $endpoints['/wp/v2/users/(?P<id>[\d]+)'] ) ) {
        unset( $endpoints['/wp/v2/users/(?P<id>[\d]+)'] );
    }
    return $endpoints;
});
```

Appendix 13. Example of registering a post type in WordPress and including it to REST API

```php
function register_cv_post_type () {

//Variables
$singular = 'Cv';
$plural = 'Cv';
$object = 'Cv';
$labels = array(
'name'                   => $plural,
'singular_name'          => $singular,
'add_name'               => 'Add new',
'add_new_item'           => 'Add new' . $singular,
'edit'                   => 'Edit',
'edit_item'              => 'Edit ' . $object,
'new_item'               => 'New ' .$singular,
'view'                   => 'Show ' . $singular,
'view_item'              => 'Show ' . $singular,
'search_term'            => 'Search ' . $object,
'not_found'              => 'Could not find ' . $object,
'not_found_in_trash'     => 'Could not find ' . $object . ' from trash’

);

$args = array(
'labels'     => $labels,
'public'     => true,
'show_in_rest' => true,
'publicly_queryable' => true,
'exclude_from_search'=> true,
'show_ui'              => true,
'show_in_menu'         => true,
'show_in_admin_bar'    => true,
'menu_position'        => 20,
'menu_icon'            => 'dashicons-id-alt',
'can_export'           => true,
'delete_with_user'     => false,
'hierarchical'         => false,
'has_archive'          => true,
'query_var'            => true,
'capability_type'      => 'post',
'map_meta_cap'         => true,
'taxonomies'          => array(
        'post_tag',
        'category'
        ),
'rewrite'             => array(
        'slug'           => 'cv',
        'with_front'     => true,
        'pages'          => true, //sivutus sallittu
        'feeds'          => false //rss sisällyttäminen
    ),
'supports'            => array(
            'title',
            'editor',
            'thumbnail',
            'excerpt',
            'custom-fields',
            'revisions',
    )
);


register_post_type('cv', $args );
}

add_action('init', 'register_cv_post_type' );
```

Appendix 14. Example how to make WordPress REST API return featured image URL

```
add_action('rest_api_init', 'register_rest_images' );
function register_rest_images(){
    register_rest_field( array('post'),
        'fimg_url',
        array(
            'get_callback'    => 'get_rest_featured_image',
            'update_callback' => null,
            'schema'          => null,
        )
    );
}
function get_rest_featured_image( $object, $field_name, $request ) {
    if( $object['featured_media'] ){
        $img = wp_get_attachment_image_src( $object['featured_media'],
'app-thumb' );
        return $img[0];
    }
    return false;
}
```

Appendix 15. Example of recommended Gatsby config file starter.

```
module.exports = {
  siteMetadata: {
    title: `Write title here`,
    description: `Write some description here
    author: `Juha Stenroos`,
  },
  plugins: [
    `gatsby-plugin-react-helmet`,
    `gatsby-transformer-sharp`,
    `gatsby-plugin-sharp`,
    {
      resolve: `gatsby-plugin-manifest`,
      options: {
        name: `YOURSITE.COM`,
        short_name: `yoursite`,
        start_url: `/`,
        background_color: `#663399`,
        theme_color: `#663399`,
        display: `minimal-ui`,
        icon: `src/images/favicon.png`,
      },
    },
    {
      resolve: "gatsby-source-wordpress",
      options: {
          baseUrl: "www.yourwordpresssite.com",
          protocol: "https",
          hostingWPCOM: false,
          useACF: true,
          verboseOutput: true,
          includedRoutes: [
            "**/*/*/posts"
          ],
      }
    },
    `gatsby-plugin-sass`,
  ],
}
```

Appendix 16. Example of Gatsby-node file

```
const Promise = require(`bluebird`);
const path = require(`path`);
const slash = require(`slash`);


const pageQuery = `
{
  allWordpressPage {
    edges {
      node {
        id
        slug
      }
    }
  }
}
`


const postsQuery = `
{
  allWordpressPost {
    edges {
      node {
        id
        slug
      }
    }
  }
}
`


exports.createPages = ({ graphql, boundActionCreators }) => {
    const { createPage } = boundActionCreators;

    return new Promise((resolve, reject) => {

      graphql(pageQuery)
      .then(result => {
          if (result.errors) {
              console.log(result.errors);
              reject(result.errors);
          }

          const pageTemplate = path.resolve("./src/templates/page.js");

          result.data.allWordpressPage.edges.forEach((edge) => {
              createPage({
                  path: `/${edge.node.slug}/`,
                  component: slash(pageTemplate),
                  context: {
                      id: edge.node.id,
                  },
              });
          });
      })

      .then(() => {

                graphql(postsQuery)
                    .then(result => {
                        if (result.errors) {
                            console.log(result.errors);
                            reject(result.errors);
                        }
                        const postTemplate = path.resolve("./src/tem-
plates/post.js");

                      result.data.allWordpressPost.edges.forEach((edge)
=> {
                            createPage({
                                path: `/post/${edge.node.slug}/`,
                                component: slash(postTemplate),
                                context: {
                                    id: edge.node.id,
                                },
                            });
                        });
                        resolve();
                    });

    })
```

```
});

}
```

Appendix 17. Example of package.json file

```
{
  "name": "gatsby-starter-default",
  "private": true,
  "description": "A simple starter to get up and developing quickly
with Gatsby",
  "version": "0.1.0",
  "author": "Kyle Mathews <mathews.kyle@gmail.com>",
  "dependencies": {
    "gatsby": "^2.13.45",
    "gatsby-image": "^2.2.7",
    "gatsby-plugin-manifest": "^2.2.4",
    "gatsby-plugin-offline": "^2.2.4",
    "gatsby-plugin-react-helmet": "^3.1.2",
    "gatsby-plugin-sass": "^2.1.4",
    "gatsby-plugin-sharp": "^2.2.9",
    "gatsby-source-filesystem": "^2.1.7",
    "gatsby-source-wordpress": "^3.1.13",
    "gatsby-transformer-sharp": "^2.2.5",
    "prop-types": "^15.7.2",
    "react": "^16.8.6",
    "react-dom": "^16.8.6",
    "react-helmet": "^5.2.1"
  },
  "devDependencies": {
    "prettier": "^1.18.2"
  },
  "keywords": [
    "gatsby"
  ],
  "license": "MIT",
  "scripts": {
    "build": "gatsby build",
    "develop": "gatsby develop",
    "format": "prettier --write src/**/*.{js,jsx}",
    "start": "npm run develop",
    "serve": "gatsby serve",
    "test": "echo \"Write tests! -> https://gatsby.dev/unit-testing\""
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/gatsbyjs/gatsby-starter-default"
  },
  "bugs": {
    "url": "https://github.com/gatsbyjs/gatsby/issues"
  }
}
```

Appendix 18. Example of heading component and related styles

```
import PropTypes from 'prop-types'
import React from 'react'
import './style.scss'

const Heading = ({ heading }) => <h2 className="articleHeading">{head-
ing}</h2>

Heading.propTypes = {
  heading: PropTypes.string,
}

Heading.defaultProps = {
  heading: '',
}

export default Heading
```

```
.articleHeading {
  margin: 0 auto 1.45rem;
  max-width: 660px;
}
```

Appendix 19. Example content component

```
import PropTypes from 'prop-types'
import React from 'react'
import './style.scss'

const Content = ({ content }) => (
  <div className="mainContent">
    <div dangerouslySetInnerHTML={{ __html: content }} />
  </div>
)

Content.propTypes = {
  content: PropTypes.string,
}

Content.defaultProps = {
  content: '',
}

export default Content
```

Appendix 20. Example DateParser component

```
import React from 'react'

const DateParser = props => {

  const date = new Date(props.date)
  const day = date.getDate()
  const month = date.getMonth()
  const year = date.getFullYear()

  return (
    <span>
      {day}.{month}.{year}
    </span>
  )
}

export default DateParser
```

Appendix 21. Example page and relates styles

```
import React from 'react'
import { graphql } from 'gatsby'
import * as R from 'ramda'
import SEO from '../components/seo'
import Heading from '../components/heading'
import Content from '../components/content'

import './style.scss

const IndexPage = ({ data }) => {
  const page = R.head(R.path(['post', 'edges'], data)).node
  return (
    <div className="jstFrontpage">
      <Heading heading={page.title} />
      <Content content={page.content} />
    </div>

  )
}

export default IndexPage

export const pageQuery = graphql`
  query frontpageQuery {
    post: allWordpressPage(filter: { slug: { eq: "frontpage" } }) {
      edges {
        node {
          title
          content
          slug
        }
      }
    }
  }
`
```

```
.jstFrontpage {
  max-width: 1080px;
  margin: 0 auto;
  background: whitesmoke;
  padding: 100px;
}
```

Appendix 22. Example template for Gatsby

```
import React from 'react'
import { graphql } from 'gatsby'
import Layout from '../components/layout'
import SEO from '../components/seo'
import Content from '../components/content'
import Heading from '../components/heading'
import DateParser from '../components/dateparser'
import './style.scss'


const Cv = ({ data }) => {
    const post = this.props.data.wordpressWpCv
    return (
      <div>
        <SEO title={post.title} description={post.excerpt} />
        <Layout>
          <div className="singlePost">
            <Heading heading={post.title} />

            <Content content={post.content} />
            <div className="articleDate">
                Last edited: <DateParser date={post.modified} />
            </div>
          </div>

        </Layout>
      </div>
    )
}

export default Cv

export const pageQuery = graphql`
  query currentCvQuery($id: String!) {
    wordpressWpCv(id: { eq: $id }) {
      title
      excerpt
      content
      modified
    }
  }
`
```

Appendix 23. Example page query from WordPress using gatsby-source-wordpress

```
export const pageQuery = graphql`
  query frontpageQuery {
    post: allWordpressWpCampaign(filter: { slug: { eq: "frontpage" } })
{
      edges {
        node {
          title
          content
          slug
          acf {
            order
            bgcolor
          }
        }
      }
    }
  }
`
```

Appendix 24. Example current post query from WordPress using gatsby-source-wordpress

```
export const pageQuery = graphql`
  query currentPostQuery($id: String!) {
    wordpressPost(id: { eq: $id }) {
      title
      content
      excerpt
    }
  }
`
```

Appendix 25. Example of save-post build hook in WordPress (Netlify version)

```
<?php

function save_any_post_type( $post_id, $post, $update ) {

  $response = wp_remote_post( 'https://api.net
lify.com/build_hooks/YOURHOOKIDCOMESHERE, array(
  'method' => 'POST',
  'timeout' => 45,
  'httpversion' => '1.0',
  'blocking' => true,
  'headers' => array(
    'Content-Type' => 'application/json',
    'Accept' => 'application/json'
  ),
));

if ( is_wp_error( $response ) ) {
  $error_message = $response->get_error_message();
  echo "Something went wrong: $error_message";
} else {
  return;
}

}
add_action( 'save_post', 'save_any_post_type', 10, 3 );
```

Appendix 26. Example of package.json file scripts

```
"scripts": {
  "build": "gatsby build",
  "develop": "gatsby develop",
  "start": "npm run develop",
  "format": "prettier --write \"src/**/*.js\"",
  "build:prod": "yarn build -- --prefix-links",
  "deploy": "yarn build:prod && firebase deploy",
  "test": "echo \"Warning: no test specified\" && exit 0"
},
```

Appendix 27. Example of travis.yml file

```
language: node_js
node_js:
  - "8"
cache: yarn
addons:
  apt:
    packages:
      - g++-4.8
    sources:
      - ubuntu-toolchain-r-test
env:
  CXX=g++-4.8
branches:
  only:
    - master
before_script:
  - "yarn global add firebase-tools"
  - "yarn global add gatsby"
script:
  - "yarn test"
  - "yarn build:prod"
after_success:
  - "firebase deploy --token=${FIREBASE_TOKEN}"
```

Appendix 28. Example of save-post build hook (Travis version)

```
function jst_save_posts( $post_id, $post, $update ) {

  $post_type = get_post_type($post_id);

  // If saved post isn't in a post type 'post', don't do anything
  // CHANGE THIS TO WHATEVER POST TYPE YOU WANT TO BE EFFECTED
  if ( "post" != $post_type ) return;

    // POST REQUEST
    $response = wp_remote_post( 'https://api.travis-
ci.com/repo/yourgithubhandle%2Fyourrepositoryname/requests', array(
    'method' => 'POST',
    'timeout' => 45,
    'httpversion' => '1.0',
    'blocking' => true,
    'headers' => array(
      'Content-Type' => 'application/json',
      'Accept' => 'application/json',
      'Travis-API-Version' => 3,
      'Authorization' => 'token YOURTOKENCOMESHERE',
    ),
    'body' => json_encode(array( 'request' => array(
      'branch' => 'master'
    ))
      ))
  );

  if ( is_wp_error( $response ) ) {
    $error_message = $response->get_error_message();
    echo "Something went wrong: $error_message";
  } else {
    return;
  }

}
add_action( 'save_post', 'jst_save_posts', 10, 3 );
```