LAPIN AMK
Lapland University of Applied Sciences

MODERN WEBSITE DEVELOPMENT WITH STRAPI AND NEXT.JS

Niko Pinnis

Thesis

Study Programme in Information and Communication Technology
Bachelor of Engineering

2022

LAPIN AMK
Lapland University of Applied Sciences

Opinnäytetyön tiivistelmä

Tieto- ja viestintätekniikan koulutus
Insinööri AMK

Tekijä	Niko Pinnis	Vuosi	2022
Ohjaaja	Anssi Ylinampa		
Toimeksiantaja	Panagiotis Sotiriou		
Työn nimi	Modernin verkkosivuston kehitys Strapi- ja Next.js-ohjelmistoja käyttäen		
Sivumäärä	44		

Nykymaailman digitalisoitumisen myötä on tullut tärkeäksi yrityksille esittää toimintaansa internetissä lisätäkseen omaa näkyvyyttään yleisölle. Yleisin keino täyttää tämä tarve on sosiaalisen median käyttö, mutta monesti yritykset ylläpitävät myös omia verkkosivustojaan, joissa ne esittävät ja myyvät omia tuotteitaan ja palvelujaan. Tämän työn toimeksiantaja halusi pienelle liikkeelleen verkkosivuston, jossa hän voi alustavasti esittää omia tuotteitaan ja jatkossa sivusto on oltava mahdollista laajentaa verkkokaupaksi.

Tässä opinnäytetyössä esitetään toimeksiantajalle toteutettu verkkosivusto painopisteenä sen arkkitehtuuri ja käytetyt teknologiat. Tutkimusmenetelmänä sovellettiin konstruktiivista tutkimusmenetelmää. Työn teoriaosuus koostuu käytettyjen teknologioiden kuvauksesta, joiden lähdeaineistona käytettiin kehittäjiensä dokumentointia. Lisäksi aineistona on käytetty muita tuotokseen liittyviä digitaalisia julkaisuja sekä tekijän omaa osaamista ja kokemusta käytetyistä teknologioista.

Tuloksena saatiin yksinkertainen, mutta toimiva verkkosivusto, joka täyttää sille asetetut vaatimukset mukaan lukien mahdollisuuden jatkokehitykselle verkkokaupaksi. Tulos on toimeksiantajan tavoitteiden mukainen. Työssä käytetyt teknologiat saavuttivat hyvin tavoitetun tuloksen, mutta käytetty arkkitehtuuri on turhan monimutkainen tämän nettisivun tarkoitukselle.

Avainsanat cms, Javascript, Next.js, Strapi, website

LAPIN AMK
Lapland University of Applied Sciences

Abstract of Thesis

Study Programme in Information and Communication Technology
Bachelor of Engineering

Author	Niko Pinnis	Year	2022
Supervisor	Anssi Ylinampa		
Commissioned by	Panagiotis Sotiriou		
Subject of thesis	Modern Website Development with Strapi and Next.js		
Number of pages	44		

The use of the Internet is becoming increasingly important for businesses in modern times. A combination of social media platforms, websites and e-commerce services are used by most businesses as part of their normal operation. The aim of this thesis study was to develop the website of a small store. The client for this project wanted to increase the Internet presence and visibility of his business and thus required a website that could initially present the store and its products. The possibility for future development of the website to an e-shop, without major changes, was also a requirement for this project.

A constructive research method was used in this study. The theoretical part consists of a description of the technologies that were used to implement a website for the client. Most of the source material used for this thesis are technical documentation along with some digital publications and the author's own knowledge and experience.

The client's requirements were met by the outcome of this project. The development to an e-shop in the future is possible with the technologies used. However, the architecture used is too complex to use on a small business website and some simplification will be required.

Key words Strapi, Nextjs, website, cms, Javascript

CONTENTS

SYMBOLS AND ABBREVIATIONS

ISR	Incremental Static Regeneration
CDN	Content Delivery Network
JSON	JavaScript Object Notation
REST	Representational State Transfer
API	Application Programming Interface
CRUD	Create, Read, Update, Delete
SEO	Search Engine Optimization
CMS	Content Management System
SSL	Secure Sockets Layer
TLS	Transport Layer Security
HTTPS	Hypertext Transfer Protocol Secure
CAPTCHA	Completely Automated Public Turing test to tell Computers and Humans Apart

1 INTRODUCTION

In the first chapter of this thesis, an overall description of the general concept of the project is given as well as its requirements, the technological stack that was used, what decisions were made along the way and an overview of the architecture.

1.1 About this Project

The outcome of this project is a simple website for a souvenir store merchant to display his store and a catalog of their products. The client is the owner of this type of store in the island of Rhodes, which normally operates during the summer tourist season limiting their revenue generation period from 7-to-8-months a year. Maintaining a presence on the internet is important for businesses in modern times, thus social media platforms have been used by the owner of the store to promote his business. The owner is exploring the possibility of expanding his business by also selling online through an e-shop. For the client was offered a simple website to be built, for starters, that meets the requirements for future development to an e-shop. The work on this project started during winter while the physical store wasn't operational, after its busy summer season. The owner did not require to have an e-commerce solution ready this winter which granted a flexible timeframe to experiment with an implementation beyond the requirements of a small store. It was decided by the author of this thesis that a more generic solution to be built, which would be more interesting from a technical perspective but an over-engineered solution for a small business.

1.2 Requirements

The requirements of the client's website not including the functionality of an e-shop are few. It should display some basic information regarding the store, and a browsable catalog of its products must be included. Due to the client's lack of knowledge, a convenient solution of adding and organizing products into categories, a simple panel was used, enabling the client to familiarize quick with the panel and make changes to some simple information like a store announcement or the open hours. The website should also include multiple languages support since its visitors are expected to be from different countries.

This website is expected to serve a small number of visitors thus it should meet the modern standards of performance, and it must have low running expenses because of its a small number of visitors. Fulfilling the requirements should not come at the expense of future development towards an e-shop website.

1.3 Research Method and Scope

In this Thesis the constructive research method was used. The problem to be solved was defined at the beginning along with the requirements for its solution. After that, the research was conducted for a suitable solution to be determined. Finally, a project offering the solution was completed and evaluated.

Building a complete modern website from scratch is not a simple task. To limit the work required for finishing this project, priority was given to producing a simple thematic appearance for the frontend. Due to the author’s limited design skills, a decision was made to keep things simple. For the backend implementation, a solution was chosen that automates most of the back-end functionality while still giving the author the tools to build everything that is needed.

1.4 Technologies Used

To build the frontend of the website, Next.js combined with TailwindCSS were chosen, two very popular and powerful frontend frameworks. Next.js provides the ability of dynamic page construction with content by using ReactJs, a widely used frontend library which the author is already familiar with. By using TailwindCSS only some CSS code was required to be written for the website saving a lot of time in terms of working hours.

For the backend Strapi was deemed as a good fit to be the CMS of the project. Strapi is built with JavaScript, a programming language used by ReactJs. MySQL, which is a popular relational database, was selected as the database of this project for reasons explained in the deployment section.

1.5 Architecture

The website in its core uses a JAMstack architecture (Figure 1). JAMstack stands for JavaScript, API and Markup. This kind of architecture is relatively new and it is gaining popularity for web application or website development. Because the JAMstack is a large topic, only a short explanation of its core ideas are given in this chapter.

JavaScript is the programming language that powers the frontend and the backend in this case. The architecture is reliant on the data exchange between the frontend and the backend, which is facilitated by JavaScript and JSON as the data format (Biilmann, M. & Hawksworth, P. 2019, 8.)

The API part of JAMstack is about building and using HTTP APIs that are reusable by different presentation layers, such as web browsers and mobile applications to name a few (Biilmann, M. & Hawksworth, P. 2019, 9.)

The elemental/core part of the web is HTML. The letter M stands for Markup, so it makes sense that JAMstack also has Markup in its core. Markup is delivered by a different model than previous architectures. By using static site generators and build-tools the HTML of a website can be combined with content and created prior to a request turning it to a static file (Biilmann, M. & Hawksworth, P. 2019, 10.)

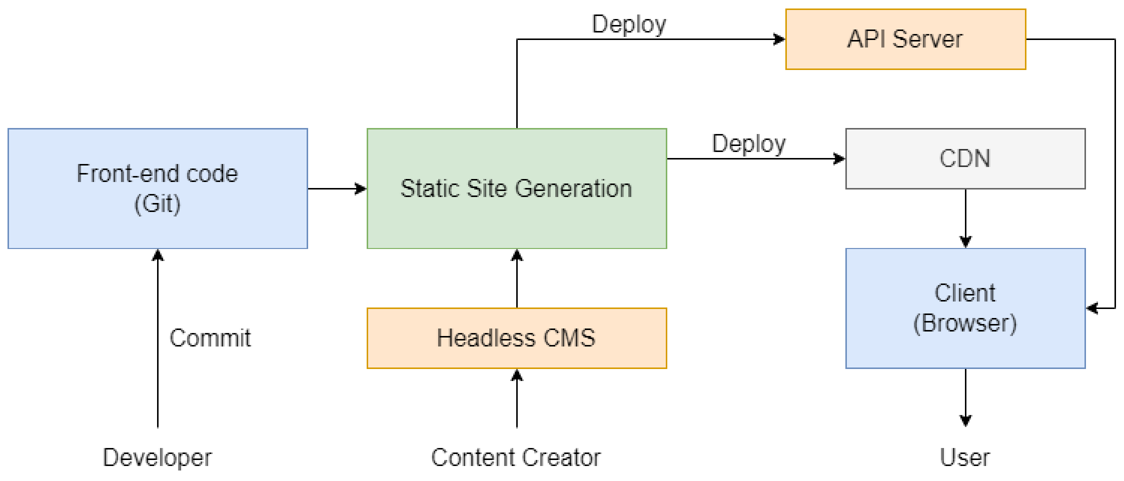


Figure 1. An example of a JAMstack workflow.

This project uses these concepts as the foundation of the website. Then the author builds on them by not only using data to describe content but also the structure of pages as well. The main idea is to store information about the appearance of a website in a CMS, where it can be easily edited within predefined specifications by a person without programming skills. The concept is similar as many widely used page builder tools but the architecture that is used for this project is specific. Building something comparable that matches the creative capabilities of current page builders was unnecessary and beyond the scope of this project/thesis.

All data used in this project are divided between two types, pages and content. Pages describe what URLs exist on the website, their contents, and a basic layout for their rendering. The content types consist of one or more collections of data that can be referenced from a page and their contents embedded in the page when it is built (Figure 2).

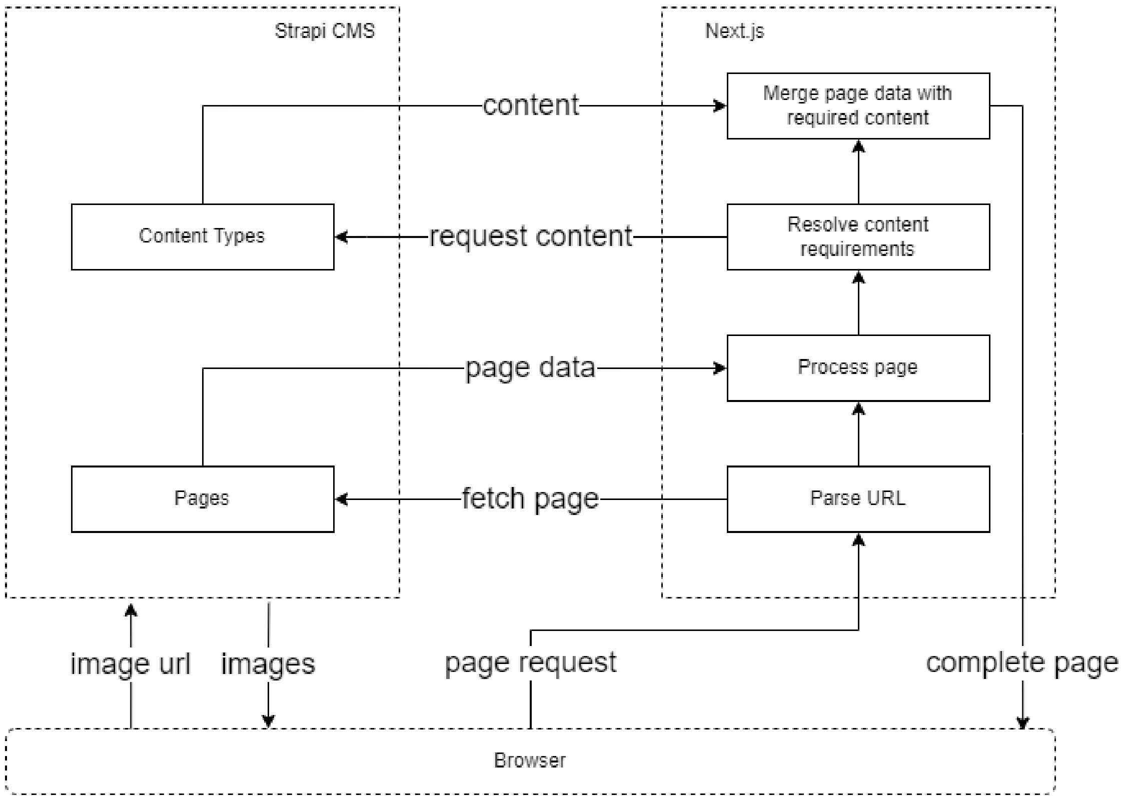


Figure 2. The flow of page rendering

2 BACKEND

2.1 Content Management Systems

For any website that requires its content to be dynamic there must be the ability of creating, editing and deleting its content. The most common approach of covering this requirement is by using a Content Management System (CMS). Individuals without technical skills have the ability to edit the content of a website by using such systems.

Most content management systems are coupled with a particular presentation system such as a website frontend or the user interface of a desktop application. Consequently, this coupling made them rigid and difficult to adapt to new delivery formats like mobile apps or web applications that are developed outside of the CMS platform (Tamturk 2016.)

To overcome this problem new systems for content management were developed that did not include a way of presenting the content making them "headless". Such systems were named Headless Content Management Systems. The purpose of a Headless CMS is to model content, enable the creation of content with authoring and organizing content in a repository. These systems do not have an opinion on how and where their content is displayed leaving that responsibility completely at the content presentation system (Tamturk 2016.)

Headless CMSs should not be confused with decoupled CMSs. Those are a regular full-stack CMS with a presentation solution but also allow the content to be leveraged by other systems (Tamturk 2016). A good example of a traditional CMS is WordPress which is in wide use all over the world powering 43% of the web. (WordPress.org 2022) In recent years WordPress has advanced towards becoming a decoupled system with the development of its JSON REST API allowing other systems to access its content.

The main advantage of a Headless CMS can also be a drawback. Since they don't feature a method of presenting their content, the presentation layer must be built separately. This adds more requirements to the owner of the system, particularly the technical skills of whatever technology is used for constructing the front-end. This can result in higher expenses during development and maintenance of the application. Another potential issue is that the content sent to the CMS by the front-end will not have an adequate amount of information about the visitors to have the content properly personalized. This issue, however, can be addressed with the use of the right business intelligence tools. Another issue is the lack of content formatting. Unlike other Content Management Systems, without a presentation solution the preview and editing of content cannot be done in the context of a page or view requiring some anticipation work to achieve the desired look. But this issue can also be solved if a preview system is implemented in the front-end (Tamturk 2016.)

2.2 Strapi

Strapi is an open-source Headless CMS development by a French company called Strapi Solutions SAS. Its name is derived from the words “Bootstrap your API”. Users are able to define content models and access them through its API.

The core feature of Strapi is that once a content model has been defined the necessary REST API endpoints are automatically generated allowing CRUD operations on the content by other systems to be performed. Content models can be created through Strapi’s own administration panel or directly by creating the required files in the source directory. After the creation of a content type the default API endpoints can be overridden and extended with as many endpoints as needed.

Relational databases are used by Strapi to store all content data. The databases supported by its major versions 3 and 4 are SQLite, MySQL/MariaDB and PostgreSQL. Support for also MongoDB, a non-relational database (NoSQL), is included in version 3, but starting from version 4 official support for it has been dropped by its developers. Database table creation and querying are automatically handled by Strapi, but manual queries can also be performed by the developer when necessary (Strapi SAS 2022.)

A plugin system is also featured by Strapi, by which developers are allowed to create their own plugins for extending the functionality of their Strapi CMS implementation. Strapi has support included for a few official plugins. These are:

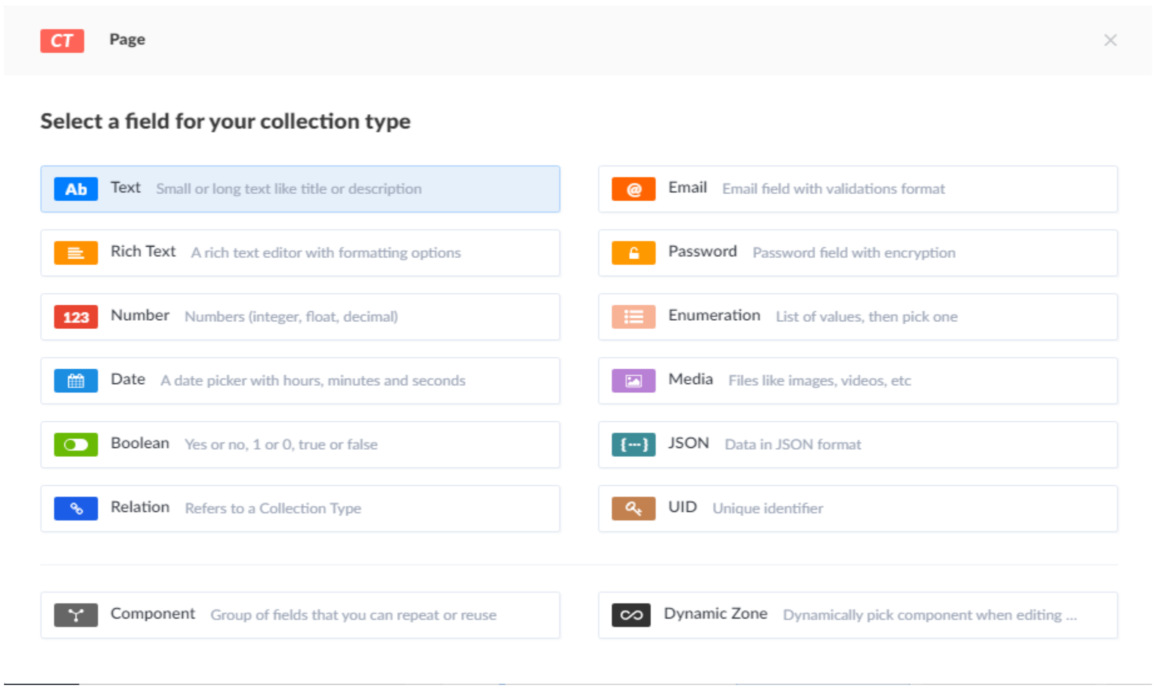
- GraphQL
- Internationalization (i18n)
- Users and Permissions (installed by default)
- Email (installed by default)
- Upload (installed by default)

GraphQL queries on the content are enabled by the use of the GraphQL plugin which adds the necessary endpoint to the API. These queries are also autogenerated by default. With the Internationalization plugin content can be created and requested separately for each language. The 'Users and Permissions' plugin adds access control to the API endpoints based on the roles defined by the Strapi administrator. Finally, email sending and media file upload features are added by the Email and Upload plugins respectively.

The user management feature from the Users and Permissions plugin is not to be confused with the user management system in the Strapi administration panel itself. A user and role management system for the use of the admin panel is included with Strapi, but its customization capabilities are limited when using the community edition of Strapi. For the full customization functionality an Enterprise license is required.

2.2.1 Content Type Definition

Content types are defined in Strapi by an administrator through the use of the admin panel which is built from source when installing Strapi. This feature is provided by the built-in Content-type Builder plugin. Support for defining the fields of a particular content model such as text, numbers, booleans, media (images, video, etc.) and more is also included (Figure 3). Fields can be grouped together as components to be reusable and relational fields can be added to connect content collections together with a relationship.


CT Page

Select a field for your collection type

Text	Small or long text like title or description
Rich Text	A rich text editor with formatting options
Number	Numbers (integer, float, decimal)
Date	A date picker with hours, minutes and seconds
Boolean	Yes or no, 1 or 0, true or false
Relation	Refers to a Collection Type
Email	Email field with validations format
Password	Password field with encryption
Enumeration	List of values, then pick one
Media	Files like images, videos, etc
JSON	Data in JSON format
UID	Unique identifier
Component	Group of fields that you can repeat or reuse
Dynamic Zone	Dynamically pick component when editing ...

Figure 3. The available fields for collection type content

2.2.2 Content Management

After at least one content-type has been defined the admin panel can be used by administrators to create and edit the content in the CMS. A simple table view of the entries in a collection are offered by the user interface of the content manager along with a search field, filters and pagination. The visible columns of the table can be customized, but only entries of a single locale at a time are displayed on the table.

By selecting an entry from the table or clicking the “Add New” button the administrator is taken to the content editing view which is also customizable (Figure 4). All the necessary input elements to edit the data of the content are contained in this view with validation for the input.

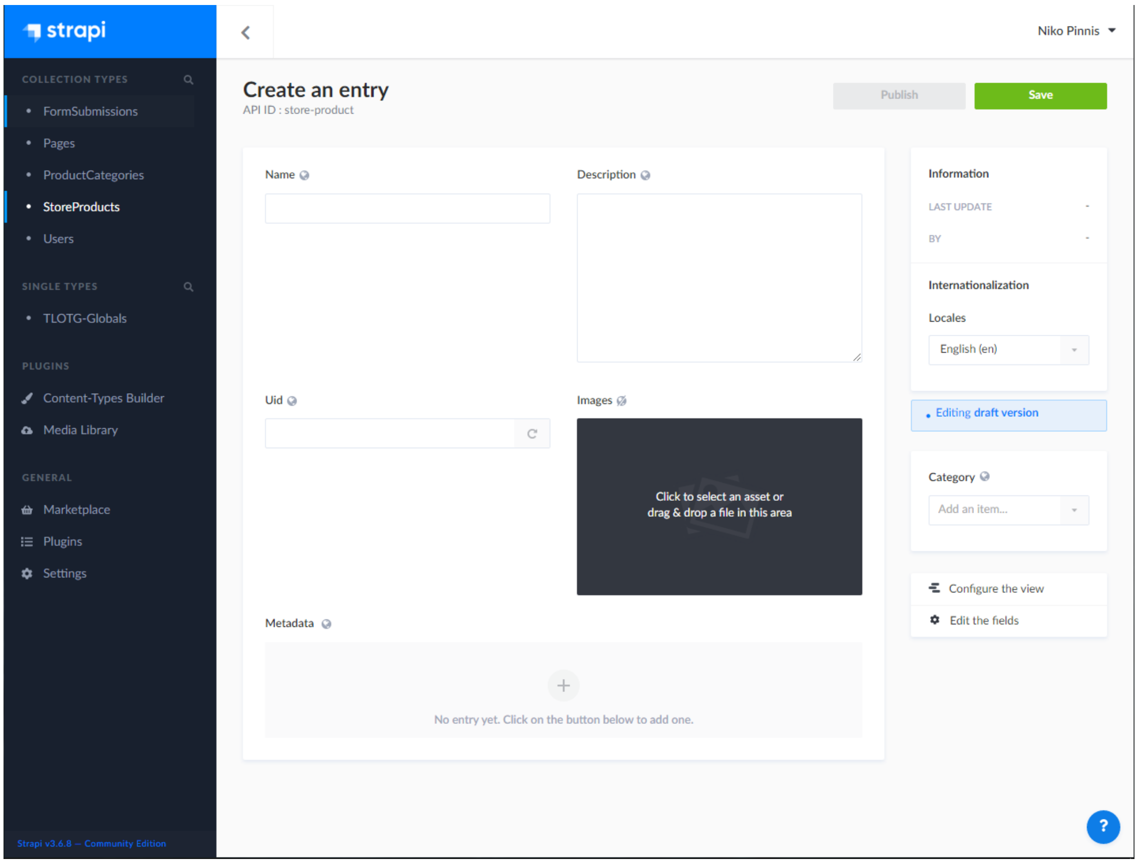


Figure 4. The new entry creation view in the admin panel

2.2.3 Image Storage

A built-in media library plugin for handling image storage is bundled with Strapi and it is activated by default (Figure 5). A provider is used by the plugin to upload files and, by default, a local file system storage provider is used to store files in the public folder of the Strapi server. This behavior can be replaced by configuring another provider for the plugin. Providers are maintained by the Strapi team for: Amazon S3, Cloudinary and Rackspace (Strapi SAS 2022). Custom providers can also be implemented by developers if necessary. For this project the local file storage provider was sufficient, and it was used with the default configuration. By default, the plugin has the images resized to different widths, by the use of an image manipulation library called sharp. An optimized image size can be chosen by the front-end for better performance when displaying images. This choice contains some caveats that are explained in the deployment section.

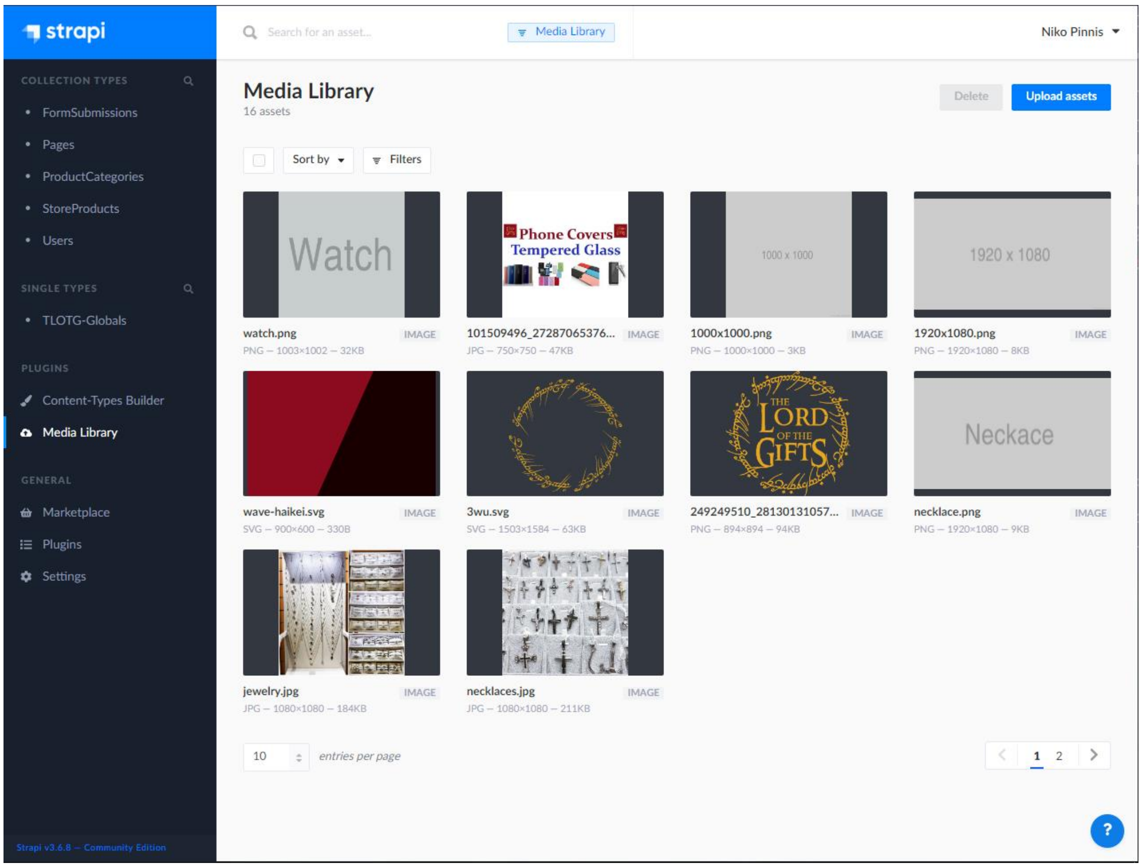

strapi
Search for an asset...
Media Library
Niko Pinnis
COLLECTION TYPES
FormSubmissions
Pages
ProductCategories
StoreProducts
Users
SINGLE TYPES
TLOTG-Globals
PLUGINS
Content-Types Builder
Media Library
GENERAL
Marketplace
Plugins
Settings
Strapi v3.6.8 — Community Edition
Media Library
16 assets
Delete
Upload assets
Sort by
Filters
watch.png IMAGE PNG — 1003×1002 — 32KB
101509496_27287065376... IMAGE JPG — 750×750 — 47KB
1000x1000.png IMAGE PNG — 1000×1000 — 3KB
1920x1080.png IMAGE PNG — 1920×1080 — 8KB
wave-haikei.svg IMAGE SVG — 900×600 — 330B
3wu.svg IMAGE SVG — 1503×1584 — 63KB
249249510_28130131057... IMAGE PNG — 894×894 — 94KB
necklace.png IMAGE PNG — 1920×1080 — 9KB
jewelry.jpg IMAGE JPG — 1080×1080 — 184KB
necklaces.jpg IMAGE JPG — 1080×1080 — 211KB
10 entries per page
1 2

Figure 5. The Strapi media library plugin

2.2.4 API Customization

Whenever a content-type is created or edited the server is restarted and a set of default API endpoints and handlers are generated for the content-type. These defaults are the bare minimum required for allowing CRUD operations to be performed on the data by a HTTP-client. These endpoints can be replaced and extended by developers with their own implementations for adding custom access control, database queries and any other features as needed.

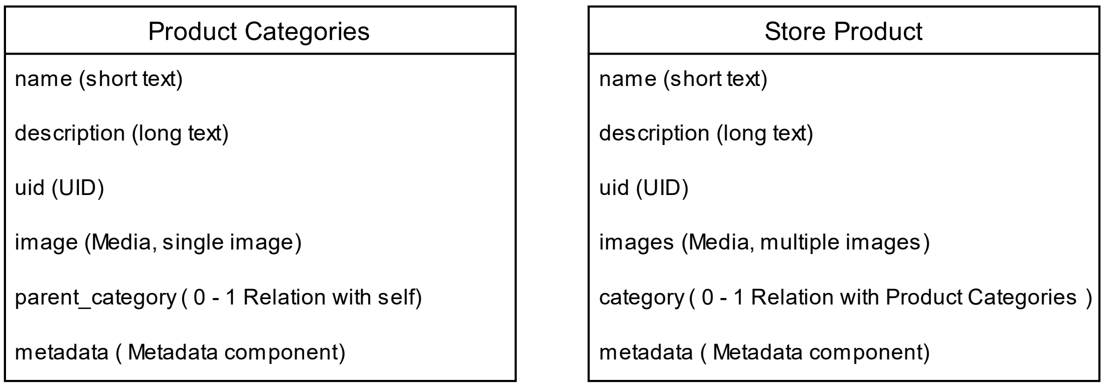
For handling requests an internal middleware is used by Strapi. Other middleware can be added by developers to all routes and the load order of Strapi's internal middleware can also be changed.

2.3 Strapi Implementation for this Project

In the implementation of the backend, content types were used to describe not only the content of the website but its structure as well. To accomplish this the pages of the website became a collection in the CMS with their entries describing the view of each page. Every page of the website is identified by its own unique slug, and this slug is used to fetch the data of a page from the CMS.

2.3.1 Website Content

The actual content in this project are two collections. One is used to store the categories where the products belong to and the other is used to store the products themselves (Figure 6). One parent category for each category is supported by the categories collection in order to define subcategories and products can only belong to one category or subcategory. This relationship is deemed sufficient by the client for the website.



Product Categories
name (short text)
description (long text)
uid (UID)
image (Media, single image)
parent_category (0 - 1 Relation with self)
metadata (Metadata component)

Store Product
name (short text)
description (long text)
uid (UID)
images (Media, multiple images)
category (0 - 1 Relation with Product Categories)
metadata (Metadata component)

Figure 6. The collection types for the content

2.3.2 Website Global Parameters

It is standard for modern websites to include a navigation header on the top of every page, and a footer on the bottom with links for different pages, social media sites and other useful information. Since these elements are common for every page in the website a content-type of type “Single” was created, which as the name implies, only has one entry in the CMS. In that single entry, named as “TLOTG-Globals” (Figure 7), the header and footer data are defined and stored for retrieval by the front-end of the website. Global data is split into components describing the navbar, the footer, a system-bar and an announcement block.

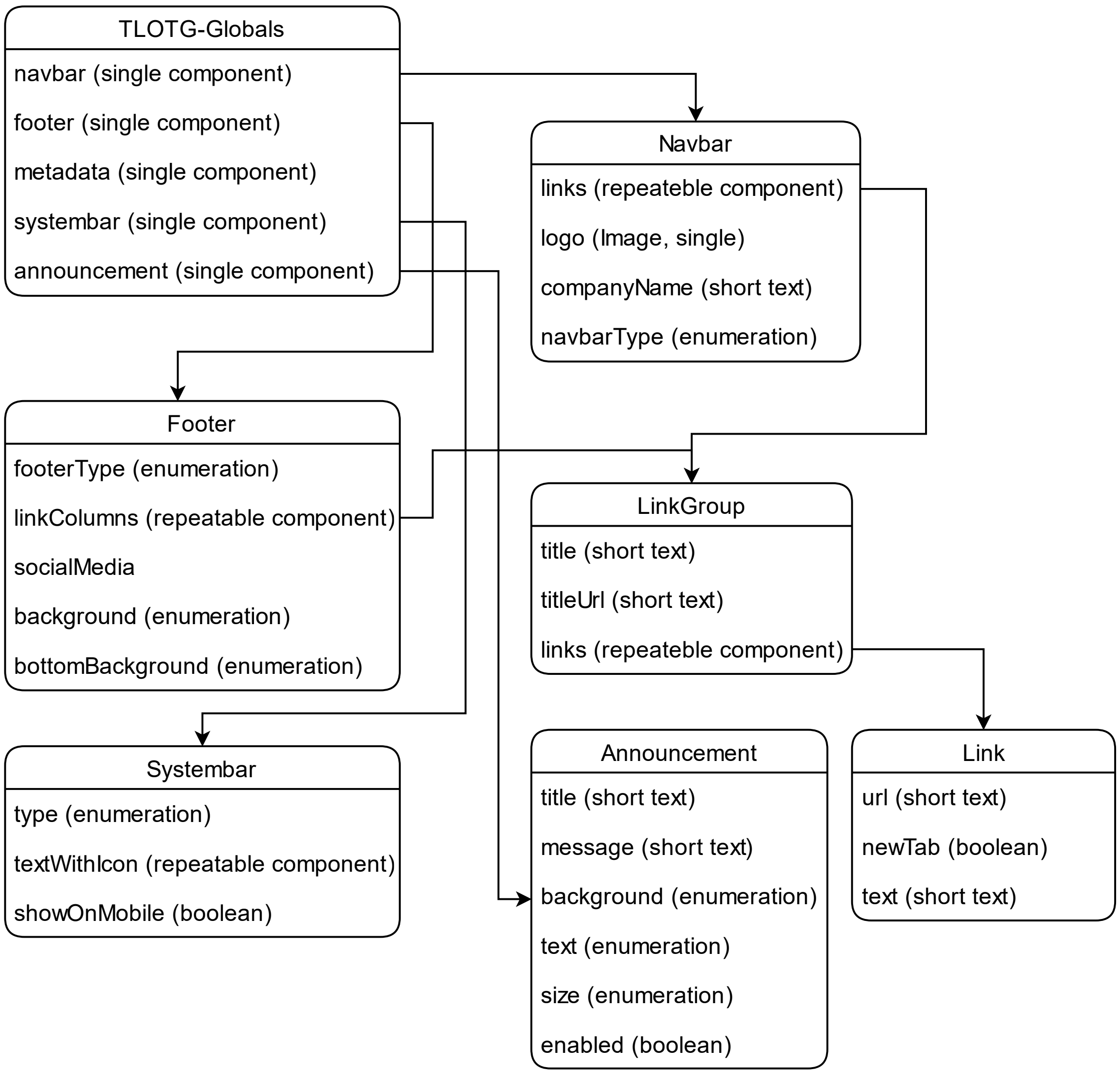


Figure 7. The component structure of the global content-type

2.3.3 Pages as Content

The structure and content of each page, on the website, exist as an entry in the Pages collection. A unique slug field that defines the URL of the page is included in each page in the collection.

When used together with a component called “Dynamic-Route” a set of dynamic URLs are formed, that the front-end can understand and display with the specific content they point to. This dynamic component points to a collection and names a field in the content that is to be used as a unique identifier for the formation of the unique URLs (Figure 8).

A field named blocks is also included in the model of the pages collection. Because this field is of a “Dynamic-zone” type, any number and combination of components from a group of components defined at the content-builder can be added to it as an entry. The structure of a page is defined by the contents of this field.

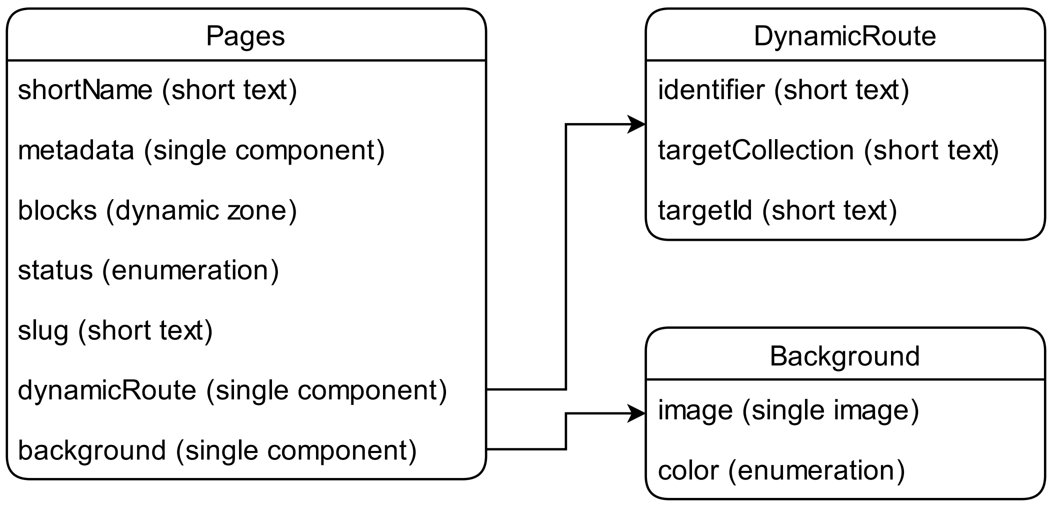


Figure 8. The Pages collection

2.3.4 Page Blocks

For the front-end of the website React, which is a JavaScript library that features components to render HTML documents, is used. That said, it makes sense for components in the CMS to be used that describe React components so that ‘React’ can render them with the data provided from the CMS as component props. These components called “blocks” are used in the Pages collection as described above and the backbone of the website is formed by them.

Several blocks were defined and built to satisfy the needs of the project. A list of the blocks that made it to production are:

- Hero, for the landing page. Features a Call-to-Action button
- Categories, to display product categories in a grid

- Products, to display products of a single category in a grid
- Contact, a simple contact form
- Markdown, to render markdown text as HTML
- FAQ, to display the FAQ section
- Google-Maps, to display a map with the location of the store

2.3.5 Metadata

Basic SEO information is added to all pages on the website by the Metadata component (Figure 9). This component was used by the following content-types listed in increasing priority:

- TLOTG-Globals single type
- Pages collection type
- Product Categories collection type
- Store Products collection type

The values in this component are read by the front-end in the above order and overwrites are performed by a higher priority value, if such value exists.

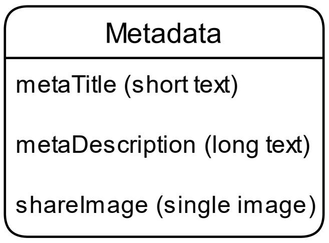


Figure 9. The Metadata component

3 FRONTEND

3.1 Tailwind CSS

A plethora of utility classes for styling websites are offered by TailwindCSS, a popular CSS framework. Almost every CSS property has a CSS class for it with multiple variants for the property value and screen size. The classes can be overridden and extended from a configuration file and the use of plugins is also supported by the framework.

By using TailwindCSS, every HTML element was given the styling it needed just by reusing the classes the library offers. To reduce some repetitiveness, the “@apply” directive was used to combine multiple Tailwind classes to a single custom class. This also helped reduce some of the verbosity in the ‘class’ attribute of the HTML elements that using tailwind causes.

3.2 Next.js

Next.js, a React framework, was developed by a company called Vercel. It has many features bundled in it allowing developers to create production ready full-stack websites and web-applications. Some of the features included in the framework are Image Optimization, Internationalization, A hybrid Static Site Generation and Server-side Rendering, Typescript support, File-system routing, API routes, Built-in CSS support and code-splitting and bundling (Vercel 2022a.) The framework can be used in the development of full-stack web applications while using any one of the popular databases and any Node.js server that meets the minimum requirements can be used to run it.

Next.js was used to build the front-end for this project since it offers everything that a website needs to run reliably and performantly. It also works well with several Content Management Systems including Strapi. It can fetch all the necessary data from the CMS already at build-time and its built-in API endpoint creation support allowed for implementing a simple form submission logic without exposing any sensitive keys to the front-end or using 3rd party services.

3.3 Pages and Routes

A filesystem-based routing system is featured by the framework. All the possible URLs of the website are defined by the structure of directories and JavaScript files inside the “pages” directory of the Next.js project source. Any URL that does not satisfy these will result to a 404-error page by default unless a redirect or a custom page has been configured. With this system several static routes can be implemented for the website by a developer. However, it is often the case that the server cannot know in advance the full URL of a route when that route depends on external data, like in the case of content fetched from a CMS. In that case the declaration of dynamic routes that accept one or more parameters is supported by Next.js allowing for the creation of dynamic routes with nesting (Vercel 2022b.)

3.3.1 Static Routes

As is stated above the static routes of the website are defined by the contents of the ‘pages’ directory. Two routes in this project are dynamic, the categories and the products route. The other routes are simple, and their URLs are static, meaning that they could be implemented directly in their own files. For example, the “About Us” page could be handled by a file named about.js at the root of the pages directory which would fetch the page content from the CMS and display the page. In this project a different approach was attempted using only dynamic routes.

3.3.2 Dynamic Routes

A route becomes dynamic when square-brackets are added to the name of the directory or file, which a developer wants to make dynamic, for example: `/pages/post/[slug].js`.

In this simple example the last part ‘/[slug].js’ of the URL became a route parameter named slug and its value can be used to fetch a specific post from a CMS.

For more complex routes this can be extended so as to catch all the paths (beginning with: /pages/post/) by the addition of three dots inside the square brackets before the name of the parameter, for example: `/pages/post/[…slug].js`. In this way URLs with deeper nesting paths can be handled. For example, the URL: `http://domain.name/post/some-post/edit` will match:

`/pages/post/[…slug].js` with the parameters passed to […slug].js as the following JavaScript object: { “slug”: [“some-post”, “edit”] }.

This type of catch-all route can be made optional by using double square brackets. The main difference between optional and non-optional catch-all routes is that if it’s optional, the route without a parameter is also matched resulting in an empty parameter object passed to the handler. It’s important to note that, when using dynamic routes if a URL matches both a dynamic and a static route precedence is taken by the static route (Vercel 2022c.)

In this project only a single optional catch-all route at the root of the “pages” directory was used for all routing. The handler file is named “[[…slug]].js”. Enabling the definition of all pages with their contents in the CMS, while the front-end is set to fetch the data required to render all the pages. By using this method, the implementation of pages with a unique slug becomes straightforward. There might be also a dependency on a content collection regarding the retrieval of data (dynamic routes), which differs of the way mentioned above. In this manner, dynamic content retrieval is implemented through a different component, which is going to be analyzed in the next section.

3.3.3 Connecting Pages to Collections

There are two content types in this project that result in the generation of dynamic routes at the front-end: categories and products. By using categories as the chosen content-type, a solution to the above challenge is presented next. To define the connection of a dynamic page with a content collection, a component named ‘DynamicRoute’ is added to the model of the pages collection (Figure 8). The relation between the page and another collection (in this case product-categories) is stored in the ‘targetCollection’ field of this component and the unique identifier field (in this case the uid) in the target collection that is used for creating unique URL paths is stored in the ‘identifier’ field. When Next.js processes the data of the page it knows to request all the content of the collection and a list of all the possible route endpoints can be built. The result for the example mentioned above is the URL scheme: `/categories/category-uid` with a single entry of the collection being referred to by category-uid.

3.4 Data Fetching

Several different data fetches are performed by Next.js to display the content in this project. These requests are executed inside four functions exported by the framework from the JavaScript files inside the ‘pages’ directory. Pages are pre-rendered by using these functions that run on the server.

3.4.1 Using getStaticPaths

This function is needed to define a list of valid paths when using dynamic routing. It runs only once at/during build time on the server, and an array of paths is returned by it with their parameters and a fallback property. When this fallback property is false then only paths in the array can be rendered, for this dynamic route, with other paths resulting in a ‘404-error’ page being displayed. When fallback is set to true then, instead of an error page, a fallback page is served while an attempt is made by the server to generate a page for the route. Upon completion the page is sent to the browser. Finally, when it is set to ‘blocking’ nothing is sent while generating the page which is then cached and served upon completion. A loading indicator can be used to inform the page visitor that the page is being built (Vercel 2022d.)

3.4.2 Using getStaticProps

This function is used to prerender a page on build time by using the props returned to the page by this function. Data fetching from a CMS or any API can be performed by it and a static page generated when the required data is not user specific. Pages implementing this function are made very performant as their resulting HTML can be cached by a CDN. When combined with Incremental Static Regeneration (ISR) any page already cached can be set to invalidate after some time by setting the ‘revalidate’ property at the returned props, which causes the function to rerun and an updated version of the page to be created and cached

(Figure 10). This is important because if the data fetched can change at some point in time and a complete rebuild of the project is not practical (Vercel 2022e.)

The pages of this website are served by using ISR, to make sure that updates in the CMS are pushed to the front-end after a short time. This way content updates can be received by the website without having to trigger a full rebuild of the front-end.

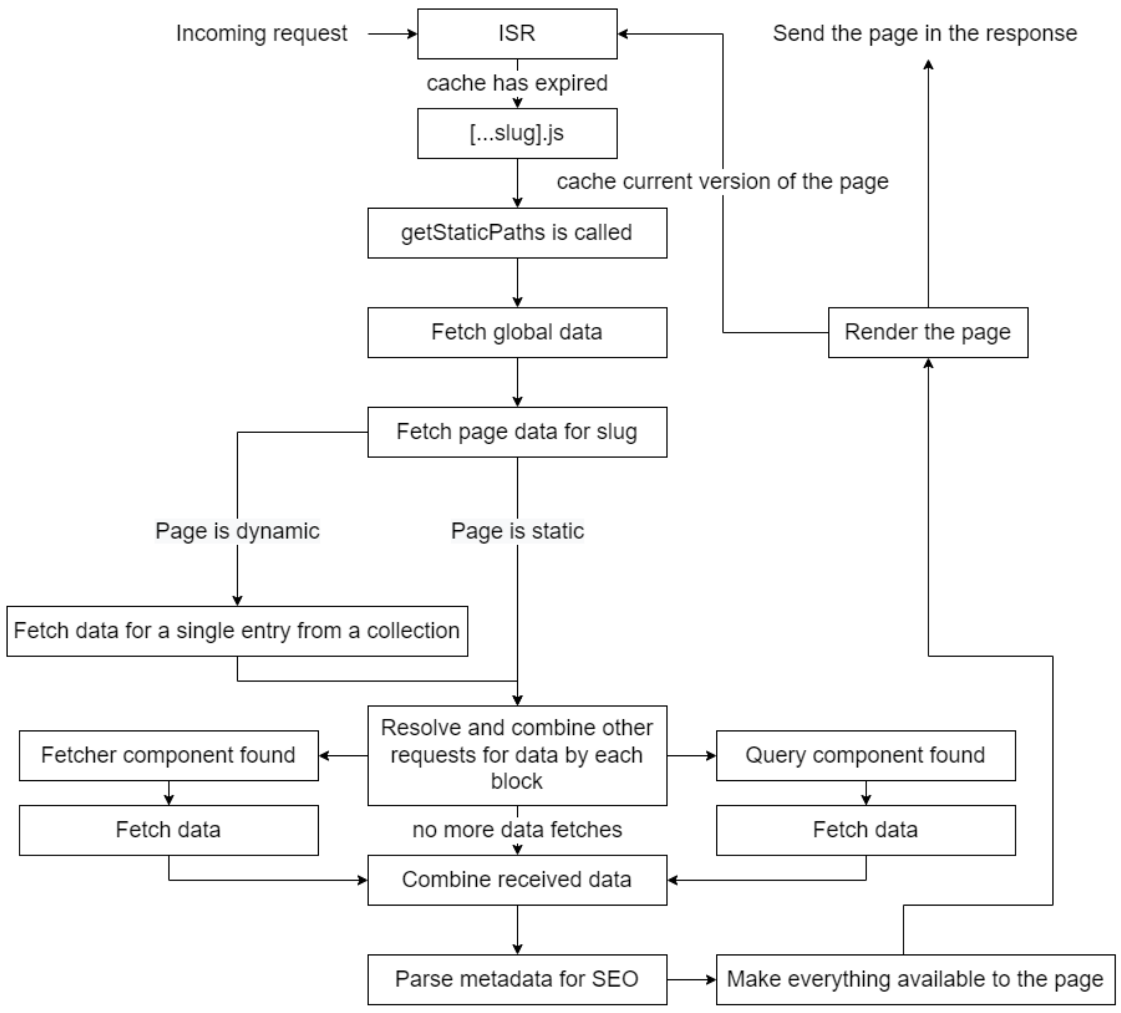


Figure 10. The flowchart of the page generation handler

In this project getStaticProps gets called for every page in an optional catch-all route handler called [[…slug]]. The slug is passed to the function as a parameter (a string array) to be parsed. Before parsing the slug, the global website data is fetched from the CMS. After it has determined whether the page slug is static or dynamic, the CMS is queried for the page data and the returned data is processed. In the case of a dynamic route the dynamic part of the slug (the last index of the slug array) is used to fetch a single entry from the CMS of the collection as indicated by the ‘DynamicRoute’ component of the page data. This data is then passed as a prop to the page component and becomes available to all the blocks in the page (Figure 10).

Next the structure of the page must be determined from the page data, specifically the blocks field. By iterating each block of the page, a list of other queries that must be sent to the CMS are assembled and the requested data is then fetched. In the end, the SEO metadata is merged according to a priority, which is mentioned on section 2.3.5 and all the data needed by the page is returned as props to the page rendering component (Figure 10).

In production this process is run for each path returned by getStaticPaths once at build time and every time the page cache expires after that due to ISR. During development the function is always called when navigating to a route. The preview mode feature of the framework can be enabled by using an API route and some cookies are set on the browser to indicate the preview mode. When the preview mode is enabled the getStaticProps function gets called for each request in production too.

3.4.3 Using getServerSideProps

This function is run on the server every time the page is requested, and it’s used to execute server-side only code on every request and when the page data is often changed. This function is not to implemented in any route of this project since the content of the website is not expected to change often according to the current requirements, and none of the website data is specific for a particular user (Vercel 2022f.)

3.4.4 Using getInitialProps

By using this function server-rendering of pages and initial population of the data in the browser are enabled, a feature useful for the SEO. However, it is recommended by the Next.js documentation to use getStaticProps and getServer-SideProps functions instead as these allow developers to have more control between static generation and server-side rendering. Automatic Static Optimization is also disabled, when using this function, which is a Next.js feature that increases page loading performance by allowing applications to emit both server-rendered and statically rendered pages. This function is not used in this project (Vercel 2022g.)

3.4.5 Client-side Fetching

Data fetching functions for on the client-side are not included in Next.js as it does not require any to function. Since the front-end is made with React, any data fetching scheme compatible with React can be used. In this project only server-side data fetching is used.

3.5 Blocks

In this project React components in the front-end and content components in the CMS together define blocks. The structure of a page is formed by these blocks, and they are stored in the ‘blocks’ dynamic zone field of the pages collection in the CMS. Two types of these blocks are designed.

3.5.1 Simple Blocks

Only data stored for them is needed by simple blocks. This data is then used to populate and customize the HTML of their corresponding React component. The blocks used to construct pages such as the ‘Contact Us’ page and the ‘FAQ’ page fall into this block type.

3.5.2 Data Driven Blocks

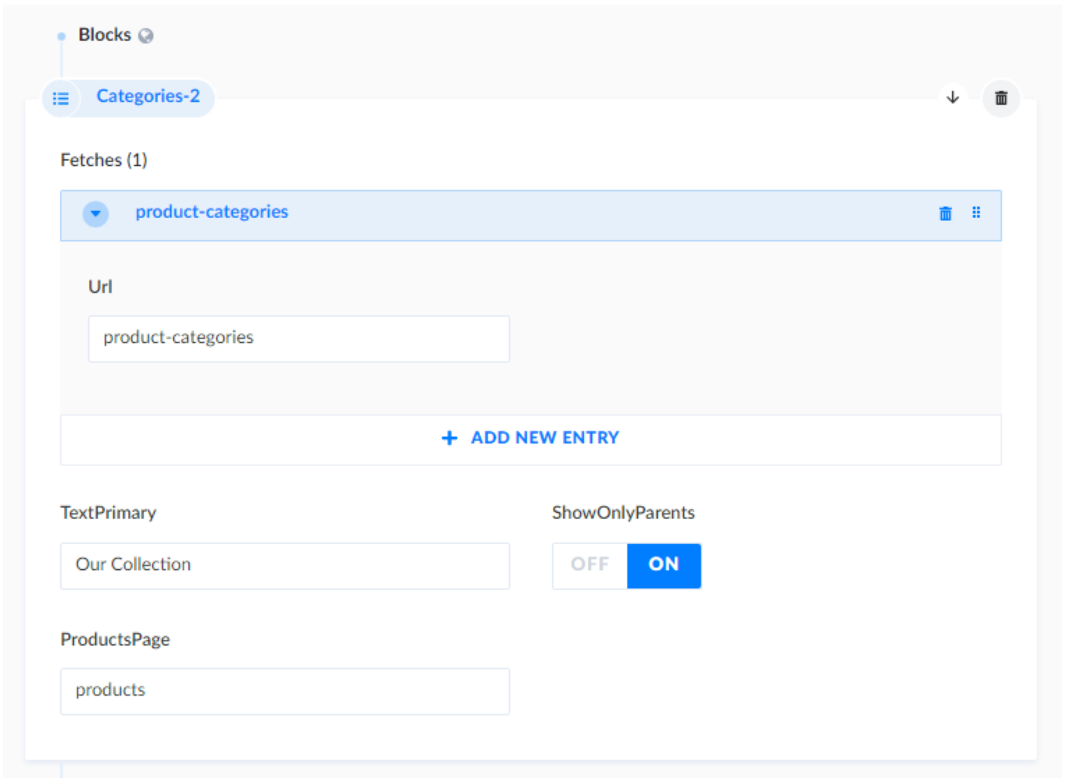
To render this type of blocks all or some entries are needed to be fetched from a content collection of the CMS. To describe these data requirements, two repeatable components were defined in Strapi, to be used optionally in this type of a block component. These components are named as Fetcher and Query component (Figure 10).

A collection name is defined by the Fetcher component, which is needed to be retrieved for the block to render. While a filter condition is defined by the Query component, which is used to only fetch a part of the collection. The filter condition was chosen to be a simple equality check between a field in the model of the collection and a value provided.

The Fetcher component was used for the Categories block since all its entries are needed by the block and the Query component was used for the Products block because only the products of a particular category are needed.

3.5.3 Categories Block

The display of all product categories is handled by the Categories block and a Strapi component (Figure 11) to describe is created. The component is then added to the blocks field of the appropriate page. A single Fetch component is used by this block to point the page generating code to the correct collection so that all the entries can be fetched during the page build. The other data fields are a simple title text field for the block, a switch to make the page show only parent categories or all categories and a text field that contains the slug of the products page in the pages collection so that the block can generate the link URLs to the dynamic pages that list the products of each subcategory.


Blocks

Categories-2

Fetches (1)

product-categories

Url

product-categories

TextPrimary

Our Collection

ShowOnlyParents

OFF [✓ ON]

ProductsPage

products

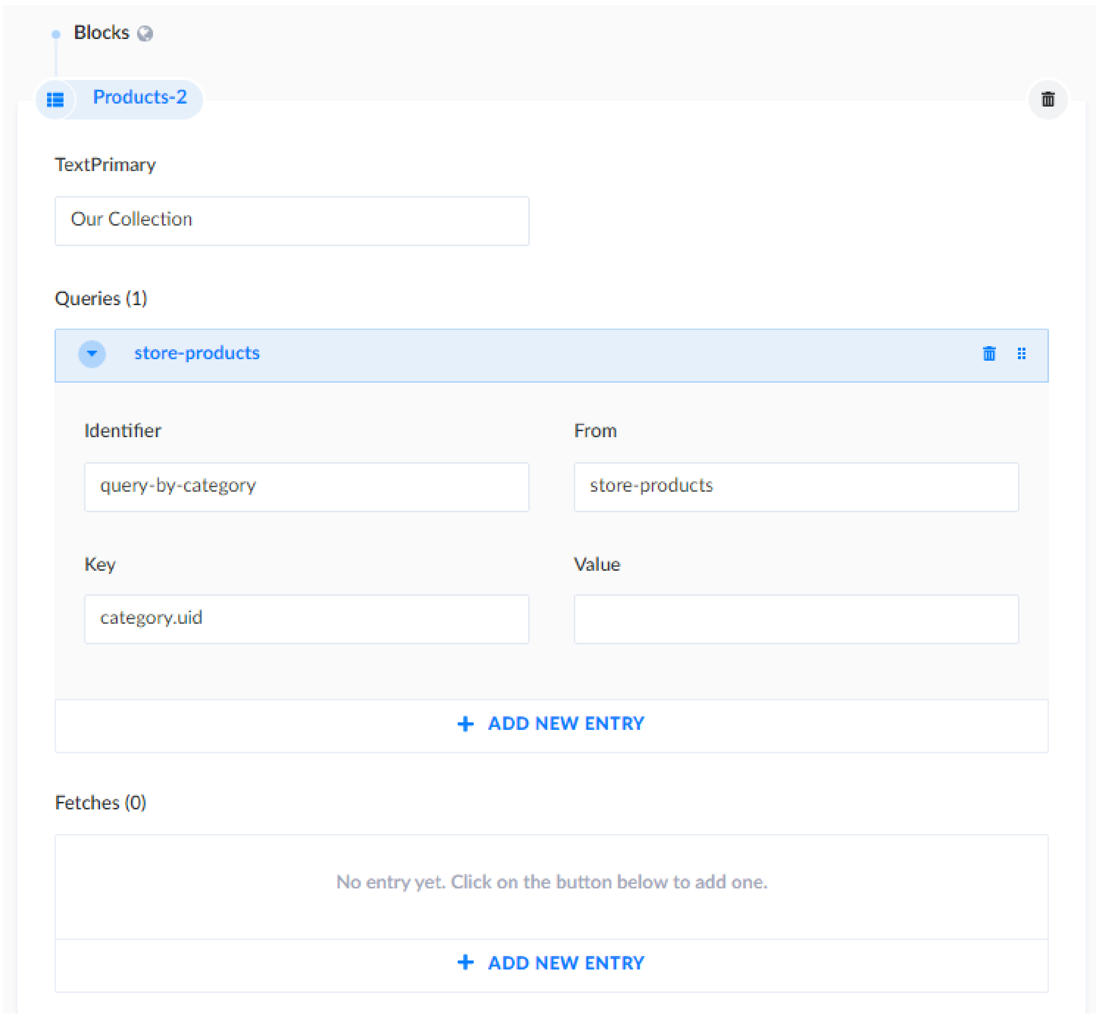
Figure 11. The categories Strapi component with the required values.

3.5.4 Products Block

For the display of products, a block named ‘Products Block’ and a Strapi component (Figure 12) to describe it is created. In this block instead of a Fetch component a Query component was needed because only some entries are needed to be retrieved from the collection.

For the Query component 4 parameters were needed to be defined. The ‘Identifier’ which is just a unique key for the query if the block has more than one Queries. The name of the collection to be queried by the page generating code is stored in the ‘From’ field. What the block is looking for in the collection is determined by the ‘Key’ and ‘Value’ pair.

The ’Key’ field of this component in the page entry (Figure 12) was filled with a value containing a dot (category.uid). Nested queries to the Strapi API are represented by using a dot and in this case, when performing the query, a match is determined by looking at the ‘uid’ field of the category entries and comparing it with the value of the ‘Value’ field. When the ‘Value’ field is left empty the dynamic part of the Dynamic Route is used instead. If the page being rendered is not dynamic that parameter does not exist, and the query is ignored.


Blocks

Products-2

TextPrimary

Our Collection

Queries (1)

store-products

Identifier: query-by-category

From: store-products

Key: category.uid

Value:

ADD NEW ENTRY

Fetches (0)

No entry yet. Click on the button below to add one.

ADD NEW ENTRY

Figure 12. The Products component with the required values filled.

3.6 Internationalization

Support for internationalization is a built-in feature of Next.js. A cookie is used by it to track which locale is currently active. Locale detection is done automatically for each visitor at the first visit, and the cookie is set. In order to serve pages with a different locale all page routes are prepended by Next.js with the locale identifier for the currently active locale, except for the default locale. A locale switch can be done programmatically using the ‘next/router’ component or by clicking a ‘next/link’ component with the appropriate props set. The activation of the internationalized routes requires the configuration of the locales supported in the ‘next.config.js’ file of the project. When switching to an unsupported locale or navigating to a non-existing page of a locale Next.js, it results in a fallback page of the default locale. This feature worked well together with Strapi’s own Internationalization plugin making the insertion of another language in the website a simple task.

3.7 SEO

The process of improving the organic traffic quality and quantity to a website is called Search Engine Optimization (SEO). This is achieved by using a set of techniques that make the website more likely to appear higher in the search results of a search engine. As a complex and evolving process that is beyond the scope of this thesis, SEO was not completely implemented, but the basics of SEO were added on the project (Beal 2001.)

As described in a previous chapter, a metadata component was added to the content data in the CMS. To use this data, a Node Package Manager (npm) package called ‘next-seo’ was also installed. A React component is provided by this package, which can be added to every page of the website. By using this component and passing into it all the SEO data available, much time and effort was saved by not having to add a custom SEO implementation for the HTML documents.

To keep things simple, only the ‘Open Graph protocol’ props of this component were filled with the metadata from the CMS together with the canonical URL prop which is constructed through accessing the full-page URL by the ‘next/router’ component. The website page title and description tags are also filled from the same metadata.

3.8 Rendering Methods

As described in the data fetching section, the rendering method used for a page upon a request in Next.js is determined by which functions are exported from the page handler. The getStaticProps should be used for static pages, so that they can be cached directly as an HTML file, while the getServerSideProps must be used when it's critical keeping up to date the latest data for every request, resulting in a fresh server-rendered page. Exporting the entire website as static HTML with every page prebuilt is possible when no server rendering takes place. This way a Next.js application can be hosted without a Node.js backend, since only static files are contained that can be served from a CDN.

Static page generation with ISR so as to serve the pages is used by this project for quicker page loading time, while still reacting to changes in the content. The use of the static export method was also considered, but due to its limitations this option was of no use but it could be useful for other projects of course. Two critical limitations of static export are the lack of API routes and internationalized routing, both features are used in this website.

3.9 Image Optimization

3.9.1 Next.js Image Optimization

Built-in image optimization is offered by using the 'next/image' component, which is an extension of the HTML <img> element. A variety of props are available for customizing how the image is placed and its use is straightforward when using an image stored in the static assets of the project.

It is often the case that the image is found in an external URL and is therefore a remote image. For security reasons the origin domain of remote images must be listed in the Next.js config before loading them is permitted by this framework. It may also be necessary (in some cases) overwriting the default loader and replacing it with a custom loader so as to enable the function of image optimization. A few image loaders for different image storage providers are a built-in feature of the Image component and they are used to build the 'scrset' attribute of the <img> element.

3.9.2 Image Loading

All the images in this project are stored and retrieved from the CMS. Different sizes are automatically created by Strapi for each image uploaded to its library, but to ensure that the correctly sized image is used by Next.js, a small custom loader for the Image component together with a wrapper component are written so that the image URLs from Strapi can be matched to the screen sizes in the loader.

3.10 Securing the CMS API Endpoints

Assigning roles for authenticated users from the Users and Permissions plugin can be used to add access control to the Strapi API endpoints. But when the user is accessing the API is 'another server' it's simpler to implement an API key authentication scheme instead of a password-based user authentication flow. When first working on this project the author of this thesis started working with version 3 of Strapi which did not include a built-in API key authentication functionality. This feature came out with version 4 of Strapi.

A simple API key validation was implemented by following a guide at the Strapi documentation. With this addition the Strapi server now checks every request for a secret token to validate it. It is important to note that this is not a permanent solution as the secret API token is placed to the query parameters of the HTTP request which makes it readable by anyone by whom the request passes through (Ploesser 2021). A better solution could be putting the token in a HTTP header since those are encrypted when using Transport Layer Security (TLS). This token check is skipped when using the administration panel or the request is by an authenticated user. Updating to Strapi Version 4 is a must for the final production release, so as to ensure continued support by its developers, and the built-in API key check used instead.

3.11 API Routes

One of the many features of Next.js is the ability to build an API backend for the web application within Next.js itself. The API routes of the framework are defined inside the `/pages/api` directory of the project and follow the same naming convention as the pages with both static and dynamic endpoints. Each endpoint exports a handler function that accepts a request and a response object as its parameters. The API routes also support the use of middleware in a similar manner of how Express, a popular Node.js back-end framework that implements middleware. Naturally these functions only run in the server and can be employed securely for a range of operations such as database queries or fetching data from another API. Depending on how the project is deployed to a host these handler functions can become and execute as serverless functions.

In this project only three API endpoints are needed to be implemented. Two of them for enabling and disabling the preview mode and one for submitting the contact form. The endpoint handler for the contact form is: `/pages/api/form-submission.js`. The handler function expects a POST request with the following body data: email, name, subject, message. These values are then sent to the CMS by a POST request, creating a new ‘form-submission’ content entry. The endpoint contains a check for a honeypot field. Honeypot fields are empty inputs in the form that are invisible to the user, but bots can see them and if they (these bots) insert text into them and submit the form the server automatically recognizes them as spam and rejects this kind of submissions.

4 DEPLOYMENT

The deployment of this project in production required two Node.js servers and a database server. Version 3 of Strapi requires a Node.js version between 10.16.0 and any minor version of 14, while only the v12 and v14 Long Term Support Node.js versions are supported. Below a table is presented with the minimum database version requirements. The hardware requirements can be significant for an API that receives heavy traffic, but by using prerendering and caching on the front-end – these requirements are scaled down. The minimum requirements are 1 CPU core and 2 GB of RAM but the recommended values are double. For low traffic operation the amount of 2 GB of RAM is not always needed by the server. This requirement comes from the heavy memory utilization (>1 GB) of the admin dashboard generation on deployment, a process done once for each full deployment. However, if the deployment is configured with a separate or no dashboard deployment the actual minimum amount of memory is much lower. The dashboard can also be pre-generated on the development machine and uploaded to any HTTP server since it's just a React app bundle with static files.

Table 1. Strapi minimum version requirement for a database.

Database	Minimum version
SQLite	3
PostgreSQL	10
MySQL	5.6
MariaDB	10.1
MongoDB	3.6

Because a persistent file storage solution is required by the project as Strapi's media uploads are stored there, using a hosting service that features saving files to a filesystem makes unnecessary the setup of a separate image provider for the upload plugin and using a 3rd party service. Ephemeral file systems are used by many hosts, for example Heroku, that wipes/deletes files generated after their deployment, and since the upload plugin creates new files in the public folder, for the uploads, those files won't remain for too long.

A Node.js server with a minimum version of 12.22.0 is required by Next.js. When using static export an Apache or a Nginx server are enough since only static files are served. This website is not expected to generate a lot of daily traffic with its current features, so the hardware resource requirements are low.

4.1 Hosting Options

While writing this report, the author had already done a full deployment of the project, but only as staging deployment. In order to limit the expenses, only the hosting options that minimize the deployment cost were considered at this stage.

4.1.1 Next.js Hosting

Based on the requirements and the current features of the website, so as to host the Next.js application, the simplest hosting option is to utilize the hosting service of its creator Vercel. A free hosting tier is featured in Vercel's pricing scheme for mostly static websites with low traffic. Unfortunately, the hosting of commercial websites is prohibited by the free tier, which is why it can only be used for staging purposes unless an upgrade to pro tier is justified. Some of Vercel's many features are preview deployments, analytics, global edge functions and more. A website is also secured by Vercel with a free SSL certificate enabling HTTPS connections for the website.

In addition to Vercel the front-end can also be hosted on any host that supports the minimum required Node.js version. Since the popularization of Node.js backends many hosting options for Node.js backends are offered by different companies. A more thorough analysis of the available options is needed when the website must go live later. A few examples of hosting platforms are AWS, Azure, Google Cloud, Heroku, Render, Netlify, Digital Ocean, Cloudflare, Linode.

4.1.2 Strapi Hosting

For the deployment of Strapi a host that also offers databases is recommended to be used. By having these hosted together, the expected distance and connection delay between the server and the database is minimized. Like the deployment of the front-end, the aim at this stage is to be able to host everything with the minimum expenses, even freely is an option.

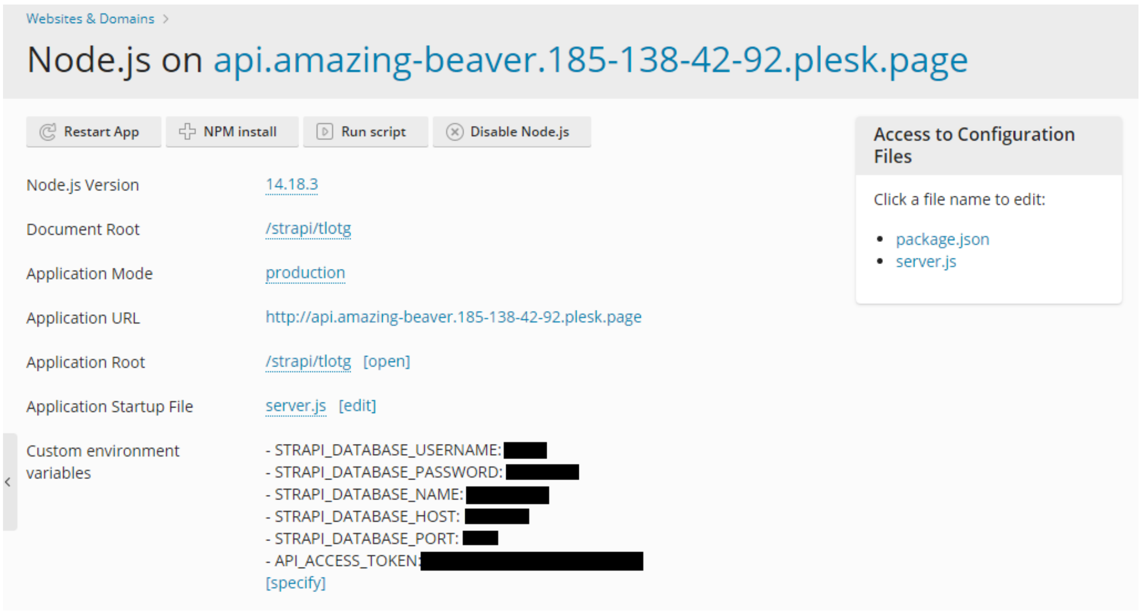
One option is to host Strapi at Heroku. Both Node.js and PostgreSQL are offered by Heroku in its free tiers. The free tiers again have some limitations. Namely the memory is limited to 512 MB, which is sufficient, but the server goes to sleep mode after 30 minutes of inactivity causing a delay for responding to the next request when the server is waking up. Additionally, the database capacity is limited to 10000 rows and 1 GB of storage. These limitations are not a problem at this stage making Heroku a good choice, but a different option is available.

The author was granted access to another hosting provider that with a paid plan. The hosting service provider is called TopHost, a company that offers multiple hosting and domain name registration services. With TopHost it was possible to host Strapi in a Linux server using a single randomly named hostname with one subdomain. After Strapi was deployed it was configured to serve the admin dashboard at the hostname, the API server was configured to listen at the subdomain “api.”. MySQL databases with unlimited capacity are also offered by TopHost, so a new production database was created and connected to the Strapi deployment. The Linux server is managed through Plesk Web Host Edition. From the panel it was also possible to generate an SSL certificate using Let’s Encrypt, a free certificate generation service. This procedure also automatically configures the Linux containers Apache or Nginx server to reverse proxy the Strapi server securing it with a HTTPS connection. A persistent filesystem is used for the Linux container, which permits using the same settings as in development for the media upload plugin and storing all images locally on the server.

4.2 Project Upload

The deployment of Next.js to Vercel is easy and straightforward. All that is needed, is creating a new project at Vercel’s dashboard and importing the GitHub repository of the front-end. Then after defining the necessary environmental variables that are to be embedded in the server-side code, by clicking ‘Deploy’ the platform builds the website and adds it as a deployment. After the initial deployment, every time a push of a new commit to the ‘Main’ branch at GitHub is made, Vercel deploys the latest version automatically. Another Git branch can also be specified for creating a separate deployment, when needed to test some change or a new feature.

Deploying the Strapi server is not that simple. The hosting provider does not have automated tools for Node.js application deployments so some things must be done manually. The Strapi backend is deployed with a random and temporary hostname, since the client had not yet registered a domain name for his website. This temporary domain is used to serve the admin dashboard. The next step was creating a subdomain for the domain. The subdomain’s name starts with ‘api’ since it hosts the Strapi API backend. The upload of the project with all the relevant project files to the server filesystem is done by using an FTP connection. The hosting package has 2 GB of RAM, so it is possible to run a build command on the host in order to build the dashboard. The next step after uploading the files of the project, is the installation of the Node.js dependencies using the command ‘npm install’. A shortcut button for the installation is located at the Node.js management view (Figure 13). Then in turn, the necessary environmental variables for production deployment are set and the Strapi deployment is ready to run.



Node.js on api.amazing-beaver.185-138-42-92.plesk.page

Node.js Version	14.18.3
Document Root	/strapi/tlotg
Application Mode	production
Application URL	http://api.amazing-beaver.185-138-42-92.plesk.page
Application Root	/strapi/tlotg [open]
Application Startup File	server.js [edit]
Custom environment variables	- STRAPI_DATABASE_USERNAME: [redacted] - STRAPI_DATABASE_PASSWORD: [redacted] - STRAPI_DATABASE_NAME: [redacted] - STRAPI_DATABASE_HOST: [redacted] - STRAPI_DATABASE_PORT: [redacted] - API_ACCESS_TOKEN: [redacted] [specify]

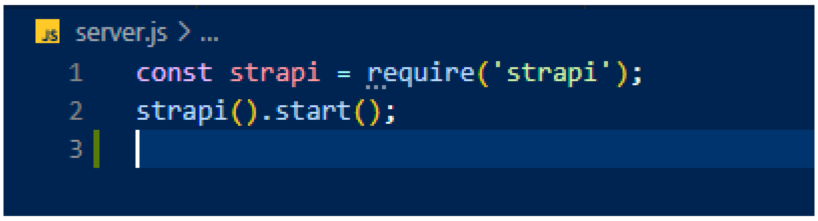
Access to Configuration Files

Click a file name to edit:

- package.json
- server.js

Figure 13. Plesk panel for Node.js configuration

The startup file for running the application can be set at the panel, so a ‘server.js’ (Figure 14) file is created at the application root. The Strapi server is enabled by executing this file with Node.js.

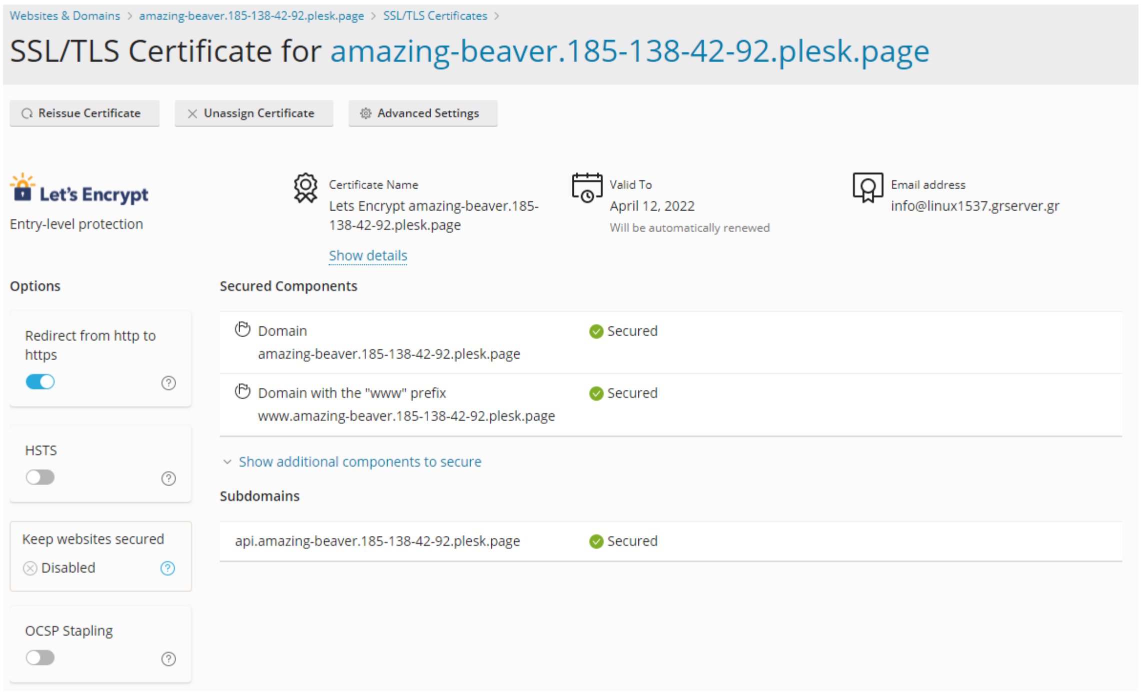


```
server.js > ...
const strapi = require('strapi');
strapi().start();

```

Figure 14. Contents of server.js

After confirming that Node.js is running the last step is to secure the connections to the server with an SSL/TLS certificate. A handy panel for this as well is provided by Plesk (Figure 15). After in turns: selecting the domain and the subdomain, clicking the generate button the certificate is ready, and the server is configured to be used. The server is also set to redirect requests from HTTP to HTTPS with a 301 redirect.



Websites & Domains > amazing-beaver.185-138-42-92.plesk.page > SSL/TLS Certificates >

SSL/TLS Certificate for amazing-beaver.185-138-42-92.plesk.page

Reissue Certificate | Unassign Certificate | Advanced Settings

Let's Encrypt
Entry-level protection

Certificate Name: Lets Encrypt amazing-beaver.185-138-42-92.plesk.page
Show details

Valid To: April 12, 2022
Will be automatically renewed

Email address: info@linux1537.grserver.gr

Options

- Redirect from http to https: on
- HSTS: off
- Keep websites secured: Disabled
- OCSP Stapling: off

Secured Components

Domain amazing-beaver.185-138-42-92.plesk.page	Secured
Domain with the "www" prefix www.amazing-beaver.185-138-42-92.plesk.page	Secured

Show additional components to secure

Subdomains

api.amazing-beaver.185-138-42-92.plesk.page	Secured

Figure 15. The Plesk panel for SSL certificate settings.

After these steps both Node.js servers are up and running in a production environment. Before the website can receive any visitors, the production database must be populated with pages and content.

5 DISCUSSION

The completion of this project took about 2 and a half months. The result appears to be modest but there were several hurdles during development with most of them being of non-technical nature. This thesis was written during a pandemic when governments implemented movement restrictions and people were generally cautious and had to stay indoors with their families. This factor alone limited the ability to interact with the client who also at some point fell sick during the pandemic. The client also didn’t have photos/images of any use for the website and due to the fact that the store had being closed for winter, thus it was not in a condition to be photographed. Despite these problems the requirements set at the start of the project were met. The website can display the product collection of the store thus giving it some visibility as the client initially wanted.

As for the technical implementation the following conclusions are drawn. The choice of the tech-stack is a good one for a custom-built website. However, as stated in the introduction, this dynamic page architecture is too complicated for a small store. Making changes to the structure of the pages on the website is usually not required after the pages are built and approved by the client. At most only some specific text, for example store announcements and open hours can be modified by the owner of the website. The main focus of the website is the display of products and their categories in the CMS and Strapi which takes care of providing an interface for their entry. After that the products are simply displayed in their respective dynamic pages whose endpoint URLs can be statically defined in the filesystem. The complexity and thus the possibility for technical issues is reduced by using a more straightforward approach. If the project had to be rebuilt with the current requirements, the filesystem routing of Next.js should be used to define the pages instead of the current method and the CMS should only store the customization data of each page.

The architecture used in this project is more suitable for a corporate website where developers can build content blocks to be used for example by marketing specialists who are then able to readjust the look and the content of the website without being so depended to developers. After the project was finished one such example was found by the author of this thesis, a corporate website using this approach, by a French company (Finary) in the financial sector.

Since the website is originally intended to evolve to an e-shop, a lot of work is needed in some areas of the website. For starters, an overhaul is needed for the appearance of the website. As it stands, the client's business is not properly represented to its visitors by the website and will probably result in a poor conversion rate if opened to the public in its current state. Also, the appropriate photographs have to be uploaded for all products and the store itself. A task that falls on the client and that limited the design options during development since barely any visual material were available. The SEO of the website is very basic but allows the proper display of the store in search results. However, there are improvements to be made such as presenting the opening hours and inserting more structured data to the pages of the website. A functioning contact form is included on the website, but it lacks modern spam prevention methods, namely a CAPTCHA challenge. A notification should trigger with every form submission, typically in the form of an email message, which has not been implement yet.

Upgrading this website to an e-shop requires some new features to be added. The elementary features that certainly are required are an inventory system, a user registration and a login system, an ordering system, a payment processing system and automated emails. Even a simple e-shop has enough complexity in its implementation and the design should not only be secure and reliable but also should have the appropriate built enabling its future expansion or modification, for example – adding more features as mentioned above.

Building this project was a great learning experience for the author of this thesis as it gave an opportunity to practice important HTML, CSS and JavaScript skills while working on something interesting. Deeper knowledge was also gained about the Next.js framework, and together with Strapi this was the author's first complete full-stack project using these tools.

With the website requiring an image upload and a storage solution some new knowledge about responsive image storage and display in a full-stack website was gained. Experience was also gained about the hosting process of a website backend. Having used services like Vercel before, the chance to use a tool like

Plesk, which allowed tinkering with setting up subdomains, webmail, SSL certificates, file transfer over FTP and more was very welcomed.

In conclusion, despite the difficulties, this project was completed while valuable experience was gained from the work. Thanks to the deeper understanding of the technologies gained from this project, the author is now better equipped to face the challenges of expanding the website to an e-shop, after this thesis, according to the original plan.

BIBLIOGRAPHY

Beal, V. 2001. SEO (Search Engine Optimization). Accessed 23 January 2022 https://www.webopedia.com/definitions/seo/ ()

Biilmann, M. & Hawksworth, P. 2019. Modern Web Development on the JAM-stack. first edition. Sebastopol, CA: O’Reilly Media.

Heroku 2022. Heroku Pricing. Accessed 21 April 2022 https://www.her-oku.com/pricing

Ploesser, K. 2021. REST API Design Best Practices for Parameter and Query String Usage. Accessed 23 January 2022 https://www.moesif.com/blog/tech-nical/api-design/REST-API-Design-Best-Practices-for-Parameters-and-Query-String-Usage

Strapi SAS 2021. Strapi corporate Nextjs template. Accessed 3 November 2021 https://strapi.io/starters/strapi-starter-next-js-corporate

Strapi SAS 2022. Strapi version 3 documentation. Accessed 12 January 2022 https://docs-v3.strapi.io/developer-docs/latest/guides/api-token.html

Tamturk, V. 2016. The Ultimate Guide for Headless Content Management Sys-tems. Accessed 20 January 2022 https://www.cms-connected.com/News-Ar-chive/December-2016/The-Ultimate-Guide-for-Headless-Content-Management

Vercel 2022a. Nextjs documentation. Accessed 1 February 2022 https://nextjs.org/docs

Vercel 2022b. Nextjs documentation Pages. Accessed 1 February 2022 https://nextjs.org/docs/basic-features/pages

Vercel 2022c. Nextjs documentation Dynamic Routes. Accessed 1 February 2022 https://nextjs.org/docs/routing/dynamic-routes

Vercel 2022d. Nextjs documentation getStaticPaths. Accessed 1 February 2022 https://nextjs.org/docs/basic-features/data-fetching/get-static-paths

Vercel 2022e. Nextjs documentation getStaticProps. Accessed 1 February 2022 https://nextjs.org/docs/basic-features/data-fetching/get-static-props

Vercel 2022f. Nextjs documentation getServerSideProps. Accessed 1 February 2022 https://nextjs.org/docs/basic-features/data-fetching/get-server-side-props

Vercel 2022g. Nextjs documentation getInitialProps. Accessed 1 February 2022 https://nextjs.org/docs/api-reference/data-fetching/get-initial-props

WordPress.org 2022. Accessed 18 February 2022 https://wordpress.org