Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of Informatics

Bachelor Thesis

Usage of Headless Content Management System for information portals of public administration

Arozhdan Baibussynov

© 2022 CZU Prague

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

BACHELOR THESIS ASSIGNMENT

Arozhdan Baibussynov

Systems Engineering and Informatics
Informatics

Thesis title

Usage of Heasless Content Management System for information portals of public administration

---

Objectives of thesis

The main aim of this thesis is to analyse the impact of the usage of headless Content Management System (CMS) on the design and programming the web pages and explain what headless CMS is and how to use it through a relevant theory (communication via API services between front and back end).

The partial objectives are such as:
-To conduct a comprehensive literature review, studying the relevant theories on current usage and prospects of traditional and headless CMS (e.g. Drupal, WordPress, Liferay, Joomla, etc.).
-To study challenges, opportunitiees, benefits and implications of headless CMS in the web design.
-To examine the impact of headless CMS on the desing and programming technique.
-To evaluate the proposed ideas, formulate recommendations and make conclusions.

Methodology

The theoretical part of this thesis is based on the author's own research and study of relevant information resources, using qualitative document analysis and external desk research. The practical part will show some usecases of how to work with headless CMS on front and back end (via API communication).

Based on the usecases through a selected theory the conclusions and implications both for theory and practice will be formulated.

The proposed extent of the thesis

30-50 pages

Keywords

Content management system, web pages, information portal, programming, design

---

Recommended information sources

LAURENČÍK, M. Tvorba www stránek v HTML a CSS. Praha: Grada Publishing,
    2019. ISBN 978-80-271-2241-7

MAYEKAR, D. Decoupling Drupal: A Decoupled Design Approach for Web Applications. Berkeley, CA:
    Apress, 2017. ISBN 978-1-4842-3320-7

Expected date of thesis defence

2021/22 SS – FEM

The Bachelor Thesis Supervisor

doc. Ing. Jan Tyrychtr, Ph.D.

Supervising department

Department of Information Engineering

Electronic approval: 1. 3. 2022

Ing. Martin Pelikán, Ph.D.

Head of department

Electronic approval: 7. 3. 2022

doc. Ing. Tomáš Šubrt, Ph.D.

Dea

Declaration

I declare that I have worked on my bachelor thesis titled "Usage of Headless Content Management System for information portals of public administration" by myself and I have used only the sources mentioned at the end of the thesis. As the author of the bachelor thesis, I declare that the thesis does not break any copyrights.

In Prague on date of submission   12.03.2022

Acknowledgement

I would love to thank my supervisor doc. Ing. Jan Tyrychtr, Ph.D., for all the help he provided to me within short period of time. For the assistance with structuring and organizing the job!

Usage of Headless Content Management System for information portals of public administration

Abstract

English

The purpose of this bachelor thesis is to learn more about "Headless" CMS and design a new website that uses data created and managed by the CMS. We should be able to define the fundamental difference in comparing more mature Traditional CMS to modern Headless CMS.

The main goal of literature review is to clarify what CMS is, then define Headless CMS, understand how it is different from "normal" CMS, define benefits, pros and cons of each approach.

The goal of practical solution is to create an instance of headless CMS. Understand what is happening behind the scene. With this, the aim is to create a fullstack application running on Headless CMS.

Czech

Keywords: Content management system, web pages, information portal, programming, design

Využití Headless Content Management System pro informační portály veřejné správy

Cílem této bakalářské práce je seznámit se s CMS "Headless" a navrhnout nové webové stránky, které využívají data vytvořená a spravovaná tímto CMS. Měli bychom být schopni definovat zásadní rozdíl v porovnání vyspělejších tradičních CMS s moderními Headless CMS.

Hlavním cílem rešerše literatury je objasnit, co je to CMS, dále definovat Headless CMS, pochopit, čím se liší od "normálního" CMS, definovat výhody, klady a zápory jednotlivých přístupů.

Cílem praktického řešení je vytvořit instanci bezhlavého CMS. Pochopit, co se děje za scénou. Díky tomu je cílem vytvořit fullstack aplikaci běžící na Headless CMS.

Keywords: Systém správy obsahu, webové stránky, informační portál, programování, design

Table of content

1 Introduction

In order to grow, businesses must have so called "representative branches" online, like mobile applications, social networks, e-shops, corporate websites. You must put yourself as widely as possible on the web by all possible ways. You also must be flexible, able to adjust quickly and easily move to new platforms.

Traditional approach is that for each platform its own architecture is developed, content is prepared, and the interface is configured. Development and support requires significant resources. This limits the ability of companies to grow and keep quality.

A new generation of CMS solves the issue. Content is now created, stored and edited regardless of the technical solutions used to present it on client equipment (browser, smartphone, smartwatch)

The way how "Traditional" CMS combines both "front-end" and "back-end" causes some inconvenience. The content is deeply associated with specific architecture and technologies.

Headless CMS is a fundamentally different management system. It is only responsible for the content itself, which can be used on many platforms and even synchronously. Now the backend is not associated with the frontend.

2 Objectives and Methodology

2.1 Objectives

The main aim of this thesis is to analyses the impact of the usage of headless Content Management

System (CMS) on the design and programming the web pages and explain what headless CMS is and how to use it through a relevant theory (communication via API services between front and back end).

The partial objectives are such as:

 - To conduct a comprehensive literature review, studying the relevant theories on current usage and prospects of traditional and headless CMS (e.g. Drupal, WordPress, Liferay, Joomla, etc.).

 - To study challenges, opportunities, benefits and implications of headless CMS in the web design.

 - To examine the impact of headless CMS on the design and programming technique.

 - To evaluate the proposed ideas, formulate recommendations and make conclusions.

2.2 Methodology

The methodology is based on analysis and study of suitable literature and electronic resources, understanding the principals of reviewed theory.

- Get familiar with the "content", "content management system", and two main approaches of managing the content of applications.
- Study most popular CMS systems, see how they are different from each other.
- Define "Headless" CMS and find out the benefits of this approach.
- Describe Client-Server architecture in case of headless approach.

Basing on this theoretical fundament:

- Create a self-hosted instance of "Headless" CMS, review the admin UI panel, as well as the processes running under the hood.
- Design fullstack application, that retrieves data from CMS and renders it on client, so that we see application, database and file structures in headless approach.
- See in practice, how frontend application can consume and display incoming content
- Describe conclusions for gained results

3 Literature Review

3.1 Content and Content Management Systems

3.1.1 What is a "Content" itself?

First of all, in order to define why we need a CONTENT management system, lets define the content. Content is information produced through editorial process and ultimately intended for human consumption via publication. Content is created and managed, then it is published and delivered (Barker, 2016). Any piece of information and/or data that we can get out of the page we access was created and published by someone in some place. Content may vary from images and videos to texts and tables. The reason we use internet and use our browser is to consume the content. The hardest thing for content creators is to effectively manage it and this is the reason for Content Management Systems to be considered. According to the W3Techs, more than half of the sites are developed with the CMS on the board. (w3techs.com, 2021)

3.1.2 Content management system

A content management system (CMS) is a software package that provides some level of automation for the tasks required to effectively manage content. (Barker, 2016). In practice, CMS is a web/desktop application that allows users to build and maintain website content. Whichever CMS you choose, you will work on the site in a special personal account - control panel. Sometimes this place is also called the admin or console. CMS gives you an opportunity to create, store, and modify a content. The reason why user usually does not even need to have any IT background, is because all of the Content Management System nowadays provides you with a very convenient, easy-to-understand graphical user interface. There you can manage all aspects of your website, you can create and edit content, add images and videos, customize the overall site design. WordPress, Shopify and Drupal are the most popular CMS currently on the web. (w3techs.com, 2021)

3.2 Traditional content management system

Nowadays, "traditional" approach is more common and according to "websitesetup.com" (Schäferhoff, 2021), the most popular true traditional CMS

systems are: "WordPress", "Joomla" and 'Drupal". So the most popular „traditional"
systems are also the most popular in general

WordPress:

Their official website tells us that 42% of the web is built on WordPress. Most
famous internet bloggers, small and large businesses, Fortune 500 companies choose
WordPress over all other options combined. (MacDonald, 2020). But how does it
work?

WordPress is the brain of your website. When user accesses a WordPress-powered
website, the WordPress software generates and delivers a fresh new page to your
visitor. Instead of creating a web page on your own, you give WordPress your raw
content you want published as an article, a product listing, a blog post, or something
else. Then, when someone visits your site, WordPress assembles that content into a
perfectly tailored page. (MacDonald, 2020). WordPress allows you to create
different templates, use multiple styles and save money on designers and
programmers. But at the same time it requires a web server to make its computations,
combine content and markup together to send you a valid page, when you make a
request.

Drupal:

Drupal is a traditional content management system, that was created by over 4,500
contributors. It does not require much specific knowledge and you don't have to be
an experienced "Drupalist" (so called Drupal user to build highly scalable and
enterprise-ready websites. Drupal is a PHP-based content management system, that
takes your content and combines it with HTML markup, CSS styles, and JavaScript.
Same as WordPress it pre-generates a static html response before it gets sent to you
(Nick Abbott, Richard Jones, Matt Glaman, Chaz Chumley, 2016)

3.2.1 What is common for traditional systems

We can already notice a common thing for these two CMS's. Actually, "Joomla", "ModX", "October CMS" and many other most popular systems use same approach. They all contain both HTML structure and content. So they insert data inside the HTML tags and send you a normal .html file as a response (with some styling and JS). That is how traditional Content Management Systems work. It is essential for them to contain all the page structure.

3.3 Headless CMS

Headless CMS is a way different from mentioned earlier systems. It is only responsible for the content and does not serve HTML structure. You divide your "body" from different "heads", so you gain a convenient way to manage content, that is displayed different way on different platform. Individual frontend can be created for each environment, that gets data from a single place.

Maintaining and delivering content to different platforms is carried out from one interface, so it is more convenient. The content can be configured for each technology to be displayed correctly on small smartphone as well as on big TV displays. This system is built from scratch and is used to store and manipulate only the content and with a set of tools. It provides creators with an admin panel to collaborate on content.

The content is stored in the database it maintains. There are two types of databases nowadays: relational and non-relational. Those would be covered in this thesis. Data exchange most often occurs in the "universal" JSON format, which allows you to adapt to any new frontend (Raevskiy, 2020). The frontend is only responsible for user interaction, API is used to manipulate and consume data.

3.4 Reasons to develop „Headless" management systems

Traditional approach came first and is way more popular even nowadays. So the question is: "Why did people need to change anything?". With all the theory described before, the benefits and disadvantages of this approach must be defined.
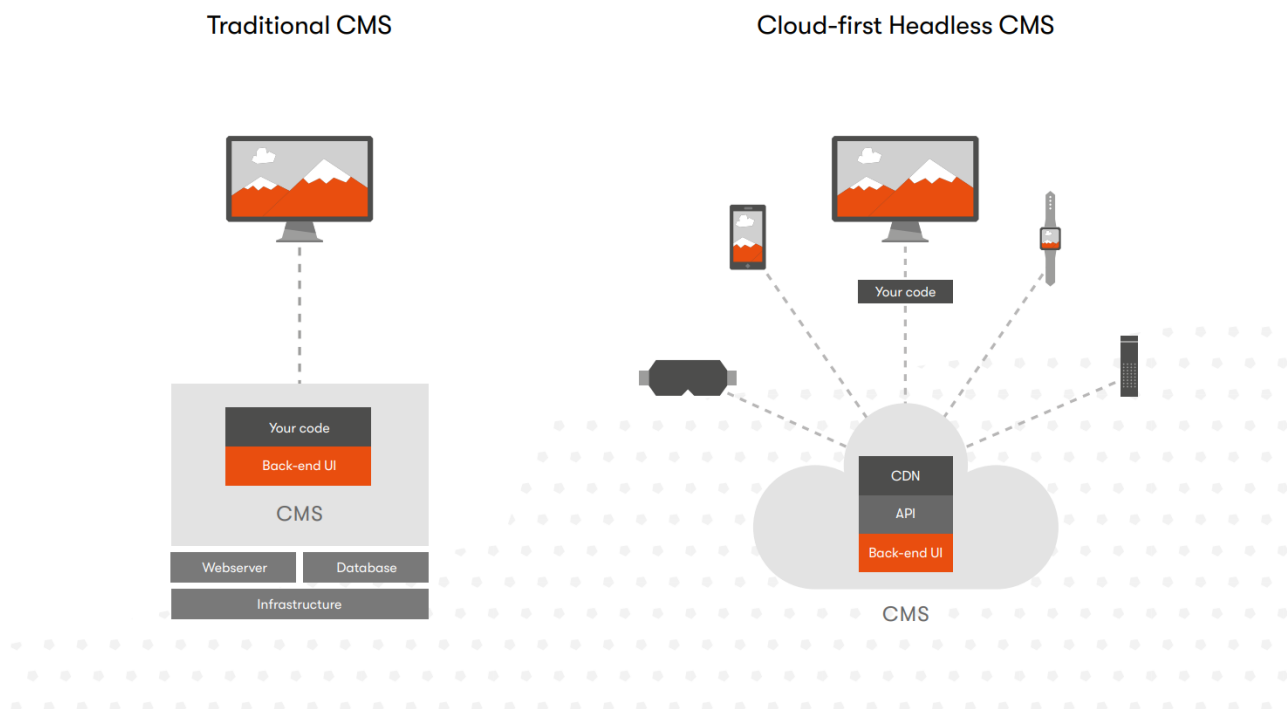
Obrázek 1Traditional vs Headless
(source:https://query.prod.cms.rt.microsoft.com/cms/api/am/binary/RWLvVL)

3.4.1 Advantages of "traditional" approach:

- Everything is stored in the same place.
- Easy access to both content and markup
- Content owner can preview the way page will look like before publishing the page
- No need to have strong knowledge in programming. You don't have to build frontend to consume your content.
- Better SEO. You can set all the required meta data, page title and all other

SEO info in CMS. SEO stands for "search engine optimization". It makes your website listed on top of search results in search engines like Google, Bing, Seznam and others. (Amerland, 2013)

- Usually a simple response. In the end of the day, it is only a normal HTML, CSS and JS files, that computers have first met many year ago, and know how to parse them.

- Popularity. If you ever run into any issue, 100% someone has already done it before. There is a great community in the web, that can help you with any problem.

- Reliability. Traditional approach has longer history than any other.

3.4.2 Disadvantages of the approach:

- Everything is stored in the same place. Yes, also is a disadvantage. Content management system by its name means the management of content. No HTML markup should be involved.

- Usually slower, comparing to Headless CMS's. It takes time to insert your texts, images and any other content into a html file, Include styles, scripts, combine all together and send you a response.

- CMS for each application individually. Since you app has data and structure closely related, it will be incredibly hard to connect multiple different applications to the same CMS. So you will need to maintain one CMS for web, one for mobile app and so on.

3.5 Integration of Headless CMS

Many times it was mentioned, that headless approach is different from the traditional one mainly by the way of integration content into the structure. Since new generation of management system does not include the markup and styling of the content but only contains data itself, you must create some frontend to consume and display your content.

Headless CMS are all API – based. This means, that they provide you with a specific integrated interface to fetch and manipulate the data.

(Palas, 2019)

3.5.1 What is API

API (Application programming interface) is a set of specific features, functions implemented by a specific software as an interface for communication with it from outside. In simple words, it's an instruction for the consuming application with the access how to use some functionality of the API provider. "I must be addressed this way, and I am obligated to do this and that if you have the rights to ask for it"

3.5.2 REST API

The name Representational state transfer (REST) was coined by Roy Fielding from the University of California (Yellavula, 2020). It allows for different clients to communicate with a single server via API calls to the specific URLs (REST endpoints).

Each type of operation uses its own HTTP request method:

1. receiving – GET
   a. GET request / rest / articles - getting information about all users
   b. GET request / rest / articles / 123 - getting information about the article with id = 125
2. adding – POST
   a. POST request / rest / articles - add a new article
3. modification – PUT
   a. PUT request / rest / articles / 123 - change information about article with id = 125
4. deletion – DELETE
   a. DELETE request / rest / articles / 123 - deleting an article with id = 125

This is a way you address the API and ask to do something for you.

3.5.3   URL, REST API endpoints

URL (Uniform Resource Locator) are used as the locators in World Wide Web. It is like a virtual address to the specific part of the internet.

It was very useful for designing the REST API, because you can easily rich very specific part of your webapp using the URLs. So called RESPI API endpoints are created with this technology that allow you to communicate with the server.
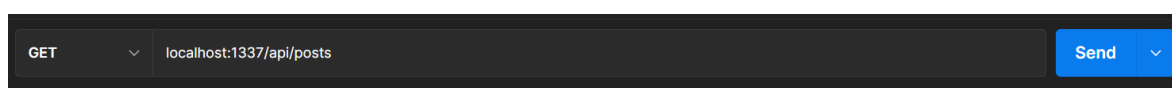
Obrázek 2 Example of REST API Endpoint (self-made)

In the example above, we can see that we address our local machine (localhost) by a specific URL (/api/posts). The second part is the endpoint for our REST API.
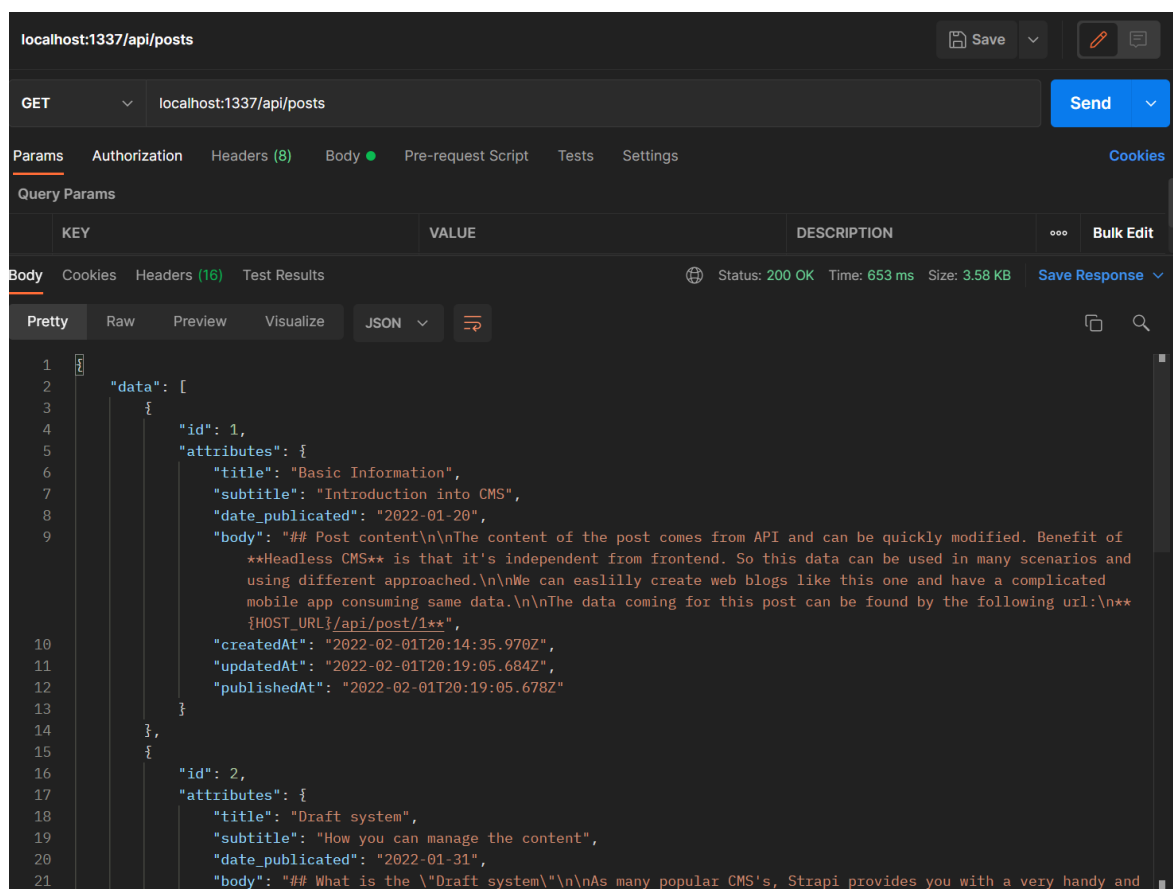
Obrázek 3Response with REST API (self-made)

This is an example of the response coming with the REST API. We can clearly see,

---

that by sending the GET request to the server, we ask it to grab all the posts in our database and send them back in JSON format.

3.5.4 GraphQL API

GraphQL is a query language for the APIs. A GraphQL query asks only for the data that it needs. It is a completely different approach since the structure and amount of data is determined by the client application (Eve Porcello, Alex Banks , 2018).

```
1 ▾ query{
2 ▾   posts{
3 ▾     data{
4         attributes{
5           title
6           subtitle
7         }
8       }
9     }
10  }
```

In this example we query our server and ask for all the posts titles and subtitles.
The client specifies exactly what data it wants to receive using a declarative, graph-like structure that closely resembles the JSON format.

Obrázek 4 GraphQL Query example (self-made)

The client asks for three fields (title, subtitle). But it can request both one field, for example name, and an arbitrary number of fields that are defined in the user type on the GraphQL server.

3.6 Databases

Now, it is clear how the data can be fetched and integrated. Another big question is how is it stored and maintained on the server. How do the REST or GraphQL API access data and send it in any convenient format to the consuming software.

Both traditional and headless systems use databases to store data. Actually, all servers that provide data use any kind of database to maintain and provide raw data.

Database is an organized structure that is designed to store, process and modify a large amount of information. (Petrov, 2019).

The primary purpose and goal of any database is to effectively store data and make a reliable availability of it to the users. Databases are used as a primary source of any data, making it possible to share data across different parts of the software.

3.6.1 SQL

SQL (Structured Query Language) is a standard language for accessing and manipulating databases. It allows us to perform queries in order to review, manipulate and access the data using human readable language. It became a standard of the ANSI (American National Standards Institute), and of the ISO (International Organization for Standardization) in 1987.

(Petrov, 2019)

```
SELECT column1, column2, ...
FROM table_name;
```

This is an example of a simple select query, that requires to return "column1" and "column2" from the table named "table_name". It is important to mention that you don't have to write these queries on your own, the CMS system does it for you when you access its API.

On these principle, the API system retrieves data from the database and creates a response in JSON format. So it is more convenient to consume on any kind of frontend and to render in with different technologies.

3.7 JavaScript

Both frontend and backend of the project described in the practical part are running on JavaScript(JS). JavaScript is one of the most popular programming languages, that mostly run on the web. Actually, all web browsers can read and execute JavaScript, the all include JavaScript interpreters, that makes it the most deployed programming language in history (Flanagan, 2020).

The name "JavaScript" does not come from "Java". These are completely different and unrelated programing languages. First one was designed to run on web and Java is mainly used to run on desktops or to develop mobile apps.

3.7.1 Client-side JS

Main purpose of JavaScript is to run on the client (web browser) and add the interactivity to static web pages. It lets you make functional components such as interactive buttons, slideshows, online calculators and much more units you can "touch". As JavaScript is widely used, it has a big number of frameworks and libraries. The most popular JS frameworks : Angular (one from google), React JS (Facebook) and Vue. For the practical part, the frontend is running on React JS. It is the most popular one, so has a lot of tutorials, plugins and even own frameworks like Next JS.

React was created by Facebook developer Jordan Walke in 2012. It allows to fetch, map and render incoming data the way we specify. JSX (JavaScript Syntax Extension) syntax is used for generating and rendering HTML templates inside JavaScript functions. (Banks, et al., 2020). So it can be called JavaScript library for building user interfaces.

3.7.2 Server-side JS

The aim of this thesis is not to describe the client side of this architecture, but the way Headless CMS run. But it was said many times, that JavaScript is a Client-side programming language that runs and executed by browsed. So the question is how it can be useful for management system that obviously run on the server, not client.

The answer is simple, developers liked JS that much, so they decided to use it both for frontend and backend development. With this in mind, NodeJS was designed. This allows developers to write the whole app with one language, not wasting time and resources to learn and include new scripting languages. One developer now is

able to do both: Frontend and Backend development. Fullstack developers nowadays are not that rare, as it used to be, because modern technologies make it easier to work with both servers and browsers. (Herron, 2020)

3.8   Strapi Headless CMS

3.8.1   Introduction to Strapi

Strapi is an open-source headless CMS, chosen to demonstrate the power and principals of the headless approach. The original purpose of Strapi was to help bootstrap the API. It was created in October 2015 by Aurélien, Jim, and Pierre as a bachelor thesis (2015).   Nowadays, it provides developers with a fast, simple to create and use the API system. Content types and the access can be created and modified with the provided Administration Panel, so it is a very quick way to create the REST and GraphQL API with a convenient graphical user interface.

3.8.2   Advantages of Strapi

It is important to specify the benefits of Strapi CMS and reasons why it was selected for practical solution.

Why Strapi?
- Open-source solution. It is 100% free to use and can be stored and running on your own local machine.
- Self-hosted solution. As it was mentioned, you can get the source code of Strapi and run it on your machine. So it can be very deeply customize and protected from outside attacks.
- Functionality. Strapi has a giant preinstalled plugins (like content-type builder, user permissions and others), and you can also find or create a bunch of different ones.
- NodeJS. Strapi is built on top of NodeJS. So it uses familiar to many developer JavaScript as a backend language, what makes us easier to use and manage it.

3.8.3 Overview of Strapi

As most of the CMSs, Strapi has its own Admin panel where non developer users without coding skills can manage and maintain content of their apps. The user interface is done using React JS.

Admin panel consists of a set of different components that must be covered in this thesis.

Dashboard
https://*strapi_base_url*/admin
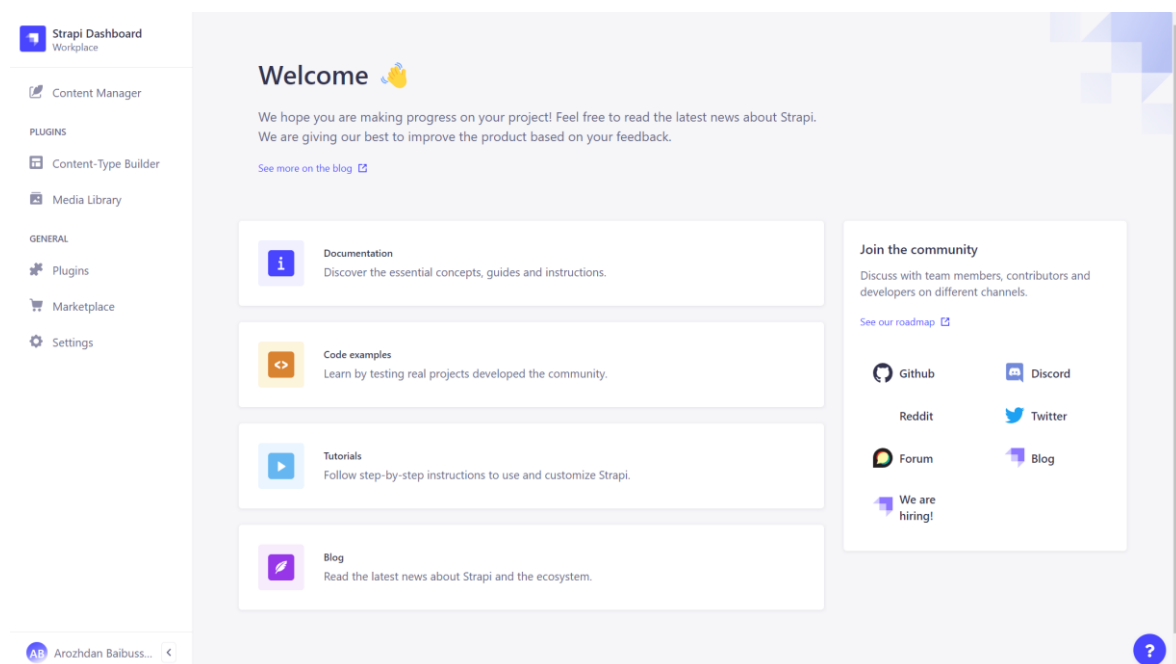
Obrázek 5 Strapi Dashboard (self-made)

Provides with the news and releases, related links, community platforms and the overview of collection types, single types, plugins and general.

Content manager
https://*strapi_base_url*/admin/content-manager/
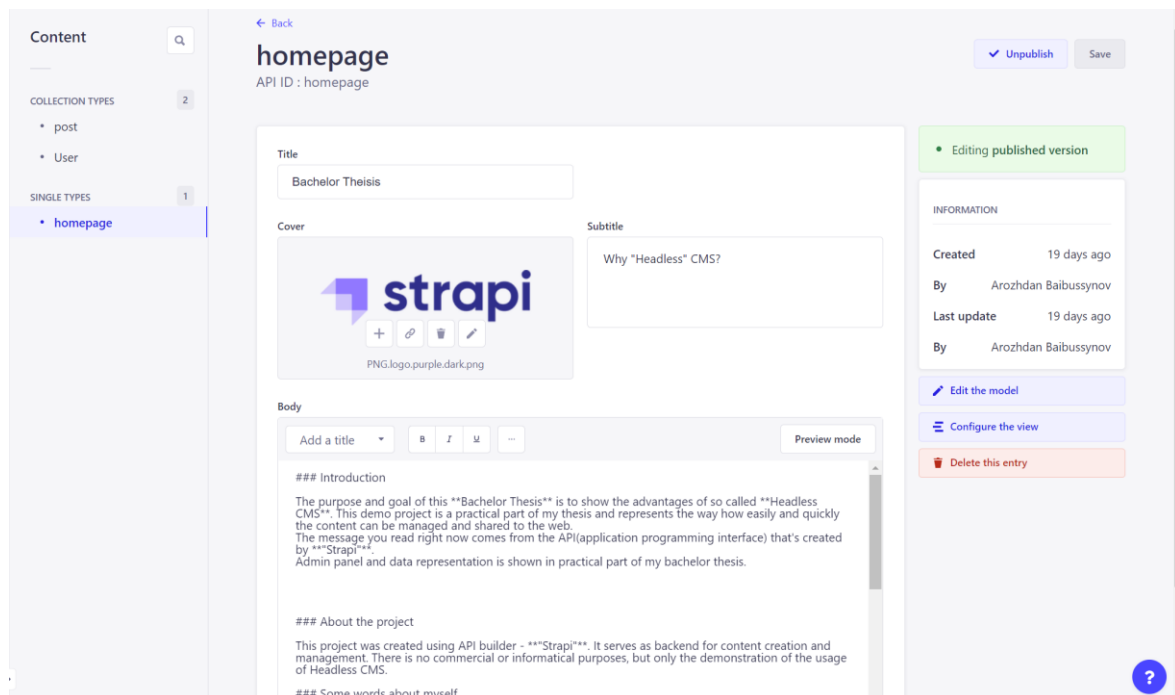
Obrázek 6 Content manager - Sinle types (self-made)

It is an overview of all the content maintained by Strapi. Content in general is divided into two groups: "Collection" and "Single" types. These both may contain same fields, but have a fundamental difference.

Single types, displayed in the figure, is a most standard and "normal" type in Strapi. It is the best way to manage specific page content (like Homepage, About page and so on). Those are directly accessible from the main navigation of the admin panel. Unlike collection types that have multiple entries, single types are not created for multiple uses. In other words, there can only be one default entry per available single type.
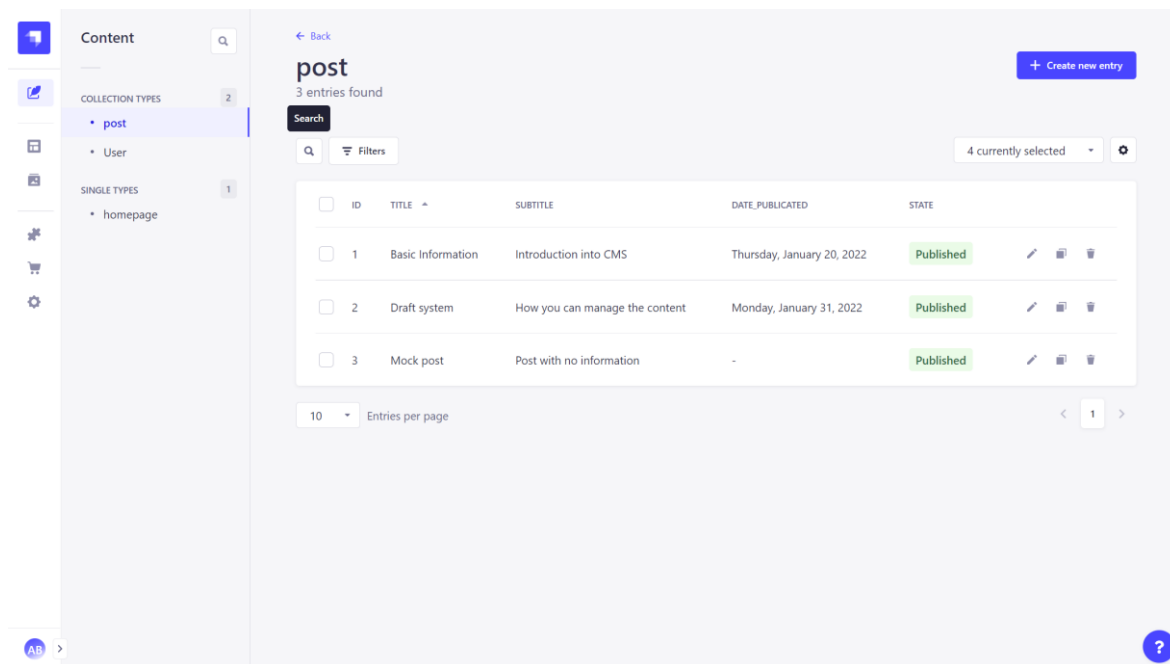
Obrázek 7 Collection types overview (self-made)

Used to create types repeating the same type of content like blog posts, products, users or any list of content you can think of. Collection types category of the Content Manager displays the list of available collection types which are directly accessible from the main navigation of the admin panel. For each available collection type multiple entries can be created. This is why each collection type is divided into 2 interfaces: the list view and the edit view. The list view of a collection type displays all entries created for that collection type.

Content-type plugins

http://*strapi_base_url*/admin/list-plugins/
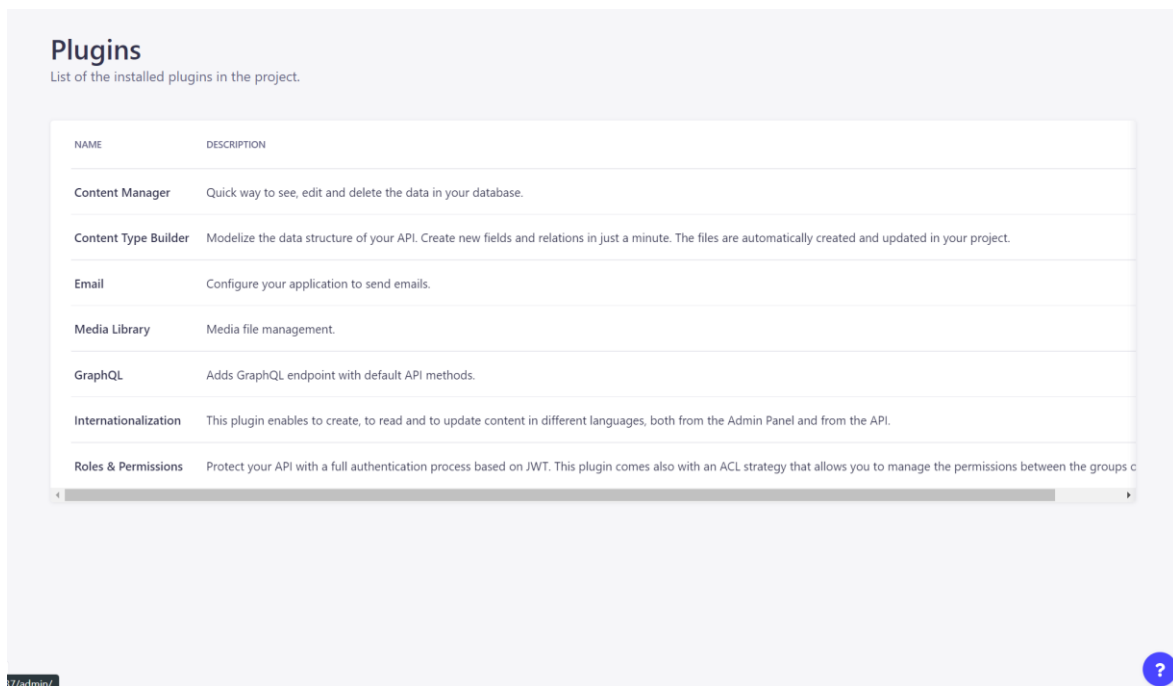
Obrázek 8 Strapi plugins list (self-made)

The list of installed plugins on specific Strapi instance. New project by default has some pre-installed ones, but others can always be add or deleted.

For the practical solution, GraphQl plugin was installed, so I could show the way it works.

Content-type builder

http://*strapi_base_url*/admin/plugins/content-type-builder
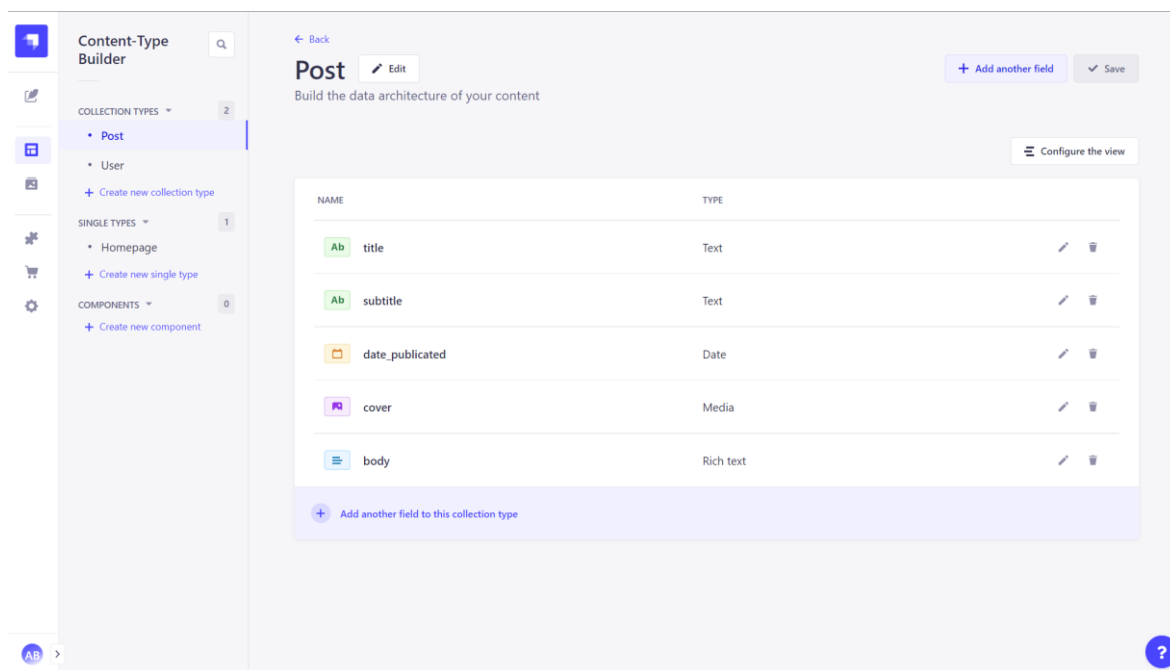
Obrázek 9 Content type builder (self-made)

The Content Manager is a core plugin of Strapi. It is a feature that is always activated by default and cannot be deleted. It is accessible both when the application is in a development and production environment.

The Content Manager is divided into 2 categories, both displayed in the main navigation: Collection types and Single types. Each category contains the available collection and single content-types, which were created beforehand using the Content-Types Builder. From these 2 categories, administrators can create, manage, and publish content.

3.8.4   File structure in Strapi CMS
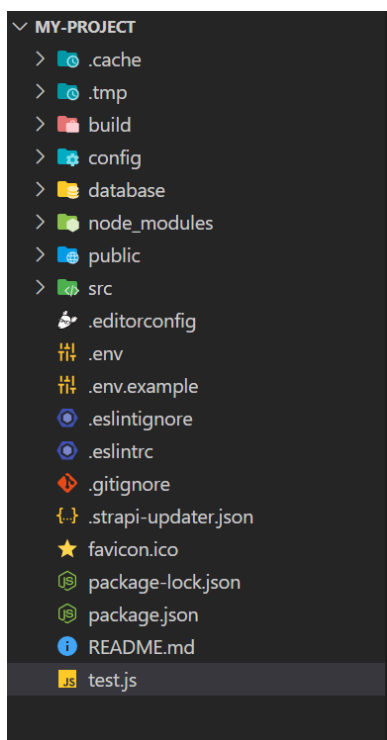
Obrázek 10 File Structure overview (self-made)

Once a new instance of Strapi is installed, the following file structure appears on the machine. The most important directories and files are described in this thesis.

API – is the place where all our models and their setting are stored.
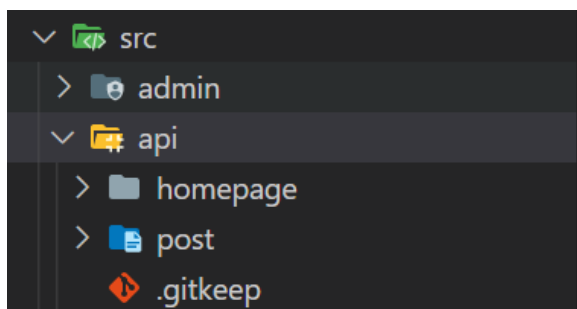
Obrázek 11 API directory (self-made)

It duplicates all the collections and single types, we have in our Admin panel. This is the place where our required fields are defined. On each manipulation with the content type builder, this directory in modified.

Routes.js file is the place to manage the REST API endpoints, and the handlers for it. It means that once we visit <u>https://*strapi_base_url*/posts</u>, Strapi will trigger

that and using the "post.find" controller send us a response.

We can always add more controllers and specify our custom routes for that if we need. For that, all we need is to add to "routes" directory JavaScript file with some scripts.

```javascript
// api/post/routes/custom-posts.js

module.exports = {
  routes: [
    {
      method: 'GET',
      path: '/posts/customRoute',
      handler: 'post.exampleMethod', // custom-posts.js: exampleMethod
      config: {
        auth: false,
      },
    },
  ],
};
```

We create a configuration for our custom route, specifying all RestAPI required options (method, URL/path, require or not authentication) and we also set the handler for this route. It will say our server what to do, once it triggers a request to this route.

```javascript
/**
 * api/post/controllers/post.js
 *  post controller
 */
const { createCoreController } = require('@strapi/strapi').factories;

module.exports = createCoreController('api::post.post', strapi=>({
  exampleMethod: ctx=>{
    return 'Hello, World!';
  }
}));
```

In our case I have created an "exampleMethod" that returns a static string. But obviously it can contain some computations, requests to database, requests to external API's and so on.

With all this set, we can test our custom API endpoint:
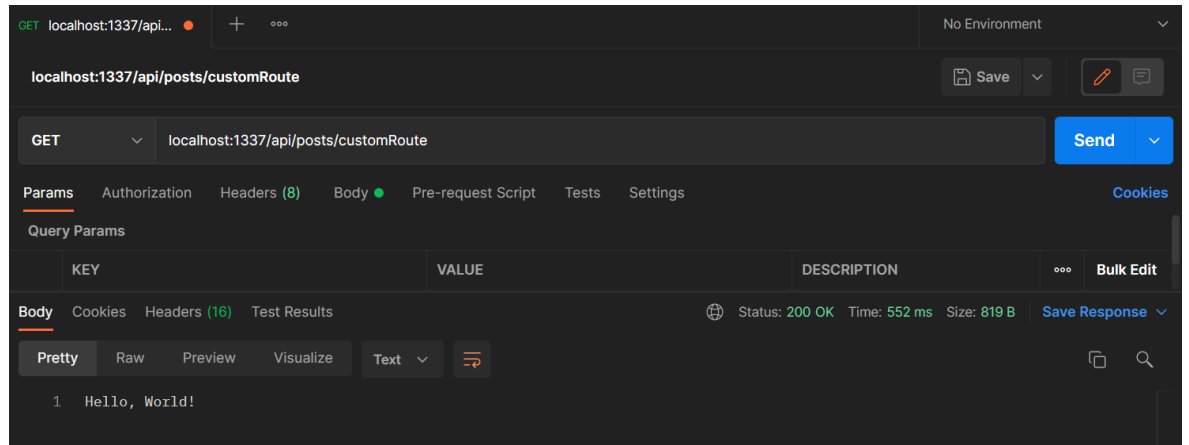
Obrázek 12 Strapi custom endpoint(self-made)

Config – Global settings and preferences are stored in this directory. This is where URL, port (by default that's 1337), database configurations can be modified.
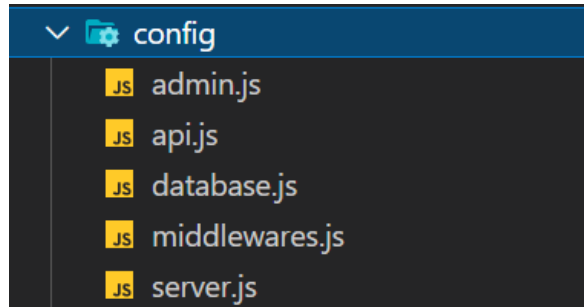
Obrázek 13 Config (self-made)

SERVER.JS

```
module.exports = ({ env }) => ({
  host: env('HOST', '0.0.0.0'), // if not in .env file, then
localhost
  port: env.int('PORT', 1337), // if not in .env file, then 1337
  admin: {
    auth: {
      secret: env('ADMIN_JWT_SECRET',
'8c57b4c67c4066a82888794ce26cfdaf'),
    },
  },
});
```

Node Modules – directory with preinstalled packages.
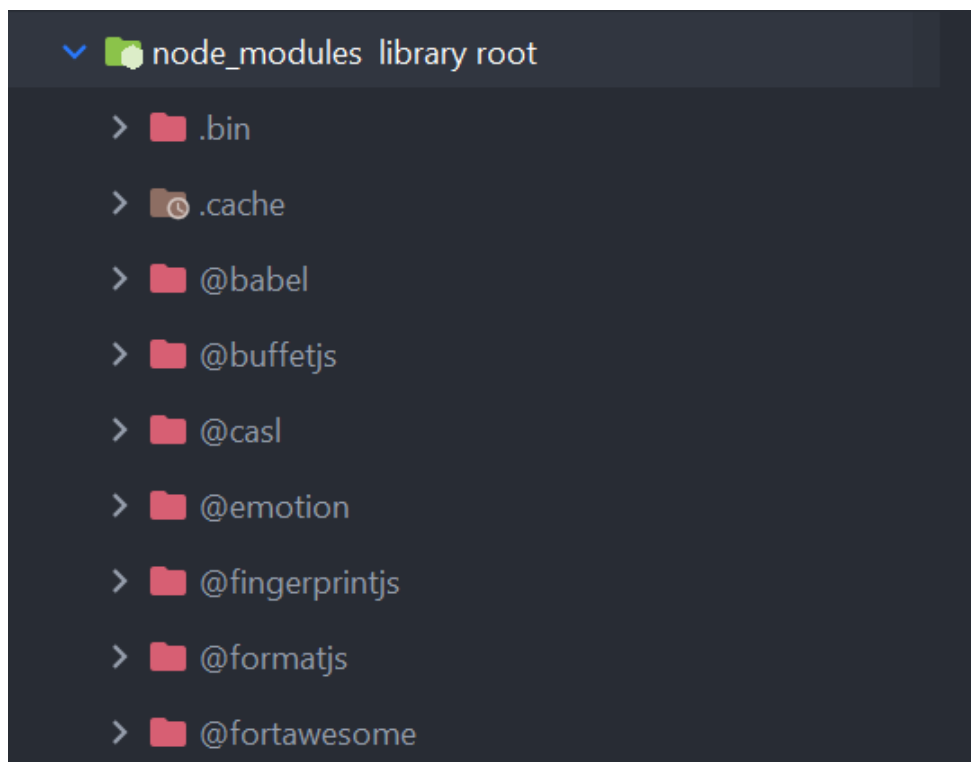
Obrázek 14 Node Modules (self-made)

Basically is the folder we wouldn't touch often. All our packages and dependencies are stored here. Now there are more than 100 subfolders with packages Strapi requires, and obviously we would not go through them.

Public – publicly accessible directory.

Obrázek 15 Public directory (self-made)

Usually contains images, downloadable files.

3.9 Node Package Manager

NPM stands for Node package manager. That's a great tool for JavaScript developers.

It is a registry / collection of node modules. So you can install and use some plugins / packages developers have built in advance and share for free use.

NPX is a package runner. NPX – Node package executer. It runs a command of a package without installing it explicitly.

(Herron, 2020)

4 Practical Part

This chapter is a summary of the process of developing fullstack (frontend and backend) online blog project using Strapi CMS as a backend administrational system, client-server communication process, developer experience and entity relation diagram that shows the database structure designed by me and created by Strapi CMS.

4.1 Backend development process

This is the process of designing and creation of the Strapi instance for managing and manipulating data. Every development starts with the architecture. The aim of this practical solution is to create a web blog. So it is required to define all the entities involved for the best content creation and management.

4.1.1 Entities required
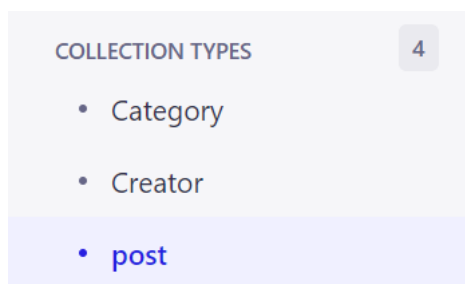
Obrázek 16 Entities for practical solution (self-made)

The most essential components of my blog website are following:

- Posts themselves.
- Author of the post
- Categories the posts are related to

And I also have created a "homepage" entity, so we can provide our web blog with some introduction and the "hottest" articles we want to display on first page.

Obrázek 17 Homepage entity (self-made)

4.1.2 Data structure

Obrázek 18 ERD (self-made)

ERD (Entity Relationship Diagram) is the best way to represent the data structure and data relationships in visual form. The diagram above is a representation of the real database that was generated by Strapi for us. It did not require a single line of SQL query from our side, but was created for us. There are much more tables, but most of them are system tables for Strapi to work well.

But, with the SQL database browser, we can make sure, that the tables have been created exactly as we needed them to, just by using a convenient admin panel, letting CMS handling the rest for us.

Obrázek 19 Database table example 1 (self-made)

Obrázek 20 Database table example 2 (self-made)

4.1.3 Post entity

Core component in our architecture. It has the most relations to other entities and actually is the purpose of any web blog. So I think it is the best to demonstrate how Strapi CMS handles the process of creation of such an entity.

The following structure for post was created in admin panel using a Content type builder.

Obrázek 21Content type builder - post (self-made)

Once the save button was clicked, the process of restructuring of file structure and database manipulations started.
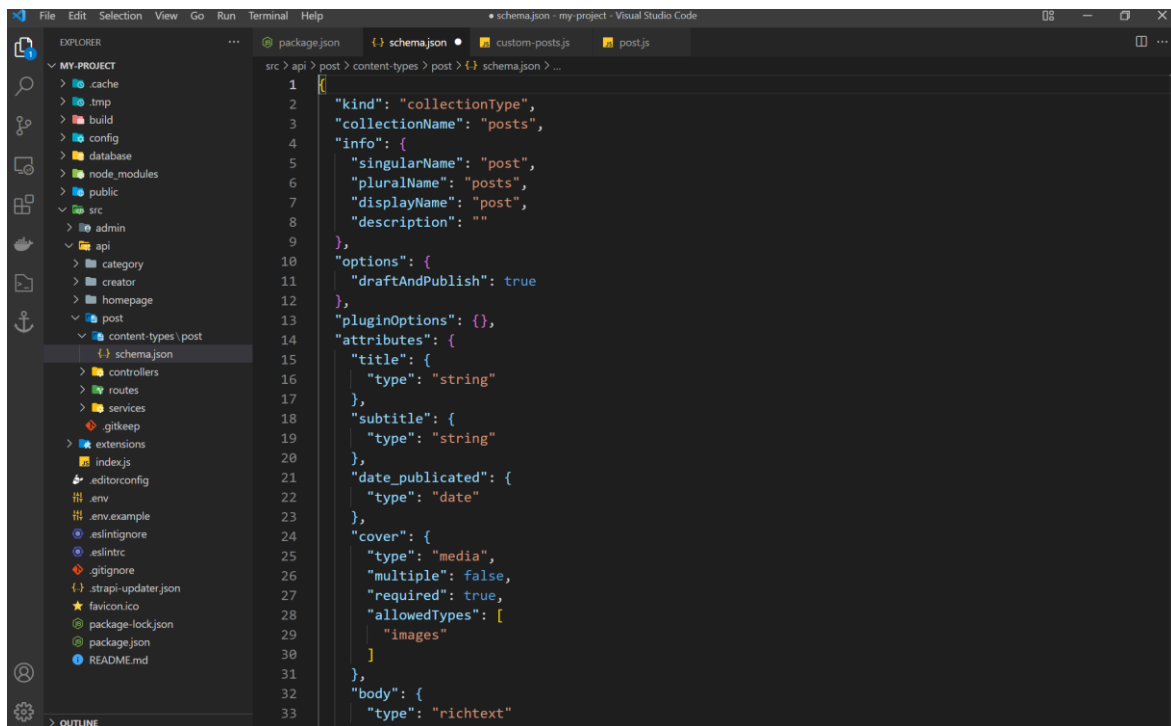
Obrázek 22 Post schema file (self-made)

| id	title	subtitle	date_publicated	body	created_at	updated_at	published_at	created_by_id	updated_by
Фил...	Фильтр	Фильтр	Фильтр	Фильтр	Фильтр	Фильтр	Фильтр	Фильтр	Фильтр
1	Basic ...	Introduction ...	2022-01-20	## Post ...	164374647...	164660521...	1643746745...	1	
2	Draft ...	How you can...	2022-01-31	## What i...	164374742...	164660435...	1643754468...	1	
3	Mock post	Post with no ...	NULL	## What i...	164375107...	164375447...	1643754471...	1	

Obrázek 23 Post database table (self-made)

We can see that the schema of post entity in file structure, database table and in Strapi are same once again.

Same happened for each content type, created with Strapi CMS. Same will happen on each content type modifications.

4.2   REST API in action

The concept of RESTful API was covered in theoretical part of the thesis. In this section, REST API endpoint for fetching content is demonstrated. Since the API requires the rights to be called from outside, it is required to set them in Strapi admin panel.
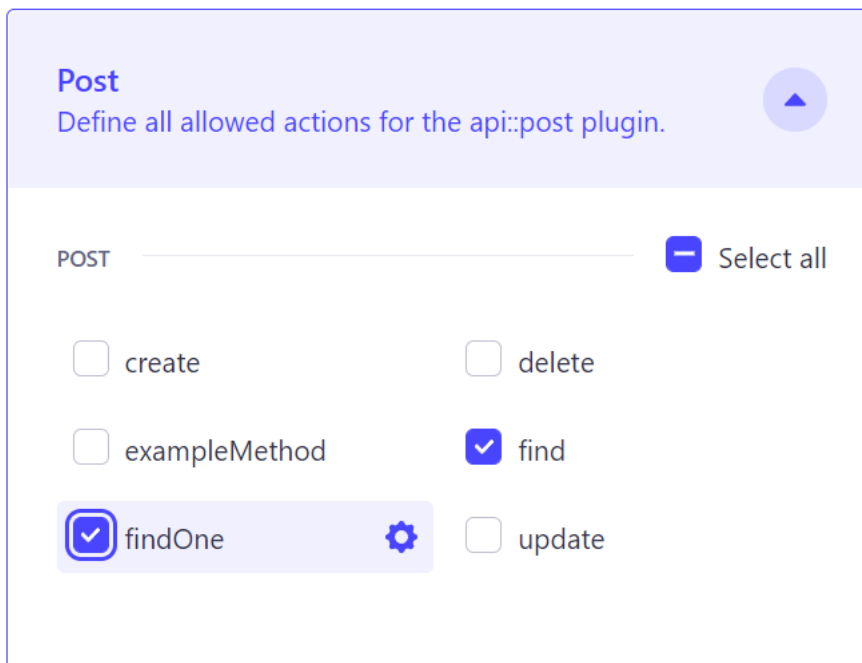
Obrázek 24 Access rights (self-made)

This way, we make post collection publicly available.
/api/posts/:id

Accessing this endpoint, we can now receive posts, stored in database

Obrázek 25 Rest API Posts collection (self-made)

4.2.1 Populating the response

At this moment, related field are not included into the response. It is done this way on purpose, so the response is simple and quick. But it is always possible to include required related data, by adding special query parameters to the request URL.

```
{
    - data: [
        - {
            id: 1,
            - attributes: {
                title: "Basic Information",
                subtitle: "Introduction into CMS",
                date_publicated: "2022-01-20",
                body: "## Post content

The content of the post comes from API and can be quickly modified. Benefit of **Headless CMS** is that it's independent from frontend. So this data can be

We can easilily create web blogs like this one and have a complicated mobile app consuming same data.

The data coming for this post can be found by the following url:
**{HOST_URL}/api/post/1**",
                createdAt: "2022-02-01T20:14:35.970Z",
                updatedAt: "2022-03-06T22:20:11.686Z",
                publishedAt: "2022-02-01T20:19:05.678Z"
            }
        },
        - {
            id: 2,
            + attributes: { … }
        },
        - {
            id: 3,
            + attributes: { … }
        }
    ],
    - meta: {
        - pagination: {
            page: 1,
            pageSize: 25,
            pageCount: 1,
            total: 3
        }
    }
}
```

Obrázek 26 Posts with related category (self-made)

```
http://localhost:1337/api/posts?populate=category
```

4.3 Frontend data fetching and rendering

4.3.1 Homepage

Obrázek 27 Homepage frontend (self-made)

As it was mentioned before, JavaScript framework was used for fetching and rendering the content. In does not matter which technology is chosen, but since I have more experience with web development, I decided to go with React JS, or actually some framework for React, called Next JS.

As it was covered in Literature review, there are two ways to access Strapi API, with REST API with many different endpoints and with different queries, or via GraphQL API. In the example below, first option is shown.

```jsx
// index.jsx
export async function getStaticProps(context) {
  const pageData = await
axios.get('http://localhost:1337/api/homepage?populate=posts,cover,posts.cover,posts.category')
  return {
    props: {
      page: pageData.data.data,
      articles: pageData.data.data.attributes.posts.data,
    },
    revalidate: 10,
  }
}
```

This is how the data is requested from backend. Already familiar URL address is used to reach the RestAPI endpoint to retrieve required data.
The returned object of this function contains data with Homepage related fields as well as the related articles.

```
{
  - data: {
      id: 1,
      - attributes: {
          title: "Bachelor Theisis",
          subtitle: "Why "Headless" CMS?",
          body: "### Introduction

The purpose and goal of this **Bachelor Thesis** is to show the advantages of so called **Headless CMS**. This demo project is a practical part of my thesis and represents the way how ea
The message you read right now comes from the API(application programming interface) that's created by **"Strapi"**.
Admin panel and data representation is shown in practical part of my bachelor thesis.


### About the project

This project was created using API builder - **"Strapi"**. It serves as backend for content creation and management. There is no commercial or informatical purposes, but only the demonst

### Some words about myself

My name is Arozhdan Baibussynov, I am 22 y.o student of 3d year at **Czech University of Life Sciences Prague** (CULS/CZU).
I came from Kazakhstan to get Europenean education and get a great life experiese, what I did throug last 5 year in Czech Republic.
I partisipated in **Erasmus** program in 2020 and have some spanish experiense in my life.


### What you can find on this website

As it was mentioned earlier, this project does not contain any sufficien information and serves only as demonstation of what Headless CMS can do, and what are the benefits of it.
There are some blog posts created with meanless information, so we can see the difference between "single" and "collection" types in Strapi CMS.
",
          createdAt: "2022-02-01T19:51:37.090Z",
          updatedAt: "2022-03-07T00:29:37.480Z",
          publishedAt: "2022-02-01T20:11:19.907Z",
        - posts: {
            - data: [
                - {
                    id: 1,
                    - attributes: {
                        title: "Basic Information",
                        subtitle: "Introduction into CMS",
                        date_publicated: "2022-01-20",
                        body: "## Post content

The content of the post comes from API and can be quickly modified. Benefit of **Headless CMS** is that it's independent from frontend. So this data can be used in many s

We can easlilly create web blogs like this one and have a complicated mobile app consuming same data.
```
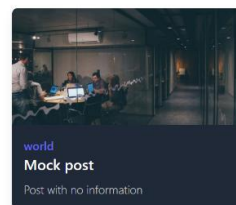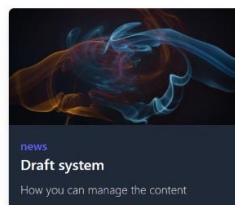
And once data is present in frontend application, it can be rendered and displayed as we need.

There is an example of simple JSX markup

```
<div className='page'>
    <div className="container mx-auto pb-12">
        <h1 className='typorgraphy-h1 mt-6'>
        {page.attributes.title}</h1>
        <h2 className='typorgraphy-h2 text-gray-800 mb-8'>
{page.attributes.subtitle}</h2>
        <div className="flex justify-between items-start">
            <ReactMarkdown className='normalize-text flex-grow-1 flex-shrink-0 w-
6/12 pb-12'>{page.attributes.body}</ReactMarkdown>
            <img className='w-3/12'
src={`http://localhost:1337${page.attributes.cover.data.attributes.url}` }
alt=''></img>
```

```
        </div>
        <div className="flex justify-between">
          {
            articles.map(article => (
              <Link key={article.id} href={`/post/${article.id}`}>
                <a>
```

```
                <Card  article={article} />
              </a>
            </Link>
        ))
      }
    </div>
  </div>
</div>
```

And this is the place, where the consumed data is placed into the JSX (HTML with JS) template.

That is the main difference in "Traditional" and "Headless" approaches. In our case, frontend is independent from the backend and can be written with any technologies.

We ask for specific data in specific files to generate specific web page. Same data can be used in any other software, like mobile apps written on Java, or desktop applications running on C++.

4.3.2 Post / article page

Basic Information

by Author 1, 2022-01-20   world

Post content

The content of the post comes from API and can be quickly modified. Benefit of **Headless CMS** is that it's independent from frontend. So this data can be used in many scenarios and using different approached.

We can easlilly create web blogs like this one and have a complicated mobile app consuming same data.

The data coming for this post can be found by the following url: **(HOST_URL)/api/post/1**

Obrázek 28 Post page (self-made)

We consume post data as well as all related fields like "creator" and "category" entities. Actually, media content is also stored in a separated table, so does not come by default.

Posts or articles pages are a little tricky, cause individual page must be generated for each row in posts database table. Of course, it is not required to add a new JS file on each publication of a new post, Next JS will handle it for us.

```js
export async function getStaticPaths() {
  const articles = await axios.get(`http://localhost:1337/api/posts`)
  const paths = articles.data.data.map(article => ({ params: { post:
`${article.id}` } }))
  return {
    paths,
    fallback: true // false or 'blocking'
  };
}
```

First, we catch all the posts. At this moment, we don't require related fields, so build process is faster. Once we do it inside "getStaticPaths" function, Next JS will load all posts and create a webpage for each of them. This is only related to this specific frontend framework and is different for each technology.

```
export async function getStaticProps(context) {
  const routeId = context.params.post
  const articleData = await
axios.get(`http://localhost:1337/api/posts/${routeId}/?populate=category,
cover,creator,creator.avatar`)
  return {
    props: {
      article: articleData.data.data,
    },
    revalidate: 10,
  }
}
```

This is how data fetching for a specific post in handled by Next JS. Since all possible routes are already stored, we just loop through them and make a GET request to fetch more data.

5 Results and Discussion

5.1 Results

The result of the practical solution is a modern full stack web application, that is running on local setup and can be hosted on any virtual server or serverless providers like AWS (Amazon Web Services). Application owner is secured, in case of software support termination, since the whole core code for both frontend and backend is under developer's control and can be deeply modified.

Strapi "Headless" content management system was chosen as backend – brains and content provider of the project, and Next JS (React JS framework) as a frontend tool to display incoming data. This approach allows to painlessly and quickly switch frontend to any other solution or/and use many data consumers on different platforms.

As the result, I have created a demo web blog, that's a functionality of creating, managing and publishing articles in a convenient web admin interface, it also allows to specify rights and permissions for different content consumers, so some data might be hidden from public and only visible for registered users if required. Each admin user also has different access level, such as super admin, administrator and editor.

5.2 Limitaions and possible solution

As a fullstack web developer, I have quite a big experience in both "traditional" and "headless" management systems, and have mentioned to myself some complications that come with a newer approach of development, I would like to discuss.

5.2.1 Human resources required

Both human and hardware resources consuming is higher.

Development requires more time and skills. When you build a website on

WordPress, for example, you can choose any of preinstalled templates for the markup and stylings, so you can build it with a small development experience. In our case, I had to build a frontend application from scratch, so I had to know any technology to consume and display incoming data. The documentations coming with most Headless CMSs are usually very descriptive and understandable, they always contain integration tutorials and manuals. And actually, it is the point of "headless" approach, when CMS is not responsible for frontend, but I think, newer generations might have provide with some website constructors, that would already know what content types are created.

5.2.2   Software resources

This is quite a big topic. Headless CMS itself requires much less computational powers than elder brothers do, but as a developer you must also think of the frontend part. There are three frontend development technologies and each have its own compromises:

- Single page applications (SPA). Render happens on client side. Does not consume server resources, but clients. Might run slow on old devices.
- Static server generators (SSG). Render happens on server side. Must be manually triggered, otherwise client does not receive updates in content. Render happens rarely, so server is not busy and client receives static well known HTML, CSS and JS.
- Server side rendering (SSR). Happens on the server on each page visit. So client receives most relevant and current state of data. Becomes a problem, if you have thousands of visitors each day.

As we can see, project owner must know about each approach and choose the best option.

While working on this thesis, Next JS (React JS framework) has released a major update, containing the changes I thought of as a possible solution and suggestion. Next JS running in SSR mode, now caches the page on first visit. Revalidate value can be set for each specific page, so it erases the page from cache after

period of time. In my case, this is the perfect solution.

5.2.3 Community and state of CMSs

Community is quite small at this moment, comparing to "Traditional" CMSs. It still requires some time, for articles to be written, questions to be answered. The state even of the most popular Headless content management systems is quite raw. New releases appear almost every week, that solve some bugs, but might also bring new ones. It is not a disadvantage of a problem of "Headless" CMSs, but of any new technology. They all require some time to evolve and grow.

5.3 Features

Never the less, newer approach brings a lot of benefits. Freedom is the biggest one. You choose what development stack to use. You can build same websites with different stacks, have multiple frontend applications manipulating same content and data source. Everything is safely synchronized and shared across the web to ones, who has rights to use to your data. You don't have to restrict the access to you website while maintenance or moving to another platform, both versions can work simultaneously.

So we can define most important features as following:

- Quick adjustment to changes.
- Less maintenance and migration complications.
- More workflow options.
- Freedom to choose development stack.
- No need to predefine target platforms. Once your API is up and running, you can connect new applications to it whenever you decide to.

6 Conclusion

This thesis was designed, so basing on theoretical knowledge, new clean instance of Strapi headless content management, could be launched. It shows, that even though, traditional approach is more popular, it deserves to be mentioned and used as an API builder and data provider. First, the required basic theory was described in Literature review, introduction to the world of content management systems, fundamental difference in two main approaches, introduction to Strapi user interface, file and data structures. Thesis aimed to show the impact of Headless CMS to web development in general, show what does it change and how it can be useful.

With this theory, demonstrational project was created, so that changes are feasible in practice. The process of content creation, storing in database and sharing with REST API was covered step-by-step. I also had a frontend service, to show how this data can be useful and consumable in real world application.

The flexibility of "Headless" approach would not be reachable in traditional management systems and building API from scratch would require much more time and knowledge.

As the conclusion, I would like to say, that I would definitely use Headless content management systems for personal projects and push this approach to be chosen by more developers.

7 References

Nick Abbott, Richard Jones, Matt Glaman, Chaz Chumley. 2016 . *Drupal 8: Enterprise Web Development.* Birmingham : Packt Publishing Ltd., 2016 . ISBN 978-1-78728-319-0.

2015. About Strapi CMS. *Strapi CMS.* [Online] 2015. https://strapi.io/about-us.

Amerland, David. 2013. *Google™ Semantic Search.* Indianapolis : Que, 2013.

Banks, Alex a Eve, Porcello. 2020. *Learning React.* Sebastopol : O'Reilly Media, Inc., 2020.

Barker, Deane. 2016. *Web Content Management.* s.l. : O'Reilly Media, Inc., 2016.

Eve Porcello, Alex Banks . 2018 . *Learning GraphQL.* Sebastopol : O'Reilly Media, 2018 . 978-1-492-03071-3.

Flanagan, David. 2020. *JavaScript: The Definitive Guide, 7th Edition.* Sebastopol : O'Reilly Media, Inc., 2020.

Herron, David. 2020. *Node.js Web Development.* Livery Place : Packt Publishing, 2020. 978-1-83898-757-2.

MacDonald, Matthew. 2020. The WordPress Landscape. *WordPress: The Missing Manual, 3rd Edition.* místo neznámé : O'Reilly Media, Inc., 2020.

NodeJS. [Online] https://nodejs.org/en/.

Palas, Petr. 2019. *The Ultimate Guide.* Brno : Kentico Software, 2019.

Petrov, Alex. 2019. *Database Internals.* Sebastopol : O'Reilly Media, Inc., 2019.

Raevskiy, Mikhail. 2020. why-headless-cms-is-the-future-of-web. *medium.* [Online] 9. Sep 2020. https://medium.com/swlh/why-headless-cms-is-the-future-of-web-759105706325.

Schäferhoff, Nick. 2021. Popular CMS by Market Share. *Websitesetup.* [Online] 1. May 2021. [Citace: 13. August 2021.] https://websitesetup.org/news/popular-cms/.

w3techs.com. 2021. Usage statistics of content management systems. *w3techs.com.* [Online] w3techs.com, 2021. https://w3techs.com/technologies/overview/content_management.

Yellavula, Naren. 2020. *Hands-On RESTful Web Services with Go Second Edition.* Livery Place : Packt Publishing, 2020. 978-1-83864-357-7.

8 List of pictures, tables, graphs and abbreviations

8.1 List of pictures

8.2 List of abbreviations

CMS – Content Management System

PHP - Personal Home Page (Hypertext Preprocessor)

HTML – Hypertext markup language

CSS – Cascadian stylesheet

JS – Javascript

NodeJS – Node JavaScript (running on the server)

SEO - Search engine optimization

RESTful - representational state transfer

GraphQL - Graph Query Language

JSON - JavaScript Object Notation

XML - extensible markup language

GitHub - Global Information Tracker

React JS – reactive JavaScript

JWT – JSON Web Token

NPM – Node package Manager