Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Diplomová práce

Porovnání headless CMS a následná implementace webové stránky ve zvoleném systému

Bc. Martin Choutka

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Martin Choutka

Informatika

Název práce

Porovnání headless CMS a následná implementace webové stránky ve zvoleném systému

Název anglicky

Comparison of headless CMS and subsequent implementation of a website in the chosen system

---

Cíle práce

Diplomová práce je zaměřena na headless content management systémy. Hlavním cílem práce je zhodnotit a porovnat nejpopulárnější headless systémy pro správu obsahu a dokázat, zda jsou vhodné pro tvorbu webové prezentace s pilotním vytvořením webové prezentace ve zvoleném systému.

Vedlejší cíle práce jsou:
- vypracování přehledu vybraných CMS,
- vypracování přehledu vybraných webových technologií.

Metodika

Metodika řešené problematiky diplomové práce je založena na studiu a analýze odborných informačních zdrojů. Vlastní práce spočívá v analýze a porovnání nejpopulárnějších headless systémů pro správu obsahu s pilotní implementací. Na základě syntézy teoretických poznatků a výsledků praktické části budou formulovány závěry diplomové práce.

Doporučený rozsah práce

60 – 80 stran textu

Klíčová slova

CMS, headless, SEO, analytika, React

Doporučené zdroje informací

BARRETT, Daniel. Efficient Linux at the Command Line: Boost Your Command-Line Skills. California:O'Reilly Media, 2022. ISBN 978-1098113407.

EDMONDSON, Mark. Learning Google Analytics: Creating Business Impact and Driving Insights. California:O'Reilly Media, 2022. ISBN 978-1098113087.

KANE, Sean a Karl MATTHIAS. Docker: Up & Running: Shipping Reliable Containers in Production. 3rd ed. California: O'Reilly Media, 2023. ISBN 978-1098131821.

PALAS, Petr. The Ultimate Guide to Headless CMS: Everything you need to know to choose the right CMS. 3rd ed. Brno: Kontent.ai, 2022. ISBN 978-1973352686.

SCHWARZMÜLLER, Maximilian. React Key Concepts: Consolidate your knowledge of React's core features. California: O'Reilly Media, 2022. ISBN 978-1803234502.

Předběžný termín obhajoby

2023/24 LS – PEF

Vedoucí práce

doc. Ing. Pavel Šimek, Ph.D.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 4. 7. 2023

doc. Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 3. 11. 2023

doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 17. 03. 2024

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Porovnání headless CMS a následná implementace ve zvoleném systému" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 31. 3. 2024 _____

Poděkování

Rád bych touto cestou poděkoval doc. Ing. Pavlu Šimkovi, Ph.D. za odborné vedení práce, jeho vstřícné rady a cenné poznámky. Dále bych rád poděkoval Ing. Petru Chalupovi za podklady, které usnadnily tvorbu webové aplikace.

Porovnání headless CMS a následná implementace webové stránky ve zvoleném systému

Abstrakt

V této diplomové práci je provedeno komplexní porovnání a analýza headless content management systémů, které v posledních letech nabývají na popularitě v oblasti webového vývoje. Headless CMS, charakterizované oddělením správy obsahu od prezentace, poskytují vývojářům flexibilitu při implementaci obsahu napříč různými platformami a zařízeními. Jsou zhodnoceny klíčové vlastnosti, přednosti a možná omezení několika vedoucích systémů na trhu, včetně Payload CMS, Strapi, Directus a Sanity CMS, a na základě těchto informací je demonstrována praktické implementace webové prezentace s využitím vybraného headless CMS. Důraz je kladen na potenciál headless CMS pro zvýšení efektivity vývoje, zlepšení uživatelské zkušenosti a integraci s moderními webovými technologiemi. Tato práce poskytuje důkladný přehled o současném stavu a možnostech využití headless CMS v praxi, nabízí směrnice pro výběr vhodné platformy v závislosti na specifických potřebách projektu a ukazuje, jak mohou tyto systémy přispět k inovaci v digitálním prostoru. Výsledky této práce jsou určeny odborníkům v oblasti digitálního marketingu, webového vývoje a designu, kteří hledají efektivní řešení pro správu a distribuci digitálního obsahu.

Klíčová slova: CMS, headless, SEO, analytika, React

Comparison of headless CMS and subsequent implementation of a website in the chosen system

Abstract

In this thesis, a comprehensive comparison and analysis of headless content management systems that have been gaining popularity in web development in recent years is performed. Headless CMS, characterized by the separation of content management from presentation, provide developers with the flexibility to implement content across different platforms and devices. The key features, strengths, and potential limitations of several market-leading systems, including Payload CMS, Strapi, Directus, and Sanity CMS, are evaluated, and a practical implementation of a web presentation using a selected headless CMS is demonstrated. Emphasis is placed on the potential of headless CMS to increase development efficiency, improve user experience and integrate with modern web technologies. This thesis provides a thorough overview of the current state and potential of headless CMS in practice, offers guidelines for selecting the appropriate platform depending on the specific needs of the project, and demonstrates how these systems can contribute to innovation in the digital space. The results of this work are aimed at digital marketing, web development and design professionals looking for effective solutions for managing and distributing digital content.

Keywords: CMS, headless, SEO, analytics, React

Obsah

1 Úvod

Ve světě neustále se vyvíjejících technologií a rostoucí digitalizace se správa obsahu stává zásadní součástí online prezentace firem a jednotlivců. Způsob, jakým organizace spravují a distribuují digitální obsah, má přímý vliv na jejich schopnost angažovat cílové publikum, zvyšovat svou online viditelnost a podporovat celkový marketingový výkon. V tomto odvětví se headless systémy pro správu obsahu jeví jako klíčové nástroje pro modernizaci webového vývoje a obsahové strategie.

Tradiční CMS, ačkoli poskytují uživatelsky přívětivé prostředí pro správu obsahu, často omezují flexibilitu a možnost personalizace prezentace obsahu kvůli jejich monolitické architektuře. Headless CMS, na druhé straně, nabízejí revoluční přístup ke správě obsahu tím, že od sebe oddělují backend a frontend. To umožňuje vývojářům plnou kontrolu nad způsobem prezentace obsahu na různých platformách a zařízeních.

Tento přístup usnadňuje organizacím rychle se přizpůsobit měnícím se trendům a požadavkům trhu, zatímco uživatelská zkušenost je optimalizována pro všechna digitální zařízení. Vzhledem k rostoucí popularitě headless CMS a jejich potenciálu transformovat digitální ekosystémy je nezbytné provést hloubkové porovnání mezi předními systémy na trhu.

2 Cíl práce a metodika

2.1 Cíl práce

Diplomová práce je zaměřena na headless content management systémy. Hlavním cílem práce je zhodnotit a porovnat nejpopulárnější headless systémy pro správu obsahu a dokázat, zda jsou vhodné pro tvorbu webové prezentace s pilotním vytvořením webové prezentace ve zvoleném systému. Vedlejší cíle práce jsou:

- vypracování přehledu vybraných CMS,
- vypracování přehledu vybraných webových technologií.

2.2 Metodika

Metodika řešené problematiky diplomové práce je založena na studiu odborných informačních zdrojů. Odbornými informačními zdroji je literatura, která se týká systémů pro správu obsahu, vývoje software, typů vykreslení obsahu, programovacího jazyka JavaScript a technologií s ním spojených. V neposlední řadě je pro aktuálnost informací analyzována dokumentace systémů pro správu obsahu a technologií. Literatura je nejnovější, aby bylo zamezeno čerpání neaktuálních informací, které se velmi rychle mění.

Praktická část práce staví na analýze jednotlivých systémů. Nejprve došlo ke srovnávacímu výkonnostnímu testu. Ten byl proveden lokálně, v ideálních podmínkách, kdy byly jednotlivé testy provedeny na stejném zařízení a verzi JavaScript runtime Node.js. Každý test proběhl s podobným dotazem. Výstup výkonnostního testu posloužil pro vícekriteriální analýzu.

Dále byl proveden test popularity. Ten, na základě počtu hvězd na platformě GitHub analyzuje, zda jsou systémy na vzestupu popularity. Výstup tohoto testu odhaluje, zda jsou některé systémy ve fázi stagnace, která může naznačovat úpadek projektu. Křivka vzestupu popularity signalizuje, které systémy mohou být v budoucnu úspěšnější než ty ostatní, a mít tak větší podíl na trhu, který se může pozitivně projevit na kvalitě systému. Stejně jako výkonnostní test, jsou výsledné hodnoty testu popularity použity ve vícekriteriální analýze variant.

Výstupem vícekriteriální metody variant s využitím metody váženého součtu je pořadí systémů dle jejich vhodnosti pro konkrétní účel na základě stanovených kritérií.

Pro vývoj byl zvolen systém, který skončil ve vícekriteriální analýze na prvním místě. Na základě systému byla zvolena infrastruktura. Vyvinutý systém byl nasazen a úspěšně otestován.

3 Teoretická východiska

3.1 Systémy pro správu obsahu

Systémy pro správu obsahu jsou balíčky softwaru, pomocí kterých lze automatizovat mnoho úkonů, díky kterým, je správa obsahu jednodušší. Systémy pro správu obsahu fungují na straně serveru. Přistupovat k takovému systému může více uživatelů naráz, a obsah bývá centralizovaný. Systémy pro správu obsahu umožňují editorům vytvářet nový obsah či ten stávající upravovat. Snahou editora je tak zpřístupnit obsah všem čtenářům. Systém se skládá z:

- uživatelského rozhraní,
- oprávnění a přístupů pomocí rolí,
- mechanismu k publikaci článků.

3.1.1 Rozdělení CMS

Web content management (WCM)

Správa obsahu je primárně určena masovému dodání obsahu koncovému uživateli. Systém dokáže velmi dobře oddělit prezentační vrstvu od obsahové. Jedná se o typ systému, který společnost potřebuje ke správě obsahu svých webových stránek a aplikací. ECM systém může poskytovat některá data (např. specifikace produktů, právní dokumenty), ale pokud je potřeba vytvořit domovskou stránku společnosti včetně blogu, přidat produktové stránky, galerie obrázků, knihovny videí nebo učinit web či aplikaci interaktivní přidáním kontaktních formulářů nebo formulářů pro zájemce, online obchodu nebo konfigurátorů produktů, je WCM systém vhodnější. (JustRelate 2020) (Barker 2016)

Enterprise content management (ECM)

Tento typ systému pro správu obsahu spíše podporuje společnosti při zpracování dokumentů a surových dat a obvykle neposkytuje způsob, jak vytvářet, spravovat a poskytovat obecný obsah pro zaměstnance (např. informace o lidských zdrojích) nebo pro externí publikum, např. na firemních webových stránkách. (Barker 2016) (JustRelate 2020)

Digital assets management (DAM)

Vhodný k uchovávání a správě médií, jako jsou obrázky, zvukové nahrávky či videa. Dokáže velmi dobře pracovat s metadaty a případně je transformovat. Liší se od ostatních typů tím, že obsahuje nástroje, které nejsou standardem. Může se jednat např. o změnu velikosti obrázků, zpracování videa (střih) atp. (Barker 2016)

Record management

Slouží ke správě dokumentů, které vznikly jako produkt businessové operace. Může se jednat o mnoho reportů, kontraktů či faktur. Takový systém je velmi dobrý v archivaci dokumentů a v řízení přístupu na základě oprávnění. (Barker 2016)

Dále lze systémy pro správu obsahu rozdělit také na tři podkategorie:

Component content management systém

Tyto systémy jsou využity ke správě drobných částí (odstavce, věty, slova), pomocí kterých lze sestavit komplexní obsah. (Barker 2016)

Learning management systémy

Určené pro interakci se studenty, vhodné ke správě studijního obsahu jako jsou materiály, výukové kurzy či videa. (Barker 2016)

Portály

Využity ke správě, prezentaci a agregaci mnoha toků dat do jednoho sjednoceného systému. (Barker 2016)

3.2 Webové aplikace

Webové aplikace lze rozdělit do čtyř různých aspektů nebo vrstev. Každý aspekt má jiné cíle a požadavky, stejně jako jiné nástroje k provedení práce.

Aplikační vrstva zahrnuje specializované backend služby, které se zabývají konkrétními úlohami, jako je zpracování dat a stav aplikace. Patří sem například dynamické vyhledávání, filtry pro katalogy produktů a správa uživatelů. Hlavními zainteresovanými stranami jsou vývojové týmy. (Baumgartner 2023)

Frontend vrstva představuje přímé rozhraní pro uživatele. Je zodpovědná za poskytování přístupného a sémantického jazyka HTML, zobrazování designu pomocí CSS a vytváření poutavého, dynamického uživatelského prostředí s přidáním jazyka JavaScript na straně klienta. Hlavními zainteresovanými stranami jsou týmy frontendu a designu. (Baumgartner 2023)

Vrstva správy obsahu se zabývá organizací obsahu, vytvářením datových struktur a pohodlnými způsoby úprav zmíněného obsahu. Jejími hlavními cíli jsou snadné používání a flexibilita struktury. Hlavními zainteresovanými stranami, které stojí za správou obsahu, jsou týmy e-marketingu a redaktoři obsahu. (Baumgartner 2023)

Vrstva runtime se zabývá nasazením nebo hostováním všech aspektů webové aplikace. Jejím hlavním cílem je najít vhodné prostředí pro technologické volby provedené ostatními týmy, aby byla zajištěna dostupnost, stabilita a bezpečnost systému. Hlavními zainteresovanými stranami jsou provozní týmy. (Baumgartner 2023)

3.3 Typy architektur

3.3.1 Monolitická architektura

Vývojové týmy obvykle vynakládají úsilí, aby zvolili technologii, která vyhovuje potřebám a zkušenostem týmu. Taková technologie je poté volena na základě zkušenosti s programovacím jazykem, rychlostí vývoje, jednoduchostí nasazení a používání. V rámci vývoje je potřeba brát v potaz několik faktorů, a monolitická architektura není nejlepší možná volba ve všech aspektech. (Baumgartner 2023)

Monolitická architektura může vynikat v jednom aspektu, ale selhávat v aspektu druhém. Typickým příkladem nesprávné volby může být monolitický WordPress. Ten může být zvolen pro jeho popularitu, či již stávající zkušenost editorů s tímto systémem. V dnešní době existuje spousta společností, které WordPress používají ve velkém měřítku. Takové společnosti jsou zvyklé na existující pluginy, které jsou v tomto systému dostupné, a nechtějí vynakládat finance či úsilí na vývoj nového systému, či přeučení stávajících editorů. (Baumgartner 2023)

Taková volba může být ale nevhodná pro aplikační tým, který v případě volby systému WordPress, musí dodatečné funkcionality psát v jazyce PHP. Je možné se spolehnout na dosavadní funkcionality systému, ale v případě absence potřebné funkcionality, kterou nelze nahradit pluginy, je takový tým odkázaný na svoji zkušenost s jazykem PHP. Pokud toto není silná stránka týmu, vývoj se může prodloužit, a ve výsledku také prodražit. Dodatečný vývoj pro WordPress bývá složitý, je nutné porozumět interním API, které nebývají nejlépe zdokumentované. (Baumgartner 2023)

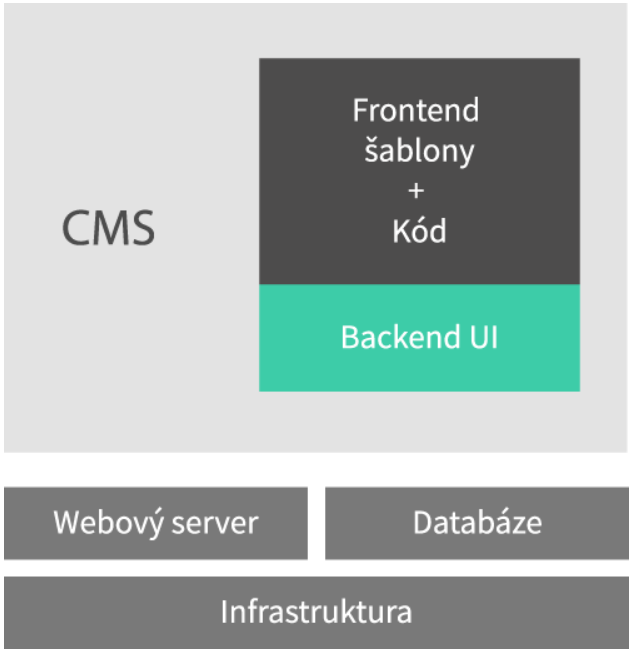
Tým, který vyvíjí frontend, je také odkázaný na limitaci systému WordPress. Ten, sám o sobě, postrádá *template engine*. Nejenom to, jako frontend vývojář je také potřeba znát, jak WordPress funguje, ať už se jedná o spoustu podmínek či funkcionalitu *partials*. Ve výsledku takový tým nevyužije své plné znalosti frontendu, spíše se musí spoléhat na znalosti systému WordPress jako samotného. To v samotném vývoji není nejlepší praktika, a takový způsob vývoje může mít za následek vznik aplikace, která nedosahuje potřebných kvalit. (Baumgartner 2023)

V neposlední řadě, je monolitický systém jako WordPress velmi limitující, co se provozu týče. Vývojář, či tým, který je odpovědný za nasazení systému musí zvolit takovou infrastrukturu, kterou WordPress vyžaduje. Jedná se o Linux server, na kterém je dostupný programovací jazyk PHP, databáze MySQL a webový server Apache. Volba takové

infrastruktury je logická pro malé stránky, pokud se jedná ale o velmi navštěvovanou aplikaci, ke které je potřeba přistupovat ze všech koutů světa nastane problém škálovatelnosti a distribuce obsahu. (Baumgartner 2023)

V dnešní době se aplikace nasazují na cloud. Taková řešení jsou poté lehce škálovatelná, v určitých případech také velmi finančně výhodná. Velmi vhodným příkladem je nasazení s pomocí cloudového poskytovatele AWS. Ten nabízí služby, které mají usnadnit nasazení monolitických systémů jakým je WordPress. Tyto služby ale u jiných cloudových poskytovatelů nemusí existovat. Nasazení systému tak může být náročné. Ve všech případech jde ale o rozhodnutí jednoho člověka, který zvolí přístup, jakým bude systém provozován. Nevhodná volba se projeví negativním dopadem na podstatnou část týmu. (Baumgartner 2023) (Amazon 2024) (Microsoft 2023) (Google 2023)

Obrázek 1: Monolitická architektura



Zdroj: vlastní zpracování dle (Palas 2022)

3.3.2 Headless architektura

Headless architektura je dělena na několik menších subsystémů, které na sebe nejsou přímo vázány. To přináší spoustu výhod, kdy největší výhodou je přenechání volby technologií na konkrétní týmy. Tým, který se stará o frontend, má k dispozici spoustu možností, jak k vývoji přistupovat, a nemusí se vázat na konkrétní řešení, která se používají

na straně aplikační. Jednotlivé části systému pak komunikují pomocí formátu JSON přes API. (Baumgartner 2023)

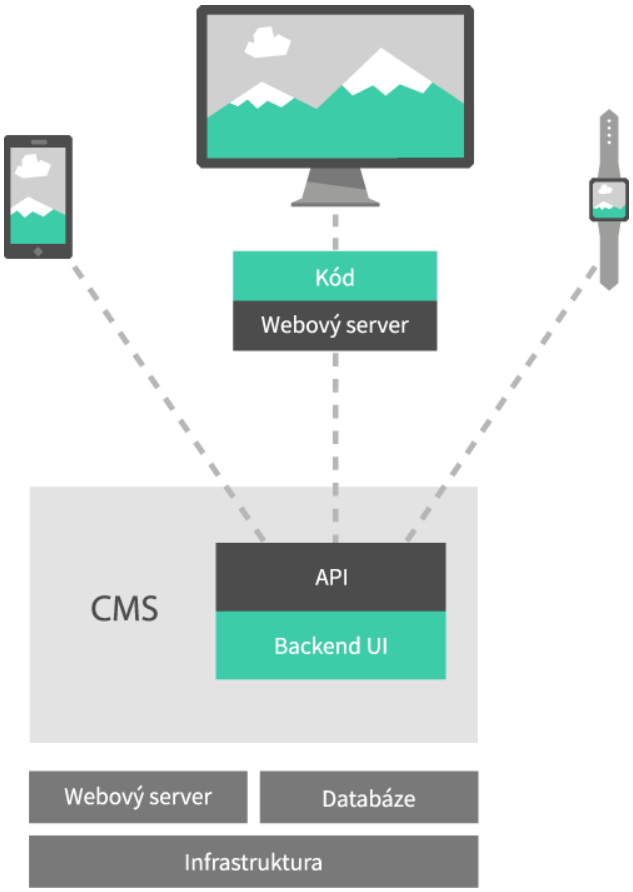
Tradiční CMS se vyvinuly v ohromné platformy, které zajišťují mnoho funkcionalit. Od samotné editace obsahu, po integraci s dalšími službami, ať již přes pluginy či vlastní řešení až po zobrazení frontend kódu. Monolitický přístup u obyčejného CMS má dva vstupy. Jedním vstupem je frontend, druhým vstupem je administrace. Systém obsahuje databázi, kdy jsou často data velmi zaměnitelně uložena. Obsah je uložený ve stejné databázi, která zároveň obsahuje data, jež slouží k funkcionalitě databáze. Mnohdy jsou data, vyprodukovaná tímto systémem špatně migrovatelná, přesun na jiné řešení je tak časově a finančně náročný. V případě, kdy je potřebné zobrazit stránku uživateli, samotná vrstva, která odpovídá za *renderování* se na data dotazuje také. Tato architektura sebou nese již popsaná negativa. (Baumgartner 2023)

Tento tradiční monolit lze rozdělit na skupinu menších, znovupoužitelných služeb. Prvním krokem je oddělení samotného systému pro správu obsahu od zbytku aplikace. Tento systém pak obsahuje pouze prvky určené pouze pro správu obsahu – logiku a databázi. Takový systém ale stále poskytuje data okolnímu světu, o kterém tento systém neví, přes rozhraní API. Toto rozhraní poskytuje data na základě formátu JSON. Takové architektuře se říká *headless*. (Baumgartner 2023)

Koncový editor, který spravuje obsah nepozná, zda se jedná o systém monolitický nebo oddělený. Funkcionalita systému zůstává stejná, kdy prvky, jako je např. vizuální editování, již nejsou doménou pouze monolitických architektur. Nespornou výhodou je možnost konzumace obsahu na více zařízeních, ať se jedná o obyčejný web, nebo mobilní aplikaci. (Baumgartner 2023)

Zatímco monolit má mnoho přístupových bodů, které potřebují různou viditelnost, včetně výstupu HTML, headless CMS má pouze dva: rozhraní pro správu a koncové body API. Přístup k API lze poskytnout s jemnou granularitou a administrační rozhraní může být plně skryto pro interní systémy. Díky tomu může být nasazení headless CMS mnohem izolovanější. Může běžet ve zcela jiném regionu nebo u zcela jiného poskytovatele cloudu. (Baumgartner 2023)

Obrázek 2: Headless architektura



Zdroj: vlastní zpracování dle (Palas 2022)

3.3.3 Decoupled architektura

Pokud je API hlavním poskytovatelem obsahu jakéhokoli druhu, je možné decentralizovat služby z hlavní aplikace. V kontextu složených webových architektur poskytují headless a decentralizované systémy hlavní bod pro kompozici. Prostřednictvím jejich flexibilních API je možné připojovat funkce dle potřeby. (Baumgartner 2023)

V případě, kdy je potřeba, je možné rozdělit aplikační vrstvu. Aplikační vrstva se skládá ze všeho mimo zobrazení spravovaného obsahu, tedy ze služeb, které nemusí být nutně součástí monolitické obsahové platformy. V systému, jako je WordPress, se jedná o ekosystém zásuvných modulů, kterými lze rozšiřovat základní funkce. (Baumgartner 2023)

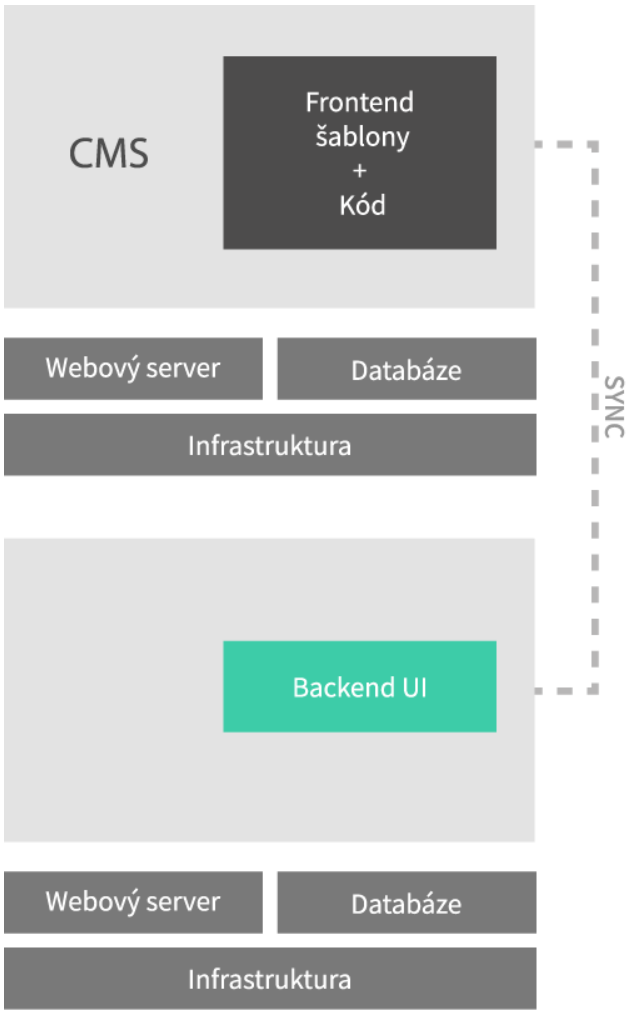
Namísto nasazení jednoúčelového aplikačního backendu se vykreslení obsahu stává samostatnou entitou, která komunikuje s API poskytovanými jednoúčelovými službami. Jednoúčelové služby jsou určeny k provádění jedné funkce nebo úlohy v rámci aplikace

nebo systému. Cílem jednoúčelových služeb je, aby byly samostatné, pracovaly izolovaně a neměly téměř žádnou závislost na jiných službách. Vlastnosti jednoúčelových služeb jsou obvykle navrženy tak, aby maximalizovaly efektivitu a výkon pro svou konkrétní funkci nebo úkol. (Baumgartner 2023)

Každá služba vykonává specifickou funkci nebo úkol. To usnadňuje údržbu a aktualizaci služby, protože změny v jednom modulu nemají vliv na ostatní moduly. Jednoúčelové služby zvládnou nárůst poptávky, aniž by došlo k problémům s výkonem. Služba je distribuována na více serverů nebo instancí, což jí umožňuje zvládnout velký objem požadavků. Mohou se zotavit ze selhání nebo chyb, aniž by to ovlivnilo celý systém. Redundantní servery nebo instance mohou v případě selhání převzít řízení. Mají minimální závislosti. Díky tomu mohou běžet efektivně a rychle, aniž by spotřebovávaly nadměrné zdroje nebo zpomalovaly ostatní služby. Mají dobře definované rozhraní pro přístup ke službě. To umožňuje snadnou integraci s jinými službami. (Baumgartner 2023)

Návrh aplikačního aspektu jako skupiny jednoúčelových služeb umožňuje hlavnímu účastníkovi tohoto aspektu – vývojářům – pracovat ve zvoleném programovacím jazyce s potřebnými frameworky a knihovnami. Mohou tak využít všechny své znalosti a rychle vytvářet nové funkce. Mohou také kombinovat technologie podle svých potřeb a stát se zcela nezávislými na volbě technologií ostatních vývojářů. Frontend vývojářů se tento způsob nedotkne, protože konzumují pouze další API. Obrovský dopad to má na provozní část. Ne všechny jednoúčelové služby vyžadují připojení k CMS. Tím dochází ke snížení počtu závislostí, a zvýšení spolehlivosti systému. (Baumgartner 2023)

Obrázek 3: Decoupled architektura



Zdroj: vlastní zpracování dle (Palas 2022)

3.4 Technologie

3.4.1 React

React je open-source JavaScript knihovna pro vývoj uživatelských rozhraní, zejména pro jednostránkové aplikace. Byla vytvořena vývojáři ze společnosti Facebook a poprvé byla představena veřejnosti v roce 2013. Hlavním cílem je umožnit rychlý vývoj dynamických webových aplikací s vysokým výkonem. Toho React dosahuje díky efektivní manipulaci s DOM (Document Object Model) prostřednictvím virtuálního DOM, což je lehká kopie skutečného DOM ve webovém prohlížeči.

React umožňuje vývojářům definovat komponenty jako znovupoužitelné kusy kódu, které mohou obsahovat vlastní logiku a stav. Tyto komponenty lze skládat dohromady k vytvoření složitějších uživatelských rozhraní. React podporuje deklarativní programování, což znamená, že vývojáři popisují, jak by měla aplikace vypadat pro různé stavy, a React se postará o aktualizace a zobrazení správného uživatelského rozhraní v reakci na změnu stavu. React dodržuje základní koncepty, na kterých funguje. (Tejas 2023)

Prvním konceptem je JSX. Jedná se o rozšíření jazyka JavaScript, které dokáže vkládat HTML kód do jazyka JavaScript přes syntaxi, která je podobná XML jazyku. Toto rozšíření se transformuje do platného kódu JavaScript, ačkoliv je nutné podotknout, že sémantika se liší v závislosti na specifikacích implementace. Samotné JSX se stalo populárním spolu s knihovnou React, v dnešní době lze nalézt implementace ale také v jiných knihovnách. (Osmani 2023)

Komponenty jsou základním kamenem jakékoli React aplikace. Jedná se o JavaScript funkce, které přijímají vstupy, kterým se říká *props* a vrací výstup, kterému se říká *React element*. React element vrací to, co by se mělo zobrazit na obrazovce. Může se jednat o samotnou komponentu, či o primitivní hodnoty, kterou může být například hodnota *null*. Výsledná aplikace se často skládá pouze z komponent, které lze nořit do sebe. (Osmani 2023) (Tejas 2023)

Props, dlouhým názvem *properties*, jsou vstupy, které vstupují do komponenty. Tyto vstupy jsou pro komponentu výchozími daty. Jsou uvedeny v deklaraci komponenty. Používají podobný zápis, jakým je definována deklarace HTML atributu. Hodnota *props* se nemění, tedy, nelze dynamicky měnit jaké *props* jsou komponentě posílány. Co je možné, je měnit hodnotu jednotlivých proměnných v rámci *props*. Tímto způsobem pak dochází ke

změně stavu komponenty, která je aktualizována překreslením (*rerendering*). (Osmani 2023)

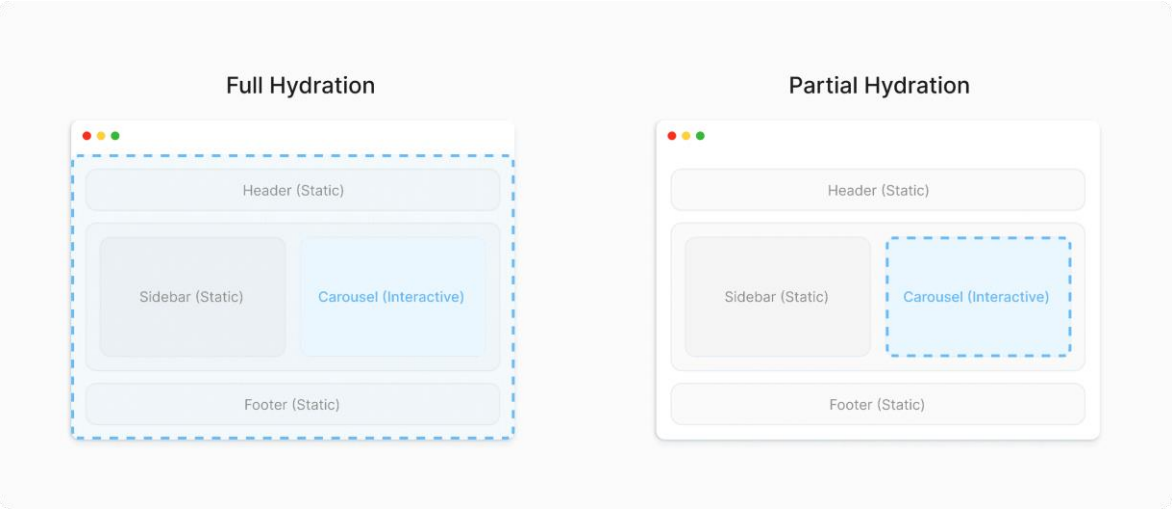
State, přeloženo do češtiny jako stav, je objekt, který obsahuje proměnné, které se v rámci životního cyklu komponenty mění. Tyto stavy jsou často spjaty se vstupy. Pokud má komponenta vstupy, jsou v určitých případech tyto vstupy použity jako výchozí hodnoty pro samotné stavy. Výhodou takového přístupu je možnost dynamicky měnit samotný stav nezávisle na samotném vstupu. (Osmani 2023)

Client-side rendering je jedním ze dvou způsobů, jak překreslit zobrazovaný obsah na stránce. V jednoduchý aplikacích, se stává, že je logika pro načítání dat a *routing* uložena pouze na straně klienta, v samotných JavaScript souborech, které jsou načítány při prvním načtení stránky. Tento přístup se velmi vyplatil v prvních počátcích samotné knihovny React, jelikož tento přístup pomohl oddělit webovou stránku vykreslovanou plně na straně serveru od aplikace, které stačí první načtení, a často se samotného serveru více nedotazuje. (Osmani 2023) (Tejas 2023)

Druhým způsobem, jak překreslit zobrazovaný obsah je *server-side rendering*. Od tohoto způsobu se neustupuje, naopak se stále používá, spíše ale v kombinaci s vykreslením na straně klienta. Tento způsob změny obsahu používají monolitické CMS ve výchozím nastavení. Systémy jako WordPress, Joomla či Drupal při přechodu na novou stránku dotazují server, který vrací nové HTML, které se má zobrazit. (Osmani 2023) (Tejas 2023)

V aplikaci vykreslované na straně serveru je kód HTML pro aktuální stránku generován na serveru a odeslán klientovi. Vzhledem ke skutečnosti, že kód byl vygenerován serverem, může jej rychle analyzovat a zobrazit na obrazovce. JavaScript potřebný k interaktivitě uživatelského rozhraní se načte až poté. Obsluhy událostí, které zajistí interaktivitu prvků uživatelského rozhraní, jako jsou tlačítka, se připojí až po načtení a zpracování balíčku JavaScript. Tento proces se nazývá *hydratace*. (Osmani 2023) (Tejas 2023)

Obrázek 4: Hydratace v React.js



Zdroj: (Partial Hydration 2022)

3.4.2 Vue.js

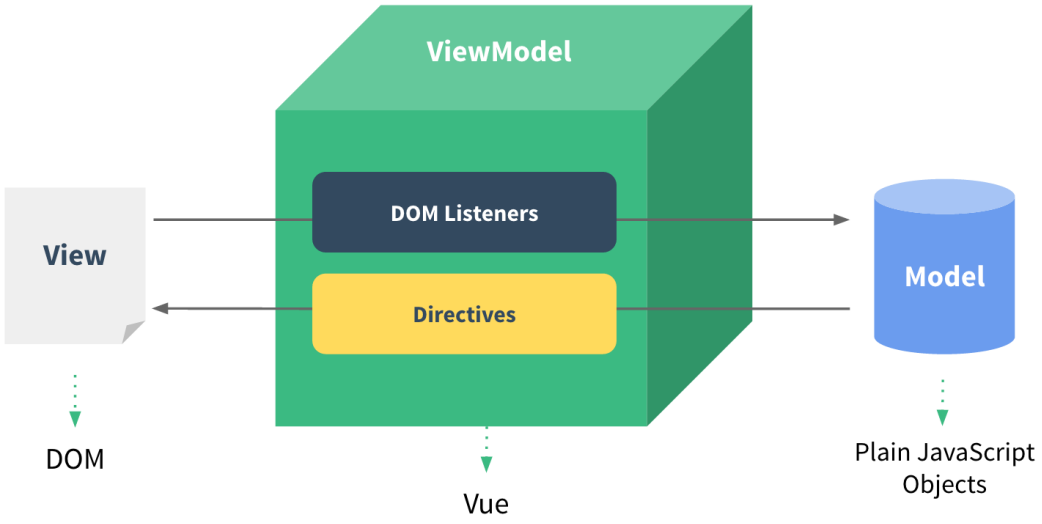
Vue.js je populární JavaScript framework pro psaní uživatelských rozhraní. Byl vyvinut Evanem You, bývalým vývojářem ve společnosti Google, který pracoval na Angular JS projektech. Vue.js se snaží využívat dobrých funkcionalit, které má Angular, ale je celkově lehčí, méně názorový, a jednodušeji se udržuje. Hlavním rozdílem oproti jiným frameworkům či knihovnám je jeho nevtíravý reaktivní systém. (Osmani 2023)

Stav komponenty ve Vue.js se skládá z reaktivních JavaScript objektů. Pokud jsou tyto objekty upraveny, samotný *view* se také aktualizuje. Tímto způsobem je *state management* velmi jednoduchým a intuitivním, ale je nutné, aby jej vývojář pochopil, některé funkcionality mohou být totiž problémové. V samotném modelu je zachycováno čtení a zapisování vlastností objektu. Různé verze Vue využívají odlišné způsoby, jak toto provádět. Vue 2 využívalo na základě limitace ze strany prohlížeče *gettery* a *settery*, ve verzi 3 jsou využity takzvané *proxy* pro samotné reaktivní objekty, a místo *getterů* a *setterů* jsou využity *refs*. Pojmenování referencí je například stejné v knihovně React. Každá instance komponenty vytváří reaktivní efekt, který vykresluje a aktualizuje DOM. Vue obsahuje Vue Composition API, které obsahuje rozhraní *ref*, *computed* či *watchEffect*. (Tejas 2023)

Vue.js začalo využívat takzvaných signálů. Mají podobnou funkcionalitu jako *refs*. Jedná se o kontejner hodnot, který umožňuje sledovat závislosti při přístupu a spouští vedlejší účinky při mutaci. Toto paradigma založené na reaktivitě není novým konceptem. Vue Options API a knihovna správy stavu React MobX jsou také založeny na stejných

principech. V důsledku používání virtuálního DOM se Vue v současné době opírá o kompilátory pro dosažení podobných optimalizací. Vue však zároveň prozkoumává novou strategii kompilace inspirovanou knihovnou Solid, zvanou Vapor Mode, která se neopírá o virtuální DOM a lépe využívá vestavěný reaktivní systém Vue. Největší předností je jednoduchost. Do HTML souboru stačí vložit knihovnu Vue pomocí tagu *script*. Dále je možné psát komponenty Vue. Vue navíc poskytuje CLI pro snadnou inicializaci nových projektů, což může být skvělý způsob, jak začít s prací na složitější aplikaci. (Tejas 2023)

Obrázek 5: ViewModel ve Vue.js



Zdroj: (Introduction 2024)

3.4.3 Next.js

Next.js je populární React framework od společnosti Vercel, a je známý pro své bohaté funkce a jednoduchost při vytváření vykreslovaných a statických webových stránek na straně serveru. Řídí se zásadou konvence nad konfigurací, čímž snižuje množství šablon a rozhodování nutných k zahájení projektu. S vydáním verze Next.js 13 bylo významným doplňkem zavedení Next.js *app routeru*. Jedním z jeho aspektů je, že na rozdíl od jiných frameworků Next.js neexponuje konfiguraci serveru, ale místo toho maskuje rozsáhlou komplexitu a umožňuje tak vývojářům soustředit se na tvorbu aplikací. (Tejas 2023)

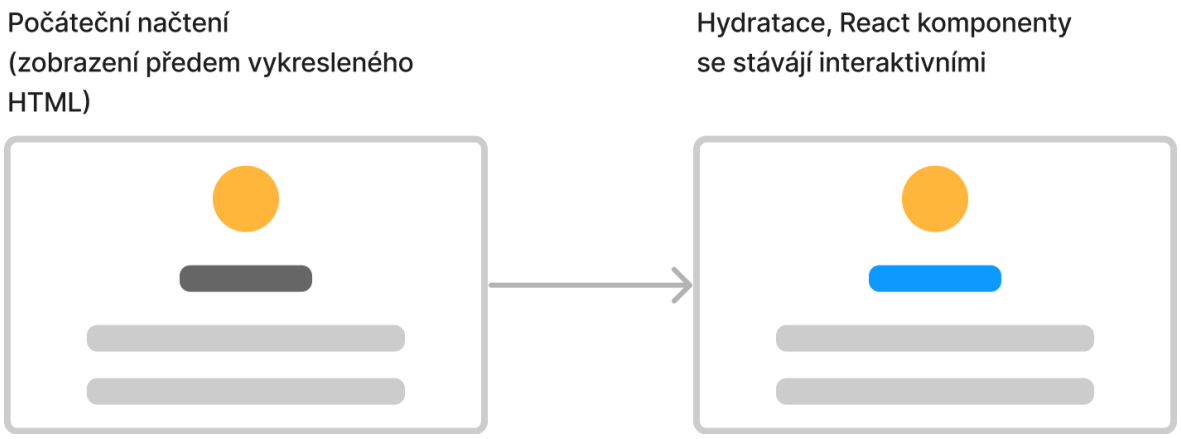
Next.js využívá nejnovější verzi knihovny React. Často se spoléhá na takzvané *canary* verze. To znamená, že je často o krok napřed a pochopení nových funkcí se může

zdát o něco náročnější. Je navržen s ohledem na flexibilitu mezi statickým a serverem vykresleným obsahem. Podporuje také zcela klientské aplikace, ačkoliv to není jeho primární případ užití. Nabízí bohatý ekosystém knihoven a integrací, které mohou výrazně urychlit vývojový proces. Někteří členové týmu, kteří vytváří React, pracují ve společnosti Vercel, kde se vyvíjí Next.js, což naznačuje velmi těsnou zpětnou vazbu vývoje mezi Next.js a knihovnou React. Next.js je navržen s důrazem na výkon a nabízí několik optimalizací pro rychlé a responzivní aplikace. (Tejas 2023) (Osmani 2023)

Podporuje automatické rozdělení kódu, což zajišťuje, že pro každou stránku je načten pouze nezbytný kód. Dále disponuje vestavěnou komponentou *Image*, která optimalizuje načítání obrázků pro lepší výkon. Hybridní model SSG/SSR umožňuje vývojářům vybrat optimální strategii načítání dat pro každou stránku, což vyvažuje výkon a aktuálnost. Stránky, které nepotřebují čerstvá data, mohou být předvykresleny v době sestavení, což vede k rychlejšímu načítání stránek. Pro stránky vyžadující čerstvá data lze použít serverové vykreslení nebo inkrementální statické vykreslení. (Tejas 2023) (Osmani 2023)

Next.js také poskytuje automatickou statickou optimalizaci pro stránky bez blokujících požadavků na data, zajišťující jejich vykreslení jako statické HTML soubory, což vede k rychlejšímu času prvního bajtu (TTFB). Využívá, kde je to možné, React server komponenty, a tím odesílá méně JavaScript kódu klientovi. To vede k rychlejšímu načítání stránek. (Tejas 2023)

Obrázek 6: Před-vykreslení



Zdroj: vlastní zpracování dle (Lizardo 2023)

3.4.4 Remix.js

Ve srovnání s Next.js je Remix novějším přírůstkem na scéně React frameworků, byl vytvořený v roce 2021. Jeho tvůrci jsou zároveň autory knihovny React Router a klade důraz na základy webu a poskytuje velkou flexibilitu. Remix může mít mírně plošší křivku učení, protože se více spoléhá na základy webu a využívá React způsobem, který byl normou před vznikem server komponent. Díky jedinečnému přístupu k *routingu* a načítání dat je Remix efektivní a výkonný. Vzhledem ke skutečnostem, že je načítání dat vázáno pouze na cesty, jsou načítána pouze data potřebná pro konkrétní cestu, což snižuje celkovou potřebu dat. Jeho strategie postupného vylepšování navíc zlepšují uživatelský komfort. Remix má jiný přístup k sestavení než Next.js. (Tejas 2023)

Namísto předvykreslování stránek využívá vykreslení na straně serveru, a odesílá HTML kód na klienta. To může vést k rychlejšímu TTFB, zejména u dynamického obsahu. Jednou z klíčových vlastností je robustní strategie ukládání do mezipaměti. Využívá nativní *fetch* a cache API prohlížeče, což vývojářům umožňuje specifikovat strategie ukládání do mezipaměti pro různé zdroje. To vede k rychlejšímu načítání stránek a odolnější aplikaci. (Osmani 2023)

3.5 Vykreslení obsahu

Vzhledem k rostoucím potřebám na dynamický obsah, a narůstajícím počtem událostí, které se začaly vyskytovat na webových stránkách, se zvýšil počet aplikací, které začaly přístup jednostránkových aplikací používat. Webové stránky však mohou být statické nebo dynamické dle toho, jakou funkci plní. Na webu je nadále velké množství statického obsahu, například blog, který lze vygenerovat na serveru a odeslat klientům v nezměněné podobě. Statický obsah je bezstavový, nevyvolává události a po vykreslení nepotřebuje rehydrataci. Naopak dynamickému obsahu (filtry, vyhledávání) musí být po vykreslení připojeny původní eventy. Na straně klienta je třeba regenerovat VDOM (virtuální DOM). V dnešní době existuje několik způsobů, jak obsah vykreslit. (Baumgartner 2023)

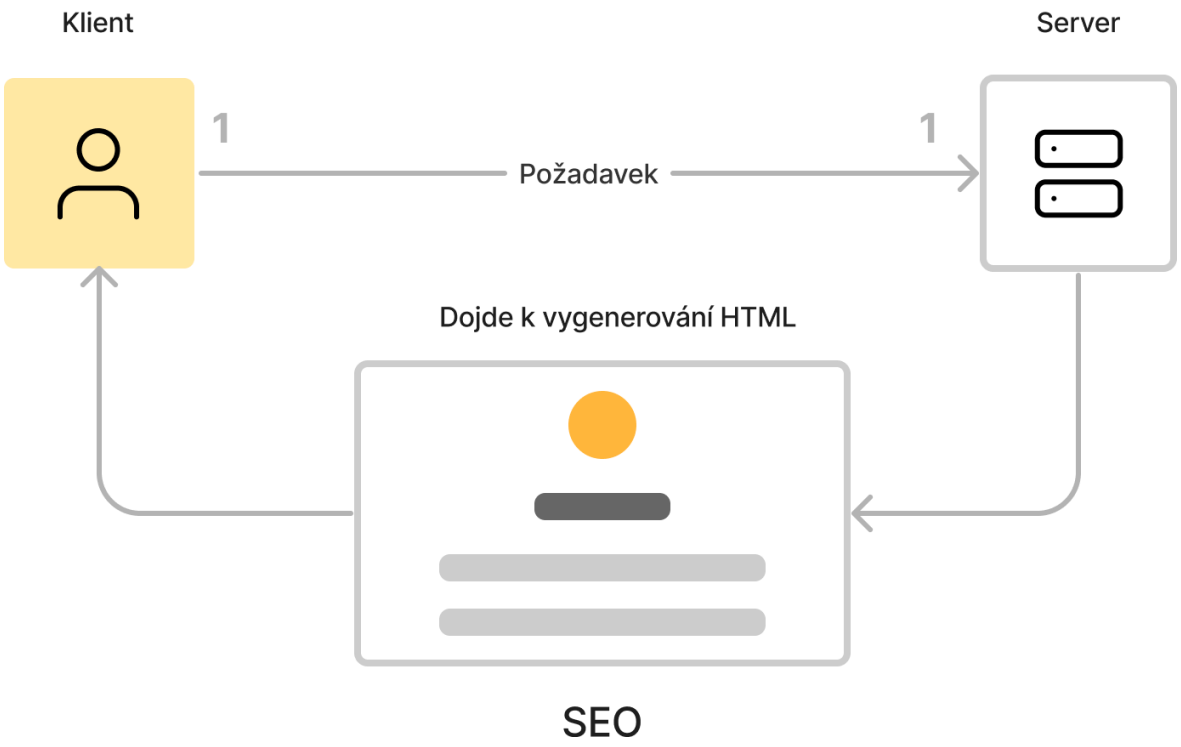
Tabulka 1: Způsoby vykreslení obsahu

Client-side vykreslení	HTML je vykreslené na straně klienta
Server-side vykreslení	Dynamické vykreslení HTML na straně serveru před hydratací na straně klienta
Statické vykreslení	Statické vykreslení HTML na straně serveru
Inkrementální statické vykreslení	Statické vykreslení HTML na straně serveru umožňující dodatečné vykreslení po prvotním sestavení
Streamované SSR	Obsah vykreslený na straně serveru je rozdělený do menších celků
Edge vykreslení	Vykreslení na straně serveru, který je nejblíže klientovi
Hybridní vykreslení	Kombinuje vykreslení na straně klienta a jednotlivé další přístupy vykreslení na straně serveru
Částečná hydratace	Hydratace, která probíhá jen u některých komponent (React server komponenty)

Progresivní hydratace	Hydratace, kde je ovlivněno pořadí hydratace komponent
Architektura ostrovů	Dynamické celky komponent, které jsou nezávislé od komponent ostatních
Progresivní vylepšení	Zajištění dostupnosti obsahu pro všechny klienty, kdy jsou dodatečné funkcionality zpřístupněny takovým klientům, kteří funkcionalitu podporují

Zdroj: (Baumgartner 2023)

Obrázek 7: Vykreslení na straně serveru



Zdroj: vlastní zpracování dle (Lizardo 2023)

Správnou volbou způsobu vykreslení obsahu lze zjednodušit vývoj, zrychlit sestavení celého projektu, či zlepšit funkcionalitu celého systému – ať už se jedná o samotný výkon, nebo pouhou uživatelskou zkušenost. Z tohoto důvodu se u webových stránek a aplikací používají základní metriky, kterým se říká *Core Web Vitals*. (Baumgartner 2023)

Tabulka 2: Metriky Core Web Vitals

Time to First Byte (TTFB)	Čas, dokud klient neobdrží od serveru první byte obsahu
First Contentful Paint (FCP)	Čas, dokud klient nevykreslí první kus obsahu po navigaci
Time to Interactive	Čas od doby, kdy se stránka načítá do doby, kdy je schopna odpovídat na vstupy klienta
Largest Contentful Paint	Čas, dokud se nevykreslí hlavní obsah stránky
Cumulative Layout Shift	Měřítko pro vizuální stabilitu obsahu, který se může v proběhu načítání stránky různě posouvat
First Input Delay	Čas od doby, kdy uživatel interaguje se stránkou, do doby, kdy proběhne odpovídající *event handler*

Zdroj: (Baumgartner 2023)

3.6 Headless systémy pro správu obsahu

Headless systémy pro správu obsahu umožňují editorům obsah editovat, ale zároveň je možné jej konzumovat pomocí API. Na rozdíl od tradičních systémů pro správu obsahu, headless systémy neobsahují prezentační vrstvu. Absence prezentační vrstvy skýtá spoustu výhod. Největší výhodou je možnost zobrazovat obsah na základě koncového kódu, tedy např. stránky která obsah konzumuje. Tím editace směřuje více k přístupu content-first, kdy se koncový editor stará pouze o obsah, nikoliv o to, jak bude zobrazen. U tradičního přístupu je konzumace takového obsahu především v mobilních aplikacích složitá. Headless systémy se dělí do dvou skupin, API-based a Git-based. (Barker 2016)

API-based CMS

Někdy se jim také říká API-driven CMS. Tyto redakční systémy si obsah ukládají v databázi a poskytují ho společně s dalšími metadaty ostatním navázaným systémům pomocí API, obvykle přes REST či GraphQL. Ke známějším zástupcům patří například Strapi, Directus nebo Payload CMS. (Barker 2016)

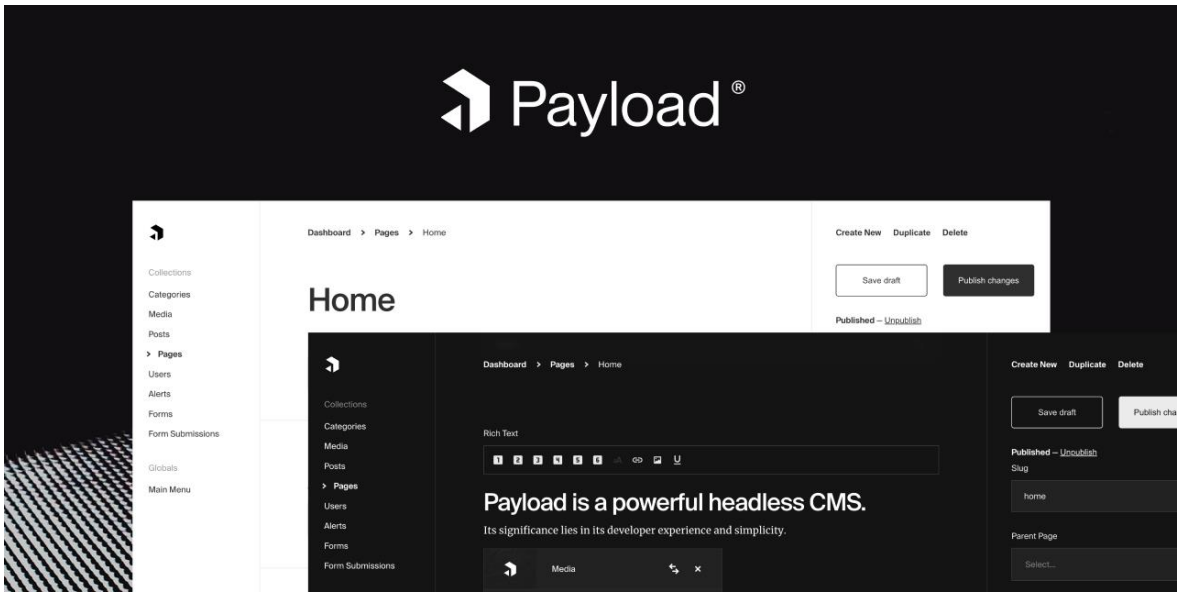
Git-based CMS

Tyto headless redakční systémy neukládají obsah do databáze. Jako úložiště používají Git, open-source systém používaný vývojáři obvykle ke správě různých verzí programů, větvení vývoje apod. Z toho plynou zajímavé výhody, jako je například možnost sledovat průběžné změny v obsahu. Ke známějším zástupcům tohoto typu headless CMS patří Netlify CMS, Jekyll Admin či Forestry. (Barker 2016)

3.6.1 Payload CMS

Payload CMS je API-based headless systém, který byl oficiálně představen 14. listopadu 2022. Na rozdíl od ostatních CMS je veškerá konfigurace systému prováděna přímo ve zdrojovém kódu. Má plnou TypeScript podporu – většina ukázek je psaných přímo s typy.

Obrázek 8: Systém Payload CMS



Zdroj: (Payload CMS 2023)

Payload CMS vyniká svojí konfigurací. Vývojář může CMS přizpůsobit přímo na míru klienta. Dle základních principů Payload obsahuje kolekce a globální kolekce. Kolekcí se rozumí sdružení položek stejného typu. Jedná se především o stránky a články. Může jich být ale celá řada – zaměstnanci, faktury, projekty aj. (Payload CMS 2024)

Globální kolekce mají podobný účel, sdružují určitou část informací. Na rozdíl od kolekcí normálních ale obsahují pouze jednu položku. Prvky, vyskytující se na webových stránkách, jako hlavička či patička, by měly být definovány jako globální kolekce, pokud neuvažujeme o použití více hlaviček či patiček, které by se na webové stránce postupně měnily. (Payload CMS 2024)

Oproti ostatním systémům je Payload velmi dobře rozšiřitelný. Jeho hlavní výhodou je možnost využití vlastních komponent pro zobrazení různorodého obsahu přímo v administraci. Je tak možné jej rozšířit o vlastní úvodní komponentu, na které lze administrátorům či editorům zobrazit základní informace, analytiku či stav serveru. (Payload CMS 2024)

Tento postoj ale nezaujímá Jan Hejzlar v jeho akademické práci, který uvádí, že možnost rozšíření o vlastní funkcionality je nedostačující, což může být v rozporu s reálným stavem, kdy Payload CMS nabízí mnohem více funkcionalit než konkurenční systémy. (Hejzlar 2023)

3.6.1.1 Kolekce

Payload se liší od jiných headless systémů svojí implementací kolekcí. U headless systémů jako je Strapi či Directus lze kolekce vytvářet, velikým rozdílem je ale postup, kterým se tvoří. U systému Payload je nutné, aby byly definovány přímo v kódu. Strapi či Directus využívají databáze, kam ukládají veškerá nastavení kolekcí, relací či bloků. Výhoda definování kolekcí v kódu spočívá v jednoduché replikaci již nasazeného systému na vícero místech. V případě smazání projektu a nového nasazení není potřeba veškerou konfiguraci provádět znova. V případě systémů Strapi či Directus by takový postup musel zahrnovat také zálohu databáze, a její pozdější obnovu. Velikou výhodou této nezávislosti je také snížení požadavků na databázi samotnou. (Payload CMS 2024)

Kolekce je možné rozšiřovat o hooky. Jedná se o jednoduché funkce, které přebírají určité argumenty od zdrojového kódu systému Payload. Představují způsob, pomocí kterého je možné rozšiřovat funkcionality systému o vlastní business logiku. Verze 1.8.2 čítá 15 hooků, kdy jsou hooky *afterChange* či *afterDelete* nejpoužívanější. Hooky mohou nabývat těchto použití: (Payload CMS LLC 2023)

- integrace uživatelských profilů s CRM třetí strany, jako je Salesforce nebo Hubspot,
- odesílání kopií nahraných souborů na Amazon S3 nebo podobné úložiště,
- automatické přidávání pole *lastModifiedBy* do dokumentu pro sledování toho, kdo v průběhu času dokument změnil,
- šifrování dat pole při ukládání a dešifrování při čtení,
- integrace s poskytovatelem plateb, jako je Stripe, pro automatické zpracování plateb při vytvoření objednávky,
- bezpečné přepočítávání ceny objednávek na straně serveru, aby bylo zajištěno, že celková cena za objednávky, které uživatelé odešlou, je přesná,
- generování a ukládání data posledního přihlášení u uživatele přidáním hooku *afterLogin*.

Všechny hooky lze psát jako synchronní nebo asynchronní funkce. Pokud má hook upravit data před aktualizací nebo vytvořením dokumentu a spoléhá se na asynchronní akce, jako je například načítání dat, dává smysl definovat hook jako asynchronní funkci, aby bylo zaručeno, že dojde k dokončení hooku před pokračováním životního cyklu operace. Asynchronní hooky se spouštějí v sérii – pokud jsou definovány dva asynchronní hooky, druhý hook počká na dokončení prvního a teprve poté se spustí. (Hooks Overview 2023)

Kromě definování hooků v rámci kolekcí, je možné je definovat také na úrovni polí. Jejich zaměření je ale mnohem menší, nepoužívají se proto běžně k úpravě celého dokumentu v rámci kolekce, ale spíše k drobným úpravám polí. Mohou nabývat těchto použití: (Payload CMS LLC 2023)

- automatické přidání vztahu vlastníka do dokumentu na základě *user.id* z požadavku,
- šifrování / dešifrování citlivého pole pomocí hooků *beforeValidate* a *afterRead*,
- omezení aktualizace dokumentu pouze jednou za X hodin pomocí hooku *beforeChange*.

Velmi využívaným hookem je *afterChange*, který se používá na úrovni kolekce v případě, kdy dojde ke změně dokumentu. Na tento hook navazuje v mnoha případech logika, která volá patřičné koncové body prezentační vrstvy, která v moment zavolání aktualizuje a revaliduje stránku s obsahem. Tento přístup se nazývá *on-demand* revalidace. (Payload CMS LLC 2023)

3.6.1.2 Databáze

Payload využívá nerelační databázi MongoDB. Jedná se dokumentovou databázi, kdy dochází k ukládání dokumentů ve formátu BSON. Struktura takové databáze je přímočará k relační databázi. Místo standardních tabulek využívá kolekcí, dokumenty nahrazují řádky a pole sloupce. Z pohledu relačního objektového modelování taková databáze postrádá strukturu a v určitých případech je obtížné s ní pracovat. U relačních databází je nutné strukturu tabulky dodržet, naopak u dokumentové databáze často nastane stav, kdy kolekce obsahuje dokumenty, jejichž struktura se liší. (Installation 2023)

Přesto, že se jedná o databázi dokumentovou, je možné používat relace na další kolekce již ze základu. Tvorba relací a jejich uchovávání je ale složitější. Payload proto využívá nástroj na relační objektové modelování Mongoose, který usnadňuje vývojáři práci s daty, datovou strukturou a především relacemi. Použití MongoDB u CMS dává smysl především v případě, kdy je konečné CMS potřeba provozovat s minimálními náklady, minimálním vytížením serveru a maximální dostupností. Dokumentové databáze velmi dobře podporují sharding, je tak možná data rozložit mezí více serverových instancí ať již v jednom regionu, či globálně. (Installation 2023) (MongoDB Inc. 2023)

Tabulka 3: Analogie dokumentové databáze k relační databázi

MongoDB	**SQL**
Database	Database
Collection	Table
Document	Row
Field	Column
Aggregation	Joins

Zdroj: vlastní zpracování dle (Matthew 2021)

Payload a jeho použití s MongoDB je vhodné, pokud již existuje dataset, který se pro budoucí účely musí zpracovávat v CMS. Příkladem může být např. soubor či objekt ve formátu JSON, který čítá milion záznamů. Oproti relačním databázím bude výkonnost při správném použití mnohonásobně vyšší. Tuto výkonnost lze zaručit správným využitím indexů, které Payload podporuje. Každá kolekce umožňuje definování databázových indexů.

Databázový index je speciální datová struktura, která umožňuje rychlejší přístup k datům a pomáhá vytvářet vysoce výkonné aplikace. Index se obvykle skládá ze dvou sloupců: vyhledávacího klíče a ukazatele dat. Klíč uchovává hodnotu, kterou je potřeba vyhledat, a ukazatel ukazuje na blok, kde se data nacházejí. (MongoDB Inc. 2023)

Payload využívá veškeré typy indexů, které MongoDB nabízí. Je proto možné pro kolekci definovat indexy typu *2d*, *2dsphere*, *geoHaystack*, *hashed* či *text*. Vývojář má plnou volbu v tom, jaké indexy využije. Payload umožňuje vytváření více indexů pro jednu kolekci. (Payload CMS LLC 2023)

Kód 1: Definice indexu pro kolekci

```
indexes: [{ fields: { title: "text" } }];
```

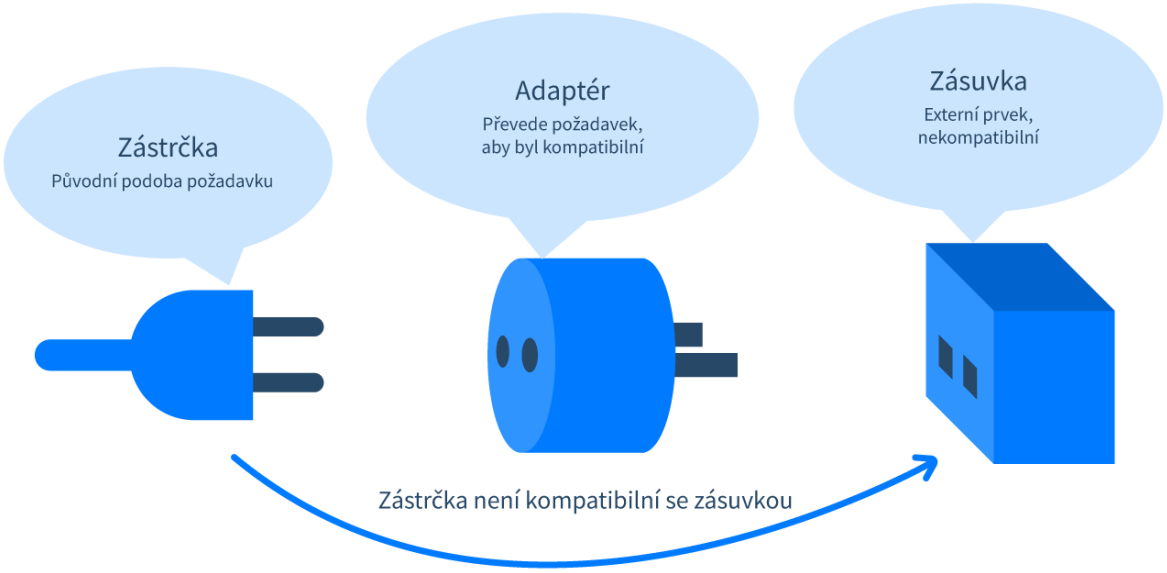
Zdroj: vlastní zpracování

3.6.1.3 PostgreSQL

Ke konci roku 2023 došlo k rozšíření možných databází o PostgreSQL. Pro využití další databáze začal Payload využívat takzvaný *adapter pattern*. Ten převádí rozhraní třídy

na jiné rozhraní, které koncový bod očekává. Adaptér umožňuje spolupráci tříd, které by jinak nemohly spolupracovat kvůli nekompatibilním rozhraním. To je vhodné pro rozšiřitelnost databází, kdy lze definovat strukturu, kterou má databázový adaptér obsahovat. Každý databázový adaptér, ať již pro MySQL nebo MariaDB, lze poté vyvinout nezávisle na systému samotném. To zaručuje určitou modularitu. Tento návrhový vzor není použitý pouze u databázových interakcí, ale také u *bundleru*, který *bundluje* uživatelský panel při sestavení, a také u *rich-text* editoru. (Mikrut 2023) (Freeman 2020)

Obrázek 9: Schéma návrhového vzoru adapter



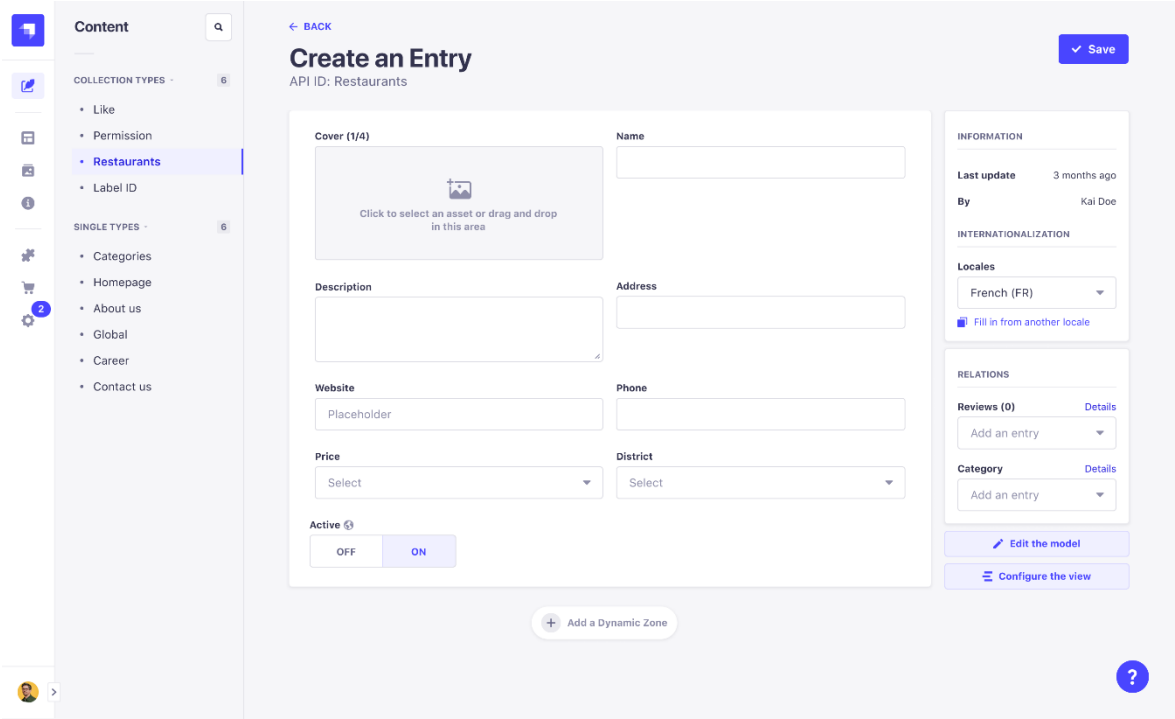
Zdroj: vlastní zpracování dle (JetBrains 2024)

Samotná MongoDB databáze je pro *use-case* systému Payload skvělá, v kontradikci s použitím SQL databáze není potřeba dokumentové databázi ustupovat. Ústupků na straně SQL je napříč celým spektrem headless systémů mnoho. Prvním ústupkem je používání *array* polí, které jsou v rámci SQL často vyřešeny jako obyčejný sloupec s JSON typem. Používání takového typu ale znamená, že je toto pole ochuzeno o možnost relací, a v rámci tohoto pole je možné používat jen primitivní typy. Druhým ústupkem je absence noření bloků (v jiných systémech komponent), kdy je takové noření velmi jednoduché v dokumentové databázi, ale poměrně složité v databázi relační. Pro zjednodušení databázových operací a zachování *type-safety* a definování dynamických databázových schémat je využit Drizzle ORM. (Mikrut 2023)

3.6.2 Strapi

Strapi je headless systém, který byl poprvé představen v říjnu 2015. Název je vytvořený ze slovní hříčky „Bootstrap API“. Jak název napovídá, jeho cílem je pomoci vývojářům rychle vytvořit API. Strapi šetří čas při vývoji API díky integrovanému a snadno použitelnému panelu správce a solidní sadě základních funkcí již v základu.

Obrázek 10: Systém Strapi



Zdroj: (Strapi 2023)

3.6.2.1 Databáze

Strapi umí pracovat s různými databázovými systémy. Lze jej nastavit a nakonfigurovat pro práci s PostgreSQL, MySQL, MongoDB a SQLite. Možnost instalace Strapi je přímočará, kdy lze využít buď možnosti *Quickstart* či *Custom*. Možnost *Quickstart* nastavuje instanci Strapi pouze se základní databází SQLite. Strapi varuje, že je pro produkční využití nevhodná, ale nespecifikuje, z jakých důvodů. (Elshafie 2022) (Strapi 2023)

SQLite je databáze, která umožňuje zapisovat a číst ze souborů *.sqlite*. Pracuje podobně jako ostatní zmíněné databáze, ale pracuje pouze s jedním souborem. Konkurenční

databáze, které jsou označovány za produkční, pracují s tisíci soubory. Ty efektivně rozdělují datové tabulky a udržují indexy. (Erz 2021)

SQLite je velmi jednoduchý, a usnadňuje tak migraci, zálohování či obnovu dat. V případě zálohování není nutné provádět export dat pomocí souborů SQL, které zvyšují skutečnou velikost kvůli mnoha příkazům INSERT v záložním souboru. Taková databáze je velmi vhodná pro použití v mobilních aplikací, u kterých je nutné, aby výsledná velikost byla co nejmenší. SQLite využívá databázový protokol, který je rychlejší než databázový protokol využívaný u konkurenčních databází. Teoretická velikost databáze SQLite je zároveň 281 terabajtů. (Erz 2021)

Množství dat, která je potřeba uchovávat, ale hraje velmi vysokou roli ve zvolení správné databáze. U redakčního systému lze předpokládat, že dat bude velmi mnoho. Data lze do databáze uložit jako nestrukturovaná, typickým příkladem může být JSON. Ten, u složitějších objektů může velmi zvýšit velikost databáze. (Erz 2021)

V případě, kdy bude databáze uchovávat velké množství dat, je naprosto nežádoucí využít SQLite. Důvodem jsou indexy, které uchovává každý databázový systém. Index je samostatný soubor, který obsahuje jeden záznam pro každou položku v původních datech. Index je mnohem menší, než jsou původní data, a z toho důvodu je mnohem rychlejší vyhledat index než celý původní záznam. Databázové systémy proto hledají index, který obsahuje odkaz na místo původního záznamu. Tyto systémy pak otevírají jeden konkrétní soubor z tisíce souborů, kdy tento soubor obsahuje pouze velmi malou část záznamů tabulky. Otevírání malého souboru je mnohem rychlejší než otevírání velmi objemného souboru. V tento moment SQLite selhává, a rychlost čtení potřebných dat se exponenciálně zvyšuje na základě objemu dat, která jsou v databázi uložená. (Erz 2021)

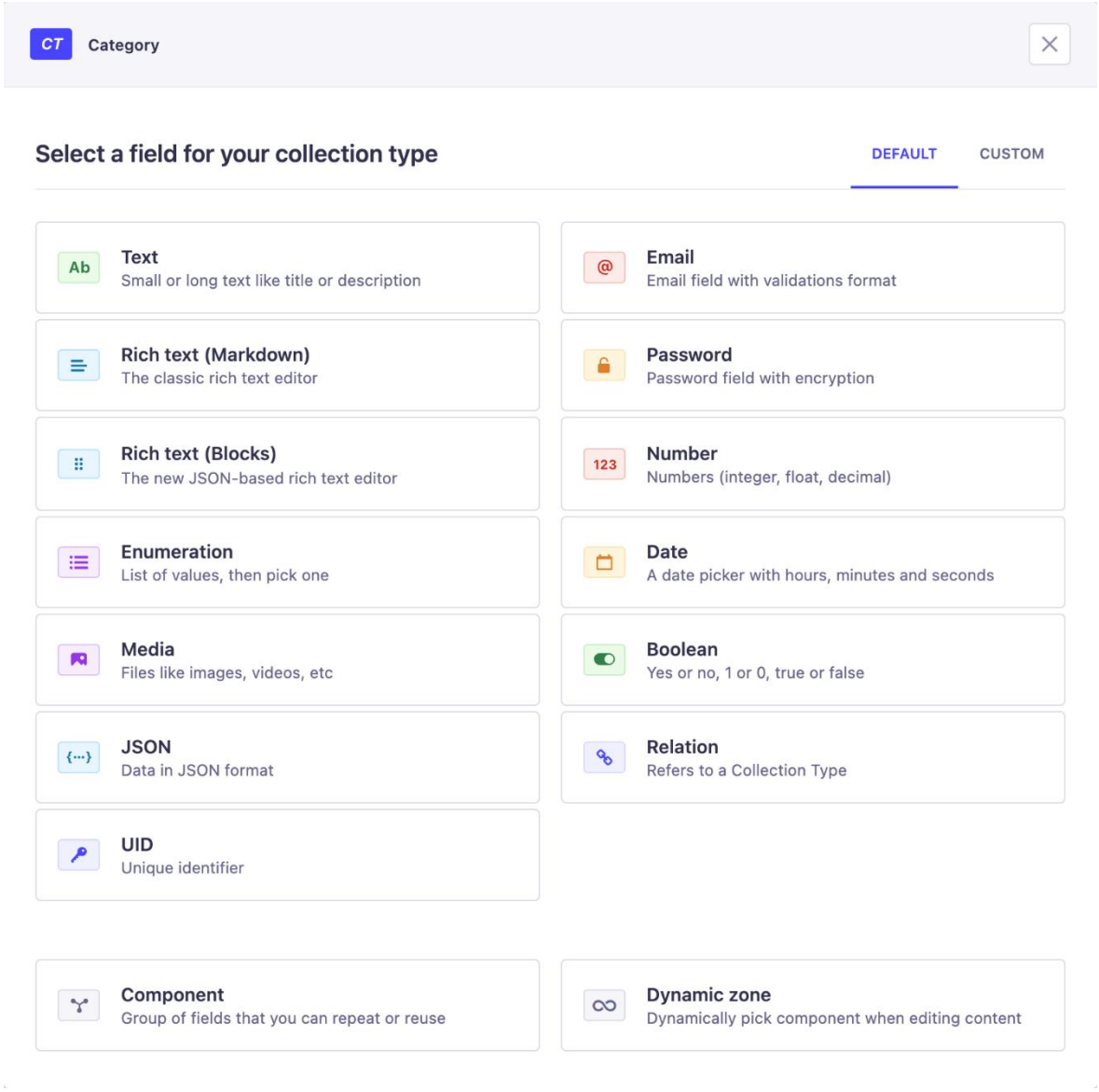
U redakčních systémů je běžné používat databázi MySQL. Databázi MySQL podporuje většina poskytovatelů hostingu. Při nasazení na vlastní prostředky je na místě zvážit zvolení PostgreSQL před MySQL z výkonnostních důvodů.

3.6.2.2 Typy obsahu

Typy obsahu jsou v systému Strapi způsobem, jak definovat entity, které tvoří systém. K vytváření a správě těchto typů obsahu existuje zásuvný modul *Content-Type Builder*. Jedná se o jeden ze základních zásuvných modulů Strapi. *Collection* typy jsou typy obsahu, které mohou spravovat několik položek. *Single* typy, jsou typy obsahu s jedním vstupem, které si lze představit jako jednořádkovou tabulku. Lze je použít ke správě věcí,

jako je nastavení aplikace. *Components* jsou vlastní datovou strukturou, kterou lze použít jak v *collection* typech, tak v *single* typech. Je možné např. vytvořit komponentu *full name*, která se skládá ze dvou textových polí (křestní jméno a příjmení), a tuto komponentu pak použít v typech obsahu *collection* a *single*. (Elshafie 2022) (Strapi 2023)

Obrázek 11: Možné typy obsahu v systému Strapi



Zdroj: vlastní zpracování dle (Strapi 2023)

V moment, kdy se nadefinuje jakýkoli typ obsahu, Strapi automaticky vytváří REST API pro konkrétní typy obsahu. Strapi definuje *createCoreRouter factory* funkci, která automaticky generuje pět základních API endpointů. Jedná se o CRUD operace (*find*, *findOne*, *create*, *update* a *delete*). Funkce *createCoreRouter* je definována v souboru

{content-type}/routes/{content-type}.js. Tento soubor je ale prázdný, až na definici factory funkce *createCoreRouter*. To je způsobeno architekturou Strapi, kdy jsou generovány soubory, které přebírají chování od výchozích komponent. Tyto soubory jsou generovány z důvodu, aby je vývojář mohl přepsat, a pozměnit tak jejich funkcionalitu. Strapi uvádí příklad generovaných endpointů, kdy je každý endpoint odpovědný za provedení jedné z CRUD operací. Endpointy konformují s REST specifikací. (Elshafie 2022)

Tabulka 4: Strapi endpointy na základě metody

HTTP metoda	**Cesta (URI)**	**Výchozí *handler***	**Operace**
GET	*/api/tutorials*	*tutorial.find*	Získá veškeré tutoriály
GET	*/api/tutorials/:id*	*tutorial.findOne*	Získá jeden tutoriál na základě ID
POST	*/api/tutorials*	*tutorial.create*	Vytvoří nový tutoriál
PUT	*/api/tutorials/:id*	*tutorial.update*	Aktualizuje tutoriál na základě uvedeného ID
DELETE	*/api/tutorials/:id*	*tutorial.delete*	Smaže tutoriál na základě uvedeného ID

Zdroj: (Elshafie 2022)

3.6.2.3 Factory funkce

Factory funkce *createCoreRouter* samotná akceptuje dva parametry. Prvním parametrem je *uid* řetězec, který identifikuje jednotlivé endpointy. Formát takového řetězce je *api::api-name.route-name*. Druhým parametrem je konfigurační objekt, který obsahuje čtyři klíče, a který umožňuje přepsat výchozí nastavení API endpointu. Prvním parametrem je prefix, který umožňuje přidat každému endpointu určitou předponu. Tato předpona je poté použita v celé API cestě při jejím zavolání. Druhým parametrem je pole *only*, které umožňuje definovat, které API endpointy pro CRUD operace budou dostupné. Strapi tímto způsobem umožňuje vývojáři např. znepřístupnit endpoint pro mazání záznamu. Opakem tohoto klíče je *except*, který naopak umožňuje definovat, které CRUD operace nemají být

dostupné. Objekt *config* poté umožňuje definovat další složitější nastavení, které se týká middlewaru. (Elshafie 2022)

Kód 2: Inicializace factory funkce createCoreRouter

```
module.exports = createCoreRouter("api::tutorial.tutorial", {
  prefix: "",
  only: [],
  except: [],
  config: {},
});
```

Zdroj: (Elshafie 2022)

Je možné přidávat vlastní API cesty. Pokud je potřeba API cestu přidat, dle pravidel Strapi se musí vytvořit soubor ve složce *{content-type}/routes/*. Strapi nedoporučuje žádný specifický název souboru, ani nenastavuje specifická pravidla pro to, jak by se takový soubor měl jmenovat. Definování samotného obsahu souboru naopak pravidla má. Na základě toho, jak je Strapi navrženo, je potřeba, aby konfigurační objekty byly exportovány jako *CommonJS* module. To je způsobeno velmi nativní architekturou Strapi, kdy je kód psán v programovacím jazyce JavaScript, a není kompilován z programovacího jazyka TypeScript. (Elshafie 2022)

Konfigurační objekt poté umožňuje definovat jednu nebo více cest v poli *routes*. Konkrétní jedna cesta poté obsahuje klíč *method*, která umožňuje definovat jednu z HTTP metod na základě standardu HTTP/1.1. Je nutné definovat konkrétní cestu a název handleru. Handler není z principu otypovaný, ani importován jako objekt, což snižuje přívětivost pro vývojáře. Posledním klíčem je objekt *config*, který, jak již bylo řečeno, umožňuje definovat další hodnoty pro specifickou konfiguraci. (Strapi 2023) (Elshafie 2022)

3.6.2.4 Entity Service API

Strapi obsahuje další vrstvu architektury pro práci s komplexními datovými architekturami, kterými mohou být např. komponenty. Této vrstvě se říká *Entity Service API*. Tato vrstva zpřístupňuje několik metod, pro CRUD práci s daty. Jedná se o (Elshafie 2022):

- *findOne*
- *findMany*
- *create*
- *update*

- *delete*

Strapi umožňuje vytvoření vlastních metod (funkcí), ze kterých lze tuto vrstvu volat. Pro volání této vrstvy je nutné dodržovat předepsanou strukturu, kdy je na prvním místě jako parametr opět *uid*. Druhým parametrem je konfigurační objekt, který udává, jak má být vrstva dotazována. Strapi umožňuje dodat parametr, např. jako *query* parametr z prohlížečového dotazu. Tento parametr může sloužit např. k filtraci. Při definici *filter* objektu, je tento dotaz v Entity vrstvě přetransformován do *where* dotazu, kterým je dotazována databáze. (Elshafie 2022)

Entity vrstva umožňuje definovat další pravidla, která jsou aplikována na samotný databázový dotaz. Je možné definovat např. pole (sloupce), která mají být z databázového dotazu vrácena. Databázový dotaz je samozřejmě také možné začínat od určitého indexu, či dotaz limitovat na určitý počet výsledků. Obě tyto varianty slouží především ke stránkování výsledků již na straně serveru. Data je možné řadit, a v neposlední řadě je možné také dotazu dodat, které sloupce mají být populovány, jedná-li se o sloupec s cizím klíčem. Je vhodné podotknout absenci kontroly psaného kódu a samotných typů, vývojáři není proto poskytnuta zpětná vazba, zda je napsaný kód správný, a zda projekt bude bez problémů sestaven či nenastanou runtime chyby. (Elshafie 2022) (Strapi 2023)

V případě, že se uživatel dotáže vytvořeného endpointu, a nebude provedena transformace výsledků z *Entity Service*, může očekávat, že struktura dat nebude konformovat s daty z výchozích endpointů. Strapi umožňuje uživateli vrátit data ve formátu, které potřebuje pro své účely. V případě, že tato data chce zobrazit ve stejném formátu, jak to dělají výchozí endpointy, je nutné použít *utility* funkci *transformResponse*. (Elshafie 2022)

V případě, kdy se uživatel dotazuje na data z tabulky, a některé sloupce jsou cizí klíče, jsou tyto klíče z výsledného výsledku opomenuty. Na základě výchozího nastavení *Entity Service* jsou všechny relace automaticky nepopulovány, populaci těchto cizích záznamů musí vždy explicitně vývojář uvést. (Elshafie 2022)

Jak je zvykem u všech CMS, je toto agresivní nastavení zvoleno z výkonnostních důvodů, kdy mohou záznamy s velkým počtem relací způsobit zpomalení serveru a databáze. Pokud je nutné záznam populovat, musí se populované pole explicitně uvést v klíči *Populate*. Zajímavostí je nestandardní pojmenovávání klíčů napříč Strapi, kdy je pojmenování některých klíčů v souladu s *camelCase* notací, zatímco některé klíče využívají

Pascal notace. Při populaci je možné uvádět přímo názvy polí, nebo využít hvězdičku *, čímž dojde k populování veškerých relací v daném záznamu.

Výsledkem populovaného pole je poté objekt, který nabývá hodnot z navázané tabulky. Populace ale může být velmi hluboká, což je jev, který se v relačních systémech vyskytuje často. Hluboká populace přes několik tabulek je ale velmi složitá výpočetní a databázová záležitost, a tudíž je rozumné, že není prováděna automaticky. V případě, kdy je potřeba populovat záznam přes několik tabulek, měla by být uvedena cesta k poli, které chceme populovat, s tím, že musí být zpopulovány tabulky předchozí. Tato funkcionalita přímo koreluje s architekturou SQL a použitím vnitřních, vnějších nebo *left* a *right* joinů.

Nevýhodou použití Entity Service API a nevýhodou Strapi je oddělené chování GUI a samotného napsaného kódu. Nastavení, které je např. u typů obsahu provedeno z GUI, není nutně respektováno při volání Entity Service API. Při dotazování se na typ obsahu, kterým může být například takový „Uživatel“, se z dotazu provedeným přes Entity Service API vrací celý uživatelský objekt, včetně polí, která jsou ve Strapi označena jako chráněná. (Elshafie 2022)

Dobrou ukázkou může být heslo, které se z dotazu vrací také. Zpřístupnění takových dat by mohlo mít vážné důsledky, je proto nutné používat utility funkci *sanitize*. V ideálním světě by služba *Entity Service* měla respektovat nastavení použitá v GUI. Tato nastavení jsou ale ukládána do databáze, služba by proto musela vždy nejdříve dotazovat databázi na nastavení ke konkrétnímu typu obsahu, což se jeví jako další nepotřebný dotaz na databázi, který ve výchozím nastavení nemusí být nutný. Z tohoto důvodu utility funkce *sanitize* funguje na stejném principu. Této funkci je možné předat výsledky, které je nutné sanitizovat, a model, podle kterého se výsledky mají sanitizovat. Na vlastnosti modelu je nutné se nejdříve dotázat voláním metody *getModel*. Výslednou sanitizací dojde k odstranění nechtěných polí u objektu, případně u každého objektu v poli. (Elshafie 2022)

Strapi umožňuje příjemnou práci s kontextem. Kontextem je objekt, který vstupuje do *controlleru,* např. u základního GET dotazu. Kontext může obsahovat *request* objekt, který lze předat *Core* controlleru, a v případě, že kontext není upraven, lze zachovat původní funkcionalitu, a zároveň v samotném controlleru data vrácená z *Core controlleru* dodatečně upravit. To umožňuje jednoduchou manipulaci s daty při zachování původní funkcionality, pokud vývojář nemá potřebu data více transformovat. Ideální představou využití takové funkcionality a celého kontextu je zachování dotazu na uživatele s výchozími filtry či

stránkováním, zároveň ale s výpočtem věku uživatelů na základě jejich narození a dnešního data. (Elshafie 2022)

3.6.2.5 Query Engine API

Stejně jak tomu je u ostatních systémů, je nutné s daty pracovat více extensivně. Samotný koloběh dat je velmi složitý, data jsou dotazována, upravována nebo mazána. Každá entita v systému, reprezentována na úrovni databáze jako tabulka, může být často spojena s okolními systémy, které nejsou reprezentovány v systému uživatele. Z tohoto důvodu Strapi poskytuje *Query Engine API*. Tato služba poskytuje spojení mezi databázovými modely a reálnými databázovými entitami. Nejen to, je také odpovědná za sestavování samotného databázového dotazu, který přebírá z *Entity Service* API v podobě konfiguračního objektu sestaveného z jednotlivých filtrů. V případě, že je nutné provádět operace před dotazy, nebo také po dotazech na databázi, je možné využít *hooky*. Ty jsou ve stejném duchu využity také v ostatních CMS. *Hooky* jsou potřebné v následujících ukázkových případech (Elshafie 2022):

- smazání, úprava či sanitizace určitých polí, než jsou vráceny *Entity* službě,
- automatická úprava pole na základě změny jiného pole,
- notifikace ostatních systémů o změně.

Hooky jsou z celé architektury nejblíže k databázi. Používají se v moment, kdy je potřeba mít jistotu, že operace proběhne na všech endpointech. K dispozici jsou tyto *hooky* (Elshafie 2022):

- *beforeFindMany, afterFindMany*
- *beforeFindOne, afterFindOne*
- *beforeCreate, afterCreate*
- *beforeCreateMany, afterCreateMany*
- *beforeUpdate, afterUpdate*
- *beforeUpdateMany, afterUpdateMany*
- *beforeDelete, afterDelete*
- *beforeDeleteMany, afterDeleteMany*
- *beforeCount, afterCount*
- *beforeCountMany, afterCountMany*

Hooky jsou v rámci Strapi řešeny stejným způsobem, jak je tomu u ostatních systémů. Do procesu vstupují parametry, či data, získaná z databáze, v případě že *hook* probíhá po databázové operaci. Po modifikaci těchto dat jsou z metody předávána dále. (Strapi 2023)

Velmi velkou výhodou je aplikace softwarových principů. Složité operace, které se volají v rámci *hooků*, se dají enkapsulovat do vlastních služeb, a přes hlavní Strapi službu je provolávat na všech místech aplikace. Tento *DRY* přístup normalizuje a zlepšuje kvalitu systému. Společně s dalšími softwarovými principy je vhodné data, která vstupují do hooku nemutovat, ale vytvářet nový objekt. To nejenom zlepšuje čitelnost, ale systém je takto více předvídatelný. Přímá mutace je také nedoporučována z výkonnostního hlediska, ačkoliv je dopad minimální u kódu, který běží na straně serveru. (Elshafie 2022)

3.6.2.6 Autorizace

Strapi umožňuje vytvoření dvou typů uživatelů. Z principu CMS je prvním typem administrátor. Ten se dokáže přihlašovat do systému jako takového, pracuje převážně s GUI. Takový uživatel může provádět veškeré konfigurační operace, ale nemůže interagovat s obsahem samotným. Z tohoto důvodu Strapi umožňuje vytvoření API uživatelů, kterým je možné přiřadit určitou roli, která je opravňuje k provádění určitých úkonů, ale nemohou nastavovat systém samotný. (Strapi 2023)

Tato funkcionalita API uživatelů není v systému přímo zahrnuta, je nutné ji využít jako plugin. Veškeré operace uživatelů fungují především na základě JWT. Token je u Strapi uložený v úložišti prohlížeče a je připojen při každém dotazu na server jako autorizační hlavička. Z mnoha postupů, jak v dnešní době pracovat s přihlašovacím tokenem, je tento jedním z nejméně bezpečných. Největším úskalím je jednoduchá čitelnost tokenu doplňky prohlížeče. Konkurenční systémy s tokeny dokáží pracovat lépe, ať už ukládáním jako cookies či HTTP-only cookies. (Strapi 2023) (Okta 2023)

Strapi při použití výše zmíněného pluginu umožňuje vytváření uživatele přes POST požadavek při správném specifikování jeho těla. Uživateli se při úspěšné registraci vrací výše zmíněný token, který se ukládá do úložiště prohlížeče. Stejnou analogii lze použít pro přihlašování. (Elshafie 2022)

Výše zmíněné nebezpečí je u Strapi umocněno nesprávným nastavením JWT. Token je ve výchozím stavu nastavený na 30denní expiraci. Dle pravidel, mají být přihlašovací tokeny nastavené jako krátkodobé, s expirací v řádu dnů, nikoli týdnů. Častým řešením

tohoto problému je poskytnutí obnovovacího tokenu, který neumožňuje přihlášení, ale umožňuje vzdálenému serveru poskytnutí nového přihlašovacího tokenu prohlížeči uživatele po splnění podmínek, které si server nadefinuje. (Okta 2023)

Důvodem použití tohoto přístupu může být mnoho, jedním z nich je například zjednodušená práce se subdoménami či autorizací v rámci mobilních aplikací. Jedná se ale o případy použití, která nejsou častá. Hlavním důvodem je tedy spíše laxní přístup k vývoji. Expiraci tokenu je možné nastavit v konfiguračním JSON souboru. Strapi umožňuje definovat, kdo může přistupovat k jednotlivým typům obsahu, a jaké operace může provádět. Rozlišují se dvě role (Elshafie 2022):

- veřejná role
- autorizovaná role

Strapi ve výchozím nastavení nenastavuje překážky, a proto je možné se dotázat na obsah všech typů obsahu bez přihlášení. To je ve výchozím nastavení bezpečnostním rizikem. Definici, kdo může přistupovat k jakému obsahu, a jaké operace na něm může provádět lze provést v rámci GUI. V případě, že je pro operaci nutné přihlášení, je kontrolována existence JWT, kdy je na základě JWT (a případných dat po dotazu na databázi) porovnávána role uživatele a role potřebná. Tento postup je standardem a nijak nevybočuje od ostatních systémů. (Elshafie 2022)

Tabulka 5: Ukázka nastavení přístupových oprávnění v systému Strapi

Aktor	**Oprávnění na třídy**	**Oprávnění na návody**
Studenti	*find findOne findTutorial*	*findOne*
Učitelé	*find findOne findTutorial*	*findOne create*
Administrátoři	Všechna	Všechna

Zdroj: (Elshafie 2022)

V případě, že je nutné umožnit uživatelům upravovat jejich vlastní obsah, nelze se u Strapi spoléhat na plugin, který je uvedený výše. Implementace je na straně vývojáře. Strapi ale implementaci poměrně zjednodušuje. Využívá *policies*. Jedná se o funkci, která probíhá před tím, než se požadavek dostane k samotnému *controlleru*. Ty je možné definovat

na globální úrovni, nebo na úrovni jednotlivých API cest. Jsou velmi jednoduché, obalují *controller* a obsahují tak celý kontext, včetně informací o požadavku, odpovědi a samotném uživateli. Na základě podmínky, kterou vývojář nadefinuje, se vrací hodnota *true* nebo *false*. V prvním případě nastane invokace *controlleru,* v druhém případě server požadavek odmítne. (Elshafie 2022)

Strapi umožňuje také definici jednotlivých OAuth poskytovatelů, a umožňuje tak přihlášení přes sociální sítě. Které OAuth poskytovatele systém akceptuje lze nastavit v GUI. Jsou podporovány autorizační služby třetích stran. Jedná se o Cognito či Auth0. Chybí podpora OIDC, to ale může být nahrazeno jednou ze dvou zmíněných služeb. (Strapi 2023)

3.6.2.7 Pluginy

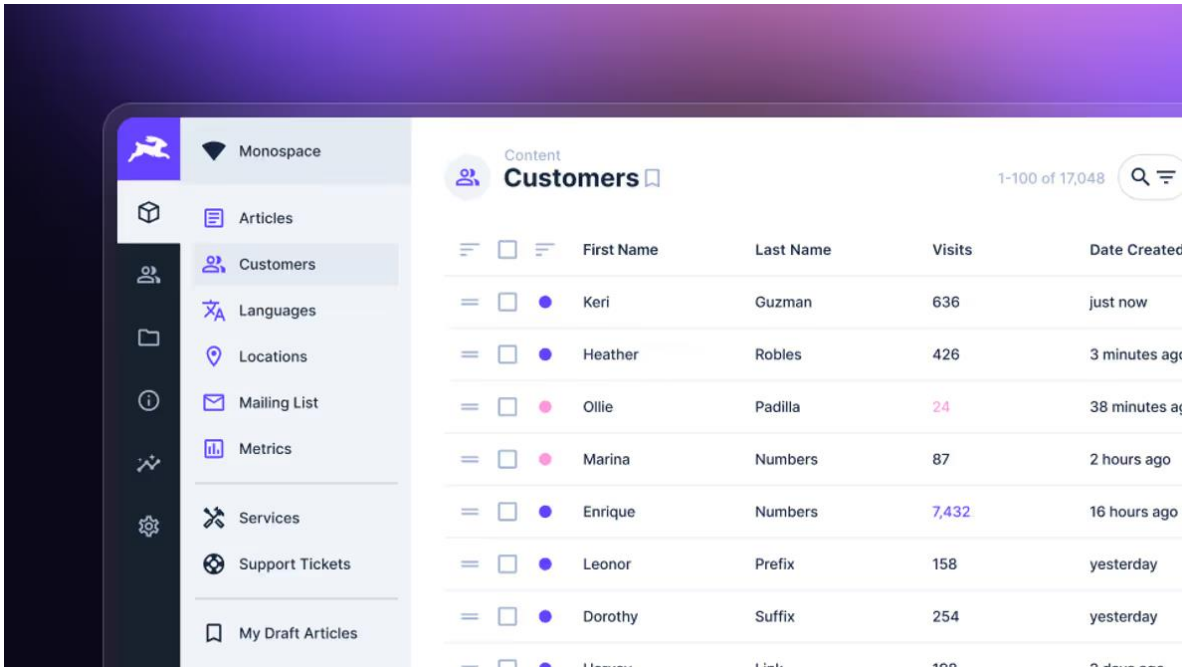
Strapi umožňuje používání pluginů pomocí *Plugin Marketplace*. Ten obsahuje 172 pluginů. Jedním z důležitějších pluginů, který se v Marketplace vyskytuje je *Documentation plugin*. Ten umožňuje jednoduché vytvoření dokumentace k API na základě specifikace OpenAPI. Pomocí tohoto pluginu je poté jednoduché tuto dokumentaci vizualizovat či automatizovat pomocí *Swagger UI*. Strapi umožňuje instalaci pluginů jak přes *Plugin Marketplace*, tak také přes příkazový řádek. (Elshafie 2022)

V případě nutnosti, umožňuje také Strapi instalaci pluginu GraphQL. Ve výchozím nastavení lze Strapi konzumovat pouze přes standard REST. Pomocí tohoto pluginu je ale možné obsah z CMS konzumovat přes GraphQL. (Strapi 2023)

3.6.3 Directus

Directus je headless systém pro správu obsahu srovnatelný se systémem Strapi. Výhodou systému Directus je využití takzvané databázové introspekce, která umožňuje čtení existující databázové struktury, a vytvoření abstrakční vrstvy. Directus podporuje mnoho SQL databází a neupravuje dosavadní databázovou strukturu. Systém Directus lze proto kdykoli přestat používat. Nevzniká zde riziko *vendor-locku*. Po databázové introspekci jsou vytvořeny REST a GraphQL endpointy, které umožňují správu dat, s granulární metodou přístupů. Pro možnost extenzivní správy je k dispozici CLI a JavaScript SDK. (Introduction 2023)

Obrázek 12: Systém Directus



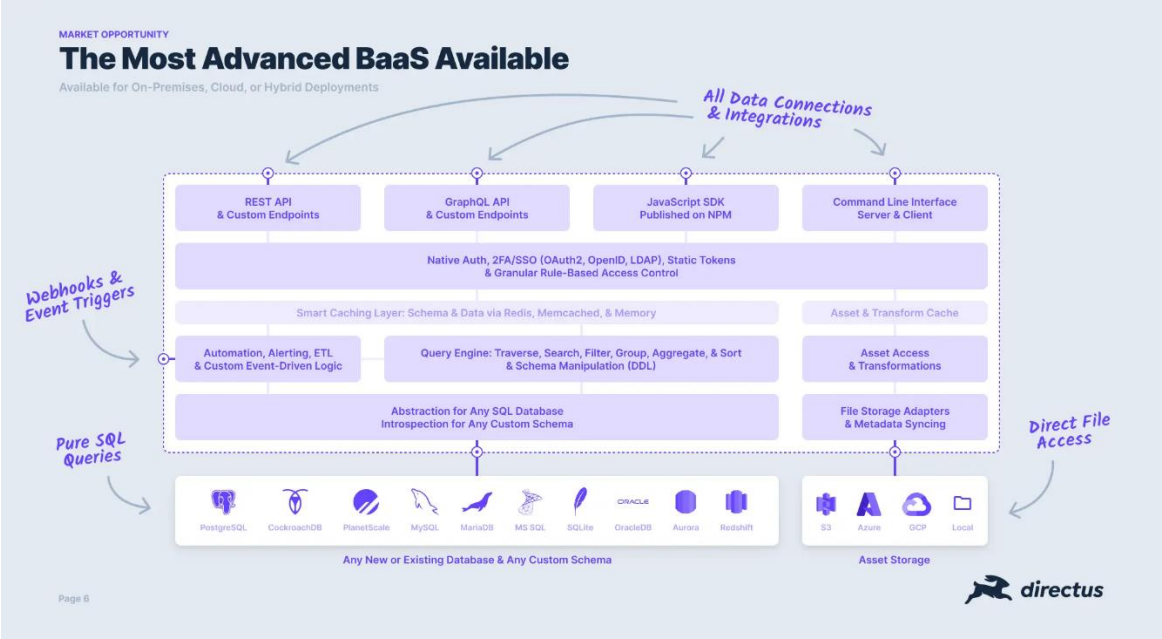
Zdroj: (Directus 2023)

3.6.3.1 Modulová architektura

Celková architektura je postavena na modulech. Každý modul přistupuje k datům určitým způsobem, a umožňuje je tak upravovat, V případě, kdy Directus neposkytuje modul, který je v systému potřeba, umožňuje Directus vyvinutí a nasazení svého vlastního. Celý projekt je postavený na programovacím jazyce JavaScript, kdy je na straně serveru využit Node.js, na straně uživatelské pak Vue.js. Celý projekt je open-source. V případě, kdy je potřeba využít funkcionalit, které Directus neobsahuje, je také možné nastavit

dodatečné služby, jako je např. SSO přihlášení či způsob ukládání souborů, kdy jsou podporovány služby AWS či Google. (Introduction 2023)

Obrázek 13: Architektura systému Directus



Zdroj: (Architecture 2023)

Architektura je lehce odlišná od ostatních CMS systémů. Zatímco ostatní systémy přistupují často k datům napřímo, Directus je použit jako vrstva nad samotnou databází. Samotná aplikace a API dynamicky zrcadlí schéma databáze a její obsah. Tento způsob, kterému se říká databázová introspekce, má mnoho výhod. Absolutní kontrola nad schématem databáze SQL, spolu s naprostou transparentností, přenositelností a zabezpečením dat, umožňuje import stávajících databází v nezměněné podobě a bez potřeby migrace.

Uživatelé mají přímý přístup k databázi, což jim umožňuje využívat plný výkon komplexních dotazů SQL. Díky optimalizacím a indexování dochází k výraznému zlepšení výkonu. Directus umožňuje uživatelům přímý přístup k jejich autentickým a neupraveným datům, což představuje zásadní odchylku od běžné praxe. Tato vlastnost systému Directus poskytuje možnost obejít střední vrstvu softwaru, jako jsou API, SDK a aplikace, a přímo se připojit k datům prostřednictvím standardních SQL dotazů. Takový přístup efektivně eliminuje potenciální úzká místa, zbytečnou latenci a proprietární omezení přístupu k datům, což přináší uživatelům významné výhody v rychlosti a flexibilitě při práci s daty. (Architecture 2023)

Architektura systému je pro koncového vývojáře snadno uchopitelná, a především výhodná. Systém Directus v moment, kdy je připojený k databázi, data uživatele nevlastní, ani nijak nemodifikuje. Je nutné dodat, že pro správnou funkcionalitu systému, je pro systém samotný, vytvořeno 20 dodatečných tabulek. Tyto tabulky ale nejsou s dosavadními daty uživatele nijak spjaty, je proto možné od systému Directus kdykoliv ustoupit. API struktura umožňuje vývojáři provádění čistých SQL dotazů. (Architecture 2023)

3.6.3.2 Data Engine

Dalším prvkem v systému je *Data engine*, který je hlavní mozek systému. Tato vrstva systému obsahuje logiku pro přístup, úpravu či dotazování se na data. Tato vrstva tak obsahuje *event triggery*, operace týkající se databázového dotazování a další funkcionality, jako jsou například transformace souborů. Samotné API, které je tvořeno z REST a GraphQL endpointů je generováno dynamicky, na základě datového modelu uživatelů, nakonfigurovaných rolí a přístupových oprávnění. Pro podporu samotného API je dostupné JavaScript SDK, které lze dodatečně nainstalovat přes NPM. Toto SDK umožňuje vývojáři využívat CLI, které umožňuje provádění akcí na straně serveru, jako jsou databázové migrace či reset uživatele. (Architecture 2023)

Directus obsahuje stejně jako ostatní systémy kolekce, kdy kolekce v systému mohou být schémata 1:1 stejná datové tabulce v SQL. Kolekce mohou být ale zároveň také skupiny ostatních kolekcí, či pohledy určené jen ke čtení (jak je typické např. pro PostgreSQL). Stejně jako kolekce v ostatních systémech, obsahuje Directus také pole, které jsou mapovány na sloupce v databázi. Tyto sloupce ale obsahují pouze čistá, nezpracovaná data. Vývojář vytvářením vlastní logiky určuje dodatečný způsob, jak se tato data mají zobrazovat. Tato logika je poté uložena zvlášť od samotných dat. Samotné záznamy pak značí řádky v jednotlivých tabulkách, Directus ale záznam nemapuje 1:1 vůči samotnému řádku v tabulce, nýbrž provádí dodatečné operace, předtím, než záznam zobrazí samotnému uživateli. Takové operace může zahrnovat např. dotazování se na navázané tabulky. (Architecture 2023)

3.6.3.3 Kolekce

Samotný systém obsahuje výchozí kolekce, které nemají s uživatelskými daty mnoho společného. Tyto výchozí kolekce zajišťují ukládání dat pro systémovou funkcionalitu. Výchozích kolekcí je okolo dvaceti, kdy každá kolekce značí určitou funkcionalitu systému.

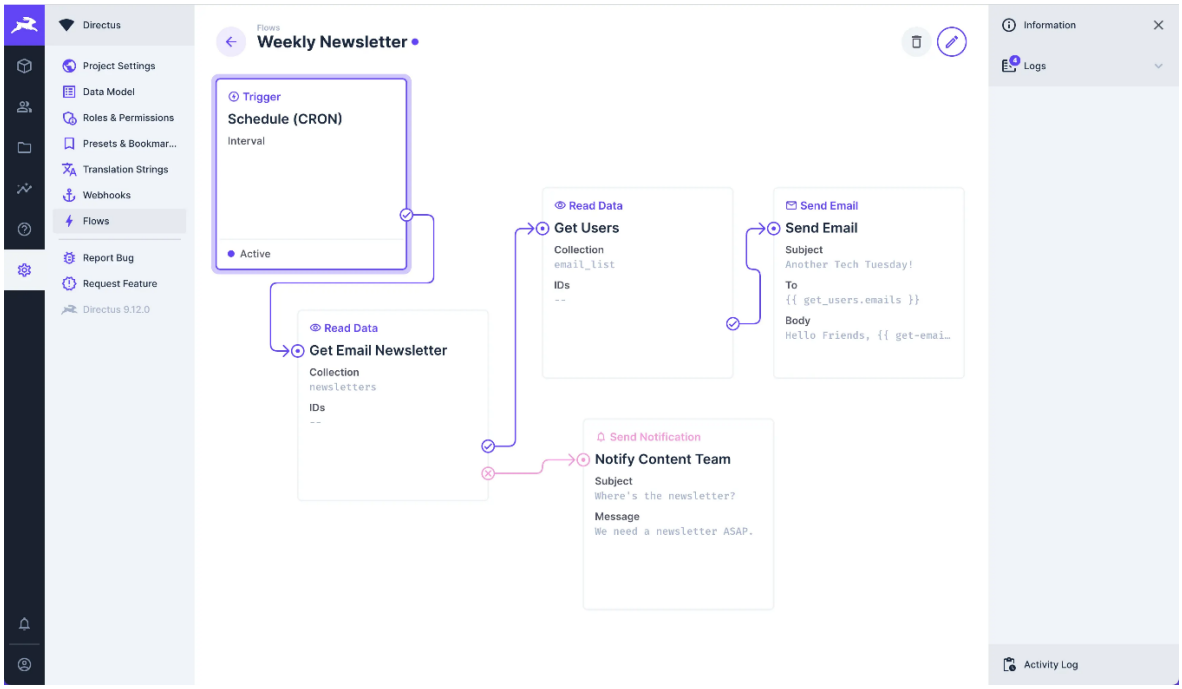
Důležitými funkcionalitami je např. udržení informací o jednotlivých událostech v systému, v případě, kdy došlo k nějaké operaci. Příkladem může být smazání určitého záznamu z kolekce, kterou definoval uživatel. Taková operace je pak zaznamenána v systémové kolekci. Další systémové kolekce mohou být např. role, notifikace či samotní uživatelé. Všechny kolekce poté tvoří jeden funkční datový celek. (Collections 2023)

3.6.3.4 Flows

Nesmírnou výhodou systému jsou takzvané *flows*. Toky umožňují vlastní, událostmi řízené zpracování dat a automatizaci úloh. Každý tok se skládá z jednoho spouštěče, po kterém následuje řada operací. Veškeré toky poté spojuje *data chain*. Samotný spouštěč může být akce nebo událost v rámci samotné aplikace, případně se může jednat o příchozí *webhook* nebo *cron* události. Operace umožňuje provádět akce, které pracují s daty, může se jednat např. o zaslání e-mailu, zaslání *push* notifikace či spuštění dodatečného *webhooku*.

Aby bylo možné v rámci celého toku přistupovat k datům průběžně, vytváří si každý tok svůj datový řetězec. Každá operace z toku má přístup k datovému řetězci, může tak z datového řetězce čerpat informace, nebo je naopak do datového řetězce vkládat. To znamená, že v rámci konkrétní operace lze čerpat data z předchozí operace. Každý tok je unikátní, v rámci první invokace do něj vstupují data, která mohou být ale v rámci druhé invokace jiná. To znamená, že operace může v první invokaci skončit úspěchem, v rámci invokace druhé ale může skončit chybou. Proto umožňuje Directus tok kontrolovat, a spouštět další operace na základě stavu operace předchozí. (Flows 2023) (Triggers 2023) (Operations 2023)

Obrázek 14: Toky v systému Directus

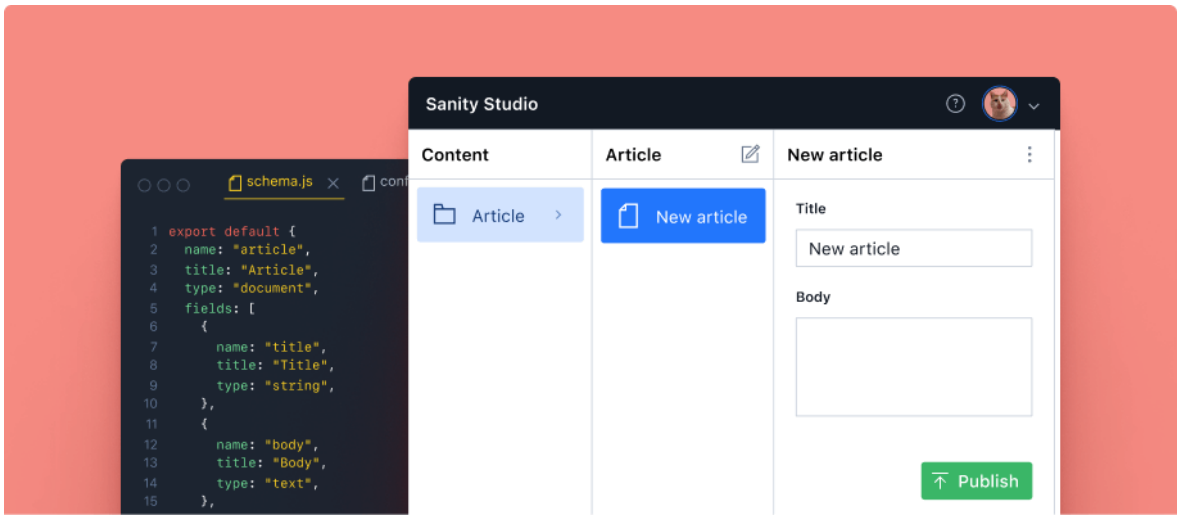


Zdroj: (Flows 2023)

3.6.4 Sanity CMS

Sanity je platformou pro strukturovaný obsah. Jedná se o službu, kde je možné spravovat svůj obsah přes API, které Sanity poskytuje. Kromě toho poskytuje také nástroje a knihovny. Pomocí Sanity lze jednoduše sestavit centralizované místo pro obsah, který lze čerpat z několika projektů. Sanity se rozděluje na službu jako takovou, jde především tedy o místo k uchování dat, zadruhé také poskytuje GROQ. (Sanity Studio 2023)

Obrázek 15: Systém Sanity



Zdroj: (Sanity Studio 2023)

Není tedy divu, že Sanity poskytuje pro své zákazníky převážně Sanity Studio. Jedná se o open-source jednostránkovou aplikaci, kterou lze jednoduše modifikovat a upravovat dle svého přání. Studio využívá programovacího jazyka JavaScript, a pro frontend kód využívá knihovny React. Pomocí něj je také možné Studio jednoduše modifikovat a personalizovat – převážně pro editory. (Sanity Studio 2023)

Datová schémata pro samotné Strapi probíhá na úrovní JavaScript objektů, které definuje koncový uživatel. Tento princip je odlišný od popisovaného Strapi či systému Directus, kde se datové schéma nastavuje na úrovní grafického rozhraní. Ačkoliv se tento způsob může zdát složitý, pro vývojáře a koncové zákazníky, kteří oplývají technickými znalostmi, je tento přístup velmi užitečný. Definování datového schématu tak může být jednoduché. Tento přístup ale není vhodný v případech, kdy již datový model existuje, a CMS je potřeba upravit dle samotného datového modelu. V určitých momentech Sanity vázne v obousměrných relacích, kdy článek může mít autora. Sanity ale automaticky nevytváří vazbu od autora ke článkům. Pokud je potřeba vytvořit takovou relaci, musí se

vytvořit ručně. V tradičním systému, který implementuje SQL, bychom definováním relace mohli získat data z obou stran – jak od autora, tak od článku. (Putting Sanity.io to the test 2020)

Sanity také poměrně limituje vnořené struktury. Sanity obecně nedoporučuje používání vnořených struktur v datovém schématu, které pramení z tendence vývojářů členit obsah do několika struktur. Takovou ukázkou může být samotná mapa webu, která je tvořena z několika stránek, které mají určité datové typy a určitý obsah, který může být rozdělený na několika drobnějších celků. Tento styl přemýšlení Sanity nedoporučuje, místo toho doporučuje zakládat každou entitu (stránka, navigace, patička) jako samostatnou strukturu obsahu. (Pecoraro 2021)

Sanity doporučuje dva přístupy. Jedním z nich je právě zmíněné vnořování. Kdy samotný *meeting* může mít v sobě vnořený *topic*, ale jako samostatnou datovou strukturu. Definování této datové struktury usnadňuje dotazování se pomocí GROQ a dodržování principu DRY. (Pecoraro 2021)

Druhý přístup zakládá *topic* jako samostatný dokumentový typ, který je dotazován přes relace, které je možné v Sanity implementovat. Tento přístup ale není jednoduchý, z podstaty relací je totiž velmi složité smazat dokument, který může být navázaný na jiný dokument z jiného dokumentového typu. Z tohoto důvodu je doporučováno relace definovat, v tomto případě, na straně *topicu*. Takto definovaná relace koncového uživatele upozorňuje na prerekvizity (jiné dokumentové typy), které je potřeba smazat, než bude možné smazat koncový dokument. Sanity editorům umožňuje použít nástroj, který mazání navázaných dokumentů provede, jeho účinnost ale není stoprocentní. (Pecoraro 2021)

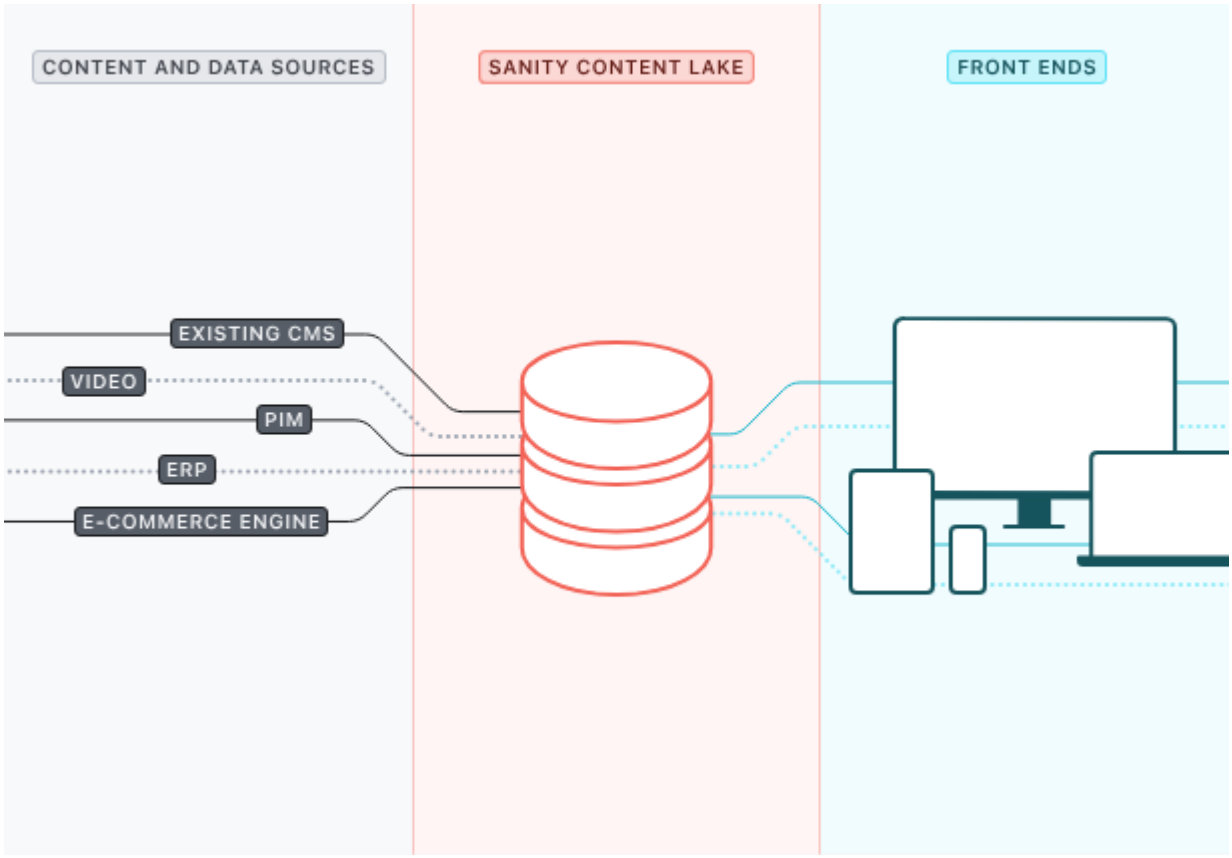
Sanity umožňuje definování několika datových typů pro jednotlivá pole. Některé z těchto polí jsou primitivní, jako například datový typ *boolean*, zatímco některá pole jsou složitější. Jedná se především o datové typy *Geopoint* či *Image*. Jak bylo zmíněno, Sanity umožňuje definování relací, tyto relace jsou ale omezené pouze na dva typy, one-to-one a one-to-many. (Pecoraro 2021) (Putting Sanity.io to the test 2020)

3.6.4.1 Datastore

Datastore, jak jej Sanity označuje, je nejdůležitějším prvkem Sanity. Celá služba je vytvořena jako strukturovaný obsah, který má být bohatší než dvoudimenzionální tabulky. Zároveň má být GROQ jednodušší na pochopení než SQL, které je spíše používáno pro

dotazování se na databázi. Z názvu je patrné, že se jedná o Graph Oriented Query, tedy jazyk, který se má dotazovat na data, která jsou uložena v grafové struktuře. (Pecoraro 2021)

Obrázek 16: Schéma Content Lake



Zdroj: (Sanity Content Lake 2023)

Samotný jazyk má čtyři hlavní body, které definují jeho funkcionalitu. Prvním z nich je expresivní filtrování, které umožňuje jednodušší dotazování a filtraci. Filtrace probíhá jako obyčejný zápis s rovnítky, kde se na levé straně definuje, které pole se má filtrovat, a na pravé straně se definuje hodnota, podle které se má filtrovat. Tento zápis je známý z GQL. Důležitou funkcionalitou je jednoduché spojování více rozdílných dokumentů, které mohou být jiného dokumentového typu. V jazyce SQL je toto prováděno zápisem s použitím vnitřních, vnějších či levých a pravých joinů. Zde je dotazování a spojování záznamů prováděno v největší míře automaticky. (Putting Sanity.io to the test 2020) (Pecoraro 2021)

Třetí funkcionalita umožňuje změnu výsledku v samotném dotazu, kdy lze určitá pole přesouvat, měnit jejich název či hodnoty. Tento zápis je podobný SELECT dotazu z SQL, ačkoliv se jedná o změnu paradigmatu, zkušenější vývojář mající zkušenost s SQL

by s těmito dotazy neměl mít problém. Čtvrtou funkcionalitou je samotná kompaktnost, výsledný dotaz bývá často kompaktnější než ekvivalent v jazyce SQL. V rámci dotazování se na Content Lake je zároveň umožněno do Sanity nainstalovat plugin Vision, který je ekvivalentem jakéhokoli GQL klienta. Takový klient pak umožňuje jednoduchou analýzu možných dotazů směrem na BE – v tomto případě na službu Content Lake. V rámci tohoto pluginu je možné provádět také samotné dotazy, kdy je možné dotazům dodávat také parametry, pomocí kterých je poté sestaven výsledný dotaz. Samotné výsledky dotazu jsou automaticky ve formátu JSON. (Putting Sanity.io to the test 2020)

SQL pracuje na paradigmatu, kdy je nutné definovat přesnou strukturu dat, aby byly dle této struktury vytvořeny tabulky, do kterých se bude výsledný obsah dat ukládat. V případě, že existují data, která je nutné do tabulky uložit, a tento obsah se neshoduje se strukturou tabulky, takový obsah nebude uložen. (Putting Sanity.io to the test 2020)

Obsahový model je rozdílný v *schemaless* přístupu, který je podobný dokumentovým databázím. Není potřeba definovat tabulky a sloupce, systém ukládá hodnoty ve formátu klíč / hodnota. Takový systém velmi zjednodušuje samotný vývoj, který se vyplatí v agilním prostředí, kde jsou nároky na systém v prvních iteracích často měněny. Tento přístup se naopak nevyplatí u velmi relačních, a na integritu dat náročných systémů. (Putting Sanity.io to the test 2020) (Pecoraro 2021)

Pro klienty, kteří nepodporují GROQ, a potřebují se dotazovat na obsah v Content Lake, je možné použít GQL. (Pecoraro 2021)

4 Vlastní práce

4.1 Výkonnostní analýza

Tato analýza se zaměřuje na porovnání výkonu systémů Payload, Directus a Strapi. Test simuluje situaci, kdy je nutné provést GraphQL dotaz na složitý *mega menu* dokument obsahující 30-50 odkazů na další stránky, příspěvky apod., a to včetně relačních vazeb na média, jako jsou ikony a obrázky. Důležité je zdůraznit, že tento dokument je potřeba načítat pro každý serverem vykreslený pohled aplikace či webové stránky, což může představovat významnou zátěž pro server, zejména pokud není systém pro správu obsahu optimalizovaný.

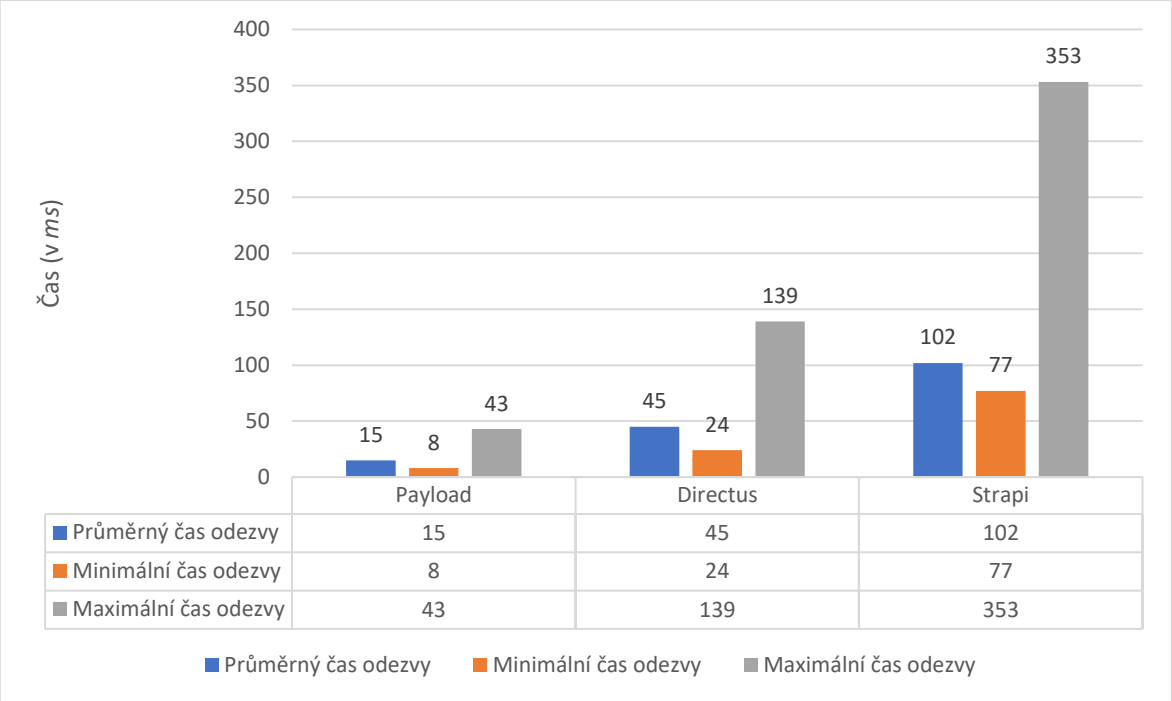
V rámci testování byla navržena struktura dokumentu s více než 60 vztahy a komplexními datovými strukturami, jako jsou skupiny, pole, vnořená pole a bloky. Dokument byl v každém systému vytvořen identicky a dotazy GraphQL byly stejné s výjimkou specifické syntaxe jednotlivých CMS. Reprezentovat výše zmíněné schéma je v systému Sanity obtížné, z důvodu, kdy není možné hostovat server Sanity. Provádět dotaz na server Sanity je zároveň neetické, a v rámci benchmarku by byly ostatní systémy znevýhodněny. Tento systém byl proto z analýzy vyřazen.

- Payload se ukázal jako nejvýkonnější s průměrnou odezvou 15 ms, maximální odezvou 43 ms a minimální odezvou 8 ms. Celkový čas testování byl 1513 ms.
- Directus vykázal průměrnou odezvu 45 ms, maximální odezvu 139 ms a minimální odezvu 24 ms. Celkový čas testování byl 4459 ms.
- Strapi mělo průměrnou odezvu 102 ms, maximální odezvu 353 ms a minimální odezvu 77 ms, což z něj činí systém s nejslabším výkonem. Celkový čas testování dosáhl 10172 ms.

Z těchto výsledků vyplývá, že Payload je výrazně efektivnější ve zpracování složitých dotazů a odpovídá rychleji než Directus a Strapi. V kontextu moderních frontendových frameworků, jako jsou Next nebo Gatsby, které často předvykreslují stránky a mohou během procesu sestavení generovat vysoký počet požadavků na API, se může výkonová účinnost systému Payload projevit jako kritický faktor pro snížení nároků na serverové zdroje a náklady. Vzhledem k výsledkům je Payload vhodnější volbou pro projekty vyžadující optimalizovaný výkon při SSR, zejména při manipulaci s datově náročnými strukturami, jako jsou *mega menu* dokumenty. Directus a Strapi, ačkoliv

poskytují vyšší flexibilitu v některých aspektech, se mohou stát limitujícím faktorem v kontextu výkonu a škálovatelnosti.

Graf 1: Výkonnostní porovnání CMS



Zdroj: vlastní zpracování

Kód 3: Ukázka funkce pro dotázání se na Strapi GraphQL endpoint

```
async function performStrapiQuery(authHeader: string, query: string) {
  await fetch('http://localhost:1337/graphql', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      Authorization: authHeader,
    },
    body: JSON.stringify({
      query,
    }),
  })
}
```

Zdroj: vlastní zpracování

4.2 Analýza popularity

V období mezi lety 2018 a 2024 lze z grafu pozorovat vývoj popularity headless systémů na základě počtu hvězd na platformě GitHub. Počet hvězd je často používán jako indikátor zájmu a přijetí uživatelské a vývojářské komunity.

Systém Strapi, reprezentovaný žlutou barvou, dosahuje výrazného nárůstu popularity, což je vyjádřeno nejvyšším počtem hvězd překračujícím 50 000 ke konci sledovaného období. Stabilní růst naznačuje, že se systém Strapi stal jedním z nejpreferovanějších headless CMS, což může být přisouzeno jeho otevřenému zdrojovému kódu, rozsáhlé funkčnosti a aktivní komunitní podpoře.

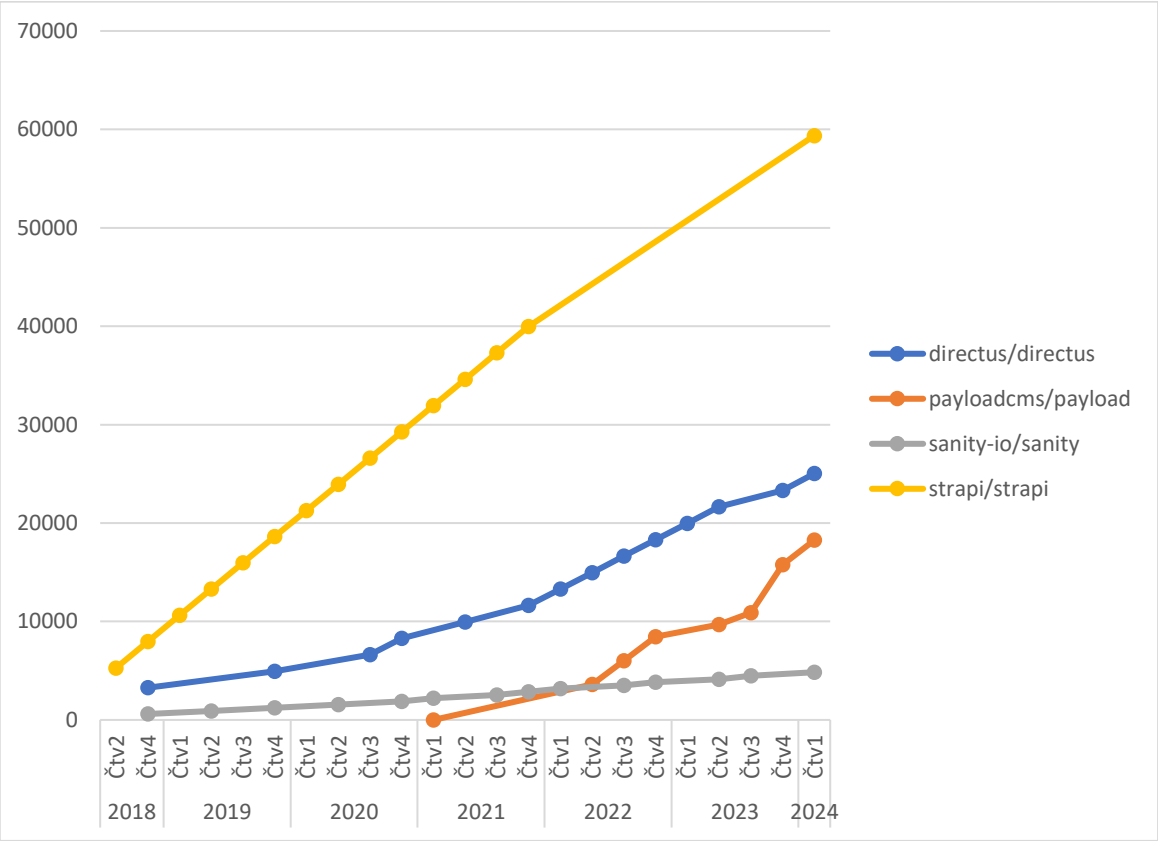
Directus, který je v grafu znázorněn modrou barvou, také ukazuje konzistentní růst, s hodnotou dosahující přibližně 30 000 hvězd v roce 2024. Popularita systému Directus může být odrazem jeho uživatelské přívětivosti a širokých možností přizpůsobení, které usnadňují práci s daty v rámci různých aplikací.

Payload CMS, indikovaný oranžovou barvou, ukazuje strmý růst a dosahuje více než 20 000 hvězd v roce 2024. Jeho uživatelská základna se postupně rozrůstá, což může odrážet jeho rostoucí přijetí mezi vývojáři, kteří hledají efektivní a moderní řešení pro správu obsahu.

Sanity, představený šedou křivkou, má v grafu nejméně hvězd, avšak i tato platforma vykazuje růst. To může signalizovat, že ačkoliv není tak široce uznáván jako jeho konkurenti, stále získává trakci mezi vývojáři, kteří hledají specifické funkce nebo přístupy, které Sanity nabízí.

S rostoucím počtem hvězd pro tyto platformy je zřejmé, že headless CMS se stávají stále populárnějšími pro správu digitálního obsahu, přičemž Strapi vede tuto skupinu s nejvyšším počtem hvězd, následovaný systémem Directus, Payload CMS a Sanity.

Graf 2: Popularita jednotlivých CMS



Zdroj: vlastní zpracování dle (Bytebase 2024)

4.3 Vícekriteriální analýza variant

K posouzení různých systémů bylo využito metod vícekriteriálního hodnocení. Specificky byla aplikována metoda Simple Additive Weighting (SAW), což je přístup založený na optimalizaci lineárního hodnotového modelu. Nejdříve je stanovena ideální varianta H a bázová varianta D, přičemž ideální varianta dostává maximální užitkovou hodnotu 1 a bázová varianta minimální hodnotu 0. Hodnoty užitku jednotlivých systémů se poté pohybují mezi těmito dvěma extrémy. Pro výpočet je vytvořena normalizovaná matice R, kdy je každý prvek vypočítán dle vztahu:

$$r_{ij} = \frac{y_{ij} - d_j}{h_j - d_j}$$

kde r_{ij} reprezentuje standardizovanou hodnotu pro dané kritérium, y_{ij} je původní hodnota kritéria, d_j je hodnota bázové varianty a h_j je hodnota ideální varianty pro toto kritérium.

Kritéria pro hodnocení v tabulce vícekriteriální analýzy jsou rozmanitá a zahrnují jak technické aspekty, tak obecnější charakteristiky. Každé kritérium je dále definováno typem (zda je lepší dosáhnout maximální hodnoty, označeno jako MAX, nebo minimální hodnoty, označeno jako MIN) a stupnicí znamenající poměrné ohodnocení. Váha každého kritéria naznačuje jeho důležitost v celkovém hodnocení, přičemž některé aspekty mají větší význam než jiné. Například "Cena" má největší váhu 0,20, což ukazuje, že je to významný faktor v rozhodovacím procesu. Kritéria zahrnutá v analýze jsou:

1. Open source – preference softwaru s otevřeným zdrojovým kódem (váha 0,10)
2. Popularita – preference softwaru, který je na trhu populárnější (váha 0,02)
3. Stáří – preference softwaru, který je na trhu déle (váha 0,01)
4. Noření komponent – schopnost systému efektivně pracovat s komponentami (váha 0,10)
5. SEO podpora – podpora pro optimalizaci vyhledávačů (váha 0,02)
6. Rozšiřitelnost – jak dobře lze systémy rozšiřovat (váha 0,05)
7. Lokalizace – podpora pro lokalizační aspekty (váha 0,10)
8. Verzování – schopnost spravovat různé verze obsahu (váha 0,02)
9. TypeScript podpora – podpora pro TypeScript (váha 0,10)
10. Rich-text editor – přítomnost a kvalita rich-text editoru (váha 0,05)
11. Zabezpečení – úroveň zabezpečení systému (váha 0,02)

12. Riziko vendor lock-in – hodnocení na základě toho, jak snadno lze systém přestat používat a začít používat systém jiný (váha 0,10)
13. Podpora Next.js - podpora pro framework Next.js (váha 0,10)
14. Cena (váha 0,20)

Celkový součet vah těchto kritérií je roven 1, což znamená, že každé kritérium má přesně definovaný vliv na konečné hodnocení a že všechny váhy dohromady tvoří celý rozhodovací model.

Tabulka 6: Kritéria pro VAV

Kritéria	Typ	Stupnice	Váha	Hodnota
Open source	MAX	POM	0,099	4
Popularita	MAX	POM	0,025	1
Stáří	MAX	POM	0,012	0,5
Noření komponent	MAX	POM	0,099	4
SEO podpora	MAX	POM	0,025	1
Rozšiřitelnost	MAX	POM	0,049	2
Lokalizace	MAX	POM	0,099	4
Verzování	MAX	POM	0,025	1
TypeScript podpora	MAX	POM	0,099	4
Rich-text editor	MAX	POM	0,049	2
Zabezpečení	MAX	POM	0,025	1
Riziko vendor lock-in	MIN	POM	0,099	4
Podpora Next.js	MAX	POM	0,099	4
Cena	MIN	POM	0,198	8
Celkem			**1**	**40,5**

Zdroj: vlastní zpracování

Na základě definovaných kritérií a přidělených vah byla stanovena následující skóre pro jednotlivé systémy. Payload CMS se s celkovým skóre 0,947 umístil na první pozici, což naznačuje jeho významnou převahu vzhledem k přiděleným váhám kritérií. Directus obdržel skóre 0,75, což mu zajistilo druhé místo v hodnocení, což ukazuje silnou konkurenceschopnost v mnoha kritických oblastech. Strapi, s celkovým skóre 0,525, byl hodnocen jako třetí, což odhaluje jeho vyváženou kombinaci silných a slabých stránek ve srovnání s ostatními hodnocenými systémy. Sanity CMS, i přes některé své silné stránky, se s celkovým skóre 0,219 umístil na poslední, čtvrté místo, což naznačuje oblasti, ve kterých by mohl být systém zlepšen.

Tabulka 7: Výsledek metody SAW

SAW	Directus	Payload CMS	Sanity CMS	Strapi
Open source	1	1	0	1
Popularita	0,667	0,333	0	1
Stáří	0,5	0	0,25	1
Noření komponent	0,333	1	0,667	0
SEO podpora	1	0	0	0
Rozšiřitelnost	0,75	1	0	0,5
Lokalizace	0,8	1	0	0,6
Verzování	1	1	0	0,5
TypeScript podpora	1	1	0,333	0
Rich-text editor	0,333	1	0	0,667
Zabezpečení	0,75	1	0,75	0
Riziko vendor lock-in	1	1	0	0,8
Podpora Next.js	0	1	1	0,5
Cena	1	1	0	0,667
Skóre	**0,75**	**0,947**	**0,219**	**0,525**
Pořadí	**2**	**1**	**4**	**3**

Zdroj: vlastní zpracování

4.4 Nastavení monorepositáře

Pro účely nasazení webové stránky společně s redakčním systémem byla zvolena metoda monorepositáře. S pomocí monorepositáře je možné mít v jednom repositáři více projektů. V tomto případě monorepositář obsahuje dva hlavní projekty, webovou aplikaci a samotnou administraci. V dnešním světe se u monorepositářů používá nástrojů, které mají samotné sestavení aplikace zjednodušit a zrychlit. Využívají cachování v cloudu, nastavují pravidla, která zabraňují cirkulárním importům aj. V tomto projektu nebyly tyto nástroje využity, z důvodu, kdy jsou pro dvě aplikace zbytečně složité a představovaly by nadbytečnou komplexitu, která nepřinese žádný reálný užitek. Z toho důvodu byly využity takzvané *workspaces*, které jsou součástí balíčkovacího systému NPM. Tento termín je také dostupný v balíčkovacích systémech, které jsou na NPM založeny. Jedná se o Yarn či PNPM.

Pro nastavení *workspaces* bylo potřeba přidat níže uvedené do samotného *package.json*, který se nachází v kořenové složce projektu. Yarn si poté dokáže jednotlivé složky dohledat, načíst *package.json* pro každý *workspace* a pracovat s ním. Jakákoli instalace balíčků se musí provádět v patřičném *workspace*, samotný stažený balíček je ale uložený v *node_modules* v kořeni projektu, nikoliv u patřičného workspace. Toto pravidlo neplatí pouze v případě, kdy je stažený balíček označený jako určený pro vývoj. Takový balíček je uložený u *workspace*, ve kterém byl nainstalován. Do kompilovaných souborů ale zahrnut není. Nevýhodou nevyužití nástrojů NX nebo Turborepo je nutnost využití balíčků *concurrently* a *wait-on*. Ty slouží k usnadnění spouštění obou projektů zároveň při lokálním vývoji. Při sestavení webové aplikace musí již běžet administrace, proto musí být před spuštěním webové aplikace dotazován *localhost* na portu 3000, zda administrace běží, než dojde k sestavení webové aplikace.

Kód 4: Definování workspaces v souboru package.json

```
"private": true,
"workspaces": [
  "app",
  "payload"
],
```

Zdroj: vlastní zpracování

4.4.1 Adresářová struktura webové aplikace

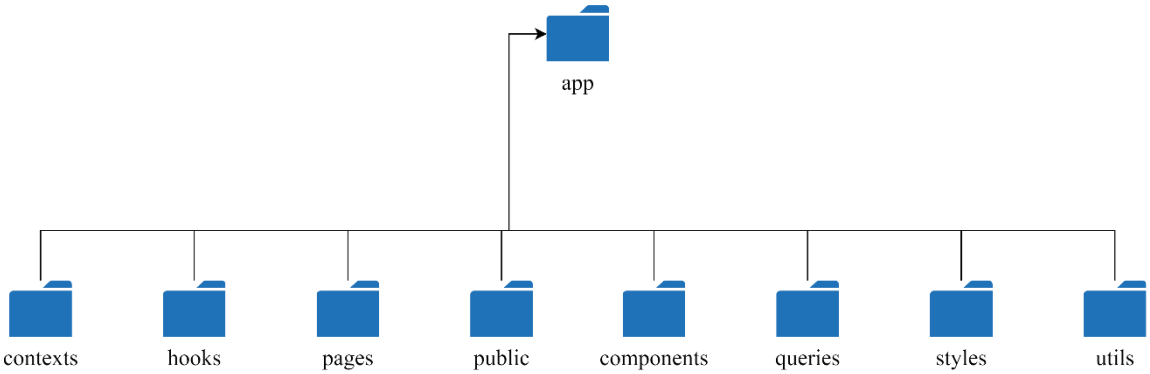
Webová aplikace kopíruje adresářovou strukturu na základě doporučených praktik pro Next.js. Ve složce *contexts*, jsou k nalezení kontexty pro samotný React. Tyto kontexty jsou využívány pro udržení informací napříč celou aplikací. Informace se do tohoto kontextu dostávají ze samotné administrace. Přenos těchto informací z administrace do kontextu probíhá v moment samotné statické generace. Složka *hooks* obsahuje jednotlivé funkce pro React, zvané *hooky*, které mají být v moment zavolání komponenty zavolány také a nepodmínečně. V těchto *hooks* je možné zároveň využívat další *hooks*, ať už vlastní, či z knihoven třetích stran. Ve složce *pages* je již samotná struktura webové aplikace, kde je k nalezení mnoho souborů, které jsou poté v samotné webové aplikaci zobrazovány jako stránky.

U tohoto projektu byl použit *pages* router, na rozdíl od nového *app* routeru. Důvodem byla nestabilita a testovací provoz *app* routeru v době tvorby webové aplikace. Samotná složka kromě jednotlivých stránek obsahuje také API funkce, ať již pro revalidace stránek či pro zasílání e-mailů. Složka *public* obsahuje soubory, které mají být veřejně dostupné samotnému klientovi. Typickým příkladem může být například mapa webu, favikony či soubor *robots.txt*. Společně s *pages* složkou je také složka *components* složkou nejobjemnější. Ta obsahuje 51 komponent, které jsou použity napříč celou webovou aplikací. Většina těchto komponent je mapována na bloky, které jsou definovány v administraci.

Queries obsahuje dotazy, definované a strukturované tak, aby se jejich použitím dala provolávat administrace v moment statické generace, a bylo tak možné z administrace získávat samotná data. *Styles* obsahuje základní globální styly, které jsou použity napříč celou webovou aplikací. Tyto globální styly jsou navíc dodatečně rozšířeny o samotnou knihovnu Tailwind CSS a její direktivy. Ve složce jsou také definovány JavaScript objekty, které v sobě obsahují konstanty, které jsou používány při dynamickém připínání stylů

v závislosti na hodnotě ze samotné administrace. V neposlední řadě složka *utils* obsahuje především pomocné funkce. Ty slouží ke stránkování, spojování jednotlivých názvů CSS tříd či serializování objektu z *rich text* editoru definovaném v administraci.

Obrázek 17: Adresářová struktura pro webovou aplikaci



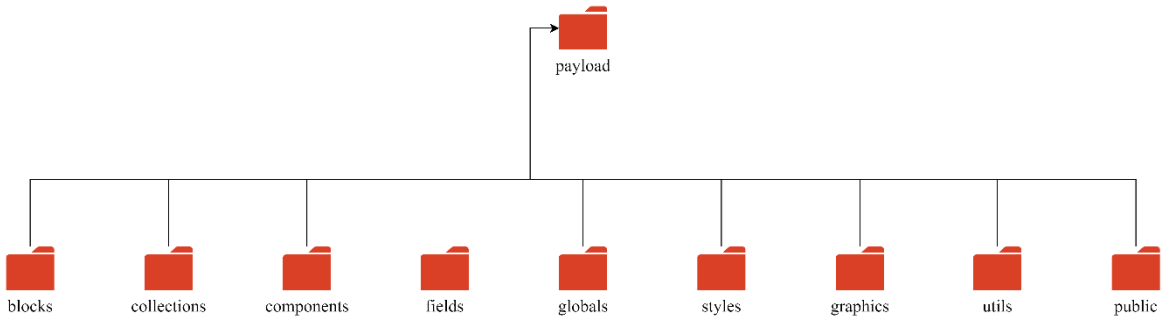
Zdroj: vlastní zpracování

4.4.2 Adresářová struktura administrace

Samotná administrace má lehce složitější strukturu. Redakční systém Payload nenastavuje žádná pravidla pro adresářovou strukturu, doporučuje ale některé postupy, které by měly být vývojáři dodržovány. Z toho důvodu obsahuje složka *blocks* jednotlivé bloky, které lze přidávat v samotné administraci do jednotlivých typů obsahu, ať už se jedná o stránky či příspěvky. Nejdůležitější složkou je *collections*. Ta obsahuje kolekce. Jedná se o stránky, příspěvky, média, videa aj. Stejně jak tomu je u webové aplikace, je možné v administraci použít vlastní React komponenty, které lze poté zobrazovat na určitých místech administrace. Tyto komponenty jsou obsaženy ve složce *components*. V rámci úpravy obsahu je možné přidávat vlastní pole, například takové, pomocí kterých lze vybrat barvu textu, či pole pomocí kterého lze zvolit, na jakou stranu se má obsah zarovnat. Tato pole jsou uložena ve složce *fields*.

Kolekce jsou jak singulární, tak globální. Globální kolekce jsou obsaženy ve složce *globals* a obsahují kolekce pro nastavení patičky, hlavičky či cookies lišty. Systém Payload ve výchozím nastavení obsahuje grafiku, kterou lze nahradit. Pro nahrazení této grafiky je vhodné vytvořit nové komponenty, které jsou uloženy ve složce *graphics*. Pro pomocné funkce stejně jako u webové aplikace existuje složka *utils*, pro soubory, které mají být dostupné klientovi pak existuje složka *public*.

Obrázek 18: Adresářová struktura pro administraci



Zdroj: vlastní zpracování

4.5 Nasazení serveru

Pro provozování administrace Payload je potřeba disponovat serverem s OS Linux, na kterém lze provozovat Docker. Vzhledem k cenám a specifikacím, které nabízejí čeští poskytovatelé Forpsi či WEDOS, byl zvolen německý poskytovatel Hetzner. Hlavním bodem pro výběr tohoto poskytovatele byl poměr cena / výkon, kdy čeští poskytovatelé nabízejí menší kapacitu úložiště, méně výkonný procesor a za vyšší cenu. U poskytovatele Hetzner byl vybrán stroj CPX11. Tento stroj nabízí 2 virtuální procesory AMD EPYC 7002, 2 GB RAM a 40 GB SSD. Mnoho poskytovatelů také na tyto servery aplikuje limit na přenos dat za měsíc. U tohoto poskytovatele je limit 20 TB. Takový přenos dat je ale skoro nemožné překročit při provozování administrace a aplikace.

Lokace serveru je v datacentru poskytovatele Hetzner ve městě Falkenštejn. Toto město přímo hraničí s Českou republikou na její severozápadní straně. Pro klienty, kteří se připojují z České republiky, je přenos dat velmi rychlý s minimální odezvou. Pokud porovnáme lokaci s jinými poskytovateli, zjistíme, že je lokace datacentra Hetzner velmi výhodná. Jiní poskytovatelé, např. takové AWS, nabízí několik lokací po celém světu. Nejbližší lokací je pro Českou republiku Frankfurt, který je ale vzdálenější než samotný Falkenštejn.

Obrázek 19: Specifikace serveru

Zdroj: (Servers 2023)

Obrázek 20: Datové centrum ve městě Falkenštejn

Zdroj: (Hetzner 2023)

4.6 Docker konfigurace

Provozování dvou aplikací na jednom serveru vyžaduje použití manažeru procesů nebo softwaru pro izolaci aplikací do kontejnerů. Nejznámější manažer procesů pro Node.js aplikace je PM2. Ten je často dostačující v určitých případech, ale u dvou aplikací, které mají uchovávat data, a komunikovat mezi sebou či okolním světem, je vhodnější použít Docker. Ten se nejenom hodí pro kontejnerizaci těchto dvou aplikací, ale zároveň je jednoduché v něm zprovoznit instanci MongoDB. Každý kontejner musí mít svůj Docker obraz, ve kterém jsou definovány instrukce, jak má být sestaven. Docker obraz pro samotnou webovou stránku v sobě obsahuje instrukce pro tři etapy. První je etapa závislostí. V této etapě dochází k instalaci závislostí na základě souboru *package.json*. V případě, že nedojde ke změně tohoto souboru, bude tato etapa uložena do mezipaměti, aby nemusela být při dalším opětovném sestavení obrazu opakována. To stejné platí pro druhou etapu samotného sestavení projektu. Ve třetí etapě samotného spuštění aplikace dochází ke zkopírování určitých souborů z druhé etapy, zpřístupnění portu 5000 a samotného spuštění serveru. Velmi podobný postup platí také u Docker souboru pro administraci.

Pro provoz těchto kontejnerů je využit Docker Compose. V Docker Compose souboru jsou definovány tři kontejnery, které budou po spuštění příkazu *docker compose up* připraveny. První kontejner je webová aplikace, druhý kontejner je administrace. Třetím kontejnerem je samotná databáze. Pro správné směřování požadavků na základě subdomény v adrese je využita reverzní proxy Traefik. Ta není přímo uvedena v Docker Compose souboru, kde je uvedena aplikace a administrace, místo toho je nasazena pomocí separátního Docker Compose souboru. Toto řešení umožňuje budoucí nasazení více aplikací, které na sobě mohou být nezávislé, ale bude na nich odkazováno přes jednu instanci proxy Traefik.

Pro správnou funkčnost administrace je nutné v Docker Compose souboru definovat *volumes*. Jedná se o svazky na disku, které mohou nebo nemusí být přiřazeny jednomu z kontejnerů. Tyto svazky jsou velmi užitečné, pokud mají být data zachována v případě, kdy dojde k zastavení nebo smazání kontejneru. Pro administraci byly vytvořeny tři svazky: videa, dokumenty a média. Pro správnou funkcionalitu MongoDB byl vytvořen čtvrtý svazek *data*, který slouží k ukládání databázových souborů.

Kód 5: Definované svazky přes Docker Compose

```
volumes:
  videos:
  documents:
  media:
  data:
  node_modules:
```

Zdroj: vlastní zpracování

Důležitým nastavením v ekosystému Docker jsou sítě. Ty umožňují komunikaci jednotlivých kontejnerů mezi sebou. Vzhledem ke skutečnosti, kdy webová aplikace načítá data ze samotné administrace, je nutné tuto síť nakonfigurovat. Stejně jako je tomu u reverzní proxy Traefik, je síť nastavena v rámci jiného Docker Compose souboru. Samotná webová aplikace a administrace je poté na tuto síť připojena. Po připojení těchto kontejnerů k síti, je možné nastavit *labels*. Jedná se o štítky, které lze nastavit u každého kontejneru a nesou informační hodnotu, ať už pro kontejner samotný či pro kontejnery na samotné síti.

Kód 6: Definování sítě přes Docker Compose

```
networks:
  network:
    name: traefik-network
```

Zdroj: vlastní zpracování

Vzhledem k využití reverzní proxy Traefik, je potřeba, aby oba kontejnery měly přiřazené štítky se správnou informací. Z tohoto důvodu webová aplikace obsahuje štítek, který zapíná Traefik pro samotný kontejner, nastavuje správnou Docker síť, port a poté nastavuje pravidla, při jaké webové adrese má být klient na tento kontejner odkazován. Z ukázky níže je patrné, že klient bude odkázán na administraci v moment, kdy se bude webová adresa přesně shodovat s *cms.energosavings.cz*. Traefik navíc disponuje generováním bezpečnostních certifikátů, které jsou automaticky prodlužovány. Pro vynucení a automatický přepis nezabezpečeného připojení na připojení šifrované je v samotném Docker Compose souboru pro Traefik definováno pravidlo. Tento přepis je označený jako permanentní, a proto v prohlížeči vyvolává status kód 301.

Kód 7: Definování štítků přes Docker Compose pro administraci

```
labels:
  - traefik.enable=true
  - traefik.docker.network=traefik-network
  - traefik.http.services.payload.loadbalancer.server.port=3000
  - traefik.http.routers.payload.rule=Host(`cms.energosavings.cz`)
  - traefik.http.routers.payload.tls.certresolver=myresolver
  - traefik.http.routers.payload.tls=true
  - traefik.http.routers.payload.tls.domains[0].main=cms.energosavings.cz
  - traefik.http.routers.payload.tls.domains[0].sans=www.cms.energosavings.cz
```

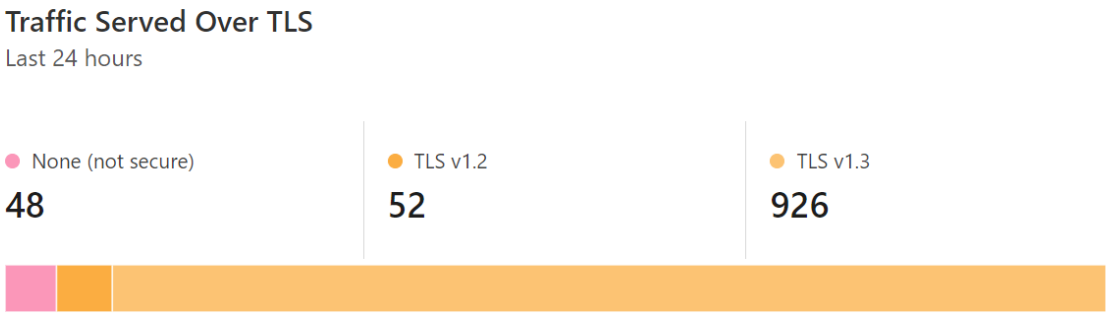
Zdroj: vlastní zpracování

4.7 Nastavení CDN

Nasazení sítě pro doručení obsahu není potřebné pouze v moment, kdy má být obsah dostupný po celém světě, ale lze ji využít i ve chvíli, kdy chceme minimalizovat přenos dat ze serveru, na kterém je webová stránka umístěna. Přenos dat na placených serverech bývá omezený, využití sítě pro doručení obsahu je proto skvělý způsob, jak objem přenášených dat minimalizovat. Pro webovou stránku bylo využito sítě pro doručení obsahu Cloudflare. Ta není využita jen jako síť pro doručení obsahu, umožňuje totiž využití pestré palety funkcionalit.

Pro webovou stránku bylo využito plného SSL/TLS módu. Ten zaručuje šifrované připojení mezi prohlížečem uživatele, samotným serverem Cloudflare a cílovým serverem, na kterém je uložený hlavní obsah. Je zajímavé porovnávat statistiky, jaký počet klientů se připojil a s jakým šifrováním. Podstatná většina se připojuje s šifrováním TLS 1.3, zatímco necelá desetina přes nižší verzi 1.2. Někteří klienti se snaží připojit bez protokolu. Jedná se pravděpodobně o boty. Samotná funkcionalita šifrování je především implementována na úrovní takzvaných *edge* certifikátů, které jsou vygenerovány pro domény **.energosavings.cz* a *energosavings.cz*. Tyto vygenerované certifikáty jsou automaticky prodlužovány, a chráněny navíc dodatečnými záložními certifikáty v případě chybné funkcionality certifikátů primárních.

Obrázek 21: Rozpad jednotlivých šifrování



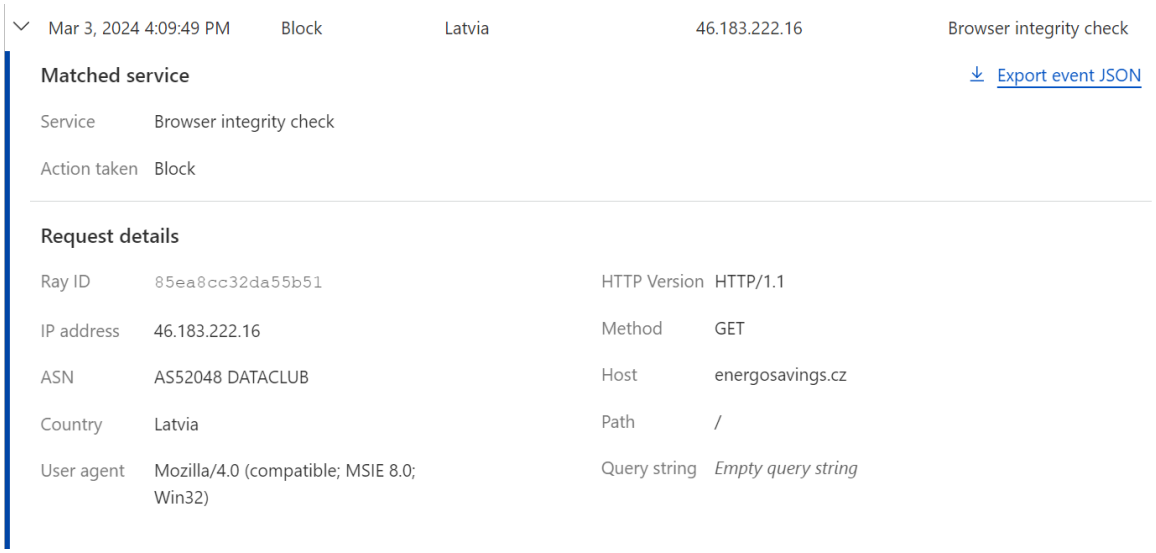
Zdroj: vlastní zpracování dle (Cloudflare 2024)

Webová stránka je také chráněna. Vše je vyřešeno pomocí Cloudflare pravidel, které jsou nastaveny automaticky. Tyto pravidla filtrují jednotlivé požadavky, které se zdají být podezřelé. Tyto požadavky se mohou zdát podezřelé již na úrovni prohlížeče, kdy dochází

k dodatečné CAPTCHA verifikaci. Pokud CAPTCHA verifikace skončí úspěchem, je uživatel na webovou stránku již odkázán, a pro další požadavky označen jako důvěryhodný agent. V opačném případě je mu přístup na webovou stránku odepřen. V dnešní době je velmi důležitou funkcionalitou ochrana proti DDOS útokům. Ty spočívají v provádění velkého počtu dotazů na cílový server, který pod náporem velkého počtu požadavku nedokáže jednotlivé požadavky obstarat. To může vést k odstavení serveru, který neodpoví na dotazy jiných klientů. Ve světě, kdy čas jsou peníze toto znamená také možnou finanční újmu.

Z tohoto důvodu je využita ochrana proti DDOS útokům od Cloudflare, s použitím výchozích pravidel. V praxi bude jakýkoli DDOS útok odstaven již na úrovni Cloudflare, jednotlivé škodlivé dotazy na cílový server nebudou mít vliv. Na obrázku níže je vidět požadavek, který byl detekován jako škodlivý a proto zablokován. Tento požadavek pochází z litevské adresy, konkrétně od poskytovatele serveru Dataclub. Lze tedy předpokládat, že se jedná o automatizovaného bota, který je provozován na konkrétním serveru od výše zmíněného poskytovatele. Nejedná se o reálného zákazníka.

Obrázek 22: Zablokovaný škodlivý požadavek pocházející z litevské adresy



Zdroj: vlastní zpracování dle (Cloudflare 2024)

Samotné zablokování škodlivých požadavků ale mnohokrát nestačí. Na webové stránce jsou uložena citlivá data. Jedná se o telefonní čísla a e-mailové adresy. Ty jsou umístěny v samotné hlavičce webové stránky, a pak u každého zaměstnance. Pokud by nebyly tyto údaje chráněny hrozí, že budou především e-mailové adresy *scrapnuty*, uloženy,

a poté využívány k zasílání spamu. Aby k tomuto nedošlo, je využita obfuskace e-mailových adres pomocí takzvaných *server-side excludes*. Funkcionalita je to poměrně jednoduchá. Cílový server vrací HTML, které je na straně Cloudflare analyzováno, v případě potřeby upraveno a odesláno klientovi. Upravené HTML je uloženo do *cache*, aby nebylo nutné tento proces opakovat při každém požadavku.

Pro zrychlení webové stránky bylo využito několika metod a technologií, které v dnešní době začínají být standardem.

Jedná se především o verze jednotlivých HTTP. Většina webových stránek v dnešní době využívá stále HTTP/1, ačkoliv se jedná již o zastaralý protokol, a měl by být využit novější HTTP/2. Ten umožňuje rychlejší přenos dat, než je tomu u protokolu HTTP/1. Zapnuta je také možnost nejnovějšího HTTP/3, který začíná být podporován u všech webových prohlížečů.

4.8 Nastavení administrace

Pro účely administrace byl zvolen redakční systém Payload. Ten se v projektu vyskytuje ve verzi 1.11.8. Vzhledem ke skutečnostem, jak Payload funguje s Node.js serverem Express, je jakákoli další konfigurace v projektu jednoduchá. Po inicializaci Express serveru byl objekt uložen do neměnné proměnné *app*. S touto proměnnou je poté možné pracovat, přidávat ji nové *request handlery*, nastavovat ji *middleware* či rovnou připnout celé aplikace, kterou je v tomto případě Payload. Pro účely diagnostiky systému v rámci administrace, byl vytvořen nový *request handler*, který využívá balíčku *check-disk-space*. Ten na základě prostředí, na kterém server běží, dokáže spočítat volné místo na disku, a tuto informaci poté z *handleru* vrátit.

Kód 8: Nastavení handleru pro výpočet volného místa

```
app.get("/check-disk-space", (req, res) => {
  const isWin = process.platform === "win32";
  const path = isWin ? process.cwd() : "/";
  checkDiskSpace(path).then((diskSpace) => {
    res.json(diskSpace);
  });
});
```

Zdroj: vlastní zpracování

Důležitá je ale především samotná inicializace systému Payload. V samotné inicializaci, která probíhá asynchronně, dochází k nastavení takzvaného *secret*. Tato hodnota slouží k šifrování hesel, a je proto skryta. Nejdůležitějšími dvěma klíči je *mongoURL* a *express*. Jak je patrné, první klíč slouží k nastavení cesty k databázi. Druhý klíč musí obsahovat neměnnou proměnnou, která odkazuje na instanci Express serveru. Systém Payload umožňuje zaslání e-mailu v případě zapomenutého hesla, či při registraci nového uživatele. Aby toto zasílání e-mailů fungovalo, byly nastaveny patřičné hodnoty v *email* objektu. Pro zasílání e-mailů byl zvolen protokol SMTP. Hostitelem pro zasílání e-mailů je poskytovatel Seznam.cz, u kterého byly vytvořeny všechny e-mailové schránky pro klienta s využitím služby Email Profi. Pro správné zasílání e-mailů byly nakonfigurovány následující hodnoty. Hodnota atributu *SMTP_PASS* byla cenzurována z bezpečnostních důvodů.

Tabulka 8: Definování SMTP hodnot pro email objekt

Název klíče	Hodnota klíče
SMTP_HOST	smtp.seznam.cz
SMTP_USER	info@energosavings.cz
SMTP_PASS	---
PORT	465
SECURE	true

Zdroj: vlastní zpracování

Po správné definici konfiguračního objektu se server pouští na portu 3000. U konfigurace samotného systému Payload byly nastaveny základní hodnoty pro správnou funkcionalitu systému. Pomocí *serverURL* byla nastavena adresa *cms.energosavings.cz*. Další nastavení se týká již administrace samotné. Prvně došlo ke konfiguraci uživatelské kolekce, která se má používat pro přihlášení. Ta byla nakonfigurována tak, aby vyžadovala při první registraci pouze jméno. V případě, že je přidán nový uživatel ze samotné administrace, je možné tomuto uživateli přidat také profilový obrázek. Administrace také obsahuje meta značky pro personalizaci uživatelského prostředí. Byla nastavena meta značka *title*, současně s tím byl nastaven také favikon a Open Graph obrázek pro sociální sítě. Dodatečná úprava, která nemusí být provedena, je dodání komponent, které se mají zobrazovat před a po samotné *dashboard* komponentě. Před tuto *dashboard* komponentu byla umístěna komponenta, která popisuje základní funkcionalitu systému, a klientovi zanechává kontakt pro možnost dodatečné komunikace. Po *dashboard* komponentě je zobrazena komponenta, která udává, kolik volného místa na disku zbývá. Tato komponenta je vhodná jak pro klienta, tak pro administrátora systému.

Kód 9: Základní nastavení systému Payload

```
serverURL: process.env.PAYLOAD_PUBLIC_SERVER_URL,
admin: {
  user: Users.slug,
  meta: {
    titleSuffix: "- Energosavings",
    favicon: "/assets/favicon.svg",
    ogImage: "/assets/logo.svg",
  },
  components: {
    beforeDashboard: [BeforeDashboard],
    afterDashboard: [AfterDashboard],
    graphics: {
      Logo,
      Icon,
    },
  },
  css: path.resolve(__dirname, "./styles/custom.scss"),
},
```

Zdroj: vlastní zpracování

Bylo upraveno bezpečnostní nastavení. Ve výchozím nastavení je limit dotazů na API nastaven na 500 dotazů za vteřinu z jedné IP adresy. Tato hodnota musela být zvýšena, z důvodu, kdy se webová aplikace dotazuje samotné administrace na mnoho informací při samotném sestavení. Aby mohly být požadavky akceptovány, i přes použití reverzní proxy Traefik, bylo nutné nastavit *trustProxy* na *true*. Došlo také ke konfiguraci, do jaké maximální hloubky mají být relace populovány. Výchozí hodnota je populace do 10 úrovně. Tato hodnota je ale velmi vysoká, a pro zlepšení výkonu samotného serveru byla hodnota snížena na populaci do 4 úrovně.

Kód 10: Pokročilé nastavení systému

```
cors: "*",
maxDepth: 4,
rateLimit: {
  trustProxy: true,
  max: 5000,
},
plugins: [
  seo({
    collections: ["pages", "posts"],
    generateTitle: ({ doc }: { doc: Document }) => {
      return `Energo Savings - ${doc.title.value}`;
    },
  }),
],
defaultDepth: 4,
```

Zdroj: vlastní zpracování

Systém obsahuje mnoho kolekcí, pro zjednodušení bude vysvětlena nejdůležitější kolekce *pages*. Tato kolekce obsahuje stránky, které se zobrazují na webové stránce. Každá stránka je veřejné přístupná, získat informace o celé stránce může proto jakýkoli klient, který si informace vyžádá. U této kolekce byly nadefinovány *hooks*.

Kód 11: Definované hooks v pages kolekci

```
hooks: {
  afterRead: [
    ({ doc }) => {
      return removeLayout(doc);
    },
  ],
  afterDelete: [
    ({ doc, req: { payload } }) => {
      deletePage({ doc, payload });
    },
  ],
  afterChange: [
    ({ req: { payload } }) => {
      regeneratePage({
        payload,
      });
    },
  ],
},
```

Zdroj: vlastní zpracování

Ty jsou pouštěny ve třech stavech. Prvním stavem je stav po čtení. V tomto stavu se volá funkce *removeLayout*, která projde celý objekt, a odebere z objektu veškeré instance objektu *layout*. K tomuto dochází z důvodu, kdy může objekt obsahovat relace na další stránky, kdy tyto stránky obsahují mnoho informací, z nichž většina je uložena právě v *layout* objektu. Tento problém by bylo možné vyřešit adopcí GraphQL. Při druhém stavu, který nastává v moment smazání záznamu z kolekce, dochází k volání funkce *deletePage*. Tato funkce má za úkol provolat samotnou webovou stránku, která obsahuje API funkce, které jsou součástí Next.js, a dynamicky revalidují dříve staticky vygenerovaný obsah. Stejný princip platí u stavu po změně webové stránky. Dochází k volání funkce *regeneratePage*, která, na základě změněného obsahu detekuje, u kterých stránek musí dojít k revalidaci. V případě úpravy hlavičky či patičky jsou revalidovány veškeré stránky. Pro samotné stránky byl zapnut systém *Drafts*. Ten umožňuje verzování změn samotné stránky, v případech nouze umožňuje také *rollback* na verze dřívější a zvyšuje uživatelskou přívětivost.

Stránka má definované pole pro název, *slug* a obsah. Název je obyčejný text, ze kterého je generován slug, který je zbaven diakritiky a mezery jsou nahrazeny pomlčkami. Samotný obsah se poté skládá z jednotlivých bloků. Stránka má na první úrovni čtyři bloky,

kdy nejdůležitějším blokem je kontejner. Ten se skládá z dalších bloků, které mohou být do kontejneru přidány.

Kód 12: Definovaný blok layout

```
{
  label: "Komponenta",
  labels: {
    singular: "Komponenta",
    plural: "Komponenty",
  },
  name: "layout",
  type: "blocks",
  blocks: [Container, Button, Hero, BottomOverflow],
},
```

Zdroj: vlastní zpracování

Tento způsob umožňuje velmi dynamickou úpravu obsahu webové stránky. U této kolekce byly také nasazeny další dva endpointy. První endpoint vrací *slug* hodnoty všech stránek jako pole řetězců. Lze ho provolat zavoláním */api/pages/slugs*. Je potřebný pro statickou generaci stránek při volání Next.js funkce *getStaticPaths*.

Kód 13: Definovaný slugs endpoint

```
{
  path: "/slugs",
  method: "get",
  handler: async (req, res, next) => {
    const { docs } = await req.payload.find({
      collection: "pages",
      pagination: false,
    });
    const slugs = docs.map((page) => page.slug);
    if (docs) {
      res.status(200).send(slugs);
    } else {
      res.status(404).send({ error: "not found" });
    }
  },
},
```

Zdroj: vlastní zpracování

Druhý endpoint vrací veškeré stránky, které jsou v administraci založeny. Je volán na adrese */api/pages/all*. Tento endpoint je využíván při generování mapy webu. Samotná generace mapy webu probíhá na straně Next.js.

5 Zhodnocení výsledků

V této diplomové práci byly analyzovány čtyři hlavní headless systémy pro správu obsahu. Tyto systémy byly zvoleny z důvodu jejich popularity na trhu. Na základě této popularity byla uzpůsobena analýza. Nejpopulárnější systémy byly analyzovány více než ty méně populární, aby došlo k odhalení případných nedostatků, které jsou z důvodu popularity u těchto systémů přehlíženy. Nejpopulárnějším systémem na trhu je Strapi. U tohoto systému došlo na podrobnou kontrolu kvality kódu, systému přihlašování, možností rozšíření a v neposlední řadě byla zkoumána bezpečnost samotného systému.

Strapi se jeví jako skvělá volba, pokud je nutné zvolit známý systém, který má k dispozici bohatou dokumentaci. Jeho hlavním nedostatkem je technologický dluh. Samotný technologický dluh se podepisuje na straně kvality kódu, kde je zvolen JavaScript, který neposkytuje dostatečnou typovou bezpečnost jako ostatní systémy. Ani bezpečnost není hlavní stránkou systému Strapi, kdy je mnoho bezpečnostních opatření zanedbáno, ať už se jedná o autorizaci a práci s citlivými tokeny, nebo o zabezpečení kolekcí. Přístup, kdy se kolekce definují na úrovni GUI a ukládají do databáze není vhodný v momentě, kdy celý systém nastavuje a udržuje vývojář. Vývojáře toto spíše odradí a vývoj zpomalí.

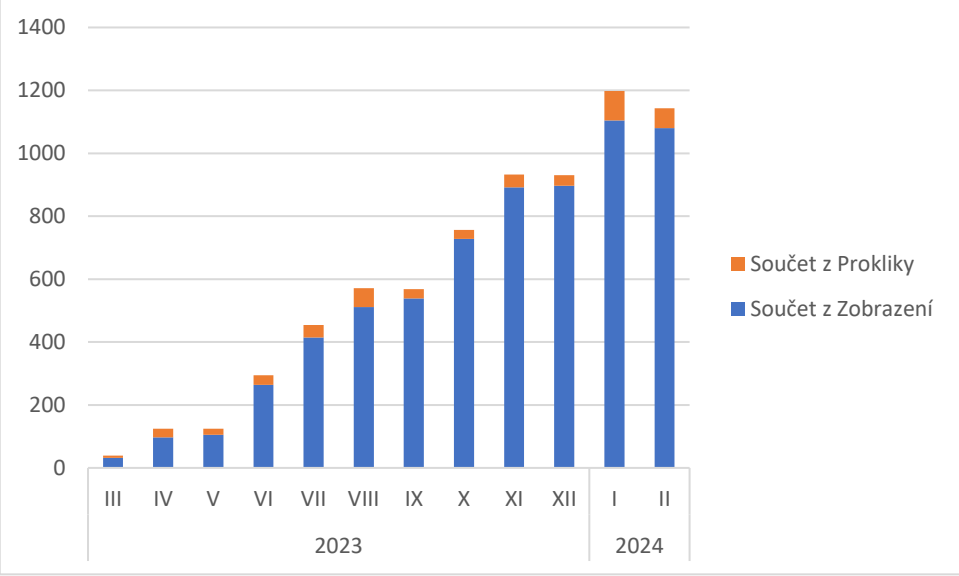
Druhým zkoumaným systémem byl Directus. Ten je na rozdíl od systému Strapi velmi technologicky pokročilým, zaručuje typovou bezpečnost a neobsahuje bezpečnostní rizika, která obsahuje systém Strapi. Využívá stejný přístup k definování kolekcí, tedy přes GUI, je ale mnohem uživatelský přívětivější. Pohled do dokumentace naznačuje, že jsou informace k systému velmi dobře strukturovány, vývoj je aktivnější, než jak tomu je u systému Strapi. Z výkonnostního hlediska má Directus také výhodu, jelikož v *benchmark* testu dokáže odbavit větší počet požadavků za časové okno.

Třetím zkoumaným systémem byl Sanity. Ten je velmi odlišný od předchozích dvou systémů a jeho kladem je rychlé a jednoduché nasazení klientské části na základě konfiguračního objektu. Tento konfigurační objekt dává strukturu datům, které jsou uloženy v datovém úložišti Content Lake, ke kterému uživatel nemá přístup. Systém je nejvíce uživatelsky přívětivý, ať už se systémem pracuje cílový uživatel nebo vývojář. Bohužel, kvůli pestré integraci s ekosystémem Sanity je u systému velké riziko *vendor lock-inu*. Data nejsou ve skutečnosti ve vlastnictví uživatele a na limity přísná *free* verze směřuje uživatele k zakoupení placeného plánu.

Poslední zkoumaný systém je Payload. Ten ostatním systémům konkuruje ve všech kritériích. U tohoto systému byla zjištěna nejlepší práce s autorizací, přihlašováním a konfigurací. Definice kolekcí v kódu je velmi příjemný přístup, a pro vývojáře se jedná o nejrychlejší způsob, jak headless systém pro správu obsahu nastavit. Na základě *bechmarku* bylo zjištěno, že je zároveň nejrychlejší, a dokáže odbavit nejvíce požadavků za určité časové okno. S největší pravděpodobností tomu tak je využitím nerelační databáze MongoDB. Ta v případě rychlého vývoje usnadňuje vývojáři práci s migracemi, které nejsou u této databáze potřeba.

V druhé části praktické práce došlo k vývoji a nasazení webové aplikace společně s administrací. Pro administrační část byl na základě kritérií zvolen systém Payload. Ten byl nasazen dle přání klienta. Výběr systému byl velmi dobrou volbou, jelikož zjednodušil a zrychlil samotný vývoj. Na základě funkcionalit tohoto systému byla tvorba samotné webové aplikace jednoduchá. Největším kladem bylo jednoduché sdílení typů administrace s webovou aplikací. Nasazení celého systému je robustní, bez zaznamenaných pádů, a s přívětivou cenou pro samotného klienta. Za posledních 12 měsíců byl zaregistrován růst zobrazení a prokliků.

Graf 3: Výkon vyhledávání za posledních 12 měsíců



Zdroj: vlastní zpracování

Pokud se porovnají výstupy diplomové práce a práce Ladislava Burgra, ukáže se, že se autor staví k *headless* systémům podobně. Práce autora analyzuje systémy Strapi, Ghost,

Directus a Payload CMS. Každý systém je ale zmíněn pouze obecně, k podrobné kritice nedochází. To není vhodné řešení, obecná analýza nemusí odhalit veškeré problémy, které systémy pro správu obsahu mohou mít. Pro vývoj byl autorem zvolen systém Directus. Systém Payload nebyl zvolen, z důvodu absence verzování obsahu, zálohování a relační databáze.

Všechny tyto funkcionality jsou v dnešní verzi systému Payload již obsaženy. To poukazuje na velmi agilní přístup systémů pro správu obsahu, které se na základě požadavků zákazníků mění. Nové přírůstky na trhu se systémy pro správu obsahu zlepšují stabilitu trhu, zvyšují konkurenci mezi systémy a snižují riziko monopolu.

6 Závěr

V této diplomové práci byla provedena komplexní analýza a srovnání headless systémů pro správu obsahu s cílem posoudit jejich vhodnost pro tvorbu a správu webové prezentace. Záměrem bylo nejen zhodnotit technické možnosti a funkčnost těchto systémů, ale také prozkoumat jejich přínosy a omezení v kontextu moderního webdesignu a online obsahu. Výzkum se opíral o důkladnou rešerši odborné literatury, analýzu existujících řešení a praktickou implementaci webové stránky v jednom z vybraných headless CMS, konkrétně v Payload CMS.

Hlavním výsledkem práce je potvrzení, že headless CMS představují významný posun ve správě webu a digitálního obsahu, nabízejí vysokou úroveň flexibility, škálovatelnosti a efektivity pro tvorbu webových aplikací. Na základě provedeného srovnání a analýzy bylo zjištěno, že headless systémy pro správu obsahu efektivně oddělují obsah od prezentace, což umožňuje vývojářům a designérům pracovat nezávisle a efektivněji, a nabízejí lepší integraci s moderními webovými technologiemi, jako jsou JavaScript frameworky a API služby.

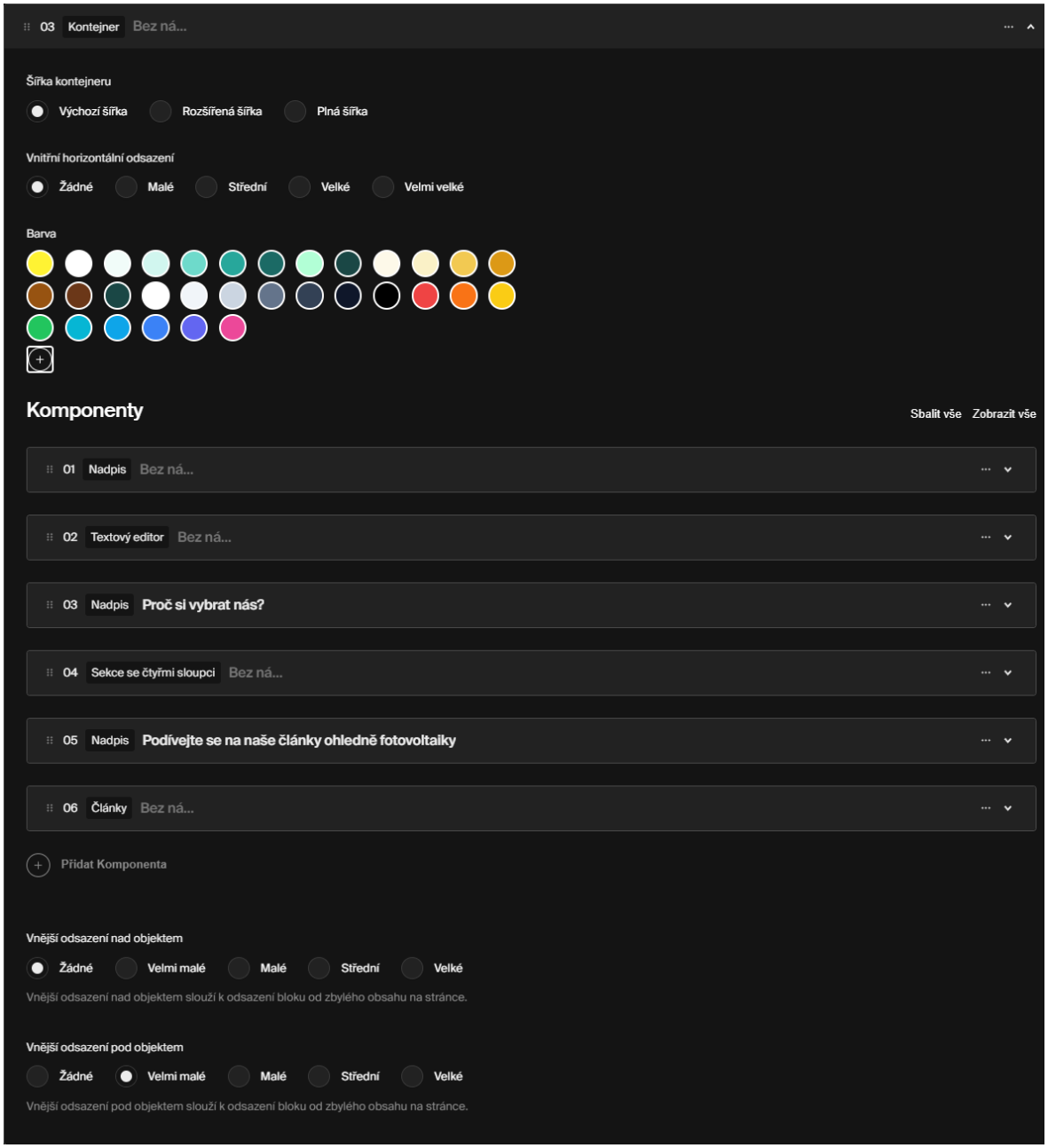
Implementace webové prezentace v Payload CMS ukázala, že volba headless CMS může značně usnadnit a zefektivnit vývojový proces, zjednodušit správu obsahu a zároveň poskytnout robustní základ pro vysoce výkonné webové aplikace. Byla zdůrazněna důležitost výběru vhodného systému na základě specifických potřeb projektu, včetně podpory pro moderní webové technologie, uživatelskou přívětivost a možnosti integrace s externími službami.

Tato práce také poukázala na významnou roli headless systémů pro správu obsahu v kontextu SEO a digitálního marketingu, kde jejich flexibilita a integrace s různými platformami a nástroji může přinést významné výhody pro zlepšení online viditelnosti a uživatelské zkušenosti. Diskuse o vedlejších cílech práce, včetně přehledu vybraných webových technologií a CMS, poskytla cenné informace pro další výzkum a vývoj v oblasti webdesignu a správy obsahu.

V závěru lze konstatovat, že headless systémy pro správu obsahu jsou klíčovou technologií pro budoucnost webového vývoje, poskytují významné přínosy pro tvorbu a správu digitálního obsahu a nabízejí odpověď na rostoucí požadavky na flexibilitu, rychlost a personalizaci v digitálním prostředí. Diplomová práce přispěla k hlubšímu

pochopení možností a výzev spojených s využíváním headless CMS a otevřela cestu pro další průzkum v této dynamicky se vyvíjející oblasti.

Obrázek 23: Nasazený systém v pohledu jednotlivých komponent



Zdroj: vlastní zpracování

Obrázek 24: Nasazená webová aplikace

Zdroj: vlastní zpracování

7 Seznam použitých zdrojů

AMAZON, 2024. Tutorial: Launch and configure a WordPress instance in Lightsail. In: AMAZON. *AWS* [online]. [cit. 2024-02-24]. Dostupné z: https://docs.aws.amazon.com/lightsail/latest/userguide/amazon-lightsail-tutorial-launching-and-configuring-wordpress.html

Architecture, 2023. In: *Directus* [online]. [cit. 2024-02-24]. Dostupné z: https://docs.directus.io/getting-started/architecture.html

BARKER, Deane, 2016. *Web Content Management* [online]. 1. Sebastopol (California): O'Reilly Media, Inc. [cit. 2023-09-13]. ISBN 978-1491908129. Dostupné z: https://learning.oreilly.com/library/view/web-content-management/9781491908112/

BAUMGARTNER, Stefan, 2023. *Decoupled Applications and Composable Web Architectures* [online]. 1. O'Reilly Media [cit. 2024-02-24]. ISBN 9781098151478. Dostupné z: https://www.oreilly.com/library/view/decoupled-applications-and/9781098151478/

BYTEBASE, 2024. *GitHub Star History* [online]. [cit. 2024-03-23]. Dostupné z: https://star-history.com/

CLOUDFLARE, 2024. *Cloudflare* [online]. [cit. 2024-02-24]. Dostupné z: https://www.cloudflare.com/

Collections, 2023. In: *Directus* [online]. [cit. 2024-02-24]. Dostupné z: https://docs.directus.io/app/data-model/collections.html

Directus [online], 2023. [cit. 2024-03-22]. Dostupné z: https://directus.io/

ELSHAFIE, Khalid a Mozafar HAIDER, 2022. *Designing Web APIs with Strapi* [online]. 1. Birmingham: Packt [cit. 2024-02-03]. ISBN 9781800560635.

ERZ, Hendrik, 2021. Why You Shouldn't Use SQLite. In: *Hendrik Erz* [online]. [cit. 2024-02-31]. Dostupné z: https://www.hendrik-erz.de/post/why-you-shouldnt-use-sqlite

Flows, 2023. In: *Directus* [online]. [cit. 2024-02-24]. Dostupné z: https://docs.directus.io/app/flows.html

FREEMAN, Eric a Elisabeth ROBSON, 2020. *Head First Design Patterns* [online]. 2. O'Reilly Media [cit. 2024-02-24]. ISBN 9781492077992. Dostupné z: https://learning.oreilly.com/library/view/head-first-design/9781492077992/

GOOGLE, 2023. WordPress on Google Cloud. In: *Google Cloud* [online]. [cit. 2024-02-31]. Dostupné z: https://cloud.google.com/wordpress/

HEJZLAR, Jan, 2023. *Headless CMS s generováním admin panelu*. Praha. Bakalářská práce. České vysoké učení technické v Praze.

HETZNER, 2023. Datacenter. In: HETZNER. *Hetzner* [online]. [cit. 2024-03-22]. Dostupné z: https://www.hetzner.com/unternehmen/rechenzentrum/

Hooks Overview, 2023. In: *Payload CMS* [online]. [cit. 2023-09-11]. Dostupné z: https://payloadcms.com/docs/hooks/overview

Installation, 2023. In: *Payload CMS* [online]. [cit. 2023-09-13]. Dostupné z: https://payloadcms.com/docs/getting-started/installation

Introduction, 2023. In: *Directus* [online]. [cit. 2024-02-24]. Dostupné z: https://docs.directus.io/getting-started/introduction.html

Introduction, 2024. In: *Vue.js* [online]. [cit. 2024-02-23]. Dostupné z: https://vuejs.org/guide/introduction.html

JETBRAINS, 2024. Adapter pattern. In: *Hyperskill* [online]. [cit. 2024-02-27]. Dostupné z: https://hyperskill.org/learn/step/17657

JustRelate [online], 2020. [cit. 2024-02-15]. Dostupné z: https://www.justrelate.com/cms-vs-wcm-wcms-vs-ecm-328222e4710738e3

LIZARDO, Pablo, 2023. Next.js Diagrams. In: *Figma* [online]. [cit. 2024-02-25]. Dostupné z: https://www.figma.com/community/file/1181669867394898197

MATTHEW, Abel, 2021. Why choose MongoDB as your next database?. In: CRIO. *Crio* [online]. [cit. 2024-02-23]. Dostupné z: https://www.crio.do/blog/why-choose-mongodb-as-your-next-database/

MICROSOFT, 2023. Vytvoření webu WordPress. In: *Microsoft Learn* [online]. [cit. 2024-02-30]. Dostupné z: https://learn.microsoft.com/cs-cz/azure/app-service/quickstart-wordpress#create-wordpress-site-using-azure-portal

MIKRUT, James, 2023. Announcing Payload 2.0: Postgres, Live Preview, Lexical RTE, and More. In: *Payload CMS* [online]. [cit. 2024-02-24]. Dostupné z: https://payloadcms.com/blog/payload-2-0

MONGODB INC., 2023. Sharding. In: *MongoDB* [online]. [cit. 2023-09-13]. Dostupné z: https://www.mongodb.com/docs/manual/sharding/

MONGODB INC., 2023. Indexes. In: *MongoDB* [online]. [cit. 2023-09-13]. Dostupné z: https://www.mongodb.com/docs/manual/indexes/

OKTA, 2023. Introduction to JSON Web Tokens. In: OKTA. *JWT* [online]. [cit. 2024-02-31]. Dostupné z: https://jwt.io/introduction

Operations, 2023. In: *Directus* [online]. [cit. 2024-02-24]. Dostupné z: https://docs.directus.io/app/flows/operations.html

OSMANI, Addy, 2023. *Learning JavaScript Design Patterns* [online]. 3. O'Reilly Media [cit. 2024-02-25]. ISBN 9781098139865. Dostupné z: https://learning.oreilly.com/library/view/learning-javascript-design/9781098139865/

PALAS, Petr, 2022. *The Ultimate Guide to Headless CMS: Everything you need to know to choose the right CMS*. 3rd ed. Brno: Kontent.ai. ISBN ISBN 978-1973352686.

Partial Hydration, 2022. In: *Gatsby* [online]. [cit. 2024-02-23]. Dostupné z: https://www.gatsbyjs.com/docs/conceptual/partial-hydration/

Payload CMS [online], 2023. [cit. 2024-02-23]. Dostupné z: https://payloadcms.com/

PAYLOAD CMS, 2024. Collections. In: *Payload CMS* [online]. [cit. 2024-02-31]. Dostupné z: https://payloadcms.com/docs/configuration/collections

PAYLOAD CMS LLC, 2023. The Payload Config. In: *Payload CMS* [online]. [cit. 2023-09-13]. Dostupné z: https://payloadcms.com/docs/configuration/overview

PAYLOAD CMS LLC, 2023. Field Hooks. In: *Payload CMS* [online]. [cit. 2023-09-13]. Dostupné z: https://payloadcms.com/docs/hooks/fields

PAYLOAD CMS LLC, 2023. Collection Hooks. In: *Payload CMS* [online]. [cit. 2023-09-13]. Dostupné z: https://payloadcms.com/docs/hooks/collections

PECORARO, Christopher a Vincenzo GAMBINO, 2021. *Jumpstart Jamstack Development* [online]. 1. Packt Publishing [cit. 2024-02-24]. Dostupné z: https://learning.oreilly.com/library/view/jumpstart-jamstack-development/9781800203495/

Putting Sanity.io to the test, 2020. In: MELVæR, Knut. *Knut Melvær* [online]. [cit. 2024-01-24]. Dostupné z: https://www.knutmelvaer.no/blog/2020/04/a-practical-application-of-the-web-project-book/

Sanity Content Lake, 2023. In: *Sanity* [online]. [cit. 2024-03-22]. Dostupné z: https://www.sanity.io/content-lake

Sanity Studio, 2023. In: *Sanity* [online]. [cit. 2024-03-22]. Dostupné z: https://www.sanity.io/studio

Servers, 2023. In: *Hetzner* [online]. [cit. 2024-03-22]. Dostupné z: https://docs.hetzner.com/cloud/servers/overview/

Strapi [online], 2023. [cit. 2024-02-23]. Dostupné z: https://strapi.io/

TEJAS, Kumar, 2023. *Fluent React* [online]. 1. O'Reilly Media [cit. 2024-02-26]. ISBN 9781098138707. Dostupné z: https://learning.oreilly.com/library/view/fluent-react/9781098138707/

Triggers, 2023. In: *Directus* [online]. [cit. 2024-02-24]. Dostupné z: https://docs.directus.io/app/flows/triggers.html

8 Seznam obrázků, tabulek, grafů a zkratek

8.1 Seznam obrázků

8.2 Seznam tabulek

8.3 Seznam grafů

8.4 Seznam ukázek kódu

8.5 Seznam použitých zkratek

API	Application Programming Interface
REST	REpresentational State Transfer
GQL	Graph Query Language
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
PHP	Hypertext Preprocessor

SQL	Structured Query Language
AWS	Amazon Web Services
JSON	JavaScript Object Notation
DOM	Document Object Model
JSX	JavaScript XML
XML	Extensible Markup Language
CLI	Command Line Interface
CRUD	Create Read Update Delete
HTTP	Hypertext Transfer Protocol
GUI	Graphical User Interface
JWT	JSON Web Token
OIDC	OpenID Connect
SSO	Single Sign On
SDK	Software Development Kit
GROQ	Graph Relational Object Queries
NPM	Node Package Manager

Přílohy

Příloha A CD se zdrojovým kódem