# Webová aplikace s dynamickým obsahem

Webové aplikace s dynamickým obsahem jsou nejčastěji vytvořeny ze dvou částí - backendu a frontendu.

## Backend

Jako **backend** („zadní konec“) se označuje část webové aplikace, která slouží k administraci webu a k zpracování dat. V případě redakčního systému (CMS) backend umožňuje vkládání a úpravy článků, zakládání účtů dalších administrátorů či třeba manipulace se strukturou celého webu. V internetovém obchodu zase backend slouží k vkládání nového zboží i k aktualizaci jeho vlastností, zejména cen.

*Backend funguje na straně webového serveru a bývá vyvíjen pomocí různých jazyků (PHP, JavaScript, Python, Java, C#, Ruby), často i s podporou tzv.* frameworků *("aplikačních rámců").*

### Softwarový framework

**Softwarový framework** je platforma pro vývoj softwarových aplikací. Poskytuje základ, na kterém mohou vývojáři softwaru podstatně rychleji a efektivněji vytvářet programy pro konkrétní platformu. Framework může také zahrnovat knihovny kódů, kompilátor a další programy používané v procesu vývoje softwaru.

*K nejoblíbenějším patří frameworky Symfony, Laravel či český Nette Framework pro jazyk PHP, Express pro JavaScript, Django a Flask pro Python nebo Ruby on Rails pro programátory v jazyce Ruby.*

### Databáze

Backendové aplikace obvykle pracují s daty, které jsou ukládány v **databázích**. Existuje celá řada typů databází, ale stále jsou nejčastěji využívány tzv. relační databáze založené na sadě vzájemně propojených datových tabulek.

Pro komunikaci s **relačními databázemi** se využívá **jazyk SQL** (Structured Query Language - "jazyk strukturovaných dotazů"). K nejpopulárnějším relačním databázím, které běží v pozadí webových aplikací, patří MySQL, MariaDB nebo PostgreSQL. Pro účely výuky a podporu vývoje menších webů lze použít i odlehčenou variantu relační databáze SQLite.

Alternativou k relačním databázím jsou tzv. **NoSQL databáze**, které pro uložení a zpracování dat používají jiné prostředky než tabulková schémata. Příkladem takové databáze je MongoDB, která ukládá data v podobě objektů **JSON** (JavaScript Object Notation) nebo BSON (Binary JSON), a je proto hojně využívána vývojáři v jazyce JS a v prostředí NodeJS.

## Frontend

Pojem frontend pochází z oblasti programování webových aplikací, kde slouží k označení části webu viditelné běžným návštěvníkům. Například u internetového obchodu slouží jako frontend katalog zboží, nákupní košík a objednávkový formulář. Na rozdíl od backendu mívá většinou mnohem propracovanější uživatelské rozhraní, zejména z hlediska přístupnosti, použitelnosti a vzhledu. Mnohem častěji také dochází k jeho změnám, aby byl v souladu s aktuálními trendy.

K tvorbě frontendu využíváme technologie podporované na straně klienta - v tomto případě webového prohlížeče (browseru). Kromě těch základních, čili jazyka HTML a kaskádových stylů (CSS), se nejvíce uplatňují technologie založené na jazyce **JavaScript**.

### Knihovny

Velmi často je k webovým stránkám připojována knihovna **jQuery**, která usnadňuje manipulaci s objekty i tvorbu animací na webových stránkách nebo zjednodušuje použití technologie **AJAX**. Kód knihovny jQuery je velmi úsporný, v minimalistické verzi obsahuje pouze několik desítek kilobajtů. Základní motto „write less, do more“ („piš méně, udělej více“) vystihuje efektivní zápis skriptů v jQuery. Knihovna JQuery může být stažena nebo na stránku připojena prostřednictvím odkazů na **CDN** (Content Delivery Network - síť pro doručování obsahu). Klíčovou roli hrají v jQuery selektory dobře známé z CSS, pomocí kterých lze spolehlivě identifikovat libovolný objekt tvořící DOM webové stránky. Pro odlišení jQuery selektorů od běžného javascriptového kódu se používá dolarová notace - $(selektor). jQuery nabízí programátorům mnoho metod, pomocí nichž lze provést různé efektní změny na webové stránce (např. skrývání a odkrývání částí stránky), vytvářet jednoduché animace, měnit dynamicky kaskádové styly apod.

Existuje velké množství **knihoven**, které frontendovým vývojářům výrazně usnadňují řešení různých konkrétních typů úloh. Příkladem populárních javascriptových knihoven je DataTables, která s vynaložením minimálního programátorského úsilí dokáže zobrazovat interaktivní datové tabulky, Chart.js pro zobrazování grafů, knihovna D3.js nabízející vizualizaci dat na webové stránce, Anime.js umožňující tvorbu animovaných prvků, knihovny řešící tvorbu fotogalerií (Lightbox, LightGallery) nebo integraci WYSIWYG editorů do webových formulářů (TinyMCE).

V současné době patří k nejoblíbenějším open-source knihovnám pro tvorbu uživatelského rozhraní (UI) webových aplikací knihovna **ReactJS**, kterou vyvíjí společnost Meta (dříve Facebook). Základním stavebním kamenem jsou tzv. komponenty (components), což jsou různé znovupoužitelné HTML elementy se zapouzdřenou funkcionalitou, jejichž skládáním vzniká komplexní aplikace. Tyto komponenty pak mají své vlastnosti (props) a spravují svůj vnitřní stav (state). Místo klasického imperativního přístupu (zapisování kódu jako sledu příkazů) se v Reactu používá spíše deklarativní způsob práce s daty, což vede k předvídatelnějšímu chování aplikace a usnadňuje to i její ladění. Navíc mohou vznikat komplexní i jednoúčelové moduly komponent, které lze začlenit do vlastní aplikace.

Příkladem je knihovna **React Bootstrap**, která dává vývojářům v Reactu možnost využít komponenty založené na oblíbeném CSS frameworku Bootstrap. Jinou populární variantou je knihovna komponent MUI (Material UI), s jejíž moderním a účelným designem se můžeme setkat na stránkách celosvětově významných společností, jako jsou Spotify, Amazon, Netflix nebo NASA.

## Redakční systémy

Redakční systém je složitá webová aplikace, která umožňuje správu obsahu webu. Obvykle běží na serveru a po přihlášení poskytuje rozhraní, přes které si uživatelé mohou spravovat obsah stránek, i když vůbec nerozumí webovým technologiím. Existují také redakční systémy, které fungují jako program spouštěný v počítači. Pro redakční systémy se používá také název publikační systém či zkratka **CMS**, z anglického Content Management System, systém pro správu obsahu.

K základním funkcím redakčních systémů patří:

* správa obsahu jednotlivých typů stránek, včetně galerií, článků, kalendářů;
* správa obrázků, multimédií, dokumentů či třeba souborů ke stažení;
* plnohodnotná správa struktury webu, včetně automatického generování nejrůznějších navigačních prvků;
* správa komentářů, diskuzí, anket a dalších prvků pro získávání zpětné vazby;
* funkce pro správu a vytváření formulářů;
* systém práv uživatelů – jeden uživatel například obsah vytváří a jiný ho schvaluje a publikuje;
* verzování obsahu – možnost vracet se zpět k předchozím podobám téhož dokumentu, tak jak se během úprav postupně měnil;
* logování aktivity uživatelů (takže lze snadno dohledat, kdo způsobil jakou změnu).

Existuje mnoho různých redakčních systémů, které můžeme členit podle jejich funkce (např. systémy pro správu webu, systémy pro správu dokumentů, blogovací systémy, wiki, e-shopová řešení) a vybírat si z komerčních i open source variant. K nejrozšířenějším patří CMS systémy vytvořené v jazyce PHP, jako jsou Wordpress, Joomla nebo Drupal, pro které existuje mnoho různých pluginů, témat (šablon s připraveným vzhledem) i jazykových mutací. V těchto tradičních CMS frontendové a backendové části webu obvykle tvoří jednu velkou aplikaci; prostřednictvím rozsáhlé a někdy i poměrně složité administrace se tak řeší nejen správa uživatelů a dat, ale konfiguruje se i vzhled stránek, jak je uvidí návštěvníci webu.

### Headless CMS

Naproti tomu tzv. **headless** ("bezhlavé") **CMS** jsou výrazně jednodušší, protože frontendovou část vůbec neobsahují (neboli nemají "hlavu"). Hotový obsah distribuují dalším systémům, jako jsou různé nezávislé weby, microsite, mobilní aplikace, sociální sítě, mailer a další marketingové kanály. Spravovaný obsah je obvykle nezávislý na zobrazení a bývá vytvářen s ohledem na znovupoužitelnost napříč různými systémy a platformami. Výhodou těchto systémů je kromě už zmíněné znovupoužitelnosti obsahu také jeho centralizace, lepší škálovatelnost i bezpečnost a především možnost volby různých řešení frontendové části, která se pro daný projekt nejvíce hodí. Usnadňuje to také případnou rychlou změnu frontendu bez ohledu na administrační část.

Headless CMS ukládají data buď v databázích a zprostředkovávají přístup k nim prostřednictvím API (např. komerční Contentful nebo nekomerční Strapi), nebo jako úložiště používají Git (Netlify CMS, Jekyll Admin).

## Webová API

**API** (zkratka pro Application Programming Interface) označuje v informatice rozhraní pro programování aplikací. Jde o komplex procedur, funkcí, tříd či protokolů nějaké knihovny (ale třeba i jiného programu nebo jádra operačního systému), které může programátor využívat. API určuje, jakým způsobem jsou funkce knihovny volány ze zdrojového kódu programu.

V programátorské praxi se můžeme setkat s celou řadou API - například s API operačních systémů (unixové systémy používají standard POSIX, OS od Microsoftu zase Windows API), grafickými API (OpenGL, DirectX, Vulkan) nebo s API databázových a informačních systémů.

API se nyní hojně využívá při vývoji mobilních i webových aplikací a tvorbě internetových stránek. Smyslem webových API je zajištění komunikace mezi dvěma platformami, které si vzájemně vyměňují data. Umožňují využívat již naprogramovaná řešení a integrovat je do vlastních webů.

V praxi se s integrací aplikačního rozhraní setkáváme třeba na stránkách zobrazujících polohu obchodu na Google mapách. V takovém případě je využito API z Google Map, které volají vrstvu kódu, jež zobrazuje požadovaná data.

API jsou využívána také čtečkami zpráv (RSS), kde umožňují výpis článků, nebo při přihlášení na webu a v mobilní aplikaci pomocí externího účtu (Facebook, LinkedIn atd.). Stejně tak umožňují přidávání komentářů například na sociální sítě. API může být součástí účetního software a díky tomu lze automaticky vygenerovat fakturu při nákupu z e-shopu a zapsat ji do databáze v účetnictví.

### SOAP

K implementaci API lze přistupovat několika způsoby. Nejstarším prostředkem zůstává takzvaný Simple Object Access Protocol, zkráceně **SOAP**, určený pro volání procedur za využití XML. Jeho cílem je snazší sdílení informací různých aplikací, běžících v různých prostředích nebo v různých jazycích. Jedná se tedy o nástroj odstraňující bariéry mezi odlišnými platformami. Dnes je využíván převážně bankami a pojišťovnami; v ČR tento typ API využívá systém EET.

### REST API

V širším měřítku je ovšem vytlačován **REST API** (někdy RESTfull API). REST API je, na rozdíl od SOAP, architektura fungující na principu klient-server, kdy veškerá uživatelská data zůstávají uložena na straně klienta. Díky tomu by nemělo dojít k úniku citlivých dat. Jedná se o pravděpodobně nejpoužívanější řešení určené k tvorbě aplikačních rozhraní. Ke své funkci využívá HTTP protokol.

Následníkem REST API je **GraphQL**, původně vyvíjené Facebookem (nyní Meta), který ho také používá v rámci své sociální sítě. Časem se z něj stal open-source a může tak být bezplatně využíván širokou veřejností. GraphQL je jazyk dotazů, který běží na straně serveru. Umožňuje stahovat data z více zdrojů najednou, což znatelně ulehčuje práci programátorů. Nabízí větší rychlost zpracování než REST API, má bohatší škálu využití, není závislý na architektuře a lze ho implementovat i nad stávající REST API.

# Tvorba backendu

## Strapi

Strapi je open source headless CMS napsaný v JavaScriptu, který umožňuje uživatelům snadno vytvářet i editovat datové objekty uložené v některé z populárních relačních databází (MySQL, MariaDB, PostgreSQl, SQLite) prostřednictvím uživatelsky přívětivého rozhraní.

Kromě toho nabízí:

* Zpřístupnění veškerá uložená data prostřednictvím REST API a GraphQL bez nutnosti psát jakýkoliv kód na straně serveru.
* Používání knihovny médií (Media library) k ukládání obrázků, zvuků, videí a dalších dokumentů na jednom místě.
* Návrh a správu obsahu webu pomocí samostatných datových komponent, které mohou vycházet jak z předdefinovaných datových typů, tak i z vlastních typů definovaných vývojáři. Komponenty tvoří základ jak tzv. collection types (datové struktury určené pro opakování stejného typu obsahu, jako jsou blogové příspěvky, produkty, uživatelé), tak pro tzv. single types (určené pro jednorázové stránky, které mají jedinečnou strukturu obsahu: domovská stránka, menu, SEO nastavení). Jednotlivé typy obsahu lze vzájemně propojovat pomocí relací; můžeme takto například propojit články s autory nebo kategoriemi. Strapi dokonce umožňuje definovat tzv. dynamické zóny (Dynamic zones), díky kterým lze provádět změny ve struktuře obsahu i za běhu aplikace.
* Správu uživatelských rolí a oprávnění včetně definování práv ke čtení, úpravám a publikování přímo v administrátorském rozhraní Strapi.
* Internacionalizaci obsahu a vytváření webových stránek v různých jazykových mutacích.
* Instalaci dalších pluginů a funkcionalit.

### Instalace Strapi

Nejprve pro náš projekt vytvoříme základní složku symbolicky pojmenovanou infoserver. Tu můžeme otevřít v příkazovém řádku nebo terminálu VSCode.

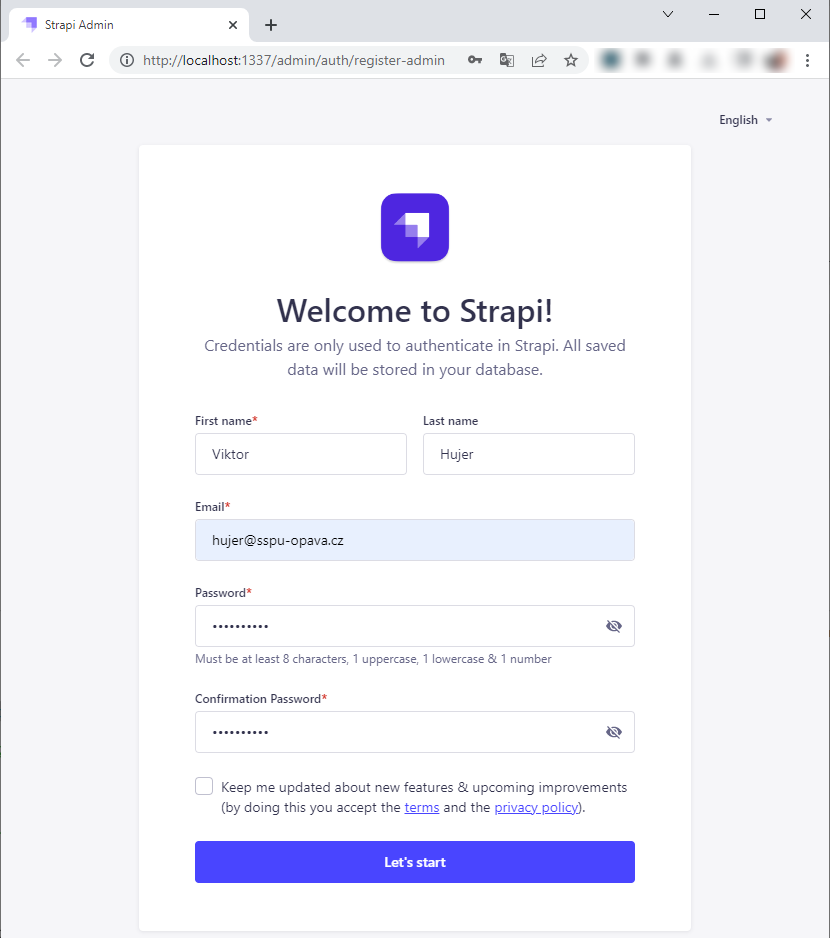
Zadáme příkaz:

**npx create-strapi-app@latest backend**

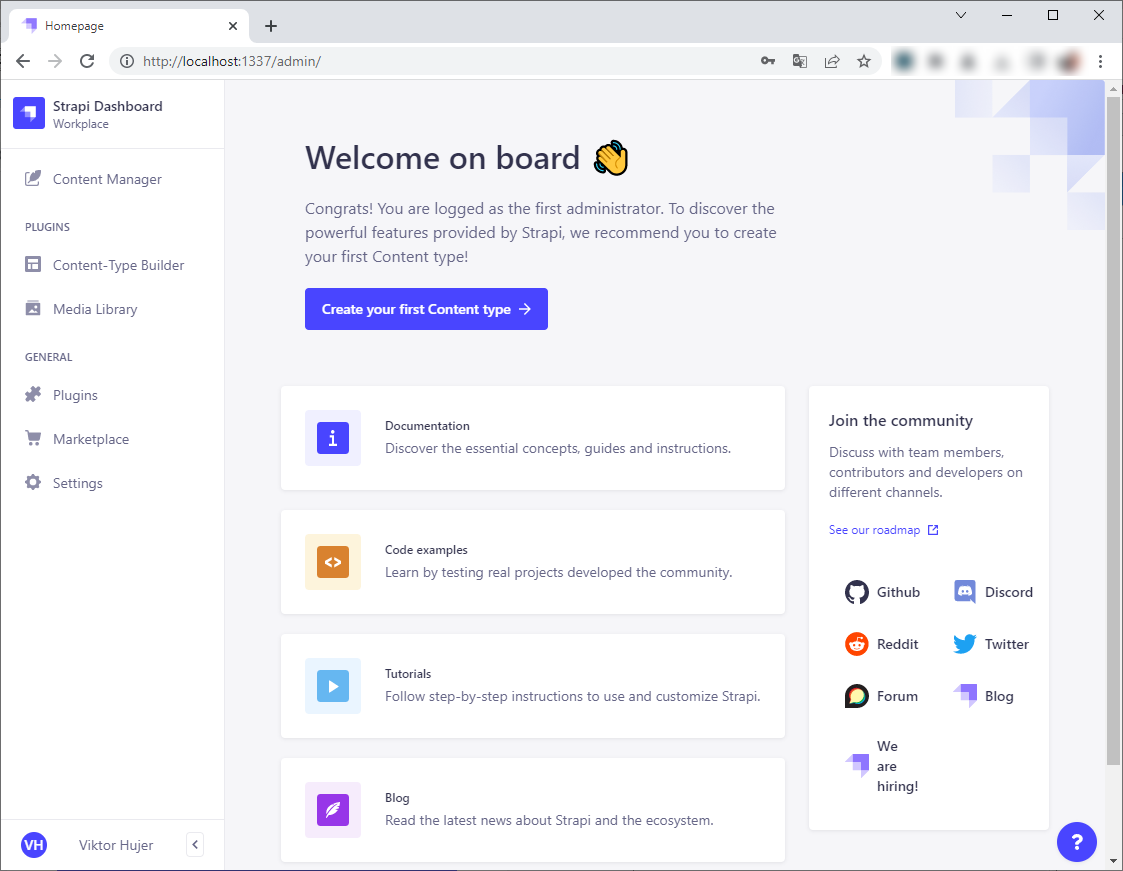
Vytvoříme jím serverovou aplikaci Strapi umístěnou do složky backend.

Na začátku instalace jsme dotázáni na typ projektu - můžeme potvrdit doporučený Quickstart.

Po úspěšné instalaci všech modulů se v prohlížeči spustí Strapi server na lokálním portu 1337 a objeví se formulář pro registraci administrátora - superuživatele:



Poté se objeví **Strapi Dashboard** - administrační rozhraní našeho serveru:



Uživatelský profil

Odhlašování

Nastavení

Uživatelé a role

Instalace pluginů

Nainstalované pluginy

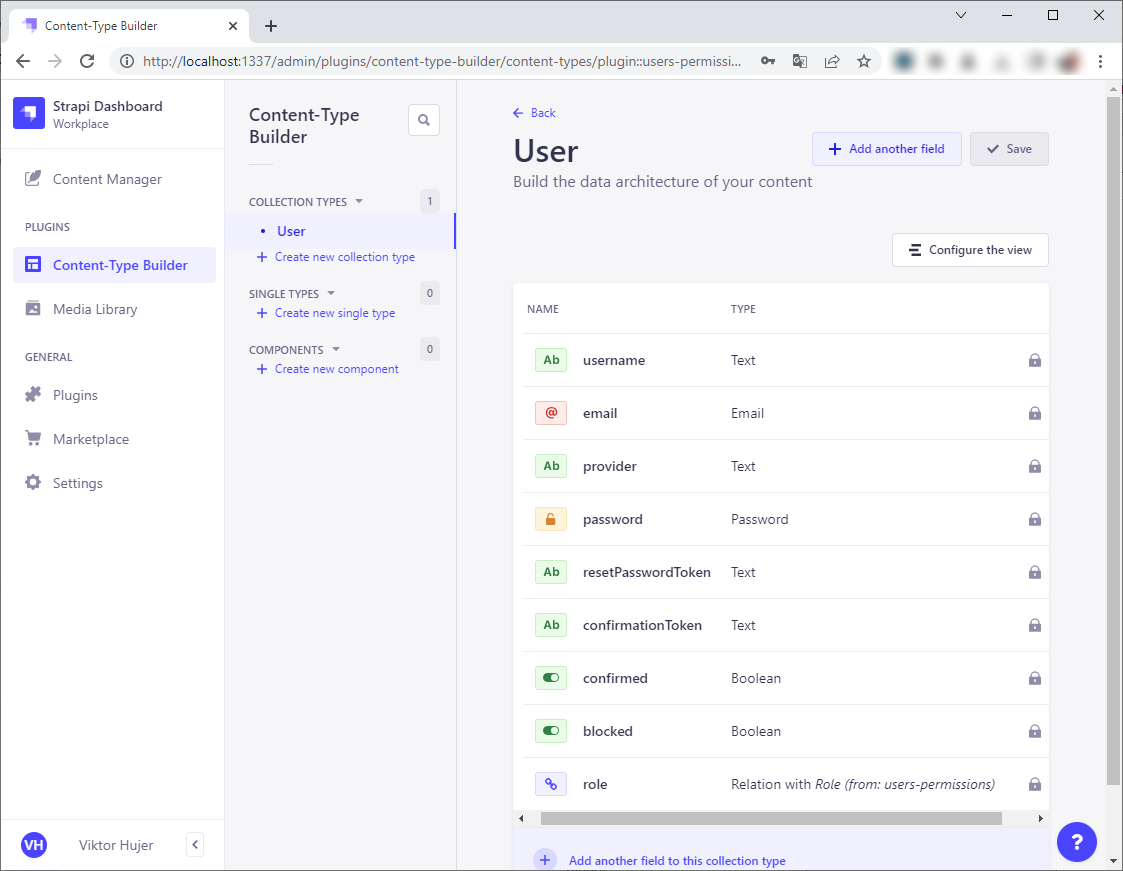
Knihovna médií

Návrhář obsahu webu

Správce obsahu webu

### Návrh obsahu

Po kliknutí na odkaz **Content-Type Builder** se zobrazí návrhář obsahu, v němž již existuje jedna předdefinovaná kolekce User. Slouží k vytváření účtů uživatelů, kteří mohou pracovat s obsahem webu jako **autentifikovaní** (přihlášení) uživatelé.

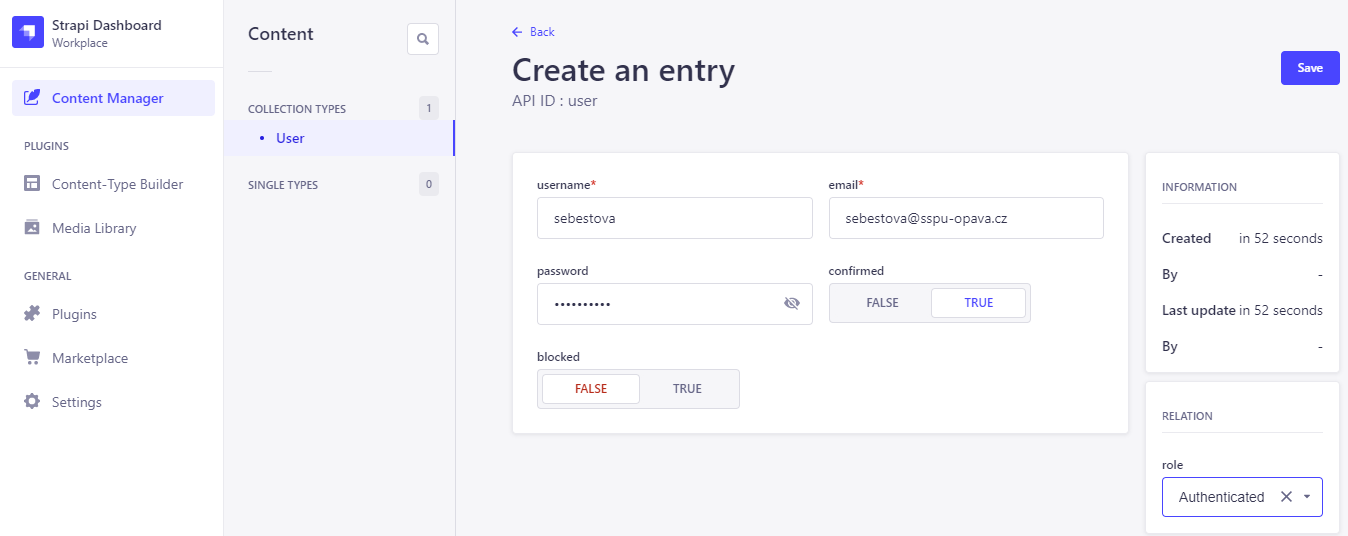


Pole **role** je pomocí relace propojeno s kolekcí **Role**

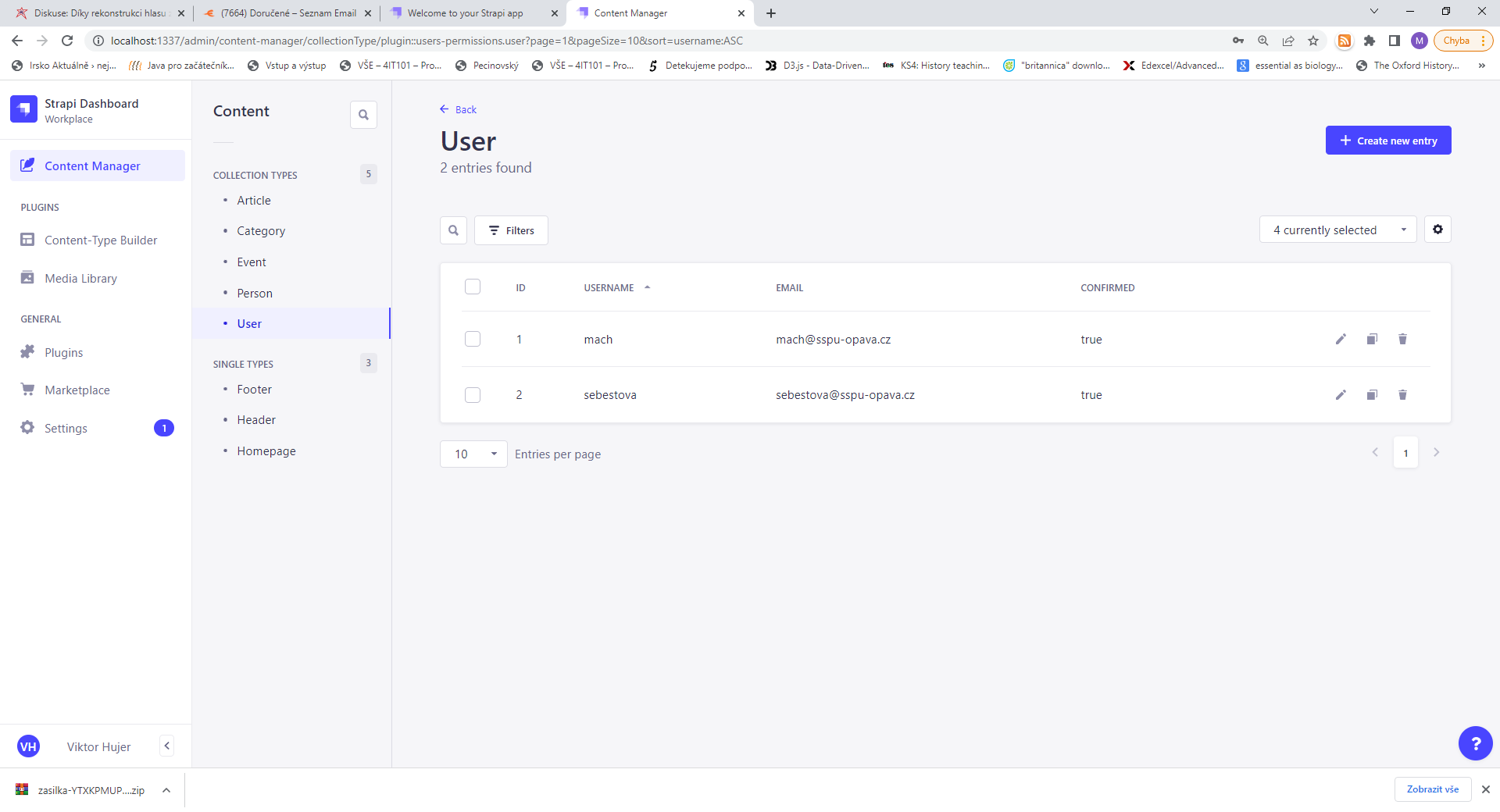
Může nám zároveň posloužit jako příklad pro návrh dalších kolekcí. Vidíme, že každá kolekce je tvořena řadou polí určitého typu a může být prostřednictvím typu **Relation** (relace) propojena s jinou kolekcí.

V této kolekci tomu tak je v případě pole **role**, které je navázáno na kolekci **Role**, jež je součástí předinstalovaného přídavného modulu *users-permissions*. Ten řeší problematiku tzv. **autorizace**, tedy přidělování oprávnění určitým skupinám uživatelů k provádění operací s určitými datovými objekty.

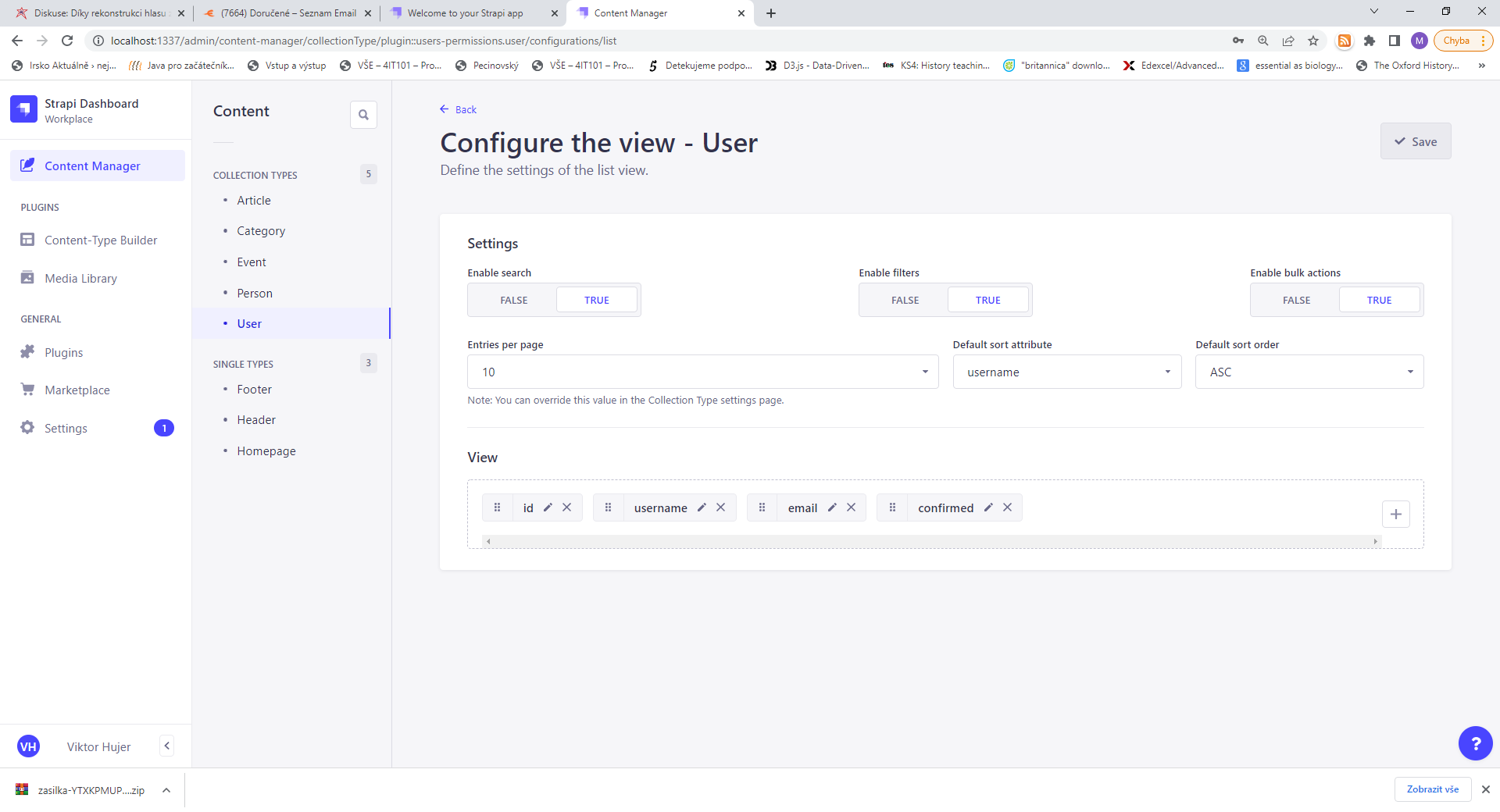
Nyní klikneme na odkaz **Content Manager** (*Správce obsahu*) a vytvoříme účet novému uživateli:



Prostřednictvím odkazu **Content Manager** (*Správce obsahu*) můžeme postupně vytvořit účty dvěma symbolickým uživatelům - Šebestové a Machovi. Oběma uživatelům přidělíme roli autentifikovaného uživatele.



Každý vytvořený datový objekt můžeme **editovat, duplikovat i smazat**. Po kliknutí na symbol **Nastavení** se zobrazí konfigurační stránka, na níž můžeme zapínat a vypínat možnosti vyhledávání či filtrování, volit počet záznamů zobrazených na jedné stránce (Entries per page) i výchozí způsob jejich seřazení. V bloku View můžeme měnit uspořádání datových polí v editačním formuláři.



### Vytváření vlastních kolekcí

Nyní můžeme v **Návrháři obsahu** (Content-Type Builder) vytvořit první vlastní kolekci nazvanou **Category**:

|  |  |
| --- | --- |
|  |  |

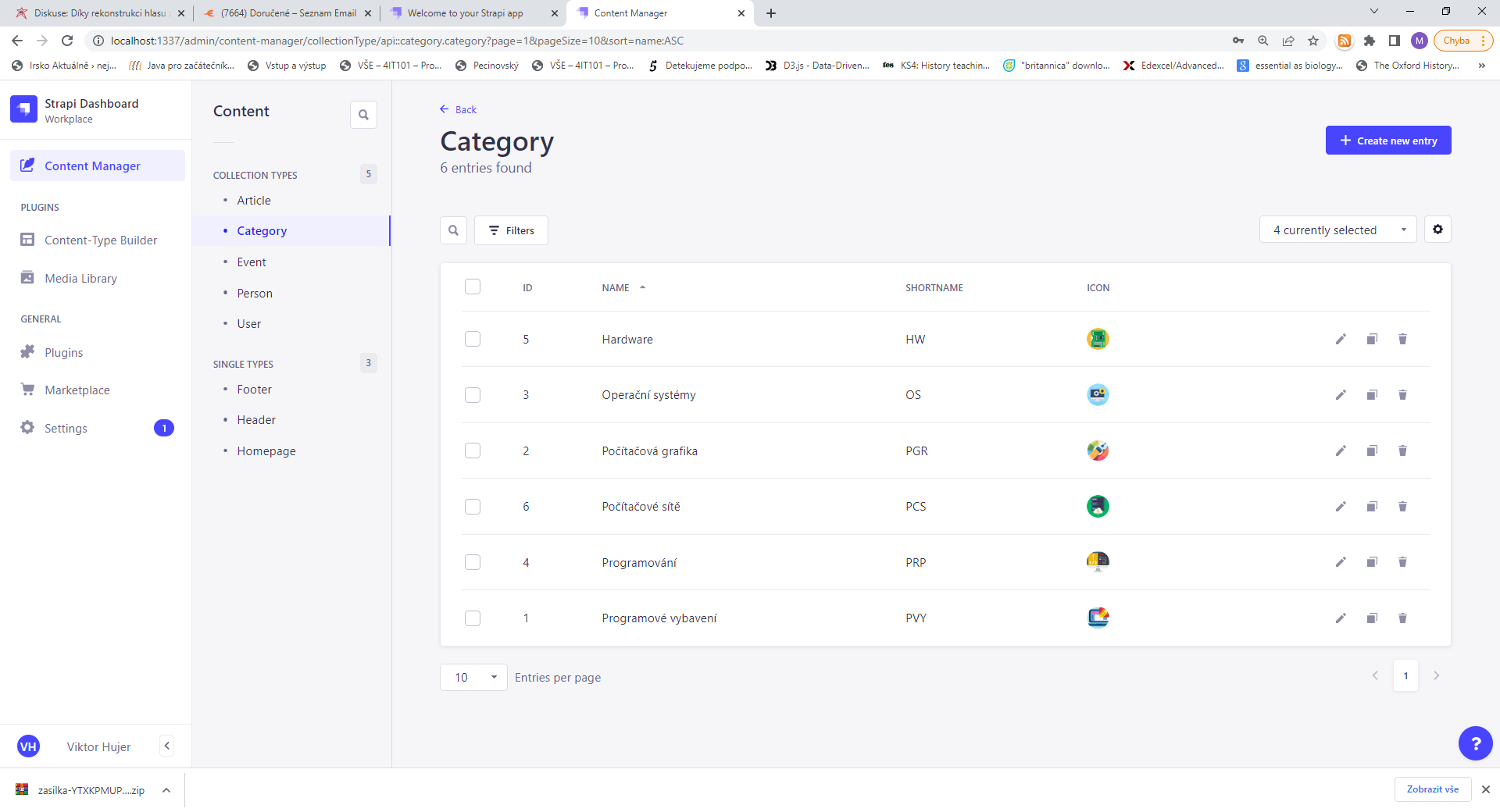
Název kolekce je vhodné zadat velkým počátečním písmenem a použít jednotné číslo. Návrhář obvykle sám připraví vhodné API ID pro označení jednoho i více datových objektů. Na kartě Advanced settings je volba Draft/Publish. V případě, že je zapnuta (true), prochází zpracování zadaných dat fází návrhu (draft) a teprve po potvrzení dojde k jejich uveřejnění (publish).

Pomocí komponent datových typů vytvoříme požadovanou strukturu kolekce. Její součástí jsou pole name a shortname (obě typu Text) a icon, kterému přiřadíme typ Media a v pokročilém nastavení vybereme pouze typ obrázků.

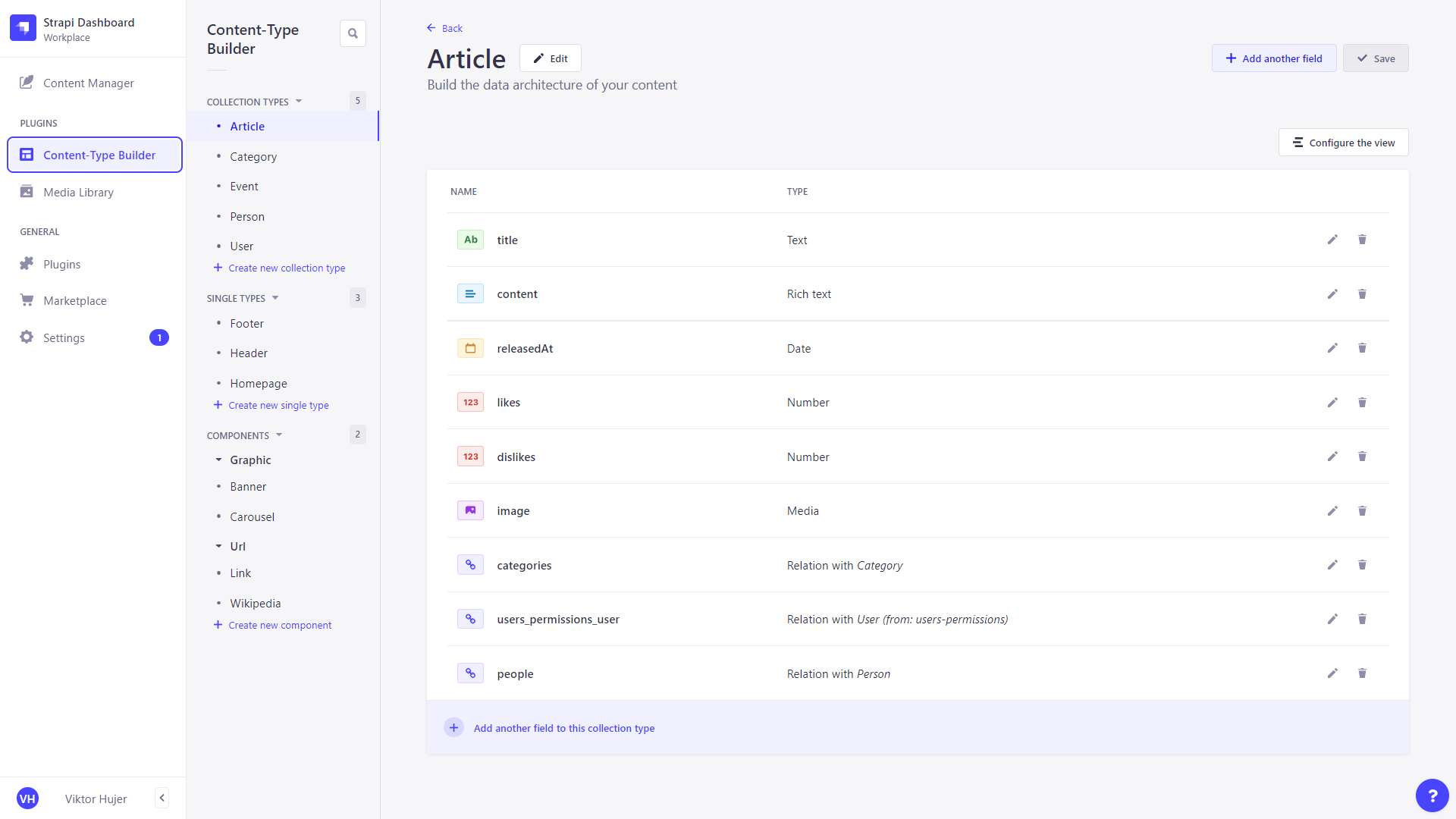
|  |  |
| --- | --- |
|  | |
|  |  |
|  |  |

Na zachycených snímcích obrazovek jsou zobrazeny různé možnosti nastavení jednotlivých polí.

Navrženou kolekci uložíme a uveřejníme. Poté přepneme opět do **Správce obsahu** (Content manager) a vložíme do nové kolekce data v podobě odborných předmětů ICT:



Zmíněné kroky opakujeme i v případě vytvoření a naplnění kolekce Article, která bude sloužit k ukládání odborných článků. Zde je struktura kolekce Article:



### Tvorba relací

Speciální pozornost si zaslouží pole categories a users\_permissions\_user. V obou případech byl použit typ **Relation** (relace, vztah), abychom pomocí něj mohli zachytit fakt, že datové objekty kolekce mohou spadat do více kategorií (předmětů) a že každý článek může být propojen s některým z uživatelů - autorem článku.

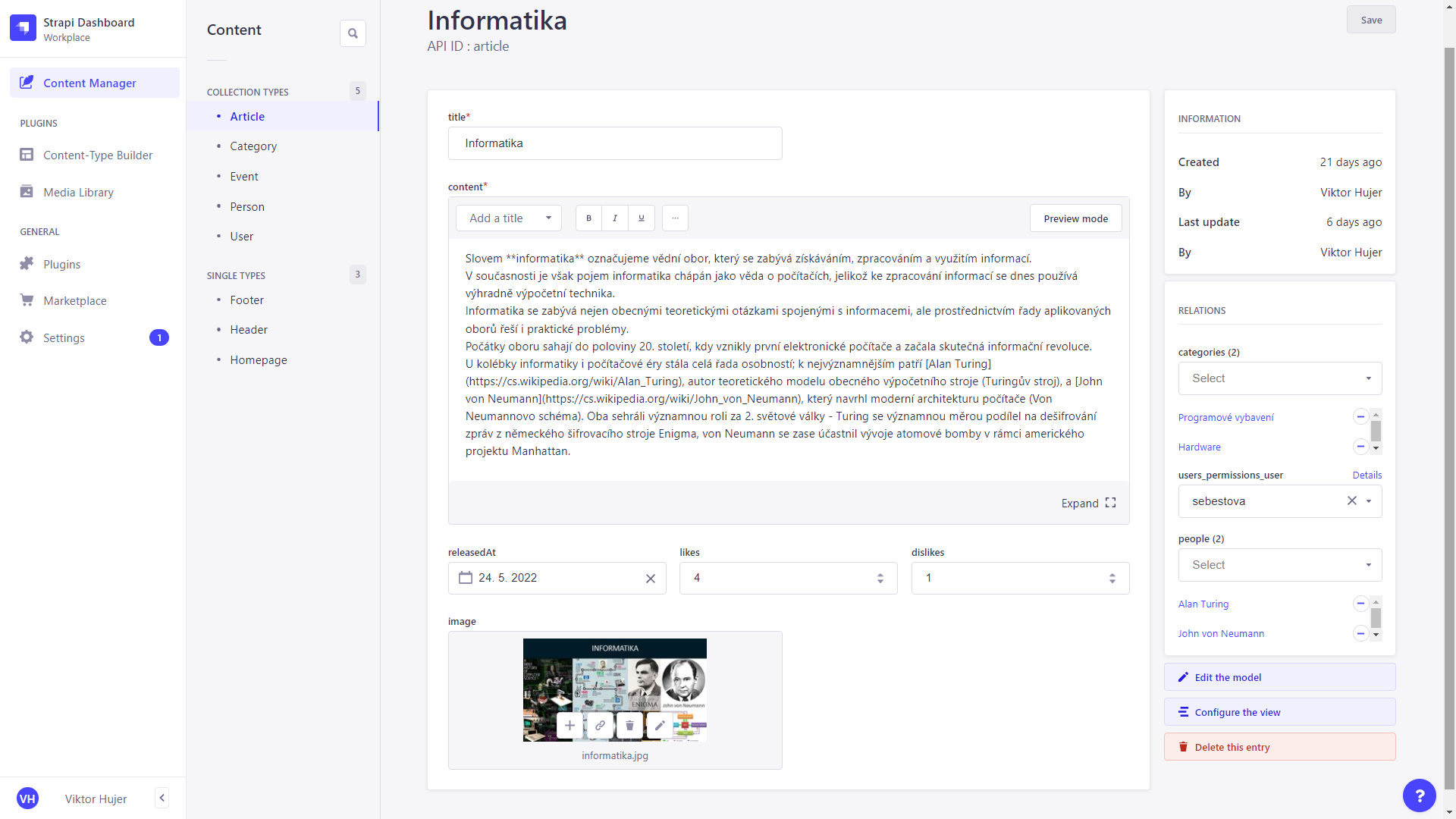
|  |  |
| --- | --- |
|  |  |

V zobrazení nastavení obou vztahů vidíme základní rozdíl ve volbě typu vztahu.

Vztah mezi články a kategoriemi je M:M (many-to-many), neboli více článků může být zařazeno do více kategorií. Tento typ vztahu se proto automaticky promítne i do kolekce kategorie, kde přibude pole articles.

Vztah mezi uživateli a články je 1:M (one-to-many), čili jeden uživatel může být spojen s více články, ale každý článek má pouze jednoho zodpovědného autora-uživatele.

Na dalším screenshotu je zachycen příklad vytvoření či editace článku ve Správci obsahu. Zatímco levá, širší část formuláře obsahuje různé formulářové prvky pro vložení samotného obsahu datového objektu, v pravém sloupci jsou kromě základních časových údajů a informaci o tvůrci obsahu umístěny relace s dalšími datovými kolekcemi. Díky tomu můžeme snadno článek zařadit do několika kategorií (zde odborných předmětů) a připojit i jméno uživatele, kterému můžeme například svěřit zodpovědnost nad úpravou článku.

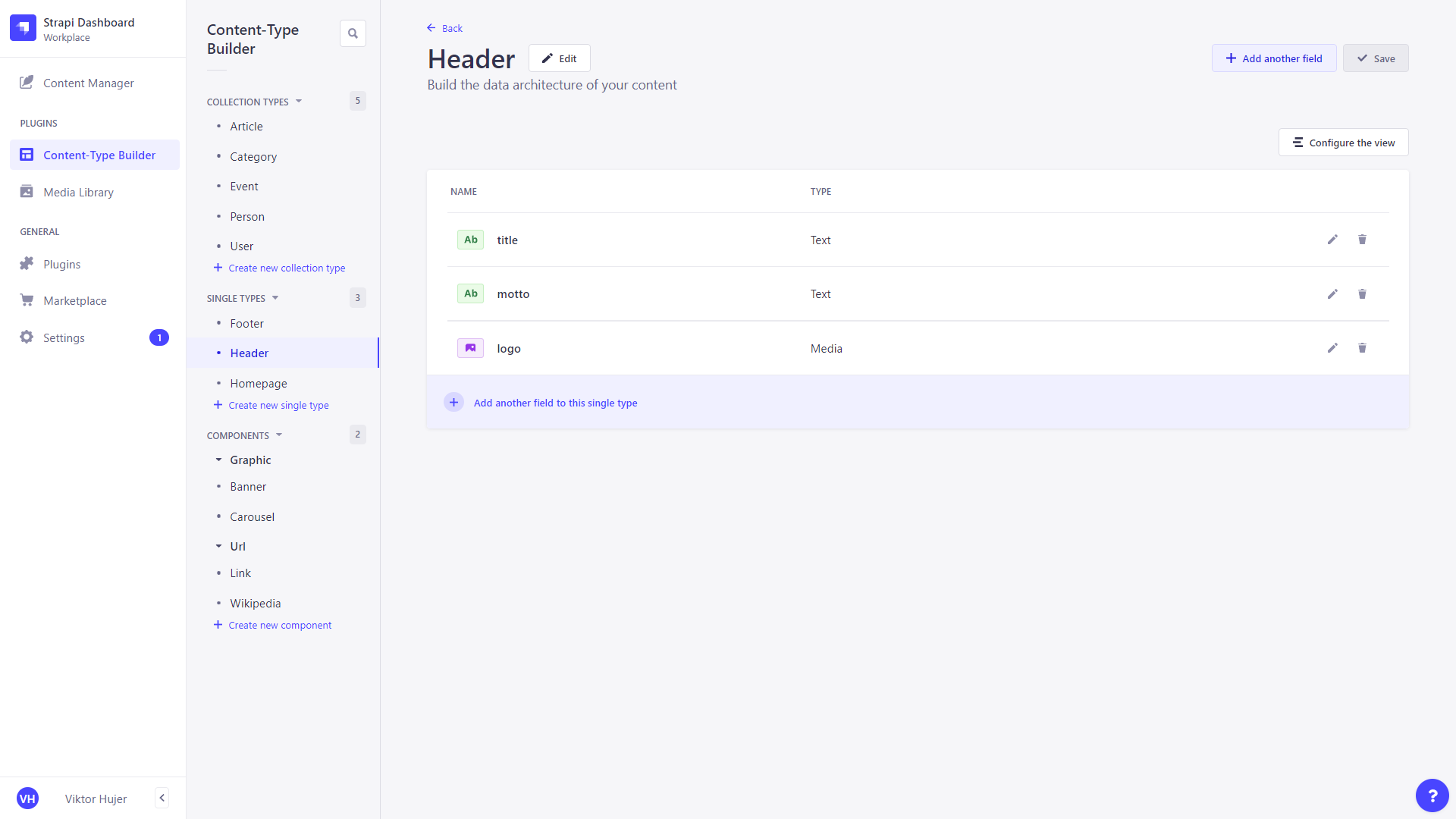


Podobných kolekcí můžeme vytvořit celou řadu; z výše uvedeného snímku obrazovky je patrné, že v backendové aplikaci tohoto InfoWebu jsou vytvořeny rovněž kolekce Event a Person (významné události respektive osobnosti v dějinách ICT). I ty lze pomocí relací svázat s jiným obsahem - například osobnosti informatiky mohou být propojeny s články, v nichž je o nich nějaká zmínka.

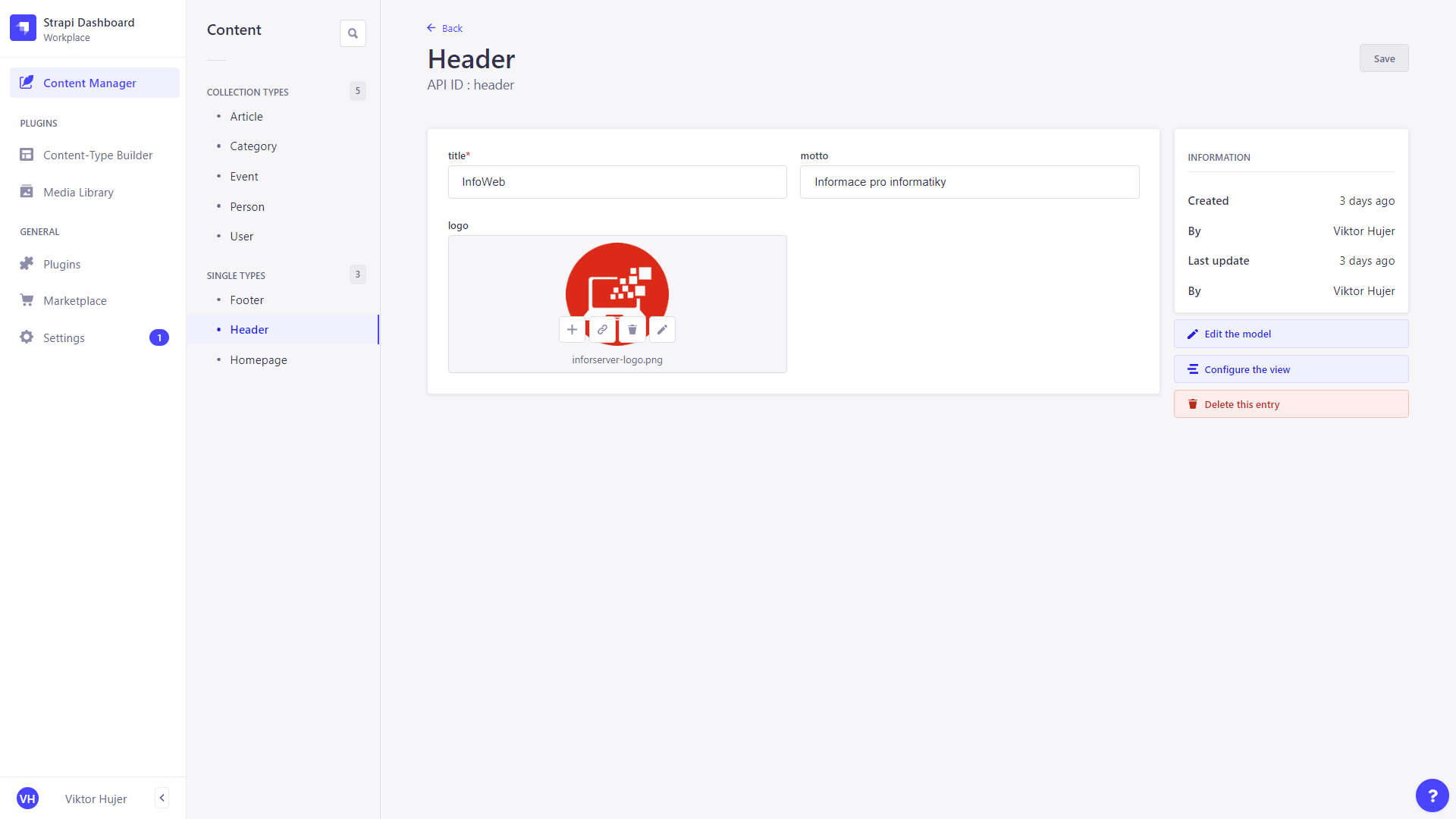
### Single Types

Vedle kolekcí, které slouží k zaznamenání vícenásobných dynamických datových prvků (článků, událostí, zboží, uživatelů apod.), můžeme CMS využít i k ukládání dat, která tvoří jedinečný a neopakující se obsah statických stránek webu, jako je např. domovská stránka, stránka s kontaktními informacemi nebo stránka představující majitele webu. Tyto **Single Types** vznikají v podstatě stejně jako kolekce, jen mohou být vytvořeny jako jedinečný datový objekt. Je možné je využít i pro opakující se části webu - typicky pro záhlaví (header), navigaci nebo zápatí (footer). Hlavní výhodou je samozřejmě snadná změna obsahu bez nutnosti zásahu do HTML kódu stránky.

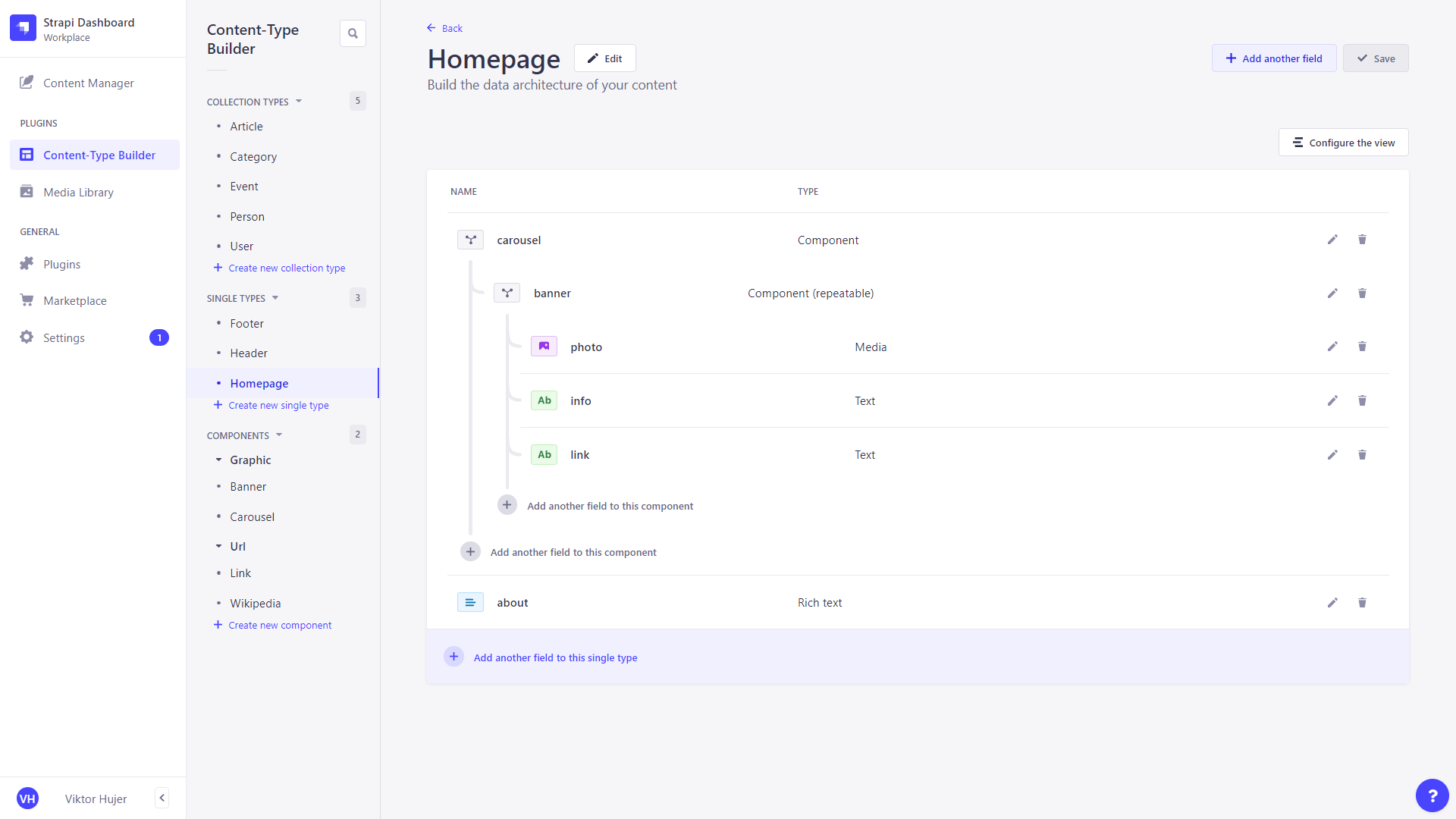
Návrh "single type" pro záhlaví stránky:



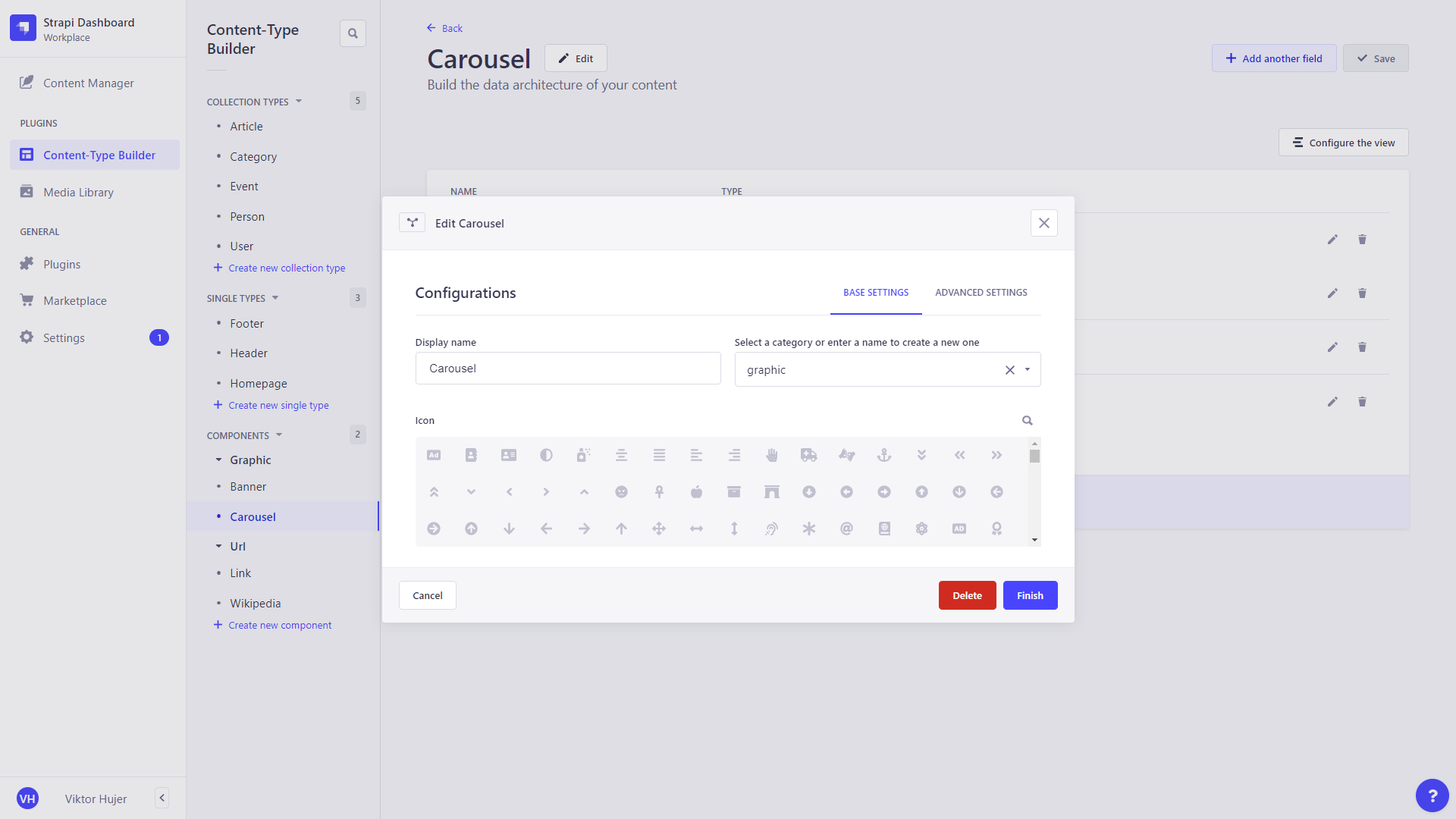
Příklad datového objektu pro záhlaví s vyplněnými daty:



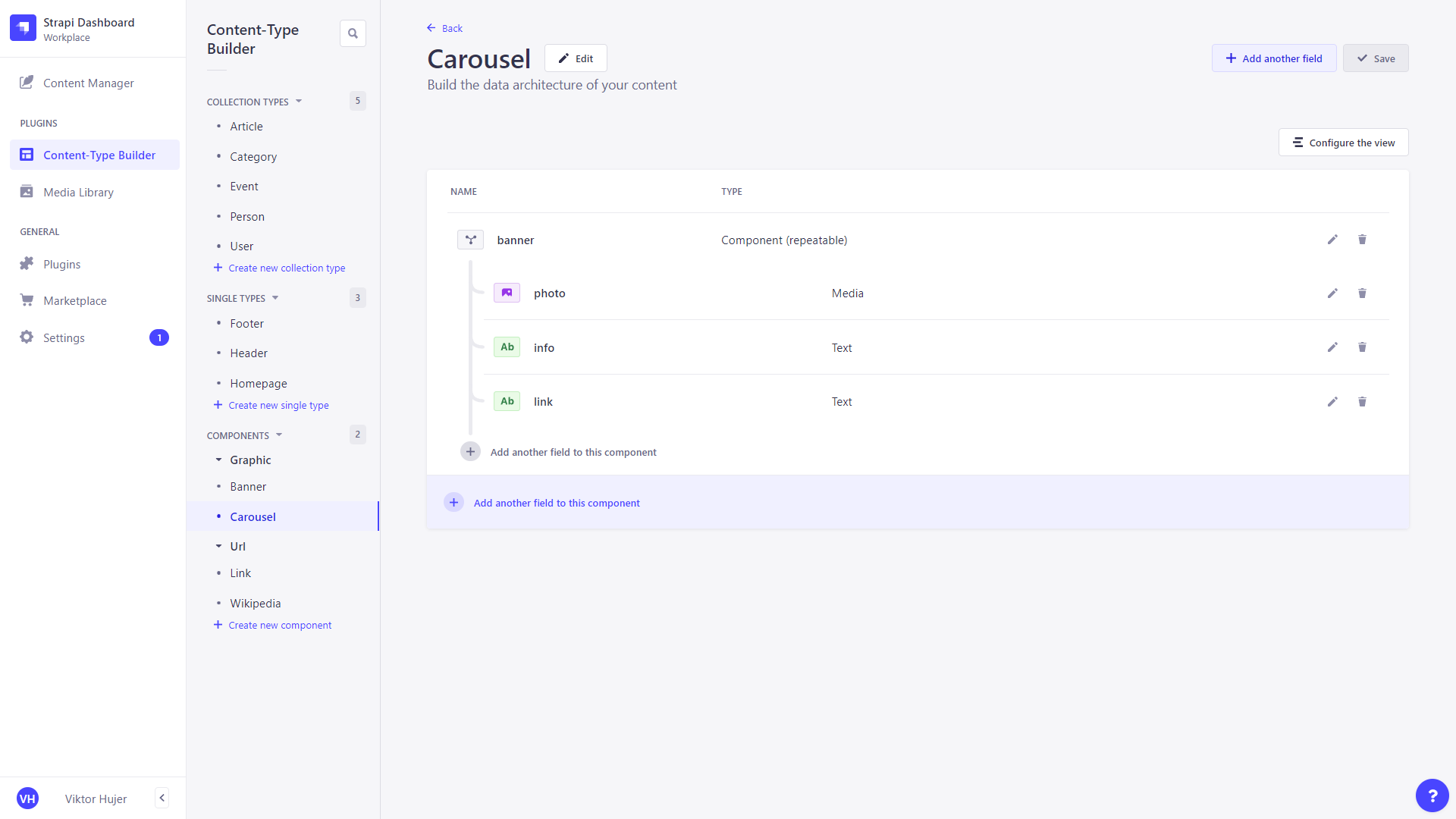
V příkladu "single type" pro domovskou stránku si můžeme ukázat i použití tzv. **komponent** (components). V tomto případě například komponentu carousel, která je tvořena ještě další vnitřní či "dětskou" opakující se (repeatible) komponentou banner.



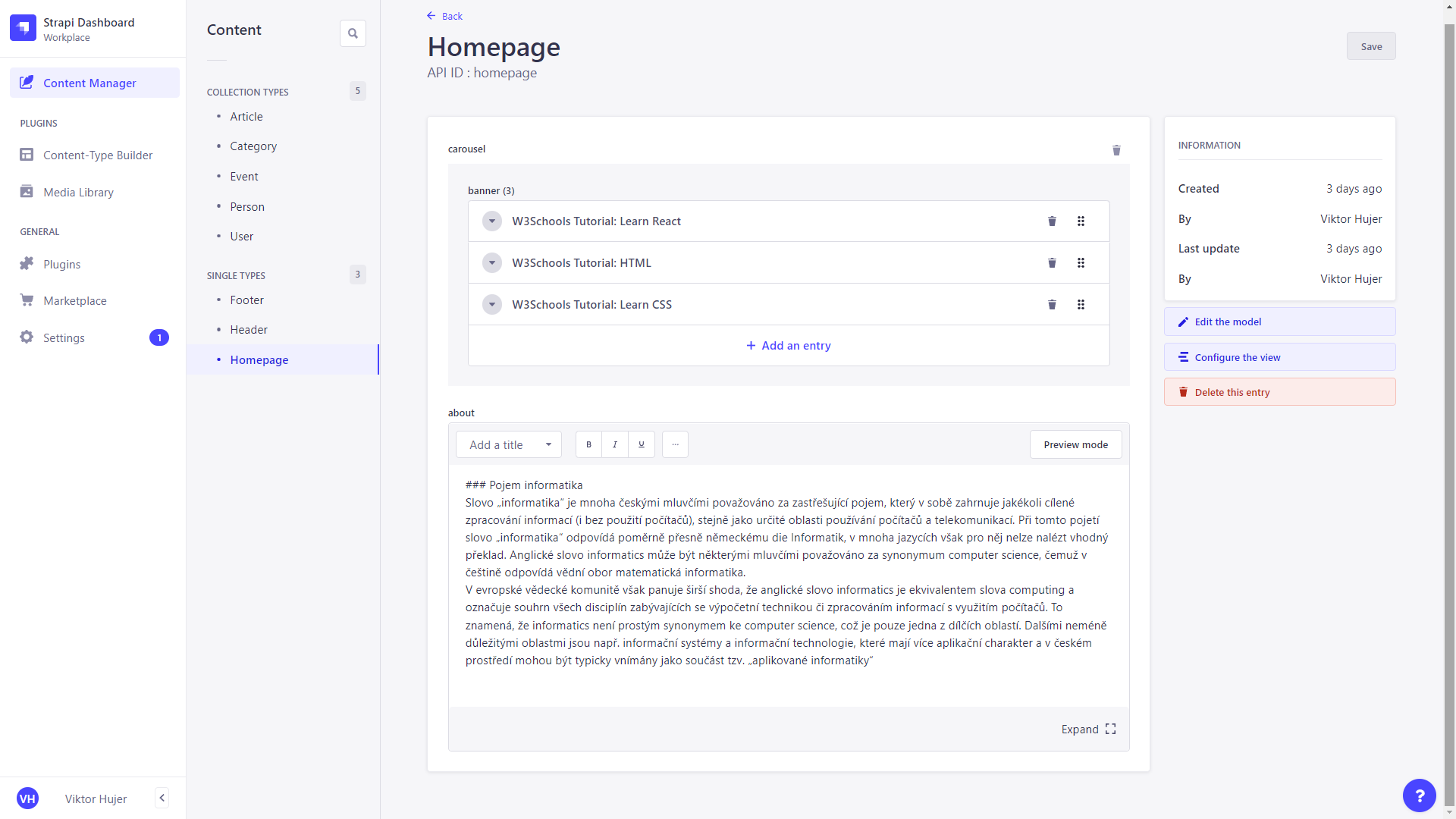
Na tomto snímku vidíme, jak lze vlastní komponentu ve Strapi vytvořit a nastavit pro ni jméno, vhodnou ikonu i kategorii.



Následující snímek ukazuje vlastní strukturu komponenty, která se v principu neliší od kolekce.



A tímto způsobem mohou být komponenty carousel i banner využity na domovské stránce a naplněny daty:



# Práce s API

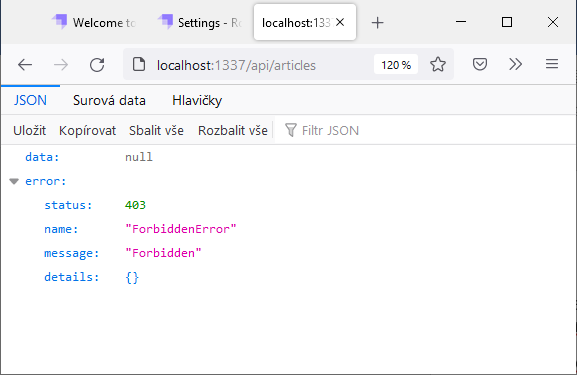
Obsah navržený i vytvořený pomocí headless CMS a uložený v některé z databází je možné zobrazit na libovolném zařízení pomocí libovolně řešené frontendové části. Důležitou roli v tom hrají API. K výhodám Strapi patří i to, že standardně v rámci běžné instalace nabízí populární rozhraní RESTful API, ale díky snadno nainstalovanému rozšíření i moderní řešení uvolněné Facebookem/Metou, které se nazývá GraphQL.

## REST API

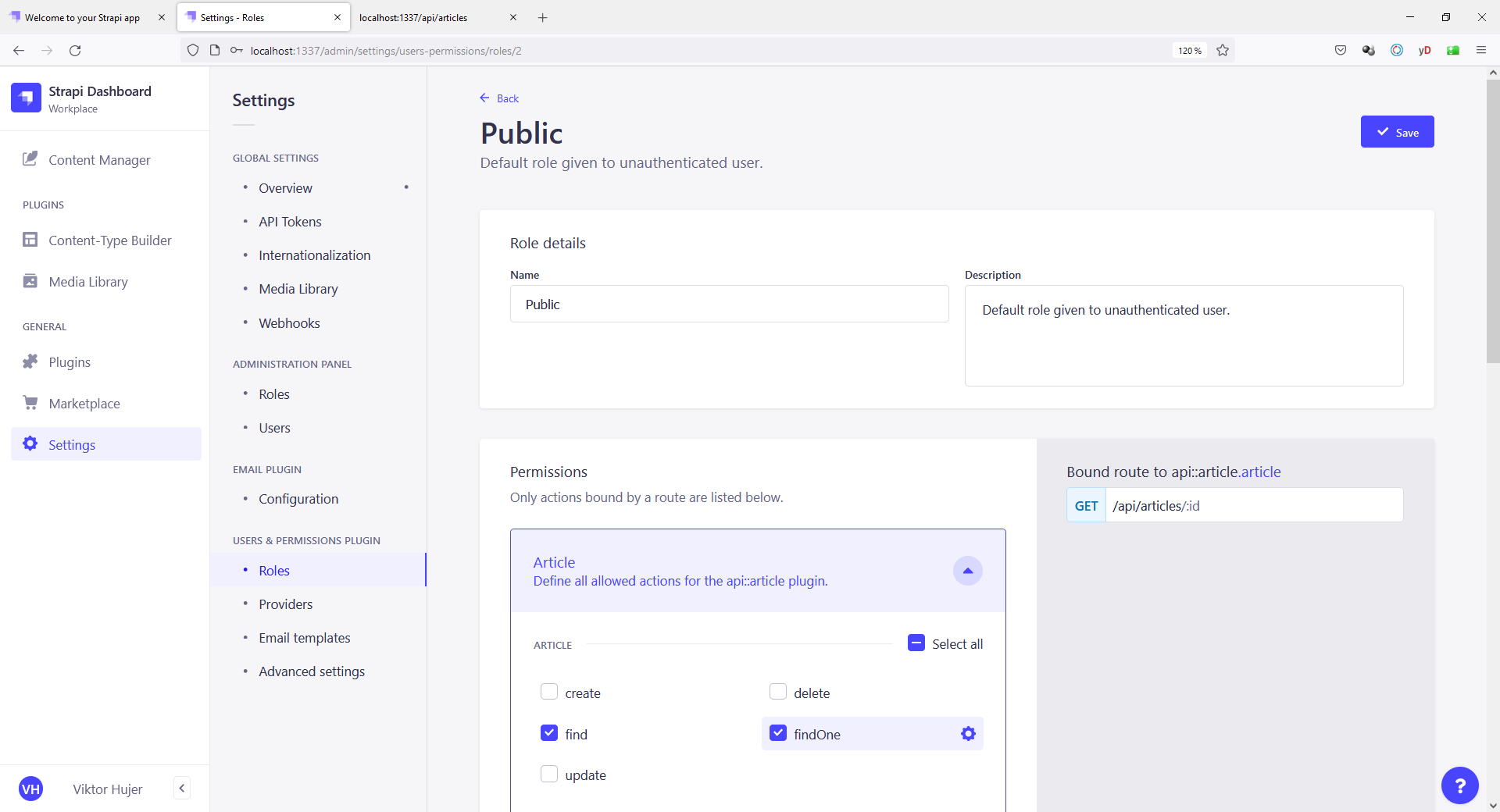
K otestování funkčnosti REST API můžeme použít běžný prohlížeč - např. Mozilla Firefox. Oficiální popis používání REST API najdeme zde:

<https://docs.strapi.io/developer-docs/latest/developer-resources/database-apis-reference/rest-api.html>

Když se ovšem nyní pokusíme zadat do prohlížeče adresu REST API endpointu pro výpis všech článků - <http://localhost:1337/api/articles> - objeví se chyba 403:



Důvodem je výchozí nastavení práv ve Strapi, které zabraňuje, aby byla data na serveru veřejně přístupná, byť jen pro prohlížení. Musíme tedy provést změnu nastavení přístupnosti článků na stránce Settings | Roles | Public:

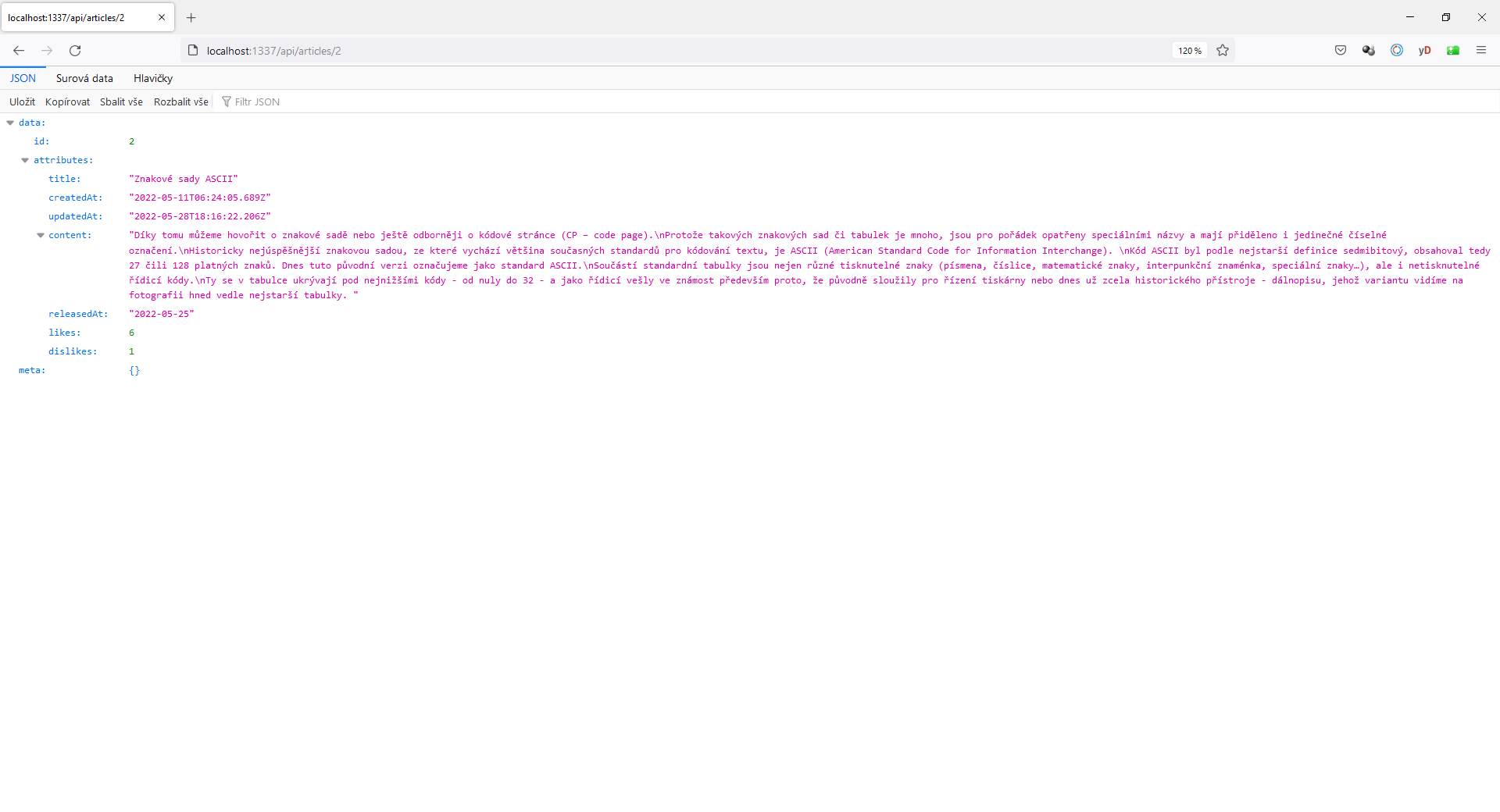


Zaškrtnutím voleb find a findOne umožníme vyhledávat i vypisovat buď více článků, nebo jeden vybraný.

Výsledek výpisu po zadání <http://localhost:1337/api/articles> obsahuje data strukturovaná do formátu JSON:



Takto vypadá výpis jednoho článku s id=2 - <http://localhost:1337/api/articles/2>:



Několik příkladů dalších výpisů:

[http://localhost:1337/api/articles?sort[0]=releasedAt&sort[1]=title](http://localhost:1337/api/articles?sort%5b0%5d=releasedAt&sort%5b1%5d=title)

(zobrazí výpis článků seřazených primárně podle data ve sloupci releasedAt a sekundárně podle sloupce title)

[http://localhost:1337/api/articles?filters[content][$contains]=info&filters[likes][$gt]=5](http://localhost:1337/api/articles?filters%5bcontent%5d%5b$contains%5d=info&filters%5blikes%5d%5b$gt%5d=5)

(zobrazí výpis článků s nastaveným filtrem pro všechny články, které mají v obsahu řetězec info a současně mají více než 5 lajků)

[http://localhost:1337/api/articles?filters[content][$contains]=info&filters[categories][shortname][$eq]=PGR](http://localhost:1337/api/articles?filters%5bcontent%5d%5b$contains%5d=info&filters%5bcategories%5d%5bshortname%5d%5b$eq%5d=PGR)

(zobrazí výpis článků s tzv. hlubokým filtrem: články musí obsahovat řetězec "info" a současně patří do kategorie označené zkratkou "PGR")

[http://localhost:1337/api/articles?populate=\*](http://localhost:1337/api/articles?populate=*)

(zobrazí u každého článku i data z propojených kolekcí - např. všechny přiřazené kategorie, připojený obrázek nebo přiřazené osobnosti)

[http://localhost:1337/api/articles?pagination[page]=1&pagination[pageSize]=3](http://localhost:1337/api/articles?pagination%5bpage%5d=1&pagination%5bpageSize%5d=3)

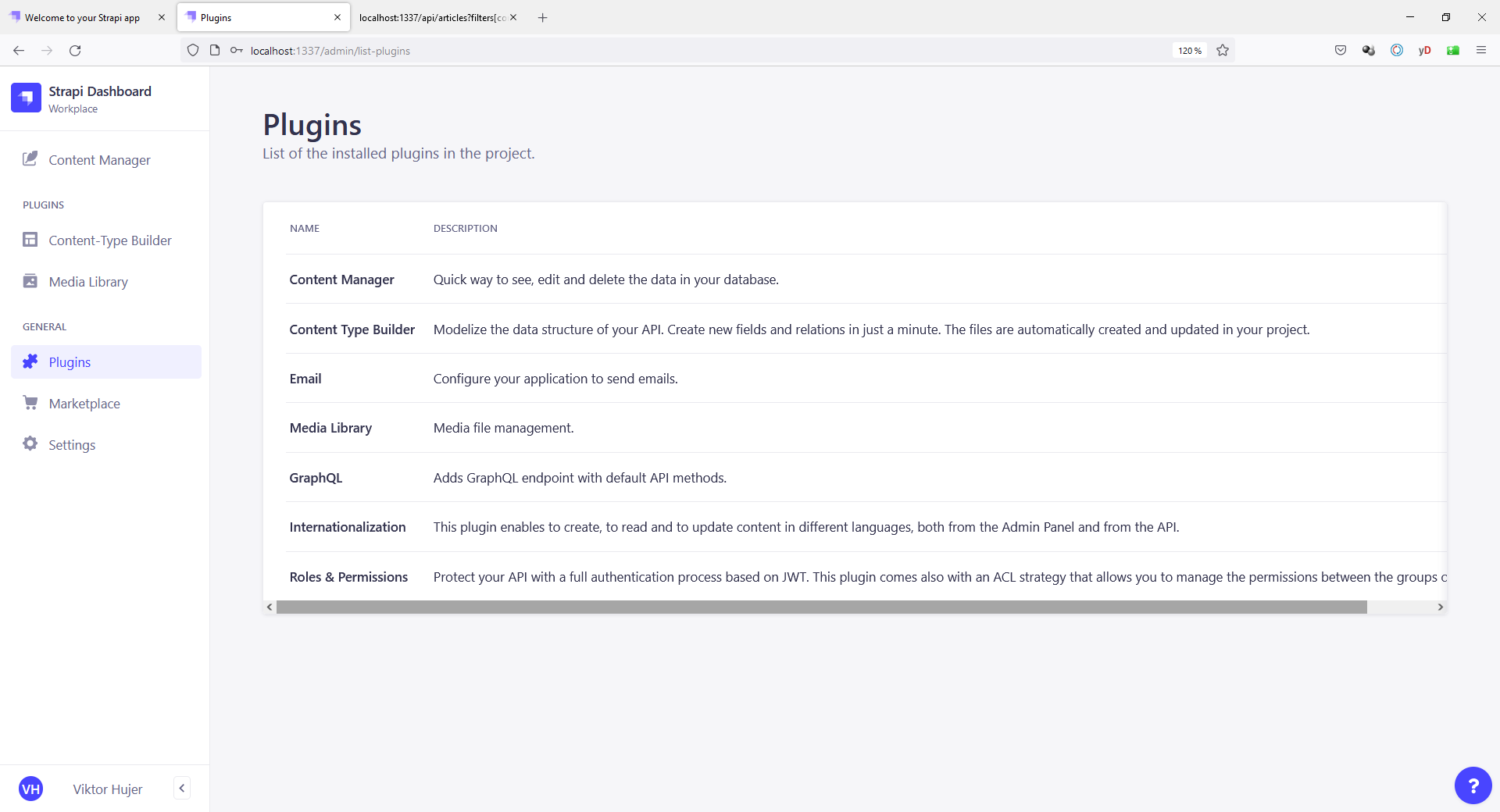
(zobrazí výpis 1 stránky článků rozdělených na stránky po 3 článcích)

## GraphQL

Pro podporu tohoto moderního API je nutné do Strapi doinstalovat potřebný plugin příkazem:

**npm run strapi install graphql**

O úspěchu instalace GraphQL se poté můžeme přesvědčit i na stránce Plugins:

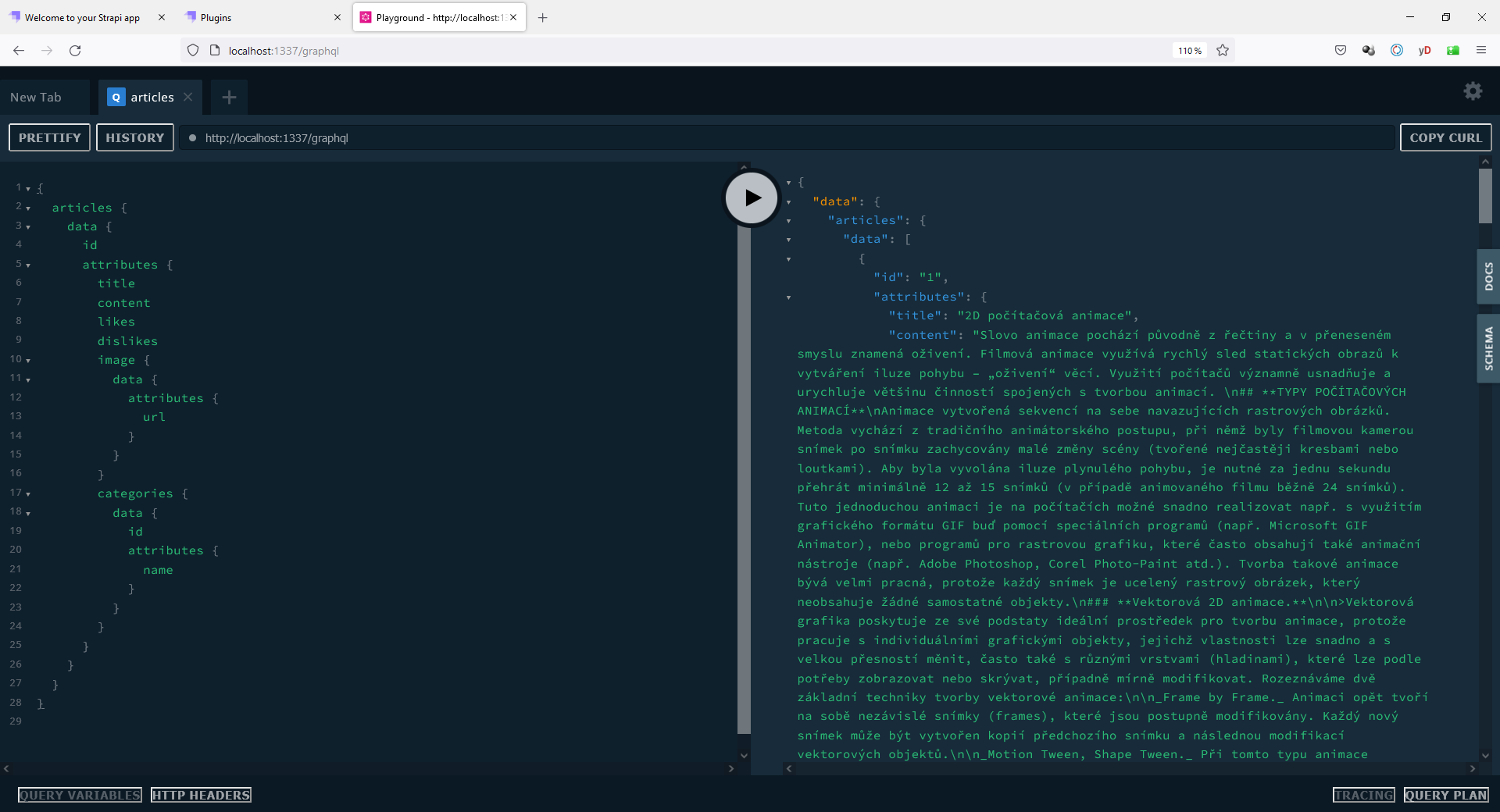


Vlastní dotazy a mutace v GraphQL můžeme testovat v prostředí Playground na adrese:

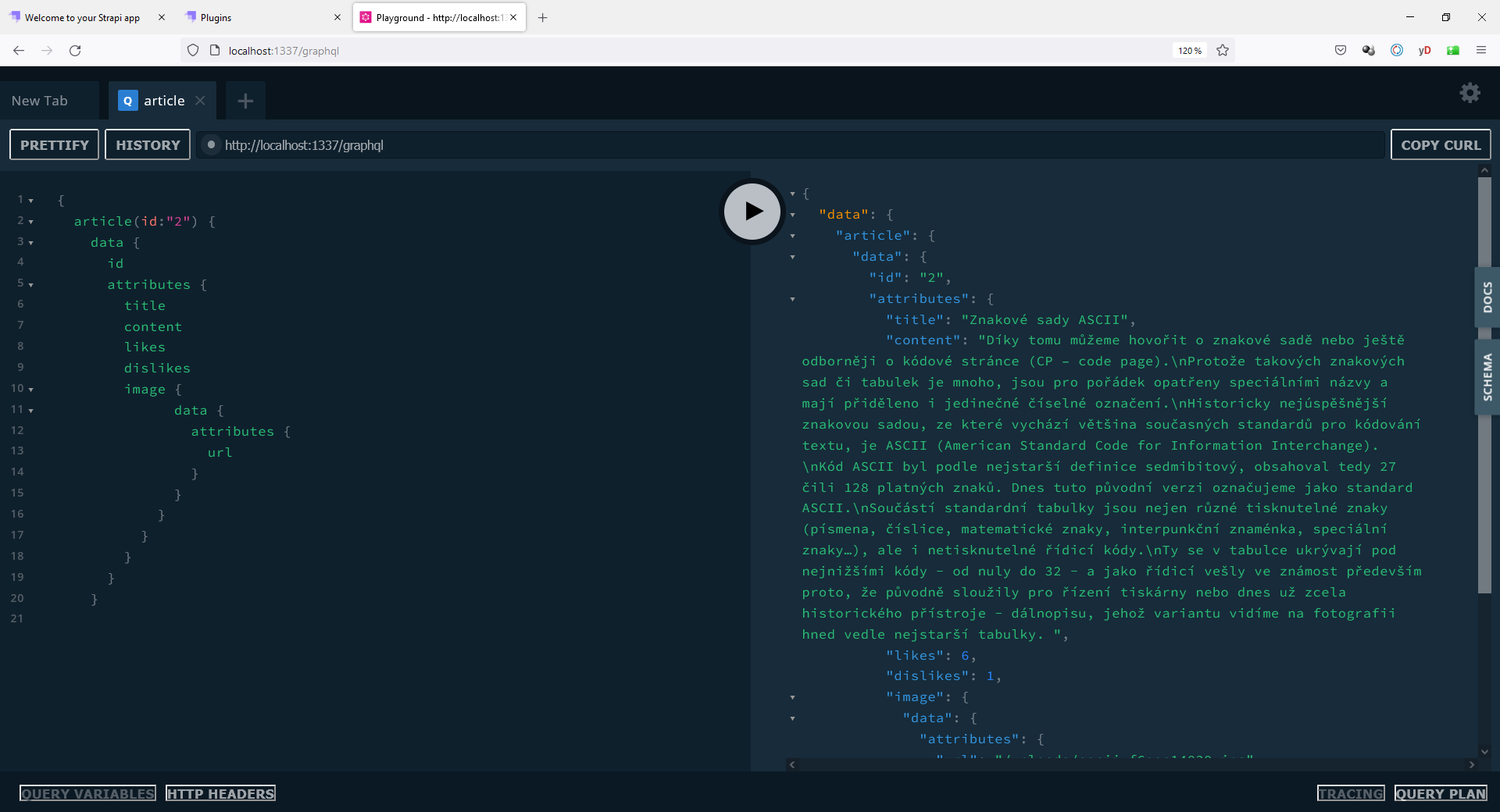
<http://localhost:1337/graphql>

Příklady některých GraphQL dotazů:

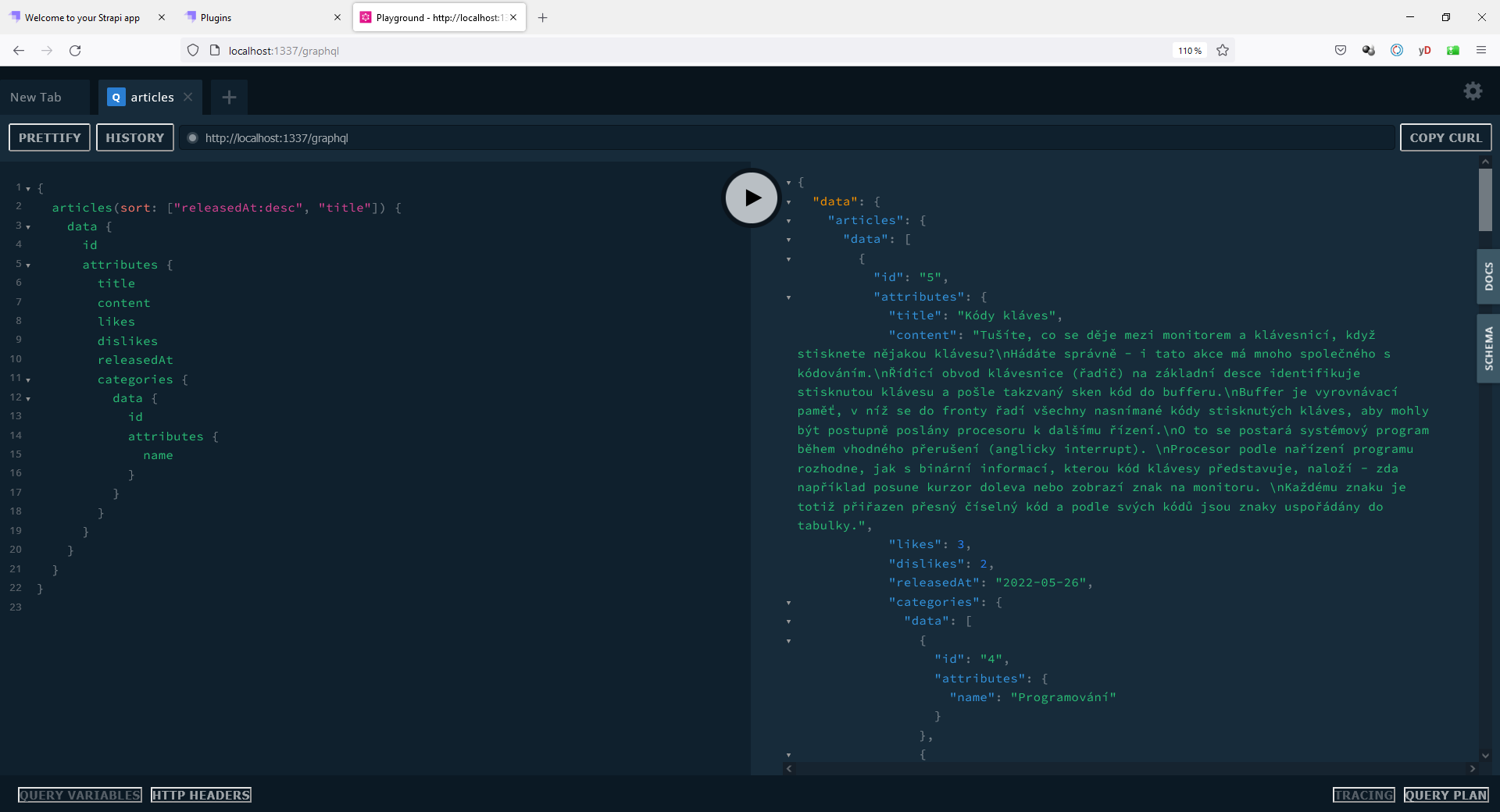
Výpis všech článků včetně kategorií a obrázků:



Výpis jednoho článku s id=2:

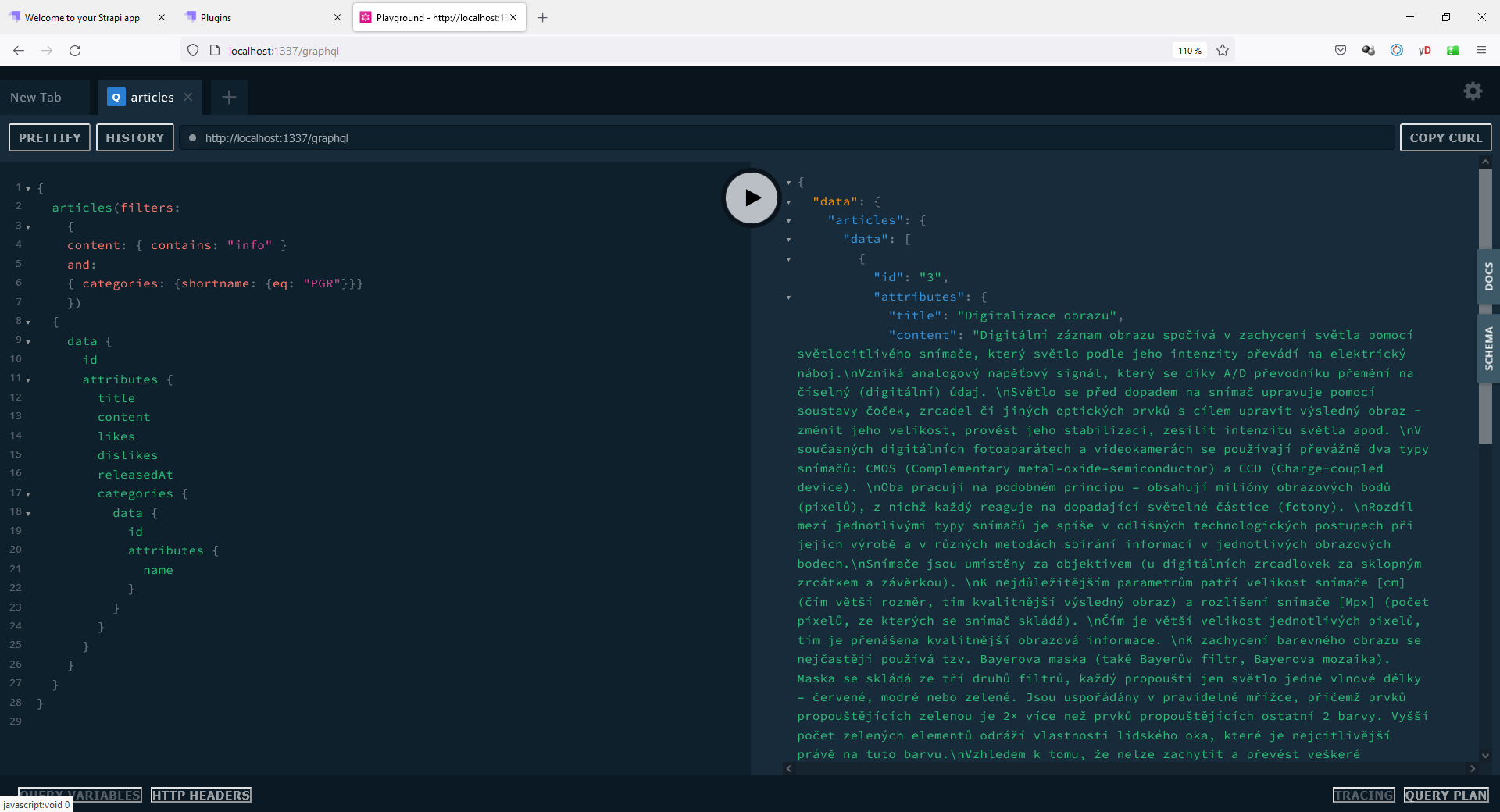


Výpis článků seřazených primárně podle data "releasedAt" sestupně (desc) a sloupce "title":



Další informace o využití GraphQL ve Strapi 4 naleznete zde:

<https://docs.strapi.io/developer-docs/latest/developer-resources/database-apis-reference/graphql-api.html#unified-response-format>



# Tvorba frontendu

## Použité technologie

K tvorbě frontendové části využijeme opět běhové prostředí (runtime environment) NodeJS. Budeme ho potřebovat pro kompilaci a spouštění javascriptového kódu využívajícího knihovnu React JS.

### React

Tato stále populárnější technologie byla vytvořena programátory Facebooku (dnes Meta) a uvolněna pro ostatní frontendové vývojáře.

React usnadňuje vytváření interaktivních uživatelských rozhraní s využitím komponent. Komponenty jsou dílčí části/bloky webové stránky, které jsou vytvářeny jako samostatné JS moduly a podle potřeby propojovány ve větší celky. Protože základem webových stránek je značkovací jazyk HTML, objevují se značky také v komponentách, ovšem ve formě speciálního jazyka JSX (JavaScript XML), která umožňuje kromě samotných značek vkládat do kódu i výrazy a proměnné v JavaScriptu.

Díky tomu můžeme snadno a dynamicky předávat stránkám data získaná z backendových serverů nejčastěji prostřednictvím některého z API (REST API, GraphQL). React je navržen tak, aby na každou změnu dat reagoval co nejrychleji a nejefektivněji. Výsledná aplikace se proto svým chováním velmi blíží aplikacím desktopovým.

### Knihovna komponent react-boostrap

Další velkou předností je tzv. znovupoužitelnost jednou vytvořených komponent - můžeme je nejen opakovaně používat na různých místech v naší vlastní aplikaci, ale poskytnout je případně i jako moduly pro jiné aplikace. A naopak máme samozřejmě možnost využívat celé sady profesionálně připravených komponent připojením externích knihoven - příkladem mohou být například knihovny **react-bootstrap** (<https://react-bootstrap.github.io/>) nebo **Material UI** (<https://mui.com/>).

### Apollo Client

**Apollo Client** (<https://www.apollographql.com/>) je knihovna pro JavaScript, která umožňuje spravovat místní i vzdálená data pomocí GraphQL. Základní knihovna @apollo/client poskytuje vestavěnou integraci s Reactem.

## Založení a základní nastavení aplikace

### Vytvoření struktury aplikace v Reactu

V základní složce našeho projektu (infoserver) spustíme příkaz:

npx create-react-app frontend

Během této operace se vytvoří nová podsložka frontend, která obsahuje základní strukturu aplikace s podporou knihovny React.

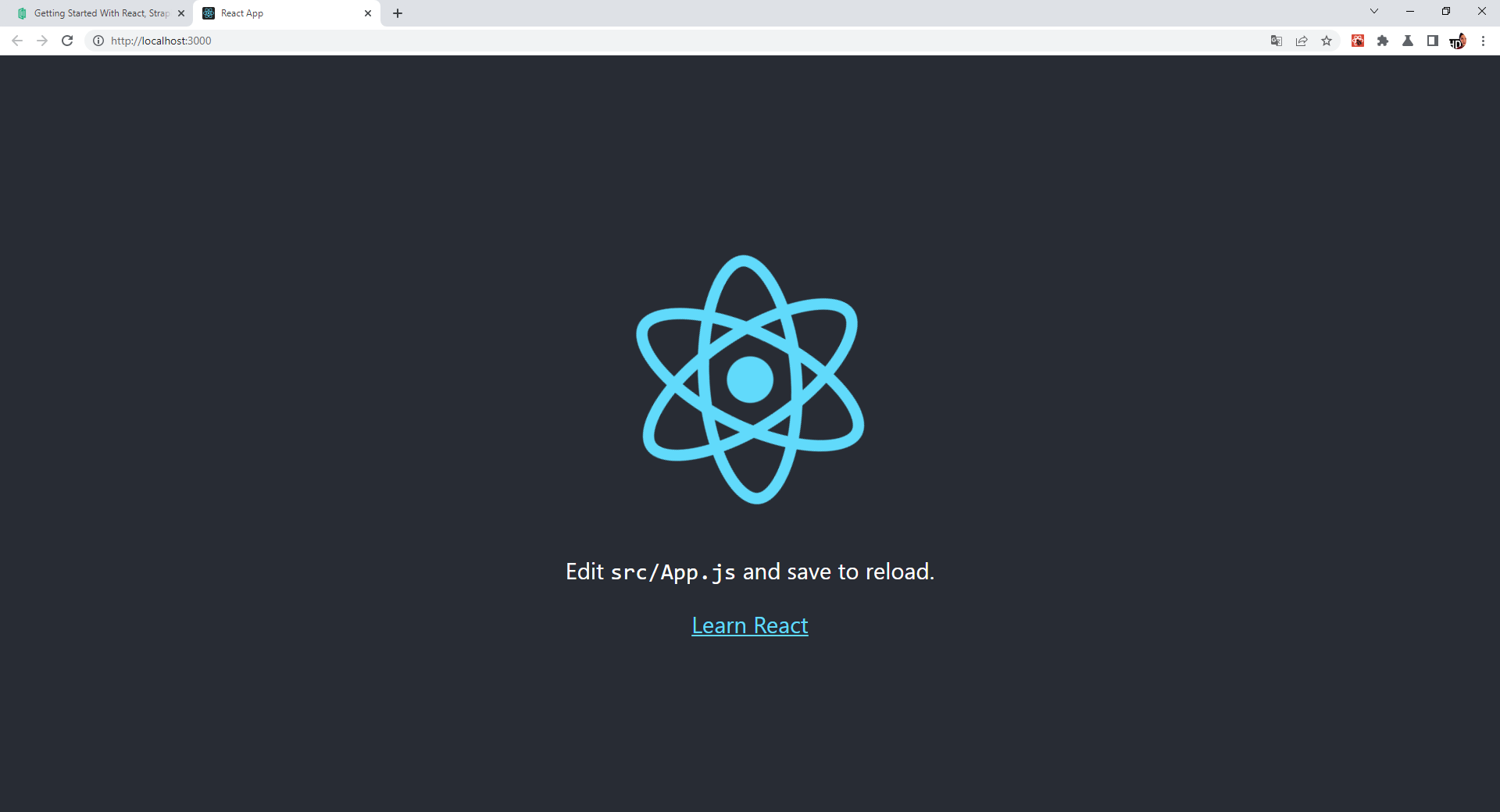
Složku frontend otevřeme jako projekt ve VSCode. V něm si můžeme prohlédnout obsah vytvořených podsložek:

|  |  |
| --- | --- |
|  | **node\_modules** - obsahuje všechny závislosti - nainstalované moduly aplikace  **public** - veřejně dostupná složka statických souborů webu, nejdůležitější je soubor   * index.html, který aplikace v Reactu využívá jako rámec pro zobrazení komponent   **src** - základní složka se zdrojovými soubory aplikace tvořené pomocí Reactu   * **index.js** - soubor, které se stará o renderování (vykreslení) komponenty App, hlavní komponenty celé aplikace * **index.css** - soubor obsahující kaskádové styly dostupné v rámci celé aplikace * **App.js** - javascriptový modul pojmenovaný stejně jako hlavní komponenta celé aplikace *App*; obsahuje komponenty důležité pro chod celé aplikace. * **App.css** - úpravy pomocí CSS přímo vázané na komponentu App |

Nyní můžeme v terminálu spustit frontendovou aplikaci příkazem:

**npm start**

Po chvíli kompilace a sestavení souboru bundle.js (pomocí webpack), který obsahuje veškerý kód v JS, se v prohlížeči objeví výchozí ukázková stránka vytvořená s využitím Reactu.



### Instalace externích knihoven

Pro potřeby naší aplikace nainstalujeme ještě několik dalších knihoven:

npm install @apollo/client graphql react-router-dom

* **apollo-client** - pomůže nám vytvářet a spravovat dotazy GraphQL v reactové aplikaci.
* **graphql** - poslouží při parsování GraphQL dotazů.
* **react-router-dom** - umožní nastavit routování aplikace.

### Instalace knihovny react-bootstrap

Abychom mohli efektivně využívat výhod známého CSS frameworku Boostrap, nainstalujeme si knihovnu připravených komponent react-bootstrap:

npm install react-bootstrap bootstrap

Kromě instalace potřebných modulů je ještě nutné importovat CSS do souboru index.js (nebo App.js):

{/\* The following line can be included in your src/index.js or App.js file\*/}

import 'bootstrap/dist/css/bootstrap.min.css';

Alternativně můžeme soubor s CSS připojit v hlavičce souboru **index.html**:

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" />

### Úprava souboru index.js

Abychom zjednodušili tvorbu naší aplikace, můžeme ze souboru **index.js** odebrat metodu reportWebVitals() včetně příslušného importu. Tento modul se dá využít k měření výkonnosti aplikace, což v tomto případě dělat nebudeme.

Nová podoba souboru index.js vypadá takto:

import React from 'react';

import ReactDOM from 'react-dom/client';

import 'bootstrap/dist/css/bootstrap.min.css';

import './index.css';

import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

  <React.StrictMode>

    <App />

  </React.StrictMode>

);

### Úprava souboru App.js

Nyní upravíme soubor obsahující hlavní komponentu aplikace:

/\* Import základní knihovny Reactu \*/

import React from 'react'

/\* Soubor s CSS pro komponentu App \*/

import './App.css';

/\* Import objektů ApolloClient, InMemoryCache, ApolloProvider z knihovny @apollo/client

   Tyto objekty usnadní práci s GraphQL dotazy a zefektivní fungování vyrovnávací paměti - cache \*/

import { ApolloClient, InMemoryCache, ApolloProvider } from "@apollo/client";

/\* Import objektů BrowserRouter, Routes, Route z knihovny react-router-dom, pomocí kterých můžeme

   efektivně vyřešit tzv. routování, neboli směrování v aplikaci - vytvoření odkazů na různá místa v aplikaci \*/

import { BrowserRouter, Routes, Route } from "react-router-dom";

/\* Import komponenty Homepage, která se nachází v podsložce pages \*/

import Homepage from './pages/Homepage';

// Inicializace ApolloClienta

const client = new ApolloClient({

  /\* Nastavení adresy, na které backendový server přijímá dotazy GraphQL \*/

  uri: "http://localhost:1337/graphql",

  /\* Základní nastavení vyrovnávací paměti (cache) pro urychlení načtení často používaných částí webu \*/

  cache: new InMemoryCache(),

});

/\* Hlavní komponenta celé aplikace \*/

function App() {

  return (

    /\* Konstrukce komponenty s využitím jazyka JSX \*/

    /\* Inicializace routeru - směrovače \*/

    <BrowserRouter>

      {/\* Inicializace služby Apollo s předáním nastavení klienta \*/}

      <ApolloProvider client={client}>

        <main className='container'>

          {/\* Nastavení routování - směrování požadavků na správné komponenty \*/}

          <Routes>

            {/\* URL adresa/cesta k domovské stránce \*/}

            <Route path="/" element={<Homepage />} />

          </Routes>

        </main>

      </ApolloProvider>

    </BrowserRouter>

  );

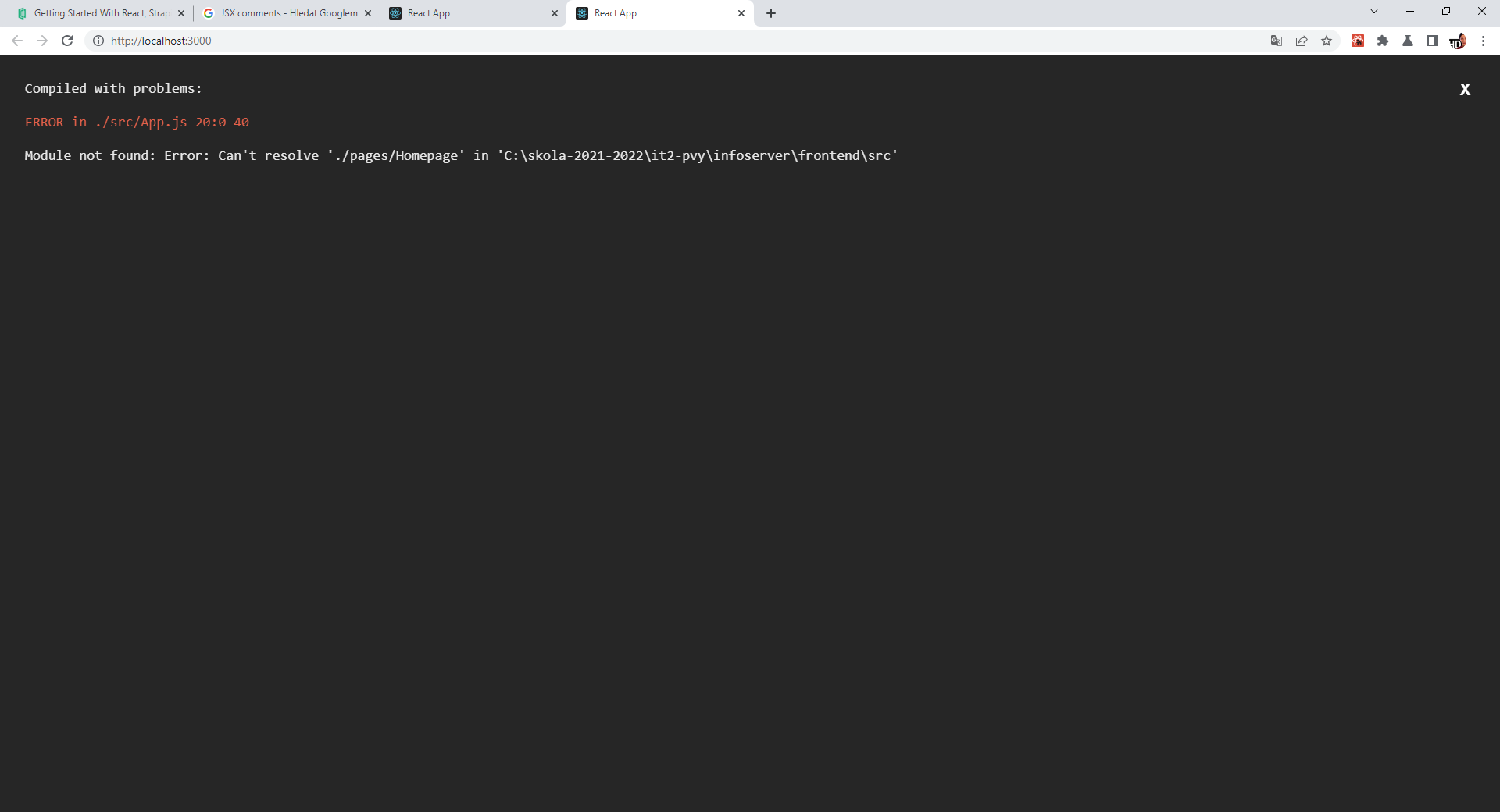
}

/\* Exportování modulu App, aby mohl být naimportován v jiných modulech \*/

export default App;

Provedeme opětovný start aplikace příkazem npm start.

Stránka v prohlížeči, ale i terminál nebo konzola příkazového řádku nyní hlásí chybu:



Problém je v absenci podsložky pages a souboru Homepage.js. Obojí proto vytvoříme.

### Vytvoření a úprava souboru Homepage.js

Jednoduchý kód souboru **Homepage.js**, který jsme vytvořili v podsložce **pages**, vypadá následovně:

/\* Import základní knihovny Reactu \*/

import React from "react";

/\* Export funkce Homepage(), která představuje komponentu základní stránky aplikace \*/

export default function Homepage() {

  return (

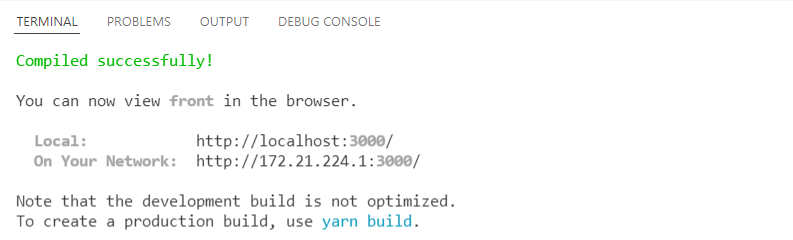
    /\* Kód v jazyce JSX, v němž je možné používat i běžné značky HTML \*/

    <h1>Domovská stránka</h1>

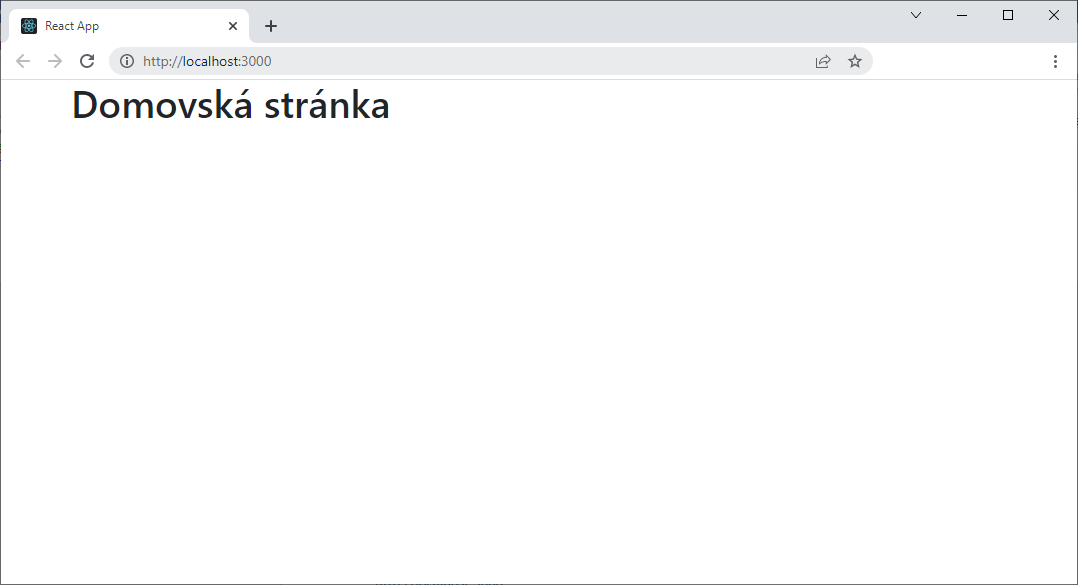
  );

}

Nyní můžeme příkazem **npm start** znovu spustit aplikaci. Pokud vše proběhne, jak má, objeví se v terminálu zpráva o úspěšné kompilaci kódu a spuštění webového serveru na adrese <http://localhost:3000>:



Po zadání zmíněné URL adresy do prohlížeče uvidíme základní stránku:



### Vytvoření komponent Header a Footer

Součástí kostry našich stránek budou minimálně dvě komponenty, které se budou objevovat v rámci celé aplikace. Ve složce **src** pro ně vytvoříme podsložku **components** a v ní postupně založíme dva samostatné moduly v JS.

**Header.js:**

/\* Import základní knihovny Reactu \*/

import React from 'react';

/\* Komponenta je vytvořena a exportována jako konstanta s využitím arrow function. \*/

export const Header = (props) => {

    /\* Komponentě mohou být při jejím volání předány atributy/props title a motto.

       Jejich hodnoty načteme do konstant title a motto. \*/

    const { title, motto } = props;

    /\* Zajistí vyrenderování komponenty - kód v jazyce JSX je převeden do podoby HTML. \*/

    return (

        /\* V jazyce JSX používáme k vkládání tříd atribut className. \*/

        <header className='bg-danger p-5 text-center'>

            {/\* Do složených závorek můžeme vkládat proměnné/konstanty nebo výrazy v JS - v tomto případě konstantu title. \*/}

            <h1 className='display-2 text-white'>{ title }</h1>

            {/\* Provede podmíněné renderování bloku small - jen pokud existuje atribut/props motto. \*/}

            { motto &&

                <small style={{fontSize: '30px', color: '#aaa', borderTop: '1px solid #aaa', textTransform: 'uppercase'}}>{ motto }</small>

            }

            {/\* V JSX lze použít i vložené styly, ale je třeba vložit je jako JS objekty, tedy i s použitím camel case syntaxe. \*/}

        </header>

    );

}

**Footer.js:**

/\* Import základní knihovny Reactu \*/

import React from 'react';

/\* Import komponenty Image z externí knihovny react-bootstrap - viz https://react-bootstrap.github.io/components/images/ \*/

import { Image } from 'react-bootstrap';

/\* Připojení souboru Footer.css, který je umístěn ve shodné složce a obsahuje nastavení CSS pro komponentu \*/

import './Footer.css';

/\* Komponenta je vytvořena a exportována jako konstanta s využitím arrow function. \*/

export const Footer = (props) => {

    /\* Komponentě mohou být při jejím volání předány atributy/props logo a copyright.

       Jejich hodnoty načteme do konstant logo a copyright. \*/

    const { logo, copyright } = props;

    /\* Konstanta logoStyle obsahuje nastavení vloženého stylu pro odstavec s logem. \*/

    const logoStyle = {

        width: '300px',

        margin: '20px auto',

        padding: '20px',

        backgroundColor: 'rgba(255,255,255,.5)',

        boxShadow: '5px 5px'

    };

    /\* Do proměnné year načteme pomocí objektu Date a metody getFullYear() aktuální rok. \*/

    let year = new Date().getFullYear();

    /\* Zajistí vyrenderování komponenty - kód v jazyce JSX je převeden do podoby HTML \*/

    return (

        <footer className='bg-dark mt-3 p-3 text-center'>

            {/\* Provede podmíněné renderování odstavce - jen pokud existuje atribut/props logo \*/}

            { logo &&

                <p style={ logoStyle }><Image src={ logo } alt="Logo" /></p>

            }

            {/\* Do složených závorek můžeme vkládat proměnné/konstanty nebo výrazy v JS \*/}

            <p className='copyright'>&copy; { year } - { copyright.projectName } - autor: <b>{ copyright.projectAuthor }</b></p>

        </footer>

    );

}

K souboru Footer.js je připojen jednoduchý soubor s kaskádovými styly **Footer.css**:

.copyright {

    font-size: small;

    color: #ddd;

}

Obě nově vytvořené komponenty vyvoláme v souboru App.js:

/\* Import základní knihovny Reactu \*/

import React from 'react'

/\* Soubor s CSS pro komponentu App \*/

import './App.css';

/\* Import objektů ApolloClient, InMemoryCache, ApolloProvider z knihovny @apollo/client

   Tyto objekty usnadní práci s GraphQL dotazy a zefektivní fungování vyrovnávací paměti - cache \*/

import { ApolloClient, InMemoryCache, ApolloProvider } from "@apollo/client";

/\* Import objektů BrowserRouter, Routes, Route z knihovny react-router-dom, pomocí kterých můžeme

   efektivně vyřešit tzv. routování, neboli směrování v aplikaci - vytvoření odkazů na různá místa v aplikaci \*/

import { BrowserRouter, Routes, Route } from "react-router-dom";

/\* Import komponenty Homepage, která se nachází v podsložce pages \*/

import Homepage from './pages/Homepage';

/\* Importy modulů obsahujících komponenty Header a Footer.

   Protože jsou exportovány jako konstanty, je nutné vložit jejich názvy mezi složené závorky. \*/

import { Header } from './components/Header';

import { Footer } from './components/Footer';

// Inicializace ApolloClienta

const client = new ApolloClient({

  /\* Nastavení adresy, na které backendový server přijímá dotazy GraphQL \*/

  uri: "http://localhost:1337/graphql",

  /\* Základní nastavení vyrovnávací paměti (cache) pro urychlení načtení často používaných částí webu \*/

  cache: new InMemoryCache(),

});

/\* Hlavní komponenta celé aplikace \*/

function App() {

  return (

    /\* Konstrukce komponenty s využitím jazyka JSX \*/

    /\* Inicializace routeru - směrovače \*/

    <BrowserRouter>

      {/\* Inicializace služby Apollo s předáním nastavení klienta \*/}

      <ApolloProvider client={client}>

        {/\* Vložení komponenty Header, které předáváme atributy/props title a motto. (viz (viz https://reactjs.org/docs/components-and-props.html)) \*/}

        <Header title="InfoWeb" motto="Informace pro informatiky" />

        <main className="container">

          {/\* Nastavení routování - směrování požadavků na správné komponenty \*/}

          <Routes>

            {/\* URL adresa/cesta k domovské stránce \*/}

            <Route path="/" element={<Homepage />} />

          </Routes>

        </main>

        {/\* Vložení komponenty Footer, které předáváme atributy/props logo a copyright.

            Soubor skola-logo.png musí být umístěn ve složce public, která je chápána jako základní složka webu (obsahuje statické soubory).

            Atribut/props copyright je v tomto případě předáván jako JS objekt, který má dvě vnitřní proměnné/atributy projectName a projectAuthor. \*/}

        <Footer logo="/skola-logo.png" copyright={{projectName: "Ročníkový projekt IT2", projectAuthor: "Viktor Hujer"}} />

      </ApolloProvider>

    </BrowserRouter>

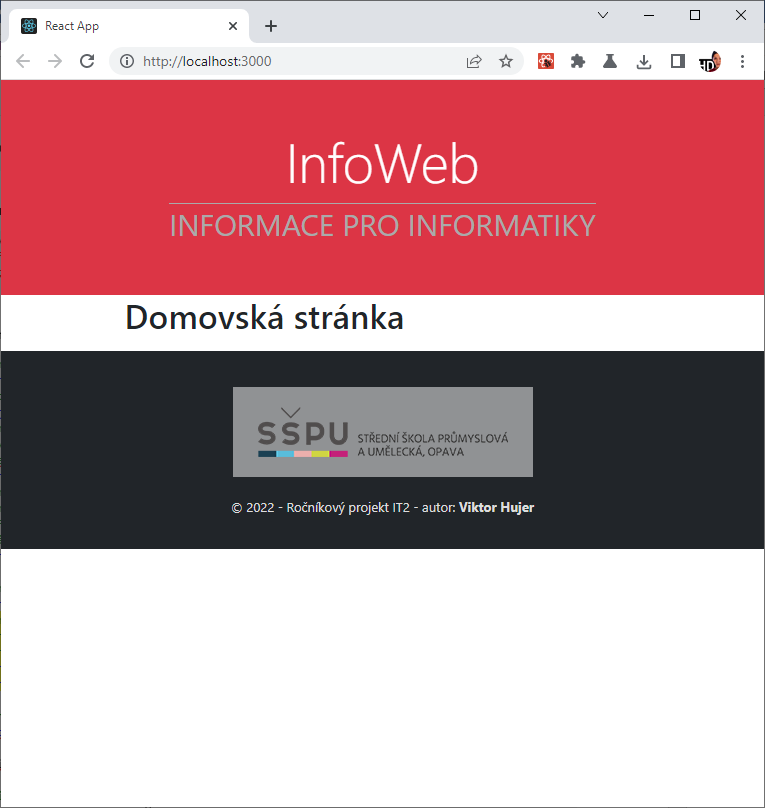
  );

}

/\* Exportování modulu App, aby mohl být naimportován v jiných modulech \*/

export default App;

Výsledkem je nová podoba úvodní stránky:



## Výpisy článků na webových stránkách

### Směrování mezi částmi webu

Než začneme na webových stránkách vypisovat údaje získané z backendového serveru, ukážeme si, jak funguje tzv. routing, neboli přesměrování mezi jednotlivými stránkami webu. Routování budeme řešit v souboru App.js, tedy v rámci základní komponenty celé aplikace. K již existující cestě, zajišťující zobrazení domovské stránky (komponenty Homepage), přidáme další dvě komponenty Route, které budou směrovat požadavky uživatelů webu na výpis všech článků a podrobné zobrazení jednoho konkrétního článku podle zadaného id.

Změněná část souboru **App.js** vypadá nyní takto:

    /\* Inicializace routeru - směrovače (viz https://v5.reactrouter.com/web/guides/primary-components) \*/

    <BrowserRouter>

      {/\* Inicializace služby Apollo s předáním nastavení klienta \*/}

      <ApolloProvider client={client}>

        {/\* Vložení komponenty Header, které předáváme atributy/props title a motto. (viz (viz https://reactjs.org/docs/components-and-props.html)) \*/}

        <Header title="InfoWeb" motto="Informace pro informatiky" />

        <main className="container">

          {/\* Nastavení routování - směrování požadavků na správné komponenty \*/}

          <Routes>

            {/\* URL adresa/cesta k domovské stránce \*/}

            <Route path="/" element={<Homepage />} />

            {/\* URL adresa/cesta k stránce zobrazující přehled článků \*/}

            <Route path="/articles" element={<Articles />} />

            {/\* URL k stránce zobrazující podrobné informace o jednom článku.

:id = parametr, který umožňuje předávat v URL id konkrétního článku. \*/}

            <Route path="/articles/:id" element={<ArticleDetail />} />

          </Routes>

        </main>

        {/\* Vložení komponenty Footer, které předáváme atributy/props logo a copyright.

            Soubor skola-logo.png musí být umístěn ve složce public, která je chápána jako základní složka webu (obsahuje statické soubory).

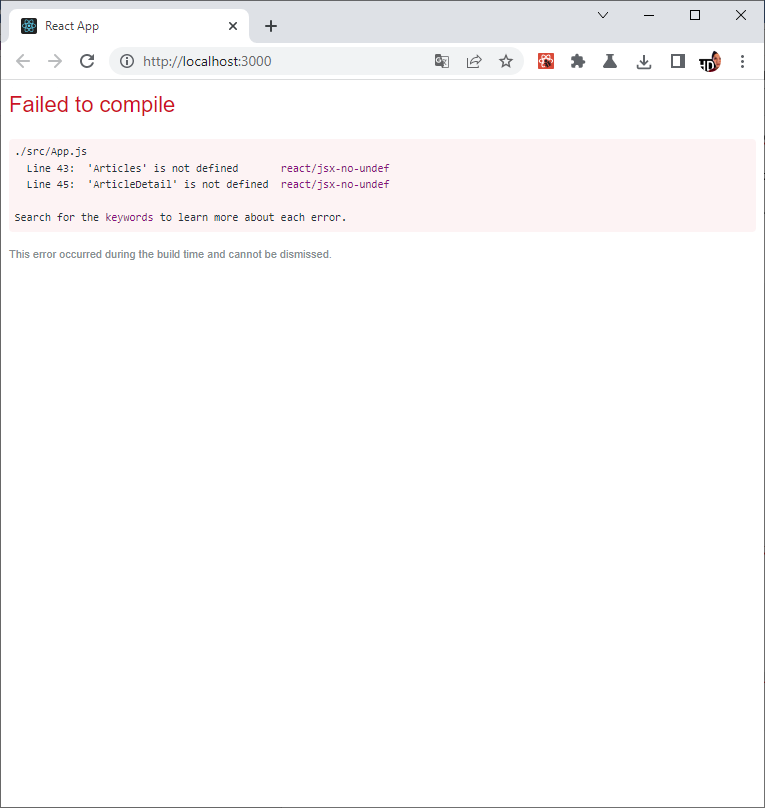
            Atribut/props copyright je v tomto případě předáván jako JS objekt, který má dvě vnitřní proměnné/atributy projectName a projectAuthor. \*/}

        <Footer logo="/skola-logo.png" copyright={{projectName: "Ročníkový projekt IT2", projectAuthor: "Viktor Hujer"}} />

      </ApolloProvider>

    </BrowserRouter>

Po této změně bude spuštěná aplikace hlásit chyby:



Je to logické, protože jsme přidali směrování na dvě komponenty (elementy), které dosud neexistují. Abychom to napravili, musíme je aspoň ve velmi jednoduché podobě vytvořit. Založíme a upravíme dva nové soubory ve složce pages - Articles.js a ArticleDetail.js. Můžeme si při tom pomoci vytvořením kopií souboru Homepage.js a jejich rychlou úpravou do následující podoby:

**Articles.js**

/\* Import základní knihovny Reactu \*/

import React from "react";

/\* Export funkce Articles(), která představuje komponentu pro výpis seznamu článků \*/

export default function Articles() {

  return (

    /\* Kód v jazyce JSX musí obsahovat kořenový element - v tomto případě oddíl div \*/

    <div>

      <h2>Seznam článků</h2>

      <p>Tady bude seznam článků</p>

    </div>

  );

}

**ArticleDetail.js**

/\* Import základní knihovny Reactu \*/

import React from "react";

/\* Import metody useParams z modulu react-router-dom. (viz https://reactrouter.com/docs/en/v6/hooks/use-params) \*/

import { useParams } from "react-router-dom";

/\* Export funkce ArticleDetail(), která představuje komponentu pro výpis podrobností o jednom článku. \*/

export default function Articles() {

  /\* Do konstanty id se uloží parametr zadaný na konec URL požadavku - číselné id vybraného článku. \*/

  /\* metodu useParams() označujeme v Reactu jako tzv. hook (viz https://www.w3schools.com/react/react\_hooks.asp) \*/

  const { id } = useParams();

  return (

    <div>

      {/\* V nadpisu se zatím zobrazí id předané v URL požadavku \*/}

      <h2>Výpis článku s id={ id }</h2>

      <p>Tady budou zobrazeny podrobnosti o jednom článku</p>

    </div>

  );

}

Ještě je nutné doplnit importy obou nových komponent v souboru App.js:

/\* Import komponent, které se nacházejí v jednotlivých modulech v podsložce pages \*/

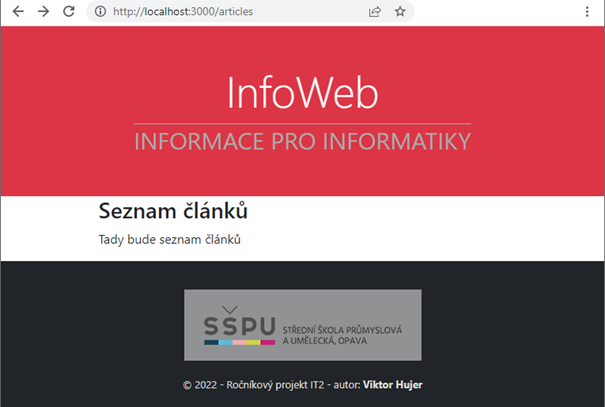
import Homepage from './pages/Homepage';

import Articles from './pages/Articles';

import ArticleDetail from './pages/ArticleDetail';

Pokud kompilace kódu proběhne bez chyb, můžeme ověřit funkční směrování (routování) postupným zadáním těchto URL adres v prohlížeči:

* <http://localhost:3000/articles>



* <http://localhost:3000/articles/5>



## Komponenta ArticlesGrid pro výpis seznamu článku

Pro výpis seznamu článků vytvoříme speciální komponentu **ArticlesGrid** v souboru *ArticlesGrid.js* umístěném do složky *components*.

Účelem této komponenty bude vypisovat články v responzivním zobrazení - tedy uspořádané do několika sloupců v případě zobrazení na širokoúhlém displeji nebo v jediném sloupci na displeji mobilního telefonu. K tomu využijeme externí knihovnu react-bootstrap a její komponenty Container, Row a Col.

### Získání dat pomocí knihovny Apollo

Ještě před samotným výpisem dat je musíme načíst z backendového serveru. Data získáme prostřednictvím graphQL API a nástrojů, které jsou součástí již námi nainstalované externí knihovny **@apollo-client**.

Nejprve si připravíme **graphQL dotaz**, kterým backend požádáme o předání dat o všech článcích. Funkčnost dotazu je dobré předem ověřit v prostředí **Playground**, které po instalaci pluginu GraphQL nabízí Strapi server. Odladěný bezchybný dotaz pošleme na backendový server hook metodou **useQuery()**, která vyžaduje minimálně jeden parametr - query QL dotaz. Metoda vrací tři objekty, zachycující tři možné stavy, k nimž může během provádění dotazu dojít:

* *loading* - v případě, že ještě probíhá komunikace se serverem;
* *error* - objekt obsahuje informace o případné chybě (ve formátu JSON), k níž došlo během operace;
* *data* - objekt obsahuje načtená data ve formátu JSON, která poté můžeme zpracovat.

### Ošetření možných chyb

V případě, že je zjištěna chyba, zobrazí se upozorňující zpráva (pomocí komponenty Alert) a připojí se podrobný výpis obsahu proměnné error (JSON.stringify(error) zajistí zřetězení všech údajů, které objekt obsahuje).

### Výpis pole objektů metodou map()

V případě úspěšného provedení dotazu můžeme podmínkou ověřit, zda databáze obsahuje aspoň jeden článek (*if (data.articles.data.length > 0)*). Jestliže ano, pokračujeme výpisem přehledu článků, když ne, zobrazí se jen alert s upozorněním, že žádný článek není k dispozici.

Protože data o všech článcích v databázi (*data.articles.data*) tvoří pole objektů, je výhodné pro jejich výpis využít metody map, která nám velmi efektivně nahradí použití klasického cyklu tím, že spojí každý objekt v poli (v našem případě article) s arrow funkcí, v níž je proveden údajů o jednotlivém článku v požadovaném formátu. V našem případě jsme opět využili několika připravených komponent z knihovny react-bootstrap (Row, Col, Figure) nebo běžných HTML tagů. Proměnné jsou v jazyce JSX vloženy mezi složené závorky.

Při použití metody map v Reactu bychom neměli zapomenout označit každý opakující se prvek jedinečnou hodnotou (ideálně primárním klíčem v podobě číselného id). Proto je součástí každého sloupce atribut **key** s přiřazenou hodnotou **article.id**.

### Podmíněné zobrazení obrázku

V některých případech musíme ošetřit možnost, že data pro danou část výpisu v databázi chybí. Můžeme pro tyto situace použít tzv. podmíněného vykreslení části stránky. Příkladem je podmíněné zobrazení doprovodného obrázku k článku:

{/\* V případě, že k článku existuje doprovodný obrázek ... \*/}

{article.attributes.image.data && (

    <Figure>

      {/\* ... použije se spojení adresy webu s cestou k danému obrázku, která je uložena v databázi backendu. \*/}

      <Figure.Image

        alt={article.attributes.title}

        src={`${process.env.REACT\_APP\_BACKEND\_URL}${article.attributes.image.data.attributes.url}`}

        rounded

      />

      <Figure.Caption>

        Obrázek: {article.attributes.title}

      </Figure.Caption>

    </Figure>

)}

Podmíněné zobrazení je zde zajištěno díky výrazu: {*(podmínka) &&* *(výpis)*} (výpis je proveden, když současně platí nějaká podmínka).

### Předání URL adresy serveru z konstanty uložené v soubou .env

Obrázky, případně další jiné přílohy, jsou na Strapi serveru uloženy ve speciální složce pod jedinečnými názvy a v databázi je zapsána pouze jejich relativní URL adresa bez adresy serveru, která se pochopitelně může měnit. Abychom u všech příloh v naší aplikaci nemuseli opakovaně psát adresu backendového serveru (např. <http://localhost:1337>) a v případě změny této adresy ji všude pracně přepisovat, vytvoříme si v kořenové složce aplikace soubor .env, do něhož můžeme uložit různé konstanty vztahující se k "prostředí" (environment) naší aplikace. Pro naše potřeby do něj zatím můžeme zapsat konstantu s adresou lokálního Strapi serveru:

#.env file

REACT\_APP\_BACKEND\_URL=http://localhost:1337

Aby aplikace dokázala tento soubor správně využívat, musíme nainstalovat externí knihovnu dotenv:

**npm install dotenv**

Poté ji ještě připojíme v souboru index.js:

import React from 'react';

import ReactDOM from 'react-dom/client';

import './index.css';

import App from './App';

require('dotenv').config()

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

  <React.StrictMode>

    <App />

  </React.StrictMode>

);

V rámci celé aplikace pak můžeme ke konstantám uloženým v souboru .env přistupovat pomocí **process.env**.KONSTANTA.

Konkrétním příkladem použití námi uložené konstanty je vytvoření absolutní adresy k obrázku:

<Image

src={`${process.env.REACT\_APP\_BACKEND\_URL}${article.attributes.image.data.attributes.url}`}

/>

### Kompletní kód komponenty ArticlesGrid

/\* Import základní knihovny Reactu \*/

import React from "react";

/\* Import použitých komponent z knihovny react-bootstrap \*/

import { Container, Row, Col, Alert, Figure, Button } from "react-bootstrap";

/\* Import nástrojů pro práci s graphQL z knihovny @apollo/client (viz https://www.apollographql.com/docs/react/get-started/) \*/

import { gql, useQuery } from "@apollo/client";

/\* Konstanta s vytvořeným graphQL dotazem, pomocí něhož získáme informace o všech uložených článcích. \*/

const ARTICLES = gql`

  query Articles {

    articles {

      data {

        id

        attributes {

          title

          content

          likes

          dislikes

          releasedAt

          categories {

            data {

              id

              attributes {

                name

                shortname

                icon {

                  data {

                    attributes {

                      url

                    }

                  }

                }

              }

            }

          }

          image {

            data {

              attributes {

                url

              }

            }

          }

        }

      }

    }

  }

`;

/\* Komponenta ArticleGrid pro zobrazení seznamu článků v rámci Bootstrap gridu \*/

export const ArticlesGrid = () => {

  /\* Odeslání graphQL dotazu pomocí hook funkce useQuery()

     Výsledkem mohou být tři stavy vyjádřené v konstantách loading, error, data \*/

  const { loading, error, data } = useQuery(ARTICLES);

  /\* Situace, kdy jsou načítána data z backend serveru. \*/

  if (loading) return <p>Probíhá načítání stránky...</p>;

  /\* Situace, kdy došlo k chybě během načítání dat. \*/

  if (error) return (

    <Container>

      <Alert variant="danger">Došlo k chybě: {JSON.stringify(error)}</Alert>

    </Container>

  );

  /\* Dotaz byl úspěšně proveden a v odeslaných datech je aspoň 1 článek: \*/

  if (data.articles.data.length > 0)

    return (

      <Container fluid>

        <Row sm={1} md={2} lg={3}>

          {/\* K postupnému výpisu všech datových objektů v poli lze efektivně použít metodu map ( viz https://www.w3schools.com/js/js\_es5.asp#mark\_array\_map).

                Jednotlivé objekty v poli jsou postupně předány do proměnné article a pomocí ní lze přistupovat k dílčím datovým atributům. \*/}

          {data.articles.data.map((article) => (

            /\* Každý opakující se prvek by měl obsahovat atribut key s přiřazeným jedinečným klíčem k datovému objektu. \*/

            <Col key={article.id}>

              <div className="border p-2 m-2">

                {/\* Zobrazení titulku článku v nadpisu. \*/}

                <h3>{article.attributes.title}</h3>

                {/\* V případě, že k článku existuje doprovodný obrázek ... \*/}

                {article.attributes.image.data && (

                  <Figure>

                    {/\* ... použije se spojení adresy webu s cestou k danému obrázku, která je uložena v databázi backendu. \*/}

                    <Figure.Image

                      alt={article.attributes.title}

                      src={`${process.env.REACT\_APP\_BACKEND\_URL}${article.attributes.image.data.attributes.url}`}

                      rounded

                    />

                    <Figure.Caption>

                      Obrázek: {article.attributes.title}

                    </Figure.Caption>

                  </Figure>

                )}

                {/\* V případě, že k článku není žádný doprovodný obrázek ... \*/}

                {!article.attributes.image.data && (

                  <Figure>

                    {/\* ... zobrazí se náhrada v podobě školního loga \*/}

                    <Figure.Image

                      alt={article.attributes.title}

                      src="/skola-logo.png"

                      rounded

                    />

                    <Figure.Caption>Obrázek: Není k dispozici</Figure.Caption>

                  </Figure>

                )}

                {/\* Zobrazí se 100 prvních znaků z obsahu článku. \*/}

                <p>{article.attributes.content.substring(0, 100)}...</p>

                {/\* Komponenta Button umožní zobrazení stránky s podrobnostmi k danému článku. \*/}

                <Button

                  variant="outline-primary"

                  href={`/articles/${article.id}`}

                >

                  Podrobnosti

                </Button>

              </div>

            </Col>

          ))}

        </Row>

      </Container>

    );

  /\* Dotaz byl úspěšně proveden, ale v odeslaných datech není žádný článek: \*/ else

    return (

      <Container>

        <Alert variant="warning">Nebyl nalezen žádný článek</Alert>

      </Container>

    );

};

## Komponenta ArticleDetail pro výpis jednoho článku

Úlohou komponenty ArticleDetail umístěné ve složce pages je provádět úplný výpis informací o jednom konkrétním článku. Opakují se v ní některé části, které už známe z řešení v komponentě ArticlesGrid včetně vytvoření graphQL dotazu a jeho odeslání pomocí metody useQuery() pocházející z knihovny @apollo-client.

Zásadní je v tomto případě předání identifikace článku pomocí primárního klíče - id. Dotazu, uloženému v konstantně ARTICLE, musíme v tomto případě předat proměnnou:

const { loading, error, data } = useQuery(ARTICLE, { variables: { id: id } });

### Úplný kód komponenty ArticleDetail

/\* Import základní knihovny Reactu \*/

import React from "react";

/\* Import metody useParams z modulu react-router-dom. (viz https://reactrouter.com/docs/en/v6/hooks/use-params) \*/

import { useParams } from "react-router-dom";

/\* Import použitých komponent z knihovny react-bootstrap \*/

import { Row, Col, Figure, ListGroup } from "react-bootstrap";

/\* Import nástrojů pro práci s graphQL z knihovny @apollo/client (viz https://www.apollographql.com/docs/react/get-started/) \*/

import { gql, useQuery } from "@apollo/client";

import { ReactMarkdown } from "react-markdown/lib/react-markdown";

import remarkGfm from "remark-gfm";

/\* Konstanta s vytvořeným graphQL dotazem, pomocí něhož získáme informace o všech uložených článcích. \*/

const ARTICLE = gql`

  query Article($id: ID!) {

    article(id: $id) {

      data {

        id

        attributes {

          title

          content

          likes

          dislikes

          releasedAt

          categories {

            data {

              id

              attributes {

                name

                shortname

              }

            }

          }

          image {

            data {

              attributes {

                url

              }

            }

          }

          users\_permissions\_user {

            data {

              attributes {

                username

                email

              }

            }

          }

        }

      }

    }

  }

`;

/\* Export funkce ArticleDetail(), která představuje komponentu pro výpis podrobností o jednom článku. \*/

export default function Articles() {

  /\* Do konstanty id se uloží parametr zadaný na konec URL požadavku - číselné id vybraného článku. \*/

  /\* metodu useParams() označujeme v Reactu jako tzv. hook (viz https://www.w3schools.com/react/react\_hooks.asp) \*/

  const { id } = useParams();

  /\* Odeslání graphQL dotazu pomocí hook funkce useQuery()

     Výsledkem mohou být tři stavy vyjádřené v konstantách loading, error, data \*/

  const { loading, error, data } = useQuery(ARTICLE, { variables: { id: id } });

  /\* Situace, kdy jsou načítána data z backend serveru. \*/

  if (loading) return <p>Probíhá načítání stránky...</p>;

  /\* Situace, kdy došlo k chybě během načítání dat. \*/

  if (error) return <p>Došlo k chybě: {JSON.stringify(error)}</p>;

  /\* Data jsou úspěšně načtena a uložena do konstanty article \*/

  const article = data.article.data;

  /\* Tímto způsobem si můžeme do konstanty realeaseDate připravit datum publikování článku tak, aby se zobrazovalo v obvyklé "české" podobě.

     Metoda split() je použita k tomu, aby z řetězce "ROK-MĚSÍC-DEN" vytvořila pole [ROK, MĚSÍC, DEN], z něhož je vytvořen nový datový objekt.

     Metoda toLocaleDateString() vypíše datum v podobě, které odpovídá místním zvyklostem (dle nastavení OS). \*/

  const releasedDate = new Date(article.attributes.releasedAt.split("-")).toLocaleDateString();

  return (

    <Row>

      {/\* V nadpisu se zobrazí titulek \*/}

      <h2 className="text-danger bg-light p-3 m-3 text-center">

        {article.attributes.title}

      </h2>

      {/\* V odstavci pod nadpisem se zobrazí dostupné informace o autorovi článku (nick a email) a připojí se připravené datum publikování. \*/}

      <p className="small text-center text-secondary mb-4">

        Autor:{" "}

        <b>

          {article.attributes.users\_permissions\_user.data.attributes.username}

        </b>{" "}

        ({article.attributes.users\_permissions\_user.data.attributes.email}),

        publikováno: <b>{releasedDate}</b>

      </p>

      <Col>

        {/\* Protože bylo k formátování obsahu článku použito ve Strapi RichText editoru umožňujícího úpravu textu s využitím značek formátu MD (MarkDown),

            který je známý např. z gitu (README.md), k správnému zobrazování upravených částí textu použijeme externí komponentu ReactMarkdown a plugin

            remarkGfm (obojí je třeba nejprve nainstalovat: npm install react-markdown remark-gfm). Nesmíme je zapomenout importovat. \*/}

        <ReactMarkdown

          children={article.attributes.content}

          remarkPlugins={[remarkGfm]}

        />

      </Col>

      <Col>

        {/\* V případě, že k článku existuje doprovodný obrázek ... \*/}

        {article.attributes.image.data && (

          <Figure>

            {/\* ... použije se spojení adresy webu uložené v souboru .env v konstantě REACT\_APP\_BACKEND\_URL s cestou k danému obrázku, která je uložena v databázi backendu. \*/}

            <Figure.Image

              alt={article.attributes.title}

              src={`${process.env.REACT\_APP\_BACKEND\_URL}${article.attributes.image.data.attributes.url}`}

              rounded

            />

            {/\* K popisu pod obrázkem použijeme titulek článku. \*/}

            <Figure.Caption>Obrázek: {article.attributes.title}</Figure.Caption>

          </Figure>

        )}

        <h4 className="bg-danger text-white p-3">Předměty</h4>

        {/\* React-bootstrap komponentu ListGroup (viz https://react-bootstrap.github.io/components/list-group/) použijeme k vypsání všech předmětů/kategorií,

            do nichž byl článek zařazen. \*/}

        <ListGroup variant="flush">

          {/\* Opět k tomu využijeme metody map, díky níž jsou opakovaně vloženy položky seznamu s názvy kategorií/předmětů. \*/}

          { article.attributes.categories.data.map(category => (

              <ListGroup.Item>{category.attributes.name}</ListGroup.Item>

            ))

          }

        </ListGroup>

      </Col>

    </Row>

  );

}