Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Dotazovací jazyky pro webové API

2023

Bc. Petr Unzeitig

Vedoucí práce:
Mgr. Radek Janoštík, Ph.D.

Studijní program: Aplikovaná informatika,
Specializace: Vývoj software

Bibliografické údaje

Autor:	Bc. Petr Unzeitig
Název práce:	Dotazovací jazyky pro webové API
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2023
Studijní program:	Aplikovaná informatika, Specializace: Vývoj software
Vedoucí práce:	Mgr. Radek Janoštík, Ph.D.
Počet stran:	91
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Bc. Petr Unzeitig
Title:	Web API Query Languages
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2023
Study program:	Applied Computer Science, Specialization: Software Development
Supervisor:	Mgr. Radek Janoštík, Ph.D.
Page count:	91
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

V rámci této diplomové práce byly zkoumány a srovnávány nejrozšířenější dotazovací jazyky pro webové API: OData, HotChocolate a JSON:API, včetně jejich implementace na platformě .NET. Dále byly analyzovány a srovnány architektury REST a GraphQL. Cílem práce bylo poskytnout hluboké porozumění těmto technologiím a srovnat jejich funkčnost a výkonnost v různých testovacích scénářích.

Synopsis

Within this thesis, the most widespread query languages for web APIs: OData, HotChocolate, and JSON:API were examined and compared, along with their implementations on the .NET platform. Additionally, the architectures of REST and GraphQL were analyzed and contrasted. The aim of the study was to provide profound understanding of these technologies and compare their functionality and performance across various testing scenarios.

Klíčová slova: dotazovací jazyky; API; REST; GraphQL

Keywords: query languages; API; REST; GraphQL

Děkuji vedoucímu této diplomové práce panu Mgr. Radku Janoštíkovi Ph.D. za odbornou pomoc a cenné rady. Děkuji také mé rodině a blízkým za trpělivost a podporu.

Odevzdáním tohoto textu jeho autor/ka místopřísežně prohlašuje, že celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

Úvod

Vývoj moderních webových aplikací využívajících webová API klade stále větší nároky na flexibilitu, efektivitu a rychlost získávání dat. Proto se stává klíčovým faktorem, jak rychle a efektivně jsou tato API schopna zpracovávat požadavky. Pro zajištění tohoto cíle je důležité vybrat vhodný dotazovací jazyk a API architekturu, která bude použita pro návrh a implementaci webového API. Tato diplomová práce se zaměřuje na nejrozšířenější dotazovací jazyky pro získávání dat pomocí webových API, jako jsou OData, Hotchocolate a JSON:API. Popisuje jejich vlastnosti a srovnává je na základě výkonnostních testů při implementaci na platformě .NET v kontextu dotazování dat. Dále se práce věnuje srovnání nejpoužívanějších API architektur, jako jsou REST a GraphQL, včetně jejich implementace na platformě .NET. Cílem práce je poskytnout komplexní přehled o nejvhodnějších technologiích pro návrh a implementaci webových API, včetně doporučení pro vývojáře při výběru nejvhodnějšího řešení pro konkrétní projekty.

1 Webové API

Termín Application Programming Interface (API) je rozhraní mezi počítačovými systémy nebo mezi různými programy v existujícím systému. Tyto systémy byly často vyvíjeny jako celek, přičemž žádný z nich nebyl označován jako server nebo klient. Například mail server může použít databázi k uložení informací, ale oba systémy byly navrženy společně, aby byly výhradně propojeny, a aby spolu bezproblémově spolupracovali.

V případě webových API je specifikován klient, což může být cokoli od webového prohlížeče po mobilní aplikaci. Klient kontaktuje webový server, přičemž s daty se dále pracuje na tomto serveru, a pokud je potřeba, data jsou vrácena zpět na klienta. Hlavní rozdíl je v tom, že vývojáři, kteří vyvíjí klienty, nejsou často stejní jako ti, kteří píší rozhraní – systémy jsou oddělené.

Při návrhu API se většinou neklade velký důraz na program, který toto rozhraní poskytuje, neřeší se tedy business logika. Spíše jde o to, aby bylo rozhraní pochopitelné pro lidi, kteří ho využívají pro psaní jiných programů nebo konkrétně klientských aplikací.

API je zodpovědné například za to, jak jsou předpovědi počasí sdíleny z nějakého důvěryhodného zdroje, jako je Národní meteorologická služba pro stovky softwarových aplikací, které tyto předpovědi prezentují. Bankovní společnosti využívají API k tomu, aby umožnili svým uživatelům zpracovávat kreditní karty a bezproblémově tak vybírat a vkládat peníze, aniž by se museli starat o detaily finančních technologií, jejich zákonů a předpisů.

API jsou stále více klíčovou součástí pro škálovatelné aplikace úspěšných internetových společností, jako jsou Google, Amazon, Netflix, Facebook a další. Je tak důležitou složkou pro společnosti, které se snaží vytvořit obchodní platformu rozšiřující se na globální trh.

K pochopení principu fungovaní webových API je nezbytné popsat také protokol – způsob, kterým spolu systémy komunikují.[1]

2 HTTP

Hypertext Transfer Protocol (HTTP) je protokol určený primárně ke komunikaci v internetové síti. Miliardy JPEG obrázků, HTML stránek, WMV videí, MP3 zvukových nahrávek, JSON či XML dokumentů a mnoho dalšího proplouvají Internetem každý den.

HTTP protokol přesouvá množství těchto informací rychle a spolehlivě z webových serverů po celém světě do webových prohlížečů a různých klientských aplikací.

Tento protokol je užitečný díky tomu, že poskytuje standardizovaný způsob vzájemné komunikace mezi počítači. Specifikuje, jak klienti data požadují, a jak servery na tyto požadavky reagují.

HTTPS

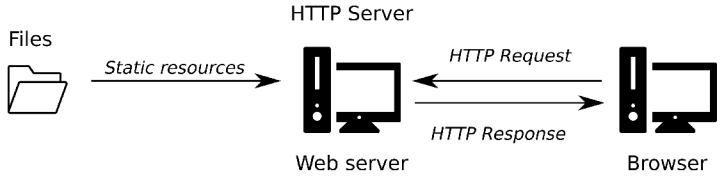
Hypertext Transfer Protocol Secure (HTTPS) je protokol umožňující zabezpečenou komunikaci v počítačové síti.

Protokol HTTP není chráněn před odposloucháváním nebo modifikací obsahu. Nezaručuje tak, že obsah, který klient obdrží, je ten obsah, který byl vytvořen na příslušném serveru.

Tyto problémy řeší protokol HTTPS. Při jeho použití dochází před odesláním každého požadavku a každé odpovědi přes síť k šifrování obsahu. Poskytuje kryptografickou bezpečnostní vrstvu na transportní úrovni pomocí certifikátu Transport Layer Security (TLS) nebo Secure Sockets Layer (SSL). [2]

2.1 Webový server a klient

Webový obsah se nachází na webových serverech. Tyto servery jsou schopny data ukládat a poskytovat vždy, když to klienti požadují. Klienti odesílají serverům HTTP požadavky a servery vracejí požadovaná data v HTTP odpovědích tak, jak je zobrazeno na obrázku 1. Společně tvoří základní komponenty celosvětové služby World Wide Web.



Obrázek 1: Architektura statického serveru [3]

Běžný uživatel internetu používá HTTP klienta každý den, a sice internetový prohlížeč. Webové prohlížeče posílají požadavky na HTTP zdroje a zobrazují je na uživatelských obrazovkách.

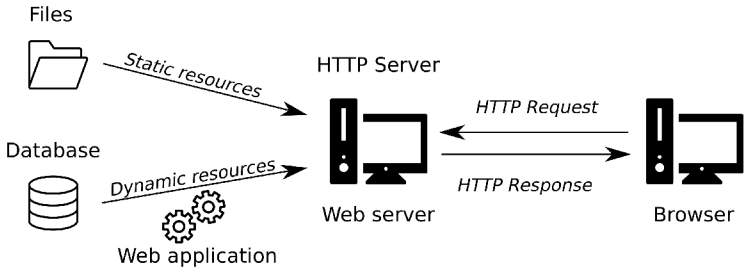
Jakmile se uživatel pokusí procházet například internetovou stránku *www.muj-web.cz/index.html*, jeho prohlížeč odešle HTTP požadavek na server *www.muj-web.cz*. Server se pokusí najít požadovaný objekt, zde konkrétně dokument *index.html*. Pokud ho najde, odešle ho v HTTP odpovědi společně s jeho informacemi (typ odesílaného dokumentu, jeho velikost a další) zpět na klienta.

Často se HTML stránky, obrázky, videa a podobné HTTP zdroje nacházejí přímo na webovém serveru, a ten tak nemusí požadovaná data generovat, jen je „jednoduše“ vrátí uživateli přesně tak, jak jsou uložena. Jelikož se jedná o statické soubory, jde o přístup statický.

Zdroje však nemusejí být jen statická data. Mohou to být i softwarové programy, které generují obsah na vyžádání. Mohou generovat obsah na základě identity uživatele, informací, které požadoval nebo i na základě denní doby. Umožňují zobrazit živé fotografie z kamer, prohledávat různé databáze nebo nakupovat v internetových obchodech.

Pokud tedy musí webový server při plnění HTTP požadavku vykonávat procesy podnikové logiky a dynamicky generovat data, jedná se o přístup dynamický.

Na obrázku 2 je ukázána architektura webového serveru, který je schopen generovat data z databáze.



Obrázek 2: Architektura dynamického serveru [3]

Jelikož se při tomto přístupu nenacházejí požadovaná data přímo na serveru, většinou už nestačí pouze odeslat požadavek ve tvaru */index.html*. Vývojář již musí znát konkrétní tvar požadavku, který je definován ve webovém rozhraní API konkrétního serveru. [2]

2.2 Typ internetového média

Jelikož jsou na Internetu sdíleny objekty tisíce různých datových typů, HTTP označí každý objekt, který je přenášen přes web speciálním štítkem. Tento štítek se nazývá Multipurpose Internet Mail Extensions (MIME) a označuje datový formát daného objektu. Tento způsob byl původně navržen k řešení problémů,

které se vyskytují při přesouvání zpráv mezi různými systémy elektronické pošty. Avšak MIME fungoval pro e-mail tak dobře, že jej HTTP přijal k popisu a označení vlastního multimediálního obsahu.

Hodnota vlastnosti *Content-type* v HTTP odpovědi označuje informaci pro server, jaký je skutečný typ vráceného objektu. V HTTP požadavku jde o informaci pro klienta, jaký typ dat je skutečně odesílán.

Webové servery připojují štítek MIME ke všem datovým objektům přenášeným přes protokol HTTP. Jakmile webový prohlížeč příjme data ze serveru, podívá se na připojený MIME typ, aby zjistil, jestli umí s objektem takového typu zacházet a jestli je schopen ho dekódovat. Většina prohlížečů dokáže zpracovat stovky oblíbených typů objektů: zobrazení obrazových souborů, analýzu a formátování souborů HTML, přehrávání zvukových souborů přes reproduktory počítače nebo spouštění externího zásuvného softwaru pro práci se speciálními formáty.

MIME typ je textový štítek skládající se ze dvou částí. První část označuje primární typ objektu a druhá část vyjadřuje specifický typ. Tyto dvě části jsou oddělené lomítkem. Tabulka 1 obsahuje vybrané typy internetových medií. [2]

Tabulka 1: Typy internetových médií

štítek	popis
text/plain	prostý text
image/jpeg	obrázek ve fomátu JPEG
image/gif	obrázek ve fomátu GIF
application/json	dokument ve formátu JSON
multipart/form-data	data vyplněného formuláře

2.3 URI

Uniform Resource Identifier (URI) označuje název zdroje na serveru. Každý zdroj, který se nachází na webovém serveru má svůj název, aby byl pro klienty rozeznatelný. Klienti tak mohou odesílat požadavky o konkrétní zdroje, o které mají zájem. URI jsou něco jako poštovní adresy internetu, jedinečně identifikují a lokalizují informační zdroje po celém světě.

Takto může například vypadat URI adresa obrázku, který se nachází na webu *www.muj-web.cz*:

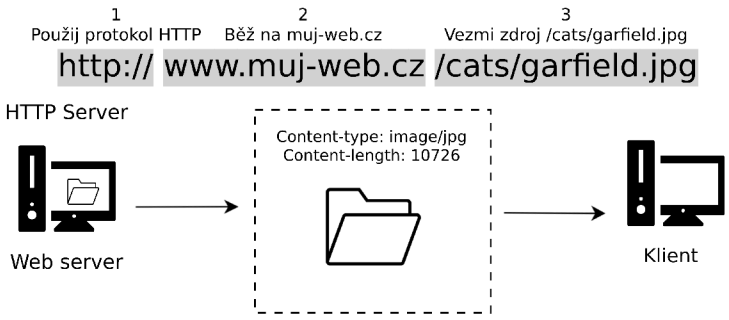
http://www.muj-web.cz/cats/garfield.jpg

Obrázek 3 ukazuje, jak URI specifikuje protokol HTTP pro přístup ke zdroji garfield.jpg na serveru s název muj-web.

URI existuje v několika variantách, které se nazývají URL, URN nebo URC.

URL

Uniform Resource Locator (URL) je nejběžnější formou identifikátoru internetových zdrojů. Adresy URL popisují konkrétní umístění zdroje na konkrétním serveru. Říkají přesně, jak získat zdroj ze vzdáleného fyzického uložiště. Tabulka 2 obsahuje několik příkladů URL adres.



Obrázek 3: Použití URL adresy

Většina adres URL má standardizovaný formát, který se skládá ze třech hlavních částí:

1. protokol používaný pro přístup ke zdroji. Tato část je často nazývaná, jako schéma (např. http://),
2. internetová adresa serveru (např. www.muj-web.cz),
3. cesta ke zdroji na webovém serveru (např. /cats/garfield.jpg).

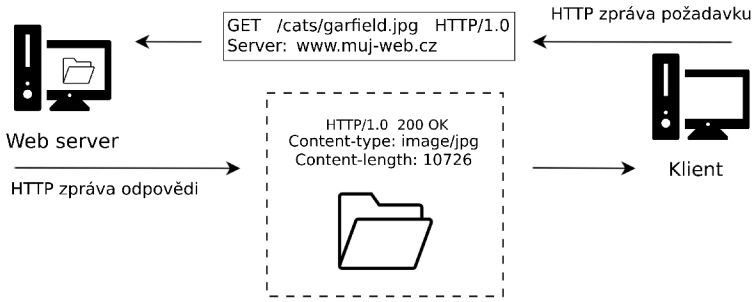
URL adresy, které slouží pro získání zdrojů z dynamických webový serverů, mohou navíc obsahovat čtvrtou část. Tato další část se nazývá řetězec dotazu (z anglického query string). Následuje těsně za cestou ke zdroji a skládá se z textové řetězce začínajícího otazníkem. Slouží serveru pro různé účely, nejčastěji jako parametr pro vyhledávání. Řetězec dotazu se obvykle skládá z dvojic jmen a hodnot, například *jmeno=garfield*. Páry jsou od sebe odděleny znakem ampersand, například *jmeno=garfield&barva=oranzova*. [2]

Tabulka 2: Příklady URL adres

URL	popis
http://muj-web.cz/index.html	Domovská stránka webu muj-web.cz.
http://muj-web.cz/images/logo.png	Logo webu muj-web.cz ve formátu PNG.
http://muj-web.cz/cats/garfield	URL pro program generující dokument s informacemi o kočce Garfield.
http://muj-web.cz/cats?barva=bila	URL pro program, který vrací informace o kočkách, které mají bílou barvu.

2.4 Transakce

Klienti komunikují s webovými servery pomocí transakcí. HTTP transakce se skládají z požadavku, který je odesílán z klienta na server a také z výsledku odpovědi, který je odesílán ze serveru zpět na klienta. Tato komunikace probíhá ve formátovaných blocích dat nazývajících se HTTP zprávy tak, jak je znázorněno na obrázku 4.



Obrázek 4: HTTP transakce

Metody

HTTP protokol definuje sadu metod požadavku k označení prováděné akce. Každá HTTP zpráva požadavku obsahuje jednu z metod, která řekne serveru, jakou akci má provést (načíst webovou stránku, spustit program, odstranit soubor atd.). Tabulka 3 uvádí nejběžnější HTTP metody.

Tabulka 3: HTTP metody

HTTP metoda	popis
GET	požadavek o konkrétní zdroj
HEAD	požadavek pouze o metadata konkrétního zdroje (velikost, typ, ...)
POST	odesílá uživatelská data na server, například vyplněné formuláře
PUT	požadavek o uložení dat na server
DELETE	požadavek o smazání konkrétního objektu ze serveru

Stavové kódy

Každá HTTP zpráva odpovědi se vrací zpět na klienta se stavovým kódem. Stavový kód je třímístný číselný kód, který klientovi sděluje, zda byl požadavek úspěšný nebo jestli je potřeba nějaké další akce. Tabulka 4 uvádí několik běžných stavových kódů.

Tabulka 4: HTTP stavové kódy

kód	název	popis
200	OK	dokument byl vrácen úspěšně
204	Žádný obsah	požadavek byl proveden úspěšně
400	Neplatný požadavek	server nedokáže zpracovat přijatý požadavek
401	Neautorizovaný	klient se musí nejdříve autentizovat
404	Nenalezen	požadovaný zdroj nebyl nalezen

Zprávy

HTTP zprávy jsou jednoduché sekvence znaků, které jsou uspořádány do řádků. Jelikož se jedná o prostý text, nikoli o binární, je pro člověka snadno čitelný. Obrázek 5 ukazuje zprávy jednoduché HTTP transakce.

Obrázek 5: HTTP zprávy

Zprávy odesílané z webových klientů na webové servery se nazývají HTTP zprávy požadavku. V opačném případě se jedná o HTTP zprávy odpovědi. Formáty obou těchto zpráv jsou si velmi podobné. Každá HTTP zpráva se skládá ze tří částí:

- počáteční řádek: první řádek zprávy označuje, co se má udělat k provedení požadavku nebo informaci, co se stalo s požadavkem k provedení,
- hlavička: za počátečním řádkem následují volitelné položky hlavičky. Každá taková položka se skládá z názvu a hodnoty oddělených dvojtečkou. Hlavička končí prázdným řádkem,
- tělo: po prázdném řádku následuje nepovinné tělo HTTP zprávy obsahující libovolný druh dat. Těla požadavků představují data přenášená na server. Těla odpovědí přenášejí data zpět na klienta. Na rozdíl od předchozích dvou částí HTTP zprávy, tělo může obsahovat i jiná data, než jen prostý text. Může obsahovat libovolná binární data (např. obrázky, zvukové stopy, json dokumenty). [2]

3 API Paradigmata

Webové API, jak je vysvětleno v kapitole 1, jsou rozhraní, určené webovým klientům pro získávání dat ze serverů přes protokol HTTP. To, jak bude toto rozhraní vypadat, určuje zvolené API paradigma, které je v různých literaturách často označované, jako architektonický styl API.

Jedná se o definovaný obecný přístup k vytváření API. Nejde o konkrétní nástroj nebo specifikaci, ale spíše o koncept. Určuje mimo jiné tvar požadavků, které jsou serverem přijímány a také to, jakou strukturu mají odpovědi.

Paradigmata jsou rozdělena do několika kategorií, ale tato práce se zabývá jen tou nejtypičtější zvanou Request–Response. API založené na této kategorii definují jeden či sadu koncových bodů. Klienti odesílají požadavky na tyto koncové body a servery vracejí odpovědi. Odpovědi jsou typicky vráceny ve formátu JSON nebo XML. Existuje několik běžných paradigmat tohoto typu: SAOP, RPC, REST a GraphQL.

RPC

Remote Procedure Call (RPC) je jedním z nejsnazších paradigmat. Jedná se o styl založený na akcích. Umožňuje klientovi spustit blok kódu na vzdáleném serveru. Klienti obvykle předají serveru název metody s argumenty a obdrží zpět JSON nebo XML. RPC API se obecně řídí dvěma jednoduchými pravidly:

- koncové body obsahují název operace, která má být provedena,
- pro volání API se používají HTTP metody následovně: GET pro požadavky pouze pro čtení a POST pro ostatní.

Celý proces vykonání požadavku začíná tak, že klient vyvolá vzdálenou proceduru. Serializuje parametry a další informace do zprávy, kterou následně odešle na server. Po přijetí zprávy server deserializuje její obsah, provede požadovanou operaci a odešle výsledek zpět klientovi.[4]

SOAP

Simple Object Access Protocol (SOAP) je paradigma založené na XML formátu, který se používá pro výměnu informací mezi systémy. Podobně jako RPC se jedná o styl založený za akcích. SOAP zapouzdřuje volání metody do zprávy, která se skládá z XML řetězců a odesílá ji přes protokol HTTP. Pro požadavky zasílané na server využívá SOAP jen HTTP metodu POST.

SAOP zpráva se skládá z tzv. obálky, která je reprezentovaná elementem *<Envelope>*. Obálka obsahuje následující tři elementy:

- *<Header>* je volitelný element, který se používá k předání informací souvisejících s aplikací, které mají být zpracovány (údaje pro ověření, platební údaje apod.),

- *<Body>* je povinný element, který obsahuje hlavní data určená pro příjemce,
- *<Fault>* je dílčí element elementu <Body>. Je určený pro hlášení chyb. Může obsahovat informace, jako jsou stavový kód, textový detail chyby atd.

3.1 REST

Representational State Transfer (REST) je API architektonický styl, který se při vývoji webových služeb řídí řadou pravidel. Byl představen jako nástupce SOAP API. Na rozdíl od SOAP není REST API omezeno na formát XML a může zpracovávat více datových formátů v závislosti na tom, co je potřeba. Mezi datové formáty podporované rozhraním REST API patří JSON, XML a YAML.

Rozhraní API, která jsou vytvořena na základě stylu REST, jsou často označována jako systémy RESTful. Vyznačují se bezstavostí a tím, že jsou systémově nezávislé.

Když klient zavolá rozhraní REST API, server přenese prostředky ve standardizované reprezentaci. Fungují tak, že vracejí informace o zdroji, který byl požadován – a jsou přeloženy do interpretovatelného formátu.

Při práci s daty používá REST API metody HTTP k provádění operací CRUD (Create, Read, Update a Delete).

3.1.1 Práce s REST API

Následující příklady zobrazují HTTP požadavky a odpovědi určené pro správu databáze knih. Příklady jsou směřovány pro server, který je postaven na architektonickém stylu REST, jedná se o tzv. RESTful API. V prvním řádku každého požadavku je klíčové slovo HTTP metody (GET, POST atd.) a část URI, která popisuje zdroj dat a číslo verze API. V prvním řádku každé odpovědi je klíčové slovo HTTP následované verzí toho protokolu a také stavový kód vyjadřující úspěšnost požadavku.

POST

Jak už bylo zmíněno v kapitole HTTP Transakce 2.4, metoda POST slouží k odeslání uživatelských dat na server. REST využívá tuto metodu k vytvoření nového prvku a jeho uložení například v databázi.

Tabulka 5: REST: vytvoření nového prvku

Požadavek	Odpověď

POST /api/2/books
Content-Type: application/json
 	
{
"title": "Algorithms to Live By"
"isbn": "1627790365",
"author": "Brian Christian",
"year": ": 2017
}
	HTTP/1.1 201 Created
	Content-Type: application/json
	Location: /api/2/books/3
	
	{
	"Id": "3"
	"title": "Algorithms to Live By"
	"isbn": "1627790365",
	"author": "Brian Christian",
	"year": ": 2017
	}

Příklad je uváděn s předpokladem, že se při vytvoření nového prvku na serveru přiřadilo tomuto prvku automaticky vygenerované identifikační číslo. Proto přibyl nový atribut s název Id v těle HTTP odpovědi.

Navíc má REST definované doporučení při vytváření nových zdrojů: zahrnout hlavičku *Location*, ukazující na cestu k nově vytvořenému objektu.

GET - všechny prvky

Pro získání všech knih uložených v databázi se použije HTTP metoda GET v následujícím tvaru.

Tabulka 6: REST: získání všech prvků

Požadavek	Odpověď

GET /api/2/books	HTTP/1.1 200 OK
	Content–Type: application/json
	
	[{
	"Id": "1"
	"title": "C# Data Structures and Algorithms"
	"isbn": "1788833732",
	"author": "Marcin Jamro",
	"year": ": 2023
	},
	{
	"Id": "2"
	"title": "Pro ASP.NET Core 6"
	"isbn": "1484279565",
	"author": "Adam Freeman",
	"year": ": 2022
	},
	{
	"Id": "3"
	"title": "Algorithms to Live By"
	"isbn": "1627790365",
	"author": "Brian Christian",
	"year": ": 2017
	}]

GET - konkrétní prvek

Pro získání jedné knihy se také použije metoda GET, avšak s přidaným identifikačním číslem konkrétní knihy, kterou chceme získat.

Tabulka 7: REST: získání konkrétního prvku

Požadavek	Odpověď

`GET /api/2/books/3`	`HTTP/1.1 200 OK`
	`Content-Type: application/json`
	
	`{`
	`"Id": "3"`
	`"title": "Algorithms to Live By"`
	`"isbn": " 1627790365",`
	`"author": "Brian Christian",`
	`"year": ": 2017`
	`}`

PUT

Metoda PUT je používána v souvislosti s aktualizací konkrétního prvku. Jedná se o nahrazení celého prvku, tedy přepsání všech jeho atributů tak, jak jsou poslány v těle požadavku. Pro aktualizaci jen části prvku se využívá metoda PATCH. Ta oproti tomu umožňuje měnit jen jeden nebo více atributů daného prvku.

Tabulka 8: REST: aktualizace prvku

Požadavek	Odpověď

`PUT /api/2/books/3`
`Content-Type: application/json`
 	
`{`
`"Id": "3"`
`"title": "Algorithms to Live By"`
`"isbn": " 1627790365",`
`"author": "Tom Griffiths",`
`"year": ": 2017`
`}`
	`HTTP/1.1 200 OK`
	`Content-Type: application/json`
	
	`{`
	`"Id": "3"`
	`"title": "Algorithms to Live By"`
	`"isbn": " 1627790365",`
	`"author": "Tom Griffiths",`
	`"year": ": 2017`
	`}`

Alternativně může být stavový kód HTTP odpovědi úspěšně provedeného požadavku pro aktualizaci prvku jen **204 No Content** a to bez těla zprávy.

DELETE

Pro smazaní daného prvku ze serveru se přirozeně použije HTTP metoda Delete.

Tabulka 9: REST: smazání prvku

Požadavek	Odpověď
`DELETE /api/2/books/3`	`HTTP/1.1 204 No Content`

3.1.2 Implementace

Jelikož je REST dnes už skoro standardem při navrhování webových API, k vytvoření RESTful API v .NET není potřeba přidávat do projektu velké množství závislostí. Platforma ASP.NET obsahuje všechny potřebné knihovny a nástroje pro tvorbu webových aplikací. Vývojáři mohou takřka ihned po vytvoření projektu začít navrhovat a programovat backend pro svůj informační systém.

Ať je to REST či GraphQL, vždy je potřeba si nejdřív vytvořit zdroj dat, ze kterého bude aplikace data číst, a do kterého bude data zapisovat. Nejčastějším příkladem pro tyto účely je databáze.

Běžným přístupem bývá také použití *objektově relačního mapování* (ORM), které zajistí automatickou konverzi mezi relační databází a objektově orientovaným programovacím jazykem.

Při uvádění příkladů bude předpokládána již vytvořená relační databáze, která je spojena s aplikací pomocí *připojovacího řetězce*. Navíc bude předpokládána implementace entit, které odpovídají tabulkám v existující databázi díky mapování ORM technikou.

API založené na kontrolerech

Hlavním pilířem Web Api v .NET jsou tzv. *kontrolery*. Kontrolery řídí příchozí HTTP požadavky a odchozí odpovědi. Jsou implementovány třídami, které dědí ze třídy *ControllerBase*. Následující zdrojový kód 1 ukazuje definici kontroleru pro zpracování požadavků spjatých výhradně s entitami představující knihy.

```
1 [ApiController]
2 [Route("[controller]")]
3 public class BooksController : ControllerBase
```

Zdrojový kód 1: API kontroler v .NET

Třída ControllerBase poskytuje mnoho vlastností a metod, které jsou užitečné pro zpracování HTTP požadavků, jako například metoda *Ok*. Tato metoda přeformátuje obsah do těla HTTP odpovědi a vrací stavový kód 200.

Následující zdrojový kód 2 ukazuje zpracování GET požadavku pro získání informací o konkrétní knize s daným *id*. Navíc obsahuje kontrolu, že pokud kniha se zadaným identifikačním číslem v databázi neexistuje, vrací stavový kód 404.

```
[HttpGet("id")]
public ActionResult<Book> GetBook(int id)
{
    var book = db.Books.FirstOrDefault(x => x.Id == id);
    if (book is null)
    {
        return NotFound();
    }

    return Ok(book);
}
```

Zdrojový kód 2: Zpracování GET požadavku v .NET

Pro správné fungování kontrolerů je nutná jejich registrace do *Dependency injection kontejneru* pomocí metody rozšíření *AddControllers*. Toho lze ve verzích .NET 5 a starších docílit ve třídě *Startup* v metodě s názvem *ConfigureServices*.

Následně je důležité přidat middleware pro mapování požadavků na odpovídající koncové body v kontrolerech. Přidání middlewarů se v .NET 5 děje v metodě *Configure*.

```
// Přidání služeb do kontejneru.
public void ConfigureServices(IServiceCollection services)
{

    services.AddControllers();
}

// Konfigurace pipeliny pro HTTP požadavky.
public void Configure(IApplicationBuilder app, IWebHostEnvironment
    env)
{
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Zdrojový kód 3: Konfigurace pro API založené na kontrolerech v .NET 5

Zdrojový kód 3 zobrazuje nutnou konfiguraci pro API založené na kontrolerech v .NET 5 v souboru *Startup.cs*.

Následující zdrojový kód 4 uvádí stejnou konfiguraci, ale pro .NET verzi 6 a vyšší. Tato konfigurace se provádí v souboru *Program.cs*.

```
var builder = WebApplication.CreateBuilder(args);

// Přidání služeb do kontejneru.
builder.Services.AddControllers();

var app = builder.Build();

// Konfigurace pipeliny pro HTTP požadavky.
app.MapControllers();
```

Zdrojový kód 4: Konfigurace pro API založené na kontrolerech v .NET 6

Minimal API

Minimal API jsou navrženy pro vytváření HTTP API s minimálními závislostmi. Jsou ideální pro mikroslužby a aplikace, které chtějí zahrnovat jen minimum souborů a funkcí v ASP.NET. Byly představeny v .NET 6 pro zjednodušení vývoje webových aplikací.

Konfigurace včetně implementace koncových bodů se děje v jediném souboru. V souboru *Program.cs* lze při použití Minimal API provést přidání služeb do kontejneru, konfigurace pipeliny pro HTTP požadavky i definovat koncové body. Následující zdrojový kód 5 zobrazuje příklad definice koncového bodu pro zpracování GET požadavku.[5]

```
app.MapGet("/books/{id}", async (int id) =>
{
    var book = db.Books.FirstOrDefault(x => x.Id == id);
    if (book is null)
    {
        return NotFound();
    }

    return Ok(book);
});
```

Zdrojový kód 5: Zpracování GET požadavku v .NET Minimal API

3.2 GraphQL

GraphQL byl vytvořen společností Facebook v roce 2015 jako vylepšení architektury REST. Pokouší se řešit jeho nedostatky a neefektivitu. Například GraphQL

umožňuje klientovi definovat konkrétní data, která budou vrácena ze serveru, což u REST není možné.

GraphQL vytváří rozhraní API, která fungují jako struktura, jejímž prostřednictvím klient odesílá a přijímá požadavky HTTP.

Jak název napovídá, GraphQL strukturuje data ve formě grafu prostřednictvím svého pokročilého dotazovacího jazyka pro získávání, extrahování a změnu dat. GraphQL je dotazovací jazyk pro webová API, ale to pouze z pohledu frontendového spotřebitele. GraphQL je také runtime vrstva, kterou je třeba implementovat na backendu.

Běhové prostředí GraphQL poskytuje serverům strukturu k popisu dat, která mají být vystavena v jejich rozhraních API. Tato struktura je to, co se ve světě GraphQL nazývá schéma.

Spotřebitel API pak může použít jazyk GraphQL k vytvoření textového požadavku. Klient odešle tento textový požadavek službě API prostřednictvím protokolu HTTP. Runtime vrstva GraphQL přijímá textový požadavek, komunikuje s ostatními službami na backendu, aby sestavil odpovídající datovou odpověď, a poté tato data odešle zpět ve formátu JSON.

3.2.1 Práce s GraphQL

Nejzákladnější pojmy související s využíváním GraphQL jsou schéma a typy. Schéma stojí na pozici prostředníka mezi frontendem a backendem, přičemž jsou na jeho základě ověřovány všechny procházející dotazy. Schéma obsahuje definice všech souvisejících typů. Typy představují objekty, o které lze server požádat.

Práce s GraphQL je rozdělena na dvě operace. Operace query a mutation. GraphQL query se používá ke čtení, zatímco mutation se používá k zápisu. V obou případech je operace jednoduchý řetězec odeslaný pomocí HTTP metody POST, který může server GraphQL analyzovat a odpovědět na něj daty ve specifickém formátu.

Obě operace je nutné volat s příslušným jménem funkce, která musí být definována na serveru s odpovídající funkčností.

Query

V tabulce 10 je zobrazen příklad operace query, která je určena k získání všech knih z databáze. Navíc lze definovat atributy, které jsou pro konzumenta potřebné. V tomto případě jen kód isbn každé knihy.

Tabulka 10: GraphQL: query

Požadavek	Odpověď
`query { Books{ isbn } }`	`[ { "isbn": "1788833732" }, { "isbn": "484279565" }, { "isbn": "1627790365" } ]`

Mutation

K vytvoření nového, případně aktualizaci již existujícího prvku nebo jeho smazání, používá GraphQL operaci mutation.

Pro přidání nového záznamu do databáze knih lze využít následující dotaz.

Tabulka 11: GraphQL: mutation

Požadavek	Odpověď
`mutation { createBook(book: { title: ""Algorithms to Live By", isbn: "1627790365", author: "Brian Christian", year: 2017 }){ id, isbn } }`	`{ "id": 3 "isbn": "1627790365" }`

3.2.2 Implementace

K implementaci GraphQL je potřeba do ASP.NET projektu zahrnout nutnou dávku závislostí. Základní je knihovna *GraphQL.NET*. Díky ní lze zprovoznit běhové prostření na straně serveru pro provádění dotazů pomocí typového systému.

Mezi další důležití závislosti patří:

- *GraphQL.Server.Transports.AspNetCore*: umožnuje provozovat GraphQL přes HTTP protokol,
- *GraphQL.SystemTextJson*: pro de/serializaci požadavků a odpovědí.

Následující příklady budou stejně jako v kapitole o implementaci REST API 3.1.2 uváděny s předpokladem již existující databáze a již vytvořených odpovídajících entit.

Služba GraphQL je vytvořena definováním typů a polí na těchto typech a poté poskytováním funkcí pro každé pole každého typu.

Schéma lze na platformě .NET implementovat dvěma různými způsoby. Jeden je tzv. *Schema first* přístup, který používá GraphQL jazyk pro definici schémat. Druhý přístup je nazývám *Code first*, a jde o vytváření jednotlivých typů na základě generické třídy *ObjectGraphType*, která je poskytována v rámci knihovny *GraphQL.NET*.

Schema first

Tento přístup spoléhá na konvence a snaží se poskytnout minimální množství syntaxe. Je jednodušší na implementaci, ale má nevýhodu ve vytváření GraphQL typů dynamicky typovým způsobem.

Zdrojový kód 6 zobrazuje vytvoření schématu použitím přístupu Schema first. Obsahuje rozhraní a implementaci dotazů pro získání všech knih a získání konkrétní knihy z databáze. Pro mapování operace do schématu se používá atribut *GraphQLMetadata*.

```
1  public class MyQuery
2  {
3    [GraphQLMetadata("book")]
4    public Book GetBook(int id)
5    {
6      return db.Books.FirstOrDefault(x => x.Id == id);
7    }
8
9    [GraphQLMetadata("books")]
10   public IQueryable<Book> GetBooks()
11   {
12     return db.Books;;
13   }
14 }
15
16 var schema = Schema.For(@"
17   type Book {
18     id: Int
19     title: String
20     isbn: String
21     author: String
22     year: Int
23   }
24
25   type Query {
26     book: Book
27     books: [Book]
28   }
29 ", _ => {
30     _.Types.Include<MyQuery>();
31 });
```

Zdrojový kód 6: Definice GraphQL schématu přístupem Schema first

Code first

Oproti tomu přístup zvaný *Code first* nebo také *GraphType* přístup je o něco robustnější, ale vzhledem ke statické typovosti bezpečnější. Zdrojový kód 7 zobrazuje implementaci stejného schématu, jako zdrojový kód 6, ale s použitím pří-

stupu Code first. Jeho součástí je vytvoření typu *BookType* z entity *Book*, která je mapovaná z databáze.

```
1  public class BookType : ObjectGraphType<Book>
2  {
3    public BookType()
4    {
5      Field(x => x.Id);
6      Field(x => x.Title);
7      Field(x => x.Isbn);
8      Field(x => x.Author);
9      Field(x => x.Year);
10   }
11 }
12
13 public class MyQuery : ObjectGraphType
14 {
15   public MyQuery()
16   {
17     Field<ListGraphType<BookType>>("books")
18       .Resolve(_ => db.Books);
19
20   Field<BookType>("book")
21     .Argument<int>("id")
22     .Resolve(context => db.Books.FirstOrDefault(x => x.Id == context.
         GetArgument<int>("id")));
23   }
24 }
25
26 var schema = new Schema { Query = new MyQuery() };
```

Zdrojový kód 7: Definice GraphQL schématu přístupem Code first

Konfigurace

GraphQL.NET poskytuje rozhraní *IGraphQLBuilder*, které zapouzdřuje konfigurační metody pro vkládání závislostí. Poskytuje tak abstraktní metodu konfigurace dependency injection kontejneru pro práci s GraphQL.NET. Pomocí *AddGraphQL* lze volat metody rozšíření pro konfiguraci této knihovny.

Zdrojový kód 8 obsahuje jednoduchý příklad konfigurace GraphQL na platformě .NET 6. Příklad obsahuje třídu *MySchema*, což je třída vytvořená z bázové třídy *Schema*. Třída Schema je poskytovaná v rámci knihovny GraphQL.NET a umožňuje snadnou tvorbu schémat přístupem Code first.

```
1  var builder = WebApplication.CreateBuilder(args);
2
3  // Přidání služeb do kontejneru.
4  builder.Services.AddGraphQL(builder => builder
5      .AddSchema<MySchema>(ServiceLifetime.Scoped)
6      .AddGraphTypes()
7      .AddSystemTextJson());
8
9  var app = builder.Build();
10
11 // Konfigurace pipeliny pro HTTP požadavky.
12 app.UseGraphQL<MySchema>("/api");
```

Zdrojový kód 8: Konfigurace pro GraphQL v .NET 6

Jak ukazuje zdrojový kód 8, v rámci rozhraní IGraphQLBuilder (builder) jsou volány tři konfigurační metody pro registraci do dependency injection kontejneru:

- *AddSchema*: zaregistruje schéma MySchema, jehož instance se vytváří jednou za HTTP požadavek,
- *AddGraphTypes*: zaregistruje všechny typy v rámci shématu MySchema,
- *AddSystemTextJson*: zaregistruje *GraphQLSerializer*, jehož instance se vytoří jednou za životní cyklus aplikace. Slouží k de/serializaci požadavků a odpovědí.

Metoda *UseGraphQL* navíc zajistí použití GraphQL middlewaru pro HTTP požadavky. Argument *"/api"* určuje cestu ke koncovému bodu, který je řízen schématem MySchema.

3.3 Srovnání REST a GraphQL

REST i GraphQL patří mezi nejpoužívanější API architektury. REST je dnes už skoro standardem a je tak nejvyužívanější API architekturou vůbec. Každý styl má své výhody a omezení. Při výběru je nutné brát v úvahu případ použití, očekávání uživatele a další aspekty.

Způsob dotazování

Základní rozdíl mezi GraphQL API a REST API spočívá ve způsobu dotazování. Zatímco REST funguje na principu koncových bodů, které se vztahují ke každému zdroji, GraphQL si lze představit jako grafovou strukturu, případně množinu bodů s pouze jedním koncovým bodem.

Načítání dat

Jedním z nejběžnějších omezení REST je tzv. *overfetching* a *underfetching*. jde o to, že jediným způsobem, jak může klient stáhnout data ze serveru, je poslat požadavek na koncový bod, který vrací pevně strukturovaná data.

Overfetching znamená získat více informací, než je potřeba. Underfetching naopak nedostatek dat a tedy nutnost odeslání dalšího dotazu. V obou případech se jedná o problémy s výkonem: buď se používá větší šířka pásma, nebo se provádí více HTTP požadavků.

GraphQL řeší tyto problémy tím, že umožňuje klientům definovat konkrétní atributy entit, které potřebuje, a ten tak nedostane nepotřebná data. Navíc poskytuje podporu vnořených atributů, klient je tak schopen jedním dotazem získat informace o požadované entitě včetně vnořených potomků.

Náročnost

Když dojde na srovnání architektur podle toho, jak je náročné se je naučit používat, GraphQL má křivku učení méně strmější, respektive je pomalejší se jej naučit. Je to z toho důvodu, že má specifický dotazovací jazyk s vlastními pravidly. REST je minimálně do začátků snadnější. Nemá téměř žádná pravidla, co by měl zdroj vracet nebo požadovat.

Cache

Při práci s rozsáhlými systémy je nezbytné mít mechanismus mezipaměti pro rychlý přístup k častěji používaným datům. V případě RESTful API může mít výsledný systém implementovanou vlastní mezipaměti nebo může standardně používat mezipaměť na úrovni HTTP. GraphQL bohužel ve výchozím nastavení nepodporuje ukládání do mezipaměti na úrovni HTTP.

Odpověď

Je důležité vědět, jaký formát odpovědi dostáváme. Na základě návrhu systému lze RESTful API nakonfigurovat tak, aby podporovala více formátů odpovědí (JSON, XML, HTML, YAML atd.). GraphQL podporuje pouze odpovědi JSON.

Tabulka 12: Srovnání GraphQL a REST

GraphQL	**REST**
Lze organizovat podle schémat.	Lze organizovat podle koncových bodů.
Má menší, rostoucí komunitu.	Má velkou, ustálenou komunitu.
Rychlost vývoje je poměrně vysoká.	Vývoje je spíše pomalejší.
Implementace v .NET je výrazně složitější.	Implementace v .NET je snadná.
Používá metadata pro validaci požadavku před odesláním.	Validace probíhá při přijetí požadavku na serveru.
Používá specifický nástroj pro dokumentaci.	Umožňuje různé možnosti dokumentace.

Tabulka 12 popisuje dodatečné srovnání vlastností architektur GraphQL a REST.

Závěr

Každá ze srovnaných architektur je vhodná pro rozdílné případy užití. Nedá se tedy říct, že by byla jedna lepší než druhá.

GraphQL funguje lépe pro následující scénáře:

- aplikace pro zařízení, jako jsou mobilní telefony či chytré hodinky, kde záleží na využití šířky pásma,
- aplikace, kde je třeba načíst vnořená data v jediném volání. Například blog nebo platforma sociálních sítí, kde je třeba načítat příspěvky spolu s vnořenými komentáři a podrobnostmi o osobě, která komentář komentuje.

Oproti tomu REST je vhodnější pro aplikace, které vyžadují robustní API s ukládáním do mezipaměti a monitorovacím systémem.

4 Dotazovací jazyky

Dotazovací jazyky pro webová API jsou jazyky, které se používají k dotazování na webové servery. Tyto dotazy slouží k získávání dat nebo k provádění určitých operací, jako je například úprava nebo mazání dat. Dotazovací jazyky pro webové API se často využívají k přístupu k databázím, kde se informace ukládají.

Existuje několik různých typů dotazovacích jazyků pro webové API, každý z nich má své specifické vlastnosti a omezení. Tyto dotazovací jazyky umožňují klientům specifikovat, jaké konkrétní data potřebují a server pak poskytuje pouze tyto data.

Dotazovací jazyky webového rozhraní API poskytují podporu pro operace jako filtrování, řazení, seskupení a agregaci dat, což umožňuje vytvářet složitější dotazy a získávat přesnější data. To je užitečné pro různé typy aplikací, jako jsou například e-commerce nebo sociální sítě.

V poslední době se stále více rozšiřuje využití dotazovacích jazyků, které umožňují větší flexibilitu a efektivitu v komunikaci mezi klientem a serverem. Tyto dotazovací jazyky poskytují silnou podporu pro specifické požadavky aplikací, jako například škálovatelnost, přehlednost a snadnou implementaci.

Výběr správného dotazovacího jazyka pro webové rozhraní API závisí na konkrétních potřebách a požadavcích projektu. Je důležité zvážit požadavky na rychlost, flexibilitu, bezpečnost a další faktory, aby byl dotazovací jazyk co nejvhodnější pro konkrétní projekt.

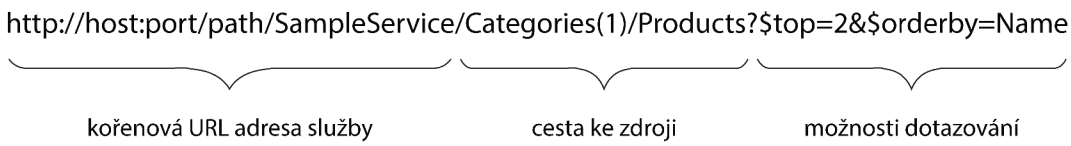
Mezi nejznámější dotazovací jazyky pro webová API patří OData, JSON:API a HotChocolate.

4.1 OData

OData (Open Data Protocol) je schválený standard, který definuje sadu nejlepších postupů pro vytváření a spotřebovávání REST API. Umožňuje vytváření služeb založených na REST, které poskytují možnost publikovat a upravovat zdroje identifikované pomocí URL a definované v datovém modelu pomocí jednoduchých HTTP zpráv.

Protokol OData je protokol na aplikační úrovni a slouží k interakci s daty prostřednictvím rozhraní RESTful. Podporuje popis datových modelů, editaci a dotazování dat podle těchto modelů. REST API, která jsou založena na OData, je snadné pochopit a používat díky metadatům, které OData poskytuje.

Jelikož se jedná o REST specifikaci, OData klienti komunikují se serverem pomocí standardních operací GET, PUT, PATCH, POST a DELETE. Adresa URL služby OData se skládá ze 3 částí.



Obrázek 6: OData URL

Obrázek 6 obsahuje strukturu URL adresy služby OData s následujícími částmi:

- **kořenová URL adresa služby**: základní adresa URL pro službu OData.

Zahrnuje protokol (http nebo https), název hostitele nebo IP adresu s číslem portu, na kterém služba běží, případně i kořenovou cestu služby,

- **cesta ke zdroji**: zdroj podle definice REST je objekt, který je přístupný přes HTTP pomocí standardních metod GET, POST, PUT, PATCH a DELETE. Může to být jeden objekt nebo sbírka podobných objektů,
- **možnosti dotazování**: jedná se o standardizované parametry ve formě řetězců, které lze předat službě OData za účelem spouštění dotazů na požadovaný zdroj. Lze díky nim provádět operace, jako je *select*, *filter*, *count*, *skip*, *order*, *search* a *format*. Všechny dotazy začínají znakem $ a nerozlišují velká a malá písmena. Například následující adresa URL vybírá produkt podle barvy: *https://server/products?$filter=color eq 'red'*.

4.1.1 Čtení dat

Služba OData podporuje čtení dat prostřednictvím požadavků HTTP GET.

Následující požadavek slouží k získání všech filmů z databáze, a vrací tak kolekci všech záznamů z tabulky *Movies*.

GET odataServiceRoot/Movies

Zdrojový kód 9 obsahuje tělo možné odpovědi dotazu, který vrací kolekci filmů. Příklad je uváděn při základní konfiguraci služby OData. Mimo požadovanou kolekci obsahuje odpověď i atribut *@odata.context*, který je určen k poskytování metadat. Obsahuje odkaz na dokument, popisující schéma dat, která jsou vrácena v odpovědi.

Níže uvedený požadavek vrací samostatnou entitu typu *Movie* s daným identifikačním číslem. Předpokládá se tedy, že sloupec *id* je v tabulce *Movie* primárním klíčem.

GET odataServiceRoot/Movies(48)

Oba předchozí dotazy vrací data ve formátu podobným běžným REST požadavkům, jak je v vysvětleno v kapitole 3.1.1.

Odlišné odpovědi přichází ve chvíli potřeby získat jen konkrétní vlastnost entity. Bez nutnosti použití složitějšího dotazování, umožňuje OData vrátit konkrétní sloupec daného záznamu. Následující požadavek vrací jméno filmu s Id rovným 48.

GET odataServiceRoot/Movies(48)/Name

Tělo odpovědi je json objekt s atributem *value*, jehož hodnota odpovídá požadované vlastnosti.

Hodnotu vlastnost lze získat i přes vnořené typy objektů. Níže uvedený požadavek vrací jméno režiséra, který režíroval konkrétní film.

GET odataServiceRoot/Movies(48)/Director/Name

Pokud klient nechce pracovat s objekty obsaženými v json odpovědi, může využít klíčové slovo *$value*. Následující požadavek vrací hodnotu vlastnosti jako prostý text.

GET odataServiceRoot/Movies(48)/Director/Name/$value

```
{
   "@odata.context": "http://odataServiceRoot/\$metadata#movies",
   "value": [
      {
         "Id": 0,
         "Name": "The Lord of the Rings: The Return of the King",
         "Year": 2003,
         "Rank": 8.9
      },
      {
         "Id": 1,
         "Name": "Pulp Fiction",
         "Year": 1994,
         "Rank": 8.8
      },

      // Vynechaná data pro stručnost
   ]
}
```

Zdrojový kód 9: Příklad těla odpovědi OData požadavku na kolekci entit

4.1.2 Možnosti dotazování

K dotazu určenému pro získání zdroje lze přidat sadu řetězců a řídit tak množství vrácených dat. Při dotazování lze přidat tyto parametry a v podstatě tak vyžádat provedení sady transformací, jako je filtrování, řazení atd. Odata podporuje parametry v každé HTTP metodě kromě DELETE.

Odata poskytuje následující operace.

Filter

Operace *filter* umožňuje klientům filtrovat kolekci zdrojů, které jsou požadovány v URL požadavku. Výraz zadaný pomocí $filter je vyhodnocen pro každý zdroj v kolekci a do odpovědi jsou zahrnuty pouze položky, u kterých je výraz vyhodnocen jako *true*. Prostředky, pro které je výraz vyhodnocen jako *false* nebo *null*, jsou z odpovědi vynechány.

Příklady:

- **Všichni herci s křestním jménem *Jack*:**

  /actors?$filter=first_name eq 'Jack'

- **Všichni herci s křestním jménem jiným než *Jack*:**
  /actors?$filter=first_name ne 'Jack'
- **Všichni herci s křestní jménem *Jack* nebo *John***
  /actors?$filter=first_name in ('Jack', 'John')
- **Všechny filmy s rokem vydání větším než *2010*:**
  /movies?$filter=year gt 2010

Expand

Operace *expand* se používá k načtení souvisejících zdrojů společně s hlavním zdrojem v jediném dotazu. Často se využívá k získání přidružených entit spojených cizím klíčem. Řeší tak typický problém REST - underfetching.

Příklady:

- **Všechny role společně s hercem a filmem, ve kterém hrál:**
  /Roles?$expand=Actor,Movie
- **Všechny role společně s filmem, který má jméno *Star Wars*:**
  /Roles?$expand=Movie($filter=name eq 'Star Wars')

Select

Operace *select* umožňuje získávat klientům konkrétní sadu vlastností každé entity nebo komplexního typu. Řeší tak další typický problém REST - overfetching.

Tato operace se často používá ve spojení s operací *expand*, která určí rozsah požadovaných zdrojů. Select poté určí podmnožinu vlastností pro každý zdroj.

Příklady:

- **Jméno a rok vydání každého filmu:**
  /Movies?$select=name,year
- **Všechny role společně se jménem filmu:**
  /Roles?$expand=movie($select=name)

OrderBy

Operace *order by* slouží k požadování zdrojů v určitém pořadí. Lze použít přepínač *asc* k vzestupnému třídění nebo *desc* k třídění sestupnému, přičemž bez uvedení směru, je použito vzestupné třídění.

Příklady:

- **Všechny filmy seřazené podle roku vydání sestupně a poté podle jména vzestupně:**
  /Movies?$orderby=year desc, name asc

- **Všechny filmy se všemi rolemi, které jsou seřazeny vzestupně podle jména:**

  /Movies?$expand=roles($orderby=name)

Top

Operace *top* určuje maximální počet záznamů, které mohou být vráceny v odpovědi.

Příklad:

- **Prvních 10 filmů:**

  /Movies?$top=10

Skip

Operace *skip* určuje počet záznamů, které mají být přeskočeny a vyřazeny tak z odpovědi. Společně s operací *top* mohou být využity k efektivnímu stránkování.

Příklad:

- **Všechny filmy, bez prvních 10:**

  /Movies?$skip=10

Count

Operace *count* umožňuje zahrnout do odpovědi počet vrácených záznamů.

Příklady:

- **Všechny filmy, společně s jejich počtem:**

  /Movies?$count=true

- **Všechny filmy s rolemi, společně s počtem rolí:**

  /Movies?$expand=roles($count=true)

4.1.3 Implementace

Implementace dotazovacího jazyka OData na platformě ASP.NET se velmi neliší od standardního vytvoření REST API založeného na kontrolerech, jenž je popsáno v kapitole 3.1.2.

V první řadě je nutné do projektu zahrnout knihovnu, která obsahuje veškerou logiku tykající se OData protokolu: *Microsoft.AspNetCore.OData*

K vytvoření kontrolerů slouží třída *ODataController*. Zdrojový kód 10 ukazuje definici OData kontroleru pro zpracování požadavků spjatých výhradně s entitami představující herce.

```
1   [Route("[controller]")]
2    public class ActorsController : ODataController
```

Zdrojový kód 10: OData kontroler

Každý kontroler by měl obsahovat metody pro zpracování konkrétních HTTP požadavků. Například zdrojový kód 11 obsahuje action metodu, která umožňuje dotazování nad všemi herci v databázi. Metoda obsahuje atribut *EnableQueryAttribute*, který umožňuje jednoduché i složitější dotazy.

```
1       [EnableQuery]
2       public ActionResult Get()
3       {
4          return Ok(db.Actors);
5       }
```

Zdrojový kód 11: Zpracování GET požadavku pomocí OData

Odata službu je nutné zaregistrovat a nakonfigurovat v aplikaci. Pro .NET 6 stačí v souboru *Program.cs* správně použít rozšiřující metodu *AddOData*, která stojí za přidání služby a middleware do dependency injection kontejneru.

Dále je možné určit, které možnosti dotazování budou zpřístupněny při dotazování. Lze povolit operace, které byly popsány v kapitole 4.1.2.

Zdrojový kód 12 zobrazuje příklad registrace OData služby. V kódu je použita metoda *EnableQueryFeatures*, která explicitně povolí všechny možnosti dotazování. Její parametr nastavuje maximální povolenou hodnotu pro použití operace *top*.

```
1 builder.Services.AddControllers().AddOData(
2    options => options.EnableQueryFeatures(250)
```

Zdrojový kód 12: Konfigurace OData služby

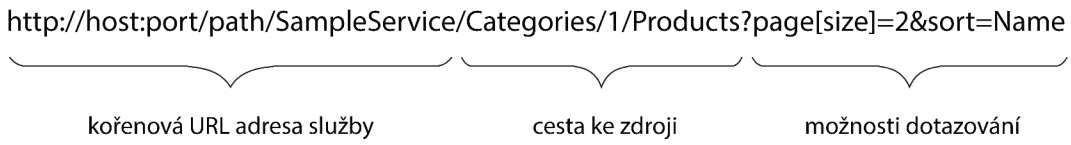
4.2 JSON:API

JSON:API je REST API specifikace, která umožňuje výměnu dat mezi klientem a serverem v podobě strukturovaných dat v formátu JSON. Tato specifikace definuje způsob, jakým jsou data organizována a poskytuje sémantické konvence pro manipulaci s daty.

Umožňuje efektivní výměnu dat mezi klientem a serverem díky minimalizaci počtu dotazů na server a zvýšení propustnosti. Klient může požadovat určité

informace o zdrojích a JSON:API odpoví jedním dotazem obsahujícím všechny požadované informace.

Definuje také způsob, jakým jsou prováděny operace CRUD na zdrojích. Pomocí JSON:API může klient vytvářet, číst, aktualizovat a mazat data pomocí standardizovaných metod HTTP jako GET, POST, PUT a DELETE.



Obrázek 7: JSON:API URL

Obrázek 7 obsahuje strukturu URL adresy služby JSON:API se třemi částmi. První dvě části - *kořenová URL adresa služby* a *cesta ke zdroji* jsou stejné, jako ty v URL adrese služby OData, jenž jsou vysvětlené v kapitole 4.1.

Poslední část - *možnosti dotazování* obsahuje parametry dotazu, kterými lze upřesnit počet a strukturu dat, vracených ze serveru. JSON:API umožňuje použití parametrů, jako jsou *limit*, *offset*, *sort*, *filter*, *fields*. Parametry dotazu začínají otazníkem (?) a oddělují se od sebe znakem ampersand (&). Například následující adresa URL vybírá produkt podle ceny:
https://server/products?filter=lessThan(price,'25').

4.2.1 Čtení dat

JSON:API podporuje čtení dat prostřednictvím požadavků HTTP GET.

Následující požadavek slouží k získání všech filmů z databáze.

GET jsonApiServiceRoot/Movies

Zdrojový kód 13 obsahuje tělo možné odpovědi dotazu, který vrací kolekci filmů. Při základní konfiguraci služby JSON:API, vrací server mimo samotná data i související odkazy. Služba se dá dále konfigurovat, aby nevracela žádná, nebo jen některá tato metadata. Mezi konfigurovatelná metadata patří aktuálně volaná url adresa, odkaz pro získání každé individuální entity nebo odkaz na informace o vztahu mezi entitami (self), případně odkaz na samotná data daného vztahu (related).

Pro získání konkrétní entity, lze využít URL adresu v následujícím tvaru.

GET jsonApiServiceRoot/Movies/48

JSON:API nepodporuje požadavky na jednotlivou vlastnost entity. Pro dosažení podobné funkcionality musí uživatel využít parametr *fields*, jenž je vysvětlen v kapitole 4.2.2 - Možnosti dotazování.

```
{
   "links": {
      "self": "jsonApiServiceRoot/api/movies"
   },
   "data": [
      {
         "type": "movies",
         "id": "0",
         "attributes": {
            "name": "The Lord of the Rings: The Return of the King",
            "year": 2003,
            "rank": 8.4
         },
         "relationships": {
            "roles": {
               "links": {
                  "self": "jsonApiServiceRoot/api/movies/0/
                     relationships/roles",
                  "related": "jsonApiServiceRoot/api/movies/0/roles"
               }
            },
            "director": {
               "links": {
                  "self": "jsonApiServiceRoot/api/movies/0/
                     relationships/director",
                  "related": "jsonApiServiceRoot/api/movies/0/
                     director"
               }
            }
         },
         "links": {
            "self": "jsonApiServiceRoot/api/movies/0"
         }
      },

      // Vynechaná data pro stručnost
   ]
}
```

Zdrojový kód 13: Příklad těla odpovědi JSON:API požadavku na kolekci entit

4.2.2 Možnosti dotazování

Při odesílání GET požadavku na koncový bod lze odpovědi upřesnit pomocí volitelných funkcí popsaných níže.

Filter

Parametr *filter* umožňuje klientům filtrovat kolekci zdrojů, která je požadována v URL požadavku. Jedná se o ekvivalenci operace *filter* v požadavku služby

ODdata.

Příklady:

- **Všichni herci s křestním jménem *Jack*:**
  /actors?filter=equals(first_name,'Jack')
- **Všichni herci s křestním jménem jiným než *Jack*:**
  /actors?filter=not(equals(first_name,'Jack'))
- **Všichni herci s křestní jménem *Jack* nebo *John***
  /actors?filter=or(equals(first_name,'Jack'),equals(first_name,'Jack')))
- **Všechny filmy s rokem vydání větším než *2010*:**
  /movies?filter=greaterThan(year,'2007')

Include

Parametr *include* se používá k načtení souvisejících zdrojů společně s hlavním zdrojem v jediném dotazu. Jedná se o ekvivalenci operace *expand* v požadavku služby ODdata.

Příklady:

- **Všechny role společně s hercem a filmem, ve kterém hrál:**
  /roles?include=actor,movie
- **Všechny role společně s filmem, který má jméno *Star Wars*:**
  /roles?$include=movie&filter=equals(movie.name,'Star Wars')

Fields

Parametr *fields* umožňuje získávat klientům konkrétní sadu vlastností každé entity. Jedná se o ekvivalenci operace *select* v požadavku služby ODdata.

Příklady:

- **Jméno a rok vydání každého filmu:**
  /movies?fields[movies]=name,year
- **Všechny role společně se jménem filmu:**
  /roles?include=movie&fields[movie]=name)

Sort

Parametr *sort* slouží k požadování zdrojů v určitém pořadí. K sestupnému třídění lze použít přepínač mínus (-). Naopak třídění vzestupné je výchozí a uvádí se bez znaménka. Jedná se o ekvivalenci operace *order by* v požadavku služby ODdata.

Příklady:

- **Všechny filmy seřazené podle roku vydání sestupně a poté podle jména vzestupně:**

  /movies?sort=-year,name

- **Všechny filmy se všemi rolemi, které jsou seřazeny vzesestupně podle jména:**

  /movies?include=roles&sort=[roles]=name

Page

Parametr *page* slouží ke stránkování výsledků v odpovědi. Jedná se kombinaci operací *top* a *skip* v OData api požadavku.

Příklad:

- **Prvních 10 filmů:**

  /movies?page[size]=10

- **Dalších 10 filmů, bez prvních 10:**

  /movies?page[size]=10&page[number]=2

4.2.3 Implementace

K implementaci dotazovacího jazyka JSON:API na platformě ASP.NET lze použít přístup REST API založené na kontrolerech, který je popsán v kapitole 3.1.2.

Logika pro specifikaci JSON:API je zahrnuta v knihovně *JsonApiDotNetCore*. Entitní modely pro mapování databázových tabulek mají odlišný tvar, než klasické entity při implementaci REST API, případně OData API.

Zdrojový kód 14 obsahuje příklad validní entity. Aby mohla být publikována pomocí kontroleru, musí dědit z generické třídy *Identifiable*, kde generický typ označuje datový typ primárního klíče. JsonApiDotNetCore bohužel nepodporuje tabulky bez primárního klíče, nebo tabulky se složeným primárním klíčem.

```
public class Movie : Identifiable<int>
{
    [Attr]
    public string Name { get; set; }

    [Attr]
    public int Year { get; set; }

    [Attr]
    public float Rank { get; set; }

    public int DirectorId { get; set; }

    [HasOne]
    public Director Director { get; set; } = null!;

    [HasMany]
    public ICollection<Actor> Roles { get; set; } = new HashSet<Actor
        >();
}
```

Zdrojový kód 14: Entitní model JSON:API

Attr atribut označuje vlastnosti, které mohou být vráceny v HTTP odpovědi. *HasOne* atribut definuje vztah *one-to-one* a *HasMany* zase vztah *one-to-many*.

K vytvoření kontroleru slouží generická třída *JsonApiController*. Zdrojový kód 15 obsahuje definici kontroleru pro vystavení kolekce s filmy.

Navíc při použití ORM systému *Entity Framework Core*, který je při využívání knihovny *JsonApiDotNetCore* nastavený jako výchozí, stačí implementovat základní konstruktor bez těla, a třída *JsonApiController* se už postará o vše ostatní.

```
public class MoviesController : JsonApiController<Movie, int>
```

Zdrojový kód 15: JSON:API Kontroler

Pro konfiguraci JsonApiDotNetCore a registraci zdrojů Entity Framework Core modelu slouží metoda *AddJsonApi*. Při použití jiného ORM systému je nutné zaregistrovat zdroje do dependency injection kontejneru klasickým způsobem, aby ho mohl odpovídající kontroler použít.

Zdrojový kód 16 obsahuje příklad registrace JSON:API s Entity Framework modelem definovaného ve třídě *DbContext*. Navíc má nakonfigurovaný URL prefix pro vystavené endpointy.

```
builder.Services.AddJsonApi<DbContext>(options =>
{
   options.Namespace = "api";
});
```

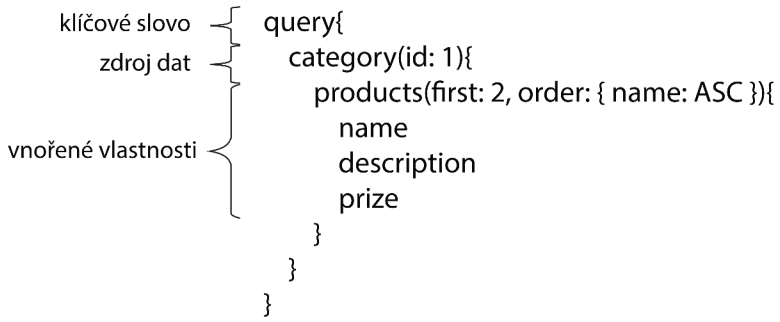
Zdrojový kód 16: Konfigurace JSON:API služby

4.3 HotChocolate

HotChocolate je knihovna pro platformu .NET, která se zaměřuje na implementaci a integraci GraphQL API. Jelikož se jedná o GraphQL specifikaci, což už je dotazovací jazyk sám o sobě, má HotChocolate oproti ostatním dotazovacím jazykům, které jsou v této práci srovnávány výhodu.

Mimo běžnou funkcionalitu GraphQL nabízí knihovna HotChocolate snadnou integraci s .NET technologiemi, jako jsou ASP.NET pro webové služby, Entity Framework pro přístup k datům nebo Blazor pro frontend.

Navíc obsahuje vestavěnou podporu pro filtraci, třídění a stránkování dat před odesláním zpět na klienta.



Obrázek 8: HotChocolate dotaz

Obrázek 8 obsahuje tělo požadavku služby GraphQL s podporou HotChocolate. Požadavek se skládá ze tří částí:

- **klíčové slovo**: označuje začátek operace dotazu GraphQL. Používá se ke čtení nebo načítání dat ze serveru,
- **zdroj dat**: požadovaný zdroj dat odpovídající tabulce v databázi. Tato část často obsahuje dodatečné možnosti dotazování jako je *order*, *where*, *first*,
- **vnořené vlastnosti**: povinný seznam vlastností hlavní entity, které mají být zahrnuty v odpovědi. Odpovídá parametru *fields* v URL adrese JSON:API požadavku nebo parametru *select* v URL adrese požadavku služby OData. Specifické vlastnosti mohou obsahovat dodatečné možnosti dotazování.

4.3.1 Čtení dat

Pro dotazování dat v HotChocolate slouží operace *Query*. Požadavek se posílá HTTP metodou POST na jediný vystavený endpoint.

Následující požadavek, respektive tělo požadavku slouží k získání všech filmů z databáze, a vrací tak kolekci všech záznamů z tabulky Movies.

```
query{
  movies{
    name,
    year,
    rank,
    roles{
      roleName
      actor{
        firstName,
        lastName
      }
    }
  }
}
```

Základní čtení dat se nijak neliší od klasického dotazování v GraphQL. Jak je vidět z předchozího příkladu, je nutné zadat všechny vlastnosti entity, které se mají objevit v odpovědi, včetně možnosti vybrat i vnořené vlastnosti, které se ve skutečnosti nacházejí v jiné tabulce. Bez nutnosti složitějšího dotazování tak řeší overfetching a underfetching.

Zdrojový kód 17 obsahuje tělo možné odpovědi dotazu, který vrací kolekci filmů. Samotná data jsou zahrnuta v atributu *data*.

Pro získání konkrétní entity, lze využít širší možnosti dotazování a filtrovat podle požadovaných vlastností, nebo využít parametrizované query tak, jak ukazuje následující příklad.

```
query{
  movie(id: 48){
    name,
    year,
    rank,
    roles{
      roleName
      actor{
        firstName,
        lastName
      }
    }
  }
}
```

```
{
   "data": {
      "movies": [
         {
            "name": "The Lord of the Rings: The Return of the King",
            "year": 2003,
            "rank": 8.4,
            "roles": [
               {
                  "roleName": "Frodo",
                  "actor": {
                     "firstName": "Elijah",
                     "lastName": "Wood"
                  }
               },
               {
                  "roleName": "Aragorn",
                  "actor": {
                     "firstName": "Viggo",
                     "lastName": "Mortensen"
                  }
               },

               // Vynechaná data pro stručnost
            ]
         },

         // Vynechaná data pro stručnost
      ]
   }
}
```

Zdrojový kód 17: Příklad těla odpovědi HotChocolate požadavku na kolekci entit

4.3.2 Možnosti dotazování

Stejně jako předchozí 2 dotazovací jazyky, i HotChocolate umožňuje efektivně a flexibilně získávat data z databáze nebo jiných zdrojů. Tato funkcionalita dovoluje klientovi získat pouze relevantní a požadované informace, což snižuje zátěž databáze a zlepšuje rychlost odezvy API.

Jelikož je HotChocolate GraphQL specifikací, nikoli specifikací architektonického stylu REST, nedisponuje externími operacemi, které umožňují přidat, či odebrat vlastnosti entity na výstupu. Tato funkčnost je již obsažena v rámci GraphQL, jak je popsáno v kapitole věnované srovnání REST a GraphQL 3.3.

HotChocolate nabízí následující operace, které má klient k dispozici při dotazování dat.

Filtrování

Stejně jako má Odata služba a služba JSON:API operaci *filter*, i HotChocolate umožňuje stejnou funkcionalitu, a to v podobě parametru *where*, který umožňuje klientovi specifikovat podmínky, podle kterých se mají data filtrovat.

Příklady:

- **Všichni herci s křestním jménem *Jack*:**

```
query {
  actors(where:{ firstName: { eq: "Jack" } }) {
    id
    firstName
    lastName
  }
}
```

- **Všichni herci s křestním jménem jiným než *Jack*:**

```
query {
  actors(where: { firstName: { neq: "Jack" } }) {
    id
    firstName
    lastName
  }
}
```

- **Všichni herci s křestní jménem *Jack* nebo *John***

```
query {
  actors(where: { firstName: { in: ["Jack", "John"] } }) {
    id
    firstName
    lastName
  }
}
```

- **Všechny filmy s rokem vydání větším než *2010*:**

```
query {
  movies(where: { year: { gt: 2010 } }) {
    id
    name
    year
  }
}
```

Řazení

K požadování zdrojů v určitém pořadí slouží operace *order*. HotChocolate podporuje řazení podle jednoho nebo více atributů vzestupně (ASC) nebo sestupně (DESC).

Příklady:

- **Všechny filmy seřazené podle roku vydání sestupně a poté podle jména vzestupně:**

```
query {
  movies(order: [{ year: DESC }, { name: ASC }]) {
    id
    name
    year
  }
}
```

- **Všechny filmy se všemi rolemi, které jsou seřazeny vzestupně podle jména:**

```
query {
  movies(order: [{ roles: { roleName: ASC } }]) {
    id
    name
    roles {
      id
      roleName
      actor {
        id
        firstName
        lastName
      }
    }
  }
}
```

Offset stránkování

Offset stránkování je jednoduchý způsob, jak získat data po stránkách, který používá číselné indexy pro definování začátku a počtu položek na stránce. V HotChocolate lze použít parametry *skip* a *take* pro použití offset stránkování. Tento typ stránkování je ekvivalentní se stránkováním použitelným v obou předchozích přístupech, OData a JSON:API.

Jakmile je na serveru nastaveno pro daný dotaz offset stránkování, automaticky se pozmění návratový typ na tzv. *CollectionSegment*. Tento typ obsahuje

hlavní data v atributu *items*. Navíc disponuje atributem *pageInfo*, který nese dvě následující pravdivostní hodnoty, které informují klienta, zda jsou další stránky k dispozici:

- hasNextPage,
- hasPreviousPage.

Příklady:

- **Prvních 10 filmů:**

```
query {
  movies(take: 10) {
    items{
      id
      name
      year
    },
    pageInfo{
      hasNextPage,
      hasPreviousPage
    }
  }
}
```

- **Dalších 10 filmů, bez prvních 10:**

```
query {
  movies(skip: 10, take: 10) {
    items{
      id
      name
      year
    },
    pageInfo{
      hasNextPage,
      hasPreviousPage
    }
}
```

Cursor stránkování

Cursor stránkování je pokročilejší způsob stránkování, který poskytuje stabilnější a konzistentnější výsledky, zejména v případě častých změn dat. Místo číselných indexů používá ukazatele pro identifikaci konkrétních záznamů v datasetu. Ukazetelé jsou generované hodnoty typu string.

Stejně jako u předchozího stránkování, i zde je odlišný návratový typ - *Connection*. Tento typ obsahuje hlavní data v atributu *nodes*. Atribut *pageInfo* obsahuje navíc ukazatele na první a poslední položku na aktuální stránce. Navíc disponuje atributem *edges*, který obahuje ukazatele na všechny záznamy na aktuální stránce.

Příklad:

- **Prvních 10 filmů po filmu s ukazatelem *CursorForMovieN48***

```
query {
  movies(first: 10, after: "CursorForMovieN48") {
    nodes{
      id
      name
      year
    },
    pageInfo{
      startCursor,
      endCursor
    },
    edges{
      cursor
    }
  }
}
```

4.3.3 Implementace

Implementace GraphQL s HotChocolate podporou se výrazně liší od implementací předchozích dvou přístupů. Jelikož výsledné API obsahuje jediný koncový bod, nevytváří se pro každou entitu samostatný kontroler. Namísto toho se vytvoří schéma obsahující *query* typ, případně i *mutation* typ. Všechny tyto termíny jsou vysvětleny v kapitole 3.2.1 Práce s GraphQL.

Nejprve je potřeba nainstalovat NuGet balíček do projektu. Základní knihovna, která je potřeba, je *HotChocolate.AspNetCore*. HotChocolate disponuje i dalšími knihovnami. Například pro lepší podporu databázových systému, jako jsou *HotChocolate.Data.EntityFramework* nebo *HotChocolate.Data.Mongo*.

Jako základní vstupní bod pro dotazování dat v GraphQL API slouží třída Query. Je to třída, která definuje dostupné dotazy pro klienty, které mohou využít k získání dat z API. Tato třída obvykle obsahuje metody, které reprezentují jednotlivé dotazy, a zpracovávají požadavky klientů na získání specifických dat.

Zdrojový kód 18 obsahuje třídu Query reprezentující stejnojmenný GraphQL typ. Třída poskytuje základní operaci pro získání seznamu herců s podporou různých funkcí, jako je stránkování, projekce, filtrování a řazení. Pro přidaní podpory různých možností dotazování k dotazům slouží v HotChocolate speciální atributy. Před jejich použitím, je nutné provést jejich registraci v konfiguraci.

```
public class Query
{
    [UsePaging]
    [UseProjection]
    [UseFiltering]
    [UseSorting]
    public IQueryable<Actor> GetActors(ImdbContext context)
        => context.Actors;
}
```

Zdrojový kód 18: Query typ v HotChocolate

Pro zaregistrování této třídy v konfiguraci HotChocolate serveru je potřeba ji přidat do konfigurace v souboru *Pogram.cs*. Pro registraci GraphQL serveru slouží metoda *AddGraphQLServer()* .

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddGraphQLServer()
    .AddQueryType<Query>()
    .AddProjections()
    .AddFiltering()
    .AddSorting();

var app = builder.Build();

app.MapGraphQL("/api");

app.Run();
```

Zdrojový kód 19: Konfigurace HotChocolate

Zdrojový kód 19 obsahuje soubor Program.cs se základní konfigurací GraphQL s HotChocolate podporou. Metoda *AddQueryType* zaregistruje query typ. Zdrojový kód obsahuje následující metody rozšíření pro přidání podpory možnosti dotazování na globální úrovni:

- **AddProjections**: používá se pro registraci podpory projekcí. Projekce umožňují klientům vyžádat pouze konkrétní podmnožinu vlastností,
- **AddFiltering**: používá se pro registraci podpory filtrování. Filtrování umožňuje klientům získávat výsledky dotazu na základě různých podmínek,
- **AddSorting**: používá se pro registraci podpory řazení. Řazení umožňuje klientům řadit výsledky dotazu podle určitých kritérií.

4.4 Srovnání dotazovacích jazyků

Všechny 3 přístupy poskytují bohatý jazyk pro dotazování, který umožňuje klientům vykonávat složité dotazy při získávání dat, jako je filtrování, řazení, stránkování a expandování souvisejících zdrojů. Umožňují klientům specifikovat, která data mají být vrácena v odpovědi, což minimalizuje množství přenesených dat. Každý z těchto jazyků má odlišnou syntaxi a flexibilitu, což ovlivňuje způsob, jakým klienti komunikují se serverem a jak je možné přizpůsobit dotazy pro různé požadavky.

Odeslání požadavku

Samotné odesílání požadavku serveru se liší v závislosti na tom, jakou architekturu podporuje daný dotazovací jazyk.

Například OData využívá REST architekturu a odesílá požadavky prostřednictvím standardních HTTP metod, jako jsou GET, POST, PUT, PATCH a DELETE. Požadavek na získání dat je odeslán pomocí HTTP GET metody s dotazovými parametry v URL pro specifikaci možnosti dotazování.

JSON:API také využívá REST architekturu, proto odesílá požadavky také prostřednictvím standardních HTTP metod, jen s odlišnou syntaxí.

HotChocolate oproti tomu využívá GraphQL architekturu a má tak vlastní způsob odesílání požadavků. Požadavek na získání dat je typicky odeslán prostřednictvím HTTP POST metody s dotazem GraphQL v těle požadavku, který obsahuje typ, vlastnosti, argumenty a proměnné pro specifikaci požadovaných dat.

V kontextu odeslání požadavků má HotChocolate vyšší úroveň flexibility a přehlednosti. Umožňuje složitější dotazy s mnoha parametry, jelikož ostatní přístupy neodesílají požadavky v těle požadavku, ale přímo v URL adrese, která má omezenou délku.

Navíc je výměna dat prostřednictví POST metody bezpečnější z pohledu ochrany citlivých dat.

4.4.1 Filtrování

Filtrování dat se liší v syntaxi, schopnostech a úrovni flexibility. Níže jsou uvedeny parametry, které se používají pro aplikování filtrování v jednotlivých dotazovacích jazycích:

- OData: *$filter*,
- JSON:API: *filter*,
- HotChocolate: *where*.

Každý dotazovací jazyk umožňuje základní i složitější operace pro porovnávání jednotlivých záznamů. Následující tabulky slouží jako rychlý přehled a pomáhají pochopit rozdíly mezi těmito jazyky pro běžné operace. Uvádí syntaxi

jednotlivých operací a pokud jazyk operaci nepodporuje, je v příslušné buňce znak lomítka (/).

Logické operace

Logické operátory jsou nezbytné pro vytváření složitějších dotazů, které kombinují různé podmínky. Následující tabulka 13 poskytuje srovnání syntaxe logických operátorů, jako jsou AND (a také), OR (nebo), NOT (negace) pro každý z těchto dotazovacích jazyků. V tabulce se objevuje výraz 3 tečky (...), který lze nahradit jakýmkoli validním výrazem daného dotazovacího jazyka.

Tabulka 13: Srovnání logických operátorů pro filtrování

Operátor	OData	JSON:API	HotChocolate
AND	... *and* ...	*and*(..., ...)	*and*: [{ ... }, { ... }]
OR	... *or* ...	*or*(..., ...)	*or*: [{ ... }, { ... }]
NOT	not ...	not(...)	/

Z tabulky 13 je patrné, že všechny tři dotazovací jazyky podporují logickou konjunkci a disjunkci, jen s mírnými odlišnostmi v syntaxi. Malá nevýhoda nastává u dotazovacího jazyka OData, kdy dochází k redundanci operátoru, když klient potřebuje zadat více argumentů dané operace. Při použití JSON:API nebo HotChocolate stačí přidat další podvýraz následovaný za čárkou ve výrazu.

Co se týče negace (NOT operátor), HotChocolate nepodporuje přímo NOT operátor pro filtrování. Místo toho lze použít alternativní způsob, jak dosáhnout negace, jako je nerovnost *neq* nebo jiné operátory, které jsou vysvětleny dále v kapitole.

Základní operace porovnávání

Tabulka 14: Srovnání základních operací pro filtrování

Operace	OData	JSON:API	HotChocolate
Rovnost	Name *eq* 'John'	*equals*(Name, 'John')	Name: { *eq*: "John" }
Nerovnost	Name *ne* 'John'	/	Name: { *neq*: "John" }
Větší než	Id *gt* 48	*greaterThan*(Id, 48)	Id: { *gt*: 48 }
Větší nebo rovno než	Id *ge* 48	*greaterOrEqual*(Id, 48)	Id: { *gte*: 48 }
Menší než	Id *lt* 48	*lessThan*(Id, 48)	Id: { *lt*: 48 }
Menší nebo rovno než	Id *le* 48	*lessOrEqual*(Id, 48)	Id: { *lte*: 48 }

V tabulce 14 je prezentováno srovnání syntaxe základních operací pro filtrování mezi OData, JSON:API a HotChocolate.

Všechny tři dotazovací jazyky používají různé syntaxe pro operandy ve svých operacích. OData používá klíčová slova, JSON:API používá funkce s parametry a HotChocolate používá objektovou notaci s klíčovými slovy.

Je důležité poznamenat, že JSON:API nepodporuje operaci nerovnosti přímo. Má však k dispozici operátor negace, který společně s operátorem rovnosti umožňuje dosáhnout nerovnosti.

V HotChocolate navíc existují následující operace pro negaci některých základních operací a není tedy nutné používat negaci explicitně.

- Není větší než: Id: { *ngt*: 48 },
- není větší nebo rovno než: Id: { *ngte*: 48 },
- není menší než: Id: { *nlt*: 48 },
- není menší nebo rovno než: Id: { *nlte*: 48 }.

Tyto operandy pro negování základních operací včetně nerovnosti zvyšují komplexnost dotazovacího jazyka a zvyšují tedy čas na jeho naučení. Na druhou stranu může být jejich použití snazší a zápis operace je kratší než při použití explicitního operátoru negace. Proto záleží především na preferencích vývojářů, které operace jsou pro ně vhodnější používat.

Textové operace

Textové operace jsou důležité pro získávání relevantních dat z velkých datasetů a umožňují uživatelům zúžit výsledky na základě konkrétních slov nebo frází.

Tabulka 15: Srovnání textových operací pro filtrování

Operace	OData	JSON:API	HotChocolate
Začíná	*startswith*(Name,'Jo')	*startswith*(Name,'Jo')	*startswith*: { Name: "Jo" }
Obsahuje	*contains*(Name,'oh')	*contains*(Name,'oh')	*contains*: { Name: "oh" }
Končí	*endsWith*(Name,'hn')	*endsWith*(Name,'hn')	*endsWith*: { Name: "hn" }

Všechny 3 dotazovací jazyky podporují textové operace, které se vyhodnocují na pravdivostní hodnoty. Tyto operace kontrolují, zda vlastnost entity začíná zadaným textem, zda text obsahuje nebo jestli textem končí. V tabulce 15 jsou tyto operace zobrazené. OData a JSON:API mají pro tyto operace připravené funkce, které mají v obou jazycích stejnou syntaxi. HotChocolate má odlišnou syntaxi, ve funkčnosti se však neliší.

HotChocolate navíc obsahuje pro všechny textové operace verze pro negaci, jelikož nemá externí operátor pro negaci:

- nezačíná: *nstartsWith*: { Name: "Jo" },
- neobsahuje: *ncontains*: { Name: "oh" },

- nekončí: *nendsWith*: { Name: "hn" }.

Ve většině aplikací jsou tyto textové operace dostatečné. OData však posunuje flexibilitu dotazovacího jazyka pro webové API na vyšší úroveň. Oproti zbylým dvou dotazovacím jazykům nabízí funkce jako jsou:

- konkatenace textů: *concat()*,
- délka textu: *length()*,
- část z textu: *substring()*,
- převod na malá písmena: *tolower()*,
- převod na velká písmena: *toupper()*
- a další.

Operace pro práci s kolekcemi

Operace pro práci s kolekcemi při filtrování jsou klíčové pro efektivní dotazování dat v one-to-many vztazích v databázi. Umožňují klientům filtrovat záznamy na základě vlastností souvisejících záznamů v kolekcích.

Obecně existují dvě operace pro filtrování dat na základě vnořených vlastností. Umožňují filtrovat záznamy na základě podmínek, které platí pro *alespoň jeden* nebo *všechny* související záznamy v kolekci.

OData podporuje pro tyto účely operace *any* a *all* ve formě lambda funkcí. Funkce, jejíž podmínka musí platit pro alespoň jeden záznam v kolekci, může vypadat například následovně. Příklad vybírá záznamy, které obsahují alespoň jednu roli s názvem *Olaf*:

- roles/any(x:x/RoleName eq 'Olaf').

Naproti tomu funkce jejíž podmínka musí platit pro všechny záznamy v kolekci může vypadat následovně. Příklad vybírá záznamy, které mají všechny role s názvem *Olaf*:

- roles/all(x:x/RoleName eq 'Olaf').

JSON:API podporuje jen jednu z těchto dvou operací a to ve formě funkce, jejíž podmínka musí platit alespoň pro jeden záznam v kolekci. Jedná se o funkci *has* a má následující tvar:

- has(roles,equals(roleName,'Olaf')).

HotChocolate, stejně jako OData podporuje obě operace. Operaci *some* (alespoň jeden) a *all* (všechny):

- roles: { some: { roleName: { eq: "Olaf"} } }),
- roles: { all: { roleName: { eq: "Olaf"} } }).

Ostatní operace

Pro většinu aplikací jsou operace zmíněné v této kapitole dostatečné pro efektivní dotazování a filtrování dat. Tyto základní operace pokrývají širokou škálu scénářů, které mohou být v běžných aplikacích požadovány. Nicméně, v některých případech mohou být potřeba další nebo specifické operace, které nejsou ve výchozím nastavení k dispozici.

Je důležité si uvědomit, že každý z dotazovacích jazyků - OData, JSON:API a HotChocolate - umožňuje implementovat dodatečné nebo specifické operace explicitně, pokud je to nutné. To znamená, že pokud je potřeba použít nějakou speciální funkcionalitu, která není ve výchozím nastavení k dispozici, lze si ji přizpůsobit a rozšířit možnosti dotazování podle potřeb.

I přesto má OData některé funkce, které ostatní dotazovací jazyky nemají. Některé z těchto funkcí zahrnují:

- aritmetické operace, které umožňují provádět základní aritmetické výpočty přímo v dotazu, jako jsou sčítání, odečítání, násobení a dělení,
- geografické a geometrické operace, které umožňují provádět dotazy a filtrace založené na geografických a geometrických údajích, jako jsou vzdálenosti, průsečíky a obsahy,
- textové operace, jako je vyhledávání podřetězců, porovnávání textu a manipulace s textem, což umožňuje provádět sofistikované textové dotazy a filtrace,
- podpora pro agregace a analytické funkce, které umožňují provádět pokročilé analýzy a zpracování dat přímo v dotazu.

Závěr

OData nabízí největší množství pokročilých funkcí a operací pro filtrování dat při dotazování, což může být užitečné pro složitější aplikace s velkým množstvím dat a s komplexními vztahy mezi entitami.

JSON:API a HotChocolate jsou oproti OData jednodušší. Ačkoli neposkytují tak širokou škálu pokročilých funkcí jako OData, oba jazyky nabízejí dostatečnou flexibilitu pro základní i pokročilé dotazování.

Někdy může být poskytnutí příliš mnoha funkcí klientovi považováno za nevýhodu, protože to může způsobit zvýšenou složitost a náročnost na výkon serveru. V takových případech mohou být jednodušší a více zaměřené jazyky, jako jsou JSON:API a HotChocolate vhodnější volbou.

Je třeba si uvědomit, že volba nejvhodnějšího dotazovacího jazyka závisí na konkrétních požadavcích aplikace a preferencích vývojářů. Některé aplikace mohou těžit z bohaté funkcionality a flexibility, kterou nabízí OData, zatímco jiné mohou dávat přednost jednoduchosti a efektivitě jazyků, jako jsou JSON:API a HotChocolate. Výběr nejlepšího dotazovacího jazyka tak závisí na analýze potřeb a očekávání vývojářů a uživatelů dané aplikace.

4.4.2 Projekce

Projekce umožňuje klientům požadovat specifické atributy entit a získávat data souvisejících entit podle jejich potřeb. I když každý dotazovací jazyk používá jiný přístup a syntaxi pro projekci, všechny poskytují možnost omezit množství vrácených dat, a také snížit počet dotazů.

OData umoňuje použít funkci *$select* pro specifikaci atributů, které mají být vráceny. Pro získání dat přidružených entit se používá funkce *$expand*, která umožňuje klientovi získat data těchto entit společně s hlavními daty.

JSON:API používá parametr *fields* pro specifikaci atributů, které mají být vráceny. Pro získání dat přidružených entit se používá parametr *include*, který umožňuje klientovi získat data těchto entit společně s hlavními daty.

HotChocolate využívající GraphQL syntaxi, umožňuje klientům přesně specifikovat, které atributy a vztahy mají být vráceny, a to přímo v těle dotazu. Tímto způsobem lze získat specifické kombinace dat podle potřeb klienta.

OData a JSON:API mají podobný přístup k projekcím, protože oba používají URL parametry pro specifikaci atributů a vztahů. Tyto syntaxe jsou jednoduché a snadno čitelné, což umožňuje vývojářům rychle a snadno implementovat projekce. Na druhou stranu, HotChocolate nabízí větší flexibilitu a kontrolu nad vrácenými daty díky GraphQL syntaxi, která umožňuje klientům přesně definovat, jaká data chtějí získat.

OData a JSON:API jsou vhodné pro aplikace, které vyžadují jednoduché a snadno čitelné projekce, zatímco HotChocolate je ideální pro aplikace, které vyžadují větší flexibilitu a kontrolu nad vrácenými daty.

4.4.3 Stránkování

Stránkování je klíčová funkce výkonného API, která umožňuje efektivně zpracovávat velké množství dat.

Tabulka 16: Srovnání možností stránkování

Funkce	OData	JSON:API	HotChocolate
Offset stránkování	Ano	Ano	Ano
Cursor stránkování	Ne	Ne	Ano
Metadata o stránkování	Ano	Ano	Ano

Offset stránkování

Offset stránkování je nejjednodušší a nejčastěji používaná metoda stránkování, která umožňuje specifikovat počet vrácených záznamů společně s počtem přeskočených záznamů. Tato metoda je podporována ve všech třech dotazovacích jazycích: OData, JSON:API a HotChocolate.

- **OData** používá parametr *$skip* pro určení počtu záznamů, které mají být přeskočeny, a textit$top pro omezení počtu vrácených záznamů.

- **JSON:API** používá parametr *page[offset]* pro určení počtu záznamů, které mají být přeskočeny, a parametr *page[limit]* pro omezení počtu vrácených záznamů.
- **HotChocolate** používá GraphQL syntaxe pro určení počtu záznamů, které mají být přeskočeny, prostřednictvím argumentu *skip* a omezení počtu vrácených záznamů pomocí argumentu *take*.

Offset stránkování má několik nevýhod, které jsou společné pro všechny tři dotazovací jazyky. Především může být pomalé, pokud jsou dotazy prováděny na velkých datových sadách, protože databáze musí přeskočit určitý počet záznamů před vrácením výsledků. Navíc offset stránkování může vést k nekonzistentním výsledkům, pokud dojde ke změně záznamů mezi jednotlivými dotazy na stránky.

Nicméně, offset stránkování je jednoduché na používání a poskytuje snadnou cestu pro získání konkrétní stránky záznamů, což může být vhodné pro jednoduché aplikace nebo prototypy.

V případě, že je offset stránkování nedostačující pro konkrétní případ, je možné zvážit použití alternativních metod stránkování, jako je cursor stránkování, které může poskytovat lepší výkon a konzistenci výsledků za cenu složitější aplikace.

Cursor stránkování

Cursor stránkování je metoda stránkování, která poskytuje výhody v oblasti výkonu a konzistence výsledků, zejména při práci s velkými datovými sadami nebo častými změnami záznamů. Místo použití číselného offsetu se tato metoda spoléhá na unikátní identifikátor nebo jiný jedinečný atribut záznamu, který slouží jako *kurzor* pro určení místa, odkud má stránkování pokračovat, případně kde má končit.

Ze tří srovnávaných dotazovacích jazyků pouze HotChocolate poskytuje nativní podporu pro cursor stránkování. OData a JSON:API nemají podporu pro tuto metodu, ale mohou být rozšířeny o vlastní implementace. Nicméně, implementace cursor stránkování může být náročná a složitá.

Metadata stránkování

Metadata stránkování se týká informací o stránkování, které jsou vráceny spolu s výsledky dotazu. Tyto informace mohou zahrnovat celkový počet záznamů, počet stránek, informace o následující stránce a další. Poskytování metadat o stránkování je užitečné pro klienty, kteří chtějí lépe pochopit strukturu výsledků nebo navigovat mezi stránkami s výsledky.

Všechny tři srovnávané dotazovací jazyky poskytují určitou formu metadat o stránkování, ačkoli se liší v rozsahu a způsobu, jakým jsou tato metadata poskytována:

- **OData** poskytuje metadata stránkování prostřednictvím dvou atributů v odpovědi. *@odata.nextLink* je odkaz na další stránku výsledků a je přítomen pouze, pokud existuje další stránka.

  Atribut *@odata.count* poskytuje celkový počet záznamů v kolekci a je vrácen pouze, pokud je k dotazu přidán parametr $count=true.

- **JSON:API** poskytuje metadata stránkování prostřednictvím hypertextových odkazů v sekci *links* v odpovědi. Tyto atributy zahrnují následující odkazy:

  - **self**: odkaz na aktuální stránku,
  - **first**: odkaz na první stránku,
  - **last**: odkaz na poslední stránku,
  - **prev**: odkaz na předchozí stránku, pokud nějaká existuje,
  - **next**: odkaz na další stránku, pokud nějaká existuje.

  Při stránkování jsou všechny tyto atributy vráceny v odpovědi. Klient si tedy nemůže zvolit, který z těchto atributů ho zajímá a ostatní z odpovědi vyřadit.

  Pro informaci o celkovém počtu záznamů v kolekci, slouží atribut *total* v sekci *meta*.

- **HotChocolate** poskytuje metadata stránkování prostřednictvím sekce *pageInfo*, která může obsahovat následující atributy:

  - **hasPreviousPage**: pravdivostní hodnota udávající, zda existuje předchozí stránka,
  - **hasNextPage**: pravdivostní hodnota udávající, zda existuje další stránka.

  Pro získání celkového počtu záznamů v kolekci slouží atribut *totalCount*.

Závěr

Všechny tři jazyky poskytují podporu pro offset stránkování, které je jednoduché na použití, nicméně má své nevýhody. Cursor stránkování je alternativní metodou, která poskytuje lepší výkon a konzistenci výsledků, ale je nativně podporována pouze v HotChocolate.

Všechny tři dotazovací jazyky poskytují metadata o stránkování, která umožňují klientům lépe pochopit strukturu výsledků a navigovat mezi stránkami výsledků. Avšak, rozsah a způsob poskytování těchto metadat se liší mezi jednotlivými jazyky.

Při volbě dotazovacího jazyka je třeba zvážit, která metoda stránkování nejlépe vyhovuje konkrétním potřebám a očekáváním. Pro jednoduché aplikace nebo prototypy může být offset stránkování dostačující, zatímco pro aplikace s velkými

datovými sadami nebo častými změnami záznamů může být vhodnější zvážit použití cursor stránkování a zvolit tak HotChocolate.

Je důležité zdůraznit, že srovnání vlastností a funkcí těchto dotazovacích jazyků nejsou jedinými faktory, které by měly být zohledněny při výběru vhodného dotazovacího jazyka pro aplikaci. Kromě těchto aspektů je také zásadní vzít v úvahu efektivitu z hlediska rychlosti a využití paměti. V další kapitole budou prováděny testy, které umožní porovnat jazyky z hlediska výkonu.

5 Výkonnostní testy

Tato kapitola je zaměřená na srovnání výkonu dotazovacích jazyků HotChocolate, OData a JSON:API. Cílem je zjistit, jaký vliv mají jednotlivé jazyky na rychlost zpracování dotazů, reakční dobu a využití systémových prostředků, jako je paměť. Pro měření těchto parametrů jsou využity dva nástroje.

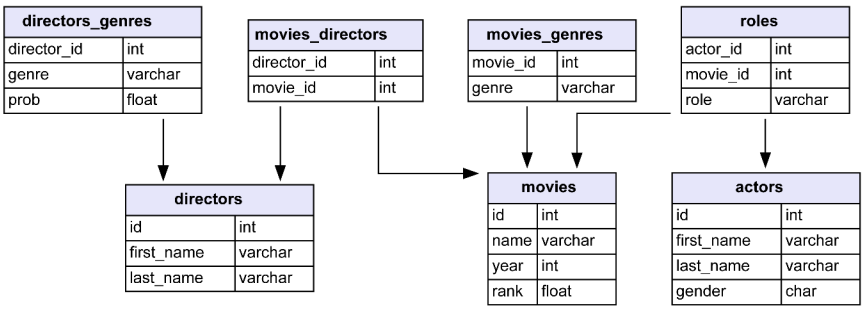
První nástroj, *k6* [6], je použit pro testování zátěže a provádění souběžných dotazů na testovací server. Tento nástroj poskytuje informace o rychlosti zpracování dotazů a reakční době na klientské straně, což umožňuje sledovat výkon serveru za různých podmínek.

Druhý nástroj, *Visual Studio Profiler* [7], je použit pro sledování a analýzu využití systémových prostředků na serverové straně, a to během těchto souběžných dotazů. Tento nástroj je schopen poskytnout detailní informace o výkonnosti a zatížení serveru, což přispívá k celkovému hodnocení výkonnosti dotazovacích jazyků.

Dotazovací jazyky jsou testovány při implementaci na platformě *.NET 6*. Jsou testovány nad stejnou Microsoft SQL databází, jejíž tabulky jsou mapovány ORM systémem *Entity Framework Core 6*.

Součástí této diplomové práce je implementace a záloha databáze pro vlastní testování, včetně návodu pro spuštění.

Databáze obsahuje tabulky a data reprezentující filmovou databázi. Byla převzata z [8] a je použita výhradně pro testovací účely. Obrázek 9 obsahuje vizualizaci této datové struktury, která je použita pro testování výkonu dotazovacích jazyků.



Obrázek 9: Databázová struktura pro testovací účely [8]

Databáze obsahuje následující modely:

- **Actor** - obsahuje informace o hercích, jako jsou jméno, příjmení a pohlaví. Každý herec má několik rolí v různých filmech,
- **Director** - obsahuje informace o režisérech, jako jsou jméno a příjmení. Každý režisér má několik filmů, které režíroval, a několik žánrů, ve kterých pracuje,
- **DirectorsGenre** - spojovací tabulka mezi režiséry a žánry, která udává preference režisérů pro jednotlivé žánry,
- **Movie** - obsahuje informace o filmech, jako jsou název, rok a hodnocení. Každý film má několik žánrů, rolí a režisérů,
- **MoviesGenre** - spojovací tabulka mezi filmy a žánry, která přiřazuje žánr k danému filmu,
- **MoviesDirector** - spojovací tabulka mezi filmy a režiséry, která přiřazuje režiséra k danému filmu,
- **Role** - spojovací tabulka mezi herci a filmy, která obsahuje informace o roli, kterou daný herec ztvárnil v daném filmu.

V rámci testování jsou prováděny pouze testy pro čtení dat. Důvodem je zaměření na srovnání efektivity dotazovacích jazyků při získávání dat z databáze. Testy jsou prováděny při co nejzákladnější konfiguraci dotazovacích jazyků, aby vracely data v co nejvíce podobné struktuře a minimalizovaly možný vliv konfigurace na výsledky testů.

Při testování výkonu dotazovacích jazyků jsou porovnávány zejména následující metriky, které jsou přehledně poskytnuty nástroji *k6* a *Visual Studio Profiler*:

- **data_received**: množství dat, které klient obdržel ze serveru,
- **data_sent**: množství dat odeslaných klientem na server,
- **http_req_duration**: průměrná doba od odeslání požadavku do doby, kdy klient obdržel odpověď,
- **max_memory**: maximální využití paměti během testu.

Pro testování výkonu dotazovacích jazyků jsou navrženy testovací scénáře, které zahrnují různé úrovně složitosti dotazů. Tyto scénáře zahrnují:

1. Dotaz na všechny filmy režiséra s daným jménem.
2. Dotaz na získání seznamu herců, kteří hráli ve filmech vydaných v roce 2000, a informace o jejich rolích a filmech.
3. Dotaz na získání seznamu herců, kteří hráli v alespoň jednom hororovém filmu s hodnocením vyšším než 9, který režíroval režisér s určitou preferencí pro komedie.

Ke každému scénáři jsou navrženy testy pro všechny dotazovací jazyky. Tyto scénáře jsou testovány nejprve jedním virtuálním uživatelem současně. Následně je vybrán jeden optimální scénář, jehož rozdíly ve výsledcích nejsou příliš velké, ale nejsou ani zanedbatelné. Tento scénář je podroben testu s více souběžně se dotazujícími virtuálními uživateli, čímž je dosaženo podrobnějšího porovnání výkonnosti jednotlivých dotazovacích jazyků.

5.1 Testování jedním uživatelem

V testech s jedním uživatelem je každý testovací scénář spuštěn 50krát pro každý dotazovací jazyk s použitím jednoho virtuálního uživatele. Tento typ testu je zaměřen na měření základního výkonu každého dotazovacího jazyka bez jakékoliv konkurence nebo zátěže z více paralelních dotazů.

5.1.1 1. scénář

Tento scénář má znění: *Dotaz na všechny filmy režiséra s daným jménem.*

Představuje jednoduchý dotaz, který vyžaduje pouze základní selekci a filtrování. Jelikož je tento dotaz považován za jednoduchý a přímočarý, očekává se, že výsledky výkonnosti budou podobné pro všechny dotazovací jazyky.

Pro tento dotaz je zvolen režisér se jménem *Steven Spielberg*.

OData

Odpovídající část URL adresy pro dotaz na server s implementací OData vypadá pro první scénář následovně:

```
/api/directors?$expand=movies&$filter=firstname eq 'Steven' and
    lastname eq 'Spielberg'&$select=movies
```

Skládá se z následujících částí:

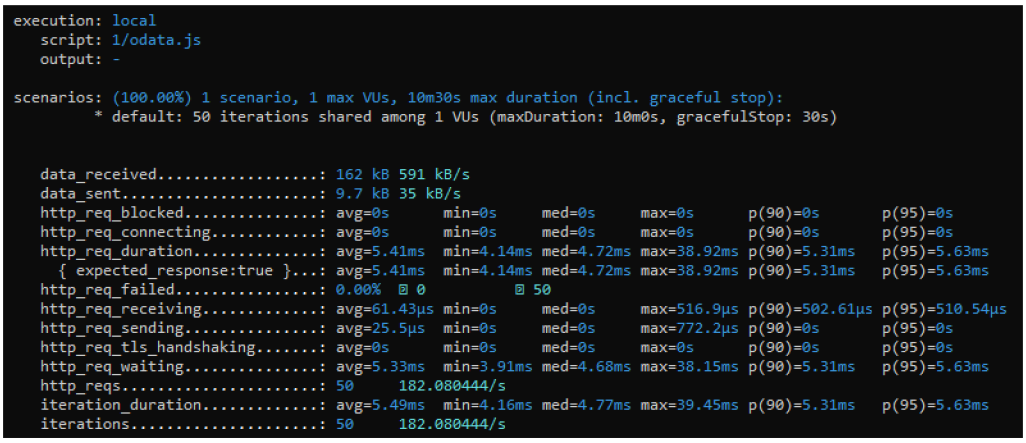
- **api/directors**: zdroj dat,
- **$expand=movies**: načtení souvisejících zdrojů,
- **$filter=firstname eq 'Steven' and lastname eq 'Spielberg'**: filtrování podle podmínky,
- **$select=movies**: omezení vrácených vlastností.

Výsledný SQL výraz generovaný službou OData pro tento scénář je prezentován v kódu 20.

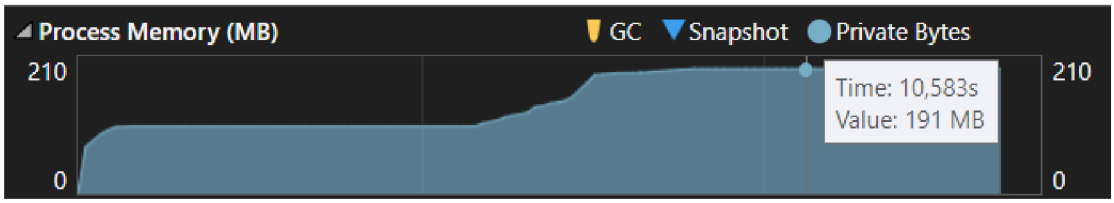
Obrázek 10 poskytuje náhled na testovací výsledky prvního scénáře pro OData získané pomocí *k6* s jedním testujícím uživatelem. Obrázek 11 pak zobrazuje výsledky shromážděné *Visual Studio Profilerem*.

```sql
SELECT [d].[id], [t].[id], [t].[name], [t].[rank], [t].[year], [t].[
    director_id], [t].[movie_id]
FROM [directors] AS [d]
LEFT JOIN (
    SELECT [m0].[id], [m0].[name], [m0].[rank], [m0].[year], [m].[
        director_id], [m].[movie_id]
    FROM [movies_directors] AS [m]
    INNER JOIN [movies] AS [m0] ON [m].[movie_id] = [m0].[id]
) AS [t] ON [d].[id] = [t].[director_id]
WHERE ([d].[first_name] = 'Steven') AND ([d].[last_name] = '
    Spielberg')
ORDER BY [d].[id], [t].[director_id], [t].[movie_id]
```

Zdrojový kód 20: SQL dotaz pro první scénář - OData



Obrázek 10: k6 - výsledky testu prvního scénáře při testování jedním uživatelem - OData



Obrázek 11: Visual Studio Profiler - využití paměti u prvního scénáře při testování jedním uživatelem - OData

JSON:API

Odpovídající část URL adresy pro dotaz na server s implementací JSON:API vypadá pro první scénář následovně:

```
/api/directors include=movies&filter=and(equals(firstName,'Steven'),
    equals(lastName,'Spielberg'))&fields[directors]=movies
```

Skládá se z následujících částí:

- **api/directors**: zdroj dat,
- **include=movies**: načtení souvisejících zdrojů,
- **filter=and(equals(firstName,'Steven'),equals(lastName,'Spielberg'))**: filtrování podle podmínky,
- **fields[directors]=movies**: omezení vrácených vlastností.

SQL dotaz odpovídající požadavku HTTP, který byl službou JSON:API vygenerován pro tento scénář, je zobrazen v kódu 21.

```
1  SELECT [d].[id], [t].[id], [t].[name], [t].[rank], [t].[year], [t].[
       director_id], [t].[movie_id]
2  FROM [directors] AS [d]
3  LEFT JOIN (
4      SELECT [m0].[id], [m0].[name], [m0].[rank], [m0].[year], [m].[
           director_id], [m].[movie_id]
5      FROM [movies_directors] AS [m]
6      INNER JOIN [movies] AS [m0] ON [m].[movie_id] = [m0].[id]
7  ) AS [t] ON [d].[id] = [t].[director_id]
8  WHERE ([d].[first_name] = 'Steven') AND ([d].[last_name] = '
       Spielberg')
9  ORDER BY [d].[id], [t].[id], [t].[director_id]
```

Zdrojový kód 21: SQL dotaz pro druhý scénář - JSON:API

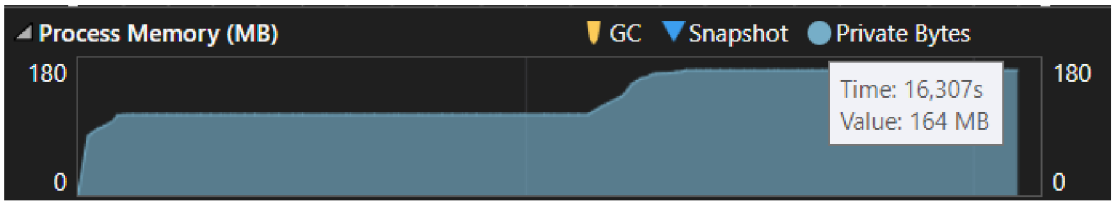
Obrázek 12 zobrazuje výsledky testu prvního scénáře, které byly získany nástrojem *k6*. Obrázek 13 obsahuje výsledky testu získaných nástrojem *Visual Studio Profiler*.

```
execution: local
   script: 1/jsonapi.js
   output: -

scenarios: (100.00%) 1 scenario, 1 max VUs, 10m30s max duration (incl. graceful stop):
        * default: 50 iterations shared among 1 VUs (maxDuration: 10m0s, gracefulStop: 30s)


   data_received..................: 306 kB 1.3 MB/s
   data_sent......................: 10 kB  42 kB/s
   http_req_blocked...............: avg=20.04µs min=0s     med=0s     max=1ms     p(90)=0s     p(95)=0s
   http_req_connecting............: avg=20.04µs min=0s     med=0s     max=1ms     p(90)=0s     p(95)=0s
   http_req_duration..............: avg=4.71ms  min=3.67ms med=4.21ms max=19.09ms p(90)=4.78ms p(95)=5.56ms
     { expected_response:true }...: avg=4.71ms  min=3.67ms med=4.21ms max=19.09ms p(90)=4.78ms p(95)=5.56ms
   http_req_failed................: 0.00%  ✓ 0        ✗ 50
   http_req_receiving.............: avg=21.45µs min=0s     med=0s     max=520.1µs p(90)=0s     p(95)=20.13µs
   http_req_sending...............: avg=1.93µs  min=0s     med=0s     max=65.19µs p(90)=0s     p(95)=7.25µs
   http_req_tls_handshaking.......: avg=0s      min=0s     med=0s     max=0s      p(90)=0s     p(95)=0s
   http_req_waiting...............: avg=4.69ms  min=3.67ms med=4.2ms  max=19.09ms p(90)=4.78ms p(95)=5.56ms
   http_reqs......................: 50     208.43008/s
   iteration_duration.............: avg=4.79ms  min=3.68ms med=4.24ms max=19.09ms p(90)=4.78ms p(95)=5.56ms
   iterations.....................: 50     208.43008/s
```

Obrázek 12: k6 - výsledky testu prvního scénáře při testování jedním uživatelem - JSON:API



Obrázek 13: Visual Studio Profiler - využití paměti u prvního scénáře při testování jedním uživatelem - JSON:API

HotChocolate

Tělo dotazu na server s implementací HotChocolate vypadá pro první scénář následovně:

```
query {
  directors(where: { and: [ { firstName: { eq: "Steven" } }, {
     lastName: { eq: "Spielberg" }}]}) {
    items{
        movies{
            id
            name
            year
            rank
        }
    }
  }
}
```

Skládá se z následujících částí:

- **query**: typ operace,
- **directors**: zdroj dat,
- **where: { and: [{ firstName: { eq: "Steven"} }, { lastName: { eq: "Spielberg"}}]}**: filtrování podle podmínky,
- **items {... }**: zahrnutí vlastností zdrojů do odpovědi.

HotChocolate překládá příchozí HTTP požadavek do SQL dotazu, jehož výsledek je zobrazen v kódu 22.

Obrázek 14 poskytuje přehled výsledků získaných jedním uživatelem pomocí *k6* pro první scénář testující HotChocolate. Pro další analýzu jsou v obrázku 15 zobrazeny výsledky získané pomocí *Visual Studio Profileru*.

```sql
SELECT [t].[id], [t0].[Id], [t0].[Name], [t0].[Year], [t0].[Rank], [
    t0].[director_id], [t0].[movie_id]
FROM (
    SELECT TOP(2147483647) [d].[id]
    FROM [directors] AS [d]
    WHERE ([d].[first_name] = 'Steven') AND ([d].[last_name] = '
        Spielberg')
) AS [t]
LEFT JOIN (
    SELECT [m0].[id] AS [Id], [m0].[name] AS [Name], [m0].[year] AS [
        Year], [m0].[rank] AS [Rank], [m].[director_id], [m].[
        movie_id]
    FROM [movies_directors] AS [m]
    INNER JOIN [movies] AS [m0] ON [m].[movie_id] = [m0].[id]
) AS [t0] ON [t].[id] = [t0].[director_id]
ORDER BY [t].[id], [t0].[director_id], [t0].[movie_id]
```

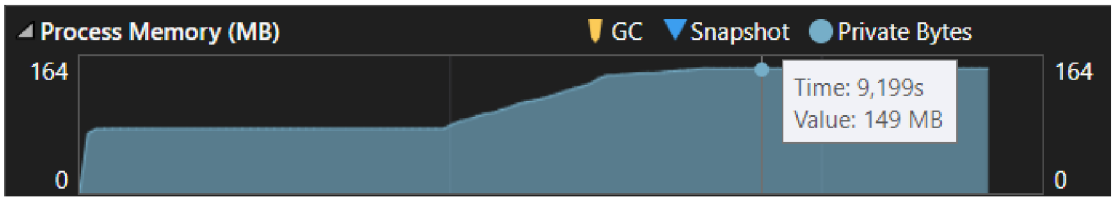
Zdrojový kód 22: SQL dotaz pro první scénář - HotChocolate

```
execution: local
   script: 1/hotchocolate.js
   output: -

scenarios: (100.00%) 1 scenario, 1 max VUs, 10m30s max duration (incl. graceful stop):
        * default: 50 iterations shared among 1 VUs (maxDuration: 10m0s, gracefulStop: 30s)


   data_received..................: 176 kB 777 kB/s
   data_sent......................: 22 kB  96 kB/s
   http_req_blocked...............: avg=115.91µs min=0s     med=0s     max=5.78ms   p(90)=0s       p(95)=0s
   http_req_connecting............: avg=0s       min=0s     med=0s     max=0s       p(90)=0s       p(95)=0s
   http_req_duration..............: avg=4.48ms   min=2.58ms med=4.21ms max=12.62ms  p(90)=5.21ms   p(95)=5.27ms
     { expected_response:true }...: avg=4.48ms   min=2.58ms med=4.21ms max=12.62ms  p(90)=5.21ms   p(95)=5.27ms
   http_req_failed................: 0.00%  ✓ 0        ✗ 50
   http_req_receiving.............: avg=73.61µs  min=0s     med=0s     max=537.29µs p(90)=517.74µs p(95)=520.83µs
   http_req_sending...............: avg=125.91µs min=0s     med=0s     max=5.78ms   p(90)=0s       p(95)=0s
   http_req_tls_handshaking.......: avg=0s       min=0s     med=0s     max=0s       p(90)=0s       p(95)=0s
   http_req_waiting...............: avg=4.28ms   min=2.58ms med=4.18ms max=6.87ms   p(90)=5.21ms   p(95)=5.25ms
   http_reqs......................: 50     221.40509/s
   iteration_duration.............: avg=4.51ms   min=2.58ms med=4.21ms max=12.62ms  p(90)=5.22ms   p(95)=5.53ms
   iterations.....................: 50     221.40509/s
```

Obrázek 14: k6 - výsledky testu prvního scénáře při testování jedním uživatelem - HotChocolate



Obrázek 15: Visual Studio Profiler - využití paměti u prvního scénáře při testování jedním uživatelem - HotChocolate

Porovnání výsledků

Všechny vygenerované SQL dotazy jsou si podobné, avšak s mírnými odlišnostmi.

1. **OData**: Tento dotaz je přímý a jeho struktura je jasná. Hlavní tabulka je *directors*, která je spojena s tabulkou *movies* prostřednictvím spojovací tabulky *movies_directors*. Dotaz vyhledává řádky, kde je jméno režiséra 'Steven' a příjmení 'Spielberg'. Výsledek je seřazen podle *id režiséra*, *director_id* a *movie_id*, což dělá jeden sloupec k seřazení redundantní.

2. **JSON:API**: Struktura tohoto dotazu je velmi podobná OData dotazu. Hlavní rozdíl je ve způsobu, jakým se výsledky seřadí. V tomto dotazu je řazení podle *id režiséra*, *id filmu* a *director_id*. Ale jelikož je zde znovu řazení podle režiséra redundantní, na výsledek to nemá vliv.

3. **HotChocolate**: Tento dotaz se liší od předchozích dvou v tom, že používá *TOP(2147483647)*. Je to díky tomu, že HotChocolate vyžaduje velikost stránky při definovaní query, proto je zde nakonfigurována výchozí velikost stránky na maximální int hodnotu. To je způsob, jak SQL Serveru říci, že chce vrátit všechny řádky, které odpovídají podmínkám WHERE. Na výkon vykonání dotazu by to však nemělo mít velký vliv. Tento dotaz také seřadí výsledky podle *id režiséra*, *director_id* a *movie_id*, což je stejné jako u OData dotazu.

Celkově lze říci, že všechny tři dotazy dosáhnou stejného výsledku, ale jejich výkon a čitelnost mohou být ovlivněny rozdílnými faktory. HotChocolate dotaz má zbytečně vynucené omezení výsledku, což však může být vhodné pro velké datové sady, kde by větší množství záznamů mohlo způsobovat výkonnostní problémy.

Tabulka 17 zobrazuje shrnutí výsledků testů jednotlivých dotazovacích jazyků pro první scénář. Jednotlivé metriky jsou vysvětleny v rámci představení v kapitole 5.

Tabulka 17: Výsledky testů prvního scénáře

	OData	JSON:API	HotChocolate
data_received	162 kB	306 kB	176 kB
data_sent	9,7 kB	10 kB	22 kB
http_req_duration	5,41 ms	4,71 ms	4,48 ms
max_memory	191 MB	164 MB	149 MB

Z hlediska dat přijatých klientem je OData nejefektivnější, s 162 kB dat, následována HotChocolate s 176 kB. JSON:API je naopak nejméně efektivní s 306 kB přijatých dat. To je důležité, protože všechny dotazovací jazyky poskytují stejné informace, respektive stejné záznamy se stejnými atributy, ale efektivnější jazyky je poskytnou v kompaktnějším balíčku, což znamená, že jsou schopny dodat stejnou hodnotu při menším zatížení sítě.

Pokud jde o data odeslaná klientem, opět se zdá, že HotChocolate je méně efektivní s 22 kB odeslaných dat, což je předpokládané, jelikož jsou data odeslaná POST metodou v těle dotazu. OData a JSON:API jsou efektivnější s hodnotami

9,7 kB a 10 kB, což znamená, že velikost URL je u OData dotazu lehce kratší. To je důležité, protože při zpracování dotazu je vysoká účinnost při odesílání dat do sítě rovněž přínosem.

Čas potřebný k provedení HTTP požadavku (http_req_duration) byl nejnižší u HotChocolate (4,48 ms), následuje JSON:API (4,71 ms) a OData (5,41 ms). Ačkoli rozdíly nejsou obrovské, tyto milisekundy mohou být významné v kontextu velkých aplikací nebo při vysoké frekvenci požadavků.

Pokud jde o spotřebu paměti, HotChocolate opět exceloval s nejnižšími nároky na paměť (149 MB), zatímco OData měla největší spotřebu paměti (191 MB). JSON:API se umístil mezi oběma s hodnotou 164 MB.

Tyto výsledky ukazují, že ačkoli existují rozdíly v efektivitě jednotlivých dotazovacích jazyků, žádný z nich není výrazně lepší nebo horší než ostatní ve všech ohledech. Výběr nejvhodnějšího jazyka tak bude záviset na konkrétních potřebách a omezeních každého jednotlivého projektu.

5.1.2 2. scénář

Tento scénář má znění: *Dotaz na získání seznamu herců, kteří hráli ve filmech vydaných v roce 2000, a informace o jejich rolích a filmech.*

Představuje mírně pokročilý dotaz, který vyžaduje kombinaci více parametrů a filtrování podle více kritérií. Tento dotaz je složitější než základní dotaz a může otestovat výkonnost a schopnosti dotazovacího jazyka při zpracování komplexnějších požadavků. Výsledky výkonnosti mohou být různé pro různé dotazovací jazyky, v závislosti na jejich schopnosti efektivně zpracovat a vrátit požadované výsledky.

Jelikož databáze pro testovací účely obsahuje mnoho dat, je pro tento scénář omezen počet vrácených záznamů na *10 000*. Tímto omezením je zaručeno, že nedojde k překročení maximální povolené velikosti HTTP odpovědi. Současně data zůstanou v dostatečné velikosti k vyhodnocení výsledku testů.

OData

Odpovídající část URL adresy pro dotaz na server s implementací OData vypadá pro druhý scénář následovně:

```
/api/actors?$expand=Roles($expand=Movie)&$filter=Roles/any(r: r/
    Movie/Year eq 2000)&$top=10000
```

Skládá se z následujících částí:

- **api/actors**: zdroj dat,
- **$expand=Roles($expand=Movie)**: načtení souvisejících zdrojů,
- **$filter=Roles/any(r: r/Movie/Year eq 2000)**: filtrování podle podmínky,
- **$top=10000**: omezení počtu záznamů.

Kód 23 prezentuje SQL dotaz vygenerovaný službou OData v reakci na HTTP požadavek pro druhý scénář.

```
SELECT [t].[id], [t].[first_name], [t].[gender], [t].[last_name], [
    t0].[Id], [t0].[actor_id], [t0].[movie_id], [t0].[role], [t0].[
    id0], [t0].[name], [t0].[rank], [t0].[year], [t0].[c]
FROM (
    SELECT TOP(10000) [a].[id], [a].[first_name], [a].[gender], [a].[
        last_name]
    FROM [actors] AS [a]
    WHERE EXISTS (
        SELECT 1
        FROM [roles] AS [r]
        INNER JOIN [movies] AS [m] ON [r].[movie_id] = [m].[id]
        WHERE ([a].[id] = [r].[actor_id]) AND ([m].[year] = 2000))
    ORDER BY [a].[id]
) AS [t]
LEFT JOIN (
    SELECT [r0].[Id], [r0].[actor_id], [r0].[movie_id], [r0].[role],
        [m0].[id] AS [id0], [m0].[name], [m0].[rank], [m0].[year],
        CAST(0 AS bit) AS [c]
    FROM [roles] AS [r0]
    INNER JOIN [movies] AS [m0] ON [r0].[movie_id] = [m0].[id]
) AS [t0] ON [t].[id] = [t0].[actor_id]
ORDER BY [t].[id], [t0].[Id]
```

Zdrojový kód 23: SQL dotaz pro druhý scénář - OData

Obrázek 16 zobrazuje výsledky testu druhého scénáře, získaných nástrojem *k6* při testování jedním uživatelem. Obrázek 17 obsahuje výsledky testu získaných nástrojem *Visual Studio Profiler*.

```
execution: local
   script: 2/odata.js
   output: -

scenarios: (100.00%) 1 scenario, 1 max VUs, 10m30s max duration (incl. graceful stop):
        * default: 50 iterations shared among 1 VUs (maxDuration: 10m0s, gracefulStop: 30s)


   data_received..................: 868 MB 2.2 MB/s
   data_sent......................: 9.0 kB 23 B/s
   http_req_blocked...............: avg=10.09µs min=0s       med=0s       max=504.9µs  p(90)=0s       p(95)=0s
   http_req_connecting............: avg=0s      min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
   http_req_duration..............: avg=7.95s   min=7.53s    med=7.81s    max=9.2s     p(90)=8.61s    p(95)=8.85s
     { expected_response:true }...: avg=7.95s   min=7.53s    med=7.81s    max=9.2s     p(90)=8.61s    p(95)=8.85s
   http_req_failed................: 0.00%  ✓ 0        ✗ 50
   http_req_receiving.............: avg=7.73s   min=7.32s    med=7.58s    max=8.97s    p(90)=8.37s    p(95)=8.63s
   http_req_sending...............: avg=10.65µs min=0s       med=0s       max=515.7µs  p(90)=0s       p(95)=0s
   http_req_tls_handshaking.......: avg=0s      min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
   http_req_waiting...............: avg=217.7ms min=191.21ms med=209.69ms max=452.74ms p(90)=240.24ms p(95)=246.1ms
   http_reqs......................: 50     0.125702/s
   iteration_duration.............: avg=7.95s   min=7.53s    med=7.81s    max=9.2s     p(90)=8.61s    p(95)=8.85s
   iterations.....................: 50     0.125702/s
   vus............................: 1      min=1      max=1
   vus_max........................: 1      min=1      max=1
```

Obrázek 16: k6 - výsledky testu druhého scénáře při testování jedním uživatelem - OData

Obrázek 17: Visual Studio Profiler - využití paměti u druhého scénáře při testování jedním uživatelem - OData

JSON:API

Odpovídající část URL adresy pro dotaz na server s implementací JSON:API vypadá pro druhý scénář následovně:

```
/api/actors?include=roles.movie&filter=has(roles,equals(movie.year
    ,'2000'))&page[size]=10000
```

Skládá se z následujících částí:

- **api/actors**: zdroj dat,
- **include=roles.movie**: načtení souvisejících zdrojů,
- **filter=has(roles,equals(movie.year,'2000'))**: filtrování podle podmínky,
- **page[size]=10000**: omezení počtu záznamů.

Kód 24 ukazuje výsledný SQL výraz, který je službou JSON:API přeložený při vykonávání HTTP požadavku pro tento scénář.

Při tomto testovacím scénáři, který se zaměřoval na zpracování velkých množství dat, se JSON:API nedokázal úspěšně vyrovnat s požadavky a výsledkem byla chyba *HTTP request timeout*. To mohlo být způsobeno mnoha faktory, avšak nejpravděpodobnější je, že JSON:API nebylo dostatečně optimalizované pro práci s velkými objemy dat. Tento výsledek slouží jako připomenutí, že při výběru přístupu a jeho implementace je důležité zohlednit konkrétní potřeby a omezení daného projektu.

```sql
SELECT [t].[id], [t].[first_name], [t].[gender], [t].[last_name], [
    t0].[Id], [t0].[ActorId], [t0].[MovieId], [t0].[RoleName], [t0
    ].[id0], [t0].[name], [t0].[rank], [t0].[year]
FROM (
    SELECT TOP(10000) [a].[id], [a].[first_name], [a].[gender], [a].[
        last_name]
    FROM [actors] AS [a]
    WHERE EXISTS (
        SELECT 1
        FROM [roles] AS [r]
        INNER JOIN [movies] AS [m] ON [r].[movie_id] = [m].[id]
        WHERE ([a].[id] = [r].[actor_id]) AND ([m].[year] = 2000))
    ORDER BY [a].[id]
) AS [t]
LEFT JOIN (
    SELECT [r0].[Id], [r0].[actor_id] AS [ActorId], [r0].[movie_id]
        AS [MovieId], [r0].[role] AS [RoleName], [m0].[id] AS [id0],
        [m0].[name], [m0].[rank], [m0].[year]
    FROM [roles] AS [r0]
    INNER JOIN [movies] AS [m0] ON [r0].[movie_id] = [m0].[id]
) AS [t0] ON [t].[id] = [t0].[ActorId]
ORDER BY [t].[id], [t0].[Id]
```

Zdrojový kód 24: SQL dotaz pro druhý scénář - JSON:API

HotChocolate

Tělo dotazu na server s implementací HotChocolate vypadá pro druhý scénář následovně:

```
query {
  actors(where: { roles: {some: {movie: {year: {eq: 2000 }}}}},
         take: 10000) {
    items{
        id
        firstName
        lastName
        gender
        roles{
            id
            roleName
            movie{
                id
                name
                year
                rank
            }
        }
    }
  }
}
```

Skládá se z následujících částí:

- **query**: typ operace,
- **actors**: zdroj dat,
- **where: {roles: {some: {movie: {year: {eq: 2000 }}}}}**: filtrování podle podmínky,
- **items {... }**: zahrnutí vlastností zdrojů do odpovědi,
- **take: 10000**: omezení počtu záznamů.

Výsledný SQL výraz, který odpovídá HTTP požadavku vykonanému pomocí služby HotChocolate, je uveden v kódu 25.

```
SELECT [t].[id], [t].[first_name], [t].[last_name], [t].[gender], [
    t0].[Id], [t0].[role], [t0].[c], [t0].[id0], [t0].[name], [t0].[
    year], [t0].[rank]
FROM (
    SELECT TOP(10000) [a].[id], [a].[first_name], [a].[last_name], [a
        ].[gender]
    FROM [actors] AS [a]
    WHERE EXISTS (
        SELECT 1
        FROM [roles] AS [r]
        INNER JOIN [movies] AS [m] ON [r].[movie_id] = [m].[id]
        WHERE ([a].[id] = [r].[actor_id]) AND ([m].[year] = 2000))
) AS [t]
LEFT JOIN (
    SELECT [r0].[Id], [r0].[role], CAST(1 AS bit) AS [c], [m0].[id]
        AS [id0], [m0].[name], [m0].[year], [m0].[rank], [r0].[
        actor_id]
    FROM [roles] AS [r0]
    INNER JOIN [movies] AS [m0] ON [r0].[movie_id] = [m0].[id]
) AS [t0] ON [t].[id] = [t0].[actor_id]
ORDER BY [t].[id], [t0].[Id]
```

Zdrojový kód 25: SQL dotaz pro druhý scénář - HotChocolate

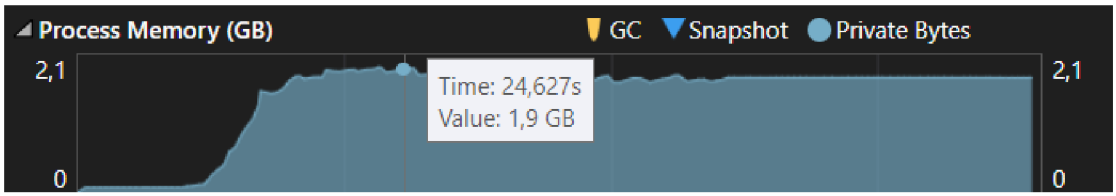
Obrázek 18 ilustruje výsledky druhého scénáře získané jedním uživatelem testujícím HotChocolate pomocí *k6*. Další výsledky získané nástrojem *Visual Studio Profileru* jsou ukázány v Obrázku 19.

```
execution: local
   script: 2/hotchocolate.js
   output: -

scenarios: (100.00%) 1 scenario, 1 max VUs, 10m30s max duration (incl. graceful stop):
         * default: 50 iterations shared among 1 VUs (maxDuration: 10m0s, gracefulStop: 30s)


   data_received..................: 721 MB 12 MB/s
   data_sent......................: 29 kB  497 B/s
   http_req_blocked...............: avg=14.77µs  min=0s       med=0s       max=738.7µs  p(90)=0s       p(95)=0s
   http_req_connecting............: avg=14.77µs  min=0s       med=0s       max=738.7µs  p(90)=0s       p(95)=0s
   http_req_duration..............: avg=1.18s    min=884.44ms med=975ms    max=2.82s    p(90)=1.61s    p(95)=2.33s
     { expected_response:true }...: avg=1.18s    min=884.44ms med=975ms    max=2.82s    p(90)=1.61s    p(95)=2.33s
   http_req_failed................: 0.00%  ▒ 0        ▒ 50
   http_req_receiving.............: avg=129.36ms min=116.96ms med=128.62ms max=151.27ms p(90)=138.24ms p(95)=140.85ms
   http_req_sending...............: avg=20.49µs  min=0s       med=0s       max=513.4µs  p(90)=0s       p(95)=0s
   http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
   http_req_waiting...............: avg=1.05s    min=759.7ms  med=841.17ms max=2.68s    p(90)=1.47s    p(95)=2.19s
   http_reqs......................: 50     0.845673/s
   iteration_duration.............: avg=1.18s    min=884.44ms med=975ms    max=2.82s    p(90)=1.61s    p(95)=2.33s
   iterations.....................: 50     0.845673/s
   vus............................: 1      min=1      max=1
   vus_max........................: 1      min=1      max=1
```

Obrázek 18: k6 - výsledky testu druhého scénáře při testování jedním uživatelem - HotChocolate



Obrázek 19: Visual Studio Profiler - využití paměti u druhého scénáře při testování jedním uživatelem - HotChocolate

Porovnání výsledků

Všechny tři dotazy jsou podobné, ale existují určité rozdíly, které mohou ovlivnit výkon, čitelnost a pořadí výsledků.

1. **OData**: Tento dotaz používá *EXISTS* subdotaz k filtrování herců, kteří hráli ve filmech vydaných v roce 2000, což je velmi efektivní způsob. Dotaz seřadí výsledky podle *id herce a id role*. To může být důležité pro pořadí výsledků. Tento dotaz také vrátí *CAST(0 AS bit) AS [c]*, což vypadá jako uměle přidaný sloupec.

2. **JSON:API**: Struktura tohoto dotazu je velmi podobná OData dotazu. Hlavní rozdíl je v názvech sloupců výsledného dotazu, kde jsou použita velká písmena na začátku názvů (ActorId, MovieId, RoleName). To neovlivňuje výkon dotazu, ale může to ovlivnit způsob, jakým je dotaz čitelný.

3. **HotChocolate**: Tento dotaz je také podobný jako předchozí dva. Stejně jako OData dotaz, také přidává umělý sloupec *CAST(1 AS bit) AS [c]*.

Všechny tři dotazy dosáhnou stejného výsledku. Hlavní rozdíly mezi nimi jsou v čitelnosti (kvůli rozdílným názvům sloupců) a v tom, jak přidávají umělé sloupce. Tyto sloupce nejsou potřebné pro výslednou funkčnost dotazu a je možné, že byly přidány kvůli specifickým požadavkům nebo omezením dotazovacích jazyků, které generují tyto SQL dotazy.

Co se týče výkonu, všechny tři dotazy by měly mít podobný výkon, protože mají podobnou strukturu a používají stejné techniky pro filtrování a spojování dat. Nicméně v závislosti na konkrétních okolnostech, jako je velikost datové sady a schopnosti databázového systému, by mohl být výkon jednoho dotazu lepší než ostatních.

Tabulka 18 ukazuje souhrn výsledků testů jednotlivých dotazovacích jazyků pro druhý scénář. Jednotlivé metriky jsou vysvětleny v rámci představení v kapitole 5.

Tabulka 18: Výsledky testů druhého scénáře

	OData	JSON:API	HotChocolate
data_received	868 MB	/	721 MB
data_sent	9,0 kB	/	29 kB
http_req_duration	7,95 s	/	1,18 s
max_memory	1,6 G	/	1,9 G

Druhý testovací scénář ukázal některé zajímavé rozdíly jen mezi dotazovacími jazyky OData a HotChocolate, jelikož JSON:API nebyl schopen tento test vykonat. Nejspíš je to díky tomu, že je tento test navržen pro zpracování velkých množství data, a protože JSON:API není příliš efektivní, co se týče dat odeslaných ze serveru, neustál takové množství.

V oblasti přijatých dat byl HotChocolate efektivnější s přijetím 721 MB dat, což je oproti OData, kde klient přijal 868 MB dat, výrazně méně. Stejně jako v předchozím scénáři, oba jazyky poskytují stejné informace, takže menší množství dat znamená lepší efektivitu.

Pokud jde o odesílaná data, OData byl efektivnější s odesláním 9,0 kB dat, zatímco klient na HotChocolate server odeslal 29 kB dat. Toto je důležité, protože menší množství odesílaných dat znamená menší zatížení sítě a tedy i efektivnější komunikaci.

V oblasti doby trvání HTTP požadavku byl HotChocolate výrazně rychlejší, jeho požadavek trval pouze 1,18 sekund, což je oproti OData, jehož požadavek trval 7,95 sekund, mnohem kratší doba. Tato rychlejší doba odezvy může mít významný dopad na výkon aplikace, zejména při vysokém počtu dotazů.

Pokud jde o spotřebu paměti, oba dotazovací jazyky byly poměrně náročné, HotChocolate spotřeboval 1,9 GB a OData 1,6 GB paměti. Toto jsou poměrně vysoké hodnoty, ale je důležité mít na paměti, že oba jazyky byly testovány na velkém objemu dat.

Tyto výsledky ukazují, že HotChocolate byl ve druhém testovacím scénáři obecně efektivnější než OData, a to zejména co se týče rychlosti zpracování do-

tazu. Jelikož JSON:API neumožnil tento scénář otestovat, končí na posledním místě.

5.1.3 3. scénář

Tento scénář má znění: *Dotaz na získání seznamu herců, kteří hráli v alespoň jednom hororovém filmu s hodnocením vyšším než 9, který režíroval režisér s určitou preferencí pro komedie.*

Představuje komplexní dotaz, který vyžaduje pokročilé funkce dotazovacího jazyka, jako je propojení více entit, složité filtrování a využití funkcí. Tento dotaz je nejnáročnější na výkonnost a schopnosti dotazovacího jazyka a slouží k otestování jeho schopnosti zvládnout složité a náročné úlohy. Výsledky výkonnosti mohou výrazně variovat v závislosti na efektivitě a optimalizaci konkrétního dotazovacího jazyka.

OData

Odpovídající část URL adresy pro dotaz na server s implementací OData vypadá pro třetí scénář následovně:

```
/api/actors?&$filter=Roles/any(r:r/Movie/Rank gt 9 and r/Movie/
    MoviesGenres/any(g:g/Genre eq 'Horror') and r/Movie/Directors/
    any(d:d/DirectorsGenres/any(g:g/Genre eq 'Comedy')))
```

Skládá se z následujících částí:

- **api/actors**: zdroj dat,
- **&$filter=Roles/any(r: ...)**: filtrování rolí,
- **r/Movie/Rank gt 9**: filtrování filmů dle hodnocení,
- **r/Movie/MoviesGenres/any(g:g/Genre eq 'Horror')**: filtrování filmů dle žánrů,
- **r/Movie/Directors/any(d:d/DirectorsGenres/any(g:g/Genre eq 'Comedy'))**: filtrování filmů dle žánrové preference režiséra.

Kód 26 zobrazuje SQL dotaz, který byl vygenerován službou OData pro tento konkrétní scénář.

Obrázek 20 zobrazuje data třetího scénáře získaná jedním testujícím uživatelem pro OData pomocí *k6*. Výsledky *Visual Studio Profileru* jsou k dispozici v obrázku 21.

```sql
SELECT [a].[id], [a].[first_name], [a].[gender], [a].[last_name]
FROM [actors] AS [a]
WHERE EXISTS (
    SELECT 1
    FROM [roles] AS [r]
    INNER JOIN [movies] AS [m] ON [r].[movie_id] = [m].[id]
    WHERE ([a].[id] = [r].[actor_id]) AND ((([m].[rank] > 9) AND
        EXISTS (
        SELECT 1
        FROM [movies_genres] AS [m0]
        WHERE ([m].[id] = [m0].[movie_id]) AND ([m0].[genre] ='Horror'
            ))) AND EXISTS (
        SELECT 1
        FROM [movies_directors] AS [m1]
        INNER JOIN [directors] AS [d] ON [m1].[director_id] = [d].[id]
        WHERE ([m].[id] = [m1].[movie_id]) AND EXISTS (
            SELECT 1
            FROM [directors_genres] AS [d0]
            WHERE ([d].[id] = [d0].[director_id]) AND ([d0].[genre] = '
                Comedy')))))
```

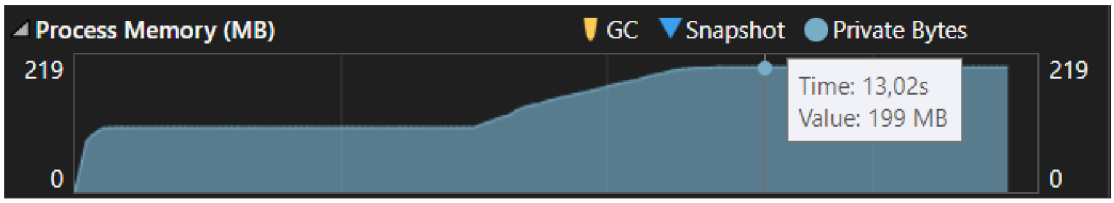
Zdrojový kód 26: SQL dotaz pro třetí scénář - OData

```
execution: local
   script: 3/odata.js
   output: -

scenarios: (100.00%) 1 scenario, 1 max VUs, 10m30s max duration (incl. graceful stop):
        * default: 50 iterations shared among 1 VUs (maxDuration: 10m0s, gracefulStop: 30s)


   data_received..................: 383 kB 132 kB/s
   data_sent......................: 14 kB  4.7 kB/s
   http_req_blocked...............: avg=24.81µs min=0s      med=0s      max=1.24ms   p(90)=0s       p(95)=0s
   http_req_connecting............: avg=19.95µs min=0s      med=0s      max=997.9µs  p(90)=0s       p(95)=0s
   http_req_duration..............: avg=58.13ms min=51.56ms med=57.36ms max=71.02ms  p(90)=62.72ms  p(95)=64.86ms
     { expected_response:true }...: avg=58.13ms min=51.56ms med=57.36ms max=71.02ms  p(90)=62.72ms  p(95)=64.86ms
   http_req_failed................: 0.00%  ✓ 0         ✗ 50
   http_req_receiving.............: avg=73.19µs min=0s      med=0s      max=545.4µs  p(90)=514.59µs p(95)=520.14µs
   http_req_sending...............: avg=35.1µs  min=0s      med=0s      max=504.49µs p(90)=0s       p(95)=386.27µs
   http_req_tls_handshaking.......: avg=0s      min=0s      med=0s      max=0s       p(90)=0s       p(95)=0s
   http_req_waiting...............: avg=58.02ms min=50.54ms med=57.36ms max=71.02ms  p(90)=62.72ms  p(95)=64.59ms
   http_reqs......................: 50     17.164369/s
   iteration_duration.............: avg=58.22ms min=51.56ms med=57.37ms max=71.02ms  p(90)=63.18ms  p(95)=65.15ms
   iterations.....................: 50     17.164369/s
   vus............................: 1      min=1       max=1
   vus_max........................: 1      min=1       max=1
```

Obrázek 20: k6 - výsledky testu třetího scénáře při testování jedním uživatelem - OData



Obrázek 21: Visual Studio Profiler - využití paměti u třetího scénáře při testování jedním uživatelem - OData

JSON:API

Odpovídající část URL adresy pro dotaz na server s implementací JSON:API vypadá pro druhý scénář následovně:

```
/api/actors?filter=has(roles,and(greaterThan(movie.rank,'9'),has(
    movie.moviesGenres,equals(genre,'Horror')),has(movie.directors,
    has(directorsGenres,equals(genre,'Comedy')))))
```

Skládá se z následujících částí:

- **api/actors**: zdroj dat,
- **has(roles, ...)**: filtrování rolí,
- **greaterThan(movie.rank,'9')**: filtrování filmů dle hodnocení,
- **has(movie.moviesGenres,equals(genre,'Horror'))**: filtrování filmů dle žánrů,
- **has(movie.directors,has(directorsGenres,equals(genre,'Comedy')))**: filtrování filmů dle žánrové preference režiséra.

Kód 27 představuje SQL dotaz, který vznikl překladem HTTP požadavku službou JSON:API.

Obrázek 22 demonstruje výsledky třetího scénáře pro JSON:API, získané pomocí *k6* jedním uživatelem. Obrázek 23 dále poskytuje výsledky *Visual Studio Profileru*.

```
SELECT [a].[id], [a].[first_name], [a].[gender], [a].[last_name]
FROM [actors] AS [a]
WHERE EXISTS (
    SELECT 1
    FROM [roles] AS [r]
    INNER JOIN [movies] AS [m] ON [r].[movie_id] = [m].[id]
    WHERE ([a].[id] = [r].[actor_id]) AND ((([m].[rank] > 9) AND
        EXISTS (
        SELECT 1
        FROM [movies_genres] AS [m0]
        WHERE ([m].[id] = [m0].[movie_id]) AND ([m0].[genre] = 'Horror
            ''))) AND EXISTS (
        SELECT 1
        FROM [movies_directors] AS [m1]
        INNER JOIN [directors] AS [d] ON [m1].[director_id] = [d].[id]
        WHERE ([m].[id] = [m1].[movie_id]) AND EXISTS (
            SELECT 1
            FROM [directors_genres] AS [d0]
            WHERE ([d].[id] = [d0].[director_id]) AND ([d0].[genre] = '
                Comedy'')))))
ORDER BY [a].[id]
```

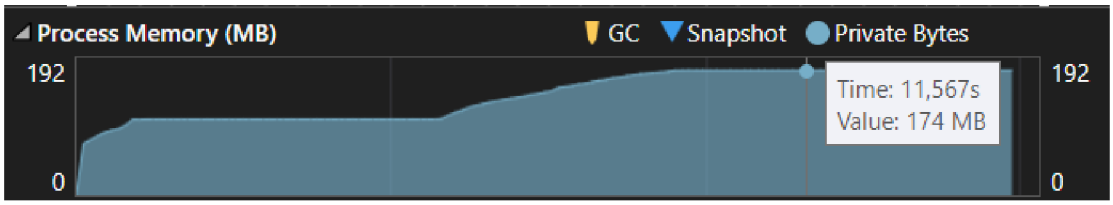
Zdrojový kód 27: SQL dotaz pro třetí scénář - JSON:API

```
execution: local
   script: 3/jsonapi.js
   output: -

scenarios: (100.00%) 1 scenario, 1 max VUs, 10m30s max duration (incl. graceful stop):
        * default: 50 iterations shared among 1 VUs (maxDuration: 10m0s, gracefulStop: 30s)


   data_received..................: 554 kB 225 kB/s
   data_sent......................: 13 kB  5.2 kB/s
   http_req_blocked...............: avg=20.05µs min=0s      med=0s      max=1ms     p(90)=0s      p(95)=0s
   http_req_connecting............: avg=0s      min=0s      med=0s      max=0s      p(90)=0s      p(95)=0s
   http_req_duration..............: avg=49.08ms min=46.67ms med=48.36ms max=56.41ms p(90)=51.58ms p(95)=54.79ms
     { expected_response:true }...: avg=49.08ms min=46.67ms med=48.36ms max=56.41ms p(90)=51.58ms p(95)=54.79ms
   http_req_failed................: 0.00%  ✓ 0         ✗ 50
   http_req_receiving.............: avg=43.36µs min=0s      med=0s      max=529.8µs p(90)=7.26µs  p(95)=520.76µs
   http_req_sending...............: avg=10.38µs min=0s      med=0s      max=519µs   p(90)=0s      p(95)=0s
   http_req_tls_handshaking.......: avg=0s      min=0s      med=0s      max=0s      p(90)=0s      p(95)=0s
   http_req_waiting...............: avg=49.03ms min=46.67ms med=48.36ms max=55.88ms p(90)=51.52ms p(95)=54.79ms
   http_reqs......................: 50     20.325678/s
   iteration_duration.............: avg=49.18ms min=47.07ms med=48.42ms max=56.41ms p(90)=51.58ms p(95)=54.97ms
   iterations.....................: 50     20.325678/s
   vus............................: 1      min=1       max=1
   vus_max........................: 1      min=1       max=1
```

Obrázek 22: k6 - výsledky testu třetího scénáře při testování jedním uživatelem - JSON:API



Obrázek 23: Visual Studio Profiler - využití paměti u třetího scénáře při testování jedním uživatelem - JSON:API

HotChocolate

Tělo dotazu na server s implementací HotChocolate vypadá pro třetí scénář následovně:

```
query {
  actors(where:
      { roles: { some:
          { movie: { and: [
              { rank: { gt: 9 }},
              { moviesGenres: { some: { genre: { eq: "Horror" }}}},
              { directors: { some: { directorsGenres: { some: {
                 genre: { eq: "Comedy" }}}}}
              ]}}}) {
           items{
               id
                firstName
                lastName
                gender
           }
    }
}
```

Skládá se z následujících částí:

- **query**: typ operace,
- **actors**: zdroj dat,
- **roles: { some: ...}**: filtrování rolí,
- **movie: { and:** [**..]}**: filtrování filmů,
- **rank: { gt: 9 }**: filtrování filmů dle hodnocení,
- **moviesGenres: { some: { genre: { eq: "Horror"}}}**: filtrování filmů dle žánrů,
- **directors: { some: { directorsGenres: { some: { genre: { eq: "Comedy"}}}}}**: filtrování filmů dle žánrové preference režiséra.

```
1  SELECT TOP(2147483647) [a].[id] AS [Id], [a].[first_name] AS [
       FirstName], [a].[last_name] AS [LastName], [a].[gender] AS [
       Gender]
2  FROM [actors] AS [a]
3  WHERE EXISTS (
4      SELECT 1
5      FROM [roles] AS [r]
6      INNER JOIN [movies] AS [m] ON [r].[movie_id] = [m].[id]
7      WHERE ([a].[id] = [r].[actor_id]) AND ((([m].[rank] > 9) AND
           EXISTS (
8          SELECT 1
9          FROM [movies_genres] AS [m0]
10         WHERE ([m].[id] = [m0].[movie_id]) AND ([m0].[genre] ='Horror'
               ))) AND EXISTS (
11         SELECT 1
12         FROM [movies_directors] AS [m1]
13         INNER JOIN [directors] AS [d] ON [m1].[director_id] = [d].[id]
14         WHERE ([m].[id] = [m1].[movie_id]) AND EXISTS (
15             SELECT 1
16             FROM [directors_genres] AS [d0]
17             WHERE ([d].[id] = [d0].[director_id]) AND ([d0].[genre] = '
                   Comedy')))))
```

Zdrojový kód 28: SQL dotaz pro třetí scénář - HotChocolate

SQL výraz odpovídající HTTP požadavku, který byl službou HotChocolate přeložen pro tento scénář, je zobrazen v kódu 28.

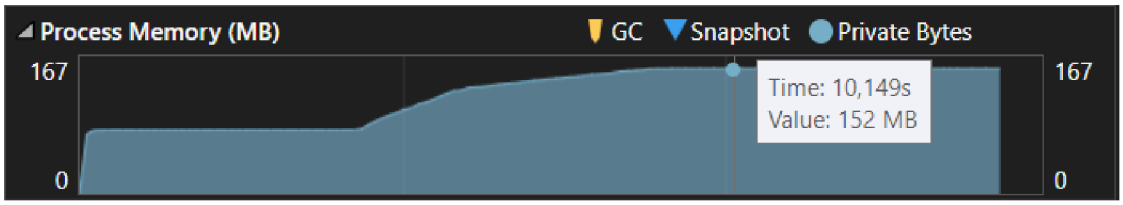
Obrázek 24 ukazuje výsledky třetího scénáře, získané jedním uživatelem testujícím HotChocolate pomocí *k6*. Obrázek 25 obsahuje zase výsledky testu získaných nástrojem *Visual Studio Profiler*.

```
execution: local
   script: 3/hotchocolate.js
   output: -

scenarios: (100.00%) 1 scenario, 1 max VUs, 10m30s max duration (incl. graceful stop):
        * default: 50 iterations shared among 1 VUs (maxDuration: 10m0s, gracefulStop: 30s)


   data_received..................: 379 kB 155 kB/s
   data_sent......................: 29 kB  12 kB/s
   http_req_blocked...............: avg=0s      min=0s      med=0s      max=0s       p(90)=0s       p(95)=0s
   http_req_connecting............: avg=0s      min=0s      med=0s      max=0s       p(90)=0s       p(95)=0s
   http_req_duration..............: avg=48.79ms min=46.32ms med=48.55ms max=56.87ms  p(90)=49.72ms  p(95)=50.68ms
     { expected_response:true }...: avg=48.79ms min=46.32ms med=48.55ms max=56.87ms  p(90)=49.72ms  p(95)=50.68ms
   http_req_failed................: 0.00%  ✓ 0         ✗ 50
   http_req_receiving.............: avg=62.97µs min=0s      med=0s      max=518.8µs  p(90)=505.59µs p(95)=515.81µs
   http_req_sending...............: avg=20.47µs min=0s      med=0s      max=518.29µs p(90)=0s       p(95)=0s
   http_req_tls_handshaking.......: avg=0s      min=0s      med=0s      max=0s       p(90)=0s       p(95)=0s
   http_req_waiting...............: avg=48.71ms min=46.32ms med=48.42ms max=56.87ms  p(90)=49.72ms  p(95)=50.4ms
   http_reqs......................: 50     20.423624/s
   iteration_duration.............: avg=48.94ms min=46.32ms med=48.64ms max=57.38ms  p(90)=50.14ms  p(95)=50.68ms
   iterations.....................: 50     20.423624/s
   vus............................: 1      min=1       max=1
   vus_max........................: 1      min=1       max=1
```

Obrázek 24: k6 - výsledky testu třetího scénáře při testování jedním uživatelem - HotChocolate



Obrázek 25: Visual Studio Profiler - využití paměti u třetího scénáře při testování jedním uživatelem - HotChocolate

Porovnání výsledků

Všechny tři SQL dotazy jsou strukturovány podobně a všechny se snaží dosáhnout stejného cíle. Zde jsou hlavní rozdíly.

1. **OData**: Tento dotaz je dobře strukturovaný a efektivně řeší požadovaný scénář. Zahrnuje několik subdotazů *EXISTS* pro otestování specifických podmínek. Tento dotaz však nezahrnuje žádné řazení výsledků.

2. **JSON:API**: Tento dotaz je velmi podobný OData dotazu, ale řadí výsledky podle id herce. To může být důležité pro pořadí, v jakém jsou výsledky vráceny, ale nemělo by to ovlivnit výkon dotazu.

3. **HotChocolate**: Tento dotaz je také téměř identický s předchozími dvěma dotazy. Hlavní rozdíl je v tom, že tento dotaz znovu obsahuje omezení vrácených řádků pomocí *TOP*. Kromě toho tento dotaz mění názvy sloupců výsledku na velká písmena (např. FirstName, LastName, Gender), což může ovlivnit čitelnost.

Celkově, všechny tři dotazy řeší zadaný problém efektivně a na první pohled není žádný z nich výrazně lepší nebo horší. Hlavní rozdíly mezi nimi spočívají v tom, jak jsou formátovány výsledky (názvy sloupců, řazení výsledků) a v použití TOP klauzule k omezení počtu výsledků. Výkon těchto dotazů by měl být podobný, protože všechny používají podobnou strukturu a techniky pro filtrování dat.

Tabulka 19 obsahuje rekapitulaci výsledků testů jednotlivých dotazovacích jazyků pro třetí scénář. Jednotlivé metriky jsou vysvětleny v rámci představení v kapitole 5.

Tabulka 19: Výsledky testů třetího scénáře

	OData	JSON:API	HotChocolate
data_received	383 kB	554 kB	379 kB
data_sent	14 kB	13 kB	29 kB
http_req_duration	58,13 ms	49,08 ms	48,79 ms
max_memory	199 MB	174 MB	152 MB

V oblasti přijatých dat byl HotChocolate nejefektivnější s přijetím 379 kB dat, následovaný velmi těsně OData s 383 kB dat. JSON:API byl méně efektivní s 554 kB dat. Všechny jazyky poskytují stejné informace, takže menší množství dat znamená lepší efektivitu.

V oblasti odeslaných dat byl nejefektivnější JSON:API s odesláním 13 kB dat, následovaný velmi těsně OData s 14 kB dat. HotChocolate odeslal 29 kB dat. Menší množství odesílaných dat znamená menší zatížení sítě a tedy i efektivnější komunikaci.

Pokud jde o dobu trvání HTTP požadavku, byly rozdíly mezi jednotlivými jazyky minimální, HotChocolate měl nejnižší dobu trvání požadavku s 48,79 ms, následovaný JSON:API s 49,08 ms a OData s 58,13 ms.

Pokud jde o spotřebu paměti, HotChocolate byl nejefektivnější se spotřebou 152 MB, následovaný JSON:API s 174 MB a OData s 199 MB.

Tyto výsledky ukazují, že ve třetím scénáři byl HotChocolate obecně nejefektivnější, ale rozdíly byly minimální a všechny dotazovací jazyky si vedly dobře. Tento scénář ukázal, že všechny tři jazyky jsou schopné efektivně zpracovávat složitější dotazy.

5.2 Testování více uživateli

V testech s více uživateli je testovací scénář spuštěn 500krát pro každý dotazovací jazyk. Iterace jsou sdíleny mezi deset virtuálních uživatelů. Tento test je zaměřen na simulaci realistického prostředí, kde je server současně dotazován z více zdrojů.

Jak již bylo zmíněno na začátku kapitoly, souběžným testováním více uživateli je podroben jen vybraný scénář. Pro tento účel byl vybrán hned první scénář, jelikož ho lze provést všemi dotazovacími jazyky a lze podle něj tedy provádět hodnocení. Současně není příliš složitý, jako například třetí scénář, u kterého je velmi malá pravděpodobnost použití v reálném prostředí.

V této části jsou výsledky jen prezentovány a zhodnoceny, jelikož samotný dotaz je stejný, jako při testování jedním uživatelem.

Vybraný scénář č. 1 má znění: *Dotaz na všechny filmy režiséra Steven Spielberg.*

OData

Výsledky paralelního testování prvního scénáře na dotazovacím jazyce OData za použití deseti uživatelů, získané nástrojem *k6*, jsou prezentovány na obrázku 26. Obrázek 27 ukazuje data shromážděná pomocí *Visual Studio Profileru.*

```
execution: local
   script: 1/odata.js
   output: -

scenarios: (100.00%) 1 scenario, 10 max VUs, 10m30s max duration (incl. graceful stop):
         * default: 500 iterations shared among 10 VUs (maxDuration: 10m0s, gracefulStop: 30s)


   data_received..................: 1.6 MB 2.4 MB/s
   data_sent......................: 97 kB  144 kB/s
   http_req_blocked...............: avg=1.05µs   min=0s     med=0s      max=503.7µs p(90)=0s       p(95)=0s
   http_req_connecting............: avg=0s       min=0s     med=0s      max=0s      p(90)=0s       p(95)=0s
   http_req_duration..............: avg=13.25ms  min=6.99ms med=12.67ms max=28.99ms p(90)=17.35ms  p(95)=19.59ms
     { expected_response:true }...: avg=13.25ms  min=6.99ms med=12.67ms max=28.99ms p(90)=17.35ms  p(95)=19.59ms
   http_req_failed................: 0.00%  ✓ 0         ✗ 500
   http_req_receiving.............: avg=96.04µs  min=0s     med=0s      max=1ms     p(90)=518.32µs p(95)=534.53µs
   http_req_sending...............: avg=134.65µs min=0s     med=0s      max=5.85ms  p(90)=0s       p(95)=23.42µs
   http_req_tls_handshaking.......: avg=0s       min=0s     med=0s      max=0s      p(90)=0s       p(95)=0s
   http_req_waiting...............: avg=13.02ms  min=6.86ms med=12.32ms max=28.99ms p(90)=17ms     p(95)=19.41ms
   http_reqs......................: 500    743.26078/s
   iteration_duration.............: avg=13.36ms  min=6.99ms med=12.95ms max=28.99ms p(90)=17.37ms  p(95)=19.59ms
   iterations.....................: 500    743.26078/s
```

Obrázek 26: k6 - výsledky testu druhého scénáře při testování více uživateli - OData

Obrázek 27: Visual Studio Profiler - využití paměti u třetího scénáře při testování více uživateli - OData

JSON:API

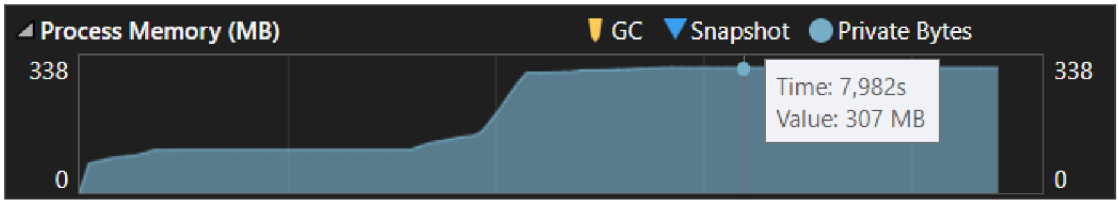
Obrázek 28 ilustruje získané výsledky při paralelním testování prvního scénáře na JSON:API s deseti uživateli, prováděného nástrojem *k6*. Další výsledky tohoto testu, zaznamenané pomocí *Visual Studio Profiler*, jsou zobrazeny na obrázku 29.

```
execution: local
   script: 1/jsonapi.js
   output: -

scenarios: (100.00%) 1 scenario, 10 max VUs, 10m30s max duration (incl. graceful stop):
         * default: 500 iterations shared among 10 VUs (maxDuration: 10m0s, gracefulStop: 30s)


   data_received..................: 3.1 MB 7.3 MB/s
   data_sent......................: 101 kB 241 kB/s
   http_req_blocked...............: avg=21.13µs min=0s  med=0s     max=1.28ms  p(90)=0s     p(95)=0s
   http_req_connecting............: avg=21.13µs min=0s  med=0s     max=1.28ms  p(90)=0s     p(95)=0s
   http_req_duration..............: avg=8.11ms  min=5ms med=8ms    max=18.56ms p(90)=9ms    p(95)=9.99ms
     { expected_response:true }...: avg=8.11ms  min=5ms med=8ms    max=18.56ms p(90)=9ms    p(95)=9.99ms
   http_req_failed................: 0.00%  0 0           500
   http_req_receiving.............: avg=80.59µs min=0s  med=0s     max=1.99ms  p(90)=7.26µs p(95)=997.6µs
   http_req_sending...............: avg=14.7µs  min=0s  med=0s     max=1ms     p(90)=0s     p(95)=0s
   http_req_tls_handshaking.......: avg=0s      min=0s  med=0s     max=0s      p(90)=0s     p(95)=0s
   http_req_waiting...............: avg=8.02ms  min=5ms med=7.99ms max=18.28ms p(90)=9ms    p(95)=9.99ms
   http_reqs......................: 500    1193.671251/s
   iteration_duration.............: avg=8.26ms  min=5ms med=8ms    max=20.08ms p(90)=9.84ms p(95)=9.99ms
   iterations.....................: 500    1193.671251/s
```

Obrázek 28: k6 - výsledky testu druhého scénáře při testování více uživateli - JSON:API



Obrázek 29: Visual Studio Profiler - využití paměti u třetího scénáře při testování více uživateli - JSON:API

HotChocolate

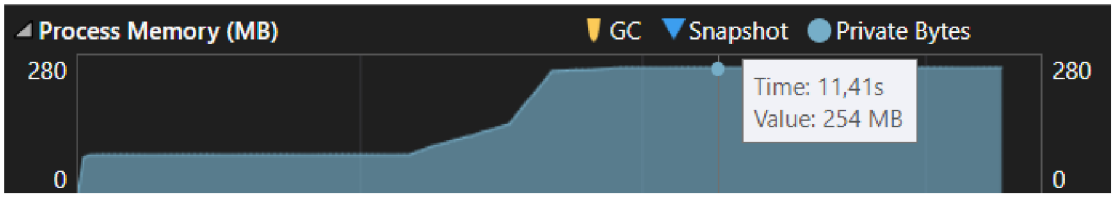
Na obrázku 30 jsou znázorněny výsledky paralelního testování prvního scénáře na HotChocolate s deseti uživateli, které byly shromážděny pomocí nástroje *k6*. Další data k tomuto testu, získaná nástrojem *Visual Studio Profiler*, lze nalézt na obrázku 31.

```
execution: local
   script: 1/hotchocolate.js
   output: -

scenarios: (100.00%) 1 scenario, 10 max VUs, 10m30s max duration (incl. graceful stop):
         * default: 500 iterations shared among 10 VUs (maxDuration: 10m0s, gracefulStop: 30s)


   data_received..................: 1.8 MB 4.6 MB/s
   data_sent......................: 240 kB 626 kB/s
   http_req_blocked...............: avg=11.23µs min=0s     med=0s     max=4ms     p(90)=0s       p(95)=0s
   http_req_connecting............: avg=0s      min=0s     med=0s     max=0s      p(90)=0s       p(95)=0s
   http_req_duration..............: avg=7.5ms   min=4.16ms med=6.86ms max=20.14ms p(90)=10.53ms  p(95)=12.22ms
     { expected_response:true }...: avg=7.5ms   min=4.16ms med=6.86ms max=20.14ms p(90)=10.53ms  p(95)=12.22ms
   http_req_failed................: 0.00%  ▒ 0          ▒ 500
   http_req_receiving.............: avg=66.95µs min=0s     med=0s     max=547.9µs p(90)=513.61µs p(95)=524.02µs
   http_req_sending...............: avg=91.73µs min=0s     med=0s     max=4ms     p(90)=0s       p(95)=12.52µs
   http_req_tls_handshaking.......: avg=0s      min=0s     med=0s     max=0s      p(90)=0s       p(95)=0s
   http_req_waiting...............: avg=7.35ms  min=4.16ms med=6.85ms max=19.61ms p(90)=10.04ms  p(95)=11.58ms
   http_reqs......................: 500    1304.737319/s
   iteration_duration.............: avg=7.61ms  min=4.16ms med=6.89ms max=20.14ms p(90)=10.64ms  p(95)=12.22ms
   iterations.....................: 500    1304.737319/s
```

Obrázek 30: k6 - výsledky testu druhého scénáře při testování více uživateli - HotChocolate



Obrázek 31: Visual Studio Profiler - využití paměti u třetího scénáře při testování více uživateli - HotChocolate

Porovnání výsledků

Tabulka 20 obsahuje shrnutí výsledků testů jednotlivých dotazovacích jazyků pro první scénář při souběžném dotazování 10 uživatelů. Jednotlivé metriky jsou vysvětleny v rámci představení v kapitole 5.

Tabulka 20: Výsledků paralelních testů prvního scénáře

	OData	JSON:API	HotChocolate
data_received	1,6 MB	3,1 MB	1,8 MB
data_sent	97 kB	101 kB	240 kB
http_req_duration	13,25 ms	8,11 ms	7,5 ms
max_memory	361 MB	307 MB	254 MB

V kontextu paralelních testů prvního scénáře se mohou ukázat jako nejdůležitější následující metriky:

- **http_req_duration** (čas potřebný k provedení HTTP požadavku): Tato metrika je důležitá pro určení, jak rychle se systém dokáže vyrovnat se zvýšenou zátěží.

- **max_memory** (maximální spotřeba paměti): To může ukázat, jaký systém lépe zvládá zátěž z pohledu spotřeby paměti, což může mít značný vliv na celkový výkon systému.

Pokud jde o *data_received* a *data_sent*, tyto hodnoty jsou stále relevantní, ale jsou v poměru totožné k výsledkům testů bez paralelního dotazovaní.

Čas potřebný k provedení HTTP požadavku (http_req_duration) byl nejnižší u HotChocolate, jehož požadavek byl zpracován za 7,5 ms. JSON:API byl těsně za ním s časem 8,11 ms, zatímco OData trvalo nejdéle, konkrétně 13,25 ms. V kontextu paralelního dotazování jsou tyto rozdíly ještě významnější, protože rychlost zpracování dotazu může výrazně ovlivnit celkovou dobu odezvy systému.

Pokud jde o spotřebu paměti, HotChocolate se opět ukázal jako nejefektivnější s hodnotou 254 MB. JSON:API spotřeboval 307 MB paměti a OData byla nejnáročnější na paměť s hodnotou 361 MB. V kontextu paralelního dotazování je spotřeba paměti zvláště důležitá, protože vyšší spotřeba může vést k nižšímu výkonu systému.

Tato srovnání ukazují, jak se jednotlivé dotazovací jazyky vyrovnávají s vyšší zátěží. HotChocolate a JSON:API se ukázaly jako efektivnější v kontextu paralelního dotazování, zatímco OData se ukázal jako méně efektivní v obou sledovaných metrikách.

Závěr

Všechny tři dotazovací jazyky – OData, JSON:API a HotChocolate jsou společně s ORM knihovnou Entity Framework schopné generovat efektivní a funkční SQL dotazy pro dané scénáře. Tyto dotazy jsou navrženy tak, aby vyhovovaly specifickým požadavkům scénářů a zdá se, že jsou schopny vrátit očekávané výsledky.

Pokud jde o strukturu dotazů, všechny tři jazyky generují podobné struktury s drobnými odlišnostmi, jako je změna názvů sloupců nebo seřazení výsledků. V zásadě však všechny dotazy využívají podobných principů a technik, jako jsou vnořené dotazy a EXISTS subdotazy.

Výkon těchto dotazů by měl být srovnatelný, pokud jsou všechny tabulky správně indexovány. Rozdíly ve výkonu mohou být způsobeny specifickou implementací databáze a jejími optimalizačními mechanismy.

Co se týče čitelnosti a srozumitelnosti, dotazy generované jednotlivými jazyky jsou všechny strukturované a relativně snadno pochopitelné pro ty, kteří jsou obeznámeni se SQL. I přesto, že nejsou tyto SQL dotazy určeny pro uživatele a toto kritérium je tedy zanedbatelné, je užitečné zahrnout do srovnávání jednotlivých dotazovacích jazyků i čitelnost.

Celkově vzato, OData, JSON:API a HotChocolate jsou efektivní nástroje pro generování SQL dotazů na základě specifických scénářů. Každý z nich má své stránky a slabé stránky, ale všechny jsou schopny splnit požadavky komplexních dotazů. Volba mezi nimi by tedy mohla záviset na specifických potřebách a preferencích vývojářů a daném kontextu použití.

V rámci srovnání byla provedena řada testů s cílem porovnat efektivitu dotazovacích jazyků. Každý z těchto jazyků byl testován ve třech různých scénářích, které byly navrženy tak, aby simulovat reálné situace, s nimiž se mohou vývojáři setkat při práci s těmito jazyky.

V prvním scénáři, kde byly jazyky testovány na jednoduchých dotazech, byly rozdíly mezi jednotlivými jazyky relativně malé, a žádný z nich se neukázal jako výrazně lepší nebo horší než ostatní ve všech ohledech. Ve druhém scénáři, který byl navržen pro zpracování velkých množství dat, byl HotChocolate výrazně efektivnější než OData, zatímco JSON:API nebyl schopen tento scénář zvládnout. Ve třetím scénáři, kde byly jazyky testovány v kontextu složitějších dotazů, byly rozdíly mezi jazyky opět poměrně malé.

Při testování paralelního dotazování 10 virtuálních uživatelů, HotChocolate a JSON:API se ukázaly jako efektivnější, zatímco OData byl méně efektivní.

Celkově lze říci, že ačkoli existují rozdíly v efektivitě jednotlivých dotazovacích jazyků, žádný z nich není výrazně lepší nebo horší než ostatní ve všech ohledech. Výběr nejvhodnějšího jazyka pro konkrétní projekt tedy bude záviset na specifických potřebách a omezeních daného projektu. Důležité je také vzít v úvahu další faktory, jako je kompatibilita s existující technologickou základnou, dostupnost a kvalita dokumentace a podpory a komfort vývojářů s daným jazykem.

Závěr

V rámci této diplomové práce byly detailně zkoumány nejrozšířenější dotazovací jazyky pro webové API, konkrétně OData, HotChocolate a JSON:API, a jejich implementace na platformě .NET. Kromě toho byly srovnány architektury REST a GraphQL a jejich možnosti implementace na stejné platformě.

Hlavními cíli práce bylo poskytnout hluboké porozumění pro tyto přístupy a jejich implementace. Srovnat jejich výkon a efektivitu v několika testovacích scénářích.

V teoretické části práce byly zkoumány každý z těchto přístupů a jejich implementace, přičemž důraz byl kladen na jejich klíčové vlastnosti, funkce a výhody a nevýhody. Tato část poskytla pevný teoretický základ pro praktické testy.

V praktické části byla provedena řada výkonnostních a efektivnostních testů na různých scénářích, které odrážejí typické i netypické situace, s nimiž se vývojáři mohou setkat při vývoji a nasazení webových aplikací.

Výsledky ukázaly, že každý z těchto přístupů a jejich implementací má své silné a slabé stránky a že neexistuje jednoznačný "nejlepší" přístup pro všechny situace. Volba přístupu a jeho implementace závisí na konkrétních potřebách a omezeních daného projektu.

Celkově tato práce přináší komplexní a detailní pohled na různé přístupy k API designu a jejich implementaci na platformě .NET. Je předpokládáno, že zjištěné informace budou užitečné pro vývojáře a architekty softwaru při rozhodování o výběru nejvhodnějšího přístupu a jeho implementace pro jejich projekty.

Conclusions

Throughout this thesis, the most widespread query languages for web APIs, specifically OData, HotChocolate, and JSON:API, and their implementations on the .NET platform were examined in detail. Additionally, the architectures of REST and GraphQL were compared, along with their implementation possibilities on the same platform.

The main objectives of the work were to provide deep understanding for these approaches and their implementations and to compare their performance and efficiency in various testing scenarios.

In the theoretical part of the work, each of these approaches and their implementations were examined, with emphasis placed on their key properties, functions, and pros and cons. This part provided a solid theoretical basis for the practical tests.

In the practical part, a series of performance and efficiency tests were carried out on different scenarios that reflect typical and also atypical situations developers may encounter when developing and deploying web applications.

The results showed that each of these approaches and their implementations have their strengths and weaknesses and that there is no unequivocal "best" approach for all situations. The choice of approach and its implementation depends on the specific needs and constraints of the given project.

Overall, this thesis offers a comprehensive and detailed look at various approaches to API design and their implementation on the .NET platform. It is anticipated that the information gleaned will be useful to developers and software architects in deciding on the selection of the most suitable approach and its implementation for their projects.

A Obsah elektronických dat

Na samotném konci textu práce je uveden stručný popis obsahu elektronických dat odevzdaných v systému katedry informatiky spolu s textem. Tato data jsou nedílnou součástí práce a tvoří (datovou) přílohu textu práce. Povinné položky struktury dat jsou:

`text/`
: Adresář s textem práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech (textových) příloh, a všechny soubory potřebné pro bezproblémové vytvoření PDF dokumentu textu (případně v ZIP archivu), tj. zdrojový text textu a příloh, vložené obrázky, apod.

`README.*`
: Textový soubor (s příponou např. `.txt`) s informacemi o opakovatelném způsobu použití ostatních dat práce – typicky plně reprodukovatelný co nejúplnější funkční postup zprovoznění software vytvořeného v rámci práce, tzn. jeho případné instalace/nasazení a spuštění, včetně uvedení všech požadavků pro bezproblémový provoz; za zprovoznění software se nepovažuje zpřístupnění (např. po Internetu) již někde zprovozněného software.

`*`
: Adresáře a soubory s veškerými ostatními autorskými daty práce (případně v ZIP archivu) – typicky spustitelné a další soubory software vytvořeného v rámci práce potřebné pro bezproblémový provoz software, případně jeho instalační program, a kompletní zdrojové texty software a další data nutná pro plně reprodukovatelné korektní vytvoření spustitelných souborů.

Dále mohou data obsahovat například:

- ukázková a testovací data použitá v práci nebo pro potřeby posouzení práce v rámci její obhajoby,
- položky bibliografie v elektronické podobě, příp. jiná relevantní literatura a dokumentace vztahující se k práci,
- cizí data (software) potřebná pro bezproblémové použití autorských dat práce (software), která nejsou standardní součástí předpokládaného (softwarového) vybavení uživatele.

U veškerých cizích obsažených materiálů jejich zahrnutí dovolují podmínky pro jejich veřejné šíření nebo přiložený souhlas držitele práv k užití. Pro všechny použité (a citované) materiály, u kterých toto není splněno a nejsou tak obsaženy, je uveden jejich zdroj, např. webová adresa, v bibliografii nebo textu práce nebo souboru `README.*`.

Literatura

[1] Hunter, Kirsten L. *Irresistible APIs: Designing web APIs that developers will love*. 2016. ISBN 9781617292552.

[2] David Gourley, Brian Totty. *HTTP: The Definitive Guide*. 2002. ISBN 9780596519926.

[3] Dorman, Michael. *Introduction to Web Mapping*. 2020. ISBN 9780367861186.

[4] Brenda Jin Saurabh Sahni, Amir Shevat. *Designing Web APIs: Designing web APIs that developers will love*. 2018. ISBN 9781492026921.

[5] *ASP.NET documentation*. Dostupný z: ⟨https://learn.microsoft.com/aspnet/core⟩.

[6] Grafana Labs. *k6 Open Source*. Dostupný také z: ⟨https://k6.io/open-source⟩.

[7] Microsoft. *Visual Studio Profiler*. Dostupný také z: ⟨https://learn.microsoft.com/visualstudio/profiling⟩.

[8] Kranjc, Janez. *IMDb*. Dostupný také z: ⟨https://relational.fit.cvut.cz/dataset/IMDb⟩.

[9] *Web technology for developers*. Dostupný z: ⟨https://developer.mozilla.org/docs/Web⟩.

[10] *OData documentation*. Dostupný z: ⟨https://learn.microsoft.com/odata⟩.

[11] *JsonApiDotNetCore documentation*. Dostupný z: ⟨https://www.jsonapi.net⟩.

[12] *HotChocolate documentation*. Dostupný z: ⟨https://chillicream.com/docs/hotchocolate⟩.