Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvantitativních metod

Technologie pro tvorbu webových API

Diplomová práce

Autor: Filip Dvořák
Studijní program: Informační management

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D.

Hradec Králové Duben 2024

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 20.4.2024 Filip Dvořák

Poděkování:

Děkuji vedoucímu diplomové práce doc. Mgr. Tomášovi Kozlovi, Ph.D. za metodické vedení práce, poskytnutá doporučení a odborné rady k mé práci.

Anotace

Tato diplomová práce se zabývá analýzou a porovnáním nejpoužívanějších technologií pro vývoj webových API s důrazem na RESTful API. V teoretické části je představen význam webových API, jsou rozebrány základní typy webových API a podrobně jsou popsány základní principy fungování RESTful API. V praktické části je pak provedena analýza a porovnání vybraných technologií pro vývoj RESTful API (ASP.NET Core, Spring, Django) ve spojení s populárními databázovými systémy (SQL Server, Oracle, PostgreSQL) z hlediska rychlosti, výkonnosti a stability. Praktická část obsahuje nasazení a konfiguraci testovaných API a databází na stejném serveru a následné testování pomocí nástroje Apache JMeter. Na základě výsledků testování jsou identifikovány nejefektivnější kombinace technologií a databází pro vývoj RESTful API z hlediska výkonnosti a stability.

Klíčová slova: Webová API, REST, ASP.NET Core, Spring, Django

Annotation

Title: Technologies for web API development

This thesis deals with the analysis and comparison of the most used technologies for web API development with emphasis on RESTful APIs. In the theoretical part, the importance of web APIs is introduced, the basic types of web APIs are discussed and the basic principles of RESTful APIs are described in detail. In the practical part, an analysis and comparison of selected technologies for RESTful API development (ASP.NET Core, Spring, Django) in conjunction with popular database systems (SQL Server, Oracle, PostgreSQL) in terms of speed, performance and stability is performed. The practical part includes deployment and configuration of the tested APIs and databases on the same server and subsequent testing using Apache JMeter. Based on the testing results, the most effective combinations of technologies and databases for developing RESTful APIs in terms of performance and stability are identified.

Keywords: Web API, REST, ASP.NET Core, Spring, Django

Obsah

1 Úvod ... 1

2 Cíl práce ... 2

2.1 Hypotézy ... 2

2.1.1 ASP.NET Core ... 2

2.1.2 Spring ... 3

2.1.3 Django ... 4

3 Metodika zpracování ... 5

3.1 Výzkumné otázky ... 6

4 Webová API ... 7

4.1 HTTP komunikace ... 7

4.1.1 HTTP požadavek ... 8

4.1.2 HTTP odpověď ... 11

4.2 Formáty pro reprezentaci dat ... 12

4.2.1 JSON ... 13

4.2.2 XML ... 14

4.3 Využití webových API ... 15

5 Architektury webových API ... 16

5.1 REST ... 17

5.1.1 Principy ... 19

5.1.2 Richardsonův model zralosti ... 20

5.2 Webhooks ... 22

5.3 GraphQL ... 23

5.4 SOAP ... 24

5.4.1 WSDL ... 25

6 Databázové systémy pro webová API 26
6.1 Typy databází 26
6.2 Relační databázové systémy 27
6.2.1 PostgreSQL 28
6.2.2 Oracle Database 29
6.2.3 Microsoft SQL Server 29
7 Technologie pro vývoj RESTful API 31
7.1 Model-View-Controller 31
7.2 C# - ASP.NET Core 32
7.3 Java – Spring 34
7.4 Python – Django 35
8 Testování rychlostí webových API 37
8.1 Testovací prostředí 37
8.2 ASP.NET Core 41
8.2.1 Představení kódu 41
8.2.2 Měření rychlostí odezvy 45
8.3 Spring 50
8.3.1 Představení kódu 50
8.3.2 Měření rychlostí odezvy 54
8.4 Django 59
8.4.1 Představení kódu 59
8.4.2 Měření rychlostí odezvy 62
8.5 Porovnání technologií 66
9 Shrnutí 71
10 Závěr 76
11 Seznam použité literatury 78

Seznam obrázků

Obrázek 1 - HTTP komunikace (upraveno z [4]) 8

Obrázek 2 – Popularita architektur webových API [19] 16

Obrázek 3 - Richardsonův model zralosti [23] 21

Obrázek 4 - Popularita relačních databázových systémů [49] 28

Obrázek 5 - Popularita technologií pro RESTful API [50] 31

Obrázek 6 - Testovací databázové schéma [Zdroj: autor] 39

Obrázek 7 - Porovnání rychlostí odezvy pro ASP.NET Core [Zdroj: autor] 48

Obrázek 8 - Porovnání rychlostí odezvy pro Spring [Zdroj: autor] 57

Obrázek 9 - Porovnání rychlostí odezvy pro Django [Zdroj: autor] 64

Obrázek 10 - Porovnání technologií s databází SQL Server [Zdroj: autor] 67

Obrázek 11 - Porovnání technologií s databází PostgreSQL [Zdroj: autor] 68

Obrázek 12 - Porovnání technologií s databází Oracle [Zdroj: autor] 70

Seznam ukázek

Ukázka 1- HTTP požadavek [Zdroj: Chat GPT] 10

Ukázka 2 - HTTP odpověď [Zdroj: Chat GPT] 12

Ukázka 3 – Data ve formátu JSON [Zdroj: Chat GPT] 13

Ukázka 4 – Data ve formátu XML [Zdroj: Chat GPT] 14

Ukázka 5 - Formát URI [21] 17

Ukázka 6 - ASP.NET Core [Zdroj: Chat GPT] 33

Ukázka 7 – Spring [Zdroj: Chat GPT] 35

Ukázka 8 – Django [Zdroj: Chat GPT] 36

Ukázka 9 - Ukázka HTTP dotazu [Zdroj: autor] 40

Ukázka 10 – Objekt Book pro mapování v ASP.NET Core [Zdroj: autor] 41

Ukázka 11 - DataContext v ASP.NET Core [Zdroj: autor] 42

Ukázka 12 - GET metoda v ASP.NET Core [Zdroj: autor] 43

Ukázka 13 - POST metoda v ASP.NET Core [Zdroj: autor] 44

Ukázka 14 - PUT metoda v ASP.NET Core [Zdroj: autor] 44

Ukázka 15 - DELETE metoda v ASP.NET Core [Zdroj: autor] 45

Ukázka 16 – Objekt Book pro mapování ve Spring [Zdroj: autor] 51

Ukázka 17 - GET metoda ve Spring [Zdroj: autor] 52

Ukázka 18 - POST metoda ve Spring [Zdroj: autor] 53

Ukázka 19 - PUT metoda ve Spring [Zdroj: autor] 53

Ukázka 20 - DELETE metoda ve Spring [Zdroj: autor] 54

Ukázka 21 – Objekt Book pro mapování v Django [Zdroj: autor] 59

Ukázka 22 - GET metoda v Django [Zdroj: autor] 60

Ukázka 23 - POST metoda v Django [Zdroj: autor] 60

Ukázka 24 - PUT metoda v Django [Zdroj: autor] 61

Ukázka 25 - DELETE metoda v Django [Zdroj: autor] 61

Seznam tabulek

Tabulka 1 - Výsledky testování ASP.NET Core s databází SQL Server [Zdroj: autor] .. 46

Tabulka 2 - Výsledky testování ASP.NET Core s databází PostgreSQL [Zdroj: autor] . 46

Tabulka 3 - Výsledky testování ASP.NET Core s databází Oracle [Zdroj: autor] 46

Tabulka 4 - Výsledky testování Spring s databází SQL Server [Zdroj: autor] 55

Tabulka 5 - Výsledky testování Spring s databází PostgreSQL [Zdroj: autor] 55

Tabulka 6 - Výsledky testování Spring s databází Oracle [Zdroj: autor] 55

Tabulka 7 - Výsledky testování Django s databází SQL Server [Zdroj: autor] 62

Tabulka 8 - Výsledky testování Django s databází PostgreSQL [Zdroj: autor].....62

Tabulka 9 - Výsledky testování Django s databází Oracle [Zdroj: autor].....63

1 Úvod

V posledních letech, kdy digitální transformace proniká do všech oblastí našeho života, se stává vývoj webových API (Application Programming Interface) klíčovou částí pro propojení, interoperabilitu a kooperaci různých systémů a aplikací. API umožňují efektivní výměnu dat a informací mezi různými částmi softwarových aplikací, což vytváří základní předpoklad pro moderní informační systémy.

Tato diplomová práce obsahuje základní přehled a analýzu nejpoužívanějších technologií pro vývoj webových API, s důrazem na RESTful API (Representational State Transfer), které se stalo v podstatě standardem pro návrh a implementaci webových rozhraní. V diplomové práci je popsáno, co vlastně webová API jsou a jak fungují, podrobněji rozebráno a popsáno je fungování RESTful API, včetně zkoumání konkrétních technologií pro jejich vývoj jako například ASP.NET Core nebo Spring. Budou zmíněny základní principy těchto technologií včetně jejich výhod a nevýhod. Kromě RESTful diplomová práce stručně zmíní a popíše další alternativy pro vývoj webových API jako například GraphQL nebo SOAP.

Teoretická část bude také věnována databázovým systémům. Bude stručně nastíněna jejich role v kontextu webových API a budou popsány vlastnosti a charakteristiky nejpoužívanějších databázových systémů současnosti jako například Microsoft SQL Server, nebo Oracle Database.

V praktické části následně bude provedena analýza a porovnání vybraných technologií pro vývoj RESTful API v oblasti rychlosti, výkonnosti a stability spolupráce s vybranými databázovými systémy na základě získaných statistických charakteristik.

Tato diplomová práce může sloužit pro vývojáře a architekty softwarových řešení v oblasti informačních technologií, kteří se zajímají o vývoj webových API a hledají hlubší porozumění technologiím, které jsou k dispozici pro tuto formu integrace a komunikace mezi aplikacemi. Práce může čtenářům poskytnout stručný pohled na problematiku vývoje webových API z hlediska rychlosti a stability a pomoci jim při výběru technologií pro implementaci rozhraní, která budou plně odpovídat potřebám jejich projektů.

2 Cíl práce

Cílem této práce je provést analýzu nejčastějších dostupných technologií pro vývoj webových API a následně provést analýzu a porovnání vybraných technologií pro vývoj RESTful API v oblasti rychlosti, výkonnosti a stability při práci s vybranými databázovými systémy na základě získaných statistik. Na základě výsledků testů bude určeno, které technologie pro vývoj RESTful API nejlépe spolupracují s jednotlivými databázovými systémy a bude posouzena vhodnost jejich použití pro různé typy projektů s především na základě rychlosti a stability.

2.1 Hypotézy

Pro zodpovězení 3. výzkumné otázky jsou stanoveny následující hypotézy, které odhalí, zda mezi jednotlivými skupinami databází existují statisticky významné rozdíly. Následně na základě výsledků těchto hypotéz v kombinaci se získanými statistickými veličinami bude zodpovězena 3. výzkumná otázka. Pro každou testovanou technologii je nejprve formulována hypotéza, která ověří, zda existuje statisticky významný rozdíl mezi alespoň jednou dvojící databázových systémů pomocí odpovídajícího testu (hypotéza 1, 4 a 7) a pokud je tato hypotéza přijata, jsou pomocí dalšího testu zkoumány statisticky významné rozdíly mezi konkrétními zkoumanými dvojicemi. (hypotéza 2, 3, 5, 6, 8 a 9). Hypotézy 1,4 a 7 budou testovány pomocí Kruskoval-Wallisova testu, který se používá k ověření existence statisticky významných rozdílů mezi třemi nebo více skupinami nezávislých pozorování. Pro zbylé stanovené hypotézy jsou použity Dunnovy post-hoc testy, které již odhalují statisticky významné rozdíly mezi konkrétními dvojicemi. Pro výpočet testů je použit statistický software Past4.03.

2.1.1 ASP.NET Core

1. Hypotéza

H_0: Neexistuje žádný statisticky významný rozdíl v časech odezvy technologie ASP.NET Core mezi všemi databázovými systémy při použití metody GET.

H_1: Existuje statisticky významný rozdíl v časech odezvy technologie ASP.NET Core mezi alespoň jednou dvojicí databázových systémů při použití metody GET.

2. Hypotéza

H_0: Neexistuje žádný statisticky významný rozdíl v časech odezvy technologie ASP.NET Core s databází SQL Server a Oracle při použití HTTP metody GET.

H_1: Existuje statisticky významný rozdíl v časech odezvy technologie ASP.NET Core s databází SQL Server a Oracle při použití HTTP metody GET.

3. Hypotéza

H_0: Neexistuje žádný statisticky významný rozdíl v časech odezvy technologie ASP.NET Core s databází SQL Server a PostgreSQL při použití HTTP metody GET.

H_1: Existuje statisticky významný rozdíl v časech odezvy technologie ASP.NET Core s databází SQL Server a PostgreSQL při použití HTTP metody GET.

2.1.2 Spring

4. Hypotéza

H_0: Neexistuje žádný statisticky významný rozdíl v časech odezvy technologie Spring mezi všemi databázovými při použití metody GET.

H_1: Existuje statisticky významný rozdíl v časech odezvy technologie Spring mezi alespoň jednou dvojicí databázových systémů při použití metody GET.

5. Hypotéza

H_0: Neexistuje žádný statisticky významný rozdíl v časech odezvy technologie Spring s databází Oracle a SQL Server při použití HTTP metody GET.

H_1: Existuje statisticky významný rozdíl v časech odezvy technologie Spring s databází Oracle a SQL Server při použití HTTP metody GET.

6. Hypotéza

H_0: Neexistuje žádný statisticky významný rozdíl v časech odezvy technologie Spring s databází Oracle a PostgreSQL při použití HTTP metody GET.

H_1: Existuje statisticky významný rozdíl v časech odezvy technologie Spring s databází Oracle a PostgreSQL při použití HTTP metody GET.

2.1.3 Django

7. Hypotéza

H_0: Neexistuje žádný statisticky významný rozdíl v časech odezvy technologie Django mezi všemi databázovými při použití metody GET.

H_1: Existuje statisticky významný rozdíl v časech odezvy technologie Django mezi alespoň jednou dvojicí databázových systémů při použití metody GET.

8. Hypotéza

H_0: Neexistuje žádný statisticky významný rozdíl v časech odezvy technologie Django s databází PostgreSQL a SQL Server při použití HTTP metody GET.

H_1: Existuje statisticky významný rozdíl v časech odezvy technologie Django s databází PostgreSQL a SQL Server při použití HTTP metody GET.

9. Hypotéza

H_0: Neexistuje žádný statisticky významný rozdíl v časech odezvy technologie Django s databází PostgreSQL a Oracle při použití HTTP metody GET.

H_1: Existuje statisticky významný rozdíl v časech odezvy technologie Django s databází PostgreSQL a Oracle při použití HTTP metody GET.

3 Metodika zpracování

Pro dosažení vytyčeného cíle, kterým je provést analýzu a porovnání vybraných technologií pro vývoj RESTful API v oblasti rychlosti a výkonnosti při práci s vybranými databázovými systémy, je nejprve nutné vybrat vhodné technologie a databázové systémy pro testování. Dané technologie jsou vybrány na základě jejich popularity, popularity programovacího jazyka, ve kterém jsou implementovány a rozšířenosti ve vývojářské komunitě.

Databázové systémy určené pro testování jsou rovněž vybrány na základě jejich popularity a podílu na trhu. Zároveň však jejich výběr závisí i na jednotlivých technologiích pro tvorbu webových API, kdy je k jednotlivým technologiím vybrána databáze, se kterou se technologie často používá v reálných projektech. Všechny technologie jsou porovnány vzájemně se všemi vybranými databázemi, aby z výsledků testování bylo patrné, zda je technologie vhodně optimalizována pro databázi, se kterou se často používá v reálných projektech, nebo zda je schopna účinně komunikovat i v kombinaci s databázemi, se kterými se technologie často nepoužívá.

Aby byl minimalizován vliv hardwarových a softwarových parametrů na testování, jsou všechna jednotlivá API i databáze nasazeny na stejném serveru. Dále je provedena taková konfigurace databází i jednotlivých API, aby jednotlivé technologie i databáze byly nastaveny co nejpodobnějším způsobem a vliv na výsledky tak byl minimální. Veškeré databázové systémy pro testování používají stejná data, aby nedošlo ke zkreslení výsledků na základě obsahu dat v databázích.

Po nasazení a konfiguraci je provedeno testování výkonu jednotlivých kombinací pomocí nástroje Apache JMeter. Testování je provedeno pro technologii ASP.NET Core v jazyce C#, Spring v jazyce Java a technologii Django v jazyce Python. Testy jsou provedeny pro základní HTTP metody (GET, POST, PUT, DELETE) přičemž největší důraz je kladen na metodu GET, která je implementována tak, že pracuje s největším objemem dat z databáze a výsledky testování této metody jsou tak nejvíce vypovídající. Aby byly eliminovány náhodné jevy a variabilita výsledků, je pro všechny jednotlivé technologie a metody provedeno 100 opakování. Na základě výsledků všech volání jsou následně určeny průměrné hodnoty, ale i další statistické charakteristiky na základě kterých je

posouzeno, které technologie nejlépe spolupracují s jednotlivými databázemi a také jsou porovnány jednotlivé technologie vzájemně mezi sebou.

Po provedení testů je provedena analýza výsledků, která provede srovnání rychlosti a výkonnosti implementací jednotlivých RESTful API pomocí různých technologií s vybranými databázovými systémy. Na základě výsledků testování a analýzy budou zodpovězeny výzkumné otázky, tedy budou porovnány jednotlivé implementaci API s jednotlivými databázemi, budou identifikovány nejefektivnější kombinace technologií a databází z hlediska rychlosti a výkonnosti a bude posouzeno, zda jednotlivé technologie optimálně pracují s nejčastěji používanými databázovými systémy.

Pro generování jednotlivých ukázek kódu v teoretické části byl využit jazykový model založený na umělé inteligenci ChatGPT.

3.1 Výzkumné otázky

1. Jak se liší rychlost, výkonnost a stability implementací RESTful API pomocí různých technologií při práci se stejnými databázovými systémy?
2. Jaké jsou nejefektivnější kombinace technologií pro vývoj RESTful API ve spojení s konkrétními databázovými systémy z hlediska dosažení optimální rychlosti, výkonnosti a stability?
3. Budou jednotlivé technologie pro vývoj webových API při použití metody GET nejrychleji pracovat s databázemi, se kterými se v praxi nejčastěji používají, oproti ostatním vybraným technologiím? Zkoumané kombinace jsou následující:
    - ASP.NET Core – SQL Server
    - Spring – Oracle
    - Django – PostgreSQL

4 Webová API

Jak již bylo zmíněno výše, API (Application Programming Interface) představují neodmyslitelnou součást většiny moderních softwarových aplikací. API představuje rozhraní, které slouží k propojení a komunikaci různých softwarových aplikací, čímž umožňuje sdílení dat a velmi tak usnadňuje vývoj webových služeb [1].

Webová API jsou pak takový druh API, která zajišťují přenos strojově čitelných dat mezi webovými systémy, tedy pomocí internetu a jeho protokolů HTTP (Hypertext Transfer Protocol) [2] nebo HTTPS (Hypertext Transfer Protocol Secure).

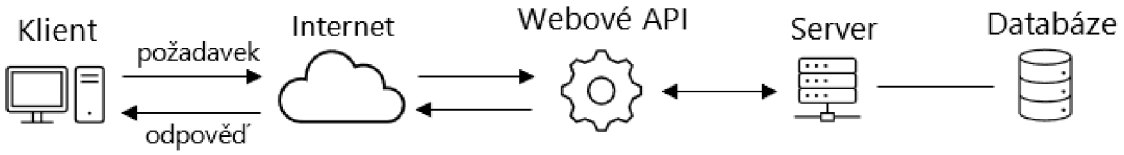
4.1 HTTP komunikace

HTTP je standardní způsob komunikace mezi webovým prohlížečem a webovým serverem. Tento protokol určuje pravidla pro přenos dat mezi počítači. V rámci internetu se pomocí něj sdílí různé typy dat, včetně textu, obrázků a multimediálních souborů. V podstatě všechny aktivity, které uživatelé provádějí ve svém webovém prohlížeči, využívají protokol HTTP. Protokol HTTPS je pak kombinací protokolu HTTP a protokolu TLS (Transport Layer Security) nebo staršího SSL (Secure Sockets Layer). Tyto protokoly zajišťují šifrovanou komunikaci a bezpečné ověření identity cílového webového serveru [3]. HTTPS tedy oproti běžnému HTTP poskytuje vyšší úroveň bezpečnosti díky certifikaci těchto dvou protokolů. Právě z důvodu bezpečnosti se proto HTTPS v dnešní době stalo de facto standardem a webové stránky, jež fungují na protokolu HTTP nejsou považované za bezpečné.

Princip fungování webových API se tedy běžně vyjadřuje prostřednictvím architektury klient-server. Klientem se rozumí jakákoliv front-end aplikace nebo rozhraní, se kterou interaguje koncový uživatel. Na straně serveru se nachází back-endová logika a databázové operace. API v tomto scénáři představuje mezivrstvu mezi klientem a serverem, která umožňuje posílat požadavky (requests) ze strany klienta a odpovědi (responses) ze strany serveru [2]. Konkrétní příklady požadavků a odpovědí budou představeny níže.

Role webových API je znázorněna pomocí následujícího diagramu. Klient iniciuje komunikaci vysláním požadavku prostřednictvím protokolu HTTP. API, jakožto

poskytovatel rozhraní, přijímá požadavky od klienta a směruje ho k webovému serveru. Server pak na základě HTTP dotazu pracuje s databází, například získá potřebná data, nebo s daty manipuluje jiným způsobem. Následná odpověď, například získaná data, je pak serverem přes API zaslány zpět na klientské zařízení. Formáty pro reprezentaci těchto dat budou rozebrány v kapitole níže.



Obrázek 1 - HTTP komunikace (upraveno z [4])

4.1.1 HTTP požadavek

HTTP požadavek je základním stavebním prvkem komunikace mezi klientem a serverem na internetu. Představuje informace nebo data, které internetové prohlížeče potřebují k načtení dat webové stránky.[5]

V kontextu webových API je HTTP požadavek způsob, jakým klient požaduje určitou akci od serveru. Ve všech HTTP požadavcích jsou přítomny následující informace.

HTTP Metoda

Protokol HTTP popisuje sadu metod, které specifikují požadovanou akci pro daný zdroj. Tyto metody umožňují klientům komunikovat se serverem a provádět různé operace s daty ze serveru, jako je například získání dat, jejich aktualizace, vytvoření nebo vymazání. HTTP popisuje následující metody[6].

1. *GET* – Tato metoda slouží pouze k získání dat ze serveru. Požadavek s metodou GET nemění stav serveru a slouží pouze k získání informací.
2. *HEAD* – Tato metoda je téměř identická s metodou GET, avšak server odpovídá pouze s hlavičkami zdroje bez těla požadavku.
3. *POST* – Jedná se o metodu, která se používá pro zaslání entity specifickému zdroji. Může ovlivnit stav serveru a zpravidla se používá pro přidávání dat do databáze.

4. *PUT* – Metoda PUT se používá k úpravě dat na serveru na základě zadaných parametrů.
5. *DELETE* – Metoda DELETE, jak již název napovídá, odstraňuje data na základě zadaných parametru.
6. *CONNECT* – Tato metoda se používá k navázání síťového spojení se serverem,
7. *OPTIONS* – Metoda OPTIONS umožňuje klientovi získat informace o možnostech nebo požadavcích spojených se zdrojem.
8. *TRACE* – Provádí message loop-back test a slouží k diagnostice a testování komunikace mezi klientem a serverem.
9. *PATCH* – Provádí částečné úpravy zdroje. Umožňuje aktualizaci zdroje, aniž by bylo nutné posílat celý zdroj.

V tomto seznamu byly uvedeny veškeré metody, které mohou být použity pro zasílání HTTP požadavků, nicméně zpravidla jsou používány pouze metody GET, POST, PUT, PATCH a DELETE, jenž jsou ekvivalenty pro operace CRUD (create, read, update, delete).[7] Pro účely této diplomové práce proto bude pracováno výhradně s těmito metodami.

Cílová adresa

Cílová adresa neboli URL (Uniform Resource Locator), je specifická adresa, která identifikuje jedinečný zdroj nebo umístění na internetu.[8] V kontextu webových API se URL využívá k určení konkrétního cíle, ke kterému se klient snaží přistoupit prostřednictvím svého požadavku. URL adresa se skládá z několika částí, a to z použitého protokolu, doménového jména a cesty k danému zdroji na serveru. URL poskytuje podstatný rámec pro správnou komunikaci mezi klientem a serverem a umožňuje přesnou lokalizaci zdroje dat, které mají být získány nebo modifikovány.

Verze protokolu HTTP

Číslo verze protokolu HTTP, kterému je zpráva přizpůsobena.

Hlavička požadavku (header)

Hlavička v HTTP požadavku slouží k poskytnutí informací o kontextu požadavku. Informace mohou serveru pomoci lépe porozumět požadavku a vhodně přizpůsobit

odpověď[9]. Požadavek zpravidla obsahuje řadu hlaviček. Například hlavičky typu Accept-* umožňují klientovi nastavit preferovaný formát odpovědi. Hlavička Authorization souží pro ověření autorizace nebo User-Agent pro získání informací o prohlížeči[9].

Tělo požadavku (body)

Tělo HTTP požadavku obsahuje informace, které mají být přeneseny z klienta na server[5]. Těla zpráv jsou vhodná pouze pro některé metody, konkrétně pro metody, které odesílají data na server, tedy například metoda POST. Požadavek s metodou GET pouze žádá o odeslání prostředku a tělo zprávy tak nemá. [10]

Těla požadavku mohou být ve formě různých formátů, jako je například JSON nebo XML [11], jenž hrají v kontextu webových API důležitou roli a budou podrobněji rozebrány v dalších částech diplomové práce. Těla požadavků však mohou mít i podobu textových řetězců, binární dat nebo jiných formátů podle konkrétních požadavků webového API.

```
POST /api/endpoint HTTP/3
Content-Type: application/json
Content-Length: 59
{
  "name": "John Doe",
  "age": 30,
  "email": "johndoe@example.com"
}
```

Ukázka 1- HTTP požadavek [Zdroj: Chat GPT]

V ukázce výše je názorně prezentována konkrétní struktura HTTP požadavku. Na prvním řádku se nachází použitá metoda, zde konkrétně metoda POST. Dále se na témže řádku nachází cílová adresa směřující k požadovanému zdroji. Zakončení prvního řádku specifikuje verzi HTTP protokolu, v tomto konkrétním příkladu aktuální verze HTTP/3. Následující dva řádky obsahují hlavičky dotazu, které přesně specifikují formát přenášených dat a zároveň definují délku obsahu v těle

požadavku. Samotné tělo požadavku následuje pod hlavičkami a je zde prezentováno ve formátu JSON.

4.1.2 HTTP odpověď

HTTP odpověď je odesílána serverem klientovi. Jejím cílem je poskytnout klientovi požadovaný zdroj, informovat ho, zda požadovaná akce byla provedena, nebo zda došlo k chybě při zpracování jeho požadavku. [12]

HTTP odpověď má podobnou strukturu jako požadavek a obsahuje následující části:

Stavový kód

Stavový kód je trojmístný kód, který sděluje, zda byl HTTP požadavek úspěšně dokončen. Tyto kódy jsou rozděleny do pěti kategorií [13].

1. *Informační odpovědi (Kód 1xx)* – Kódy začínající číslem 1 informují, že žádost byla přijata a je v procesu. Tato odpověď je dočasná, a to do doby kdy probíhá zpracování žádosti.
2. *Úspěšné odpovědi (Kód 2xx)* – Tato kategorie označuje, že požadavek, který chtěl klient vykonat, je úspěšně dokončen.
3. *Přesměrování (Kód 3xx)* – Tato kategorie označuje, že klient může provést další kroky k dokončení požadavku. Obvykle je dodatečnou akcí přesměrování klienta na jinou adresu.
4. *Chyby na straně klienta (Kód 4xx)* – Odpověď s kódem začínajícím číslem 4 oznamují, že požadavek nelze splnit, jelikož došlo k chybě na straně klienta. Zpravidla se jedná o špatnou syntaxi nebo chybějící autorizaci.
5. *Chyby na straně serveru (Kód 5xx)* – Poslední kategorie začínající číslem 5 značí buďto chybu na straně serveru nebo to, že server není schopen požadavek provést.

Hlavička odpovědi

Hlavička odpovědi má stejný význam jako v případě HTTP požadavku a obsahuje dodatečné informace o odpovědi a o serveru, který ji odeslal. [12]

Tělo odpovědi

Tělo odpovědi, stejně jako v případě těla požadavku, obsahuje informace, které mají být přeneseny, nicméně tentokrát opačným směrem ze strany serveru ke klientovi. Stejně jako v případě požadavků mají zpravidla podobu JSON nebo XML.

HTTP odpověď může mít například následující podobu:

```
HTTP/3 200 OK
Date: Mon, 27 Jul 2021 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Content-Length: 80
Content-Type: text/xml

<?xml version="1.0" encoding="UTF-8"?>
<note>
     <to>John</to>
     <from>Jake</from>
     <message>Hello world!</message>
</note>
```

Ukázka 2 - HTTP odpověď [Zdroj: Chat GPT]

Na počátečním řádku je opět jako v případě požadavku specifikována verze HTTP protokolu. Následující hlavičky poskytují důležité informace o zdroji, ze kterého byla odpověď zaslána, spolu s relevantními informacemi o přenesených datech. Samotné tělo odpovědi pak prezentuje potřebné informace, v uvedeném případě ve formátu XML.

4.2 Formáty pro reprezentaci dat

Při používání nebo vytváření webových API jsou používány různé formáty pro reprezentaci dat, které slouží k efektivní komunikaci a výměně informací mezi softwarovými aplikacemi. V této části budou rozebrány dva nejčastější formáty JSON a XML.

4.2.1 JSON

JSON (JavaScript Object Notation) je standardní textový formát pro reprezentaci strukturovaných dat založený na objektové syntaxi jazyka JavaScript. Běžně se používá k přenosu dat ve webových aplikacích, především k odeslání informací z klienta na server a naopak, čímž umožňuje zobrazení těchto dat na webových stránkách. [14].

Je navržen tak, aby byl odlehčený a snadno čitelný pro lidi. Jeho textový charakter a nezávislost na jazyce [15], díky čemuž se JSON stává vhodným jazykem pro výměnu dat mezi různými systémy a aplikacemi, umožňuje snadnou komunikaci a spolupráci mezi různými technologickými prostředími a usnadňuje interoperabilitu mezi různými softwarovými platformami.

Struktura

Objekt JSON obsahuje nula, jedna nebo více dvojic klíč-hodnota, kterým se také říká atributy. Objekt je obklopen složenými závorkami. Každá dvojice klíč-hodnota je oddělena čárkou. Dvojice klíč-hodnota se skládá z klíče a hodnoty oddělených dvojtečkou. Klíč je řetězec, který identifikuje dvojici klíč-hodnota[16].

V následující ukázce je příklad dat ve formátu JSON znázorňující jednotlivé datové typy.

```
{
  "retezec": "Text",
  "desetinne_cislo": 3.14,
  "pole": [1, 2, 3, 4, 5],
  "objekt": {
    "vnoreny_retezec": "Vnoreny text",
    "vnorene_cislo": 123
  },
  "boolean": true,
}
```

Ukázka 3 – Data ve formátu JSON [Zdroj: Chat GPT]

4.2.2 XML

XML (Extensible Markup Language) je značkovací jazyk, který se používá především k definování a přenosu dat ve strukturovaném a sdíleném formátu.[17] Díky svým předdefinovaným pravidlům XML usnadňuje bezproblémový přenos dat po sítích a zajišťuje, že informace mohou být přesně interpretovány a zpracovány různými softwarovými systémy.

XML dokument obsahuje několik základních částí. Deklarace XML popisuje specifikace verze a kódování, kořenový element, který slouží jako primární značka definující zastřešující strukturu. Dále dokument obsahuje elementy XML reprezentující konkrétní data s jejich atributy a vnořenými elementy.

V následující ukázce je příklad dat ve formátu XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giovanni Rana</author>
  </book>
  <book category="WEB">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
  </book>
</bookstore>
```

Ukázka 4 – Data ve formátu XML [Zdroj: Chat GPT]

Jak je již v příkladu vidět, jazyk XML nerozlišuje datové typy, nicméně narozdíl od JSONu nebyl navržen pouze jako formát pro serializaci, a proto je oproti jazyku JSON mnohem komplexnější a složitější ke čtení.

Oproti tomu však XML těží z dobře definovaných schémat, která umožňují validovat strukturu a data, což zajišťuje konzistenci dat a redukuje riziko chyb.

Přestože každý z uvedených formátů pro reprezentaci dat má své výhody a nevýhody a jsou oba používány pro přenos dat. V současné době je JSON díky své jednoduchosti a čitelnosti populárnější. [18]

4.3 Využití webových API

Komunikace přes HTTP dotazy a univerzální formáty pro reprezentaci dat zajišťující komunikaci mezi různými druhy systémů nezávisle na využité technologii nebo rozhraní činí z webových API mocný nástroj, který může značné urychlit a zjednodušit vývoj nových softwarových aplikací.

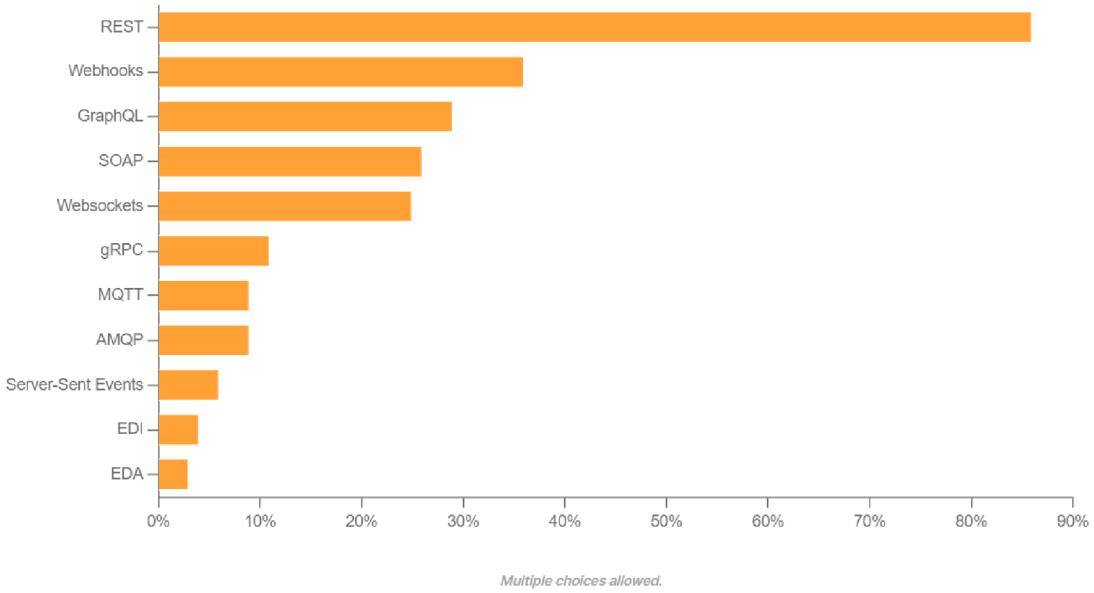
Vývojáři mohou pomocí webových API do svých aplikací přidávat funkce (např. zpracování plateb nebo získávání dat z různých databází) od jiných poskytovatelů do svých stávajících řešení nebo vytvářet kompletně nové aplikace využívající služby, které dané API poskytuje. Vývojáři se tak nemusí zabývat zdrojovým kódem a pochopením daného řešení k dosáhnutí potřebné funkcionality a jednoduše přes webové API připojí svůj software k jinému, který danou funkcionalitu realizuje.

Jinými slovy, rozhraní API slouží jako abstrakční vrstva mezi dvěma systémy, jenž skrývá složitost a fungování druhého systému.[2]

5 Architektury webových API

Architektura webových API definuje základní návrh a strukturu webových rozhraní, které umožňují softwarovým aplikacím vzájemně komunikovat a vyměňovat si data prostřednictvím internetu. Slouží tak k definování pravidel, podle kterých mohou aplikace komunikovat a vyměňovat si data. Kromě toho také poskytuje strukturu, podle níž jsou data organizována a přenášena, což zajišťuje, že obě strany (klient a server) budou schopny spolehlivě spolupracovat.

Na základě průzkumu z roku 2023 společnosti Postman [19], který zkoumal popularitu různých architektur pro tvorbu webových API, byly získány následující výsledky. Respondenti měli možnost vybrat více než jednu možnost.



Obrázek 2 – Popularita architektur webových API [19]

Ačkoliv architektura REST zaznamenala drobný pokles na 86 % oproti 89 % z roku 2022 a 92 % z roku 2021, stále je jednoznačně nejpoužívanější architekturou pro tvorbu webových rozhraní. Výrazný propad však zaznamenala architektura SOAP, kterou v letošním roce používalo pouze 26 % respondentů oproti 34 % v loňském roce. Díky tomuto propadu ji na třetím místě v žebříčku nahradila architektura GraphQL, kterou letos využívalo 29 % účastníků průzkumu [19]. Pro účely této kapitoly budou vybrány čtyři nejpoužívanější architektury na základě výše

uvedeného průzkumu. Podrobnějšímu zkoumání pak bude podrobena architektura REST, právě kvůli její neustále převažující popularitě a významnému vlivu na tvorbu webových rozhraní. Z tohoto důvodu také budou v dalších částech práce podrobněji zkoumány a porovnávány technologie právě pro tvorbu této architektury.

5.1 REST

REST (Representational State Transfer), je architektura pro vytváření webových služeb, kterou poprvé v roce 2000 ve své práci disertační práci představil Roy Fielding [22], jenž je zároveň jedním ze spoluautorů protokolu HTTP. Od té doby se architektura REST společně s protokolem HTTP stala v podstatě standardem pro vývoj webových rozhraní.

Architektura REST je orientovaná na zdroje. Zdroje jsou objekty s typem, přidruženými daty, vztahy k jiným zdrojům a metodami, které se zdroji pracují, což se dá přirovnat k objektům v objektově orientovaných jazycích s tím rozdílem, že zdroje zpravidla využívají pouze čtyři standardní CRUD metody, které již byly zmíněny výše (HTTP GET, POST, PUT a DELETE). Zatímco instance objektů v objektově orientovaných jazycích má zpravidla větší množství metod. Zdroje je možné seskupovat do kolekcí, přičemž každá kolekce obsahuje pouze jeden typ zdroje [20], podobně jako v objektově orientovaných jazycích.

Klíčové vlastnosti, které určují architekturu REST definoval Bill Burke ve své knize RESTful Java with JAX-RS [21]. Tyto vlastnosti by měly být následující:

Adresovatelnost

Myšlenka adresovatelnosti znamená, že každý zdroj by měl být dosažitelný pomocí jedinečného identifikátoru, což umožňuje propojitelnost a spolupráci mezi různými aplikacemi a službami. V případě architektury REST je adresovatelnost řízena pomocí URI, které identifikuje umístění zdroje v síti. Formát URI je standardizován způsobem, který zobrazuje ukázka 5:

```
scheme://host:port/path?queryString#fragment
```

Ukázka 5 - Formát URI [21]

Schéma označuje protokol komunikace (zpravidla výše zmíněné HTTP nebo HTTPS). Hostitel je pak doménové jméno, popřípadě IP adresa, využívající se pro identifikaci zdroje v síti následované nepovinným portem. Následuje cesta, což je seznam textových segmentů oddělených znakem "/", identifikující konkrétní umístění zdroje v síti nebo na serveru. Řetězec dotazu je nepovinný a je oddělen znakem "?", obsahuje seznam parametrů ve formě dvojic klíč - hodnota, oddělených znakem "&". Segment ohraničený znakem "#" se zpravidla používá jako odkaz na určité místo v dokumentu.

Jednotné rozhraní

Princip jednotného rozhraní klade důraz na omezení množství operací, které mohou být na zdrojích prováděny. REST omezuje rozhraní na konečnou sadu operací, které jsou standardem pro protokol HTTP, jako jsou především GET, POST, PUT a DELETE. Omezené rozhraní v architektuře REST přináší výhody prostřednictvím jednoduché a předvídatelné struktury, která usnadňuje pochopení a integraci webových služeb pro vývojáře i klienty. Díky využití standardních metod HTTP zajišťuje snadnou interoperabilitu mezi různými systémy a umožňuje efektivní využití ukládání do mezipaměti, což vede k vyšší škálovatelnosti a výkonu.

Zaměření na reprezentaci

Tento princip v rámci architektury REST zdůrazňuje, že zdroje identifikované jednou URI mohou být reprezentovány různými formáty. Různé aplikace mohou vyžadovat různé formáty. Tento formát je v HTTP definován dle hlavičky Content-Type a rovněž je možné pomocí hlavičky Accept si stanovit formát preferovaného formátu odpovědi. Využívání různých formátů tak umožňuje komunikaci mezi systémy používající odlišné technologie.

Bezstavovost

Bezstavovost znamená, že server neukládá žádné relace klienta. Veškerý stav je udržován a spravován naopak klientem, který přenáší relevantní data na server s každým požadavkem dle potřeby. Server tak zůstává bezstavový a zaměřuje se

pouze na správu prostředků, které poskytuje. Tím, že server nemusí udržovat a spravovat relace klientů se snižuje náročnost aplikace a je umožněno efektivní škálování. Tento princip můžeme také nazvat jako tzv. tenký klient, jehož opakem je tlustý klient, který udržuje veškerý stav relace na serveru, což může být náročné na správu a údržbu systému.

HATEOAS (Hypermedia As The Engine Of Application State)

Poslední vlastností určující architekturu REST podle Burkeho [21] je HATEOAS. Tato vlastnost umožňuje klientům objevovat a interagovat s aplikací prostřednictvím hypertextových odkazů vracených serverem v rámci dat. Výstupem tedy nemusí být vždy pouze data, ale i odkazy na další zdroje. Například pokud chceme pomocí metody GET získat informace o produktech, server nevrátí statický seznam všech produktů najednou, což by mohlo vést k velké zátěži na klienta, ale vrátí pouze částečný seznam s odkazem na další produkty.

5.1.1 Principy

Výše uvedené definice vycházejí ze základních principů, které ve své disertační práci Architectural Styles and the Design of Network-based Software Architectures [22] definoval Roy Fielding. Architektura REST je založena na následujících principech:

Klient – server (Client Server)

Princip klient-server neboli princip oddělení zájmů, zvyšuje přenositelnost uživatelského rozhraní mezi platformami a vylepšuje škálovatelnost systému úpravou serverových komponent. V kontextu webového prostředí hraje významnou roli tím, že umožňuje nezávislý vývoj komponent, což podporuje požadavky různých organizací na internetové úrovni. Tímto způsobem se nejen zvyšuje efektivita a škálovatelnost systému, ale též podněcuje inovace a vývoj komponent, což je klíčové pro dynamiku a adaptabilitu v internetovém prostředí.

Bezstavovost (Stateless)

Princip bezstavovosti zajišťuje, že server neukládá žádné relace klienta a veškerý stav je spravován klientem. Tento princip je již podrobněji popsán v přechozí části.

Ukládání do mezipaměti (Cache)

Princip ukládání do mezipaměti hraje důležitou roli při optimalizaci výkonu a snižování zátěže na server. Tento princip spočívá v tom, že klientská aplikace si ukládá odpovědi na předchozí dotazy na server, aby předešla opakovaným požadavkům na stejná data. To znamená, že při každém dotazu na server se kontroluje, zda má klient již odpověď v mezipaměti. Pokud ano, může odpověď získat přímo z mezipaměti místo opakovaného dotazování serveru, což výrazně zrychluje odezvu a snižuje zátěž na server

Jednotné rozhraní (Uniform Interface)

Princip jednotného rozhraní popisuje komunikaci mezi klientem a serverem pomocí standardních a generických operací. Tento princip již byl opět podrobněji popsán výše.

Vícevrstvý systém (Layered System)

Strukturování systému do hierarchických vrstev umožňuje efektivní oddělení funkcí a zjednodušení celkové struktury. Omezení viditelnosti každé komponenty na bezprostřední vrstvu snižuje složitost a poskytuje nezávislost na ostatních komponentách, což usnadňuje správu a údržbu. Zprostředkovatelé v rámci vrstveného systému hrají důležitou roli při zlepšení škálovatelnosti. Vrstvený systém však může způsobit zvýšenou režii a latenci.

Kód na vyžádání (Code-On-Demand)

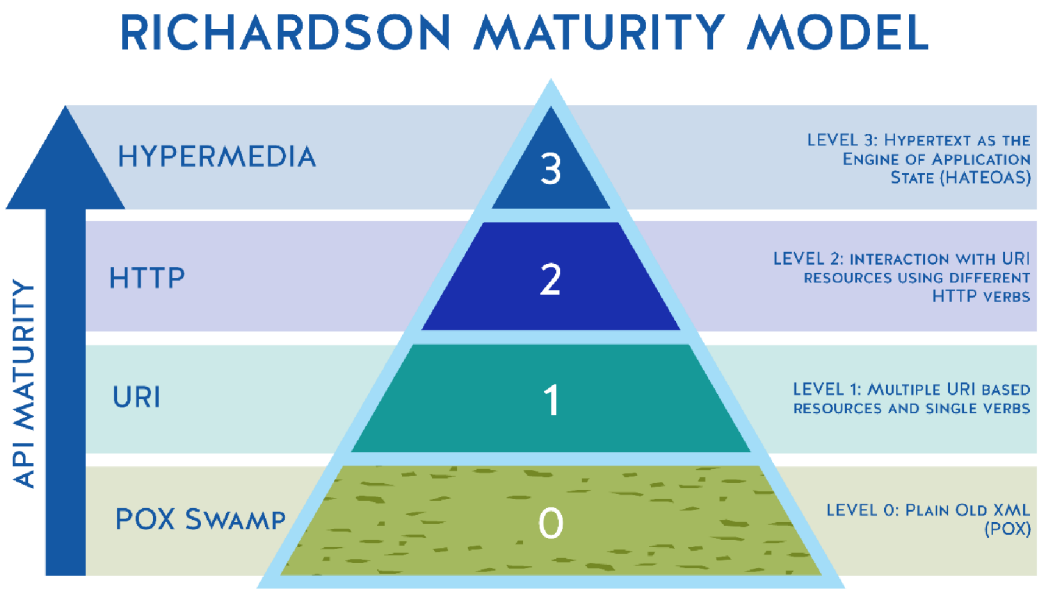
Posledním principem architektury REST je princip code-on-demand, který umožňuje klientům dynamicky získávat a spouštět kód ze serveru v průběhu provozu. Tento přístup významně snižuje nároky na předem definované implementace klienta a podporuje flexibilitu systému tím, že umožňuje rozšíření funkcí po nasazení.

5.1.2 Richardsonův model zralosti

Richardsonův model zralosti (Richardson Maturity Model), představený Leonardem Richardsonem [23], představuje konceptuální rámec pro hodnocení úrovně zralosti

(maturity level) RESTových služeb a slouží k jejich hodnocení a posouzení kvality návrhu a implementace.

Richardsonův model zralosti definuje celkově čtyři úrovně zralosti znázorněné na následujícím obrázku.



Obrázek 3 - Richardsonův model zralosti [23]

Úroveň 0 – Swamp of POX

Na nulté úrovni, nazývané jako Swamp of POX (Plain Old XML), se rozhraní chová spíše jako klasická webová služba než plnohodnotné RESTful API. Komunikace je na této úrovni omezena pouze na HTTP POST požadavky a chybí zde jednoznačná identifikace zdrojů pomocí URI. Tato úroveň představuje počáteční fázi, kde implementace REST principů teprve začíná formovat. [23]

Úroveň 1 – Zdroje

Na první úrovni modelu dochází k začlenění identifikace zdrojů pomocí URI. Komunikace je však stále omezena pouze na metodu HTTP POST pro veškerou manipulaci s daty. Tato úroveň představuje začátek strukturovanějšího přístupu k definici a manipulaci s daty, avšak stále zůstává na začáteční úrovni v porovnání s plně vyspělými RESTful rozhraními. [23]

Úroveň 2 – HTTP metody

Na druhé úrovni, anglicky označované jako HTTP Verbs, nastává významné posunutí směrem k plnohodnotnému RESTful rozhraní. Zde se začínají plně využívat standardní metody HTTP, jako jsou GET, POST, PUT a DELETE, pro manipulaci s daty. Na této úrovni se také již používají návratové kódy a je možné kontrolovat, zda nedošlo k chybám. [23]

Úroveň 3 – HATEOAS

Na nejvyšší úrovni modelu dochází k plnému využití konceptu HATEOAS. Tato úroveň je tedy charakterizována přítomností odkazů na zdroje, což umožňuje dynamické a samo popisující ovládání klienta. Díky HATEOAS se vytváření spojení mezi zdroji stává snadným, bez nutnosti lidského zásahu, což výrazně přispívá k automatizaci řízené klientem. [23]

5.2 Webhooks

Webhooks představují typ rozhraní API řízeného událostmi [24], který umožňuje jednomu systému (klientovi) automaticky informovat druhý systém (server) o událostech, které nastaly [26]. Tento typ integrace zvyšuje flexibilitu a schopnost rychle reagovat na události.

Princip fungování webhooks spočívá ve sledování konkrétních událostí na klientovi a zasílání informací o těchto událostech serveru. Klient nastaví webhook tím, že poskytne serveru jedinečnou adresu URL a specifikuje události, které je třeba sledovat. Poté, když se na klientovi vyskytne registrovaná událost, server automaticky odešle HTTP požadavek s informacemi o události na URL, kterou si server zaregistroval [25]. Tato notifikace může obsahovat různé informace, jako jsou data, čas, typ události atd. Server může poté na základě této notifikace provést příslušné akce a aktualizace v reálném čase.

Z tohoto důvodu jsou webhooks známé také jako reverzní API nebo push API, protože odpovědnost za komunikaci nespočívá v aktivním dotazování klienta, ale v automatickém odesílání dat serverem při výskytu daných událostí [25]. Webhooky tímto způsobem minimalizují zbytečnou komunikaci mezi klientem a serverem, což

zvyšuje efektivitu a snižuje latenci. Eliminuje se tím také potřeba pravidelného dotazování na změny a snižuje se riziko ztráty dat.

Webhooky nacházejí uplatnění především v oblastech, kde je potřeba sledovat nebo reagovat na určité události. Příkladem mohou být notifikace na sociálních sítích, sledování událostí v e-commerce transakcích, nebo v platebních systémech [26].

5.3 GraphQL

GraphQL je dotazovací jazyk pro manipulaci s daty prostřednictvím API a běhový engine pro provádění těchto dotazů. Byl vyvinut společností Meta (dříve Facebook) a v roce 2015 byl vypuštěn jako open-source projekt.[27]

Jednou z hlavních vlastností GraphQL je možnost deklarativního načítání dat, což znamená, že klient může přesně specifikovat, jaká data potřebuje z rozhraní API. Na rozdíl od tradičního přístupu s několika koncovými body, které vracejí samostatná data, GraphQL server poskytuje jediný koncový bod a odpovídá přesně s daty, která byla klientem vyžádána.[27]

Tým společnosti Meta, který stál za vývojem GraphQL, formuloval následující principy tohoto jazyka [28].

- **Definice datového formátu** – Přesná definice struktury odpovědí na dotazy usnadňující psaní dotazů přizpůsobených potřebám aplikace.
- **Hierarchická Struktura** – Efektivní spolupráce s grafově strukturovanými daty, která umožňuje vytvářet uživatelská rozhraní s hierarchickou strukturou.
- **Silná typovost** – Každá část dotazu odpovídá určitému typu, poskytující výstižné chybové zprávy před vykonáním dotazu.
- **Protokol, ne úložiště** – Podpora volitelných funkcí pro každé pole na serveru, využívající existující kódy a datové modely.
- **Introspektivní přístup** – Poskytování introspektivního přístupu pro dotazování na podporované typy, což podporuje rychlé učení a zkoumání rozhraní API.

- **Verze bez omezení** – Struktura dat je plně determinována klientem, což umožňuje přidávání nových polí bez vlivu na existující klienty a zpětně kompatibilní odstraňování starších funkcí.

Díky výše uvedeným vlastnostem a principům nabízí GraphQL flexibilitu v dotazech a minimalizaci nadbytečných dat při komunikaci mezi klientem a serverem. Klienti mohou specifikovat přesně, jaká data potřebují, což umožňuje efektivnější výměnu informací. Tím, že klienti definují strukturu svých dotazů, není nutné přenášet zbytečné informace, což vede ke zrychlení odezvy.

GraphQL je tak díky svým vlastnostem vhodné používat například tam kde je potřeba pružnější kontrola nad daty, která jsou získávána ze serveru. To mohou být například mobilní aplikace, chytré hodinky nebo IoT zařízení, kde velmi záleží na rychlostech [29]. Vhodné využití má také pro spojování dat z různých zdrojů.

5.4 SOAP

SOAP (Simple Object Access Protocol) je protokol založený na XML, který se používá k výměně informací v decentralizovaných a distribuovaných systémech. Zprávy ve formátu SOAP jsou přenášeny pomocí libovolného vhodného transportního mechanismu, přičemž v současné specifikaci je preferován protokol HTTP [30].

Struktura dokumentů v protokolu SOAP je organizována kolem kořenových elementů a podřízených elementů s hodnotami a dalšími specifikacemi. Komunikace mezi klientem a serverem začíná odesláním XML dokumentu obsahujícího požadavek, který specifikuje požadovanou metodu a případné parametry. Server poté reaguje odpovídajícím XML dokumentem, který obsahuje výsledky provedené metody [30].

Navzdory klesající popularitě SOAP nabízí řadu výhod. Jeho rozšiřitelnost umožňuje vývojářům flexibilně přidávat funkce a rozšiřovat možnosti pomocí jmenných prostorů XML a vlastních prvků záhlaví. SOAP může pro komunikaci využívat řadu jiných protokolů, kromě HTTP to může být například SMTP (Simple Mail Transfer Protocol). Použití schématu XML přispívá k lepšímu ověřování dat a snižuje riziko chyb v komunikaci [34].

Na druhé straně má SOAP i řadu nevýhod. Velikost zpráv založených na XML může způsobit nižší výkon, což způsobuje méně efektivní přenos dat. Složitá povaha XML zpráv a celkově protokolu SOAP může přinést do systému nadměrnou složitost [34].

I když popularita protokolu SOAP neustále klesá ve prospěch jiných přístupů, stále existují oblasti, kde se SOAP používá. Své využití nachází zejména v odvětvích, kde jsou bezpečnost či transakční spolehlivost důležitým aspektem, což jsou například finanční služby či zdravotnictví.

5.4.1 WSDL

WSDL (Web Services Description Language) slouží k popisu webových služeb založených na protokolu SOAP [31] pomocí formátu XML. Soubor WSDL obsahuje informace o koncových bodech dané webové služby a také definuje, jaké operace nebo funkce jsou v dané webové službě k dispozici [32], což zjednodušuje interoperabilitu mezi systémy používající protokol SOAP.

Soubor WSDL obsahuje řadu tagů, které definují datové typy, specifikují strukturu zpráv nebo identifikují operace služby. Dále propojuje protokol s transportním mechanismem a specifikuje koncový bod (URL). [33]

6 Databázové systémy pro webová API

Databáze je systematicky uspořádaný soubor strukturovaných informací nebo dat, nejčastěji uložených elektronicky v počítačovém systému. Ovládání databáze je obvykle svěřeno systému pro správu databází DBMS (Database Management System). Souhrnně se data, DBMS a související aplikace, nazývají databázovým systémem [35].

Databázové systémy v kontextu webových API slouží k ukládání, správě a zabezpečení dat potřebných pro API. Webová API využívají databáze k získávání informací, aktualizaci dat a zajištění konzistence. Celkově vzájemná integrace mezi webovým API a databázemi umožňuje bezpečnou výměnu informací mezi různými částmi softwarového systému.

6.1 Typy databází

Existuje řada různých typů databázových systémů, každý je navržený s ohledem na specifické potřeby dat, která jsou zpracovávána. Tato kapitola popisuje nejčastější typy databázových systémů [36].

Relační databáze

Relační databáze jsou založeny na relačním datovém modelu, který ukládá data ve formě řádků a sloupců, které společně tvoří tabulku. Pro manipulaci a správu dat využívají relační databáze jazyk SQL (Structured Query Language). Každá tabulka v databázi obsahuje klíč, který činí data unikátními v porovnání s ostatními.

NoSQL databáze

NoSQL (Not Only SQL) představuje typ databáze určený pro ukládání různorodých datových sad. Na rozdíl od relačních databází, NoSQL ukládá data nejen ve formě tabulek, ale využívá několik různých formátů. NoSQL bylo vytvořeno především s cílem lépe zpracovávat velké množství různorodých dat. Relační databáze totiž mohou narazit na omezení při práci s velkými objemy dat nebo daty s nestandardní strukturou, což vedlo k potřebě nových přístupů k ukládání a zpracování dat.

Objektově orientovaná databáze

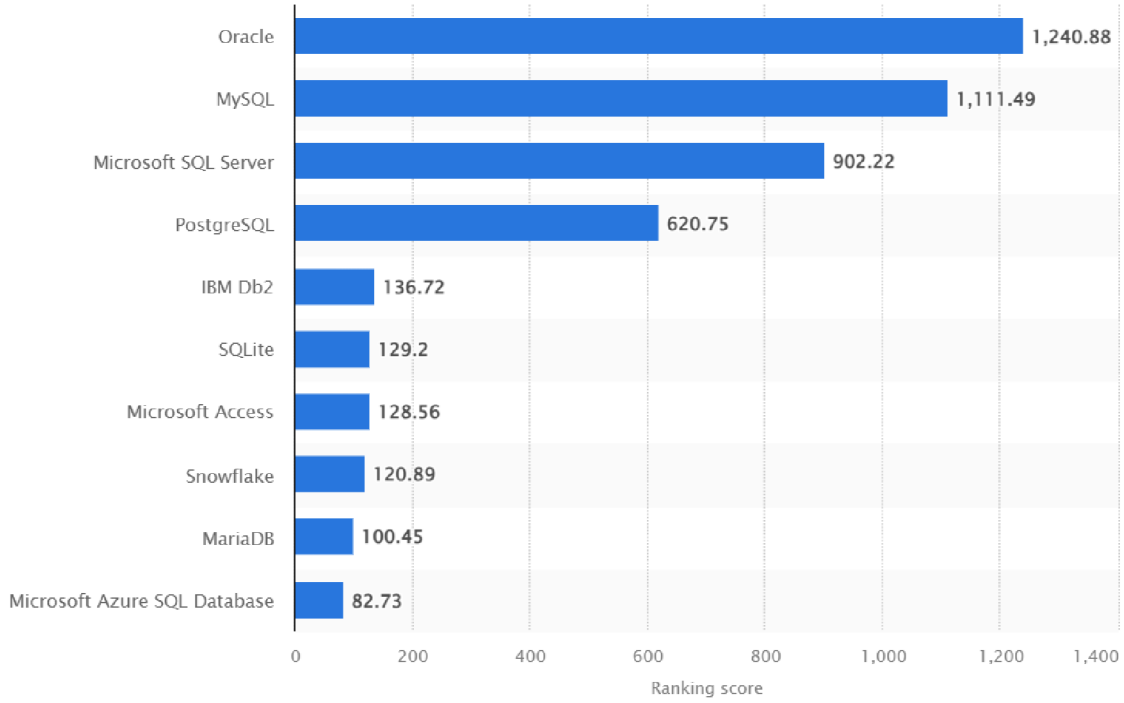
Využívá objektově orientovaný model pro ukládání dat v databázi. Data jsou reprezentována a ukládána jako objekty, které odpovídají objektům používaným v objektově orientovaných programovacích jazycích.

Hierarchická databáze

Jedná se o typ databáze, který uspořádává data ve formě stromové struktury vztahů rodič-dítě. Záznamy jsou uloženy jako uzly a propojené odkazy. Každý podřízený záznam ve stromu má právě jednoho rodiče, zatímco každý rodičovský záznam může mít více záznamů podřízených.

6.2 Relační databázové systémy

Relační databázové systémy v současnosti patří k nejpopulárnějším, a to i přes velký vzestup nových databázových systémů, především výše zmíněného NoSQL. Tato kapitola je věnována vybraným relačním databázovým systémům, jimiž jsou Oracle Database, Microsoft SQL Server a PostgreSQL, viz Obrázek 4. Všechny zmíněné databázové systémy následně jsou vybrány i pro testování s jednotlivými technologiemi pro vytváření webových API. Ačkoliv databáze MySQL je populárnější než databáze PostgreSQL, byla vybrána právě databáze PostgreSQL, protože se jedná o databázi, která je nejčastěji používána s technologií Django, která bude podrobněji rozebrána níže. Veškeré popsané databáze budou následně použity při testování.



Obrázek 4 - Popularita relačních databázových systémů [49]

6.2.1 PostgreSQL

PostgreSQL je open-source relační databázový systém, jehož vývoj započal v rámci projektu POSTGRES na Kalifornské univerzitě v Berkeley v roce 1986. S více než třicetiletou historií aktivního vývoje se PostgreSQL řadí mezi přední relační databázové platformy s výrazným zaměřením na robustnost, integritu dat a schopností efektivně řešit náročné datové úlohy. [37]

Přednost databáze PostgreSQL její otevřený zdrojový kód, což poskytuje uživatelům flexibilitu a možnost rozšíření podle konkrétních potřeb. Databázový systém podporuje rozmanité datové typy, od primitivních a strukturovaných po dokumentové a geografické.

Dokumentové datové typy zahrnují JSON a JSONB (binary JSON) [37] a jsou užitečné při práci s nestrukturovanými nebo polostrukturovanými daty. Geografické datové typy jsou vhodné pro práci s geografickými informacemi a prostorovými daty. Tuto funkci plní rozšíření PostGIS [37], které nabízí pokročilé geografické funkce, umožňující uživatelům provádět prostorové dotazy a analýzy. PostgresSQL tak

může být vhodná pro projekty zaměřené na geografickou informační analýzu, mapování, nebo prostorové modelování.

Bezpečnostní prvky, jako například sofistikované metody ověřování, robustní systémy řízení přístupu a vícefaktorové ověřování [37], zajišťují velmi vysokou úroveň zabezpečení databáze PostgreSQL. Pomocí rozsáhlých funkcí, které zahrnují pokročilé indexování a optimalizaci dotazů až po různé mechanismy obnovy po havárii [37], poskytuje PostgreSQL poměrně komplexní řešení pro různé aplikace.

6.2.2 Oracle Database

Oracle Database je relační systém pro správu databází vyvinutý a distribuovaný společností Oracle Corporation. Databázový systém byl vytvořen v roce 1977 Larrym Ellisonem a dalšími inženýry a tato databáze se stala jedním z nejprestižnějších a nejrozšířenějších systémů pro ukládání, organizaci dat [38].

Oracle databáze klade důraz na vysokou dostupnost, kterou zajišťuje především funkce Oracle Data Guard, která umožňuje plynulé přepínání mezi primární a sekundární databází [38], a díky které se minimalizuje pravděpodobnost výpadků.

Dalším prvkem, na který se Oracle databáze zaměřuje je bezpečnost, kterou Oracle zajišťuje pomocí funkcí jako například Transparent Data Encryption (TDE), které šifruje data u zdroje i při exportu [38].

V oblasti analýzy dat, Oracle poskytuje nástroje, jako například Oracle Analytic Processing a Advanced Analytics. Ty umožňují provádět složité analytické výpočty nad obchodními daty a vytvářet prediktivní obchodní modely [38].

Poslední vlastností, která bude v případě Oracle Database stojí za zmínku je její snadná přenositelnost. Tento databázový systém podporuje velkou řadu síťových protokolů a hardwarových platforem, čímž Oracle Database značně usnadňuje přechod na jiné technologie [38].

6.2.3 Microsoft SQL Server

Microsoft SQL Server představuje relační systém pro správu databází, známý pro svou schopnost podpory transakčního zpracování, business intelligence a analytiky

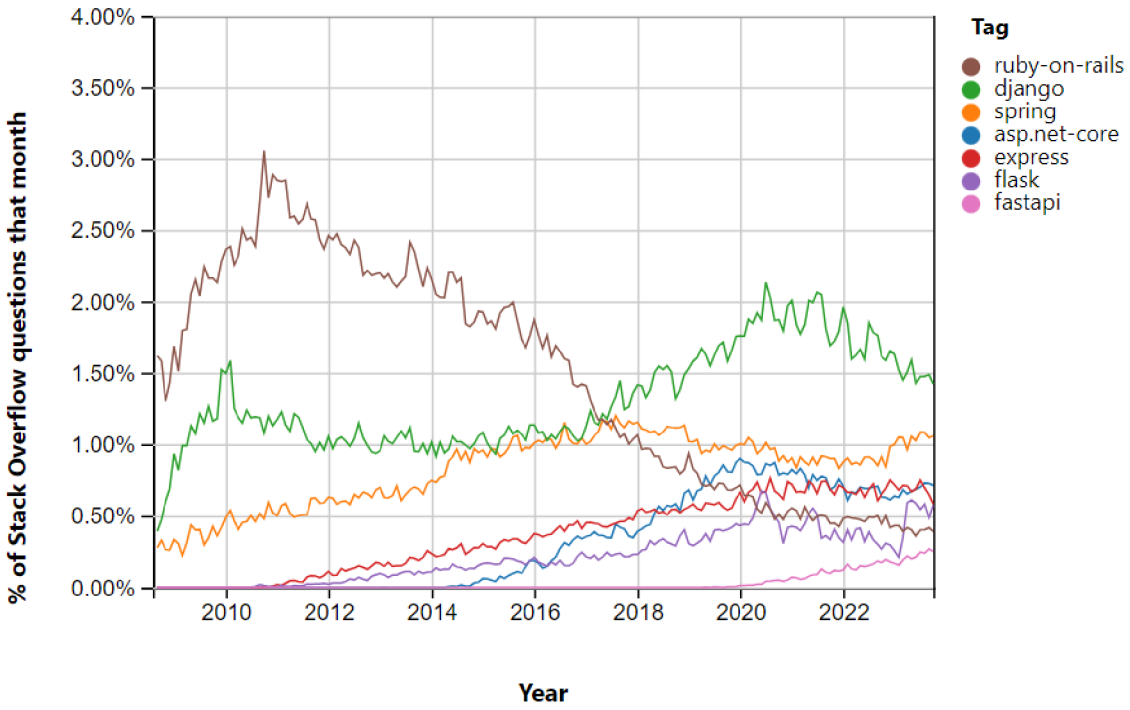
v podnikovém IT prostředí. Microsoft SQL Server je stejně jako předchozí databázové systémy postaven na SQL, avšak využívá jazyk Transact-SQL (T-SQL) [39], který přidává specifické funkce a vlastnosti pro použití v produktech společnosti Microsoft. Architektura Microsoft SQL Server obsahuje Database Engine skládající se z relačního engine pro zpracování dotazů a úložného engine pro správu dat. Pod Database Engine je SQL Server Operating System (SQLOS), který zajišťuje nižší úrovňové funkce.[39]

Microsoft SQL Server poskytuje velké množství nástrojů pro správu a analýzu dat, což zahrnuje funkce pro business intelligence. Integrace dat do aplikací a jejich analýza jsou snadno proveditelné díky integrovaným možnostem Microsoft SQL Serveru, které jsou dostupné jak v lokálním, tak i v cloudovém prostředí, což zajišťuje flexibilitu a škálovatelnost pro uživatele [40].

Možná největší předností Microsoft SQL Server je pak velmi snadná integrace a spolupráce s ostatními produkty a technologiemi, které má společnost Microsoft v portfoliu. Sem patří také framework ASP.NET pro tvorbu webových API, který bude podrobněji rozebrán v další kapitole.

7 Technologie pro vývoj RESTful API

Tato kapitola je zaměřena na konkrétní technologie, které umožňují implementaci RESTful API. Výběr konkrétních technologií je určen pomocí tzv. tagů na webu Stackoverflow [50], kde byly vybrány nepoužívanější technologie pro tvorbu webových API, z nichž jsou vybrány stejně jako v případě relačních databázových systémů čtyři nejzmiňovanější technologie, kterými jsou Django, Spring a ASP.NET Core. Vybrané technologie také jsou použity pro testování v další části práce.



Obrázek 5 - Popularita technologií pro RESTful API [50]

7.1 Model-View-Controller

Všechny frameworky, které jsou v této kapitole zmíněny, jsou postavené na architektonickém vzoru model-view-controller (MVC).

MVC se používá ve vývoji softwaru k organizaci a strukturování kódu. Jeho hlavním cílem je oddělit logiku aplikace od zobrazení a spravovat tok dat a událostí efektivním způsobem. Tento vzor rozděluje aplikaci do tří hlavních komponent: Model, View a Controller [41].

Model je komponenta nezávislá na uživatelském rozhraní obsahující logiku aplikace, výpočty, validace a přístup k databázi. Funkce modelu spočívá v přijetí parametrů a vydání dat ven.

View se zaměřuje na prezentaci dat uživateli. Jeho úkolem je vizuálně prezentovat informace poskytnuté modelem, obvykle pomocí šablon nebo značkovacího jazyka, bez obsahu složité logiky.

Controller funguje jako prostředník mezi komponentou model a view. Přijímá uživatelský vstup, zpracovává ho a komunikuje s výše uvedenými komponenty model nebo view. Jeho hlavním úkolem je efektivní řízení toku dat a událostí v aplikaci pro bezproblémovou komunikaci.

7.2 C# - ASP.NET Core

Technologie ASP.NET Core, nástupce její starší verze ASP.NET Core, je vybrána do seznamu technologií pro RESTful API, jelikož se jedná o preferovanou technologii pro vývojáře, kteří se rozhodnou vyvíjet webová API v jazyce C# a s pomocí frameworku .NET.

ASP.NET Core je open-source webový framework, který je vyvíjen společností Microsoft ve spolupráci s komunitou vývojářů [42]. Jeho využití zahrnuje tvorbu různých webových rozhraní, především pak rozhraní RESTful, ale také tvorbu moderních webových a cloudových aplikací.

Jak již bylo výše zmíněno, nespornou výhodou využívání produktů společnosti Microsoft je snadná spolupráce s ostatními produkty, které společnost Microsoft nabízí. Mezi ně můžeme zařadit vývojové prostředí Microsoft Visual Studio, které podstatně usnadňuje práci s touto technologií a zároveň nabízí přístup k dalším službám, které společnost Microsoft nabízí, jako například cloudová platforma Microsoft Azure. Další výhodou je snadné importování podpůrných knihoven, frameworků a balíčků fungujících na základě jazyka C# a .NET díky nástroji NuGet Pakage Manager.

.NET Core je designován jako modulární [42], díky čemuž je možné zahrnout pouze potřebné součásti frameworku a je tak možné vytvářet aplikace s minimálními

závislostmi a zbytečným kódem. To ve výsledku vede k menší velikosti aplikace a zlepšuje celkovou spravovatelnost a efektivitu vývoje. Modularita následně hraje roli při optimalizaci výkonu jádra .NET Core. Celkový design frameworku je zaměřen na rychlost a efektivitu, zejména pro aplikace nasazené na straně serveru.

Psaní webových API pomocí ASP.NET Core značně zjednodušuje Entity Framework (EF), což je objektově relační mapovač, který umožňuje vývojářům pracovat s relačními daty pomocí objektově orientovaných konceptů [43], na kterých je založen jazyk C#. EF tak eliminuje potřebu rozsáhlého kódu pro přístup a manipulaci s daty a zajišťuje mapování databázových tabulek na objekty v jazyce C#.

Oproti těmto výhodám má ASP.NET Core i řadu nevýhod. Mezi nejzásadnější patří omezená kompatibilita mezi různými verzemi frameworku a omezená podpora starších frameworků [44], to může v určitých případech působit potíže.[44]

Následující ukázka představuje jednoduchý controller v rámci ASP.NET Core aplikace napsané v jazyce C#.

```
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    // Implementace metody GET /api/products
    [HttpGet]
    public IEnumerable<ProductModel> GetAllProducts()
    {
        // Zjednodusene nacitani produktu
        return _productsRepository.GetProducts();
    }
}
```

Ukázka 6 - ASP.NET Core [Zdroj: Chat GPT]

Controller je dostupný na cestě "/api/products", kde se název cesty ke zdroji určuje pomocí názvu daného controlleru (products) a obsahuje metodu GetAllProducts, označenou atributem [HttpGet], která vrací seznam všech produktů v databázi.

7.3 Java – Spring

Spring je nejčastější volbou pro vývoj webových API v jazyce Java. Jedná se o open-source nástroj navržený pro zjednodušení vývoje podnikových aplikací. Cílem frameworku Spring je poskytnout robustní infrastrukturu pro tvorbu komplexních a rozšiřitelných aplikací. Framework je rozdělen do modulů, které lze používat nezávisle nebo ve spojení, což umožňuje vytvářet aplikace pro různá prostředí, včetně tradičních aplikačních serverů, či cloudových prostředí [45].

Spring Framework využívá princip Inversion of Control (IoC) [51], který zajišťuje, že kontrola běhu programu je předána frameworku. To umožňuje definovat komponenty a jejich závislosti, s následným spravováním vytvářených objektů a jejich závislostí Spring kontejnerem, což je kontejner, do kterého jsou registrovány komponenty a jejich závislosti. Tím je dosaženo snadné konfigurovatelnosti a změny chování aplikace bez potřeby zásahů do kódu.

Na rozdíl od ASP.NET Core, který se primárně zaměřuje na integraci s Visual Studio, je Spring více flexibilní ve volbě vývojových prostředí (IDE). ASP.NET Core nejvíce vyniká ve spojení s Visual Studio, které mu poskytuje nejvyšší podporu. Naopak Spring umožňuje vývojářům vybírat z několika různých IDE.

V následující ukázce je znázorněn příklad controlleru pomocí frameworku Spring v jazyce Java.

```
@RestController
@RequestMapping("/api/products")
public class ProductsController {

    // Implementace metody GET /api/products
    @GetMapping
    public List<ProductModel> getAllProducts() {
        // Zjednodusene nacitani produktu
        return _productsRepository.getProducts();
    }
}
```

Ukázka 7 – Spring [Zdroj: Chat GPT]

Princip práce s frameworkem Spring je velmi podobný jako princip u ASP.NET Core. Controller obstarává HTTP požadavky na zdroje v daném controlleru s cestou /api/products jako v přechozím případě. Anotace GetMapping poté znázorňuje, že se jedná o http metodu GET.

7.4 Python – Django

Django je vysokoúrovňový webový framework napsaný v jazyce Python, který usnadňuje rychlý vývoj bezpečných a udržovatelných webových stránek.

Django implementuje architekturu zvanou Model-View-Template (MVT). MVT je modifikovanou verzí klasického MVC vzoru zmíněného výše, který je specifický pro Django, kde Template má na starost prezentaci dat a část controlleru je řízena samotným frameworkem.[46]

Framework Django je známý především tím, že vyniká v oblasti bezpečnosti a pomáhá vývojářům vyhnout se řadě bezpečnostních rizik jako je například SQL injection či cross-site scripting[47].

Django je dále známý svou schopností škálovat. S využitím architektury "shared-nothing" lze jednotlivé části aplikace nezávisle škálovat, ať už jde o cachovací servery, databázové servery nebo aplikační servery.[47]

Django je vhodnou volbou, pokud je prioritou co nejrychlejší vývoj dané aplikace, jelikož nabízí řadu vestavěných nástrojů a funkcí pro co nejrychlejší vývoj [47].

Jistou nevýhodou tohoto frameworku je pak nižší podpora ze strany Windows, jelikož je nutné používat řadu dodatečného softwaru. Zároveň dle internetových diskuzí [48] používání v systému Linux nabízí značně vyšší výkon oproti systému Windows.

Následující ukázka zobrazuje příklad controlleru pomocí frameworku Django a jazyka Python.

```
@api_view(['GET'])
def get_all_products(request):
     product_repository = ProductRepository()
     products = product_repository.get_all_products()
     return JsonResponse({'products': list(products.values())})
```

Ukázka 8 – Django [Zdroj: Chat GPT]

Python se oproti předchozím uvedeným jazykům značně liší v syntaxi. I tak zde můžeme najít řadu podobností, pro příklad dekorátor @api_view(['GET']), který určuje jaká HTTP metoda je použita. Cesta ke zdroji se obvykle definuje pomocí souboru urls.py. V tomto souboru se nastavují URL cesty a mapují se na odpovídající funkce obsluhující požadavky.

8 Testování rychlostí webových API

V následující kapitole je provedena analýza a porovnání vybraných technologií pro vývoj RESTful API v oblasti rychlosti, výkonnosti a stability při práci s vybranými databázovými systémy. Na základě výsledků je následně určeno, jaké technologie nejlépe spolupracují s jednotlivými databázovými systémy. Vybrané technologie pro tvorbu webových API jsou ASP.NET Core, Spring Boot a Django. Jako testovací databáze pak jsou vybrány SQL Server, Oracle a PostgreSQL.

8.1 Testovací prostředí

Pro účely testování rychlostí webových API je vytvořen virtuální server s operačním systémem Windows Server 2022 Datacenter přes platformu Microsoft Azure. Tento operační systém je vybrán s ohledem na jeho schopnost poskytnout robustní platformu pro nasazení a testování aplikaci. Server disponuje dvěma virtuálnímu CPU, 8 GB RAM paměti a SSD diskem o velikosti 127 GB.

Výše zmíněné parametry serveru mohou mít vliv na rychlosti technologií pro webová API a databázové systémy, avšak v souladu s cílem zajistit konzistentní a objektivní výsledky testů jsou všechny použité technologie pro vývoj API i databázové systémy nasazeny právě na tomto serveru. Tato rovnováha prostředí zajišťuje, že výsledky testů nejsou zkresleny vlivem odlišného hardwarového nebo síťového prostředí a umožní přesné srovnání rychlosti a výkonnosti jednotlivých technologií.

Aby bylo zajištěno co nejmenší ovlivnění výsledků ze strany databázových systémů, budou vlastnosti databází, které mohou mít vliv na rychlost daných databázových systémů, nakonfigurovány co nejpodobnějším způsobem. Všechny používané databáze jsou nakonfigurovány tak, aby byly vyloučeny veškeré šifrovací mechanismy, které by mohly mít vliv na výsledky testování, resp. je vše nastaveno na defaultní hodnoty a je překontrolováno, zda některá z databází šifrování nevyužívá.

Dále jsou pro všechny databázové systémy nakonfigurovány používané paměti. Nastavení paměťových limitů pro databázové systémy představuje poměrně náročnou úlohu, neboť vybrané databázové systémy využívají a spravují paměť různými způsoby. Aby byl tento vliv alespoň z části eliminován, jsou všechny databázové systémy

nakonfigurovány s identickými paměťovými limity. Ve všech případech je pro databázové systémy uvolněno 4 GB, tedy 50 % pamětí RAM, kterou disponuje server.

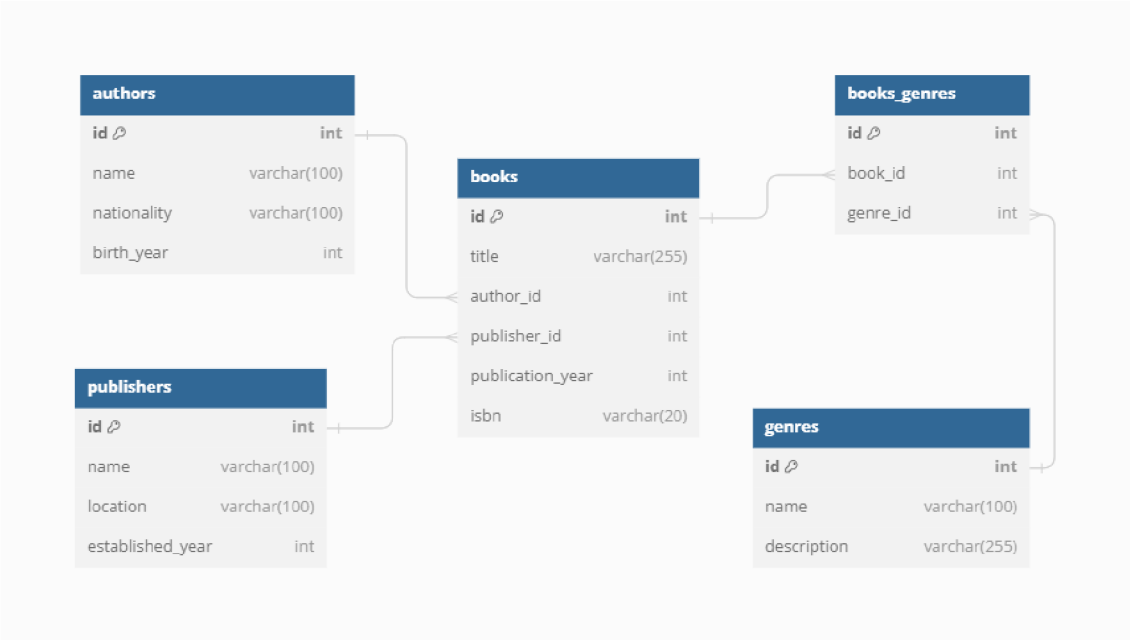
Konkrétní nastavení jsou následující:

- SQL Server má nastavenou maximální možnou paměť pomocí nastavení 'max server memory ' na hodnotu 4096, tedy 4 GB. Max server memory v SQL Serveru určuje maximální množství paměti, které může SQL Server alokovat pro svou interní potřebu, včetně bufferování dat (buffer pool) a cache plánu dotazů.
- Pro Oracle je nastavena hodnota sga_max_size rovněž na hodnotu 4 GB. SGA (System Global Area), je oblast paměti obsahující různé paměťové struktury a datové oblasti.
- Pro PostgreSQL jsou sdílené vyrovnávací zóny (shared buffers) nastaveny na 4 GB. Shared buffers v PostgreSQL určuje množství paměti použité pro sdílené buffery, což je oblast paměti používaná pro cache datových bloků.

Pro dosažení co největší podobnosti z hlediska databázových uživatelů jsou vytvořeny uživatelské účty, které využívají klasické SQL přihlašování pomocí uživatelských jmen a hesel. Všichni uživatelé disponují neomezeným přístupem ke všem databázím a mají kontrolu nad databázovým prostředím. Uživateli v SQL Server je přidělena role sysadmin, u Oracle je to role DBA a v případě databáze PostgreSQL je uživateli přidělena role superuser.

Pro zajištění konzistence výsledků z hlediska struktury tabulek v databázi jsou ve všech třech databázích vytvořeny totožné tabulky a veškeré vazby mezi nimi. Pro testování je zvolena struktura databáze knih, která je velmi dobře uchopitelná a může dobře simulovat reálné prostředí různých systémů. Zároveň, aby byla zajištěna i co největší konzistence

ze strany dat v tabulkách, všechny tabulky obsahují naprosto totožná data. Následující diagram zobrazuje strukturu databázových tabulek, která je pro testování použita.



Obrázek 6 - Testovací databázové schéma [Zdroj: autor]

Hlavní tabulka books má řadu atributů a také obsahuje vazbu 1*N na tabulku s autory a vydavateli, kdy jedna kniha může mít pouze jednoho autora a jednoho vydavatele. Tabulka books však může mít více žánrů, přičemž vzniká vztah N*N, k čemuž slouží pomocná tabulka books_genres zachycující tyto vazby.

API implementovaná pomocí jednotlivých technologií jsou nasazena s nejnovějšími dostupnými verzemi. ASP.NET Core tak bude disponovat verzí .NET 8.0, Spring Boot je nasazen na verzi 3.2.2 a Django na verzi 5.0.1. Pro mapování databázových tabulek všechny technologie využívají nejčastější řešení, což je v případě ASP.NET Core Entity Framework, v případě Spring Boot se jedná o JPA (Java Persistence API) a Django užívá ORM (Object-Relational Mapping).

API jsou následně spuštěna na serveru, kde jsou nakonfigurována tak, aby komunikovala s odpovídající databází pomocí nastaveného connection stringu a aby bylo možné je volat přes předem nastavený port.

K testování API se využívá program Apache JMeter, což je open-source nástroj určený pro testování výkonu webových aplikací. Program Apahce JMeter je nakonfigurován tak, aby současně prováděl pouze jedno volání současně a tedy aby nedošlo ke zkreslení výsledků z hlediska databází, které zpravidla spravují připojení k databázi jiným způsobem. Následně je provedeno 100 opakování daného volání. API jsou volána ze serveru pomocí HTTP dotazů, odezva jednotlivých volání je zaznamenávána, na základě čehož jsou následně určeny všechny potřebné charakteristiky. Následující ukázka znázorňuje příklad HTTP dotazu pomocí veřejné IP adresy serveru, který vrátí seznam knih z databáze ve formátu JSON.

```
http://98.71.48.234:8080/api/books
```

Ukázka 9 - Ukázka HTTP dotazu [Zdroj: autor]

Před samotným testováním je nejprve provedeno několik volání API s cílem stabilizovat časy odezvy a minimalizovat variabilitu výsledků. Jednotlivá testování probíhají pro základní HTTP metody, které budou v API implementovány. Těmito metodami jsou HTTP GET, POST, PUT a DELETE, přičemž největší důraz je kladen především na metodu GET. Metody pro všechny technologie jsou nakonfigurovány tak, aby vždy byly na stejné URL cestě.

- GET - http://98.71.48.234:8080/api/books
- POST - http://98.71.48.234:8080/api/books
- PUT - http://98.71.48.234:8080/api/books/{id}/update
- DELETE - http://98.71.48.234:8080/api/books/{id}/delete

Metoda GET je implementována tak, že získává všechna data v dané databázi a pracuje tak s mnohem větším objemem dat než ostatní zmíněné metody. Výsledky testování této metody tak jsou nejvíce směrodatné. Veškerá provedená testování jsou vždy provedena se stejnými daty pro každou databázi. Konkrétně metoda GET získává 1500 totožných údajů v jednom volání a z toho důvodu má tato metoda mnohem delší časy odezvy než ostatní HTTP metody. Vytváření nových údajů pomocí metody POST je stejně tak prováděno vždy pomocí stejných dat ve formátu JSON, která jsou vložena do těla požadavku. Podobně je testována i metoda PUT, která vždy pro všechny databáze nahrazuje stejná data pomocí jiných dat ve formátu JSON v těle požadavku. Metoda

DELETE je pak opět testována tak, že jsou mazána stejná data pro všechny technologie a databáze. Jednotlivé technologie a databáze jsou vybrány na základě popularity, ale i s ohledem na vybrání takových kombinací, se kterými je možné se často setkat v praxi. Z výsledků pak může být patrné, zda jsou tyto kombinace optimalizovány tak, že jsou schopny spolupracovat rychleji než méně časté kombinace. Porovnání však jsou provedeny mezi všemi technologiemi a všemi databázemi, aby byly vidět možné rozdíly v rychlostech odezvy mezi těmito kombinacemi.

8.2 ASP.NET Core

První testovanou technologií pro vývoj webových API je technologie ASP.NET Core vyvíjená v jazyce C#. Pro mapování databázových tabulek do objektů v jazyce C# je použit nástroj Entity Framework, což je pro mapování relačních databázových tabulek do objektů a celkovou práci s databázemi v ASP.NET Core zpravidla nejčastější řešení.

8.2.1 Představení kódu

Následující ukázka představuje objekt Book se kterým je v rámci testování pracováno a který obsahuje vazby na ostatní potřebné objekty.

```
public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int Author_id { get; set; }
    public int Publisher_id { get; set; }
    public int Publication_Year { get; set; }
    public string Isbn { get; set; }
    [ForeignKey("Author_id")]
    public virtual Author Author { get; set; }
    [ForeignKey("Publisher_id")]
    public virtual Publisher Publisher { get; set; }
    [InverseProperty("Book")]
    public virtual ICollection<Book_Genre> Genres { get; set; }
}
```

Ukázka 10 – Objekt Book pro mapování v ASP.NET Core [Zdroj: autor]

Hlavní třída Book obsahuje kromě vybraných atributů také vazbu na autora a vydavatele na základě jejich ID. K tomu slouží anotace [ForeignKey] použitá nad atributy Author a Publisher, která explicitně určuje sloupce v databázi, které reprezentují cizí klíče pro vztahy s těmito entitami.

Podobně je vytvořena vazba mezi knihou a žánry pomocí asociativní tabulky Books_Genres. Tato tabulka obsahuje záznamy, které propojují knihy s jedním nebo více žánry. Entita Book_Genre obsahuje vlastnosti pro identifikátor záznamu, identifikátory knihy a žánru a odkazy na odpovídající knihu a žánr, kde atribut [InverseProperty] definuje opačný konec asociace v entitním modelu. Tato deklarace pomáhá Entity Frameworku správně určit vztah mezi entitami a optimalizovat načítání spojených dat. Celá tato implementace tak znázorňuje vztah N*N kdy jedna kniha může mít více žánrů a jeden žánr může být přidružen k více knihám.

Klíčové slovo virtual před deklarací přidružených objektů je využito pro tzv. lenivé načítání (lazy loading), což umožňuje načítat přidružené entity z databáze až tehdy, kdy jsou vyžádány.

V databázovém kontextu, který se v Entity Framework používá pro propojení ASP.NET Core aplikace a databáze, jsou následně definovány vlastnosti typu DbSet<T>, které již reprezentují konkrétní tabulky v databázi a umožňují nad těmito entitami provádět CRUD operace.

```
public DbSet<Author> Authors { get; set; }
public DbSet<Publisher> Publishers { get; set; }
public DbSet<Genre> Genres { get; set; }
public DbSet<Book> Books { get; set; }
public DbSet<Book_Genre> Books_Genres { get; set; }
```

Ukázka 11 - DataContext v ASP.NET Core [Zdroj: autor]

Pro účely testování je v aplikaci vytvořen controller, ve kterém jsou implementovány čtyři základní HTTP metody, které jsou podrobeny testování. Těmito metodami jsou GET, POST, PUT a DELETE.

GET

První implementovaná metoda je metoda GetBooks zpracovávající GET požadavky. Tato metoda má za cíl získat všechny knihy z databáze a vrátit je jako odpověď na požadavek.

Metoda získává všechny záznamy z tabulky Books v databázi pomocí výše zmíněného databázového kontextu. Následně pomocí metody Include jsou načteny i přidružené entity Author, Publisher a Genres, aby byly k dispozici v rámci jediného dotazu. Výsledná data jsou následně vrácena jako odpověď metody pomocí Ok(books), která vrací seznam knih a HTTP status kód 200, což označuje úspěšné dokončení požadavku.

```
[HttpGet]
public IActionResult GetBooks()
{
    var books = _context.Books
        .Include(x => x.Author)
        .Include(x => x.Publisher)
        .Include(x => x.Genres)
        .ThenInclude(x => x.Genre)
        .ToList();
    return Ok(books);
}
```

Ukázka 12 - GET metoda v ASP.NET Core [Zdroj: autor]

POST

Druhá metoda CreateBook, zpracovává POST požadavky na výše zmíněné URL /api/books/create. Tato metoda slouží k vytvoření nové knihy v databázi. Tato metoda přijímá parametr book, který reprezentuje novou knihu, která má být vytvořena. Nová kniha je následně přidána do databázového kontextu a suložena do databáze.

Po správném provedení je vytvořena odpověď na požadavek pomocí metody CreatedAtAction. Tato metoda vytváří odpověď s kódem 201, což znamená, že

vytvoření knihy bylo úspěšné. Odpověď obsahuje informace o nově vytvořené knize včetně jejího ID a také samotný objekt nové knihy jako tělo odpovědi.

```
[HttpPost("create")]
Public IActionResult CreateBook([FromBody] Book book)
{
    _context.Books.Add(book);
    _context.SaveChanges();

    return CreatedAtAction(nameof(GetBooks),
    new { id = book.Id }, book);
}
```

Ukázka 13 - POST metoda v ASP.NET Core [Zdroj: autor]

PUT

Třetí testovaná metoda UpdateBook je metoda zpracovávající HTTP PUT požadavky na URL /api/books/{id}/update, kde {id} je identifikátor aktualizované knihy.

Metoda UpdateBook přijímá dva parametry. Prvním z nich je ID, které slouží k identifikaci knihy, která má být aktualizována, a objekt book, který obsahuje aktualizované informace o knize. Pomoci identifikátoru je kniha načtena z databáze, aktualizována na základě objektu book a uložena do databáze. Následně je vrácena odpověď s kódem 200 a upravenou knihou v těle odpovědi.

```
[HttpPut("{id}/update")]
Public IActionResult UpdateBook(int id, [FromBody] Book book) {
    var existingBook = _context.Books.Find(id);

   if (existingBook == null)
       return NotFound();

   existingBook.Title = updatedBook.Title;
   existingBook.Author_id = updatedBook.Author_id;
   existingBook.Publisher_id = updatedBook.Publisher_id;
   existingBook.Publication_Year = updatedBook.Publication_Year;
   existingBook.Isbn = updatedBook.Isbn;
   existingBook.Genres = updatedBook.Genres;

   _context.SaveChanges();
   return Ok(existingBook);
}
```

Ukázka 14 - PUT metoda v ASP.NET Core [Zdroj: autor]

DELETE

Poslední testovaná metoda DeleteBook zpracovává HTTP DELETE požadavky na URL /api/books/{id}/delete, kde {id} je identifikátor knihy, kterou chceme smazat.

Metoda DeleteBook přijímá jeden parametr id, který slouží k identifikaci knihy, která má být smazána. Pokud kniha existuje, metoda nejprve vyhledá související žánry knihy v asociativní tabulce Books_Genres a odebere všechny tyto záznamy. Poté je smazána i sama kniha z tabulky Books. Změny jsou následně uloženy do databáze a pro potvrzení klientovi o smazání je vrácena odpověď s kódem 204, což znamená, že požadavek byl úspěšně zpracován, ale nemá žádný obsah.

```
[HttpDelete("{id}/delete")]
Public IActionResult DeleteBook(int id)
{

    var existingBook = _context.Books.Find(id);

    if (existingBook == null)
       return NotFound();

    var relatedGenres = _context.Books_Genres.Where(bg =>
    bg.Book_Id == id);

    _context.Books_Genres.RemoveRange(relatedGenres);

     _context.Books.Remove(existingBook);
     _context.SaveChanges();

     return NoContent();
 }
```

Ukázka 15 - DELETE metoda v ASP.NET Core [Zdroj: autor]

8.2.2 Měření rychlostí odezvy

Všechny výše uvedené HTTP metody napsané pomocí technologie ASP.NET Core jsou testovány se všemi vybranými databázemi, kterými jsou SQL Server, PostgreSQL a Oracle. Z hlediska rychlostí odezvy a spolupráce technologie s databází se očekává, že ASP.NET Core bude v kombinaci s Entity Framework nejrychleji spolupracovat s databází SQL Server, jelikož se jedná o produkty od společnosti Microsoft, což v tomto kontextu může znamenat lepší optimalizaci

a integraci mezi touto technologií a databází. Výsledky testování pro technologii ASP.NET Core v milisekundách jsou následující.

Výsledky testování ASP.NET Core s databází SQL Server (v ms)						
HTTP metoda	**Průměr**	**Medián**	**Min**	**Max**	**Směrodatná odchylka**	**Variační koeficient**
GET	841	824	593	1764	160	19 %
POST	52	51	48	104	7	13,5 %
PUT	52	50	47	101	6	11,5 %
DELETE	53	51	48	98	7	13,2 %

Tabulka 1 - Výsledky testování ASP.NET Core s databází SQL Server [Zdroj: autor]

Výsledky testování ASP.NET Core s databází PostgreSQL (v ms)						
HTTP metoda	**Průměr**	**Medián**	**Min**	**Max**	**Směrodatná odchylka**	**Variační koeficient**
GET	878	832	630	1776	197	22,4 %
POST	52	49	46	171	15	28,8 %
PUT	49	48	46	99	6	12,2 %
DELETE	52	50	48	94	5	9,6 %

Tabulka 2 - Výsledky testování ASP.NET Core s databází PostgreSQL [Zdroj: autor]

Technologie ASP.NET Core s databází Oracle (v ms)						
HTTP metoda	**Průměr**	**Medián**	**Min**	**Max**	**Směrodatná odchylka**	**Variační koeficient**
GET	967	902	623	1845	224	23,2 %
POST	51	50	48	103	6	11,8 %
PUT	52	50	47	107	6	11,5 %
DELETE	51	50	48	94	5	9,8 %

Tabulka 3 - Výsledky testování ASP.NET Core s databází Oracle [Zdroj: autor]

Výsledky naměřených hodnot průměrných dobových odezev a mediánů u metody GET ukazují, že technologie ASP.NET Core očekávaně vykazuje nejrychlejší odezvu

při spolupráci s databází SQL Server. Průměrná doba odezvy u této kombinace technologií činí 841 ms, což je nižší než u ostatních zkoumaných databázových systémů, stejně tak jako v případě mediánu, který má hodnotu 824 ms. Tato pozorovaná rychlost spojení mezi ASP.NET Core a SQL Serverem může být důsledkem výše zmíněné optimálnější integrace této technologie s databází SQL Server či efektivního zpracování dotazů mezi touto technologií a databází.

Ačkoliv jsou naměřené hodnoty odezvy dle očekávání nejnižší s databází SQL Server, výrazně ASP.NET Core nezaostává ani s databází PostgreSQL, která je oproti databázi SQL Server v průměru pomalejší o 37 ms a rozdíl mediánů je pak pouze 8 ms. Výrazněji pak zaostává technologie ASP.NET Core ve spolupráci s databází Oracle, kde byla naměřena průměrná hodnota o 126 ms pomalejší než při spolupráci s databází SQL Server a rozdíl v mediánech je pak 78 ms.

Všechny ostatní HTTP metody (POST, PUT, DELETE) vykazují poměrně vyrovnané doby odezev napříč testovanými databázovými systémy. Tento jev může být způsobem tím, že tyto metody nepracují s tak velkým objemem dat jako metoda GET a nemusí tak vyniknout případné rozdíly v rychlostech napříč jednotlivými databázovými systémy. Právě z tohoto důvodu pak metoda GET může být nejvíce vypovídající o rychlostech ASP.NET Core s jednotlivými databázemi.

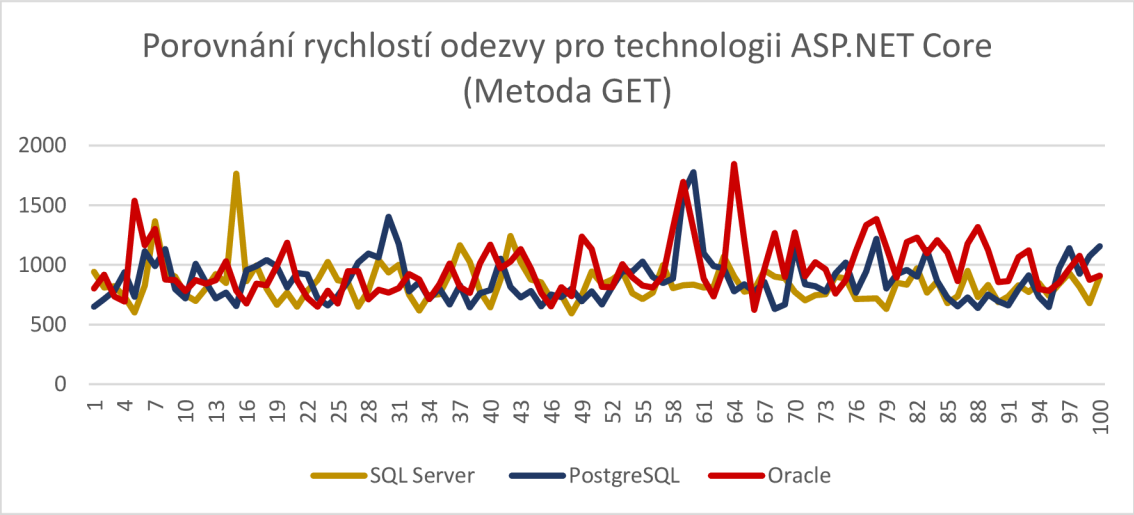
Hodnoty variačního koeficientu, které vyjadřují míru variability dat vzhledem k průměrným hodnotám mohou v kontextu měření rychlostí odezvy webových API nastínit informace o konzistenci jednotlivých volání, a tedy i o celkové stabilitě dané technologie. Nižší míra variačního koeficientu naznačuje, že většina požadavků není příliš rozptýlena od průměru, což znamená, že technologie je více předvídatelná z hlediska jednotlivých měření. Naopak vyšší hodnota variačního koeficientu může naznačovat, že časy odezvy jsou více rozptýlené vzhledem k průměrné hodnotě a mohou vést k nekonzistentnímu chování aplikace z hlediska jednotlivých volání.

Na základě naměřených hodnot variačních koeficientů pro ASP.NET Core v případě metody GET lze říci, že nejlépe tato technologie spolupracuje v kombinaci s databází SQL Server, kde je vykázána nejnižší míra variačního koeficientu. To stejně jako v případě rychlostí naznačuje, že technologie ASP.NET Core je vhodně

optimalizována právě pro práci s databází SQL Server. ASP.NET Core v kombinaci s databází PostgreSQL a Oracle pak vykazuje poměrně podobné míry variačních koeficientů, které jsou však vyšší, než tomu je v případě měření s databází SQL Server.

U většiny ostatních HTTP metod nebyly zaznamenány výrazné rozdíly v konzistenci a stabilitě jednotlivých volání stejně jako v případě měření rychlostí. Výraznější rozdíl v konzistenci jednotlivých měření nastal pouze v případě metody POST s databází PostgreSQL, a to i s ohledem na naměřenou maximální hodnotu.

Srovnání průměrných dob odezvy pro metodu GET jednotlivých volání technologie ASP.NET Core pro veškeré vybrané databáze zobrazuje následující graf, kde na ose X jsou jednotlivá volání a osa Y ukazuje hodnotu odezvy v milisekundách pro dané volání.



Obrázek 7 - Porovnání rychlostí odezvy pro ASP.NET Core [Zdroj: autor]

Z grafu lze vypozorovat zvýšenou dobu odezvy při spolupráci s databází Oracle především v porovnání s databází SQL Server. Uvedený graf však lépe znázorňuje poměrně velké výkyvy dob odezvy především u kombinace s databází Oracle, ale také s PostgreSQL, což potvrzuje vyšší hodnoty směrodatných odchylek právě pro kombinace s těmito dvěma databázemi. Ačkoliv u databáze SQL Server lze také

vypozorovat určité výkyvy, celkově se kombinace s touto databází jeví jako konzistentnější.

Ověření hypotéz

Ověření stanovených hypotéz pro ASP.NET Core je nejprve provedeno na základě Kruskal-Wallisova testu, který určí, zda existují statisticky významné rozdíly napříč všemi testovanými databázemi pro metodu GET (1. hypotéza). Pokud tato hypotéza bude přijata, budou proveden Dunnův post-hoc test, který již odhalí, mezi kterými skupinami existují statisticky významné rozdíly (2. a 3. hypotéza). Postup řešení stanovených hypotéz je následující:

1. Hypotéza

H_0: Neexistuje žádný statisticky významný rozdíl v časech odezvy technologie ASP.NET Core mezi všemi databázovými při použití metody GET.

H_1: Existuje statisticky významný rozdíl v časech odezvy technologie ASP.NET Core mezi alespoň jednou dvojicí databázových systémů při použití metody GET.

Vzhledem k tomu, že p-hodnota Kruskal-Wallisova testu je mnohem nižší než hladina významnosti 0,05 (p = 2.536E-05), zamítáme nulovou hypotézu H_0. **To znamená, že existují statisticky významné rozdíly mezi alespoň jednou dvojicí databázových systémů v časech odezvy technologie ASP.NET Core při použití metody GET**. Následně tak mohou být formulovány hypotézy 2 a 3, které jsou ověřeny pomocí Dunnova post-hoc testu.

2. Hypotéza

H_0: Neexistuje žádný statisticky významný rozdíl v časech odezvy technologie ASP.NET Core s databází SQL Server a Oracle při použití HTTP metody GET.

H_1: Existuje statisticky významný rozdíl v časech odezvy technologie ASP.NET Core s databází SQL Server a Oracle při použití HTTP metody GET.

Na základě Dunnova post-hoc testu je p-hodnota pro porovnání mezi SQL Server a Oracle velmi nízká (p = 9.18E-06). Tento výsledek je nižší než hladina významnosti 0,05, což nám umožňuje zamítnout nulovou hypotézu H_0 a akceptovat alternativní

hypotézu H1. To znamená, že existuje statisticky významný rozdíl v časech odezvy technologie ASP.NET Core mezi databázemi SQL Server a Oracle při použití HTTP metody GET. Při pohledu na nižší hodnoty průměrů a mediánů v kombinaci s těmito dvěma databázemi lze říci, že **v případě metody GET je ASP.NET Core rychlejší ve spolupráci s databází SQL Server než s databází Oracle**.

3. Hypotéza

H_0: Neexistuje žádný statisticky významný rozdíl v časech odezvy technologie ASP.NET Core s databází SQL Server a PostgreSQL při použití HTTP metody GET.

H_1: Existuje statisticky významný rozdíl v časech odezvy technologie ASP.NET Core s databází SQL Server a PostgreSQL při použití HTTP metody GET.

Vzhledem k vysoké p-hodnotě (0.2457) pro porovnání mezi SQL Server a PostgreSQL není dostatečný důkaz k zamítnutí nulové hypotézy. To znamená, že není statisticky významný rozdíl v časech odezvy technologie ASP.NET Core mezi těmito dvěma databázemi při použití HTTP metody GET. Ačkoliv je tedy průměrná hodnota i medián pro ASP.NET Core v kombinaci s databází SQL Server nižší než s databází PostgreSQL, **tento rozdíl není pro metodu GET významný**.

8.3 Spring

Další testovanou technologií pro vývoj RESTful API je technologie Spring Boot, která se vyvíjí v jazyce Java. Pro objektově relační mapování je v případě této technologie použit nástroj Java Persistence API (JPA), což je stejně jako v případě Entity Framework v ASP.NET Core nejčastější řešení pro mapování databázových tabulek do objektů v případě Spring Boot aplikací.

8.3.1 Představení kódu

Pro mapování tabulek do objektů jsou stejně jako v případě ASP.NET Core vytvořeny objekty. Následující ukázka představuje objekt Book se kterým je při testování zpravidla pracováno.

```
@Entity
@Table(name = "Books")
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int Id;

    private String Title;

    @ManyToOne
    @JoinColumn(name = "Author_id")
    private Author Author;

    @ManyToOne
    @JoinColumn(name = "Publisher_id")
    private Publisher Publisher;

    private int Publication_Year;

    private String Isbn;

    @ManyToMany(cascade = CascadeType.PERSIST,
                fetch =  FetchType.EAGER)
    @JoinTable(name = "Books_Genres",
    joinColumns = @JoinColumn(name = "Book_id",
referencedColumnName = "Id"),
    inverseJoinColumns = @JoinColumn(name = "Genre_id",
referencedColumnName = "Id"))
    private Set<Genre> Genres = new HashSet<>();
}
```

Ukázka 16 – Objekt Book pro mapování ve Spring [Zdroj: autor]

Anotace @Entity v kontextu JPA označuje třídu jako entitu, což znamená, že tato třída má přímý vztah k databázové tabulce. Když třída obsahuje tuto anotaci, JPA předpokládá, že instance této třídy může pracovat s databází. ID daných objektů je upřesněn anotací @Id a @GeneratedValue, která upřesňuje, že daná hodnota je automaticky generována databází.

Stejně jako v případě objektů v ASP.NET Core mají všechny objekty ve Spring vybrané atributy a také vazby na další objekty. V případě třídy Book tyto vazby zajišťují anotace @ManyToOne upřesňující typ vazby na 1*N a anotace @JoinColumn, která propojuje objekty na základě identifikátoru cizího klíče, který je uveden v anotaci. Stejně jako v případě ASP.NET Core objekty definují i vazby N*N pomocí anotace @ManyToOne a anotace @JoinTable, která propojuje tabulku Books

a Genres na základě asociativní tabulky Books_Genres pomocí cizích klíčů. Atribut cascade nastavený na PERSIST v tomto kontextu upřesňuje, že pokud je hlavní entita (například kniha) persistována (tj. vložena do databáze), budou provedeny operace persist i na všech spojených entitách (například autor nebo vydavatel), které jsou součástí této asociace. Atribut fetch je alternativou k výše zmíněnému lazy loadingu v ASP.NET Core. Nastavení tohoto atributu na hodnotu EAGER zajišťuje načítání spojených dat ihned při načítání hlavní entity, stejně jako v případě ASP.NET Core a metody Include.

Pro účely testování je stejně jako v případě ASP.NET Core vytvořen controller, ve kterém jsou implementovány čtyři základní HTTP metody určené k testování. Anotace @RequestMapping("/api/books") zajišťuje, že všechny akce v tomto controlleru budou dostupné pod URL cestou /api/books.

GET

První implementovaná metoda je metoda GetBooks, která zpracovává GET požadavky a má za cíl získat všechny knihy z databáze a vrátit je jako odpověď na požadavek. Metoda získává všechny záznamy z tabulky Books v databázi pomocí JPA Repository. Výsledná data jsou následně vrácena jako odpověď metody.

```
@GetMapping
public ResponseEntity<List<Book>> getBooks()
{
    List<Book> books = bookRepository.findAll();
    return ResponseEntity.ok(books);
}
```

Ukázka 17 - GET metoda ve Spring [Zdroj: autor]

POST

Další implementovaná metoda pro testování v rámci Java Spring je metoda createBook, která naslouchá na cestě api/books/create a přijímá HTTP POST požadavek. Tento požadavek očekává objekt typu Book v těle požadavku. Na základě

poskytnutých identifikátorů žánrů jsou následně ke knize přiřazeny jednotlivé objekty žánrů a kniha je uložena do databáze.

```
@PostMapping("/create")
public ResponseEntity<Book> createBook(@RequestBody Book book) {
  Set<Genre> genres = book.getGenres();
  Set<Genre> attachedGenres = new HashSet<>();

  for (Genre genre : genres) {
      Genre attachedGenre = genreRepository.findById(genre.getId())
      .orElseThrow(() -> new RuntimeException("Not found"));
      attachedGenres.add(attachedGenre);
  }

  book.setGenres(attachedGenres);

  Book savedBook = bookRepository.save(book);
  return ResponseEntity.status(HttpStatus.CREATED).body(savedBook);
}
```

Ukázka 18 - POST metoda ve Spring [Zdroj: autor]

PUT

Metoda updateBook v rámci Java Spring očekává HTTP PUT požadavek pro aktualizaci knihy s vybraným ID. Stejně jako v případě ASP.NET Core metoda očekává identifikátor knihy, která má být upravena společně s tělem objektu Book s jehož hodnotami má být vybraná kniha aktualizována.

```
@PutMapping("/{id}/update")
public ResponseEntity<Book> updateBook(@PathVariable Integer id,
@RequestBody Book updatedBook)
{
    Book existingBook = bookRepository.findById(id)
    .orElseThrow(() -> new EntityNotFoundException("Not found“));

    existingBook.setTitle(updatedBook.getTitle());
    existingBook.setAuthor(updatedBook.getAuthor());
    existingBook.setPublisher(updatedBook.getPublisher());
    existingBook.setIsbn(updatedBook.getIsbn());
    existingBook.setGenres(updatedBook.getGenres());
    existingBook.setYear(updatedBook.getYear());

    Book updated = bookRepository.save(existingBook);
    return ResponseEntity.ok(updated);
}
```

Ukázka 19 - PUT metoda ve Spring [Zdroj: autor]

DELETE

Poslední testovaná metoda je metoda DeleteBook je označená atributem sloužícím ke smazání knihy z databáze zpracovávající HTTP DELETE požadavky na URL /api/books/{id}/delete, kde {id} je identifikátor knihy, která má být smazána.

Na základě poskytnutého identifikátoru je vyhledána kniha vybraná ke smazání a jsou odstraněny všechny záznamy z asociativní tabulky spojené s danou knihou. Následně je smazána i samotná kniha a stejně jako v případě ASP.NET Core je vrácen kód 204 indikující, že kniha byla úspěšně smazána.

```
@DeleteMapping("/{id}/delete")
public ResponseEntity<Void> deleteBook(@PathVariable Integer id)
{
    Optional<Book> optionalBook = bookRepository.findById(id);
    if (!optionalBook.isPresent()) {
        return ResponseEntity.notFound().build();
    }

    optionalBook.get().getGenres().clear();

    bookRepository.deleteById(id);
    return ResponseEntity.noContent().build();
}
```

Ukázka 20 - DELETE metoda ve Spring [Zdroj: autor]

8.3.2 Měření rychlostí odezvy

Metody implementované pomocí technologie Spring jsou opět testovány se všemi vybranými databázovými systémy. Z hlediska rychlostí odezvy, ale i z hlediska konzistence jednotlivých volání se očekává, že technologie Spring bude nejlépe spolupracovat s databází Oracle, nebo s databází PostgreSQL, jelikož tato technologie se v mnoha případech implementuje právě s těmito databázovými systémy. Naopak tato technologie může z hlediska rychlostí zaostávat ve spolupráci s databází SQL Server, jelikož využívání této kombinace není příliš časté a technologie Spring tak nemusí být vhodně optimalizována pro spolupráci s databází SQL Server. Výsledky testování pro technologii Spring v milisekundách jsou následující.

Technologie Spring s databází SQL Server (v ms)						
HTTP metoda	**Průměr**	**Medián**	**Min**	**Max**	**Směrodatná odchylka**	**Variační koeficient**
GET	2648	2384	3150	143	143	5,4 %
POST	61	57	105	6	6	9,8 %
PUT	63	56	180	18	18	28,6 %
DELETE	64	56	282	23	23	37,7 %

Tabulka 4 - Výsledky testování Spring s databází SQL Server [Zdroj: autor]

Technologie Spring s databází PostgreSQL (v ms)						
HTTP metoda	**Průměr**	**Medián**	**Min**	**Max**	**Směrodatná odchylka**	**Variační koeficient**
GET	1177	1120	899	2829	237	20,1 %
POST	53	52	50	94	6	11,3 %
PUT	63	57	52	275	25	39,7 %
DELETE	56	54	51	115	9	16 %

Tabulka 5 - Výsledky testování Spring s databází PostgreSQL [Zdroj: autor]

Technologie Spring s databází Oracle (v ms)						
HTTP metoda	**Průměr**	**Medián**	**Min**	**Max**	**Směrodatná odchylka**	**Variační koeficient**
GET	1409	1254	761	2891	430	30,5 %
POST	57	55	52	106	7	12,3 %
PUT	61	55	52	178	20	32,8 %
DELETE	56	55	50	103	7	12,5 %

Tabulka 6 - Výsledky testování Spring s databází Oracle [Zdroj: autor]

Z naměřených hodnot průměrných dobových odezev u metody GET vyplývá, že technologie Spring má nejrychlejší průměrnou odezvu při spolupráci s databází PostgreSQL. Průměrná doba odezvy u této kombinace technologií činí 1177 ms a medián je 1120 ms. Nicméně v měření značně nezaostává ani kombinace s databází Oracle, kde je naměřena průměrná hodnota 1409 ms s mediánem 1254

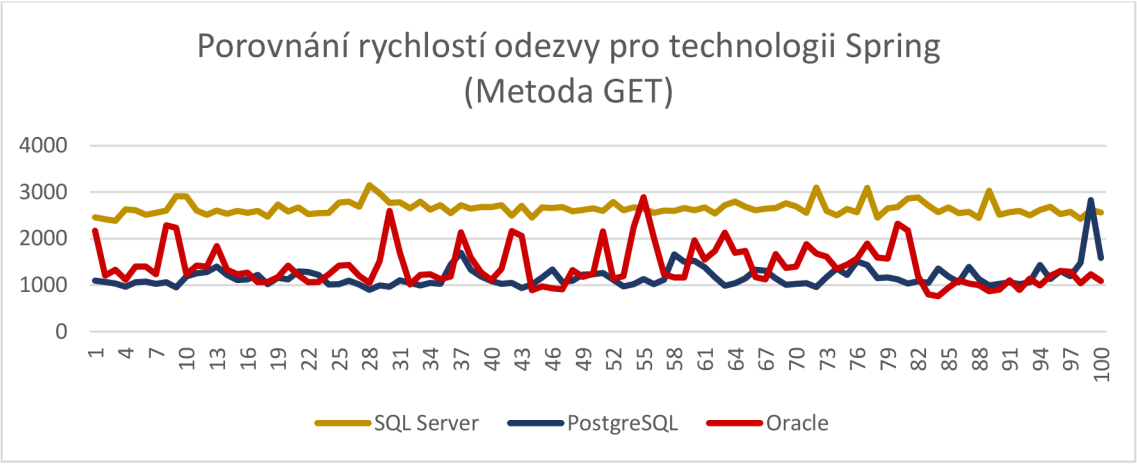
ms. Dá se tak říci, že z hlediska rychlostí technologie Spring nejlépe spolupracuje s databázemi, které se zároveň zpravidla pro tuto technologii používají v praxi oproti databázi SQL Server. Technologie Spring při testování GET metody poměrně zaostává v kombinaci s databází SQL Server, kde byl naměřen průměrný čas odezvy 2648 ms, což je zhruba o 125% pomalejší průměrný čas odezvy než v kombinaci s databází PostgreSQL a téměř o 88% pomalejší průměrný čas odezvy než v kombinaci s databází Oracle. Naměřené průměrné hodnoty i mediány tak mohou potvrzovat fakt, že technologie Spring je vhodně optimalizována ke spolupráci s databázemi PostgreSQL a Oracle, se kterými se zpravidla používá nejčastěji. Naopak není příliš vhodně optimalizována pro spolupráci s databází SQL Server, která se v praxi téměř nepoužívá.

Ostatní testované HTTP metody, stejně jako v případě technologie ASP.NET Core, vykazují poměrně vyrovnané doby odezev napříč testovanými databázovými systémy. Mírně větší hodnoty lze zaznamenat v případě kombinace s databází SQL Server, kde všechny testované metody přesahují hodnotu 60 ms, nicméně tento rozdíl je prakticky zanedbatelný.

Na základě hodnot variačních koeficientů u metody GET lze říci, že nejkonzistentněji Spring poměrně značně spolupracuje v kombinaci s databází SQL Server, kde je vykázána nejnižší hodnota směrodatné odchylky. Ačkoliv je tedy spolupráce s databází SQL Server jednoznačně nejpomalejší, hodnoty všech volání jsou blízko průměrným hodnotám. Naopak ke větším výkyvům dochází v případě zbylých dvou databází, a to především v případě databáze Oracle, která vykazuje značně nejvyšší hodnotu směrodatné odchylky.

V případě ostatních HTTP metod byly zaznamenány značné rozdíly v konzistenci a stabilitě jednotlivých volání především u metod PUT, a to v případě všech databází. Vyšší směrodatná odchylka je také zaznamenána u metody DELETE v kombinaci s databází SQL Server.

Porovnání průměrných dob odezvy všech jednotlivých volání technologie Spring pro všechny vybrané databáze je opět zobrazen na následujícím grafu.



Obrázek 8 - Porovnání rychlostí odezvy pro Spring [Zdroj: autor]

Z grafu lze vypozorovat výše postavenou křivku pro databázi SQL Server, což znázorňuje značně vyšší rychlosti odezvy v kombinaci s touto databází. Nicméně z křivky grafu také vyplývá vyšší konzistence jednotlivých volání, kdy jednotlivá hodnoty volání nejsou příliš odchýlena od průměru. U databází Oracle a PostgreSQL lze pak vidět nižší naměřené hodnoty, nicméně také značná nekonzistence v jednotlivých voláních, a to především v případě databáze Oracle.

Ověření hypotéz

Ověření stanovených hypotéz pro technologie Spring je stejně jako v případě technologie ASP.NET Core nejprve provedeno na základě Kruskal-Wallisova testu, který potvrdí či vyvrátí existenci statisticky významných rozdílů mezi všemi testovanými databázemi pro metodu GET (4. hypotéza). Pokud je 4. hypotéza přijata, je proveden Dunnův post-hoc test. (5. a 6. hypotéza). Postup řešení stanovených hypotéz je následující:

4. Hypotéza

H_0: Neexistuje žádný statisticky významný rozdíl v časech odezvy technologie Spring mezi všemi databázovými při použití metody GET.

H_1: Existuje statisticky významný rozdíl v časech odezvy technologie Spring mezi alespoň jednou dvojicí databázových systémů při použití metody GET.

Vzhledem k nízké p-hodnotě Kruskal-Wallisova testu (p = 4.019E-44) zamítáme nulovou hypotézu H_0. **To znamená, že existují statisticky významné rozdíly v časech odezvy technologie Spring mezi alespoň jednou dvojicí databázových systémů při použití metody GET.** Lze tedy formulovat další hypotézy testované pomocí Dunnova post-hoc testu, který odhalí jednotlivé statisticky významné rozdíly mezi konkrétními dvojicemi.

5. Hypotéza

H_0: Neexistuje žádný statisticky významný rozdíl v časech odezvy technologie Spring s databází Oracle a SQL Server při použití HTTP metody GET.

H_1: Existuje statisticky významný rozdíl v časech odezvy technologie Spring s databází Oracle a SQL Server při použití HTTP metody GET.

Dunnův post-hoc test pro porovnání mezi Oracle a SQL Server vykazuje velmi nízkou p-hodnotu (1.469E-25), což je značně nižší než stanovená hladina významnosti 0,05. Tedy zamítáme nulovou hypotézu H_0 a přijímáme alternativní hypotézu H_1. Existuje statisticky významný rozdíl v časech odezvy technologie Spring mezi databázemi Oracle a SQL Server při použití HTTP metody GET. Na základě získaných průměrných hodnot i mediánu lze pak jednoznačně říci, **že Spring je dle očekávání s databází Oracle rychlejší než ve spolupráci s databází SQL Server**.

6. Hypotéza

H_0: Neexistuje žádný statisticky významný rozdíl v časech odezvy technologie Spring s databází Oracle a PostgreSQL při použití HTTP metody GET.

H_1: Existuje statisticky významný rozdíl v časech odezvy technologie Spring s databází Oracle a PostgreSQL při použití HTTP metody GET.

Při porovnání mezi Oracle a PostgreSQL je p-hodnota (0.002523) Dunnova post-hoc testu nižší než stanovená hladina významnosti 0,05, což umožňuje zamítnout nulovou hypotézu H_0 a přijmout alternativní hypotézu H_1. Existuje statisticky významný rozdíl v časech odezvy technologie Spring mezi databázemi Oracle a PostgreSQL při použití HTTP metody GET. Na základě naměřených průměrných hodnot a mediánů v případě obou databází pak lze jednoznačně určit**, že technologie Spring s databází PostgreSQL je rychlejší než Spring v kombinaci s databází Oracle**. Ačkoliv bylo očekáváno, že

technologie Spring bude rychleji spolupracovat spíše s databází Oracle což je poměrně častá kombinace, v praxi je možné se často setkat právě i s kombinací technologie Spring a databáze PostgreSQL. Zároveň rozdíl v případě těchto dvou databází není na základě p-hodnot obou testů tak statisticky významný jako rozdíl právě mezi databází Oracle a SQL Server.

8.4 Django

Poslední testovanou technologií pro vývoj RESTful API je technologie Django, která se implementuje v jazyce Python. Pro mapování databázových tabulek se pro Django zpravidla používá systém ORM (Object-Relational Mapping), který bude používán i pro testování.

8.4.1 Představení kódu

Následující ukázka představuje třídu Book v souboru models.py, ve kterém jsou definovány objekty pro práci s databází. Pro všechny objekty jsou v souboru serializers.py také implementovány serializéry a deserializéry pro převádění objektů do formátu pro přenos přes HTTP.

```
class Book(models.Model):
  title = models.CharField(max_length=100)
  author = models.ForeignKey('Author', on_delete=models.CASCADE)
  publisher=models.ForeignKey('Publisher',on_delete=models.CASCADE)
  publication_year = models.IntegerField()
  isbn = models.CharField(max_length=20)
  genres = models.ManyToManyField('Genre', through='BookGenre')
```

Ukázka 21 – Objekt Book pro mapování v Django [Zdroj: autor]

Stejně jako v případě předchozích dvou technologií jsou v objektu Books definovány vazby na ostatní objekty pomocí cizích klíčů. Zároveň je nastavena i vazba N*N na dané žánry pomocí objektu BooksGenres, který mapuje asociativní tabulku Books_Genres. Argument on_delete u objektu BooksGenres nastavený na CASCADE zajišťuje, že pokud je smazán záznam dané knihy, na kterou odkazuje záznam i v asociativní tabulce Books_Genres, je tento záznam automaticky smazán také.

Pro účely testování HTTP požadavků ve frameworku Django je vytvořen soubor views.py, ve kterém jsou definovány veškeré metody pro práci s HTTP požadavky.

Jedná se tak o obdobu controllerů používaných v předchozích technologiích ASP.NET Core a Spring Boot.

GET

Metoda get_books stejně jako ve dvou předchozích případech vrací všechny knihy v databázi. Dekorátor @api_view(['GET']) zajišťuje, že metoda bude obsluhovat požadavky GET. Serializer následně zajistí, že všechny knihy v databázi budou vráceny ve formátu JSON.

```
@api_view(['GET'])
def get_books(request):
    books = Book.objects.all()
    serializer = BookSerializer(books, many=True)
    return Response(serializer.data)
```

Ukázka 22 - GET metoda v Django [Zdroj: autor]

POST

Metoda create_book obsluhuje požadavky POST, konkrétně požadavek na přidání knihy do databáze. Po oveření, zda je serializovaný objekt validní, následuje získání ID přidružených autorů, vydavatelů a žánrů. Následně jsou z databáze získány objekty podle daných identifikátorů a jsou přiřazeny k vytvořené knize při ukládání. Vrácen je vytvořený objekt a kód 201 indikující, že objekt byl v pořádku přidán.

```
@api_view(['POST'])
def create_book(request):
    serializer = BookCreateSerializer(data=request.data)
    if serializer.is_valid():
        author_id = request.data.get('author')
        publisher_id = request.data.get('publisher')
        genre_ids = request.data.get('genres')

        try:
            author = Author.objects.get(pk=author_id)
            publisher = Publisher.objects.get(pk=publisher_id)
            genres = Genre.objects.filter(pk__in=genre_ids)
        except (Author.DoesNotExist, Publisher.DoesNotExist) as e:
            return JsonResponse({'error': str(e)}, status=400)

        book = serializer.save(author=author, publisher=publisher)
        book.genres.set(genres)

        return JsonResponse(serializer.data, status=201)
    return JsonResponse(serializer.errors, status=400)
```

Ukázka 23 - POST metoda v Django [Zdroj: autor]

PUT

Aktualizace vybrané knihy opět obsluhuje metoda update_book. V případě, že kniha s vybraným identifikátorem v databázi existuje a serializovaná data jsou validní, kniha je v databázi aktualizována a je vrácen aktualizovaný objekt.

```
@api_view(['PUT'])
def update_book(request, id):
    try:
        book = Book.objects.get(id=id)
        serializer = BookUpdateSerializer(instance=book,
data=request.data)
        if serializer.is_valid():
            serializer.save()
            return JsonResponse(serializer.data)
        return JsonResponse(serializer.errors, status=400)
    except Book.DoesNotExist:
        return JsonResponse({'error': 'Book not found'},
status=404)
```

Ukázka 24 - PUT metoda v Django [Zdroj: autor]

DELETE

Metoda pro mazání knih delete_book se od metod v předchozích dvou technologiích liší tím, že tato metoda již explicitně nemusí implementovat mazání referencí na danou knihu v tabulce Books_Genres, ale smazání je automaticky zajištěno nastavením parametru on_delete na hodnotu CASCADE, který byl zmíněn výše.

```
@api_view(['DELETE'])
def delete_book(request, id):
    try:
        book = Book.objects.get(id=id)
    except Book.DoesNotExist:
        return Response(status=404)

    book.delete()

    return Response(status=204)
```

Ukázka 25 - DELETE metoda v Django [Zdroj: autor]

URL cesty k jednotlivým metodám jsou nastaveny stejně jako v případě předchozích dvou technologií a jsou definovány v soubory urls.py.

8.4.2 Měření rychlostí odezvy

Z testování HTTP metod se očekává, že technologie Django bude nejrychleji spolupracovat s databází PostgreSQL, jelikož se tato databáze se často používá ve spojení s technologií Django. Naopak tato technologie může z hlediska rychlostí zaostávat především ve spolupráci s databází SQL Server. Výsledky testování pro technologii ASP.NET Core jsou následující.

Technologie Django s databází SQL Server (v ms)						
HTTP metoda	**Průměr**	**Medián**	**Min**	**Max**	**Směrodatná odchylka**	**Variační koeficient**
GET	4760	4725	4435	5824	195	4 %
POST	156	155	150	198	6	3,8 %
PUT	162	155	151	293	19	11,7 %
DELETE	169	155	142	454	51	30,1 %

Tabulka 7 - Výsledky testování Django s databází SQL Server [Zdroj: autor]

Technologie Django s databází PostgreSQL (v ms)						
HTTP metoda	**Průměr**	**Medián**	**Min**	**Max**	**Směrodatná odchylka**	**Variační koeficient**
GET	4083	3966	3657	5755	351	8,5 %
POST	204	202	198	296	12	5,8 %
PUT	232	230	215	552	35	15 %
DELETE	232	218	200	458	41	17,7 %

Tabulka 8 - Výsledky testování Django s databází PostgreSQL [Zdroj: autor]

Technologie Django s databází Oracle (v ms)						
HTTP metoda	**Průměr**	**Medián**	**Min**	**Max**	**Směrodatná odchylka**	**Variační koeficient**
GET	4195	4122	3821	5271	252	6 %
POST	206	202	189	295	14	6,8 %
PUT	210	200	183	469	45	21,4 %
DELETE	196	186	177	340	26	13,3 %

Tabulka 9 - Výsledky testování Django s databází Oracle [Zdroj: autor]

Na základě naměřených hodnot průměrných dob odezvy a mediánů u metody GET je patrné, že technologie Django dle očekávání vykazuje nejrychlejší odezvu při spolupráci s databází PostgreSQL s naměřenou průměrnou hodnotou 4083 ms a mediánem 3966 ms. Nicméně Django nezaostává ani ve spolupráci s databází Oracle s průměrnou hodnotou 4195 ms a mediánem 4122 ms. Delší průměrnou dobu odezvy oproti databázím PostgreSQL a Oracle pak Django vykazuje v kombinaci s databází SQL Server s průměrnou hodnotou odezvy 4760 ms a mediánem 4725 ms.

Ostatní testované HTTP metody, na rozdíl od předchozích dvou metod, vykazují rozdíly v rychlostech i mezi jednotlivými databázovými systémy. Poměrně překvapivě technologie Django vykazuje nižší rychlosti odezvy v kombinaci s databází SQL Server, a to i přesto, že v případě metody GET tato kombinace byla nejpomalejší. Naopak Django v kombinaci s PostgreSQL zaostává při metodách PUT a DELETE, ačkoliv při práci s metodou GET byla databáze PostgreSQL nejrychlejší.

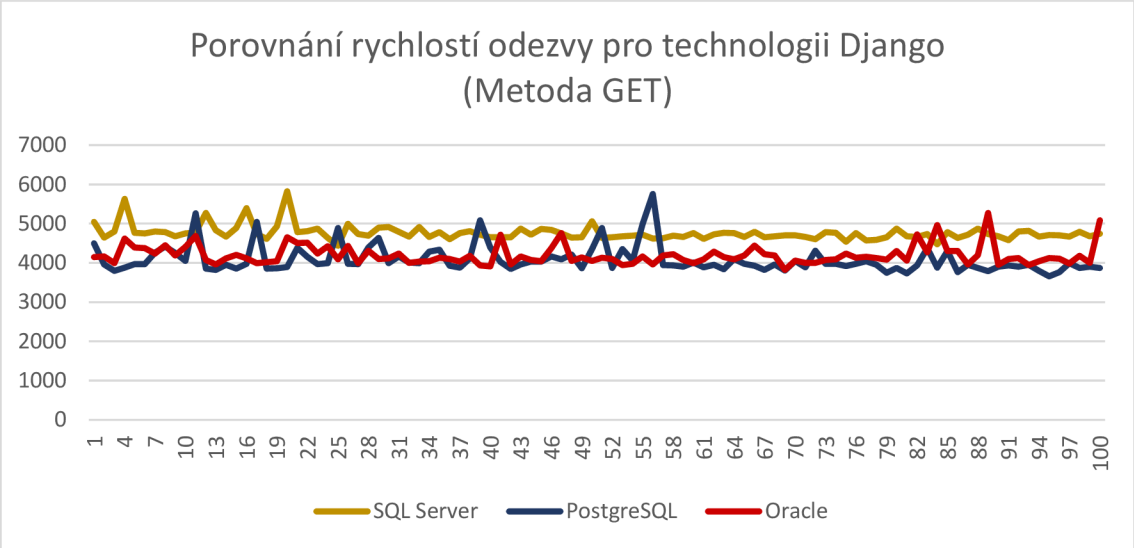
Stejně jako u předchozích dvou technologií je Django v případě metody GET nejstabilnější ve spolupráci s databází SQL Server, kde je zaznamenána nejnižší míra variačního koeficientu. Django však není nejméně stabilní v kombinaci s databází Oracle jak tomu bylo v případě předchozích dvou technologií, nicméně nejméně stabilní je s databází PostgreSQL kde byla zaznamenána nejvyšší míra variačního koeficientu.

U ostatních HTTP metod byly zaznamenány poměrně velké výkyvy na základě směrodatné odchylky ale i maximálních naměřených hodnot u metody DELETE

v kombinaci s databází SQL Server. Dále pak u metody PUT a DELETE v kombinaci s databází PostgreSQL a také v případě metody PUT ve spolupráci s databází Oracle.

Na základě dat z testování si lze všimnout, že technologie Django z hlediska rychlostí odezvy zaostává za předchozími testovanými technologiemi, a to jak v případě metod GET, tak i ostatních HTTP metod, které zpravidla byly u ASP.NET Core a Spring časově vyrovnané. Podrobnější porovnání technologií vzájemně bude provedeno v následující části práce.

Na následujícím grafu je zobrazeno srovnání průměrných časů odezvy pro metodu GET pro všechny testované databáze pro technologii Django.



Obrázek 9 - Porovnání rychlostí odezvy pro Django [Zdroj: autor]

Z grafu lze vypozorovat celkové vyšší časy odezvy pro technologii Django v kombinaci s databází SQL Server než v případě konfigurace s databází PostgreSQL a Oracle. Z grafu je také patrný nízké odchýlení hodnot pro databázi SQL Server, a naopak také nekonzistence především v případě databáze PostgreSQL.

Ověření hypotéz

Pro ověření hypotéz stanovených pro technologii Django je stejně jako v případě předchozích dvou technologií použit nejprve Kruskal-Wallisův test (7. Hypotéza). Následně pokud je přijata alternativní hypotéza, mohou být formulovány hypotézy

pro rozdíly v případě jednotlivých dvojic databázových systémů (8. a 9. hypotéza). Tyto hypotézy jsou opět vyhodnoceny na základě Dunnových post-hoc testů.

7. Hypotéza

H_0: Neexistuje žádný statisticky významný rozdíl v časech odezvy technologie Django mezi všemi databázovými při použití metody GET.

H_1: Existuje statisticky významný rozdíl v časech odezvy technologie Django mezi alespoň jednou dvojicí databázových systémů při použití metody GET.

Vzhledem k nízké p-hodnotě Kruskal-Wallisova testu (p = 5.567E-37) zamítáme nulovou hypotézu H_0. **To znamená, že existují statisticky významné rozdíly v časech odezvy technologie Django mezi alespoň jednou dvojicí databázových systémů při použití metody GET.** A je možné tak stanovit další hypotézy testované pomocí Dunnova testu.

8. Hypotéza

H_0: Neexistuje žádný statisticky významný rozdíl v časech odezvy technologie Django s databází PostgreSQL a SQL Server při použití HTTP metody GET.

H_1: Existuje statisticky významný rozdíl v časech odezvy technologie Django s databází PostgreSQL a SQL Server při použití HTTP metody GET.

Dunnův post-hoc test pro porovnání mezi PostgreSQL a SQL Server vykazuje velmi nízkou p-hodnotu (2.207E-19), která je značně nižší než stanovená hladina významnosti. Je možné tedy zamítnout nulovou hypotézu H_0 a přijmout alternativní hypotézu H_1. Existuje statisticky významný rozdíl v časech odezvy technologie Django mezi databázemi PostgreSQL a SQL Server při použití HTTP metody GET. Na základě získaných průměrných hodnot a mediánů lze říci, **že Django je rychlejší s databází PostgreSQL než s databází SQL Server.**

9. Hypotéza

H_0: Neexistuje žádný statisticky významný rozdíl v časech odezvy technologie Django s databází PostgreSQL a Oracle při použití HTTP metody GET.

H_1: Existuje statisticky významný rozdíl v časech odezvy technologie Django s databází PostgreSQL a Oracle při použití HTTP metody GET.

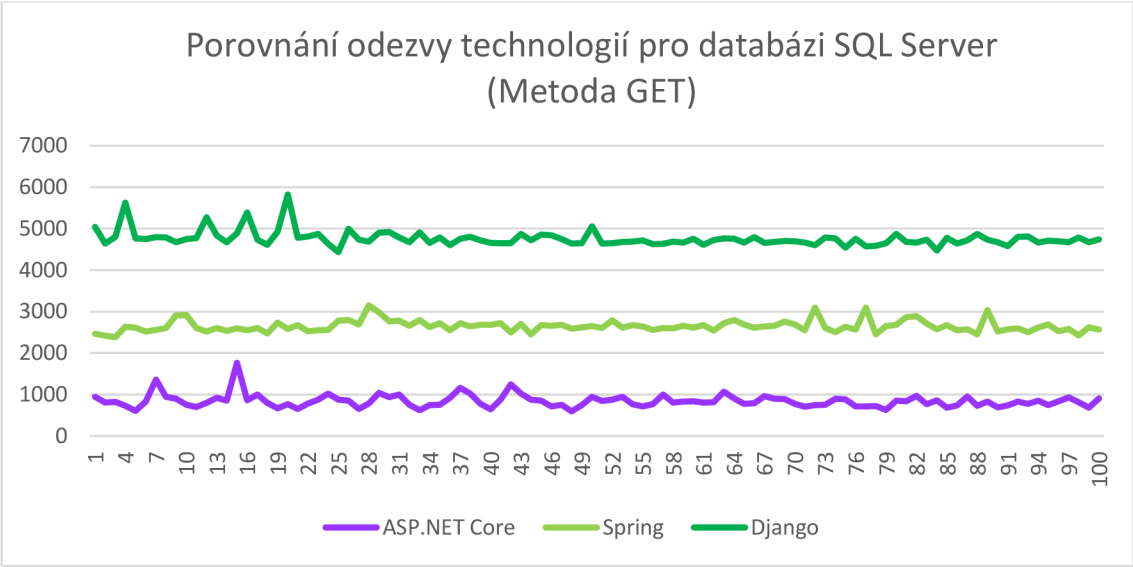
Při porovnání mezi PostgreSQL a Oracle je p-hodnota (0.000422) nižší než stanovená hladina významnosti 0,05, takže stejně jako v případě předešlých dvojic je možné zamítnout nulovou hypotézu H_0 a přijmout alternativní hypotézu H_1. Existuje statisticky významný rozdíl v časech odezvy technologie Django mezi databázemi PostgreSQL a Oracle při použití HTTP metody GET. Získané průměrné hodnoty a mediány poté naznačují, **že Django ve spolupráci s databází PostgreSQL je rychlejší než s databází Oracle.**

8.5 Porovnání technologií

V následující části práce jsou porovnávány vybrané technologie vzájemně mezi sebou při práci s jednotlivými databázovými systémy na základě dat uvedených v předchozích kapitolách pro jednotlivé technologie.

SQL Server

Na základě získaných dat všech technologií v kombinaci s databází SQL Server lze určit, že s touto databází dle očekávání jednoznačně nejrychleji pracuje technologie ASP.NET Core s hodnotou 841 ms, což je o více než trojnásobně rychlejší spolupráce než v případě technologie Spring, kde je zaznamenána hodnota 2648 ms. Mnohem pomalejší odezvy však dosáhla ještě technologie Django, která v kombinaci s databází SQL Server dosáhla průměrné odezvy 4760 ms a značně tak zaostává v porovnání s technologií Spring a především ASP.NET Core. Z hlediska stability jednotlivých volání si vzhledem ke své průměrné hodnotě nejlépe vede technologie Django s nejnižší mírou variačního koeficientu (4 %), nicméně za zmínku stojí i technologie Spring, která rovněž vzhledem k naměřené průměrné hodnotě vykazuje poměrně nízkou hodnotu (5,4 %). Překvapivě pak s databází SQL nejhůře ze všech technologií spolupracuje technologie ASP.NET Core, kde je variační koeficient nejvyšší (19 %). Na následujícím grafu lze vidět grafické srovnání všech technologií s databází SQL Server právě pro metodu GET.



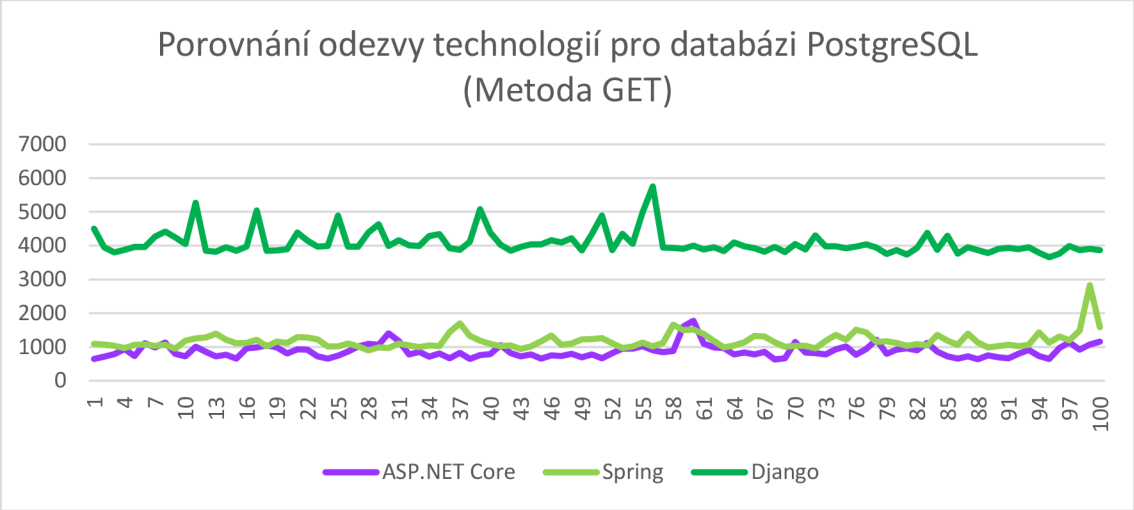
Obrázek 10 - Porovnání technologií s databází SQL Server [Zdroj: autor]

Na grafu lze vidět rozdíly v rychlostech mezi jednotlivými technologiemi kdy technologie ASP.NET Core je v kombinaci s databází SQL Server očekávaně jednoznačně nejrychlejší. Naopak z hlediska průměrných časů odezvy je značně pomalejší technologie Django, která ztrácí na technologie ASP.NET Core, ale i na technologii Spring.

V případě ostatních testovaných HTTP metod nejsou na základě průměrných hodnot mezi technologiemi ASP.NET Core a Spring značně rozdíly, kdy se v případě těchto dvou technologií všechny metody pohybují kolem hodnot 50 až 60 milisekund. Nicméně poměrně značně oproti těmto dvěma technologiím zaostává technologie Django, kde se všechny průměrné hodnoty pohybují kolem 160 milisekund. Z hlediska konzistence volání v případě metody POST si vzhledem ke svým hodnotám vede nejlépe technologie Django s mírou 3,8 %. V případě metody PUT si z hlediska konzistence téměř stejně vede technologie ASP.NET Core (11,5 %) a Django (11,7 %). Naopak poměrně značně u této metody zaostává technologie Spring (28,6 %). Z hlediska metody DELETE si pak nejlépe vede metoda ASP.NET Core (9,8 %) a zásadně zaostávají technologie Spring (37,7 %) a Django (30,1 %).

PostgreSQL

V případě kombinace jednotlivých technologií s databází PostgreSQL si stejně jako v případě databáze SQL Server vede v případě metody GET nejlépe technologie ASP.NET Core, která dosáhla s rychlostí 878 ms pouze mírně pomalejšího průměrného času odezvy než s databází SQL Server. Dle očekávání si značně lépe, než v případě databáze SQL Server vede technologie Spring, která má s časem 1177 ms rychlejší čas odezvy téměř o 1,5 sekundy než právě s databází SQL Server. Zrychlení průměrné odezvy v porovnání s předchozí databází vykazuje i technologie Django, s rychlostí odezvy 4083 ms. Nicméně tato technologie i přesto značně zaostává za technologiemi ASP.NET Core a Spring. Technologie Django si však v případě metody GET vede nejlépe z hlediska konzistence vzhledem k nejnižší naměřené míře variačního koeficientu s hodnotou 8,5 %. Technologie ASP.NET Core (22,4 %) a Spring (20,1 %) si z hlediska konzistence vzhledem k průměrné hodnotě vedou poměrně podobně. Následující graf nabízí srovnání všech technologií s databází PostgreSQL pro metodu GET.



Obrázek 11 - Porovnání technologií s databází PostgreSQL [Zdroj: autor]

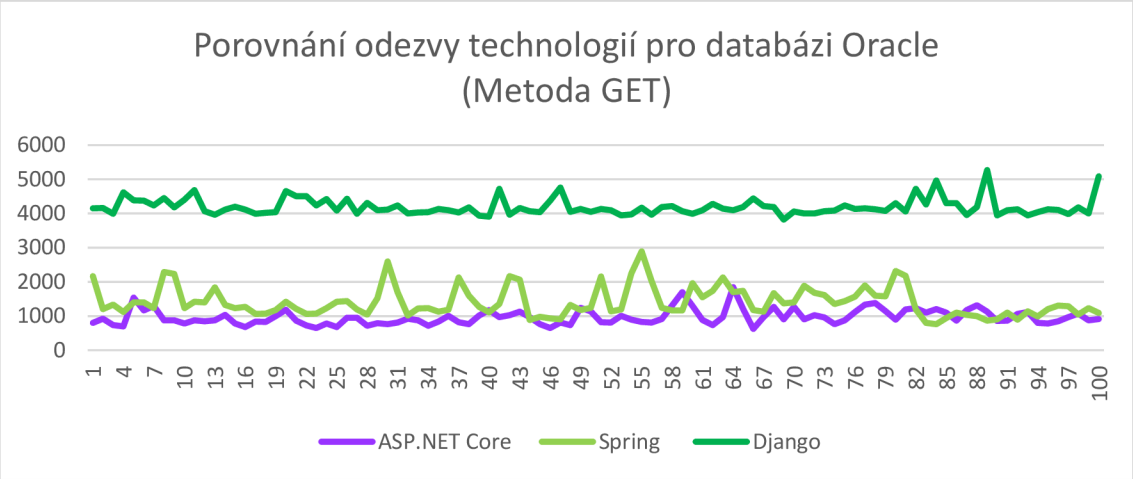
Grafu lze vyčíst, že technologie Spring je časově vyrovnanější s ASP.NET Core, než tomu bylo v případě kombinace s databází SQL Server. A ačkoliv technologie Django dosáhla nižšího času odezvy než v porovnání s předchozí databází, stále velmi značně ztrácí na technologii ASP.NET Core a technologii Spring.

U ostatních HTTP nejsou na základě testování zaznamenány značné rozdíly mezi technologiemi ASP.NET Core a Spring, stejně jako tomu bylo i v případě s databází SQL Server. Stejně jako v případě SQL Serveru se průměry rychlostí odezvy pohybují okolo 50-60 milisekund. Opět však značně zaostává technologie Django, kde se průměrné hodnoty všech metod pohubují okolo 200-230 milisekund, což je více než čtyřnásobek, než tomu je v případě technologií ASP.NET Core a Spring. Analýza stability na základě variačního koeficientu ukázala, že v případě metody POST si i vzhledem k vysokým hodnotám odezvy vede nejlépe technologie Django (5,8 %). V případě metody PUT pak byla nejnižší míra variačního koeficientu zaznamenána u technologie ASP.NET Core (12,2 %), nicméně značně opět nezaostává ani technologie Django (15 %). Naopak u této metody značně zaostává technologie Spring s mírou variačního koeficientu 39,7 %. V případě metody DELETE pak byly zaznamenány poměrně blízké hodnoty variačního koeficientu u technologií Spring (16 %) a Django (17,7 %), nicméně nejlepší stabilitu vykazuje technologie ASP.NET Core s hodnotou 9,6 %.

Oracle

Na základě výsledků testování technologií v kombinaci s databází Oracle pro metodu GET lze stejně jako v případě předchozích dvou databázových systémů určit, že nejrychleji ve spolupráci s databází Oracle komunikuje technologie ASP.NET Core s časem odezvy 967 ms, a to i navzdory tomu, že tato technologie v kombinaci s databází Oracle není příliš častá. Za technologií ASP.NET Core v kombinaci s databází Oracle značně nezaostává ani technologie Spring s průměrným časem odezvy 1409 ms. Stejně jako v případě testování s předchozími databázovými systémy však poměrně výrazně zaostává technologie Django s časem 4195 ms. Na rozdíl od rychlosti si však technologie Django v případě metody GET poměrně značně vede nejlépe z hlediska stability vzhledem k průměrným hodnotám s mírou variačního koeficientu 6 %, přičemž naopak poměrně zaostává technologie

ASP.NET Core a Spring. Následující graf zobrazuje srovnání jednotlivých technologií v kombinaci s poslední testovanou databází Oracle.



Obrázek 12 - Porovnání technologií s databází Oracle [Zdroj: autor]

Z grafu lze vyčíst nízké naměřené hodnoty pro technologii ASP.NET Core stejně jako tomu bylo v případě kombinace s předchozími databázemi. Lehce vyšších časů pak dosahuje technologie Spring a opět značně zaostává technologie Django.

V případě ostatních HTTP metod jsou opět naměřeny zanedbatelné rozdíly v rychlostech mezi technologií ASP.NET Core a Spring, kde se průměrné hodnoty všech metod u obou technologií pohybují kolem okolo 50-60 milisekund. Opět však v případě těchto metod značně zaostává technologie Django, kde se průměrné hodnoty všech metod pohybují okolo 200 milisekund. Z hlediska konzistence si v případě metody POST si u této databáze vede nejlépe technologie Django s mírou variačního koeficientu 6,8 %. Technologie ASP.NET Core (11,8 %) a Spring (12,3 %) v případě této metody z hlediska stability s databází Oracle pracují na podobně úrovni. V případě metody PUT si nejlépe vede technologie ASP.NET Core s variačním koeficientem 11,5 %, zatímto technologie Spring (32,8 %) i Django (21,4 %) poměrně zaostávají. U metody DELETE si pak všechny technologie vzhledem ke svým průměrným hodnotám vedou poměrně podobně, nicméně nejvyšší smíru konzistence vykazuje ASP.NET Core (ASP.NET Core 9,8 %, Spring 12,5 % a Django 13,3 %).

9 Shrnutí

Na základě provedeného výzkumu, kde byla zkoumána rychlost, výkonnost a stabilita implementací RESTful API pomocí různých technologií při práci se stejnými databázovými systémy získané výsledky naznačují, že technologie ASP.NET Core z hlediska rychlosti dosahuje nejlepších výsledků, a to poměrně překvapivě se všemi vybranými databázovými systémy. Technologie ASP.NET Core byla schopna v případě metody GET se všemi databázovými systémy dosáhnout rychlostí pod 1000 ms, čehož žádná další technologie nedosáhla. V případě ostatních HTTP metod se tyto hodnoty zpravidla pohybují kolem 50 až 60 milisekund, což je srovnatelné s technologií Spring a značně rychlejší než technologie Django. Z hlediska stability jednotlivých volání si však v případě metody GET technologie však ASP.NET Core mírně zaostává, jelikož se všechny míry variačního koeficientu pohybují kolem 20 %, a to i v kombinaci s databází SQL Server, kde byla očekávána nejlepší úroveň optimalizace. U ostatních HTTP si pak technologie ASP.NET Core vede poměrně obstojně a vykazuje výsledky na podobné úrovni jako ostatní technologie.

Obstojně si však vede i technologie Spring, která sice nedosahovala tak nízkých časů odezvy jako technologie ASP.NET Core, ale stále poskytovala solidní výkonnost, a to především s databázemi PostgreSQL a Oracle. Spring pak dle očekávání zaostává ve spolupráci s databází SQL Server. V případě ostatních testovaných HTTP metod si technologie Spring vede podobně jako technologie ASP.NET Core, jak již bylo zmíněno výše. Z hlediska stability tato technologie poměrně překvapivě dosáhla velmi dobrého výsledku v kombinaci s databází SQL Server, a naopak zaostávala v kombinaci s ostatními databázemi, kde byla dokonce u databáze Oracle zaznamenána hodnota variačního koeficientu přes 30 %. Výrazné odchylky byly v případě této metody zaznamenány i v případě ostatních metod, kde byla nejednou zaznamenána hodnota i přes 30 %.

Technologie Django při testování vykázala jednoznačně nejdelší časy napříč všemi databázovými systémy, kdy všechny průměrné hodnoty přesáhly hranici 4000 milisekund což je více než čtyřnásobná hodnota která byla naměřena v případě

ASP.NET Core. Technologie Django však oproti ASP.NET Core a Spring také značně zaostávala i v případě ostatních metod, kde byly v případě databáze SQL Server zaznamenány hodnoty pohybující se kolem 160 milisekund a v případě databáze Oracle a PostgreSQL dokonce i přes 200 milisekund. Přednost této technologie je však na základě měření konzistence volání pomocí této technologie, kdy v případě metody GET všechny hodnoty variačního koeficientu ani s jednou databází nepřesahují 10 %. U ostatních HTTP metod pak již byly naměřeny určité výchylky, nicméně například ve srovnání s technologií Spring tyto odchylky nejsou tak značné.

Na základě výše zmíněných poznatků pak lze najít odpověď na výše uvedenou výzkumnou otázku.

1. Jak se liší rychlost, výkonnost a stabilita implementací RESTful API pomocí různých technologií při práci se stejnými databázovými systémy?

Při použití metody GET s databází SQL Server si očekávaně vede nejlépe technologie ASP.NET Core a poměrně značně pak zaostávají zbylé dvě technologie Spring a Django. Při pohledu na výsledky měření ostatních HTTP metod lze také vidět, že ASP.NET Core rovněž vykazuje nejrychlejší časy odezvy, nicméně technologie Spring se těmto časům velmi blíží a tento rozdíl je prakticky zanedbatelný. Z hlediska stability technologií s touto databází pak lze vidět, že při použití metody GET poměrně překvapivě nejlépe funguje technologie Spring a Django a to i vzhledem ke značně vyšším rychlostem odezvy. V případě ostatních HTTP metod pak ASP.NET Core poměrně stabilně funguje se všemi metodami, zatímco technologie Spring i Django v některých případech zaznamenaly značné odchylky.

Metoda GET všech technologií při spolupráci s databází PostgreSQL vykazuje poměrně překvapivě nejlepší výsledky s technologií ASP.NET Core, ačkoliv se očekávala lepší spolupráce spíše se zbylými dvěma technologiemi. Stejně jako v případě databáze SQL Server vykazují technologie ASP.NET Core a Spring poměrně podobné rychlosti odezvy, zatímco Django velmi značně zaostává. Z hlediska stability volání s databází PostgreSQL si u metody GET jednoznačně vede nejlépe technologie Django a naopak značně zaostávají technologie ASP.NET Core a Spring. Určité odchylky pak ve spolupráci s touto databází obsahují všechny technologie.

U databáze Oracle, stejně jako v případě předchozích dvou databází u metody GET vykazuje nejrychlejší čas odezvy technologie ASP.NET Core a to opět velmi překvapivě. Stejně jako v případě předchozích dvou databází si i v případě databáze Oracle ostatní HTTP metody vedou poměrně podobně u technologií ASP.NET Core a Spring a opět značně zaostává technologie Django. Nejvyšší úroveň stability u metody GET stejně jako u předchozích dvou databází vykazuje technologie Django a značně pak zaostávají zbylé dvě technologie. ASP.NET Core pak vykazuje poměrně stabilní doby odezvy v případě ostatních HTTP metod, přičemž Spring i Django u některých metod vykazují odchylky.

Na základě analýzy naměřených hodnot pak lze najít odpověď i na další výzkumnou otázku.

2. Jaké jsou nejefektivnější kombinace technologií pro vývoj RESTful API ve spojení s konkrétními databázovými systémy z hlediska dosažení optimální rychlosti, výkonnosti a stability?

Na základě výsledků měření lze říci, že ze všech technologií dosahuje ASP.NET Core nejlepších výsledků napříč všemi databázemi, nicméně dle očekávání dosahuje nejlepších výsledků s databází SQL Server. Vzhledem k naměřeným hodnotám tak lze tuto kombinaci považovat za jednu z nejefektivnějších z hlediska stability ale především rychlosti odezvy. V případě kombinace s ostatními databázovými systémy byly u této technologie rovněž naměřeny poměrně nízké hodnoty, nicméně, kombinace ASP.NET Core s SQL Serverem dává větší smysl než s databázemi jako PostgreSQL nebo Oracle z důvodů optimalizace, integrace a výkonnostních výhod spojených s ekosystémem Microsoftu a kompatibilitou těchto technologií.

Z výsledků je možné vyvodit další poměrně vhodné kombinace pro spolupráci, především v případě technologie Spring, která zaznamenala poměrně dobré výsledky, a to především v kombinaci s databází PostgreSQL a Oracle. Na základě naměřených hodnot nelze tyto kombinace považovat za tak efektivní jako kombinaci technologie ASP.NET Core a SQL Server, nicméně i přesto byly vykázány poměrně rychlé doby odezvy. Ačkoliv u těchto kombinací byly v případě některých metod naměřeny poměrně vysoké odchylky, z hlediska rychlostí se tyto kombinace jeví jako velmi efektivní.

Jak již bylo výše zmíněno, technologie Django vykazuje značně vyšší doby odezvy napříč všemi databázovými systémy. Ačkoliv tato technologie vykazovala poměrně dobré nízké

míry variačního koeficientu určující stabilitu jednotlivých volání, z hlediska značně pomalých rychlostí nebude doporučena žádná vhodná kombinace obsahující technologii Django. Na základě měření tak byly vybrány následující kombinace technologií a databázových systémů.

- ASP.NET Core – SQL Server
- Spring – PostgreSQL
- Spring – Oracle

Na základě stanovených hypotéz, které byly otestovány pomocí výše zmíněných testů, lze odpovědět i na poslední formulovanou výzkumnou otázku.

3. Budou jednotlivé technologie pro vývoj webových API při použití metody GET nejoptimálněji pracovat s databázemi, se kterými se v praxi nejčastěji používají, oproti ostatním vybraným technologiím? Zkoumané kombinace jsou následující:
    - ASP.NET Core – SQL Server
    - Spring – Oracle
    - Django – PostgreSQL

V případě první stanovené kombinace byly určeny hypotézy porovnávající statisticky významné rozdíly mezi kombinací technologie ASP.NET Core a SQL Server a poté kombinací technologie se zbylými dvěma databázemi. V případě porovnání databáze SQL Server a Oracle byl zjištěn statisticky významný rozdíl mezi touto dvojicí. Vzhledem k naměřeným průměrným hodnotám a mediánům lze říci, že ASP.NET Core s databází SQL Server vykazuje rychlejší čas odezvy než v kombinaci s databází Oracle. V případě porovnání databáze SQL Server s databází PostgreSQL bylo zjištěno, že mezi těmito kombinacemi neexistuje statisticky významný rozdíl. Ačkoliv je tedy na základě průměrných hodnot a mediánů ASP.NET Core s SQL Server rychlejší, tento rozdíl není statisticky významný. V případě ASP.NET Core tedy nelze jednoznačně určit, že tato technologie nejoptimálnější funguje s databází SQL Server z hlediska rychlosti.

Pro druhou stanovenou technologii byly stanoveny hypotézy zkoumající statisticky významné rozdíly mezi kombinací technologie Spring a Oracle a opět zbylými dvěma databázemi. V případě porovnání databáze Oracle a SQL Server byly zjištěny statisticky významné rozdíly a na základě průměrných hodnot i mediánů pak lze jednoznačně určit,

že kombinace Spring s databází Oracle je rychlejší než s databází SQL Server. U porovnání s databází PostgreSQL byly rovněž zaznamenány statisticky významné rozdíly, nicméně vzhledem k naměřeným statistikám vyšlo najevo, že Spring je rychlejší s databází PostgreSQL než s databází Oracle. Lze tedy říci, že Spring nejoptimálnější nespolupracuje s databází Oracle, nýbrž s databází PostgreSQL, přičemž tato kombinace databáze a technologie je v praxi také poměrně častá.

V případě technologie Django se očekávalo, že tato technologie bude rychlostně nejoptimálnější pracovat s databází PostgreSQL. Z provedených testů vyplývá, že mezi kombinací technologie Django a PostgreSQL jsou statisticky významné rozdíly mezi kombinací s databází Oracle i SQL Server. Průměrné hodnoty pak naznačují, že kombinace s databází PostgreSQL je rychlejší než kombinace s databází Oracle i s databází SQL Server. Je možné tedy říci, že Django z hlediska rychlosti odezvy očekávaně nejoptimálněji pracuje s databází PostgreSQL

10 Závěr

V rámci této práce bylo provedeno porovnání rychlosti a stability technologií pro vývoj RESTful API, konkrétně ASP.NET Core, Spring a Django, společně s databázemi SQL Server, Oracle a PostgreSQL. Práce splnila hlavní cíle, kterými bylo identifikovat nejen nejefektivnější kombinace technologií z hlediska rychlostí a stability jednotlivých technologií, ale i také identifikovat, jak se liší chování technologií ve spolupráci se stejnými databázemi či zda budou jednotlivé technologie nejoptimálnější spolupracovat s databázemi se kterými se tyto technologie zpravidla nejčastěji používají.

Výzkum poskytl užitečné poznatky o rychlostech odezvy a stabilitě jednotlivých technologií v kombinaci s různými databázemi, nicméně k výsledkům měření je nutné dodat, že při výběru vhodné technologie, případně kombinace technologie a databáze nelze přihlížet pouze k rychlostem odezvy a stabilitě odezvy jednotlivých kombinací, nicméně k řadě dalších faktorům jako jsou škálovatelnost, podpora a komunita, bezpečnost, náklady a údržba, flexibilita a rozšiřitelnost, dostupnost a odolnost, kompatibilita s jinými technologiemi či dlouhodobá udržitelnost a business strategie.

Zároveň je nutné dodat, že technologie byly testovány za specifických podmínek a mohou se lišit testování v jiných podmínkách v závislosti na mnoha faktorech. Velký vliv na testování může mít pravděpodobně využití asynchronity v rámci jednotlivých technologií, kde by patrně z velké části záleželo na tom, jak jsou jednotlivé technologie schopny asynchronně zpracovávat různé požadavky. Značný vliv na výsledky rovněž může mít výběr testovacího prostředí, kde pro účely této diplomové práce byl vybrán operační systém Windows Server, což alespoň zčásti může vysvětlovat fakt, že technologie ASP.NET Core byla nejrychlejší napříč všemi testovanými systémy. Naopak tento fakt mohl částečně ovlivnit zbylé dvě technologie, a to především Django, které na základě diskuzí pracuje lépe na různých Linuxových distribucích.

Do budoucna lze z hlediska optimalizace u architektury REST, ale i u ostatních architektur očekávat důraz na stále větší optimalizaci těchto technologií právě z hlediska rychlostí a stability. Jedním z dalších pravděpodobných směrů zlepšování je další pokrok v oblasti efektivního využití výše zmíněných asynchronních operací a paralelního zpracování, což může výrazně zvyšovat výkon a odezvu aplikací.

Další oblastí, na kterou bude do budoucna kladen stále větší důraz a která by měla být zdokonalována, je zvyšování bezpečnosti a odolnosti proti různým typům útoků a poruch. S narůstajícím počtem kybernetických hrozeb je z hlediska aplikací důležité nejen minimalizovat možnosti zneužití systému, ale také rychle a efektivně reagovat na případné incidenty. S tím souvisí neustálé zdokonalování nástrojů pro monitorování a správu aplikací, což pomůže včasnému odhalení a řešení potenciálních problémů a zlepší celkovou správu a údržbu systémů. V reakci na stále narůstající počet uživatelů aplikací založených právě na těchto technologiích a jejich narůstajících požadavcích by pak v neposlední řadě by do budoucna být věnována pozornost také udržitelnosti a škálovatelnosti technologií.

Z hlediska dalšího výzkumu by neměla být věnována pozornost pouze rychlostem či stabilitě jako tomu bylo v této práci, ale i výše uvedeným vlastnostem jako je právě bezpečnost či škálovatelnost. Technologie a jejich vlastnosti by měly být zkoumány z různých úhlů pohledu, v různých prostředích a za odlišných podmínek, které mohou značně ovlivnit výsledky měření. Zkoumání chování těch nejpopulárnějších technologií a databází, které byly vybrány v této práci, ale i dalších méně známých technologií je pak velmi zajímavým předmětem dalšího výzkumu.

11 Seznam použité literatury

[1] Demystifying Web API vs Rest API: A Comparison. In: Astera [online]. Updated on: August 23rd, 2023. c2023 [cit. 2023-10-28]. Dostupné z: https://www.astera.com/knowledge-center/web-api-vs-rest-api/.

[2] What is an API: Definition, Types, Specifications, Documentation. In: Altexsoft [online]. Last Updated: 21 Nov, 2022 [cit. 2023-10-28]. Dostupné z: https://www.altexsoft.com/blog/engineering/what-is-api-definition-types-specifications-documentation/.

[3] Difference between http:// and https://. In: Geeks for Geeks [online]. Last Updated: 08 Aug, 2023 [cit. 2023-10-28]. Dostupné z: https://www.geeksforgeeks.org/difference-between-http-and-https/.

[4] BHUDATA, Teja. API Integration: A Practical Guide to Maximizing Business Efficiency. In: Exalate [online]. c2023, Updated on October 17, 2023 [cit. 2023-10-28]. Dostupné z: https://exalate.com/blog/api-integration/

[5] HTTP Full Form. In: Geeks for Geeks [online]. Last Updated : 29 May, 2023 [cit. 2023-10-28]. Dostupné z: https://www.geeksforgeeks.org/http-full-form/

[6] HTTP request methods. In: MDN Web Docs [online]. c1998–2023, Last modified on Apr 10, 2023 [cit. 2023-10-28]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

[7] Different kinds of HTTP requests. In: Geeks for Geeks [online]. Last Updated : 31 May, 2022 [cit. 2023-10-28]. Dostupné z: https://www.geeksforgeeks.org/different-kinds-of-http-requests/

[8] What is a URL? In: MDN Web Docs [online]. c1998–2023, Last modified on Aug 2, 2023 [cit. 2023-10-28]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/Common_questions/Web_mechanics/What_is_a_URL

[9] Request header. In: MDN Web Docs [online]. c1998–2023, Last modified on Jun 8, 2023 [cit. 2023-10-28]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Request_header

[10] HTTP requests. In: IBM [online]. c2015-2021, Last Updated: 2023-06-07 [cit. 2023-10-28]. Dostupné z: https://www.ibm.com/docs/en/cics-ts/5.3?topic=protocol-http-requests

[11] MING JUN, Chin. Sending JSON HTTP Request Body in Terminal. In: Baeldung on Linux [online]. Last updated: June 5, 2023 [cit. 2023-10-28]. Dostupné z: https://www.baeldung.com/linux/json-http-request-body-terminal

[12] HTTP responses. In: IBM [online]. c2014-2021, Last Updated: 2021-03-03 [cit. 2023-10-28]. Dostupné z: https://www.ibm.com/docs/en/cics-ts/5.2?topic=protocol-http-responses

[13] HTTP response status codes. In: Mlytics Learning Center [online]. c2023 [cit. 2023-10-28]. Dostupné z: https://learning.mlytics.com/the-internet/http-response-status-codes/

[14] Working with JSON. In: MDN Web Docs [online]. c1998–2023, Last modified on Jul 3, 2023 [cit. 2023-10-28]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON

[15] Introducing JSON. In: Introducing JSON [online]. [cit. 2023-10-28]. Dostupné z: https://www.json.org/json-en.html

[16] ADHIKARY, Tapas. JSON for Beginners – JavaScript Object Notation Explained in Plain English. In: FreeCodeChamp [online]. 2021 [cit. 2023-10-28]. Dostupné z: https://www.freecodecamp.org/news/what-is-json-a-json-file-example/

[17] What is XML? In: Amazon AWS [online]. c2023 [cit. 2023-10-28]. Dostupné z: https://aws.amazon.com/what-is/xml/

[18] What's the Difference Between JSON and XML? In: Amazon AWS [online]. c2023 [cit. 2023-10-28]. Dostupné z: https://aws.amazon.com/compare/the-difference-between-json-xml/

[19] 2023 State of the API Report. Online. In: Postman. C2023. Dostupné z: https://www.postman.com/state-of-api/api-technologies/#api-technologies. [cit. 2023-12-01].

[20] JANSEN, Geert. Resources. Online. In: Thoughts on RESTful API Design. 2011. Dostupné z: https://restful-api-design.readthedocs.io/en/latest/resources.html. [cit. 2023-12-01].

[21] BURKE, Bill. RESTful Java with JAX-RS 2.0. 2nd ed. Sebastopol: O'Reilly Media, 2014. ISBN 978-1-449-36134-1.

[22] FIELDING, Roy. Architectural Styles and the Design of Network-based Software Architectures. Disertační práce. Irvine: University of California, 2000.

[23] ANURADHA, C. a Arvind PADMANABHAN. Richardson Maturity Model. Online. In: Devopedia. 2020, Accessed 2023-11-12. Dostupné z: https://devopedia.org/richardson-maturity-model. [cit. 2023-12-01].

[24] STRAUSS, Luke. Webhook vs. API: differences (and when to use each). Online. In: Zapier. C2023. Dostupné z: https://zapier.com/blog/webhook-vs-api/. [cit. 2023-12-01].

[25] What is a webhook? Online. In: Red Hat. C2023. Dostupné z: https://www.redhat.com/en/topics/automation/what-is-a-webhook. [cit. 2023-12-01].

[26] What is a webhook? Online. In: Hookdeck. C2022. Dostupné z: https://hookdeck.com/webhooks/guides/what-are-webhooks-how-they-work#what-is-a-webhook. [cit. 2023-12-01].

[27] Basics Tutorial - Introduction. Online. In: How to GraphQL. Dostupné z: https://www.howtographql.com/basics/0-introduction/. [cit. 2023-12-02].

[28] BYRON, Lee. GraphQL: A data query language. Online. In: META PLATFORMS. Engineering at Meta. Dostupné z: https://engineering.fb.com/2015/09/14/core-infra/graphql-a-data-query-language/. [cit. 2023-12-02].

[29] Why and When to Use GraphQL. Online. In: DZone. Updated Jul. 19, 21. Dostupné z: https://dzone.com/articles/why-and-when-to-use-graphql-1. [cit. 2023-12-02].

[30] Simple Object Access Protocol (SOAP). Online. In: OpenText. C2023. Dostupné z: https://www.microfocus.com/documentation/silk-performer/205/en/silkperformer-205-webhelp-en/GUID-FEFE9379-8382-48C7-984D-55D98D6BFD37.html. [cit. 2023-12-02].

[31] Working with WSDLs. Online. In: SoapUI. C2023. Dostupné z: https://www.soapui.org/docs/soap-and-wsdl/working-with-wsdls/. [cit. 2023-12-02].

[32] What is a WSDL file? Online. In: FileFormat. C2001-2023. Dostupné z: https://docs.fileformat.com/cs/web/wsdl/. [cit. 2023-12-02].

[33] KUBA, Martin. Tutorial Web Services. Online. In: ICS MUNI. C2001-2023. Dostupné z: https://dior.ics.muni.cz/~makub/soap/tutorial.html#wsdl. [cit. 2023-12-02].

[34] STAUDINGER, Chris. REST vs. SOAP APIs. Online. In: Postman. C2023. Dostupné z: https://blog.postman.com/soap-vs-rest/. [cit. 2023-12-02].

[35] What Is a Database? Online. In: Oracle. C2023. Dostupné z: https://www.oracle.com/cz/database/what-is-database/. [cit. 2023-12-02].

[36] Types of Databases. Online. In: JavaTpoint. C2011-2021. Dostupné z: https://www.javatpoint.com/types-of-databases. [cit. 2023-12-02].

[37] What is PostgreSQL? Online. In: PostgreSQL. C1996-2023. Dostupné z: https://www.postgresql.org/about/. [cit. 2023-12-02].

[38] What is Oracle? Online. In: JavaTpoint. C2011-2021. Dostupné z: https://www.javatpoint.com/what-is-oracle. [cit. 2023-12-02].

[39] AWATI, Rahul. Microsoft SQL Server. Online. In: TechTarget. C2023, last updated in June 2019. Dostupné

z: https://www.techtarget.com/searchdatamanagement/definition/SQL-Server. [cit. 2023-12-02].

[40] PERÉZ, Sergio Darias. What is Microsoft SQL Server and what is it for? Online. In: Intelequia. C2023. Dostupné z: https://intelequia.com/en/blog/post/what-is-microsoft-sql-server-and-what-is-it-for. [cit. 2023-12-02].

[41] HARTINGER, David. MVC Architektura. Online. In: ITNetwork. C2023. Dostupné z: https://www.itnetwork.cz/navrh/mvc-architektura-navrhovy-vzor. [cit. 2023-12-02].

[42] ROTH, Daniel, Rick ANDERSON a Shaun LUTTIN. Přehled ASP.NET Core. Online. In: Microsoft. 2023. Dostupné z: https://learn.microsoft.com/cs-cz/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-8.0. [cit. 2023-12-02].

[43] Entity Framework. Online. In: Microsoft. 2022. Dostupné z: https://learn.microsoft.com/en-us/aspnet/entity-framework. [cit. 2023-12-02].

[44] UNADKAT, Darshan. Pros and Cons of .Net Core. Online. In: Medium. 2022. Dostupné z: https://medium.com/@darshanunadkat67/pros-and-cons-of-net-core-37ec451edd0. [cit. 2023-12-02].

[45] Spring Framework Overview. Online. In: Spring.io. C2005-2023. Dostupné z: https://docs.spring.io/spring-framework/reference/overview.html. [cit. 2023-12-02].

[46] Difference between MVC and MVT design patterns. Online. In: Geeks for Geeks. Last Updated : 14 Sep, 2022. Dostupné z: https://www.geeksforgeeks.org/difference-between-mvc-and-mvt-design-patterns/. [cit. 2023-12-02].

[47] Django introduction. Online. In: MDN Web Docs. C1998–2023. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction. [cit. 2023-12-02].

[48] Python/Django development, windows or linux?: Diskuze. Online. In: Stackoverflow. 2012. Dostupné z: https://stackoverflow.com/questions/11338382/python-django-development-windows-or-linux. [cit. 2023-12-02].

[49] TAYLOR, Petroc. Ranking of the most popular relational database management systems worldwide. Online. In: Statista. 2023. Dostupné z: https://www.statista.com/statistics/1131568/worldwide-popularity-ranking-relational-database-management-systems/. [cit. 2023-12-02].

[50] Stack Overflow Trends. Online. In: . 2023. Dostupné z: https://insights.stackoverflow.com/trends?tags=asp.net-

core%2Cdjango%2Cspring%2Cexpress%2Cruby-on-rails%2Cflask%2Cfastapi. [cit. 2023-12-02].

[51] Introduction to Spring Framework. Online. In: Spring.io. C2005-2023. Dostupné z: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/overview.html. [cit. 2023-12-03].

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání diplomové práce

Autor:	**Bc. Filip Dvořák**
Studium:	I2200817
Studijní program:	N0688A140001 Informační management
Studijní obor:	Informační management
Název diplomové práce:	**Technologie pro tvorbu webových API**
Název diplomové práce AJ:	Technologies for Web API Development

Cíl, metody, literatura, předpoklady:

Cíl: Provedení analýzy dostupných technologií pro vývoj webových API, především technologií pro vývoj RESTful API. Jejich základní popis, charakteristiky a následné porovnání vybraných technologií pro vývoj RESTful API.

Osnova:

1. Úvod
2. Cíl a metodika práce
3. Webová API
4. RESTful API
5. Technologie pro vývoj RESTful API
6. Alternativní technologie pro vývoj webových API
7. Databázové systémy pro webová API
8. Analýza a porovnání technologií pro vývoj RESTful API
9. Závěr
10. Seznam použité literatury

[1] AMUNDSEN, Michael a DVORAK, Katharine. Design and build great web APIs: robust, reliable, and resilient. Raleigh: Pragmatic bookshelf, 2020. ISBN 978-1-68050-680-8.

[2] RICHARDSON, Leonard, Michael AMUNDSEN a Sam RUBY. RESTful Web APIs. Sebastopol: O'Reilly, 2013. ISBN 978-1-449-35806-8.

[3] JIN, Brenda, Saurabh SAHNI a Amir SHEVAT. Designing Web APIs. Sebastopol: O'Reilly Media, 2018. ISBN 978-1-492-02692-1.

Zadávající pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	doc. Mgr. Tomáš Kozel, Ph.D.
Datum zadání závěrečné práce:	10.10.2023