Možnosti využití frameworku gRPC pro tvorbu webových aplikací

Bc. Martin Feigl

Diplomová práce
2021

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav počítačových a komunikačních systémů

Akademický rok: 2020/2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Martin Feigl**
Osobní číslo:	**A19436**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Počítačové a komunikační systémy**
Forma studia:	**Kombinovaná**
Téma práce:	**Možnosti využití frameworku gRPC pro tvorbu webových aplikací**
Téma práce anglicky:	**The Possibilities of using the gRPC Framework for Web Application Development**

Zásady pro vypracování

1. Popište současný stav webových služeb a souvisejících technologií.
2. Zaměřte se na framework gRPC a jeho aplikaci v prostředí .NET a Blazor.
3. Navrhněte ukázkovou aplikaci s využitím frameworku gRPC.
4. Vytvořte vzorová řešení demonstrující klíčové prvky použití gRPC.
5. Porovnejte navržená řešení s řešením využívající REST webové služby.
6. Formulujte závěry.

Forma zpracování diplomové práce: **Tištěná/elektronická**

Seznam doporučené literatury:

1. gRPC – A high-performance, open source universal RPC framework. gRPC [online]. [cit. 09.11.2020]. Dostupné z: https://grpc.io/
2. GitHub – grpc/grpc-dotnet: gRPC for .NET. [cit. 09.11.2020]. Dostupné z: https://github.com/grpc/grpc-dotnet
3. Blazor | Build client web apps with C# | .NET. .NET | Free. Cross-platform. Open Source. [online]. Dostupné z: https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor
4. WebAssembly [online]. [cit. 2019-11-14]. Dostupné z: https://webassembly.org
5. NAGEL, Christian. Professional C# 7 and .NET Core 2.0. 11th edition. Indianapolis: Wrox, a Wiley Brand, 2018. ISBN 978-1119449270.
6. ALBAHARI, Joseph a Ben ALBAHARI. C# 7.0 in a nutshell. 7th edition. Sebastopol: O´Reilly, 2018. ISBN 978-1491987650.
7. V. HAAS, Andreas, Andreas ROSSBERG, Derek L. SCHUFF, et al. Bringing the web up to speed with WebAssembly. HAAS, Andreas, Andreas ROSSBERG, Derek L. SCHUFF, et al. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY: ACM, 2017, s. 185-200. ISBN 978-1-4503-4988-8.

Vedoucí diplomové práce: **Ing. Erik Král, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání diplomové práce: **15. ledna 2021**
Termín odevzdání diplomové práce: **17. května 2021**

doc. Mgr. Milan Adámek, Ph.D. v.r.
děkan

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
DĚKAN
L.S.

prof. Ing. Karel Vlček, CSc. v.r.
garant oboru

Ve Zlíně dne 15. ledna 2021

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 27.05.2021 Martin Feigl v.r.

ABSTRAKT

Hlavným cieľom diplomovej práce je priblíženie využitia frameworku gRPC pri tvorbe webových aplikácii v prostredí .NET a Blazor. Teoretická časť popisuje súčasný stav webových služieb a primárne sa zameriava na popis frameworku gRPC. Praktická časť sa venuje návrhu a vytvoreniu vzorovej aplikácie, ktorá demonštruje kľúčové prvky použitia technológii gRPC a REST.

Kľúčové slová: gRPC, REST, SOAP, .NET, Blazor, webová služba, webová aplikácia

ABSTRACT

The main aim of the thesis is to describe the usage of the gRPC framework in the creation of web applications in .NET and Blazor environments. The theoretical part describes the current state of web services and primarily focuses on describing the gRPC framework. The practical part of the thesis is devoted to designing and creating a sample application that demonstrates the critical elements of gRPC and REST technology.

Keywords: gRPC, REST, SOAP, .NET, Blazor, web service, web application

Na tomto mieste by som rád poďakoval rodine, priateľom, profesorom a všetkým ľudom, ktorí ma v priebehu celého štúdia podporovali. Osobitné poďakovanie patrí pánovi Ing. Erikovi Královi Ph.D. za jeho vedenie pri tejto diplomovej práci, za jeho trpezlivosť, rady a pripomienky.

Prehlasujem, že odovzdaná verzia diplomovej práce a verzia elektronická nahraná do IS/STAG sú totožné.

OBSAH

ÚVOD

Dnešná moderná komunikácia je založená na internete. Počítače a dátové siete sa čím ďalej viacej podieľajú na našich každodenných činnostiach. Rozvíjajúce sa informačné technológie prinášajú nové vymoženosti a obohacujú náš život o možnosti, ktoré by sme si pred niekoľkými rokmi nedokázali predstaviť. S postupným pokrokom a rastúcou digitalizáciou sa neustále zvyšuje množstvo prenášaných informácii a je preto potreba riešiť efektívny spôsob komunikácie a prenosu dát po internete.

Teoretická časť diplomovej práce sa zaoberá spôsobom, akým vzájomne komunikujú medzi sebou stroje alebo aplikácie v sieti. Popisuje tradičný spôsob komunikácie pomocou webových služieb, ktoré sú založené na protokole SOAP. Jedná sa o jeden z najstarších spôsobov komunikácie, ktorý je postavený na vzdialenom volaní procedúr. Populárnou alternatívou posledných rokov je architektúra rozhrania REST, ktorá je orientovaná dátovo. REST predstavuje jednotný a jednoduchý prístup pomocou HTTP metód. Primárna časť práce sa zaoberá frameworkom gRPC, ktorý bol prvotne vyvíjaný spoločnosťou Google pre interné účely. Neskôr bol tento framework verejne predstavený s príchodom protokolu HTTP/2 a poskytuje nový binárny formát, ktorý disponuje množstvom výhod.

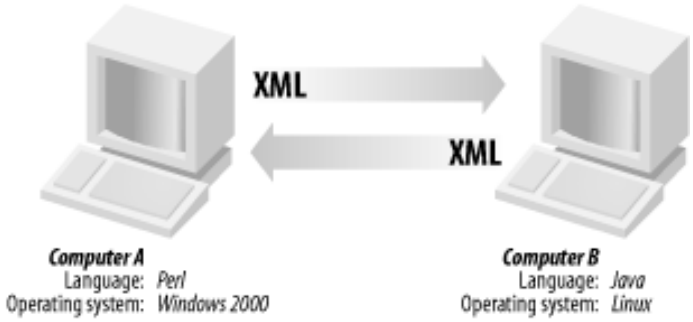
V praktickej časti je navrhnutá a vytvorená jednoduchá webová vzorová aplikácia využívajúca technológie gRPC a REST v prostredí .NET a Blazor. Aplikácia demonštruje kľúčové prvky jednotlivých technológii. Jej úlohou je ukázať základnú komunikáciu založenú na architektúre klient-server a porovnať jednotlivé navrhnuté riešenia komunikácie.

Motiváciou pre výber tejto témy diplomovej práce bola možnosť bližšieho zoznámenia sa s technológiou gRPC, ktorá predstavuje alternatívu s veľkým potenciálom ku bežne používaným riešeniam komunikácie pomocou protokolu SOAP a architektúry REST.

I. TEORETICKÁ ČASŤ

1 WEBOVÉ SLUŽBY

Webová služba je softwarový systém umožňujúci vzájomnú komunikáciu medzi aplikáciami alebo strojmi cez sieť. Jedná sa o zapuzdrenú logiku jednej aplikácie, ktorú využíva druhá aplikácia. Je možné teda hovoriť o programovom rozhraní procedúr alebo metód, ktoré poskytujú nástroj pre ich vzdialené volanie pomocou internetu. Webové služby sú nezávislé na operačnom systéme a programovacom jazyku [1].



Obrázok 1 Komunikácia s webovou službou [1].

Medzinárodné konzorcium W3C, ktoré sa podieľa na vývoji webových štandardov pre Word Wide Web, definuje webové služby ako softvérový systém navrhnutý na podporu interoperabilnej interakcie medzi strojmi v sieti. Má rozhranie opísané v strojovo spracovateľnom formáte (konkrétne WSDL). Ostatné systémy interagujú s webovou službou spôsobom predpísaným v jej popise pomocou správ SOAP, ktoré sa zvyčajne prenášajú pomocou protokolu HTTP so sériovým usporiadaním XML v spojení s inými štandardmi súvisiacimi s webom [2].

Webové služby od ostatných webových aplikácii odlišuje to, že sú určené len na podporu komunikácie medzi aplikáciami [1]. Ostatné webové aplikácie podporujú komunikáciu medzi ľuďmi alebo aplikáciou a človekom. Laickou verejnosťou je často termín webová služba nesprávne interpretovaný a webová služba je chápaná ako služba poskytovaná webovými stránkami a aplikáciami. Webové služby sú navrhnuté tak, aby umožňovali aplikáciám komunikovať bez ľudského zásahu.

Typicky sa pod pojmom webová služba rozumie prenos správ pomocou protokolu SOAP a pridružených technológii WSDL a UDDI [1]. Viaceré definície zaraďujú ku webovým službám aj služby, ktoré využívajú architektúru rozhrania REST. Alternatívu alebo obdobný prístup poskytujú technológie ako sú XML-RPC, CORBA, Java RMI, DCOM alebo gRPC. Technológiám SOAP, REST a gRPC sú venované nasledujúce kapitoly.

1.1 SOA

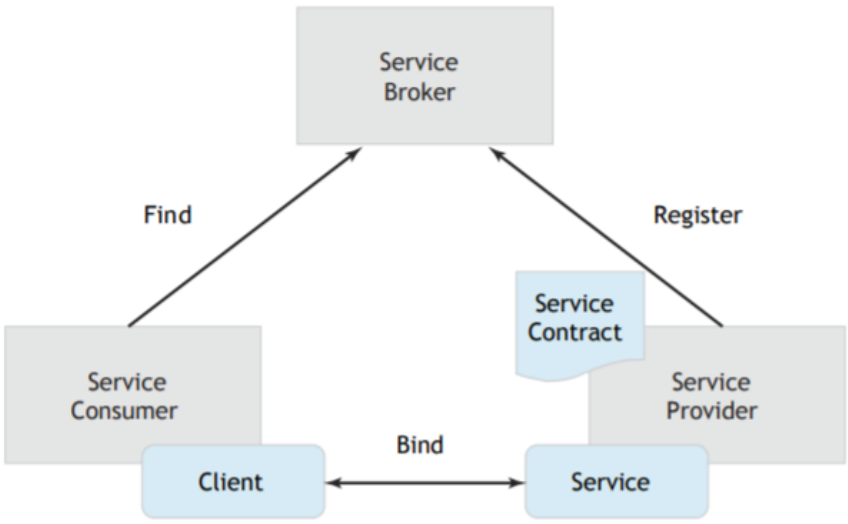
Webové služby sú postavené na architektúre SOA (Service Oriented Architecture). SOA, architektúra orientovaná na služby, je otvorená, agilná, rozšíriteľná a skladateľná architektúra pozostávajúca z autonómnych služieb [2]. Služba je nezávislý softwarový program, ktorý poskytuje presne definovanú funkcionalitu. Metódy každej služby sú zverejnené programovým rozhraním, pomocou ktorého spotrebitelia alebo konzumenti služby komunikujú s danou službou [3].

1.1.1 Základné entity SOA

V koncepte SOA, vystupujú tri základné role alebo entity [3]:

- poskytovateľ služby (service provider),
- spotrebiteľ/konzument služby (service consumer/requestor),
- databáza služieb (service broker/registry).

Vzťah medzi týmito tromi rolami je možné vidieť na obrázku č. 2. Poskytovateľ služby, sprístupňuje samotnú službu a poskytuje jej funkcionalitu. Zverejňuje popis rozhrania služby a prípadne službu registruje v databáze služieb. Nie je povinné registrovať každú službu v databáze služieb a v súčasnosti to prevažne nie je zvykom. V databáze služieb si môže užívateľ nájsť funkcionalitu, ktorú potrebuje a následne pomocou dostupných informácií nadviaže pomocou sieťového pripojenia komunikáciu s poskytovateľom služieb [3].



Obrázok 2 SOA architektúra [3].

1.1.1 Princípy SOA

Architektúra SOA sa riadi viacerými základnými princípmi, medzi ktoré patrí [4]:

- štandardizovaný kontrakt služby,
- slabé väzby medzi komponentmi,
- abstrakcia,
- znovupoužiteľnosť.
- autonómnosť,
- bezstavovosť,
- objaviteľnosť,
- skladateľnosť.

1.1.1.1 Štandardizovaný kontrakt služby

Jedná sa o najdôležitejší princíp na ktorý sa musí brať veľký dôraz pri navrhovaní verejného technického rozhrania služby. Servisný kontrakt, teda predpis služby, musí byť presne definovaný, musí byť vytýčený účel a funkcionalita služby, musia byť opísané dátové typy, modely a všetky technické prostriedky potrebné pre úspešnú komunikáciu so službou [4]. Cieľom je, aby stanované koncové body služby boli dlhodobo konzistentné a spoľahlivé pre spotrebiteľov.

1.1.1.2 Slabé väzby medzi komponentmi

Je kladený dôraz na znižovanie závislosti medzi poskytovanou funkcionalitou služby, jej implementáciou a spotrebiteľmi služby [4]. Na samotnej službe sa môže podieľať viacero komponentov. Slabé väzby a závislosť medzi nimi podporujú nezávislý dizajn a vývoj implementácie služby, zatiaľ čo stále zaručujú spoľahlivosť pre spotrebiteľov.

1.1.1.3 Abstrakcia

Úlohou abstrakcie je čo najviac skryť základné implementačné podrobnosti služby. Zohráva rolu pri skladaní a navrhovaní zložitých systémov. Podporuje princíp slabých väzieb medzi komponentmi [4]. Rozsah abstrakcie môže ovplyvniť konečné náklady služby, má vplyv na úsilie vynaložené na správu a prípadné úpravy implementácie služby.

1.1.1.4 Znovupoužiteľnosť

Znovupoužiteľnosť je významná vlastnosť, na ktorú sa kladie dôraz pri analýze a návrhu služby. Pri návrhu služby alebo komponentu služby by sa malo počítať s jej možným využitím na viacerých projektoch, pri ktorých sa môžu opakovať requesty [4]. Znovupoužitie existujúcich komponentov šetrí čas na implementáciu a správu.

1.1.1.5 Autonómnosť

Aby služba mohla vykonávať správne a spoľahlivo fungovať, musí mať jej základná logika riešenia značnú mieru kontroly nad svojím prostredím a zdrojmi. Úrovne izolácie a úvahy o normalizácii služieb sa berú do úvahy na dosiahnutie vhodného stupňa autonómie, najmä pre opakovane použiteľné služby, ktoré sú často zdieľané [4].

1.1.1.6 Bezstavovosť

Správa nadmerných stavových informácií môže narušiť dostupnosť služby a znížiť jej potenciál škálovateľnosti. Služba by mala byť navrhnutá tak, aby zostala stavová iba v prípade potreby. Službu, ktorá nie je závislá na predošlom stave je možné okamžite jednoducho znovupoužiť [4]. Pre bezstavovosť je treba minimalizovať uchovávanie informácii potrebných pre danú aktivitu. Pre obsluhu requestu by sa nemalo využívať akéhokoľvek kontextu uloženého na servery.

1.1.1.7 Objaviteľnosť

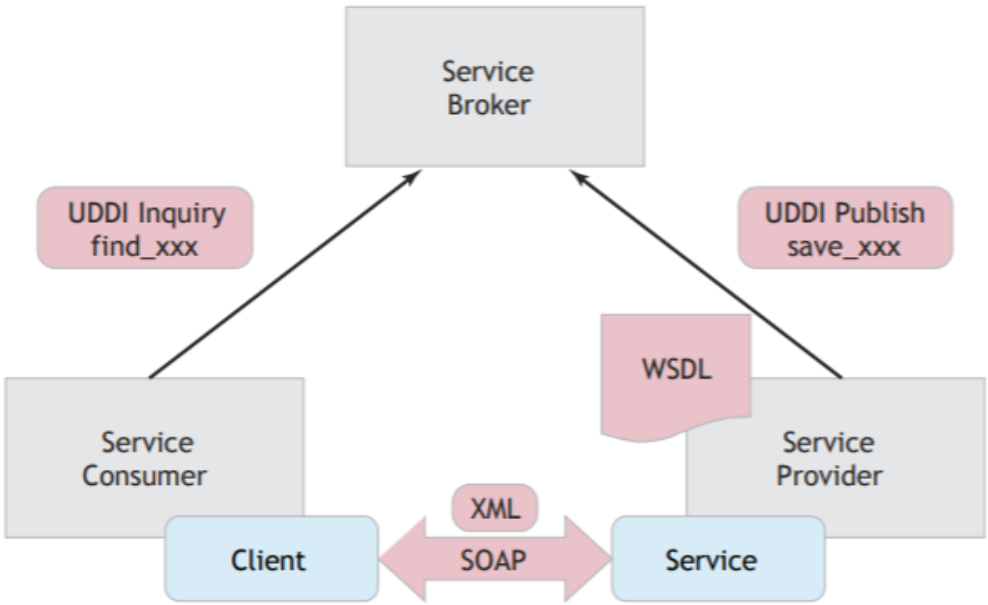
Aby bola služba viacej prístupná pre užívateľov a následne viacej používaná, tak by mala byť jednoducho dostupná, identifikovateľná a pochopiteľná. Služba je navrhnutá tak, aby bola popisná a mohla byť objavená pomocou dostupných vyhľadávacích mechanizmov [4].

1.1.1.8 Skladateľnosť

Každý zložitý komplexný problém by mal byť rozložený na menšie ľahko riešiteľné podproblémy. Skladateľnosť umožňuje spájať nezávislé jednotlivé služby a ich komponenty, ktoré riešia jednotlivé podproblémy [4].

1.2 Architektúra SOAP

SOA architektúra s komunikáciou pomocou protokolu SOAP využíva pre prenos formát jazyku XML, popis služby jazyk WSDL a pre registráciu a vyhľadávanie služieb technológiu UDDI [3]. Ich vzťah môžeme vidieť na obrázku. č. 3.



Obrázok 3 SOAP architektúra [3].

1.2.1 SOAP

SOAP je protokol založený na XML slúžiaci pre výmenu informácií. Predstavuje základný kameň architektúry webových služieb, ktorý umožňuje aplikáciám jednoduchú výmenu služieb a dát [1].

Frameworky a technológie ako CORBA, DCOM a Java RMI poskytujú podobné funkcie ako SOAP, ale správy SOAP sú napísané výlučne v jazyku XML, a preto sú nezávislé na platforme a jazyku. Protokol SOAP môže využívať pre prenos rôzne transportné protokoly, ako sú napríklad FTP, SMTP, TCP, UDT, XMPP, WebDAV, Telnet, Gopher, BEEP, JMS [1]. Hlavným štandardizovaným zameraním SOAP je vzdialené volanie procedúru pomocou protokolu HTTP a využitím jeho metódy POST.

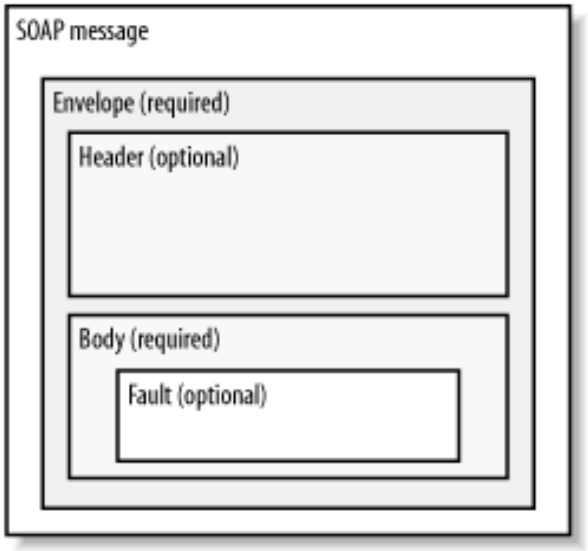
Využitie týchto transportných protokolov má výhodu v tom, že sú obvykle akceptované väčšinou firewallov [1]. SOAP má širokú podporu a existujú desiatky implementácií pre programovacie jazyky.

1.2.1.1 SOAP message

Každý request od klienta alebo response zo serveru sa nazýva správa. Správa je XML dokument, ktorý sa skladá z [1]:

- Obálka (envelope) – Koreňový povinný prvok, zapuzdruje XML dokument a určuje, že ide o SOAP správu. Obsahuje definíciu menných priestorov (namespaces).
- Hlavička (header) – Nepovinný prvok, špecifikuje požiadavky, ktoré musia byť vykonané alebo obsahuje dodatočné informácie pre spracovanie správy.
- Telo (body) – Povinný prvok, ktorý obsahuje všetky prenášané informácie. Ide o názov metódy, parametre metódy alebo jej návratové hodnoty.
- Chyba (fault) – Nepovinný prvok, ktorý sa používa v prípade, že nastane chyba. Obsahuje kód chyby a jej popis.

Štruktúry správa a vzťahy medzi jej časťami správami je možné vidieť na obrázku č. 4. Ukážku SOAP requestu a response je možné vidieť na obrázku č. 5 a obrázku č. 6.



Obrázok 4 Štruktúra SOAP správy [1].

```
<?xml version='1.0' encoding='UTF-8' ?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://www.w3.org/2001/09/soap-envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:getWeather
          xmlns:ns1="urn:examples:weatherservice"
          SOAP-ENV:encodingStyle="http://www.w3.org/2001/09/soap-encoding/">
          <zipcode xsi:type="xsd:string">10016</zipcode>
       </ns1:getWeather>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Obrázok 5 SOAP request [1].

```
<?xml version='1.0' encoding='UTF-8' ?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://www.w3.org/2001/09/soap-envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:getWeatherResponse
         xmlns:ns1="urn:examples:weatherservice"
         SOAP-ENV:encodingStyle="http://www.w3.org/2001/09/soap-encoding/">
         <return xsi:type="xsd:int">65</return>
      </ns1:getWeatherResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Obrázok 6 SOAP response [1].

1.2.2 XML

XML je značkovací jazyk, ktorý bol vyvinutý pre jednoduchý prenos a ukladanie dát. Je navrhnutý tak, aby bol ľahko čitateľný pre človeka a aj pre stroj. XML formát je jeden z najrozšírenejších formátov a spôsobov pre textovú reprezentáciu štruktúrovaných dát [2].

Dokumenty XML sú tvorené ako stromy prvkov. Každý strom začína od koreňového prvku a rozvetvuje sa na podradené prvky. Prvok (element) je tvorený začiatočnou značkou, obsahom prvku a koncovou značkou. Začiatočná značka môže navyše obsahovať pridané informácie (atribúty) [2].

1.2.2.1 Vlastnosti XML

Medzi základne vlastnosti XML patrí [2]:

- Štandardný formát – XML vytvorilo konzorcium W3C ako platforme nezávislý štandard, ktorý ponúka jednoduchý otvorený formát s nenáročnou implementáciou.
- Jazykové sady – Podpora definície jazykovej sady a Unicode znakov.
- Opisnosť – Štruktúra, názvy dátových značiek a ich hodnoty sú sebapopisné.
- Miera informácii – Vďaka štruktúrovaným dátam je možné rozlíšiť dôležitosť a význam dát. XML dokument obsahuje veľkú mieru informácii, oproti značkovacím jazykom zameraných na prezentáciu a vzhľad dát.
- Rozšíriteľnosť – XML neobsahuje preddefinované značky a je treba definovať vlastné. Je možné vytvoriť vlastné schémy a formáty.
- Hypertextové odkazy – Možnosť vytvárať hypertextové odkazy v rámci jedného aj viacerých dokumentoch.

1.2.3 WSDL

WSDL je jazyk založený na XML, ktorý popisuje webovú službu. Popisuje štyri hlavné kategórie údajov [1]:

- verejne dostupné funkcie rozhrania,
- dátové typy vstupných parametrov a návratových hodnôt funkcií,
- informácie ohľadom použitého transportného protokolu,
- adresa pre vyhľadanie služby.

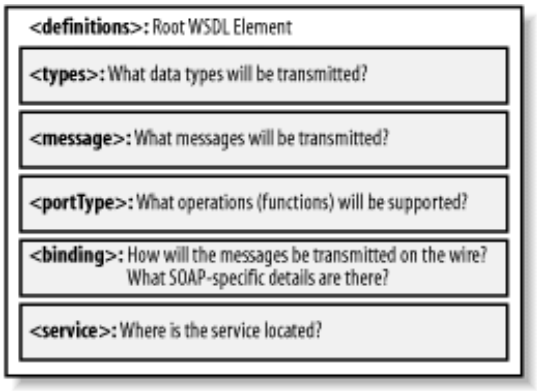
1.2.3.1 Štruktúra WSDL

Jazyk WSDL má presne danú štruktúru a je možné ho považovať ako dokumentáciu pre SOAP klienta. WSDL súbor býva dostupný pomocou protokolu HTTP. Každý XML dokument obsahuje šesť hlavných prvkov [1]:

- definitions – Koreňový prvok celého dokumentu. Definuje názov webovej služby, deklaruje menné priestory (namespaces) a obsahuje všetky ostatné prvky.
- types – Obsahuje všetky dátové typy používané pre komunikáciu.
- message – Popisuje request klienta a response serveru.
- portType – Kombinuje request klienta a response serveru, udáva ktoré funkcie sú dostupné.

- binding – Informácie ohľadom transportného protokolu.
- service – Adresa pre zavolanie služby, obvykle ide o URL adresu.

Prehľad štruktúry s používanými prvkami je možné vidieť na obrázku č. 7 a ukážku WSDL súboru na obrázku č. 8.



Obrázok 7 Štruktúra WSDL dokumentu [1].

Okrem hlavných prvkov môže WSDL obsahovať dva doplňujúce prvky [1]:

- documentation – Dokumentácia vo formáte čitateľnom pre človeka.
- import – Používa sa pre import ďalších WSDL dokumentov alebo XML schém.

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions name="WeatherService" targetNamespace="http://www.ecerami.com/wsdl/WeatherService.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.ecerami.com/wsdl/WeatherService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <message name="getWeatherRequest">
      <part name="zipcode" type="xsd:string" />
   </message>

   <message name="getWeatherResponse">
      <part name="temperature" type="xsd:int" />
   </message>

   <portType name="Weather_PortType">
      <operation name="getWeather">
         <input message="tns:getWeatherRequest" />
         <output message="tns:getWeatherResponse" />
      </operation>
   </portType>

   <binding name="Weather_Binding" type="tns:Weather_PortType">
      <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
      <operation name="getWeather">
         <soap:operation soapAction="" />
         <input>
            <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="urn:examples:weatherservice" use="encoded" />
         </input>
         <output>
            <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="urn:examples:weatherservice"
               use="encoded" />
         </output>
      </operation>
   </binding>

   <service name="Weather_Service">
      <documentation>WSDL File for Weather Service</documentation>
      <port binding="tns:Weather_Binding" name="Weather_Port">
         <soap:address location="http://localhost:8080/soap/servlet/rpcrouter" />
      </port>
   </service>
</definitions>
```

Obrázok 8 Ukážka WSDL dokumentu [1].

1.2.4 UDDI

Jedná sa o databázu alebo register SOAP služieb. Umožňuje jednotný a systematický spôsob pre registrovanie a vyhľadávanie služieb [5]. Užívatelia webových služieb majú možnosť vyhľadať stanovenú funkcionalitu, ktorú poskytuje iná aplikácia.

Popis služby je založený na formáte XML a každý záznam o službe obsahuje tri druhy informácii [5].

- white pages – názov, kontakt, adresa a ďalšie informačné údaje o poskytovateľovi služby,
- yellow pages – informácie pre kategorizáciu typu služby,
- green pages – technické informácie o službe.

```
<businessService
   serviceKey="618167a0-3e64-11d5-98bf-002035229c64"
   businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
   <name>XMethods Currency Exchange Rates</name>
   <description xml:lang="en">
      Returns exchange rates between 2 countries currencies
   </description>
   <bindingTemplates>
      <bindingTemplate
         serviceKey="618167a0-3e64-11d5-98bf-002035229c64"
         bindingKey="618474e0-3e64-11d5-98bf-002035229c64">
         <description xml:lang="en">
            SOAP binding for currency exchange rates service
         </description>
         <accessPoint URLType="http">
            http://services.xmethods.net:80/soap
         </accessPoint>
         <tModelInstanceDetails>
            <tModelInstanceInfo
               tModelKey="uuid:e092f730-3e63-11d5-98bf-002035229c64" />
         </tModelInstanceDetails>
      </bindingTemplate>
   </bindingTemplates>
<businessService>
```

Obrázok 9 Ukážka popisu služby v UDDI [1].

Súkromné UDDI databázy využívajú najmä veľké spoločnosti k integrácii a zefektívneniu prístupu k ich vlastným interným SOAP službám. Dve hlavné verejné UDDI databázy spravovali spoločnosti IBM a Microsoft. V praxi sa ukázalo, že mnoho záznamov bolo neaktuálnych a nefungujúcich. Druhým problémom verejných UDDI je, že nedokážu zaručiť dôveryhodnosť poskytovateľov služieb [6]. Tieto skutočnosti prispeli k tomu, že UDDI nebolo prijaté do takej miery ako sa očakávalo.

Z týchto dôvodov spoločnosti IBM a Microsoft v roku 2006 oznámili uzavretie svojich verejných UDDI a následne spojili úsilie, ktoré bolo pretavené do vytvorenia jazyku WSIL. WSIL ponúka opačný prístup oproti UDDI. Najprv si užívateľ nájde dôveryhodného poskytovateľa a následne ho požiada o popis jeho služby [6].

1.3 REST

Populárnou alternatívou ku webovým službám využívajúcich protokol SOAP je architektonický štýl rozhrania REST (Representational state transfer). S týmto pojmom prišiel prvý krát Roy Fielding, jeden zo zakladateľov protokolu HTTP, vo svojej dizertačnej práci "Architectural Styles and the Design of Network-based Software Architectures" [7].

Kľúčovou abstrakciou pri REST je zdroj (resource). Zdrojom môže byť akákoľvek informácia, ktorú je možné pomenovať. Zdrojom sú teda dáta alebo aj stav aplikácie, s ktorým je možné pracovať pomocou metód HTTP protokolu. REST poskytuje jednotný a jednoduchý prístup ku zdrojom. Každý zdroj má pridelenú URI. URI je jedinečná postupnosť znakov, ktorá slúži ku presnej špecifikácii zdroja informácii, ide teda o názov a adresu zdroja. Oproti SOAP, ktoré je orientované procedurálne, tak REST je orientovaný dátovo [7].

1.3.1 Princípy REST

Webové služby, aby vyhovovali architektúre REST, tak musia spĺňať 6 základných princípov. Službu, ktorá sa riadi danými princípmi nazývame RESTful. Tieto princípy ovplyvňujú spôsoby, ktorými server spracováva a odpovedá na requesty klientov. Správnym dodržaním týchto zásad majú vplyv na nefunkčné požiadavky RESTového rozhrania. Ide o požiadavky, ktoré definujú vlastnosti systému. Medzi nefunkčné požiadavky patrí výkon, škálovateľnosť, jednoduchosť, modifikovateľnosť, viditeľnosť, prenositeľnosť a spoľahlivosť. Medzi základné princípy patrí [7]:

- architektúra klient-server,
- bezstavovosť,
- cache,
- jednotné rozhranie,
- vrstvený systém,
- kód na požiadanie.

1.3.1.1 Architektúra klient-server

Architektúra klient-server patrí medzi najbežnejšie používané architektonické štýly pre sieťové aplikácie. Strana serveru poskytuje svoje služby a reaguje na požiadavky klienta. Strana klienta, ktorá chce prístup ku službe pošle požiadavku (request) strane serveru, server žiadosť vykoná a odošle odpoveď (response) klientovi, prípadne požiadavku zamietne [7].

Tento architektonický štýl prispieva k oddeleniu zodpovednosti (separation of concerns). Oddelením používateľského rozhrania (klient) od strany, ktorá ukladá a pracuje s dátami (server), sa zlepšuje prenosnosť klienta naprieč rôznymi platformami a zlepšuje sa škálovateľnosť serveru [7]. Najvýraznejšou výhodou oddelenia zodpovednosti je možnosť nezávislého vývoja klienta a serveru.

1.3.1.2 Bezstavovosť

Ako už bolo zmienené v kapitole 1.1.2.6, bezstavovosť znamená schopnosť serveru pochopiť a obslúžiť request klienta bez použitia akékoľvek uloženého kontextu na serveri. Stav aplikácie je úplne na klientovi a každý jeho request musí obsahovať všetky potrebné informácie pre jeho obsluhu na strane serveru [7].

Medzi výhody bezstavovosti patrí zlepšenie viditeľnosti, spoľahlivosti a škálovateľnosti [7]. Viditeľnosť je zlepšená tým, že stačí monitorovať a vyhodnotiť obsah jediného requestu. Spoľahlivosť je zvýšená tým, že prípadná chyba jedného requestu nemá vplyv na nasledujúci request. Skutočnosť, že server nemusí ukladať žiadne stavy requestov zlepšuje škálovateľnosť, lebo server je schopný rýchlo uvoľniť svoje prostriedky pre obsluhu ďalšieho requestu.

Nevýhodou bezstavovosti je zníženie výkonu siete zvýšením opakujúcich sa údajov pri každom requeste. Umiestnenie stavu aplikácie na stranu klienta znižuje kontrolu servera nad konzistentným správaním aplikácie [7].

1.3.1.3 Cache

Pre zlepšenie efektívnosti siete sa používa medzipamäť cache, ktorá slúži pre ukladanie odpovedí serveru [7]. Pre možnosť uloženia odpovede do cache pamäte musí byť odpoveď označená ako cacheable. Typicky sa využíva pre ukladanie odpovedí, ktoré sú vracané metódou GET HTTP protokolu. Každá uložená odpoveď má nastavený čas expirácie.

Ak je odpoveď cachovaná, tak má táto klientská vyrovnávacia pamäť možnosť využiť uloženú odpoveď pre ekvivalentné requesty [7]. Tým sa zvýši škálovateľnosť, výkon a zníži sa priemerná doba odozvy serveru. Nevýhodou je zníženie spoľahlivosti, ktoré môže nastať ak by sa uložené zastarané dáta líšili od údajov, ktoré by vrátil server v prípade priameho requestu .

1.3.1.4 Jednotné rozhranie

REST kladie dôraz na jednotné rozhranie medzi komponentmi. Jednotnosť má vplyv na zjednodušenie architektúry, zlepšuje viditeľnosť jednotlivých interakcií a nezávislosť jednotlivých častí [7]. Jednotnosť vyjadruje, že každé rozhranie bude používať protokol HTTP rovnakým spôsobom. Vďaka jednotnému rozhraniu akákoľvek služba prijíma a spracováva rovnako požiadavky ako iná služba.

Aby rozhranie bolo jednotné, tak musí dodržiavať nasledujúce podmienky [7]:

- Identifikácia zdroja – Každý zdroj musí byť jednoznačne identifikovateľný. Zdroj je oddelený od odpovede, ktorá sa vracia klientovi. Klient pracuje len s reprezentáciou zdroja.
- Manipulácia so zdrojmi prostredníctvom reprezentácii – Ak klient prijal reprezentáciu zdroja a k tomu jeho metadáta, tak je schopný s daným zdrojom manipulovať.
- Samopopisné správy – Každá správa má obsahovať dostatok informácii pre jej ďalšie spracovanie.
- HATEOAS (Hypermedia as the Engine of Application State) – Po prístupe k počiatočnej URI zdroju a následnej odpovedi serveru, by klient mal byť schopný pracovať so zdrojom pomocou odkazov, ktoré získa v odpovedi. Inými slovami, klienti by mali komunikovať len na základne znalostí prijatých od serveru.

1.3.1.5 Vrstvový systém

REST umožňuje rozdeliť architektúru medzi viaceré hierarchické vrstvy. Každá vrstva vidí a spolupracuje len s bezprostrednými vrstvami, s ktorými priamo komunikuje [7]. Sám klient nemusí vedieť či komunikuje priamo s koncovým serverom alebo s medzivrstvou slúžiacou ako sprostredkovateľom ku serveru.

Jednotlivé vrstvy je možné použiť pre zapuzdrenie starších služieb a oddelenie nových služieb od starších klientov. Vrstvenie umožňuje odľahčiť celkovú zložitosť systému a podporuje nezávislosť samotných vrstiev [7].

Napríklad pre zabezpečenie je možné použiť samostatnú vrstvu a tým docielime oddelenie aplikačnej logiky od zabezpečenia. Použitie cache pamäte pri vrstvách zvyšuje výkon, ale na druhú stranu môže použitie vrstvových systémov zvyšovať režijné náklady a latenciu pri spracovaní údajov [7].

1.3.1.6 Kód na požiadanie

Kód na požiadanie je nepovinná podmienka REST architektúry. Spočíva v možnosti rozšíriť funkcionalitu klienta prenosom spustiteľného kódu zo serveru, ako sú napríklad Java appely alebo kódy Javascriptu [7]. Výhodou kódu na požiadanie je rozšírenie funkcionality, ale nevýhodou je zhoršenie prehľadnosti a bezpečnosti.

1.3.2 Metódy pre prístup ku zdrojom

REST využíva primárne protokol HTTP a jeho základné metódy, ktorú umožňujú CRUD (create, read, update, delete) operácie. Parametre metód sa prenášajú priamo v hlavičke správy ako súčasť URI adresy alebo v tele správy. Odpoveď serveru obsahuje hlavičku a prípadne telo správy. Hlavička odpovede obsahuje stavový kód a slovný popis stavového kódu. Stavový kód vyjadruje či bol request úspešne vykonaný. Medzi metódy pre prístup ku zdrojom patrí [8]:

- GET (Read) – Základná metóda pre prístup ku zdrojom slúžiaca pre získanie dát zo zdroja.
- POST (Create) – Slúži pre vytvorenie nového zdroja.
- PUT (Update) – Umožňuje editovať existujúci zdroj. Dá sa použiť aj pre vytvorenie nového zdroja.
- DELETE – Odstráni zdroj zo serveru.

1.3.3 JSON

Pre výmenu dát podporuje REST širokú škálu dátových formátov ako sú napríklad JSON, XML, YAML, HTML, RSS, ATOM. Najpoužívanejší formát je JSON (JavaScript Object Notation). JSON je jednoduchý textový formát, založený na JavaScriptových objektoch, ktorý je jednoducho čitateľný pre ľudí aj stroje [8]. Pre počítače sa jednoducho parsuje a generuje.

Pozostáva zo štruktúry kľúč-hodnota. JSON je populárny v kombinácii s REST rozhraním a asynchrónnou AJAX komunikáciou [8]. Rovnako ako XML je platforme a jazykovo nezávislý. Na obrázku č. 10 môžeme vidieť odpoveď serveru na metódu GET, telo odpovede je vo formáte JSON.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
...

{
  "collection" :
  {
    "version" : "1.0",
    "href" : "http://localhost:1337/api/",

    "items" :
    [{
      "href": "http://localhost:1337/api/2csl73jr6j5",
      "data": [
        {
          "name": "text",
          "value": "Bird"
        },
        {
          "name": "date_posted",
          "value": "2013-01-24T18:40:42.190Z"
        }
      ]
    }],

    "template" : {
      "data" : [
        {"prompt" : "Text of message", "name" : "text", "value" : ""}
      ]
    }
  }
}
```

Obrázok 10 Response metódy GET vo formáte JSON [8].

2 GRPC

gRPC je moderný výkonný open source framework založený na vzdialenom volaní procedúr. Za vznikom frameworku gRPC stojí spoločnosť Google. Google využíval viac ako desať rokov pre prepojenie veľkého množstva svojich interných mikroservís (mikroslužieb) v rámci svojich dátových centier jednu univerzálnu infraštruktúru s názvom Stubby [9].

Spoločnosť Google zažívala veľmi rýchly rast a preto potrebovala efektívnu, bezpečnú a spoľahlivú infraštruktúru, ktorá pokryje jeho zvyšujúce sa nároky na údržbu a správu jeho celého systému. Infraštruktúra Stubby taktiež disponovala pripojiteľnou podporou pre vyrovnávanie záťaže, sledovanie, kontrolu stavu a autentifikáciu [9].

Jednalo sa teda o internú technológiu Googlu úzko spojenú s ich vlastnou infraštruktúrou, ktorá nebola založená na žiadnom štandarde, a preto nebola vhodná pre verejné použitie. To sa zmenilo s príchodom technológii SPDY, HTTP/2 a QUIC. Tieto technológie poskytovali viaceré nové funkcie ako štandardy [9].

Niektoré tieto funkcie využívalo už Stubby predtým, a tak Google využilo výhody tejto štandardizácie, prepracovalo Stubby a rozšírilo jeho uplatniteľnosť okrem mikroservís aj pre využitie vo webových a mobilných aplikáciách, internetu vecí a cloude. Táto nová verzia Stubby bola sprístupnená v roku 2015 ako open source framework gRPC s licenciou Apache License 2.0. Od roku 2017 má na starosť vývoj a smerovanie gRPC spoločnosť Cloud Native Computing Foundation (CNCF) [9]. Samotná skratka „gRPC“ je rekurzívny akronym pre „gRPC Remote Procedure Call“, písmeno „g“ je niekedy neoficiálne označované ako „Google“ alebo „general“.

2.1 Princípy gRPC

CNCF zaviedlo súbor zásad, ktorými sa bude gRPC uberať. Primárne sa zaoberajú maximalizáciou prístupnosti a použiteľnosti. Medzi tieto princípy patrí [9]:

- Servisy a správy – Filozofia návrhu mikroservís a výmeny správ medzi systémami. Použitie servís miesto objektov a správ miesto referencií.
- Pokrytie a jednoduchosť – Dostupnosť na všetkých populárnych platformách, životaschopnosť na zariadeniach s obmedzeným procesorovým výkonom alebo pamäťou.
- Open source – Všetky gRPC funkcie a pluginy by mali byť open source pre uľahčenie a nebránenie prijatia gRPC vývojármi.

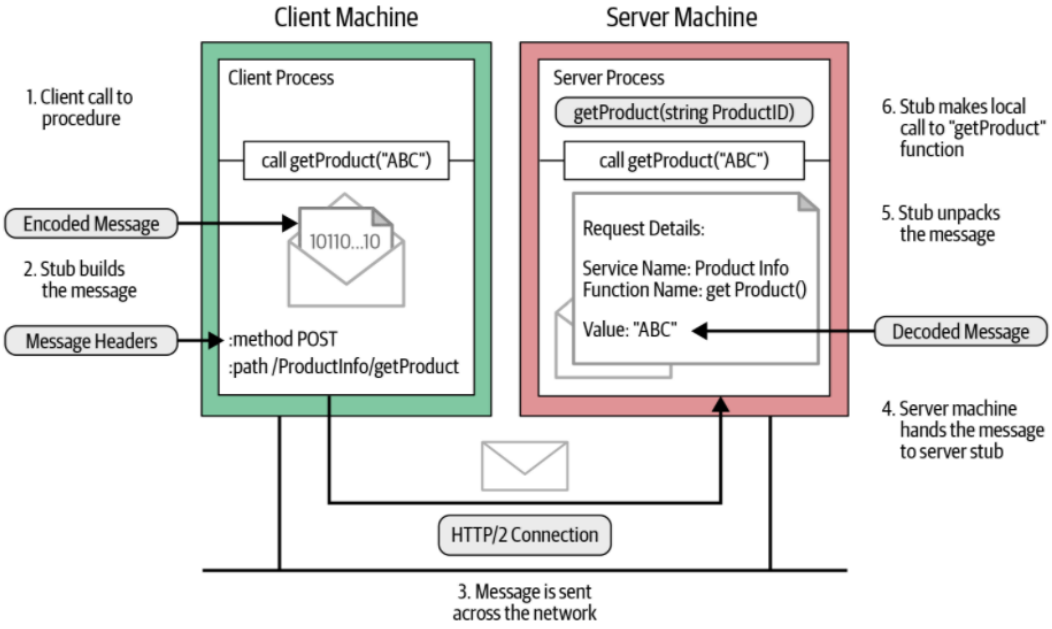
- Interoperabilita a dosah – Prenos pomocou všeobecne dostupných sieťových štandardov.
- Všeobecnosť a výkon – Použiteľnosť pre čo najširšiu škálu prípadov použitia bez ohrozenia zníženia výkonu.
- Vrstvenie – Kľúčové aspekty musia byť schopné nezávislého vývoja bez narušenia väzieb aplikačnej vrstvy.
- Dátový formát – Podpora rôznych dátových formátov a kódovania, podpora formátov ako sú protocol buffers, JSON, XML, Thrift.
- Streamovanie – Poskytnutie sémantiky streamovania pre veľké súbory dát alebo pre asynchrónne zasielanie správ.
- Blokovanie a neblokovanie – Podpora asynchrónneho aj synchrónneho spracovania sekvencie správ vymieňaných medzi klientom a serverom.
- Zrušenie a časový limit – Možnosť zrušenia dlhotrvajúcich a náročných operácii pre spätné získanie zdrojov servera. Klient má možnosť nastavenia vlastného timeoutu.
- Lameducking – Server je možné vypnúť a odmietnuť nové požiadavky bez ovplyvnenia spracovania existujúcich požiadaviek.
- Flow control – Výpočtový výkon a kapacita siete sú medzi klientom a serverom často nevyvážené. Riadenie toku umožňuje lepšiu správu medzipamäte a poskytuje ochranu pred DDoS útokmi.
- Pripojiteľnosť – Možné pripojenie rozšírení a pluginov, ako sú napríklad zabezpečenie, kontrola stavu, vyvažovanie záťaže, failover, monitorovanie, sledovanie a logovanie.
- Rozšíriteľnosť ako API – Rozšírenia, ktoré vyžadujú spoluprácu medzi službami by mali uprednostňovať použitie API a nie rozšírenie protokolu.
- Výmena metadát – Oddelenie a samostatné spracovanie neaplikačných dát, ako sú napríklad autentifikačné tokeny.
- Štandardizované stavové kódy – Obmedzenie variability stavových kódov pre jasnejšie spracovanie chýb. Prípadné rozšírenie stavových kódov je možné zaručiť v rámci výmeny metadát.

2.2 Základ gRPC

Ako už vyplýva z názvu samotného gRPC, základ komunikácie tohto frameworku, rovnako ako pri komunikácii pomocou protokolu SOAP, spočíva vo vzdialenom volaní procedúr (RPC). Server implementuje množinu funkcií (metód), ktoré je možné vyvolať na diaľku. Klient môže vygenerovať stub, modul slúžiaci ako rozhranie klienta, ktorý poskytuje abstrakciu pre volanie funkcií poskytovaných serverom [10].

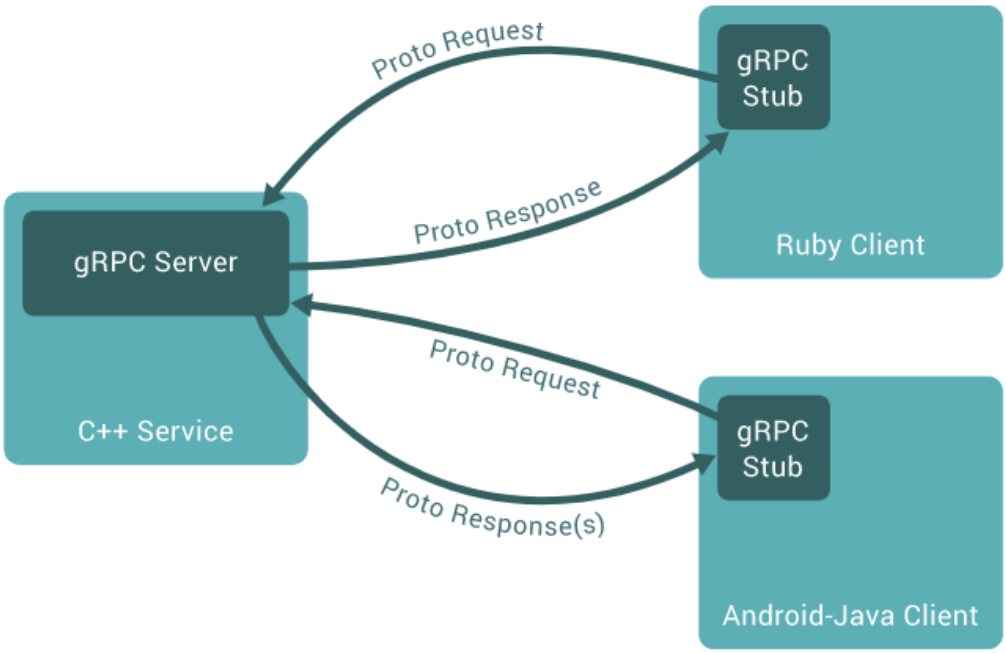
Samotný workflow komunikácie pri zavolaní funkcie je možné vidieť na obrázku č. 11. Postup spočíva v [10]:

1. Klient zavolá funkciu servera pomocou vygenerovaného stubu.
2. Klientský stub vytvorí HTTP POST požiadavku so zakódovanou správou. Všetky gRPC požiadavky sú ako HTTP POST požiadavky s content-type s predponou application/grpc. Názov volanej funkcie sa odošle ako samostatná HTTP hlavička.
3. HTTP požiadavka so správou sa pošle cez sieť na server.
4. Server prijme správu a preskúma hlavičku správy, aby zistil ktorú funkciu je treba zavolať, správu dekóduje a predá serverovému stubu.
5. Stub naparsuje bajty správy do dátových štruktúr špecifikovaných pre použitý jazyk.
6. Stub lokálne zavolá danú funkciu, ktorej predá naparsované dáta.
7. Odpoveď z funkcie je zakódovaná a odoslaná rovnakým spôsobom klientovi.



Obrázok 11 Workflow gRPC volania funkcie [10].

gRPC je nezávislé na platforme a na použitom programovacom jazyku, pri implementácii klienta a servera môžu byť použité rozličné technológie (obrázok č. 12). Najnovšie rozhrania Google API poskytujú svoje verzie aj v gRPC, čím uľahčujú používanie Google funkcií v systémoch založených na gRPC [9].



Obrázok 12 gRPC komunikácia [9].

2.3 Protocol buffers

Najväčším rozdielom gRPC oproti iným RPC frameworkom je využitie protocol buffers. gRPC využíva protocol buffers ako primárny prenosový formát a zároveň ako jazyk definujúci rozhranie (IDL) [9].

2.3.1 .proto súbor

Definícia rozhrania spolu s použitými dátovými štruktúrami je obsiahnutá v súbore s príponou *.proto*. Proto súbor nemusí obsahovať všetky použité message a v prípade potreby je možné ich importovať z iných proto súborov [9].

Ukážku štruktúry proto súboru môžeme vidieť na obrázku č. 13 a delí sa na nasledujúce časti [9]:

- syntax – Verzia použitého protocol buffer.
- package – Názov balíku slúžiaci pre definíciu menného priestoru.
- service – Súbor metód, ktoré je možné vzdialene volať.
- rpc – Vzdialená metóda, každá metóda obsahuje vstupný a návratový dátový typ.
- message – Užívateľom definovaná dátová štruktúra.

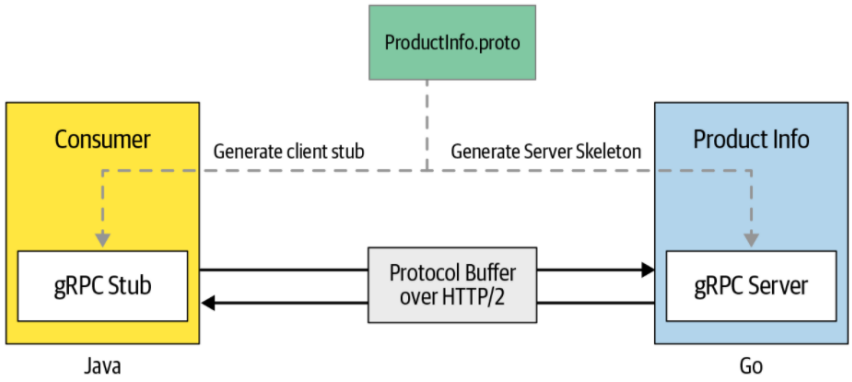
```
// ProductInfo.proto
syntax = "proto3";
package ecommerce;

service ProductInfo {
    rpc addProduct(Product) returns (ProductID);
    rpc getProduct(ProductID) returns (Product);
}

message Product {
    string id = 1;
    string name = 2;
    string description = 3;
}

message ProductID {
    string value = 1;
}
```

Obrázok 13 Štruktúra proto súboru [10].

Proto súbor je možné použiť pre vygenerovanie kódu (obrázok č. 14) na strane servera (server skeleton/stub) aj na strane klienta (client stub) v ľubovoľnom podporovanom programovacom jazyku [9]. Všetky podrobnosti o komunikácii na nízkej úrovni sú implementované do generovaného kódu, ktorý poskytuje abstrakciu na vysokej úrovni. Táto abstrakcia slúži pre jednoduchú prácu s volaniami RPC a odstraňuje potrebu vedieť podrobnosti samotnej implementácie RPC [10]. Na strane servera a klienta sa vygeneruje základná kostra funkcií obsiahnutých v definícii rozhrania a na vývojárovi je implementovať vnútornú aplikačnú logiku.

Protocol buffers nie je samopopisný dátový formát na rozdiel napríklad od formátu JSON, kde je možné dešifrovať pole a hodnotu v dátovej štruktúre iba skúmaním samotnej dátovej

štruktúry. Protocol buffers potrebuje „manuál“ pre klienta a pre server, ktorý slúži ako mechanizmus pre serializáciu a deserializáciu posielaných dát, a týmto manuálom je práve proto súbor [10]. Užívateľ sa teda nemusí zaoberať spôsobmi fungovania serializácie dát, sieťovej komunikácie pomocou protokolu HTTP, autentifikácie a ani ostatných vecí ako samotné RPC funguje v sieti [9].



Obrázok 14 Generovanie gRPC kódu [10].

Špeciálny nástroj pre generovanie tohoto kódu, ktorý je zodpovedný za serializáciu a deserializáciu dát, sa nazýva protoc [9]. Zo serverového proto súboru je možné vygenerovať aj kód pre klienta, klient teda nemusí mať fyzicky kópiu proto súboru. Protoc umožňuje klientovi vygenerovanie kódu, ktorý bude mať vložený logický model schémy serverového proto súboru alebo taktiež umožňuje pre klienta dynamicky pristupovať ku serverovému proto súboru, ktorého schému načíta priamo do pamäte a použije ju pre spracovanie binárnej správy [11].

Tento dynamický prístup sa používa pre prostredia, v ktorých sa server a klient nachádza v rámci rovnakého fyzického systému, ktorý umožňuje, aby obidve strany mali prístup za behu k identickej kópii proto súboru [11]. Táto logika má za výhodu synchronizáciu všetkých zmien v proto súbore, obidve strany budú pristupovať k jednému proto súboru.

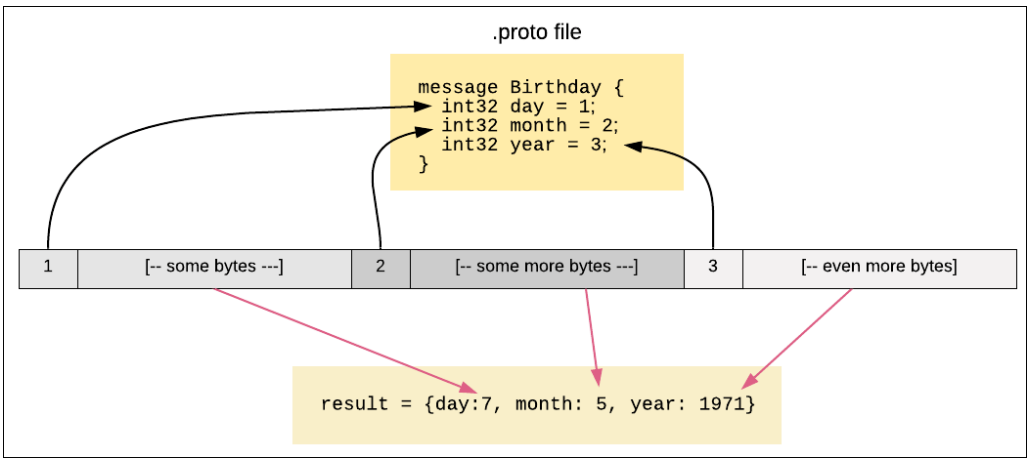
Riešením pre prostredia, kde nie je možné zaručenie spoľahlivého lokálneho prístupu klienta ku serverovému proto súboru, je fyzické skopírovanie verejne dostupného serverového proto súboru na stranu klienta [11]. Tento prístup ma nevýhodu v tom, že do proto súboru klienta je nutné vždy prenášať všetky zmeny z proto súboru serveru.

2.3.2 Prenos správy

Protocol buffers, skrátene protobuff, je jazykovo agnostický, platforme nezávislý spôsob ako efektívne serializovať a deserializovať štruktúrované dáta [9]. gRPC podporuje taktiež formáty JSON, XML alebo Thrift. Na rozdiel od JSON a XML, ktoré používa textový formát, tak protocol buffers využíva pre prenos binárny formát.

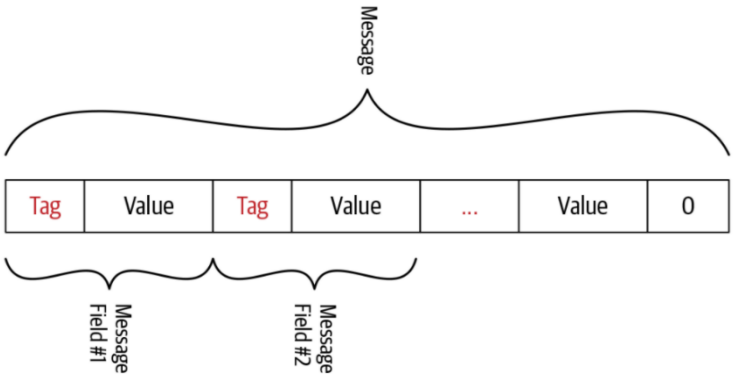
Vstupné a návratové dátové typy sú pri gRPC metódach definované pomocou messages (správ) [9]. Tieto správy sú posielané medzi klientom a serverom. Jedná sa o základnú protobuff dátovú štruktúru, každá správa je malým logickým záznamom informácii obsahujúcich sériu key-value párov nazývaných fields (polia) [10].

Správu je možné prirovnať v objektovo orientovanom programovaní ku classe (triede), ktorá obsahuje jednotlivé property. Rovnako ako každá property, tak aj každý field má určený dátový typ. Field musí mať navyše priradený jednoznačný číselný identifikátor unikátny v rámci definície správy [9]. Identifikátor poľa slúži pre rozpoznanie jednotlivých polí v binárnom formáte, aby každé pole bolo správne naparsované pri deserializácii. Užívateľ je zodpovedný za správnu definíciu správy, ostatné záležitosti ohľadom kódovania rieši priamo gRPC [10]. Zjednodušený princíp posielania správy je možné vidieť na obrázku č. 15.



Obrázok 15 Zjednodušený prenos správy

Na pozadí sa pri kódovaní správ do binárneho formátu prevedú tagy a hodnoty polí na postupnosť bajtov (obrázok č. 16). Ako prvý sa zakóduje tag. Pod pojom tag rozumieme číselný identifikátor poľa a wire type (obrázok č. 17). Wire type je preddefinovaná hodnota, ktorá prislúcha jednotlivým dátovým typom (tabuľka č. 1) [12].



Obrázok 16 Štruktúra správy pri prenose [10].

Obrázok 17 Štruktúra tagu [10].

Tabuľka 1 Wire types a dátové formáty [12].

Type	Meaning	Used For
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64-bit	fixed64, sfixed64, double
2	Length-delimited	string, bytes, embedded messages, packed repeated fields
3	Start group	groups (deprecated)
4	End group	groups (deprecated)
5	32-bit	fixed32, sfixed32, float

Obidve časti tagu sa prevedú do binárnej reprezentácie a pomocou binárneho posunu o 3 bity sa získa hodnota tagu. Prvé bity budú predstavovať číselný identifikátor poľa, posledné tri bity predstavujú wire type. Pomocou wire type je možné zistiť pri deserializácii dĺžku nasledujúcej hodnoty poľa [12]. Na obrázku č. 18 je možné vidieť hodnotu tagu v binárnej forme, ktorá obsahuje číselný identifikátor s hodnotou 1 a wire type s hodnotou 2.

Následne sa prevedie do binárnej podoby hodnota daného poľa [12]. Jednotlivé hodnoty sa kódujú rozlične podľa dátového typu. Napríklad hodnota poľa dátového typu int32 by sa zakódovala pomocou varint kódovania, hodnota dátového typu sint32 pomocou zigzag a varint kódovania alebo string pomocou UTF-8 [10].

```
Tag value = (field_index << 3) | wire_type
Tag value = (00000001 << 3) | 00000010
          = 000 1010
```

Obrázok 18 Výpočet hodnoty tagu [12].

Pri textovom formáte, ktorý napríklad používa JSON, sú dáta, rovnako ako pri binárnom formáte, pri posielaní správy prevádzané na postupnosť bitov, ale pred prevodom sa používa obvykle UTF alebo ASCII kódovanie, ktoré zodpovedá ľudsky čitateľným znakom [10]. Binárny formát prevádza dáta priamo na postupnosť bitov. Binárny formát je teda všeobecne kratší a jednoduchší na spracovanie.

2.3.3 Porovnanie rýchlosti JSON a Protocol buffers

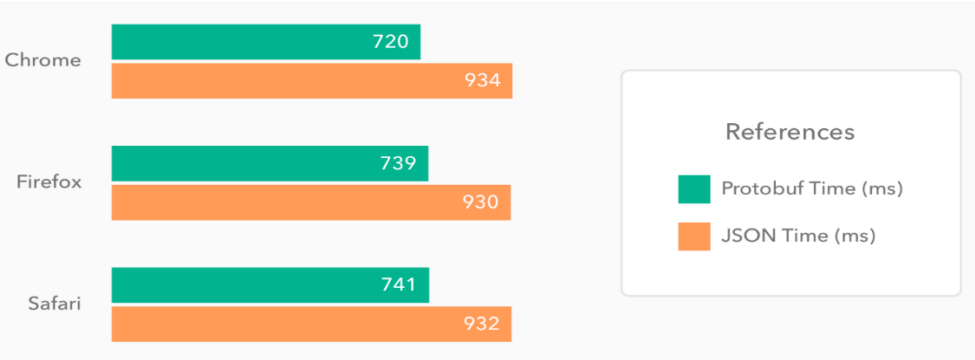
Binárny formát protocol buffers je na prvý pohľad nečitateľný pre človeka, ale efektívnejšie využíva šírku pásma. Protocol buffers oddeľuje kontext a samotné dáta. Polia sú zoradené podľa číselných identifikátorov, pre každé pole sa prenáša len jeho hodnota a príslušný číselný identifikátor, ktorý slúži ako kľúč pri deserializácii. To umožňuje rýchlejšie spracovanie oproti textových formátom ako JSON, pri ktorých môžu byť polia posielané v akomkoľvek poradí, a pri ktorých sú polia identifikované podľa názvu poľa, ktorý sa musí vždy posielať spolu s poľom [11].

Hlavnou výhodou binárneho formátu protocol buffers je rýchla serializácia dát a deserializácia dát, pri väčších dátových prenosoch sa uvádza niekoľko násobná väčšia rýchlosť prenosu dát oproti formátom ako JSON a XML [9].

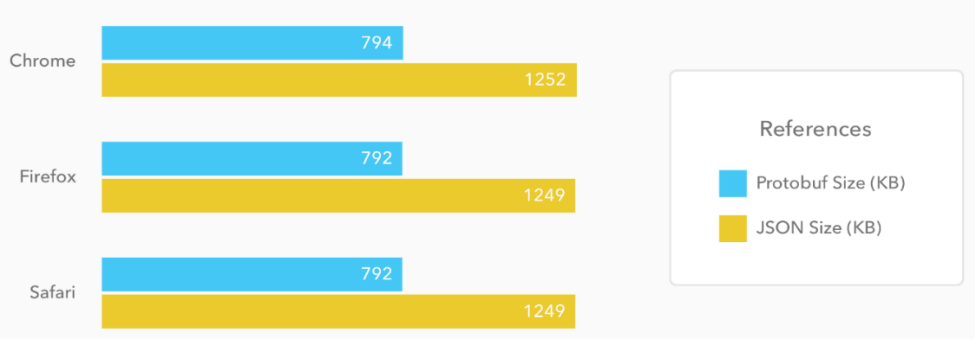
Existuje viacero publikácii a príspevkov, ktoré porovnávajú rýchlosť prenosu a veľkosť prenášaných dát pri použití formátov JSON a protocol buffers. Jeden z takýchto pokusov bol uverejnený na webovej stránke Auth0.

Tento pokus pozostával z dvoch častí. V prvej časti bola porovnaná komunikácia Javascript webovej aplikácie s Java serverom, konkrétne bol použitý Spring Boot framework. Testovali sa webové prehliadače Chrome, Firefox a Safari. Pre pokus sa použil GET request, ktorý vrátil údaje o 50 tisíc ľuďoch a POST request, ktorý poslal údaje o jednom človeku. Každý človek bol reprezentovaný menom, kolekciou mobilných čísiel, kolekciou e-mailových adries a kolekciou adries, pričom každá adresa pozostávala s názvu ulice a čísla domu [13].

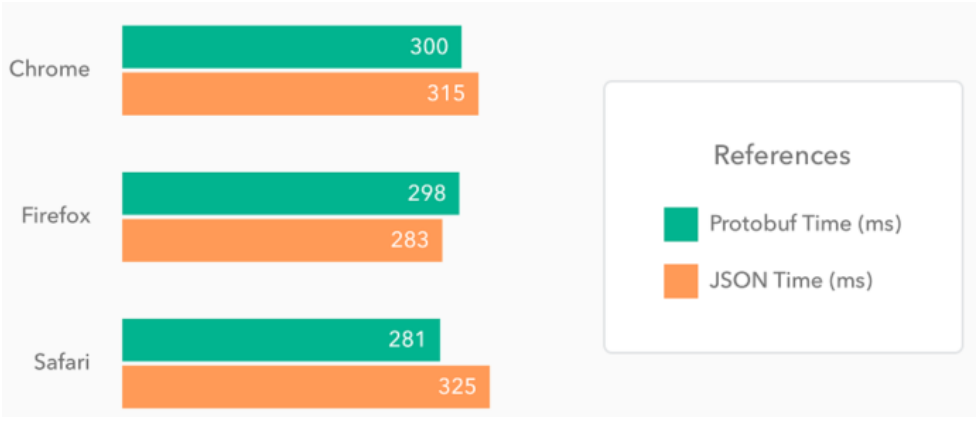
Nasledujúce grafy znázorňujú priemerné výsledky 50 spustení. Je možné vidieť, že pri GET requeste a použití protocol buffers boli dáta poslané o približne 21% rýchlejšie (obrázok č. 19) a boli o 34% menšie (obrázok č. 20) pri nezapnutej Spring boot kompresii. Protocol buffers dosiahol lepšie výsledky, aj napriek tomu, že musel byť navyše prevedený z binárneho formátu na JSON, ktorý používa Javascript ako svoj natívny formát správ. Pri POST requeste boli rozdiely menšie, z dôvodu posielania malých správ. Dáta boli prenesené za približne rovnaký čas (obrázok č. 21) a boli rovnako veľké (obrázok č. 22) [13].



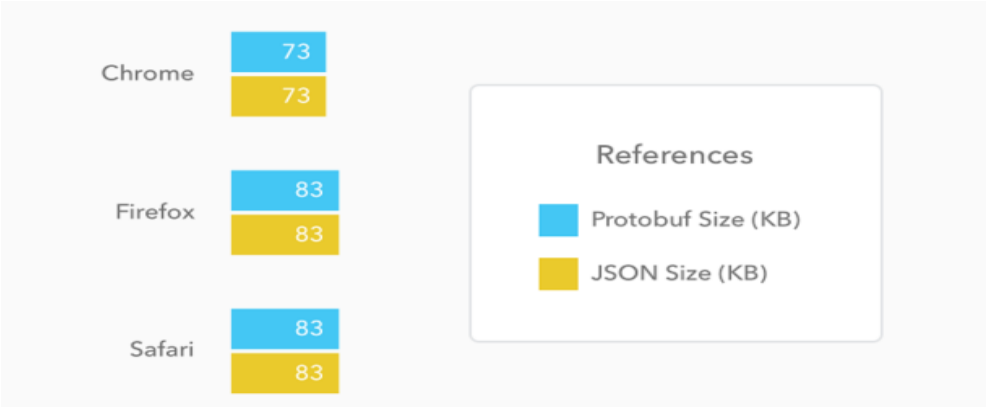
Obrázok 19 Porovnanie rýchlosti JSON a protobuf pre GET request [13]



Obrázok 20 Porovnanie veľkosti dát JSON a protobuf pre GET request [13].

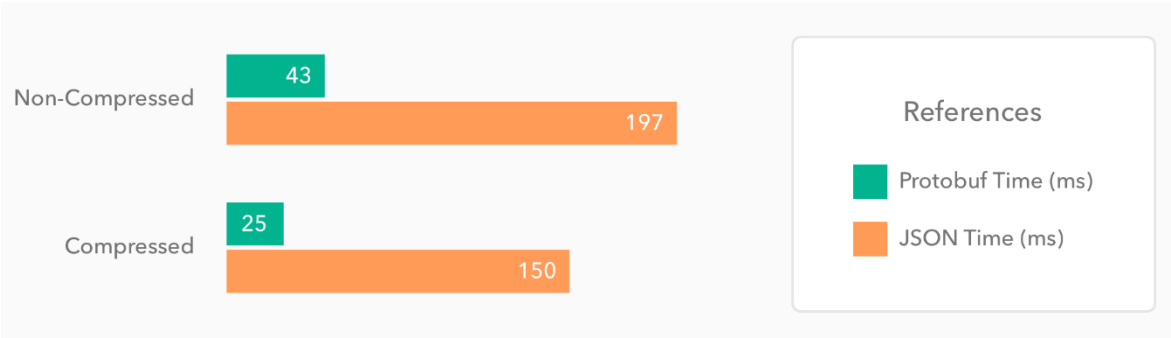


Obrázok 21 Porovnanie rýchlosti JSON a protobuf pre POST request [13].



Obrázok 22 Porovnanie veľkosti dát JSON a protobuf pre POST request [13].

Druhá časť pokusu pozostávala s testovania komunikácie medzi dvomi Java aplikáciami. Každá aplikácia bola nasadená na rozdielny virtuálny stroj a simulovala komunikáciu dvoch mikroservís. Pri tomto pokuse bolo 500 GET requestov a protocol buffer naplno prejavil svoju efektivitu (obrázok č. 23), nebol obmedzený Javascriptom a podarilo sa mu dosiahnuť priemerne až 5 násobne väčšiu rýchlosť pri nezapnutej kompresii dát a takmer 6 násobne väčšiu rýchlosť pri zapnutej kompresii dát [13].



Obrázok 23 Porovnanie rýchlosti JSON a protobuf pre GET request [13].

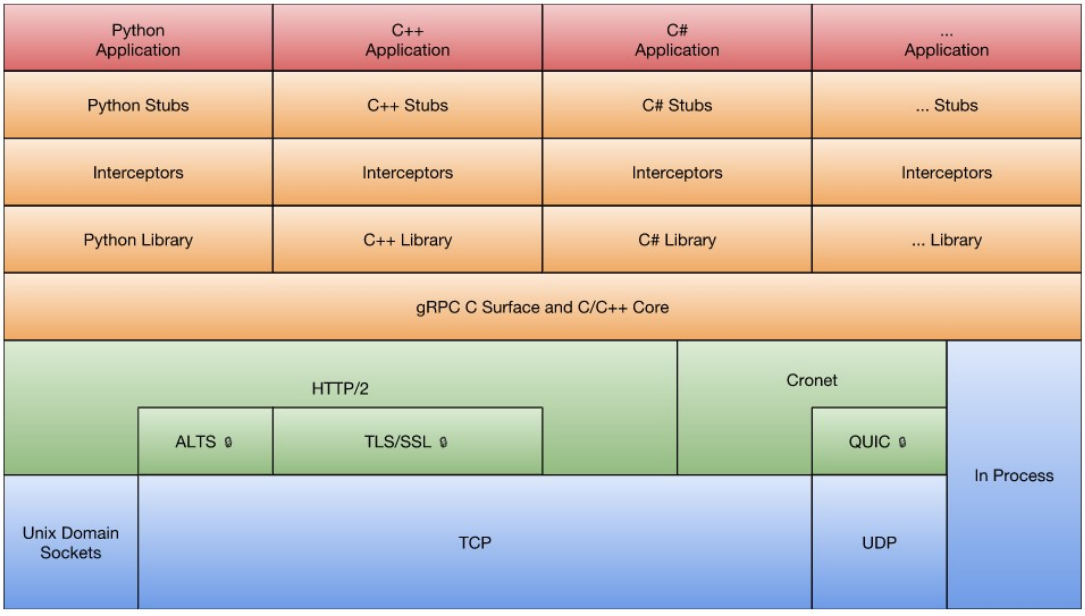
2.4 Jadro gRPC

Samotné jadro gRPC tvoria tri knižnice napísané v C-Core, Go a Jave. Toto jadro predstavuje jednotnú horizontálnu vrstvu, ktorá abstrahuje všetky podporované programovacie jazyky. Všetky tieto jazyky sú tenký obal jadra gRPC [9]. To znamená, že všetky gRPC volania bez ohľadu na jazyk aplikácie, sú vnútorne preložené do jazykov C-Core, Go alebo Java.

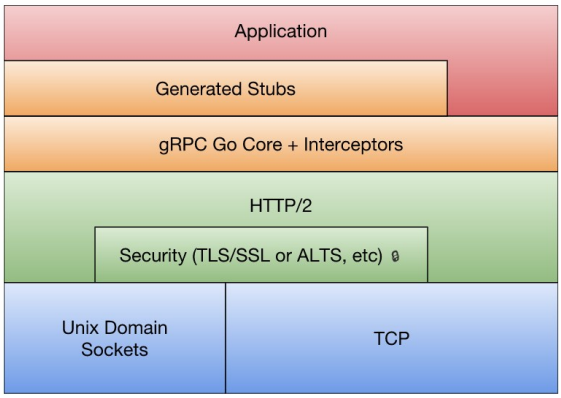
Všetky jazyky, okrem jazyka Go a Java, používajú jadro postavené na C-Core. Samotná komunikácia pozostáva z viacerých vrstiev [9]. Všetky vrstvy komunikácie, ktoré sú založené na C-Core, je možné vidieť na obrázku č. 24.

Aplikácia zavolá pomocou stubu RPC metódu, ktorá prechádza cez interceptory. Interceptor je handler udalosti pred alebo po vykonaní metódy RPC, môže sa jednať napríklad o logovanie, ktoré sa spustí pred a po vykonaní metódy [9].

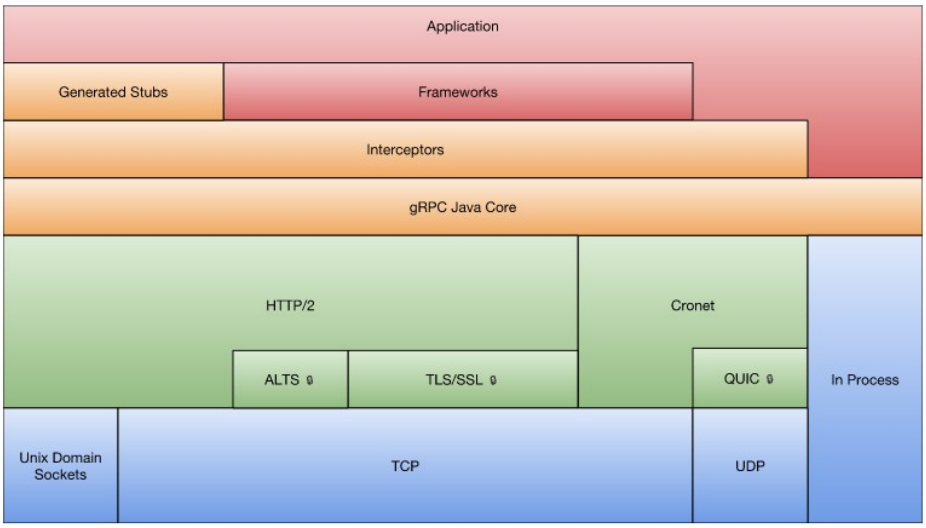
Toto volanie je obalené knižnicou jazyka danej aplikácie a následne preložené do C-Core. C-Core volanie je ďalej prenesené transportnými protokolmi (zelená a modrá farba). Defaultne využíva gRPC komunikáciu pomocou HTTP/2 v kombinácii s TLS/SSL a TCP, avšak gRPC umožňuje nastaviť konkrétne transportné protokoly, ktoré sa majú využívať [9]. Komunikáciu aplikácie písanej v jazyku Go môžeme vidieť na obrázku č. 25 a pre jazyk Java na obrázku č. 26.



Obrázok 24 C-Core základ komunikácie [9].



Obrázok 25 Go základ komunikácie [9].



Obrázok 26 Java základ komunikácie [9].

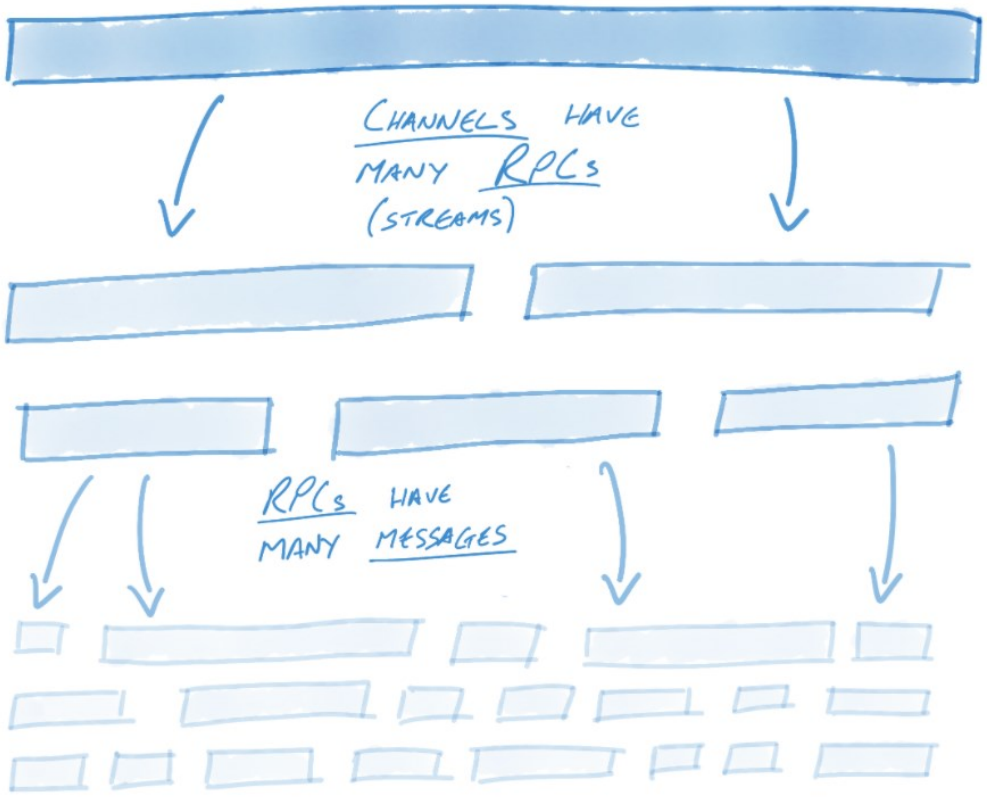
2.5 Streaming

gRPC plne využíva možnosti protokolu HTTP/2 a umožňuje streamovanie. Na rozdiel od RESTu, ktorý podporuje iba model request-response dostupný v HTTP/1. x. sa jedná o zásadnú výhodu. REST je zo svojej podstaty bezstavový a bez kontextu klienta nad rámec jeho requestu nie je preto streamovanie možné [7]. gRPC ponúka 4 spôsoby komunikácie a to unárne RPC, server streaming, client streaming a bidirectional streaming [10]

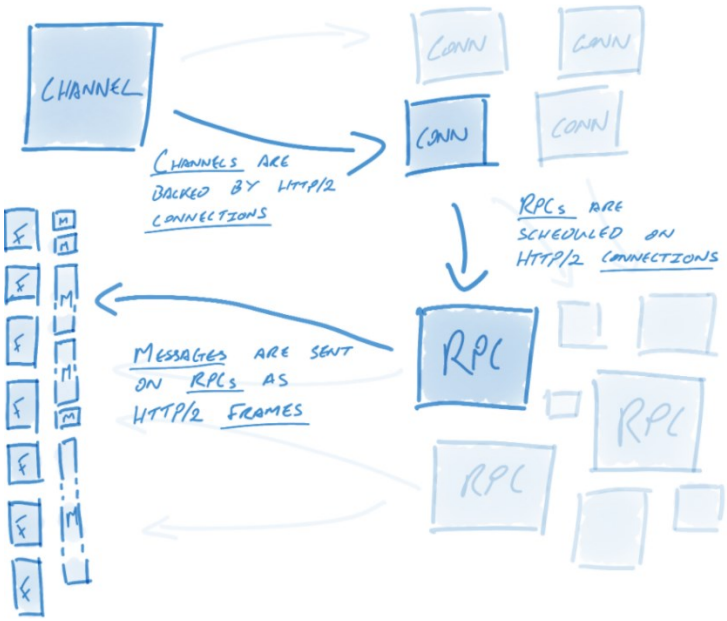
Protokol HTTP/2 je plne multiplexovaný, čo znamená, že umožňuje odosielať viacero požiadaviek súbežne prostredníctvom jedného pripojenia TCP. Vďaka tomu sú aplikácie, ktoré využívajú HTTP/2 rýchlejšie, jednoduchšie a robustnejšie. HTTP/2 poskytuje základ pre dlhodobé komunikačné toky v reálnom čase [14].

Pod pojmom stream je možné rozumieť tok bajtov v rámci nadviazaného spojenia. Každý stream môže obsahovať jednu alebo viacej správ. Správa je kompletná sekvencia rámcov, ktoré sa mapujú na logickú správu HTTP, ktorá sa skladá z jedného alebo viacerých rámcov [14]. Rámec je najmenšia komunikačná jednotka v rámci HTTP/2. Každý rámec obsahuje hlavičku, ktorá identifikuje stream, do ktorého rámec patrí [10].

V gRPC je kľúčovým konceptom kanál (channel). Streamy v protokole HTTP/2 umožňujú viacero súbežných konverzácií na jednom pripojení, kanál rozširuje tento koncept tým, že umožňuje viacero streamov cez viacero súbežných pripojení [9]. Kanál predstavuje virtuálne pripojenie ku koncovému bodu, ktoré v skutočnosti môžu byť podporené viacerými pripojeniami HTTP/2 [10]. Vzťah kanálov, pripojení, správ a rámcom je možné vidieť na obrázku č. 27 a na obrázku č. 28.



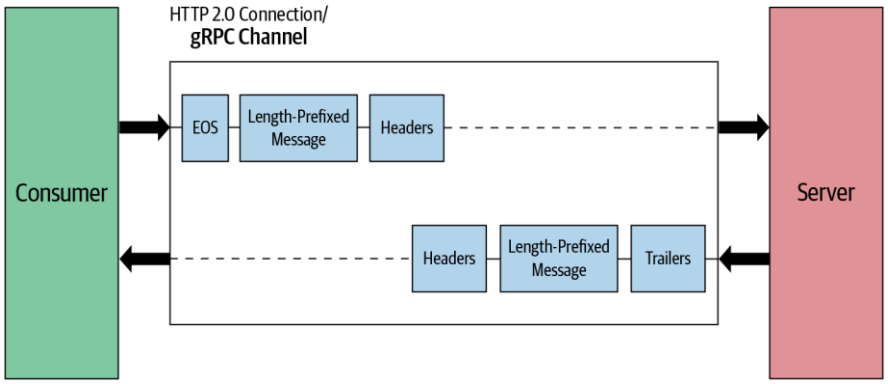
Obrázok 27 Vzťah konceptu gRPC a HTTP/2 [9].



Obrázok 28 Vzťah konceptu gRPC a HTTP/2 [9].

2.5.1 Unárne RPC

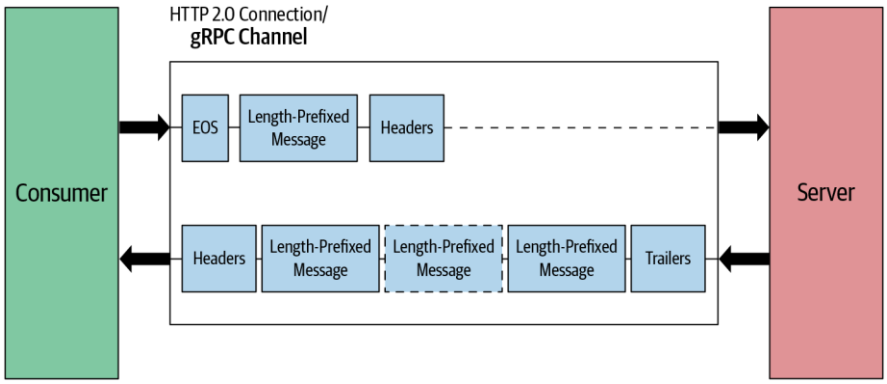
Unárne alebo jednoduché RPC je najjednoduchší typ komunikácie. Jedná sa o typický model, pri ktorom klient pošle jednu požiadavku serveru, od ktorého následne dostane späť jednu odpoveď [9]. Požiadavka na server obsahuje hlavičku, za ktorou nasleduje správa, ktorá sa môže skladať z jedného alebo viacerých rámcov [10]. Na konci je pridaný príznak o konci toku (EOS). Týmto príznakom je ukončené odosielanie požiadavky klienta na server. Po prijatí celej správy server vytvorí odpoveď, ktorú pošle klientovi. Táto odpoveď rovnako obsahuje úvodnú hlavičku, samotnú správu a koncovú hlavičku so stavovými podrobnosťami.



Obrázok 29 Unárne RPC [10].

2.5.2 Server streaming

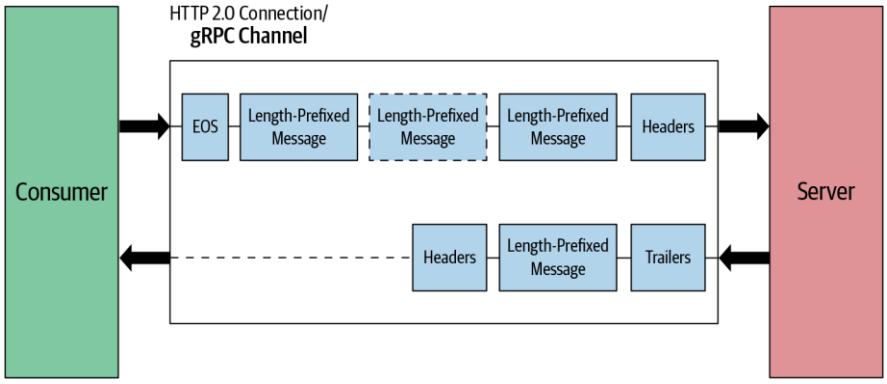
Z pohľadu klienta má unárne RPC rovnaký tok požiadavky ako pri server streamingu. V obidvoch prípadoch je posielaná požiadavka s jednou správou. Hlavný rozdiel je na strane servera, ktorý po spracovaní požiadavky umožňuje odoslať klientovi viacero správ [10].



Obrázok 30 Server streaming [10].

2.5.3 Client streaming

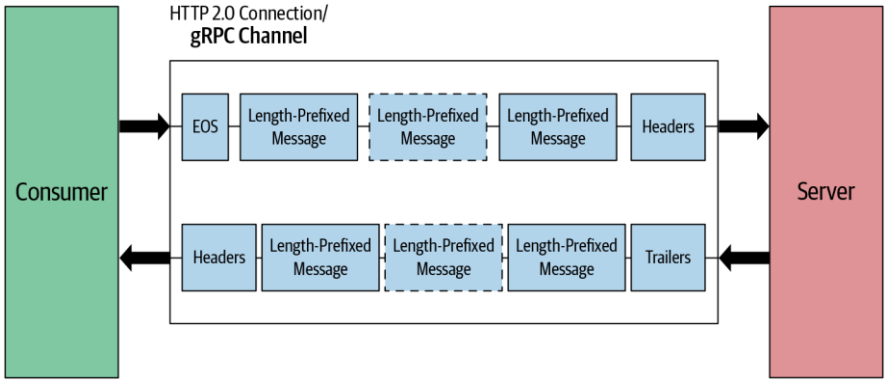
Client streaming je opak server streamingu. Klient odosiela požiadavku, ktorá sa skladá z viacerých správ. Server po prijatí všetkých správ, pošle späť klientovi odpoveď, ktorá obsahuje len jednu správu [10].



Obrázok 31 Client streaming [10].

2.5.4 Bidirectional streaming

Pri bidirectional, teda obojsmernom, streamingu dokáže klient aj server posielať viacero správ. Obidva streamy sú nezávislé, to znamená, že klient a server môžu prijímať a posielať správy v ľubovoľnom poradí, bez toho aby čakali na dokončenie posielania druhej strany [10].



Obrázok 32 Bidirectional streaming [10].

2.6 Pluginy

gRPC ponúka množstvo pripojiteľných pluginov, ktoré slúžia ako výrazné rozšírenie funkcionality. Je možné nájsť pluginy, ktoré poskytujú napríklad autentifikáciu, šifrovanie, deadlines, timeouty, výmenu metadát, kompresiu, vyvažovanie záťaže, monitorovanie alebo logovanie [9]. Bližšie budú stručne popísané pluginy gRPC-Web a gRPC-Gateway, ktoré budú použité v praktickej časti.

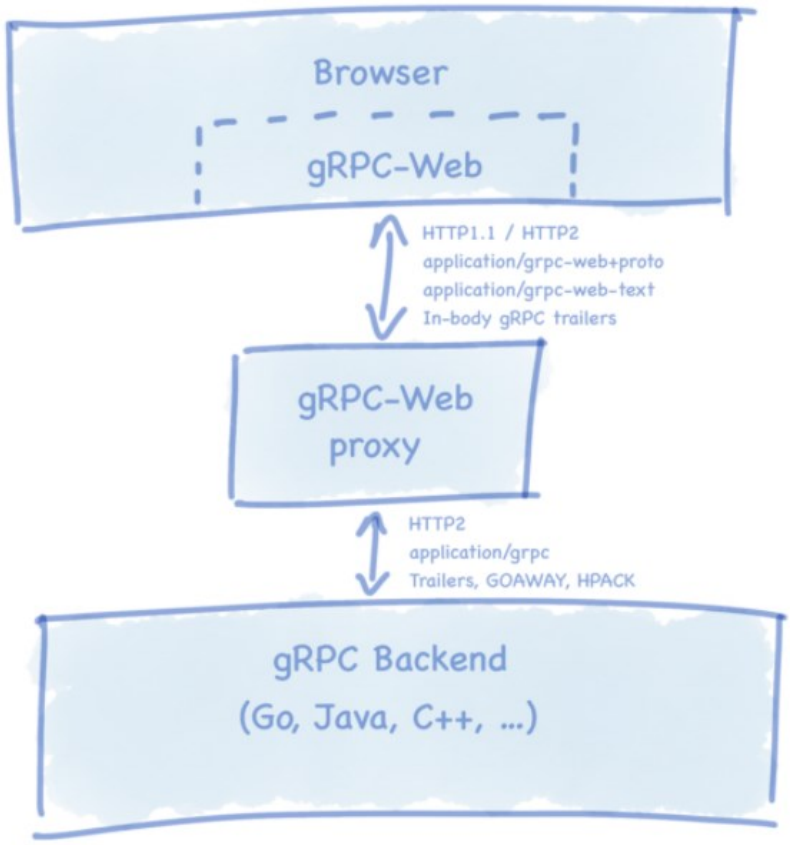
2.6.1 gRPC-Web

gRPC používa pre komunikáciu protokol HTTP/2, ktorý komplikuje využitie gRPC pri webových aplikáciách. V súčasnosti žiadny moderný internetový prehliadač nemá prístup ku HTTP/2 rámcom, a preto nie je umožnená priama komunikácia internetového prehliadača s gRPC [15].

gRPC-Web vytvára proxy server, ktorý slúži na preklad HTTP/1.x požiadaviek zo strany webového klienta na požiadavky HTTP/2 serveru [15]. Tým pádom webový klient, ktorý využíva JavaScript alebo Blazor, je schopný komunikovať pomocou gRPC [16]. Kód klienta a serveru je priamo vytvorený z proto súborov a rovnako sa správy posielajú v binárnom

protobuf formáte. Pri komunikácii s internetovými prehliadačmi je možné využívať len unárne RPC alebo serverové streamovanie, klientské a obojsmerné streamovanie nie je možné [16].

Pre správne fungovanie gRPC-Web musí byť povolené CORS (Cross-origin resource sharing) a musí byť správne nakonfigurovaný gRPC server [16]. CORS je mechanizmus umožňujúci zdieľanie zdrojov webovej stránky pre aplikáciu inej domény.

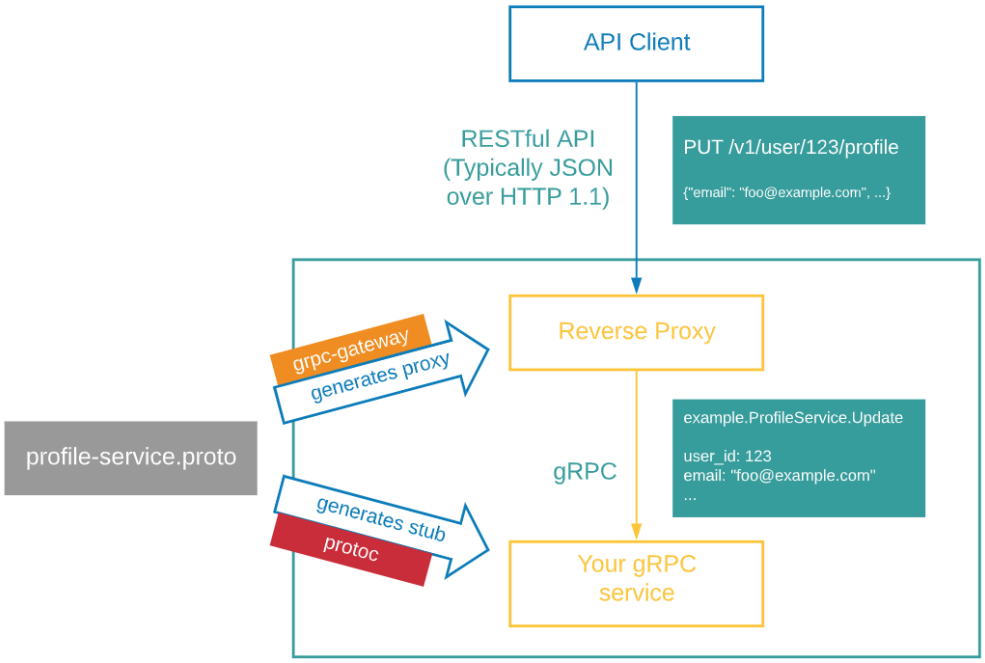


Obrázok 33 gRPC-Web [9].

2.6.2 gRPC-Gateway

gRPC-Gateway je plugin, ktorý umožňuje komunikovať REST klientom s gRPC servermi. Toto použitie je veľmi výhodné, pokiaľ je treba zachovať spätnú kompatibilitu so stávajúcou REST implementáciou alebo pre podporu jazykov a klientov, ktorí nie sú dobre podporovaní v gRPC [17]. gRPC-Gateway rovnako poskytuje čiastočnú alternatívu ku gRPC-Web, pretože internetové prehliadače nemajú problém s komunikáciou pomocou REST.

V definiící v proto súbore je nutné pridať RPC metódam mapovanie pre REST volania, pre každú metódu je nutné určiť jej URL cestu a parametre, ktoré bude využívať REST. Z proto súboru je následne vygenerovaný kód pre stranu serveru a vygenerovaný reverzný proxy server [17]. Tento proxy server slúži pre prijímanie REST requestov, prekladá JSON správy na protocol buffers správy, ktoré ďalej predá gRPC serveru.



Obrázok 34 gRPC-Gateway [17].

2.7 Bezpečnosť

Aplikácie založené na gRPC komunikujú navzájom po sieti. Aplikácia vystavuje svoje vstupné endpointy ostatným, ktorí s ňou potrebujú komunikovať. Pre bezpečnú komunikáciu medzi klientom a serverom je preto dôležité zabezpečiť aspoň minimálne bezpečnostné požiadavky. Každá aplikácia musí byť schopná spracovávať šifrované správy, šifrovať celú komunikáciu a autentifikovať všetky posielané správy [10].

V gRPC existujú dva typy poverení (credential), kanálové (channel) a volania metódy (call). Pod pojmom credential je možné rozumieť certifikát, privátny kľúč alebo iné dáta požadované systémom pre overenie práva prístupu k informáciám alebo prostriedkom systému. Pre zvýšenú bezpečnosť je možné kombinovať tieto dva typy poverení a využívať ich súčasne [9, 10].

Pri kanálovom overení sú dáta spojené s kanálom a sú overené pri nadviazaní spojenia. Jedná sa typicky o overenie pomocou SSL/TLS, prípadne ALTS [9]. Môže sa jednať o jednosmerné overenie, pri ktorom klient overí server, aby sa zabezpečilo, že prijíma údaje zo zamýšľaného serveru [10].

Pri vytvorení spojenia server zdieľa svoj verejný certifikát s klientom, ktorý validuje prijatý certifikát. Druhou možnosťou je obojsmerné kanálové overenie, ktorého hlavným zámerom je kontrola serveru nad klientmi, ktorí sa k nemu pripájajú. TLS server je nakonfigurovaný na prijímanie pripojení od obmedzenej skupiny overených klientov. Obidve strany navzájom zdieľajú svoje verejné certifikáty a overujú druhú stranu [10].

Druhým spôsobom autentifikácie dát je serverové overovanie klienta pri každom jeho volaní RPC metódy. Obvykle sa jedná o overenie pomocou vygenerovaného tokenu, pri ktorom je možné určiť, ako dlho je platný [10].

gRPC podporuje mechanizmy ako Basic Authentication, OAuth 2.0, JWT, Bearer, Google token-based authentication, Azure Active directory, Client Certificate, IdentitiyServer, OpenID Connect, WS-Federation [10, 16]. gRPC je navrhnuté na prácu s rôznymi druhmi autentifikačných mechanizmov a je schopné rozšíriť svoju podporu pridaním vlastného autentifikačného mechanizmu [9]. Vďaka tomu je použitie gRPC pre komunikáciu s inými systémami jednoduché a bezpečné.

2.8 Použitie

Spoločnosť Google, ktorá stála za vznikom gRPC, používa gRPC a jeho predošlé neverejné verzie vo svojich systémoch dlhé roky. Prvá verejná verzia bola sprístupnená v roku 2015 a od tej doby našla využitie v spoločnostiach ako Microsoft, Netflix, Spotify, Salesforce, Slack, Nokia, Mux, Nulab, VSCO, uSwitch, PingCAP, Qihoo360, The New York Times, OpenAI, Northwestern Mutual, NetEase, Intuit, HubSpot, Apester, AppDirect, Capital One a v mnoho ďalších [18].

gRPC poskytuje v istej forme alternatívu ku technológiám ako je SOAP a REST a jeho využitie je praktické v prípadoch ako [9, 16, 19]:

- mikroservisy,
- mobilný klient komunikujúci s cloudovým serverom,
- vysoko škálovateľné distribuované systémy s nízkou latenciou,
- komunikácia v reálnom čase, systémy s duplexným streamovaním,

- medziprocesová komunikácia (IPC),
- systémy s obmedzenými sieťovými prostredkami, internet vecí,
- cloudové systémy,
- viacjazyčné prostredia a systémy,
- architektúra klient-server.

2.9 Podpora

Framework gRPC podporuje viaceré bežne používané programovacie jazyky. Podporuje aj jazyky pre vývoj mobilných aplikácii. Prehľad oficiálne podporovaných jazykov, operačných systémov, kompilátorov a SDK je možné vidieť v tabuľke č. 1. Existujú aj implementácie pre jazyky Flutter a Swift, pre ktoré sa plánuje zaradenie do oficiálnej podpory [9].

Tabuľka 2 Oficiálna podpora gRPC [9].

Language	OS	Compilers / SDK
C/C++	Linux, Mac	GCC 4.9+, Clang 3.4+
C/C++	Windows 7+	Visual Studio 2015+
C#	Linux, Mac	.NET Core, Mono 4+
C#	Windows 7+	.NET Core, NET 4.5+
Dart	Windows, Linux, Mac	Dart 2.2+
Go	Windows, Linux, Mac	Go 1.13+
Java	Windows, Linux, Mac	JDK 8 recommended (Jelly Bean+ for Android)
Kotlin	Windows, Linux, Mac	Kotlin 1.3+
Node.js	Windows, Linux, Mac	Node v8+
Objective-C	macOS 10.10+, iOS 9.0+	Xcode 7.2+
PHP	Linux, Mac	PHP 7.0+
Python	Windows, Linux, Mac	Python 3.5+
Ruby	Windows, Linux, Mac	Ruby 2.3+

2.10 Výhody

Výhody, ktoré framework gRPC prináša, sú kľúčové pre jeho ďalší rast a jeho všeobecné prijatie. Medzi hlavné výhody gRPC patrí [9, 10, 19, 20]:

- Rýchlosť – Protocol buffer predstavuje efektívny a rýchly dátový formát. Vo viacerých prípadoch je prenos pomocou protocol buffer niekoľko násobne krát rýchlejší ako riešenie REST a JSON alebo SOAP a XML.
- Modernosť – Moderný framework navrhnutý s ohľadom na aktuálne potreby vývojárov.
- Typovosť – Protocol buffer je silno typový formát, ktorý jasne definuje dátové typy používaných správ.
- Podpora programovacích jazykov – gRPC podporuje viacero programovacích jazykov.
- Podpora dátových formátov – Hlavný formát, ktorý gRPC využíva je protocol buffer, ale zároveň umožňuje prácu s formátmi ako sú napríklad JSON, XML a Thrift.
- Duplexné streamovanie – Natívna podpora pre duplexné streamovanie, ktoré nebolo možné napríklad pri prístupe REST alebo SOAP.
- Rozšírenia – Vstavaná podpora pre autentifikáciu, šifrovanie, deadlines, timeouty, výmenu metadát, kompresiu, vyvažovanie záťaže, monitorovanie, logovanie atď.
- Cloud – gRPC je integrovaný do cloudových natívnych ekosystémov. Viaceré projekty v rámci CNFC, ako napríklad Envoy, podporujú gRPC ako komunikačný protokol.
- Prijatie – gRPC je prijaté a používané veľkými technologickými spoločnosťami a nemusí sa obávať o svoj nasledujúci vývoj a smerovanie.
- RPC prístup – Návrat ku funkcionálnemu prístupu API, ktorý vychádza z RPC. REST prístup ku zdrojom nemusí byť pre niekoho populárny.
- Znížená latencia – gRPC je založený na protokole HTTP/2, ktorý umožňuje rýchlejšie a dlhodobejšie pripojenia.
- Jednoduchosť – Rýchla, jednoduchá inštalácia a práca s gRPC.
- Generovanie kódu – Automaticky generovaný kód pre stranu servera aj klienta.
- Dokumentácia – Dostupná rozsiahla dokumentácia, podrobné quickstarty, tutoriály a ukážky kódov pre všetky dostupné programovacie jazyky.

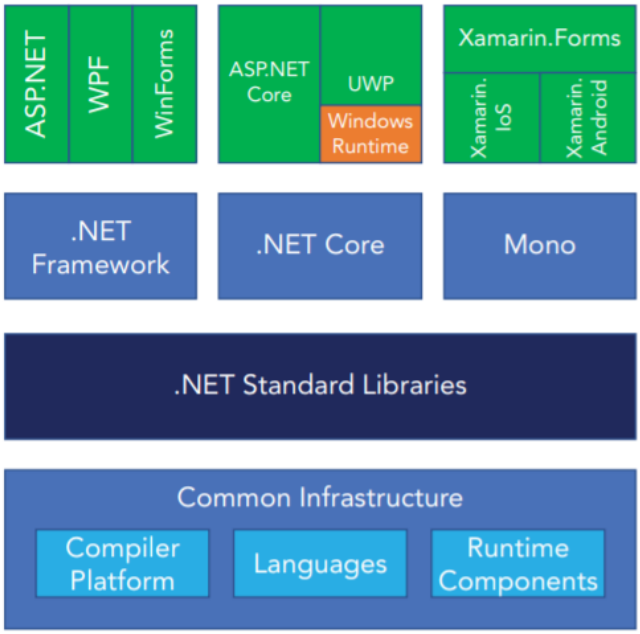
2.11 Nevýhody

Vývoj s frameworkom gRPC obnáša aj určité nevýhody, s ktorými je nutné počítať. Medzi nevýhody patrí [9, 10, 19, 20]:

- Webové aplikácie – gRPC využíva protokol HTTP/2, ktorý internetovým prehliadačom neposkytuje prístup ku svojim rámcom, a preto nie je možná priama komunikácia webového klienta a gRPC serveru, je nutné používať proxy servery.
- Rozšírenia – gRPC ponúka pre rozšírenie funkcionality množstvo pluginov, bez ktorých sa niekedy nedá zaobísť a ich potrebu je nutné brať na vedomie.
- Nečitateľnosť prenosu – Protocol buffer je menej čitateľný formát pre človeka.
- Dátové typy – Protocol buffer má obmedzenú škálu dátových typov.
- Externé služby – Silne typová povaha a definícia služby gRPC môže brániť flexibilite, ktorá je poskytovaná externým službám. Externý spotrebitelia nemusia mať rovnako povedomie a skúsenosti s gRPC.
- Kontrakt služby – Pri veľkých zlomových zmenách v definícii služby je nutná regenerácia kódu pre server a aj pre stranu klienta. Tento proces je potrebné začleniť do existujúceho procesu nepretržitej integrácie a môže komplikovať celkový životný cyklus vývoja.
- Nezrelosť – V porovnaní s viacej zaužívaným RESTom je technológia gRPC relatívne menej zrelá a ponúka menší ekosystém.

3 GRPC V PROSTREDÍ .NET

.NET je bezplatná, multiplatformová, open source vývojová platforma od spoločnosti Microsoft so širokým uplatnením pri tvorbe webových, mobilných a desktop aplikácii, pri docker mikroservisách, vývoji hier, machine learningu, cloude alebo pri internete vecí. Umožňuje používať programovacie jazyky ako sú C#, Visual Basic. NET alebo F# [21]. .NET platforma (obrázok č. 35) sa skladá z troch hlavných implementácii a to .NET Framework, .NET Core a Mono [21]. Pokračovaním .NET Core je .NET 5 vydaný v novembri 2020.



Obrázok 35 Prehľad .NET platformy [21].

3.1 Podpora v .NET

Medzi podporované jazyky gRPC patrí od roku 2016 jazyk C#. V začiatkoch gRPC neexistovala spoľahlivá použiteľná C# knižnica pre prácu s HTTP/2. gRPC sa preto rozhodlo postaviť C# implementáciu na svojej natívnej knižnici založenej na jadre C-Core. Využitie natívnej knižnice znížilo čas na vývoj a vznikla stabilná a výkonná C# implementácia, označovaná podľa názvu nuget balíku ako Grpc.Core [9].

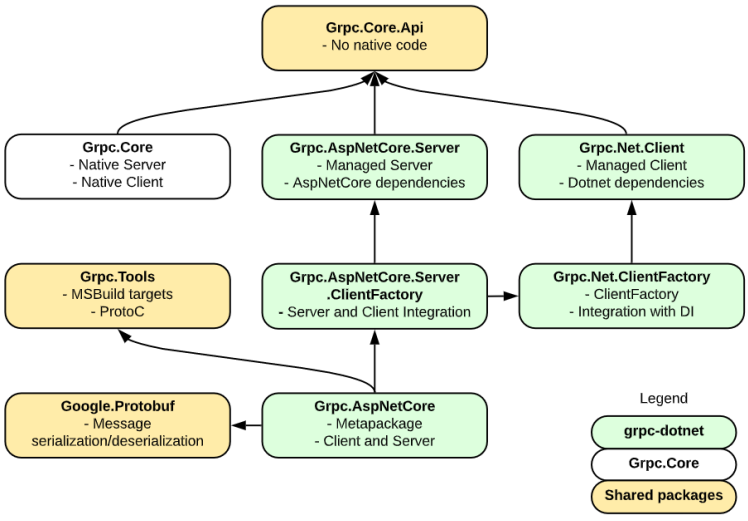
V novembri 2018 začal spolupracovať Microsoft .NET vývojársky tím s vývojármi gRPC a v septembri 2019 predstavili implementáciu označenú ako grpc-dotnet. Jednalo sa o novú implementáciu napísanú celú v C# bez natívnych závislostí založenú na novo vydanom .NET Core 3.0, ktorý umožňuje prácu s protokolom HTTP/2 [9].

Na začiatku koexistovali obidve implementácie vedľa seba. Vyplývalo to zo skutočnosti, že samotný .NET Core bola novinka a predovšetkým z dôvodu, že grpc-dotnet nebol rovnako zrelý ako stabilná implementácia Grpc.Core [9]. Časom sa stal .NET Core populárny a samotný grpc-dotnet priniesol nové funkcie.

Čistá C# implementácia grpc-dotnet sa ukázala z dlhodobého hľadiska ako modernejšia implementácia s väčším potenciálom, ktorá je dobre implementovaná do moderných verzií .NET a vhodnejšia pre C# komunitu. Zároveň sa znižoval prínos natívnej implementácie Grpc.Core a preto sa gRPC rozhodlo, aby nemuselo udržovať a niesť náklady pre vývoj dvoch implementácií postupné ukončenie podpory implementácie Grpc.Core [9].

Ukončenie podpory spočíva v dvoch fázach. Prvá fáza začala v máji 2021 a spočíva v režime údržby. S novými verziami Grpc.Core nebudú poskytované nové funkcionality a vylepšenia, budú riešené len chyby a problémy so zabezpečením. Prvá fáza potrvá jeden rok a v máji 2022 prejde Grpc.Core do druhej fáze, kedy sa stane zastaranou implementáciou (deprecated), ktorá nebude už oficiálne podporovaná a prestanú byť poskytované opravy chýb [9].

Viaceré nuget balíky sú využívané obidvomi implementáciami Grpc.Core a grpc-dotnet. Tieto zdieľané balíky (žltá farba na obrázku č. 36) sú nezávisle na natívnych komponentov [9]. Pri prechode z Grpc.Core na grpc-dotnet budú potrebné len minimálne zmeny v konfigurácii gRPC kanálov a serverov, ktoré sú obvykle oddelené od aplikačnej logiky.



Obrázok 36 Prehľad základných gRPC .NET nugetov [9].

3.2 Požiadavky v .NET

Existujú rozdielne požiadavky pre používanie gRPC serveru a gRPC klienta v prostredí .NET.

3.2.1 gRPC server

gRPC server je možné používať na všetkých operačných systémoch, ktoré podporujú .NET Core. Jedná sa teda o najrozšírenejšie operačné systémy Windows, Linux a macOS. Operačné systémy Windows a Linux umožňujú používať protokol HTTPS/2, operačný systém macOS kvôli chýbajúcej podpore ALPN umožňuje pri gRPC len použitie HTTP/2 bez TLS. Rovnako prácu s gRPC podporujú všetky .NET Core vstavané servery, ako sú napríklad Kester, TestServer, IIS a HTTP.sys [16].

Hosting gRPC serveru je možný pre prostredie .NET [16].

- .NET 5 alebo novšia verzia,
- .NET Core 3 alebo novšia verzia.

3.2.2 gRPC klient

Požiadavky na gRPC klienta je možné vidieť v tabuľke č. 3. Plnú podporu gRPC volaní pomocou protokolu HTTP/2 umožňuje len .NET 5 a .NET Core 3, respektíve ich novšie verzie. .NET framework pre prácu s HTTP/2 potrebuje konfiguráciu WinHttpHandler a minimálnu verziu Windows 10 Build 19622, čo môže vyžadovať použitie Windows Insider. Pri ostatných .NET implementáciách je možné využiť plugin gRPC-Web [16].

Tabuľka 3 gRPC klient požiadavky [16].

.NET implementation	gRPC over HTTP/2	gRPC-Web
.NET 5 or later	✔	✔
.NET Core 3	✔	✔
.NET Core 2.1	✘	✔
.NET Framework 4.6.1	⚠[†]	✔
Blazor WebAssembly	✘	✔
Mono 5.4	✘	✔
Xamarin.iOS 10.14	✘	✔
Xamarin.Android 8.0	✘	✔
Universal Windows Platform 10.0.16299	✘	✔
Unity 2018.1	✘	✔

3.3 Blazor

Blazor je bezplatný, multiplatformový, open source UI framework od spoločnosti Microsoft. Slúži pre vytváranie interaktívnych webových užívateľských rozhraní na strane klienta za pomoci jazyka C# namiesto JavaScriptu [23]. Umožňuje zdieľať aplikačnú logiku na strane serveru a klienta a je založený na komponentoch. Komponenty sú znovupoužiteľné bloky kódu, ktoré sú tvorené pomocou jazyka C# a Razor.

Blazor podporuje dva formy hostingu a to Blazor WebAssembly a Blazor server. Blazor WebAssembly slúži pre vytváranie single-page aplikácii (SPA). Spúšťanie .NET kódu vo webových prehliadačoch umožňuje WebAssembly [23]. Do webového prehliadača klienta sa stiahne Blazor aplikácia spolu s .NET runtimom. WebAssembly je binárny formát inštrukcií optimalizovaný pre rýchle sťahovanie aplikácie a efektívne vykonávanie kódu [24, 25]. Pri hostingu pomocou Blazor servera aplikácia nebeží vo webovom prehliadači, ale priamo na serveri. Komunikácia a aktualizácia užívateľského rozhrania vo webovom prehliadači sa vykonáva pomocou SignalR pripojenia so serverom [23].

Ako vidíme v tabuľke č. 3, Blazor neumožňuje priamo prácu s gRPC a protokolom HTTP/2 a pre prácu s gRPC je nutné využiť plugin gRPC-Web.

II. PRAKTICKÁ ČASŤ

4 NÁVRH APLIKÁCIE S VYUŽITÍM GRPC

Framework gRPC má široký potenciál využitia. Vzorová aplikácia demonštruje základné kľúčové prvky komunikácie pomocou gRPC založenej na architektúre klient-server. Navrhnuté riešenie sa delí na dve časti a to z webovej aplikácie a zo serverovej časti.

Webová aplikácia predstavuje CRM systém pre realitných maklérov. Navrhnutý CRM systém slúži pre jednoduchú správu zákaziek. Je možné zobraziť prehľad zákaziek, vytvoriť, editovať a zmazať zákazku. Realitný CRM systém by bol v skutočnosti omnoho rozsiahlejší a komplexnejší. Účelom tejto aplikácie nie je vybudovať kompletný CRM systém, ale len ukázať základnú prácu pomocou gRPC a porovnať ju s REST riešením.

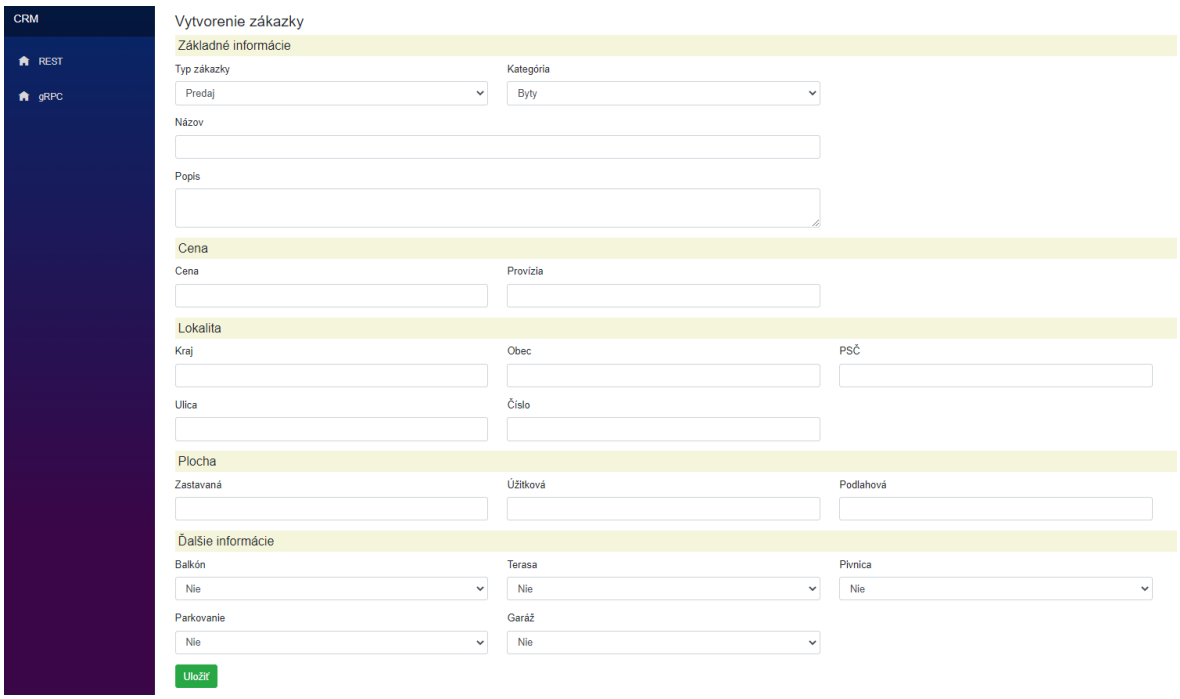
Webová aplikácia je rozdelená na dve časti – gRPC a REST. Obidve časti vyzerajú vizuálne rovnako, rozdiel spočíva v princípe komunikácie so serverom. Všetky údaje o zákazkách sú poslané a uložené na serveri. Každá operácia so zákazkou vytvára request na server. Serverová časť pozostáva z gRPC servisy a REST API. Request vytvorený z gRPC webovej časti je smerovaný na gRPC servisu, request vytvorený z REST webovej časti je smerovaný na REST API.

Na prechod medzi týmito dvomi časťami vo webovej aplikácii slúži bočná navigácia. Domovskú stránku predstavuje prehľad zákaziek. Jedná sa o tabuľku zákaziek, pri každom zázname tabuľky sú zobrazené základné údaje o zákazke a tri tlačidlá, ktoré slúžia pre zobrazenie detailu zákazky, editáciu a zmazanie zákazky. Nad tabuľkou zákaziek sa nachádza tlačidlo pre vytvorenie zákazky.

Vytvorenie, detail a editácia zákazky využíva rovnaký formulár zákazky. Tlačidlo pre vytvorenie presmeruje na prázdny formulár zákazky, tlačidlo pre detail a editáciu presmerujú na predvyplnený formulár podľa predošlých uložených hodnôt zákazky. Formulár otvorený z detailu zákazky je needitovateľný, formulár otvorený z editácie zákazky je editovateľný.

Základný prehľad zákaziek je zobrazený na obrázku č. 37 a formulár zákazky na obrázku č. 38.

Obrázok 37 Prehľad zákaziek.



Obrázok 38 Formulár zákazky.

5 IMPLEMENTÁCIA APLIKÁCIE

Pre implementáciu aplikácie je použité vývojové prostredie Visual Studio 2019. Aplikácia využíva jazyk C# a je tvorená v prostredí .NET Core 3.1. Ako bolo spomenuté v kapitole o podpore gRPC v prostredí .NET, existujú dve verzie gRPC pre .NET a to Grpc.Core a grpc-dotnet. Aplikácia bude pracovať s modernejšou verziou grpc-dotnet. Webová aplikácia, ktorá slúži ako klient a serverová časť sú oddelené ako samostatné solutions, Server a Client Solution.

5.1 Server

Server Solution je rozdelené na tri samostatné projekty a to na Data, gRPC a REST.

5.1.1 Data

Ukladanie a správu dát má na starosti projekt Data. Projekt je vytvorený ako šablóna class library a poskytuje in-memory databázu, ktorá je dostačujúca pre navrhnutú aplikáciu. In-memory databáza ukladá dáta do RAM pamäte počítača, databáza tak nie je perzistentná a dáta sú uložené len po dobu spustenia serveru. Je využitá In-memory Entity Framework Core databáza, ktorá je dostupná po nainštalovaní nuget balíku Microsoft.EntityFrameworkCore.InMemory. Jedná sa o užitočný nástroj, ktorý umožňuje prácu bez inštalácie databázy a bez réžie skutočných databázových operácii. Slúži výhradne pre testovacie účely, nejedná sa o klasickú relačnú databázu.

Samotný projekt Data pozostáva zo zložky Models, ktorá obsahuje model zákazky, databázový kontext a DTO (Data Transfer Object) modely. DTO modely slúžia pre prenos dát medzi procesmi. Jedná sa o dátovú štruktúru, ktorá predstavuje medzivrstvu medzi modelom databáze a modelom aplikačnej vrstvy. Obvykle nie je žiaduce, aby všetky property databázového modelu boli prístupné a prenášané po sieti. Ukážku databázového kontextu, súbor PropertyContext, ktorý pozostáva zo zákaziek je možné vidieť na ukážke kódu č. 1.

```
public class PropertyContext : DbContext
{
        public PropertyContext(DbContextOptions<PropertyContext> options) : base(options)
        {

        }

        public DbSet<Property> Properties { get; set; }
}
```

Ukážka kódu 1 Databázový kontext.

5.1.2 REST

Projekt REST server je vytvorený ako prázdna ASP.NET Core Web Application šablóna. Základ celého serveru predstavuje Property Controller, ktorý obsahuje HTTP metódy, ktoré slúžia ako endpointy pre CRUD operácie so zákazkami.

Daný Property Controller je označený route atribútom. Route atribút je nepovinný, ale jeho použitie zvyšuje kontrolu nad URI vo vytvorenom webovom API. Routing (smerovanie) je spôsob, akým webové rozhranie API priraďuje URI ku controllerom a metódam. Property controller s použitým route atribútom a konštruktorom je možné vidieť na ukážke kódu č. 2, metódy controlleru budú dostupné pod prefixom „api/property“.

Pri Property Controlleri využijeme návrhový vzor dependency injection (vkladanie závislosti). Dependency injection slúži pre zníženie závislosti medzi jednotlivými časťami systému. Je využitý constructor injection, pri ktorom sú definované závislosti controlleru priamo v konštruktore. Controller bude využívať databázový kontext pre prácu s databázou a property map servisu, ktorá bude slúžiť pre mapovanie DTO a DB modelov.

Pre správu závislostí je použitý IoC kontajner Autofac. Autofac je dostupný po nainštalovaní nuget balíkov Autofac a Autofac.Extensions.DependencyInjection. Jeho použitie je nutné registrovať v súbore Program.cs (ukážka kódu č.3). DB kontext, použité servisy a routovanie controllerov je potrebné zaregistrovať v súbore Startup.cs (ukážka kódu č. 4).

```
[ApiController]
[Route("api/[controller]")]
public class PropertyController : ControllerBase
{
    private readonly Data.Models.PropertyContext _context;
    private readonly Services.IPropertyMapService _propertyMapService;

    public PropertyController(
       Data.Models.PropertyContext context,
       Services.IPropertyMapService propertyMapService
    )
    {
       _context = context;
       _propertyMapService = propertyMapService;
    }
```

Ukážka kódu 2 PropertyController.

```
public class Program
{
        public static void Main(string[] args)
        {
                CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
           Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                        webBuilder.UseStartup<Startup>();
                })
                .UseServiceProviderFactory(new AutofacServiceProviderFactory());
}
```

Ukážka kódu 3 Program.cs.

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<Data.Models.PropertyContext>(p =>
           p.UseInMemoryDatabase("Properties")
        );
        services.AddControllers();
    }

    public void ConfigureContainer(ContainerBuilder builder)
    {
        builder.RegisterType<Services.PropertyMapService>()
        .As<Services.IPropertyMapService>()
        .InstancePerLifetimeScope();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

         app.UseHttpsRedirection();
         app.UseRouting();
         app.UseAuthorization();

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

Ukážka kódu 4 Startup.cs

Samotný Property Controller obsahuje 5 metód, ktoré sú dostupné ako endpointy REST API. Každá metóda je označená atribútom HttpMethodAttribute, ktorý určuje, o ktorý typ HTTP metódy sa jedná a zároveň umožňuje definovať názov metódy pri routovaní. Atribút môže obsahovať priamo v názve metódy aj použité parametre, ak sú posielané v hlavičke requestu ako to je napríklad pri metódach typu HttpGet a HttpDelete. Controller obsahuje metódy:

- PropertyCreate (ukážka kódu č. 5) – metóda typu HttpPost, vytvorenie zákazky,
- PropertyDetail (ukážka kódu č. 6) – metóda typu HttpGet, detail zákazky,
- PropertyUpdate (ukážka kódu č. 7) – metóda typu HttpPut, editovanie zákazky,
- PropertyDelete (ukážka kódu č. 8) – metóda typu HttpDelete, zmazanie zákazky,
- Properties (ukážka kódu č. 9) – metóda typu HttpGet, detail zákaziek.

Každá metóda spočíva v jednoduchej práci s databázovým kontextom a mapovaním databázových a DTO modelov. REST pristupuje za použitia HTTP metód ku zdrojom, ktoré sú identifikovateľné pomocou URI a následne nad týmito zdrojmi vykonáva dané operácie.

```
[HttpPost("PropertyCreate")]
public async Task<ActionResult<int>> PropertyCreate(PropertyDTO propertyDTO)
{
    var property = _propertyMapService.GetProperty(propertyDTO);

    _context.Properties.Add(property);

    await _context.SaveChangesAsync();

    return property.PropertyID;
}
```

Ukážka kódu 5 Metóda PropertyCreate.

```
[HttpGet("PropertyDetail/{id}")]
public async Task<ActionResult<PropertyDTO>> PropertyDetail(int id)
{
    var property = await _context.Properties.FindAsync(id);

    if (property == null)
    {
        return NotFound();
    }

    return _propertyMapService.GetPropertyDTO(property);
}
```

Ukážka kódu 6 Metóda PropertyDetail.

```
[HttpPut("PropertyUpdate")]
public async Task<IActionResult> PropertyUpdate(PropertyDTO propertyDTO)
{
    var property = await _context.Properties.FindAsync(propertyDTO.PropertyID);

    if (property == null)
    {
        return NotFound();
    }

    _propertyMapService.MapProperty(property, propertyDTO);

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    when (!PropertyExists(property.PropertyID.Value))
    {
        return NotFound();
    }

    return NoContent();
}
```

Ukážka kódu 7 Metóda PropertyUpdate.

```
[HttpDelete("PropertyDelete/{id}")]
public async Task<IActionResult> PropertyDelete(int id)
{
    var property = await _context.Properties.FindAsync(id);

    if (property == null)
    {
        return NotFound();
    }

    _context.Properties.Remove(property);

    await _context.SaveChangesAsync();

    return NoContent();
}
```

Ukážka kódu 8 Metóda PropertyDelete.

```
[HttpGet("Properties")]
public async Task<ActionResult<IEnumerable<PropertyListItemDTO>>> Properties()
{
    return await _context.Properties.Select(p =>
        _propertyMapService.GetPropertyListItemDTO(p)
    ).ToListAsync();
}
```

Ukážka kódu 9 Metóda Properties.

5.1.3 gRPC

Pre vytvorenie projektu gRPC server je využitá gRPC Service šablóna. Ako už bolo spomenuté základom gRPC je proto súbor, ktorý obsahuje definíciu rozhrania spolu (ukážka kódu č. 10) s použitými dátovými štruktúrami (ukážka kódu č. 11). Samotné messages v proto súbore predstavujú DTO modely, ktoré budú slúžiť ako posielané objekty pri komunikácii klienta so serverom. V prostredí .NET sa proto súbor pridáva ako protocol buffer file. Pre prácu s gRPC a protocol buffers je potrebné mať nainštalované nuget balíky Grpc.AspNetCore, Grpc.Tools a Google.Protobuf.

Metódy rozhrania sa v gRPC odlišujú od REST riešenia alebo iných RPC frameworkov tým, že majú vždy povinný len jeden vstupný parameter a majú vždy návratovú hodnotu. To znamená, že každá metóda musí prijímať alebo vracať minimálne prázdnu message, ktorá nemá žiadne fields.

Ďalším rozdielom je to, že gRPC nepodporuje všetky dátové typy, ktoré bežne používajú programovacie jazyky. Nepodporuje dátové typy ako sú napríklad decimal alebo datetime. Taktiež nepozná klasické pole (array) alebo list, ekvivalentom pre kolekcie je použitie kľúčového slova repeated pred daným fieldom.

Pri najnovšej verzii protocol buffers, proto3, sú všetky fieldy v základe gRPC voliteľné (optional), nikdy nenadobúdajú hodnotu null, nie sú nullable, sú vždy inicializované defaultnými hodnotami (0, empty string...). To znamená, že každé priradenie null hodnoty do fieldu vráti NullPointerException. Preto je nutné pri každom priradení hodnoty fieldu kontrolovať hodnotu, či neobsahuje null a v prípade, že áno, tak hodnotu neuložiť do fieldu, čo môže byť nekomfortné pre programátora. Napríklad ani dátový typ string, ktorý je obvykle pri programovacích jazykoch nullable, nedovoľuje priradenie hodnoty null. Zároveň tým, že celá message je inicializovaná defaultnými hodnotami dátových typov, tak gRPC metóda nikdy nevráti null.

Jedným z možných riešení problémov s dátovými typmi, ktoré je aj použité v navrhnutej aplikácii, je použitie Google proto súborov. Tieto proto súbory sú obsiahnuté priamo v nuget

balíku Google.Tools a ich presnú definíciu je možné nájsť v Google GitHub protocol buffers repozitári.

Medzi tieto proto súbory patrí napríklad empty proto súbor, ktorý sa importuje ako google/protobuf/empty.proto. Obsahuje prázdnu message s názvom google.protobuf.Empty, ktorá nahradzuje potrebu pri každej gRPC službe implementovať vlastnú prázdnu message.

Databázový model zákazky obsahuje property Commision a Price, ktoré majú dátový typ decimal. gRPC nepozná tento dátový typ, preto je namiesto neho využitý dátový typ float, ktorý je jednoducho konvertovateľný na decimal.

Zložitejšia situácia je pri propertách zákazky DateCreated a DateUpdated, ktoré majú dátový typ datetime. Pre prácu s datetime je nutné importovať proto súbor google/protobuf/timestamp, ktorý rozširuje gRPC o dátový typ google.protobuf.Timestamp, ktorý je možné konvertovať na datetime.

Druhým využitým importovaným súborom v aplikácii je wrappers súbor (google/protobuf/wrappers), ktorý ponúka nullable dátové typy ako napríklad google.protobuf.StringValue (nullable string), google.protobuf.Int32Value (nullable int) alebo google.protobuf.FloatValue (nullable float).

```
syntax = "proto3";

import "google/protobuf/timestamp.proto";
import "google/protobuf/wrappers.proto";

option csharp_namespace = "gRPC.Protos";

package property;

service Property {
    rpc PropertyCreate (PropertyItem) returns (PropertyID);
    rpc PropertyDetail (PropertyID) returns (PropertyItem);
    rpc PropertyUpdate (PropertyItem) returns (Empty);
    rpc PropertyDelete (PropertyID) returns (Empty);
    rpc Properties (Empty) returns (PropertyList);
}
```

Ukážka kódu 10 Definícia rozhrania v proto súbore.

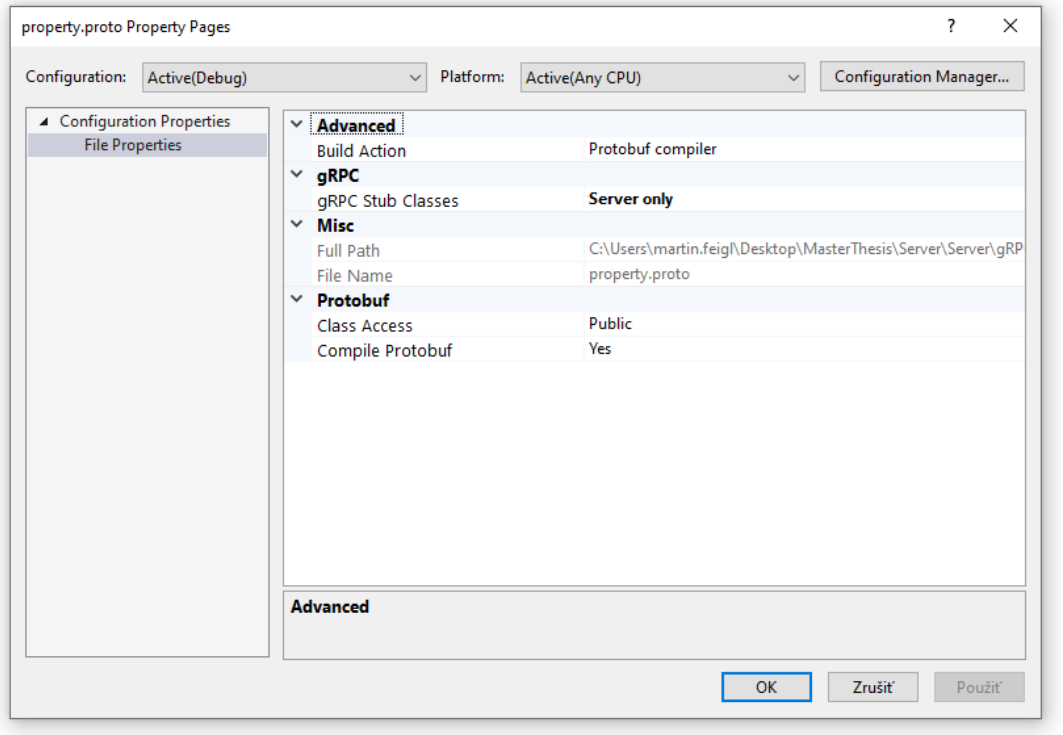
```
message Empty {

}

message PropertyID {
    int32 PropertyID = 1;
}

message PropertyListItem {
    google.protobuf.Timestamp DateCreated = 1;
    google.protobuf.Timestamp DateUpdated = 2;
    google.protobuf.FloatValue Price = 3;
    int32 PropertyID = 4;
    google.protobuf.StringValue Title = 5;
}

message PropertyList {
    repeated PropertyListItem Properties = 1;
}

message PropertyItem {
    bool IsBalcony = 1;
    bool IsCellar = 2;
    bool IsGarage = 3;
    bool IsParking = 4;
    bool IsTerrace = 5;
    google.protobuf.FloatValue Commision = 6;
    google.protobuf.FloatValue Price = 7;
    google.protobuf.FloatValue BuildingArea = 8;
    google.protobuf.FloatValue FloorArea = 9;
    google.protobuf.FloatValue UsableArea = 10;
    int32 PropertyCategoryTypeID = 11;
    int32 PropertyTypeID = 12;
    google.protobuf.Int32Value PropertyID = 13;
    google.protobuf.StringValue City = 14;
    google.protobuf.StringValue Description = 15;
    google.protobuf.StringValue PostCode = 16;
    google.protobuf.StringValue Region = 17;
    google.protobuf.StringValue Street = 18;
    google.protobuf.StringValue StreetNo = 19;
    google.protobuf.StringValue Title = 20;
}
```

Ukážka kódu 11 Dátové štruktúry v proto súbore.

Vo vlastnostiach každého proto súboru (obrázok č. 39) je potrebné nastaviť Build Action na hodnotu Protobuf compiler a v možnosti gRPC Stub Classes vybrať spôsob generovania kódu z proto súboru. Toto nastavenie určuje, pre ktorú stranu je možné generovať kód. Na výber sú možnosti Client and Server, Client only, Server only, Do not generate a inherit from parent or project default. V ukážkovej aplikácii je nastavená hodnota Server only, pre klienta je použitá vlastná verzia proto súboru.

Nastaviť spôsob generovania kódu, samotné generovanie a kompilovanie proto súborov je možné vďaka použitiu spomenutého nuget balíka Grpc.Tools. Tento balík obsahuje binárne súbory gRPC kompilátoru protoc a protobuf C# pluginu, ktoré sú potrebné pre vygenerovanie kódu z proto súboru. Od verzie 1.17 je tento nuget integrovaný vo Visual Studiu spolu

s MSBuild a umožňuje automatické generovanie C# kódu z proto súborov. Nie je tak potrebné pre generovanie kódu používať protoc príkazy v príkazovom riadku, nuget Grpc.Tools automaticky vykonáva samotné generovanie kódu.



Obrázok 39 Nastavenie proto súboru.

Generovaný kód, ktorý je automaticky vytvorený z proto súborov sa nachádza v projektovej zložke obj/debug/netcoreapp3.1/Protos. Pre každý proto súbor sú vytvárané dva vygenerované súbory. Aplikácia obsahuje proto súbor s názvom Property, pre tento súbor boli vygenerované dva klasické C# súbory s názvom Property.cs a PropertyGrpc.cs. Pri každej uloženej zmene v proto súbore sú tieto súbory ihneď pregenerované.

Tieto súbory obsahujú preklad formátu protocol buffers do jazyka C#. Majú na starosti serializáciu a deserializáciu dát, celkovú komunikáciu pomocou gRPC. Obsahujú základnú abstraktnú definíciu rozhrania, ktoré obsahuje všetky RPC metódy.

Ukážky z automaticky vygenerovaného kódu je možné vidieť na ukážkach kódu č. 12, 13 a 14.

```
// <auto-generated>
//     Generated by the protocol buffer compiler.  DO NOT EDIT!
//     source: Protos/property.proto
// </auto-generated>
#pragma warning disable 1591, 0612, 3021
#region Designer generated code

using pb = global::Google.Protobuf;
using pbc = global::Google.Protobuf.Collections;
using pbr = global::Google.Protobuf.Reflection;
using scg = global::System.Collections.Generic;
namespace gRPC.Protos {

  /// <summary>Holder for reflection information generated from Protos/property.proto</summary>
  public static partial class PropertyReflection {

    Descriptor

  }
  Messages

}

#endregion Designer generated code
```

Ukážka kódu 12 Automaticky vygenerovaný kód z proto súboru.

```
/// <summary>Base class for server-side implementations of Property</summary>
[grpc::BindServiceMethod(typeof(Property), "BindService")]
public abstract partial class PropertyBase
{
  public virtual global::System.Threading.Tasks.Task<global::gRPC.Protos.PropertyID> PropertyCreate(global::gRPC.Prot
  {
    throw new grpc::RpcException(new grpc::Status(grpc::StatusCode.Unimplemented, ""));
  }

  public virtual global::System.Threading.Tasks.Task<global::gRPC.Protos.PropertyItem> PropertyDetail(global::gRPC.Pr
  {
    throw new grpc::RpcException(new grpc::Status(grpc::StatusCode.Unimplemented, ""));
  }

  public virtual global::System.Threading.Tasks.Task<global::gRPC.Protos.Empty> PropertyUpdate(global::gRPC.Protos.Pr
  {
    throw new grpc::RpcException(new grpc::Status(grpc::StatusCode.Unimplemented, ""));
  }

  public virtual global::System.Threading.Tasks.Task<global::gRPC.Protos.Empty> PropertyDelete(global::gRPC.Protos.Pr
  {
    throw new grpc::RpcException(new grpc::Status(grpc::StatusCode.Unimplemented, ""));
  }

  public virtual global::System.Threading.Tasks.Task<global::gRPC.Protos.PropertyList> Properties(global::gRPC.Protos
  {
    throw new grpc::RpcException(new grpc::Status(grpc::StatusCode.Unimplemented, ""));
  }
}
```

Ukážka kódu 13 Automaticky vygenerovaný kód z proto súboru.

```
public static partial class Property
{
  static readonly string __ServiceName = "property.Property";

  static void __Helper_SerializeMessage(global::Google.Protobuf.IMessage message, grpc::SerializationContext context)
  {
    #if !GRPC_DISABLE_PROTOBUF_BUFFER_SERIALIZATION
    if (message is global::Google.Protobuf.IBufferMessage)
    {
      context.SetPayloadLength(message.CalculateSize());
      global::Google.Protobuf.MessageExtensions.WriteTo(message, context.GetBufferWriter());
      context.Complete();
      return;
    }
    #endif
    context.Complete(global::Google.Protobuf.MessageExtensions.ToByteArray(message));
  }
```

Ukážka kódu 14 Automaticky vygenerovaný kód z proto súboru.

Pre samotnú obsluhu RPC metód je nutné implementovať definíciu rozhrania proto súboru. Property Servisa (ukážka kódu č. 15), ktorá implementuje aplikačnú logiku dedí od základnej abstraktnej Base triedy (Property.PropertyBase). Táto Base trieda sa nachádza v automaticky vygenerovanom kóde z proto súboru a obsahuje virtuálne metódy, ktoré je nutné prepísať.

```
public class PropertyService : Property.PropertyBase
{
    private readonly PropertyContext _context;
    private readonly Map.IPropertyMapService _propertyMapService;

    public PropertyService(
        PropertyContext context,
        Map.IPropertyMapService propertyMapService
    )
    {
        _context = context;
        _propertyMapService = propertyMapService;
    }
```

Ukážka kódu 15 PropertyService.

Samotná logika metód je podobná ako pri REST API, spočíva v jednoduchej práci s databázovým kontextom a mapovaním databázových modelov na proto správy. Implementované metódy rozhrania je vidieť na ukážkach kódu č. 16, 17, 18, 19 a 20.

```
public override async Task<Protos.PropertyID> PropertyCreate(Protos.PropertyItem request, ServerCallContext con-
text)
{
    var property = _propertyMapService.GetProperty(request);

    _context.Properties.Add(property);

    await _context.SaveChangesAsync();

    return await Task.FromResult(_propertyMapService.GetProtoPropertyID(property.PropertyID));
}
```

Ukážka kódu 16 Metóda PropertyCreate.

```
public override async Task<Protos.PropertyItem> PropertyDetail(Protos.PropertyID request, ServerCallContext con-
text)
{
    var property = await _context.Properties.FindAsync(request.PropertyID_);

    if (property == null)
    {
        return await Task.FromResult(new Protos.PropertyItem());
    }

    return await Task.FromResult(_propertyMapService.GetProtoProperty(property));
}
```

Ukážka kódu 17 Metóda PropertyDetail.

```
public override async Task<Protos.Empty> PropertyUpdate(Protos.PropertyItem request, ServerCallContext context)
{
    var property = await _context.Properties.FindAsync(request.PropertyID);

    if (property == null)
    {
        return await Task.FromResult(new Protos.Empty());
    }

    _propertyMapService.MapProperty(property, request);

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException) when (!PropertyExists(request.PropertyID.Value))
    {
        return await Task.FromResult(new Protos.Empty());
    }

    return await Task.FromResult(new Protos.Empty());
}
```

Ukážka kódu 18 Metóda PropertyUpdate

```
public override async Task<Protos.Empty> PropertyDelete(Protos.PropertyID request, ServerCallContext context)
{
    var property = await _context.Properties.FindAsync(request.PropertyID_);

    if (property == null)
    {
        return await Task.FromResult(new Protos.Empty());
    }

    _context.Properties.Remove(property);

    await _context.SaveChangesAsync();

    return await Task.FromResult(new Protos.Empty());
}
```

Ukážka kódu 19 Metóda PropertyDelete.

```
public override async Task<Protos.PropertyList> Properties(Protos.Empty request, ServerCallContext context)
{
    var properties = await _context.Properties.ToListAsync();

    return await Task.FromResult(_propertyMapService.GetProtoProperties(properties));
}
```

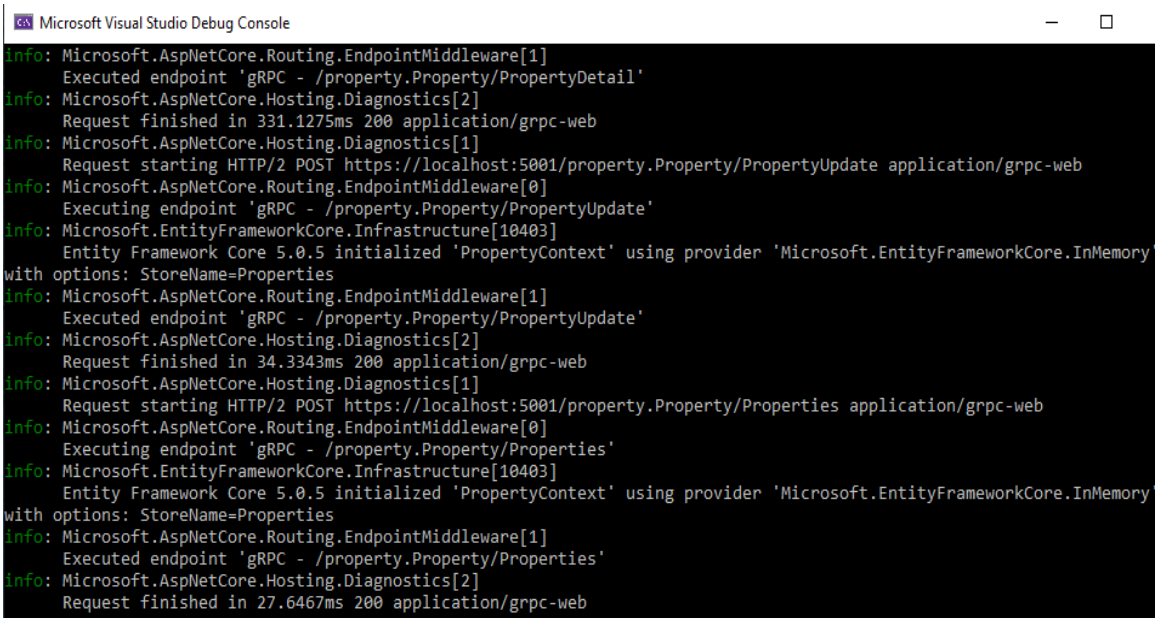
Ukážka kódu 20 Metóda Properties

Na strane serveru je taktiež využitá dependency injection a IoC kontajner Autofac, ktorý treba rovnako registrovať ako pri REST riešení v súboroch Program.cs a Startup.cs. Klient, ktorý bude komunikovať s gRPC serverom je webová aplikácia využívajúca Blazor. Aby webový prehliadač mohol komunikovať s gRPC serverom, tak server musí mať nastavené použitie pluginu gRPC-Web a povolené CORS. gRPC-Web je dostupné ako nuget balíček Grpc.AspNetCore.Web. Povoliť použitie gRPC-Web a CORS je treba nutné spraviť v Startup.cs (ukážka kódu č. 21).

Pri úspešnom spustení gRPC serveru sa zároveň spustí aj konzola (obrázok č. 40), v ktorej je možné sledovať stav všetkých requestov.

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<Data.Models.PropertyContext>(p => p.UseInMemoryDatabase("Properties"));
        services.AddGrpc();
        services.AddCors(o => o.AddPolicy("AllowAll", builder =>
        {
            builder.AllowAnyOrigin()
                   .AllowAnyMethod()
                   .AllowAnyHeader()
                   .WithExposedHeaders("grpc-status", "grpc-message", "grpc-encoding", "grpc-accept-encoding");
        }));
    }

    public void ConfigureContainer(ContainerBuilder builder)
    {
        builder.RegisterType<Services.Map.PropertyMapService>()
               .As<Services.Map.IPropertyMapService>()
               .InstancePerLifetimeScope();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseRouting();
        app.UseGrpcWeb();
        app.UseCors();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGrpcService<PropertyService>().EnableGrpcWeb().RequireCors("AllowAll");
        });
    }
}
```

Ukážka kódu 21 Startup.cs.



Obrázok 40 Konzola gRPC serveru.

5.2 Klient

Klient je implementovaný ako blazor server aplikácia, ktorá je schopná komunikovať s REST API a zároveň aj s gRPC servisou. Webové rozhranie aplikácie je rozdelené na dve časti, ktoré predstavujú jednotlivé REST a gRPC riešenia.

Samotné riešenia využívajú rovnaké view a razor komponenty. Jediný rozdiel je v tom, na ktorý server sa vykonávajú požiadavky, tzn. CRUD operácie vyvolané z REST riešenia smerujú na REST server, operácie vyvolané z gRPC riešenia smerujú na gRPC server. Pre každé riešenie je nutné vytvoriť klienta, ktorý bude komunikovať so serverom. Následne ak volaná metóda vracia model, tak je nutné ho namapovať na model, ktorý je použitý v daných razor komponentoch.

Rovnako ako pri serveri, tak aj pri klientovi je využité dependency injection a je nutné podobne nastaviť súbor Startup.cs a Program.cs.

5.2.1 REST

Základ REST klienta predstavuje HttpClient. Pre HttpClienta sa nastaví základná URI adresa servera. Následne sa z neho zavolá HTTP metóda (GetAsync, PostAsync, PutAsync, DeleteAsync) s konkrétnou URI adresou zdroja, nad ktorým sa vykonáva požiadavka.

Metóda môže vyžadovať posielanie parametrov alebo dát v hlavičke alebo v tele požiadavky. Ak je nutné posielať parametre priamo v hlavičke, tak sú súčasťou priamo URI (HTTP metóda GET a DELETE). Posielané dáta v tele požiadavky je nutné serializovať (HTTP metóda POST a PUT), rovnako je potrebné deserializovať každú odpoveď servera. Následne sa deserializovaný DTO model mapuje na model používaný v aplikácii.

Použitie HTTP metód pre request na REST API je možné vidieť na ukážkach kódu č. 22, 23, 24, 25, 26 a 27.

```
public HttpClient GetHttpClient()
{
    var client = new HttpClient();
    client.BaseAddress = new Uri("https://localhost:44310");
    client.DefaultRequestHeaders.Accept.Clear();

    return client;
}
```

Ukážka kódu 22 Vytvorenie HttpClienta.

```
public async Task<int> PropertyCreate(Data.Property property)
{
    using (var httpClient = GetHttpClient())
    {
        var propertyDTO = _propertyMapService.GetPropertyDTO(property);
        var json = new StringContent(JsonConvert.SerializeObject(propertyDTO), Encoding.UTF8, "application/json");
        var response = httpClient.PostAsync($"/api/property/propertycreate", json).Result;
        var returnResult = response.Content.ReadAsStringAsync().Result;
        var propertyID = JsonConvert.DeserializeObject<int>(returnResult);

        return await Task.FromResult(propertyID);
    }
}
```

Ukážka kódu 23 Metóda PropertyCreate.

```
public async Task<Data.Property> PropertyDetail(int propertyID)
{
    using (var httpClient = GetHttpClient())
    {
        var response = httpClient.GetAsync($"/api/property/propertydetail/{propertyID}").Result;
        var returnResult = response.Content.ReadAsStringAsync().Result;
        var propertyDTO = JsonConvert.DeserializeObject<PropertyDTO>(returnResult);
        var property = _propertyMapService.GetProperty(propertyDTO);

        return await Task.FromResult(property);
    }
}
```

Ukážka kódu 24 Metóda PropertyDetail.

```
public async Task<bool> PropertyUpdate(Data.Property property)
{
    using (var httpClient = GetHttpClient())
    {
        var propertyDTO = _propertyMapService.GetPropertyDTO(property);
        var json = new StringContent(JsonConvert.SerializeObject(propertyDTO), Encoding.UTF8, "application/json");
        var response = httpClient.PutAsync($"/api/property/propertyupdate", json).Result;
        var returnResult = response.Content.ReadAsStringAsync().Result;

        return await Task.FromResult(true);
    }
}
```

Ukážka kódu 25 Metóda PropertyUpdate.

```
public async Task<bool> PropertyDelete(int propertyID)
{
    using (var httpClient = GetHttpClient())
    {
        var response = httpClient.DeleteAsync($"/api/property/propertydelete/{propertyID}").Result;
        var returnResult = response.Content.ReadAsStringAsync().Result;

        return await Task.FromResult(true);
    }
```

Ukážka kódu 26 Metóda PropertyDelete.

```
public async Task<List<PropertyListItem>> Properties()
{
    using (var httpClient = GetHttpClient())
    {
        var response = httpClient.GetAsync("/api/property/properties").Result;
        var returnResult = response.Content.ReadAsStringAsync().Result;
        var propertiesDTO = JsonConvert.DeserializeObject<List<PropertyListItemDTO>>(returnResult);
        var properties = propertiesDTO.Select(p => _propertyMapService.GetPropertyListItem(p)).ToList();

        return await Task.FromResult(properties);
    }
}
```

Ukážka kódu 27 Metóda Properties.

5.2.2 gRPC

Pri gRPC je obdobou HttpClienta GrpcChannel a GrpcClient. Pre tento kanál je rovnako nastavená základná adresa servera. Blazor neumožňuje priamo komunikovať pomocou HTTP/2 a je preto nutné použiť plugin gRPC-Web. Použitie tohto pluginu je veľmi jednoduché a stačí pre GrpcChannel pridať GrpcWebHandler. Pre gRPC-Web je potrebné stiahnuť nuget Grpc.Net.ClientWeb, pre prácu s protocol buffers sú potrebné nugety Grpc.Tools a Google.Protobuf a pre prácu s gRPC klientom je nutný nuget Grpc.Net.Client.

Ako gRPC dátový formát je využitý protocol buffers, pre ktorý je nutné vytvoriť proto súbor. Pri správnom nastavení serverového proto súboru je možné serverový proto súbor použiť aj na strane klienta, avšak pre reálnejšiu ukážku práce s gRPC je v aplikácii na strane klienta vytvorený nový proto súbor, ktorý má rovnakú štruktúru ako proto súbor na strane servera. Pre tento proto súbor na strane klienta je nastavené generovanie kódu s hodnotou Client Only.

Vygenerovaný kód z proto súboru obsahuje klienta, ktorý obsluhuje dostupné metódy z daného proto súboru. Tento klient využíva vytvorený GrpcChannel a následne je z neho možné zavolať jednotlivé metódy. Prenášané dáta nie je potrebné serializovať a deserializovať, na pozadí sa o to stará automaticky vygenerovaný kód. Rovnako ako pri REST riešení, tak následne každá odpoveď serveru je mapovaná pomocnými servisami na model, ktorý je používaný v aplikácii.

Použitie gRPC kanálu a gRPC klienta je možné vidieť na ukážkach kódu č. 28, 29, 30, 31, 32 a 33.

```
public Protos.Property.PropertyClient GetGrpcClient()
{
    var channel = GrpcChannel.ForAddress("https://localhost:5001", new GrpcChannelOptions
    {
        HttpHandler = new GrpcWebHandler(new HttpClientHandler())
    });
    var client = new Protos.Property.PropertyClient(channel);

    return client;
}
```

Ukážka kódu 28 Vytvorenie gRPC klienta.

```
public async Task<int> PropertyCreate(Data.Property property)
{
    var client = GetGrpcClient();
    var protoProperty = _propertyMapService.GetProtoProperty(property);
    var response = await client.PropertyCreateAsync(protoProperty);

    return await Task.FromResult(response.PropertyID_);
}
```

Ukážka kódu 29 Metóda PropertyCreate.

```
public async Task<Data.Property> PropertyDetail(int propertyID)
{
    var client = GetGrpcClient();
    var protoPropertyID = _propertyMapService.GetProtoPropertyID(propertyID);
    var response = await client.PropertyDetailAsync(protoPropertyID);
    var property = _propertyMapService.GetProperty(response);

    return await Task.FromResult(property);
}
```

Ukážka kódu 30 Metóda PropertyDetail.

```
public async Task<bool> PropertyUpdate(Data.Property property)
{
    var client = GetGrpcClient();
    var protoProperty = _propertyMapService.GetProtoProperty(property);
    var response = await client.PropertyUpdateAsync(protoProperty);

    return await Task.FromResult(true);
}
```

Ukážka kódu 31 Metóda PropertyUpdate.

```
public async Task<bool> PropertyDelete(int propertyID)
{
    var client = GetGrpcClient();
    var protoPropertyID = _propertyMapService.GetProtoPropertyID(propertyID);
    var response = await client.PropertyDeleteAsync(protoPropertyID);

    return await Task.FromResult(true);
}
```

Ukážka kódu 32 Metóda PropertyDelete.

```
public async Task<List<PropertyListItem>> Properties()
{
    var client = GetGrpcClient();
    var response = await client.PropertiesAsync(new Protos.Empty());
    var properties = response.Properties.Select(p => _propertyMapService.GetPropertyListItem(p)).ToList();

    return await Task.FromResult(properties);
}
```

Ukážka kódu 33 Metóda Properties.

6 POROVNANIE REST A GRPC RIEŠENIA

Jednotlivé REST a gRPC riešenie je porovnané z hľadiska rýchlosti a veľkosti prenášaných dát. Porovnávanie pozostáva z vyhodnotenia 50 GET requestov na metódu pre vrátanie zoznamu zákaziek. Pred porovnávaním bolo pre každé riešenie uložených rovnakých 5000 zákaziek.

Porovnanie bolo prevedené za použitia programov Postman pre REST server a pomocou programu Kreya pre gRPC server. Tieto programy predstavujú testovacích klientov, jedná sa o nástroje pomocou, ktorých je možné vytvoriť requesty na server a merať rýchlosť a veľkosť prenesených dát. V súčasnosti neexistuje spoľahlivý jednotný testovací klient pre REST a gRPC riešenia, použitie dvoch rozdielnych testovacích klientov môže mať za následok čiastočné skreslenie výsledkov kvôli rozdielnej réžii pri volaní a spracovaní metód.

Volanie GET metódy pre vrátenie zoznamu zákaziek v programe Postman trvalo priemerne 90,82 ms a veľkosť daných dát bola 785 KB. Na druhej strane nameraná priemerná rýchlosť pomocou programu Kreya bola 74,32 ms a veľkosť dát bola 261 KB. Vo výsledku gRPC riešenie bolo o 22,2 % rýchlejšie a veľkosť prenášaných dát bola tretinová.

Rozdiel v rýchlosti by bol niekoľkonásobne väčší, ak by sa mohli prejaviť naplno výhody HTTP/2, na ktorom je gRPC postavené. HTTP/2 aplikácie sú rýchlejšie, jednoduchšie a robustnejšie. HTTP/2 zároveň neposkytuje internetovým prehliadačom prístup ku svojim dátovým rámcom, a preto nie je možná priama komunikácia webového klienta a gRPC serveru, je nutné používať proxy servery. Z toho dôvodu pri použití Blazoru je nutné využívať napríklad plugin gRPC-Web, ktorý vytvára proxy server pre preklad HTTP/1 požiadaviek klienta na HTTP/2 požiadavky serveru. Rozdiel vo veľkosti prenášaných dát je výrazný a to z dôvodu použitia najnovšej verzie protocol buffers, ktorá poskytuje vylepšenú kompresiu dát.

Samotné riešenia predstavujú dva rozdielne štýly komunikácie. Komunikácia pomocou REST API je populárny prístup, ktorý je orientovaný dátovo. Spočíva v princípe prístupu ku zdrojom, ktoré sú identifikovateľné pomocou URI. Ku zdrojom sa pristupuje pomocou jednotného a jednoduchého použitia základných HTTP metód. Ako prenosový formát je využitý textový formát JSON.

gRPC je založené na vzdialenom volaní procedúr a jeho základom je proto súbor, ktorý predstavuje definíciu rozhrania a použité dátové štruktúry. Využíva binárny prenosový formát protocol buffers, ktorý je maximálne optimalizovaný na rýchlosť a veľkosť prenosu.

Z proto súboru je automaticky generovaný kód, ktorý slúži pre základnú komunikáciu a serializáciu dát na strane serveru aj na strane klienta. Práca s gRPC v prostredí .NET je jednoduchá, nuget balíky navyše obsahujú protoc kompilátor pre protocol buffers, ale napriek tomu implementovať gRPC rozhranie zaberie pravdepodobne dlhšiu dobu ako REST API.

ZÁVER

Hlavným cieľom diplomovej práce bolo opísať moderný framework gRPC, ktorý slúži pre komunikáciu typu klient-server a je založený na vzdialenom volaní procedúr.

Teoretická časť sa venuje opisu komunikácie dvoch strojov alebo aplikácii pomocou webových služieb. Popisuje základné entity a princípy architektúry orientovanej na služby a približuje komunikáciu založenú na protokole SOAP a jazyku XML. Ďalej opisuje komunikáciu založenú na dátovo orientovanej architektúre rozhrania REST, opisuje jej základné pravidlá a prístup pomocou HTTP metód.

Primárna časť sa venuje frameworku gRPC, ktorý predstavuje výkonnú alternatívu ku bežne používaným spôsobom komunikácie. Jedná sa o relatívne nový framework založený na komunikácii pomocou protokolu HTTP/2. gRPC oproti bezstavovému RESTu plno podporuje všetky formy streamovania a taktiež prináša binárny prenosový formát protocol buffers, ktorý je maximálne optimalizovaný na rýchlosť prenosu a veľkosť prenášaných dát. To predurčuje gRPC v prípadoch použitia ako mikroservisy, komunikácia v reálnom čase, vysoko škálovateľné distribuované systémy s nízkou latenciou alebo internet vecí.

V praktickej časti je navrhnutá a vytvorená webová aplikácia, ktorá využíva REST a gRPC v prostredí .NET a Blazor. Aplikácia približuje a porovnáva základné prvky komunikácie jednotlivých technológii. Webové aplikácie predstavujú pre gRPC slabinu, a to z dôvodu obmedzeného prístupu internetových prehliadačov ku dátovým rámcom protokolu HTTP/2, ktorý využíva gRPC. Napriek tomu je gRPC výkonný framework, ktorý sa nemusí obávať o svoju budúcnosť a smerovanie, jedná sa o perspektívnu novinku, za ktorou stoja veľké technologické firmy na čele so spoločnosťou Google.

ZOZNAM POUŽITEJ LITERATÚRY

[1] CERAMI, Ethan. *Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL*. Sebastopol: O'Reilly, 2002. ISBN 0-596-00224-6.

[2] *World Wide Web Consortium (W3C)* [online]. [cit. 2021-03-21]. Dostupné z: https://www.w3.org

[3] *Systinet: Introduction to Web Services Architecture* [online]. In: . 2002 [cit. 2021-03-21]. Dostupné z: https://www.immagic.com/eLibrary/ARCHIVES/GENERAL/SYSTINET/S020716S.pdf

[4] ERL, Thomas. *SOA: Principles of Service Design*. Crawfordsville: R.R. Donnelley, 2014. ISBN 0-13-234482-3.

[5] CURBERA, Francisco, Matthew DUFTLER, Rania KHALAF, William NAGY, Nirmal MUKHI a Sanjiva WEERAWARANA. *Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI*, IEEE Internet Computing, VI(2), 86-93, 2002, doi: 10.1109/4236.991449.

[6] KUBA, M. *Web Services*. Zpravodaj ÚVT MU. 2003, XIII(3), 9-14. ISSN 1212-0901.

[7] FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Irvine, 2000. Dizertačná práca. University of California. Vedoucí práce Richard N. Taylor.

[8] LEONARD, Richardson, Mike AMUNDSEN a Sam RUBY. *RESTful Web APIs: Services for a Changing World*. Sebastopol: O'Reilly, 2013. ISBN 978-1-449-35806-8.

[9] *GRPC – A high-performance, open source universal RPC framework.* gRPC [online]. [cit. 09.11.2020]. Dostupné z: https://grpc.io/ [online].

[10] INDRASIRI, Kasun a Danesh KURUPPU. *GRPC: Up and Running: Building Cloud Native Applications with Go and java for Docker and Kubernetes*. Sebastopol: O'Reilly, 2020. ISBN 9781492058335.

[11] RESELMAN, Bob. *Understanding the Essentials of gRPC*. ProgrammableWeb [online]. 2014 [cit. 2021-5-23]. Dostupné z: https://www.programmableweb.com/news/understanding-essentials-grpc/analysis/2020/10/14

[12] *Protocol Buffers* [online]. Google Developers [cit. 2021-5-23]. Dostupné z: https://developers.google.com/protocol-buffers/docs/overview

[13] KREBS, Bruno. *Beating JSON performance with Protobuf*. *Auth0* [online]. 2017 [cit. 2021-5-23]. Dostupné z: https://auth0.com/blog/beating-json-performance-with-protobuf/

[14] DE KLERK, Jean. *HTTP/2: Smarter at scale*. Clound Native Computing Foundation [online]. 2018 [cit. 2021-5-23]. Dostupné z: https://www.cncf.io/blog/2018/07/03/http-2-smarter-at-scale

[15] GitHub - grpc/grpc-web: *gRPC Web* [cit. 2021-5-23]. Dostupné z: https://github.com/grpc/grpc-web

[16] *Introduction to gRPC services* [online]. Microsoft Docs [cit. 2021-5-23]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/grpc

[17] GitHub - grpc-ecosystem/grpc-gateway: *gRPC-Gateway* [cit. 23.05.2021]. Dostupné z: https://github.com/grpc-ecosystem/grpc-gateway

[18] *Case studies.* Clound Native Computing Foundation [online]. [cit. 2021-5-23]. Dostupné z: https://www.cncf.io/case-studies/?_sft_lf-project=grpc

[19] *What is gRPC: Main Concepts, Pros and Cons, Use Cases*. Altexsoft [online]. [cit. 2021-5-23]. Dostupné z: https://www.altexsoft.com/blog/what-is-grpc/

[20] *When to use gRPC over REST*. Medium [online]. [cit. 2021-5-23]. Dostupné z: https://medium.com/@sankar.p/how-grpc-convinced-me-to-chose-it-over-rest-30408bf42794

[21] ALBAHARI, Joseph a Ben ALBAHARI. *C# 7.0 in a nutshell*. 7th edition. Sebastopol: O´Reilly, 2018. ISBN 978-1491987650.

[22] NAGEL, Christian. *Professional C# 7 and .NET Core 2.0*. 11th edition. Indianapolis: Wrox, a Wiley Brand, 2018. ISBN 978-1119449270.

[23] *Blazor | Build client web apps with C#* | .NET. .NET | Free. Cross-platform. Open Source. [online]. Dostupné z: https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor

[24] *WebAssembly* [online]. [cit. 2021-5-23]. Dostupné z: https://webassembly.org

[25] V. HAAS, Andreas, Andreas ROSSBERG, Derek L. SCHUFF, et al. *Bringing the web up to speed with WebAssembly*, Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY: ACM, 2017, s. 185-200. ISBN 978-1-4503-4988-8.

ZOZNAM POUŽITÝCH SYMBOLOV A SKRATIEK

AJAX	Asynchronous JavaScript And XML
ALPN	Application Layer Protocol Negotiation
ALTS	Application Layer Transport Security
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BEEP	Blocks Extensible Exchange Protocol
CNCF	Cloud Native Computing Foundation
CORBA	Common Object Request Broker Architecture
CORS	Cross-Origin Resource Sharing
CRM	Customer Relationship Management
CRUD	Create, Read, Update and Delete
DB	Database
DCOM	Distributed Component Object Model
DDoS	Distributed Denial of Service
DTO	Data Transfer Object
EOS	End Of Stream
FTP	File Transfer Protocol
gRPC	gRPC Remote Procedure Call
HATEOAS	Hypermedia as the Engine of Application State
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IIS	Internet Information Services
IoC	Invesrion of Control
Java RMI	Java Remote Method Invocation

JMS	Java Message Service
JSON	JavaScript Object Notation
JWT	JSON Web Token
OAuth	Open Authorization
QUIC	QUIC User Datagram Protocol Internet Connection
RAM	Random Access Memory
REST	Representational State Transfer
RPC	Remote Procedure Callrs
RSS	Really Simple Syndication
SDK	Software Development Kit
SMTP	Simple Mail Transfer Protocol
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SPA	Single Page Application
SPDY	SPeeDY
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
Telnet	Teletype Network
TLS	Transport Layer Security
UDDI	Universal Description, Discovery, and Integration
UDT	User Datagram Protocol-Based Data Transfer
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UTF	Unicode Transformation Format
W3C	World Wide Web Consortium
WebDAV	Web-based Distributed Authoring and Versioning

WSDL	Web Service Definition Language
WS-Federation	Web Services Federation
WSIL	Web Services Inspection Language
XML	Extensible Markup Language
XML-RPC	Extensible Markup Language – Remote Procedure Call
XMPP	Extensible Messaging and Presence Protocol
YAML	YAML Ain't Markup Language

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

ZOZNAM UKÁŽOK KÓDU

ZOZNAM PRÍLOH

PRÍLOHA P I: CD

CD obsahuje:

- Zdrojové kódy webovej aplikácie: prilohy.zip / srcClient
- Zdrojové kódy serveru: prilohy.zip / srcServer
- Diplomová práca vo formáte pdf: fulltext.pdf