HO GENT

Departement IT en Digitale Innovatie

Een vergelijkende studie tussen gRPC met Protocol Buffers en REST met JSON in opdracht van Fashion Society.

Sven Pepermans

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Antonia Pierreux
Co-promotor:
Kristof van Moorter

Instelling: Fashion Society

Academiejaar: 2020-2021

Tweede examenperiode

Departement IT en Digitale Innovatie

Een vergelijkende studie tussen gRPC met Protocol Buffers en REST met JSON in opdracht van Fashion Society.

Sven Pepermans

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Antonia Pierreux
Co-promotor:
Kristof van Moorter

Instelling: Fashion Society

Academiejaar: 2020-2021

Tweede examenperiode

Woord vooraf

Ik heb deze bachelorproef geschreven voor het voltooien van mijn opleiding Toegepaste Informatica met als afstudeerrichting Mobile Apps. Ik heb dit onderzoek rond performantie tussen gRPC met Protocol Buffers en REST met JSON voor de implementatie van The Fashion Society uitgevoerd omdat ik grote interesse heb in het ontwikkelen van back-end applicaties en services. Daarnaast vind ik de back-end structuur van de Fashion Society waarmee ik leren werken heb gedurende mijn stage zeer intrigerend en vond ik het passend om te onderzoeken of het systeem performanter te maken is door enkel het gebruikte dataformaat, en de bijhorende structuur, aan te passen. Dit onderzoek heeft mij doen inzien dat er meer dan enkel REST en JSON is voor back-end applicaties en services. Waar ik voordien standaard JSON gebruikte zal ik nu eerst grondig nadenken of het niet beter is om een alternatief dataformaat te gebruiken.

Deze bacherlorproef zou echter niet tot stand zijn gekomen zijn zonder de hulp en bijstand van enkele mensen. Wat hierop volgt is een bedanking aan alle mensen die mij gesteund en geholpen hebben bij het ontwikkelen van deze bachelorproef.

Eerst en vooral zou ik de Fashion Society en specifiek Yoerick Lemmelijn willen bedanken voor het vertrouwen dat zij in mij hebben gestoken met het toegang verlenen tot de interne test server waarop een kopie van hun systeem draait en dewelke ook gebruik maakt van gevoelige data zoals klantgegevens. Als volgt wil ik zeer graag mijn co-promotor Kristof van Moorter bedanken. Dankzij zijn uitgebreide kennis van zowel het interne systeem van de Fashion Society als back-end applicaties, microservices en dataformaten is toch wel een groot deel van mijn bachelorproef en om specifiek te zijn mijn Proof-Of-Concept mogelijk gemaakt.

Alsook wil ik graag mijn promotor Antonia Pierreux bedanken voor de bijstand en feedback

op de inhoud van deze bachelorproef, zij stond altijd klaar voor het verbeteren van een nieuwe toevoeging aan deze bachelorproef en hiervan had ik ook steeds snel de feedback, daarnaast wil ik haar ook bedanken voor het altijd klaar staan voor te antwoorden op soms, wat ik zelf kan beschrijven als domme, vragen.

Vervolgens zou ik ook mijn ouders, vriendin en vrienden willen bedanken voor mij te pushen op de momenten dat het nodig was zodanig dat deze bachelorproef voor de deadline zou afgeraakt zijn en om regelmatig mijn tussentijdse versies te proof readen.

Tot slot wil ik ook mijn medestudenten van campus Aalst bedanken voor de behulpzaamheid en alle verduidelijkingen die zij via de Discord server gegeven hebben in tijden van onzekerheid en onduidelijkheid.

Bij deze wens ik u een aangename leeservaring toe.

Samenvatting

Op basis van dit onderzoek kan de Fashion Society kiezen om de huidige structuur van de Discovery API, zijnde REST met JSON, te behouden of over te schakelen naar gRPC met Protocol Buffers. Dit onderzoek is interessant doordat de Fashion Society als maar blijft uitbreiden en er dagelijks duizenden requests passeren door zowel de Orchestrator als de Discovery API en opmerkelijke verbeteringen in performantie door de eindgebruiker, zijnde het personeel van de Fashion Society en onderliggende bedrijven, duidelijk gevoeld kunnen worden. In dit onderzoek worden REST met JSON en gRPC met Protocol Buffers onderzocht en met elkaar vergeleken. Voor het uitvoeren van dit onderzoek wordt gebruik gemaakt van de testserver van de Fashion Society waar de huidige structuur reeds op geïmplementeerd is. De twee structuren worden vergeleken op basis van performantie in twee groeperingen, de kleine payload bestaande uit vierduizend requests en de grote payload bestaande uit tienduizend requests. Verder in dit document zult u een inleiding tot het onderwerp vinden waarin onder andere de huidige stand van zaken, de onderzoeksvraag en het verdere verloop van deze bachelorproef beschreven staan. Alsook vindt u voor elk van beide structuren de resultaten van beide payloads.

In dit onderzoek is duidelijk geworden dat gRPC met Protocolbuffers duidelijk een performantere implementatie heeft dan REST met JSON voor de Fashion Society. Wel is gebleken dat het niet aan de verwachtte snelheidsverbetering komt maar dat het verschil in snelheid van gRPC ten opzichte van REST relatief beperkt blijft. Doordat gRPC alsnog een performanter alternatief is voor REST is het naar de toekomst toe aan te raden aan de Fashion Society om de communicatie tussen de Orchestrator API en Discovery API om te zetten naar gRPC met ProtocolBuffers. Dit onderzoek heeft zich enkel gericht op de communicatie tussen de Orchestrator API en Discovery API van de Fashion Society. Toekomstig onderzoek kan gedaan worden naar de performantie in communicatie tussen de interne services en de Discovery API voor de huidige REST implementatie en een

gRPC implementatie alsook kan toekomstig onderzoek gedaan worden naar de winst in performantie bij het omschakelen naar een volledige gRPC structuur waarbij REST en JSON nog amper tot niet meer gebruikt worden. Echter zal dit laatste onderzoek in praktijk moeilijk te behalen zijn door de omvang van het huidige aantal REST services binnen de Fashion Society.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

1. Inleiding

1.1 Probleemstelling

The Fashion Society is constant aan het uitbreiden, dit onder meer door overnames en het openen van eigen nieuwe ZEB winkels. Elk van deze nieuwe winkels werkt via het centraal systeem en maakt gebruik van de Discovery API bij ondermeer het informeren van de stock, verkopen van een artikel, etc. De Discovery API is een API die aan de hand van een binnenkomende call het adres van de gevraagde API gaat ophalen om aan te spreken. Naarmate er meer winkels zijn zullen er ook meer gelijktijdige requests afgehandeld worden door de Discovery API. Door deze toename aan requests kan de performantie dalen. Met dit onderzoek wordt hierop ingespeeld en wordt gekeken of gRPC met Protocol Buffers een performanter alternatief is op de huidige implementatie.

1.2 Onderzoeksvraag

De onderzoeksvragen die gevormd worden bij de vergelijking tussen gRPC met Protocol Buffers en REST met JSON zijn als volgt.

- Welk dataformaat is performanter voor de Discovery API van The Fashion Society?
- Is gRPC met ProtocolBuffers sneller dan REST API met JSON voor de implementatie van The Fashion Society?
- Is gRPC met ProtocolBuffers efficiënter dan REST API met JSON op gebied van CPU gebruik voor de implementatie van The Fashion Society?

1.3 Onderzoeksdoelstelling

Aan de hand van een proof-of-concept wordt verwacht dat gRPC performanter zal zijn voor de Discovery API van The Fashion Society. Om deze doelstelling te behalen moet aan de onderzoeksvraag “Is gRPC met ProtocolBuffers sneller dan REST API met JSON voor de implementatie van The Fashion Society?” voldaan worden.

1.4 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk 2 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie.

In Hoofdstuk 3 wordt het praktijk onderzoek toegelicht, hierin worden de requirement analyse, plan van aanpak en de resultaten van de proof-of-concept besproken.

In Hoofdstuk 4, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

2. Stand van zaken

In de inleiding is duidelijk geworden dat het onderzoek gericht zal zijn op twee mogelijke dataformaten aan de hand van 2 verschillende structuren, namelijk JSON aan de hand van REST en gRPC aan de hand van Protocol Buffers. Om dit onderzoek volledig te kunnen begrijpen is het belangrijk om de werking en de basisprincipes van deze dataformaten en de daarbij behorende structuren te begrijpen. Om deze reden zal eerst kort uitgelegd worden wat een API is, vervolgens zal de werking en basisprincipes van JSON en RESTful API uitgelegd worden, en als volgt die van gRPC en Protocol Buffers. Alsook wordt de interne werking van de architectuur van de Fashion Society uitgelegd aan de hand van de gebruikte ontwerppatronen en een praktisch voorbeeld.

2.1 API

2.1.1 Wat is een API?

Een API of Application Programming Interface is een programma dat wordt gebruikt als tussenlaag die het mogelijk maakt voor twee applicaties om met elkaar te communiceren (**HubSpire**). Meer in detail is het een interface die de interacties tussen meerderere applicaties, bestaande uit zowel hardware als software, gaat vastleggen. Deze interacties zijn in de vorm van requests, de interface legt vast hoe deze requests moeten afgehandeld worden, welke dataformaten gebruikt moeten worden, welke richtlijnen en conventies er gevolgd moeten worden etc.

Daarnaast kan een API op verschillende manieren ontworpen worden, volledig op maat, specifiek voor een bepaalde applicatie of component of op basis van een standaard om

interoperabiliteit te kunnen garanderen.

2.1.2 Hoe werkt een API?

Een API communiceert op basis van vooropgestelde regels die bepalen hoe applicaties en machines met elkaar kunnen communiceren. In dit proces is de API een tussenlaag tussen twee applicaties of machines die met elkaar willen connecteren voor een specifieke taak.

Een simpel voorbeeld hiervan is het opzoeken van gegevens in Google Search. In de webpagina wordt een zoekterm opgegeven en eens er op de zoekknop gedrukt wordt of op enter gedrukt wordt dan stuurt de webpagina een request naar de achterliggende API die vervolgens op de servers alle gerelateerde data gaat ophalen en terug weergeven aan de webpagina.

2.1.3 Verschillende API types

Voor het onderzoek wordt er gebruik gemaakt van Web APIs, dit zijn APIs die gecontacteerd kunnen worden aan de hand van het HTTP protocol. Dit soort APIs kunnen we onderverdelen in vier categoriëen (**Stoplight**).

Open APIs

Open APIs staan ook wel bekend als publieke APIs, dit zijn APIs die toegankelijk zijn met minimale restricties voor vrijwel iedereen. Mogelijks kan een Open API registratie vereisen of het gebruik van een API key. Dit soort APIs hebben als doel om data of services toegankelijk te maken voor externe gebruikers.

Interne APIs

Interne APIs zijn APIs die gebruikt worden binnen een bepaald bedrijf voor intern resources te kunnen delen zoals data en services. Op deze manier kunnen verschillende diensten of teams binnen éénzelfde bedrijf elkaars resources gebruiken. Voordelen van interne APIs zijn beveiliging en een gestandariseerde interface voor het connecteren van meerdere services.

Partner APIs

Partner APIs zijn gelijkend aan Open APIs, echter zitten hier veel meer restricties op en worden deze gecontrolleerd door een API Gateway (hoofdstuk 2.6.2). Deze APIs worden het meest gebruikt voor doeleinden zoals het toegang verlenen tot betalende services en worden vaak gebruikt in Software as a Service.

Samengestelde APIs

De samengestelde APIs laten het toe om meerdere endpoints te contacteren aan de hand van één request. Dit soort APIs zijn vooral handig in microservice architecturen (hoofdstuk

2.6.1) in situaties waar data nodig is van meerdere services om één specifieke taak uit te voeren. Het gebruik van dit soort APIs kan voordelen opleveren zoals een vermindering in server load en verbetering van de performantie van een applicatie.

2.2 JSON

2.2.1 Algemeen

JSON is een dataformaat zoals bijvoorbeeld XML en de afkorting staat voor JavaScript Object Notation. JSON is beschreven in Standard ECMA-404 (ECMA, 2017) en is een structuur van accolades, haakjes, dubbele punten en komma's die zeer nuttig kunnen zijn in verschillende contexten, profielen en applicaties. Een voorbeeld hiervan is te zien in figuur 2.1.

```
{
  "klant": {
    "voornaam": "Sven",
    "achternaam": "Pepermans",
    "geboortedatum": "18-06-2000",
    "adres": {
      "stad": "Geraardsbergen",
      "postcode": 9506,
      "straat": "Smeerebbestraat",
      "huisnummer": 35
    },
    "aankopen": ["TV", "Computer", "GSM"]
  }
}
```

Figuur 2.1: Een voorbeeld van een JSON.

JSON mag niet gezien worden als een specificatie van een gegevensuitwisseling. Bij een zinvolle gegevensuitwisseling wordt er een overeenstemming over de structuur die gekoppeld is aan een gebruik van JSON tussen producent en consument vereist. JSON kan wel gezien worden als een syntactisch raamwerk waaraan een specifieke structuur kan worden gekoppeld.

Doordat er veel verschillende types getallen zijn zoals decimale en binaire getallen, kiest JSON enkel voor een weergave van getallen die mensen gebruiken, namelijk een reeks cijfers. Ook al zijn alle programmeertalen het niet altijd eens over de interne representaties van getallen, ze weten wel hoe ze cijferreeksen moeten begrijpen.

Objecten kunnen op veel verschillende manieren voorgesteld worden, met andere woorden

kunnen de modellen van objecten heel erg uiteenlopen. Om dit probleem aan te pakken biedt JSON een eenvoudige notatie aan voor het uitdrukken van verzamelingen met naam / waarde-paren. Om zulke verzamelingen weer te geven hebben de meeste programmeertalen reeds een functie zoals struct, map, hash en object. Daarnaast biedt JSON ook ondersteuning voor geordende zoeklijsten, alsook hiervoor hebben alle programmeertalen een functie om deze weer te geven zoals array, vector en list. Aangezien objecten en arrays zich kunnen nesten, kunnen aan de hand van JSON complexe structuren zoals boomstructuren worden gerepresenteerd.

Hieruit kan worden geconcludeerd dat door het aanvaarden van JSON's simpele conventies, complexe datastructuren uitgewisseld kunnen worden tussen wat anders incompatiebele programmeertalen zijn.

2.2.2 JSON in detail bekeken

Een JSON-tekst bestaat uit een reeks tokens die gevormd zijn uit Unicode-codepunten en die in overeenstemming zijn met de achterliggende JSON grammatica. Zo een reeks tokens bevat tekenreeksen, cijfers, zes structurele tokens en drie letterlijke naamtokens. Hieronder volgt een opsomming van de zes structurele tokens en de drie literal name tokens.

De zes structurele tokens:

-] Linker vierkante haak
-] Rechter vierkante haak
- { Linker accolade
- } Rechter accolade
- : Dubbele punt
- , Komma

De drie literal name tokens:

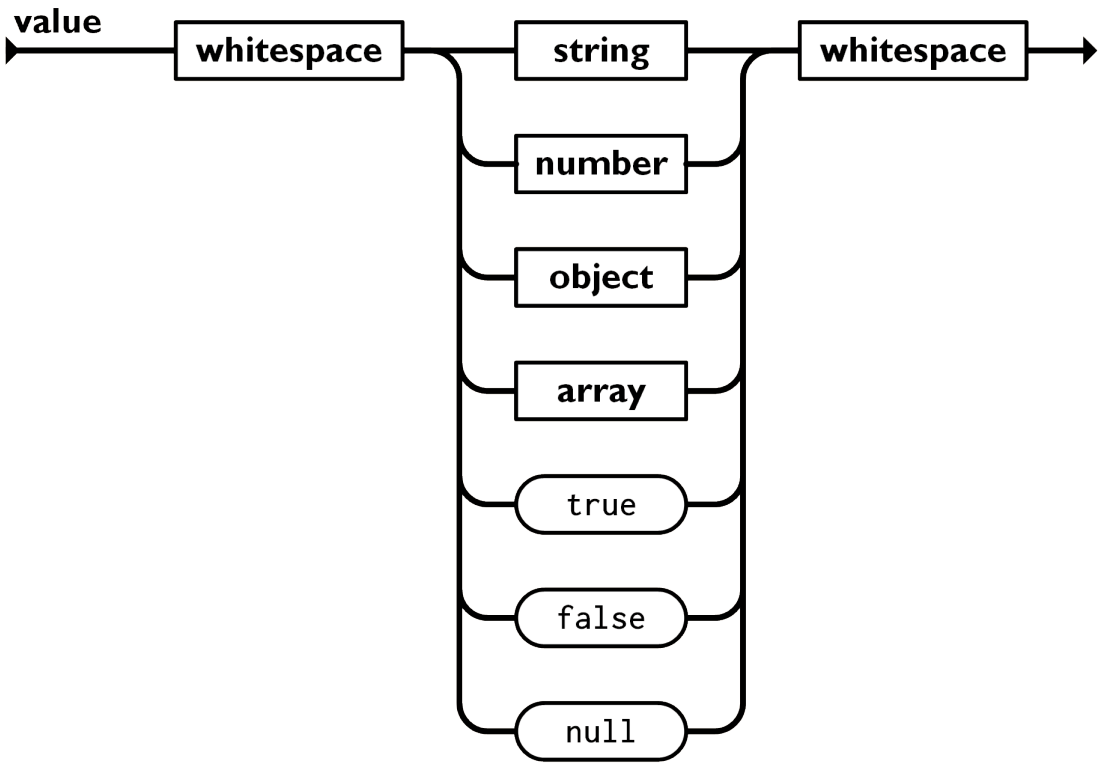
- true
- false
- null

Voor of na de meeste tokens is het toegestaan om witruimte te gebruiken, maar niet binnen in een token zelf, een spatie is hierop een uitzondering en is wel toegestaan in strings. deze witruimte kan beschreven worden als een willekeurige reeks van één of meerdere van onderstaande codepunten.

- Character tabulation
- Line feed
- Carriage return
- Spatie

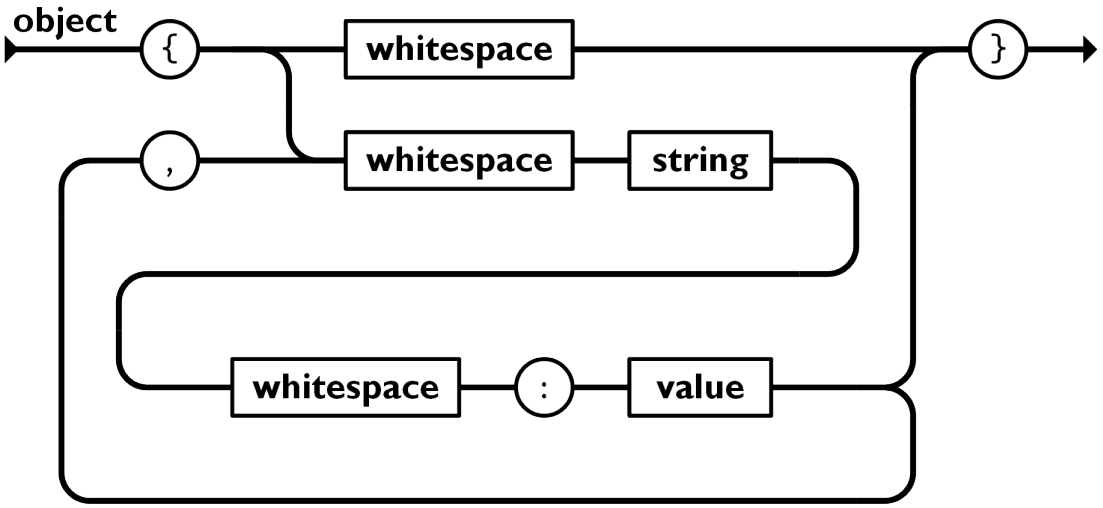
2.2.3 Waarden

De bovengestelde tokens vormen de ruggengraat van een JSON-bestand, echter is het de bedoeling dat een JSON-bestand data overdraagt. Deze data kan aan de hand van verschillende waarden worden voorgesteld, namelijk als objecten, arrays, nummers, strings, true, false en null. Alle mogelijke waarden zijn zichtbaar in figuur 2.2.



Figuur 2.2: De structuur van een JSON-waarde met alle mogelijke waarden die deze kan bevatten. (Crockford, g.d.)

De eerste waarden die besproken worden zijn de objecten, deze worden voorgesteld door een paar accolades die geen of meerdere name/value paren kunnen omringen zoals te zien is in figuur 2.3 en in figuur 2.4. In een name/value paar is de naam een string en wordt gevolgd door een dubbele punt die de naam en waarde van elkaar onderscheidt. Na de waarde kan optioneel een komma gezet worden, deze onderscheidt de waarde en de volgende naam van elkaar. Tot slot moeten de namen niet uniek zijn en moeten ze geen bepaalde ordening volgen.

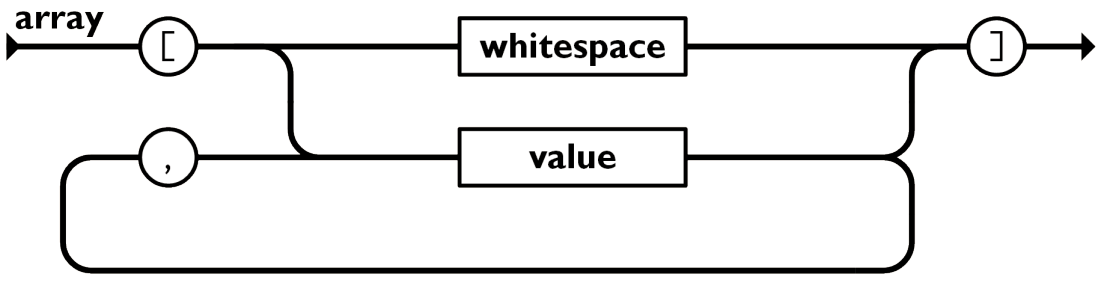


Figuur 2.3: De structuur van een JSON object. (Crockford, g.d.)

```
{
  "klant": {
    "voornaam": "Sven",
    "achternaam": "Pepermans",
    "geboorteDatum": "18-06-200",
    "klantenNummer": "KLANT01"
  }
}
```

Figuur 2.4: Voorbeeld van een JSON object.

Een tweede waarde zijn de arrays, dit zijn vierkante haken die geen of meerdere waarden omringen. Het bijzondere aan arrays is dat deze niet beperkt zijn tot een name/value paar en dus ook andere arrays kunnen bevatten en genest kunnen worden. Dit kan afgeleid worden uit figuur 2.5. De array structuur wordt net zoals bij objecten geen beperkingen opgelegd, echter worden deze vooral gebruikt in situaties waar de ordening wel enig belang heeft. Figuur 2.6 is hier een duidelijk voorbeeld van.

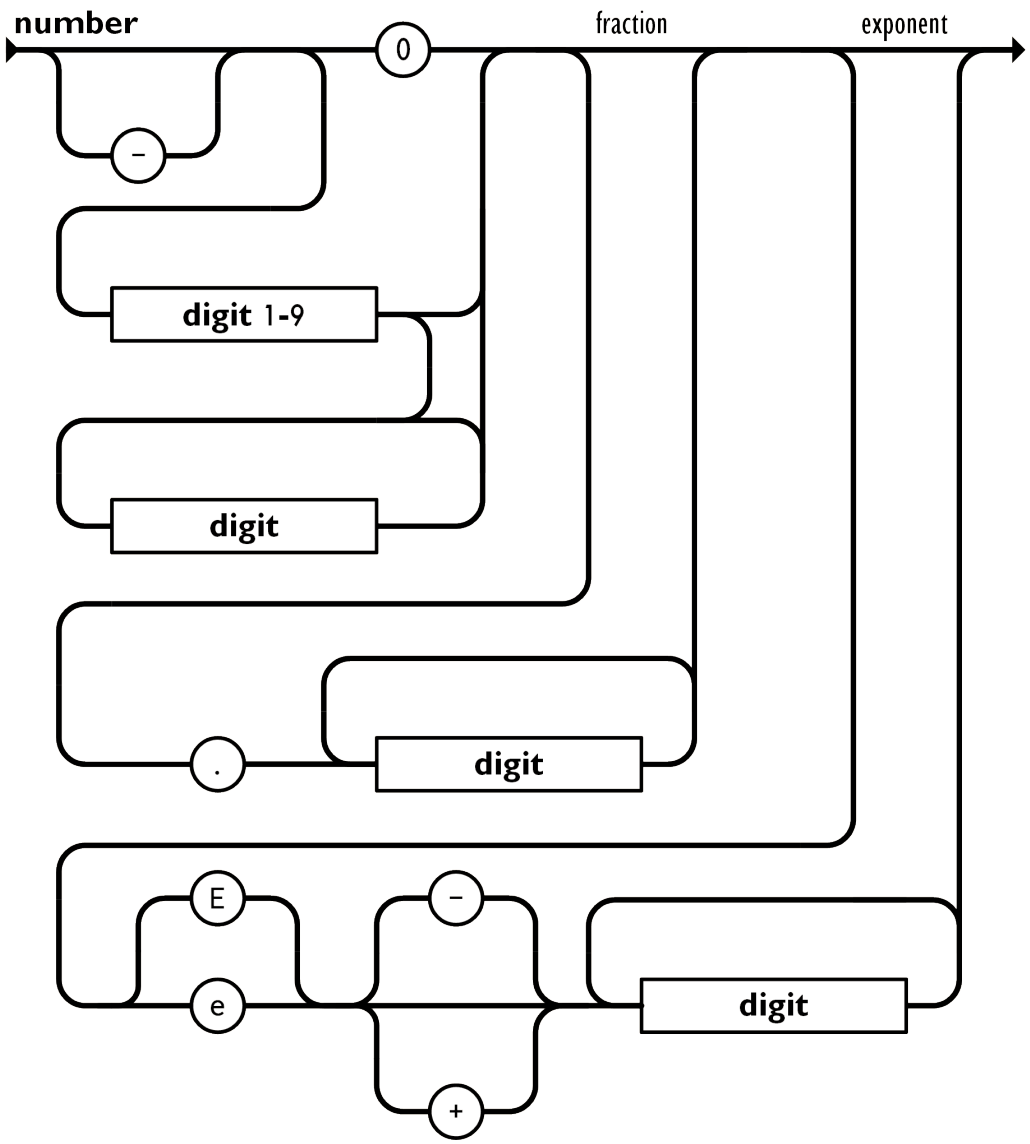


Figuur 2.5: De structuur van een JSON array.(Crockford, g.d.)

```
{
  "klanten": ["Sven", "Fleur", "Kristof"]
}
```

Figuur 2.6: Een voorbeeld van een JSON Array.

Vervolgens zijn er de nummers, een nummer kan gedefinieerd worden als een sequentie van decimale cijfers. Bij deze sequentie cijfers is er geen overbodige voorloopnul, een nummer kan wel voorafgegaan worden door een min-teken, als ook kan een nummer voorafgegaan worden door zowel een kleine als een grote e om een exponent aan te duiden dat eventueel kan worden bijgestaan door een plus- of min-teken. Om te werken met kommagetallen wordt een decimaal punt gebruikt. Deze structuur is te zien in figuur 2.7 en een voorbeeld hiervan in figuur 2.8.



Figuur 2.7: De structuur van een JSON nummer. (Crockford, g.d.)

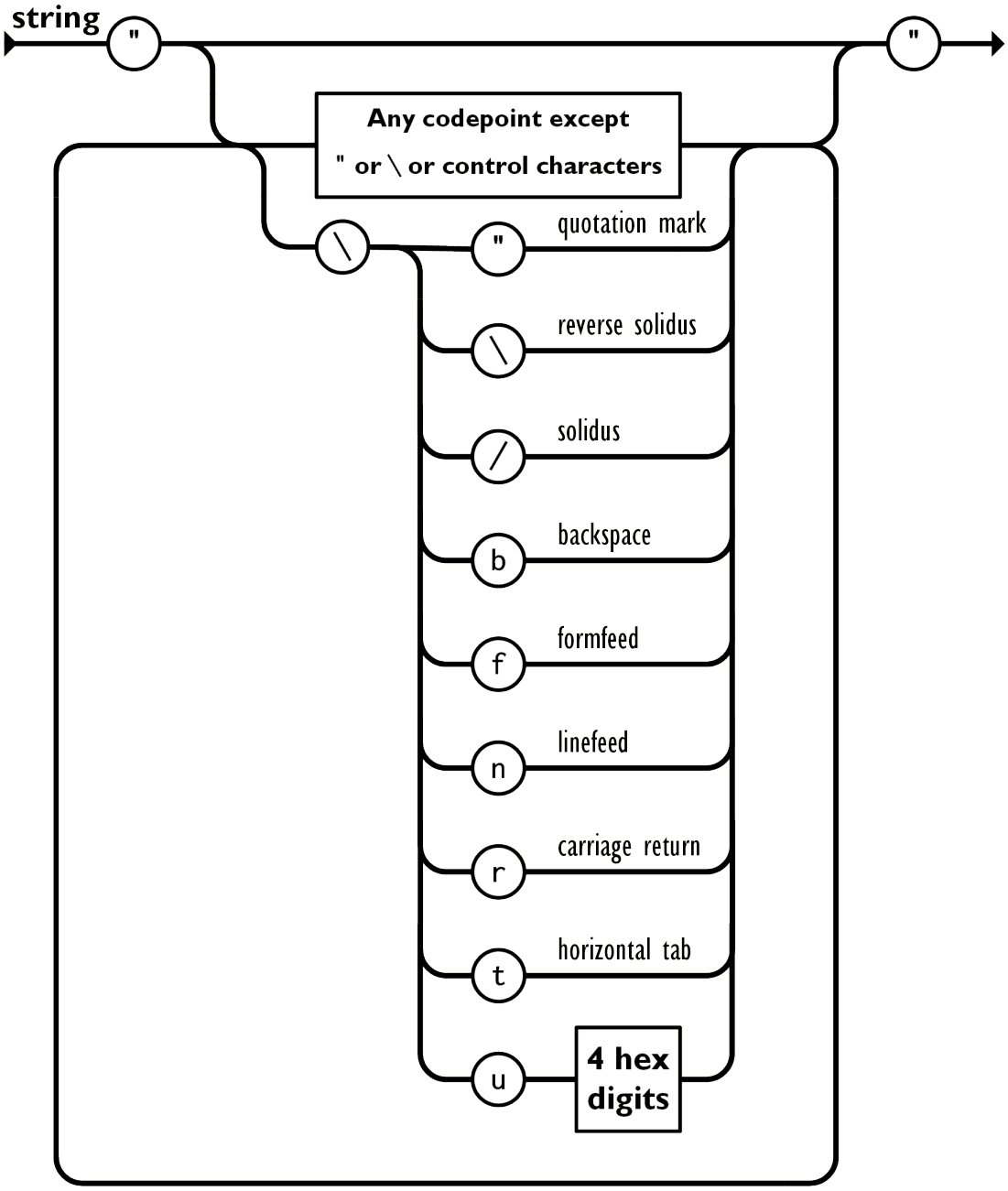
```
{
 "aantal": 35
}
```

Figuur 2.8: Een voorbeeld van een JSON nummer.

Tot slot zijn er de strings, deze representeren een tekst-waarde bestaande uit een sequentie van Unicode codepunten en zijn omringd door aanhalingstekens. Binnen deze aanhalingstekens zijn er echter enkele codepunten die niet gebruikt mogen worden, namelijk de karakters die moeten worden geëscaped. Dit zijn dan aanhalingstekens, backslash en de

control karakters. Voor sommige escape-reeksweergaven bestaande uit twee tekens bestaat er wel een representatie binnen de string, een representatie hiervan is te zien in 2.9.

Daarnaast kan ook elk codepunt weergegeven worden als een hexadecimale sequentie, waarvan de betekenis is vastgelegd in ISO/IEC 10646. Hexadecimale getallen kunnen zowel cijfers als kleine letters en hoofdletters van A tot F zijn. De codepunten die zich bevinden in het Basic Multilingual Plane worden gerepresenteerd als een sequentie van zes karakters, namelijk een backslash gevolgd door een kleine letter u, en tot slot gevolgd door vier hexadecimale getallen die een codepunt encoderen. In figuur 2.9 is de structuur van een string te zien.



Figuur 2.9: De structuur van een JSON string. (Crockford, g.d.)

2.3 RESTful API

Representational State Transfer (REST) is zoals beschreven door Fielding (2000) een architecturale stijl voor het ontwerpen van gedistribueerde hypermediasystemen.

De combinatie van REST en API is een RESTful API, wanneer een RESTful API gecalled wordt dan zal de server een representatie van de huidige staat van de gevraagde resource weergeven.

2.3.1 Design principes

Om een API als RESTful te kunnen beschrijven moet deze voldoen aan zes design principes die hieronder zullen worden verduidelijkt aan de hand van Long (g.d.) en Naeem (2021)

Client-Server

Dit principe zegt dat zowel de client als de server apart moeten kunnen worden ontwikkeld en moeten los staan van elkaar. Op deze manier wordt de handelbaarheid en schaalbaarheid aanzienlijk verbeterd aangezien de gebruikersinterface los staat van de gegevensopslag.

Stateless

Met stateless wordt bedoeld dat elke request die er naar de API gedaan wordt onafhankelijk is, een request heeft dus geen resultaat nodig van een andere request om te kunnen verder gaan. Met andere woorden moet de server geen vorige requests en states opslaan wat de benaming 'Stateless' verklaard.

Cacheable

Een REST API moet de mogelijkheid hebben om data te cachen, dit is het opslaan van data in digitaal geheugen. Volgens het Cacheable principe moet data in een response duidelijk gecategoriseerd worden als cacheable of non-cacheable. Indien een response cacheable is dan mag de cliënt cache deze data opslaan om gelijkaardige requests in de toekomst te beantwoorden.

Uniform interface

Om aan het client-server principe te voldoen is een uniforme interface nodig die autonome ontwikkeling van de applicatie mogelijk maakt zonder de acties, modellen en services te koppelen aan de API-laag. Door dit principe wordt de hele architectuur gestroomlijnd en wordt de visibiliteit van de communicatie verbeterd. Er zijn echter wel verschillende architecturale controles nodig die de performantie binnen de architectuur sturen om een uniforme interface te bereiken.

REST bevat vier interfacecontroles, namelijk:

1. Identificatie van bronnen: Een Uniform Resource Locator (URL) identificeert de

online locatie van een resource.
2. Beheer van bronnen door middel van representatie: Bronnen worden weergegeven aan de hand van media types (N. Freed, 1996) zoals JSON en XML.
3. Zelfbeschrijvende communicatie: Een bericht bevat alle informatie die de ontvanger nodig heeft om de informatie te begrijpen. Er mag geen extra informatie in een aparte documentatie of apart bericht zitten. Deze zelfbeschrijvende communicatie gebeurt door het gebruik van de accept en content-type HTTP headers die de inhoud die verzonden of aangevraagd wordt beschrijft.
4. Hypermedia: Hypermedia is data die verzonden wordt van de server naar de cliënt met informatie over wat de cliënt vervolgens kan doen. Hyper Text Markup Language (HTML) (W3schools, g.d.) is hier een voorbeeld van.

Layered system

De architectuur van een RESTful API bestaat uit verschillende lagen die door samen te werken een hiërarchie vormen die helpt bij het vormen van een flexibele, meer schaalbare applicatie. Dankzij het deze gelaagde structuur is de gevormde applicatie beter beveiligd, dit komt doordat componenten niet kunnen interageren buiten de eigen laag en de volgende laag. Daarnaast biedt het gedeelde caches aan die de schaalbaarheid verbeteren.

Deze gelaagde architectuur zorgt tot slot voor meer stabiliteit doordat het de componenten beperkt zodanig dat een component niet verder kan 'zien' als de laag waarmee het interageert.

Code on demand

Het 'Code on demand' principe zorgt ervoor dat codering kan worden gecommuniceerd via de API voor gebruik binnen de applicatie. De definitie van een RESTful API maakt het mogelijk dat functionaliteit aan de cliënt-kant kan worden uitgebreid door codering zoals applets of scripts te downloaden en te implementeren. Dit stroomlijnt cliënts door het aantal functies te verminderen die vooraf moeten worden geïmplementeerd.

Naast statische resources zoals XML en JSON kan dus ook uitvoerbare code aangeleverd worden door de server.

Resources

Bij REST kan elk stuk informatie een resource zijn, dit kan van alles zijn zoals documenten, services, collecties van andere resources, afbeeldingen, en dergelijke. In sommige gevallen wordt "Everything as a resource" een zevende principe van REST genoemd.

2.3.2 Hoe werkt een RESTful API?

Zoals eerder vermeld in subsectie 2.3.1 heeft elke resource een unieke URL, zo een URL wordt een request genoemd waarbij de gereturnde data gekend staat als een response. In een RESTful API worden deze requests of transacties in vier componenten verdeeld zoals uitgelegd in REST API (cursus van Karine Samyn, 29 maart 2020, opleidingsonderdeel

WebApplicaties IV).

- **GET**: Deze request haalt een representatie van een resource op.
- **PUT**: Met PUT wordt een bestaande resource geupdatet.
- **POST**: Aan de hand van POST kunnen nieuwe resources en sub-resources aangemaakt worden.
- **DELETE**: De DELETE request gaat een reeds bestaande resource verwijderen.

2.4 Protocol Buffers

In dit hoofdstuk zullen Protocol Buffers verduidelijkt worden, dit is het dataformaat dat gebruikt zal worden bij de implementatie van gRPC. In dit onderzoek zijn Protocol Buffers de tegenhanger van JSON.

2.4.1 Wat zijn protocol buffers?

Protocol buffers (protobuf) zijn net zoals JSON een manier om data te verzenden of op te slaan in bestanden en is ontwikkeld door Google zoals beschreven door Kurian (2020). Een voorbeeld van een protobuf is te zien in figuur 2.10.

```
syntax = "proto3";

message Person {
  uint64 id = 1;
  string email = 2;
  bool is_active = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

   message PhoneNumber {
    string number = 1;
    PhoneType type = 2;
   }

  repeated PhoneNumber phones = 4;

}
```

Figuur 2.10: Een voorbeeld van een .proto bestand. (Kurian, 2020)

Formaten zoals JSON en XML zijn zeer flexibel maar zijn niet geoptimaliseerd voor het uitwisselen van data tussen meerdere microservices (Lewis, 2014) ongeacht het gebruikte

platform. Protobuf is ontwikkeld met het oog op simpliciteit en performantie, om specifiek te zijn was het ontwikkeld om kleiner en sneller te zijn dan XML.

Protobuf onderscheidt zich van andere dataformaten door vele verantwoordelijkheden te verwijderen die normaal door het dataformaat aangepakt worden. Hierdoor gaat protobuf zich vooral focussen op het zo snel mogelijk serialiseren en deserialiseren van data. Daarnaast is een tweede optimalisatie de bandbreedte die gebruikt wordt door het zo klein mogelijk maken van de data die verzonden wordt.

Natuurlijk kunnen protobufs niet alleen voordelen hebben, een objectief nadeel dat protobuf heeft is dat het minder leesbaar is in vergelijking met JSON en XML.

Er bestaan meerdere versies van Protocol buffers, momenteel is de standaard protocol buffers versie 3 (proto3).

2.4.2 Kerneigenschappen

Binair formaat

Protobuf is een binair dataformaat, dit betekent dat de verzonden data omgezet wordt naar ééntjes en nulletjes. Deze eigenschap zorgt voor een verbetering van de transmissiesnelheid doordat het dataformaat minder bandbreedte en ruimte inneemt. Tot slot zorgt de omzetting van string naar binair formaat ook voor compressie, hierdoor zal het CPU gebruik ook lager zijn.

Splitsing van context en data

Zoals in 2.11 te zien is maakt een formaat zoals JSON gebruik van key-value paren waar de data en de context in éénzelfde bestand zitten wat ook meer ruimte in beslag neemt. Bij protobuf is dit anders, hier worden data en context opgesplitst in een configuratiebestand ook wel gekend als een .proto bestand, waarin de message wordt gedefinieerd, zie figuur 2.10, en geëncodeerde data die verzonden wordt aan de hand van het configuratiebestand.

```
{
  first_name: "Arun",
  last_name: "Kurian"
}
```

Figuur 2.11: Een simpel voorbeeld van een key-value JSON bestand. (Kurian, 2020)

2.4.3 Message formaat

Zoals hierboven vermeld is het configuratiebestand de plaats waarin de context gedefinieerd wordt, met andere woorden wordt hier de structuur van een bepaald object bepaald. Deze structuur wordt gedeclareerd door het “message” woord gevolgd door een zelfgekozen naam voor de message. Net zoals bij JSON worden de velden gedeclareerd binnen de

accolades, deze velden kunnen elk in vier delen opgedeeld worden, namelijk veldregels, veldtypes, veldnamen en veldnummers (Google, 2020).

Veldregels

In proto3 wordt enkel nog de "repeated" regel gebruikt, deze regel zegt dat de message dit veld meerdere keren mag bevatten met verschillende waarden. Met andere woorden, duidt dit op een array van een bepaald veldtype.

In proto2 waren er ook de required en optional regels. Required zorgt ervoor dat een message een bepaald veld exact één keer bevat en de optional regel zorgt ervoor dat een message een veld nul of één keer kan bevatten.

Veldtypes

Het field type is het datatype van een field, dit zijn er vier:

- **Scalaire types**: Dit zijn de traditionele types zoals strings, integers, booleans, ...
- **Enum**: Binnen een message is het mogelijk om een attribuut te hebben dat enkel één van een aantal vooraf bepaalde waarden mag hebben.
- **Message types**: Net zoals bij andere dataformaten is het mogelijk dat een message andere messages als attribuut bevat.

Veldnamen

Veldnamen moeten volledig in kleine letters zijn en indien ze uit meerdere woorden bestaan moeten deze opgesplitst worden door een underscore. In tegenstelling tot JSON wordt hier dus geen camelcasing gebruikt. Deze conventies zijn vastgelegd omdat dit ook de conventies zijn die gevolgd worden door de Protoc compiler die code gaat genereren op basis van het .proto bestand.

Veldnummer

Elk veld heeft een uniek nummer, deze nummers worden gebruikt om de velden te identificeren in het binaire formaat. Als gevolg mogen deze niet veranderd worden eens het message type reeds gebruikt is.

2.5 gRPC

2.5.1 Wat is gRPC?

gRPC is open source Remote Procedure Call (RPC) framework en is in 2015 ontstaan uit zijn voorganger Stubby, deze was een single general-purpose RPC infrastructuur (gRPC Authors, g.d.). Deze technologie laat het toe voor een programma om procedures te starten op andere computers in, eventueel, verschillende ip-ranges aan de hand van een smal comminucatiekanaal (Nelson, 1981).

2.5.2 Design Principes en requirements

Geen objecten maar services en Messages in plaats van Referenties

Hiermee is het de bedoeling om de mircoservices design philosofie van coarse-grained berichtenuitwisseling te promoten zonder de problemen van gedistribueerde objecten (Fowler, 2014) en de valkuilen van het negeren van het netwerk (Rotem-Gal-Oz, 2008) te hebben.

Coverage en Simpliciteit

gRPC moet beschikbaar zijn op elk populair ontwikkelplatform en moet gemakkelijk zijn om op te bouwen op een platform naar keuze. Alsook moet gRPC werken op toestellen die CPU en geheugen gelimiteerd zijn.

Gratis en Open

De fundamentale features van gRPC zijn gratis om te gebruiken en alle artifacten worden gereleased als open-source efforts, hierbij moet licensing er voor zorgen dat het aannemen van het framework wordt vergemakkelijkt en niet vermoeilijkt.

Interoperabiliteit en Bereik

Het wire protocol moet zonder problemen kunnen reizen doorheen de internet infrastructuur.

Algemeen Doel en Performant

De stack moet bruikbaar zijn in een grote aanbod van use-cases zonder dat performantie verloren gaat ten opzichte van specifieke stack voor die use-case.

Gelaagd

Onderdelen van de stack moeten zich individueel evolueren, bij het updaten van het wire-formaat met het vernieuwde onderdeel mogen de bindingen tussen de applicatielagen niet verstoord worden.

Payload Agnostisch

gRPC en de implementatie ervan zorgen ervoor dat elke service een andere message type en encoding systeem kan gebruiken zoals protocol buffers en JSON. Alsook zorgt gRPC er voor dat verschillende payload compressie mechanismes kunnen gebruikt worden aangezien deze kunnen verschillen per use-case.

Streaming

gRPC moet bruikbaar zijn met streaming, onder andere opslagsystemen maken hier gebruik van samen met flow control om grote data-sets uit te drukken. Er zijn ook andere services zoals voice-to-text die afhankelijk zijn van streaming voor berichtsequenties te

representeren die een tijdelijke relatie hebben met elkaar. Streaming is enkel mogelijk dankzij het gebruik van het HTTP/2 protocol (Grigorik, 2013)

Blocking en Non-Blocking

gRPC ondersteunt zowel asynchrone als synchrone processing van de data uitwisseling tussen client en server. Dit requirement is van groot belang op vlak van schaalbaarheid en het omgaan met streams.

Annulering en Timeout

Doordat sommige operaties lang en "duur" kunnen zijn op vlak van resources laat annulering het toe voor een server om de resources terug ter beschikking te zetten wanneer clients zich goed gedragen. Annulering kan ook cascaden wanneer een causale keten is gedetecteerd.

Lameducking

Een server moet zich kunnen afsluiten door reeds lopende requests af te werken maar geen nieuwe requests meer te accepteren.

Flow Control

Flow control zorgt voor betere buffer management dat kan ontstaan door een onevenwichtige balans van computing power en netwerk capaciteit tussen client en server. Als ook biedt flow control bescherming tegen DOS door een over actieve peer.

Pluggable

Een wire protocol zoals gRPC is enkel een onderdeel van een API infrastructuur. Implementaties van gRPC moeten extensiepunten voorzien voor het inpluggen van features zoals logging, health-checking, tracing,... en indien nuttig een basis implementatie van de feature.

Extensies als APIs

Extensies waarvoor een samenwerking tussen meerdere services nodig is moeten de voorkeur hebben van een API te gebruiken in plaats van protocol extensies. Voorbeelden van zulke extensies zijn health-checking en load-monitoring.

Metadata uitwisseling

Zaken zoals authenticatie zijn afhankelijk van het gebruik van metadata, dit is data die geen onderdeel is van de gegevensuitwisseling die gedefinieerd is in de interface of service.

Gestandaardiseerde statuscodes

Er zijn voor de client maar een gelimiteerd aantal manieren voor het omgaan met error die verkegen zijn van de API. Om de error handling beslissingen duidelijker te maken moet de

statuscode namespace afgebakend worden, indien er meer specifieke statussen nodig zijn kan hiervoor metadata gebruikt worden.

2.5.3 Werking van gRPC

Bij gRPC kan een client applicatie een bepaalde methode die op een remote server draait rechtstreeks aanspreken alsof het een lokaal object zou zijn, dit maakt het gemakkelijk voor het gebruiken van distribueerde applicaties en services.

gRPC definieert net zoals in vele RPC systemen, een service waarin de methodes die kunnen worden aangeroepen worden gespecificeerd samen met hun parameters en return types. Deze interface wordt op de server geïmplementeerd en draait een gRPC server om client calls af te handelen. De client heeft een stub die dezelfde methodes als de server bevat.

Op deze manier kunnen gRPC servers en clients draaien en tegen elkaar praten in verschillende omgevingen. Dit laat het bijvoorbeeld toe om een gRPC server in C# te draaien met clients die draaien in Python.

Standaard worden door gRPC Protocol Buffers gebruikt, hierbij worden de services gedefinieerd in ordinaire proto bestanden met RPC methode parameters en return types als protocol buffer messages. Aan de hand van het protoc commando samen met een gRPC plugin wordt automatisch client en server code gegenereerd op basis van het proto bestand samen met de standaard protobuf code voor het populeren, serialiseren en ophalen van de gespecificeerde message types.

2.6 Interne architectuur Fashion Society

De Fashion Society werkt met een architectuur gebouwd rond microservices, om de gehele flow binnen de architectuur te garanderen is deze gebaseerd op basis van enkele design patronen. Om de architectuur te verstaan zullen de design patronen die van toepassing zijn in deze bachelorproef kort uitgelegd worden aan de hand van hun huidige toepassing.

2.6.1 Microservice architectuur patroon

Dit patroon zegt dat de architectuur bestaat uit meerdere los van elkaar staande, samenwerkende services waarin elke service:

- Onderhoudbaar en testbaar is. Hierdoor kunnen services snel aangepast en gedeployed worden.
- Losstaand is van andere services, dit zorgt ervoor dat elke service kan draaien zonder beïnvloed te worden door veranderingen in andere services.
- Zelfstandig deployable is.
- Ontwikkelbaar door een klein team.

(Richardson, g.d.-b)

Binnen de Fashion Society wordt aan elk van deze requirements voldaan en communiceren de services aan de hand van HTTP/REST protocollen.

2.6.2 API-gateway patroon

In dit patroon wordt gedefinieerd dat de architectuur een API gateway heeft. Deze is het enige entry point voor alle clients (Richardson, g.d.-a).

Dit patroon staat bij de Fashion Society in nauwe samenwerking met het Service registry patroon en het Self Registration patroon en wordt de Orchestrator genoemd.De Orchestrator heerst en verdeelt over de micro services, elke request, inclusief de interne requests tussen APIs moeten via de Orchestrator verlopen.

2.6.3 Service Registry patroon

Het Service Registry patroon zegt dat de architectuur een service registry bevat, dit is een database van services en hun gegevens. (Richardson, g.d.-d) Bij de Fashion Society is dit de een SQL database die aan de Discovery API hangt en de servicenaam, server en poort bevat. Deze database wordt aangeroepen door de endpoints in de Discovery API. Deze aanroepingen gebeuren aan de hand van requests die de gegevens van een bepaalde service kunnen opvragen zoals de url van de locatie van een service of het volledige service object.

2.6.4 Self Registration patroon

Elke instantie van een service is zelf verantwoordelijk om bij het opstarten zichzelf te registreren bij de Service Registry. (Richardson, g.d.-c)

In de Startup klasse van elke service zit er een stuk code waarin de service zich gaan aanmelden bij de Discovery API aan de hand van een POST request zijnde RegisterService, dit gebeurt enkel wanneer er in een online omgeving wordt gewerkt zoals de test, UAT en live omgevingen, dit is om te voorkomen dat een bepaalde service al beschikbaar is in de Discovery API maar in realiteit nog niet draait.

2.6.5 SOLID Principes

De SOLID principes zijn een onderdeel van Object-Oriented Programming en zijn vijf richtlijnen voor het ontwikkelen van software zodanig dat deze eenvoudiger zijn om te onderhouden en eventueel te schalen. Hieronder een verduidelijking van elke letter in het woord SOLID. (**Thelma**)

Single Responsibility

Het eerste principe is single responsibility, dit principe zegt dat een klasse of in de context van deze bachelorproef, een micro-service maar één verantwoordelijkheid mag hebben. Naar mate een klasse of service meer verantwoordelijkheden heeft, wordt het risico bij het aanpassen van een verantwoordelijkheid groter dat een bug opduikt bij een andere verantwoordelijkheid.

Dit principe gaat gepaard met punt twee van sectie 2.6.1 dat zegt dat elke service losstaand moet zijn van andere services.

Een customer service mag dus geen data behandelen dat te maken heeft met vouchers, dat is de taak van de voucher service.

Open-Closed

Een klasse of service moet open zijn voor uitbreiding maar gesloten voor aanpassing. Het gedrag van een bepaalde endpoint in een service mag niet zomaar aangepast worden, dit heeft namelijk invloed op alle services of applicaties die deze endpoint gebruiken. Indien een endpoint van een service extra functionaliteit moet hebben is het beter om een nieuwe endpoint aan te maken die voor deze extra functionaliteit zorgt.

Het doel van dit principe is om extra functionaliteit te voorzien zonder problemen te veroorzaken bij services of applicaties waar de huidige functionaliteit gebruikt wordt.

Liskov Substitution

Als een klasse S van een bepaald subtype T is dan kunnen klasses van type T vervangen worden door klasses van type S zonder dat het invloed heeft op de propeties van het programma.

Concreet wil dit zeggen dat als een klasse overerft van een andere klasse, deze minstens hetzelfde moet kunnen doen als de ouder. Dit principe heeft als doel om consistentie te verzekeren doorheen een programma, zodanig dat een kind klasse en ouder klasse foutloos op dezelfde manier kunnen gebruikt worden.

Interface Segregation

Het is beter om meer specifieke interfaces te hebben dan één algemene interface. Een client zou geen methodes moeten implementeren die deze niet gebruikt. Het implementeren van methodes die niet gebruikt worden en dus nutteloos zijn heeft als grootste nadeel dat deze onverwachte bugs kunnen creëren.

Met andere woorden moet een klasse enkel methodes implementeren die nodig zijn voor uitvoeren van zijn taken, andere methodes die niet van toepassing zijn moeten dus verwijderd worden of verplaatst naar een andere interface indien ze gebruikt worden door een andere klasse.

Het doel hiervan is om methodes op te splitsen in meerdere interfaces zodanig dat een klasse enkel de methodes moet implementeren die hij nodig heeft.

Dependency Inversion

Tot slot is er Dependency Inversion, dependency inversion zegt dat men niet afhankelijk moet zijn van implementaties maar van abstracties. Dit kan bereikt worden door het gebruik van Dependency Injection (**Ilyana**).

Het doel van Dependency Inversion is om de afhankelijkheid van een high-level klasse op een low-level klasse te beperken. Beide moeten afhankelijk zijn van abstracties.

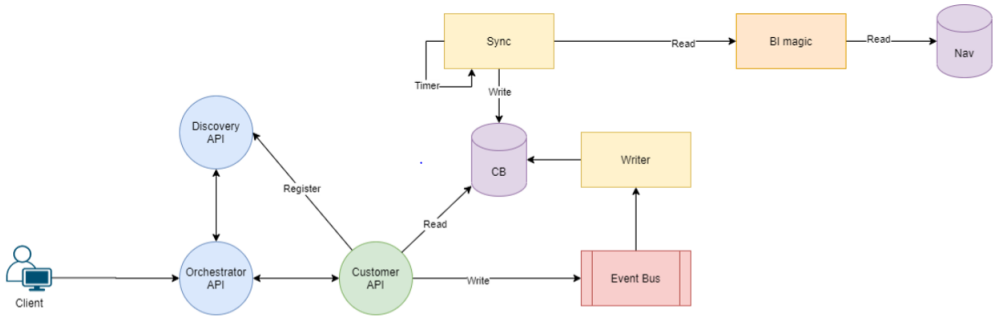
2.6.6 Algemene werking voorbeeld

Een personeelslid van team Customer Service wil graag een klant opzoeken om deze verder te helpen. Hij geeft de gegevens in in een tool die klanten ophaalt, dit is nu de Client.

De Client komt binnen bij de Orchestrator met de request om een bepaalde klant op te halen, in deze request zit ook de naam van de benodigde API, zijnde de CustomerAPI. Vervolgens stuurt de Orchestrator een request naar de DiscoveryAPI voor de gegevens van de gevraagde API, de DiscoveryAPI zal deze request beantwoorden met de server en poort waarop de CustomerAPI te vinden is.

Vervolgens kopieert de Orchestrator de originele request van de Client naar een nieuwe call die wordt verzonden naar de CustomerAPI op basis van de verkregen server en poortnummer. Deze call zal dan beantwoord worden met de juiste klantgegevens en gaat via de Orchestrator terug tot bij de Client.

Dit gehele proces kan gevolgd worden in figuur 2.12



Figuur 2.12: De interne werking voor het afhandelen van klantengegevens

3. Praktijk onderzoek

In hoofdstuk 2 is het onderzoek gestart met een literatuurstudie waarin besproken werd wat REST, JSON, gRPC en Protobufs exact zijn en hoe de interne architectuur van Fashion Society er uitziet.

In dit hoofdstuk gaat het onderzoek verder met een requirementanalyse en een plan van aanpak waarin beschreven wordt hoe de proof-of-concept tot stand komt en wat de resultaten hiervan zijn.

3.1 Requirementanalyse

In deze analyse worden de functionele en niet-functionele vereisten opgelijst waaraan een alternatieve implementatie van de Discovery API moet voldoen.

Functionele requirements:

- Implementatie aan de hand van .NET Core 3.1 voor long-term support.
- Moet een grote hoeveelheid aan parallelle requests kunnen afhandelen.

Niet-functionele requirements:

- Moet (relatief) gemakkelijk uitbreidbaar zijn.
- Moet snel requests kunnen afhandelen.
- Moet schaalbaar zijn.

3.2 Plan van aanpak

3.2.1 Opstellen van gRPC service in .NET Core 3.1

Uit de requirementanalyse kan afgeleid worden dat de service zal moeten draaien in .NET Core 3.1 en niet in .NET Core 5.0 omdat deze geen long-term support heeft van Microsoft.

Zodanig dat gebruik gemaakt kan worden van de voordelen van gRPC zoals codegeneratie zal het project opgebouwd worden vanuit een grpc service project vanuit Visual Studio zelf, dit project bevat de Grpc.AspNetCore package die alles gRPC gerelateerd afhandelt.

Voor de logica in de twee endpoints GetService en GetServiceUrl die respectievelijk een service object en een service url ophalen zal code gerecupereerd worden van de reeds bestaande Discovery API.

De reeds bestaande Discovery API maakt gebruik van het repository patroon voor het ophalen en verwerken van data uit een SQL database in de gRPC service zal dit patroon wegvallen en zal de databasecontext rechtstreeks in de Service geïnjecteerd worden. Uit een verdere analyse van de bestaande code kan er ook geconcludeerd worden dat beide methoden gebruik maken van éénzelfde Get methode in de repository, deze wordt een private methode in de Service die zal worden aangesproken door beide endpoints.

Het protobestand “services.proto” waaruit de bijhorende modellen worden gegenereerd bevat de 2 rpc methodes GetService en GetServiceUrl, beiden hebben deze een input parameter van het messagetype “ServiceName” met als veld een string “name”. De GetService methode zal een message van het type “ServiceObject” teruggeven, deze bevat de “name”, “address”, “port”, “addNameToUrl” en “hasSwaggerDoc” velden. Deze velden zijn licht afwijkend van het reeds bestaande Model in de Discovery API maar is in samenspraak met mijn co-promotor. Tot slot is er de GetServiceUrl methode, deze geeft een message van het type “ServiceUrl” terug dewelke een enkel veld bevat zijnde “url”.

In figuur 3.1 is het gebruikte protobestand te zien.

```
syntax = "proto3";

option csharp_namespace = "DiscoverygRPC";

serivce Service{
  rpc GetService(ServiceName) returns (ServiceObject);
  rpc GetServiceUrl(ServiceName) returns (ServiceUrl);
}

message ServiceName{
  string name = 1;
}

message ServiceObject{
  string name = 1;
  string address = 2;
  int32 port = 3;
  bool addNameToUrl = 4;
  bool hasSwaggerDoc = 5;
}

message ServiceUrl{
  string url = 1;
}
```

Figuur 3.1: Het gebruikte protobestand

3.2.2 Client applicaties voor REST en gRPC

Om de snelheid van beide implementaties te testen moeten de services worden aangesproken van een client. Hiervoor worden twee identieke implementaties gebruikt in twee Console applicaties (.NET Core). De enige verschillen tussen de twee applicaties zijn de aangesproken endpoints. Deze applicaties zullen aan de hand van een lijst van vier servicenamen calls doen naar de service. Er is voor vier servicenamen gekozen zodanig dat er rekening wordt gehouden met eventuele verschillen in het ophalen van de verschillende data.

Tijdens het gehele proces zal een timer lopen die de totale tijd van de calls zal opnemen.

3.2.3 Het verloop van het onderzoek

Het onderzoek wordt uitgevoerd in drie categorieën, een kleine en grote payload voor het resultaat van parallele requests en een request loop van tienduizend requests voor de response time voor één request te berekenen. In de kleine payload zullen er in totaal vierduizend calls gemaakt worden, elke servicenaam zal tien keer parallel gebruikt worden om vervolgens honderd requests te versturen. In de grote payload zullen er tienduizend calls gemaakt worden en hier zal elke servicenaam vijfentwintig keer parallel gebruikt worden om vervolgens honderd requests te versturen.

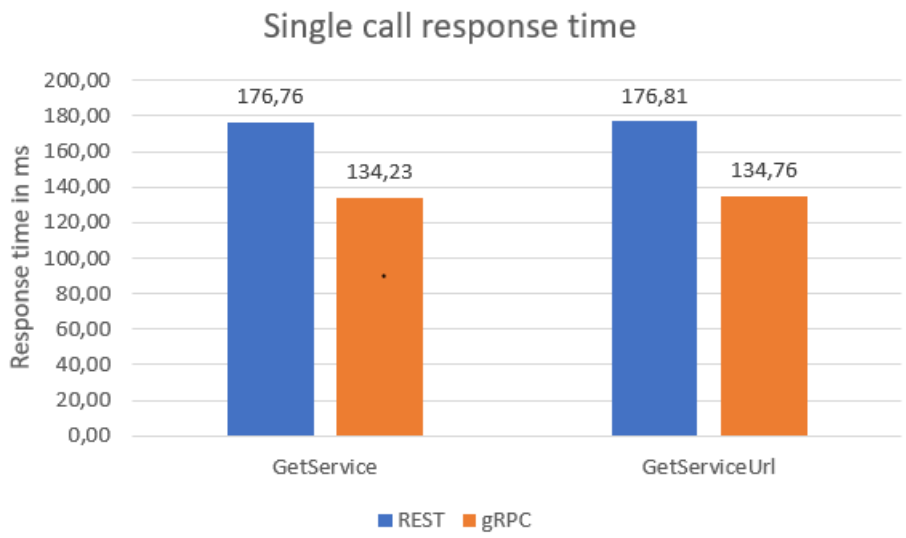
Om tot een concreet resultaat van één request te bekomen zal ook deze tienduizend keer worden uitgevoerd. Dit hoge aantal is gekozen om de invloed van eventuele netwerk vertragingen te beperken. De kleine en grote payload zullen daarintegen vier keer uitgevoerd worden om tot een concreet resultaat te bekomen. Hier zullen eventuele netwerk vertragingen om individuele calls minder doorwegen door het reeds hoge aantal calls in elke uitvoering.

Het CPU gebruik zal manueel in de gaten gehouden worden maar zal in samenspraak met de co-promotor niet in acht genomen worden in de resultaten omdat er geen accurate manier beschikbaar is om het CPU gebruik op de server te registreren.

3.3 Resultaat Proof-Of-Concept

3.3.1 Single Call

Uit het onderzoek is gebleken dat gRPC met ProtocolBuffers voor zowel de GetService methode (-24,06%) als de GetServiceUrl methode (-23,78%) sneller is dan REST met JSON.

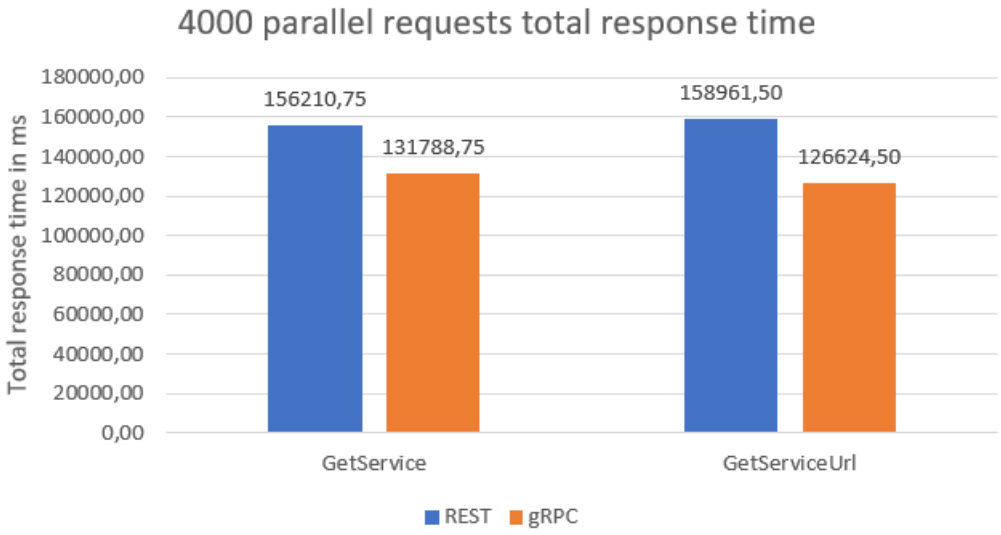


Figuur 3.2: Resultaat van het onderzoek naar Single call response time.

De verwachting van dit onderdeel was dat het resultaat bij de single call response time een marginaal verschil ging zijn in voordeel van gRPC doordat de response op de calls een kleine hoeveelheid data is voor beide methodes, echter toonde het resultaat een groter dan verwacht verschil in snelheid in voordeel van gRPC.

3.3.2 Vierduizend Parallelle Calls

Uit het onderzoek is gebleken dat gRPC voor zowel de GetService methode (-15,63%) als voor de GetServiceUrl methode (-20,34%) sneller is dan REST bij het afhandelen van vierduizend parallelle calls.

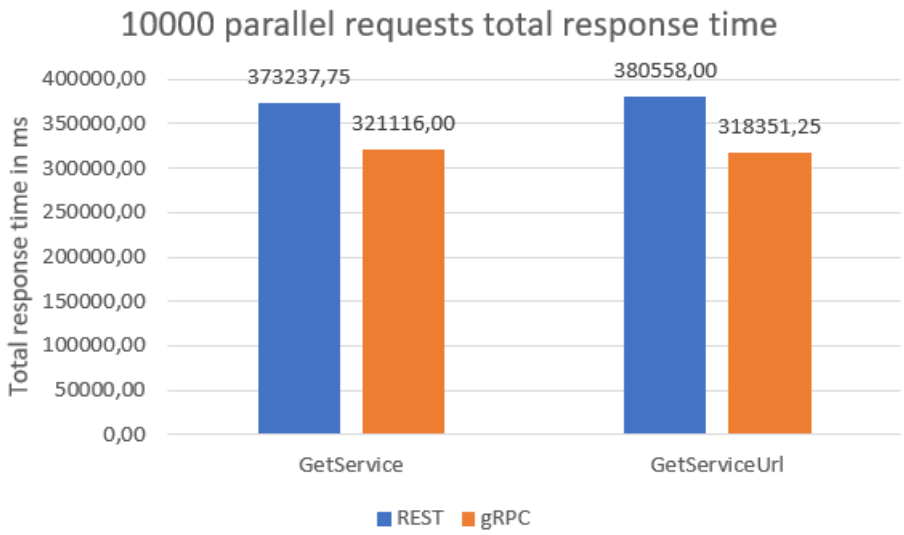


Figuur 3.3: Resultaat van het onderzoek naar een kleine hoeveelheid parallelle calls.

Bij dit onderdeel is het resultaat onder de verwachtingen, de verwachtingen waren dat gRPC opmerkelijk sneller ging zijn dan REST door de hoeveelheid aan af te handelen requests. Echter is het gebleken dat dit een verkeerde redenering was voor de verwachting, de hoeveelheid data in de response op elke call blijft namelijk nog altijd klein.

3.3.3 Tienduizend Parallelle Calls

Uit het onderzoek is gebleken dat gRPC voor zowel de GetService methode (-13,96%) als voor de GetServiceUrl methode (-16,35%) sneller is dan REST bij het afhandelen van tienduizend parallelle calls.



Figuur 3.4: Resultaat van het onderzoek naar een grote hoeveelheid parallelle calls.

Net zoals bij de kleine hoeveelheid parallelle calls is het resultaat hier onder de verwachtingen. gRPC is voor zowel de GetService methode (-13,96%) als voor de GetServiceUrl methode (-16,35%) sneller dan REST maar ook hier is voor de verwachting dezelfde verkeerde redenering gebruikt als bij de kleine hoeveelheid parallelle calls. Het aantal parallelle calls is wel verhoogd maar de hoeveelheid data in de response op de calls blijft nog altijd hetzelfde.

4. Conclusie

De opzet van dit onderzoek is om een antwoord te geven op de onderzoeksvragen “Welk dataformaat is performanter voor de Discovery API van de Fashion Society?”, “Is gRPC met ProtocolBuffers sneller dan REST API met JSON voor de implementatie van de Fashion Society?” en “Is gRPC met ProtocolBuffers efficiënter dan REST API met JSON op gebied van CPU gebruik voor de implementatie van de Fashion Society?”. De conclusie is afhankelijk van de drie uitgevoerde testen, single call response time, totale response tijd voor vierduizend parallelle calls en totale response tijd voor tienduizend parallelle calls, die voor de methode GetService en GetServiceUrl zijn uitgevoerd.

Uit het onderzoek kunnen twee van de drie onderzoeksvragen beantwoord worden. Uit het onderzoek kan geconcludeerd worden dat gRPC met ProtocolBuffers sneller is dan REST met JSON voor de implementatie van The Fasion Society en dat ProtocolBuffers performanter zijn dan JSON voor de implementatie van de Fashion Society. Deze conclusies zijn getrokken uit de resultaten van de proof-of-concept waaruit blijkt dat gRPC met Protobufs in zowel single call, kleine payload als grote payload sneller is dan REST met JSON.

Op de onderzoeksvraag “Is gRPC met ProtocolBuffers efficiënter dan REST API met JSON op gebied van CPU gebruik voor de implementatie van de Fashion Society?” kan geen antwoord gegeven worden omdat deze gegevens niet exact meetbaar waren gedurende het uitvoeren van de proof-of-concept. Hierbij is in overleg met de co-promotor het besluit gevormd dat de Fashion Society een snel groeiend bedrijf is waarbij performantie belangrijk is en dat als er meer CPU gebruik zou zijn dit een prijs is dat het bedrijf gewillig is om te betalen in ruil voor een performantere implementatie.

Dankzij dit onderzoek heeft de Fashion Society een beter inzicht in zowel gRPC en Protobufs zelf als in de effecten die het heeft op de performantie bij het overschakelen.

Hierdoor kan de Fashion Society een gewogen beslissing maken om al dan niet over te schakelen naar gRPC met Protobufs.

De vastgestelde antwoorden op de onderzoeksvragen waren zoals verwacht, echter zoals vermeld in 3.3 waren de vastgestelde waarden waarmee gRPC en Protobufs sneller waren niet zoals verwacht. De verwachting lag veel hoger in het voordeel van gRPC met Protobufs een mogelijke reden hiervoor is dat gRPC met Protobufs vooral voordeliger is bij grote data en het streamen hiervan. Dit vloeit tot een nieuwe vraag die uitnodigt tot eventueel verder onderzoek, namelijk "Is het verschil in snelheid tussen gRPC met Protobufs en REST met JSON groter in het voordeel van gRPC en Protobufs voor grotere data?". Deze mogelijke onderzoeksvraag past samen met onderzoek naar de performantie verschillen in communicatie tussen de interne services en de Discovery API voor de huidige REST implementatie van de Fashion Society en een gRPC implementatie. De communicatie tussen de interne services en de Discovery API wisselen onderling heel veel data uit dewelke kan variëren van 2 properties tot een geneste structuur die meerdere niveaus diep gaat. Deze communicatie zou dus een ideaal scenario zijn om het verschil in snelheid te testen.

A. Onderzoeksvoorstel

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

A.1 Introductie

Zonder dat het te weten of te beseffen maakt de internetgebruiker continue gebruik van dataformaten en protocollen. Deze komen in verschillende soorten, XML, JSON, Protocol Buffers, GraphQL, ... Deze data formaten zorgen aan de hand van een achterliggende implementatie in een Application Programming Interface (API) voor dat mensen hun vrienden hun nieuwe Facebook of Instagrampost kunnen zien in een internetbrowser of op een app. APIs maken het meest gebruik van XML en JSON, indien toch een dominerend formaat gekozen moet worden zal dit JSON zijn. JSON is veel sneller om te lezen en te schrijven dan XML.

Echter wil dit niet zeggen dat JSON het snelste data formaat is. De Fashion Society is een moederbedrijf met verschillende kledingsketens onder zich. Om alles vlot te laten verlopen wordt gebruik gemaakt van een centraal beheersysteem voor zowat alles dat nodig is, dit houdt in maar is niet beperkt tot pickorders, webshops, ... Dit interne systeem maakt gebruik van het Façade patroon, dit is een structuur waar er maar één toegangspunt is tot een collectie ojecten met diensten en services. In het Façade patroon van de Fashion Society is de Orchestrator het toegangspunt tot een collectie van meerdere API's, deze krijgt een request binnen voor een bepaalde service of dienst maar weet niet naar welke hij deze request moet doorsturen. Om dit op te lossen wordt gebruik gemaakt van de Discovery API, de Orchestrator zal deze aanroepen en de Discovery zal vervolgens de

juiste Service ophalen en terugsturen naar de Orchestrator zodanig dat deze de originele request kan doorsturen naar de correcte API.

De Fashion Society wordt alsmaar groter en is op zoek naar een manier om de Discovery API performanter te laten wezen om zo meer realtime requests te kunnen afhandelen. De huidige Discovery API is een REST API dat gebruik maakt van JSON. In deze Bachelorproef zal onderzocht worden of Protocol Buffers ge¨ımplementeerd aan de hand van gRPC eventueel een even performant of zelfs performanter alternatief kan zijn voor de huidige implementatie van de Fashion Society dat gebruik maakt van een REST API met JSON. De bijhorende onderzoeksvragen zullen dan ook als volgt zijn:

- Is gRPC met ProtocolBuffers sneller dan een REST API met JSON?
- Is gRPC met ProtocolBuffers efficiënter dan een REST API met JSON?

A.2 State-of-the-art

gRPC is open source Remote Procedure Call (RPC) framework en is in 2015 ontstaan uit zijn voorganger Stubby, deze was een single general-purpose RPC infrastructuur (gRPC Authors, g.d.). Deze technologie laat het toe voor een programma om procedures te starten op andere computers in, eventueel, verschillende adresruimten aan de hand van een smal comminucatiekanaal (Nelson, 1981). gRPC zelf is geen dataformaat zoals JSON maar kan gebruik maken van verschillende data formaten, standaard is dit Protocol buffers, ook wel Protobufs genoemd.

JSON en Protocol Buffers hebben zowel gelijkenissen als grote verschillen, beide zijn een language-neutral dataformaat zo blijkt uit de documentatie van Protocol Buffers (Google, g.d.) en JSON (ECMA, 2017). Het grootste, visuele verschil voor de gebruiker is te vinden in de structuur. JSON is een collectie van naam/value paren of een geordende lijst van waarden, daarintegen maken Protobufs gebruik van een soort model, er moet maar eenmaal gedefinieerd worden hoe de data gestructureerd zal zijn, daarna kan gebruik gemaakt worden van gegenereerde code om gemakkelijk data van en naar een variëteit van data streams te lezen en schrijven. Dit kan allemaal geprogrammeerd worden in heel wat verschillende programmeertalen.

gRPC wordt reeds gebruikt door verschillende grote bedrijven zoals Cisco en Netflix. Cisco gebruikt gRPC als het ideale, uniforme transportprotocol voor modelgestuurde configuratie en telemetrie. Dit komt dankzij de ondersteuning voor hoogwaardige bidirectionele streaming, op TLS gebaseerde beveiliging en het brede scala aan programmeertalen. Netflix koos dan weer voor gRPC omdat ze belang hechtte aan de architectonische kennis in de IDL (proto) dat een onderdeel is van gRPC en aan de, van deze proto-afgeleide, codegeneratie. Daarnaast speelde ook de cross-language compatibility en codegeneratie in gRPC een belangrijke rol bij de keuze voor gRPC bij Netflix (Foundation, 2018).

Eerder werd er nog geen officieel onderzoek uitgevoerd naar dit onderwerp, echter zijn wel online artikels te vinden die gRPC met REST gaan vergelijken aan de hand van een benchmark. Eén zo een onderzoek is uitgevoerd door Fernando (2019). Dit onderzoek was

een benchmark tussen gRPC met Protocol Buffers en REST met JSON concludeerd dat gRPC sneller was dan REST behalve bij het streamen van data, hier is gRPC iets trager als REST.

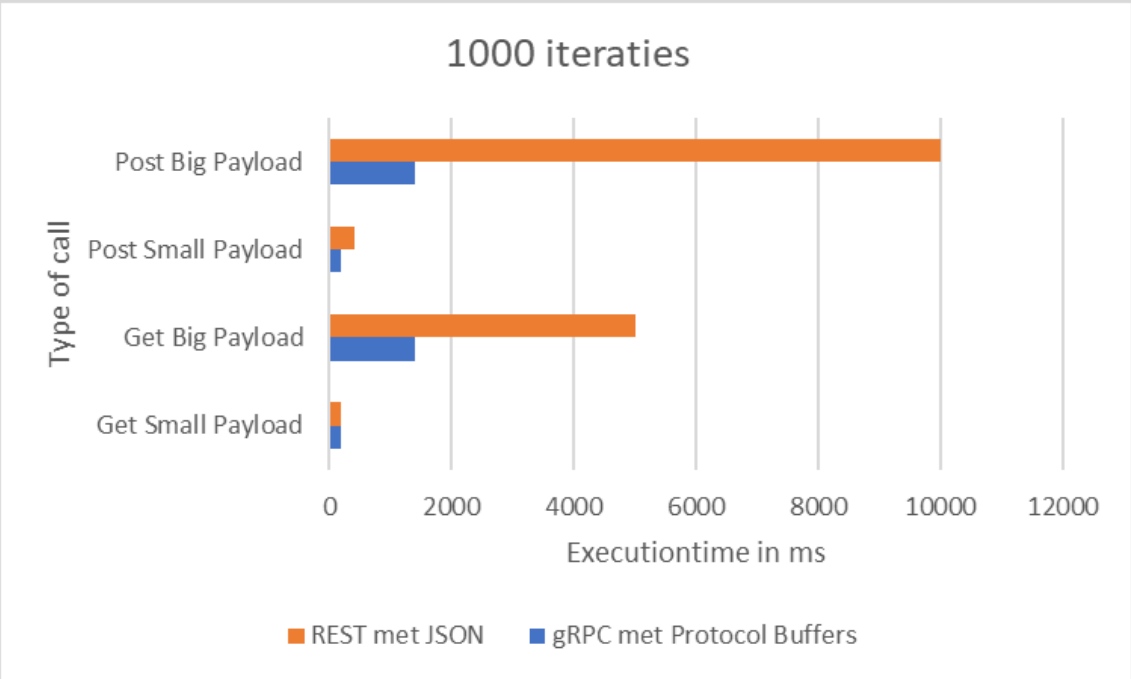
Het onderzoek dat in deze Bachelorproef zal uitgevoerd worden is gebaseerd op dat van Ruwan Fernando, hiermee wordt bedoeld dat deze proof-of-concept geprogrammeerd zal worden in C# en dat de verschillende impelementaties met elkaar vergeleken zullen worden aan de hand van een benchmark. Het verschil met Fernando R. zijn onderzoek kan men vinden in de verwerking van de data, in dit onderzoek zal de data uit een aanhangende databank gehaald worden. Alsook zal in dit onderzoek een tienvoud van het aantal iteraties doen.

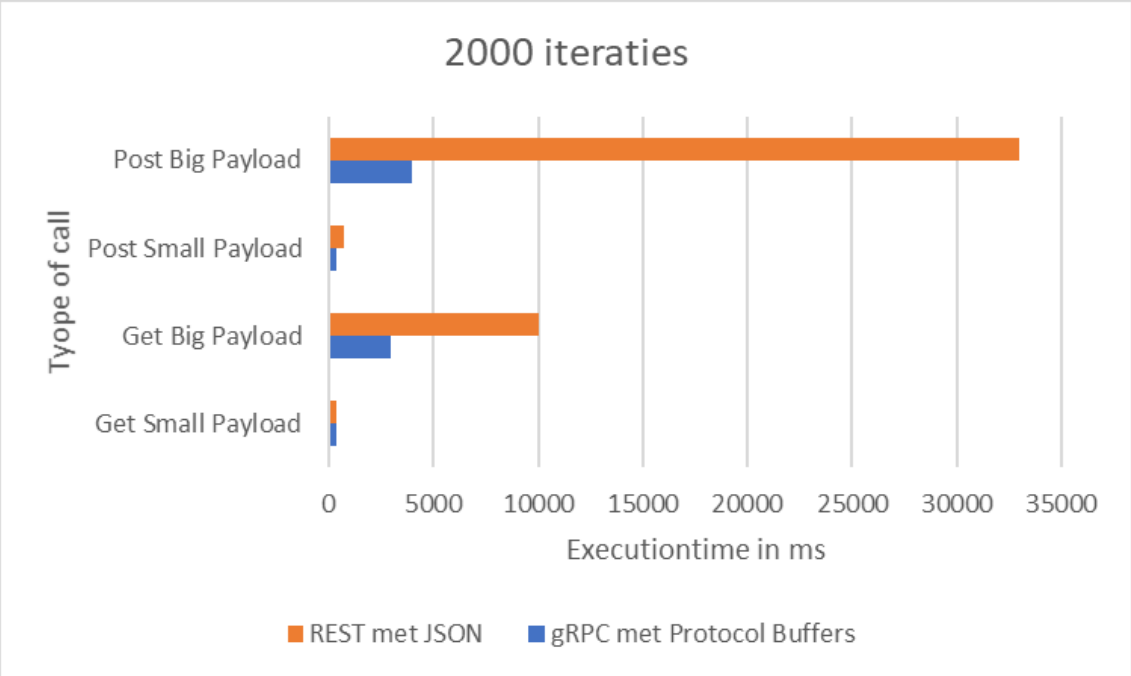
A.3 Methodologie

Het onderzoek zal gevoerd worden aan de hand van simulaties en experimenten in een Proof-of-Content (PoC). Er zullen 2 identieke APIs opgesteld worden in C#, een .NET Core REST API voor JSON, en .NET Core API voor gRPC. Aan de hand van een benchmark tool, dewelke nog niet specifiek gekozen is, zullen er 2 verschillende categorieën aan GET en POST calls uitgevoerd worden. Big payload calls, hier zal substantief meer data opgehaald en verzonden worden dan bij de Small payload calls. Daarnaast zal het experiment meerdere malen uitgevoerd worden met 1000 en 2000 iteraties, dit om inaccurate metingen van kleine uitvoertijden te verminderen en een duidelijke conclusie te kunnen vormen. Eens de resultaten van de experimenten verzameld zijn zullen we deze vergelijken en conclusies trekken.

A.4 Verwachte resultaten

Bij de grote payloads wordt een duidelijk verschil in voordeel van gRPC en Protocol Buffers verwacht, echter wordt er wel ook verwacht dat REST API met JSON in de kleine payloads nog degelijk zal presteren en misschien zelf nog licht overwegend beter zal zijn dan gRPC en Protocol Buffers.





A.5 Verwachte conclusies

Dit onderzoek moet doen blijken of gRPC en Protocol Buffers sneller en efficiënter zijn dan een REST API met JSON. In de PoC zal duidelijk worden dat gRPC en Protocol Buffers een sneller en efficiënter alternatief zijn voor JSON. Dit resultaat zou eventueel verklaard kunnen worden doordat JSON een ouder dataformaat is en dus niet geoptimaliseerd is voor de huidige technologieën terwijl gRPC zeer recent ontworpen is

om optimaal met de huidige technologieën om te gaan.

B. Copyright Notice Ecma International

Bibliografie

Crockford, D. (g.d.). *Introducing JSON*. Verkregen 4 april 2021, van https://www.json.org

ECMA. (2017). *The JSON Data Interchange Syntax*. ECMA International. Verkregen 16 december 2020, van https://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-404%201st%20edition%20October%202013.pdf

Fernando, R. (2019, april 3). *Evaluating Performance of REST vs. gRPC*. https://medium.com/@EmperorRXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures* (proefschrift). University of California, Irvine.

Foundation, C. N. C. (2018, december 4). *Netflix: Increasing developer productivity and defeating the thundering herd with gRPC*. https://www.cncf.io/case-studies/netflix/

Fowler, M. (2014, augustus 13). *Microservices and the first law of distributed objects*. Verkregen 11 april 2021, van https://martinfowler.com/articles/distributed-objects-microservices.html

Google. (g.d.). *Protocol Buffers*. Google. https://developers.google.com/protocol-buffers

Google. (2020, december 17). *Protocol Buffers Language guide*. https://developers.google.com/protocol-buffers/docs/overview

Grigorik, I. (2013, september 1). Introduction to HTTP/2. *High Performance Browser Networking*. https://developers.google.com/web/fundamentals/performance/http2

gRPC Authors. (g.d.). *gRPC, A high-performance, open source universal RPC framework*. https://grpc.io/

Hubspire. (g.d.). *Application Programming Interface*. Verkregen 21 maart 2021, van https://www.hubspire.com/resources/general/application-programming-interface/

Kurian, A. M. (2020, april 22). *Understanding Protocol Buffers*. https://betterprogramming.pub/understanding-protocol-buffers-43c5bced0d47

Lewis, M. F. J. (2014, maart 25). *Microservices*. https://martinfowler.com/articles/microservices.html

Long, L. (g.d.). *What RESTful actually means* (A. M. A. Vakil, Red.). Verkregen 21 maart 2021, van https://codewords.recurse.com/issues/five/what-restful-actually-means

N. Freed, N. B. (1996). *Multipurpose Internet Mail Extensenions (MIME) Part One: Format of Internet Message Bodies*. Verkregen 14 maart 2021, van https://tools.ietf.org/html/rfc2045

Naeem, T. (2021, februari 4). *REST API Definition: What is REST API (RESTful API)?* Astera. Verkregen 21 maart 2021, van https://www.astera.com/type/blog/rest-api-definition/

Nelson, B. J. (1981). *Remote Procedure Call* (proefschrift). Carnegie-Mellon University. https://ia801900.us.archive.org/22/items/bitsavers_xeroxparctoteProcedureCall_14151614/CSL-81-9_Remote_Procedure_Call.pdf

Richardson, C. (g.d.-a). *Patten: API-gateway*. Verkregen 18 april 2021, van https://microservices.io/patterns/apigateway.html

Richardson, C. (g.d.-b). *Pattern: Microservice architecture*. Verkregen 18 april 2021, van https://microservices.io/patterns/microservices.html

Richardson, C. (g.d.-c). *Pattern: Self Registration*. Verkregen 18 april 2021, van https://microservices.io/patterns/self-registration.html

Richardson, C. (g.d.-d). *Pattern: Service Registry*. Verkregen 18 april 2021, van https://microservices.io/patterns/service-registry.html

Rotem-Gal-Oz, A. (2008, januari 1). *Fallacies of Distributed Computing Explained*. https://www.researchgate.net/publication/322500050_Fallacies_of_Distributed_Computing_Explained

W3schools. (g.d.). *HTML Introduction*. Verkregen 21 maart 2021, van https://www.w3schools.com/html/html_intro.asp