

Tom Van der Cruyssen

Professionele Bachelor Elektronica-ICT, afstudeerrichting ICT

Academiejaar 2022/2023

**Welke API-structuur is het meest geschikt om Kreonta
Edge te refactoren?**

Kreonta

Schoolweg 11

1750 Lennik

Auteur

Tom Van der Cruyssen

Opleiding

Elektronica-ICT, afstudeerrichting Elektronica

Academiejaar

2022/2023

Interne promotor

Serge Fabre

In samenwerking met

Kreonta

Schoolweg 11

1750 Lennik

Externe promotor

Hannes Vanderheyden

Abstract

Welke API-structuur is het meest geschikt voor het refactoren van Kreonta Edge ?

T. Van der Cruyssen

Het doel van dit onderzoek is het analyseren en vergelijken van verschillende API-structuren om de performantie van de applicatie Kreonta Edge te verhogen. Hierbij wordt niet enkel de snelheid in rekening gebracht, maar ook het geheugengebruik en het gemak om code te implementeren. De vergeleken API-structuren zijn REST, SOAP, gRPC en GraphQL.

Het onderzoek begint met een literatuurstudie om een inschatting te maken van de mogelijkheden van de verschillende structuren. De theoretische sterktes en zwaktes worden opgesomd. De literatuurstudie vormt tegelijk een kennismaking met de gebruikte terminologie.

In het praktische gedeelte van het onderzoek wordt een testomgeving opgezet die een beeld geeft van zowel de client-side als de server-side. Een aantal kernonderdelen van Kreonta Edge worden herschreven in de bovenvermelde API-structuren. De ervaringen van deze implementatie worden per API-structuur beschreven.

Tot slot worden voor elke API-structuur testen uitgevoerd waarbij de uitvoeringstijd en het geheugengebruik worden gemeten.

Uit de metingen blijkt gRPC als meest geschikte architectuur naar voor te komen voor Kreonta Edge, dit wegens zijn hoge snelheid en laag geheugengebruik. De mogelijkheid om streaming toe te passen binnen gRPC, is hierbij een doorslaggevende factor.

Voorwoord

Dit onderzoek vloeit voort uit mijn stage, die ik in het kader van mijn professionele bacheloropleiding Elektronica-ICT aan de Odisee Hogeschool heb gelopen bij het bedrijf Kreonta.

Het doel van dit onderzoek was om een grondige analyse te maken van het refactoren van de applicatie Kreonta Edge. In deze bachelorproef worden verschillende aspecten van mijn project belicht, waarbij de lezer inzicht krijgt in verschillende soorten API's en hun praktische implementaties.

De keuze voor dit specifieke project komt voort uit mijn wens om me in mijn toekomstige carrière te specialiseren in API's. Ik wilde onderzoeken wat de voor- en nadelen zijn van verschillende API-architecturen en hoe deze zich vertalen naar de context van deze specifieke applicatie. Het was een uitdagende taak waarbij ik veel kennis en ervaring heb opgedaan.

Tijdens het implementeren van verschillende architecturen stuitte ik echter op enkele obstakels. Met name het coderen van SOAP en GraphQL verliep moeizaam. Kleine details die vaak over het hoofd worden gezien, kunnen leiden tot foutieve code en frustraties.

Graag wil ik mijn oprechte dank uitspreken aan iedereen die heeft bijgedragen aan het tot stand komen van dit project. Allereerst wil ik mijn ouders bedanken voor hun onschatbare steun, zowel bij het nalezen van mijn schrijfwerk als bij hun geduld gedurende mijn studie. Daarnaast gaat mijn dank uit naar Serge Fabre, mijn begeleider tijdens deze bachelorproef, om mij te motiveren en gezonde druk te zetten tijdens dit traject. Ik wil ook Kreonta bedanken voor het openstellen van hun software voor dit onderzoek. Ten slotte ben ik Johan Donné dankbaar voor het beantwoorden van mijn vele vragen gedurende mijn opleiding.

Lennik, mei 2023

Tom Van der Cruyssen

Inhoudsopgave

Figurenlijst.....	5
Tabellenlijst	6
Codefragmentenlijst.....	7
Afkortingenlijst	8
Inleiding	9
1 Kreonta Edge	10
1.1 Omschrijving Kreonta Edge	10
1.2 Belangrijke klassen	11
1.2.1 DataGatheringProcessService	11
1.2.2 DataTransmissionProcessService	13
1.2.3 Controller klassen.....	14
2 Soorten API's	15
2.1 SOAP	15
2.2 REST	16
2.3 GRPC	17
2.4 GraphQL.....	20
2.5 Overzicht	22
2.6 Conclusie	23
3 Testomgeving	24
3.1 GetDataAPI	25
3.2 Client	25
3.2.1 ExecutionTime	26
3.2.2 Memory	26
3.2.3 AddEndpointCall.....	27
4 Proof of concept verschillende API's.....	28
4.1 REST Situatie As Is	28
4.1.1 Server	28
4.1.2 Client	28
4.1.3 Conclusie	29
4.2 POC SOAP	30
4.2.1 Server	30
4.2.2 Client	30
4.2.3 Conclusie	32
4.3 POC GRPC	32
4.3.1 Server	32

4.3.2	Client	34
4.3.3	Streaming	35
4.3.4	Conclusie	35
4.4	POC GraphQL.....	35
4.4.1	Server	35
4.4.2	Client	38
4.4.3	Conclusie	38
5	Testresultaten	40
5.1	REST.....	40
5.2	SOAP.....	43
5.3	GRPC.....	46
5.4	GRAPHQL.....	49
5.5	DataTransmission.....	52
	Conclusie	55
	Literatuurlijst.....	56

Figurenlijst

Figuur 1: software architectuur Kreonta.....	10
Figuur 2: properties van een "reading" object.....	12
Figuur 3: SOAP architectuur [1].....	16
Figuur 4: REST architectuur [3].....	17
Figuur 5: GRPC architectuur [4].....	19
Figuur 6: GraphQL architectuur [5]	21
Figuur 7 Testomgeving Schema	24
Figuur 8 Solution Explorer GetDataAPI	25
Figuur 9 Client Form	25
Figuur 10 Build opties GRPC.....	34
Figuur 11 Grafiek Testresultaten REST.....	40
Figuur 12 Grafiek Uitvoeringstijd FindWebServiceById REST	42
Figuur 13 Grafiek Testresultaten SOAP	43
Figuur 14 Grafiek Vergelijking Uitvoeringstijd REST-SOAP	44
Figuur 15 Grafiek Vergelijking Geheugen REST-SOAP	44
Figuur 16 Grafiek Uitvoeringstijd SOAP	45
Figuur 17 Grafiek Testresultaten GRPC.....	46
Figuur 19 Grafiek Vergelijking Uitvoeringstijd REST-GRPC	Error! Bookmark not defined.
Figuur 20 Grafiek Vergelijking Geheugen REST-GRPC.....	47
Figuur 21 Grafiek Uitvoeringstijd GRPC	48
Figuur 22 Grafiek Testresultaten GRAPHQL.....	49
Figuur 23 Grafiek Vergelijking Uitvoeringstijd REST-GRAPHQL	50
Figuur 24 Grafiek Vergelijking Geheugen REST-GRAPHQL.....	50
Figuur 25 Grafiek Uitvoeringstijd GRAPHQL	51
Figuur 26 Grafiek Vergelijking prestaties DataTransmission	52
Figuur 27 Grafiek Uitvoeringstijd DataTransmissionGRPCStreaming	53
Figuur 28 Grafiek Uitvoeringstijd DataTransmissionREST.....	54

Tabellenlijst

Tabel 1: Properties reading object	12
Tabel 2: Voor- en nadelen API architecturen	22
Tabel 3 Testresultaten REST	40

Codefragmentenlijst

Codefragment 1: DataGatheringProcessService klasse	11
Codefragment 2: DataTransmissionProcessService klasse	13
Codefragment 3: WebserviceController klasse.....	14
Codefragment 4: GetAllWebServicesGraph.....	25
Codefragment 5: CalculateSize	27
Codefragment 6: AddEndpointCall	27
Codefragment 7: REST endpoint aanspreken	29
Codefragment 8: SOAP endpoint mapping	30
Codefragment 9 SOAP IWebServiceController	30
Codefragment 10: SOAP Query.....	30
Codefragment 11: SOAP endpoint aanspreken	31
Codefragment 12: ConvertXmlToWebServiceDTO	32
Codefragment 13: GRPC WebService proto-bestand	33
Codefragment 14: GRPC endpoint aanspreken	34
Codefragment 15: GRAPHQL endpoint mapping	36
Codefragment 16: GRAPHQL-schema	36
Codefragment 17: GRAPHQL WebServiceInputType	37
Codefragment 18: GRAPHQL endpoint aanspreken	38
Codefragment 19: GRAPHQL QueryResponse	38

Afkortingenlijst

API	Application Programming Interface
REST	Representational state transfer
SOAP	Simple Object Acces Protocol
gRPC	Remote Procedure Calls
IDL	Interface Definition Language
OPC UA	Open Platform Communications Unified Architecture
XML	Extensible Markup Language
SSL	Secure Socket Layer
TLS	Transport Layer Security

Inleiding

Deze bachelorproef bestaat erin om te onderzoeken welke API-structuren het meest geschikt zijn om de performantie van de applicatie Kreonta Edge, momenteel gebaseerd op een REST API, te verhogen.

Kreonta Edge is de API van Kreonta Connect, een webapplicatie die data uit een productieomgeving ontvangt en verwerkt voor analyse en rapportering. De data worden in realtime of met bepaalde intervallen doorgestuurd. De huidige applicatie voldoet momenteel nog aan de verwachtingen, maar een hogere snelheid zou een meerwaarde bieden.

Verschillende API-structuren worden onderzocht en met elkaar vergeleken. Naast de snelheid is ook de complexiteit relevant in dit onderzoek.

Het onderzoek wordt opgesplitst in een theoretisch en een praktisch gedeelte. In het theoretische gedeelte wordt in hoofdstuk 1 Kreonta Edge beschreven en de werking uitgelegd. In hoofdstuk 2 wordt op basis van een literatuurstudie onderzocht welke verschillende API-structuren het meeste potentieel hebben om de prestaties van Kreonta Edge te verbeteren. De structuren SOAP, REST, gRPC en GraphQL vormen het onderwerp van de literatuurstudie.

In het praktische gedeelte worden de bovenvermelde API-structuren toegepast op een aantal kernonderdelen van Kreonta Edge. In hoofdstuk 3 wordt de testomgeving besproken. Het refactoren wordt geëvalueerd in hoofdstuk 4 op basis van de hoeveelheid zelf te schrijven code, de beschikbare functies en de syntax. Tot slot worden de prestaties vergeleken en geëvalueerd in hoofdstuk 5 op basis van de applicatiesnelheid en het geheugengebruik. Deze worden gemeten tijdens simulaties van een productieomgeving.

Op basis van de bovenvermelde evaluatie wordt er een aanbeveling gedaan voor de beste API-structuur voor de Kreonta Edge API.

1 Kreonta Edge

1.1 Omschrijving Kreonta Edge

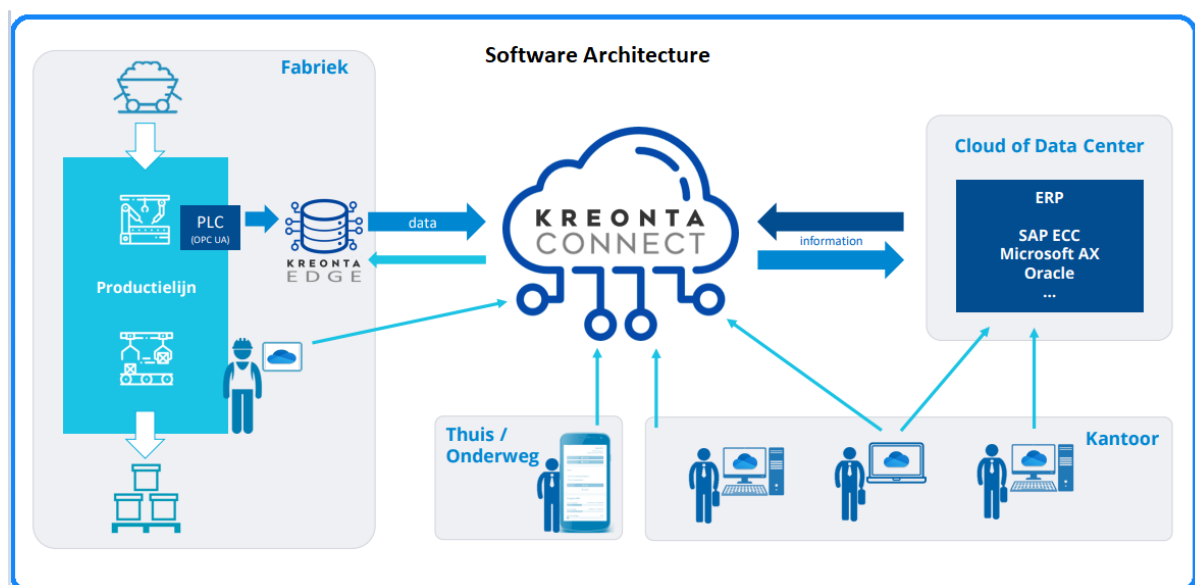
Kreonta Edge fungeert als de backend en API voor Kreonta Connect, waarmee gebruikers toegang hebben tot de data van hun productielijnen via een webbrowser. De data worden eerst doorgestuurd vanaf de productielijnen naar Kreonta Edge, waar ze worden verwerkt en opgeslagen in een MySQL-database.

Kreonta Edge is het eerste punt van contact voor de data van de productielijnen. Voor deze bachelorproef wordt de Reference server gebruikt. Dit is een OPC simulatieserver, die wordt gebruikt om de data van de productielijnen te simuleren. Deze data worden vervolgens via OPC UA-communicatie (een protocol voor het communiceren met industriële apparatuur) naar Kreonta Edge gestuurd. Kreonta Edge verzamelt en verwerkt de data en slaat deze op in een MySQL-database. Kreonta Edge fungeert ook als een interface voor andere systemen die toegang nodig hebben tot de data.

De internetbrowser is het eindpunt voor de gebruikers en biedt de mogelijkheid om de data te bekijken en te bewerken via gepaste endpoints. Via de browser kunnen gebruikers ook opdrachten geven aan Kreonta Edge om specifieke acties uit te voeren met betrekking tot de data.

De database-laag fungeert als de opslagplaats voor de data die is verzameld door Kreonta Edge. Hier worden alle gegevens opgeslagen in een MySQL-database voor later gebruik en bewerking.

In essentie werkt de API van Kreonta Edge als een interface tussen de verschillende lagen van de applicatie. De Edge-laag verzamelt en verwerkt de data van de productielijnen en slaat deze op in de database, terwijl de browserlaag toegang biedt tot deze data en de mogelijkheid biedt om deze te bewerken en opdrachten te geven aan de Edge-laag. De API maakt het mogelijk om op een gestructureerde en gecontroleerde manier toegang te krijgen tot de data van de productielijnen en deze te gebruiken in verschillende toepassingen.



Figuur 1: software architectuur Kreonta

1.2 Belangrijke klassen

Om te verduidelijken hoe Kreonta Edge werkt, worden hierna de belangrijkste klassen beschreven. Het gaat om de volgende klassen:

- DataGatheringProcessService
- DataTransmissionProcessService
- InstantTransmission
- Alle controller klassen (WebService als voorbeeld)

1.2.1 DataGatheringProcessService

DataGatheringProcessService is een klasse die verantwoordelijk is voor het verzamelen van gegevens van OPC-servers.

```
11 references
public class DataGatheringProcessService
{
    private readonly LoggingService _logger;
    19 references
    public static List<Connection> Connections { get; set; } = new List<Connection>();
    2 references
    private ApplicationConfiguration Configuration { get; set; }
    private readonly Timer reconnectTimer = new(60000);

    private readonly string conn = new ConnectionString().connectionString;

    private readonly InstantTransmission _IT;

    1 reference
    internal uint Write(string serverid, string tagid, uint value) {...}

    0 references
    public DataGatheringProcessService(OpcApplicationService opcAppService, LoggingService logger) {...}

    1 reference
    private void ReconnectTimer_Elapsed(object sender, ElapsedEventArgs e) {...}

    1 reference
    public void InitConnections() {...}

    2 references
    public void AddConnection(string ServerId) {...}

    2 references
    private Server FindServerById(string id) {...}

    3 references
    public async void UpdateConnection(string serverid) {...}

    4 references
    private void MonitoredItem_Notification(MonitoredItem monitoredItem, MonitoredItemNotificationEventArgs e) {...}
    1 reference
    public void RemoveConnection(string serverId) {...}

    2 references
    private void AddReadingToTag(string tagid, Reading r) {...}

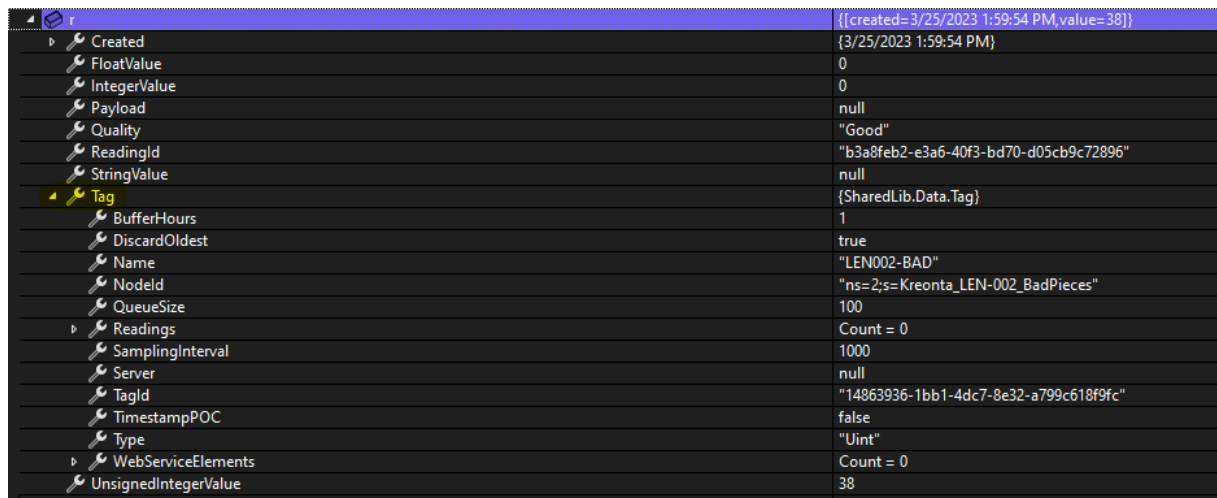
    1 reference
    private List<Tag> GetTagsForNodeId(string serverid, NodeId nodeId) {...}

    1 reference
    private void AddWebServiceValuesForReading(Reading reading, string tagId) {...}
}
```

Codefragment 1: DataGatheringProcessService klasse

Wanneer er data van de productie komen, wordt dit gedetecteerd in de methode “MonitoredItem_Notification”. Hier worden de data omgezet naar een Reading object.

Aan deze reading wordt ook een tag toegekend zodat we weten van welke productielijn de data komen. Daarna wordt er gekeken of er een webservice geabonneerd is op deze tag en worden de data toegevoegd aan deze webservice zodat deze doorgestuurd kunnen worden naar de client.



Created	{/created= 3/25/2023 1:59:54 PM,value= 38}}
FloatValue	{3/25/2023 1:59:54 PM}
IntegerValue	0
Payload	0
Quality	null
ReadingId	"Good"
StringValue	"b3a8feb2-e3a6-40f3-bd70-d05cb9c72896"
Tag	{SharedLib.Data.Tag}
BufferHours	1
DiscardOldest	true
Name	"LEN002-BAD"
NodeId	"ns=2;s=Kreonta_LEN-002_BadPieces"
QueueSize	100
Readings	Count = 0
SamplingInterval	1000
Server	null
TagId	"14863936-1bb1-4dc7-8e32-a799c618f9fc"
TimestampPOC	false
Type	"UInt"
WebServiceElements	Count = 0
UnsignedIntegerValue	38

Figuur 2: properties van een "reading" object

Hierboven staat een voorbeeld van een reading die verstuurd is vanuit de productielijn LEN002. Deze reading is gebonden aan de tag "BadPieces" en vertelt ons dat op dit moment productielijn LEN002 38 foute stukken heeft gemaakt.

Tabel 1 legt uit welke variabelen er gebruikt worden.

Tabel 1: Properties reading object

Variable naam	Type	Uitleg
Created	DateTime	Deze variabele houdt bij wanneer de reading is aangekomen.
FloatValue	Float	Als de waarde van het type Float is, komt hier de waarde in.
IntegerValue	Integer	Als de waarde van het type Integer is, komt hier de waarde in.
Payload	String	Deze variabele bevat eventuele extra informatie over de ontvangen data.
Quality	String	Deze variabele geeft de status van de reading weer.
ReadingId	GUID	Dit is de unieke waarde van een reading zodat deze geïdentificeerd kan worden.
StringValue	String	Als de waarde van het type String is, komt hier de waarde in.
UnsignedIntegerValue	UInt	Als de waarde van het type UInt is, komt hier de waarde in.

1.2.2 DataTransmissionProcessService

Een webservice heeft twee mogelijkheden, onmiddellijk data doorsturen of data een bepaalde tijd bijhouden en dan doorsturen. Hiervoor zijn twee klassen "DataTransmissionProcessService" en "InstantTransmission". Het enige verschil tussen beide klassen is dat InstantTransmission geactiveerd wordt bij het ontvangen van de data en dat DataTransmission geactiveerd wordt na het verloop van de geselecteerde timer.

```
0 references
public class DataTransmissionProcessService
{
    private readonly LoggingService _logger;
    private readonly Timer _timer = new Timer(1);
    private readonly List<DestinationController> _controllers = new List<DestinationController>();
    private readonly List<WebServiceTimer> _timers = new List<WebServiceTimer>();
    private readonly string conn = new ConnectionString().connectionString;
    private int i = 0;
    0 references
    public DataTransmissionProcessService(LoggingService logger) {...}

    0 references
    public DataTransmissionProcessService(LoggingService logger, bool instant) {...}

    1 reference
    private void Timer_Elapsed(object sender, ElapsedEventArgs e) {...}

    1 reference
    public void GetCallReady() {...}

    3 references
    private void SendCall(WebService ws, WebServiceCall callparam, DestinationController destController) {...}

    1 reference
    private void SaveWebServiceCall(WebServiceCall call, WebServiceCallExecution execution = null) {...}

    1 reference
    private void UpdateAllTimers() {...}

    2 references
    public void UpdateTimer(Guid webServiceId) {...}

    5 references
    private class DestinationController {...}

    3 references
    private class RestController {...}
}
```

Codefragment 2: DataTransmissionProcessService klasse

Dit codefragment 2 definieert de klasse "DataTransmissionProcessService". De klasse is bedoeld om gegevens over te dragen via een reeks van webservices. De klasse maakt gebruik van een "Timer" die elke seconde afgaat en de methode "Timer_Elapsed" aanroept. Deze methode haalt gegevens op uit de database. Vervolgens wordt er voor elke webservice een call gestuurd naar de bestemming die is opgegeven in de database.

Er wordt een "DestinationController" gemaakt voor elke bestemming. Als er al een controller bestaat voor de bestemming, wordt de bestaande controller gezocht en gebruikt voor het versturen van de call. Vervolgens worden voor alle webservices alle onafgewerkte calls opgehaald. Als de optie "KeepSequence" is ingeschakeld, wordt alleen de oudste onafgewerkte call verstuurd. Anders worden alle onafgewerkte calls verstuurd.

Voor elke call wordt de methode "SendCall" aangeroepen, die de webservice ophaalt en de call verstuurt naar de juiste bestemming.

1.2.3 Controller klassen

Aangezien er redelijk wat controller klassen zijn wordt de controller klasse “WebServiceController” als voorbeeld genomen.

Een controller klasse fungeert als een tussenpersoon tussen de client en de database of andere services. Het bevat de logica voor het verwerken van verzoeken, zoals het ophalen, updaten of verwijderen van gegevens en het terugsturen van de resultaten aan de client.

```
public class WebServiceController : ControllerBase
{
    private readonly DataTransmissionProcessService _dtps;
    private readonly LoggingService _logger;
    private readonly string conn = new ConnectionString()._connectionString;
    0 references
    public WebServiceController(DataTransmissionProcessService dtps, LoggingService logger) {...}

    [HttpGet]
    0 references
    public ActionResult<List<WebServiceDTO>> GetAllWebServices() {...}

    [HttpGet("{id}")]
    1 reference
    public ActionResult<WebServiceDTO> FindWebServiceById(Guid id) {...}

    [HttpPost]
    [Authorize(Roles = UserRoles.Kreonta + ", " + UserRoles.Admin)]
    0 references
    public ActionResult<WebServiceDTO> UpdateWebService(WebServiceDTO req) {...}

    [HttpDelete("{id}")]
    [Authorize(Roles = UserRoles.Kreonta + ", " + UserRoles.Admin)]
    0 references
    public ActionResult<string> RemoveWebService(Guid id) {...}

    [HttpGet("{id}/element")]
    0 references
    public ActionResult<List<WebServiceElementDTO>> GetElementsForWebService(Guid id) {...}

    [HttpGet("{id}/calls")]
    0 references
    public ActionResult<List<WebServiceCallDTO>> GetCallsForWebService(Guid id) {...}

    [HttpPost("{id}/calls/filtered")]
    0 references
    public ActionResult<CallsFilterResponse> FilterCallsForWebService(Guid id, CallsFilterRequest req) {...}

    [HttpGet("{id}/calls/{amt}")]
    0 references
    public ActionResult<List<WebServiceCallDTO>> GetLatestCallsForWebService(Guid id, int amt) {...}

    [Authorize(Roles = UserRoles.Kreonta + ", " + UserRoles.Admin)]
    [HttpPost("{id}/element")]
    0 references
    public ActionResult<WebServiceElementDTO> AddElementToWebService(Guid id, WebServiceElementDTO req) {...}
}
```

Codefragment 3: WebserviceController klasse

Elke functie waarbij HttpGet staat is een functie waarbij de client data opvraagt aan Kreonta Edge. Bij “GetAllWebServices” vraagt de client alle webservices op die opgeslagen zijn in de databank. Het is ook mogelijk voor de client om gedetailleerde acties uit te voeren, zoals bij “FindWebServiceById”. Hier staat HttpGet(“{id}”) wat ervoor zorgt dat de client een ID kan meegeven, de API stuurt dan de specifieke webservice met die ID terug. Het is ook mogelijk voor de client om zelf data in de databanken toe te voegen, of bestaande data te wijzigen. Dit kan door middel van HttpPost, zoals gebruikt bij de functie “UpdateWebService”. De client kan ook, indien gemachtigd, data uit de databank verwijderen met HttpDelete zoals bij de functie “RemoveWebService”.

2 Soorten API's

2.1 SOAP

SOAP (Simple Object Access Protocol) is een berichtenprotocol waarmee toepassingen via het internet met elkaar kunnen communiceren. Het is een protocol dat communicatie op afstand tussen softwaretoepassingen via het internet mogelijk maakt, en het wordt gebruikt om gestructureerde gegevens uit te wisselen tussen client en server. [8]

SOAP is gebaseerd op XML (Extensible Markup Language), en gebruikt een gestandaardiseerde reeks regels voor het coderen van gegevens in een berichtformaat dat kan worden begrepen door elke softwaretoepassing die SOAP ondersteunt. Het berichtformaat bestaat uit een pakket dat informatie bevat over het bericht en de gegevens die worden uitgewisseld. [8]

Een SOAP-bericht bevat doorgaans een kopgedeelte dat kan worden gebruikt om details over het bericht te specificeren, zoals de authenticatie-informatie, en een bodygedeelte dat de uitgewisselde gegevens bevat. Het bodygedeelte kan elk type gegevens bevatten, waaronder tekst, afbeeldingen of binaire gegevens. [1][7]

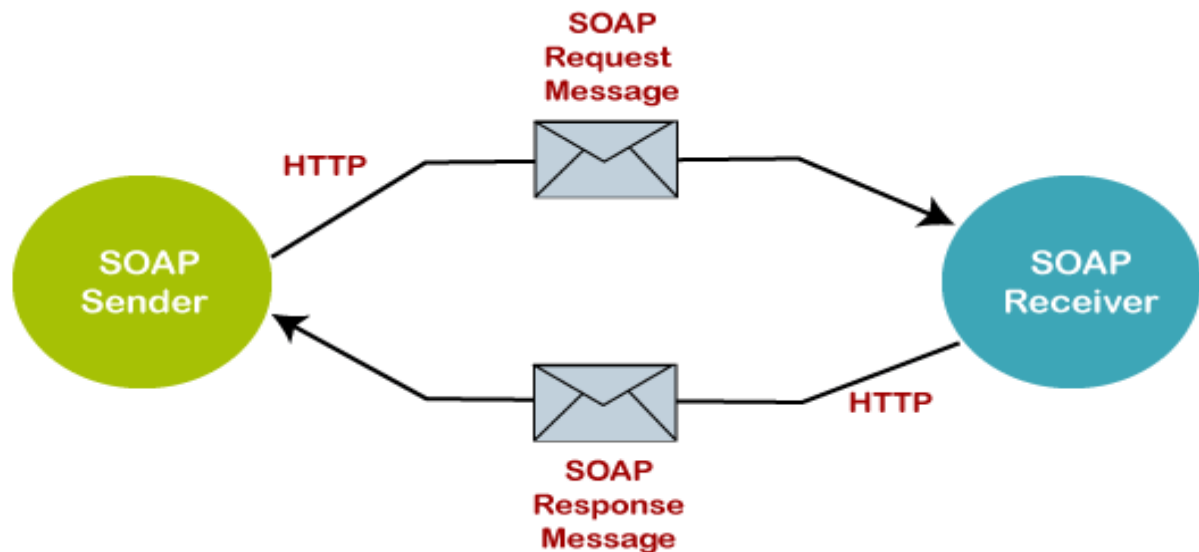
SOAP wordt gewoonlijk gebruikt voor het bouwen van webservices die functionaliteit aan andere toepassingen via het internet beschikbaar stellen. Deze webservices zijn toegankelijk voor client-toepassingen via verschillende protocollen, waaronder HTTP, SMTP en FTP. [1]

Voordelen van SOAP:

- Platformonafhankelijkheid: SOAP is gebaseerd op XML, wat betekent dat het kan worden gebruikt op elk platform en in elke programmeertaal. Dit maakt het gemakkelijk om de API te implementeren en te integreren in verschillende systemen. [7]
- Betrouwbaarheid: SOAP heeft een ingebouwd fout afhandelingssysteem en garandeert dat gegevens consistent en betrouwbaar worden afgeleverd. Het bevat ook een mechanisme voor transacties, wat betekent dat een reeks SOAP-verzoeken als één transactie kan worden behandeld. [1]
- Beveiliging: SOAP ondersteunt verschillende beveiligingsprotocollen zoals SSL en TLS, wat zorgt voor een veilige gegevensoverdracht. Het ondersteunt ook verschillende mechanismen voor authenticatie en autorisatie [1]
- Sterke typen: SOAP gebruikt een sterke typesysteem voor het definiëren van gegevensstructuren en -berichten, wat zorgt voor een hoge mate van interoperabiliteit. [1]
- Beveiliging: SOAP ondersteunt beveiligingsmechanismen zoals authenticatie en autorisatie.

Nadelen van SOAP:

- Complexiteit: de SOAP-standaard is vrij complex en kan moeilijk te begrijpen zijn voor beginners. [7]
- Overhead: SOAP-berichten zijn relatief groot en kunnen veel overhead veroorzaken, wat kan leiden tot trage prestaties. [1]
- Minder efficiënt: vanwege de overhead en het grotere formaat van SOAP-berichten kan het minder efficiënt zijn dan andere webservices protocollen, zoals REST. [1]



Figuur 3: SOAP architectuur [1]

Samengevat is SOAP een berichtenprotocol waarmee toepassingen via het internet met elkaar kunnen communiceren. Het gebruikt een gestandaardiseerde reeks regels voor het coderen van gegevens in een berichtformaat dat kan worden begrepen door elke softwaretoepassing die SOAP ondersteunt. SOAP wordt gewoonlijk gebruikt voor het bouwen van webservices die functionaliteit aan andere toepassingen via het internet beschikbaar stellen.

2.2 REST

REST (Representational State Transfer) is een architectuurstijl die wordt gebruikt om webservices te ontwikkelen. Het is geen protocol zoals SOAP, maar eerder een set van principes voor het ontwerpen van webservices die kunnen worden gebruikt voor het opvragen en behandelen van gegevens.

REST maakt gebruik van HTTP-protocollen zoals GET, POST, PUT en DELETE om gegevens op te vragen en te behandelen. Het is ontworpen om eenvoudig te zijn en gemakkelijk te begrijpen, waardoor het populair is geworden voor het bouwen van webservices.

Een REST API is een verzameling van endpoints die kunnen worden opgeroepen door clients om gegevens op te halen of te behandelen. Deze endpoints worden geïdentificeerd door URI's (Uniform Resource Identifiers), en de gegevens die worden uitgewisseld zijn doorgaans in JSON (JavaScript Object Notation) of XML-formaat. [9]

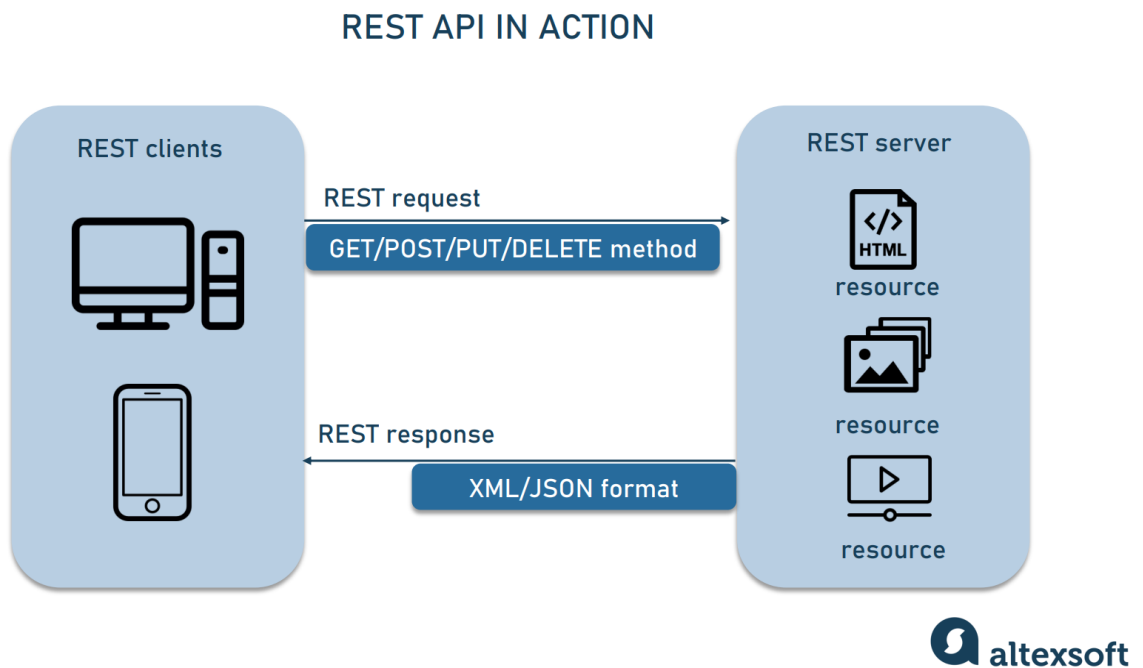
Voordelen van REST API:

- **Schaalbaarheid:** REST is ontworpen om schaalbaar te zijn, waardoor het gemakkelijk kan worden gebruikt voor grotere applicaties met veel gebruikers. [2][3]
- **Efficiëntie:** REST API's hebben over het algemeen minder overhead dan SOAP-gebaseerde API's, wat resulteert in snellere prestaties. REST maakt gebruik van HTTP-protocollen, wat betekent dat het gemakkelijk te implementeren en te gebruiken is. [2][3]
- **Simpliciteit:** REST API's zijn eenvoudig te ontwerpen, te implementeren en te gebruiken.

- Beter cachingbeheer: REST API's ondersteunen caching op clientniveau, wat betekent dat clients gegevens kunnen opslaan en opnieuw kunnen gebruiken zonder ze telkens op te halen van de server. [8]

Nadelen van REST API:

- Gebrek aan standaardisatie: hoewel REST API's eenvoudig te ontwerpen en te implementeren zijn, kan het ontbreken van een standaardisatie leiden tot inconsistenties in de implementatie.
- Beveiligingsproblemen: REST API's vereisen aanvullende beveiligingsmaatregelen om te voorkomen dat ongeautoriseerde gebruikers toegang krijgen tot gevoelige gegevens.
- API-versiebeheer: omdat REST API's geen contractueel formaat hebben, kan het moeilijk zijn om veranderingen in de API te beheren en te documenteren.



Figuur 4: REST architectuur [3]

Samengevat is REST een architectuurstijl die wordt gebruikt om webservices te ontwikkelen, en een REST API is een verzameling van endpoints die kunnen worden opgeroepen door clients om gegevens op te halen of te behandelen. REST API's zijn eenvoudig te ontwerpen, implementeren en gebruiken, maar vereisen aanvullende beveiligingsmaatregelen en het kan lastig zijn om de versies te beheren.

2.3 gRPC

gRPC is een modern, open-source Remote Procedure Call (RPC) framework dat door Google is ontwikkeld. Het biedt een efficiënte en eenvoudige manier om client-toepassingen met server-toepassingen te laten communiceren, ongeacht de programmeertaal en het platform waarop ze

draaien. Met gRPC kunnen ontwikkelaars eenvoudig gedistribueerde systemen bouwen die op grote schaal en op hoge snelheid kunnen werken.[10]

Het gRPC-framework maakt gebruik van de nieuwste protocollen zoals Protocol Buffers (protobufs) en HTTP/2 om efficiënte en betrouwbare communicatie tussen client en server te bieden. Protocol Buffers is een platformonafhankelijk taalafhankelijk binair formaat voor het serialiseren van gestructureerde gegevens. Het biedt een compacte en efficiënte manier om gegevens te coderen, en het maakt de overdracht van grote hoeveelheden gegevens over het netwerk mogelijk. [10]

In gRPC wordt de client-code gegenereerd op basis van een IDL (Interface Definition Language)-bestand dat de gegevensstructuren en de methodehandtekeningen van de server definieert. Dit betekent dat de client en de server gemakkelijk en snel met elkaar kunnen communiceren zonder zich zorgen te hoeven maken over de complexiteit van de communicatie. Dit resulteert in efficiëntere communicatie en snellere prestaties. [2][4]

gRPC ondersteunt verschillende programmeertalen, waaronder Java, Python, C++, Ruby, Go, Node.js, Objective-C, PHP en C#. Het maakt ook gebruik van HTTP/2-protocollen die multiplexing, server push en header-compressie ondersteunen. Dit vermindert de netwerkoverhead en verbetert de prestaties van het systeem. [2][4]

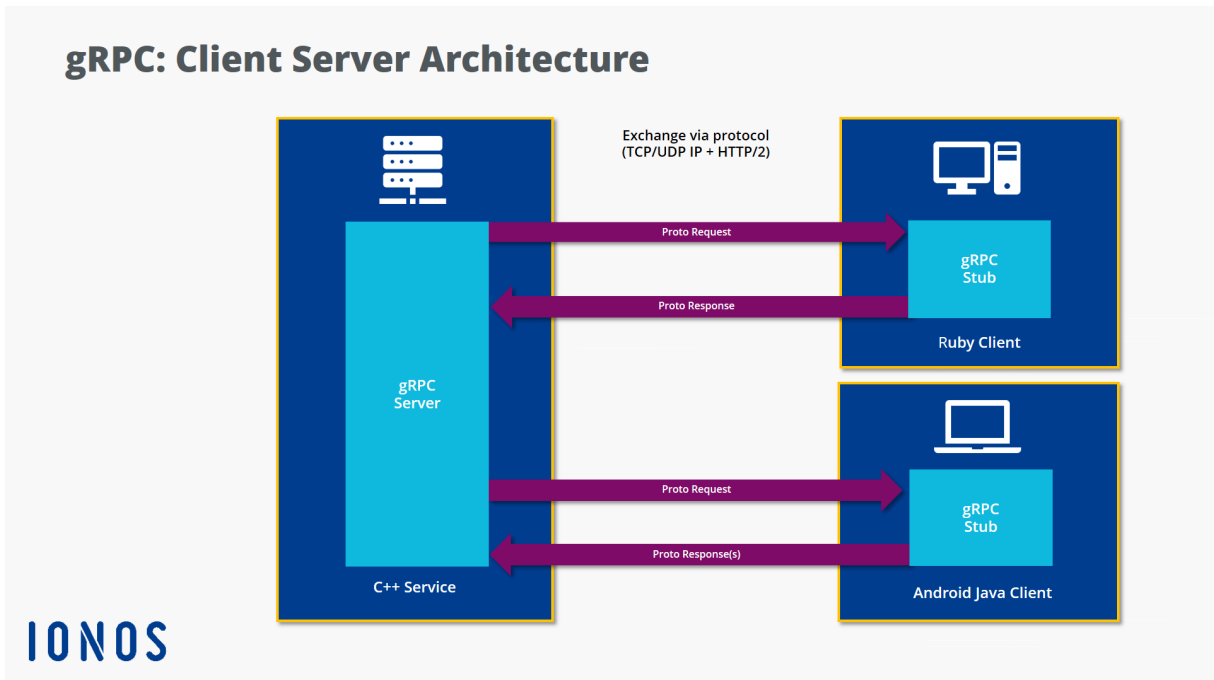
Voordelen van gRPC:

- Snelheid: gRPC gebruikt het Protocol Buffers-serialisatieprotocol en HTTP/2 om snelle en efficiënte communicatie tussen client en server mogelijk te maken. [10]
- Minder overhead: gRPC is licht en heeft minder overhead dan REST, wat betekent dat het minder bandbreedte en CPU-bronnen verbruikt. [2][4]
- Platformonafhankelijkheid: gRPC ondersteunt verschillende programmeertalen en kan op verschillende platformen worden gebruikt. [2][4]
- Automatische generatie van code: gRPC genereert clientcode op basis van een IDL-bestand, waardoor het gemakkelijk te gebruiken en foutloos is. [2][4]
- Ondersteuning voor streaming: gRPC ondersteunt zowel server- als client-streaming, wat betekent dat clients continu gegevens kunnen ontvangen van de server en omgekeerd. [10]

Nadelen van gRPC:

- Leercurve: gRPC heeft een steile leercurve, vooral voor beginners die niet bekend zijn met de bijbehorende technologieën.[10]
- Complexe configuratie: gRPC vereist een complexere configuratie dan sommige andere protocollen, zoals REST. [2][4]
- Minder flexibiliteit: gRPC is minder flexibel dan sommige andere API's, zoals GraphQL, omdat het ontworpen is om de focus te leggen op het beschikbaar stellen van services in plaats van op het opvragen van gegevens. [2][4]

- Beperkte ondersteuning: gRPC heeft nog niet de brede ondersteuning en aanvaarding gekregen die sommige andere API's al hebben, wat kan leiden tot problemen met betrekking tot het vinden van hulpmiddelen en ondersteuning. [2][4]



Figuur 5: GRPC architectuur [4]

Samengevat is gRPC een modern en efficiënt RPC-framework. Het maakt gebruik van het Protocol Buffers-protocol voor het definiëren van de servicecontracten en berichtformaten. Hierdoor is gRPC een ideale keuze voor het bouwen van schaalbare, gedistribueerde systemen in een microservices-architectuur. gRPC biedt meerdere voordelen, zoals hoge prestaties, ondersteuning voor meerdere talen, bi-directionele streaming en automatische generatie van client- en servercode op basis van de servicecontracten. Daarnaast ondersteunt gRPC verschillende authenticatie- en beveiligingsmechanismen en kan het worden geconfigureerd voor gebruik in zowel interne als externe netwerken.

2.4 GRAPHQL

GraphQL is een moderne open-source querytaal en runtime voor API's, ontwikkeld door Facebook. Het biedt een efficiënte en flexibele manier voor client-toepassingen om te communiceren met servertoepassingen, ongeacht de programmeertaal of het platform waarop ze draaien. Met GraphQL kunnen ontwikkelaars gemakkelijk gedistribueerde systemen bouwen die op schaal kunnen werken en zich kunnen aanpassen aan veranderende bedrijfsbehoeften. [5]

Met het GraphQL-framework kunnen clients precies die gegevens specificeren die ze nodig hebben en alleen die gegevens ontvangen, waardoor de noodzaak van meerdere retourritten naar de server wegvalt. Het gebruikt een typesysteem om het schema voor de API te definiëren, wat sterke typering en schemavalidatie mogelijk maakt. Dit maakt het voor ontwikkelaars gemakkelijker om met de API te werken en fouten vroeg in het ontwikkelingsproces op te sporen. [5]

Een van de belangrijkste voordelen van GraphQL is het vermogen om overfetching en underfetching van gegevens, wat kan leiden tot inefficiënte communicatie tussen client en server, te verminderen. Met GraphQL kunnen clients alleen de gegevens opvragen die ze nodig hebben, waardoor minder onnodige gegevens over het netwerk worden gestuurd, wat de prestaties verbetert en de latentie vermindert. [5]

GraphQL ondersteunt ook realtime gegevens en abonnementen, waardoor clients in realtime updates kunnen ontvangen zodra deze beschikbaar zijn. Dit is bijzonder nuttig voor toepassingen die realtime updates vereisen, zoals chat-toepassingen of beursapps. [5]

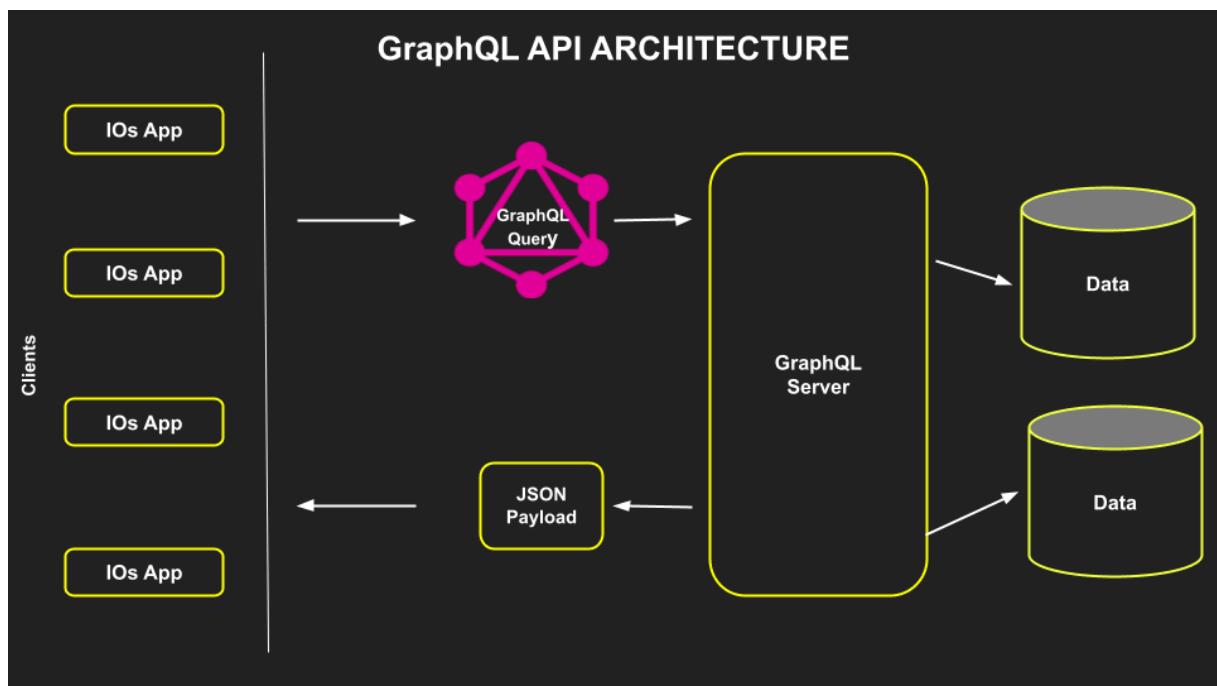
Een ander voordeel van GraphQL is dat het met meerdere programmeertalen en platformen kan werken. Het kan worden gebruikt met verschillende programmeertalen, waaronder JavaScript, Java, Python, Ruby en meer. Dit maakt het gemakkelijker voor ontwikkelaars om te werken met de technologieën waar ze zich het beste bij voelen. [2][5]

Voordelen van GraphQL:

- **Efficiëntie:** GraphQL vermindert overfetching en underfetching van gegevens, wat leidt tot efficiëntere communicatie tussen client en server. [2][5]
- **Flexibiliteit:** GraphQL biedt een flexibele manier voor klanten om de gegevens op te vragen die ze nodig hebben, zodat ze zich kunnen aanpassen aan veranderende bedrijfsvereisten. [2][5]
- **Sterke typering:** GraphQL gebruikt een typesysteem om het schema voor de API te definiëren, waardoor sterke typering en schemavalidatie mogelijk zijn. [2]
- **Realtime gegevens en abonnementen:** GraphQL ondersteunt realtime gegevens en abonnementen, waardoor klanten in realtime updates kunnen ontvangen. [2][5]

Nadelen van GraphQL:

- Leercurve: GraphQL heeft een steile leercurve, vooral voor beginners die niet bekend zijn met de bijbehorende technologieën. [2]
- Caching: GraphQL biedt geen ingebouwde caching, wat tot prestatieproblemen kan leiden als er niet goed mee wordt omgegaan. [2]
- Complexiteit: GraphQL kan complexer zijn om op te zetten en te configureren dan sommige andere API's, zoals REST. [2][5]



Figuur 6: GraphQL architectuur [5]

Samengevat is GraphQL een moderne en flexibele querytaal en runtime voor API's die een efficiëntere en effectievere manier biedt voor clienttoepassingen om te communiceren met servertoepassingen. Het vermindert de hoeveelheid onnodige gegevens die over het netwerk worden verzonden, ondersteunt realtime gegevens en abonnementen, en kan werken met meerdere programmeertalen en platformen. Hoewel het een steile leercurve heeft en het opzetten en configureren ingewikkelder kan zijn dan sommige andere API's, maken de voordelen het een uitstekende keuze voor het bouwen van schaalbare, gedistribueerde systemen in een microservices-architectuur.

2.5 Overzicht

Tabel 2: Voor- en nadelen van API architecturen

API-protocol	Voordelen	Nadelen
SOAP	<ul style="list-style-type: none">• Uitgebreide standaard• Betrouwbaar• Platformonafhankelijk• Veiligheid	<ul style="list-style-type: none">• Grootte van de payload• Minder flexibel• Minder leesbaar
REST	<ul style="list-style-type: none">• Eenvoudig te begrijpen en te implementeren• Hoge prestaties• Brede ondersteuning• Caching mogelijk	<ul style="list-style-type: none">• Minder gestandaardiseerd• Beveiligingsproblemen• Geen standaard voor foutafhandeling
GraphQL	<ul style="list-style-type: none">• Efficiëntere gegevensoverdracht• Betere documentatie• Flexibiliteit• Abonneren	<ul style="list-style-type: none">• Steile leercurve• Minder caching• Meer complexiteit
gRPC	<ul style="list-style-type: none">• Weinig overhead• Platformonafhankelijk• Snelheid (HTTP/2)• Ondersteunt streaming	<ul style="list-style-type: none">• Vereist protocol buffers als gegevensformaat• Minder geschikt voor publieke API's• Beperkte ondersteuning voor programmeertalen• Steile leercurve

2.6 Conclusie

Voor Kreonta Edge zijn er verschillende API-protocollen waaruit gekozen kan worden, elk met zijn eigen voor- en nadelen. Bij het kiezen van een API-protocol is het belangrijk om rekening te houden met de specifieke behoeften van de applicatie.

REST en gRPC zouden bijvoorbeeld goede opties kunnen zijn voor applicaties met hoge prestatie-eisen. REST heeft een brede acceptatie en ondersteuning in de industrie, is eenvoudig te implementeren en gemakkelijk te begrijpen. REST is vooral geschikt voor situaties waarin er veel leesacties zijn en er relatief weinig schrijfacties zijn. gRPC daarentegen is geschikt voor situaties waarin er veel schrijfacties zijn, zoals in realtime data streaming. gRPC is sneller en efficiënter dan REST en kan gegevens in binair formaat overdragen, waardoor het minder overhead heeft dan REST.

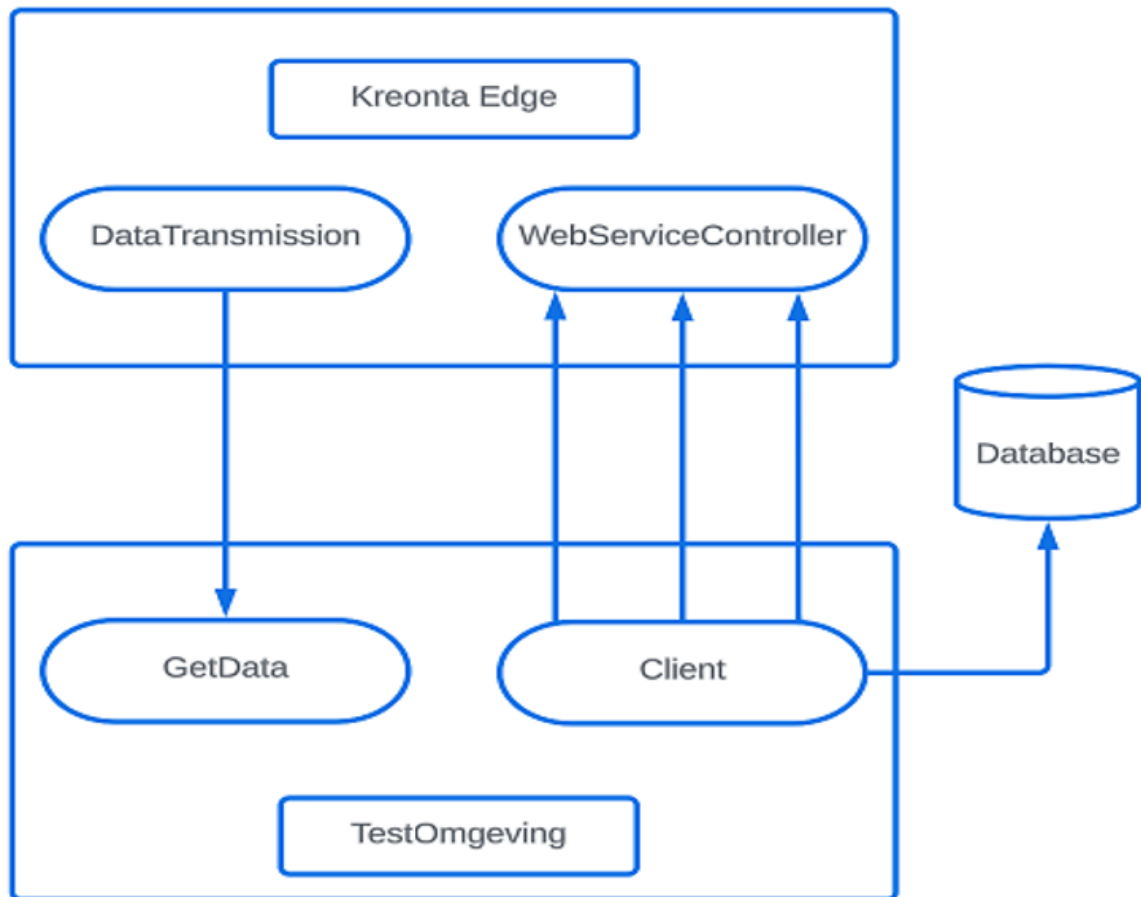
GraphQL zou een goede optie kunnen zijn voor efficiëntere gegevensoverdracht en flexibiliteit. In tegenstelling tot REST, waarbij de client altijd de volledige dataset krijgt, stelt GraphQL de client in staat om specifiekere verzoeken te doen naar het gewenste data en alleen de benodigde data terug te krijgen. Dit kan resulteren in snellere en efficiëntere gegevensoverdracht.

gRPC zou ook geschikt kunnen zijn voor InstantTransmission van de OPC UA data vanwege de hoge prestaties en de mogelijkheid om in realtime te streamen.

SOAP zou een goede keuze kunnen zijn voor applicaties die hoge betrouwbaarheid en veiligheid vereisen, met name wanneer de data van de productielijnen gevoelig zijn. SOAP heeft de mogelijkheid om complexe transacties uit te voeren en heeft ingebouwde beveiligingstechnologieën die de communicatie tussen client en server veilig houden.

3 Testomgeving

Als testomgeving wordt er een C# client gemaakt die de data van de server ontvangt en de controller van de server aanspreekt. De resultaten van elke API call worden dan opgeslagen in een databank die lokaal draait.



Figuur 7 Schema Testomgeving

De testomgeving bestaat uit twee projecten:

- GetDataAPI: Deze API is verantwoordelijk voor het ontvangen van de WebServiceCalls van Kreonta Edge.
- Client: Dit project is verantwoordelijk voor het aanspreken van endpoints van WebServiceController in de Kreonta Edge.

In Kreonta Edge worden er twee klassen aangepast:

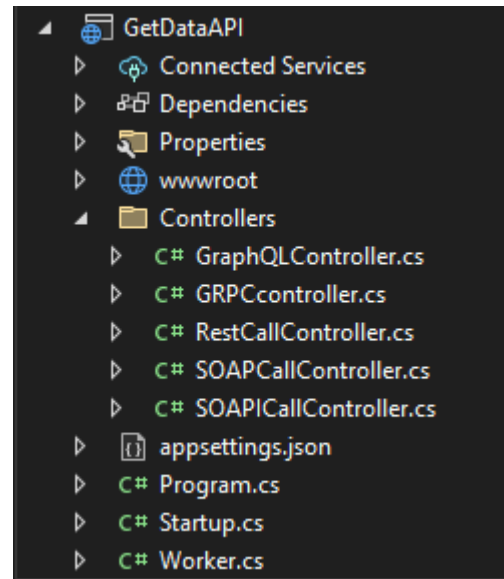
- DataTransmission: Deze klasse verstuurt WebServiceCalls naar geabonneerde clients.
- WebServiceController: Deze controller bevat alle endpoints die de testomgeving aanspreekt.

3.1 GetDataAPI

Deze API is niet traditioneel en bestaat uit meerdere API structuren. Dit maakt het gemakkelijk om DataTransmission te testen. In dit project zijn twee belangrijke onderdelen: Controllers en Startup.cs

In de folder Controllers word er voor elke API structuur een controller gemaakt met maar één endpoint, deze endpoint ontvangt de data die wordt verstuurd vanuit Kreonta Edge.

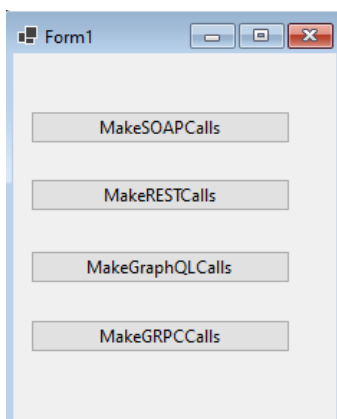
De Startup klasse is verantwoordelijk voor de configuratie van de API. Standaard is deze geconfigureerd voor een REST API, maar met een paar aanpassingen is het mogelijk om andere soorten API te gebruiken.



Figuur 8 Solution Explorer GetDataAPI

3.2 Client

Voor de client is er een windows form waarbij er voor elke structuur er een knop is die de WebService endpoints gaat aanroepen in Kreonta Edge.



Figuur 9 Client Form

```
public async void button14_Click(object sender, EventArgs e)
{
    Guid webServiceId = new("8f8c6e28-eb56-4b4d-b993-157ac9545535");
    WebServiceDTO webService = new();
    for (int i = 0; i < 300; i++)
    {
        List<WebServiceDTO>? response2 = await GetAllWebServicesGraph();
        WebServiceDTO? response1 = await UpdateWebServiceGraph3(webService);
        WebServiceDTO? response3 = await FindWebServiceByIdGraph(webServiceId);
        string? response4 = await RemoveWebServiceGraph(webServiceId);
    }
}

1 reference
public async Task<List<WebServiceDTO>> GetAllWebServicesGraph()
{
    string query = "...";
    HttpRequestMessage? request = new(HttpMethod.Post, graphqlEndpoint)...;
    Stopwatch? stopwatch = new();
    stopwatch.Start();
    try {
        HttpResponseMessage? response = await httpClient.SendAsync(request);
        stopwatch.Stop();
        decimal executionTime = stopwatch.ElapsedMilliseconds;
        long memory = CalculateSize(request) + CalculateSize(response);
        AddEndpointCall("GetAllWebServicesGraph", true, executionTime, memory);
        string? responseContent = await response.Content.ReadAsStringAsync();
        QueryResponse? queryResponse = JsonConvert.DeserializeObject<QueryResponse>(responseContent);
        return queryResponse?.Data?.GetAllWebServices;
    }
    catch (Exception e)
    {
        AddEndpointCall("GetAllWebServicesGraph", false, 0, 0);
        return new List<WebServiceDTO>();
    }
}
```

Codefragment 4: GetAllWebServicesGraph

In het bovenstaande codefragment 4 staat beschreven wat er gebeurt wanneer er op de knop “MakeGraphQLCalls” geklikt wordt.

Wanneer er op de knop geklikt wordt, wordt elke GraphQL endpoint in Kreonta Edge 300 keer aangesproken. In de praktijk zal een endpoint nooit 300 keer snel na elkaar aangesproken worden, maar dit zorgt voor een stresstest. Hetzelfde wordt gedaan voor de andere protocollen.

3.2.1 ExecutionTime

ExecutionTime is de tijd in milliseconden die de client nodig heeft om zijn verzoek te verzenden naar Kreonta Edge en een antwoord terug te krijgen. Om correcte metingen te krijgen is het belangrijk om de meest accurate toepassing te gebruiken. Hiervoor werden er twee mogelijkheden vergeleken: stopwatch en datetime.

De stopwatch klasse telt de timer ticks in het onderliggend timer systeem. Indien de hard- en software het toelaat, maakt de stopwatch gebruik van de high-resolution performance counter. De high-resolution performance counter is een timer in de CPU met precisie tot op de nanoseconde. De stopwatch stuurt dan queries naar het OS om deze op te vragen. Intens CPU gebruik kan een lichte impact hebben op de precisie, maar de precisie zal nog steeds tot op minder dan een nanoseconde correct zijn.[6]

Een alternatieve manier om de tijd tussen het begin en het einde van de omzetting te meten, is door de DateTime functie te gebruiken. Het principe is om de tijd bij het begin en op het einde van de omzetting op te vragen. Dit zou als voordeel kunnen hebben dat het een minimale impact heeft op de CPU tijdens de meting zelf. Maar dit heeft als nadeel dat dit niet zo nauwkeurig kan gebeuren als de stopwatch.

Vermits nauwkeurigheid voor dit project het belangrijkste aspect is, werd er gekozen voor de stopwatch.

3.2.2 Memory

Memory is het totale geheugen van het verzoek en het antwoord dat wordt verstuurd in bytes. Om dit te berekenen, werd een functie gemaakt die elk object serialiseert naar een string. Daarna wordt de string geconverteerd met behulp van UTF8-codering naar een array van bytes. De lengte van deze array is de uiteindelijke grootte van het object in bytes.

```

public static long CalculateSize<T>(T obj)
{
    JsonSerializerOptions? options = new()
    {
        DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull
    };
    string json = System.Text.Json.JsonSerializer.Serialize(obj, options);
    byte[] bytes = Encoding.UTF8.GetBytes(json);
    var totalSize = bytes.Length;
    return totalSize;
}

```

Codefragment 5: CalculateSize

3.2.3 AddEndpointCall

Deze functie krijgt de nodige argumenten mee na er een endpoint is aangesproken, daarna wordt het object EndpointCall ingevuld met de waarden van deze argumenten. Als het object gemaakt is slaat hij deze op in de lokale MySQL databank.

```

public void AddEndpointCall(string functionName, bool success, decimal executionTime, long memory)
{
    EndpointCall? call = new()
    {
        CallTime = DateTime.Now,
        FunctionName = functionName,
        Success = success,
        ExecutionTime = executionTime,
        Memory = memory
    };
    _context.EndpointCalls.Add(call);
    _context.SaveChanges();
}

```

Codefragment 6: AddEndpointCall

4 Proof of concept verschillende API's

In dit hoofdstuk wordt er per alternatieve API beschreven wat er nodig is om de REST API te refactoren evenals de voor- en nadelen van de alternatieven.

Het eerste deel van dit hoofdstuk beschrijft de vertreksituatie met de REST API. De volgende delen bespreken telkens een alternatieve API.

4.1 REST Situatie As Is

4.1.1 Server

Bij het aanmaken van een API in Visual Studio wordt automatisch zo goed als alles in orde gebracht voor een REST API.

```
[Route("api/[controller]")]
[Authorize]
[ApiController]
1 reference
public class WebServiceController : ControllerBase
{
    [HttpGet("{id}")]
    1 reference
    public ActionResult<WebServiceDTO> FindWebserviceById(Guid id)
    {
        try
        {
            using (MySQLConnection mConnection = new MySQLConnection(conn)) ...
        }
        catch (Exception e)
        {
            return BadRequest(e.Message);
        }
    }
}
```

Codefragment 7 REST WebServiceController

Bovenstaande codefragment presenteert de WebServiceController en daarin de endpoint FindWebserviceById.

Om een controller te configureren moet er een route gedefinieerd worden waarop de API bereikbaar is, in dit geval kan de controller bereikt worden op: <https://localhost:433/api/webservice>.

Een endpoint wordt toegevoegd door boven een functie het gepaste HTTP-attribuut (HttpGet, HttpPost,...) toe te voegen. De endpoint in het voorbeeld kan aangesproken worden door het verzoek: <https://localhost:433/api/webservice/id>. Hier wordt het argument "id" meegegeven zodat de server kan zoeken naar de gepaste WebService.

4.1.2 Client

REST is een zeer solide systeem en wordt beschouwd als de de facto standaard voor een API. REST biedt ontwikkelaars een eenvoudige manier om HTTP-requests te maken naar API-endpoints en maakt daarbij gebruik van de bestaande HTTP-methoden zoals GET, POST, PUT en DELETE.

Een groot voordeel van REST is dat er veel goede documentatie beschikbaar is. Ontwikkelaars kunnen snel aan de slag met het maken van API's door middel van talloze tutorials, codevoorbeelden en andere documentatie die beschikbaar is op het internet. Bovendien zijn er veel handige libraries beschikbaar die het ontwikkelen van REST API's nog eenvoudiger maken.

Een ander groot voordeel van REST is dat het een lichte architectuur heeft. REST API's vereisen geen extra lagen van middleware of andere complexe architectuur om te werken. Hierdoor kan er met weinig code veel gedaan gekregen worden.

```
HttpClient _client = new HttpClient();  
Cursor.Current = Cursors.WaitCursor;  
HttpResponseMessage result = _client.GetAsync("https://localhost:443/api/webservice").Result;  
response = result.Content.ReadAsStringAsync().Result;  
if (result.IsSuccessStatusCode)  
{  
    if (string.IsNullOrEmpty(response))  
    {  
        return default;  
    }  
    return JsonSerializer.Deserialize<T>(response);  
}
```

Codefragment 8: REST endpoint aanspreken

Deze code is een voorbeeld van een REST-request in C# om een API-endpoint aan te roepen. REST is een architectuurstijl om gegevens uit te wisselen tussen systemen en maakt hiervoor gebruik van HTTP-requests en -responses. In dit voorbeeld wordt een GET-request gestuurd naar een API-endpoint met behulp van de HttpClient-class.

De functie begint met het initialiseren van een HttpClient-object, waarmee de request zal worden verstuurd. Vervolgens wordt de cursor veranderd naar de "wachten"-cursor om aan te geven dat er een actie wordt uitgevoerd.

Met behulp van de GetAsync-methode wordt een GET-request verstuurd naar het opgegeven URL "https://localhost:443/api/webservice". De ontvangen response wordt opgeslagen in de HttpResponseMessage-variabele "result".

Om de inhoud van de response te lezen, wordt de ReadAsStringAsync-methode van de response.Content-property aangeroepen. De inhoud wordt opgeslagen in de "response"-variabele.

Als de HTTP-statuscode van de response aangeeft dat de request succesvol was (bijv. 200 OK), dan wordt de JSON-inhoud van de response gedeserialiseerd naar het generieke type T met behulp van de JsonSerializer.Deserialize-methode. Als de response geen inhoud bevat, wordt er standaard een null-waarde geretourneerd.

Deze code maakt gebruik van de handige HttpClient-class in C#, waardoor het versturen van een HTTP-request en het lezen van de response-inhoud zeer eenvoudig wordt. Dit zorgt voor een efficiënte en gebruiksvriendelijke implementatie van RESTful API-calls in C#.

4.1.3 Conclusie

Een API opzetten in REST is zeer gemakkelijk. Met weinig aanpassingen kan er al een goede basis opgezet worden. Door de grote gemeenschap is er ook veel informatie te vinden op het internet. Duidelijke foutmeldingen zorgen ook voor het efficiënt debuggen van foutieve code.

4.2 POC SOAP

4.2.1 Server

In tegenstelling tot bij REST moet er extra code toegevoegd worden, zodat de API bereikbaar is.

```
app.UseSoapEndpoint<IServerController>("/WebService.asmx", new SoapEncoderOptions(), SoapSerializer.XmlSerializer);
```

Codefragment 9: SOAP endpoint mapping

Per controller zou dus de bovenstaande lijn code moeten toegevoegd moeten worden, zodat de endpoints van de controller bereikbaar zouden zijn.

```
[ServiceContract(Namespace = "https://localhost:443/WebService")]
[XmlSerializerFormat]
3 references
public interface IWebServiceController
{
    [OperationContract]
    1 reference
    Task<List<WebServiceDTO>> GetAllWebServices();
}
```

Codefragment 10 SOAP IWebServiceController

Per controller wordt er ook een interface aangemaakt waarbij de controller via een servicecontract gelinkt wordt aan een url die clients kunnen gebruiken. Daarna wordt de effectieve controller aangemaakt waar functies van de interface geïmplementeerd worden.

4.2.2 Client

Om de huidige REST API te herschrijven naar een SOAP API was er aan de server-kant weinig verandering nodig aan de code en dit lukte vrij vlot. Er zit echter een groot verschil aan de client kant. Een simpele Get request wordt al snel complex en omslachtig.

```
private WebServiceDTO FindWebserviceById(Guid id)
{
    string _action = "https://localhost:443/WebService/FindWebserviceById";
    string soapRequest = $@"<soapenv:Envelope xmlns:soapenv=""http://schemas.xmlsoap.org/soap/envelope/"><soapenv:Header/>
    <soapenv:Body>
        <ser:FindWebserviceById>
            <ser:id>{id}</ser:id>
        </ser:FindWebserviceById>
    </soapenv:Body>
    </soapenv:Envelope>";
    string? soapResult = MakeAPICall(_action, soapRequest, "FindWebserviceByIdSOAP");
    return ConvertXmlToWebServiceDTO(soapResult, "FindWebserviceByIdResult").First();
}
```

Codefragment 11: SOAP Query

Bovenstaande code is een voorbeeld van een SOAP-request in C# om een API-endpoint aan te roepen. SOAP is een protocol om gegevens uit te wisselen tussen systemen en maakt hiervoor gebruik van XML-berichten. In dit voorbeeld wordt een SOAP-request gestuurd naar een API-endpoint met behulp van een POST-request.

In de functie FindWebserviceById wordt een SOAP-request aangemaakt met een URL van het API-endpoint en een XML-requestbody. De functie MakeAPICall wordt aangeroepen om deze SOAP-request te versturen. Na ontvangst van het antwoord wordt het XML-bericht omgezet naar een WebServiceElementDTO-object en getourneerd.

```

public string MakeAPICall(string action, string soapRequest, string functie)
{
    string? _url = "https://localhost:443/Service2.asmx";

    var stopwatch = new Stopwatch();
    stopwatch.Start();
    XmlDocument? soapEnvelopeXml = CreateSoapEnvelope(soapRequest);
    HttpWebRequest? webRequest = CreateWebRequest(_url, action);
    int memoryUsage = soapRequest.Length * 2;
    InsertSoapEnvelopeIntoWebRequest(soapEnvelopeXml, webRequest);
    string soapResult;
    using (WebResponse? asyncResponse = webRequest.GetResponse())
    using (StreamReader? streamReader = new(asyncResponse.GetResponseStream()))
    {
        soapResult = streamReader.ReadToEnd();
    }

    stopwatch.Stop();
    int executionTime = (int)stopwatch.ElapsedMilliseconds;
    AddEndpointCall(functie, true, executionTime, memoryUsage);
    return soapResult;
}

1 reference
private static HttpWebRequest CreateWebRequest(string url, string action)
{
    HttpWebRequest? webRequest = (HttpWebRequest)WebRequest.Create(url);
    webRequest.Headers.Add("SOAPAction", action);
    webRequest.ContentType = "text/xml; charset=utf-8";
    webRequest.Accept = "text/xml";
    webRequest.Method = "POST";
    return webRequest;
}

```

Codefragment 12: SOAP endpoint aanspreken

De functie MakeAPICall stuurt de SOAP-request naar het API-endpoint door een HTTP POST-request te maken. Hierbij wordt de CreateWebRequest-functie gebruikt om het HTTP-verzoek op te stellen en wordt de CreateSoapEnvelope-functie gebruikt om de SOAP-request om te zetten naar een XmlDocument-object. Vervolgens wordt de requestbody ingevoegd in het HTTP-verzoek en wordt het verzoek verzonden. Wanneer de API een antwoord terugstuurt, gebeurt dit in de vorm van een XML-bestand. Het deserialiseren van dit bestand is niet onmiddellijk mogelijk en dus wordt er momenteel voor elk object een nieuwe converter aangemaakt. Dit kan leiden tot vertragingen en inefficiëntie in het verwerkingsproces.

```

public static List<WebServiceDTO> ConvertXmlToWebServiceDTO(string xml, string _namespace)
{
    XNamespace ns = "https://localhost:443/WebService";
    XElement root = XElement.Parse(xml);
    List<WebServiceDTO> webServiceDTOs = root.Descendants(ns + _namespace).Select(e => new WebServiceDTO
    {
        WebServiceId = (string)e.Element(ns + "WebServiceId"),
        Name = (string)e.Element(ns + "Name"),
        Delay = (uint)e.Element(ns + "Delay"),
        Bundle = (bool)e.Element(ns + "Bundle"),
        KeepSequence = (bool)e.Element(ns + "KeepSequence"),
        RetryCount = (uint)e.Element(ns + "RetryCount"),
        RetryDelay = (uint)e.Element(ns + "RetryDelay"),
        HistoryHours = (uint)e.Element(ns + "HistoryHours"),
        Enabled = (bool)e.Element(ns + "Enabled"),
        Instant = (bool)e.Element(ns + "Instant"),
        DestinationId = (string)e.Element(ns + "DestinationId"),
        GroupId = (string)e.Element(ns + "WebServiceGroupId"),
        ElementIds = e.Element(ns + "ElementIds").Descendants(ns + "string").Select(x => (string)x).ToList(),
        CallIds = e.Element(ns + "CallIds").Descendants(ns + "string").Select(x => (string)x).ToList()
    }).ToList();
    return webServiceDTOs;
}

```

Codefragment 13: ConvertXmlToWebServiceDTO

Het opzetten van een correcte request is soms wat uitdagend en er kunnen gemakkelijk fouten worden gemaakt.

Een van de mogelijke fouten is bijvoorbeeld het verkeerd formatteren van de XML-requestbody, waarbij de elementen verkeerd zijn genest, de namen of attributen van de elementen fout zijn gespeld of er ongeldige tekens worden gebruikt. Dit kan leiden tot syntaxfouten die de server niet kan begrijpen waardoor hij als resultaat een foutmelding zal terugsturen.

Andere veelvoorkomende fouten kunnen optreden wanneer de URL van het API-endpoint onjuist is gespecificeerd of wanneer de server niet bereikbaar is. Dit kan resulteren in een time-outfout of een foutmelding die aangeeft dat de server niet kan worden gevonden.

Foutmeldingen die kunnen optreden bij het versturen van een SOAP-request, worden in de vorm van XML-foutcodes geretourneerd. Dit maakt het niet gemakkelijk om de gemaakte fouten op te sporen en op te lossen.

4.2.3 Conclusie

Door het gebruik van XML-requestbodies kunnen snel fouten gemaakt worden en dit gecombineerd met onduidelijke foutmeldingen kan zorgen voor veel frustraties. Aangezien SOAP niet echt veel gebruikt wordt en ook een iets oudere structuur is, is het ook niet altijd even gemakkelijk om hulpvolle informatie te vinden.

4.3 POC gRPC

4.3.1 Server

In eerste instantie moeten de gRPC-service en gRPC-endpoint mapping toegevoegd worden. Daarnaast moet er nog redelijk veel aangepast worden om de huidige API in gRPC te kunnen laten functioneren.

gRPC is niet zoals REST of SOAP geschreven in C# code. Bij gRPC wordt de communicatie tussen client en server gedefinieerd in een .proto-bestand met behulp van de Protobuf-taal. Dit .proto-bestand

fungeert als een contract tussen de client en server, waarin de berichtstructuren en service-definities worden gespecificeerd. Het beschrijft welke methoden beschikbaar zijn, de parameters en de geretourneerde berichten. Het .proto-bestand fungeert als de gemeenschappelijke taal tussen verschillende programmeertalen en stelt gRPC in staat om codegeneratie toe te passen om stubs en clients in verschillende talen te genereren.

Dit betekent ook wel dat wanneer de server het .proto bestand wil aanpassen, deze wijziging ook moet doorgevoerd worden naar de client. Anders worden de wijzigingen niet ondersteund. Dit kan zorgen voor ingewikkelde situaties.

```
syntax = "proto3";

option csharp_namespace = "KreontaEdgeService.GRPC";

service WebServiceService {
  rpc GetAllWebServices (EmptyGetAllWebServiceRequest) returns (WebServiceList);
  rpc FindWebServiceById (FindWebServiceByIdRequest) returns (WebServiceData);
  rpc UpdateWebService (UpdateWebServiceRequest) returns (UpdateWebServiceResponse);
}

message WebServiceData {
  string id = 1;
  string name = 2;
  int32 delay = 3;
  bool bundle = 4;
  bool keep_sequence = 5;
  int32 retry_count = 6;
  int32 retry_delay = 7;
  int32 history_hours = 8;
  bool enabled = 9;
  bool instant = 10;
  string destination_id = 11;
  string web_service_group_id = 12;
  repeated string element_id = 13;
  repeated string call_id = 14;
}

message UpdateWebServiceRequest {
  WebServiceData web_service = 1;
}
```

Codefragment 14: gRPC WebService proto-bestand

De bovenstaande code is een definitie van een gRPC-service en de bijbehorende berichttypes in het Protocol Buffers-formaat (proto3).

In de code wordt als eerste de syntaxversie gespecificeerd met `syntax = "proto3"`, wat aangeeft dat de code is geschreven in het proto3-syntax-formaat.

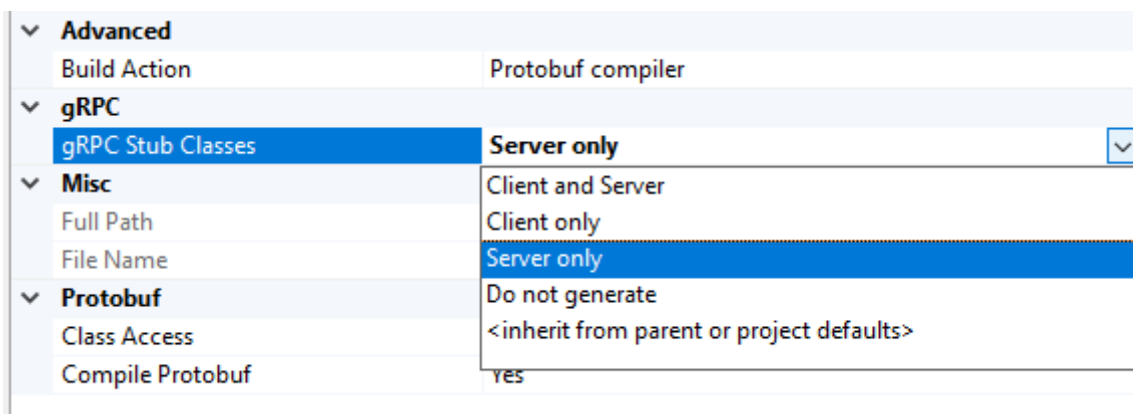
Daarna wordt met `option csharp_namespace = "KreontaEdgeService.GRPC"` aangegeven dat de gegenereerde C#-code voor de gRPC-service en berichttypes zal worden geplaatst in de C#-namespace "KreontaEdgeService.GRPC".

De `WebServiceService` is de naam van de gRPC-service die wordt gedefinieerd. Deze service bevat drie endpoints: `GetAllWebServices`, `FindWebServiceById`, en `UpdateWebService`. Elke methode heeft een request-parameter en een response-type.

Het berichttype WebServiceData wordt gedefinieerd met verschillende velden zoals "id", "name", "delay", enz. Dit berichttype wordt gebruikt om gegevens van en naar de gRPC-service te verzenden.

Het berichttype UpdateWebServiceRequest heeft één veld genaamd "web_service" van het type WebServiceData. Dit berichttype wordt gebruikt als de request-parameter voor de UpdateWebService endpoint.

Het gebruik van protobuf-bestanden in combinatie met gRPC vereist de juiste NuGet-packages om de nodige functionaliteit te bieden. Deze packages omvatten de gRPC- en protobuf-bibliotheken, die belangrijke hulpmiddelen en resources bevatten om protobuf-bestanden te verwerken.



Figuur 10 Build opties GRPC

Een cruciale component van deze packages is de toevoeging van extra Build Actions aan protobuf-bestanden. Door het uitvoeren van de protobuf compiler tijdens het buildproces wordt het protobuf-bestand gecompileerd en geanalyseerd. Deze gegenereerde code biedt essentiële functies, zoals het opzetten van de communicatiekanalen tussen clients en servers, het omzetten van berichten in een efficiënt overdraagbaar formaat en het afhandelen van de juiste deserialisatie aan de ontvangende kant.

4.3.2 Client

Door de automatisch gegenereerde code bij het bouwen van de proto file zijn alle naamgeving en structuren er al. Dit maakt dat het programmeren van de effectieve controller efficiënt gaat.

```
var webServiceId = "a133cf83-e7a7-40c5-adf2-e2fb8237a0b5";
stopwatch.Restart();
var channel = GrpcChannel.ForAddress("https://localhost:443");
var client = new WebServiceService.WebServiceServiceClient(channel);
var response = client.FindWebServiceById(new FindWebServiceByIdRequest { Id = webServiceId });
memory = response.CalculateSize();
```

Codefragment 15: GRPC endpoint aanspreken

In de bovenstaande code wordt een gRPC-client geïmplementeerd om een specifieke webservice op te zoeken en de grootte van het bijbehorende antwoord te berekenen.

Daarna wordt een gRPC-kanaal geïntanceerd met `GrpcChannel.ForAddress` ("https://localhost:443"). Dit kanaal specificeert het adres waar de gRPC-service wordt gehost, in dit geval "https://localhost:443". Hiermee wordt een verbinding tot stand gebracht met de gRPC-service.

Met behulp van het gRPC-kanaal wordt een client-object van het type `WebServiceService.WebServiceServiceClient` gemaakt. Deze client wordt gebruikt om gRPC-aanroepen naar de service uit te voeren.

Vervolgens wordt de `FindWebServiceById` endpoint aangeroepen op de client, waarbij een nieuwe `FindWebServiceByIdRequest` wordt meegegeven met de `webServiceId` als parameter. Dit verzoek wordt naar de gRPC-service gestuurd om de webservice met de opgegeven `webServiceId` op te zoeken.

Het antwoord van de gRPC-service, dat een `WebServiceData`-object bevat met informatie over de gevonden webservice, wordt toegewezen aan de variabele `responseld`.

4.3.3 Streaming

Streaming is een krachtige functionaliteit die gRPC biedt, waarmee efficiënte en realtime communicatie mogelijk is tussen client en server. Met streaming kunnen clients en servers gegevens in de vorm van streams naar elkaar verzenden, in plaats van één enkel bericht per call.

Om streaming-functionaliteit toe te voegen aan een gRPC-service, kunnen de proto-bestanden worden aangepast om streaming endpoints te definiëren. Dit omvat het specificeren van de juiste streaming-modus (server streaming of bidirectionele streaming) voor de gewenste methoden in de service-definitie. De gegenereerde code zal deze streaming-functionaliteit ondersteunen en kan dan geïmplementeerd worden in de controller.

4.3.4 Conclusie

Het opzetten van een API met gRPC kan in het begin enige moeilijkheden opleveren, omdat het concept aanzienlijk verschilt van REST. Eenmaal de basisinfrastructuur opgezet is, is het echter relatief eenvoudig om de functionaliteit uit te breiden. De officiële documentatie van gRPC biedt nuttige en begrijpelijke informatie. De foutmeldingen die worden gegenereerd, zijn over het algemeen duidelijk en vergemakkelijken het debuggen van de applicatie. Voor een recente technologie is de community redelijk groot. Hierdoor is heel wat waardevolle informatie te vinden op het internet, waardoor problemen snel opgelost geraken.

4.4 POC GraphQL

4.4.1 Server

Net zoals de andere protocollen moeten er in de startup klasse een paar configuraties gemaakt worden zodat de API-endpoints bereikbaar zijn voor clients.

```

services.AddGraphQLServer().AddQueryType<QueryType>().AddMutationType<MutationType>();
}

0 references
public void Configure(IApplicationBuilder app, IWebHostEnvironment env, EdgeDataContext context, LoggingService logger)
{
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGraphQL("/graphql");
        endpoints.MapControllers();
    });
}

```

Codefragment 16: GRAPHQL endpoint mapping

Deze code configureert en voegt GraphQL toe als een service. Daarna moet er een GraphQL-schema gemaakt worden waar de root query en mutation-type gedefinieerd worden. Het schema is gemaakt met behulp van “Hot Chocolate”, dit is een populaire bibliotheek die op een eenvoudige manier GraphQL-functionaliteiten toevoegt.

```

public class QueryType : ObjectType<Query>
{
    0 references
    protected override void Configure(IObjectTypeDescriptor<Query> descriptor)
    {
        descriptor.Field(q => q.GetAllWebServices())
            .Type<NonNullType<ListType<NonNullType<WebServiceType>>>>().Name("GetAllWebServices");

        descriptor.Field(q => q.FindWebServiceById(default)).Type<WebServiceType>()
            .Argument("id", arg => arg.Type<NonNullType<IdType>>()).Name("FindWebServiceById");
    }
}

1 reference
public class MutationType : ObjectType<Mutation>
{
    0 references
    protected override void Configure(IObjectTypeDescriptor<Mutation> descriptor)
    {
        descriptor.Field(m => m.UpdateWebService(default)).Type<WebServiceType>()
            .Argument("req", arg => arg.Type<WebServiceDTOInput>()).Name("UpdateWebService");

        descriptor.Field(m => m.RemoveWebService(default)).Type<NonNullType<StringType>>()
            .Argument("id", arg => arg.Type<NonNullType<IdType>>()).Name("RemoveWebService");
    }
}

```

Codefragment 17: GRAPHQL-schema

In de QueryType worden de resolver-methoden van de Query-klasse gekoppeld aan de corresponderende GraphQL-velden, met behulp van de functionaliteit die Hot Chocolate biedt. Hier worden de velden GetAllWebServices en FindWebServiceById gedefinieerd met hun respectievelijke types en argumenten.

Op een vergelijkbare manier worden in de MutationType de resolver-methoden van de Mutation-klasse gekoppeld aan de mutatie-velden. Hier worden de velden UpdateWebService en RemoveWebService gedefinieerd met hun types en argumenten.

Query in GraphQL wordt gebruikt om gegevens op te vragen aan de API en kan vergeleken worden met een GET-request in REST.

Mutation daarentegen wordt gebruikt om gegevens te wijzigen of te muteren, dit kan vergeleken worden met een POST-request in REST.

Net zoals bij gRPC worden er nieuwe objecten aangemaakt die compatibel zijn met GraphQL.

```
public class WebServiceInputType : InputObjectType<WebServiceDTO>
{
    0 references
    protected override void Configure(IInputObjectTypeDescriptor<WebServiceDTO> descriptor)
    {
        descriptor.Field(x => x.WebServiceId).Type<IdType>();
        descriptor.Field(x => x.Name).Type<StringType>();
        descriptor.Field(x => x.Delay).Type<IntType>();
        descriptor.Field(x => x.Bundle).Type<BooleanType>();
        descriptor.Field(x => x.KeepSequence).Type<BooleanType>();
        descriptor.Field(x => x.RetryCount).Type<IntType>();
        descriptor.Field(x => x.RetryDelay).Type<IntType>();
        descriptor.Field(x => x.HistoryHours).Type<IntType>();
        descriptor.Field(x => x.Enabled).Type<BooleanType>();
        descriptor.Field(x => x.Instant).Type<BooleanType>();
        descriptor.Field(x => x.DestinationId).Type<StringType>();
        descriptor.Field(x => x.WebServiceGroupId).Type<StringType>();
        descriptor.Field(x => x.ElementIds).Type<ListType<StringType>>();
        descriptor.Field(x => x.CallIds).Type<ListType<StringType>>();
    }
}
```

Codefragment 18: GraphQL WebServiceInputType

Hot Chocolate maakt het mogelijk om op een gestructureerde en gevalideerde manier inputparameters te definiëren voor mutaties. Dit wordt bereikt door het gebruik van de `WebServiceInputType`, een klasse die is afgeleid van `InputObjectType` in GraphQL.NET, wat wordt ondersteund door Hot Chocolate. Deze `inputobjecttypes` fungeren als contracten tussen de client en de server, waarbij ze specifieke velden en hun types specificeren die kunnen worden gebruikt om gegevens naar de server te sturen.

Door het gebruik van een `InputObjectType` kan GraphQL een duidelijke structuur en validatie bieden voor de inputgegevens. Het stelt de client in staat om alleen de vereiste velden en hun waarden op te geven bij het uitvoeren van een mutatie, waardoor onnodige gegevens worden vermeden en de server alleen de relevante velden kan verwerken en bijwerken.

4.4.2 Client

```
public async Task<List<WebServiceDTO>> GetAllWebServicesGraph()
{
    string query = @"
    query {
        GetAllWebServices {
            webServiceId
            name
            delay
            bundle
            keepSequence
            retryCount
            retryDelay
            historyHours
            enabled
            instant
            destinationId
            webServiceGroupId
            elementIds
            callIds
        }
    }";
    HttpRequestMessage? request = new(HttpMethod.Post, graphqlEndpoint)
    {
        Content = new StringContent(JsonConvert.SerializeObject(new { query }), System.Text.Encoding.UTF8, "application/json")
    };
    Stopwatch? stopwatch = new();
    stopwatch.Start();
    try {
        HttpResponseMessage? response = await httpClient.SendAsync(request);
        stopwatch.Stop();
        decimal executionTime = stopwatch.ElapsedMilliseconds;
        long memory = CalculateSize(request) + CalculateSize(response);
        AddEndpointCall("GetAllWebServicesGraph", true, executionTime, memory);
        string? responseContent = await response.Content.ReadAsStringAsync();
        QueryResponse? queryResponse = JsonConvert.DeserializeObject<QueryResponse>(responseContent);
        return queryResponse?.Data?.GetAllWebServices;
    }
}
```

Codefragment 19: GraphQL endpoint aanspreken

Eerst wordt er een query opgesteld waarin de gewenste gegevens worden gespecificeerd. Vervolgens worden de benodigde variabelen aan de query toegevoegd. De query wordt omgezet naar het juiste JSON-formaat.

Hierna wordt de JSON-query verzonden als een HTTP POST-verzoek naar de API. De API verwerkt het verzoek en stuurt een antwoord terug. Het ontvangen antwoord wordt gedeserialiseerd naar het juiste query-responstype.

```
private class QueryResponse
{
    1 reference
    public QueryData Data { get; set; }
}

1 reference
private class QueryData
{
    1 reference
    public List<WebServiceDTO> GetAllWebServices { get; set; }
}
```

Codefragment 20: GraphQL QueryResponse

4.4.3 Conclusie

Net zoals gRPC is de werking van GraphQL anders dan die van REST. Het is een complexe structuur, zelfs om een basis op te zetten. Omdat GraphQL een flexibele en aanpasbare technologie is, kan de manier waarop het geïmplementeerd wordt sterk verschillen. Elk project heeft andere vereisten en

unieke kenmerken zodat het moeilijk kan zijn om tutorials te vinden die bij de specifieke behoeften aansluiten.

Bovendien is GraphQL een relatief jonge technologie, zodat het aanbod aan documentatie en tutorials nog beperkt is.

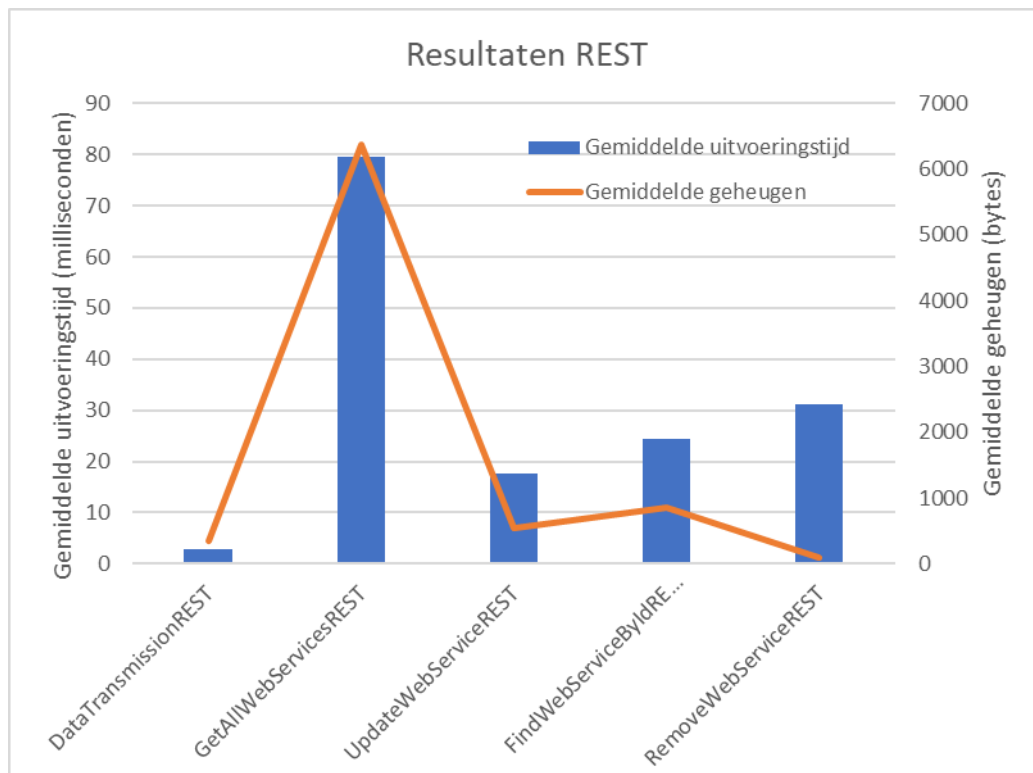
Hoewel de foutmeldingen in GraphQL over het algemeen behoorlijk informatief zijn, kan het voor onervaren gebruikers soms uitdagend zijn om ze volledig te begrijpen. Het kan voorkomen dat bij het debuggen van een GraphQL-applicatie bepaalde foutmeldingen enigszins verwarrend zijn, vooral als de query complex is of als er meerdere resolvers en gegevensbronnen betrokken zijn.

Het is wel de moeite waard om te vermelden dat de GraphQL-community groeit en actief is, waardoor er voortdurend nieuwe bronnen en tools beschikbaar komen om ontwikkelaars te ondersteunen.

5 Testresultaten

5.1 REST

De testresultaten van REST worden beschouwd als de referentiewaarden voor de andere structuren. De gemiddelde uitvoeringstijd en het gemiddelde geheugengebruik worden weergegeven in figuur 11.



Figuur 11 Grafiek Testresultaten REST

Tabel 3 Testresultaten REST

Functienaam	Gemiddelde Executie Tijd(ms)	Gemiddelde Geheugen(bytes)
DataTransmission	2,8779	347,7586
GetAllWebServices	79,5367	6373,35
UpdateWebService	17,5427	548
FindWebServiceById	24,3596	863,049
RemoveWebService	31,1667	102

DataTransmission:

- Gemiddelde Executie Tijd: 2,88 ms
- Gemiddeld geheugen: 348 bytes
- Uitleg: De DataTransmission-functie is verantwoordelijk voor het verzenden van gegevens. Er wordt verwacht dat het een korte uitvoeringstijd heeft, omdat het voornamelijk gaat om het

verzenden en ontvangen van gegevens zonder complexe bewerkingen. Het gemiddelde geheugen is relatief laag omdat er geen uitgebreide opslag nodig is voor gegevensoverdracht.

GetAllWebServices:

- Gemiddelde Executie Tijd: 79,54ms
- Gemiddeld Geheugen: 6373 bytes
- Uitleg: Deze functie haalt alle beschikbare webservices op. De relatief hogere uitvoeringstijd kan worden gelinkt aan het grote aantal webservices dat moet worden opgehaald en verwerkt. Het gemiddelde geheugen is hoger vanwege de opslagvereisten voor het bewaren van de opgehaalde gegevens.

UpdateWebService:

- Gemiddelde Executie Tijd: 17,54 ms
- Gemiddeld geheugen: 548 bytes
- Uitleg: De functie UpdateWebService wijzigt een bestaande webservice. De uitvoeringstijd is relatief korter omdat het gaat om het bijwerken van specifieke velden binnen een webservice. Het gemiddelde geheugen is lager in vergelijking met GetAllWebServices omdat er enkel een webservice wordt teruggestuurd.

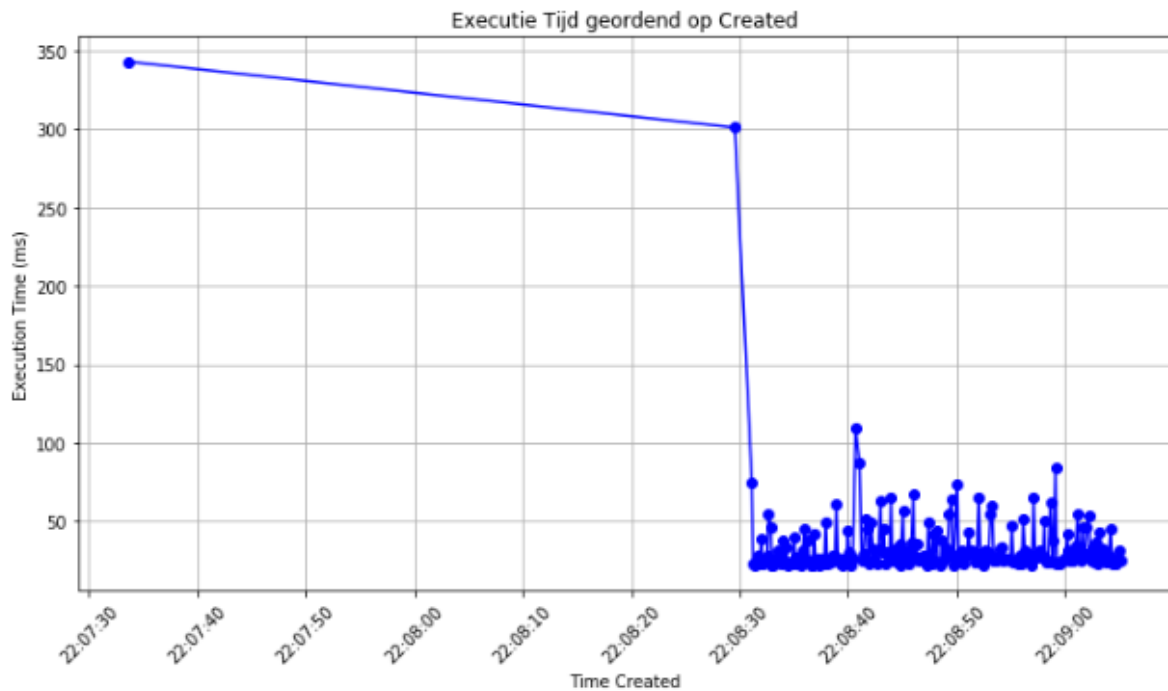
FindWebServiceById:

- Gemiddelde Executie Tijd: 24,36 ms
- Gemiddeld Geheugen: 963 bytes
- Uitleg: Deze functie zoekt naar een specifieke webservice. De uitvoeringstijd is relatief korter omdat het gaat om het doorzoeken van een specifiek record op basis van de ID, wat efficiënt kan worden gedaan. Het gemiddelde geheugen is lager in vergelijking met GetAllWebServices omdat er enkel een webservice wordt terug gestuurd.

RemoveWebService:

- Gemiddelde Executie Tijd: 31,17 ms
- Gemiddeld geheugen: 102 bytes
- Uitleg: De functie RemoveWebService verwijdert een webservice van het systeem. De uitvoeringstijd is relatief korter omdat het gaat om het verwijderen van een specifiek record uit de database. Het gemiddelde geheugen is laag omdat er enkel een ID wordt meegestuurd en een string ontvangen wordt.

Om een beter inzicht te hebben en te zien of de gemiddelde tijd wordt beïnvloed door piekwaarden, is het interessant om te bekijken hoe de uitvoeringstijd varieert per call.



Figuur 12 Grafiek Uitvoeringstijd FindWebServiceById REST

Uit bovenstaande grafiek kan er afgeleid worden dat vooral de eerste calls veel tijd in beslag nemen.

Tijdens de eerste calls moet de server van alles inladen, dit proces wordt ook wel de “warm-up phase” genoemd. De server moet mogelijk databaseverbindingen tot stand brengen, configuraties laden, cachingmechanismen instellen en andere noodzakelijke bewerkingen uitvoeren. Deze activiteiten dragen bij aan de verhoogde uitvoeringstijd van de eerste calls.

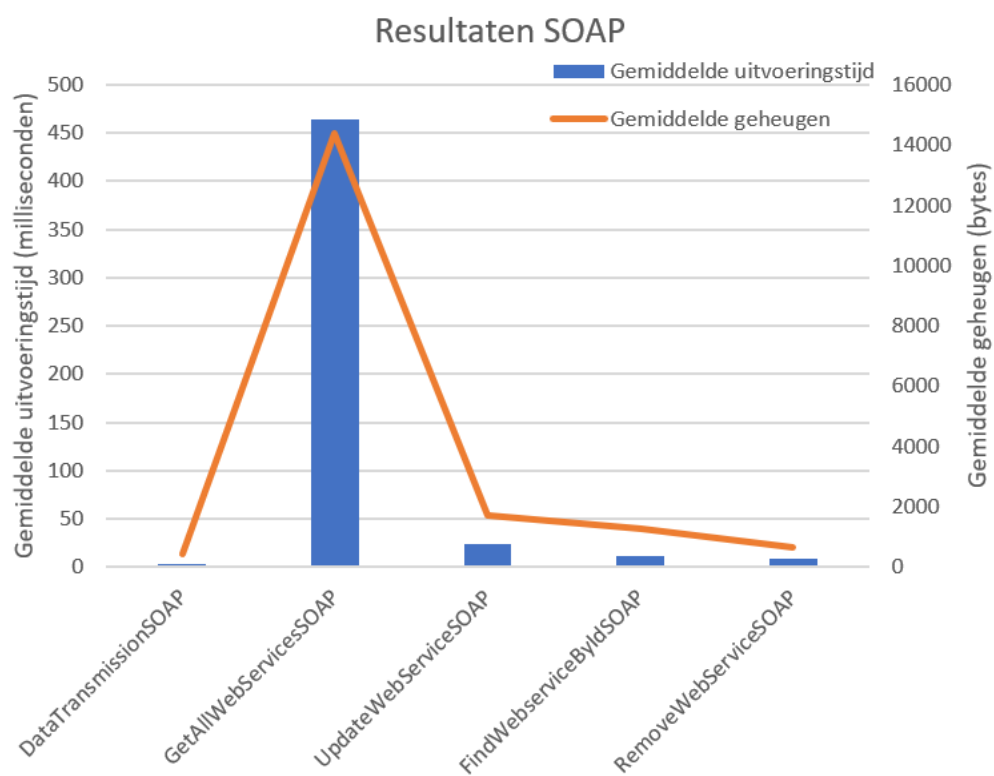
Als opeenvolgende calls worden gemaakt naar de API, komen er caching-mechanismen in het spel. Caching stelt de server in staat om de resultaten van eerdere verzoeken op te slaan en deze direct te verwerken voor volgende identieke verzoeken. Door te voorkomen dat dezelfde gegevens herhaaldelijk moeten worden verwerkt, verkort caching de uitvoeringstijd aanzienlijk. Naarmate er meer calls worden gedaan, neemt de kans op het ophalen van gegevens in de cache toe, wat resulteert in snellere uitvoeringstijden.

5.2 SOAP

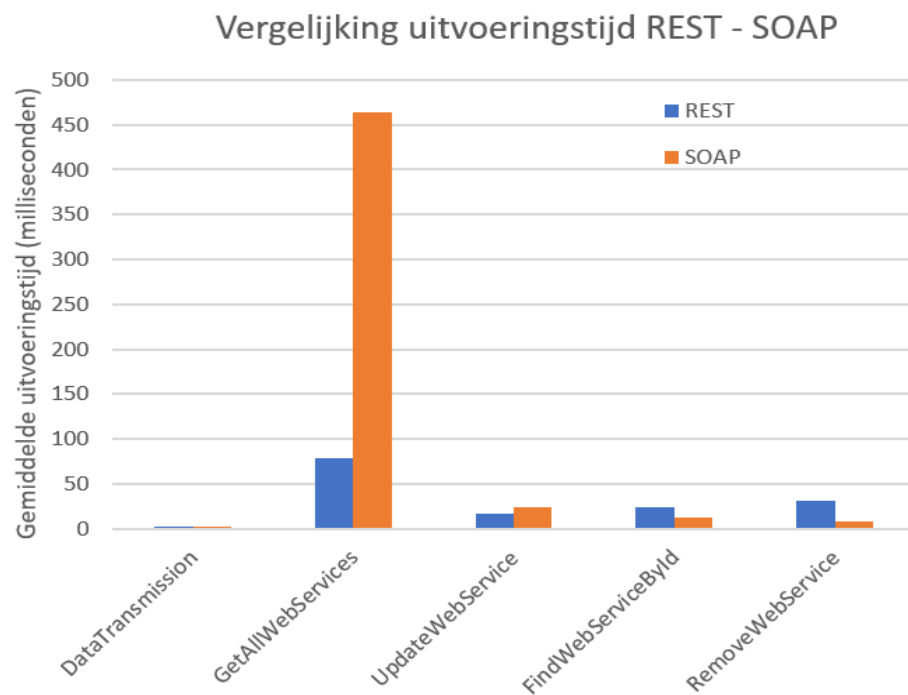
Tabel 4 en figuur 13 geven de testresultaten weer van de SOAP-API. Figuren 14 en 15 tonen het verschil ten opzichte van de REST-API.

Tabel 4 Testresultaten SOAP

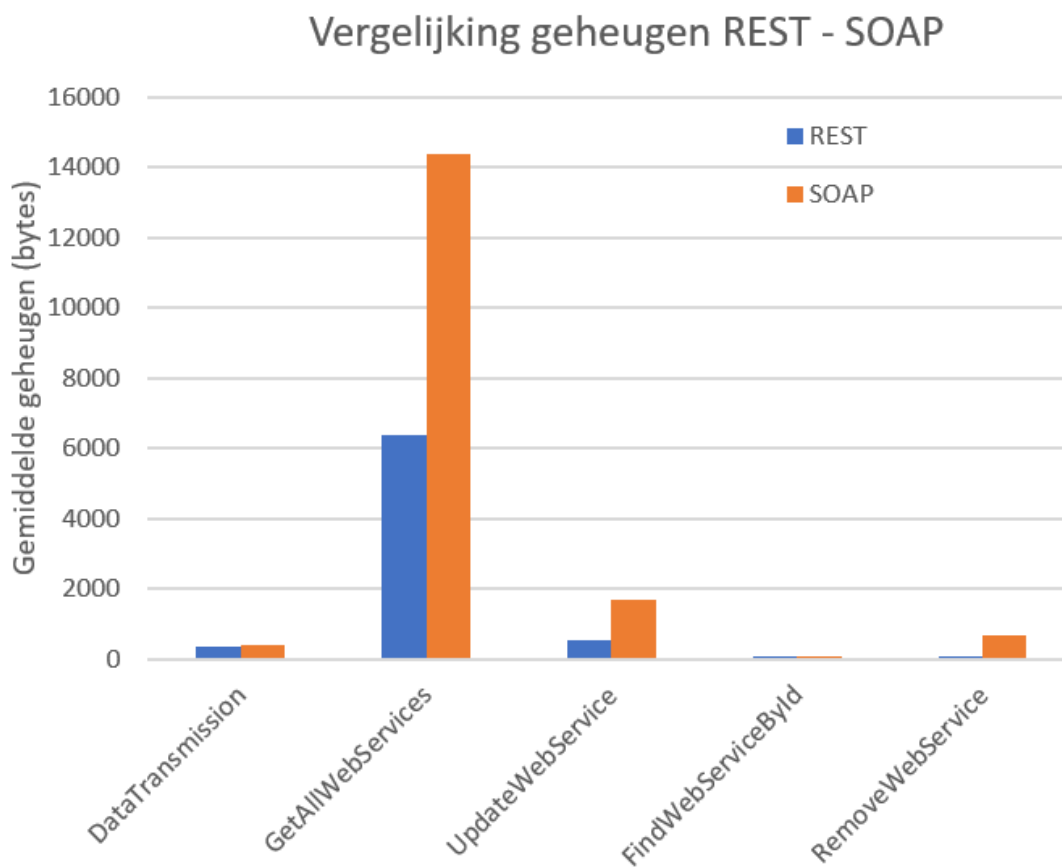
Functienaam	Gemiddelde Executie Tijd(ms)	Gemiddelde Geheugen(bytes)
DataTransmission	3,3796	422,9254
GetAllWebServices	463,4383	14368,7
UpdateWebService	24,6533	1694
FindWebServiceById	12,07	1261
RemoveWebService	8,9583	661



Figuur 13 Grafiek Testresultaten SOAP

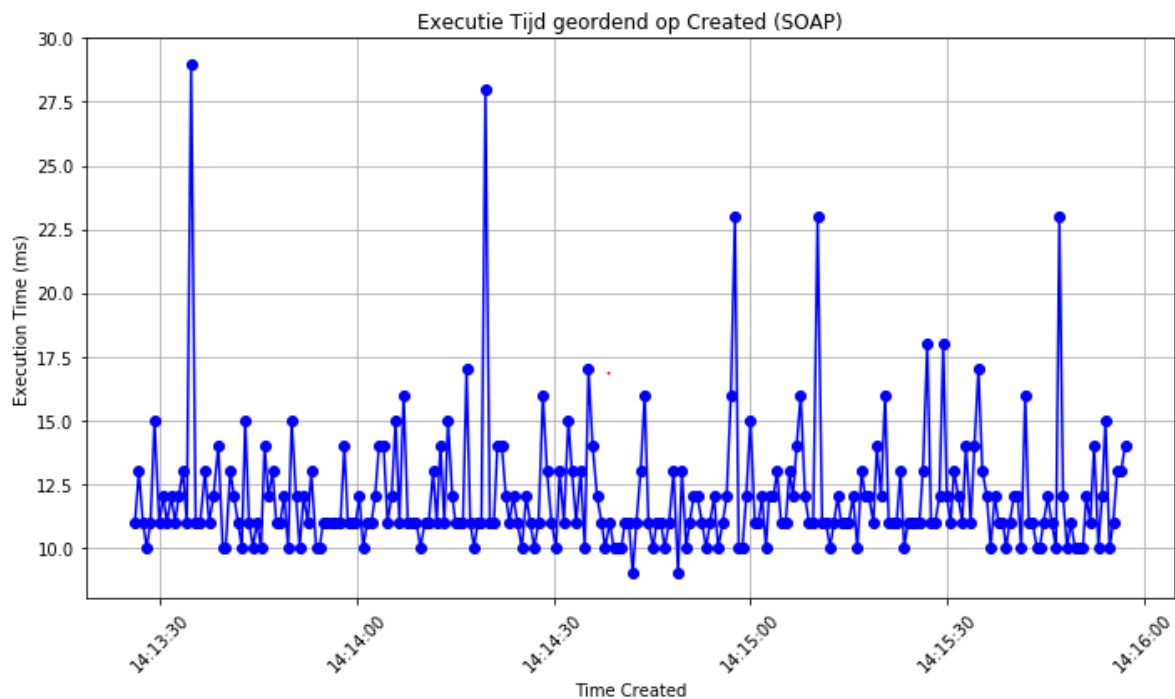


Figuur 14 Grafiek Vergelijking Uitvoeringstijd REST-SOAP



Figuur 15 Grafiek Vergelijking Geheugen REST-SOAP

SOAP lijkt iets trager te zijn dan REST in het algemeen. Dit kan te wijten zijn aan het overhead van het SOAP-protocol, dat extra XML-verwerking en parsing vereist. SOAP gebruikt XML om gegevens te coderen, wat extra verwerkingstijd kan vergen in vergelijking met REST dat meestal lichtere formaten zoals JSON gebruikt. REST kan daarentegen rechtstreeks gebruikmaken van de HTTP-protocollagen zonder extra lagen, waardoor het sneller kan zijn. In een paar gevallen is SOAP wel sneller. Dit lijkt voor te vallen wanneer de functie weinig geheugen in beslag neemt. In deze gevallen kan het zijn dat door dat REST tijd verliest met zijn warm-up phase.



Figuur 16 Grafiek Uitvoeringstijd SOAP

Het verloop van de uitvoeringstijd per call (Figuur 16) geeft een ander beeld dan bij REST. We zien hier kleine pieken. Dit kan komen door fluctuaties in de databank/API verwerkingstijd.

In tegenstelling tot bij REST zien we geen heel grote piek bij de eerste calls, dit komt doordat SOAP minder gevoelig is voor de warm-up phase.

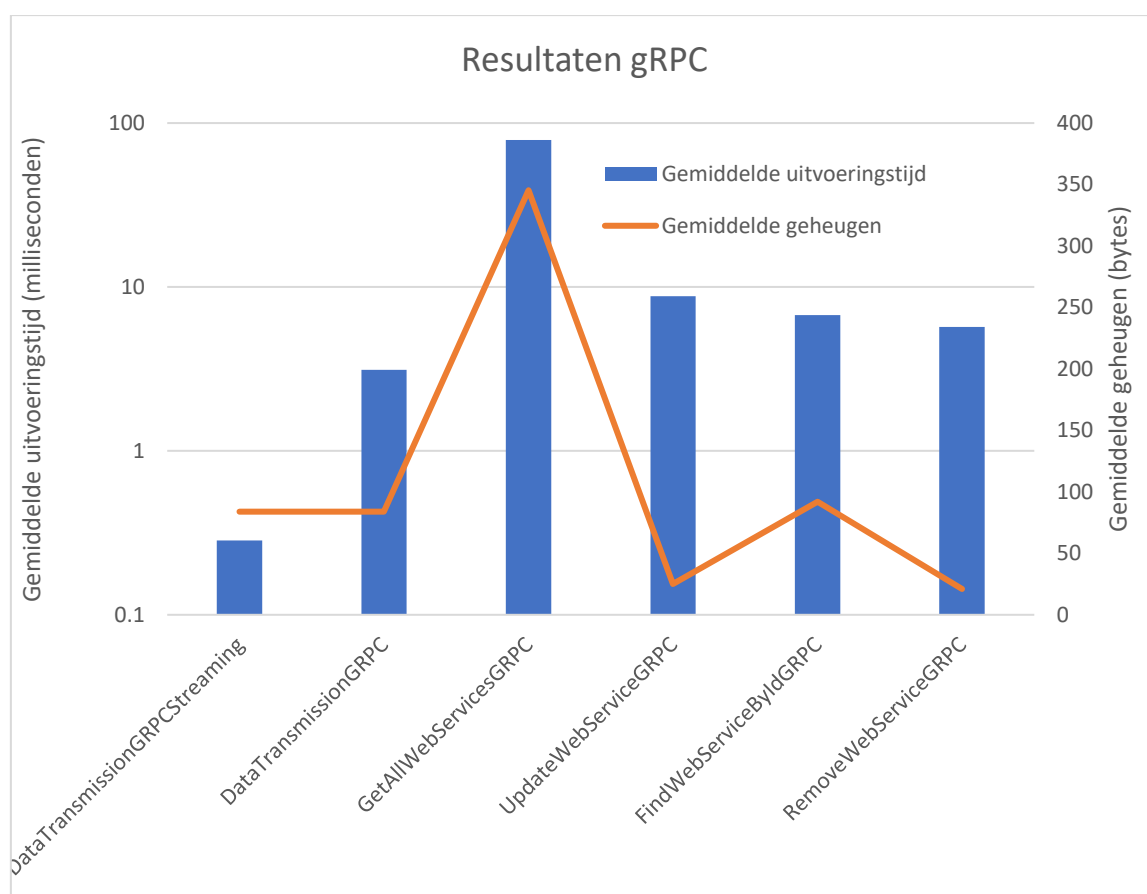
Bij SOAP worden er meestal WSDL en XML-berichten gebruikt om de communicatie tussen de client en de server te definiëren. Deze XML-berichten bevatten gedetailleerde informatie over de structuur en inhoud van de gegevens die worden uitgewisseld. Bij elke SOAP-request moet de server de XML-berichten parsen en verwerken, wat enige overhead met zich meebrengt. Dit kan resulteren in relatief hogere uitvoeringstijden voor SOAP-webservices in vergelijking met REST-webservices.

5.3 GRPC

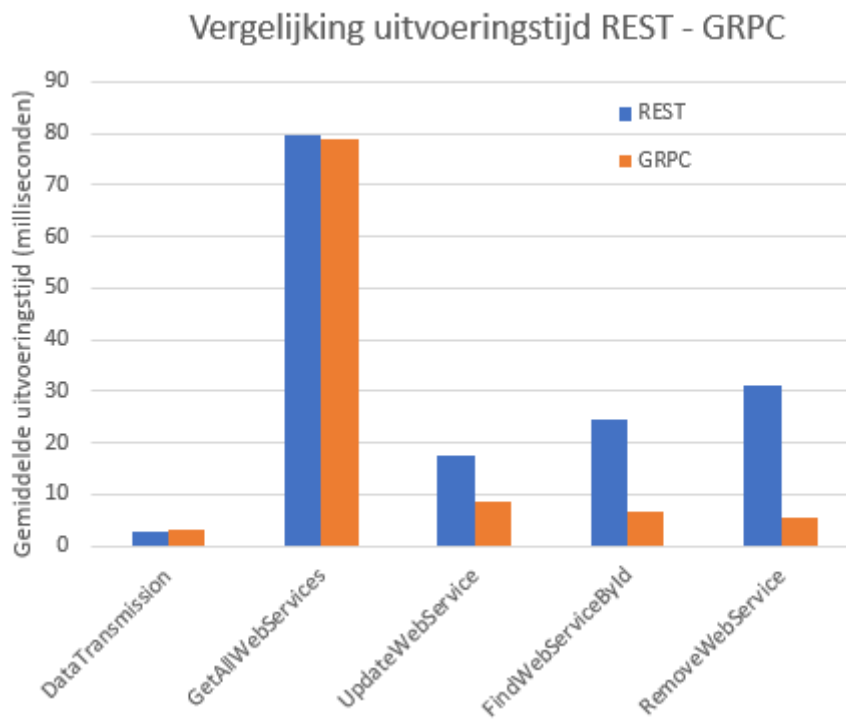
Tabel 5 en figuur 17 geven de testresultaten weer van de gRPC-API. Figuren 18 en 19 tonen het verschil ten opzichte van de REST-API.

Tabel 5 Testresultaten GRPC

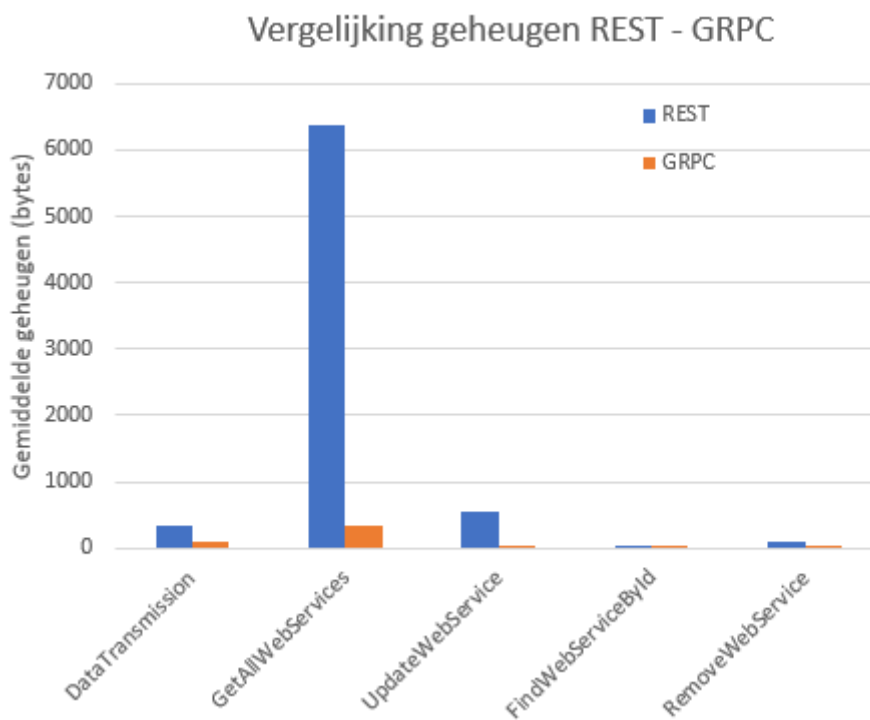
Functienaam	Gemiddelde Executie Tijd(ms)	Gemiddelde Geheugen(bytes)
DataTransmission	3,1246	84
GetAllWebServices	78,7133	345,354
UpdateWebService	8,7667	25
FindWebServiceById	6,7333	92
RemoveWebService	5,7136	21



Figuur 17 Grafiek Testresultaten GRPC



Figuur 18 Grafiek Vergelijking Uitvoeringstijd REST-GRPC



Figuur 19 Grafiek Vergelijking Geheugen REST-GRPC

Uit de testresultaten blijkt dat gRPC-calls over het algemeen zeer snel zijn, vergelijkbaar met of zelfs sneller dan REST-calls. Dit is te danken aan het gebruik van het efficiënte Protocol Buffers-protocol (protobuf) voor de berichtuitwisseling. Protocol Buffers maakt gebruik van een compact binair

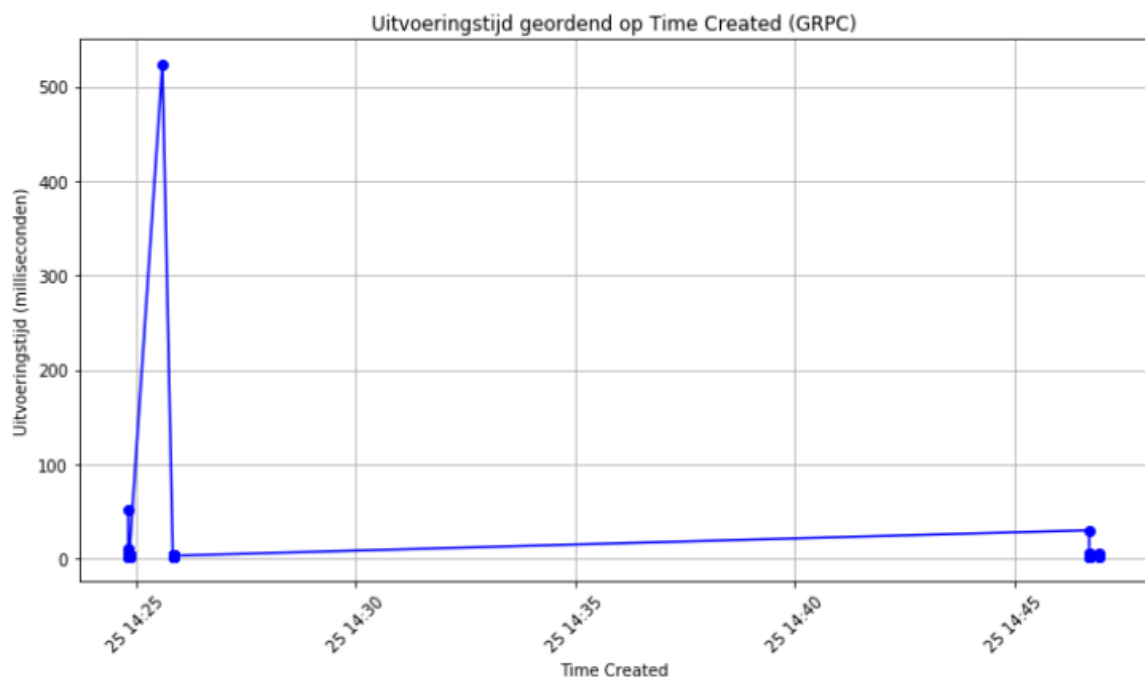
formaat, wat resulteert in kleinere bestandsgroottes in vergelijking met XML-gebaseerde protocollen zoals SOAP.

Het lichte gewicht van gRPC komt voort uit het gebruik van het HTTP/2-protocol als transportlaag. HTTP/2 biedt ondersteuning voor multiplexing en compressie van gegevens, waardoor efficiëntere communicatie mogelijk is tussen de client en de server. Dit draagt bij tot een betere prestatie van gRPC, vooral bij het verwerken en verzenden van grote hoeveelheden gegevens.

Het is belangrijk op te merken dat voor de functie GetAllWebServices zowel REST als gRPC enkele pieken in uitvoeringstijden laten zien. Dit suggereert een mogelijk probleem met de database- of queryprestaties die specifiek zijn voor deze functie. Verder onderzoek naar de databaseprestaties, query-optimalisatie en processen voor het ophalen van gegevens zou nuttig zijn om de onderliggende oorzaak van deze pieken te identificeren.

In tegenstelling tot SOAP, waarbij de nadruk ligt op uitgebreide functionaliteit en flexibiliteit, richt gRPC zich op hoge prestaties. Het maakt gebruik van een op servicecontracten gebaseerd model, waarbij de communicatie tussen clients en servers wordt gedefinieerd door middel van een Interface Definition Language (IDL). Dit zorgt voor een gestroomlijnde ontwikkeling van API's en bevordert de interoperabiliteit tussen verschillende programmeertalen.

Globaal genomen biedt gRPC snellere uitvoeringstijden en efficiënter geheugengebruik in vergelijking met REST. Het maakt gebruik van Protocol Buffers voor efficiënte data-uitwisseling en profiteert van de voordelen van het HTTP/2-protocol voor verbeterde communicatieprestaties. De focus op hoge prestaties en de gestandaardiseerde ontwikkeling van API's maken gRPC een krachtige keuze voor moderne applicatieontwikkeling.



Figuur 20 Grafiek Uitvoeringstijd GRPC

Het verloop van de uitvoeringstijd per call wordt weergegeven in figuur 20. In de grafiek kunnen we een consistente trend van lage uitvoeringstijden waarnemen, met uitzondering van een grote piek na ongeveer 30 calls. Deze piek kan mogelijk worden toegeschreven aan factoren zoals overbelasting

van de Kreonta Edge-server of de database. Het is belangrijk op te merken dat deze piek na korte tijd verdwijnt, waarna de uitvoeringstijden weer op een laag niveau stabiliseren.

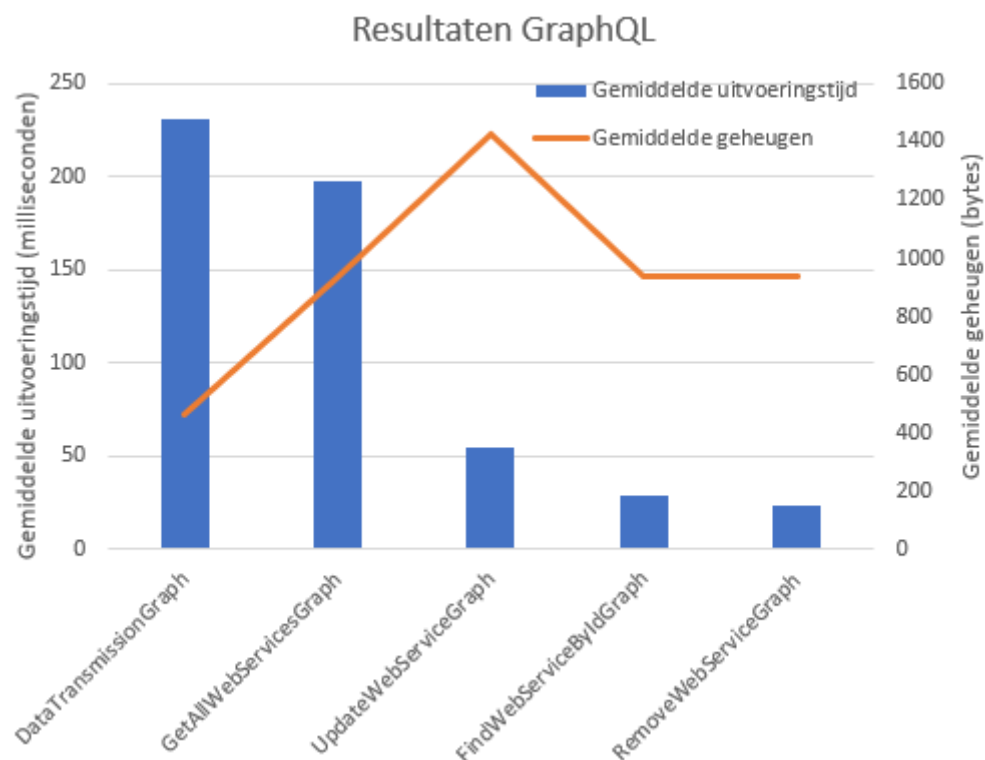
Het is interessant om op te merken dat dit fenomeen niet specifiek verband houdt met een warm-up phase bij gRPC. In tegenstelling tot sommige andere technologieën, vereist gRPC geen specifieke initiatie- of warm-up phase om verbindingen tot stand te brengen of interne resources te initialiseren. Bij gRPC wordt de verbinding efficiënt opgezet en kunnen oproepen direct worden uitgevoerd zonder extra overhead.

5.4 GRAPHQL

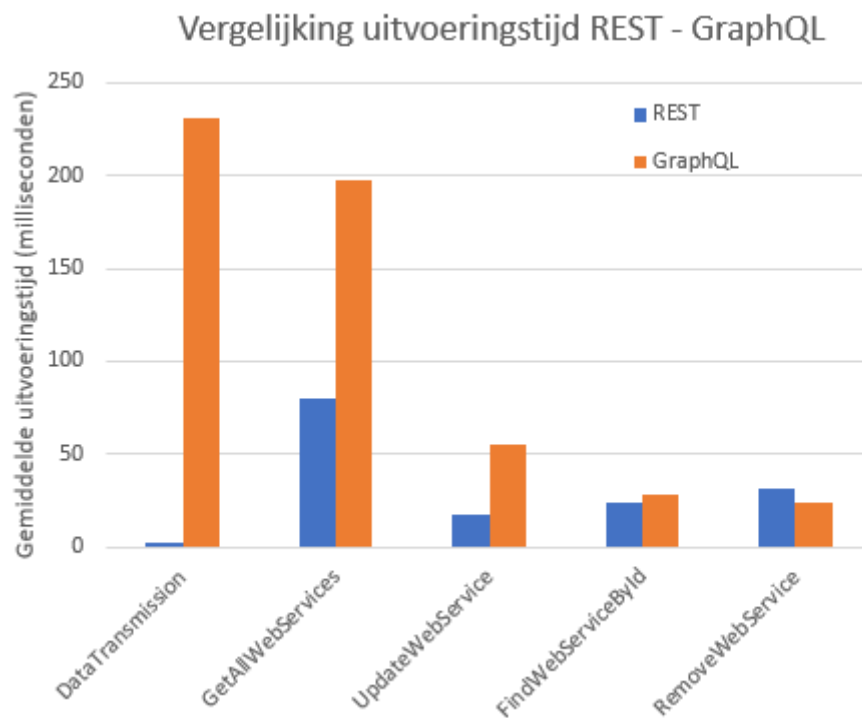
Tabel 6 en figuur 21 geven de testresultaten weer van de GraphQL-API. Figuren 22 en 23 tonen het verschil ten opzichte van de REST-API.

Tabel 6 Testresultaten GraphQL

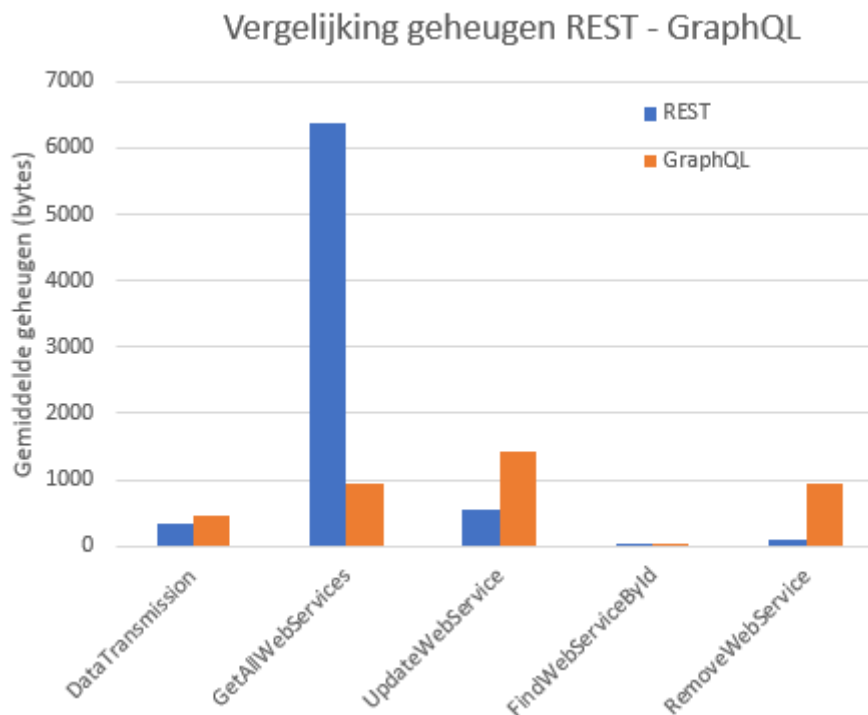
Funcienaam	Gemiddelde Executie Tijd(ms)	Gemiddelde Geheugen(bytes)
DataTransmission	231,2279	460
GetAllWebServices	197,515	941
UpdateWebService	54,8717	1424
FindWebServiceById	28,7817	941
RemoveWebService	23,9067	941



Figuur 21 Grafiek Testresultaten GRAPHQL



Figuur 22 Grafiek Vergelijking Uitvoeringstijd REST-GRAPHQL



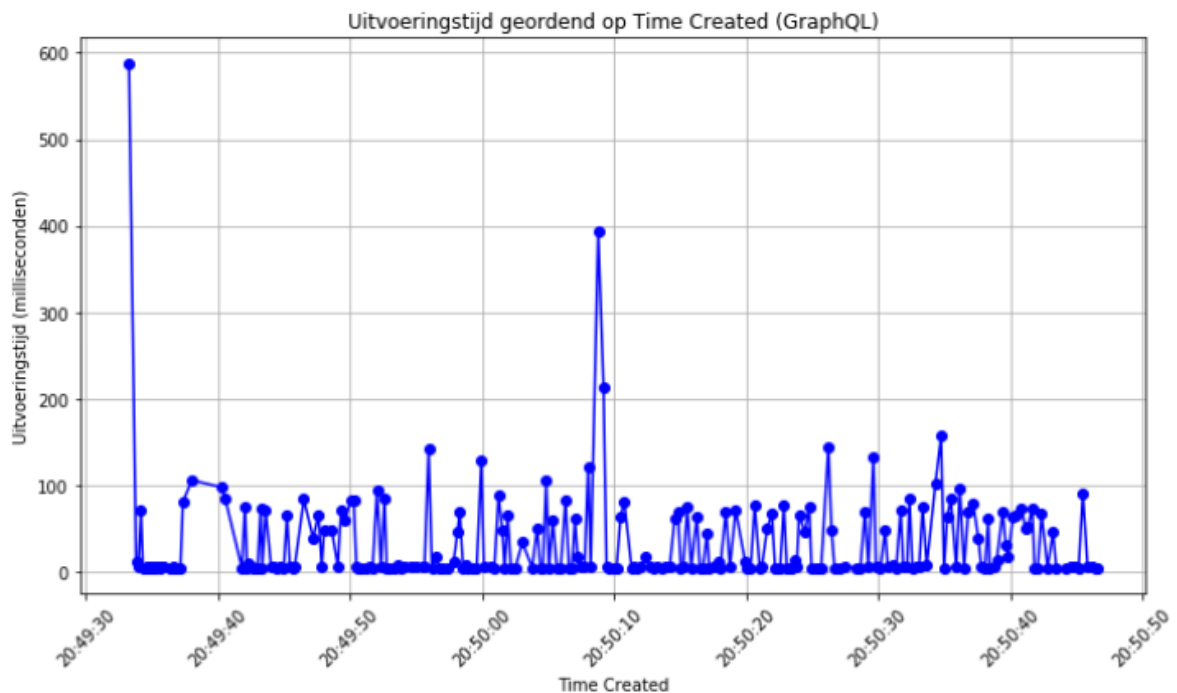
Figuur 23 Grafiek Vergelijking Geheugen REST-GRAPHQL

Uit de testresultaten blijkt dat GraphQL in vergelijking met REST in sommige gevallen langere uitvoeringstijden vertoont. Bijvoorbeeld, de functie GetAllWebServicesREST heeft een gemiddelde uitvoeringstijd van 79,54 ms, terwijl GetAllWebServicesGraph een gemiddelde uitvoeringstijd heeft van 197,52 ms. Dit suggereert dat de GraphQL-implementatie in dit specifieke geval trager is dan de REST-implementatie.

Er kunnen verschillende redenen zijn waarom GraphQL in dit geval trager is:

Overfetching van gegevens: mogelijk zijn de queries niet optimaal geschreven. Hierdoor kan er overfetching gebeuren van gegevens.

N+1-probleem: In sommige gevallen kan GraphQL te maken krijgen met het N+1-probleem. Dit doet zich voor wanneer een enkele GraphQL-query meerdere achterliggende databaseverzoeken vereist om de gevraagde gegevens op te halen. Als gevolg hiervan kan de uitvoeringstijd toenemen, vooral wanneer er grote hoeveelheden gegevens worden opgevraagd. [11]



Figuur 24 Grafiek Uitvoeringstijd GRAPHQL

Bij de grafiek van de uitvoeringstijd per call van GraphQL zien we een opmerkelijk patroon met betrekking tot de uitvoeringstijden. Er is een aanzienlijke piek bij de eerste call en vervolgens nog een piek in het midden van de calls. Buiten deze piekmomenten blijven de uitvoeringstijden echter relatief stabiel.

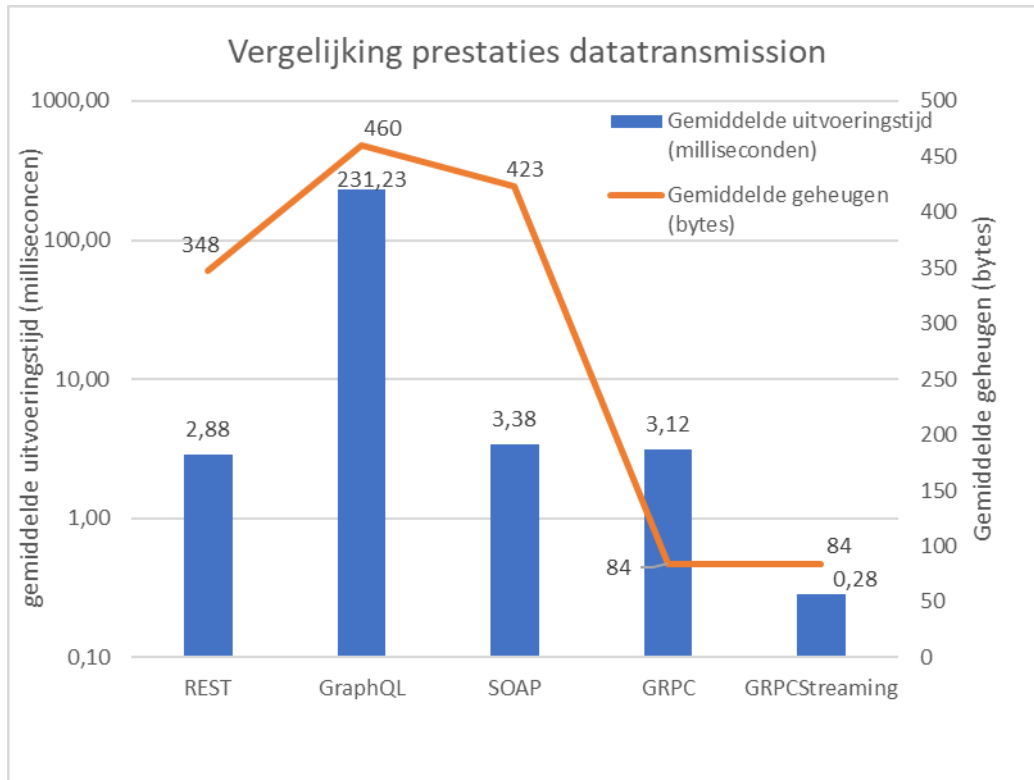
De piek bij de eerste call kan worden toegeschreven aan verschillende factoren. Bij het gebruik van GraphQL moet de server de query en de bijbehorende resolver-logica analyseren en verwerken om de juiste gegevens terug te geven. Dit kan enige overhead veroorzaken bij de eerste call, aangezien de nodige resources moeten worden geïnitialiseerd en de resolver-logica moet worden uitgevoerd.

De piek in het midden van de calls kan te wijten zijn aan verschillende redenen, waaronder mogelijk overbelasting van de server of andere externe factoren die van invloed kunnen zijn op de prestaties.

Het feit dat de uitvoeringstijden buiten de piekmomenten relatief stabiel zijn, geeft aan dat GraphQL in staat is om consistent te presteren zodra de initiële overhead is overwonnen en de server zijn resources optimaal kan benutten.

5.5 DataTransmission

Deze functie is voor Kreonta Edge de belangrijkste vermits DataTransmission een permanente service is die draait, terwijl de andere endpoints in de praktijk maar een paar keer worden gebruikt. Daarom wordt deze functie apart besproken. De prestaties van de verschillende structuren worden in figuur 25 weergegeven. Omdat er grote verschillen zijn qua uitvoeringstijd, werd een logaritmische schaal gebruikt om de verschillende uitvoeringstijden aanschouwelijk te kunnen voorstellen.



Figuur 25 Grafiek Vergelijking prestaties DataTransmission

DataTransmissionGraph heeft de langste gemiddelde uitvoeringstijd van 231.2279 ms en het hoogste gemiddelde geheugengebruik van 460 bytes. Dit kan wijzen op een inefficiënte implementatie of zwaardere verwerking van gegevens in de GraphQL-datatransmissiemethode.

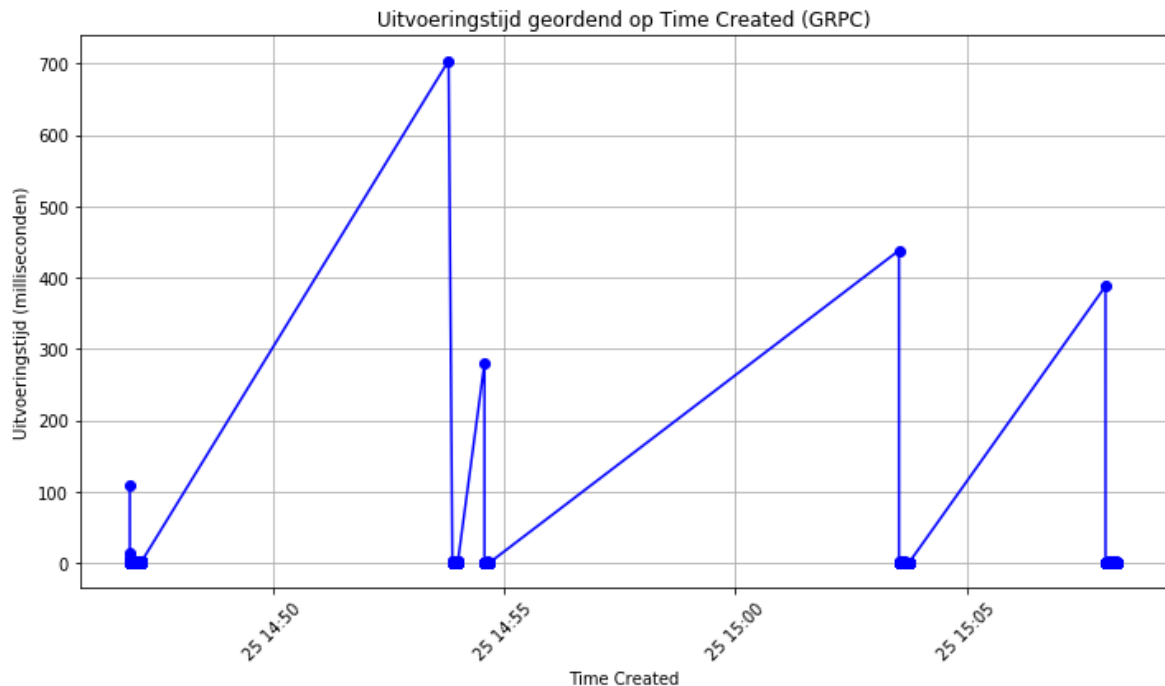
Als we GRPCStreaming buiten beschouwing laten, heeft DataTransmissionREST de kortste gemiddelde uitvoeringstijd van 2,88 ms en een gemiddeld geheugengebruik van 348 bytes. Het presteert beter qua uitvoeringstijd en geheugen in vergelijking met DataTransmissionGraph, SOAP en GRPC.

DataTransmissionSOAP heeft een iets langere gemiddelde uitvoeringstijd van 3,38 ms en een gemiddeld geheugengebruik van 423 bytes. Het is vergelijkbaar met DataTransmissionREST, maar met iets hogere uitvoeringstijden en geheugengebruik.

DataTransmissionGRPC heeft een korte gemiddelde uitvoeringstijd van 3,12 ms. Het gebruikt ook relatief weinig geheugen, met een gemiddeld geheugengebruik van 84 bytes. Het is vergelijkbaar met DataTransmissionREST, maar met iets hogere uitvoeringstijden en maar een kleiner geheugengebruik.

DataTransmissionGRPCStreaming heeft de korste gemiddelde uitvoeringstijd van 0,28 ms. Zoals de gewone gRPC heeft het een geheugengebruik van 84 bytes.

Hieruit kan besloten worden dat de resultaten van GRPCStreaming er bovenuit steken.

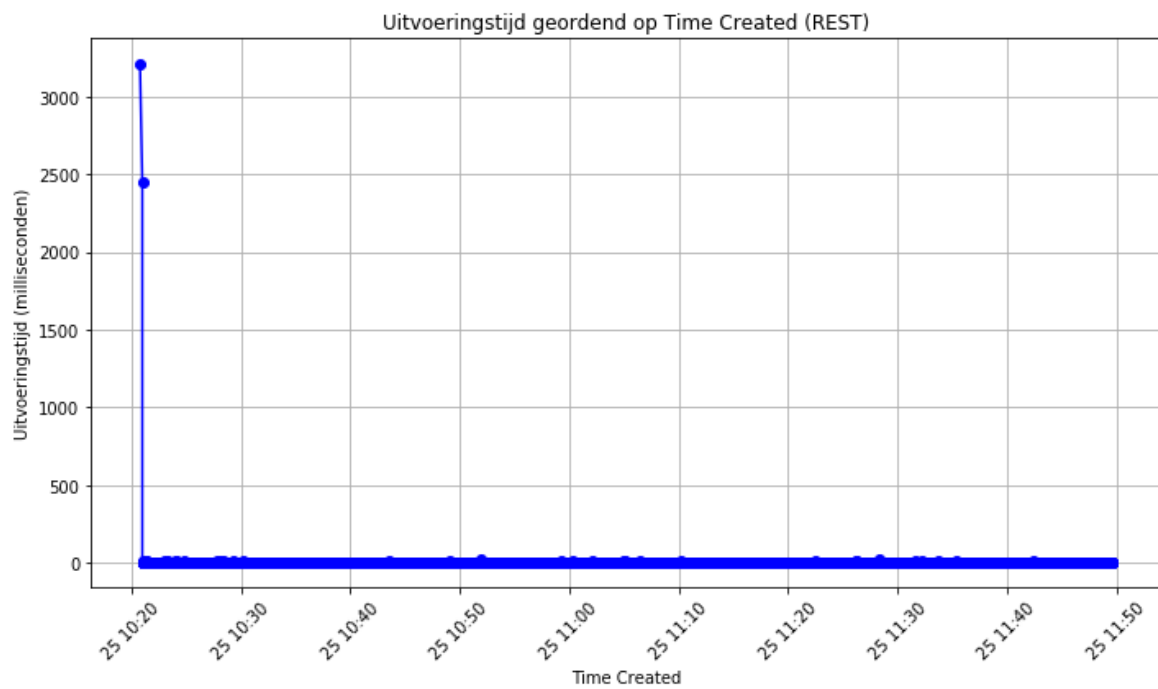


Figuur 26 Grafiek Uitvoeringstijd DataTransmissionGRPCStreaming

Uit de grafiek van uitvoeringstijden per call kan afgeleid worden dat GRPCStreaming redelijk stabiel is buiten vier grote pieken die eruit springen. De pieken kunnen verklaard worden:

Serververwerkingstijd: De pieken kunnen optreden wanneer de server meer tijd nodig heeft om de ontvangen gegevens te verwerken. Dit kan te wijten zijn aan verschillende factoren, zoals complexe verwerking van gegevens of andere taken die de server uitvoert parallel aan het verwerken van de gegevens.

Lokale systeembronnen: De pieken kunnen worden beïnvloed door de beschikbaarheid en het gebruik van lokale systeembronnen, zoals de CPU en het geheugen. Als andere processen of taken op het systeem de beschikbare middelen in beslag nemen, kan dit leiden tot variabiliteit in de uitvoeringstijden.



Figuur 27 Grafiek Uitvoeringstijd DataTransmissionREST

We zien een gelijkaardige grafiek bij de originele REST-endpoint, waarbij de uitvoeringstijd tijdens de warm-up phase hoog is en daarna stabiel laag blijft.

Conclusie

Uit de analyse van testresultaten na refactoring blijkt gRPC de meeste geschikte API-structuur om de prestatie van de applicatie Kreonta Edge te verhogen. De gRPC-structuur overtuigt op het gebied van prestaties, met uitstekende uitvoeringstijden en een efficiënt geheugengebruik. Het is een krachtig datatransmissieprotocol dat, dankzij de mogelijkheid om streaming te gebruiken, vooral uitblinkt bij toepassingen waar snelle en betrouwbare gegevensoverdracht van groot belang is.

Hoewel gRPC de voorkeur verdient voor veel toepassingen, zijn er ook situaties waarin andere datatransmissiemethoden hun eigen sterke punten hebben. REST blijft een goede keuze voor kleine, eenvoudige API's waar snelheid niet de hoogste prioriteit heeft. Het biedt eenvoudige implementatie, brede ondersteuning in verschillende programmeertalen en platformafhankelijkheid.

SOAP daarentegen vertoont slechte resultaten in termen van snelheid en efficiëntie. Het protocol is complex en heeft minder goede resultaten laten zien. Hoewel SOAP voordelen heeft in enterprise-omgevingen met betrekking tot interoperabiliteit en beveiliging, kan het in de context van de applicatie Kreonta Edge niet opwegen tegen de prestatievoordelen van gRPC.

Bij het evalueren van GraphQL bleek dat de theoretische voordelen van een snelle werking en efficiënt geheugengebruik niet werden weerspiegeld in de testresultaten. Mogelijk zijn niet alle mogelijkheden van GraphQL optimaal benut tijdens de implementatie, wat resulteerde in minder bevredigende prestaties. Bovendien werd het implementeren van GraphQL ervaren als complex en uitdagend.

Het maken van de juiste keuze tussen gRPC, REST, SOAP en GraphQL hangt dus af van de specifieke vereisten en kenmerken van de applicatie. Het is belangrijk om de noden van het project zorgvuldig af te wegen, inclusief de gewenste prestaties, complexiteit, interoperabiliteit en flexibiliteit van de datatransmissiemethode. Door het belang van datatransmissie is bij Kreonta Edge de keuze voor gRPC echter onbetwistbaar. De mogelijkheid tot streaming biedt een grote meerwaarde.

Literatuurlijst

- [1] „jvatpoint,” [Online]. Available: <https://www.jvatpoint.com/soap-and-rest-web-services>.
- [2] R. Crowley, „Youtube,” NDC Conferences, 28 Februari 2020. [Online]. Available: https://www.youtube.com/watch?v=l_P6m3JTyp0&t=2854s.
- [3] „altexsoft.com,” Altexsoft, 19 November 2022. [Online]. Available: <https://www.altexsoft.com/blog/rest-api-design/>.
- [4] „ionos.com,” Ionos, 7 Oktober 2020. [Online]. Available: <https://www.ionos.com/digitalguide/server/know-how/an-introduction-to-grpc/>.
- [5] S. Esemé, „Kinsta.com,” Kinsta, 26 September 2022. [Online]. Available: <https://kinsta.com/nl/blog/graphql-vs-rest/>.
- [6] K. Nayyeri, „Dzone,” 25 Mei 2011. [Online]. Available: <https://dzone.com/articles/darkness-behind-datetimenow>.
- [7] A. Monus, „Raygun,” 17 Oktober 2022. [Online]. Available: <https://raygun.com/blog/soap-vs-rest-vs-json/>.
- [8] „Stoplight.io,” Stoplight, [Online]. Available: <https://stoplight.io/api-types/soap-api>.
- [9] „Redhat.com,” RedHat, 8 April 2019. [Online]. Available: <https://www.redhat.com/en/topics/integration/whats-the-difference-between-soap-rest>.
- [10] „Mulesoft.com,” Mulesoft, [Online]. Available: [https://www.mulesoft.com/api-university/what-grpc-api-and-how-does-it-work#:~:text=gRPC%20is%20an%20API%20framework%20that%20supports%20point%2Dto%2Dpoint,of%20the%20World%20Wide%20Web\)..](https://www.mulesoft.com/api-university/what-grpc-api-and-how-does-it-work#:~:text=gRPC%20is%20an%20API%20framework%20that%20supports%20point%2Dto%2Dpoint,of%20the%20World%20Wide%20Web)..)
- [11] J. Wallace, „Hygraph.com,” hygraph, 20 december 2022. [Online]. Available: <https://hygraph.com/blog/graphql-n-1-problem>.