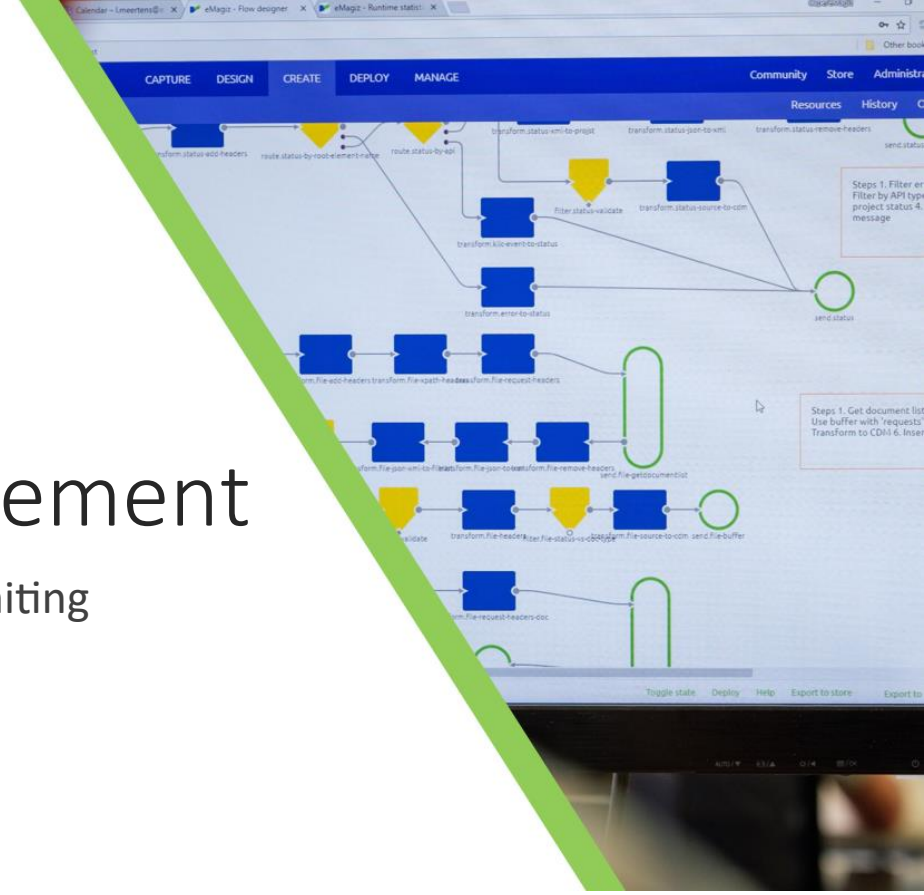


Traffic Management

eMagiz throttling & rate limiting

15-1-2024

Student
Naam: Edwin Heuver
Studentnummer: 139566
Email: 139566@student.saxion.nl

Bedrijf
Bedrijf: eMagiz b.v.
Bedrijfsbegeleider: Bas Elzinga

Onderwijsinstelling
Instelling: Hogeschool Saxion
Locatie: Enschede
Afstudeerbegeleider: Jan Jaap Sandee

Voorwoord

Voor u ligt de scriptie getiteld “Traffic management: eMagiz Throttling & Rate limiting”. Deze scriptie is geschreven ter afronding van mijn vierjarige studie HBO-ICT, richting Software Engineering aan Saxion Hogeschool te Enschede.

In de periode van september 2023 tot februari 2024 heb ik in opdracht van eMagiz B.V. onderzoek verricht naar de implementatie van throttling en rate limiting binnen verschillende integratiepatronen op het eMagiz-platform. Gebaseerd op de resultaten van dit onderzoek heb ik een bijdrage aan een open source project mogen leveren waarmee rate limiting configuraties gewijzigd kunnen worden in een gedistribueerde omgeving zonder dat hiervoor een nieuwe deployment nodig is.

Deze opdracht heeft mij de kans gegeven veel nieuwe kennis op te doen over de verschillende integratiepatronen in het eMagiz platform en de achterliggende architectuur. Ook heb ik veel geleerd over diverse manieren van throttling en rate limiting en hoe verschillende libraries dit implementeren. Ik ben dankbaar dat eMagiz mij de kans heeft geboden om een bijdrage te leveren aan dit open source project, waardoor ik nieuwe kennis en ervaring heb kunnen opdoen over onder andere het Spring framework en diverse soorten cache providers.

Graag wil ik een aantal personen bedanken voor hun steun en bijdrage aan dit afstudeeronderzoek. Allereerst wil ik mijn bedrijfsbegeleider Bas Elzinga bedanken voor de uitstekende begeleiding en waardevolle inzichten tijdens de wekelijkse voortgangsgesprekken en reviews. Zijn technische kennis heeft mij erg geholpen bij het verkrijgen van inzicht in de werking van de verschillende integratiepatronen. Daarnaast waren onze sparringsessies, zeker in de beginfase van het project, een belangrijke bijdrage voor nieuwe ideeën wanneer ik vast was gelopen.

Verder wil ik mijn dank uitspreken naar Samet Kaya en Mark De la Court voor hun bijdrage in het bedenken en verstrekken van de opdracht en het in kaart brengen van de verschillende requirements. Ook wil ik mijn collega’s van het development team bedanken, met in het bijzonder Jelle Smits, voor hun hulp bij de ontwikkeling van het prototype en de sparringsessies bij het koffiezetapparaat.

Ten slotte wil ik mijn schoolbegeleider Jan Jaap Sandee bedanken voor de goede begeleiding, feedback en adviezen gedurende het afstudeertraject.

Edwin Heuver

Hengelo, 21 december 2023

Samenvatting

Het eMagiz platform geeft gebruikers de mogelijkheid om gemakkelijk data- en integratie oplossingen te realiseren zonder dat daar uitgebreide programmeerkennis voor nodig is. Bij het maken van deze integraties hebben gebruikers de keuze uit verschillende soorten integratiepatronen. Geen van deze integratiepatronen biedt echter de mogelijkheid om dataverkeer eenvoudig te limiteren, terwijl daar voor de API-gateway- en Event Streaming patronen wel vraag naar is.

Het doel van dit project is het realiseren van een prototype waarmee beperkingen geconfigureerd kunnen worden voor de toegestane hoeveelheid dataverkeer van API-gateway en Event Streaming integraties. Om te onderzoeken hoe dit gerealiseerd kan worden is de volgende onderzoeksvraag opgesteld: *Hoe kan een generieke oplossing ontwikkeld worden waarmee throttling en/of rate limiting voor zowel API-gateway als Event Streaming geïmplementeerd kan worden?*

Om antwoord te kunnen geven op deze onderzoeksvraag is begonnen met het in kaart brengen van de requirements doormiddel van interviews met de belangrijkste stakeholders. Vervolgens is een analyse uitgevoerd van de werking van- en overeenkomsten tussen beide integratiepatronen. Hieruit bleek dat een generieke oplossing niet mogelijk is binnen de gestelde requirements, waarmee de onderzoeksvraag in principe beantwoord was.

In overleg met de stakeholders is toen besloten de focus te leggen op het realiseren van een oplossing voor de API-gateway integraties. Hiervoor is onderzoek gedaan naar bestaande throttling en rate limiting oplossingen. Uit dit onderzoek bleek de RateLimiterRequestHandlerAdvice van Spring Integration een eenvoudige toepasbare throttling oplossing te zijn welke aan de wensen voldoet. Voor rate limiting bleek de Bucket4J-spring-boot-starter library aan bijna alle requirements van eMagiz te voldoen.

Eén van de requirements waar de rate limiting library echter niet aan voldeed is het kunnen wijzigen van limieten zonder dat een nieuwe deployment nodig is. In overleg met de stakeholders van eMagiz en de schoolbegeleider is toen de keuze gemaakt deze functionaliteit te ontwerpen en realiseren, met het doel om een bijdrage aan de open-source library te leveren.

Het uiteindelijke resultaat van dit project is een werkend, op cache gebaseerd updatemechanisme, waarmee limieten dynamisch kunnen worden gewijzigd zonder dat een deployment nodig is. Bovendien is de oplossing toepasbaar in een schaalbare gedistribueerde omgeving, waarbij limieten en configuraties worden gesynchroniseerd tussen de verschillende machines. Ten slotte is doormiddel van een proof-of-concept aangetoond dat de oplossing werkt in combinatie met de Infinispan-componenten van de eMagiz runtime.

Op het moment van schrijven is het pull-request op GitHub geaccepteerd en gemerged door de eigenaar van de library. Zodra een nieuwe release wordt uitgebracht, zal de functionaliteit voor alle gebruikers toegankelijk zijn.

Figuren- en tabellenlijst

Inhoudsopgave

1 Inleiding

In de snel ontwikkelende wereld van digitale technologieën wordt het beheren van de stroom van gegevens en verzoeken naar API-servers steeds belangrijker voor het waarborgen van de stabiliteit, prestaties en beveiliging van applicaties. Dit is een probleem waar ook gebruikers van het eMagiz platform tegenaan lopen bij het realiseren van integraties tussen systemen.

EMagiz is een softwarebedrijf dat hun klanten een Enterprise Integration Platform as a Service (iPaaS) oplossing biedt. Op dit platform kunnen gebruikers in low-code integraties tussen diverse systemen ontwikkelen en beheren. Dit houdt in dat klanten zonder uitgebreide programmeerervaring toch integraties kunnen ontwikkelen om de verschillende systemen waar ze gebruik van maken aan elkaar te verbinden.

Hoewel het platform erg veel mogelijkheden biedt, ontbreekt momenteel een adequate traffic managementoplossing. Onder traffic management wordt bijvoorbeeld throttling, rate limiting, quota management en auto scaling verstaan. Wanneer integraties, al dan niet opzettelijk, te maken krijgen met zeer grote hoeveelheden gegevens of verzoeken, kan dit leiden tot performance problemen in de achterliggende systemen. In extreme gevallen kan dit zelfs leiden tot het omvallen van systemen, met potentieel verlies van gegevens tot gevolg. Naast de potentiële performance problemen, maakt het gebrek aan controle over de gegevensstroom het ook lastig voor klanten om de systemen goed te kunnen schalen.

De groeiende vraag vanuit klanten naar een oplossing voor dit probleem heeft geleid tot deze afstudeeropdracht. Het doel van de opdracht is om een prototype te realiseren waarmee gebruikers meer controle krijgen over het dataverkeer dat via hun integraties loopt. Daarbij wordt specifiek gekeken naar een generieke oplossing die toepasbaar is op de integratiepatronen “API-gateway” en “Event Streaming”.

2 Achtergrondinformatie

In dit hoofdstuk wordt ingegaan op eMagiz als bedrijf, het iPaaS platform en de achterliggende technieken. Ook wordt informatie gegeven over de verschillende integratiepatronen welke op het platform worden aangeboden.

2.1 Het bedrijf

De opdrachtgever voor het afstudeerproject is eMagiz Services b.v. te Enschede. EMagiz is ontstaan vanuit het droombeeld van een oplossing dat een bedrijfskundige in staat stelt zonder diepgaande technische kennis elke integratie te realiseren die vanuit de business noodzakelijk is (Vregelaar, 2023). Het bedrijf begon als onderdeel van zusterbedrijf CAPE groep, waar het product door consultants werd gebruikt om klanten te helpen in het digitaliseren van bedrijfsprocessen. In 2011 was het product dusdanig gegroeid dat besloten is om als zelfstandig bedrijf verder te gaan. Niet veel later werd het product voor het eerst als cloud-based service op de markt aangeboden. Ondanks de afsplitsing wordt nog steeds intensief samengewerkt tussen beide bedrijven en worden resources als personeel en het pand gedeeld.

In de loop van de jaren is het eMagiz iPaaS platform uitgegroeid tot een volwaardig product met een zeer uitgebreid pakket aan mogelijkheden. Naast het ontwikkelen van integraties, biedt het platform ook de mogelijkheid de integraties te testen, deployen en monitoren. Dit kan zowel in één van de cloud omgevingen als bij de klant op locatie. Voor het monitoren kan gedacht worden aan het inzichtelijk maken van statistieken over de verschillende klantomgevingen, het geven van alert-notificaties en het inzichtelijk maken van logging en errors etc.

Om het product te blijven verbeteren, nieuwe functionaliteiten te ontwikkelen en klanten te ondersteunen, heeft eMagiz +/- 30 werknemers in dienst. Hiervan zijn ongeveer 20 medewerkers werkzaam als software developer.

2.2 Het eMagiz platform

Om de opdracht goed te begrijpen is het van belang om te begrijpen wat de eMagiz iPaaS is en hoe deze werkt. In veel bedrijven wordt gebruik gemaakt van een groot aantal systemen waarbij het probleem ontstaat dat integraties tussen deze systemen nodig zijn. Het idee achter eMagiz was om dit te doen op basis van platformen en oplossingen die geen diepgaande technische (programmeer-) kennis en vaardigheden vragen. (*Het ontstaan van eMagiz Enterprise IPAAS*, 2022). Om dit mogelijk te maken is het eMagiz platform ontwikkeld.

2.2.1 Ontwikkelfasen

Het eMagiz platform is een volledige oplossing waarin gebruikers integraties kunnen ontwikkelen volgens het “Integration Life-cycle Management (ILM)” perspectief.

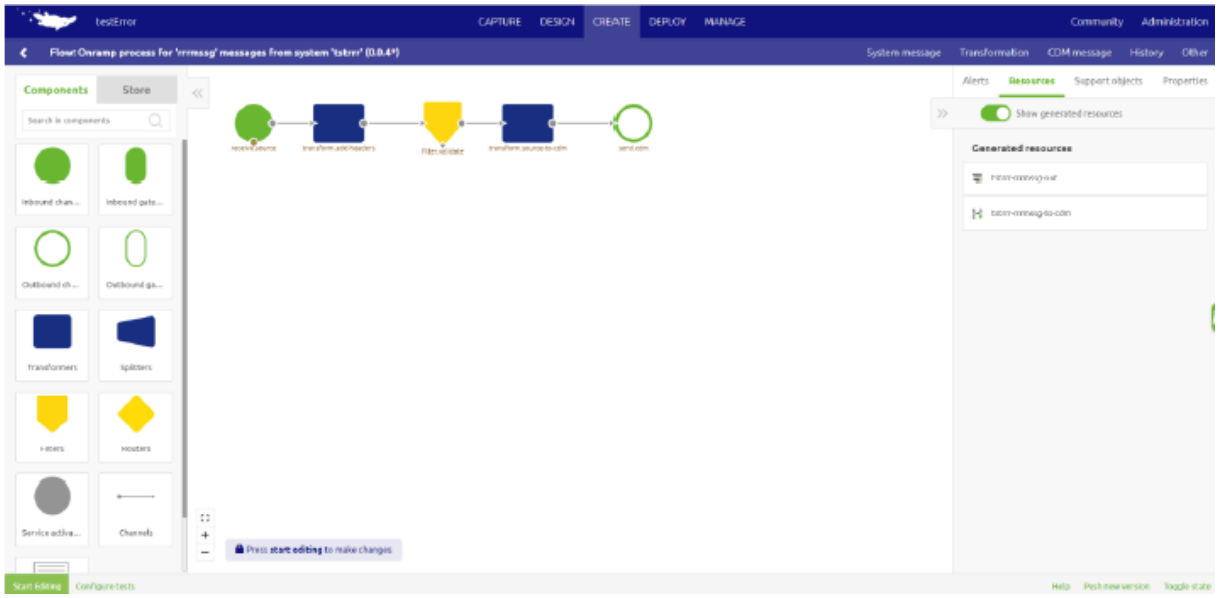
- **Capture**: het integratielandschap visueel in kaart brengen en vereisten en informatie voor integraties verzamelen en vastleggen;
- **Design**: een solution design maken, een keuze maken uit verschillende integratie patronen en een Canonical Data Model en deployment architectuur ontwerpen
- **Create**: realisatie van solution design & modelleren van integraties;
- **Deploy**: test, acceptatie en productie omgeving inrichten & integraties deployen, testen, accepteren en in productie nemen;
- **Manage**: monitoring van omgevingen, transacties, performance, centrale foutafwikkeling en notificaties naar beheerders;
- **Improve**: trend analyses uitvoeren vanuit statistieken en eenvoudig omgevingsconfiguraties aanpassen.

Figuur 1 eMagiz integratie levenscyclus

In de verschillende ontwikkelfasen (zie Figuur 1) kunnen gebruikers hun integratiebehoeften in kaart brengen, integraties ontwerpen, creëren, deployen en managen zonder diepgaande technische kennis.

2.2.2 De Techniek

Het eMagiz platform is ontwikkeld met behulp van Mendix, een low-code Application Development platform. Voor de logica in het platform zijn verschillende widgets ontwikkeld met React en Typescript en worden de Mendix low-code microflows aangevuld met op maat gemaakte Java-actions. Een van deze widgets is de eMagiz flow-designer (zie Figuur 2).



Figuur 2 eMagiz flow-designer

Wanneer de gebruiker in de “Capture” en “Design” fase zijn/haar integratiebehoeften heeft ingevoerd, maakt het platform automatisch integratieflows aan. Ieder van deze flows bestaat uit verschillende componenten welke gebaseerd zijn op het Spring integration framework. Indien gewenst kunnen deze flows met behulp van de flow-designer worden bewerkt. Als de flows naar wens zijn kunnen deze gedeployed worden en wordt ieder van de componenten vertaald naar Spring XML-format, zodat deze kunnen worden opgestart met de eMagiz runtime. De eMagiz runtime is een Java image, gemaakt met Gradle, Spring en verscheidene open-source libraries om het grote aantal opties in het platform te ondersteunen. Ook bevat de image door eMagiz ontwikkelde Spring Beans voor bijvoorbeeld de monitoring van de klantomgeving.

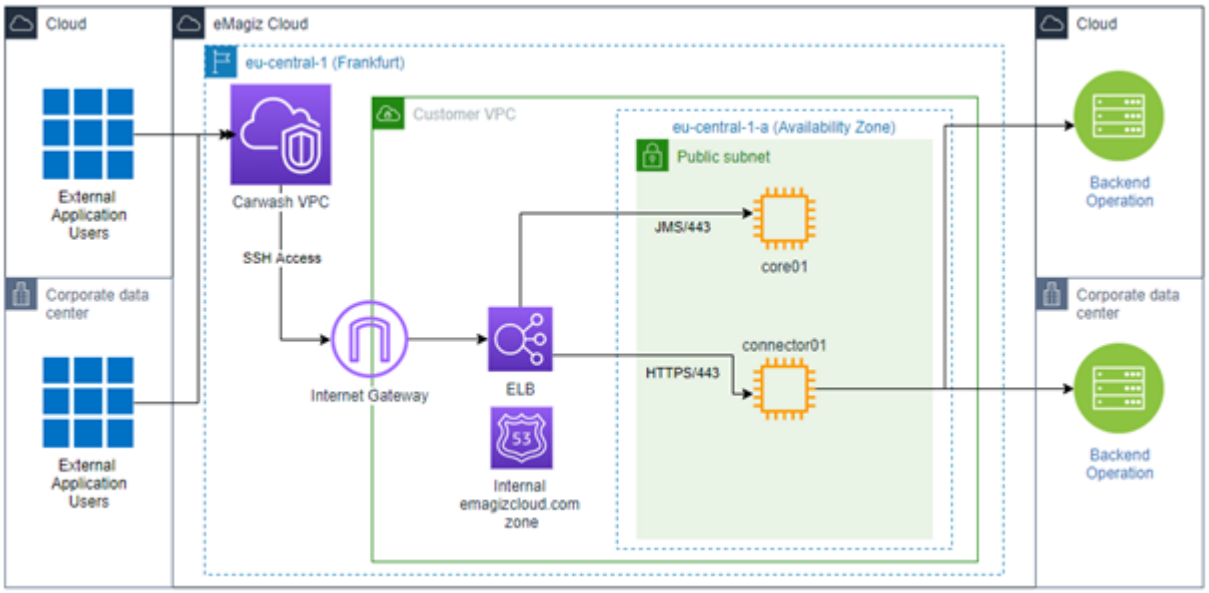
Voor de hosting van de klantomgevingen en systemen van eMagiz wordt vooral gebruik gemaakt van de AWS cloud. Standaard wordt ieder klant model op een eigen VPC gehost voor scheiding van verantwoordelijkheden tussen klanten en modellen. Voor de veiligheid verloopt al het verkeer naar deze VPC’s via een zogeheten carwash, welke de toegang tot de omgevingen beperkt. Indien gewenst hebben klanten echter ook de optie om (delen van) hun integraties on-premise te draaien. Enkele technieken die worden toegepast in de cloud zijn Kubernetes, Kafka topics en Elasticsearch.

2.2.3 Integratiepatronen

Om een zo passend mogelijke oplossing voor hun problemen te vinden hebben gebruikers de keuze uit drie verschillende integratiepatronen; Event Streaming, Messaging en API Gateway. Daarbij kunnen endpoints zich zowel on-premise als in de Cloud bevinden. Deze afstudeeropdracht beperkt zich tot de API-Gateway en Event Streaming integratiepatronen, messaging zal niet verder worden toelicht.

API-Gateway

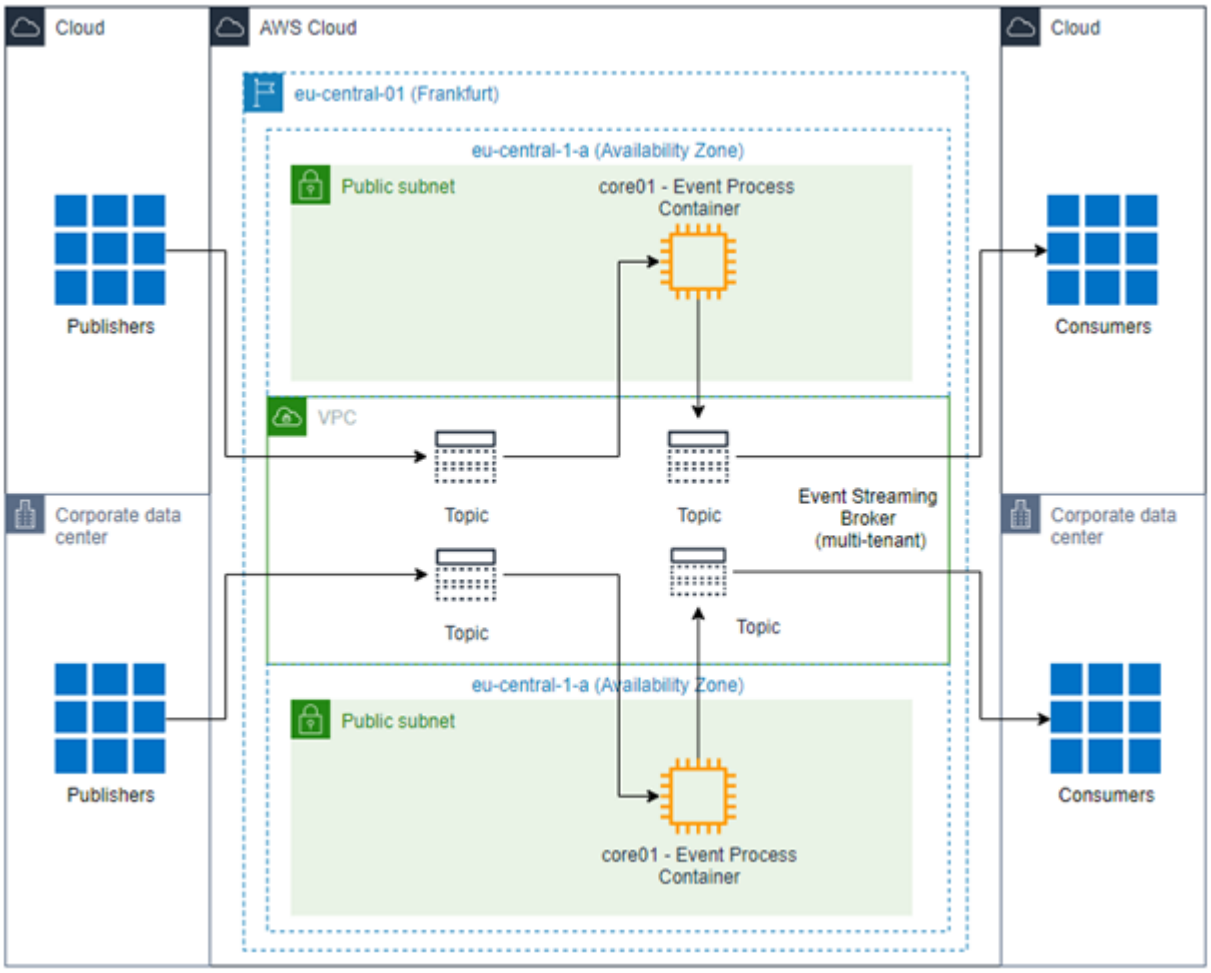
Het API-gateway integratiepatroon wordt voornamelijk gebruikt als tussenlaag tussen consumers en verschillende applicaties en/of microservices. Op die manier kan ongeacht het protocol toegang gekregen worden tot services, rekening houdend met o.a. security en versiebeheer. (Torken, z.d.). Dit integratiepatroon kan binnen eMagiz worden gebruikt als puur doorgeefluik tussen systemen (passthrough), maar het is ook mogelijk om data binnen de integratie te filteren, transformeren, splitten of aanvullende logica te implementeren. Bij het deployen van de integratie wordt deze in de cloud gehost op een aan de klant toegewezen AWS cloudslot, toegankelijk via de eMagiz cloud en voorzien van beveiliging, load balancing en verschillende andere features. Een vereenvoudigde weergave van deze architectuur is te zien in Figuur 3. De volledige cloud configuratie en hosting wordt door eMagiz afgehandeld, waardoor de gebruiker zich kan focussen op het ontwikkelen van de integraties. Meer informatie over het API-Gateway integratiepatroon is te vinden op https://emagiz.com/api-management/.



Figuur 3 API-gateway architectuur (vereenvoudigde weergave)
Noot. Overgenomen uit EMagiz API Gateway – eMagiz, door E. Bakker, 2022 (https://docs.emagiz.com/bin/view/Main/eMagiz%20Academy/Fundamentals/fundamental-api-gateway-introduction)

Event Streaming

Event Streaming is een integratiepatroon waarmee data gepubliceerd kan worden door een bron (publisher), waar andere applicaties zich op kunnen abonneren (subscriber). In tegenstelling tot de API-Gateway wordt de data bij dit patroon asynchroon verwerkt. EMagiz maakt dit mogelijk doormiddel van moderne frameworks, waaronder ActiveMQ, Artemis en Apache Kafka. (*API Management - Integration without Boundaries - eMagiz*, 2023b). Net als bij het API-gateway patroon kunnen gebruikers bij Event Streaming kiezen tussen een passthrough- of een transformation integratie. In het geval van een transformation wordt een aanvullende “Event Process Container” aangemaakt en gehost waarin de data transformaties plaats kunnen vinden. In Figuur 4 is een architectuur overzicht te zien van een dergelijke integratie. Wanneer voor passthrough wordt gekozen worden slechts de topics gehost en vervalt de architectuur in het lichtgroene gedeelte in het diagram. Meer informatie over het Event Streaming integratiepatroon is te vinden op https://emagiz.com/event-streaming/.



Figuur 4 Event Streaming architectuur (vereenvoudigde weergave)
Noot. Overgenomen uit EMagiz Event Streaming - eMagiz, door E. Bakker, 2023
(https://docs.emagiz.com/bin/view/Main/eMagiz%20Academy/Fundamentals/fundamental-event-streaming-introduction)

3 Projectopdracht

3.1 Aanleiding

Zoals in hoofdstuk 2.2 is besproken, ondersteunt het eMagiz platform verschillende soorten integratiepatronen. Afhankelijk van het soort integratie dat ontwikkeld wordt kan gekozen worden uit de Messaging-, API-Gateway- en Event Streaming integratiepatronen of een combinatie hiervan.

Zowel de API-gateway- als Event Streaming integratiepatronen hebben momenteel nog geen traffic management mogelijkheden, terwijl daar wel vraag naar is vanuit gebruikers. Vooral naar throttling en rate limiting is veel vraag vanuit klanten waardoor dit al lange tijd op de roadmap staat binnen het bedrijf. Deze toenemende vraag vanuit klanten en de wens van eMagiz om dit aan te kunnen bieden hebben geleid tot deze afstudeeropdracht.

3.2 Probleemanalyse

Het is voor klanten momenteel nog niet mogelijk om traffic management in te stellen voor API-gateway en Event Streaming integraties. Doordat hier nog geen oplossing voor is kan een foutieve integratie flow in potentie leiden tot een dusdanig aantal requests of hoeveelheid data, dat de machine het niet aan kan.

Traffic management is een erg breed begrip, waar bijvoorbeeld throttling, rate limiting, quota management en auto scaling onder vallen. Deze opdracht richt zich specifiek op de onderdelen throttling en rate limiting voor de API-gateway en Event Streaming integraties.

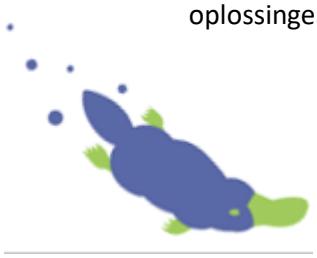
Throttling en rate limiting zijn vergelijkbare concepten, beide gericht op het beschermen van API-Services tegen overbelasting. Rate limiting richt zich vooral op het beperken van het aantal requests aan een API per tijdseenheid. (Srivastava, 2023). Daarbij wordt een limiet ingesteld voor het aantal verzoeken dat een gebruiker of applicatie binnen een specifieke tijdsperiode kan versturen. Wanneer de gebruiker van het endpoint over dit limiet heen komt worden verdere requests geblokkeerd.

Throttling richt zich voornamelijk op het uitspreiden van piekbelasting. Het verschil met rate limiting is dat bij throttling de toegang tot het endpoint niet gelijk wordt geblokkeerd wanneer veel requests binnenkomen. In plaats daarvan wordt de verwerking van de requests vertraagd zodra de limiet bereikt is (Holcombe, 2023). Om te voorkomen dat requests te lang in de wachtrij staan wordt een maximale timeout geconfigureerd. Als het throttling mechanisme merkt dat de vertraging door de wachtrij langer zal duren dan de maximale timeout, wordt gelijk een foutmelding teruggegeven.

Het probleem bij de API-gateway en Event Streaming integraties van eMagiz is dat gebruikers momenteel nog geen throttling of rate limiting kunnen instellen. Hierdoor kan het lastig zijn voor gebruikers om de omgevingen correct te schalen of quota's te managen. Daarnaast hebben gebruikers veel vrijheid in het ontwerpen van flows waardoor voor bijna ieder integratieprobleem een oplossing gemaakt kan worden. Deze vrijheid kan echter leiden tot een foutieve flow welke in potentie een dusdanig aantal requests verstuurt dat een machine omvalt, met alle gevolgen van dien.

De ideale situatie is een generieke oplossing waarmee zowel synchrone- als asynchrone calls gethrottled en/of rate limited kunnen worden, zodat het voor zowel de API-gateway als event streaming toegepast kan worden. De focus van het onderzoek ligt op het onderzoeken van de mogelijkheden voor een generieke oplossing en het realiseren van een prototype waarmee throttling en/of rate limiting mogelijk wordt.

Om een idee te geven van potentiële benaderingen, wordt hieronder voor beide traffic management oplossingen enkele voorbeelden gegeven.



Eén van de mogelijkheden voor throttling is generieke throttling, waarbij standaard instellingen op alle verzoeken wordt toegepast zonder onderscheid te maken in de afzender. Deze manier van throttling zou bijvoorbeeld gebruikt kunnen worden op klantomgevingen, waarbij gebruikers zelf limieten configureren. Een andere manier van throttling is op basis van IP-adres of API-key. Deze manier zou mogelijk toegepast kunnen worden in een proxy voor event-streaming, waarbij limieten per klant of omgeving ingesteld kunnen worden. Welke manier het best bij de casus past zal uit de requirements analyse en de onderzoeksfase moeten blijken.

Ook voor rate limiting zijn verschillende mogelijkheden. Het is bijvoorbeeld mogelijk om een limiet per klantomgeving, IP-adres of API-key in te stellen, maar dit kunnen ook algemene limieten zijn. Ook zijn er verschillende manieren van afhandeling van requests wanneer de limiet is bereikt. De eenvoudigste manier is het negeren van de requests of een foutcode teruggeven, wat potentieel kan leiden tot data verlies. Een alternatief is het implementeren van een queue-mechanisme, waarbij requests in een wachtrij gezet worden voor latere afhandeling. Ook zou bijvoorbeeld prioriteit aan andere requests kunnen worden gegeven. Uit de requirements analyse en onderzoeksfase zal blijken welke van deze manieren de voorkeur heeft of het best toepasbaar is binnen de opdracht.

3.3 Probleemstelling

In de huidige architectuur van de API-gateway en Event Streaming integratiepatronen mist de optie voor klanten om throttling en/of rate limiting te configureren. Door het gebrek aan deze traffic management oplossingen hebben sommige klanten moeite met het correct schalen van hun omgevingen, quota management en het beveiligen tegen omvallen van machines bij verkeerd geconfigureerde flows.

3.4 Doelstelling

Het doel van de onderzoeksfase is om tot een ontwerp te komen waarmee throttling en/of rate limiting geïmplementeerd kan worden voor beide integratiepatronen. Het uiteindelijke doel van deze afstudeeropdracht is het opleveren van een prototype waarmee throttling en/of rate limiting geïmplementeerd kan worden voor de API-gateway en Event Streaming integratiepatronen.

3.5 Hoofdvraag

De hoofdvraag voor het onderzoek is als volgt:

Hoe kan een generieke oplossing ontwikkeld worden waarmee throttling en/of rate limiting voor zowel API-gateway als Event Streaming geïmplementeerd kan worden?

3.6 Deelvragen

Om de hoofdvraag te kunnen beantwoorden zijn de volgende deelvragen opgesteld:

1. *Aan welke vereisten moet een throttling- en/of rate limiting oplossing voor API-gateway en Event Streaming voldoen?*
2. *Wat zijn vergelijkbare kenmerken of patronen tussen de synchrone API-gateway calls en asynchrone Event Streaming events die kunnen worden gebruikt om een generieke throttling-oplossing te ontwikkelen?*
3. *Welke bestaande, binnen het Spring ecosysteem passende, (open-source) frameworks, bibliotheken of technologieën kunnen helpen bij het ontwikkelen van een generieke throttling- en/of rate limiting oplossing?*
4. *Hoe kan een throttling- en/of rate limiting oplossing worden ontworpen voor zowel API-gateway calls als Event Streaming events, rekening houdend met de vereisten en vergelijkbare kenmerken?*

4 Onderzoeksaanpak

Om tot een antwoord op elk van de deelvragen te komen zijn verschillende onderzoeksmethoden toegepast. In dit hoofdstuk wordt per deelvraag toegelicht welke methoden zijn gebruikt, waarom voor deze methoden is gekozen en wat de resultaten waren.

4.1 Deelvraag 1: Aan welke vereisten moet een throttling- en/of rate limiting oplossing voor API-gateway en Event Streaming voldoen?

4.1.1 Methode

Om deze onderzoeksvraag te kunnen beantwoorden is een requirements analyse uitgevoerd. Bij deze analyse is allereerst een Stakeholder analyse uitgevoerd om de belangrijkste stakeholders in kaart te brengen. Stakeholders zijn alle partijen die interesse hebben in een op te leveren product of project. Het zijn mensen die het project kunnen beïnvloeden of door het project worden beïnvloed. (Van Lier, 2023). Vervolgens zijn de belangrijkste stakeholders uitgenodigd voor een persoonlijk interview waarin hun beeld van de opdracht en eisen aan het prototype in kaart gebracht werden. Door gebruik te maken van persoonlijke interviews kunnen eventuele onduidelijkheden gelijk opgehelderd worden doormiddel van vervolgvragen. Dit type interview is tijdsintensief, maar zeer geschikt voor het verkrijgen van kwalitatieve informatie.

Om zowel brede als diepgaande informatie te verkrijgen is gekozen voor gestructureerde interviews met een combinatie van generieke en specifieke vragen. De generieke vragen kunnen helpen om nieuwe inzichten te verkrijgen waar vooraf niet aan gedacht is, terwijl de specifieke vragen meer diepgaande en technische informatie kunnen geven.

4.1.2 Resultaten

De volledige requirements analyse, inclusief de uitwerking van de interviews, is te lezen in bijgevoegde document "Requirements Analyse.pdf".

Allereerst is begonnen met het in kaart brengen van de belanghebbenden van deze casus. Hieruit is de volgende lijst naar voren gekomen:

Vastgestelde belanghebbenden:

1. Tech Lead (Bas Elzinga)
2. Product Manager & Product Owner (Mark De la Court)
3. Manager Expert Service (Samet Kaya)
4. Development Team
5. Eindgebruikers

Vervolgens is van elk van deze stakeholders bepaald hoeveel belang en invloed zij hebben op het uiteindelijke resultaat. De resultaten hiervan zijn weergegeven in een power-influence matrix, welke te zien is in Figuur 5 Power-influence matrix. In deze matrix wordt op de horizontale as weergegeven hoeveel belang een stakeholder heeft bij het uiteindelijke prototype. De verticale as geeft aan hoeveel invloed de stakeholder heeft op het prototype. Hoe verder naar de rechterbovenhoek een stakeholder geplaatst is, hoe belangrijker het is om de stakeholder geïnformeerd te houden en te betrekken bij het onderzoek en te maken keuzes. Uit deze matrix is af te lezen dat de belanghebbenden 1, 2 en 3 het belangrijkste zijn voor dit project. Doordat deze belanghebbenden elk een andere achtergrond hebben kunnen ze ook inzichten geven vanuit verschillende perspectieven. Dit maakt dat dit een ideale selectie was voor de interviews.

Figuur 5 Power-influence matrix

Vervolgens zijn de één-op-één interviews gehouden met de geselecteerde stakeholders. Op basis van de antwoorden tijdens de interviews is een lijst met requirements opgesteld. Deze lijst is vervolgens besproken in een gezamenlijke meeting met alle deelnemers waar zij feedback konden geven. Ook is tijdens deze meeting de prioritering is bepaald volgens de Moscow-methodiek (Must, Could, Should, Won't). Ten slotte zijn de requirements onderverdeeld in Functionele (F) en non-functionele (NF) requirements. Na het verwerken van de feedback resulteerde de volgende lijst met requirements, welke tevens antwoord op de deelvraag geeft:

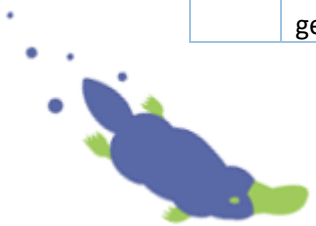
Functionele requirements:

Nr.	Requirement	MoSCoW
F01	Het systeem heeft de mogelijkheid om rate limits in te stellen voor specifieke API-endpoints.	Could
F02	Het systeem heeft de mogelijkheid om rate limits in te stellen op basis van de combinatie van een endpoint en een specifieke gebruiker.	Should
F03	Het systeem moet de mogelijkheid bieden om rate limits in te stellen op basis van de gebruikersrol per endpoint, waarbij limieten per gebruiker worden bijgehouden.	Must
F04	De rate limits moeten configureerbaar zijn op basis van verschillende tijdseenheden, waaronder seconden, minuten, uren.	Must
F05	De rate limits kunnen geconfigureerd worden op basis van verschillende tijdseenheden, waaronder dagen, weken, maanden.	Could
F06	De rate limits moeten configureerbaar zijn op basis van aantallen berichten per tijdseenheid.	Must
F07	De rate limits kunnen geconfigureerd worden op basis van hoeveelheid bytes per tijdseenheid.	Could
F08	Het systeem heeft de mogelijkheid om een limiet op de grootte van berichten in te stellen.	Won't
F09	Wanneer rate limits overschreden worden moeten verdere berichten worden afgewezen met een foutcode waarmee voor de afzender duidelijk wordt dat het limiet is overschreden.	Must
F10	Het systeem heeft de mogelijkheid om de snelheid waarmee berichten naar backend systemen gestuurd worden te limiteren. (Throttling)	Should
F11	Het systeem heeft de mogelijkheid om te configureren hoeveel berichten maximaal per tijdseenheid naar een backend systeem verstuurd mogen worden.	Should

F12	Bij het overschrijden van een limiet voor uitgaande berichten worden verdere berichten in een first-in-first-out wachtrij geplaatst, zodat deze in het volgende tijdsframe afgehandeld kunnen worden.	Should
F13	Het systeem heeft de mogelijkheid om het maximale aantal berichten te configureren dat in de wachtrij kan worden geplaatst voor verzending naar het backend systeem.	Should
F14	Het systeem zou een foutcode terug moeten geven voor nieuwe berichten wanneer het maximale aantal berichten in de wachtrij bereikt is.	Should
F15	Het systeem heeft de mogelijkheid om te configureren hoe lang bericht gethrottled mogen worden voordat een timeout gegeven wordt.	Could
F16	Het systeem stemt rate limits op binnenkomende berichten automatisch af op de limieten voor uitgaande berichten.	Could
F17	Het systeem heeft de mogelijkheid om een limiet van een gebruiker dynamisch te wijzigen zonder dat daar een nieuwe deployment voor nodig is, door een andere, al bestaande, rol aan de gebruiker te koppelen.	Should
F18	Het systeem heeft de mogelijkheid om een limiet van een gebruiker dynamisch te wijzigen zonder dat daar een nieuwe deployment voor nodig is, zonder dat de rollen van de gebruiker aangepast hoeven te worden.	Could
F19	Rate limits met een tijdsframe van meer dan een uur kunnen persistent worden bijgehouden, zodat een herstart van een omgeving geen invloed heeft op de limieten.	Should
F20	De oplossing kan actuele limieten delen tussen alle rate-limiting machines in zowel een double-lane omgeving als een volledig dynamisch schaalbare omgeving. (Implementatie)	Could
F21	Het systeem moet toepasbaar zijn in een multi-threaded omgeving.	Must
F22	Het systeem heeft de mogelijkheid om inzicht te krijgen in de actuele limieten, zodat monitoring mogelijk is.	Should
F23	Het systeem heeft de mogelijkheid om waarschuwingen in te stellen wanneer een limiet overschreden wordt.	Could
F24	De rate limits zijn toepasbaar op zowel de API-gateway als Event Streaming integratie patronen.	Could

Non-functionele requirements

Nr.	Requirement	MoSCoW
NF01	Het systeem zou horizontaal schaalbaar moeten zijn, zodat kan worden omgaan met een groeiend aantal gebruikers en verkeer in een dynamisch schaalbare gedistribueerde omgeving, zonder dat dit invloed heeft op het functionele gedrag van de rate limiter.	Should
NF02	De rate limiting implementatie mag maximaal 100 milliseconden impact hebben op de verwerking van berichten.	Must
NF03	Het systeem mag geen belemmeringen opleveren voor het implementeren van quota checks in de toekomst.	Must
NF04	De oplossing moet op een dusdanige manier ontworpen zijn dat het geen belemmeringen vormt om in de toekomst rate limits te kunnen delen tussen verschillende containers in zowel een double-lane omgeving als een dynamisch schaalbare omgeving. (Ontwerp)	Must
NF05	De oplossing moet toepasbaar zijn in een multi-tenancy omgeving, waarbij gebruikers alleen toegang hebben tot gegevens waar zij rechten voor hebben.	Must
NF06	De interface voor externe systemen moet hetzelfde blijven, zodat de impact voor gebruikers minimaal blijft.	Must



4.2 Deelvraag 2: Wat zijn vergelijkbare kenmerken of patronen tussen de synchrone API-gateway calls en asynchrone Event Streaming events die kunnen worden gebruikt om een generieke throttling-oplossing te ontwikkelen?

Het doel van deze onderzoeksvraag is om een duidelijk beeld te krijgen van de overeenkomsten tussen de API-gateway en Event Streaming integratiepatronen. Dit is nodig om te kunnen bepalen of een generieke oplossing mogelijk is, of dat twee afzonderlijke oplossingen wenselijker zijn. Daarnaast geeft dit onderzoek meer inzicht in de integratiepatronen, wat zal bijdragen bij het opstellen van een ontwerp.

4.2.1 Methode

Om antwoord op deze onderzoeksvraag te vinden is voornamelijk gebruik gemaakt van literatuuronderzoek. EMagiz heeft een uitgebreid, goed onderhouden, online documentatie portaal waar veel informatie te vinden is over het eMagiz platform en de verschillende integratiepatronen. Deze informatiebron is gekozen omdat het actief onderhouden wordt en een goede samenvatting geeft van de werking van de integratiepatronen. Waar nodig is aanvullende informatie verkregen door deskundigen, zoals ontwikkelaars van het platform en de backend, te raadplegen en code te bestuderen.

Het onderzoek is begonnen met het diepgaand bestuderen van de onderliggende architectuur, technieken, security en datastructuur van beide integratiepatronen. Vervolgens is op basis van de analyses bepaald waar eventuele overeenkomsten zitten en of daar een generieke oplossing op toegepast kan worden. Het resultaat van deze vergelijking is het antwoord op de deelvraag.

4.2.2 Resultaten

De volledige systeemanalyse is te lezen in bijgevoegde document "Systeem Analyse.pdf".

API-Gateway

Waar organisaties vroeger gebruik maakten van monolithische systemen, bewegen ze met hun applicaties steeds meer richting de cloud. Deze beweging heeft ertoe geleid dat systemen zijn opgesplitst in kleine, schaalbare en gedistribueerde componenten, welke elk hun eigen taak hebben. In een landschap vol microservices worden API-gateways vaak gebruikt als tussenlaag tussen verschillende applicaties en consumers (Oltwater, 2023).

Met het API-gateway integratiepatroon van eMagiz kunnen klanten REST-endpoints opzetten om verschillende systemen aan elkaar te kunnen koppelen. Bij deze integratie bestaat de keuze om het request gelijk door te zetten naar een back-end systeem (passthrough), of de data in de flow te transformeren voordat het request doorgestuurd wordt (transformation). In beide situaties wordt het verkeer via een zo geheten carwash gerouteerd naar de juiste klantomgeving, waar het met een load balancer naar de juiste container wordt doorgezet. In het geval van een double lane omgeving (waar alle integratiecomponenten dubbel draaien) zal de load balancer de HTTP-requests evenredig proberen te verdelen over beide connector containers. Voor de beveiliging ondersteunt dit integratiepatroon verschillende manieren van authenticatie en autorisatie, waaronder API-key en (externe) identity providers zoals OAuth 2.0. De afhandeling van deze authenticatie en autorisatie checks vinden in de klantomgeving plaats met behulp van Spring Security.

Event Streaming
Event Streaming is een integratiepatroon waarmee op asynchrone wijze data wordt verwerkt. Bij dit integratiepatroon wordt data door een bron gepubliceerd (publisher/producer), waarop andere applicaties zich kunnen abonneren (subscriber/consumer) (*What is event streaming? A deep dive*, z.d.). In het eMagiz platform wordt dit patroon aangeboden doormiddel van Kafka topics. Dit zijn kanalen waarin data kan worden gepubliceerd en geconsumeerd binnen het Apache Kafka-streamingplatform. Voor het beheer van deze topics wordt gebruik gemaakt van Aiven, een externe Event Streaming Broker service.

Net als bij API-gateway hebben gebruikers de keuze tussen passthrough en transformation integraties. Doordat externe klantsystemen rechtstreeks met de broker communiceren is het mogelijk dat gebruikers een passthrough integratie maken waarbij events volledig buiten de eMagiz cloud blijven. In het geval van een transformation integratie worden de events wel binnen de eMagiz infrastructuur verwerkt. De transformatie vindt dan plaats in een event processing container op de klantomgeving in de AWS-cloud.

De beveiliging van het verkeer van- en naar de topics is opgezet met 2-way SSL/TLS, waarbij de voorkeur ligt op gebruik van keystore en truststore bestanden. De keystore wordt gebruikt om de systemen te authentiseren bij de Event Streaming Broker, waarna met behulp van een access control list wordt gecontroleerd of het systeem geautoriseerd is. De events zelf zijn meestal in JSON format, maar ook XML en EDI is mogelijk. Qua inhoud zijn de berichten volledig afhankelijk van de producers en de klantconfiguratie in het portaal.

Conclusie
Bij deze analyse is gekeken naar zowel de API-gateway als Event Streaming integratiepatronen om te bepalen of er overeenkomsten zijn waarop een generieke throttling en/of rate limiting oplossing toegepast kan worden. Uit deze analyse is gebleken dat de beide integratiepatronen erg verschillend zijn en slechts weinig overeenkomsten hebben.

Hoewel beide patronen de keuze tussen passthrough- en transformation integraties bieden, is de opzet van deze patronen compleet verschillend. Ook de beveiliging wordt op verschillende manieren afgehandeld. In het geval van API-Gateway worden endpoints beveiligd met Spring Security op basis van een API-key, OAuth 2.0 of OpenID connect, terwijl dit voor Event Streaming volledig door de externe broker service wordt afgehandeld met keystore en truststore bestanden. Wel zitten er overeenkomsten tussen de ondersteunde dataformaten. Dit is voor beide patronen standaard in JSON, maar ook XML-format wordt ondersteunt. Slechts Event Streaming heeft daarnaast ook nog EDI mogelijkheden. Heel globaal gezien kan gesteld worden dat de grootste overeenkomst is dat zowel API-requests als events geteld kunnen worden.

Het antwoord op deze deelvraag is dan ook dat er slechts een heel beperkt aantal overeenkomsten tussen de API-Gateway en Event Streaming integratiepatronen zijn. Door de grote verschillen tussen deze systemen kan echter gesteld worden dat geen van deze overeenkomsten kan worden gebruikt om tot een generieke throttling en/of rate limiting oplossing te komen.

Rekening houdend met de requirement dat de interface voor gebruikers niet gewijzigd mag worden en eMagiz in een overgangstraject zit naar een andere Event Streaming broker, is de aanbeveling om de focus te leggen op een oplossing voor slechts de API-gateway.

4.3 Deelvraag 3: Welke bestaande, binnen het Spring ecosysteem passende, (open-source) frameworks, bibliotheken of technologieën kunnen helpen bij het ontwikkelen van een generieke throttling- en/of rate limiting oplossing?

Het doel van deze deelvraag is om te onderzoeken of er bestaande, goed onderhouden oplossingen zijn welke de ontwikkeltijd van het prototype kunnen verkorten. Deze werkwijze is in lijn met de werkwijze binnen eMagiz, waar veel gebruik gemaakt wordt van goed onderhouden open-source projecten.

4.3.1 Methode

Om antwoord te kunnen geven op deze deelvraag is gebruik gemaakt van literatuuronderzoek. Het onderzoek is daarbij opgedeeld in Throttling en Rate limiting oplossingen. Op basis van het resultaat uit deelvraag twee is daarnaast besloten ook te onderzoeken of er proxymogelijkheden zijn voor de Event Streaming integraties waarmee een generieke oplossing toch mogelijk wordt.

Dit onderzoek is uitgevoerd volgens de desk research methodiek van het DOT-framework (Vogel, z.d.). Dit houdt in dat gebruikt is gemaakt van online zoekmachines als Google om bestaande oplossingen in kaart te brengen. Om de resultaten te beperken is voorafgaand aan ieder deelonderzoek een lijst met criteria opgesteld waar de oplossing aan moet voldoen. Oplossingen welke niet aan cruciale criteria voldeden zijn bij de voorselectie gelijk afgevallen. De resterende resultaten zijn diepgaand geanalyseerd om meer informatie te verkrijgen en te kunnen bepalen of ze voldoen aan de overige criteria. De intentie was om vervolgens de beste match te bepalen op basis van een multi-criteria analyse. Door het geringe aantal matches bleek dit echter niet nodig en kon eenvoudig een conclusie worden getrokken.

4.3.2 Resultaten

De volledige resultaten van dit onderzoek zijn te lezen in bijgevoegde document "Onderzoek bestaande libraries.pdf".

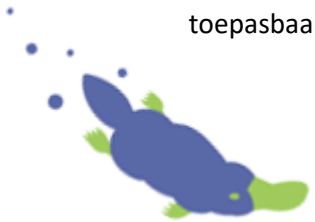
Kafka proxy

Het onderzoek is begonnen met onderzoek naar Kafka proxies. Het idee achter dit onderzoek is dat de rate limiting of throttling oplossing mogelijk aan een proxy toegevoegd kan worden om toch tot een generieke oplossing te komen. Daarbij is specifiek gekeken naar een proxy welke in combinatie met Aiven Kafka gebruikt kan worden en 2-way SSL/TLS beveiliging ondersteunt. Door deze voorselectie vielen opties als Confluent en Solace af omdat dit alternatieven op Aiven zijn en niet bedoeld zijn om samen te gebruiken.

Na de voorselectie bleef de volgende lijst over:

- Grepplabs – Kafka-proxy
- Dajudge – kafkaproxy
- Aiven – Kafka REST proxy
- Aklivity – Zilla
- Conductor Gateway

Deze overgebleven libraries/producten zijn vervolgens verder bestudeerd, waaruit bleek dat slechts Conductor Gateway mogelijkheden biedt voor het toevoegen van eigengemaakte uitbreidingen. Dit is een cruciale criteria om de proxy te kunnen gebruiken voor een generieke throttling en/of rate limiting oplossing, waardoor de overige oplossingen gelijk afvielen. Conductor Gateway biedt deze optie echter alleen voor Enterprise abonnementen, welke extra kosten met zich meebrengen. Door de aankomende migratie naar een andere Event Streaming provider, waarop Conductor Gateway niet toepasbaar is, wegen deze extra kosten niet op tegen de voordelen.



Op basis van dit onderzoek kan worden gesteld dat er geen bestaande oplossingen zijn die aan de requirements voldoen en het mogelijk maken om een generiek toepasbaar prototype te ontwikkelen. Om deze reden kan de focus voor een throttling en/of rate limiting oplossing verschoven worden naar API-gateway in plaats van een generiek toepasbare oplossing.

Throttling
Uit de requirements analyse bleek dat de wens om throttling aan te bieden voortkomt uit de wens van klanten om hun backend systemen te kunnen beschermen. Het is daarbij minder belangrijk van wie de requests afkomstig zijn, maar wel belangrijk dat een outbound flow gelimiteerd kan worden in het aantal berichten dat doorgelaten wordt. Omdat het synchrone HTTP-requests betreft is het daarnaast belangrijk dat een maximale timeout geconfigureerd kan worden. Met deze informatie in gedachten is vervolgens gezocht naar bestaande oplossingen.

Omdat het aanbod aan throttling oplossingen erg groot is, is bij de voorselectie alleen gekeken naar Spring compatible oplossingen. Het resultaat van dit onderzoek is de volgende lijst:

- Weddini – Spring boot throttler
- Resilience4j
- Bucket4j
- Spring Integration – rate-limiter-advice

Bij het verder bestuderen van deze resultaten viel Weddini af omdat deze niet langer onderhouden wordt. Bucket4j en Resilience4j bleken daarentegen beide geschikt te zijn om throttling te implementeren. Bucket4j is generiek opgezet en ondersteunt verschillende opties zoals distributed caching, terwijl Resilience4j voornamelijk gericht is op het throttlen van method calls. De Spring Integration RateLimiterRequestHandlerAdvice is volledig gebaseerd op Resilience4j en is daarmee ook een geschikte oplossing. Het voordeel van dit component is dat deze eenvoudig te configureren is doormiddel van Spring XML flows. Ook kan deze gemakkelijk aan Spring Integration HTTP Outbound componenten en Service Activators gekoppeld worden doormiddel van een zo gehete adviceChain. Dit maakt het eenvoudig te implementeren in de huidige eMagiz integraties.

Aangezien uit de requirements analyse bleek dat de wensen voor throttling voornamelijk gericht zijn op het beperken van het aantal uitgaande berichten op flow basis, lijkt de Spring Integration RateLimiterAdvice de beste optie. Deze optie vergt weinig configuratie en kan eenvoudig aan een Outbound component gekoppeld worden. Het delen van limieten tussen verschillende systemen is niet mogelijk met deze oplossing, maar voor de geschetste throttling use-case zou dat geen probleem moeten zijn.

Rate limiting
Rate limiting kan op veel verschillende manieren. Zo kan bijvoorbeeld gekozen worden voor het Token bucket algoritme, Leaky Bucket algoritme of sliding window algoritme. Ook qua architectuur zijn er veel mogelijkheden. Zo kan bijvoorbeeld gekozen worden voor een cloud based oplossing, een gateway oplossing of het implementeren in de integratieflow zelf. De keuzes die hierin gemaakt worden zijn vooral afhankelijk van het doel van de oplossing en de bijbehorende requirements.

Bij dit onderzoek is gekozen om breed te oriënteren en oplossingen voor elk van de drie hiervoor genoemde plekken in de architectuur in kaart te brengen. Uit dit onderzoek bleek dat er zeer veel aanbod is aan mogelijkheden, elk met hun eigen voor- en nadelen.

De gevonden cloud gebaseerde oplossingen zijn vooral gericht op het tegengaan van DDOS aanvallen, wat op zichzelf een belangrijke beveiliging is, maar niet overeenkomt met de requirements voor deze casus. Ook gaat dit op basis van IP-adres, terwijl eMagiz op basis van gebruikers wil kunnen limiteren.

De volgende optie waar naar is gekeken is het implementeren van een gateway. Met een gateway kan een centrale ingang voor alle HTTP-requests worden gemaakt waar requests gefilterd worden voordat deze doorgezet worden naar de juiste integratie flow. Door rate limiting in een dergelijke gateway te implementeren kan de oplossing eenvoudig op- en afgeschaald worden, zonder dat de achterliggende integratie flows mee geschaald hoeven te worden. Wel is het belangrijk dat authenticatie en autorisatie in deze gateway geïmplementeerd kan worden, zodat de rate limiting op basis van gebruikersaccounts of rollen geconfigureerd kan worden. Hoewel er veel gateway oplossingen bestaan, zijn er slechts enkele die gratis zijn, rate limiting ondersteunen en actief onderhouden worden. Twee gateways welke wel aan deze requirements voldoen zijn Tyk-gateway en Spring Cloud Gateway. Tyk-Gateway is echter minder dynamisch toe te passen dan de Spring Cloud Gateway. Zo kan de rate limiting bijvoorbeeld niet los gebruikt worden, waardoor het niet toepasbaar is op andere integratiepatronen. Vooral het feit dat Spring Cloud Gateway custom ratelimiters ondersteunt en eenvoudig te integreren is met Spring Security maakt dit een mogelijke oplossing. Van zichzelf heeft het echter geen rate limiting opties.

Ten slotte is gekeken naar libraries welke binnen de integratieflows toegepast kunnen worden. Bij dit onderzoek bleek dat de Bucket4j-spring-boot-starter library voldoet aan praktisch alle requirements. Zo kunnen filters in de application.properties geconfigureerd worden per URL en biedt de library ondersteuning voor Spring Expression Language (SPEL-Expressies). Dit maakt het mogelijk om limieten per gebruiker, per rol, per endpoint, etc. te configureren. Ook heeft het de optie om metrics te exporteren en ondersteunt het een zeer divers aantal caches, waaronder het reeds door eMagiz gebruikte Infinispan. Ten slotte is het ook mogelijk om deze library in combinatie met de Spring Cloud Gateway te gebruiken. Stel dat eMagiz de keuze maakt om de Spring Security checks naar een gateway container te verplaatsen, dan kan de rate limiting dus eenvoudig mee gemigreerd worden. Het enige nadeel is dat limieten niet tijdens runtime gewijzigd kunnen worden, terwijl dit wel als requirement uit de requirements analyse naar voren kwam. Aangezien dit een requirement met lagere prioriteit was weegt dit niet op tegen alle voordelen.

Conclusie
Uit het onderzoek is gebleken dat er een zeer divers aanbod is aan frameworks, bibliotheken of technologieën waarmee throttling en/of rate limiting geïmplementeerd kan worden. Door de opzet van Event Streaming binnen eMagiz en het gebrek aan geschikte proxy mogelijkheden voor Aiven Kafka is echter geen van deze oplossingen generiek toepasbaar op beide integratiepatronen.

Aangezien bij de requirements analyse was vastgesteld dat API-gateway de voorkeur had indien een generieke oplossing niet mogelijk was, wordt het antwoord dan ook daarop bijgesteld.

Voor het implementeren van Throttling voor API-gateway integraties kan gebruik gemaakt worden van de Spring Integration RateLimiterRequestHandlerAdvice. Dit component biedt de mogelijkheid om per flow te kunnen throttlen, is gebaseerd op de open-source Resilience4J library en is onderdeel van het Spring ecosysteem. Ook is het erg eenvoudig te implementeren in de huidige eMagiz flows, waarmee het een zo goed als kant- en klare throttling oplossing is.

Voor het implementeren van Rate Limiting is de Bucket4J-spring-boot-starter library de beste match. Deze library voldoet aan praktisch alle rate-limiting requirements en biedt zelfs enkel functionaliteiten welke voor de casus in eerste instantie als optionele requirements gezien werden. De library is open-source en is specifiek gemaakt voor het Spring ecosysteem, waarmee het een bijna kant- en klare oplossing is voor het ontwikkelen van een rate-limiting oplossing. Het enige nadeel is dat het geen ondersteuning biedt voor het dynamisch aanpassen van rate limits.

4.4 Deelvraag 4: Hoe kan een throttling- en/of rate limiting oplossing worden ontworpen voor zowel API-gateway calls als Event Streaming events, rekening houdend met de vereisten en vergelijkbare kenmerken?

Het oorspronkelijke doel van deze deelvraag was om tot een ontwerp te komen waarmee een prototype ontwikkeld kan worden dat throttling en/of rate limiting voor beide integratiepatronen mogelijk maakt. Gebaseerd op de resultaten van de eerdere deelvragen moest dit doel echter bijgesteld worden. Door de grote verschillen in de beide integratiepatronen is een generieke oplossing niet langer het doel. In plaats daarvan ligt de focus op de API-gateway integraties. Vervolgens bleek uit het onderzoek van deelvraag 3 dat voor zowel throttling als rate-limiting kant- en klare oplossingen bestaan, welke samen vrijwel alle requirements oplossen.

Voor eMagiz waren deze resultaten erg mooi. Voor het afstudeeronderzoek betekent dit echter dat de HBO-competenties ontwerpen en realiseren op een andere manier aangetoond moeten worden. In overleg met de belangrijkste stakeholders binnen eMagiz is toen besloten om de open-source Bucket4J-spring-boot-starter library te wijzigen zodat limieten dynamisch aangepast kunnen worden. Het doel is daarbij om de wijzigingen doormiddel van een pull-request als bijdrage aan de library te laten accepteren. Voor het geval de pull-request niet geaccepteerd wordt is het noodzakelijk dat de aanpassingen zo dicht mogelijk bij de huidige werking blijven, zodat toekomstige updates van de library eenvoudiger overgenomen kunnen worden.

Deelvraag 4 is hierdoor gewijzigd in: *Hoe kan de Bucket4J-Spring-boot-starter library ontworpen worden zodat rate limits zonder herstart van de applicatie aanpasbaar zijn, met minimale impact op de bestaande codebase?*

4.4.1 Methode

Omdat de scope van de opdracht is gewijzigd is een aanvullend literatuuronderzoek uitgevoerd. Bij dit onderzoek is de documentatie van de Bucket4J-spring-boot-starter uitgebreid bestudeerd. Daarnaast is de documentatie van Bucket4J, de library waarop Bucket4J-spring-boot-starter is gebaseerd, bestudeerd om inzicht te krijgen in de werking en mogelijkheden die deze biedt.

Vervolgens is een class- en code analyse uitgevoerd van de Bucket4J-Spring-boot-starter library. Deze analyse was erop gericht om de huidige structuur van de library in kaart te brengen en te bepalen welke mogelijkheden en obstakels er zijn voor het dynamisch kunnen wijzigen van limieten.

Op basis van de bevindingen van de analyse is vervolgens een ontwerp opgesteld. Tijdens het ontwerpen is gebruik gemaakt van kleine proof-of-concepts (POC's). Met behulp van deze POC's konden eventuele twijfels worden weggenomen over de werking van de library. Ook kon hiermee worden gevalideerd of onderdelen van het opgestelde ontwerp werkten zoals verwacht. Daarnaast zijn ontwerpkeuzes waar nodig afgestemd met de bedrijfsbegeleider.

4.4.2 Resultaten

De volledige resultaten van de analyse, het ontwerp en overwogen alternatieven zijn te lezen in bijgevoegd document "Systeem ontwerp.pdf".

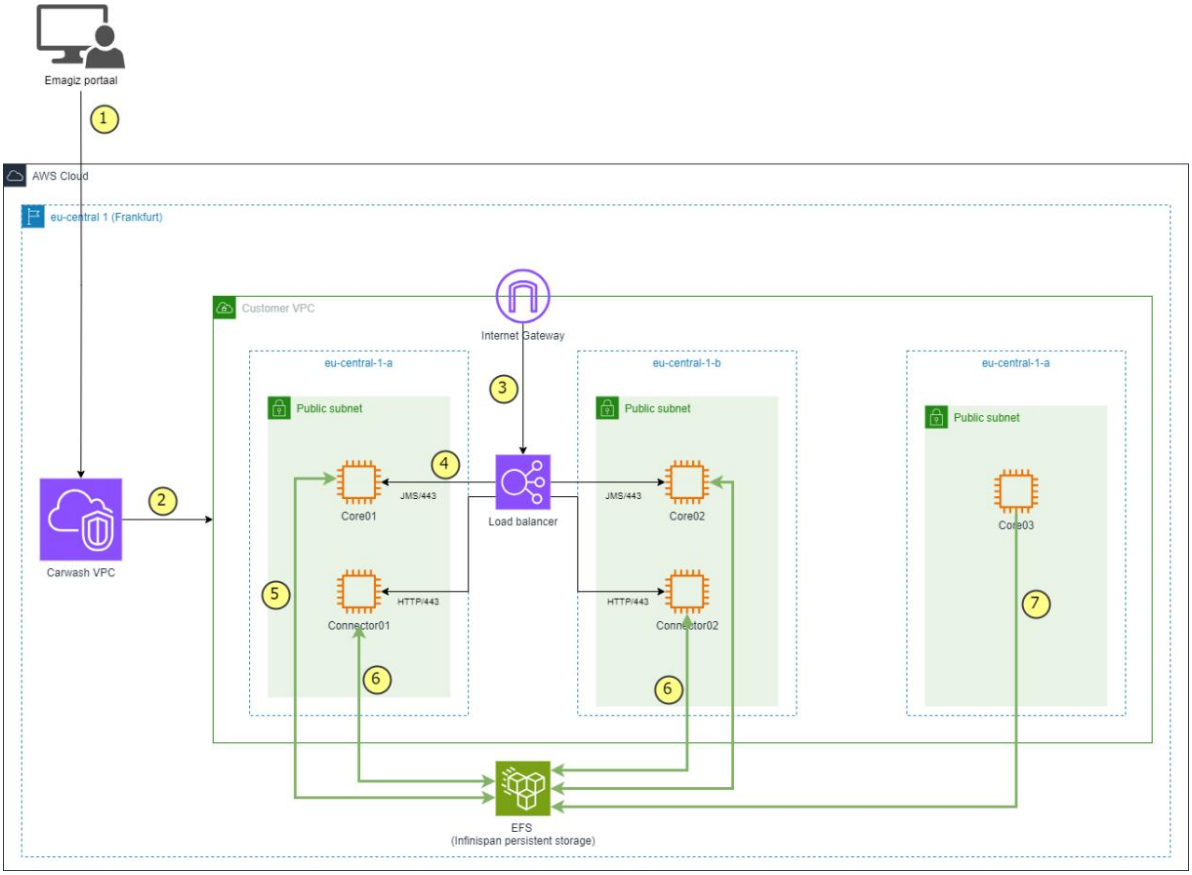
Architectuur

Voordat naar de daadwerkelijke code is gekeken, is nagedacht over de architectuur. Om limieten dynamisch aan te kunnen passen is het namelijk noodzakelijk dat het eMagiz portaal in staat is om gewijzigde configuraties naar een in de AWS-cloud gedeployde integratieflow te sturen. Daarnaast moest een oplossing bedacht worden voor het synchroniseren van limieten en configuraties tussen containers en het persisteren van data. Zonder een oplossing voor dit obstakel zal het dynamisch kunnen wijzigen überhaupt niet mogelijk zijn voor eMagiz en is het wijzigen van de library niet nodig.

Zoals in hoofdstuk 2.2.3 is besproken en in diepgang is te lezen in het document "Systeem analyse.pdf", loopt al het verkeer naar de integratie containers via de Carwash van eMagiz. In het geval van een gen3 double-lane omgeving, waar alle containers dubbel draaien, zal altijd slechts één core-container actief zijn. De andere core-container wordt in "slaapstand" gehouden zolang de eerste actief is. Al het JMS-verkeer wordt door de carwash naar deze actieve container gerouteerd terwijl HTTP-verkeer door een load balancer verdeeld wordt over beide connector containers. Doordat slechts één container JMS verkeer ontvangt en niet vast te stellen is welke container een HTTP-request zal verwerken, wordt het via deze weg lastig om wijzigingen vanuit het portaal naar beide connector containers te sturen.

Een meevaller is dat eMagiz al een systeem heeft waarmee het portaal doormiddel van JMS berichten events naar alle containers van een model kan sturen. Deze functionaliteit wordt onder andere gebruikt voor het tijdens runtime kunnen starten of stoppen van een specifieke flow. Dit systeem zou hergebruikt kunnen worden voor het versturen van een nieuwe rate limit configuratie naar de containers. Eén van de use-cases voor het dynamisch maken van rate limits was echter dat limieten ook vanuit een container gewijzigd kunnen worden. Dit kan bijvoorbeeld nodig zijn wanneer limieten verlaagd moeten worden als het CPU-gebruik hoog is. Doordat containers geen directe toegang hebben tot andere containers, is het synchroniseren van configuraties in deze situatie niet mogelijk.

De uiteindelijk bedachte oplossing is om configuraties te synchroniseren via een gedistribueerde cache. EMagiz maakt al gebruik van Infinispan. Dit is een cacheprovider waarmee caches gesynchroniseerd kunnen worden tussen gedistribueerde containers. Ook biedt Infinispan ondersteuning om te luisteren naar updates in de cache. Met deze oplossing kan het portaal een gewijzigde rate limit configuratie naar één van de containers sturen, welke deze naar een specifieke configuratie cache schrijft. Alle andere containers kunnen luisteren naar updates in deze cache en op deze manier de update ook ontvangen. Een visuele weergave hiervan is te zien in Figuur 6.



Figuur 6 Architectuuroverzicht

In dit diagram is met nummers weergegeven hoe een configuratie wijziging verloopt. De eerste stap is de gebruiker die een configuratie in het portaal wijzigt. Deze nieuwe configuratie wordt vervolgens naar één van de containers doorgezet, welke de configuratie wegschrijft naar de Infinispan configuratie cache. Voor persistentie wordt dit ook weggeschreven naar EFS (stap 5). De overige containers zullen vervolgens allemaal de cache synchroniseren via het netwerk. Bij deze synchronisatie wordt een CacheUpdateEvent getriggerd waarmee de containers de interne configuratie kunnen updaten. Wanneer een nieuwe container opstart kan deze de meest actuele configuratie ook uit de cache inlezen. Dit is weergegeven met nummer 6. Ten slotte biedt deze opzet ook de mogelijk om configuraties te updaten buiten het portaal om. Als voorbeeld hiervan is stap 7 opgenomen. Dit kan bijvoorbeeld een monitoring container zijn, welke limieten verlaagt of verhoogt op basis van CPU of memory gebruik.

Class analyse

Toen duidelijk was dat de architectuur het toelaat om wijzigingen vanuit het portaal naar alle containers te sturen, is de code van de Bucket4J-spring-boot-starter library geanalyseerd.

De analyse is begonnen met het in kaart brengen van de architectuur van de library doormiddel van een class diagram. Dit diagram is te vinden in Figuur 9 in Bijlage 1. Het doel van dit diagram is om een beter beeld te krijgen van de structuur van de library en inzicht te krijgen in hoe rate limit configuraties worden ingeladen en toegepast. Gezien het grote aantal classes in de library is het diagram sterk vereenvoudig om het overzichtelijk te houden. Dit is gedaan door slechts de Infinispan gerelateerde classes weer te geven. Daarnaast is slechts de servlet variant van de Bucket4JAutoConfiguration uitgewerkt, aangezien de verschillen met de andere implementaties beperkt zijn. Ten slotte zijn classes met betrekking tot Tests, Exceptions en Metrics weggelaten.

Aan de linkerzijde in het class diagram is de Bucket4JBootProperties class te zien. Deze class en alle classes eronder zijn afkomstig uit de bucket4j-spring-boot-starter-context package. Uit de analyse bleek dat de classes in de context package dienen als data objecten voor het inlezen van configuraties uit de application properties. Deze properties worden ingeladen doormiddel van Spring annotaties op de Bucket4JBootProperties class en gevalideerd met behulp van de Jakarta validator. Alle overige classes in het diagram zijn afkomstig uit de bucket4j-spring-boot-starter package.

Aan de rechterzijde van het diagram zijn de CacheResolver gerelateerde classes te zien. Deze classes worden gebruikt om de cache interacties van Bucket4J met de verschillende soorten caches zoveel mogelijk te abstraheren. Bij deze resolvers is een duidelijk onderscheid te zien tussen synchrone en asynchrone cache resolvers. Het type resolver dat aangemaakt wordt is van belang voor het instantiëren van de Bucket4JAutoConfiguration subclasses. Zo zullen Servlet filters alleen geïnstantieerd worden wanneer een synchrone cache resolver bestaat en vereisen Webflux en Gateway filters dat een asynchrone resolver bestaat.

Midden onder in het diagram is het ServletRequestFilter te zien. Dit is een door Spring gemanagede Filter Bean waar de daadwerkelijke rate limiting op requests wordt toegepast op basis van FilterConfiguration objecten. Ten slotte is linksboven in het diagram de Bucket4JBaseConfiguration met de verschillende implementaties hiervan te zien. Deze classes zijn de connectie tussen alle hiervoor genoemde classes. Deze classes zijn verantwoordelijk voor het parsen van de data objecten naar daadwerkelijke filter configuraties. Ook worden hier de filter beans aangemaakt en geregistreerd in de Spring applicatie context.

Hoewel dit diagram de kern van de library goed in kaart brengt, is de volledige structuur lastig in één diagram vast te leggen door de vele (functionele) interfaces, geneste classes en hoeveelheid ondersteunde caches.

Code analyse

Nadat de structuur van de applicatie in kaart was gebracht is met meer diepgang gekeken naar de feitelijke code. Het doel van deze code analyse was om in kaart te brengen wat de mogelijkheden en potentiële obstakels waren voor het dynamisch kunnen wijzigen van limieten. Hierna worden de belangrijkste bevindingen toegelicht.

Eén van de belangrijkste ontdekkingen bij deze analyse was de manier van bouwen van configuraties voor Filters. Deze worden bij het opstarten van de applicatie via een protected methode in de Bucket4JBaseConfiguration class gebouwd en kunnen dus alleen vanuit de subclasses aangemaakt worden. Dit betekent dat een wijziging van een configuratie ook via de subclasses moet lopen of dat de structuur van de base class aangepast moet worden.

Een andere belangrijke ontdekking was de opstart volgorde. Alle Filters worden aangemaakt en geregistreerd bij de Spring applicatie context tijdens het opstarten. Uit verder onderzoek bleek dat dit gedaan wordt omdat Spring het registreren of verwijderen van request filters beans alleen ondersteunt tijdens het opstarten van de applicatie. Ook bleek dat de Filters gebruik maken van FilterConfiguration objecten voor het daadwerkelijke filteren. Dit geeft de mogelijkheid om het object te vervangen en daarmee de filter configuratie te wijzigen. Het herleiden van de filter is echter lastig doordat de beans een naam krijgen op basis van de index in de configuratie. Wanneer deze index niet bekend is zal het onmogelijk zijn het juiste filter te herleiden.

Bij het bestuderen van de code van de cache resolvers bleek dat gebruik gemaakt wordt van cache-specifieke ProxyManagers van de Bucket4J library. Om te begrijpen hoe deze werken is de documentatie van Bucket4J geraadpleegd. In deze documentatie is te lezen dat de configuratie van rate limit buckets in de cache worden opgeslagen op het moment dat deze aangemaakt wordt. Wanneer op een later moment de bucket wordt opgevraagd met een andere configuratie, wordt deze nieuwe configuratie genegeerd. Dit is cruciaal om te weten wanneer configuraties vervangen moeten worden. Gelukkig biedt de Bucket4J library de optie om zowel expliciet als impliciet configuraties te vervangen. Bij expliciet vervangen moet de key in de cache bekend zijn, wat lastig te bepalen is wanneer dit op basis van gebruikersnamen gaat. In het geval van impliciet vervangen wordt dit gedaan op basis van versienummers van de configuratie bij het opvragen van de bucket. Wanneer een nieuwere versie gedetecteerd wordt vervangt Bucket4J automatisch de configuratie.

Ten slotte was een belangrijke constatering dat Bucket4J het vervangen van configuraties wel ondersteunt, maar dat dit complicaties heeft wanneer meerdere bandwidths per rate limit gebruikt worden. Deze bandwidths worden gebruikt om te bepalen wat de capaciteit van een bucket is en hoe snel deze aangevuld moet worden. Wanneer er meerdere bandwidths zijn en actuele limieten overgezet moeten worden kan de library niet bepalen welke bandwidth is aangepast (Bucket4J 8.5.0 reference, z.d.). Dit kan worden opgelost door de limieten te resetten, maar voor langlopende quota's is dit niet wenselijk. Wanneer limieten wel overgenomen moeten worden kan dit opgelost worden door het configureren van een id voor elke bandwidth. De Bucket4J-spring-boot-starter library biedt momenteel echter nog geen ondersteuning voor het configureren van id's.

Ontwerp

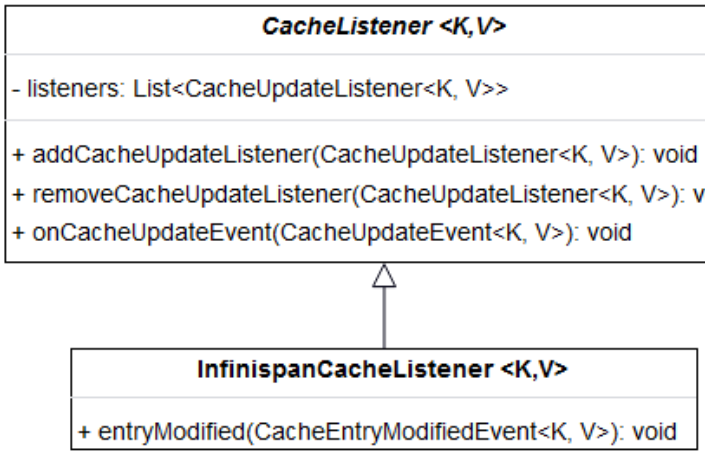
Nadat de belangrijkste obstakels en mogelijkheden in kaart waren gebracht is een ontwerp opgesteld waarmee het wijzigen van configuraties tijdens runtime mogelijk wordt. Een vereenvoudigd class diagram van dit nieuwe ontwerp is te vinden in Figuur 10 in Bijlage 2. Alle in het groen weergegeven classes zijn nieuwe toegevoegd. De belangrijkste wijzigingen in het ontwerp zullen hierna worden toegelicht. De volledige lijst met wijzigingen is terug te lezen in het systeem ontwerp document.

Aan het begin van dit deelhoofdstuk zijn de architectuur wijzigingen toegelicht. Daarbij is de keuze gemaakt om een cache-event gebaseerd systeem te implementeren voor het wijzigen en synchroniseren van configuraties. Ook bij het ontwerpen is allereerst gekeken naar hoe dit in de library geïmplementeerd kan worden.

Omdat elke cacheprovider een eigen implementatie heeft voor het luisteren naar wijzigingen in de cache, is besloten om voor elke provider een eigen Listener class te maken. In het diagram is dit weergegeven als de InfinispanCacheListener<K, V>, maar er zal ook bijvoorbeeld een JCacheListener en IgniteCacheListener komen. Deze cache listeners maken gebruik van generics om bij het aanmaken te bepalen welk type data in de cache zit. Door deze generieke opzet wordt het gebruik van de listeners niet beperkt tot de configuratie caches, maar kan het indien gewenst ook op andere caches toegepast worden. Wanneer een listener een wijziging in de cache detecteert zal deze een CacheUpdateEvent genereren. Dit event bevat de key, (indien mogelijk) de oude data en de nieuwe data.

Het gegenereerde CacheUpdateEvent heeft echter geen waarde wanneer het slechts in de listener beschikbaar is. Om het event naar andere systemen te krijgen is gekozen voor het observer-pattern. Met het observer-pattern kunnen classes een interface implementeren en zichzelf registreren als luisteraar voor een specifiek event bij een publisher class. Zodra een dergelijk event plaatsvindt worden alle geregistreerde observers daarvan op de hoogte gebracht door een methode van de interface aan te roepen met het event. Op deze manier kan een class luisteren naar wijzigingen in een

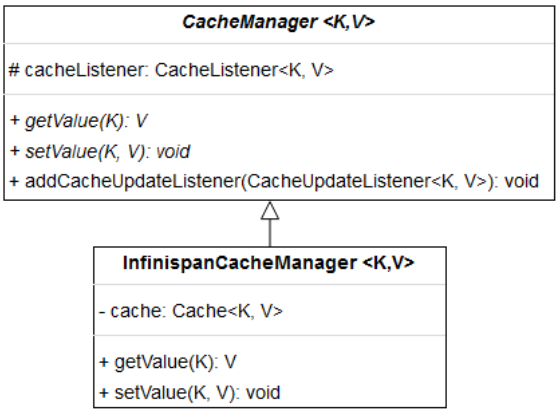
cache, de cache update omzetten naar een CacheUpdateEvent en de ‘onCacheUpdateEvent’ methode van alle observers aanroepen. Effectief betekent dit dat elke cache listener een lijst heeft met CacheUpdateListeners en methoden waarmee deze geregistreerd of verwijderd kunnen worden. Ook zullen ze allemaal een methode hebben waarmee de onCacheUpdate methode van alle geregistreerde CacheUpdateListeners wordt aangeroepen. Ter voorkoming van code duplicatie is daarom een abstracte class genaamd CacheListener gemaakt, welke als baseclass van elk van de listeners zal dienen. In onderstaand Figuur 7 zijn deze classes te zien.



Figuur 7 CacheListener en Infinispan implementatie

Naast het luisteren naar updates van de configuratie cache, zal het ook mogelijk moeten zijn om te lezen en schrijven naar deze cache. Om dit mogelijk te maken zal een CacheManager geïntroduceerd worden. Net als voor het luisteren naar updates, zal de implementatie hiervan per cacheprovider verschillen. Ook voor de cache managers zal dus per type provider een eigen class aangemaakt worden. Door de nauwe connectie tussen cache managers en listeners zal de manager daarnaast verantwoordelijk worden voor het instantiëren van de bijbehorende listener.

Deze manier van instantiëren betekent echter wel dat slechts de manager beschikking heeft over de listener instanties. Wanneer een CacheUpdateListener zich bij de cache listener wil registreren zal dit dus via de manager moeten lopen. Omdat dit voor iedere cache manager het geval is zal een abstracte CacheManager class geïntroduceerd worden met een addCacheUpdateListener methode, welke dit gelijk doorzet naar de cache listener instantie. Daarnaast zal deze class abstracte methoden genaamd getValue en setValue hebben, om af te dwingen dat dit door elke cache implementatie wordt ondersteund. In onderstaand Figuur 8 is de CacheManager met de Infinispan variant te zien.



Figuur 8 CacheManager en Infinispan implementatie

Het aanmaken van de CacheManager instanties zal gebeuren via de CacheResolver. Deze keuze is gemaakt omdat voor iedere cacheprovider al een resolver class bestaat met toegang tot de caches. Daarnaast hebben de Bucket4JBaseConfiguration subclasses al beschikking over de cache resolver, waardoor de bijbehorende methode eenvoudig aangeroepen kan worden.

De laatste grote wijziging welke in Figuur 10 te zien is, is dat de Bucket4JBaseConfiguration class de CacheUpdateListener interface implementeert. Door deze interface te implementeren in de base class wordt elk van de drie subclasses geforceerd om de bijbehorende onCacheUpdateEvent methode te implementeren. Het implementeren van de interface maakt het daarnaast mogelijk dat deze subclasses zich via de cache manager kunnen registreren als observer voor cache updates. Het is noodzakelijk om dit vanuit de AutoConfiguration classes te doen omdat dit de enige plek is waar de build methode voor filter configuraties aangeroepen kan worden. Omdat dit alleen hier mogelijk is, zal ook in deze class bij het opstarten van de applicatie gecontroleerd moeten worden of de cache een nieuwere versie bevat dan de application.properties. Indien de cache een nieuwere versie bevat zal die gebruikt worden voor het bouwen van de filters. De logica voor het opvragen uit de cache zal in de Bucket4JBaseConfiguration geplaatst worden omdat dit voor elk van de subclasses hetzelfde is.

Naast de in het class diagram weergeven wijzigingen zullen nog enkele inhoudelijke wijzigingen doorgevoerd moeten worden. Dit is onder andere het toevoegen van een id en versienummers aan de Bucket4JConfiguration class. Om onderscheid te kunnen maken tussen configuratie wijzigingen vanuit het portaal en interne wijzigingen worden deze versienummers opgedeeld in interne en externe versies. Deze worden vervolgens gecombineerd tot één long value zodat ze gebruikt kunnen worden door de Bucket4J proxy managers. Daarnaast moet elke CacheResolver implementatie worden aangepast om configuraties impliciet te vervangen op basis van dit versienummer. Het id van de Bucket4JConfiguration zal worden gebruikt voor de naam van de Filter beans zodat deze eenvoudig op te vragen zijn via de applicatie context.

Verder moet de BandWidth class uitbereid worden met een id property en moeten deze id's bij het bouwen van bucket configuraties worden toegepast. Dit is om het mogelijk te maken token limieten over te kunnen nemen bij het vervangen van een configuratie. Daarnaast zal de TokensInheritanceStrategy als property aan de RateLimit class worden toegevoegd zodat dit daadwerkelijk geconfigureerd kan worden.

Aangezien niet iedere gebruiker van de library gebruik zal willen maken van de configuratie caching functionaliteit zal het ook mogelijk moeten zijn dit uit te zetten. Dit wordt mogelijk gemaakt door een extra boolean property aan de Bucket4JBootProperties toe te voegen waarmee de caching aan gezet kan worden. Ook zal een property toegevoegd worden met de naam van de configuratie cache, zodat deze door de gebruiker gewijzigd kan worden.

Ook zal gevalideerd moeten worden dat de nieuwe properties correct geconfigureerd worden. Dit zal gedaan worden door de al bestaande validaties uit te bereiden. Zo zal worden gevalideerd dat alle filters een uniek id hebben en dat bandwidths een id moeten hebben wanneer een andere TokensInheritanceStrategy dan reset wordt gebruikt. Daarnaast wordt gevalideerd dat alle bandwidths een unieke id hebben binnen de RateLimit en dat het gecombineerde versienummer binnen de maximale waarde van een Long valt. Voor elk van deze validaties zullen ook unittests worden geschreven om te garanderen dat de validaties werken. Dit zal extra vertrouwen geven bij het doorvoeren van toekomstige wijzigingen doordat eenvoudig gecontroleerd kan worden dat de wijzigingen geen invloed hebben op deze functionaliteiten.

Ten slotte zal ieder van de voorbeeld projecten worden uitgebreid met een Rest endpoint voor het updaten van een filter configuratie. Dit geeft gebruikers van de library een idee hoe het dynamisch wijzigen van configuraties geïmplementeerd kan worden. Daarnaast maakt dit het mogelijk om integratietests te schrijven voor het vervangen van configuraties voor elk van de ondersteunde cache typen. Bij deze integratie tests zal worden gevalideerd dat het vervangen van een configuratie ook daadwerkelijk wordt toegepast op daaropvolgende requests.

Ontwerpkeuzes
Bij het opstellen van het ontwerp moesten verschillende implementatiekeuzes gemaakt worden. Hierna worden enkele van de alternatieven besproken en waarom hier niet voor gekozen is.

De eerste keuze was bij het toevoegen van een id aan filters. In het huidige ontwerp is dat geïmplementeerd door een nieuwe property aan de Bucket4JConfiguration toe te voegen. Een alternatief daarop is de lijst met configuraties in de Bucket4JBootProperties vervangen voor een HashMap. Met een map kan worden afgedwongen dat iedere configuratie een unieke key heeft, welke als id gebruikt kan worden. De reden dat hier niet voor is gekozen is omdat het een grote impact heeft op het format van de application.properties en application.yml. Met deze wijziging zouden bestaande gebruikers een groot deel van hun configuratie moeten herschrijven, zelfs als ze geen gebruik willen maken van het cachen en updaten van filter configuraties.

Hetzelfde geldt voor het id van bandwidths. Ook daar is gekeken naar de optie voor een HashSet, maar het configureren van een id is slechts in specifieke situaties noodzakelijk. Het afdwingen van een id via een HashSet schiet daarmee het doel voorbij.

Verder is gekeken naar de Spring ApplicationEventPublisher voor het publiceren van CacheUpdateEvents. Dit zou veel code schelen omdat de observers zich dan niet langer rechtstreeks bij de listeners en managers hoeven te registreren. In plaats daarvan kunnen alle Spring Beans dan eenvoudig doormiddel van een annotatie op een methode luisteren naar events.

Het obstakel is echter dat de CacheListeners niet door spring gemanaged worden, waardoor ze niet zomaar beschikking hebben over de ApplicationEventPublisher. Eventueel zou de publisher via de constructor geïnjecteerd kunnen worden door de CacheManager, welke ook geen Spring Bean is en het dus ook geïnjecteerd moet krijgen. Dit voegt extra dependencies toe aan elk van de tussenliggende classes wat niet wenselijk is.

Een oplossing voor bovenstaand obstakel zou kunnen zijn om de CacheManager en CacheListener door Spring als Bean aan te laten maken. De CacheManager zou dan niet langer via de CacheResolver geïnstantieerd worden door de Bucket4JAutoConfiguration classes, maar via de constructor geïnjecteerd worden. Het is echter mogelijk dat het cachen van configuraties uit gezet wordt waardoor de manager Bean niet zal bestaan. Als dit een constructor argument is zal Spring de CacheManager niet kunnen injecteren en zal een exceptie ontstaan bij het opstarten van de applicatie. Verder zou de meest logische plek om dergelijke beans te instantiëren in de CacheConfiguration class zijn omdat daar beschikking is over de CacheContainer. Deze configuratie wordt echter nooit uitgevoerd wanneer een gebruiker een eigen CacheResolver instantie aanmaakt, waardoor dit niet betrouwbaar is. Vanwege deze obstakels is de keuze gemaakt om de CacheManager en CacheListener niet als Spring Bean aan te maken.

De laatste keuze was omtrent het instantiëren van de cache listeners. Aangezien de managers en listeners altijd dezelfde cache moeten gebruiken, is de keuze gemaakt de managers verantwoordelijk te maken voor het instantiëren van de listeners. Om af te dwingen dat alle manager implementaties dit daadwerkelijk doen, is de CacheListener class als constructor argument aan de CacheManager class toegevoegd. Het probleem hierbij was echter dat het aanroepen van de super constructor in Java altijd op de eerste regel van de constructor moet. Dit betekent dat het instantiëren van de listener binnen de super() methode moet. Als daarna andere acties met deze listener uitgevoerd moeten worden zal deze eerst via de super opgevraagd moeten worden, wat niet ideaal is. Eén van de bekeken alternatieven is de listener net als de manager door de CacheResolver aan te laten maken en via de constructor mee te geven. Dit geeft echter nog meer dependencies voor de resolver die niet persé nodig zijn. Daarnaast is gekeken naar het implementeren van een abstracte methode waarmee de listener aangemaakt kan worden en deze methode door de CacheManager in de constructor aan te laten roepen. Deze optie is echter niet gekozen omdat het mogelijk is dat een CacheListener aanvullende parameters nodig heeft, waar de CacheManager baseclass niet over beschikt.

Conclusie

Met het hiervoor besproken ontwerp zou het in theorie mogelijk moeten zijn om rate limits dynamisch te wijzigen zonder dat de applicatie herstart hoeft te worden. Wanneer een gedistribueerde cache zoals Infinispan gebruikt wordt zal de oplossing ook toepasbaar zijn in een horizontaal schaalbare omgeving zonder dat verschillen in configuraties tussen de containers ontstaan. Door de generieke opzet zou de oplossing voor elk van de ondersteunde cacheproviders geïmplementeerd moeten kunnen worden. Dit vergroot de kans dat de wijzigingen geaccepteerd worden als bijdrage aan het open-source project. Het ontwerp vereist wel wijzigingen aan de bestaande code, maar dit is zoveel mogelijk beperkt om de impact op de codebase zo klein mogelijk te houden. Daarmee voldoet het ontwerp aan de opgestelde deelvraag.

5 Realisatie

Nadat alle deelvragen beantwoord waren is gewerkt aan het realiseren van het prototype. In tegenstelling tot het oorspronkelijke plan om een prototype te realiseren dat als generieke oplossing zou dienen, is een prototype voor het dynamisch kunnen wijzigen van rate limits met de Bucket4J-spring-boot-starter library gerealiseerd. In dit hoofdstuk wordt besproken hoe het wijzigen van de library is aangepakt en tot welk resultaat dit heeft geleid.

5.1 Het proces

Voor het realiseren van de wijzigingen werd gewerkt volgens de Scrum-methodiek met wekelijkse sprints. Daarbij werd dagelijks deelgenomen aan de stand-up waarin de huidige werkzaamheden werden besproken en eventuele obstakels waar tegenaan gelopen werd. Daarnaast is wekelijks een voortgangsgesprek gehouden met de bedrijfsbegeleider. Tijdens deze gesprekken werd het resultaat van de afgelopen sprint/week besproken en overlegd over eventuele obstakels. Ook zijn deze momenten gebruikt voor het reviewen van code en documentatie. Door deze wekelijkse sprints kon snel worden ingespeeld op eventuele feedback.

Om wijzigingen aan de library te kunnen maken is een fork gemaakt van de master branch van de Bucket4J-spring-boot-starter library op GitHub. Op deze geforkte repository zijn vervolgens branches gemaakt voor de verschillende onderdelen waar aan gewerkt werd. Wanneer een onderdeel afgerond en getest was, werden de code wijzigingen besproken met de bedrijfsbegeleider. Eventuele feedback tijdens deze reviews is vervolgens verwerkt, waarna alle wijzigingen naar een centrale branch gemerged werden.

Om te valideren dat code werkt zoals verwacht is tijdens het ontwikkelen doorlopend handmatig getest. Dit werd gedaan door demo projecten van de library te starten en met behulp van Postman requests naar een endpoint te sturen. Omdat handmatig testen veel tijd kost zijn daarnaast verscheidene unit- en integratie tests opgesteld. Met deze tests kon eenvoudig gevalideerd worden dat nieuwe wijzigingen geen impact hadden op eerder gerealiseerde code. Voordat een branch gemerged werd is dan ook altijd eerst gecontroleerd of alle testcases nog steeds slaagden.

Omdat alle wijzigingen bij elkaar tot een zeer grote pull-request leiden is ongeveer drie maanden voor de deadline van het afstudeerproject contact gelegd met de maintainer van de library. Bij dit contact is de maintainer op de hoogte gebracht van de werkzaamheden en is gevraagd of hij interesse heeft om de feature aan de library toe te voegen. Op dit voorstel werd erg positief gereageerd, waarna in overleg een vroegtijdig pull-request is aangemaakt. Dit vroegtijdige pull-request stelde de maintainer in staat om tijdens het ontwikkelen alvast feedback te geven. Hierdoor was het mogelijk feedback eerder in het proces te verwerken, waarmee de kans op uiteindelijke goedkeuring verhoogd werd.

Hoewel de focus van de opdracht lag op het dynamisch maken van de library, was de reden van deze wijzigingen dat de library aan de requirements van eMagiz moest voldoen. Om aan te tonen dat de library inclusief de wijzigingen ook daadwerkelijk toepasbaar is voor API-gateway integraties is de afrondende fase van het project gebruikt om een proof of concept op te stellen. Hiervoor is een aparte branch in het eMagiz project aangemaakt waar de rate-limiting library aan toegevoegd kon worden.

5.2 De Ontwikkelomgeving

Voor het wijzigen van de library en het uitvoeren van testcases is gebruik gemaakt van IntelliJ Ultimate. Om te voorkomen dat doorgevoerde wijzigingen verloren gaan in het geval van hardware problemen is GitHub gebruikt als versiebeheer systeem. De keuze voor GitHub wijkt af van de standaard binnen eMagiz, waar gebruik gemaakt wordt van Bitbucket. Toch is hiervoor gekozen omdat het forken van de library eenvoudiger is in GitHub. Daarnaast is het plan om de wijzigingen bij de library in te mergen, waarna het niet langer nodig zal zijn de code in een aparte eMagiz repository te bewaren.

Voor het builden van de code maakt de library gebruik van Maven. Ook dit is een afwijking van eMagiz, waar gebruik gemaakt wordt van Gradle. Om toch lokaal te kunnen builden is hierom Maven versie 3.9.5 geïnstalleerd.

Ten slotte is gebruik gemaakt van Docker Desktop. Docker was noodzakelijk voor het kunnen testen van de implementaties voor Reddison, Lettuce en Jedis caches, welke verbinding moeten kunnen maken met een Redis instantie. Hiervoor is gebruik gemaakt van de Redis image versie 7.2. Daarnaast maakt de library gebruik van de TestContainers library voor het uitvoeren van testcases. Ook deze vereist dat Docker containers aangemaakt kunnen worden. Ten slotte is Docker gebruikt voor het kunnen testen van de rate limiting library in combinatie met de eMagiz runtime en Infinispan in een gedistribueerde omgeving.

5.3 Het prototype

De volledige code van de library is toegevoegd aan het portfolio, voorzien van begeleidend schrijven met een omschrijving van de aangebrachte wijzigingen. De GitHub pull-request is te vinden op: https://github.com/MarcGiffing/bucket4j-spring-boot-starter/pull/204#

Hierna zal worden toegelicht hoe het prototype is gerealiseerd, welke obstakels daarbij zijn overwonnen en waarom op sommige plekken is afgeweken van het oorspronkelijke ontwerp.

CacheListener

Het realiseren van de wijzigingen is begonnen met het toevoegen van de CacheListener interface en bijbehorende implementaties voor elk van de cache providers. Bij het implementeren van de cache specifieke listeners kwam echter gelijk een probleem met het ontwerp naar voren. Vrijwel alle cacheproviders bieden de mogelijkheid om te luisteren naar wijzigingen in de cache door het implementeren van een interface of het gebruik van annotaties. Bij het vooronderzoek was echter geen diepgaand onderzoek gedaan naar de exacte implementatie hiervan voor elk van de caches. Bij het implementeren van de JedisCacheListener bleek dat Jedis de uitzondering op de regel is en vereist dat listeners de JedisPubSub class extenden. Java biedt echter geen ondersteuning voor het extenden van meerdere classes, waardoor het extenden van de abstracte CacheListener class niet mogelijk is. Als oplossing hiervoor is de CacheListener gewijzigd naar een interface, maar dit betekende wel dat elk van de CacheListener implementaties dezelfde code bevat voor het registreren van observers. Later gaf een collega hierbij nog als feedback dat ook een combinatie van de abstracte class en een interface mogelijk was geweest. Op die manier kan een uitzondering gemaakt worden voor Jedis, zonder dat code duplicatie voor de andere listeners nodig is. Op het moment van implementeren was dit echter niet bedacht en zoals verder op in dit hoofdstuk te lezen is was dit achteraf ook niet langer nodig.

CacheManager
De vervolgstap was het implementeren van de CacheManager class en de implementaties hiervan voor elke cacheprovider. Voor veel van de caches was het een eenvoudige opgave om lezen en schrijven naar de cache mogelijk te maken en listeners te registreren. Vooral de verschillende Redis varianten waren ook hier de uitzondering op de regel. Ten eerste maakt Redis geen onderscheid in caches zoals bijvoorbeeld Infinispan en Ignite. Als oplossing hiervoor is de keuze gemaakt om de data in een map op te slaan, waarbij de cache naam als key van de map gebruikt wordt. Het volgende probleem is dat Jedis en Lettuce data als byte array of String opslaan. Dit betekent dat de data zowel bij het lezen als schrijven naar de cache geparsed moet worden. Om dit mogelijk te maken wordt gebruik gemaakt van de Jackson ObjectMapper, welke eenvoudig objecten naar String format kan parsen. Bij het testen bleek echter dat dit niet gelijk werkte omdat gebruik gemaakt wordt van generics. Als oplossing hiervoor wordt via de constructor meegegeven van welk type de key en value in de cache zijn, zodat naar deze typen geparsed kan worden. Ook was het noodzakelijk de Serializable interface te implementeren voor sommige van de property classes.

Het laatste probleem in de CacheManagers was het publiceren van update events wanneer een waarde in de cache gewijzigd is. Veel van de cache providers doen dit automatisch wanneer naar de cache geschreven wordt. Ook hier waren de Redis varianten de uitzondering. Voor zowel Jedis, Lettuce als Redisson is dit opgelost door zelf een event te publiceren op een specifiek channel wanneer een update plaatsvindt. Ook deze events moeten gepubliceerd worden in String format. Dit had tot gevolg dat ook de CacheListeners aangepast moesten worden om binnengekomen event Strings terug te parsen naar een CacheUpdateEvent. Hiervoor geldt hetzelfde generics probleem als bij het lezen en schrijven van de cache, waardoor de key en value typen ook via de constructor aan de listeners meegegeven moeten worden. Bij het testen hiervan bleek echter dat dit nog steeds niet werkte voor Jedis. Na diepgaand onderzoek werd de oplossing gevonden door de Jedis versie vast te zetten op 4.4.6. Bij versie 5.0.0 of hoger wordt de onMessage() methode van de Listener nooit aangeroepen. Een andere oplossing voor dit probleem kon niet worden gevonden.

CacheResolver
Nadat alle listeners en managers geïmplementeerd waren zijn de cacheResolvers aangepast. Dit is in overeenstemming met het ontwerp gedaan door de “resolveCacheManager(cacheName)” methode aan de CacheResolver interface toe te voegen. Hiermee werd automatisch afgedwongen dat de methode in elk van de CacheResolver implementaties werd geïmplementeerd. Zo kon ook eenvoudig bepaald worden of het implementeren ergens vergeten was.

Nieuwe properties
Nadat alle cacheResolvers waren aangepast zijn de nieuwe properties toegevoegd aan de Bucket4JBootProperties en onderliggende property classes. Dit zijn onder andere het id veld voor de Bucket4JConfiguration en Bandwidth classes, TokensInheritanceStrategy voor de RateLimit class en filterConfigCachingEnabled voor de Bucket4JBootProperties class. Een overzicht van alle nieuwe properties is te zien in Figuur 12. Ook zijn deze nieuwe properties gelijk voorzien van validaties. Eenvoudige validaties, zoals dat een property geen null of lege String mag zijn, zijn afgehandeld doormiddel van Jakarta Validation annotaties boven de properties. Complexere validaties zijn geïmplementeerd doormiddel van validatie methoden in de Bucket4JBaseConfiguration class. Een voorbeeld daarvan is het valideren van bandwidth id’s, welke altijd uniek moeten zijn binnen de RateLimit en alleen vereist zijn wanneer de RateLimit zowel een andere TokensInheritanceStrategy dan ‘RESET’ heeft als meer dan één bandwidth bevat. Om zeker te weten dat de validaties werken zoals verwacht zijn daarnaast verscheidene unit-tests opgesteld.

CacheUpdateEvent handling
Na het toevoegen van de properties is de Bucket4JBaseConfiguration class aangepast zodat deze de CacheUpdateListener interface implementeert, gevolgd door het implementeren van de bijbehorende methode in de sub-classes. Om de sub-classes als listener te kunnen registreren is de CacheManager als constructor argument aan de baseclass toegevoegd, waar deze zichzelf registreert. Dit garandeert dat de sub-classes altijd luisteren naar update events. Om te kunnen controleren of caching aan staat moest daarnaast de Bucket4JBootProperties meegegeven worden via de constructor. Ten slotte moesten de sub-classes bij het opstarten controleren of de cache een nieuwere versie bevat dan de application.properties en de cache updaten als de configuratie nog niet in de cache zit. Omdat dit gedrag voor elk van de sub-classes hetzelfde is, is dit als protected methode opgenomen in de base class.

Integratie testen en validatie refactoring
De vervolg stap was valideren of het updaten van configuraties werkt. Hiervoor is de TestController class in één van de voorbeeldprojecten uitgebreid met een POST-endpoint. Met dit endpoint kan een nieuwe configuratie in de cache geïnjecteerd worden. Bij het testen van dit endpoint kwam aan het licht dat in de huidige opzet niet alle validaties uitgevoerd konden worden voordat de nieuwe configuratie in de cache geplaatst werd. Dit komt doordat de validatie methoden in de Bucket4JBaseConfiguration protected zijn en dus niet vanuit externe classes aan te roepen zijn. De eenvoudigste oplossing leek om deze validatie methoden public en static te maken, waarmee het probleem opgelost zou zijn. Dit bleek echter niet mogelijk te zijn doordat één van de validaties gericht is op het valideren van predicate namen op basis van ExecutePredicate beans. Deze beans worden in de sub-classes van de Bucket4JBaseConfiguration geïnjecteerd bij het opstarten en tijdens het valideren opgevraagd. Hierdoor was het static maken van de validatie methoden niet mogelijk. Daarnaast zou dit betekenen dat configuraties zowel met de Jakarta validator als de Bucket4JBaseConfiguration gevalideerd moeten worden, wat extra foutgevoeligheid introduceert. Omdat het refactoren van de validaties vrij grote impact heeft op de bestaande code van de library is eerst overlegd met de bedrijfsbegeleider. Daarbij is voorgesteld om de validaties in de Bucket4JBaseConfiguration te herschrijven naar (custom) Jakarta validation annotaties. De bedrijfsbegeleider stemde hiermee in, waarna de validaties zijn herschreven.

Bij het herschrijven van de validaties was wederom het valideren van de predicate definities een groot obstakel. Bij het valideren wordt onderscheid gemaakt tussen Webflux en Servlet predicates, welke als geautowired worden door Spring. Dit geeft gebruikers erg veel vrijheid doordat zij ook zelf predicate beans kunnen creëren, maar maakt het valideren ook erg ingewikkeld. De uiteindelijke oplossing hiervoor was het implementeren van een nieuwe validatie annotatie en validator class. In deze validator class worden de predicates via de constructor geïnjecteerd door Spring. Om onderscheid te kunnen maken tussen de Webflux en Servlet predicates zijn deze opgesplitst in twee parameters, namelijk de ExecutePredicate<HttpServletRequest> en ExecutePredicate<ServerHttpRequest>. Ook dit bleek problemen te geven doordat de context package geen dependency had voor HttpServletRequest. Als oplossing was de Jakarta-servlet-api dependency toegevoegd. Dit was een werkende oplossing, maar de library maintainer gaf aan dat hij hier bezwaar tegen had. Dit introduceert namelijk een nieuwe dependency welke niet nodig is wanneer gebruik gemaakt wordt van Webflux. Het voorstel van de maintainer om de scope van de dependency op provided te zetten leek in eerste instantie te werken, tot getest werd met een Webflux applicatie en een ClassNotFound exceptie ontstond. Dit was een complex probleem wat uiteindelijk is opgelost door alle ExecutePredicates als één constructor argument te injecteren en vervolgens te splitsen op generiek type met behulp van Springs GenericTypeResolver.

Om te voorkomen dat alsnog een ClassNotFound exception optreedt bij het splitsen, is ervoor gekozen de typecheck voor de HttpServletRequest te doen op basis van de class naam in plaats van class type. Na al deze tegenslagen was het resultaat dat de dependency toch optioneel gemaakt kon worden en alle validaties doormiddel van de Jakarta validator konden plaatsvinden.

Refactoren van het observer pattern

Na het refactoren van de validaties is gekeken of het toch mogelijk is om de CacheListeners gebruik te laten maken van Springs ApplicationEventPublisher. In het oorspronkelijke ontwerp was dit niet nodig omdat het registreren van- en publiceren naar observers in de abstracte CacheListener class zou gebeuren. Doordat dit een interface werd ontstond echter veel duplicate code tussen alle CacheListener implementaties. Gebruik maken van de ApplicationEventPublisher zou dit probleem oplossen, maar dit vereist wel dat de listeners Spring managed Beans worden zodat de publisher autowired kan worden. Dit is gerealiseerd door de Configuration classes waarin CacheResolvers worden geïnstantieerd te bewerken zodat hier ook, onder voorwaarden, de CacheListeners als bean aangemaakt worden. Voor deze wijziging moesten echter ook alle CacheManagers aangepast worden, omdat deze niet langer de CacheListener aan hoefde te maken. Bij het doorvoeren van deze wijzigingen is ook de keuze gemaakt de CacheManagers niet langer via de CacheResolver te laten aanmaken, maar in dezelfde configuration classes als de Listener. Dit voorkomt dat per ongeluk meerdere cachemanagers aangemaakt worden en maakt het ook beter inzichtelijk dat de listener en manager gebruik maken van dezelfde cache. Ook konden hierdoor alle wijzigingen aan de CacheResolver classes teruggedraaid worden, waardoor de impact op bestaande code verminderd werd. Na het refactoren van de code bleek dat de CacheListener interface overbodig was geworden, net als de methode waarmee observers zich konden registreren bij de CacheManager.

De CacheUpdateListener interface is wel behouden, maar de onCacheUpdateEvent methode van deze interface is voorzien van de @EventListener(CacheUpdateEvent.class) annotatie. Door deze minimale wijziging is het niet nodig de implementatie van deze methode in de Bucket4JAutoConfiguration sub-classes te wijzigen. Wel kon hierdoor de code voor het registreren als observer uit de constructor van de base-class verwijderd worden. Wel moesten de sub-classes aangepast worden zodat deze de CacheManager bean via de constructor geïnjecteerd kregen, in plaats van dat deze via de CacheResolver wordt opgevraagd. Omdat deze bean mogelijk niet bestaat als caching uit staat is deze gewrapped met de Optional class. Alles bij elkaar was dit een erg grote wijziging aan het oorspronkelijke ontwerp, maar het bood wel de kans om veel code duplicatie te verwijderen. Daardoor werd niet alleen de leesbaarheid van de code verbeterd, maar ook de impact op de bestaande codebase verkleind en het aantal dependencies tussen classes verminderd.

Overige obstakels

Naast bovengenoemde wijzigingen zijn tijdens het implementeren verschillende andere obstakels opgelost. Zo werd de onMessage methode van de JedisPubSub nooit aangeroepen bij Jedis versie 5.0.0 of hoger en gaf het Hazelcast voorbeeld project telkens een BindException bij het opstarten. Verder bleek een bug te zitten in het afhandelen van rate limit checks wanneer MatchingStrategy.ALL werd gebruikt bij Webflux of Gateway filters. Ook ontstond in eerste instantie raar gedrag bij het integratie testen van configuratie updates, waarbij de oude configuratie werd gebruikt met het nieuwe versienummer. Dit bleek te komen doordat de versienummers uit de oorspronkelijke configuratie gebruikt werd binnen de rate limit checks, in combinatie met de gebouwde configuratie. Tijdens de testcase werd echter het originele object aangepast, wat resulteerde in dit vreemde gedrag. Dit is opgelost door het originele object te klonen in de testcase en die aan te passen.

Feedback van de library maintainer
Na bovenstaande wijzigingen doorgevoerd te hebben was er een werkende oplossing en is in overleg met de maintainer van de library een pull-request aangemaakt. Dit stelde hem in staat om tijdens de afronding alvast feedback te geven. Eén van de feedback punten is hiervoor al besproken, namelijk de dependency van Jakarta-servlet-api. Een ander feedback punt ging over het valideren van gewijzigde properties welke aan de TestController waren toegevoegd in de voorbeeld projecten. Zo werd gevalideerd of de cacheName, filterMethod en filterOrder properties niet gewijzigd waren. De validaties op zichzelf waren geen probleem, maar de feedback hierop was dat van gebruikers niet verwacht mag worden dat zij weten welke properties gevalideerd moeten worden. Als oplossing hiervoor is een Utils class aangemaakt met een methode waarmee eenvoudig gevalideerd kan worden of een update valide is. De library maintainer stemde in met deze oplossing. Verder waren er enkele kleine feedback puntjes, welke snel opgelost konden worden.

Proof of Concept met de eMagiz runtime
Toen aangetoond was dat de library wijzigingen werkten voor de verschillende demo projecten is een proof of concept opgesteld met de eMagiz runtime. Het doel van dit proof of concept was om aan te tonen dat de gewijzigde library ook werkt in combinatie met de eMagiz implementatie van Infinispan. Hier ontstond echter gelijk een probleem doordat de eMagiz runtime nog op Java 8 draait, terwijl de library voor Spring boot 3.1 met Java 17 is gemaakt. Dit leek bij aanvang van het project geen probleem omdat eMagiz voornemend was om te migreren naar Java 17, maar door vertraging in de roadmap was dit nog niet gedaan. Aangezien het slechts een proof of concept betrof, is besloten alle niet benodigde packages te deactiveren en de resterende packages, waaronder de Infinispan componenten, naar Spring boot 3 te migreren. Ondanks verschillende complicaties is het uiteindelijk gelukt dit werkend te krijgen. Om gebruik te kunnen maken van de nieuwe functionaliteiten in de Bucket4J-spring-boot-starter library is de dynamic-rate-limiting branch lokaal gebouwd met behulp van Maven. Dit leidde tot twee .jar bestanden, één voor de context package en één voor de library zelf. Deze beide bestanden zijn op basis van bestandslocatie als dependency aan de eMagiz runtime toegevoegd zodat ze gebruikt konden worden.

Om het proof of concept zo nauw mogelijk te laten aansluiten bij de werkelijkheid, is ook gekeken naar de mogelijkheid om rate limits te configureren via XML-flows. Het eMagiz portaal is ontworpen om integratie flows om te zetten naar XML-flows, waardoor deze manier van configureren de voorkeur heeft over het gebruik van de application.properties. In eerste instantie deden zich problemen voor als gevolg van verschillen in de opstartvolgorde, maar deze konden worden opgelost door aan te geven dat sommige beans afhankelijk zijn van andere beans. Verder moest de Bucket4JBootProperties bean in de XML-flow als primary worden aangemerkt zodat het de application.properties overschreef. Om te valideren of de configuraties daadwerkelijk werkten is een eenvoudig endpoint aangemaakt met de Spring Integration HttpInboundChannelAdapter en Spring security met diverse accounts en verschillende rollen. Het testen van de rate limits op dit endpoint is handmatig gedaan met behulp van Postman. Bij het testen bleek dat het uitvoeren van een request spontaan vele malen langer duurde dan voorheen, soms zelfs tot meer dan een seconde. Dit was een groot probleem aangezien één van de requirements stelde dat dit maximaal 100ms mocht zijn. Bij nader onderzoek bleek de Spring Security migratie naar Spring boot 3 hier de oorzaak van te zijn. Omdat deze migratie op zichzelf geen onderdeel van het proof of concept was, is hier geen verder onderzoek naar gedaan. Wel zijn de bevindingen gedeeld met de collega die de Spring boot 3 migratie zal uitvoeren.

Nadat gevalideerd was dat het proof of concept werkte als de applicatie in IntelliJ uitgevoerd werd, is een BootJar van de applicatie gemaakt. Van deze bootjar is vervolgens een Docker image gemaakt, waarvan twee Docker containers zijn gemaakt. Deze containers hadden dezelfde configuratie, maar draaiden op twee verschillende poorten zodat een gedistribueerde omgeving geïmiteerd kan worden. Vervolgens is gevalideerd dat rate limits gedeeld werden tussen de containers en dat het updaten van een configuratie bij één container ook werd toegepast op de andere container. Na enkele kleine wijzigingen in de configuraties werkte alles zoals verwacht en was aangetoond dat de library wijzigingen voldoen aan de requirements van eMagiz.

5.4 Ontwerp wijzigingen

In het hierboven besproken realisatie process zijn diverse wijzigingen aan het oorspronkelijke ontwerp doorgevoerd. Vooral het wijzigen van het observer pattern naar gebruik maken van de Spring ApplicationEventPublisher had grote impact op de afhankelijkheden tussen verschillende classes en de algehele structuur. Om deze gewijzigde structuur inzichtelijker te maken is een nieuw vereenvoudigd class diagram opgesteld. Dit diagram is te vinden in Figuur 11 in Bijlage 3.

In dit diagram is goed te zien dat de cache managers, cache listeners en cache resolvers nu geen onderlinge afhankelijkheden meer bevatten. Dit is een groot voordeel doordat het de kans op problemen vermindert wanneer één van deze componenten aangepast moet worden. Verder is onder in het nieuwe diagram de cache configuratie class opgenomen. Deze class was in het oorspronkelijke diagram niet opgenomen omdat die niet gewijzigd hoefde te worden en functioneel niet van belang was. In het nieuwe ontwerp is deze configuratie class echter uitgebreid zodat deze ook de CacheManager en CacheListener beans aanmaakt. Om deze reden is de class nu wel in het diagram opgenomen. Ten slotte is boven in het diagram de Bucket4JUtils class toegevoegd, welke de methode bevat om te valideren of een configuratie update valide is. Inhoudelijke wijzigingen aan de classes zijn niet opgenomen omdat deze uitgebreid zijn besproken in de voorgaande hoofdstukken.

5.5 Alternatieven

Zoals besproken was één van de review opmerkingen dat gebruikers eenvoudig moeten kunnen controleren of een configuratiewijziging valide is. Dit is opgelost doormiddel van een Utils class, wat voldoende was in de ogen van de library maintainer. Een mooier alternatief zou echter zijn om elke CacheManager variant te laten extenden door een ConfigCacheManager. Doordat dit een specifieke implementatie is kunnen de generics van de CacheManager worden vastgezet op String en Bucket4JConfiguration. Dit maakt het vervolgens mogelijk om configuratie-specifieke validaties in de setValue methode te implementeren. Voor eventuele foutieve configuraties kunnen dan custom Exceptions gegooid worden. Hoewel deze oplossing mooier is, is het ook aanzienlijk meer werk om te implementeren. Doordat het alternatief goedgekeurd is en de deadline van het project dichtbij was, is de keuze gemaakt dit niet te implementeren maar als aanbeveling in het rapport op te nemen. Ook is deze aanbeveling aan de library maintainer doorgegeven als reactie op zijn review commentaar.

6 Conclusie

Voor de afstudeeropdracht is onderzoek gedaan naar de mogelijkheid om een generiek prototype te ontwikkelen waarmee rate limiting en/of throttling kan worden toegepast op zowel Event Streaming- als API-gateway integraties.

Het onderzoek is aangevangen met het in kaart brengen van de vereisten waar de oplossing aan moet voldoen. Vervolgens is de huidige architectuur en werking van beide integratiepatronen geanalyseerd. Uit deze analyse bleek dat de patronen zo goed als geen overeenkomsten bevatten, waardoor een generieke oplossing niet langer wenselijk was. Hiermee was de hoofdvraag in principe al beantwoord, namelijk dat een generieke oplossing niet mogelijk is binnen de gestelde requirements.

Uit de requirements analyse bleek echter ook dat wanneer een generieke oplossing niet mogelijk is, de focus op API-gateway zou moeten liggen. Op basis hiervan is onderzoek gedaan naar bestaande libraries waarmee throttling en/of rate limiting voor de API-gateway ingevoerd kon worden. Uit dit onderzoek kwamen voor zowel throttling als rate limiting oplossingen naar voren welke samen het overgrote deel van de requirements oplosten. Eén van de requirements welke echter niet opgelost werd is het kunnen wijzigen van rate limit configuraties zonder dat daar een nieuwe deployment voor nodig is.

Op basis van de resultaten van de library analyse is, in overleg met de opdrachtgever, besloten om de open source Bucket4J-spring-boot-starter library aan te passen zodat dynamisch wijzigen van limieten mogelijk wordt. Om dit mogelijk te maken is een ontwerp opgesteld. Bij het ontwerpen is geprobeerd de impact op de bestaande code base zo beperkt mogelijk te houden om de kans te vergoten dat de wijzigingen geaccepteerd worden als bijdrage aan het open source project.

Na goedkeuring van het ontwerp door de bedrijfsbegeleider is een fork van de library gemaakt en zijn de wijzigingen doorgevoerd. Door nieuwe inzichten tijdens het ontwikkelen is op bepaalde punten afgeweken van het oorspronkelijke ontwerp, met als resultaat dat minder afhankelijkheden tussen de nieuwe classes bestaan. Ook kon door deze wijzigingen code duplicatie aanzienlijk verminderd worden. Toen bleek dat alle wijzigingen werkten zoals verwacht, is een proof of concept opgesteld met de Infinispan componenten van de eMagiz runtime. Ook dit proof of concept is uitvoerig getest in zowel single instance als een gedistribueerde omgeving. Met deze tests is aangetoond dat het met de doorgevoerde wijzigingen mogelijk is om rate limits te wijzigen in een gedistribueerde omgeving zonder dat hiervoor een nieuwe deployment nodig is.

Het resultaat van deze thesis is dat voor zowel throttling als rate limiting een oplossing is gevonden welke toepasbaar is op API-gateway integraties en voldoet aan de gestelde requirements van eMagiz. Wel moet hierbij vermeld worden dat de throttling oplossing niet is getest doordat de focus, in overleg, op het wijzigen van de Bucket4J-spring-boot-starter library is komen te liggen. Aangezien de oplossing een component van Spring Integration betreft is de verwachting echter dat het implementeren hiervan geen problemen zal opleveren.

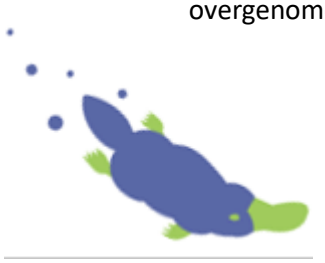
7 Reflectie

Terugkijkend op het project ben ik erg tevreden over het verloop, het behaalde resultaat en de nieuwe kennis die ik heb kunnen opdoen. Doordat ik mijn stage ook bij eMagiz had gelopen had ik al enige kennis van het eMagiz platform. Ondanks deze voorkennis heb ik toch veel bijgeleerd over de verschillende integratie patronen en de achterliggende architectuur. Ook heb ik veel geleerd over de verschillende methoden van rate limiting en throttling, en mijn kennis en ervaring van het Spring-framework kunnen vergroten.

Procesmatig verliep niet alles volgens de aanvankelijke planning. Bij het opstellen van de opdracht was al rekening gehouden met de mogelijkheid dat een library gevonden zou worden waarmee rate limiting geïmplementeerd kon worden. Het plan was om de opdracht in een dergelijke situatie uit te breiden met bijvoorbeeld het implementeren van metrics of rol-gebaseerde rate limiting. Het was echter niet voorzien dat een library werd gevonden die ook deze uitbreidingen al bevat. Om toch voldoende te kunnen laten zien voor de realisatie competentie is toen in overleg met de stakeholders besloten de library aan te passen zodat limieten dynamisch gewijzigd kunnen worden. Met deze uitbreiding kon niet alleen de competentie realiseren worden aangetoond, maar kon ook extra functionaliteit voor zowel eMagiz als overige gebruikers van de library worden gerealiseerd. Hoewel de oorspronkelijke planning niet strikt werd gevolgd, ben ik ervan overtuigd dat de situatie op een effectieve manier is aangepakt. De betrokken partijen zijn goed op de hoogte gehouden en beslissingen zijn samen genomen. Ook gaf de bedrijfsbegeleider aan dat hij dit als erg prettig had ervaren en dat het voor hem een goede indicatie was dat er gestructureerd en procesmatig gewerkt is.

Tijdens het ontwerpen en realiseren waren er momenten dat ik vastliep of dat belangrijke keuzes gemaakt moesten worden. Wanneer dit het geval was, is een sparringsessie aangevraagd met de bedrijfsbegeleider. In deze sessies legde ik het probleem of de keuze uit en gaf ik verschillende mogelijke oplossingen welke ik zelf al bedacht of geprobeerd had, samen met de voor- en nadelen. De bedrijfsbegeleider gaf vervolgens zijn mening of kwam met nieuwe inzichten waar ik nog niet aan gedacht had. Deze sessies waren een belangrijk onderdeel van het proces en hebben bij sommige keuzes veel invloed gehad op het uiteindelijke ontwerp. Een voorbeeld hiervan is de keuze om configuratie updates via de cache te laten verlopen in plaats van deze naar iedere runtime te sturen. Zowel ik als de bedrijfsbegeleider hebben deze sparringsessies als zeer prettig ervaren en ik ben ervan overtuigd dat dit de kwaliteit van de oplossing heeft verhoogd.

Een nadeel aan het aanpassen van de library was dat deze erg breed is opgezet. Dat wil zeggen dat de Bucket4J-spring-boot-starter voor Gateway, Servlet en Webflux endpoints gebruikt kan worden en daarnaast ook zeer veel cacheproviders ondersteunt. Op zichzelf is dit geen probleem, maar voor de casus van eMagiz is slechts Servlet ondersteuning met Infinispan caching benodigd. Dit betekent dat veel code geschreven moest worden voor caches die nooit door eMagiz gebruikt zullen worden. Een alternatief was geweest om een eigen rate limiting oplossing te programmeren met de Bucket4J library. Wel hadden functionaliteiten als SPEL-Expression parsing, metrics en predicates dan ook opnieuw geïmplementeerd moeten worden. Per saldo was dit waarschijnlijk evenveel werk geweest. Toch ben ik van mening dat ook hier de goede keuze is gemaakt. De reden hiervoor is dat binnen eMagiz veel gebruik gemaakt wordt van goed onderhouden open source libraries. Dit wordt onder andere gedaan zodat eenvoudig updates meegenomen kunnen worden en het wiel niet opnieuw uitgevonden hoeft te worden. Door de wijziging te ontwikkelen als bijdrage aan de library kan deze werkwijze nageleefd worden en kunnen toekomstige updates en nieuwe functionaliteiten eenvoudig overgenomen worden.



Ten slotte ben ik ervan overtuigd dat het verstandig was om de maintainer van de library tijdens het ontwikkelen te betrekken bij de code wijzigingen. De wekelijkse reviews met de bedrijfsbegeleider waren waardevol voor feedback tijdens het ontwikkelen, maar om de wijzigingen geaccepteerd te krijgen moet de library maintainer ze ook accepteren. Door hem tijdens het ontwikkelen de mogelijkheid te geven feedback te geven kon ik zijn opmerkingen gelijk verwerken. Daarmee is de kans zo groot mogelijk gemaakt dat het uiteindelijke pull-request geaccepteerd wordt.

Ondanks de onverwachte obstakels en wijzigingen ten opzichte van het originele plan, ben ik erin geslaagd om een product te realiseren dat voldoet aan de gestelde requirements. Ook heb ik met een proof-of-concept kunnen aantonen dat de wijzigingen functioneren in een gedistribueerde omgeving, gebruik makend van de eMagiz Infinispan componenten. Ik ben dan ook erg tevreden over het uiteindelijke resultaat en ben er van overtuigd dat eMagiz hun klanten hiermee een schaalbare en dynamisch aanpasbare rate limiting oplossing kan aanbieden voor de API-gateway integraties.

8 Literatuurlijst

Bakker, E. (2022, 13 Juni). *EMagiz API Gateway - eMagiz*. Geraadpleegd op 8 December 2023, van https://docs.emagiz.com/bin/view/Main/eMagiz%20Academy/Fundamentals/fundamental-api-gateway-introduction

Bakker, E. (2023, 17 April). *EMagiZ Event Streaming - eMagiz*. Geraadpleegd op 8 Augustus 2023, van https://docs.emagiz.com/bin/view/Main/eMagiz%20Academy/Fundamentals/fundamental-event-streaming-introduction

Bucket4J 8.5.0 reference. (Z.d.). Geraadpleegd op 7 November 2023, van https://bucket4j.com/8.5.0/toc.html#why-configuration-replacement-is-not-trivial

Holcombe, J. (2023, 6 Juni). *API rate Limiting: The Ultimate guide*. Kinsta®. Geraadpleegd op 4 December 2023, van https://kinsta.com/knowledgebase/api-rate-limit/

Oltwater, S. (2023, 22 Augustus). *API Management - Integration without Boundaries - eMagiz*. eMagiz. Geraadpleegd op 4 September 2023, van https://emagiz.com/api-management/

Srivastava, V. (2023, 23 Februari). *API rate limiting vs. API throttling: How are they different? | Nordic APIs |*. Nordic APIs. Geraadpleegd op 4 September 2023, van https://nordicapis.com/api-rate-limiting-vs-api-throttling-how-are-they-different/

Torken, E. (z.d.). *API Gateway - Introduction - eMagiz*. Geraadpleegd op 4 September 2023, van https://docs.emagiz.com/bin/view/Main/eMagiz%20Academy/Microlearnings/Crash%20Course/Crash%20Course%20API%20Gateway/crashcourse-api-gateway-introduction

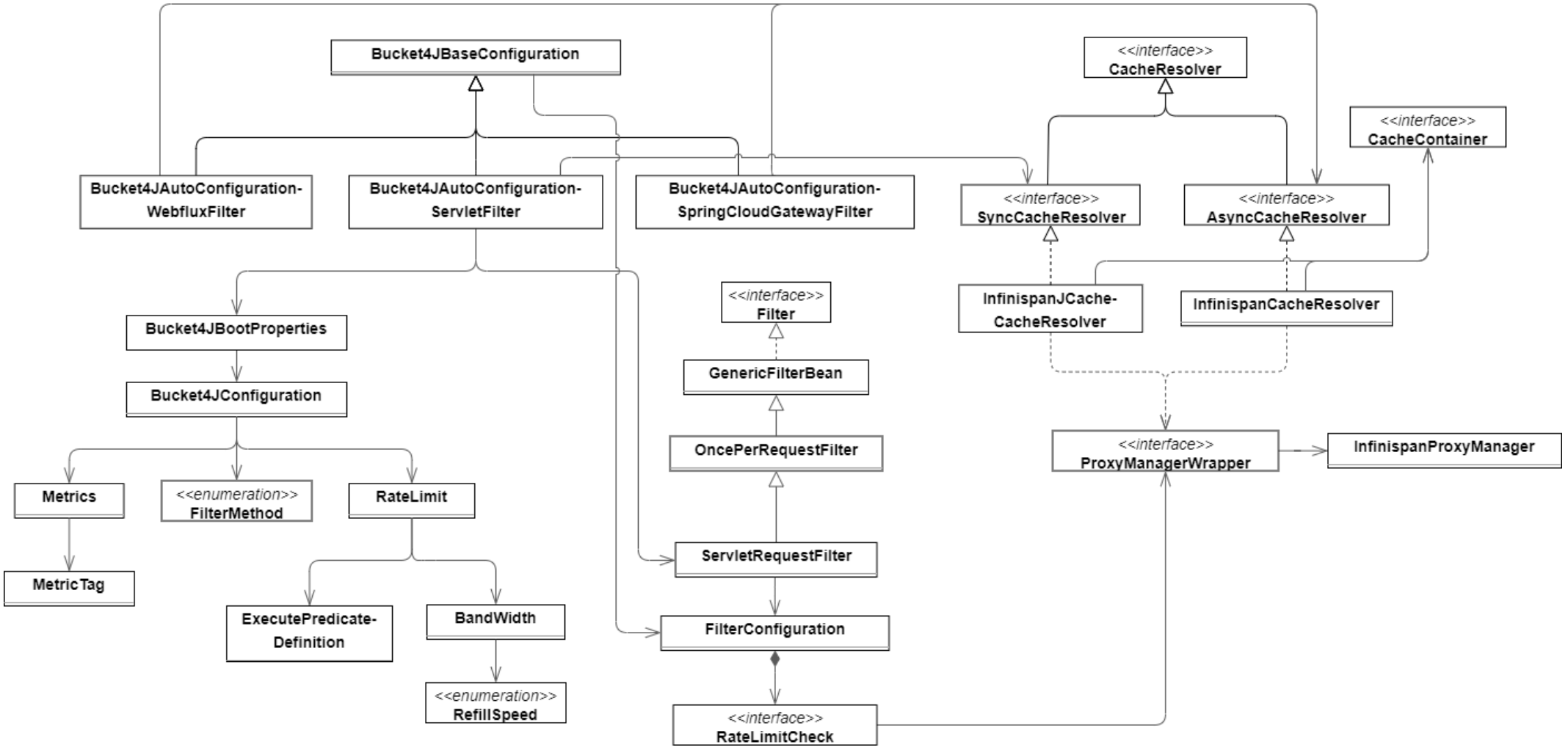
Van Lier, W. (2023, 12 Juli). *Stakeholdermanagement in projecten met scrum*. Agile Scrum Group. Geraadpleegd op 20 September 2023, van https://agilescrumgroup.nl/stakeholder-management-matrix-model/

Vogel, J. (z.d.). *ICT Research Methods — Methods Pack for research in ICT*. ICT Research Methods. Geraadpleegd op 7 December 2023, van https://ictresearchmethods.nl/dot-framework/

Vregelaar, T. T. (2023, 29 September). *Het ontstaan van eMagiz Enterprise IPAAS*. eMagiz. Geraadpleegd op 22 November 2023, van https://emagiz.com/ontstaan/

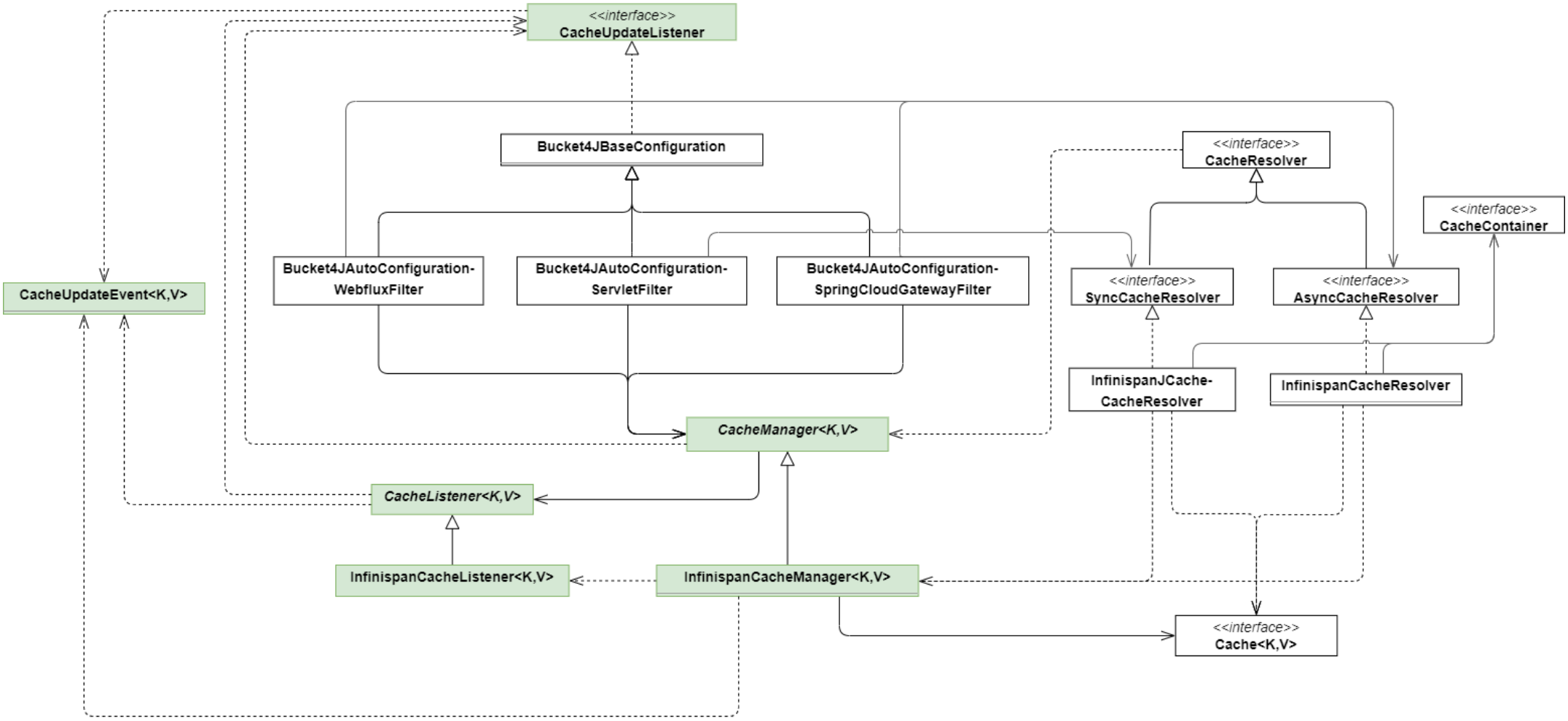
What is event streaming? A deep dive. (z.d.). Ably Realtime. Geraadpleegd op 20 November 2023, van https://ably.com/topic/event-streaming

Bijlage 1: Class diagram oorspronkelijke situatie



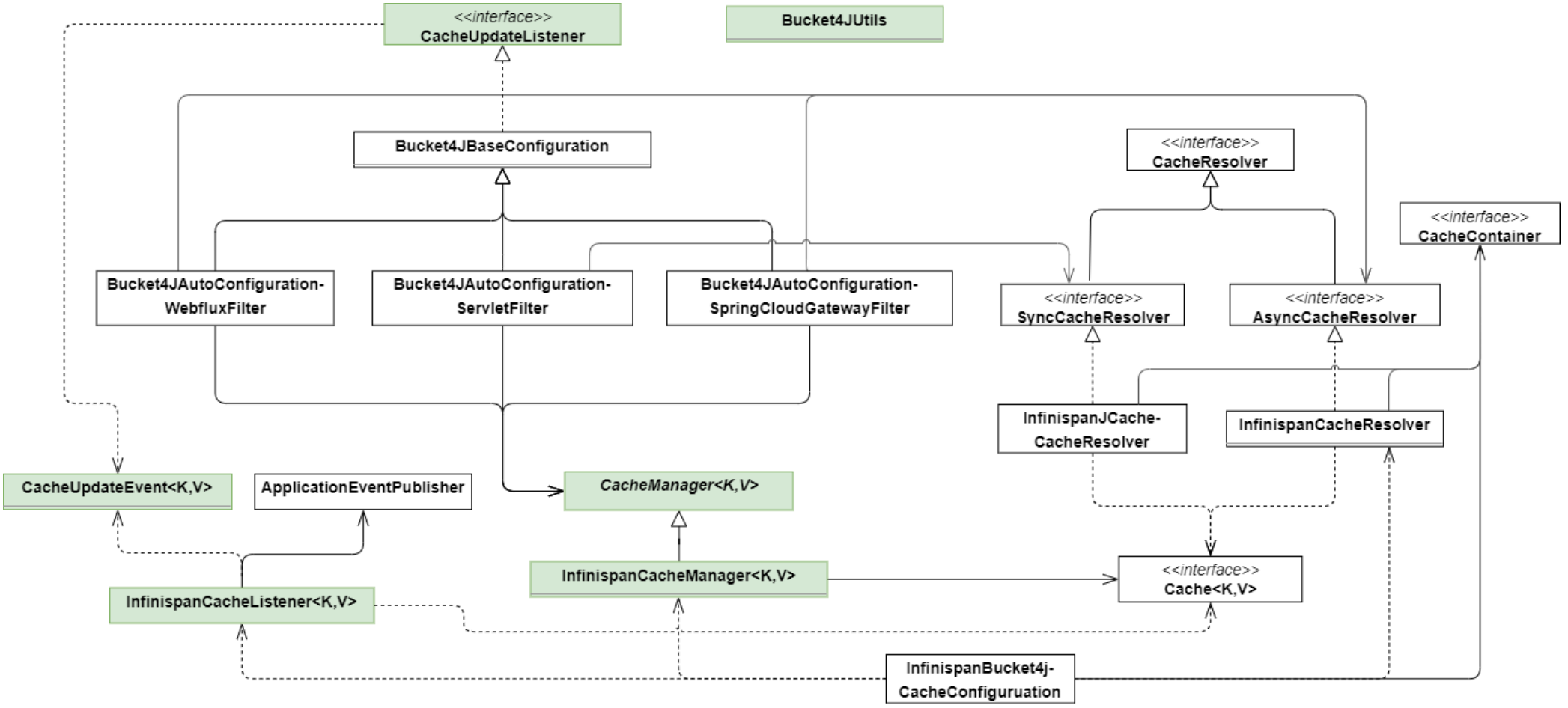
Figuur 9 Class diagram Bucket4J-spring-boot-starter (Sterk vereenvoudigd)

Bijlage 2: Class diagram nieuwe situatie



Figuur 10 Class diagram nieuwe situatie

Bijlage 3: Class diagram uiteindelijke resultaat



Figuur 11 Class diagram eindresultaat

Bijlage 4: Overige figuren en diagrammen

```
bucket4j.filter-config-caching-enabled=true
bucket4j.filter-config-cache-name=filterConfig
bucket4j.filters[0].id=filter1
bucket4j.filters[0].major-version=4
bucket4j.filters[0].minor-version=8
bucket4j.filters[0].rate-limits[0].tokens-inheritance-strategy=reset
bucket4j.filters[0].rate-limits[0].bandwidths[0].id=bandwidth1
```

Figuur 12 Nieuw toegevoegde application.properties