|  |  |
| --- | --- |
| Versie | 1.02 |
| Datum | 24 nov 2020 |
| Titel | Beschrijving API gateway |
| Auteur(s) | Zino Onomiwo, Henk Smit |

# Technisch Ontwerp van de API-gateway:

Nog te beschrijven volgens de laatste inzichten

* Endpoint(s) (http) en de bijbehorende deelschema’s per informatieservice
* Microservice per endpoint
* Tool om deelschema’s te genereren vanuit configuratiebestand(en)
* Beschrijving configuratiebestanden voor informatieservice versus bericht/gegeven (Excel sheet) opbouw
* Use-case verwerkt request voor gegeven informatieservice
* Leveringsvoorwaarden
  + Waar komen ze terecht
  + Wat moet het doen
  + Distributie
  + Hoe leggen wij dit vast: Configuratie, Database

# Doel en Verwerking

De API-gateway is een microservice om een GraphQL aanvraag te ontvangen, te verwerken en een correct antwoord in JSON-formaat terug te sturen.

Het TWI infomatie platfom bevat meerdere informatie-services. Elke informatie-service is gekoppeld aan een eigen API-gateway.

Een aanvraag wordt ontvangen via één enkel GraphQL HTTP endpoint. Dit endpoint verwacht een POST request met correcte headers en een body. Er worden twee headers verwacht. De eerste header specifieert de afnemer met als sleutel: “oin” en een string als waarde. De tweede header specifieert om welke information service het gaat met als sleutel: “service” en een identificatie string in UTF-8 als waarde.

Anders dan de headers, verwacht het HTTP endpoint een aanvraag zoals in de GraphQL specificatie beschreven is. Zie <https://graphql.org/learn/serving-over-http/#post-request>.

Nadat een aanvraag is ontvangen zal het verwerkt moeten worden om de vraag om gegevens te beantwoorden. Dit wordt gedaan met behulp van de datafetchers binnen de API-gateway. De datafetchers is een object, met als doel het ophalen gegevens uit de SUWI-broker. Elke afzonderlijke datafetcher is beperkt tot het opvragen van één enkel soort SUWI bericht. De datafetcher vraagt de gegevens op van een Translator microservice die het bericht van de broker opvraagt. De communicatie tussen de datafetcher en de Translator is geabstraheerd in een ServiceStub object, dit object neemt als input de parameters die de broker verwacht in een aanvraag.

Met het oog op de toekomst en kernwaarden van BKWI met betrekking tot hergebruik en flexibiliteit, is besloten om Translator microservices te bouwen. Dit geeft ontwikkelaars de mogelijkheid geeft om SUWI-berichten op te vragen met het gRPC protocol, binnen het TWI-project maar ook daarbuiten. Een bevraging met gRPC aan de Translator wordt doorverbonden als SOAP aanvraag aan de broker. Vervolgens wordt het antwoord van de broker, in SOAP formaat, in de Translator omgezet tot een gRPC antwoord.

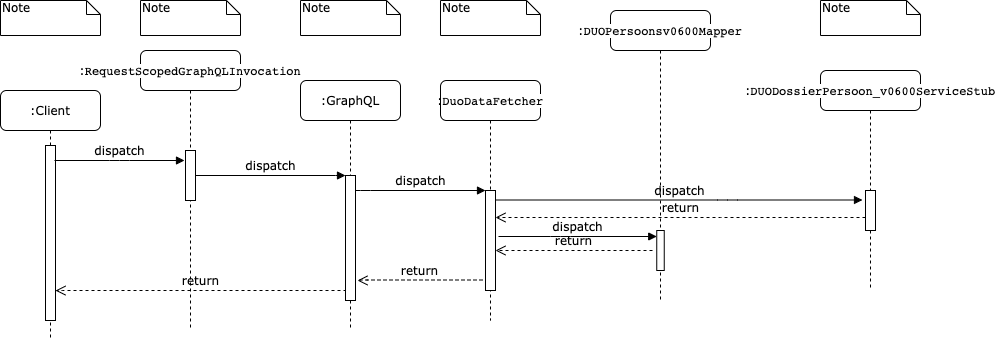
Datafetchers zijn niet afhankelijk van elkander. Als een aanvraag meerdere datafetchers gebruikt, dan kunnen deze in parallel draaien zonder bijkomstigheden zoals race-conditions.

Nadat een datafetchers voltooid is met het ophalen van diens gegevens worden de gegevens vertaald vanuit het gRPC protocol, in protobuf, naar een POJO dat gebruikt kan worden in de GraphQL-engine. Deze vertaalslag gebeurt met mapper-functies die gegenereerd worden met behulp van MapStruct. De uiteindelijk vertaalde gegevens worden verzameld door een GraphQL-engine dat de opgevraagde gegevens in een JSON-object verzameld en teruggestuurd als antwoord aan de aanvrager.

De vertaalslag moet worden uitgevoerd omdat GraphQL objecten en Protobuf objecten een andere benaming strategie aanhouden met betrekking tot lijsten.

De GraphQL Engine gebruikt java introspectie en verwacht dat een boom van objecten dezelfde velden hebben zoals gespecifieerd in het GraphQL schema. De Protobuf objecten hebben niet dezelfde benaming van velden als het schema.

De tooling waarmee de Protobuf klasses worden gegenereerd plakt een ‘-list’ suffix aan velden die een lijst type hebben. Als deze ‘-list’ suffix niet toegevoegd werd, dan had er geen extra vertaalslag hoeven plaatsvinden, maar de optie om dit uit te schakelen is er niet.



De afbeelding geeft een flow van aangeroepen methodes binnen het API-gateway project.

1. De client verzoekt de API-gateway om informatie, hierin geeft het parameters mee zoals het burgerservicenr.
2. De API-gateway ontvang een aanvraag van de Client in de RequestScopedGraphQLInvocation klasse.
3. De aanvraag wordt doorgegeven aan de GraphQL engine
4. De GraphQL engine valideert de GraphQL aanvraag op syntax
5. De GraphQL schakelt benodigde datafetchers in.
6. De datafetcher haalt gegevens op van de broker met behulp van de ServiceStub, hierin worden de parameters van stap 1 meegegeven.
7. De gegevens wordt ontvangen in protobuf-formaat
8. De gegevens wordt vertaald naar een formaat dat door GraphQL object vertaald kan worden.
9. De GraphQL verzameld de gegevens uit de datafetchers en verstuurd een antwoord naar de Client in JSON-formaat
10. De Client ontvangt het antwoord op zijn aanvraag.

# Eisen aan het schema

Bij een aanvraag moet de GraphQL query conform een schema zijn. Dit schema laat zien welke gegevens beschikbaar zijn om op te vragen. Hieronder wordt een voorbeeld van een schema vertoond, aangepast voor leesbaarheid.

|  |
| --- |
| type Query {  info(bsn: String!): InfoResponse!  }  type InfoResponse {  AanvraagPersoonResponse: **AanvraagPersoonResponse!**  BijstandsregelingenInfoResponse: **BijstandsregelingenInfoResponse!**  DUOPersoonsInfoResponse: **DUOPersoonsInfoResponse!**  } |

Het Query type is het beginpunt van een query. Hierbinnen staan de verschillende operaties die uitgevoerd kunnen worden, in het voorbeeld hierboven is er maar een. Het ontwikkelaarsteam heeft besloten om maar één operatie te gebruiken om zo conformiteit met betrekking tot het Burgerservicenummer af te dwingen.

Binnen het type InfoResponse staan meerdere velden, deze velden verwijzen naar types die gebaseerd zijn op een SUWI-bericht. BKWI probeert deze types, waar mogelijk, overeen te laten komen met de types zoals beschreven in de WSDL-definities. Deze types worden gegenereerd met behulp van BKWI-tooling en verdere configuratie voor leveringsvoorwaarden.

Binnen het team is besloten om voor elke informatie-service een eigen API-gateway project te bouwen. Dit betekent dat al deze projecten hun eigen GraphQL schema krijgen. Hierin staan alleen de gegevens die de informatie-service nodig heeft.

Binnen de WSDL-definities kunnen verschillende types dezelfde naamgeving hebben maar verschillende gegevens bevatten. Dit wordt opgelost door voor botsende namen een prefix te zetten.

Nadat berichten voor een GraphQL aanvraag zijn opgehaald, dan wordt een antwoord opgesteld met de opgevraagde berichten in JSON-formaat zoals van GraphQL verwacht kan worden. Met deze berichten kan ook een FWI-blok worden meegestuurd. Deze FWI-blokken bevatten een lijst van fouten, waarschuwingen of andere informatie wat vertaald zal worden naar een lijst dat meegegeven wordt met het JSON-antwoord. De lijsten voor alle verschillende SUWI-berichten worden samengevoegd in de GraphQL engine.

In diezelfde lijst worden ook alle overige fouten opgenomen en beschreven, dit zijn de fouten die voorkomen tijdens het gehele proces van ontvangen, verwerken en beantwoorden.

#### Fout codes:

Foutcodes volgens SuwiML transactiestandaad 4.0

<https://www.bkwi.nl/media/ex2lepsh/suwiml-transactiestandaard-versie-4-0.pdf>

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Omschrijving | Categorie | Toelichting |
| 0001 | Invalide soap envelope | 1 | Voldoet niet aan verwachte syntax. Structuur van de envelope matcht niet met wat er verwacht wordt |
| 0002 | Niet geautoriseerd | 3 | Service niet beschikbaar (QoS). Door gebrek aan bevoegdheden. |
| 0003 | Invalide soapaction | 1 | De inhoud leidt niet tot een voltooide actie, is niet gedefinieerd of onbegrijpelijk: Protocol fout. |
| 0004 | Niet conform xsd | 1 | Voldoet niet aan verwachte syntax |
| 0005 | WS-Addressing header “to” ontbreekt | 1 | Ontbreekt of voldoet niet aan verwachte syntax |
| 0006 | WS-Addressing header “action” ontbreekt | 1 | Ontbreekt of voldoet niet aan verwachte syntax |
| 0007 | WS-Addressing header “messageID” ontbreekt | 1 | Ontbreekt of voldoet niet aan verwachte syntax |
| 0008 | WS-Addressing header “relates to” ontbreekt | 1 | Ontbreekt of voldoet niet aan verwachte syntax |
| 0009 | Niet volgens UTF | 1 | Voldoet niet aan verwachte characterset |
| 0010 | Headers anders dan WSA-headers | 1 | Voldoet niet aan verwachte syntax |
| 0011 | Headers andere waarde dan voorgeschreven | 1 | Voldoet niet aan verwachte spec/waarde |
| 0051 | Service niet beschikbaar | 3 | Service niet beschikbaar (QoS). Door gebrek aan resource en/of verwerkingscapaciteit. |

|  |
| --- |
| {  “data”: { … },  “errors”: [ … ]  } |

# Schema generatie

De API-gateway maakt gebruik van een gegenereerd GraphQL-schema en gegenereerde POJO’s op basis van dit schema. Deze worden gegenereerd door tools gemaakt binnen BKWI, deze tools zijn te vinden binnen het bind-generator project en de bind-generator-maven-plugin.

De generator krijgt opgegeven welke SUWI-berichten binnen een gecombineerd schema moeten worden opgenomen. Dit levert een schema op met meerdere berichten die één-op-één zijn afgeleid uit hun WSDL-definities. Om de types van verschillende WSDL-definities uit elkaar te houden wordt het XML-pad als prefix aan de naam toegevoegd. Dit vermindert het aantal botsingen tussen de benamingen. Dit levert echter namen op als: ‘*UWVWbDossierPersoonGSDV0200UwvWbPersoonsInfoResponseClientSuwiAtlRelatieDienstverleningUwvWbKnelpuntKlantsituatieOplossingKnelpuntKlantsituatie*’.

***Configuratie van leveringsvoorwaarden***

# gRPC generatie

De API-gateway communiceert met Translator projecten via gRPC standaarden genaamd protobuf. Deze standaarden worden gegenereerd met behulp van BKWI-tooling en open-source tooling (<https://gitlab.bkwi.nl/twi-fase4/utilities/protobuf-maven-plugin>).

Net zoals de GraphQL-schema, zijn de Protobuf schemas één-op-één vertaald uit de WSDL-definities. Hierdoor komen de GraphQL en de Protobuf overeen qua structuur.

# Translatie tussen Protobuf en GraphQL

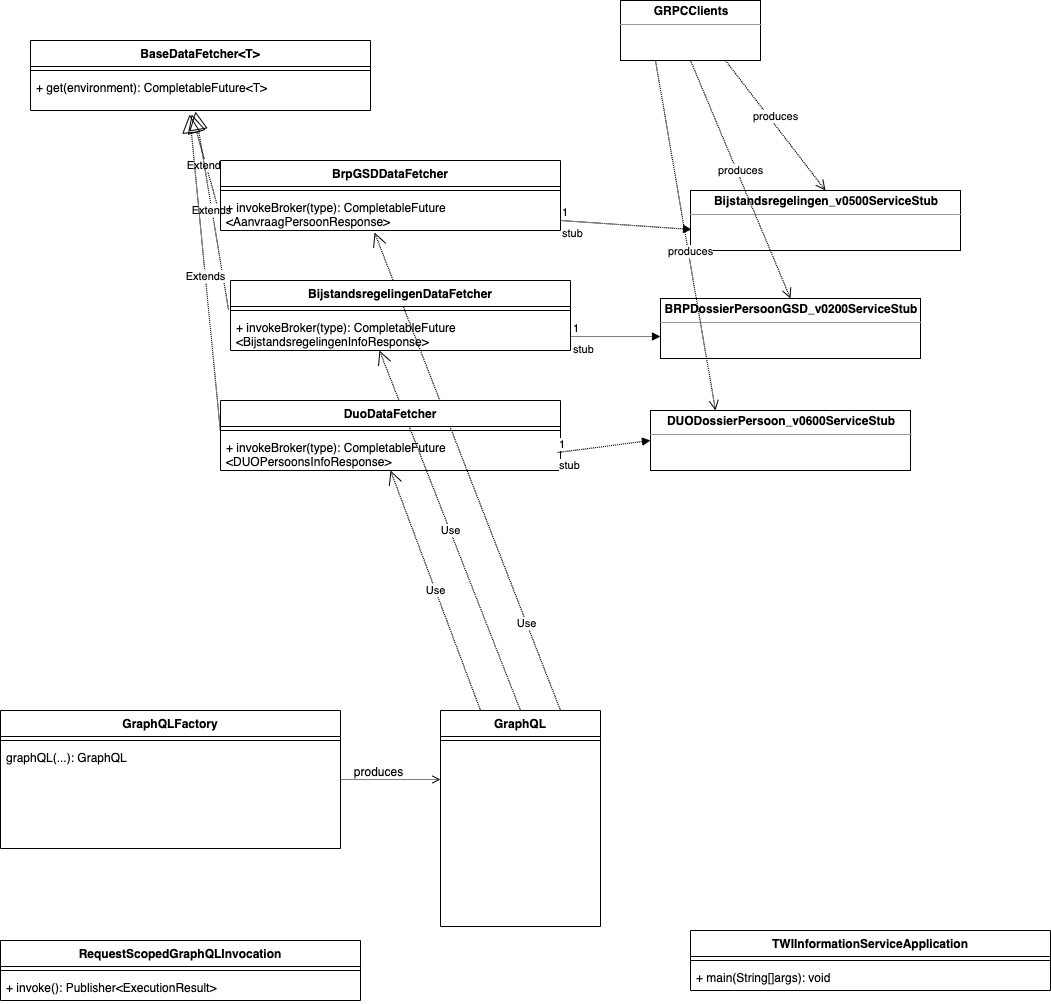
De translatie tussen Protobuf en GraphQL wordt gedaan met MapStruct (<https://mapstruct.org/>). MapStruct is een code generator dat methodes genereert voor de vertaling tussen twee verschillende types POJO’s. MapStruct maakt de veronderstelling dat beide types overeenkomen in velden, de benaming van die velden, en dat diens types ook recursief overeenkomen.

De GraphQL en de Protobuf POJO’s representeren dezelfde gegevens, maar met andere benamingen van velden en types. De benaming van types zijn onbelangrijk want MapStruct zoekt op namen van velden. Zoals eerder vermeld komt het merendeel van de velden in de GraphQL en Protobuf POJO’s overeen, met uitzonderingen van lijsten waar Protobuf POJO’s een ‘-list’ suffix hebben. Deze uitzonderingen worden niet automatisch door MapStruct ontdekt. De API-gateway maakt gebruik van een eigen ‘AccessorNamingStrategy’ voor MapStruct (<https://gitlab.bkwi.nl/twi-fase4/utilities/mapstruct-list-accessor-naming-strategy>). Dit zorgt ervoor dat velden in de GraphQL POJO’s worden gepaard met velden in de Protobuf POJO’s die een ‘-list’ suffix bevatten.

# Architectuur

De API-gateway maakt gebruik van het Micronaut framework. Dit framework levert functionaliteit voor het creëren van microservices. Met behulp van een GraphQL library wordt een controller gemaakt dat GraphQL aanvragen kan opvangen en asynchroon afhandelen.

Bij het opstarten van de API-gateway moet het geheel geconfigureerd worden. Een GraphQL bean wordt opgebouwd in de GraphQLFactory: hier wordt het GraphQL-schema ingelezen en gekoppeld aan datafetchers. Een datafetcher per SUWI-bericht. De datafetchers bevatten een asynchroon kanaal naar de Translator service. De GrpcClients klasse produceert deze kanalen. De RequestScopedGraphQLInvocation ontvangt de GraphQL aanvraag en plaatst parameters uit de header in een GraphQL context object. De klasse TWIInformationServiceApplication is de klasse die de Micronaut applicatie opstart.





* RequestScopedGraphQLInvocation
  + Vangt GraphQL aanvragen op
  + Bevat een GraphQL object met GraphQL logica
* GraphQLFactory
  + Creëert een Singleton GraphQL object met configuratie voor het injectie-systeem van Micronaut
* TWIInformationServiceApplication
  + Start de micronaut application
* GraphQL
  + Het GraphQL object met configuratie
  + Gebruikt de drie datafetchers om aanvragen af te handelen
* BaseDataFetcher<T>
  + Implementeert generieke logica die gebruikt wordt door een concreet geïmplementeerd datafetcher voor een SUWI-bericht.
* BrpGSDDataFetcher
  + Een concrete implementatie van de BaseDataFetcher<T> voor het BrpDossierPersoonGSD bericht.
  + Maakt gebruik van BrpDossierPersoonGSD\_ServiceStub om met de Translator te praten
* BijstandDataFetcher + DuoDataFetcher
  + Zelfde als BrpGSDDataFetcher maar voor hun eigen berichten
* GRPCClients
  + Produceert ServiceStubs voor het injectie-systeem van Micronaut.
  + De service stubs bevatten configuratie over de URL en mogelijke gRPC acties.
* <T>ServiceStub
  + Gegenereerde service beschrijvingen
  + Abstraheert de gRPC communicatie tussen de API-gateway en de Translator

## Versiebeheer

Op het moment dat nieuwe versies uitgerold moeten worden, dan handelt Kubernetes dit af.

Versiebeheer met betrekking tot het GraphQL schema is nog ter discussie.

# Leveringsvoorwaarden

Leveringsvoorwaarden bekijkt per opgevraagd veld of de afnemer het mag inlezen.

Op het moment zijn er drie ideeën van implementatie voorgesteld:

* Met behulp van GraphQL instrumentatie
* Met behulp van microservices tussen de API-gateway en de translators, waarbij een teruggegeven antwoord wordt gefiltreerd.
* Een API-gateway per informatie-service met gespecialiseerde GraphQL-schema.

Er is interesse getoond in het gebruik van een autorisatie-framework en/of microservices om configuratie te verbergen voor de API-gateway.

Overigens is er ook een mogelijkheid om het TWI-platform de mogelijkheid te geven om leveringsvoorwaarden te laten bekijken, mits de bronnen hier mee akkoord gaan.

### GraphQL Instrumentation

Leveringsvoorwaarden met behulp van instrumentatie dat de GraphQL aanvraag evalueert.

<https://www.graphql-java.com/documentation/master/instrumentation/#field-validation-instrumentation>

De eerste optie krijgt een GraphQL aanvraag als input, evalueert deze, om vervolgens de aanvraag toe te stemmen of af te wijzen.

Een leveringsvoorwaarde microservice bekijkt de velden en afnemer en bepaald of de GraphQL aanvraag uitgevoerd mag worden. Als de aanvraag niet uitgevoerd mag worden zal er geen data als antwoord worden opgestuurd.

Het voordeel van deze aanpak is dat de leveringsvoorwaarden op één plek worden bekeken, en dus configuratie op één plek hoeft worden toegevoegd, en onbevoegde aanvragen worden tegengehouden nog voor Translators worden bevraagd.

### Intercepting microservices

De tweede optie plaatst microservices tussen de datafetchers en de translators. De microservice krijgt een gRPC aanvraag binnen, en bekijkt of de afnemer dit bericht mag bekijken. Als dat het geval.

De microservice speelt een aanvraag door naar de Translator, maar alleen als de oorspronkelijke afnemer bevoegd is. Ook speelt de microservice het antwoord door, maar haalt eerst gegevens weg die de afnemer niet mag inzien.

Het voordeel van deze aanpak is… ?

### Een API-gateway project per informatie-service

De laatste optie introduceert per informatie-service een API-gateway. Een API-gateway gebruikt dan een kleiner schema dat enkel toegestane berichtelementen bevat. Deze optie heeft momenteel de voorkeur.

Het voorstel luidt als volgt:

* Een API-gateway per informatie-service met een schema bepekt tot de gegevens die daadwerkelijk bij die informatieservice horen.
  + Voordeel: helderheid naar de afnemer, voor een deel zit de leveringscheck dan al in dat schema.
  + Voordeel: kleinere en overzichtelijkere API’s.
  + Voordeel: geen extra software (service) nodig om de te leveren informatie per informatieservice nog eens te controleren.
  + Tijdens het uitrollen ligt maar één informatie-service stil, i.p.v. alle services tegelijk.
* Het schema voor een API-gateway wordt gegenereerd vanuit configuratie van de informatie-service.
* De microservice die de aanvragen verwerkt (1 microservice per informatie-service) wordt opgestart met het configuratiebestand voor die informatieservice. Deze microservices zijn qua code exact hetzelfde. Ze maken allen gebruik van hetzelfde "achterland", alleen zal de ene informatieservice bepaalde Translators wel gebruiken en andere informatieservices niet.
* Geen versienummers en misschien geen berichtnamen in de schema’s.
  + Als nieuwe berichten of nieuwe bronnen bij de schema’s komen, worden bestaande services niet of weinig geraakt.
  + De afnemer merkt niets van nieuwe berichtversies omdat die er niets van merkt. Alleen de translator wordt uitgebreid voor nieuw toegevoegde velden. Als velden verdwijnen of anders gaan heten (komt nauwelijks voor) die de afnemer gebruikt zijn er gevolgen voor de afnemer.
  + Botsingen met namen worden anders opgelost
  + Alleen als de informatiebehoefte van een informatie-service verandert moet er iets veranderen in de API.

# Technische appendix

## Git

Alle projecten gebruikt binnen het TWI-team van BKWI zijn te vinden op Gitlab <https://gitlab.bkwi.nl/twi-fase4>, en kunnen gebouwd worden met behulp van het Maven 3+ bouw systeem.

De API-gateway en Translators overlappen in het gebruik van Protobuf definities. Daarom zijn de Protobuf definities en de gegenereerde POJO-klassen geïsoleerd als een library en in aparte projecten gezet. Elke SUWI-bericht heeft zijn eigen Protobuf project.

## Omgevingen

Momenteel zijn geen ontwikkelaars omgevingen beschikbaar en moet alles lokaal gedraaid worden. De uitvoer hiervan ligt bij Schuberg Philis, de leverancier van de omgevingen, en de aanvraag loopt via Joost Huijbregts (BKWI). De ontwikkel en test omgeving wordt in sprint van 16/11 – 30/11 opgeleverd. De acceptatie omgeving de sprint daarna (30/11 – 14/12) en de productie omgeving de sprint daarna (14/12 – 28/12).

Uiteindelijk komt er een CI/CD pipeline in gitLab (gitLab CI) die naar de verschillende omgevingen deployt. De release gaat naar de volgende omgeving als de testen in de pipeline voor de huidige omgeving slagen. De deployment naar de productie omgeving wordt handmatig gedaan.

**Ontwikkelomgeving**: Een kubernetes omgeving zonder connectie naar buiten. Bedoeld om de deployment te testen en nieuwe features (of POCs) te proberen. Daarmee is het de meest instabiele omgeving.

**Test omgeving**: Een kubernetes omgeving zonder connectie naar buiten. Deze omgeving is bedoeld voor het ontwikkelteam om features te testen. Het team bepaald zelf welke features gedeployd worden en daarmee de stabiliteit van de omgeving.

**Acceptatie omgeving**: Een kubernetes omgeving zonder connectie naar buiten. Deze omgeving is bedoeld om de integratie tussen alle onderdelen van de keten te testen. Deployment zal moeten worden afgestemd met de andere gebruikers van deze omgeving om de stabiliteit te kunnen garanderen.

**Productie omgeving**: Een kubernetes omgeving met connectie naar het DigiNetwerk (geen connectie met het internet). Deze omgeving is bedoelt om de uiteindelijke dienst naar de afnemer te draaien.

De rechten voor toegang tot de verschillende omgevingen wordt door team techsupport (BKWI) ingeregeld. Hier is nog niet bekend welke rollen hier te onderscheiden zijn.

De uiteindelijke wordt binnen een kubernetes omgeving gedraaid. De services worden geconfigureerd met <https://Istio.io>.

Istio regelt veilige communicatie en het monitoren binnen de omgeving. Ook zorgt het voor correcte headers om te communiceren met de broker.

## Draaien van API-gateway

Om zinnige data van de API-gateway terug te krijgen moeten de translators en een lokale broker draaien. De gegevens die de broker terug kan geven kunnen worden gevonden binnen het SAS-project.

* De API-gateway wordt gestart met de commando
  + ‘mvn clean install’
  + ‘mvn mn:run’
* De translators kunnen worden gestart met de volgende commando’s. Na een aanpassing binnen de configuratie moet de translator opnieuw geinstalleerd worden.
  + ‘mvn clean install’
  + ‘mvn exec:exec’
* De docker broker kan gestart worden met
  + ‘d -dbu’