THOMAS MORE

Ontwikkeling van een REST API

Realisatiedocument

Nicolas Van Dyck
Student Bachelor in de Toegepaste Informatica – Applicatieontwikkeling

2024 - 2025

Voorwoord

Dit document maakt deel uit van de stageopdracht die werd uitgevoerd in het kader van het afsluitende traject van de bacheloropleiding Toegepaste Informatica, met een specialisatie in applicatieontwikkeling.

De stage vond plaats bij Pension Architects, waar de opdracht bestond uit het verder ontwikkelen van een bestaande REST API zodat deze in de toekomst effectief in gebruik kan worden genomen.

Wat mij in het bijzonder aansprak in deze stageopdracht, was dat de backend – en dus ook de API – ontwikkeld is in Java, een taal die ik tijdens mijn opleiding heb leren waarderen. Daarnaast bood de opdracht ook de kans om te werken met technologieën en ontwikkelstrategieën zoals AWS, Docker en infrastructure-as-code. Dit zijn tools waar ik in mijn vrije tijd al mee geëxperimenteerd had, maar die binnen de opleiding niet of nauwelijks aan bod kwamen. Toch zijn het essentiële vaardigheden voor een moderne developer, en deze stage gaf mij de mogelijkheid om ze verder te verkennen en in de praktijk toe te passen.

Woord van dank

Graag wil ik van deze gelegenheid gebruik maken om enkele mensen te bedanken die mij tijdens mijn stage en het schrijven van deze bachelorproef hebben ondersteund.

In de eerste plaats wil ik mijn stagementor bedanken voor de begeleiding, het waardevolle advies en de feedback gedurende de hele stageperiode. Ook het volledige team van Pension Architects verdient mijn oprechte dank voor de aangename samenwerking en de bereidheid om kennis en ervaring te delen.

Daarnaast ben ik bijzonder dankbaar tegenover Ivan, de eigenaar van Pension Architects, voor het geven van deze kans. Het is niet vanzelfsprekend om als werktrajectstudent een stageplaats te vinden, en deze opportuniteit betekende dan ook veel voor mij.

Ook mijn familie wil ik bedanken voor hun onvoorwaardelijke steun tijdens de vier jaar van mijn opleiding. Hun aanmoediging en begrip zijn van onschatbare waarde geweest.

Tot slot wil ik de school en alle docenten bedanken voor hun begeleiding en inzet doorheen de opleiding Toegepaste Informatica. Dankzij hun kennis en toewijding heb ik de kans gekregen om mezelf zowel professioneel als persoonlijk te ontwikkelen.

Inhoudsopgave

1. Inleiding

Voorafgaand aan de uitwerking van deze opdracht werd een Plan van Aanpak opgesteld. Hierin wordt de opdracht in meer detail toegelicht, inclusief de reden waarom een API noodzakelijk is voor het bedrijf en wie de beoogde gebruikers zijn. Daarnaast worden ook de scope van de opdracht en de bijhorende planning uiteengezet.

Om deze scope en planning te kunnen bepalen, werd eerst onderzocht welke onderdelen vereist zijn voor een productieklare API. Op basis van deze analyse volgde een tweede, meer gedetailleerd onderzoek naar de ontbrekende onderdelen die we wilden implementeren. De bevindingen resulteerden in een reeks taken die werden gegroepeerd in tickets, telkens met een inschatting van de benodigde tijd.

Tijdens de stage evolueerden zowel de scope als de planning — een logisch gevolg binnen een professionele omgeving. Aangezien de API gekoppeld is aan meerdere systemen, kwamen er geregeld nieuwe taken bij die initieel buiten scope of niet ingepland waren.

In het hoofdstuk *Analyse* worden de resultaten van de voorbereidende onderzoeken besproken, die de basis vormden voor het bepalen van de scope en planning. De daaropvolgende hoofdstukken bevatten een gedetailleerde beschrijving van elk ticket, eventueel aangevuld met gerichte onderzoeken en relevante codefragmenten. Zo wordt elk aspect van de opdracht behandeld, van het eerste onderzoek en de planning tot aan de implementatie van de verbeteringen binnen de API.

2. Analyse

2.1. Inleiding

In dit hoofdstuk worden de onderzoeken van de gehele API beschreven.
Zoals eerder vermeld in dit document is er een eerste onderzoek dat een analyse maakt van een productieklare API. Op basis van dit eerste onderzoek is er dan besproken welke onderdelen we tijdens de stageopdracht graag zouden implementeren. De onderdelen die we graag zouden implementeren zijn dan in een meer gedetailleerd onderzoek onderzocht en op basis van dat tweede onderzoek is dan de scope en planning bepaald alsook zijn er enkele tickets met een geschatte tijdsduur opgesteld.

2.2. Verkennend onderzoek

In dit eerste onderzoek som ik een aantal punten op die essentieel zijn voor een productieklare API. Er wordt ook kort een beschrijving gegeven waarom deze punten essentieel zijn voor een productieklare API.

2.2.1. Essentiele punten voor een productieklare API (Obregon, 2023)

2.2.1.1. Documentatie

De API-documentatie wordt momenteel ondersteund door Swagger. Verbeteringen kunnen worden doorgevoerd door:

- Gebruik te maken van uitgebreide annotaties.
- Beschrijvingen te verbeteren voor een betere leesbaarheid en bruikbaarheid.

2.2.1.2. Security

De API maakt gebruik van Spring Security voor authenticatie en autorisatie. De volgende aspecten vereisen verdere optimalisatie:

- **Authenticatie en autorisatie:** Overstappen van Basic Auth naar JWT-tokens kan extra beveiliging bieden.
- **Rate Limits en Quota:** Implementeren van rate limiting om misbruik van de API te voorkomen.

2.2.1.3. Caching Strategie

Caching verbetert de responstijd en verlaagt de belasting op de backend:

- **In-memory caching:** Momenteel wordt Ehcache gebruikt in het adminpanel. Caffeine biedt mogelijk betere prestaties.
- **Distributed caching:** Implementatie van Redis of Memcached voor schaalbaarheid.
- **HTTP-level caching:** Gebruik van **Etag** of **AWS CloudFront** om netwerkverkeer te verminderen.
- **Database-level caching:** Gebruik van Redis of Spring Cache met Hibernate caching.

2.2.1.4. Observability

Een robuust observability systeem is essentieel.
Observability is gebaseerd op 3 pijlers loggen, meten en traceren:

- **Monitoring:** Loggen van HTTP-methode, URI, statuscode, inhoudstype en payload.
- **Error Handling:** Foutenlogboeken met type, bericht, stack trace en timestamp.
- **Performance Metrics:** Metingen van responstijd, latentie, doorvoer en beschikbaarheid.
- **Security Metrics:** Monitoring van authenticatie, autorisatie, encryptie en rate limiting.
- **Gebruik Metrics:** Aantal verzoeken, gebruikers en sessies.
- **Visualisatie:** JSON-logformaat gebruiken voor compatibiliteit met **AWS Cloudwatch**.

2.2.1.5. Deployment Strategie

Een efficiënte deployment-strategie is cruciaal voor een stabiele API:

- **CI/CD Pipeline:** Evalueren of deze volledig geautomatiseerd is en waar verbeteringen nodig zijn.
- **Canary Deployment:** Gefaseerde uitrol om risico's te minimaliseren.
- **Feature Flags:** Mogelijkheid om functionaliteiten geleidelijk te activeren zonder volledige redeploy.

2.2.1.6. Testing

Om de kwaliteit en betrouwbaarheid van de API te waarborgen, is een gedegen teststrategie noodzakelijk. Verschillende testmethodes kunnen worden toegepast:

- **Unit Testing:** Gebruik van Mockito en JUnit om kritieke functionaliteiten te testen en af te dekken.
- **Integration Testing:** Testcontainers en WireMock kunnen worden ingezet om integraties met externe componenten te valideren.
- **Load Testing:** Tools zoals JMeter en Gatling helpen om prestaties en schaalbaarheid te testen, gebaseerd op verwachte gebruikersaantallen en belasting.

2.2.1.7. Versiebeheer

Een solide versiebeheerstrategie voorkomt brekende wijzigingen en maakt compatibiliteit mogelijk. Mogelijke methoden:

- URI-based versioning (bv. /v1/resource)
- Parameter-based versioning (bv. ?version=1)
- Header-based versioning (bv. Accept: application/vnd.myAPI.v1+json)

2.2.1.8. Microservices en API Gateway

Overwegen van een API Gateway kan de architectuur vereenvoudigen en beveiligen:

- Vermindert het aantal directe endpoints.
- Centraliseert authenticatie en autorisatie.

2.2.2. Conclusie

De API beschikt al over een sterke basis, maar er zijn nog verbeterpunten op het gebied van security, versiebeheer en logging/monitoring. Deze bevindingen werden voorgelegd aan mijn stagementor en op basis hiervan werd er beslist welke van deze onderdelen we verder zullen uitwerken.

2.3. Gedetailleerd onderzoek

In dit onderzoek worden drie belangrijke onderdelen van een moderne, productieklare API in detail onderzocht: logging, testing en rate limiting. Deze topics werden geselecteerd in overleg met de stagiair en de stagementor, op basis van hun relevantie en meerwaarde voor het project. Voor elk onderdeel wordt toegelicht waarom het belangrijk is voor de API, worden meerdere geschikte libraries besproken, en wordt per library een korte WRM (waarom-rationale-matrix) opgesteld om een gefundeerde vergelijking te kunnen maken. Waar relevant, worden codefragmenten toegevoegd ter illustratie. Elk hoofdstuk eindigt met een conclusie waarin de voorkeur wordt gemotiveerd en uitgelegd hoe de gekozen oplossing zou kunnen worden geïntegreerd in de bestaande codebase.

2.3.1. Logging

Voor het Loggen ligt onze keuze al vast op Log4J. Dit framework wordt nu al gebruikt en geniet daarom onze voorkeur.

Ik ga nog wel een aantal frameworks vergelijken, dit kan handig zijn in het geval dat er toch ooit beslist zou worden om van framework te wisselen.

De volgende frameworks zullen verder onder de loep worden genomen. Java util logging (JUL), LOG4J, TinyLOG en logback

Bij elk framework zal er dus ook een WRM gemaakt worden op basis van de volgende criteria die ik belangrijk vind.

- Bestaande implementatie:
  Dit criteria spreekt voor zich. Wordt het framework al gebruikt of niet?

- Future proof (actief onderhouden):
  Wordt het framework onderhouden en is het dus ook toekomstgericht een goede keuze. Omdat de API cruciaal is en in een productieomgeving draait is het wel van belang dat het framework ook in de toekomst ondersteund wordt.

- Performantie:
  Hoeveel impact heeft het framework op de applicatie.

- Integratie met AWS:
  Kan het framework worden geïntegreerd in een AWS-omgeving? Omdat heel de app gehost wordt in AWS is dit ook wel een punt om rekening mee te houden.

- Integratie met monitoring frameworks:
  Omdat we ook willen dat de API gemonitord wordt met een framework zou het mooi zijn als dit en het monitoring framework met elkaar kunnen samenwerken.

- Integratie met SLF4J:
  SLF4J is een façade voor verschillende logging frameworks. Het grootste voordeel hiervan is de flexibiliteit: door gebruik te maken van SLF4J kan er eenvoudig gewisseld worden van onderliggend logging framework zonder dat aanpassingen aan de bestaande applicatiecode nodig zijn. Hierdoor blijft de codebase overzichtelijk en toekomstbestendig.

2.3.1.1. Java util log

Java Util Logging (JUL) is het ingebouwde logging-framework van Java, beschikbaar sinds Java 1.4. Het stelt ontwikkelaars in staat om logberichten te genereren en te beheren zonder externe bibliotheken. Met JUL kunnen logs worden geclassificeerd op verschillende niveaus, zoals INFO, WARNING en SEVERE. Logs kunnen naar de console, bestanden of andere outputstromen worden gestuurd met behulp van handlers.
Een eenvoudig voorbeeld van JUL in gebruik:

```
1. import java.util.logging.Logger;
2.
3. public class LoggingVoorbeeld {
4.      private static final Logger logger = Logger.getLogger(LoggingVoorbeeld.class.getName());
5.
6.      public static void main(String[] args) {
7.               logger.info("Dit is een info-bericht");
8.               logger.warning("Dit is een waarschuwing");
9.              }
10. }
11.
```

Criterium	Gewicht	Score	Toelichting
Bestaande implementatie	15	10	Deze zit standaard in Java, maar wordt niet gebruikt.
Toekomstgericht	10	5	Technisch gezien wordt JUL geupdate en onderhouden omdat dit een standaard library is van java. Maar JUL zelf krijgt nauwelijks nieuwe features of updates sinds de release (2002)
Performance	5	0	JUL heeft zeer slechte performance als het gaat om grotere projecten.
Integratie met AWS	10	5	Er is integratie met AWS mogelijk via de AWS sdk bv. maar er is geen native koppeling beschikbaar

Integratie met monitoring framework	10	5	Ook hier is er geen native integratie beschikbaar met bv. micrometer.
Integratie met SLF4J	5	2	Er is integratie mogelijk met slf4j, maar deze wordt niet aangeraden aangezien er betere frameworks zijn voor slf4j
Totaal	**55**	**27**	**49%**

2.3.1.2. Log4j (Apache, 2025)

Log4j is een krachtig en uitbreidbaar logging framework van Apache waarmee applicaties logberichten kunnen vastleggen en naar verschillende bestemmingen kunnen sturen, zoals de console, bestanden of externe logsystemen. Het biedt ondersteuning voor verschillende logniveaus (DEBUG, INFO, WARN, ERROR) en maakt configuratie mogelijk via XML, JSON of YAML. Door de asynchrone loggingopties kan Log4j efficiënter omgaan met loggegevens zonder de prestaties van de applicatie te beïnvloeden.

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.22.1</version>
</dependency>

```

```
<Configuration status="WARN">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n"/>
        </Console>
    </Appenders>
    <Loggers>
        <Root level="info">
            <Appender-ref ref="Console"/>
        </Root>
    </Loggers>
</Configuration>

```

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Log4j2WithoutSLF4J {
    private static final Logger logger = LogManager.getLogger(Log4j2WithoutSLF4J.class);

    public static void main(String[] args) {
        logger.info("Dit is een informatiebericht");
        logger.warn("Dit is een waarschuwing");
        logger.error("Dit is een foutmelding");
    }
}

```

Criterium	Gewicht	Score	Toelichting
Bestaande implementatie	15	15	Wordt nu gebruikt in de ganse applicatie, en heeft daarom dus ook de voorkeur.
Toekomstgericht	10	10	Log4j krijgt regelmatig updates van de Apache Software Foundation. Doordat het framework ook een industriestandaard is de verwachting dan ook dat dit ook in de toekomst zo zal zijn.

Performance	5	5	Log4j kan asynchroon werken waardoor performantie geen probleem is.
Integratie met AWS	10	10	Log4j heeft een AWS Cloudwatch appender waardoor logs direct kunnen worden doorgestuurd naar AWS.
Integratie met monitoring framework	10	5	Er is geen native integratie mogelijk met micrometer. Er is wel een module beschikbaar.
Integratie met SLF4J	5	5	Er is integratie met SLF4J mogelijk.
Totaal	**55**	**50**	**91%**

2.3.1.3. TinyLog (Tinylog, 2025)

TinyLog is een lichtgewicht en eenvoudig loggingframework voor Java dat minder configuratie vereist dan Log4j of Logback. Het is ontworpen om minimale overhead te hebben en ondersteunt zowel synchronische als asynchronische logging. TinyLog heeft geen ingewikkelde configuratiebestanden nodig en kan eenvoudig via code of een eenvoudige properties-bestand worden ingesteld.

```
<dependency>
    <groupId>org.tinylog</groupId>
    <artifactId>tinylog-API</artifactId>
    <version>2.6.2</version>
</dependency>
<dependency>
    <groupId>org.tinylog</groupId>
    <artifactId>tinylog-impl</artifactId>
    <version>2.6.2</version>
</dependency>

```

```
tinylog.writer = console
tinylog.level = info
tinylog.format = {date:yyyy-MM-dd HH:mm:ss} [{level}] {class}.{method}(): {message}
```

```
import org.tinylog.Logger;

public class TinyLogExample {
    public static void main(String[] args) {
        Logger.info("Dit is een informatieve log");
        Logger.warn("Dit is een waarschuwing");
        Logger.error("Dit is een foutmelding met een uitzondering", new
RuntimeException("Fout!"));
    }
}
```

Criterium	Gewicht	Score	Toelichting
Bestaande implementatie	15	0	Tinylog wordt niet gebruikt.
Toekomstgericht	10	5	Wordt niet zo frequent geupdate als de andere frameworks. laatste versie dateert van 2023.

Performance	5	4	In kleinere toepassingen is Tinylog sneller dan logback of log4J. In grotere toepassingen is Tinylog trager.
Integratie met AWS	10	4	Er is geen officiële integratie met AWS.
Integratie met monitoring framework	10	4	Er is geen officiële integratie met micrometer. Ook niet met een module.
Integratie met SLF4J	5	5	SLF4J kan worden gebruikt als facade voor Tinylog.
Totaal	**55**	**22**	**40%**

2.3.1.4. Logback (Logback, 2025)

Logback is een krachtig en flexibel logging-framework voor Java-applicaties. Het wordt beschouwd als de opvolger van Log4j en is de standaard logging-implementatie voor SLF4J. Logback biedt betere prestaties, configuratiemogelijkheden via XML en Groovy, en ingebouwde ondersteuning voor asynchrone logging en RollingFile-appender.

```
1. <dependency>
2.     <groupId>ch.qos.logback</groupId>
3.     <artifactId>logback-classic</artifactId>
4.     <version>1.4.14</version>
5. </dependency>
6.
```

```
 1. <configuration>
 2.     <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
 3.         <encoder>
 4.             <pattern>%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} -
%msg%n</pattern>
 5.         </encoder>
 6.     </appender>
 7.
 8.     <root level="info">
 9.         <appender-ref ref="STDOUT"/>
10.     </root>
11. </configuration>
```

```
 1. import ch.qos.logback.classic.Logger;
 2. import ch.qos.logback.classic.LoggerContext;
 3. import ch.qos.logback.classic.joran.JoranConfigurator;
 4. import ch.qos.logback.core.util.StatusPrinter;
 5. import org.slf4j.impl.StaticLoggerBinder;
 6.
 7. public class LogbackWithoutSLF4J {
 8.     public static void main(String[] args) {
 9.         LoggerContext context = new LoggerContext();
10.         JoranConfigurator configurator = new JoranConfigurator();
11.         configurator.setContext(context);
12.
13.         try {
14.             configurator.doConfigure("logback.xml");
15.         } catch (Exception e) {
16.             e.printStackTrace();
17.         }
18.
```

```
19.         StatusPrinter.printInCaseOfErrorsOrWarnings(context);
20.         Logger logger = context.getLogger(LogbackWithoutSLF4J.class);
21.
22.         logger.info("Dit is een informatiebericht");
23.         logger.warn("Dit is een waarschuwing");
24.         logger.error("Dit is een foutmelding");
25.     }
26. }
27.
```

Criterium	Gewicht	Score	Toelichting
Bestaande implementatie	15	10	Wordt nu niet gebruikt. Zit wel standaard in Spring Boot.
Toekomstgericht	10	8	Wordt regelmatig geüpdatet door hetzelfde team als dat van SLF4J, maar niet zo frequent als Log4J 2
Performance	5	3	Heeft ook async logging zoals log4j, alleen is die iets minder in performance
Integratie met AWS	10	10	Logback heeft een AWS Cloudwatch appender waardoor logs direct kunnen worden doorgestuurd naar AWS.
Integratie met monitoring framework	10	5	Er is geen native integratie mogelijk met micrometer. Er is wel een module beschikbaar
Integratie met SLF4J	5	5	Er is integratie met SLF4J mogelijk.
Totaal	**55**	**41**	**74,5%**

2.3.1.5. Besluit

Het framework dat het beste uit de WRM's komt is Log4J. Al zou ik persoonlijk voor SLF4J kiezen als facade, dan ben je niet afhankelijk van 1 framework.

2.3.2. Observability

Tijdens de stage werd het oorspronkelijke onderzoek geüpdatet naar aanleiding van de overstap van Spring Boot 2 naar Spring Boot 3. Deze upgrade introduceerde nieuwe ingebouwde *observability*-functionaliteiten die relevant zijn voor de evaluatie van de gebruikte technologieën.

Observability (M., 2022) (Mina, 2023) is het vermogen om op basis van externe signalen inzicht te krijgen in de interne toestand van een systeem. Dit concept rust op drie pijlers:

- Logging – Het registreren van gebeurtenissen en foutmeldingen tijdens de uitvoering van de applicatie.

- Metrics – Het meten en monitoren van systeemgedrag zoals responstijden en geheugengebruik.

- Tracing – Het volgen van individuele requests doorheen verschillende componenten binnen een gedistribueerd systeem.

In het oorspronkelijk onderzoek werd reeds aandacht besteed aan *logging*. Door de overstap naar Spring Boot 3 werd het relevant om ook de twee andere pijlers — *metrics* en *tracing* — in detail te onderzoeken. In dit aanvullende luik van het onderzoek worden enkel frameworks geëvalueerd die beide pijlers ondersteunen. Compatibiliteit met het reeds gebruikte logging-framework is daarbij een voordeel.

Daarnaast wordt in deze analyse rekening gehouden met integratiemogelijkheden met AWS Cloudwatch, gezien er tijdens de stage een AWS API Gateway werd opgezet die inkomende requests observeert en logt naar Cloudwatch. Een naadloze integratie met dit observatieplatform is van belang voor een uniforme monitoring van het volledige systeem.

Er wordt een vergelijk gemaakt tussen deze 4 frameworks Micrometer, OpenTelemetry, New Relic, Datadog

Hiervoor hanteren we de Criterium.

- Actieve ondersteuning en onderhoud – De API draait in een productieomgeving. Het is daarom cruciaal dat het gekozen framework actief onderhouden wordt en toekomstbestendig is.

- Performantie/impact – Hoeveel overhead introduceert het framework op de applicatie? Een lichtere oplossing heeft de voorkeur, zeker in performantiegevoelige toepassingen.

- Integratie met AWS Cloudwatch – Omdat we AWS Cloudwatch gebruiken om onze data te visualiseren, krijgt dit criterium een hoger gewicht. Een goede integratie is essentieel voor een vlotte monitoring-setup.

- Integratie met het bestaande logging framework (Log4j) – We gebruiken momenteel Log4j voor logging. Frameworks die hier eenvoudig op kunnen aansluiten, krijgen een hogere score.

- Kostprijs – Frameworks die gratis of open-source zijn, hebben een streepje voor, zeker in het kader van kostenefficiëntie.

2.3.2.1. Micrometer (SigNoz, 2024)

Micrometer is een bibliotheek die zich richt op het verzamelen van metrics via een vendor-neutraal model. Het fungeert als een facade over verschillende monitoring backends (zoals Prometheus, Cloudwatch, Graphite, ...), waardoor je met één integratie meerdere outputs kunt ondersteunen.

Sinds Spring Boot 3 is Micrometer uitgebreid met Micrometer Tracing, een module die tracing mogelijk maakt via onderliggende tracers zoals OpenTelemetry of Brave. Daarnaast introduceerde Micrometer ook de Observation API, een uniforme observability-laag waarmee je je code één keer kunt instrumenteren om daarna te observeren.

Instrumenteren betekent ervoor zorgen dat je code zowel metrics, tracing als logs kan genereren. (OpenTelemetry, 2025)

```
 1. import io.micrometer.core.annotation.Timed;
 2. import io.micrometer.core.instrument.observation.Observation;
 3. import org.springframework.beans.factory.annotation.Autowired;
 4. import org.springframework.web.bind.annotation.GetMapping;
 5. import org.springframework.web.bind.annotation.RequestParam;
 6. import org.springframework.web.bind.annotation.RestController;
 7.
 8. import org.slf4j.Logger;
 9. import org.slf4j.LoggerFactory;
10.
11. @RestController
12. public class MyRestController {
13.
14.     private static final Logger logger = LoggerFactory.getLogger(MyRestController.class);
15.
16.     @Autowired
17.     private ObservationRegistry observationRegistry;
18.
19.     @GetMapping("/process")
20.     public String process(@RequestParam String input) {
21.         // Start een nieuwe observatie (timing, tracing en logging)
22.         Observation observation = Observation.createNotStarted("process.request",
observationRegistry);
23.
24.         // Begin de observatie
25.         observation.start();
26.
27.         try {
28.             logger.info("Processing request with input: {}", input);
29.             // Voeg een fictieve taak toe voor de meting
30.             String result = "Processed input: " + input;
31.             return result;
32.
33.         } finally {
34.             // Eindig de observatie
35.             observation.stop();
36.             logger.info("Request processed successfully.");
37.         }
38.     }
39. }
40.
```

Criterium	Gewicht	Score	Toelichting
Toekomstgericht	10	10	Micrometer wordt zeer goed ondersteund en is de facto de standaard voor metrics in Spring Boot. Daarom verwacht ik dat deze bibliotheek ook in de toekomst regelmatig zal worden bijgewerkt.
Performantie	5	5	Micrometer is zeer performant. Door gebruik te maken van een aantal technieken zoals Asynchrone verwerking, Lazily instantiated meters en Batching en sampling is de overhead minimaal.
Integratie met AWS	15	15	Micrometer heeft een ingebouwde module waarmee je rechtstreeks metrics naar AWS Cloudwatch kunt sturen.
Integratie met logging framework	10	10	Ook hiervoor biedt Micrometer een module die automatisch logs verzamelt. Daarnaast is het mogelijk om tags aan logs toe te voegen, zodat ze bijvoorbeeld met Sleuth getraceerd kunnen worden.
Gratis	10	10	Deze bibliotheek is volledig gratis
Totaal	**55**	**53**	**96%**

2.3.2.2. OpenTelemetry (SigNoz, 2024)

OpenTelemetry is een open source framework voor het verzamelen van metrics, logs en traces. Het biedt een gestandaardiseerde manier om observability-data te verzamelen en te exporteren naar systemen zoals Prometheus, Jaeger of Cloudwatch.

Micrometer maakt in Spring Boot 3 gebruik van OpenTelemetry voor tracing. Zo kan je via de eenvoudige Micrometer-API toch gebruik maken van de kracht van OpenTelemetry.

```
1.  @RestController
2.  public class MyRestController {
3.
4.      private static final Logger logger = LoggerFactory.getLogger(MyRestController.class);
5.      private static final OpenTelemetry openTelemetry = OpenTelemetry.getGlobalTracer("my-
application");
6.
7.      // Voor metrics
8.      private static final Meter meter = openTelemetry.getMeter("my-meter");
9.      private static final LongCounter requestCounter = meter
10.         .counterBuilder("requests.counter")
11.         .setDescription("Number of requests")
12.         .setUnit("1")
13.         .build();
14.
15.     @GetMapping("/process")
16.     @Timed(value = "process.request", extraTags = {"type", "example"}, longTask = true)
17.     public String process(@RequestParam String input) {
18.         // Begin tracing
19.         Span span = openTelemetry.getTracer("my-
application").spanBuilder("processRequest").startSpan();
20.         try (Scope scope = span.makeCurrent()) {
21.
22.             // Metrics: Verhoog de request teller
23.             requestCounter.add(1);
24.
25.             // Start logging
26.             logger.info("Processing request with input: {}", input);
27.
28.             // Fictieve taak (verwerkingslogica)
29.             String result = "Processed input: " + input;
30.
31.             // Log success
32.             logger.info("Request processed successfully.");
33.
34.             return result;
35.
36.         } catch (Exception e) {
37.             // Log error
38.             logger.error("Error processing request", e);
39.
40.             // Stuur de fout naar OpenTelemetry voor tracing (automatisch getraceerd)
41.             span.recordException(e);
42.
43.             return "Error: " + e.getMessage();
44.         } finally {
45.             // Sluit de trace af
46.             span.end();
47.         }
48.     }
49. }
```

Criterium	Gewicht	Score	Toelichting
Toekomstgericht	10	10	OpenTelemetry is ook zeer toekomstgericht. Deze wordt ondersteund door de Cloud Native Computing Foundation. Ook kan je OpenTelemetry gebruiken voor niet enkel Java waardoor het waarschijnlijk de standaard gaat worden voor observability
Performantie	5	4	OpenTelemetry is op zich heel performant, maar iets minder dan micrometer.
Integratie met AWS	15	15	Er is een OpenTelemetry AWS SDK dus de integratie verloopt zeer soepel
Integratie met logging framework	10	10	OpenTelemetry heeft een eigen logging oplossing, of met een bridge kan deze gebruikt worden samen met log4j,SLF4J
Gratis	10	10	Ja, OpenTelemetry is gratis.
Totaal	**55**	**52**	**94,5%**

2.3.2.3. New Relic (New Relic, 2025)

New Relic is een commercieel observability-platform dat ondersteuning biedt voor metrics, tracing en logging. Het platform richt zich op het bieden van realtime inzicht in zowel applicaties als infrastructuur en beschikt over een brede waaier aan integraties met uiteenlopende technologieën. In tegenstelling tot open-source alternatieven zoals OpenTelemetry en Micrometer, is New Relic een alles-in-één oplossing met gebruiksvriendelijke dashboards en geavanceerde analysetools. Deze eenvoud en volledigheid gaan echter gepaard met een abonnementsformule, waardoor het kosten met zich meebrengt. Dit onderscheidt het van de open-source frameworks, die gratis inzetbaar zijn maar doorgaans meer configuratie vereisen.

```
 1. import com.newrelic.API.agent.NewRelic;
 2. import com.newrelic.API.agent.Transaction;
 3. import org.slf4j.Logger;
 4. import org.slf4j.LoggerFactory;
 5. import org.springframework.web.bind.annotation.GetMapping;
 6. import org.springframework.web.bind.annotation.RequestParam;
 7. import org.springframework.web.bind.annotation.RestController;
 8.
 9. @RestController
10. public class MyRestController {
11.
12.     private static final Logger logger = LoggerFactory.getLogger(MyRestController.class);
13.
14.     @GetMapping("/process")
15.     public String process(@RequestParam String input) {
16.         // Haal de huidige New Relic transaction op
17.         Transaction txn = NewRelic.getAgent().getTransaction();
18.
19.         // Voeg custom event of metric toe (bijvoorbeeld custom timing)
20.         txn.addCustomAttribute("input", input);
21.
22.         // Start logging
23.         logger.info("Processing request with input: {}", input);
24.
25.         try {
26.             // Fictieve taak (verwerkingslogica)
27.             String result = "Processed input: " + input;
28.             logger.info("Request processed successfully.");
29.             return result;
30.
31.         } catch (Exception e) {
32.             logger.error("Error processing request", e);
33.             // Stuur de fout naar New Relic voor tracking
34.             NewRelic.noticeError(e);
35.             throw e;
36.         }
37.     }
38. }
39.
```

Criterium	Gewicht	Score	Toelichting
Toekomstgericht	10	8	New Relic wordt regelmatig bijgewerkt, maar omdat hetgeen open-source platform is, blijft de ontwikkeling afhankelijk van het bedrijf erachter. Bij eventuele wijzigingen of stopzetting vormt dat een risico.
Performantie	5	3	Vrij performant, maar het hangt allemaal af van de configuratie. Voor heel zware applicaties is een combinatie met OpenTelemetry beter.
Integratie met AWS	15	0	Omdat New Relic een observability platform is kan het enkel data van AWS ophalen, geen data sturen naar AWS. Of toch niet direct.
Integratie met logging framework	10	10	New Relic kan zowel zelf loggen als integreren met een logging framework via een appender.
Gratis	10	0	New Relic is niet gratis.
Totaal	**55**	**26**	**47%**

2.3.2.4. Datadog (Datadog, 2025)

Datadog is een cloudgebaseerd, commercieel platform voor monitoring en observability, met volledige ondersteuning voor tracing, metrics en logging. Het biedt krachtige, real-time analyses voor applicaties en infrastructuur, en een gebruiksvriendelijke interface voor visualisatie. In tegenstelling tot OpenTelemetry en Micrometer, die open-source frameworks zijn, biedt Datadog een geïntegreerde oplossing die klaar is voor gebruik, maar waarvoor een betaald abonnement vereist is voor de geavanceerde functies. Terwijl OpenTelemetry en Micrometer meer configuratie en aanpassingen vereisen, biedt Datadog een compleet pakket zonder de noodzaak van extra setup, maar dit komt tegen een prijs.

```
 1. import io.micrometer.core.annotation.Timed;
 2. import org.slf4j.Logger;
 3. import org.slf4j.LoggerFactory;
 4. import org.springframework.web.bind.annotation.GetMapping;
 5. import org.springframework.web.bind.annotation.RequestParam;
 6. import org.springframework.web.bind.annotation.RestController;
 7.
 8. @RestController
 9. public class MyRestController {
10.
11.     private static final Logger logger = LoggerFactory.getLogger(MyRestController.class);
12.
13.     @GetMapping("/process")
14.     @Timed(value = "process.request", extraTags = {"type", "example"}, longTask = true)
15.     public String process(@RequestParam String input) {
16.         try {
17.             // Start logging
18.             logger.info("Processing request with input: {}", input);
19.
20.             // Fictieve taak (verwerkingslogica)
21.             String result = "Processed input: " + input;
22.
23.             // Log success
24.             logger.info("Request processed successfully.");
25.
26.             return result;
27.
28.         } catch (Exception e) {
29.             // Log error
30.             logger.error("Error processing request", e);
31.
32.             // Stuur de fout naar Datadog voor error tracing (automatisch gedaan door de
agent)
33.             // Exception wordt automatisch getraceerd door de Datadog agent
34.
35.             return "Error: " + e.getMessage();
36.         }
37.     }
38. }
```

Criterium	Gewicht	Score	Toelichting
Toekomstgericht	10	8	Op zich worden er best veel updates gemaakt, enige nadeel is dat Datadog niet opensource is. Dus als er iets met het bedrijf zou gebeuren stopt de ontwikkeling.
Performantie	5	4	Beter dan New Relic.
Integratie met AWS	15	0	Omdat Datadog een observability platform is kan het enkel data van AWS ophalen, geen data sturen naar AWS. Of toch niet direct.
Integratie met logging framework	10	10	Datadog kan niet zelf loggen maar je kan wel logs naar Datadog sturen via een appender.
Gratis	10	0	Neen, Datadog is niet gratis.
Totaal	**55**	**27**	**49%**

2.3.2.5. Besluit

Micrometer komt als beste framework uit de vergelijking. Micrometer scoort het best in de vergelijking op basis van de opgestelde criteria. Het is een facade voor verschillende monitoringsystemen, waardoor we niet vastzitten aan één specifieke backend. Momenteel gebruiken we AWS Cloudwatch voor visualisatie, maar met Micrometer kunnen we eenvoudig overstappen naar bijvoorbeeld Prometheus — zonder aanpassingen aan de instrumentatiecode.
Dankzij de Observability API van Micrometer is de instrumentatie bovendien korter en leesbaarder dan bij bijvoorbeeld OpenTelemetry. Ook is Micrometer eenvoudiger te configureren en volledig geïntegreerd in Spring Boot, wat zorgt voor een vlotte implementatie en lange-termijnondersteuning.
Tot slot is Micrometer gratis en open-source, wat het een aantrekkelijk alternatief maakt ten opzichte van commerciële oplossingen zoals Datadog of New Relic.

2.3.3. Testing

Met unit testen kunnen de functionaliteiten van de API voldoende getest worden. We kunnen bijkomend ook JMeter gebruiken om de performantie, schaalbaarheid en betrouwbaarheid van onze API te testen. JMeter is een externe tool die gebruikers kan simuleren. Hierdoor kunnen we bv. 10.000 gebruikers simuleren die op hetzelfde moment de API aanspreken.

2.3.4. Rate limiting (Bucket4j, 2024)

We zouden ook graag de API voorzien van een rate limit/quota. Op deze manier kunnen we de API beschermen en ervoor zorgen dat deze minder belast zal worden. Ook zal de API beschermd worden tegen misbruik, DDOS aanvallen, … We kunnen

deze rate limits toepassen op een hoger niveau of op applicatieniveau. Op hoger niveau kunnen we gebruik maken van de AWS API gateway en op applicatieniveau van bucket4j. Deze library maakt gebruik van het token algoritme. Kort uitgelegd wil dit zeggen dat we een emmer met tokens voorzien en elke request naar de API verbruikt een token. Als onze emmer leeg is worden er geen requests meer toegelaten. Onze emmer vult ook tokens terug bij aan volgens een ratio die we zelf kiezen.

```
 1. import io.github.bucket4j.Bucket;
 2.
 3. public class ThrottlingFilter implements javax.servlet.Filter {
 4.
 5.     private Bucket createNewBucket() {
 6.          return Bucket.builder()
 7.             .addLimit(limit -> limit.capacity(50).refillGreedy(10, Duration.ofSeconds(1)))
 8.             .build();
 9.     }
10.
11.     @Override
12.     public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
FilterChain filterChain) throws IOException, ServletException {
13.         HttpServletRequest httpRequest = (HttpServletRequest) servletRequest;
14.         HttpSession session = httpRequest.getSession(true);
15.
16.         String appKey = SecurityUtils.getThirdPartyAppKey();
17.         Bucket bucket = (Bucket) session.getAttribute("throttler-" + appKey);
18.         if (bucket == null) {
19.             bucket = createNewBucket();
20.             session.setAttribute("throttler-" + appKey, bucket);
21.         }
22.
23.         // tryConsume returns false immediately if no tokens available with the bucket
24.         if (bucket.tryConsume(1)) {
25.             // the limit is not exceeded
26.             filterChain.doFilter(servletRequest, servletResponse);
27.         } else {
28.             // limit is exceeded
29.             HttpServletResponse httpResponse = (HttpServletResponse) servletResponse;
30.             httpResponse.setContentType("text/plain");
31.             httpResponse.setStatus(429);
32.             httpResponse.getWriter().append("Too many requests");
33.         }
34.     }
35.
36. }
```

2.3.5. Conclusie

Door elk onderdeel van het onderzoek grondig te analyseren, zijn er enkele waardevolle inzichten naar boven gekomen. Allereerst werd het logging framework onder de loep genomen. Hoewel de keuze voor Log4j op voorhand al vastlag, bleek het alsnog zinvol om meerdere alternatieven te vergelijken. Dit bevestigde dat de huidige keuze toekomstbestendig is, over alle nodige functionaliteiten beschikt, en bovendien goed integreert met het volgende onderdeel van het onderzoek: observability.

Logging vormt één van de drie pijlers van observability, samen met metrics en tracing. Het is dan ook een groot voordeel dat het bestaande loggingframework behouden kan blijven, en eenvoudig uitgebreid kan worden met ondersteuning voor de andere pijlers.

Voor metrics en tracing kwamen verschillende frameworks in aanmerking, maar Micrometer kwam hier als duidelijke winnaar uit de vergelijking. Aangezien Pension Architects met gevoelige data werkt, is het belangrijk dat de gekozen oplossing veilig, betrouwbaar én kostenefficiënt is. Commerciële oplossingen zoals Datadog en New Relic bieden uitgebreide functionaliteit, maar vallen omwille van licentiekosten en privacy-overwegingen af.

Micrometer daarentegen is gratis, open-source en sluit naadloos aan bij Spring Boot. Bovendien fungeert het als een facade voor OpenTelemetry, waardoor het eenvoudig is om later verder uit te breiden met meer geavanceerde tracing-oplossingen indien nodig. De introductie van de Observability API binnen Micrometer zorgt er bovendien voor dat de benodigde code korter en overzichtelijker wordt.

Voor het testen van de API is de keuze beperkter: klassieke unit tests volstaan in de meeste gevallen. Voor performance- of loadtesting kan aanvullend gebruikgemaakt worden van tools zoals JMeter.

Tot slot werd ook rate limiting onderzocht. Op applicatieniveau is Bucket4j momenteel de beste beschikbare oplossing. Op de netwerklaag biedt de AWS API Gateway aanvullende bescherming. Samen zorgen deze twee maatregelen ervoor dat de API op zowel applicatie- als netwerklaag beschermd is tegen misbruik en overbelasting.

2.4. Besluit

In dit hoofdstuk werden twee opeenvolgende onderzoeken uitgevoerd die samen de basis vormden voor de scope en planning van de stageopdracht. Het verkennend onderzoek bracht in kaart welke onderdelen essentieel zijn voor een productieklare API. Op basis daarvan werd in overleg met de stagementor een selectie gemaakt van prioritaire topics die vervolgens in het gedetailleerd onderzoek verder werden uitgewerkt.

De diepgaande analyse van onder meer logging, observability, testing en rate limiting leverde niet alleen waardevolle inzichten op, maar zorgde er ook voor dat onderbouwde keuzes konden worden gemaakt met het oog op implementatie. Deze keuzes houden rekening met factoren zoals toekomstgerichtheid, performantie, integratiemogelijkheden met bestaande infrastructuur (zoals AWS en Log4j), en kostenefficiëntie.

3. Technische realisaties

Dit hoofdstuk beschrijft welke taken er tijdens de stageopdracht zijn uitgevoerd. Deze worden gemanaged door Jira, de projectmanagement tool waar Pension Architects gebruik van maakt. De taken of tickets maken deel uit van een groter geheel genaamd epics. Deze epics zijn gemaakt aan de hand van de analyses of onderzoeken die in het vorige hoofdstuk staan beschreven. Zoals al eerder vermeld zijn deze ook aangepast in de loop van de stage om te kunnen inspelen op de op dat moment belangrijkste noden van de stageplaats. Elke taak bevat een omschrijving van de taak, een onderzoek of de problemen waar we tegen aan liepen, een uitgewerkte oplossing met eventuele screenshots of code en tot slot een besluit.

3.1. Logging

3.1.1. Inleiding

De eerste opdracht bestond erin de API te voorzien van logging op drie niveaus: lokaal in een bestand, zichtbaar in de console en extern via een logstream naar AWS CloudWatch.
Voor elk inkomend verzoek diende een logregel gegenereerd te worden die minstens volgende gegevens bevat:

- De gebruiker die de request uitvoerde
- Het aangesproken endpoint
- De statuscode van het antwoord
- De tijdsduur van het request

3.1.2. Uitwerking

In eerste instantie was geen bijkomend onderzoek nodig, aangezien deze opdracht rechtstreeks kon worden uitgewerkt op basis van het eerder gemaakte gedetailleerde onderzoek rond logging.

Naarmate de implementatie vorderde, bleek echter dat er extra configuratie nodig was om logs correct door te sturen naar AWS Cloudwatch.
Aangezien ik nog geen ervaring had met AWS, heb ik mij eerst verdiept in de werking van AWS en in het bijzonder in de dienst CloudFormation en het principe van Infrastructure-as-Code (IaC). Binnen Pension Architects wordt alle AWS-infrastructuur namelijk beheerd via code: configuraties worden geschreven in CloudFormation-scripts, die vervolgens automatisch omgezet worden naar live-omgevingen. Een belangrijk voordeel hiervan is dat omgevingen snel en betrouwbaar kunnen worden hersteld bij fouten of incidenten, en dat er eenvoudig backups van de infrastructuur kunnen worden gemaakt.

Als eerste stap heb ik een lokale logger opgezet op één specifiek endpoint. Dit liet toe om de configuratie van Log4j te testen en te verfijnen.

```
1. <RollingFile name="AccessLogFile" fileName="${sys:user.home}/logs/pensionarchitects-API-
access.log" filePattern="${sys:user.home}/logs/pensionarchitects-API-access.log.%i">
2.   <PatternLayout pattern="%d %-5p [%t] %-17c{2} (%13F:%L) - %m\n"/>
3.   <Policies>
4.     <SizeBasedTriggeringPolicy size="10485760"/>
5.   </Policies>
6.   <DefaultRolloverStrategy max="1" fileIndex="min"/>
7. </RollingFile>
```

```
1. <Logger name="AccessLogger" level="info" additivity="false">
2.   <AppenderRef ref="Console"/>
3.   <AppenderRef ref="AccessLogFile"/>
4. </Logger>
5.
```

In deze configuratie is de <Logger> met de naam AccessLogger gekoppeld aan de <Appender> AccessLogFile.
De Logger is het onderdeel dat in de code, bijvoorbeeld in een controller, wordt aangeroepen om berichten te loggen. De Appender bepaalt vervolgens waar en hoe deze logberichten worden verwerkt — in dit geval worden ze weggeschreven naar een lokaal bestand in de logs-directory.

Omdat de inhoud van de logberichten voor elk endpoint dezelfde gegevens bevatte, heb ik ervoor gekozen om de logging niet in elke controller afzonderlijk te implementeren, maar onder te brengen in een filter.
Een filter is een concept binnen Spring Boot waarmee je logica kan injecteren die bij elke inkomende request wordt uitgevoerd, vóór deze bij de controller aankomt.
Het grote voordeel van deze aanpak is dat de logging slechts één keer gecodeerd moet worden, en automatisch toegepast wordt op alle requests naar de API.

```
 1. @Component
 2. public class LoggingFilter extends OncePerRequestFilter {
 3.
 4.     private static final Logger LOG = LogManager.getLogger("AccessLogger");
 5.
 6.     @Override
 7.     protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain filterChain)
 8.           throws ServletException, IOException {
 9.
10.        long time = System.currentTimeMillis();
11.
12.        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
13.        // A null check on 'authentication' is necessary because if a request is made to an
endpoint
14.        // not secured by Spring Security, 'authentication' will be null, resulting in 'user'
being "Anonymous".
15.        String user = (authentication != null) ? authentication.getName() : "Anonymous";
16.
17.        try {
18.           filterChain.doFilter(request, response);
19.        } finally {
20.           time = System.currentTimeMillis() - time;
21.           LOG.info(
22.                 "User {} has called {} using a {} request, Request takes {}ms, Response code:
{}",
23.                 user,
24.                 request.getRequestURI(),
25.                 request.getMethod(),
26.                 time,
27.                 response.getStatus());
28.        }
29.
30.     }
31. }
```

De volgende stap was het opzetten van een logstream naar AWS Cloudwatch. Omdat Pension Architects hun projecten in containers op AWS host, was hiervoor bijkomende configuratie nodig. De container moest gebruik maken van een nieuwe Log4j-configuratie waarin de instellingen voor de logstream naar Cloudwatch waren opgenomen.
Om dit mogelijk te maken, werd de Log4j-configuratie opgeslagen in de AWS Parameter Store via het Infrastructure-as-Code (IaC)-project.

```
PensionArchitectsAPILogParams:
  Type: AWS::SSM::Parameter
  Properties:
    Name: !Sub '/${Environment}/${Client}/pensionarchitects-API/log4j2.xml'
    Type: String
    Value: !Sub
      - |
        <?xml version="1.0"?>
        <Configuration name="Log4j1">
          <Appenders>
            <CloudWatchAppender name="accesslog">
              <logGroup>/${Environment}/${Client}</logGroup>
              <logStream>pensionarchitects-API-accesslog</logStream>
              <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss} [%p] %c{2} %m%n" />

<clientFactory>be.pensionarchitects.logging.factory.CloudWatchClientFactory.createClient</clientF
actory>
              <dedicatedWriter>true</dedicatedWriter>
            </CloudWatchAppender>
          </Appenders>
          <Loggers>
            <Logger name="AccessLogger" level="info" additivity="false">
              <AppenderRef ref="accesslog"/>
            </Logger>
          </Loggers>
        </Configuration>
      - Environment: !Sub '${Environment}'
        Client: !Sub '${Client}'

```

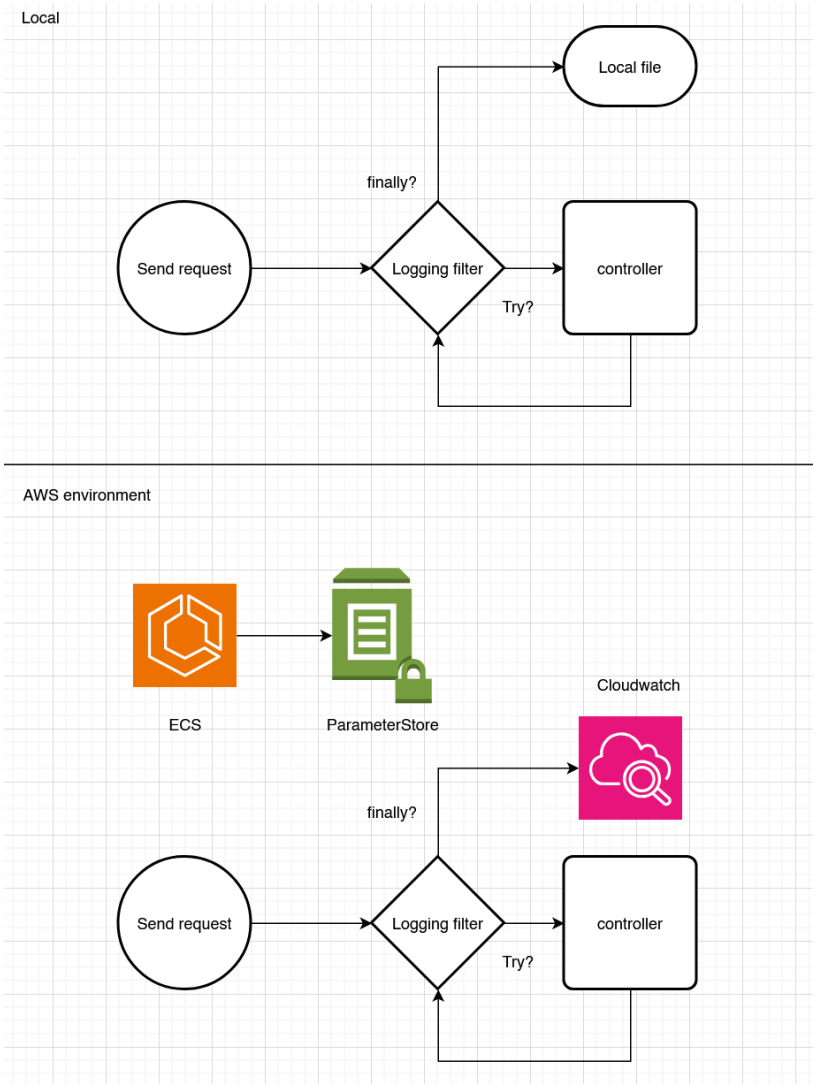
De image werd vervolgens aangepast zodat deze bij het opstarten de configuratie uit de Parameter Store ophaalt en de lokale configuratie overschrijft.
Hierdoor kan de container automatisch de juiste logging configuratie toepassen zonder manuele tussenkomst.

```
FROM amazoncorretto:17-alpine3.18


RUN apk update
RUN apk add AWS-cli
RUN apk add jq

ENV AWS_DEFAULT_REGION=eu-west-1
ENV TZ=Europe/Brussels

COPY target/pa-pensionarchitects-API-*[^tests].jar app.jar
COPY pensionarchitects-API.sh pensionarchitects-API.sh
RUN chmod +x pensionarchitects-API.sh

CMD ./pensionarchitects-API.sh
```

In het script pensionarchitects-API.sh zit een functie die de log4j configuratie uit de parameterstore haalt en die de log4j configuratie uit de container overschrijft met de nieuwe opgehaalde configuratie.

```
function replace_log4j_properties() {

    local location="/${environment}/${client}/pensionarchitects-API/log4j2.xml"
    local log4j=$(AWS ssm get-parameters --names "$location" | jq --raw-output
'.Parameters[].Value')
    if [[ -z "$log4j" ]]; then
       echo "no log4j found for pensionarchitects-API environment: $environment and client:
$client"
       exit 1
    fi

    echo "$log4j" > "log4j2.xml"
}

replace_log4j_properties

if [[ -n "$port" ]]; then
    vm_arguments=-Dserver.port=8081
fi

java $vm_arguments -jar app.jar

```

De flow lokaal en in de cloud ziet er dan als volgt uit.



Figuur 1 Logging flow lokaal en in de cloud

3.1.3. Besluit

Het configureren van Log4j verliep vlot, maar het integreren van de configuratie in de AWS-infrastructuur via Parameter Store en IaC was bijzonder leerzaam. De samenwerking tussen logging, cloudconfiguratie en containerisatie gaf een waardevol inzicht in hoe moderne applicaties beheerd en opgeschaald worden in een productieomgeving.

3.2. SNS-service

3.2.1. Inleiding

Als vervolgopdracht moest ik ervoor zorgen dat er bij kritieke problemen in de API automatisch meldingen worden verstuurd via Slack (een communicatieplatform vergelijkbaar met Microsoft Teams). Zo blijven developers in real-time op de hoogte

en kunnen ze meteen ingrijpen indien nodig. Concreet moet de API een SNS-service aanroepen, die vervolgens een bericht met de foutmelding doorstuurt naar een AWS Lambda. Deze Lambda zorgt er op zijn beurt voor dat het bericht in het juiste Slack-kanaal terechtkomt.

3.2.2. Uitwerking

Voor deze opdracht moest ik een manier vinden om foutmeldingen of exceptions op te vangen en deze door te sturen naar een lambda. Na wat opzoekwerk en overleg met mijn stagementor(s) had ik gekozen voor een global exception handler (H.K., 2024) (Chapman, 2013) (Baeldung, 2024). Deze maakte het mogelijk om exceptions op een centrale plaats op te vangen en te verwerken. Zo hoefde ik de code voor de sns service maar één plaats te schrijven en een bijkomend voordeel van een global exceptionhandler was dat ik voor elke exception, custom of niet, een aparte afhandeling kon maken. Wat de oplossing ook heel toekomstgericht maakt.

Voor de uitwerking van de opdracht ben ik begonnen met het schrijven van de global exception handler. Dit kunnen we realiseren door gebruik te maken van de @ControllerAdvice annotatie. Vervolgens kunnen we methodes gaan schrijven die in theorie verschillende type exceptions kunnen afhandelen. Dit kunnen default exceptions zijn maar ook custom exceptions. Voor deze taak was het genoeg om enkel Runtime exceptions af te handelen. Toekomstgericht kunnen we deze controller uitbreiden om meerdere exceptions af te handelen. We moeten er wel voor zorgen dat we de status van de exception (500, 403, …) terug aan de response koppelen. Dit bleek een probleem te zijn waar we pas later zijn achter gekomen. Als we dit niet doen werkt onze logging filter niet naar behoren.

```
1. @ControllerAdvice
2. public class GlobalExceptionHandler {
3.
4.     private static final Logger log = LogManager.getLogger(GlobalExceptionHandler.class);
5.     private static final int STACKTRACE_LIMIT = 1_000;
6.     private static final String SNS_TOPIC = "be.pensionarchitects.API.sns.topic";
7.     private static final String PROJECT = "pensionarchitects-API";
8.
9.     @Inject
10.    @Autowired
11.    private SnsService snsService;
12.
13.    @Inject
14.    @Autowired
15.    private ConfigurationService configurationService;
16.
17.    @ExceptionHandler(RuntimeException.class)
18.    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)

19.    public void handleRuntimeException(RuntimeException exception, WebRequest request,
HttpServletResponse response) {

    response.setStatus(HttpStatus.INTERNAL_SERVER_ERROR.value());
20.        log.error("Unexpected runtime exception: " + exception.getMessage(), exception);
21.
```

```
22.         String snsTopic = configurationService.getPropertyValue(PROJECT, SNS_TOPIC);
23.         String client = configurationService.getClient();
24.
25.         // Create a limited stack trace message
26.         String message;
27.         if (ExceptionUtils.getStackTrace(exception).length() <= STACKTRACE_LIMIT) {
28.            message = ExceptionUtils.getStackTrace(exception);
29.         } else {
30.            message = ExceptionUtils.getStackTrace(exception).substring(0, STACKTRACE_LIMIT)
31.                  + "\n ----- \n"
32.                  + ExceptionUtils.getStackTrace(ExceptionUtils.getRootCause(exception));
33.         }
34.
35.         // Send notification to SNS
36.         String url = request.getDescription(false);
37.         snsService.publishToTopic(snsTopic, client + ": " + url + "\n" + message);
38.     }
39. }
40.
```

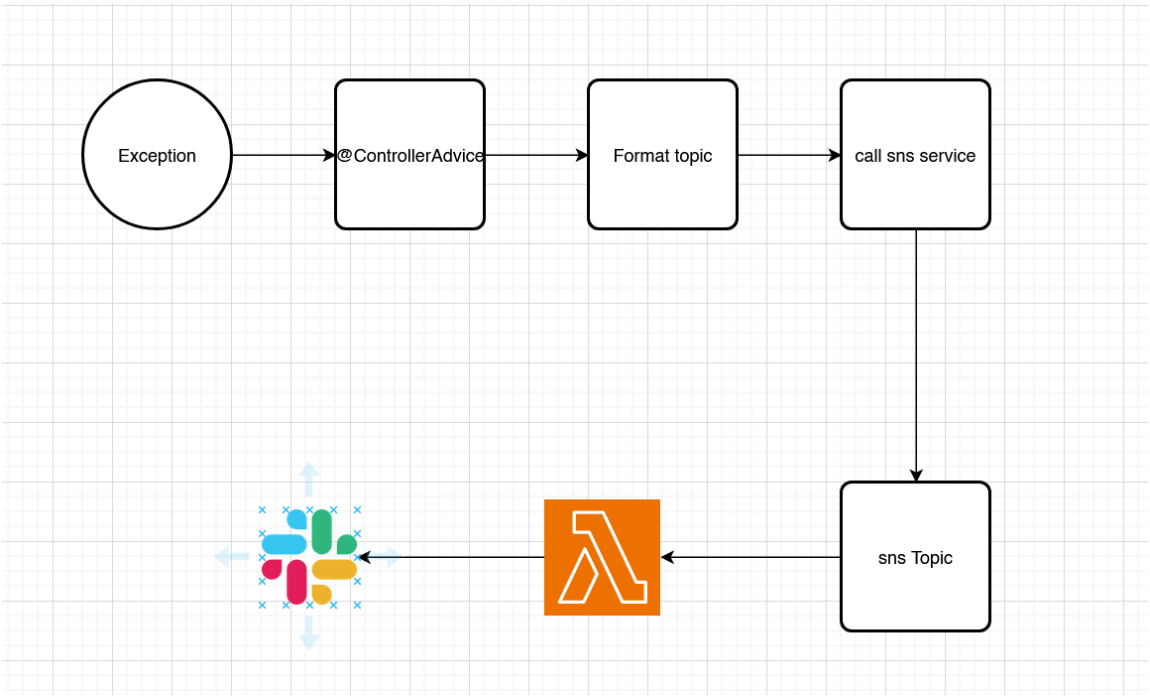
Een probleem waar we ook pas later zijn achter gekomen was dat als een request niet geautoriseerd is, deze nooit bij onze logging filter geraakt. En dus ook niet gelogd wordt. Om dit probleem aan te pakken moesten we een manier vinden om in te haken tijden het authenticatie proces. Gelukkig heeft Spring hier een manier voor namelijk een authenticationEntryPoint (Spring.io, 2025). Hierin schrijven we dezelfde code als in onze logging filter, alleen zetten we hier de status van de response op Unauthorized.

```
 1. @Component
 2. public class CustomAuthenticationEntryPoint implements AuthenticationEntryPoint {
 3.
 4.     private static final Logger LOG = LogManager.getLogger(ApiConstants.ACCESS_LOGGER_NAME);
 5.
 6.     @Override
 7.     public void commence(
 8.           HttpServletRequest request,
 9.           HttpServletResponse response,
10.           AuthenticationException authException) {
11.
12.        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
13.        LOG.info(AccessLogMessageFactory.create(request, response, 0L));
14.     }
15. }
16.
```

Het volgende deel de opdracht was de configuratie van de IAC in orde maken zodat de lambda functie ook daadwerkelijk de melding van de service kan verwerken en kan doorsturen naar de verschillende Slack kanalen.
We moesten een aantal dingen in orde maken zoals:

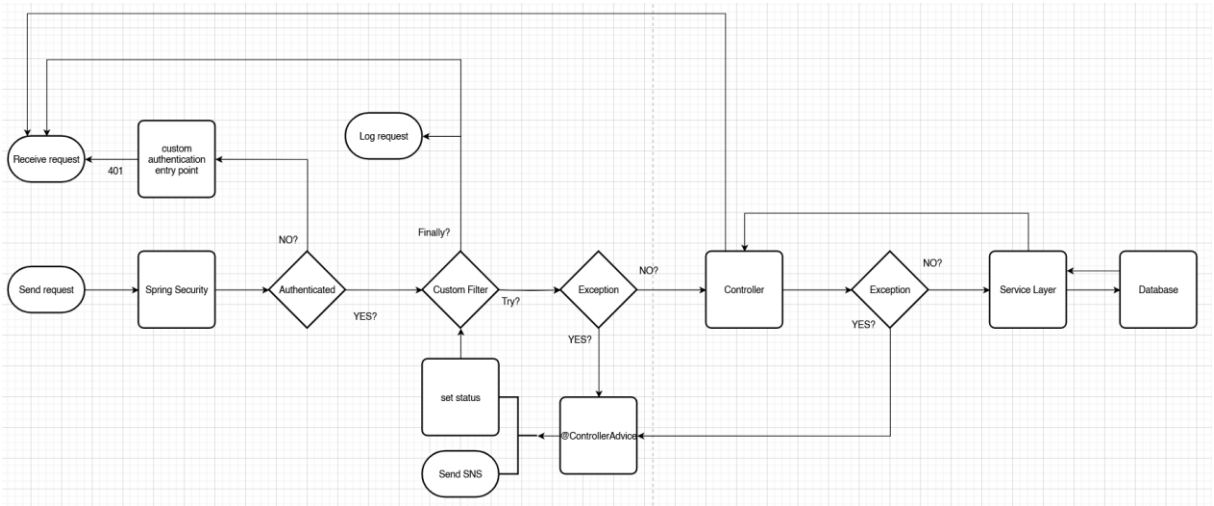
- Rechten in orde maken
- Sns topic opslaan in de parameterstore
- Topic subscriben op de lambda
- Slack kanalen toevoegen aan de lambda

De flow voor het versturen naar Slack ziet er dan als volgt uit:



Figuur 2 Flow voor een exception te versturen naar Slack

Als we dan beide taken, het loggen en het afhandelen van exceptions, gaan bekijken ziet de flow er als volgt uit:



Figuur 3 Volledige flow van logging en exception handling

3.2.3. Besluit

Bij het uitvoeren van deze opdracht heb ik vooral geleerd hoe exceptions op een globale manier kunnen worden opgevangen en afgehandeld. Daarnaast werd al doende duidelijk dat binnen het bestaande platform reeds een aantal services beschikbaar zijn, en dat het aanbevolen is om deze herbruikbare componenten waar

mogelijk in te zetten. Deze aanpak draagt bij aan consistentie en onderhoudsvriendelijkheid van de code. Al met al was dit een leerrijke en relevante opdracht die niet alleen bijdraagt aan de robuustheid van de applicatie, maar ook een concrete stap vormt richting een productieklare API.

3.3. JWT-tokens

3.3.1. Inleiding

De API is nu enkel beveiligd met Spring Security Basic Authentication. Dit is gewoon een username wachtwoord. Mijn opdracht bestaat erin om de API nog verder te beveiligen met een token systeem. Eerst werd er mij ook gevraagd of we de connectie tussen API en applicatie ook konden beveiligen met een certificaat. Hiervoor heb ik mij ingelezen in de mogelijkheden en deze doorgesproken met mijn stagementor. Naarmate dat gesprek vorderde kwamen er steeds meer vragen en obstakels naar boven. Uiteindelijk zijn ik en mijn stagementor tot het besluit gekomen dat we dit niet meer zouden implementeren in de stageopdracht en daarom werd de opdracht beperkt tot het genereren van tokens (JWT) voor de users.

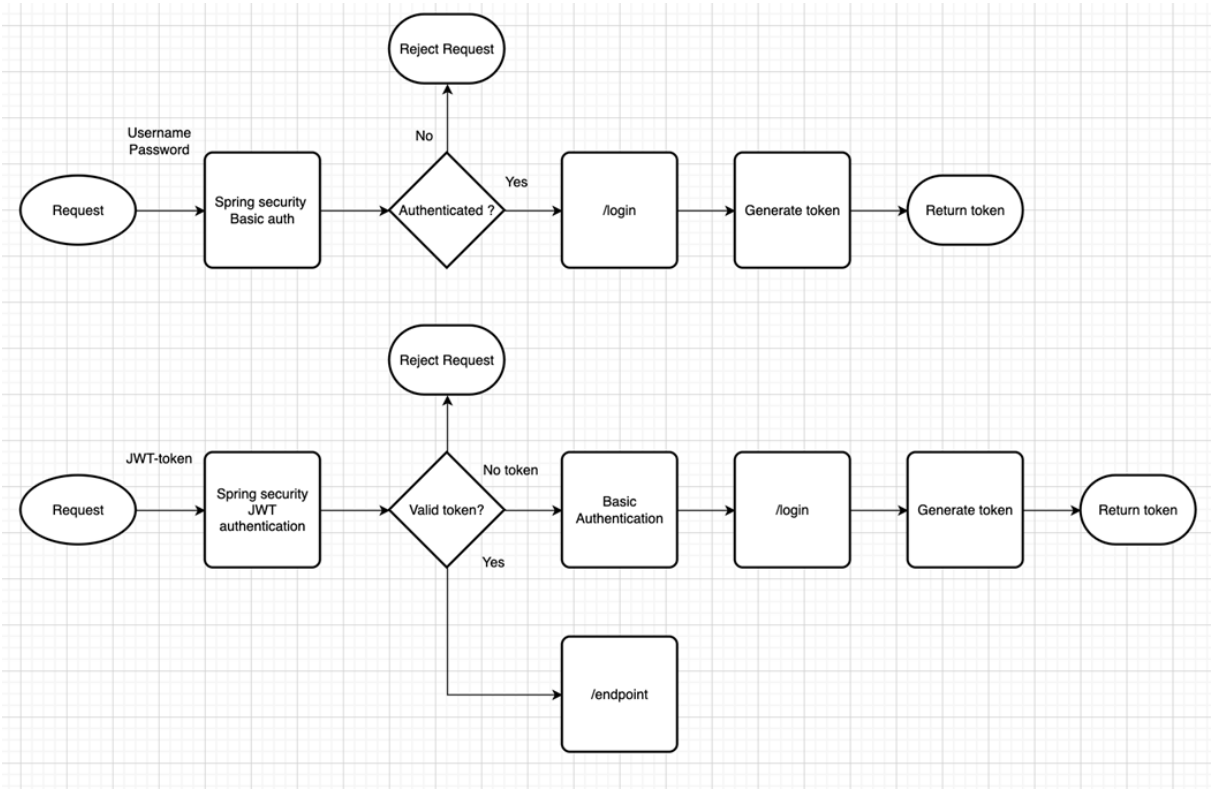
3.3.2. Uitwerking

Voor deze taak was er geen onderzoek nodig. Voor het implementeren van JWT-functionaliteit kon ik rekenen op mijn ervaring van het Project 4.0 dat ik eerder dit jaar heb gemaakt.

Voor het implementeren van de JWT-functionaliteit moesten we een aantal zaken aanmaken.

- Service voor het manipuleren en genereren van JWT-tokens.
- Filter waarin we tokens kunnen valideren tijdens een request
- Een aantal DTO-objecten
- Endpoints voor het aanvragen en hernieuwen van JWT-tokens
- Manier om de signing key voor de JWT-tokens maandelijks te roteren

De service is zoals eerder al vermeld enkel gericht op het manipuleren en genereren van JWT-tokens. Die service wordt dan aangeroepen in de filter samen met de userDetailsService. Die laatste is nodig om een gebruiker te kunnen identificeren.



Figuur 4 Flowchart van een request met en zonder jwt filter

In de filter gaan we kijken of er een token aanwezig is, is dat niet het geval wordt de request gewoon doorgestuurd naar de volgende filters. In het geval er wel een JWT-token aanwezig is wordt die token gedecrypt met de signing key. Als die key niet werkt wordt de request afgewezen, werkt de key wel wordt de username uit de token gehaald en wordt er een userdetails object van die user aangemaakt. We maken dan een authenticatietoken van Spring aan. Hiermee kunnen we een userdetailsobject manueel authenticeren door deze token toe te voegen aan de securityContext. Vanaf dan is onze gebruiker geauthenticeerd en kunnen we die gebruiken doorheen onze applicatie.

3.3.3. Besluit

Deze taak was meer een herhaling van wat ik al gedaan had tijdens mijn project 4.0. De moeilijkheid lag dan ook meer in het implementeren van deze oplossing in de bestaande codebase. Soms werkte dependency injection niet zoals verwacht, sommige stukken code zijn herwerkt om zo de code style van Pension Architects te respecteren. Er is veranderd van library om JWT-tokens te genereren. Dus ook al had ik deze taak in een vorig project gemaakt, toch waren er veel zaken waar rekening mee moest worden gehouden dat maakte dat ook deze taak weer zeer leerrijk was.

3.4. AWS API gateway (AWS, 2025)

3.4.1. Inleiding

Omdat in de toekomst bepaalde endpoints van de API publiek toegankelijk zullen worden gemaakt, werd mij gevraagd om een AWS API Gateway op te zetten. Deze gateway fungeert als het externe aanspreekpunt voor inkomende verzoeken en stuurt deze door naar een load balancer, die de requests vervolgens verdeelt over containers binnen de Elastic Container Service (ECS) — een door Amazon beheerde containeromgeving vergelijkbaar met Docker.

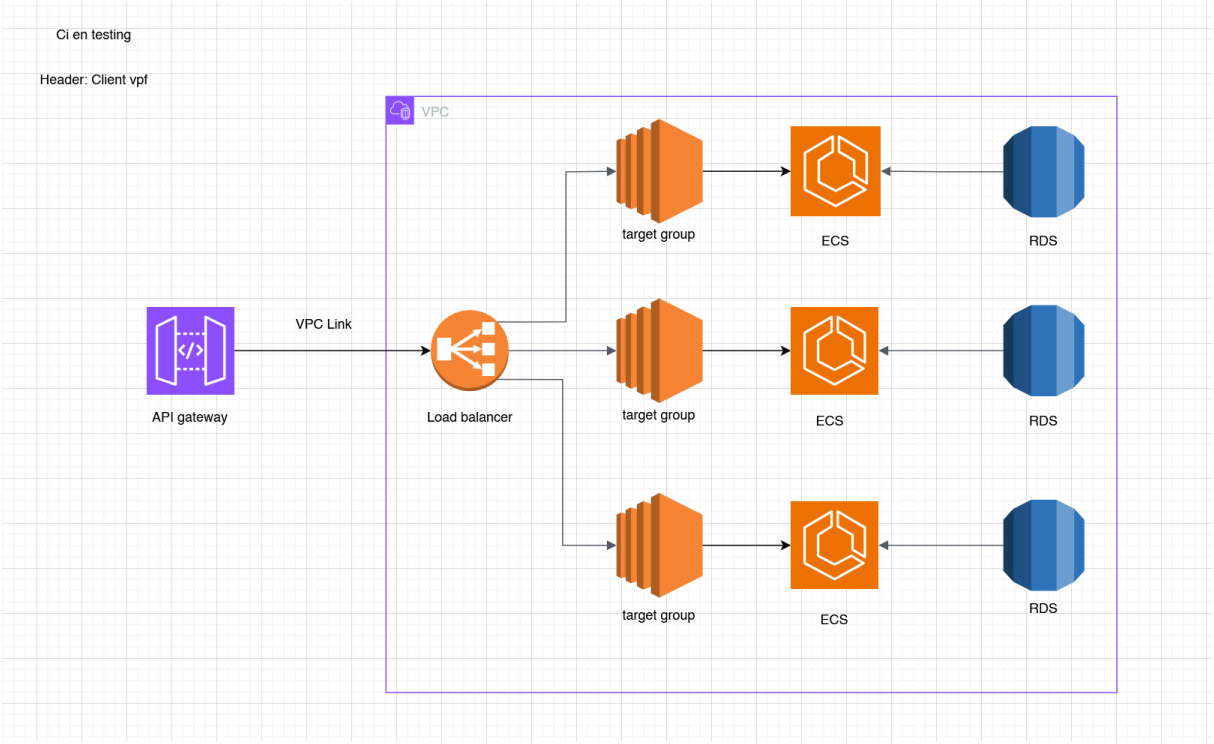
Na het handmatig opzetten van deze infrastructuur in de AWS-omgeving, was het de bedoeling om dezelfde configuratie om te zetten naar Infrastructure-as-Code (IaC). Dit werd gerealiseerd met behulp van AWS CloudFormation, waarmee de volledige setup in code wordt beheerd. Op deze manier blijft de infrastructuur volledig code-first en kan ze eenvoudig worden gerepliceerd of aangepast in andere omgevingen.

3.4.2. Uitwerking

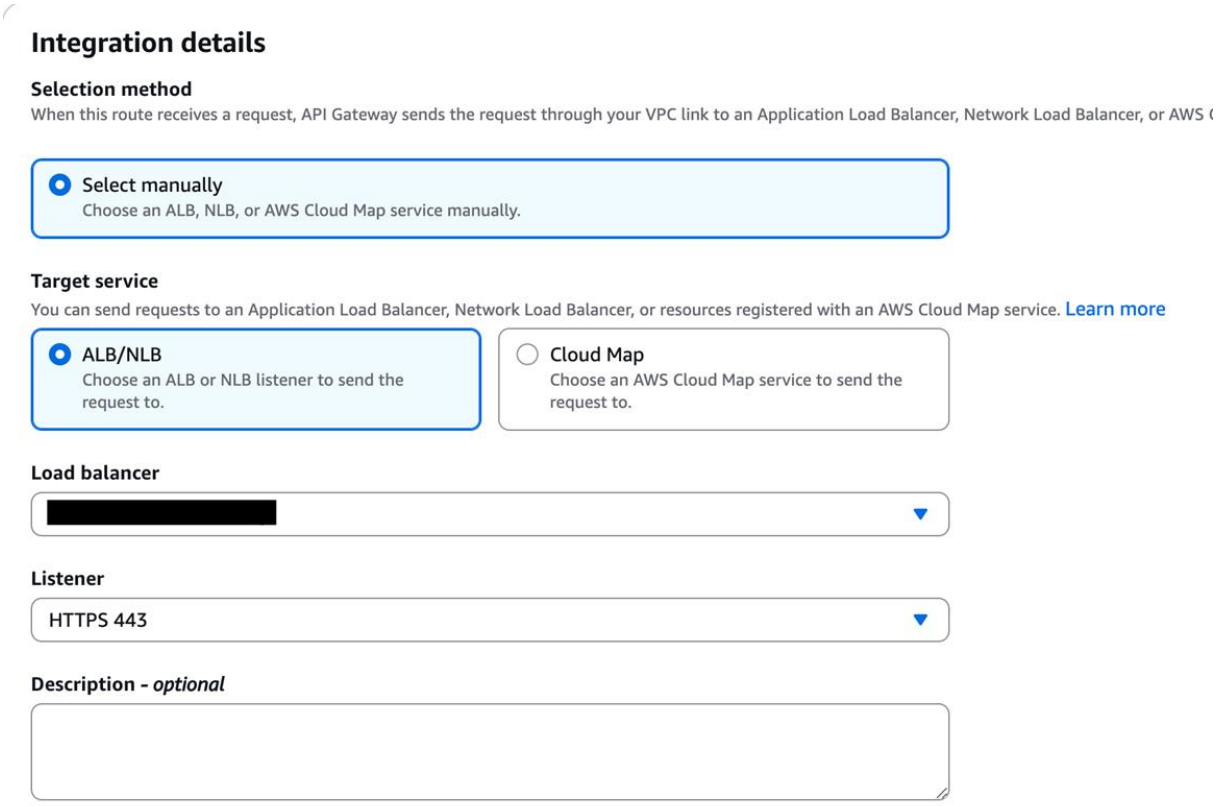
Voor deze taak heb ik eerst enkele dagen besteed aan het begrijpen van de bestaande architectuur: hoe de verschillende componenten met elkaar verweven zijn en op welke manier de API correct in de infrastructuur kan worden geïntegreerd. Ik heb me verdiept in de verschillende AWS-services die gebruikt worden binnen het project, waarbij ik voornamelijk beroep deed op de officiële documentatie van AWS. Voor het opzetten van de API via CloudFormation was het belangrijk dat ik deze documentatie correct kon interpreteren. Hoewel ze zeer volledig is, vereist ze de nodige ervaring om efficiënt te kunnen gebruiken. (AWS, 2025) (Guru, 2021)

Als eerste stap heb ik de AWS API Gateway manueel opgezet via de AWS Management Console. Op die manier kreeg ik een duidelijk beeld van de benodigde configuratie en instellingen, wat later als basis diende om de setup volledig in CloudFormation op te nemen binnen het Infrastructure-as-Code-project.

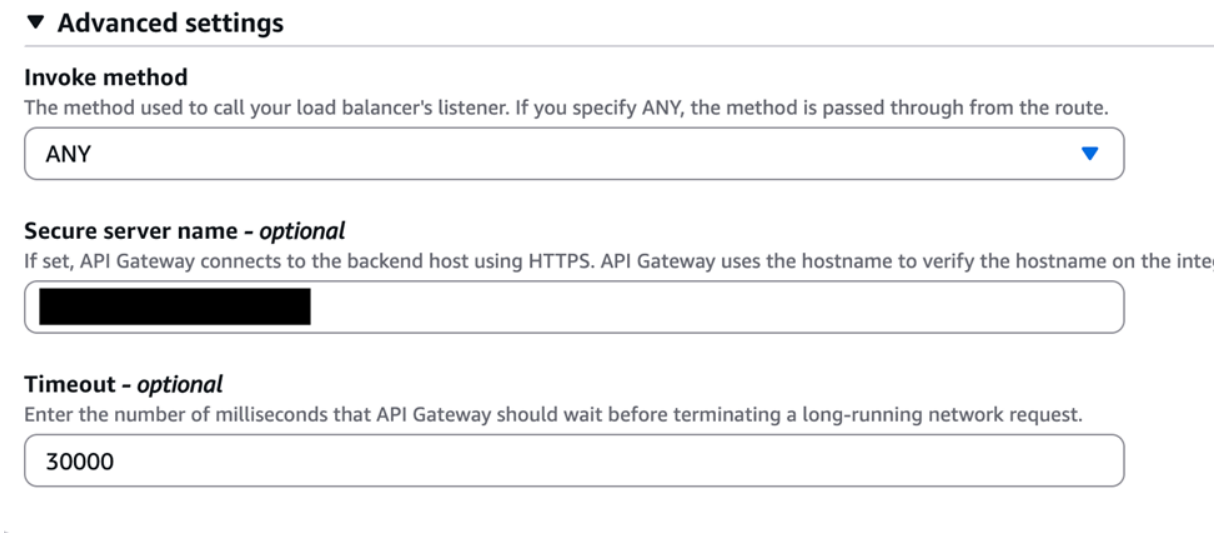
Na het opzetten van de gateway moest deze worden gekoppeld aan de bestaande infrastructuur. Dit gebeurt via zogenaamde integraties. Een integratie in AWS definieert de service waarmee de API Gateway verbonden is. In dit geval werd gekozen voor een Application Load Balancer (ALB), die inkomende requests verder afhandelt en doorstuurt naar de onderliggende containeromgeving.



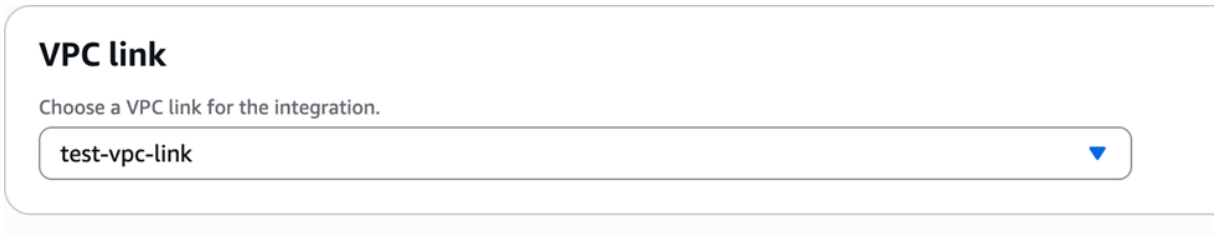
Figuur 5 Overzicht van de testing omgeving in AWS



Figuur 6 Aanmaak scherm voor een AWS API gateway Deel 1



Figuur 7 Aanmaak scherm voor een AWS API gateway Deel 2

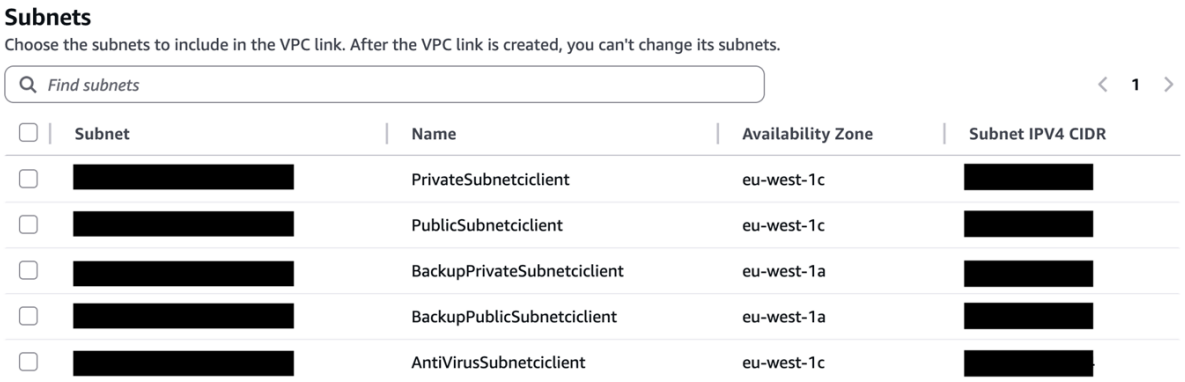


Figuur 8 Aanmaak scherm voor een AWS API gateway Deel 3

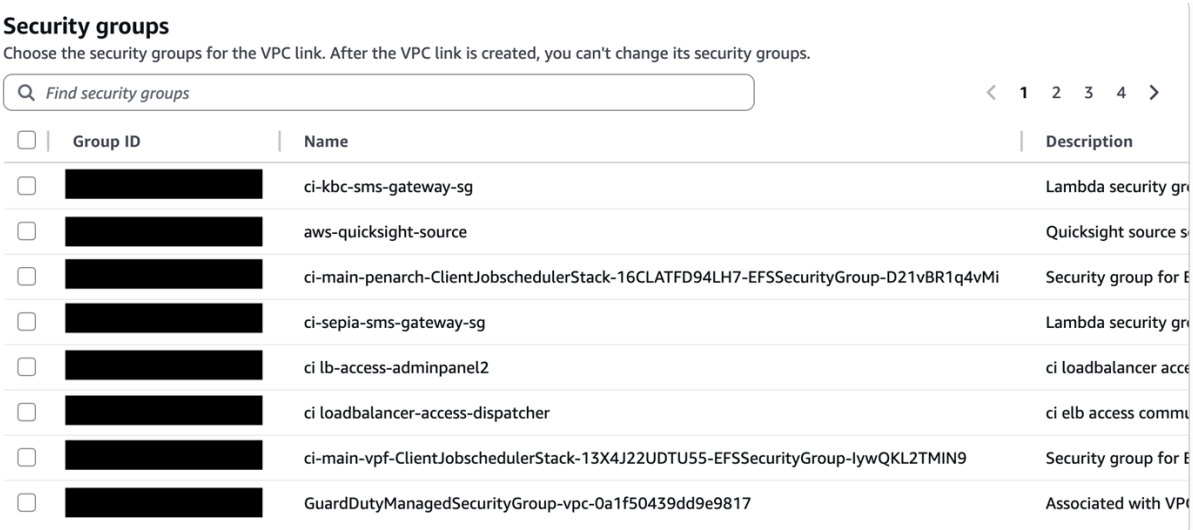
De integratie met de Application Load Balancer (ALB) vereist ook een koppeling met een Virtual Private Cloud (VPC). Een VPC is een virtueel netwerk binnen AWS waarin resources zoals containers, databases en load balancers op een veilige en geïsoleerde manier worden beheerd. Net zoals in een traditioneel netwerk bepaal je zelf het IP-bereik, de subnetten, de routering en de firewallregels.

Aangezien de API Gateway zelf buiten de VPC opereert, is een VPC Link nodig om deze met de interne infrastructuur — in dit geval de ALB — te verbinden. Deze link zorgt ervoor dat verkeer van de publieke API Gateway op een veilige manier toegang krijgt tot de private resources binnen de VPC.

Figuur 9 Aanmaak scherm voor een vpc link Deel 1



Figuur 10 Aanmaak scherm voor een vpc link Deel 2

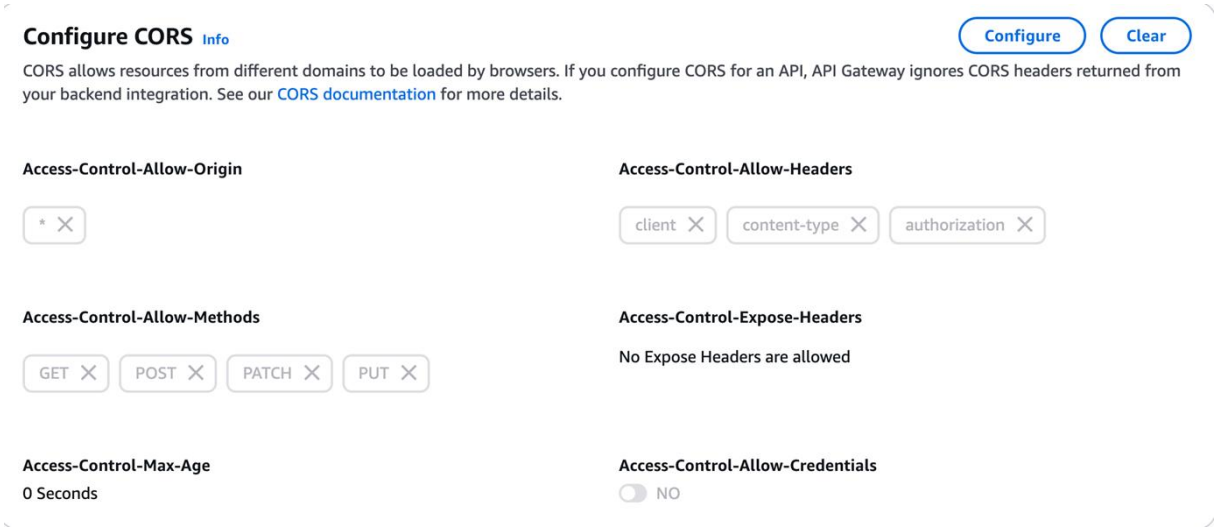


Figuur 11 Aanmaak scherm voor een vpc link Deel 3

Bij het opzetten van de VPC Link moeten een aantal essentiële instellingen worden meegegeven:

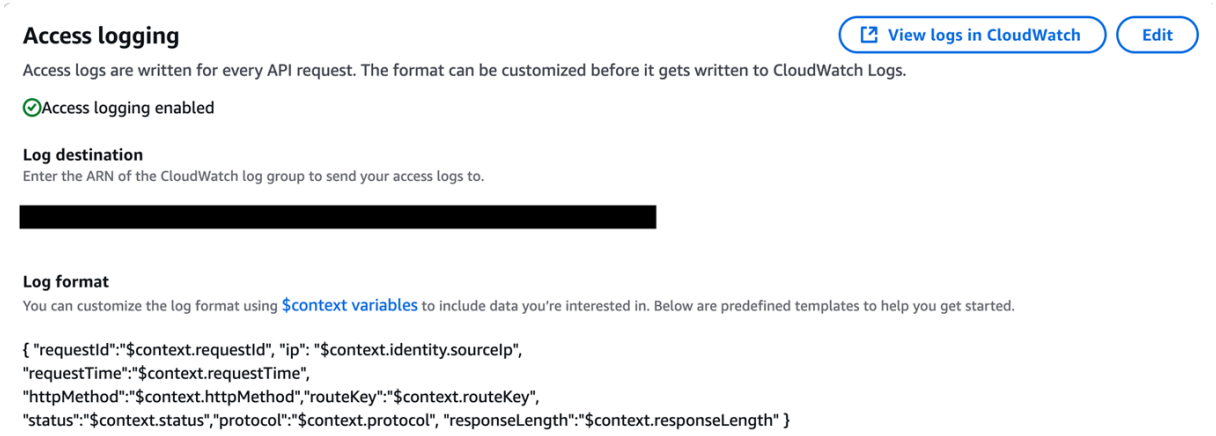
- Voor welke VPC de link bedoeld is
- Welke subnetten aan de link gekoppeld moeten worden
- Welke rechten of IAM-permissies de link krijgt om toegang te verlenen tot interne resources

Deze configuratie bepaalt hoe en onder welke voorwaarden de API Gateway met de private infrastructuur binnen de VPC kan communiceren.
Daarnaast moest ook CORS (Cross-Origin Resource Sharing) correct worden geconfigureerd. CORS is een mechanisme dat het mogelijk maakt om via een webbrowser verzoeken uit te voeren tussen verschillende domeinen. Zonder deze instelling zouden verzoeken vanuit bijvoorbeeld een frontend-applicatie op een ander domein door de browser worden geblokkeerd. Door CORS correct in te stellen op de API Gateway, wordt het mogelijk om op een veilige en toegelaten manier requests van externe domeinen te verwerken.



Figuur 12 Aanmaak scherm voor Cors instellingen

Ook is ervoor gezorgd dat er een logging mechanisme aanwezig is. Zo kunnen we de AWS API gateway in Cloudwatch integreren.



Figuur 13 Aanmaak scherm voor het activeren van Access logging

Na het succesvol opzetten van de AWS API Gateway via de webconsole van Amazon was de volgende stap om dezelfde configuratie op te zetten via CloudFormation, zodat deze kon worden opgenomen in het bestaande Infrastructure-as-Code-project.

Om hiermee aan de slag te gaan, heb ik me eerst verdiept in de documentatie van CloudFormation. Daarbij leerde ik niet alleen de inhoud, maar ook hoe ik de structuur van de documentatie efficiënt kon gebruiken om de juiste parameters, types en referenties te vinden.

Eenmaal vertrouwd met de documentatie ben ik begonnen met het schrijven van de CloudFormation-configuratie. De opbouw startte met het definiëren van de API Gateway zelf, waarna stap voor stap bijkomende componenten werden toegevoegd. Waar in de webconsole bepaalde configuraties gegroepeerd worden op één scherm, worden deze in CloudFormation afzonderlijk als aparte resources beschreven. Denk hierbij aan onderdelen zoals integrations, routes en stages, die elk hun eigen blok code vereisen en vervolgens gelinkt worden aan de hoofdcomponent van de API Gateway.

```
1. AWSTemplateFormatVersion: 2010-09-09
2. Description: Template for API-gateway
3.
4. Parameters:
5.   Client:
6.     Type: String
7.   Environment:
8.     Type: String
9.
10. Resources:
11.   PensionarchitectsAPIVpcLink:
12.     Type: AWS::APIGatewayV2::VpcLink
13.     Properties:
14.       Name: !Sub "${Environment}-${Client}-VpcLink"
15.       SecurityGroupIds:
16.         # Change to DispatcherLoadBalancerSecurityGroup after r202508 is deployed
17.         - Fn::ImportValue: !Sub '${Environment}-common-
DispatcherLoadBalancerSecurityGroupTemp'
18.       SubnetIds:
19.         - Fn::ImportValue: !Sub '${Environment}-common-PublicSubnet'
20.         - Fn::ImportValue: !Sub '${Environment}-common-BackupPublicSubnet'
```

```yaml

  PensionarchitectsAPIGateway:
    Type: AWS::APIGatewayV2::API
    Properties:
      Name: !Sub "${Environment}-${Client}-API-gateway"
      CorsConfiguration:
        AllowCredentials: false
        AllowHeaders:
          - "client"
          - "content-type"
          - "authorization"
        AllowMethods:
          - "GET"
          - "POST"
          - "PUT"
          - "PATCH"
        AllowOrigins:
          - "*"
      Description: "API gateway"
      ProtocolType: "HTTP"

  PensionarchitectsAPIIntegration:
    Type: AWS::APIGatewayV2::Integration
    Properties:
      APIId: !Ref PensionarchitectsAPIGateway
      ConnectionId: !Ref PensionarchitectsAPIVpcLink
      ConnectionType: "VPC_LINK"
      IntegrationMethod: "ANY"
      IntegrationType: "HTTP_PROXY"
      IntegrationUri:
        Fn::ImportValue: !Sub '${Environment}-common-InternalLoadbalancerHttpsListener'
      PayloadFormatVersion: "1.0"
      TlsConfig:
        ServerNameToVerify:
          Fn::ImportValue: !Sub '${Environment}-common-PensionarchitectsAPIDomainName'

  PensionarchitectsAPIRoutes:
    Type: AWS::APIGatewayV2::Route
    Properties:
      APIId: !Ref PensionarchitectsAPIGateway
      OperationName: "GetPlans"
      RouteKey: "GET /plans"
      Target: !Sub "integrations/${PensionarchitectsAPIIntegration}"

  PensionarchitectsAPIStage:
    Type: AWS::APIGatewayV2::Stage
    Properties:
      AccessLogSettings:
        DestinationArn: !GetAtt PensionarchitectsAPILogGroup.Arn
        Format: '{ "requestId":"$context.requestId", "ip": "$context.identity.sourceIp",
"requestTime":"$context.requestTime",
"httpMethod":"$context.httpMethod","routeKey":"$context.routeKey",
"status":"$context.status","protocol":"$context.protocol",
"responseLength":"$context.responseLength" }'
      APIId: !Ref PensionarchitectsAPIGateway
      AutoDeploy: true
      StageName: $default

  PensionarchitectsAPILogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub '${Environment}-${Client}'
      RetentionInDays: 7

```

Zodra de CloudFormation-code het verwachte resultaat opleverde, was de volgende stap om deze integratie op te nemen in een bestaande CloudFormation-stack. Op die manier wordt de volledige configuratie — inclusief de AWS API Gateway — automatisch uitgerold bij het opnieuw opstarten of bijwerken van de stack. Door de componenten op de juiste manier te ordenen en onderlinge afhankelijkheden aan te geven, wordt gegarandeerd dat de API Gateway in de correcte volgorde wordt geconfigureerd ten opzichte van de andere infrastructuurelementen.

3.4.3. Besluit

Deze opdracht was bijzonder boeiend om uit te voeren. Door het implementeren van de AWS API Gateway kreeg ik een breder inzicht in de algemene architectuur van Pension Architects. Het werd duidelijk hoe de verschillende onderdelen van de infrastructuur met elkaar verbonden zijn, en waarom bepaalde technische keuzes gemaakt zijn.

Daarnaast heb ik veel bijgeleerd over AWS. Voor het correct koppelen van de API Gateway aan de bestaande infrastructuur moest ik me verdiepen in verschillende AWS-services en hun onderlinge configuratie. Deze opdracht droeg dan ook sterk bij aan mijn algemene kennis van cloudarchitectuur en infrastructuurbeheer.

3.5. File upload

3.5.1. Inleiding

Om deze opdracht te begrijpen, is het belangrijk eerst de bestaande werking rond het uploaden van bestanden toe te lichten. In de huidige situatie kunnen aangeslotenen Pension Architects contacteren via verschillende communicatiekanalen. Binnen de webapplicatie kunnen beheerders deze communicatie koppelen aan de juiste persoon. Bij het toevoegen van een brief, bijvoorbeeld in PDF-formaat, wordt het bestand eerst opgeslagen op Amazon S3. De gegenereerde locatie van het bestand wordt vervolgens via een communicatieobject opgeslagen in de database.

Voor dit proces werden tot nu toe twee afzonderlijke services gebruikt: één voor het uploaden van het bestand, en één voor het registreren van de communicatie. De opdracht bestond erin om deze te combineren in één uniforme service. Daarnaast moesten alle bestaande toepassingen die de oude werkwijze gebruikten, worden aangepast naar deze nieuwe aanpak. Ook het bestaande API-endpoint, dat momenteel enkel de tweede stap uitvoert, moest uitgebreid worden zodat het voortaan beide taken kan uitvoeren.

Het uiteindelijke doel is om te evolueren naar een meer gestroomlijnde en toekomstgerichte aanpak, waarin de frontend eenvoudiger kan losgekoppeld worden van de service layer. Door de API reeds aan te passen, leggen we hiervoor nu al de nodige technische basis.

Een gelijkaardige wijziging moest worden doorgevoerd in het Camel-dispatcher (Apache, 2025) project, dat verantwoordelijk is voor het verwerken van uitgaande communicatie. Om de impact van deze aanpassing te begrijpen, is het nodig eerst het volledige proces van briefverwerking te schetsen. Hoewel er meerdere communicatiekanalen bestaan, focust deze opdracht zich uitsluitend op brieven.

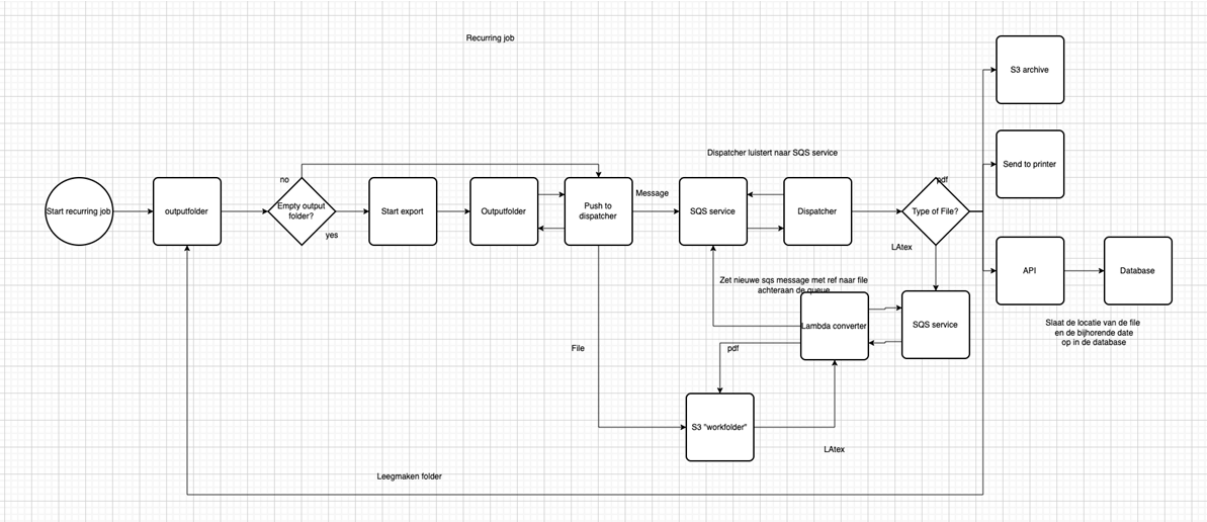
Wekelijks wordt er een terugkerende job uitgevoerd die kan worden opgesplitst in meerdere opeenvolgende stappen. De job start met het controleren van de outputfolder die gekoppeld is aan de terugkerende job via Fargate. Indien deze map leeg is, wordt het bijhorende exportproject automatisch opgestart. Dit project maakt gebruik van LaTeX-templates die worden ingevuld met gegevens uit de database. Een voorbeeld hiervan is een query die alle personen selecteert die in de komende week met pensioen gaan. Voor elk van deze personen wordt vervolgens een gepersonaliseerde brief gegenereerd. Tegelijkertijd wordt er voor elke brief een .fd-bestand aangemaakt — een tekstbestand met JSON-inhoud dat metadata bevat, waaronder een verwijzing naar het gegenereerde LaTeX-document.

Na afloop van de exportjob wordt de output doorgegeven aan het project PushToDispatcher. Dit project verplaatst de gegenereerde brieven naar een specifieke S3-map, de zogenaamde *workfolder*, waarin alle output van een klant centraal wordt beheerd. Daarnaast wordt het bijhorende .fd-bestand als bericht verstuurd naar een AWS SQS-queue.

Dispatcher luistert naar deze queue en bepaalt op basis van het bestandstype in de referentie van het bericht welke actie ondernomen moet worden. Indien het om een LaTeX-bestand gaat, stuurt dispatcher het bericht door naar een tweede SQS-queue waar een AWS Lambda-functie op geabonneerd is. Deze Lambda haalt op basis van de referentie het bestand uit de workfolder en converteert het LaTeX-document naar een PDF. De gegenereerde PDF wordt opnieuw geüpload naar de workfolder, waarna de Lambda een nieuw bericht aanmaakt met de referentie naar dit PDF-bestand. Dit bericht wordt vervolgens opnieuw in de oorspronkelijke SQS-queue geplaatst.

Wanneer dispatcher een bericht ontvangt met een referentie naar een PDF-bestand, voert het drie acties uit. Ten eerste wordt de PDF verplaatst naar een S3-archief zodat deze beschikbaar is in de webapplicatie, bijvoorbeeld via een downloadknop. Vervolgens wordt het document doorgestuurd naar de printer of een ander communicatiemiddel. Tot slot wordt ook de API aangeroepen om de communicatie correct te registreren in de database.

Wanneer alle stappen succesvol zijn uitgevoerd, wordt de outputfolder opgeschoond zodat het proces de volgende week opnieuw kan starten.



Figuur 14 Flow voor het versturen van brieven

3.5.2. Uitwerking

De eerste uitdaging bij deze opdracht was het correct opzetten van een endpoint dat zowel een bestand als bijhorende data (in de vorm van een DTO) kan ontvangen. Hiervoor bestaan meerdere mogelijkheden, en tijdens het verloop van de opdracht is de gekozen aanpak meermaals aangepast. In eerste instantie werd geprobeerd om het bestand op te nemen in de DTO zelf, maar dit gaf problemen bij de serialisatie. Spring Boot verwacht een stringrepresentatie van de parameters die vervolgens automatisch worden gemapped naar de DTO. Een File object kan niet omgezet worden naar een String representatie in een DTO.

Vervolgens werd gekozen om het bestand en de DTO afzonderlijk te versturen en deze vervolgens in de controller samen te voegen. Deze oplossing werkte correct in Postman-tests, maar gaf onverwachte problemen toen het project "Camel-dispatcher" de endpoint begon aan te roepen. Bij het versturen van een DTO naar een API worden immers alle velden, inclusief niet-gevulde, geserialiseerd. Hierdoor werd ook het lege file-object meegestuurd, wat leidde tot fouten. Na overleg met mijn stagementor werd besloten om een afzonderlijke DTO te definiëren die uitsluitend gebruikt wordt tijdens het versturen. In de controller worden deze gegevens vervolgens gemapt naar het originele communicatieobject.

Met deze oplossing in place werd ook dispatcher aangepast. Hier doken echter bijkomende obstakels op. Dispatcher luistert naar berichten in een SQS-queue en verwacht een .fd-bestand met referenties naar bestanden in S3. Hoewel AWS LocalStack reeds beschikbaar was als lokale testomgeving, ontbrak een methode om

een .fd-bestand effectief in de queue te plaatsen. Samen met mijn stagementor heb ik een testmethode opgesteld door een aangepaste test te schrijven die dit alsnog mogelijk maakt.

Daarna botsten we op een volgende uitdaging: dispatcher maakt gebruik van een OpenAPI-gegenereerde client die geen ondersteuning bood voor multipart/form-data. Deze encoding is echter essentieel wanneer we zowel bestanden als DTO's samen willen versturen. Na grondig onderzoek bleek de enige werkbare oplossing het updaten van de OpenAPI-generator. De update loste het oorspronkelijke probleem op, maar veroorzaakte een conflict met de gebruikte JUnit-versie. Door ook de testafhankelijkheden (zoals spring-boot-starter-test) te actualiseren, kon dit compatibiliteitsprobleem worden opgelost.

In de uiteindelijke implementatie wordt gebruik gemaakt van de annotatie @RequestPart om zowel de file als de DTO correct te verwerken. Omdat het mappen van de verzend-DTO en het bestand naar het effectieve communicatieobject manueel moet gebeuren, werd hiervoor een utilityklasse geschreven. Zo blijft de controller overzichtelijk:

```
 1. @Transactional(DefaultFactory.class)
 2. @PostMapping(value = "/policy/{policyNumber}/communication", consumes =
MediaType.MULTIPART_FORM_DATA_VALUE)
 3. public void addCommunication(
 4.         @PathVariable("policyNumber") String policyNumber,
 5.         @RequestPart(value = "file", required = false) MultipartFile file,
 6.         @RequestPart("fileCommunicationDataDTO")
 7.         FileCommunicationDataDTO fileCommunicationDataDTO) throws IOException {
 8.
 9.     log.debug("Started processing policy: {}", policyNumber);
10.     NewCommunicationDTO communicationDTO =
CommunicationUtil.mapFileCommunicationToNewCommunication(
11.           fileCommunicationDataDTO, file);
12.     MinimalPolicyDTO policy = new MinimalPolicyDTO(new PolicyNumber(policyNumber));
13.     communicationService.addCommunication(policy, communicationDTO, false);
14. }
15.
```

Deze methode vervangt de vorige aanpak waarin eerst het bestand apart werd geüpload, waarna de gegenereerde locatie werd opgeslagen via een tweede service. Onderstaande codefragment illustreert de oude werkwijze:

```
1. for (NewCommunicationWithFilesDTO communicationWithFiles :
getWizardData().getNewCommunications()) {
2.
3.     NewCommunicationDTO newCommunication = communicationWithFiles.getCommunication();
4.     ServerFileDTO documentLocation =
communicationService.uploadCommunicationFileToUniqueLocation(
5.           contract.getPolicy(), communicationWithFiles.getUploadedFile());
6.     newCommunication.setLocation(documentLocation.getLocation());
7.     communicationService.addCommunication(contract.getPolicy(), newCommunication, true);
8. }
9.
```

Met de nieuwe service kan dit sterk vereenvoudigd worden:

```
1. for (NewCommunicationWithFilesDTO communicationWithFiles :
getWizardData().getNewCommunications()) {
2.
3.      NewCommunicationDTO newCommunication = communicationWithFiles.getCommunication();
4.      newCommunication.setFile(communicationWithFiles.getUploadedFile());
5.      communicationService.addCommunication(contract.getPolicy(), newCommunication, true);
6. }
7.
```

Door beide services samen te voegen, wordt niet alleen het codevolume verminderd, maar ook de kans op fouten. Bovendien draagt dit bij aan de toekomstige loskoppeling van frontend en backend.

Alle plaatsen waar de oude services gebruikt werden, zijn intussen aangepast naar de nieuwe uniforme methode. Daarna kon de integratie met dispatcher worden afgerond. Een cruciale stap hierin was het correct downloaden van bestanden vanuit S3 op basis van gegevens in het SQS-bericht.

Aangezien dispatcher geen Spring Boot-applicatie is, moest dependency injection (DI) voor de service manueel worden toegepast. Dit gebeurde in drie stappen:

- Stap 1: Aanmaken van de service in de main-methode

```
1. ServerFileService serverService = new S3ServiceImpl(s3Client);
2.
```

- Stap 2: Injectie in de custom RouteBuilder

Vervolgens injecteren we dit object via de constructor van de CustomEndpointRouteBuilder, samen met andere afhankelijkheden:

```
 1. configuration.addRoutesBuilder(new CustomEndpointRouteBuilder(
 2.         config,
 3.         eboxConfig,
 4.         config.getObject(environment, client, "external_print/properties",
EndpointConfig.class),
 5.         config.getObject(environment, client, "pushBackup/properties", EndpointConfig.class),
 6.         config.getObject(environment, client, "sigedis/properties", EndpointConfig.class),
 7.         getSigedisSignerConfig(),
 8.         encrypterConfig(),
 9.         CamelApplication::resolve,
10.         serverService // injectie van de service
11. ));
12.
```

De constructor van deze class ziet er als volgt uit:

```
1. private final ServerFileService serverFileService;
```

```
 1. public CustomEndpointRouteBuilder(
 2.         ConfigurationService configService,
```

```
        EndpointConfig eboxConfig,
        EndpointConfig externalPrintConfig,
        EndpointConfig pushBackupConfig,
        EndpointConfig sigedisConfig,
        SignerConfig sigedisSignerConfig,
        EncrypterConfig sigedisEncrypterConfig,
        SqlResolver documentSqlResolver,
        ServerFileService serverFileService) {

    // Initialisatie van de velden
    this.serverFileService = serverFileService;
}

```

- Stap 3: Verder injecteren in. process(...)

De serverFileService kan vervolgens doorgegeven worden aan een custom Processor, zoals CommunicationAPIProcessor, opnieuw via constructor injectie:

```
.process(new CommunicationAPIProcessor(serverFileService)).id("ImportAPI")

```

De processor zelf:

```
private ServerFileService serverFileService;

public CommunicationAPIProcessor(ServerFileService serverFileService) {
    this(new LetterControllerAPI(APIClientFactory.createClient()));
    this.serverFileService = serverFileService;
}

```

Na het opzetten van deze configuratie was het mogelijk om de bestanden succesvol te downloaden, te koppelen aan de juiste DTO en door te sturen naar de, via OpenAPI gegenereerde, API-client.
Deze opdracht illustreert mooi hoe een relatief eenvoudige functionele vraag technische implicaties kan hebben over meerdere lagen heen: van controllerlogica en DTO-structuren tot CI-configuratie, dependencybeheer en infrastructuur-instellingen.

3.5.3. Besluit

Deze opdracht bleek uiteindelijk de meest complexe van de hele stageperiode. Waar het initieel ging om het samenvoegen van twee services tot één gestroomlijnd proces, bracht de opdracht tal van technische uitdagingen met zich mee die verder reikten dan de initiële scope. Elke stap in het proces bracht nieuwe inzichten en vereiste grondig onderzoek, overleg en iteratie.

De opdracht bood mij de mogelijkheid om diepgaand te leren werken met multipart-verzoeken in Spring Boot, het omgaan met DTO-serialisatieproblemen en het

ontwikkelen van utilityklassen voor herbruikbare mappinglogica. Daarnaast werd mijn kennis van Maven sterk verdiept door al de problemen met dependencies.

Ook het integreren met bestaande infrastructuur — zoals AWS S3, SQS en de LocalStack testomgeving — bood een waardevolle inkijk in hoe moderne cloudgebaseerde architecturen worden opgebouwd en getest. Doordat dispatcher geen Spring Boot-project is, moest ik leren hoe dependency injection handmatig opgezet wordt in een Java-project, iets wat me een beter inzicht gaf in hoe frameworks als Spring onder de motorkap werken.

Het feit dat ik voortdurend op onvoorziene situaties botste, heeft ervoor gezorgd dat ik niet alleen technische kennis heb opgedaan, maar ook beter heb leren omgaan met verandering en het belang van duidelijke communicatie binnen een team. Elke uitdaging bracht nieuwe leermomenten met zich mee, en heeft me geholpen om als ontwikkelaar te groeien.

Kortom: deze opdracht vormde een mooie weerspiegeling van de complexiteit en de veelzijdigheid van het werkveld waarin ik als softwareontwikkelaar aan de slag wil gaan.

4. Besluit

Tijdens deze stage ben ik tot het inzicht gekomen dat programmeren slechts één onderdeel is van het takenpakket van een softwareontwikkelaar. Vooraleer er effectief aan de implementatie kan worden begonnen, gaat vaak een grondige fase van analyse, overleg en planning vooraf. In de praktijk blijkt een vooraf opgestelde aanpak bovendien zelden definitief: tijdens de ontwikkeling wordt regelmatig geëvalueerd en bijgestuurd. Dit was bij Pension Architects niet anders.

Wat deze organisatie in positieve zin onderscheidt, is de open en ondersteunende teamcultuur. Iedereen is bereid om kennis te delen, en technische beslissingen worden besproken met het hele team. Die samenwerking en bereidheid om elkaar te helpen, beschouw ik als een belangrijk deel van hun succes. Tijdens mijn opdracht heb ik veel gehad aan die ondersteuning: van uitleg over de bestaande codebase en architectuur, tot feedback over mijn eigen aanpak en codekwaliteit.

Als ik terugblik op de stageopdracht ben ik erg tevreden over wat ik heb kunnen doen en leren. Ik heb gewerkt met Java, zowel met als zonder framework, en heb hands-on ervaring opgedaan met AWS, Docker, infrastructure-as-code en observability. Al deze technologieën kwamen samen in één geïntegreerd project, wat mij een realistisch en breed beeld gaf van de complexiteit en samenwerking binnen een moderne softwareomgeving.

De opdracht is nog niet volledig afgerond, maar er ligt nu een technische basis waarop verder gebouwd kan worden. Ik kijk met voldoening terug op een intensieve, maar

bijzonder leerrijke stageperiode die mijn verwachtingen van het beroep volledig bevestigd heeft.

5. Literatuurlijst

Apache. (2025). *Apache Camel Documentation*. Opgehaald van Apache Camel: https://camel.apache.org/docs/

Apache. (2025). *Log4j*. Opgehaald van Log4j: https://logging.apache.org/log4j/2.x/index.html

AWS. (2025). *Aws api gateway documentation*. Opgehaald van AWS api gateway: https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html

AWS. (2025). *AWS Cloudformation Documentation*. Opgehaald van AWS Documentation: https://docs.aws.amazon.com/cloudformation/

Baeldung. (2024, Januari 8). *Java Global Exception Handler*. Opgehaald van Baeldung: https://www.baeldung.com/java-global-exception-handler

Bucket4j. (2024, Augustus 18). *Bucket4j 8.14.0 Reference*. Opgehaald van Bucket4j Docs: https://bucket4j.com/8.14.0/toc.html#quick-start-examples

Chapman, P. (2013, November 1). *Exception Handling in Spring MVC*. Opgehaald van spring.io: https://spring.io/blog/2013/11/01/exception-handling-in-spring-mvc

Datadog. (2025). *Datadog docs*. Opgehaald van Datadog: https://docs.datadoghq.com/

Guru, S. (2021). *Tips for working with awscloudformation docs*. Opgehaald van Youtube: https://www.youtube.com/watch?v=Q0QIWgcE6XQ

H.K. (2024, Oktober 24). *Spring Boot Global Exception Handling*. Opgehaald van Medium: https://medium.com/@hk09/spring-boot-global-exception-handling-212261666c56

Logback. (2025). *Logback documentation*. Opgehaald van Logback: https://logback.qos.ch/documentation.html

M., G. (2022, 10 12). *Observability with Spring Boot 3*. Opgehaald van Spring.io: https://spring.io/blog/2022/10/12/observability-with-spring-boot-3

Mina. (2023, 10 14). *Monitoring and Observability with Spring Boot 3*. Opgehaald van Medium: https://medium.com/@minadev/monitoring-and-observability-with-spring-boot-3-2cb9cdb74a85

New Relic. (2025). *New Relic documentation*. Opgehaald van New Relic: https://docs.newrelic.com/

Obregon, A. (2023, December 4). *Effective API design in Java*. Opgehaald van Medium: https://medium.com/@AlexanderObregon/effective-api-design-in-java-a-guide-to-creating-robust-and-user-friendly-apis-e75942348bc0

OpenTelemetry. (2025, March 17). *Instrumentation*. Opgehaald van OpenTelemetry Docs: https://opentelemetry.io/docs/concepts/instrumentation/

SigNoz. (2024, 10 22). *OpenTelemetry vs Micrometer - Choosing the Right Metrics Tool*. Opgehaald van SigNoz: https://signoz.io/comparisons/opentelemetry-vs-micrometer/

Spring.io. (2025). *Spring.io Servlet authentication*. Opgehaald van Spring.io: https://docs.spring.io/spring-security/reference/servlet/authentication/architecture.html

Tinylog. (2025). *Tinylog configuration*. Opgehaald van Tinylog: https://tinylog.org/configuration/