THOMAS MORE

# Realisatie document

Sens van Aert
Student Bachelor Artificial Intelligence

2025 – 2026

# Inhoudstafel

# 1. Ometa

Ometa is een softwarebedrijf met expertise in systeemintegratie en digitale samenwerking. Sinds de oprichting in 2001 ontwikkelt het bedrijf oplossingen die organisaties ondersteunen bij het structureren, beheren en integreren van data en informatie. Daarbij ligt de nadruk op het verbinden van verschillende applicaties, databronnen en bedrijfsprocessen, zodat informatie op een efficiënte en consistente manier beschikbaar wordt binnen de organisatie.

Een centrale bouwsteen binnen het bedrijf is het Ometa Framework, een low-code platform dat toelaat om koppelingen tussen uiteenlopende systemen op een flexibele manier op te zetten. Het framework ondersteunt onder meer integraties met ERP-, CRM- en databronnen zoals SAP, Salesforce, Microsoft Dynamics, SQL-databanken en REST-gebaseerde toepassingen. Hierdoor kunnen bedrijven hun bestaande IT-landschap behouden, terwijl gegevensuitwisseling, synchronisatie en procesondersteuning aanzienlijk worden verbeterd.

Ometa positioneert zich daarmee als een partner die niet enkel inzet op technische integratie, maar ook op het optimaliseren van informatiestromen en samenwerking binnen organisaties. Door gestructureerde en ongestructureerde informatie samen te brengen, ondersteunt het bedrijf efficiëntere bedrijfsvoering en een betere toegankelijkheid van relevante data.

# 2. Inleiding

De stage bij Ometa situeerde zich binnen een project rond de analyse en visualisatie van logging data uit het Ometa Framework. Binnen dit framework worden grote hoeveelheden logs gegenereerd, die essentieel zijn voor het opsporen van fouten, performantieproblemen en andere afwijkingen. In de praktijk bleek echter dat deze logging data moeilijk te interpreteren was door de omvang, de beperkte duidelijkheid van sommige logs en de beperkingen van de bestaande rapportering en dashboards. Hierdoor werd het opsporen van problemen vaak een tijdrovend proces.

Om deze problematiek aan te pakken, werd binnen de stage gewerkt aan een verbeterde technische oplossing die relevante logging data op een gestructureerde, performante en bruikbare manier beschikbaar maakt. Het project combineerde elementen van data-analyse, datamodellering en backend ontwikkeling. Mijn bijdrage lag hierbij op het niveau van de database en de backend, waar ik instond voor de analyse van de logging data, de uitwerking van een analytisch datamodel en de ontwikkeling van een API-laag voor de ontsluiting van de gegevens naar het dashboard. Tijdens de stage werden verschillende onderdelen concreet gerealiseerd, waaronder het opzetten van een stermodel, het verwerken van data via database procedures en het ontwikkelen van meerdere backend functionaliteiten zoals filtering, paginering en specifieke endpoints voor logs, errors en health-data.

In dit realisatie document wordt toegelicht hoe deze oplossing technisch werd uitgewerkt en welke resultaten tijdens de stage werden bereikt. Daarbij ligt de nadruk op de gerealiseerde componenten, de gemaakte ontwerpkeuzes en hun meerwaarde binnen de context van Ometa.

Bij het opstellen van dit document werd ChatGPT van OpenAI gebruikt als ondersteunend hulpmiddel voor taalcorrectie, herformulering en het verbeteren van de leesbaarheid. Alle technische inhoud en conclusies zijn gebaseerd op mijn eigen werkzaamheden. De gegenereerde teksten werden door mij gecontroleerd en aangepast, en ik blijf verantwoordelijk voor de definitieve inhoud.

# 3. Begrippenlijst

| | |
|---|---|
| Ometa Framework | Een low-code integratieplatform van Ometa waarmee verschillende bedrijfsapplicaties en databronnen met elkaar verbonden worden. |
| BAM | *Business Activity Monitoring*: het registreren en analyseren van activiteiten binnen het Ometa Framework in de vorm van logs opgeslagen in de BAM-Database.<br>(Ometa, n.d.) |
| BAM-Database | De database waarin de operationele logs van het Ometa Framework worden opgeslagen.<br>(Ometa, n.d.) |
| BAM-Dashboard | Een dashboard waarmee medewerkers de logging en werking van het Ometa Framework kunnen analyseren. |
| BCM | *Business Connection Manager*: Service van het Ometa Framework dat verantwoordelijk is voor het managen van binnenkomend requests van cliënten van het Framework.<br>(Ometa, n.d.) |
| Method(e) | Een uitvoerbare actie of verwerking binnen het Ometa Framework. |
| Object | Een functionele representatie van een bedrijfsgegeven binnen het Ometa Framework. Een object bevat velden en kan gekoppeld worden aan methodes, views en andere functionaliteiten om gegevens uit externe systemen te raadplegen of te verwerken.<br>(Ometa, n.d.) |
| Profile | Een configuratie binnen het Ometa Framework die de gegevens bevat die nodig zijn om verbinding te maken met een externe databron of applicatie. Een profile wordt gebaseerd op een template en vult de bijbehorende verbindingsvelden in, zoals server- en databasegegevens.<br>(Ometa, n.d.) |
| Physical Profile | Een concreet profile dat een rechtstreekse verbinding met een specifieke databron of omgeving beschrijft. Een physical profile kan afzonderlijk gebruikt worden of als fysiek element deel uitmaken van een virtual profile, dat op basis van regels bepaalt welk physical profile gebruikt moet worden.<br>(Ometa, n.d.) |
| AppDomainName | De naam van het applicatiedomein of frameworkproces waarin een logbericht werd gegenereerd. Deze waarde helpt om te bepalen uit welk onderdeel van het Ometa Framework de logging afkomstig is.<br>(Ometa, n.d.) |
| Context Record | Een record of gegevensitem dat tijdens een methodeuitvoering verwerkt wordt. |
| Stermodel | Een analytisch datamodel met een centrale facttabel en meerdere dimensietabellen. |
| Facttabel | Een tabel waarin meetbare gebeurtenissen of waarden worden opgeslagen, zoals logs, duurtijden en scores. |

| | |
|---|---|
| Dimensietabel | Een tabel die extra beschrijvende context bevat, zoals methodes, gebruikers, componenten of tijdstippen. |
| Foreign Key | Een kolom die verwijst naar een record in een andere tabel en zo een relatie vormt. |
| Join | Een bewerking waarmee gegevens uit verschillende tabellen gecombineerd worden. |
| Query | Een opdracht waarmee gegevens uit een database worden opgehaald of aangepast. |
| ETL Proces | *Extract, Transform, Load:* gegevens ophalen, verwerken in een andere structuur en opslaan. |
| Execution plan | Het uitvoeringsplan waarmee SQL Server toont hoe een query uitgevoerd zal worden. |
| Timestamp | Het exacte tijdstip waarop een gebeurtenis plaatsvond. |
| LogId | Een unieke indentificatiecode van een logrecord. |
| Endpoint | Een specifieke route binnen een API waarmee een bepaalde actie of gegevensverzameling beschikbaar is. |
| Route | Het webadres binnen de API dat naar een bepaald endpoint verwijst. |
| Controller | Een backendklasse die inkomende API-aanvragen ontvangt en doorstuurt naar de juiste logica. |
| Service | Een klasse waarin een afgebakend deel van de bedrijfs- of verwerkingslogica wordt uitgevoerd. |
| Helper | Een herbruikbare functie of klasse voor ondersteunende logica. |
| Datadomein | Een functioneel afgebakende groep gegevens, zoals logs, errors of methodes. |
| Achtergrondtaak | Een taak die automatisch en zonder directe gebruikersinteractie op de achtergrond wordt uitgevoerd. |
| SwaggerUI | Een visuele webpagina waarmee API-endpoints gedocumenteerd en getest kunnen worden. |
| XML Summaries | Beschrijvende opmerkingen in C# die gebruikt kunnen worden om API-documentatie te genereren. |
| Parrallellisatie | Het gelijktijdig uitvoeren van meerdere bewerkingen. |
| Loadtest | Een test waarbij veel aanvragen worden uitgevoerd om het gedrag van een systeem onder belasting te meten. |
| JMeter | Een hulpmiddel waarmee belasting en responstijden van een API getest kunnen worden. |
| Logman | Een Windows-hulpmiddel waarmee performantiemetingen, zoals CPU- en geheugengebruik, verzameld worden. |

# 4. Context van het project

Binnen het Ometa Framework wordt zeer veel informatie over de werking opgeslagen in de vorm van logs. Deze logs bevatten informatie over uitgevoerde methodes, foutmeldingen, componenten, applicatiedomeinen, gebruikerscontext, gezondheidsinformatie van het framework en de server waarop deze draait met technische details zoals CPU- en geheugengebruik. Die informatie is waardevol, maar wordt pas bruikbaar wanneer ze op een gestructureerde manier kan worden gefilterd, gegroepeerd en gevisualiseerd.

Naar deze logs en manier van logging wordt gerefereerd als **BAM**, wat staat voor **Business Activity Monitoring.**

## 4.1. Huidige situatie

Wanneer gebruikers van het framework problemen ervaren zoals acties die niet doen wat ze horen te doen of zeer traag zijn, gberuikt men de gegenereerde logs om op onderzoek te gaan. Men kan verschillende Excel-rapporten genereren maar deze geven een verspreide analyse die niet overzichtelijk is. Ook kan er gebruik gemaakt worden van het BAM-Dasahboard, dit dashboard kan gebruikt worden om de activeiten in het framework te onderzoeken, maar het BAM-dashboard is zeer traag en geeft ook geen duidelijke inzichten.

## 4.2. Doel van het project

Het project had als doel om logging data beter beschikbaar te maken voor analyse en visualisatie. Concreet moest het dashboard gebruikers helpen om sneller te zien welke methodes traag zijn, welke errorgroepen vaak voorkomen, welke trends optreden in de tijd en of het gebruik van het dashboard zelf geen negatieve impact heeft op het Ometa Framework.

Mijn concrete verantwoordelijkheid bestond uit het analyseren van de bestaande logging data, het ontwerpen en implementeren van een aangepast datamodel in SQL Server en het uitwerken van de backend die deze data gecontroleerd beschikbaar maakt voor het dashboard.

# 5. Architectuur van de oplossing

De gerealiseerde oplossing is opgebouwd volgens een gelaagde architectuur. Deze architectuur scheidt de opslag en verwerking van data, de backendlogica en de presentatie naar de gebruiker. Door deze scheiding blijft de oplossing beter onderhoudbaar, uitbreidbaar en testbaar.

De eerste laag is de datalaag in SQL Server. Hier wordt de ruwe logging data uit het Ometa Framework verwerkt naar een analytisch datamodel. De oorspronkelijke logs bevatten veel detailinformatie en zijn niet altijd rechtstreeks geschikt voor snelle analyse. Daarom werd een stermodel uitgewerkt met een centrale facttabel en meerdere dimensietabellen. Dit model laat toe om logs te analyseren op basis van tijd, methode, component, ernste graad van de log of ook wel severity genoemd, app domain, correlatie en errorinformatie.

De verwerking naar dit stermodel gebeurt via stored procedures. Deze procedures verwerken de logging data batchgewijs, wat betekent dat de data of records in kleinere groepen worden opgesplitst, vullen de dimensietabellen aan en schrijven de relevante records weg naar de facttabellen. Daarnaast worden ook samenvattende tabellen opgebouwd, zoals methodHistory en healthHistory. Zij houden een samenvatting bij van het afgelopen uur van methodeuitvoeringen en gezondheidsinformatie. Door deze verwerking in

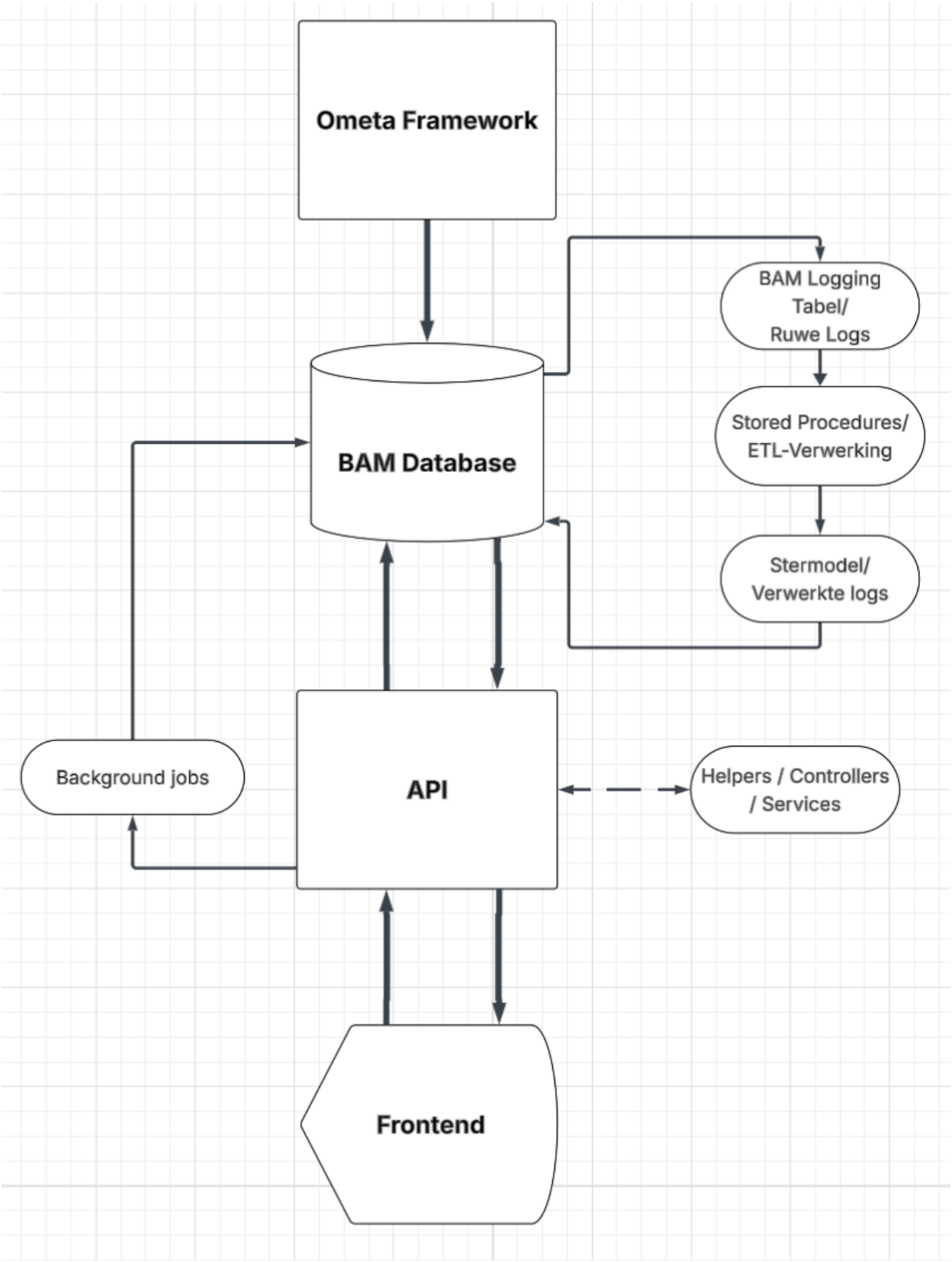
SQL Server te organiseren, kan de ruwe logging data gestructureerd en herhaaldelijk worden omgezet naar bruikbare analytische data.

Boven op de datalaag bevindt zich de backendlaag. Deze laag bestaat uit een .NET 8 API. De API vormt de schakel tussen de database en het dashboard. De backend bevat controllers voor de verschillende domeinen, zoals logs, errors, errorgroepen, healthdata, methodes en solutions. De controllers spreken services aan waarin de eigenlijke logica wordt uitgevoerd. Hierdoor blijft de controllerlaag beperkt tot het ontvangen van aanvragen en teruggeven van antwoorden.
Voor de communicatie met de frontend worden DTO's gebruikt. Deze zorgen ervoor dat de frontend enkel de gegevens ontvangt die nodig zijn voor visualisatie en analyse. De interne databasestructuur, met foreign keys en navigatie-eigenschappen, wordt daardoor niet rechtstreeks blootgesteld.

Omdat de hoeveelheid logging data groot kan worden, bevat de backend ook herbruikbare optimalisaties. Voor logoverzichten wordt cursorgebaseerde paginering gebruikt op basis van Timestamp en LogId. Daarnaast zijn filters gebundeld in aparte filterklassen en helperfuncties, zodat dezelfde filterlogica op meerdere endpoints kan worden toegepast. Voor regelmatig terugkerende of zwaardere taken worden achtergrondtaken gebruikt, bijvoorbeeld voor errorgroepering, een samenvatting van het afgelopen uur van errors die zijn voorgevallen en de framework score die aangeeft hoe goed het framework er op dat moment voor staat.

De bovenste laag is de presentatielaag. Deze bestaat uit het Angular-dashboard. Het dashboard gebruikt de endpoints van de .NET API om data op te halen en visueel voor te stellen. Gebruikers kunnen via het dashboard filters toepassen, grafieken bekijken, errors onderzoeken, methodes analyseren en algemene healthinformatie opvolgen.

Door deze architectuur is er een duidelijke verdeling van verantwoordelijkheden. SQL Server staat in voor opslag, structurering en batchverwerking van data. De .NET API staat in voor ontsluiting, validatie, filtering, paginering, berekeningen en achtergrondtaken. Het Angular-dashboard staat in voor de visuele presentatie en interactie met de gebruiker.



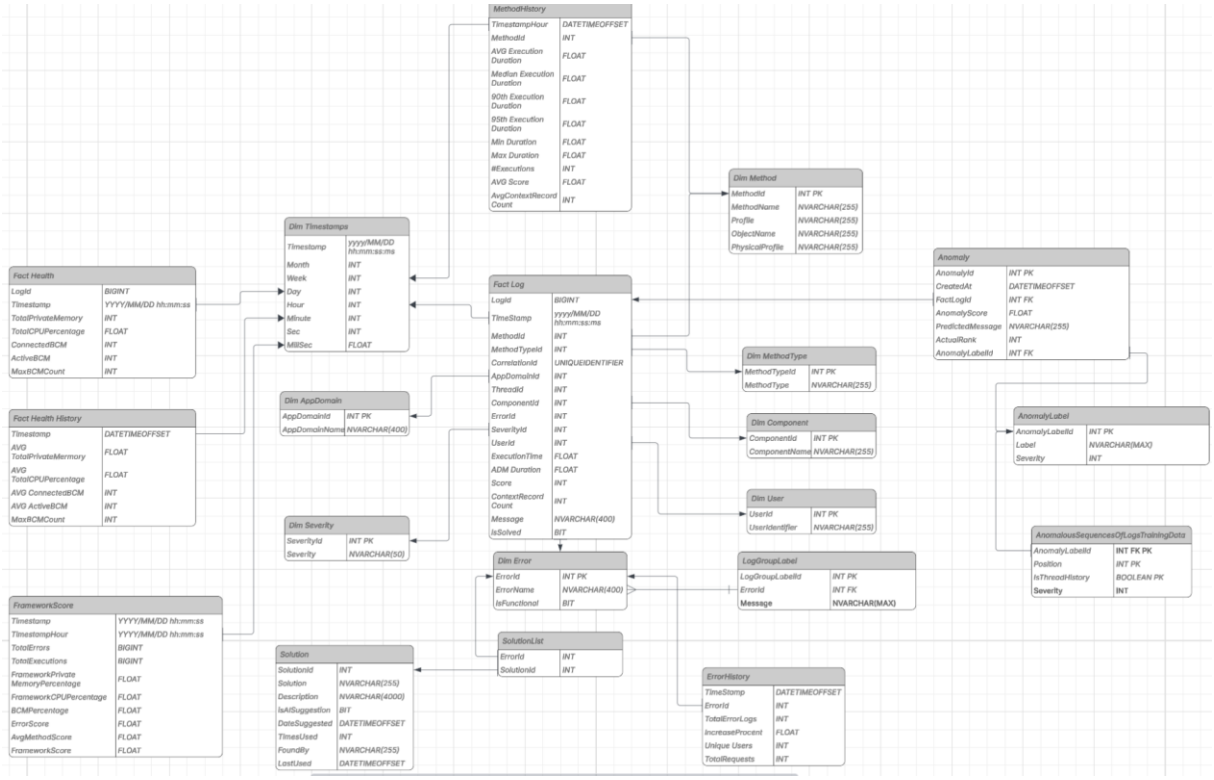

*Figuur 1: Diagram van de architectuur*

# 6. Realisatie van de database

## 6.1. Analyse en keuze van het datamodel

In de eerste fase van de stage werden de bestaande BAM-data en de gegenereerde rapportering geanalyseerd. Daarbij werd onderzocht welke velden relevant waren voor analyse, welke gegevens historisch bewaard moesten blijven en hoe de databewerking performant kon gebeuren.
Op basis van die analyse werd gekozen voor een stermodel. Die keuze werd niet louter theoretisch gemaakt, maar iteratief getoetst tijdens de stage.

In de eerste iteraties bleek het oorspronkelijke model te veel detail op te slaan. Na overleg met medewerkers en op basis van praktijk feedback werd het model bijgestuurd. Logs moesten beter gegroepeerd kunnen worden en enkel relevante informatie zou behouden blijven. Ook de verwerking van gebruikersinformatie werd vereenvoudigd: in plaats van meerdere variabelen uit de logs afzonderlijk te bewaren, werd onderzocht welke waarde voldoende was om unieke gebruikers te kunnen onderscheiden zonder gevoelige informatie onnodig te tonen.




*Figuur 2: Diagram stermodel*

## 6.2. Opbouw van het stermodel in SQL Server

Na de analyse werd het stermodel effectief in de database opgebouwd. Daarbij werden eerst de dimensietabellen opgezet en gevuld, waarna de fact tabel uitgewerkt werd. Tijdens deze fase moest rekening gehouden worden met de schaal van de logging data en met de performance van joins en inserts. Bij het vullen van de fact tabel werd gebruikgemaakt van batching. Records worden in groepen verwerkt, zodat fouten niet onmiddellijk een volledige rollback, het ongedaan maken van wijzigingen in de database, van de volledige verwerking veroorzaaken, ook is dit gedaan om de invloed op de server te beperken. Door de batching en de duidelijke bewaking van verwerkte records blijft de verwerking controleerbaar en schaalbaar.

Tijdens de verdere realisatie werd het model uitgebreid met bijkomende onderdelen voor methodHistory welke een samenvatting van het afgelopen uur van methodes opslaat, health welke informatie over de gezondheid van het framework en de server opslaat en healthHistory welke van de gezondheidinformatie een samenvatting opslaat van het afgelopen uur. Zo is het stermodel niet enkel een structuur om recente data te analyseren maar ook opslag en analyse van historische data.

Voor een kijkje in de logica van batching refereer ik naar de bijlage Code Fragment *1*

## 6.3. Stored procedures en geautomatiseerde verwerking

Om de verwerking van de logging data structureel en herhaalbaar te maken, werden de SQL-scripts omgevormd naar stored procedures. Stored procedures zijn SQL-scripts die zijn opgeslagen in de database zelf, we kunnen ze uitvoeren wanneer nodig. In totaal werden onder meer procedures voorzien voor het verwerken van dimensies en facts, voor methodHistory en voor healthHistory. De hoofdprocedure fungeert als orchestrator en roept de onderliggende procedures in de juiste volgorde aan.
De keuze voor stored procedures maakt het mogelijk om de verwerking logica centraal te bewaren, herbruikbaar te maken en gecontroleerd uit te voeren. Daarnaast wordt extra logging toegevoegd aan cruciale stappen in de verwerking, zodat fouten en performantieproblemen eenvoudiger opgevolgd kunnen worden.
Voor het periodiek uitvoeren van deze verwerking werd uiteindelijk niet vastgehouden aan SQL Server Agent als vaste oplossing. Afhankelijk van de omgeving kan de aanroep gebeuren via methodes binnen het Ometa framework zelf. In een Azure-omgeving kan daarnaast gebruikgemaakt worden van Elastic Jobs om de verwerking in te plannen.

Omdat er meerdere stored procedures uitgevoerd moeten worden werd er gekozen voor het maken van een centrale ‘hoofd’ stored procedure, die de nodige procedures uitvoerd. Hoe dit eruit ziet, is te zien in de bijlage Code Fragment *2*.

### 5.3.1. FillDimAndFactTables

FillDimAndFactTables wordt als eerste procedure uitgevoerd. Deze procedure haalt logs op die relevant zijn voor errors of methodes binnen de geselecteerde tijdspanne. De verwerking gebeurt batchgewijs. Per batch worden eerst de dimensies aangevuld op basis van informatie uit onder meer FormattedMessage, Message, Component, AppDomainName en Severity, wat allemaal velden zijn uit de originele BAM-logging. Daarna wordt de relevante informatie in de fact tabel geplaatst met verwijzingen naar de dimensietabellen in plaats van met herhaalde tekstwaarden.

Deze aanpak voorkomt dat dezelfde contextinformatie telkens opnieuw als tekst wordt opgeslagen. Daardoor worden analyses sneller, worden joins voorspelbaarder en blijft het model beter schaalbaar.

### 5.3.2. CalculateMethodHistory

CalculateMethodHistory groepeert methodelogs per methode en per tijdsvenster. Voor elke groep wordt een samenvatting gemaakt met het aantal uitvoeringen, gemiddelde duur, mediaan, P90, P95, minimumduur, maximumduur, het LogId van de traagste uitvoering, het gemiddelde aantal context records en de gemiddelde score.

Deze samenvatting vermijdt dat het dashboard telkens alle ruwe methodlogs moet verwerken voor historische grafieken. Tegelijk blijft de detailinformatie in Fact_Log beschikbaar wanneer een gebruiker wil doorklikken naar concrete logs.

### 5.3.3. CalculateHealthHistory

CalculateHealthHistory verwerkt gezondsheidsinformatie uit de logging van het Ometa Framework. Wanneer gezondheidslogging actief is, verschijnt er periodiek informatie over server- en framework verbruik

in de BAM-database. Deze procedure zet die informatie om naar een gestructureerde fact tabel en maakt een uurgebonden samenvatting.

Deze samenvatting, ook wel healthHistory genoemd, bevat onder meer CPU-gebruik van de server, CPU-gebruik van het framework, geheugengebruik van de server, geheugengebruik van het framework, het percentage vrij geheugen, het aantal verbonden BCM's, het aantal actieve BCM's en het maximum aantal BCM's dat verbonden mag zijn.

### 5.3.4 Sanitisatie script

Wanneer demo's worden gegeven aan klanten gebruiken we graag data van de klant zelf. Deze data bevat soms gevoelige gegevens die niet zomaar door iedereen gezien mogen worden. Daarom heb ik een script geschreven om deze data te anonimiseren. Hier worden enkel de velden en informatie bijgehouden die hoogst noodzakelijk zijn voor de analyse en geen gevoelige informatie bevatten.

De aanpak bestaat uit twee delen: regels voor het anonimiseren van Message en een whitelist voor toegelaten variabelen in FormattedMessage. De whitelist-aanpak is belangrijk omdat ze expliciet maakt welke informatie wel mag worden verwerkt. Alles wat niet op de whitelist staat, wordt niet meegenomen. Daardoor wordt het risico beperkt dat gevoelige klantdata per ongeluk in demo's of analyses terechtkomt.

## 6.4. Optimalisaties in de database

Omdat performantie cruciaal was voor het project, werd tijdens de stage veel aandacht besteed aan databaseoptimalisatie. Al bij de implementatie werd onderzocht welke indexen zinvol waren en hoe tabellen het best ingericht werden. Later werd dit verder onderzocht met execution plans, Query Store en sp_Blitz.

Een terugkerende optimalisatie was het verplaatsen van berekeningen wanneer bleek dat ze te zwaar wogen op inserts of joins. Zo werd de scoreberekening voor methodes en framework health meerdere keren herbekeken. Sommige berekeningen bleven nuttig in SQL omdat ze sterk op aggregatie zijn gericht. Andere berekeningen werden verplaatst naar de backend omdat ze beter te controleren, beter te testen of beter te timen waren.

Ook resourcegebruik werd onderzocht. Er werd bekeken hoe databaseverwerking minder impact kon hebben op CPU, geheugen en parallellisatie. Niet elke techniek bleek bruikbaar binnen de gebruikte SQL Server-versie, maar de analyse leidde wel tot gerichtere keuzes, zoals het beperken van parallelle verwerking waar mogelijk en het bewuster opbouwen van zware queries.

Verder werd er gebruikgemaakt van 'Query Store', een oplossing van SQL Server Management, om de zwaarste queries te onderzoeken en optimaliseren door bijvoorbeeld extra indexes toe te voegen of bestaande aan te passen. Dat verbeterde niet alleen de controle op de verwerking, maar maakte ook foutopsporing eenvoudiger.

Indexes passen we toe op SQL tabellen om gegevens sneller te kunnen opzoeken en verwerken. Indexes zijn te vergelijken met de inhoudsopgave in een boek. In plaats van dat we elke regel in de tabel te moeten doorzoeken, kunnen we rechtstreeks naar de relevante rijen navigeren.

Welke indexes gebruikt zijn kan teruggevonden worden in de bijlage Code Fragment *4*.

# 7. Realisatie van de backend

## 7.1. Opstart van de .NET 8 API

Na de eerste realisaties in de database werd gestart met de backend als .NET 8 API. Deze API vormt de schakel tussen de database en het dashboard. De backend werd opgezet om filtering, validatie, berekeningen, autorisatie en gestructureerde datatoegang op één centrale plaats te organiseren.

In de beginfase werd onderzocht welke calls nodig waren voor de frontend en hoe verantwoordelijkheden verdeeld moesten worden tussen database, backend en frontend. Er werd niet gekozen voor één generiek endpoint, maar voor gerichte routes per type visualisatie of functionaliteit. Dat beperkt de hoeveelheid data die per aanvraag aan de API teruggestuurd moet worden en maakt verdere uitbreiding eenvoudiger. Swagger UI werd gebruikt om endpoints te documenteren en later werden XML summaries toegevoegd om de documentatie te verbeteren.

## 7.2. Uitgewerkte endpoints en data domeinen

Tijdens de stage werden verschillende API-onderdelen gerealiseerd. Eerst werden endpoints uitgewerkt voor algemene logs, methodes, errors, errorgroepen en health-gerelateerde data. Later kwamen daar solution-gerelateerde endpoints, zoekfunctionaliteit, exportfunctionaliteit en bijkomende routes voor frontendnoden bij.

Bij Solutions moest rekening worden gehouden met een many-to-many-relatie tussen Solutions en Errorgroepen. Daarom werd een tussentabel voorzien. Een solution kan aan meerdere errorgroepen gekoppeld zijn en een errorgroep kan meerdere solutions hebben. Die relatie had gevolgen voor querycomplexiteit en performantie, waardoor schaalbaarheid expliciet onderzocht werd.

Een lijst van endpoints van de API kan in de bijlage *Figuur 4: Voorbeeld lijst van endpoints op swagger* gevonden worden.

Om abstractie te behouden en de volledige vorm van het databasemodel niet bloot te stellen aan de front-end werd er ook gebruikgemaakt van DTO's. Deze DTO's vervangen entiteiten en zijn de vormen van de antwoorden gestuurd door de API, ze behouden enkel de hoogstnoodzakelijke velden en informatie. Entiteiten bevatten technische relaties, foreign keys en navigatie-eigenschappen. DTO's bevatten enkel de velden die nodig zijn voor het antwoord van de API. Daardoor blijft de vorm van het antwoord van de API stabieler en is de frontend niet afhankelijk van de interne databasestructuur.

**Voorbeeld FactLog entiteit:**

```
public class FactLog {
    [Key]
    public required long LogId { get; set; }
    public required DateTimeOffset Timestamp { get; set; }
    public required DateTimeOffset TimestampHour { get; set; }
    public required int SeverityId { get; set; }
    public required Guid CorrelationId { get; set; }
    public int? MethodId { get; set; }
    public int? MethodTypeId { get; set; }
    public required int ComponentId { get; set; }
    public required int AppDomainNameId { get; set; }
    public required int ProcessId { get; set; }
    public required int ThreadId { get; set; }
```

```
    public int? ErrorId{ get; set; }
    public int? UserId { get; set; }
    public double? ExecutionDuration { get; set; }
    public double? AdmDuration { get; set; }
    public int? Score { get; set; }
    public int? ContextRecordCount { get; set; }
    public string? Message { get; set; }
    public bool IsSolved { get; set; }

    public required DimTimestamp DimTimestamp { get; set; }
    public required DimSeverity DimSeverity { get; set; }
    public DimMethod? DimMethod { get; set; }
    public DimMethodType? DimMethodType { get; set; }
    public required DimComponent DimComponent { get; set; }
    public required DimAppDomainName DimAppDomainName { get; set; }
    public DimError? DimError { get; set; }
}
```

**Voorbeeld DTO van de FactLog entiteit:**

```
public class FactLogDto {
    public required long LogId { get; set; }
    public required DateTimeOffset Timestamp { get; set; }
    public required DateTimeOffset TimestampHour { get; set; }
    public required string Severity { get; set; }
    public required Guid CorrelationId { get; set; }
    public string? MethodName { get; set; }
    public string? ObjectName { get; set; }
    public string? Profile { get; set; }
    public string? PhysicalProfile { get; set; }
    public string? MethodType { get; set; }
    public required string Component { get; set; }
    public required string AppDomainName { get; set; }
    public required int ProcessId { get; set; }
    public required int ThreadId { get; set; }
    public string? Error { get; set; }
    public int? UserId { get; set; }
    public double? ExecutionDuration { get; set; }
    public double? AdmDuration { get; set; }
    public int? Score { get; set; }
    public int? ContextRecordCount { get; set; }
    public string? Message { get; set; }
    public bool IsSolved { get; set; }
}
```

We zien dat het verschil hier bij de Dimensies zit. In plaats van gebruik te maken van ID's en dimensies die ze verbinden, is de DTO een reeds ingevuld antwoord waarbij de juiste informatie op een gestructureerde manier kan worden teruggegeven aan de front-end. Dit maakt de antwoorden van de API voorspelbaarder en is de documentatie op Swagger ook correcter.

## 7.3. Filters, tijdspannes en cursorpaginering

Een belangrijk deel van de backend realisatie bestond uit filters, tijdspannes en paginering. Verschillende routes gebruiken gelijkaardige parameters, zoals methodes, objecten, profielen, componenten, app domains, error groepen, gebruikers en correlatie-id's. Om duplicatie te vermijden, werd filterlogica herbruikbaar gemaakt.

Voor optionele filters werd een WhereIf-extensie gebruikt. Deze extensie voegt een Where-clause alleen toe wanneer de bijhorende conditie waar is. Hierdoor blijven queries leesbaar, ook wanneer een endpoint veel optionele filters ondersteunt.

**Codefragment van de WhereIf-extensie:**

```
public static class QueryableExtensions
{
    // Function to add filters to queries ("AND field = value" in SQL)
    public static IQueryable<T> WhereIf<T>(
        this IQueryable<T> query,
        bool condition,
        Expression<Func<T, bool>> predicate)
    {
        return condition ? query.Where(predicate) : query;
    }
}
```

Voor routes die dezelfde groep filters gebruiken, werd de filterset gebundeld in een LogFilters-klasse. Daarna werd een ApplyLogFilters-helper voorzien die deze filters centraal toepast. Dit maakt endpoints korter en vermindert de kans dat dezelfde filter op verschillende plaatsen net anders wordt toegepast.

**Class met meest gebruikte filters:**

```
public sealed class LogFilters
{
    public Guid[]? CorrelationIds { get; init; }
    public int[]? MethodIds { get; init; }
    public string[]? MethodNames { get; init; }
    public string[]? ObjectNames { get; init; }
    public string[]? Profiles { get; init; }
    public string[]? PhysicalProfiles { get; set; }
    public int[]? ComponentIds { get; init; }
    public int[]? AppDomainNameIds { get; init; }
    public int[]? ErrorIds { get; init; }
    public int[]? UserIds { get; set; }
}
```

**Functie die de filters van in de class 'LogFilters' toepast op de gegeven query:**

```
public static class LogQueryExtensions
{
    public static IQueryable<FactLog> ApplyLogFilters(
        this IQueryable<FactLog> query,
        LogFilters logFilters)
    {
        return query
            .WhereIf(logFilters.CorrelationIds != null &&
     logFilters.CorrelationIds.Length > 0,
                x =>
logFilters.CorrelationIds!.Contains(x.CorrelationId))
            .WhereIf(logFilters.MethodIds != null &&
logFilters.MethodIds.Length > 0,
                x => x.MethodId.HasValue &&
logFilters.MethodIds!.Contains((int)x.MethodId))
    }
}
```

Voor grote datasets werd cursorpaginering uitgewerkt. Bij deze vorm van paginering maken we gebruik van een cursor waarbij het laatste opgehaalde record als vertrekpunt voor de volgende pagina wordt gebruikt. De cursor binnen ons project combineert Timestamp en LogId, zodat de volgorde deterministisch blijft, ook wanneer meerdere logs dezelfde timestamp hebben. Hierdoor kan de API zowel de volgende als vorige pagina's correct ondersteunen. Later werd de logica uitgebreid zodat dezelfde endpoint zowel chronologische als antichronologische volgorde kan ondersteunen via een IsReverse-parameter.

Bij routes met paginering kan de gebruiker van de API vragen naar informatie met de filters die toegepast kunnen worden op die specifieke route, de grootte van de pagina met een maximum tot 5000, in welke volgorde men de data wil ontvangen (Oudste eerst of nieuwste eerst) en of men de volgende of vorige pagina wilt zien mits men al eerder een pagina heeft opgevraagd en er is data aanwezig voor of na de cursor.

Om deze logica beter te begrijpen refereer ik graag naar bijlage Code Fragment *5*.

## 7.4. Backend services en periodieke jobs

### 7.4.1. Error groepering

Voor error groepering werd een bestaande aanpak verder geïntegreerd in de backend. De oorspronkelijke groepering van errors op basis van gelijkaardige foutboodschappen gebeurde in een Python-script. Tijdens de stage werd deze aanpak omgevormd naar een bruikbare C#-service. Hiervoor werd gewerkt met een .NET-communityvariant van Drain3, zodat foutboodschappen kunnen worden gegroepeerd op basis van terugkerende templates.

De service kan via een API-call aangeroepen worden en werd later ook geschikt gemaakt om periodiek via een job te draaien. Hoewel deze functionaliteit op termijn mogelijk door een AI-oplossing vervangen kan worden, was de geïntegreerde service tijdens de stage al bruikbaar om onbekende of gelijkaardige foutmeldingen te structureren.

### 7.4.2. Error history

Naast het groeperen van errors werd ook errorHistory voorzien. Deze service maakt per uur een samenvatting van error groepen. Daarbij worden onder meer het aantal error logs, het aantal unieke gebruikers en het totaal aantal aanvragen naar de API waarbij een bepaalde error is voorgevallen bijgehouden. Door die informatie historisch te bewaren, kan het dashboard evoluties tonen in plaats van enkel de actuele toestand.

De timing wanneer errorHistory gemaakt wordt is belangrijk. De berekening moet gebeuren nadat de logs van het afgelopen uur verwerkt zijn en nadat error groepen beschikbaar zijn. Daarom werd onderzocht hoe deze verwerking het best via een achtergrondtaak kan worden aangestuurd.

### 7.4.3. Framework score job

De framework score werd initieel in de stored procedure “CalculateHealthHistory” berekend. Later bleek dat dit niet ideaal was. De score gebruikt informatie over health, methodes en error groepen. Omdat error groepen op het moment van de SQL-verwerking nog niet altijd aangemaakt of toegewezen zijn, werd de berekening naar de backend verplaatst. Met een .NET achtergrondtaak kan de timing beter gestuurd worden.

De framework score combineert meerdere signalen: CPU-gebruik, geheugengebruik, actieve BCM’s, methode scores en errors. Door deze berekening als service en job te modelleren, blijft de verantwoordelijkheid duidelijker gescheiden en kan de berekening afzonderlijk getest en aangepast worden.

Dit deel van de code zal pas uitgevoerd kunnen worden nadat de errors in groepen zijn ingdeeld. Om er voor te zorgen dat dit ook effectief gebeurt zal deze code worden aangesproken door een job die elk uur draait na het indelen van die errors. Voor een diepere kijk in de job kan men kijken naar de bijlage Code Fragment *6*.

## 7.5. Berekeningen en herbruikbare backendlogica

Niet alle berekeningen bleven in de database. Tijdens de uitwerking bleek dat bepaalde bewerkingen beter in de backend uitgevoerd konden worden. Dat was vooral het geval wanneer de timing belangrijk was, wanneer de berekening moeilijk te onderhouden werd in SQL of wanneer de logica beter testbaar moet zijn.

### 7.5.1. Method summaries en percentielen

De samenvatting van methodes in een geselecteerde tijdspanne werd initieel grotendeels via SQL berekend. Later werd de aanpak aangepast: de backend haalt enkel de nodige ruwe waarden op en voert de aggregaties daarna uit in een service. Daardoor blijft de query eenvoudiger en wordt de berekeningslogica beter testbaar. Percentielen worden in een aparte helper berekend om de Single Responsibility Principle beter te respecteren.

Eerst halen we de Ids van methodes op en kijken we of deze uitvoeringen heeft binnen de geselecteerde filters. Als dit het geval zullen er gemmideldes genomen worden van de duur van de uitvoeringen per methode, ook zal er gebruik gemaakt worden van de vooraf genoemde percientiele helper om deze te berekeke. Ook dit is per methode dit uitgevoerd is binnen de gegeven filters.

Hoe dit er in de code uitziet kan men bekijken in Bijlage Code Fragment *7* & *8*.

### 7.5.2. Caching

Caching is het tijdelijk bewaren van gegevens in het geheugen. Door dit te doen hoeft de backend niet bij elke aanvraag opnieuw dezelfde vraag naar de database te sturen. Dit kan laadtijden verkorten en belasting op de server beperken mits het juist wordt toegepast. Chaching is geschikt voor gegevens die niet vaak veranderen, zoals filterwaarden, “DaysWithData” wat later duidelijker uitgelgd zal worden, ...

Tijdens de performantieanalyse werd bekeken welke routes vaak worden aangeroepen en welke data relatief stabiel is. Vooral filterdata leent zich goed tot caching, omdat deze gegevens vaak hergebruikt worden door het dashboard en minder vaak veranderen dan ruwe logs. Bij het toepassen van caching moest wel gecontroleerd worden dat filters nog correct blijven werken wanneer een gebruiker combinaties van filterwaarden meegeeft.

Andere routes die gebruik maken van caching zijn “DaysWithData” van het stermodel en de originele BAM Database, deze informatie verandert enkel na middernacht en zal dus altijd zeer stabiel zijn. Ook routes die gebruik maken van het SeverityId zullen gecached worden, deze waarden, vooral die van error en information veranderen nooit en kunnen voor zeer lange tijd gecached worden.

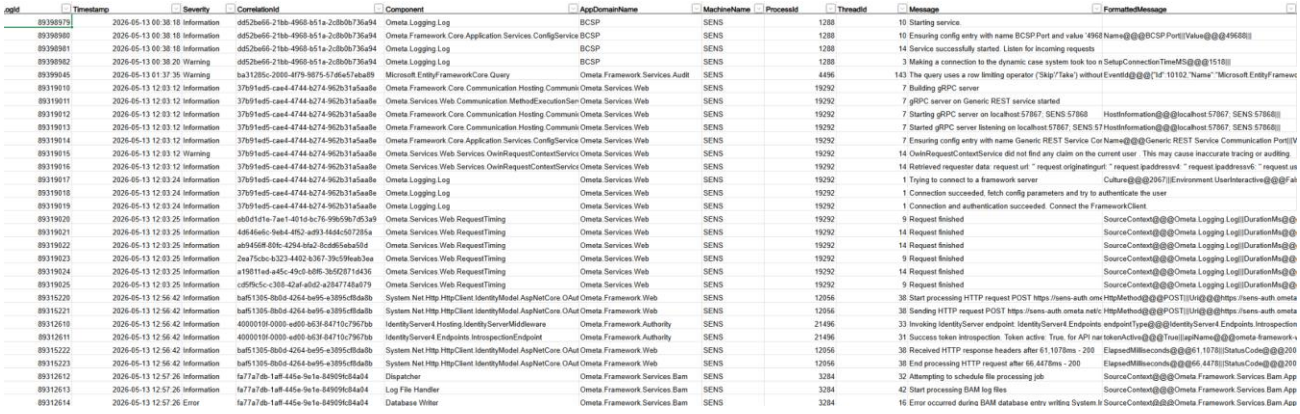
### 7.5.3. Excel-export met streaming en batching

Een bijkomende backendfunctionaliteit was de export naar Excel. Omdat exports grote datasets kunnen bevatten, werd onderzocht welke library geschikt was. ClosedXML werd bekeken, maar Aspose.Cells bleek beter geschikt voor grotere aantallen rijen en voor streaming. De export werd als aparte service opgezet zodat dezelfde logica op meerdere endpoints herbruikbaar is.

Bij deze export was vooral het geheugengebruik belangrijk. Zonder streaming zou eerst de volledige dataset uit de database opgehaald en in het geheugen opgebouwd worden voordat het Excel-bestand kan worden aangemaakt. Bij grote hoeveelheden data kan dit leiden tot een hoog geheugengebruik en mogelijk tot vertragingen of fouten.

Met streaming worden de gegevens daarentegen stapsgewijs verwerkt. De backend leest telkens een beperkt aantal records uit de database en schrijft deze onmiddellijk weg naar het Excel-bestand. Hierdoor hoeft de volledige dataset nooit tegelijk in het geheugen aanwezig te zijn. Dit maakt de export beter schaalbaar en vermindert de belasting op de server.

Om deze verwerking betrouwbaar te laten verlopen, moest de volgorde van de records deterministisch blijven. Daarom werd geordend op Timestamp en LogId. Zo blijft de volgorde stabiel, ook wanneer meerdere records hetzelfde tijdstip hebben. Deze aanpak sluit aan bij dezelfde principes als cursorpaginering: een vaste ordening en een voorspelbare verwerking van grote hoeveelheden data.



| ogId | Timestamp | Severity | CorrelationId | Component | AppDomainName | MachineName | ProcessId | ThreadId | Message | FormattedMessage |
|---|---|---|---|---|---|---|---|---|---|---|
| 89398979 | 2026-05-13 00:38:18 | Information | dd52be66-21bb-4968-b51a-2c8b0b736a94 | Ometa.Logging.Log | BCSP | SENS | 1288 | 10 | Starting service. | |
| 89398980 | 2026-05-13 00:38:18 | Information | dd52be66-21bb-4968-b51a-2c8b0b736a94 | Ometa.Framework.Core.Application.Services.ConfigService | BCSP | SENS | 1288 | 10 | Ensuring config entry with name BCSP:Port and value '4968 | Name=@@@BCSP:Port\|\|Value=@@@49688\|\| |
| 89398981 | 2026-05-13 00:38:18 | Information | dd52be66-21bb-4968-b51a-2c8b0b736a94 | Ometa.Logging.Log | BCSP | SENS | 1288 | 14 | Service successfully started. Listen for incoming requests | |
| 89398982 | 2026-05-13 00:38:20 | Warning | dd52be66-21bb-4968-b51a-2c8b0b736a94 | Ometa.Logging.Log | BCSP | SENS | 1288 | 3 | Making a connection to the dynamic case system took too n | SetupConnectionTimeMS@@@1518\|\| |
| 89399045 | 2026-05-13 01:37:35 | Warning | ba31285c-2080-4f79-9875-57d6e57eba89 | Microsoft.EntityFrameworkCore.Query | Ometa.Framework.Services.Audit | SENS | 4496 | 143 | The query uses a row limiting operator ('Skip'/'Take') without | EventId@@@{"Id":10102,"Name":"Microsoft.EntityFramewo |
| 89319010 | 2026-05-13 12:03:12 | Information | 37b91ed5-cae4-4744-b274-962b31a5aa8e | Ometa.Framework.Core.Communication.Hosting.Communi | Ometa.Services.Web | SENS | 19292 | 7 | Building gRPC server | |
| 89319011 | 2026-05-13 12:03:12 | Information | 37b91ed5-cae4-4744-b274-962b31a5aa8e | Ometa.Services.Web.Communication.MethodExecutionSer | Ometa.Services.Web | SENS | 19292 | 7 | gRPC server on Generic REST service started | |
| 89319012 | 2026-05-13 12:03:12 | Information | 37b91ed5-cae4-4744-b274-962b31a5aa8e | Ometa.Framework.Core.Communication.Hosting.Communi | Ometa.Services.Web | SENS | 19292 | 7 | Starting gRPC server on localhost:57867. SENS:57868 | HostInformation@@@localhost:57867; SENS:57868\|\| |
| 89319013 | 2026-05-13 12:03:12 | Information | 37b91ed5-cae4-4744-b274-962b31a5aa8e | Ometa.Framework.Core.Communication.Hosting.Communi | Ometa.Services.Web | SENS | 19292 | 7 | Started gRPC server listening on localhost:57867. SENS:57 | HostInformation@@@localhost:57867; SENS:57868\|\| |
| 89319014 | 2026-05-13 12:03:12 | Information | 37b91ed5-cae4-4744-b274-962b31a5aa8e | Ometa.Framework.Core.Application.Services.ConfigService | Ometa.Services.Web | SENS | 19292 | 7 | Ensuring config entry with name Generic REST Service Cor | Name@@@Generic REST Service Communication Port\|\|V |
| 89319015 | 2026-05-13 12:03:12 | Warning | 37b91ed5-cae4-4744-b274-962b31a5aa8e | Ometa.Services.Web.Services.OwinRequestContextServic | Ometa.Services.Web | SENS | 19292 | 14 | OwinRequestContextService did not find any claim on the current user . This may cause inaccurate tracing or auditing. | |
| 89319016 | 2026-05-13 12:03:12 | Information | 37b91ed5-cae4-4744-b274-962b31a5aa8e | Ometa.Services.Web.Services.OwinRequestContextServic | Ometa.Services.Web | SENS | 19292 | 14 | Retrieved requester data: request.url: '' request.originatingurl: '' request.ipaddressv4: '' request.ipaddressv6: '' request.us | |
| 89319017 | 2026-05-13 12:03:24 | Information | 37b91ed5-cae4-4744-b274-962b31a5aa8e | Ometa.Logging.Log | Ometa.Services.Web | SENS | 19292 | 1 | Trying to connect to a framework server | Culture@@@2067\|\|Environment.UserInteractive@@@Fal |
| 89319018 | 2026-05-13 12:03:24 | Information | 37b91ed5-cae4-4744-b274-962b31a5aa8e | Ometa.Logging.Log | Ometa.Services.Web | SENS | 19292 | 1 | Connection succeeded, fetch config parameters and try to authenticate the user | |
| 89319019 | 2026-05-13 12:03:24 | Information | 37b91ed5-cae4-4744-b274-962b31a5aa8e | Ometa.Logging.Log | Ometa.Services.Web | SENS | 19292 | 1 | Connection and authentication succeeded. Connect the FrameworkClient | |
| 89319020 | 2026-05-13 12:03:25 | Information | eb0d1d1e-7ae1-401d-bc76-99b59b7d53a9 | Ometa.Services.Web.RequestTiming | Ometa.Services.Web | SENS | 19292 | 9 | Request finished | SourceContext@@@Ometa.Logging.Log\|\|DurationMs@@ |
| 89319021 | 2026-05-13 12:03:25 | Information | 4d646e6c-9eb4-4f62-adf3-f4d4c507285a | Ometa.Services.Web.RequestTiming | Ometa.Services.Web | SENS | 19292 | 14 | Request finished | SourceContext@@@Ometa.Logging.Log\|\|DurationMs@@ |
| 89319022 | 2026-05-13 12:03:25 | Information | ab9456ff-80fc-4294-bfa2-8cdd65eba50d | Ometa.Services.Web.RequestTiming | Ometa.Services.Web | SENS | 19292 | 14 | Request finished | SourceContext@@@Ometa.Logging.Log\|\|DurationMs@@ |
| 89319023 | 2026-05-13 12:03:25 | Information | 2ea75cbc-b323-4402-b367-39c59feab3ea | Ometa.Services.Web.RequestTiming | Ometa.Services.Web | SENS | 19292 | 9 | Request finished | SourceContext@@@Ometa.Logging.Log\|\|DurationMs@@ |
| 89319024 | 2026-05-13 12:03:25 | Information | a19811ed-a45c-49c0-b8f6-3b5f2871d436 | Ometa.Services.Web.RequestTiming | Ometa.Services.Web | SENS | 19292 | 14 | Request finished | SourceContext@@@Ometa.Logging.Log\|\|DurationMs@@ |
| 89319025 | 2026-05-13 12:03:25 | Information | cd5f9c5c-c308-42af-a0d2-a2847748a879 | Ometa.Services.Web.RequestTiming | Ometa.Services.Web | SENS | 19292 | 9 | Request finished | SourceContext@@@Ometa.Logging.Log\|\|DurationMs@@ |
| 89315220 | 2026-05-13 12:56:42 | Information | baf51305-8b0d-4264-be95-e3895cf8da8b | System.Net.Http.HttpClient.IdentityModel.AspNetCore.OAut | Ometa.Framework.Web | SENS | 12056 | 38 | Start processing HTTP request POST https://sens-auth.omeg | HttpMethod@@@POST\|\|Uri@@@https://sens-auth.ometa |
| 89315221 | 2026-05-13 12:56:42 | Information | baf51305-8b0d-4264-be95-e3895cf8da8b | System.Net.Http.HttpClient.IdentityModel.AspNetCore.OAut | Ometa.Framework.Web | SENS | 12056 | 38 | Sending HTTP request POST https://sens-auth.ometa.net/c | HttpMethod@@@POST\|\|Uri@@@https://sens-auth.ometa |
| 89312610 | 2026-05-13 12:56:42 | Information | 4000010f-0000-ed00-b63f-84710c7967bb | IdentityServer4.Hosting.IdentityServerMiddleware | Ometa.Framework.Authority | SENS | 21496 | 33 | Invoking IdentityServer endpoint: IdentityServer4.Endpoints | endpointType@@@IdentityServer4.Endpoints.Introspection |
| 89312611 | 2026-05-13 12:56:42 | Information | 4000010f-0000-ed00-b63f-84710c7967bb | IdentityServer4.Endpoints.IntrospectionEndpoint | Ometa.Framework.Authority | SENS | 21496 | 31 | Success token introspection. Token active: True, for API nar | tokenActive@@@True\|\|apiName@@@ometa-framework-v |
| 89315222 | 2026-05-13 12:56:42 | Information | baf51305-8b0d-4264-be95-e3895cf8da8b | System.Net.Http.HttpClient.IdentityModel.AspNetCore.OAut | Ometa.Framework.Web | SENS | 12056 | 38 | Received HTTP response headers after 61.1078ms - 200 | ElapsedMilliseconds@@@61.1078\|\|StatusCode@@@200 |
| 89315223 | 2026-05-13 12:56:42 | Information | baf51305-8b0d-4264-be95-e3895cf8da8b | System.Net.Http.HttpClient.IdentityModel.AspNetCore.OAut | Ometa.Framework.Web | SENS | 12056 | 38 | End processing HTTP request after 66.4478ms - 200 | ElapsedMilliseconds@@@66.4478\|\|StatusCode@@@200 |
| 89312612 | 2026-05-13 12:57:26 | Information | fa77a7db-1aff-445e-9e1e-84909fc84a04 | Dispatcher | Ometa.Framework.Services.Bam | SENS | 3284 | 32 | Attempting to schedule file processing job | SourceContext@@@Ometa.Framework.Services.Bam.App |
| 89312613 | 2026-05-13 12:57:26 | Information | fa77a7db-1aff-445e-9e1e-84909fc84a04 | Log File Handler | Ometa.Framework.Services.Bam | SENS | 3284 | 42 | Start processing BAM log files | SourceContext@@@Ometa.Framework.Services.Bam.App |
| 89312614 | 2026-05-13 12:57:26 | Error | fa77a7db-1aff-445e-9e1e-84909fc84a04 | Database Writer | Ometa.Framework.Services.Bam | SENS | 3284 | 16 | Error occurred during BAM database entry writing System.In | SourceContext@@@Ometa.Framework.Services.Bam.App |

*Figuur 3: Gegenereerde Excel*

# 8. Kwaliteit, beveiliging en performantie

Naast de functionele endpoints werd ook gewerkt aan kwaliteit, beveiliging en performantie. Deze onderdelen zijn belangrijk omdat het dashboard gebruikt wordt bovenop dezelfde omgeving als het Ometa Framework. Het dashboard mag de werking van het framework dus niet merkbaar verstoren.

## 8.1. Authenticatie en autorisatie

De API werd afgestemd op de authenticatie- en autorisatie aanpak van het Ometa Framework. Voor beheerdersfunctionaliteit werd een beleid gebruikt dat controleert of de gebruiker of client de juiste administratorclaim heeft, deze zit in de token die de gebruiker met zijn aanvraag aan de API mee moet sturen. In debugcontext kan dit tijdelijk versoepeld worden, maar in productie moet het beleid een geldige token en de correcte claim vereisen.

## 8.2. Testen

Tijdens de stage werden unittesten toegevoegd. Dit zijn testen die de functionaliteit van kleine stukjes code, zoals helpers, services of functies, controleren. Ze geven een bepaalde invoer en gaan na of dit het verwachte resultaat geeft.

Het gebruik van unittesten maakt het makkelijker om fouten sneller te ontdekken. Dit is zeker handig na het aanpassen van functionaliteiten om te controleren of de aangepaste functionaliteit nog correct werkt.

Eerst werd geëxperimenteerd met een in-memory database, een tijdelijke database die volledig in het geheugen draait voor de duur van de testen, maar die bleek niet geschikt voor alle EF Core-functionaliteit en bulkachtige updates. Daarom werd overgeschakeld naar SQLite voor testen. De tests werden in een apart testproject geplaatst en de naamgeving werd aangepast na code review.

Er werden onder meer tests voorzien voor de helpers van onderdelen zoals tijdspannes, cursorpaginering en percentielberekening, maar ook voor endpoints. Tijdens het testen kwamen afwijkingen aan het licht, vooral rond paginering en databasevertaling. Dit leidde tot concrete verbeteringen in de backend.

## 8.3. Load testing en resource-analyse

Voor performantieonderzoek werd JMeter gebruikt om de API onder belasting te testen. Daarbij werden scenario's opgesteld die het gedrag van een normale gebruiker nabootsen. Omdat loadtests de metingen kunnen vertekenen wanneer ze op dezelfde server draaien als de API, werd onderzocht hoe JMeter vanaf een andere machine gebruikt kon worden.

Naast JMeter werd logman gebruikt om CPU- en geheugengebruik op te volgen van de API, de database en het Ometa Framework. Zo kon nagegaan worden of het dashboard een merkbare impact had op de omgeving. Voor databaseanalyse werden Query Store, execution plans en sp_Blitz gebruikt om zware queries, execution plans, indexgebruik en geheugen-impact te onderzoeken.

## 8.4. Logging van het ETL-proces

Om debugging en kwaliteitsopvolging van het ETL-proces op te volgen is er ook hier een vorm van logging toegepast die aansluit op die van het Ometa Framework zelf. Hierdoor kunnen we zien hoelang de verwerking duurt, waar het al dan niet fout ging en wanneer de verwerking gestart is. Deze logging is geplaatst op de cruciale momenten in de verwerking.

Deze manier van logging is ook opgeslagen in een stored procedure. Niet omdat dit sneller of efficienter is maar om de code appart te houden en herbruikbaar te maken. Wat er in de proocedure staat is te zien in Code Fragment *9*.

# 9. Toepassing, feedback en bijsturing

De gerealiseerde oplossing werd niet in isolatie ontwikkeld. Doorheen de stage werd regelmatig afgestemd met collega's, key users en andere stagiairs. Reeds in de analysefase werd via meetings onderzocht op basis van welke inzichten het dashboard het meest bruikbaar moest worden.
Ook tijdens de backendontwikkeling werd de API meerdere keren besproken en aangepast na review. Code werd nagekeken door collega's, opmerkingen werden verwerkt en er werd bijgestuurd wanneer uit feedback bleek dat bepaalde keuzes of functionaliteiten verfijnd moesten worden.
Daarnaast werd de oplossing ook concreet gebruikt in demo's en tests. Bij het ontvangen van data van klanten werd deze na verwerking in de database op het dashboard geplaatst ter analyse. Hieruit zijn problemen in het framework gevonden, configuratiefouten maar ook verkeerd gebruik van APIs.
Het dashboard met backend werd bovendien op een testomgeving bij Niras geplaatst. Tijdens die testfase werd ook technische ondersteuning voorzien, wat erop wijst dat de gerealiseerde componenten reeds in een werkende context gebruikt konden worden.

# 10. Samenvatting van de effectieve realisaties

Tijdens de stage werd een werkende technische basis gerealiseerd. Die basis bestaat uit een analytisch datamodel in SQL Server en een .NET 8 API als backend. De database werd ingericht om logging data gestructureerd, historisch en performant te verwerken. De backend werd opgebouwd rond gerichte endpoints, filters, cursor paginering, services, achtergrondtaken en aanvullende logica om dashboards en analyses te ondersteunen.

- analyse van bestaande logging data, BAM-rapporten en noden van gebruikers;
- uitwerking en iteratieve bijsturing van een stermodel;
- implementatie van fact- en dimensietabellen in SQL Server;
- ontwikkeling van stored procedures voor ETL-verwerking, methodHistory en healthHistory;
- batchgewijze verwerking van logs naar het stermodel;
- sanitisatiescript voor geanonimiseerde klantdata;
- onderzoek naar periodieke uitvoering via SQL Server Agent, Ometa-methodes of Elastic Jobs;
- optimalisatie met indexen, execution plans, Query Store en sp_Blitz;
- opstart en uitbouw van een .NET 8 API;
- endpoints voor logs, errors, error groepen, health, methods, solutions, search en export;
- DTO's om de vorm van de antwoorden van de API los te koppelen van het databasemodel;
- filtering, tijdspannes en cursorgebaseerde paginering met Timestamp en LogId;
- centralisatie van filterlogica via WhereIf, LogFilters en ApplyLogFilters;
- error groepering in C# met een Drain3-variant;
- errorHistory-service en bijhorende periodieke verwerking;
- framework score-service en achtergrondtaak;
- method summary-berekeningen en percentielberekening in de backend;
- Excel-export met aandacht voor streaming, batching en geheugengebruik;
- unit- en integratietesten in een apart testproject;
- load testing met JMeter en resource-analyse met logman;
- demo's, technische ondersteuning en gebruik met interne medewerkers en klantomgevingen.

# 11. Besluit

Tijdens deze stage werd een concrete technische basis gerealiseerd voor een verbeterde analyse van logging data binnen het Ometa Framework. De oplossing bestaat uit een analytisch datamodel in SQL Server, een verwerkingslaag via stored procedures en een .NET 8 API die de nodige data op een performante en bruikbare manier beschikbaar maakt voor het dashboard.

De realisaties tonen aan dat het project verder ging dan een verkennend ontwerp. Er werden werkende componenten gebouwd, geoptimaliseerd, getest, afgestemd met gebruikers en toegepast in demo's en klantgerichte testomgevingen. Daarbij werd niet alleen aandacht besteed aan functionaliteit, maar ook aan schaalbaarheid, resourcegebruik, beveiliging, testbaarheid en onderhoudbaarheid.

De gerealiseerde oplossing vormt daardoor een bruikbare basis voor verdere uitbouw binnen Ometa. Verdere stappen kunnen bestaan uit het verfijnen van AI-ondersteunde error groeperingen, het verder automatiseren van deployment, het uitbreiden van monitoring en het blijven optimaliseren van queries op basis van realistisch gebruik.

# 12. Bijlage

*1*

*Code Fragment 1: Batching logica*

```
/*BATCH INGESTION*/
WHILE 1=1
BEGIN
    IF OBJECT_ID('tempdb..#ParsedLog') IS NOT NULL
        DROP TABLE #ParsedLog;
    IF OBJECT_ID('tempdb..#Identifiers') IS NOT NULL
        DROP TABLE #Identifiers;
------------------------------------------------------------
-    CREATE TEMPORARY TABLE #ParsedLog
------------------------------------------------------------

    ;WITH Batch AS
    (
        SELECT TOP(@batchSize) *
        FROM [dbo].[Log] l --WITH (NOLOCK)
        WHERE NOT EXISTS
        (
            SELECT 1
            FROM [dbo].[Fact_Log] f
            WHERE f.[LogId] = l.[LogId]
        )
        AND l.[Timestamp] >= @startRangeTimestamp
        AND l.[Timestamp] <  @endRangeTimestamp
        AND
        (
            (
                @isNewLogVersion = 1
                AND l.[Message] LIKE N'Finished method execution %'
            )
            OR (
                @isNewLogVersion = 0
                AND l.[Message] IN
                (
                    'Finished method execution',
                    'Finished method execution of output extension',
                    'Finished method execution of input extension'
                )
            )
            OR l.[Message] = N'Request finished'
            OR l.[Severity] IN ('Error', 'Critical')
        )
    )

   -------------------
   -- Further Batch ingenstion Logic
   ----------------------

    -- LIMIT PARALLELL OPERATIONS
    OPTION (MAXDOP 2);

    -- Check not affected rows
    IF @@ROWCOUNT = 0
    BEGIN
        BREAK;
    END
```

*2*

*Code Fragment 2: Hoofd procedure*

```
CREATE OR ALTER PROCEDURE FullETL (
      @startRangeTimestamp DATETIMEOFFSET,
    @endRangeTimestamp  DATETIMEOFFSET
)
AS

DECLARE
    @processSchema sysname = OBJECT_SCHEMA_NAME(@@PROCID),
    @processName sysname = OBJECT_NAME(@@PROCID),
    @machineName NVARCHAR(128) = @@SERVERNAME,
    @appDomainName NVARCHAR(128) = APP_NAME(),
    @processId int = @@PROCID,
    @threadId int = @@SPID,
    @message NVARCHAR(255),
    @formattedMessage NVARCHAR(MAX),
    @correlationId UNIQUEIDENTIFIER = NEWID(),
    @duration float,
    @endTime datetimeoffset,
    @startedAt datetimeoffset;

BEGIN
DECLARE
-- First check if there is a contextRecordCount (differentiate old from new
logging)
      @isNewLogVersion BIT

-- We can see it is a new version of logging logic when the field
'ContextRecordCount' is present in the formattedMessage of a method execution
-------------------------------------------------------------------------
-- Normally we would execute a query to set the @isNewLogVersion variable
-------------------------------------------------------------------------

      EXEC dbo.FillDimAndFactTables
            @startRangeTimestamp = @startRangeTimestamp,
            @endRangeTimestamp = @endRangeTimestamp,
            @isNewLogVersion = @isNewLogVersion,
            @correlationId = @correlationId;

      EXEC dbo.CalculateMethodHistory
            @startRangeTimestamp = @startRangeTimestamp,
            @endRangeTimestamp = @endRangeTimestamp,
            @correlationId = @correlationId;

      EXEC dbo.CalculateHealthHistory
            @startRangeTimestamp = @startRangeTimestamp,
            @endRangeTimestamp = @endRangeTimestamp,
            @correlationId = @correlationId;
END
```

*3*

*Code Fragment 3: Voorbeeld Whitelist en Message anonimisatie regel*

```
DECLARE  @Redacted nvarchar(50) = '<Redacted>';
DECLARE @MessageMap table
(
    RuleOrder INT,
    MatchPattern     nvarchar(4000),
    ReplacementText  nvarchar(4000)
);
DECLARE @WhiteList table
(
    VariableName sysname
);

------------------------------------------------------------
-- MESSAGE SANITIZATION RULES
------------------------------------------------------------
INSERT INTO @MessageMap (RuleOrder, MatchPattern, ReplacementText)
VALUES
(
    1,
    'Fetching all claims for the user with user name % and id %',
    'Fetching all claims for the user with user name ' + @Redacted + ' and id
' + @Redacted
),
(
    2,
    'Generating system claims for the user % - % and all its users',
    'Generating system claims for the user ' + @Redacted + ' - ' + @Redacted
+ ' and all its users'
);
------------------------------------------------------------
-- FORMATTEDMESSAGE WHITELIST
------------------------------------------------------------
INSERT INTO @WhiteList (VariableName)
VALUES
    ('identity.person.id'),
    ('Method'),
    ('Object'),
    ('Profile'),
    ('PhysicalProfileName'),
    ('ContextRecordCount'),
    ('Duration'),
    ('DurationMs'),
    ('ReportJson');
```

*4*

*Code Fragment 4: Gebruikte indexes*

```
CREATE INDEX IX_Dim_Methods
ON dbo.Dim_Methods([MethodName], [ObjectName], [Profile]);

CREATE UNIQUE INDEX UX_Dim_Components_ComponentName
ON dbo.Dim_Components([ComponentName]);

CREATE UNIQUE INDEX UX_Dim_AppDomainNames_AppDomainName
ON dbo.Dim_AppDomainNames([AppDomainName]);

CREATE UNIQUE INDEX UX_Dim_Severity_Severity
ON dbo.Dim_Severity([Severity]);

CREATE UNIQUE INDEX UX_Dim_Errors_ErrorName
ON dbo.Dim_Errors([ErrorName]);

CREATE UNIQUE INDEX UX_Dim_TimestampHours
ON dbo.Dim_TimestampHours([TimeStampHour]);

CREATE UNIQUE INDEX UX_Dim_Users_Identifier
ON dbo.Dim_Users ([UserIdentifier]);

CREATE INDEX IX_MethodHistory_MethodId_TimestampHour
ON dbo.MethodHistory (MethodId, TimestampHour DESC)
INCLUDE
(
  AvgExecutionDuration,
  MedianExecutionDuration,
  [90thExecutionDuration],
  [95thExecutionDuration],
  AvgContextRecordCount
);

CREATE UNIQUE INDEX UX_ErrorHistory_TimestampHour_ErrorId
ON dbo.ErrorHistory (TimestampHour, ErrorId);

CREATE NONCLUSTERED INDEX NI_Fact_Log
ON [dbo].[Fact_Log] ([Timestamp],[MethodId]) INCLUDE
```

*5*

*Code Fragment 5: Pagineering*

```
public static async Task<PagedResult<T>> ApplyPaging<T>(
IQueryable<T> query,
        Expression<Func<T, DateTimeOffset>> timestampSelector,
        Expression<Func<T, long>> idSelector,
        CursorPaginationParams pagination,
        CancellationToken cancellationToken)
    {
        query = ApplyOrdering(query, timestampSelector, idSelector,
pagination.IsReverse);
```

```
        var decodedCursor = Decode(pagination.Cursor);
        var direction = decodedCursor?.Direction;
        var shouldReverseAfterFetch = false;

        if (decodedCursor is not null)
        {
            var ts = decodedCursor.Timestamp;
            var id = decodedCursor.Id;

            if (!pagination.IsReverse)
            {
                if (direction == CursorDirection.Next)
                {
                    query =
query.Where(BuildBeforePredicate(timestampSelector, idSelector, ts,
id));
                }
                else
                {
                    query =
query.Where(BuildAfterPredicate(timestampSelector, idSelector, ts, id))
                        .OrderBy(timestampSelector)
                        .ThenBy(idSelector);
                    shouldReverseAfterFetch = true;
                }
            }
            else
            {
                if (direction == CursorDirection.Next)
                {
                    query =
query.Where(BuildAfterPredicate(timestampSelector, idSelector, ts,
id));
                }
                else
                {
                    query =
query.Where(BuildBeforePredicate(timestampSelector, idSelector, ts,
id))
                        .OrderByDescending(timestampSelector)
                        .ThenByDescending(idSelector);
                    shouldReverseAfterFetch = true;
```

```
                }
            }
        }

        var fetchedItems = await query
            .Take(pagination.PageSize+1)
            .ToListAsync(cancellationToken);

        var hasExtraItem = fetchedItems.Count > pagination.PageSize;

        var pageItems = hasExtraItem ? [..
fetchedItems.Take(pagination.PageSize)] : fetchedItems;

        if (shouldReverseAfterFetch)
        {
            pageItems.Reverse();
        }

        var timestampAccessor = timestampSelector.Compile();
        var idAccessor = idSelector.Compile();

        var previousCursor = CreatePreviousCursor(pageItems, direction,
hasExtraItem, timestampAccessor, idAccessor);

        var nextCursor = CreateNextCursor(pageItems, direction,
hasExtraItem, timestampAccessor, idAccessor);

        // Build the paged response with next/previous cursors.
        return new PagedResult<T>(pageItems, nextCursor,
previousCursor);
    }
```

*6*

*Code Fragment 6: Framework score job*

```
public sealed class FrameworkScoreJob(
    IServiceScopeFactory scopeFactory,
    ILogger<FrameworkScoreJob> logger) : BackgroundService
{
    private readonly IServiceScopeFactory _scopeFactory = scopeFactory;
    private readonly ILogger<FrameworkScoreJob> _logger = logger;
```

```
    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        using var timer = new PeriodicTimer(TimeSpan.FromMinutes(1));
        while (await timer.WaitForNextTickAsync(stoppingToken))
        {
            var now = DateTimeOffset.Now;
            var nextRun = new DateTimeOffset(now.Year, now.Month,
now.Day,now.Hour, 10, 0, now.Offset);
            if (now >= nextRun)
            {
                nextRun = nextRun.AddHours(1);
            }
            var delay = nextRun - now;
            _logger.LogInformation("Next FrameworkScore job will run at
{NextRun}", nextRun);
            await Task.Delay(delay, stoppingToken);
            await RunForPreviousHourAsync(stoppingToken);
        }
    }
    private async Task RunForPreviousHourAsync(CancellationToken
cancellationToken)
    {
        using var scope = _scopeFactory.CreateScope();
        var frameworkScoreService = scope.ServiceProvider
            .GetRequiredService<IFrameworkScoreService>();
        var now = DateTimeOffset.Now;
        // Score the fully completed previous hour.
        var hourTo = new DateTimeOffset(now.Year, now.Month, now.Day,
now.Hour, 0, 0, now.Offset);
        var hourFrom = hourTo.AddHours(-1);
```

```
        _logger.LogInformation("Running FrameworkScore job for range
{HourFrom} - {HourTo}", hourFrom, hourTo);

        await
frameworkScoreService.CalculateFrameworkScoreAsync(hourFrom, hourTo,
cancellationToken);

    }

}
```

7

*Code Fragment 7 : Berekening van Samenvatting van methodes*

```
// Final response form with calculated percentiles
        var result = rawRows
            // Make groups in runtime
            .GroupBy(x => x.MethodId)
            .Select(g =>
            {
                var rows = g.ToList();
                var method = methods[g.Key];

                // Get all durations from a group
                var durations = rows
                    .Where(x => x.ExecutionDuration.HasValue)
                    .Select(x => x.ExecutionDuration!.Value)
                    .OrderBy(x => x)
                    .ToList();

                var durationCount = durations.Count;

                var maxDuration = durationCount > 0 ? durations[^1] :
0.0;

                var maxLogId = durationCount > 0
                    ? rows.Where(x => x.ExecutionDuration.HasValue &&
x.ExecutionDuration.Value == maxDuration)
                    .Select(x => x.LogId)
                    .FirstOrDefault()
                    : 0;

                return new MethodSummaryDto
                {
                    MethodId = method.MethodId,
```

```
                    MethodName = method.MethodName,
                    ObjectName = method.ObjectName,
                    Profile = method.Profile,
                    PhysicalProfile = method.PhysicalProfile ?? null,
                    AvgScore = (int)rows.Average(x => x.Score ?? 100),
                    TotalExecutions = rows.Count,
                    AvgExecutionDuration = durationCount > 0 ?
durations.Average() : 0.0,
                    MinDuration = durations.Count > 0 ? durations[0] :
0.0,
                    MaxDuration = maxDuration,
                    MaxLogId = maxLogId,
                    MedianExecutionDuration =
PercentileCalculator.Percentile(durations, 0.50),
                    NinetyExecutionDuration =
PercentileCalculator.Percentile(durations, 0.90),
                    NinetyFifthExecutionDuration =
PercentileCalculator.Percentile(durations, 0.95),
                    AvgContextRecordCount =
(int)Math.Round((decimal)rows.Average(x => x.ContextRecordCount ?? 0))
                };
            })
            .OrderBy(x => x.MethodId)
            .ToList();
```

*8*

*Code Fragment 8: Percentiele Helper*

```
public static double Percentile(IReadOnlyList<double> sortedValues,
double percentile)
    {
        if (percentile is < 0 or > 1)
        {
            throw new ArgumentOutOfRangeException(nameof(percentile),
"Percentile must be between 0 and 1.");
        }
        // Return 0 when the list is empty, because there is no value
to calculate from.
        if (sortedValues.Count == 0) return 0;

        // If there is only one value, that value is the percentile
result.
        if (sortedValues.Count == 1) return sortedValues[0];
```

```
        // Calculate the exact position of the percentile in the sorted
list.
        // Example:
        // for 50% in a list of 5 items, the position will fall in the
middle.
        var position = (sortedValues.Count - 1) * percentile;
        // Get the lower and upper indexes around that position.
        var lower = (int)Math.Floor(position);
        var upper = (int)Math.Ceiling(position);

        // If both indexes are the same, the percentile falls exactly
on one element.
        if (lower == upper) return sortedValues[lower];

        // Otherwise, interpolate between the lower and upper values.
        // This gives a smoother percentile result when the position is
between two indexes.
        var fraction = position - lower;
        return sortedValues[lower] + (sortedValues[upper] -
sortedValues[lower]) * fraction;
    }
```

*9*

*Code Fragment 9: Stored procedure met Logging logica*

```
CREATE OR ALTER PROCEDURE dbo.LogMessage
    @timestamp datetimeoffset,
    @severity nvarchar(20),
    @machineName nvarchar(128),
    @appDomainName nvarchar(128),
    @processId int,
    @threadId int,
    @correlationId UNIQUEIDENTIFIER,
    @message nvarchar(400),
    @formattedMessage nvarchar(MAX)
AS
BEGIN
    SET NOCOUNT ON;

    INSERT INTO dbo.[Log]
    (
        Timestamp,
        Severity,
        Component,
        MachineName,
        AppDomainName,
        ProcessId,
        ThreadId,
        CorrelationId,
        Message,
```

```
        FormattedMessage
    )
    VALUES
    (
        @timestamp,
        @severity,
        'Ometa.BAM.StarModel',
        @machineName,
        @appDomainName,
        @processId,
        @threadId,
        @correlationId,
        @message,
        @formattedMessage
    );
END;
```

*Figuur 4: Voorbeeld lijst van endpoints op swagger*