2024 - 2025

|  |
| --- |
|  |
| Worksense |
| Realisatiedocument |

Niels Van Nerum

Student Bachelor in de Toegepaste Informatica – Applicatieontwikkeling

Inhoudsopgave

[1. Inleiding 4](#_Toc199513778)

[1.1. Context van het project 4](#_Toc199513779)

[1.2. Doel van de stageopdracht 4](#_Toc199513780)

[1.3. Koppeling met het projectplan 4](#_Toc199513781)

[1.4. Overzicht van gebruikte technologieën en methodologieën 5](#_Toc199513782)

[1.5. Wat de lezer mag verwachten in dit document 5](#_Toc199513783)

[2. Analyse & Technologiekeuze 5](#_Toc199513784)

[2.1. Inleiding 5](#_Toc199513785)

[2.2. Onderzochte componenten 6](#_Toc199513786)

[2.2.1. Backend framework 6](#_Toc199513787)

[2.2.2. Frontend technologie 6](#_Toc199513788)

[2.2.3. Sensorintegratieplatform 7](#_Toc199513789)

[2.2.4. Beveiligingsmethode 7](#_Toc199513790)

[2.2.5. Database 8](#_Toc199513791)

[2.2.6. Hostingplatform 9](#_Toc199513792)

[2.3. Conclusie 9](#_Toc199513793)

[3. Projectmanagement & Werkwijze 10](#_Toc199513794)

[3.1. Scrum & Agile Methodologie 10](#_Toc199513795)

[3.2. Atlassian Stack in de praktijk 11](#_Toc199513796)

[4. Technische Architectuur 13](#_Toc199513797)

[4.1. Overzicht architectuur 14](#_Toc199513798)

[4.2. Backend-architectuur (Spring Boot) 14](#_Toc199513799)

[4.3. Frontend-overzicht (Angular) 15](#_Toc199513800)

[5. Beveiliging & Authenticatie 16](#_Toc199513801)

[5.1. OAuth2 in de praktijk 16](#_Toc199513802)

[5.2. Beveiliging van API’s en data 20](#_Toc199513803)

[6. Database & Data Management 22](#_Toc199513804)

[6.1. Databasemodel 23](#_Toc199513805)

[6.2. Spring Data JPA 25](#_Toc199513806)

[7. Sensorintegratie & ThingsBoard 26](#_Toc199513807)

[7.1. Wat is ThingsBoard 26](#_Toc199513808)

[7.2. Koppeling met sensoren 27](#_Toc199513809)

[8. Visualisatie & Gebruikerservaring 30](#_Toc199513810)

[8.1. Tijdlijnvisualisaties 30](#_Toc199513811)

[8.2. Alarmfunctionaliteit 31](#_Toc199513812)

[8.3. Mobiele toegankelijkheid 31](#_Toc199513813)

[9. Testing & Kwaliteitsborging 32](#_Toc199513814)

[9.1. Unittests 32](#_Toc199513815)

[9.2. Integratietests 33](#_Toc199513816)

[9.3. Handmatige testen & tools 33](#_Toc199513817)

[9.4. CI/CD 33](#_Toc199513818)

[10. Cloudhosting & Infrastructuur (AWS) 34](#_Toc199513819)

[10.1. Overzicht gebruikte AWS-diensten 34](#_Toc199513820)

[10.2. Architectuuroverzicht en dataflow 34](#_Toc199513821)

[11. Besluit 35](#_Toc199513822)

[Literatuurlijst 37](#_Toc199513823)

[AI-gebruik in dit document 38](#_Toc199513824)

[Bijlagen 39](#_Toc199513825)

# Inleiding

## Context van het project

Dit document beschrijft de uitwerking van mijn inhouse-stageproject bij Faros. Het project kwam voort uit een experimenteel prompt via ChatGPT, waarbij Faros de ambitie had om via metingen inzicht te verwerven in hoe bepaalde omgevingsfactoren op kantoor evolueren. Deze inzichten zouden het bedrijf in staat stellen gerichte maatregelen te nemen om zowel de ecologische voetafdruk te verkleinen als het welzijn van de medewerkers te verbeteren.

De stageopdracht maakte deel uit van een breder duurzaamheids- en welzijnsinitiatief binnen de organisatie, met als centrale doelstelling het verkrijgen van meer inzicht in de impact van het binnenklimaat op de kantooromgeving.

Voor meer achtergrondinformatie verwijs ik naar het stagevoorstel dat bij aanvang van het project werd opgesteld. Zie bijlage A: Stagevoorstel Faros 2025 – Meten is weten.

## Doel van de stageopdracht

Op basis van een voorafgaande AI-analyse werd onderzocht welke omgevingsfactoren het grootste effect hebben op het welzijn van werknemers. Hieruit werden zes kernparameters geselecteerd die gedurende het project continu werden gemonitord en geanalyseerd:

* Temperatuur
* CO₂-concentratie
* Luchtvochtigheid
* Lichtintensiteit
* Geluidsniveau (in decibel)
* Aanwezigheid

De meetgegevens moesten niet alleen visueel weergegeven worden op een tijdlijn, maar ook geaggregeerd worden tot gemiddelden over intervallen van 4, 12 en 24 uur. Daarnaast moest een alarmsysteem worden geïmplementeerd, dat bij het overschrijden van vooraf ingestelde drempelwaarden automatisch een waarschuwing verstuurt via e-mail, sms of notificatie. De beveiliging van deze data was cruciaal: enkel geautoriseerde gebruikers mochten toegang hebben tot deze informatie.

## Koppeling met het projectplan

Het volledige project werd ontwikkeld op basis van een vooraf opgesteld projectplan. Dit plan diende als leidraad voor het gehele ontwikkeltraject en bevatte duidelijke mijlpalen, sprintindelingen en technische keuzes.

Het projectplan is terug te vinden in bijlage B: Projectplan Faros 2025.

## Overzicht van gebruikte technologieën en methodologieën

Voor de technische realisatie werd gekozen voor een moderne, professionele technologie-stack:

* Spring Boot (Java) voor de backend
* Angular of Flutter voor de frontend
* ThingsBoard als platform voor sensorintegratie
* OAuth2 voor veilige authenticatie
* Een relationele database (vrij te kiezen)
* AWS of Azure voor cloudhosting en infrastructuurbeheer

De ontwikkeling gebeurde volgens de Agile/Scrum-methodiek, ondersteund door de Atlassian-stack (Jira, Confluence, Bitbucket), wat zorgde voor een iteratieve aanpak, transparante samenwerking en duidelijke documentatie.

## Wat de lezer mag verwachten in dit document

In dit document geef ik een volledig overzicht van de realisatie van mijn stageproject bij Faros. Ik begin met een toelichting op de gehanteerde werkwijze en methodologie (Scrum/Agile) en licht toe hoe de Atlassian-tools deze aanpak ondersteunden.

Daarna bespreek ik de technische opbouw van het project, beginnend met de algemene architectuur. Vervolgens wordt dieper ingegaan op de gebruikte technologieën zoals Spring Boot, OAuth2, Java en de frontend-oplossing (Angular of Flutter). Ook het databeheer met JPA en het onderliggende databasemodel komen uitgebreid aan bod.

Verder behandel ik de sensorintegratie via ThingsBoard en hoe de sensordata werd verwerkt. Visualisaties op tijdlijnen, de configuratie van alarmfunctionaliteiten en de mobiele toegankelijkheid van het systeem worden eveneens besproken.

Ook testing en kwaliteitsborging komen aan bod, met aandacht voor unit- en integratietests, CI/CD en versiebeheer. Vervolgens leg ik uit hoe de applicatie werd uitgerold op AWS, inclusief hosting, monitoring en schaalbaarheid.

In een afsluitend hoofdstuk reflecteer ik op de belangrijkste uitdagingen, de oplossingen die ik daarvoor vond, en de onderdelen van het project waar ik bijzonder trots op ben. Het document sluit af met een evaluatie van het eindresultaat en aanbevelingen voor mogelijke toekomstige uitbreidingen.

Doorheen het document wordt zowel aandacht besteed aan technische diepgang als aan de professionele aanpak van softwareontwikkeling.

# Analyse & Technologiekeuze

## Inleiding

Het analyseren van tools en technologieën is van essentieel belang in een softwareproject. Niet alleen om de meest geschikte oplossingen te kiezen voor de specifieke context, maar ook om binnen het team duidelijke afspraken te maken over welke tools en technologieën gebruikt zullen worden. Dit bevordert de samenwerking aanzienlijk. Zelfs wanneer bepaalde keuzes vooraf vastliggen, blijft het zinvol om stil te staan bij het waarom van deze keuzes: waarom gebruiken we net deze tools?

## Onderzochte componenten

In de volgende paragrafen bespreken we de verschillende ontwikkelomgevingen, frameworks en platformen die tijdens dit project werden overwogen. Ook technisch meer geavanceerde onderwerpen zoals databases, cloudomgevingen en beveiliging komen aan bod.

### Backend framework

Hoewel de keuze voor Java in dit project grotendeels vastlag – Faros is immers een gespecialiseerd Java-consultancybedrijf – loont het de moeite om de inhoudelijke waarde van deze keuze toe te lichten.

Java krijgt soms de reputatie een 'zware' taal te zijn vanwege zijn langere compileertijden, strikte typevereisten en verplichte objectgeoriënteerde aanpak. Voor kleinere of snellere projecten kan dit als beperkend aanvoelen. Toch zijn het net deze eigenschappen die Java zo geschikt maken voor grootschalige, langdurige softwareprojecten. De taal biedt robuustheid, voorspelbaarheid en een rijk ecosysteem aan volwassen libraries en frameworks. Binnen Faros wordt Java dan ook actief ingezet als stabiele basis voor het bouwen van schaalbare en onderhoudsvriendelijke backends.

Een interessant alternatief is Quarkus, een modern framework dat inspeelt op enkele traditionele beperkingen van Java. Quarkus biedt onder meer snellere opstarttijden, een kleinere geheugengebruik en ondersteuning voor native executables via GraalVM. Bovendien stimuleert het live coding, waarbij wijzigingen in de code vrijwel onmiddellijk zichtbaar zijn tijdens het ontwikkelproces — wat de ontwikkelervaring aanzienlijk verbetert.

Tijdens mijn stage volgde ik een workshop over Quarkus, waarin deze voordelen duidelijk naar voren kwamen. Het is een innovatief en veelbelovend framework dat goed inspeelt op de noden van moderne softwareontwikkeling.

Toch kozen we bewust voor Spring Boot binnen dit project. Deze keuze werd onderbouwd door de ruime ervaring van de ontwikkelaars bij Faros, waardoor we konden bouwen op bestaande expertise. Daarnaast beschikt Spring Boot over een grote community en uitgebreide documentatie. Waar Quarkus nog in opmars is, is Spring Boot reeds jaren een gevestigde waarde in de sector. Voor een stageproject waar betrouwbaarheid en kennisdeling centraal stonden, bleek Spring Boot de meest solide en toekomstgerichte keuze.

### Frontend technologie

Voor de frontend gold een gelijkaardige redenering als voor de backend. Binnen Faros is er uitgebreide expertise in Angular, wat deze technologie tot de voor de hand liggende keuze maakte. Daarnaast had ik persoonlijk reeds een Angular-certificaat behaald via Udemy, wat de keuze versterkte om deze kennis verder in de praktijk toe te passen.

Hoewel Angular en Java onafhankelijk van elkaar functioneren, vormen ze samen een vaak voorkomende en goed werkende combinatie. Flutter biedt in theorie een hogere performance, maar de documentatie en ondersteuning voor Angular zijn momenteel uitgebreider.

We kozen uiteindelijk voor Angular vanwege de aanwezige expertise binnen het team en de bredere ondersteuning die het framework biedt.

### Sensorintegratieplatform

De keuze voor ThingsBoard werd grotendeels bepaald door de interesse binnen het bedrijf om hiermee meer ervaring op te doen. Toch bleek het ook inhoudelijk een logische keuze. ThingsBoard maakt het mogelijk om verschillende platformen en sensoren te integreren. Dit is bijzonder interessant in situaties waarin een klant al sensoren heeft geïnstalleerd via een ander systeem, zoals The Things Stack. Dankzij de compatibiliteit van ThingsBoard kunnen deze bestaande sensoren eenvoudig worden overgenomen en geïntegreerd.

Hoewel ThingsBoard relatief duur is, biedt het een breed scala aan functionaliteiten. Zo beschikt het onder andere over een ingebouwde MQTT-server en een onderliggende database. Bovendien is het platform zeer gebruiksvriendelijk: het laat zich eenvoudig opzetten en werkt quasi onmiddellijk, wat het bijzonder geschikt maakt voor snelle integratie en demonstraties.

### Beveiligingsmethode

Beveiliging is een cruciaal onderdeel van onze applicatie. De verwerking van gegevens brengt potentiële veiligheidsrisico’s met zich mee, zoals het blootleggen van aanwezigheidspatronen van personeel. Een datalek zou bijvoorbeeld misbruikt kunnen worden voor fysieke inbraak. Daarom was het essentieel om een veilige, schaalbare en betrouwbare structuur voor authenticatie en autorisatie te implementeren.

We onderzochten drie mogelijke oplossingen:

* **JWT (JSON Web Tokens)**: Meestal gebruikt in combinatie met een eigen gebruikersbeheer.
* **OAuth2**: Werkt met een externe identity provider (zoals Google, Auth0 of Keycloak) die tokens uitreikt.
* **Basic Authentication**: Eenvoudige combinatie van gebruikersnaam en wachtwoord die bij elk verzoek wordt meegestuurd.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Criteria | Gewicht (%) | OAuth2 | JWT | Basic Auth |
| Beveiligingsniveau | 30% | 9 | 7 | 4 |
| Schaalbaarheid | 25% | 9 | 7 | 3 |
| Beheerbaarheid | 20% | 9 | 5 | 2 |
| Complexiteit implementatie | 15% | 6 | 8 | 9 |
| Community & Ondersteuning | 10% | 8 | 8 | 6 |
| Totaalscore | **100%** | **8.3** | 6.9 | 4.2 |

We kozen uiteindelijk voor een OAuth2-provider: **Keycloak** tijdens de ontwikkeling, en **Amazon Cognito** voor de productieomgeving. Cognito sloot bovendien mooi aan op onze cloudinfrastructuur, waarover later in dit document meer uitleg volgt.

### Database

De keuze van een geschikte database was een belangrijk beslissingspunt waarin we volledige vrijheid kregen. We bekeken verschillende modellen grondig om een weloverwogen keuze te maken. De eerste belangrijke knoop was: kiezen we voor een **relationeel**, **document-gebaseerd** of **tijdgebaseerd** model?

* **Relationele databases** bieden krachtige querymogelijkheden, ondersteunen complexe transacties en garanderen sterke dataconsistentie en integriteit. Nadelen zijn echter dat ze minder goed omgaan met ongestructureerde data, een vooraf vastgelegde structuur vereisen en minder flexibel zijn qua schaalbaarheid.
* **Document-gebaseerde databases** slaan gegevens op in JSON- of BSON-formaat. Ze zijn geoptimaliseerd voor (semi-)ongestructureerde data en bieden uitstekende prestaties bij het lezen en schrijven van documenten. Daarentegen zijn ze minder geschikt voor complexe transacties en bieden ze minder dataconsistentie dan relationele databases.
* **Tijdgebaseerde databases** zijn ontworpen voor het opslaan en analyseren van tijdsgebonden gegevens, zoals sensormetingen. Ze worden vaak gebruikt in IoT-toepassingen, monitoring en logging. Bekende voorbeelden zijn InfluxDB, TimescaleDB en Amazon Timestream. De keerzijde is dat ze minder geschikt zijn voor relationele gegevensmodellen en soms beperkt zijn in functionaliteit buiten tijdreeksen.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Criteria | Gewicht (%) | Relationeel | Document-based | Time-based (TimescaleDB) |
| Tijdsreeks-optimalisatie | 30% | 5 | 4 | 9 |
| Schaalbaarheid bij veel meetdata | 25% | 6 | 8 | 9 |
| Opzet & onderhoud | 15% | 7 | 7 | 7 |
| Geavanceerde query’s & aggregaties | 20% | 9 | 6 | 8 |
| Flexibiliteit & uitbreidbaarheid | 10% | 5 | 9 | 7 |
| Totaalscore | **100%** | **6.4** | **6.6** | **8.4** |

Onze keuze viel op **TimescaleDB**, een extensie op PostgreSQL. Deze hybride oplossing combineert de sterktes van relationele databases met de voordelen van tijdgebaseerde opslag. Binnen TimescaleDB kan een tabel worden gemarkeerd als een **hypertabel**. Deze wordt geoptimaliseerd voor tijdreeksen, waarbij de primaire sleutel gebaseerd is op een timestamp die niet uniek hoeft te zijn. Op deze tabellen kunnen krachtige tijdsgebonden query’s worden uitgevoerd met betere performance.

Later in dit document bespreek ik TimescaleDB verder in detail, aangezien het een belangrijk onderdeel van het project vormde.

### Hostingplatform

Voor de hosting van onze applicatie vergeleken we **Amazon Web Services (AWS)** met **Microsoft Azure**. Beide platformen zijn krachtig, veelzijdig en marktleiders binnen het cloudlandschap. Toch kozen we uiteindelijk voor AWS als het meest geschikte platform voor dit project.

De doorslaggevende factor was de reeds aanwezige expertise binnen Faros, wat resulteerde in een vlotte opstart en een beperkte leercurve. Daarnaast bood AWS voldoende ondersteuning op het gebied van monitoring, databanken en schaalbaarheid.

|  |  |  |  |
| --- | --- | --- | --- |
| Criteria | Gewicht | AWS | Azure |
| Dienstenaanbod | 25% | 9 | 8 |
| Schaalbaarheid | 20% | 9 | 8 |
| Prijsstructuur | 15% | 7 | 8 |
| Gebruiksgemak | 15% | 6 | 8 |
| Interne expertise Faros | 25% | 10 | 5 |
| Totaalscore | **100%** | **8.4** | **7.5** |

AWS bleek voor dit project de meest logische keuze, dankzij de bestaande kennis in het team en de sterke ondersteuning voor schaalbare en betrouwbare cloudoplossingen.

## Conclusie

Veel van de keuzes die we tijdens dit project maakten, zijn sterk beïnvloed door de aanwezige expertise binnen Faros. Dit was geen toevallige omstandigheid, maar een bewuste strategie: door aan te sluiten bij de kennis en ervaring van de professionals binnen het bedrijf, kregen wij als studenten de kans om maximaal bij te leren – niet enkel over technologie, maar ook over het hanteren van een professionele en doordachte aanpak.

Het leerproces stond centraal, zowel voor Faros als voor ons. Daarom werkten we waar mogelijk volgens best practices en industry standards. Verderop in dit document wordt duidelijk hoe deze keuzes hebben bijgedragen aan de kwaliteit, robuustheid en schaalbaarheid van onze oplossing.

# Projectmanagement & Werkwijze

Voor dit project kozen we ervoor om te werken volgens de **Agile/Scrum-methodologie** – een aanpak die veelvuldig wordt toegepast binnen IT-projecten en ook bij Faros geldt als gevestigde standaard.

We werden begeleid door **Arne**, onze projectcoördinator, die ons actief ondersteunde bij het correct en consequent toepassen van de Scrum-principes. Zijn rol was cruciaal in het structureren van onze samenwerking en het behouden van een duidelijk ritme binnen de verschillende sprints.

In het volgende hoofdstuk geef ik een korte toelichting op wat Agile en Scrum precies inhouden, gevolgd door een bespreking van hoe deze werkwijze ons project concreet heeft beïnvloed. Aansluitend behandel ik de **Atlassian Stack** – een verzameling tools die ons hielp om Scrum efficiënt te implementeren en op te volgen.

## Scrum & Agile Methodologie

Scrum ontstond nog vóór Agile, als een praktisch framework dat organisaties helpt hun projecten beter te organiseren. Het werkt met kleine, zelfsturende teams van ongeveer drie tot negen personen. Deze teams werken in korte, afgebakende periodes – zogenaamde **sprints** – waarin telkens duidelijke doelstellingen worden geformuleerd. Zo wordt een groot project opgedeeld in kleinere, beheersbare onderdelen. Scrum vergemakkelijkt de planning en maakt het mogelijk om na elke sprint de voortgang te demonstreren aan de **product owner**. (Timon, 2020)

Agile ontstond later, specifiek binnen de wereld van softwareontwikkeling. Het introduceerde een reeks waarden en principes, zoals: **individuen en interacties boven processen en tools, werkende software boven uitgebreide documentatie, samenwerking met de klant boven contractonderhandelingen, en reageren op verandering boven het volgen van een plan**. Agile ontstond als antwoord op de vaak logge en inflexibele structuren van grote organisaties. (Timon, 2020)

Afbeelding met tekst, schermopname, diagram, ontwerp

Door AI gegenereerde inhoud is mogelijk onjuist.

1. Agile Iteraties

Wat is het belangrijkste verschil tussen beide? **Scrum is een concreet framework**, terwijl **Agile een mindset** is: een manier van denken over hoe succesvolle projecten moeten verlopen.

Ook binnen ons project pasten we deze principes toe. Zo hielden we dagelijks een **stand-upmeeting**, waarbij teamleden in een korte, staande bespreking hun voortgang deelden. Dit zorgde voor transparantie, bevorderde onderlinge samenwerking en gaf de coördinator de kans om tijdig bij te sturen.

We werkten in **wekelijkse sprints**. Aan het einde van elke sprint gaven we een demo en hielden we een **sprint planning** voor de volgende sprint. Daarnaast organiseerden we tweewekelijks een **retrospective**, waarbij we de afgelopen sprint evalueerden aan de hand van drie categorieën:

* **Glad** – Wat verliep goed?
* **Sad** – Wat kon beter?
* **Mad** – Wat frustreerde ons of moet dringend veranderen?

We noteerden onze inzichten op post-its en bespraken ze in teamverband.

Ook de Agile-mindset kwam duidelijk tot uiting: we waren flexibel in onze technologische keuzes, pasten prioriteiten aan waar nodig, en beperkten de documentatie tot het strikt noodzakelijke. Onderlinge kennisdeling werd gestimuleerd via regelmatige teammeetings.

## Atlassian Stack in de praktijk

Om de Agile/Scrum-werkwijze optimaal te ondersteunen, maakten we gebruik van tools uit de **Atlassian Stack**, met name **Jira**, **Confluence** en **Bitbucket**. Deze tools integreren naadloos met elkaar en worden algemeen beschouwd als de industry standard binnen moderne softwareontwikkeling.

Met **Jira** beheerden we onze sprints via **tickets**, waarbij elke taak of functionaliteit werd voorgesteld als een afzonderlijk item. Elk ticket bevatte een duidelijke omschrijving en **acceptatiecriteria**: een lijst met vereisten waaraan voldaan moest zijn voordat het ticket als “afgewerkt” werd beschouwd. Tickets werden toegewezen aan teamleden en doorliepen verschillende fasen: **To Do**, **In Progress**, **Waiting for Feedback**, en **Done**. Dit gaf ons voortdurend een helder overzicht van de voortgang.

Daarnaast beheerden we in Jira een overzichtelijke **backlog**: een lijst van alle toekomstige taken en gewenste functionaliteiten. Tijdens de **sprintplanning** selecteerden we relevante tickets uit deze backlog om op te nemen in de volgende sprint.

Afbeelding met tekst, nummer, software, schermopname

Door AI gegenereerde inhoud is mogelijk onjuist.

1. Project backlog

**Confluence** is een platform voor het schrijven en delen van documentatie binnen teams. Het stelt gebruikers in staat om eenvoudig een onderwerp aan te maken, bijvoorbeeld *Jira Guidelines*, en dit vervolgens aan te vullen met tekst, tabellen, afbeeldingen of diagrammen. De kracht van Confluence ligt in de samenwerking: documenten kunnen worden bekeken, aangepast en vooral ook gerefereerd in Jira, wat zorgt voor een sterke integratie tussen documentatie en projectmanagement.

Voor ons project speelde Confluence een belangrijke rol. Het stelde ons in staat om complexere of technische onderwerpen uitgebreider toe te lichten. Zo kon Faros onze bevindingen achteraf herbekijken en konden teamleden elkaar beter volgen door te begrijpen waar anderen mee bezig waren.

Onderstaand vindt u een illustratie van hoe Confluence eruitziet in de praktijk.

Afbeelding met tekst, schermopname, nummer, software

Door AI gegenereerde inhoud is mogelijk onjuist.

1. Confluence met documentatie

**Bitbucket** is Atlassian’s versiebeheertool en lijkt sterk op GitHub, maar met een nauwere integratie binnen de rest van de Atlassian Stack. Het stelt ons in staat om Git-versiebeheer te gebruiken, terwijl we tegelijkertijd rechtstreeks vanuit Jira kunnen werken. Zo kunt u bijvoorbeeld vanuit een ticket meteen een nieuwe branch aanmaken, wat het ontwikkelproces versnelt en vereenvoudigt.

Git zelf is een krachtig hulpmiddel dat ontwikkelaars in staat stelt om verschillende versies van hun code bij te houden. Het maakt het mogelijk om wijzigingen te vergelijken, code afzonderlijk te ontwikkelen en deze later samen te voegen in de main branch. Dankzij pull requests en code reviews kunnen wijzigingen grondig worden bekeken voordat ze definitief worden geïntegreerd, wat de kwaliteit en stabiliteit van de code ten goede komt.

Bitbucket bood ons een gestroomlijnde manier om samen te werken aan code, fouten te voorkomen en efficiënt wijzigingen te beheren.

Onderstaand vindt u een afbeelding van Bitbucket als visuele ondersteuning.

Afbeelding met tekst, schermopname, nummer, Lettertype

Door AI gegenereerde inhoud is mogelijk onjuist.

1. BitBucket

Deze drie tools – Jira, Confluence en Bitbucket – vormden samen een krachtig geheel dat ons in staat stelde om Scrum en Agile op een gestructureerde en efficiënte manier toe te passen. Dankzij hun integratie konden we onze workflows optimaliseren, communicatie verbeteren en documentatie centraal beheren. Het mag duidelijk zijn dat een groot deel van het succes van ons project te danken is aan deze tools. Niet voor niets worden ze beschouwd als industry standards: ze maken het beheer van projecten niet alleen eenvoudiger, maar ook een stuk effectiever.

# Technische Architectuur

Een heldere en goed doordachte architectuur vormt de ruggengraat van elke succesvolle softwaretoepassing. Zij bepaalt niet alleen hoe de verschillende componenten van het systeem met elkaar communiceren, maar zorgt ook voor schaalbaarheid, onderhoudbaarheid en flexibiliteit tijdens de ontwikkeling.

In dit hoofdstuk geef ik een diepgaand overzicht van de technische opbouw van ons project. Ik leg uit hoe we de verschillende systemen en lagen logisch van elkaar hebben gescheiden en hoe dit bijdraagt aan een overzichtelijke en robuuste structuur.

Daarnaast wordt beschreven hoe de data door de applicatie stroomt: van de sensoren via ThingsBoard naar de backend, waar deze wordt verwerkt en beveiligd, tot de frontend, waar de gegevens op een gebruiksvriendelijke manier worden gepresenteerd aan de gebruiker. Ook de gekozen ontwikkelprincipes, patronen en frameworks komen hierbij aan bod.

## Overzicht architectuur

We beginnen bij het fysieke niveau van onze architectuur: de sensoren. Deze draaien op C++-code en zijn verantwoordelijk voor het uitlezen (pollen) van sensordata. De gegevens worden vervolgens verzonden via het MQTT-protocol, een lichtgewicht communicatieprotocol dat werkt volgens het publish-subscribe-model. Een centrale MQTT-broker beheert deze communicatie: apparaten (publishers) versturen data naar een specifiek topic, waarop andere systemen (subscribers) zich kunnen abonneren om deze data te ontvangen.

In ons project maken we gebruik van **ThingsBoard** als IoT-platform. ThingsBoard bevat een ingebouwde MQTT-broker waarop onze sensoren hun data publiceren. Het platform fungeert als subscriber, ontvangt de inkomende data en visualiseert deze via eigen dashboards.

Onze **Spring Boot-backend** communiceert vervolgens met ThingsBoard via de beschikbare REST API. Deze API gebruiken we om sensorwaarden en eigenschappen op te halen, op basis van een zelfgedefinieerd polling-interval. De opgehaalde data wordt opgeslagen in een **TimescaleDB**, een relationele tijdreeksdatabase gebaseerd op PostgreSQL.

De **frontend**, gebouwd met Angular, maakt gebruik van deze API-endpoints om scenario-specifieke, gestructureerde gegevens op te halen. Deze gegevens worden visueel gepresenteerd via grafieken en tabellen. Daarnaast kan de frontend ook data terugsturen, bijvoorbeeld voor het wijzigen van alarminstellingen.

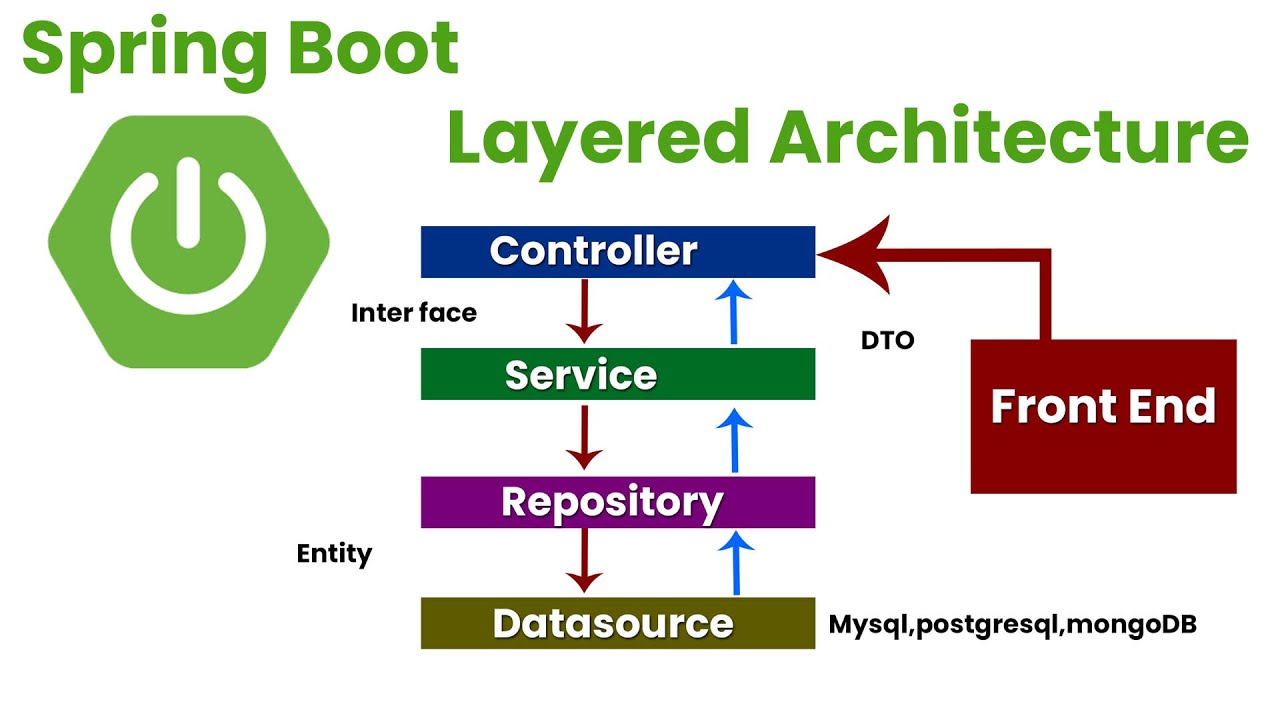
Op deze manier ontstaat een vloeiende datastroom van sensor tot eindgebruiker, ondersteund door een robuuste architectuur die schaalbaar, onderhoudbaar en efficiënt is.

## Backend-architectuur (Spring Boot)

Voor de backend van onze applicatie kozen we voor **Spring Boot**, een krachtig en populair framework binnen het Java-ecosysteem. Spring Boot maakt het mogelijk om snel en efficiënt RESTful API’s op te zetten, met een sterke focus op gebruiksgemak, schaalbaarheid en onderhoudbaarheid. Aangezien Faros veel ervaring heeft met Java en Spring Boot, was dit een logische keuze die ons toeliet te leren van de aanwezige expertise.

Onze backend is opgebouwd volgens het eerder besproken **layered architecture-principe**. Dit zorgt voor een duidelijke scheiding van verantwoordelijkheden en maakt het systeem eenvoudig uitbreidbaar. De structuur bestaat uit drie hoofdcomponenten:

* **Controller Layer**  
  Deze laag bevat de REST-controllers die inkomende HTTP-verzoeken afhandelen. Hier worden de endpoints gedefinieerd die toegankelijk zijn voor de frontend en andere externe systemen. De controllers zorgen voor de routering van data en sturen inkomende requests door naar de servicelaag.
* **Service Layer**  
  In deze laag bevindt zich de kernlogica van de applicatie. Hier worden gegevens gevalideerd, verwerkt en getransformeerd. Dit is de plaats waar de functionele logica van het systeem wordt afgehandeld, bijvoorbeeld het automatisch aanmaken van schedules voor het pollen van sensoren.
* **Data Access Layer (Repository Layer)**  
  Deze laag staat in voor de communicatie met de database. Via Spring Data JPA worden repositories opgezet die CRUD-operaties uitvoeren op onze TimescaleDB. Spring Boot automatiseert hierbij veel boilerplatecode, zodat de focus op de functionele logica kan blijven liggen.



1. Spring Boot Layered Architecture

Daarnaast bevat de backend een aantal **geplande taken (scheduled tasks)** die periodiek data ophalen via de ThingsBoard API. Deze gegevens worden lokaal opgeslagen in de database, zodat ze snel en performant beschikbaar zijn voor de frontend.

Tot slot zorgt **Spring Security**, in combinatie met **OAuth2**, voor een veilige toegang tot de API-endpoints. Gebruikers moeten correct geauthenticeerd zijn om bepaalde acties uit te voeren. Dit is essentieel voor de bescherming van gevoelige informatie. In een later hoofdstuk ga ik hier dieper op in.

## Frontend-overzicht (Angular)

De frontend van onze applicatie is ontwikkeld met **Angular**, een krachtig framework waarmee we een moderne en responsieve **single-page application (SPA)** kunnen bouwen. Angular maakt gebruik van componentgebaseerde architectuur, waarbij elk component verantwoordelijk is voor een specifiek onderdeel van de gebruikersinterface. Components kunnen genest zijn: een hoofdcomponent kan meerdere subcomponents bevatten. Dit zorgt voor een duidelijke structuur en bevordert herbruikbaarheid.

De communicatie met de backend verloopt via **services**. Wanneer een component data nodig heeft, roept het een bijhorende service aan. Deze service verstuurt een HTTP-verzoek naar een REST-endpoint en ontvangt de gevraagde gegevens. De binnenkomende data wordt vervolgens gemapt naar **TypeScript-interfaces**, wat zorgt voor typeveiligheid en verhoogde leesbaarheid van de code.

Navigatie binnen de applicatie gebeurt via **Angular Routing**, waarmee we ook toegangsbeheer kunnen implementeren. Zo kunnen we bepalen welke gebruikers toegang hebben tot specifieke routes en componenten, wat bijdraagt aan de veiligheid en het beheer van gebruikersrechten.

Voor de vormgeving en visuele consistentie maken we gebruik van **PrimeNG**, een uitgebreide bibliotheek van gebruiksklare UI-componenten. Door deze componenten te integreren en onze eigen stijlen hierop af te stemmen, realiseren we een uniforme, professionele **look & feel** binnen de applicatie.

Onderaan vindt u een visueel overzicht van de volledige technische architecturale opbouw zoals beschreven in dit hoofdstuk.

Afbeelding met tekst, schermopname, Lettertype, nummer

Door AI gegenereerde inhoud is mogelijk onjuist.

1. Totale technische architectuur

# Beveiliging & Authenticatie

Beveiliging vormt een essentieel onderdeel van onze applicatie, met name vanwege de aard van de gegevens (zoals aanwezigheidsinformatie, sensorwaarden en alarminstellingen). Indien deze informatie in verkeerde handen terechtkomt, zou dit bijvoorbeeld kunnen leiden tot het identificeren van een locatie als “onbewaakt”. Daarom hebben we gekozen voor een doordachte strategie met betrekking tot authenticatie en autorisatie.

## OAuth2 in de praktijk

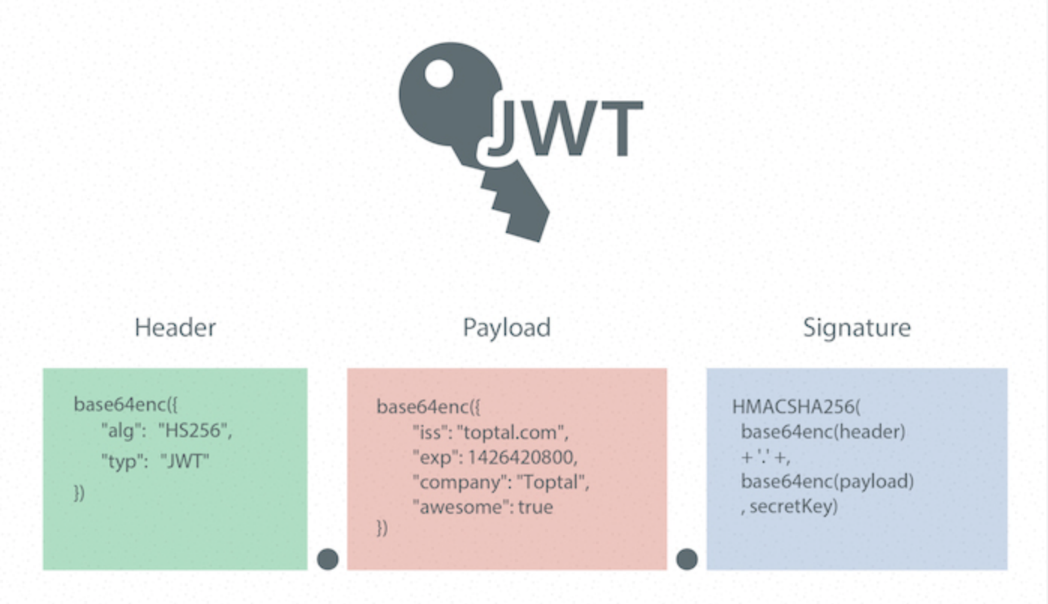
Onze keuze voor OAuth2 is gebaseerd op het evenwicht tussen gebruiksgemak en veiligheid. Functionaliteiten zoals inlogprocedures, beheer van inloggegevens en versleuteling worden volledig verzorgd door de OAuth2-provider. Hierdoor kunnen wij ons focussen op een degelijke integratie en tegelijkertijd profiteren van de robuuste beveiliging van het protocol.

OAuth2 maakt gebruik van een externe inlogflow: gebruikers worden doorgestuurd naar een betrouwbare loginpagina van de OAuth2-provider. Na een succesvolle authenticatie valideert de provider de inloggegevens en stuurt vervolgens een zogenaamd JWT-token (JSON Web Token) terug naar onze applicatie. Dit token bevat geverifieerde informatie over de gebruiker, zoals e-mailadres, gebruikersnaam, ID en rollen.

Deze JWT-token wordt lokaal opgeslagen in de session storage of local storage van de browser en fungeert als sleutel voor alle beveiligde interacties binnen onze applicatie:

* Het token wordt uitgelezen om de rol van de gebruiker te bepalen, waarna de gebruikersinterface hierop wordt aangepast. Zo heeft een admin toegang tot meer functionaliteiten dan een standaardgebruiker.
* Angular’s routeringslogica controleert bij elke navigatiepoging of de gebruiker beschikt over de juiste toegangsrechten voor de gevraagde route of component.
* Elk API-verzoek bevat de JWT-token in de headers. Aan de backendzijde wordt deze token geverifieerd om de geldigheid en herkomst van het verzoek te controleren.

Een JWT is een compacte, URL-veilige token die bestaat uit drie delen: een header, payload en signature. De payload bevat informatie over de gebruiker (zoals rollen), en de signature garandeert dat de inhoud niet ongemerkt gewijzigd kan worden. Zo verzekeren we de betrouwbaarheid en integriteit van de verstuurde gegevens.



1. JWT token opbouw

In onderstaande configuratie maken we gebruik van de angular-auth-oidc-client-bibliotheek, die het implementeren van OAuth2-authenticatie in Angular aanzienlijk vereenvoudigt.

We definiëren onder meer de authority, oftewel de URL van de OAuth2-provider die instaat voor de loginflow en de validatie van gebruikers. Daarnaast stellen we de redirectUrl in, die bepaalt waar de gebruiker na een succesvolle login naartoe wordt geleid. Voor mobiele applicaties gebruiken we hiervoor een aangepaste URI; in de webapplicatie wordt de gebruiker doorgestuurd naar een specifieke route (/floor-overview).

Verder specificeren we parameters zoals de clientId, de gewenste scope, en het responseType. In dit geval maken we gebruik van de authorization code flow, wat betekent dat we een autorisatiecode ontvangen die later wordt omgewisseld voor een toegangstoken. Ook configureren we de postLogoutRedirectUri, zodat gebruikers na het uitloggen automatisch worden doorverwezen naar de juiste pagina.

provideAuth({  
 config: {  
 authority: `${***environment***.authenticatorUrl}`,  
 redirectUrl: mobileApp ? 'be.faros.worksense://callback' : ***window***.location.origin + '/floor-overview',  
 postLoginRoute: '/floor-overview',  
 postLogoutRedirectUri: mobileApp ? 'be.faros.worksense://callback' : ***window***.location.origin + '/login',  
 clientId: `${***environment***.clientId}`,  
 scope: `${***environment***.authScope}`,  
 responseType: 'code',  
 logLevel: LogLevel.*Debug*,  
 },  
})

Binnen onze AuthService beheren we de volledige authenticatieflow aan de hand van de angular-auth-oidc-client-bibliotheek. Deze service biedt methodes om:

* De loginprocedure te starten
* Gebruikers uit te loggen
* Te controleren of een gebruiker is ingelogd
* De JWT-token op te halen
* De rollen van de ingelogde gebruiker te achterhalen

Dankzij deze centrale aanpak wordt het eenvoudig om authenticatie en autorisatie consistent en veilig toe te passen binnen onze Angular-applicatie.

Onderstaand fragment toont hoe de AuthService is opgebouwd:

export class AuthService {  
 private tokenKey = ***environment***.tokenKey;  
  
 constructor(private oidcSecurityService: OidcSecurityService) { }  
  
 isLoggedIn(): Observable<boolean> {  
 return this.oidcSecurityService.isAuthenticated$.pipe(  
 map(({ isAuthenticated }) => isAuthenticated)  
 );  
 }  
  
 login() {  
 if (***window***.sessionStorage && ***window***.localStorage) {  
 ***window***.sessionStorage.clear();  
 ***window***.localStorage.clear();  
 }  
  
 this.oidcSecurityService.authorize();  
 }  
  
 logout() {  
 if (***window***.sessionStorage && ***window***.localStorage) {  
 ***window***.sessionStorage.clear();  
 ***window***.localStorage.clear();  
 }  
  
 ***window***.location.href = `${***environment***.logoutUri}`;  
  
 this.oidcSecurityService.logoff().subscribe();  
 }  
  
 getToken(): string | null {  
 const tokenData = ***localStorage***.getItem(this.tokenKey);  
 if (tokenData) {  
 const parsedData = ***JSON***.parse(tokenData);  
 return parsedData.authnResult?.access\_token || null;  
 }  
 return null;  
 }  
  
 getRole(): string[] {  
 const token = this.getToken();  
  
 if (token) {  
 const payload = ***JSON***.parse(atob(token.split('.')[1]));  
  
 return payload?.['cognito:groups'] || payload?.role || [];  
 }  
  
 return [];  
 }  
}

Deze service zorgt voor een duidelijke scheiding van verantwoordelijkheden: authenticatie wordt centraal afgehandeld, terwijl elders in de applicatie eenvoudig rollen gecontroleerd of tokens opgehaald kunnen worden waar nodig.

Naast de AuthService zijn er verschillende andere onderdelen binnen de frontend die inspelen op authenticatie en autorisatie. Zo maken we gebruik van een **AuthGuard**, die telkens geactiveerd wordt wanneer een gebruiker naar een andere route navigeert. Deze guard controleert op basis van de aanwezige JWT-token of de gebruiker voldoende rechten heeft om de gewenste pagina te openen.

Verder zorgen we ervoor dat de JWT-token automatisch wordt toegevoegd aan elke HTTP-request richting de backend. Dit wordt gerealiseerd via een **HTTP-interceptor**, die het token ophaalt en toevoegt aan de Authorization-header van elke uitgaande request. Hierdoor blijft onze API beschermd en kunnen enkel geauthenticeerde gebruikers toegang verkrijgen tot beveiligde endpoints.

## Beveiliging van API’s en data

Ook aan de backendzijde worden de API’s zorgvuldig beveiligd. De eerste verdedigingslijn bestaat uit het beperken van inkomende requests tot enkel het IP-adres van onze frontend. Hierdoor wordt de kans verkleind dat externe partijen rechtstreeks verbinding proberen te maken met onze endpoints.

Maar IP-filtering op zich is niet voldoende. Een kwaadwillende partij zou zich nog steeds kunnen voordoen als onze frontend en proberen data te onderscheppen of manipuleren. Daarom maken we gebruik van **Spring Security** om een extra laag van validatie toe te voegen.

Elke request moet vergezeld zijn van een geldige JWT-token, die in de backend wordt gecontroleerd. Deze token wordt gevalideerd via de OAuth2-authorizer, zodat we zeker zijn dat het verzoek afkomstig is van een correct geauthenticeerde gebruiker. Bovendien analyseren we de in de token opgenomen rollen. Bepaalde endpoints zijn immers enkel toegankelijk voor gebruikers met specifieke rechten, zoals ADMIN.

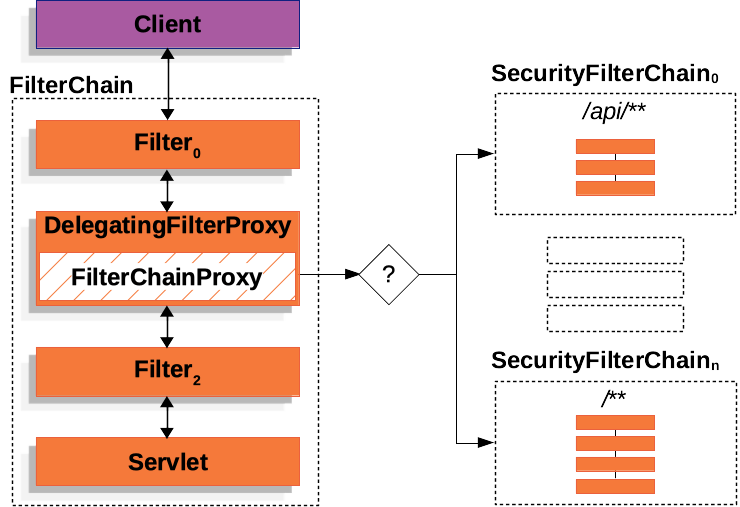
Zo garanderen we dat alleen geautoriseerde gebruikers toegang hebben tot gevoelige gegevens of acties, en blijft onze API afgeschermd tegen ongewenste toegang.

Binnen ons project gebruiken we Spring Security om de backend-API’s te beschermen. Door Spring Security toe te voegen, kunnen we een centrale configuratie opstellen waarin de toegangsregels worden gedefinieerd. Dit gebeurt via een **SecurityFilterChain**, waarin we bepalen wie toegang heeft tot welke endpoints, en onder welke voorwaarden.

In onderstaande methode tonen we hoe deze SecurityFilterChain is opgebouwd. Hierin specificeren we dat bepaalde routes uitsluitend toegankelijk zijn voor gebruikers met de rol ADMIN of SUPER\_ADMIN. Daarnaast schakelen we **CSRF-bescherming** uit (omdat onze API stateless is), en activeren we **CORS-configuratie** via een aangepaste configuratieklasse. De applicatie werkt volledig stateless en maakt gebruik van JWT-validatie via een OAuth2-resource server.

@Bean  
public SecurityFilterChain securityFilterChain(HttpSecurity http, CustomCorsConfig customCorsConfig) throws Exception {  
 http  
 .cors(cors -> cors.configurationSource(customCorsConfig.corsConfigurationSource()))  
 .csrf(AbstractHttpConfigurer::disable)  
 .authorizeHttpRequests(auth -> auth  
 .requestMatchers("/sensors/sync").hasAnyRole("ADMIN", "SUPER\_ADMIN")  
 .requestMatchers("/locations/admin", "/locations/admin/\*\*").hasAnyRole("ADMIN", "SUPER\_ADMIN")  
 .requestMatchers("/groups/admin","/groups/admin/\*\*").hasAnyRole("ADMIN", "SUPER\_ADMIN")  
 .requestMatchers("/rooms/admin","/rooms/admin/\*\*").hasAnyRole("ADMIN", "SUPER\_ADMIN")  
 .requestMatchers("/users/admin","/users/admin/\*\*").hasAnyRole("ADMIN", "SUPER\_ADMIN")  
 .requestMatchers(HttpMethod.*OPTIONS*, "/\*\*").permitAll()  
 .requestMatchers("/\*\*").authenticated()  
 .anyRequest().authenticated()  
 )  
 .oauth2ResourceServer(oauth2 -> oauth2  
 .jwt(jwt -> jwt.jwtAuthenticationConverter(jwtAuthenticationConverter()))  
 )  
 .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.*STATELESS*));  
  
 return http.build();  
}

Zoals aangetoond, is Spring Security relatief eenvoudig te configureren en biedt het een krachtige en flexibele oplossing om de beveiliging van een applicatie te garanderen. Door gebruik te maken van OAuth2 in combinatie met JWT-tokens, kunnen gebruikers op een veilige en schaalbare manier worden geauthenticeerd en geautoriseerd, zonder dat we zelf gevoelige gegevens, zoals wachtwoorden, hoeven te beheren.



1. Spring Security FilterChain

Daarnaast biedt Spring Security uitgebreide mogelijkheden om de toegang tot API-endpoints fijnmazig te regelen op basis van gebruikersrollen. Hierdoor wordt gegarandeerd dat alleen geautoriseerde gebruikers toegang hebben tot gevoelige gegevens en functionaliteiten.

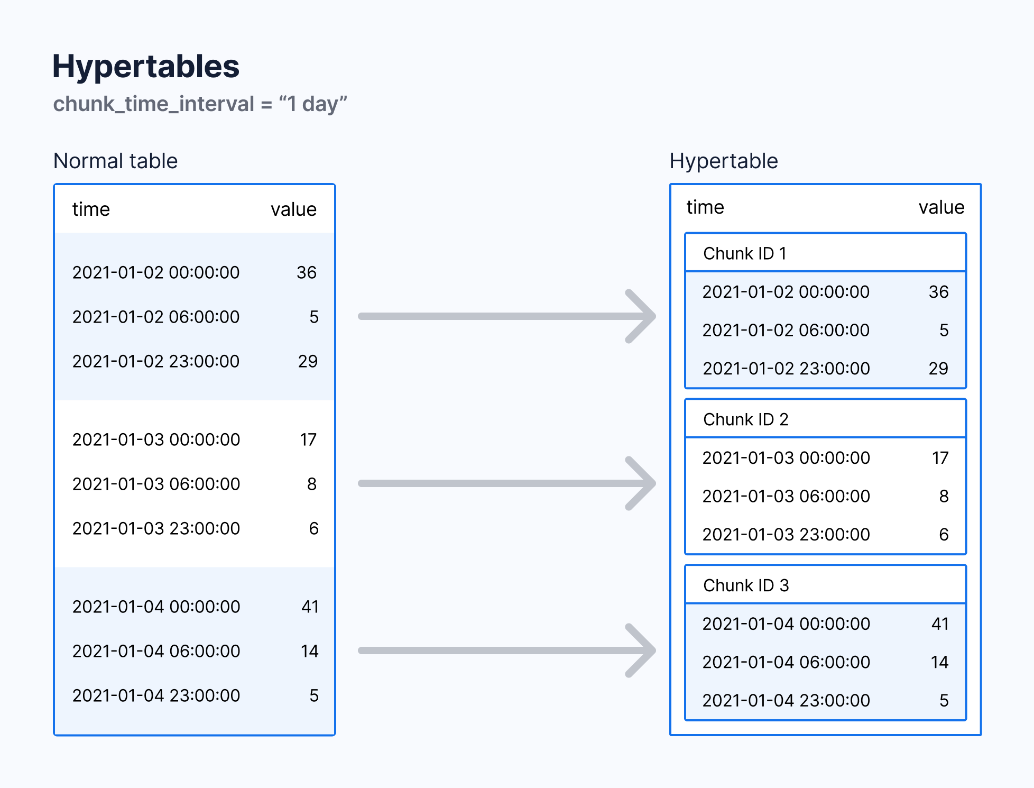
In combinatie met onze frontend, die automatisch tokens toevoegt aan HTTP-verzoeken en beveiligingscontroles uitvoert via guards en interceptors, resulteert dit in een robuust en modern beveiligingssysteem dat klaar is voor productiegebruik. Deze aanpak beschermt niet alleen de data, maar verhoogt ook het vertrouwen van gebruikers in het platform.

# Database & Data Management

Voor de opslag en verwerking van onze data kozen we, zoals eerder besproken, voor **TimescaleDB** – een krachtige extensie van PostgreSQL, specifiek geoptimaliseerd voor het verwerken van tijdreeksen (time-series data), zoals sensorwaarden.

Een van de belangrijkste voordelen van TimescaleDB is het gebruik van **hypertables**: een abstractielaag bovenop reguliere PostgreSQL-tabellen die geoptimaliseerd zijn voor tijdgebaseerde gegevens. In tegenstelling tot traditionele tabellen met een ID als primaire sleutel, maken hypertables gebruik van een **timestamp-kolom** voor indexering en partitionering. Hierdoor wordt de opslag logischer gestructureerd en efficiënter afgestemd op continue datastromen zoals sensorgegevens.

Een ander belangrijk kenmerk van TimescaleDB is de **automatische opsplitsing van hypertables in zogenoemde chunks**, standaard per zeven dagen. Deze chunking voorkomt dat tabellen onbegrensd blijven groeien, wat de prestaties van query’s ten goede komt. Door slechts de relevante chunks te doorzoeken bij het uitvoeren van een query, kan TimescaleDB zelfs bij grote datasets een hoge snelheid behouden.



1. TimescaleDB chunks

Daarnaast biedt TimescaleDB krachtige functionaliteiten voor gegevensaggregatie, zoals het berekenen van gemiddelden over vaste intervallen (bijvoorbeeld per uur of per dag). Dit maakt het mogelijk om data te groeperen, analyseren en visualiseren op een manier die zowel nuttig is voor eindgebruikers als geschikt voor het opsporen van trends en afwijkingen.

Dankzij deze eigenschappen is TimescaleDB uitermate geschikt gebleken voor onze toepassing, waarin we grote hoeveelheden sensorinformatie efficiënt willen opslaan, opvragen en analyseren.

## Databasemodel

Een belangrijk voordeel van TimescaleDB is dat het volledig gebaseerd is op PostgreSQL. Hierdoor behouden we alle mogelijkheden van een krachtig relationeel databasemodel, waaronder ondersteuning voor relaties, constraints, joins en andere klassieke SQL-functionaliteiten. Dit was cruciaal voor ons project, dat verschillende logische verbanden tussen entiteiten vereiste.

Zo waren er enkele functionele vereisten die een relationeel model noodzakelijk maakten. Een persoon moest bijvoorbeeld gekoppeld zijn aan een groep, waarbij een groep op zijn beurt meerdere locaties omvat. Elke locatie bevat verschillende kamers, en in die kamers bevinden zich uiteindelijk de sensoren. Deze hiërarchische structuur laat zich uitstekend modelleren via relationele tabellen en foreign key-relaties.

Een ander belangrijk onderdeel was de implementatie van meldingen (alerts). Een alert is gekoppeld aan zowel een persoon als een kamer. Dit betekent dat een gebruiker een persoonlijke melding kan instellen voor een specifieke ruimte — bijvoorbeeld om notificaties te ontvangen wanneer bepaalde drempelwaarden worden overschreden. Dit vereiste een many-to-one-relatie tussen alerts, kamers en personen, die zonder problemen kon worden ondersteund door TimescaleDB/PostgreSQL.

Dankzij deze relationele structuur konden we niet alleen sensorwaarden efficiënt opslaan via hypertables, maar ook de bredere context en configuratie van het systeem op een overzichtelijke en schaalbare manier modelleren.

Onderstaand diagram toont het databasemodel dat we ontworpen hebben voor deze applicatie.

Afbeelding met schermopname, ontwerp

Door AI gegenereerde inhoud is mogelijk onjuist.

1. Klassendiagram

Bij het opstellen van het databasemodel hebben we zoveel mogelijk best practices toegepast om de structuur logisch, schaalbaar en performant te houden. Een van de technieken die we hierbij hebben gebruikt, is het inzetten van enumeraties (enums) voor kolommen met een beperkt en voorspelbaar aantal mogelijke waarden.

Dergelijke velden komen vaak voor bij bijvoorbeeld statussen, types of rollen. Door deze waarden als enums in de database te definiëren, vermijden we ongewenste variatie, zoals inconsistent gebruik van hoofdletters of typfouten (bijvoorbeeld "Admin" versus "admin" of "Administrator"). Dit bevordert dataconsistentie, vereenvoudigt validatie en verhoogt de betrouwbaarheid van rapportages.

Daarnaast dragen enums bij aan de datakwaliteit en leesbaarheid van het model, doordat de toegestane waarden expliciet zijn vastgelegd. Dit is met name waardevol bij samenwerking in grotere teams of wanneer externe systemen interageren met onze database. Door het gebruik van enums minimaliseren we het risico op fouten en creëren we een stabiele en goed gestructureerde basis voor verdere ontwikkeling van de applicatie.

## Spring Data JPA

Voor het beheer van data in onze applicatie maakten we gebruik van **Spring Data JPA**, een krachtige bibliotheek binnen het Spring-ecosysteem die het werken met relationele databases aanzienlijk vereenvoudigt. In plaats van manueel SQL-queries te schrijven, konden we met behulp van JPA repository-interfaces veelvoorkomende database-operaties automatisch beschikbaar maken, zoals findAll(), save() en deleteById().

Elk domeinobject in onze applicatie werd gemapt naar een JPA-entity, die rechtstreeks correspondeert met een tabel in onze TimescaleDB-database. Hiervoor gebruikten we annotaties zoals @Entity, @Table, @Id, @Column, enzovoort. Ook relaties tussen tabellen (zoals @OneToMany of @ManyToOne) werden via annotaties vastgelegd. Hieronder volgt een voorbeeld van een entity:

@Entity  
@Table(name = "SENSOR\_READING")  
@Getter  
@Setter  
@NoArgsConstructor  
@AllArgsConstructor  
public class SensorReading {  
 @EmbeddedId  
 @NotNull(message = "SensorReadingId cannot be null!")  
 private SensorReadingId id;  
  
 @ManyToOne  
 @MapsId("sensorId")  
 @JsonBackReference  
 @NotNull(message = "Sensor cannot be null!")  
 private Sensor sensor;  
  
 @Column(name = "VALUE")  
 @NotNull(message = "Value cannot be null!")  
 private Double value;  
}

Voor elke entiteit definieerden we een bijbehorende JpaRepository. Hierdoor konden we snel gebruikmaken van standaardmethoden voor het opvragen en bewerken van data. Daarnaast was het mogelijk om op basis van naamconventies aangepaste querymethoden te definiëren, zoals:

@Repository  
public interface SensorRepository extends JpaRepository<Sensor, UUID> {  
 Optional<List<Sensor>> findSensorsBySensorType(SensorType sensorType);  
 Optional<List<Sensor>> findBySensorTypeAndRoomId(SensorType sensorType, UUID roomId);  
  
 @Query(value = """  
 SELECT DISTINCT s.sensor\_type  
 FROM room r  
 JOIN sensor s ON r.id = s.room\_id  
 WHERE r.location\_id = ?1  
 ORDER BY 1  
 """, nativeQuery = true)  
 Optional<List<Integer>> findSensorTypesByLocationId(UUID locationId);  
}

Indien complexere zoekopdrachten vereist waren, konden we gebruikmaken van de @Query-annotatie om aangepaste SQL- of JPQL-query’s te schrijven, bijvoorbeeld voor specifieke filtering of joins.

Omdat TimescaleDB gebaseerd is op PostgreSQL, was het volledig compatibel met JPA. Hoewel JPA standaard niet alle Timescale-specifieke functionaliteiten ondersteunt (zoals time\_bucket() of hypertable-partitionering), konden we de basisinteracties — CRUD-operaties, joins en filters — probleemloos uitvoeren via Spring Data. Voor specifieke analysequery’s maakten we gebruik van native queries om de volledige kracht van TimescaleDB te benutten.

Deze technologie stelde ons in staat om een schaalbare, onderhoudbare en veilige data access layer op te zetten, zonder onnodige boilerplate-code. Spring Data JPA was een essentieel onderdeel van de backendarchitectuur en droeg bij aan een snellere en consistenter ontwikkelcyclus.

# Sensorintegratie & ThingsBoard

In dit hoofdstuk lichten we toe wat ThingsBoard is, waarom het belangrijk was binnen ons project, hoe we het precies hebben ingezet, en welke voordelen en beperkingen we ervaren hebben. We gebruikten ThingsBoard als centraal platform voor het ontvangen, beheren en visualiseren van sensordata. De keuze voor ThingsBoard kwam deels voort uit de projectvereisten — Faros wilde graag ervaring opdoen met dit platform — maar bleek ook technisch gezien een waardevolle keuze.

De kracht van ThingsBoard ligt vooral in de brede waaier aan integratiemogelijkheden. Organisaties die reeds bestaande infrastructuur gebruiken (zoals sensoren die via The Things Stack of andere MQTT-brokers communiceren), kunnen dankzij kant-en-klare connectoren eenvoudig integreren met ThingsBoard. Dit maakt het platform bijzonder flexibel en inzetbaar in uiteenlopende contexten.

## Wat is ThingsBoard

ThingsBoard is een open-source IoT-platform dat ontworpen is om op schaal sensorgegevens te verzamelen, apparaten te beheren en inzichten te visualiseren via interactieve dashboards. Het ondersteunt gangbare protocollen zoals MQTT, HTTP en CoAP, waardoor het eenvoudig integreert met diverse hardware-oplossingen.

Belangrijke kenmerken van ThingsBoard zijn:

* **Device management**: het registreren, groeperen en configureren van apparaten.
* **Realtime data**: inkomende data worden onmiddellijk verwerkt en getoond op dashboards.
* **Rule engine**: via regels kunnen acties worden gedefinieerd (bijvoorbeeld een melding sturen bij overschrijding van een drempelwaarde).
* **Dashboards**: gebruiksvriendelijke dashboards kunnen visueel samengesteld worden voor directe inzichten.
* **API’s**: ThingsBoard biedt een uitgebreide REST API waarmee externe systemen gegevens kunnen opvragen of wijzigen.

In ons project fungeerde ThingsBoard als een tussenlaag tussen de fysieke sensoren en onze backendapplicatie. De sensoren — geprogrammeerd in C++ — verstuurden gegevens via het MQTT-protocol naar de ThingsBoard MQTT-broker. Daar werd de data opgevangen en opgeslagen. Onze backend maakte op zijn beurt gebruik van de REST API van ThingsBoard om op vaste tijdsintervallen relevante informatie (zoals sensorwaarden, metadata en configuraties) op te halen. Deze aanpak zorgde voor een duidelijke scheiding van verantwoordelijkheden en ontlastte onze backend.

In de volgende paragraaf gaan we dieper in op de technische koppeling tussen sensoren en ThingsBoard, evenals de gegevensstroom binnen het systeem.

## Koppeling met sensoren

Voordat we dieper ingaan op de technische koppeling tussen onze sensoren en ThingsBoard, is het zinvol om kort stil te staan bij het type hardware dat we gebruikten en de motivatie achter deze keuze.

Voor dit project kozen we voor de **SenseCAP D1 Pro**, een modulaire sensorhub ontwikkeld door Seeed Studio. Dit toestel is ontworpen met het oog op toegankelijkheid voor zowel beginnende als gevorderde ontwikkelaars en blinkt uit in uitbreidbaarheid en gebruiksgemak.

De SenseCAP D1 Pro is standaard uitgerust met ingebouwde sensoren voor onder meer temperatuur, luchtvochtigheid en lichtintensiteit, en kan eenvoudig worden uitgebreid met een breed scala aan Grove-sensoren. Grove is een plug-and-play ecosysteem van Seeed Studio, waarmee componenten zonder solderen of complexe bedrading aangesloten kunnen worden. Deze flexibiliteit was een belangrijk voordeel, aangezien we tijdens het project met verschillende sensortypes experimenteerden en onze opstellingen regelmatig moesten aanpassen.

Daarnaast beschikt de hub over een geïntegreerd display, waarop onder meer de wifi-verbinding geconfigureerd kan worden en waarop in real-time sensorwaarden en tijdsinformatie worden weergegeven. Dit maakte het tijdens de ontwikkelings- en testfase bijzonder eenvoudig om te controleren of de hardware correct functioneerde.



1. SenseCAP D1 Pro

Aangezien onze ervaring met embedded software development beperkt was, maakten we dankbaar gebruik van de open-source voorbeelden die Seeed Studio beschikbaar stelde. Een concreet voorbeeld hiervan is de **SenseCAP Home Assistant-integratie**, beschikbaar via GitHub:

<https://github.com/Seeed-Solution/SenseCAP_Indicator_ESP32>

Deze voorbeeldcode bood een solide basis om een verbinding op te zetten met een MQTT-broker. Door eenvoudig onze ThingsBoard-configuratiegegevens (zoals broker-URL, gebruikersnaam en toegangstoken) in te vullen in het configuratiebestand, konden we snel starten met het verzenden van gegevens.

We breidden deze code vervolgens uit om ook extra aangesloten Grove-sensoren automatisch te detecteren en uit te lezen. Hun waarden werden via het MQTT-protocol naar ThingsBoard verstuurd. Op die manier realiseerden we een robuuste en schaalbare oplossing, waarbij nieuwe sensoren toegevoegd konden worden zonder manuele herconfiguratie van het toestel.

Afbeelding met tekst, schermopname, nummer, Lettertype

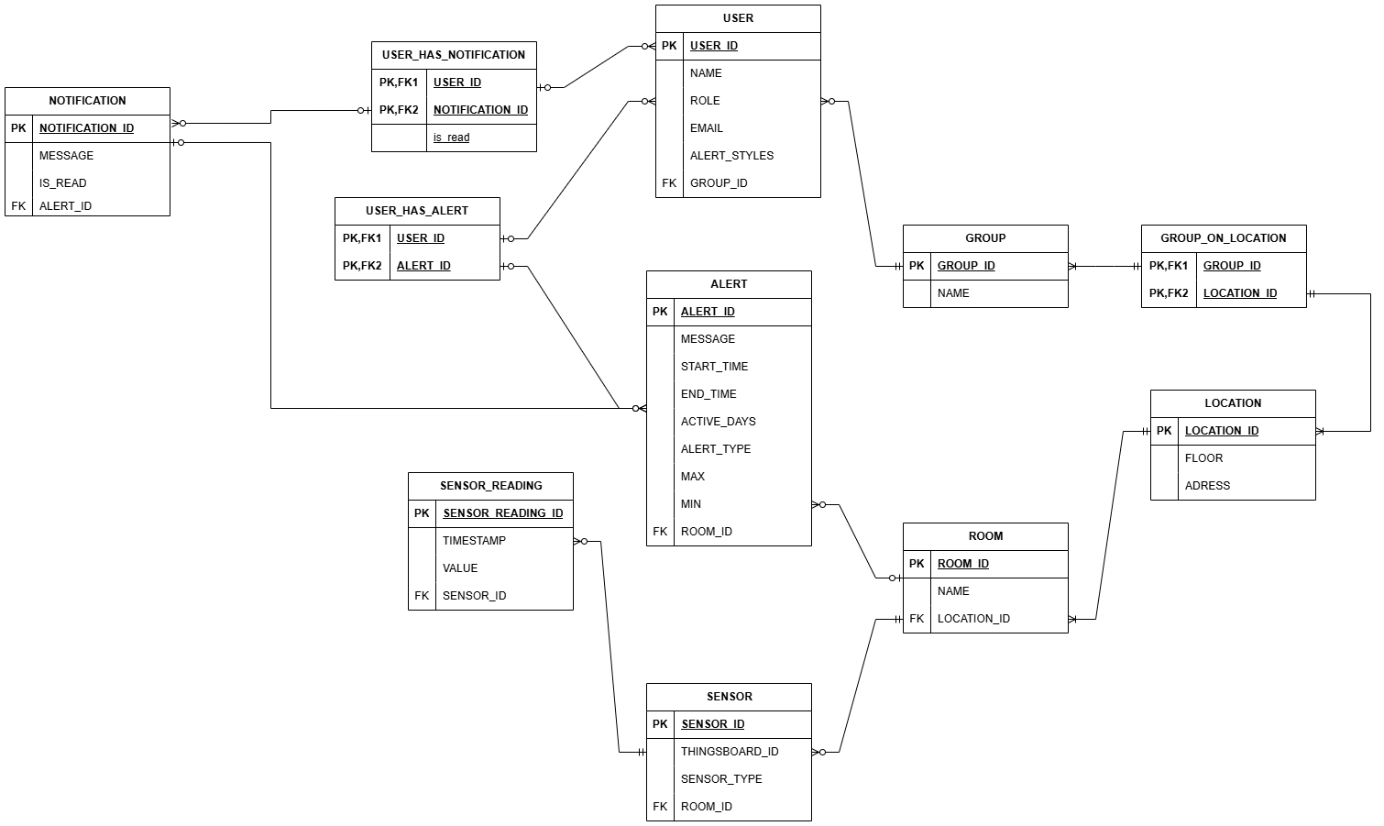
Door AI gegenereerde inhoud is mogelijk onjuist.

1. Sensorwaarden op ThingsBoard

Zodra de data op ThingsBoard binnenkwam, moest deze op een gestructureerde manier verwerkt worden. ThingsBoard maakt het mogelijk om telemetriedata in JSON-formaat te definiëren, waardoor de structuur van de inkomende gegevens duidelijk is voor externe applicaties.

Om dit proces gebruiksvriendelijk te maken voor eindgebruikers, ontwikkelden we in ons adminpanel een zogenaamde “JSON-wizard”. Deze wizard begeleidde gebruikers stap voor stap bij het aanmaken van de juiste JSON-structuur voor hun sensoren. Hierbij konden veldnamen, gegevenstypes en de bijbehorende sleutels voor binnenkomende sensordata gedefinieerd worden.

Een essentieel onderdeel in dit JSON-bestand was de toevoeging van een roomId. Hiermee kon onze backend bij het ophalen van data via de ThingsBoard API direct bepalen aan welke kamer de gegevens gekoppeld moesten worden. Dit was cruciaal voor een correcte visualisatie en verwerking binnen onze applicatie.



Afbeelding met tekst, schermopname, Lettertype, nummer

Door AI gegenereerde inhoud is mogelijk onjuist.

1. JSON Telemetry

De combinatie van toegankelijke hardware, krachtige MQTT-integratie en flexibele JSON-configuratie stelde ons in staat om snel en betrouwbaar sensordata te integreren in ons systeem. ThingsBoard fungeerde hierbij als een robuuste tussenlaag tussen de fysieke wereld (de sensoren) en onze digitale verwerkingsstructuur (backend en database). Dankzij de uitbreidbaarheid van zowel de hardware als de software is het systeem bovendien schaalbaar en eenvoudig aanpasbaar voor toekomstige toepassingen.

# Visualisatie & Gebruikerservaring

Een van de belangrijkste aspecten van onze applicatie is ongetwijfeld de visualisatie van data en de algemene gebruikerservaring. De applicatie moet niet alleen functioneel sterk zijn, maar ook intuïtief en visueel aantrekkelijk, zodat gebruikers snel inzicht krijgen in de verzamelde sensorinformatie. Aangezien we omgevingsfactoren monitoren om daaruit conclusies te trekken, is het belangrijk dat de visualisaties helder en doelgericht zijn opgebouwd.

We hebben verschillende technieken toegepast om de gebruikerservaring te optimaliseren:

* **Live-updates van sensorwaarden**: Dankzij Server-Sent Events (SSE) worden sensorwaarden in real-time geüpdatet in de interface. Dit maakt het eenvoudig om te controleren of een sensor actief is en maakt het mogelijk om een permanent dashboard op een groot scherm te tonen zonder handmatige verversing.
* **Gebruikersfeedback bij fouten**: Bij eventuele fouten tonen we steeds een duidelijke melding met uitleg. Dit voorkomt verwarring en verhoogt het gebruiksgemak.
* **Meertaligheid**: De applicatie is volledig vertaald naar het Engels, wat de toegankelijkheid voor een internationaal publiek aanzienlijk vergroot.

## Tijdlijnvisualisaties

Om trends in sensorwaarden – zoals temperatuur, luchtvochtigheid of aanwezigheid – inzichtelijk te maken, ontwikkelden we interactieve tijdlijncomponenten. Deze componenten tonen de evolutie van specifieke sensortypes binnen een kamer, met ondersteuning voor verschillende tijdsintervallen: 1 uur, 4 uur, 12 uur en 24 uur.

Afbeelding met tekst, schermopname, Perceel, Lettertype

Door AI gegenereerde inhoud is mogelijk onjuist.

1. Grafiek met sensorwaarden

Dankzij TimescaleDB konden we deze functionaliteit efficiënt implementeren. Deze database maakt het mogelijk om op eenvoudige wijze gemiddelde waarden op te vragen per uur of per ander tijdsvenster – een cruciale eigenschap voor het genereren van overzichtelijke en informatieve grafieken.

**Voorbeelden van inzichten uit deze visualisaties zijn:**

* De evolutie van de temperatuur doorheen de dag (bijvoorbeeld een snelle temperatuurdaling 's avonds).
* De aanwezigheid van mensen op kantoor, afgeleid uit CO₂- of bewegingssensoren.
* Of de verlichting nog actief is buiten de kantooruren.

Voor de visuele presentatie van deze data gebruikten we onder andere PrimeNG, waarmee we grafieken op een aantrekkelijke en responsieve manier konden weergeven.

## Alarmfunctionaliteit

Het kunnen instellen van waarschuwingen (alerts) was een belangrijke functionele vereiste binnen het project. We ontwikkelden een flexibele alarmmodule waarmee gebruikers meldingen kunnen configureren op basis van:

* Specifieke kamers
* Sensorwaarden (zoals CO₂, temperatuur of beweging)
* Instelbare minimum- en maximumdrempels
* Tijdschema’s (bijvoorbeeld enkel tijdens kantooruren)

Afbeelding met tekst, schermopname, Lettertype, ontwerp

Door AI gegenereerde inhoud is mogelijk onjuist.

1. Alarm functionaliteit

Gebruikers kunnen zelf kiezen hoe ze deze meldingen ontvangen: via pushberichten, e-mails of interne notificaties binnen de applicatie. Dankzij deze functie kunnen gebruikers snel ingrijpen wanneer bepaalde drempelwaarden worden overschreden, zoals bij slechte luchtkwaliteit.

Hiermee gaat de toepassing verder dan enkel het raadplegen van data: het stelt gebruikers in staat om proactief actie te ondernemen.

## Mobiele toegankelijkheid

Gebruikers willen ook onderweg snel kunnen nagaan hoeveel mensen er aanwezig zijn op kantoor, en of de temperatuur in een bepaalde ruimte nog binnen de grenzen valt.

Afbeelding met tekst, overdekt, computer, whiteboard

Door AI gegenereerde inhoud is mogelijk onjuist.

1. Applicatie op tablet

Vanaf de start van het project kozen we daarom bewust voor een **mobile-first aanpak**. Enkele concrete toepassingen daarvan zijn:

* **Capacitor**: We gebruikten Capacitor om onze Angular-app eenvoudig te kunnen omzetten naar een APK voor Android. Hierdoor kan de applicatie rechtstreeks geïnstalleerd worden op smartphones.
* **Progressive Web App (PWA)**: De app is beschikbaar als PWA, wat betekent dat gebruikers deze aan hun startscherm kunnen toevoegen, zonder tussenkomst van een app store.
* **Responsief design**: Dankzij het gebruik van flexibele componenten en lay-outprincipes uit PrimeNG past de interface zich automatisch aan verschillende schermgroottes aan, wat zorgt voor een optimale gebruikerservaring op zowel desktop als mobiel.

# Testing & Kwaliteitsborging

Bij Faros hechten ze veel belang aan het grondig testen van applicaties. Een stabiele en betrouwbare werking is essentieel, vooral in professionele omgevingen waar uitval geen optie is. Java is bijzonder geschikt voor robuuste toepassingen. Hoewel het soms als traag wordt ervaren, is dit vaak het gevolg van de grondige validaties die de JVM bij het opstarten uitvoert. Deze vroege foutdetectie draagt bij aan de stabiliteit en veiligheid van de applicatie.

Binnen dit project kozen we dan ook bewust voor een uitgebreide teststrategie, waarmee we optimaal gebruikmaakten van de kracht van het Java-ecosysteem en tegelijkertijd de kwaliteit van onze code duurzaam konden waarborgen.

## Unittests

Unittests zijn gericht op het testen van kleine, afzonderlijke onderdelen van de applicatie – meestal methoden binnen een service of klasse – in volledige isolatie. Er wordt hierbij geen gebruikgemaakt van een echte database of externe services. In plaats daarvan worden afhankelijkheden vaak gemockt, zodat enkel de logica van het te testen onderdeel wordt geëvalueerd.

Onze unittests zijn opgebouwd volgens het *Given-When-Then*-principe, een best practice binnen test-driven development:

* **Given:** Initialisatie van objecten en testdata
* **When:** Uitvoeren van de te testen actie
* **Then:** Controleren of het verwachte resultaat is behaald

Een voorbeeld van zo’n unit test:

@Test  
public void testAssociateUserWithGroup() {  
 // GIVEN  
 Group group = new Group(UUID.*randomUUID*(), "IT Department", new ArrayList<>(), new ArrayList<>());  
 User user = new User(UUID.*randomUUID*(), "John Doe", Role.*ADMIN*, "john.doe@example.com", group, new ArrayList<>());  
  
 // THEN  
 *assertNotNull*(user.getGroup(), "User should be associated with a Group");  
 *assertEquals*("IT Department", user.getGroup().getName(), "Group name should be 'IT Department'");  
}

## Integratietests

Waar unittests zich beperken tot individuele methoden, testen integratietests de samenwerking tussen verschillende onderdelen van het systeem. Denk aan scenario’s waarbij interactie plaatsvindt met de database, services en controllers tegelijk.

Voor onze integratietests maakten we gebruik van **Testcontainers** – een moderne technologie gebaseerd op Docker, waarmee we tijdelijke, geïsoleerde databases konden opzetten tijdens het uitvoeren van tests. Hierdoor is elke testomgeving voorspelbaar en volledig onafhankelijk van de ontwikkelomgeving.

**Voordelen hiervan:**

* Volledige controle over testdata
* Vermijden van conflicten met bestaande databases
* Herhaalbare, consistente resultaten

Een goed voorbeeld: als een test verwacht dat er zeven gebruikers zijn, moeten er ook exact zeven worden aangemaakt binnen de testcontainer. Door dit niet op de ontwikkelomgeving te doen, vermijden we inconsistente resultaten of *false positives*.

## Handmatige testen & tools

Naast geautomatiseerde tests voerden we ook regelmatig handmatige controles uit. Hiervoor maakten we gebruik van verschillende tools om de kwaliteit van onze code te waarborgen:

* **Linting & PMD:**
  + Linting werd toegepast op de frontend om syntaxfouten en inconsistenties op te sporen.
  + PMD werd ingezet voor de backend om code smells, duplicatie en kwetsbaarheden te detecteren.
  + Beide tools werden geconfigureerd met een combinatie van strikte en soepelere regels, afhankelijk van de testfase.
* **Postman:**  
  Postman was ons voornaamste instrument voor het testen van backend-endpoints. Hiermee konden we HTTP-verzoeken versturen, headers toevoegen, tokens injecteren en responses inspecteren. Postman ondersteunt bovendien het schrijven van testscripts en het automatiseren van scenario’s.
* **IntelliJ IDEA:**  
  IntelliJ bood een krachtige omgeving voor het testen en debuggen van code. Via de geïntegreerde databaseverbinding konden we eenvoudig controleren of data correct werd opgeslagen. De debugger stelde ons in staat stap voor stap door de code te lopen en op elk moment de waarde van variabelen te analyseren.

## CI/CD

Om te garanderen dat de codebase continu getest en gevalideerd wordt, implementeerden we een CI/CD-pijplijn. Deze pipeline werd automatisch geactiveerd bij elke push naar een branch en voerde onder meer de volgende acties uit:

* Uitvoeren van unit- en integratietests
* Toepassen van linting en code-analyse
* Validatie van de buildkwaliteit

Dankzij deze aanpak konden we met vertrouwen nieuwe features ontwikkelen en fouten snel detecteren, nog vóór ze in de hoofdbranch terechtkwamen.

# Cloudhosting & Infrastructuur (AWS)

Om de applicatie betrouwbaar, schaalbaar en continu beschikbaar te maken, kozen we voor Amazon Web Services (AWS). AWS is een van de meest gebruikte cloudplatformen ter wereld en biedt een uitgebreid scala aan diensten voor infrastructuurbeheer, schaalbare hosting en monitoring. Bij Faros is bovendien reeds veel expertise opgebouwd rond AWS, wat het tot de logische keuze maakte voor dit project.

Dankzij deze cloudoplossing kunnen we onze backend-applicatie ononderbroken laten draaien, realtime sensorwaarden blijven ophalen en gebruikers op elk moment toegang geven tot het dashboard — ongeacht hun locatie of toestel.

In dit hoofdstuk bespreken we de belangrijkste AWS-diensten die we hebben ingezet, en hoe we de infrastructuur en deployments georganiseerd hebben.

## Overzicht gebruikte AWS-diensten

Voor dit project maakten we gebruik van diverse AWS-diensten, elk met een specifieke rol in de infrastructuur. Deze combinatie resulteerde in een stabiele, schaalbare en goed beheerbare omgeving:

* **EC2 (Elastic Compute Cloud)**  
  Onze backend werd gehost op een EC2-instance: een virtuele server waarop we volledige controle hebben over het besturingssysteem, de configuratie en de runtime. De Spring Boot-applicatie is zo geconfigureerd dat ze automatisch opstart bij een reboot van de machine. Ook onze database draaide op een aparte EC2-instance, wat ons maximale flexibiliteit en controle gaf over de configuratie van TimescaleDB.
* **S3 (Simple Storage Service)**  
  S3 werd ingezet voor meerdere doeleinden. De backend werd gecompileerd tot een .jar-bestand en in een S3-bucket geplaatst, zodat de EC2-instance steeds de meest recente versie kan ophalen tijdens een deployment. Daarnaast hostten we de frontend (Angular-build) in een aparte S3-bucket.
* **CloudFront**  
  De S3-bucket voor de frontend werd gekoppeld aan CloudFront, het Content Delivery Network (CDN) van AWS. CloudFront zorgt ervoor dat de frontend wereldwijd wordt gerepliceerd en geserveerd vanaf de dichtstbijzijnde edge location, wat resulteert in snellere laadtijden. Bovendien biedt CloudFront een eigen domeinnaam, waarmee de frontend toegankelijk wordt gemaakt.
* **Cognito**  
  Voor authenticatie gebruikten we AWS Cognito als OAuth2-provider. Cognito is volledig configureerbaar via de AWS-console en biedt alle functionaliteiten van een traditionele identity provider: gebruikersbeheer, authenticatieflows, tokenverwerking en meer. Rollen die in Cognito worden gedefinieerd, kunnen worden gekoppeld aan IAM-policies, wat een fijnmazige toegangscontrole binnen de infrastructuur mogelijk maakt.

## Architectuuroverzicht en dataflow

Onderstaand overzicht beschrijft hoe de infrastructuur functioneert en hoe de data binnen onze applicatie stroomt.

A screenshot of a computer

AI-generated content may be incorrect.

1. Architectuur AWS

Wanneer een gebruiker de website bezoekt, wordt er een verbinding gemaakt via onze CloudFront-distributie. CloudFront fungeert als wereldwijd verspreid CDN en haalt de frontendbestanden (de Angular-applicatie) op uit de gekoppelde S3-bucket. Deze bestanden worden vervolgens lokaal geserveerd aan de gebruiker, wat zorgt voor snelle laadtijden.

Na het laden van de frontend en het aangaan van interactie door de gebruiker, worden HTTP-verzoeken verstuurd naar onze backend, die draait op een EC2-instance. De backend verwerkt deze verzoeken, voert indien nodig databasequeries uit en stuurt de resultaten terug naar de frontend.

De database draait op een afzonderlijke EC2-instance, zodat prestaties en beveiliging apart beheerd kunnen worden. Dit biedt een duidelijke scheiding van verantwoordelijkheden binnen de infrastructuur.

Alle communicatie tussen frontend, backend en database is beveiligd. We maken gebruik van CORS-beleid (Cross-Origin Resource Sharing) om uitsluitend toegestane oorsprongen toe te laten, wat voorkomt dat onbevoegde frontends verbinding maken met onze backend.

Tot slot wordt de volledige infrastructuur gehost binnen een VPC (Virtual Private Cloud). Dit is een virtueel, geïsoleerd netwerk binnen AWS waarin al onze services zich bevinden. Hierdoor is het systeem volledig afgeschermd van de buitenwereld, tenzij expliciet anders geconfigureerd. De VPC biedt zowel beveiliging als controle over netwerksegmentatie, firewallregels en toegangsbeheer.

# Besluit

In dit project ontwikkelden we een volledig functioneel sensorplatform waarmee omgevingsdata efficiënt kan worden verzameld, verwerkt, gevisualiseerd en geïnterpreteerd. Door gebruik te maken van moderne technologieën zoals Spring Boot, Angular, TimescaleDB en AWS, slaagden we erin een schaalbare, performante en gebruiksvriendelijke applicatie op te bouwen.

We kozen bewust voor een modulaire architectuur en pasten best practices toe op het vlak van databankmodellering, gebruikerservaring en softwaretesting. Zo konden we zowel aan de technische als de functionele vereisten van het projectplan voldoen. Dankzij het gebruik van onder meer unit- en integratietests en een goed uitgeruste CI/CD-pijplijn, konden we de codekwaliteit voortdurend bewaken en optimaliseren.

Een belangrijk doel binnen dit project was het verkennen van ThingsBoard en het integreren van sensoren via MQTT. Ondanks onze beperkte voorkennis op het gebied van embedded systemen en IoT-integraties, slaagden we erin om betrouwbare communicatie tussen de sensoren en de backend tot stand te brengen. We ontwikkelden zelfs een eigen JSON-wizard, waarmee eindgebruikers op een intuïtieve manier sensorconfiguraties kunnen aanmaken.

We kijken met trots terug op de realisatie van dit platform. Niet alleen functioneert het technisch zoals verwacht, maar het biedt ook een duidelijke en toegankelijke gebruikersinterface waarmee snel inzichten kunnen worden verkregen in omgevingsdata. Dankzij de flexibiliteit van het systeem is het bovendien klaar voor uitbreiding of integratie met andere sensortechnologieën.

Voor de toekomst bevelen we aan om:

* het platform verder te testen in reële omstandigheden
* het aantal ondersteunde sensortypes uit te breiden
* gebruikersfeedback actief te verzamelen om de gebruikerservaring verder te verfijnen

Tot slot kunnen we besluiten dat de vooropgestelde doelstellingen werden behaald. We hebben niet alleen een technische oplossing gerealiseerd, maar ook waardevolle inzichten verworven in het opzetten van een professioneel softwareproject — van concept tot productie.

Literatuurlijst

Timon. (2020, July 9). Wat is nu precies het verschil tussen Scrum en Agile? - Geweldige service, blije klanten - TOPdesk. *TOPdesk - NL*. https://www.topdesk.com/nl/blog/verschil-scrum-agile/

AI-gebruik in dit document

Artificiële intelligentie (AI) wordt steeds populairder, en terecht: het verhoogt de productiviteit en kan bijdragen aan een vlottere, beter afgewerkte tekst. Toch is transparantie over het gebruik ervan belangrijk, zeker binnen een academische of professionele context. Daarom geef ik graag mee op welke manier AI werd ingezet bij het opstellen van dit document.

Bij het schrijven van dit verslag heb ik AI gebruikt als hulpmiddel bij het verbeteren van de leesbaarheid van mijn teksten. Voor elk hoofdstuk schreef ik eerst een eigen, ruwe versie waarin ik alle inhoudelijke elementen verwerkte die ik belangrijk vond. Deze tekst plakte ik vervolgens in ChatGPT met de volgende opdracht:

**“Verbeter deze tekst, maak het vlotter leesbaar en professioneler, groepeer logische stukken, en corrigeer taalfouten. Wijzig de inhoud niet.”**

Door deze aanpak kon ik mijn eigen inhoudelijke input behouden, terwijl de taal, structuur en stijl werden verfijnd. Alle gegenereerde resultaten werden uiteraard steeds kritisch nagelezen en waar nodig handmatig aangepast.

Bijlagen

**Bijlage A** – Stagevoorstel Faros 2025 – *Meten is weten*  
Bestand: Stagevoorstel Faros 2025 - Meten is weten.docx

**Bijlage B** – Projectplan Faros 2025  
Bestand: Projectplan Faros 2025.docx