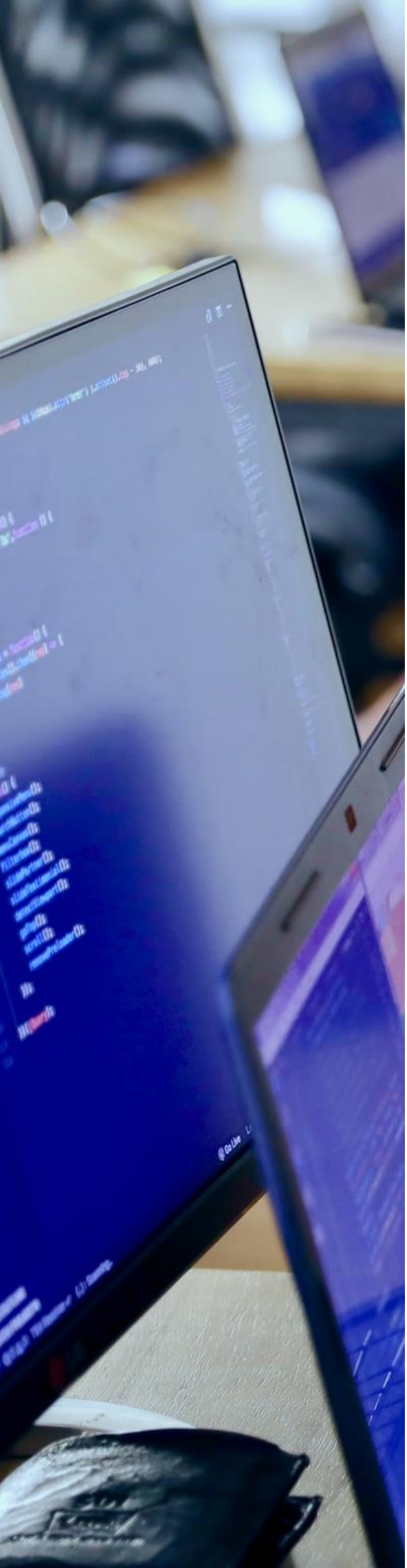
# Technisch Document

Arno Van den Bergh / Stage XitechniX / 2023



# INHOUDSOPGAVE


INHOUDSOPGAVE ..... 2

Inleiding ..... 3

Technologie stack ..... 4

Modelleren ..... 7

Technische realisaties ..... 12

# INLEIDING

In dit technische document wordt een gedetailleerd overzicht gegeven van de gebruikte technologie stack, het modelleren en de technische realisaties binnen het stageproject. De verschillende onderdelen van de stack, van frontend tot backend, worden behandeld, evenals belangrijke aspecten zoals de databasemodellering en de implementatie van specifieke functionaliteit.

In het eerste deel wordt de focus gelegd op de technologie stack. De frontend-, backend- en database technologieën worden toegelicht. Bij het bespreken van de verschillende technologieën zal telkens een beschrijving van de technologie worden gegeven met aansluitend een persoonlijke ervaring.

Het tweede deel van dit document richt zich op het modelleren van de applicatie. Hierin wordt aandacht gegeven aan de boom structuur waar het gross van de data in dit stageproject op steunt en het volledige klassendiagram dat wordt geïplementeerd.

Als laatste worden de technische realisaties van het project besproken. Er wordt gestart met een algemeen overzicht, waarna dieper wordt ingegaan op de meer interessante verwezenlijkingen in frontend en backend.

Dit document is bedoeld om een gedetailleerd inzicht te geven in de gebruikte technologieën, het modelleren en de technische implementaties in het project. Om een beter zicht te krijgen over het functionele aspect van het project wordt er aangeraden om eerst het functioneel document door te lezen.

# TECHNOLOGIE STACK

Voor deze stage werd een technologie stack aangereikt die intern binnen Xitechnix ook gebruikt wordt. Hieronder worden de verschillende technologieën besproken die gebruikt werden voor de frontend, backend en database. Daarbij volgt telkens een korte beschrijving van elk van de technologieën met aansluitend de ervaringen en uitdagingen die naar voren kwamen bij het gebruiken van de technologie.

## Frontend

### React

**Beschrijving**

React is een javascript library waarmee je vrij efficiënt, met behulp van componenten, een user interface kan opbouwen.

**Ervaring**

Persoonlijk ben ik wel een fan van React, zeker als je dit gaat vergelijken met andere component based libraries en frameworks zoals bijvoorbeeld Angular. Ik vind het leuk dat de logica, template en styling allemaal gegroepeerd zit in één bestand. Een challenge dat daar dan ook wel bij komt kijken is dat componenten snel onoverzichtelijk kunnen worden doordat de code snel groeit. Hierbij is het belangrijk dat je op tijd je code splitst over verschillende componenten zodat iedere component zijn eigen verantwoordelijkheid behoudt.

### Typescript

**Beschrijving**

Typscript is als programmeertaal een strikte superset van JavaScript en wordt voor runtime ook gecompileerd naar javascript. Typescript voegt typing toe aan javascript, wat als grote voordeel heeft dat type errors kunnen worden opgevangen bij compileren. Daarbovenop introduceert typescript ook auto completion doordat de types achter de code ook telkens gekend zijn.

**Ervaring**

In combinatie met React was het voor mij in het begin wel even aanpassen, maar eens ik daarmee weg was vond ik Typescript vooral een hulp. Met de type errors kan je bugs uit je code halen nog voor de code gecompileerd is en ook de auto completion viel bij mij goed in de smaak.

### Tailwind

**Beschrijving**

Tailwind is een css framework dat heel dicht bij css zelf ligt. De verschillende tailwind klassen zijn bijna één op één te mappen op de gelijknamige css properties.

**Ervaring**

Ik vind Tailwind een heel sterke tool vanwege 3 redenen.

- Ik hoef zelf geen klassen meer te schrijven want ik kan de Tailwind klassen meteen toevoegen in mijn componenten. Zeker in combinatie met React vind ik dit erg fijn want dan hoef ik mijn styling niet te voorzien in een apart css bestand, maar zit meteen geïntegreerd in mijn component.
- Vergeleken een css framework zoals bootstrap heb je met tailwind veel meer vrijheid. Bij een framework zoals bootstrap zie je nogal snel dat dit een “bootstrap” site is. Je moet al heel wat custom css schrijven om daar je eigen draai aan te geven. Dat is bij tailwind niet het geval omdat de klassen heel dicht liggen tegen puur css.
- Daarbovenop heeft Tailwind ook een sterke community. Telkens als ik een wel gekend UI component wou toevoegen. Startte ik mijn zoektocht met een simpele “google” search en dan krijg je genoeg voorbeeld componenten die je makkelijk kan aanpassen zodat deze binnen jouw layout passen.

Het enige minpuntje aan Tailwind voor mij is dat de structuur van de html wat verloren kan gaan als je te veel tailwind klassen toevoegd. Maar ook dit kan weer opgevangen worden door je componenten op tijd op te splitsen.

### React router

**Beschrijving**

React router is de meest populaire javascript library voor react om frontend page routing te kunnen gebruiken.

**Ervaring**

Met mijn huidige kennis kan ik react router niet echt met iets anders vergelijken. De routing gebeurt vrij intuïtief, wat het makkelijk maakt om te kunnen gebruiken. Ook de documentatie is in orde, al had ik op de officiële website graag wat meer in depth tutorials en voorbeelden gezien.

### Redux toolkit

**Beschrijving**

Redux toolkit is de nieuwe versie van Redux, een tool voor global state management die vaak in combinatie met React gebruikt wordt.

**Ervaring**

Ik heb deze tool vanuit mijn stage opgelegd gekregen omdat ze daar zelf in hun eigen projecten ook mee werken. Persoonlijk vond ik global state management in mijn stage project een twijfel geval. Het is zeker geen must en het vereist wel wat setup om hier vlot mee aan de slag te kunnen.

In vergelijking met andere global state management libraries zoals Recoil vind ik Redux Toolkit wel heel uitgebreid, wat het voor een project van kleinere omvang wat overbodig maakt.

Los daarvan biedt de tool wel heel wat opties naar debugging toe wat het voor grotere teams wel interessant kan maken.

### Axios

**Beschrijving**

Axios is een heel eenvoudige http request client.

**Ervaring**

Mijn ervaring loopt eigenlijk volledig analoog met de beschrijving. Het werkt zeer eenvoudig en doet wat het moet doen.

# Backend

### ASP.NET

**Beschrijving**

ASP.NET is deel van het .NET developer platform wat bestaat een hoop tools, programmeer talen en libraries waarmee je verschillende types van applicaties kan maken. ASP.NET is specifiek gericht op het bouwen van web applicaties.

**Ervaring**

Zelf heb ik ASP.NET gebruikt in combinatie met C#. Het duurt voor mij altijd even om comfortabel te worden met zo'n groot platform. Werken met ASP.NET biedt zoveel mogelijkheden en als beginner het soms lastig om door het bos de bomen nog te zien. Het basis project waar ik van gestart ben was een ASP.NET Rest API applicatie. Daar bestaan wel enkele goede tutorials voor om mee van start mee te geraken, maar in het begin was het zeker zoeken.

### Mongodb.driver

**Beschrijving**

MongoDB driver voor .NET/C# is de officiële driver voor .NET wat het mogelijk maakt om vanuit de code te communiceren met je MongoDB database.

**Ervaring**

Ik vond het over het algemeen wel aangenaam werken met MongoDB driver. Je bent er vrij in of je de driver gebruikt in combinatie met POCO's (gewone c# objecten) of BsonDocumenten (de MongoDB default). Voor het grootste stuk kan je jezelf wel behelpen met POCO objecten, wat het makkelijkst werkt. Maar specifieke gevallen kan je er niet buiten om te werken met BsonDocumenten in dat geval kan je niet genieten van de Auto Completion en de beter foutopsporing die POCO's wel bieden, maar het geeft meer vrijheid.

# Database

### Mongodb

**Beschrijving**

MongoDB is een NoSQL database programma, dat gebruikt maakt van collecties en documenten om zijn data in te structureren.

**Beschrijving**

Ik ben persoonlijk wel fan van MongoDB, juist vanwege de NoSQL aanpak. Ik vind, zeker tijdens development, dat dat zijn vruchten afwerpt. Doorheen het programmeren heb ik regelmatig velden

moeten toevoegen aan mijn databank. Omdat MongoDB geen schema vereist ging dat ook perfect, vaak zelfs zonder mijn data in de database te hoeven aanpassen.

# MODELLEREN

Één van de challenges waar ik tijdens deze stage tegen aanliep was het modelleren van de data binnen het systeem. Het doel van de stage was om een NoSQL database documentatie systeem in elkaar te steken en daarbij kwamen voor mij twee vragen naar boven.

- Hoe modelleer ik het datamodel dat achter een databank of systeem zit zodat ik dit achteraf terug kan ophalen?
- Hoe structureer ik meerdere modellen binnen het database documentatie systeem?

Mijn antwoord op de eerste vraag zal worden uitgelicht in volgend deeltje “Model tree structure”. Het antwoord op de tweede vraag is me door de stage gever aangereikt geweest, dat bespreek ik hieronder in “Klassen diagram”.

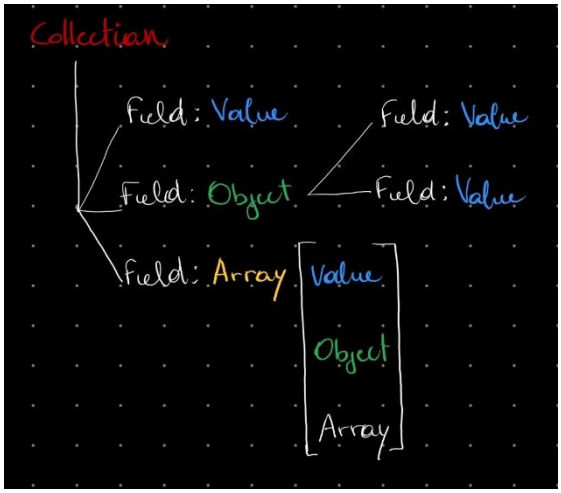
## Model tree structure

Hoe modelleer ik het datamodel dat achter een databank of systeem zit zodat ik dit achteraf terug kan ophalen?

Bij deze vraag moet er ook rekening worden gehouden met het feit dat elk datamodel van elkaar verschilt. Het antwoord moet daarom veralgemeenbaar zijn naar alle mogelijke modellen, en niet alleen het Humqu model.

**De eerste draft**

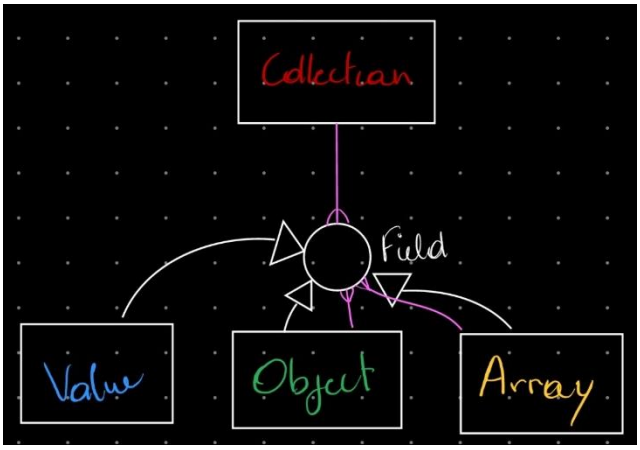
Voor het uitdenken van een manier om een snaphshot van een datamodel te modelleren ben ik begonnen met het uittekenen van een algemene structuur van een document based NoSQL datamodel.




Meestal bevat een document binnen een NoSQL database JSON data. Zo heb ik mijn algemeen schema dan ook opgebouwd. Zo kan elk document binnen een collectie 3 verschillende soorten velden bevatten:

- Een veld kan **een enkele waarde** van een bepaald datatype bevatten: een naam, telefoonnummer, datum, …
- Een veld kan een **object** bevatten: een adres met straat, gemeente, postcode, …
- Een veld kan een **array** bevatten: een lijst van telefoonnummers

Van hieruit is dan ook mijn eerste schets voor het klassendiagram ontstaan. Een Collectie klasse die verschillende Velden onder zich heeft. Waarbij een veld Een gewone Waarde, een object of een Array kan zijn. Objecten en Arrays kunnen op hun beurt ook terug velden bevatten.

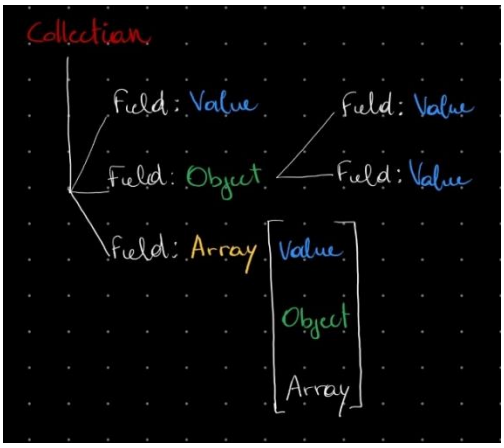


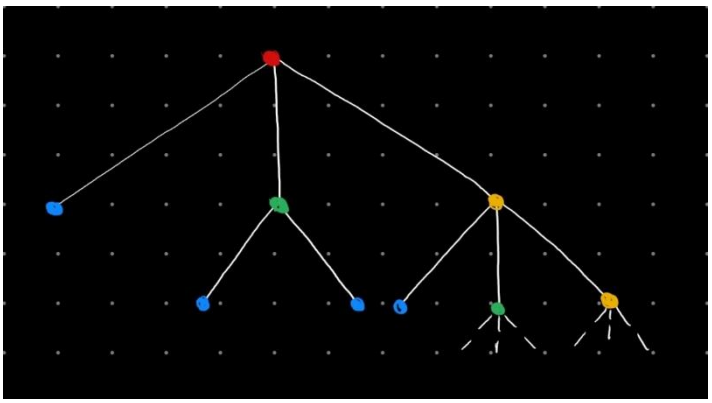

Op het eerste zicht leek dit voor mij de meest voor de hand liggende oplossing, maar qua uitbreidbaarheid en simpliciteit in de code was ik nog niet helemaal tevreden.

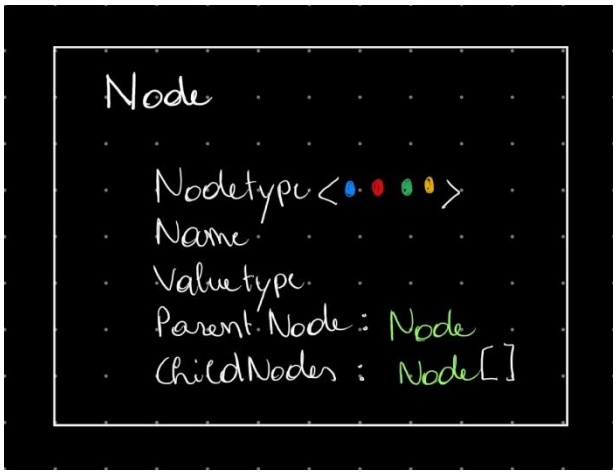
**Een boom diagram**

Als ik een probleem doorheen de dag niet opgelost krijg neem ik dat (als is het onbewust) mee naar huis en blijf ik daar wel eens op door denken. Soms kan dat tot interessante resultaten leiden. Binnen mijn stage is dit daar een mooi voorbeeld van.

In de lessen AI dit jaar hebben we kennis gemaakt met enkele “tree search algorithms”. Het is van daaruit dat ik de data structuur ben beginnen zien als een boom structuur.





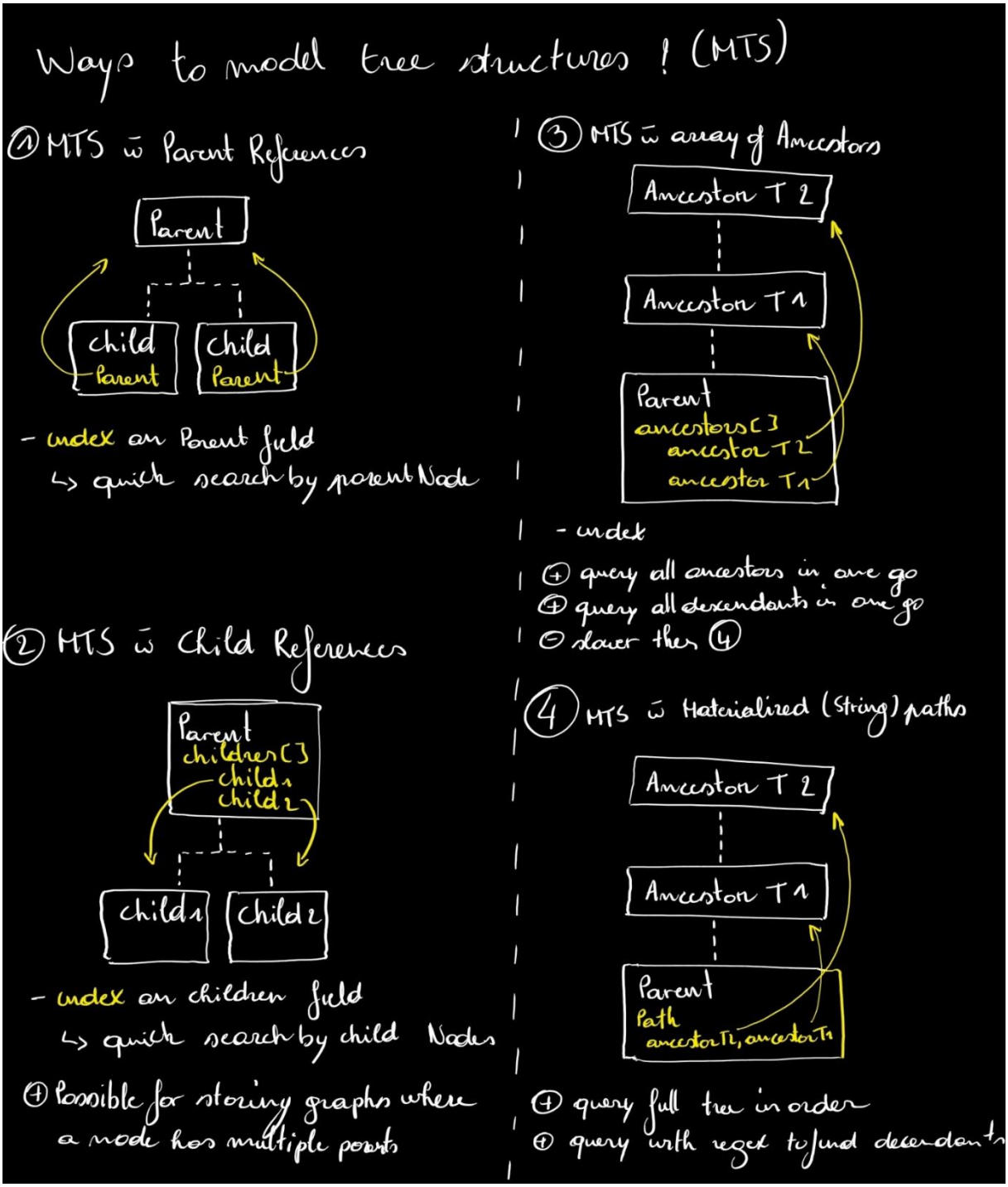

Op deze manier kan je de kan je de verschillende klassen die hierboven geïntroduceerd werden herleiden tot één Node klasse.

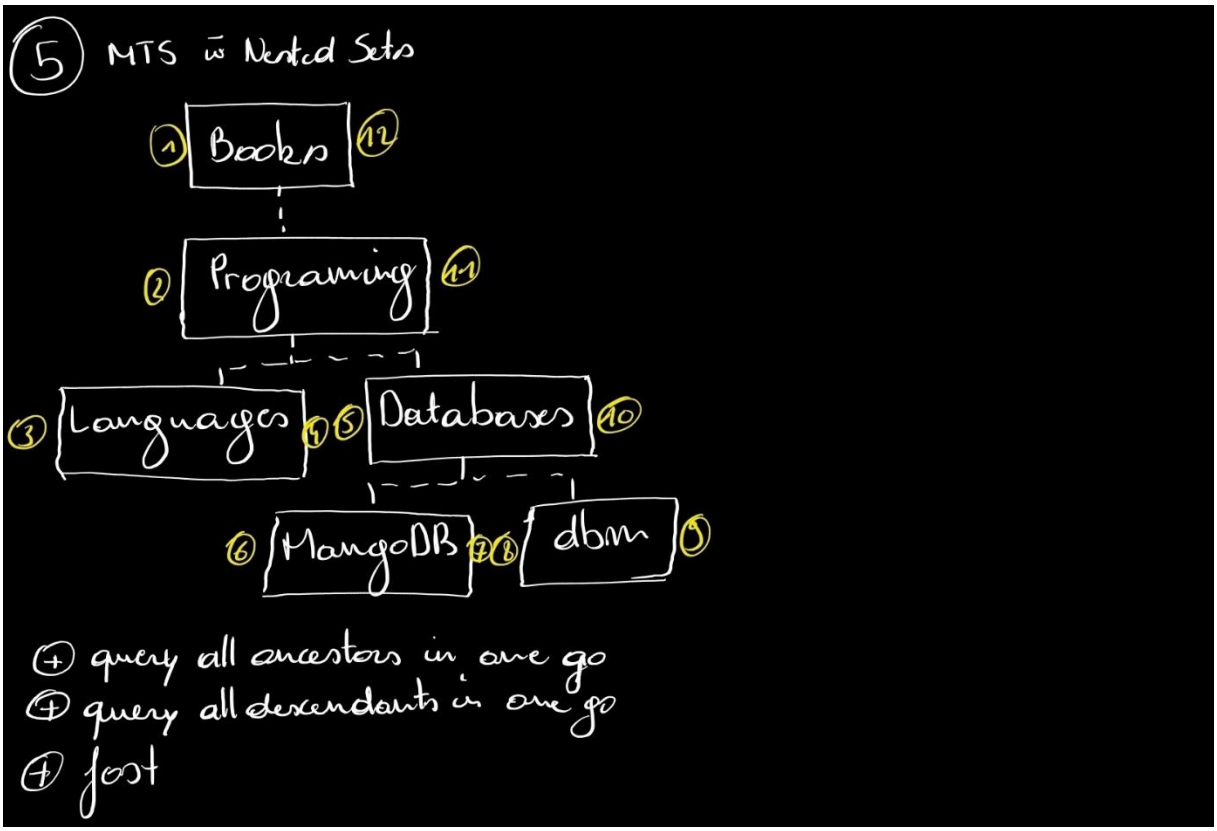
Zo hoeft de code bij het maken van een snapshot maar rekening te houden met één klasse. Kan al deze data in één collectie worden weggeschreven in de database, wat het schrijven van query’s een pak makkelijker maakt.

**De relaties tussen de nodes bijhouden**

Een node op zich heeft weinig betekenis. Het zijn de relaties tussen de verschillende de nodes die de boom vorm geven. Intuïtief was mijn eerste aanvoelen om voor elke node de direct parent node en child nodes bij te houden. Op deze manier kan je de boom van een node uit tekenen door over de parent en child nodes te itereren.

Deze manier van structureren brengt ook wel wat beperkingen met zich mee. Zo kan je met één query hoogstens de direct parent en / of directe child nodes terugvinden. Wanneer je ineens alle ascendant of descendant nodes wilt ophalen moet je meerdere query's doen. Deze beperking had me curieus gemaakt naar welke andere mogelijkheden er nog zijn om de relaties binnen een boom structuur te modelleren. Daarom heb ik tijdens mijn stage even de tijd genomen om een kleine case study te doen rond dit onderwerp. Om niet te ver uit te weiden plaats ik hieronder gewoon de samenvatting die ik voor mezelf had gemaakt en bespreek ik de gekozen methode.






Voor mijn stageproject ben ik gestrand op de 5de optie. De heeft namelijk twee grote troeven. Je kan in één trek alle ancestor en descendant nodes met een query ophalen. Met deze methode houd je als het ware een pad bij doorheen de boom: je houdt een linker en rechter index bij. Door middel van een simpele vergelijking met deze indexen kan je makkelijk alle nodige nodes ophalen. Dat brengt ineens het tweede voordeel naar boven. Doordat het gaat om eenvoudige vergelijking tussen getallen is deze manier ook erg snel. Het enige nadeel is dat het lastig is om achteraf nodes toe te voegen aangezien dan het volledige pad opnieuw getekend moet worden. Gelukkig is het niet de bedoeling om in dit project achteraf nog nodes te gaan toevoegen.

## Klassen diagram

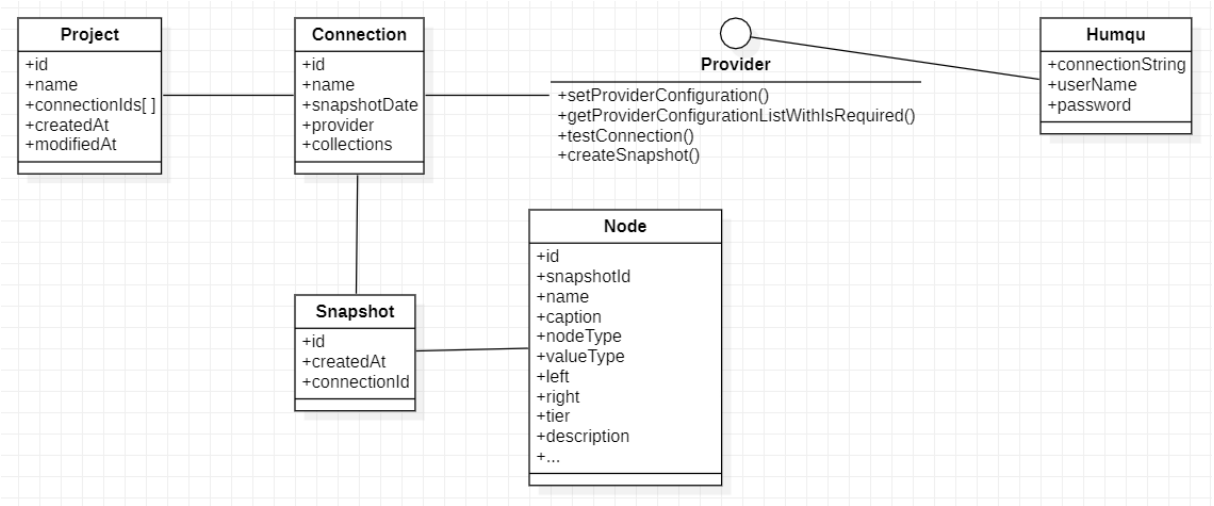
Nu dat de eerste vraag van het modelleren beantwoord is kan er meteen worden overgegaan naar vraag 2:

Hoe structureer ik meerdere modellen binnen het database documentatie systeem?

In samenspraak met mijn stage mentor is er gekozen om de snapshots te structureren in projecten en connecties. Zo is het de bedoeling dat je een project kan aanmaken en daar verschillende connecties kunt toevoegen en voor elk van die connecties snapshots kunt maken.

Achter een connectie zit een provider van een bepaald systeem die het mogelijk om een snapshot te maken van het gekozen systeem. Belangrijk hierbij was dat bij het aanmaken van een connectie de gebruiker zelf kan kiezen welke provider die wil gebruiken en er dus ook ondersteuning moet zijn voor meerdere gebruikers. (Zie verder voor meer uitleg).

Al het bovenstaande wordt opgevangen in volgend klasse diagram:


Project
+id
+name
+connectionIds[ ]
+createdAt
+modifiedAt
Connection
+id
+name
+snapshotDate
+provider
+collections
Provider
+setProviderConfiguration()
+getProviderConfigurationListWithIsRequired()
+testConnection()
+createSnapshot()
Humqu
+connectionString
+userName
+password
Snapshot
+id
+createdAt
+connectionId
Node
+id
+snapshotId
+name
+caption
+nodeType
+valueType
+left
+right
+tier
+description
+...

# TECHNISCHE REALISATIES

Het volledige project kan gezien worden als een verzameling van technische realisaties die door met elkaar in communicatie te gaan het eind product opleveren. In deze sectie worden de belangrijkste realisaties apart van elkaar besproken. Daarbij wordt er eerst algemeen gekeken over het hele project heen. Nadien wordt er gekeken naar de frontend en backend afzonderlijk.

## Algemeen

### Architectuur

Omdat het voor mij de eerste keer is dat ik in een project ben gestapt van deze grote, heb ik stil moeten staan bij de opbouw van mijn code. In de klas hebben we hier natuurlijk een aantal concepten en best practices rond gezien, maar elk project verschilt natuurlijk van elkaar. Vandaar dat ik voor mezelf een weg heb gezocht om de backend en frontend van dit project vorm te geven.

**Backend**

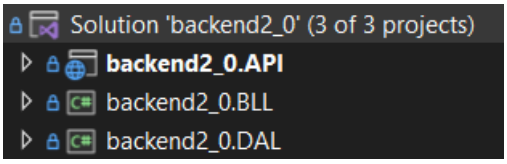
Voor de backend heb ik in mijn zoektocht voornamelijk 2 architecturen zien terugkomen voor het structuren van de code. Een onion architectuur waarbij de domein klasse het centrum van het project zijn en N-lagen model waarbij je in de bovenste laag de UI hebt en in de onderste laag de connectie met andere instantie zoals database.

Onion architectuur

Hierbij heb je helemaal centraal in het project de domein entiteiten en al de andere lagen zijn daar omheen gebouwd. Op die manier ligt de focus niet op de technologieën en de gebruikte frameworks, maar is alles afhankelijk van de domein entiteiten. Voor kleinere projecten kan dit wat overkill zijn. Omdat ik zelf nog geen ervaring heb met dit model en mijn project niet van zo'n grote omvang is ben ik hier niet veel dieper op in gegaan.

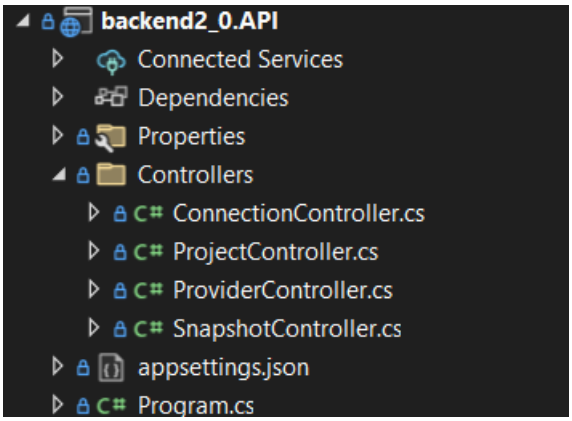
N-lagen architectuur

Bij de N-Lagen architectuur heb je verschillende lagen die elks hun eigen verantwoordelijkheid hebben. Deze architectuur is geschikt voor kleinere projecten. Vandaar dat ik er voor gekozen heb om hier in verder te gaan en een model met 3 lagen te hanteren.

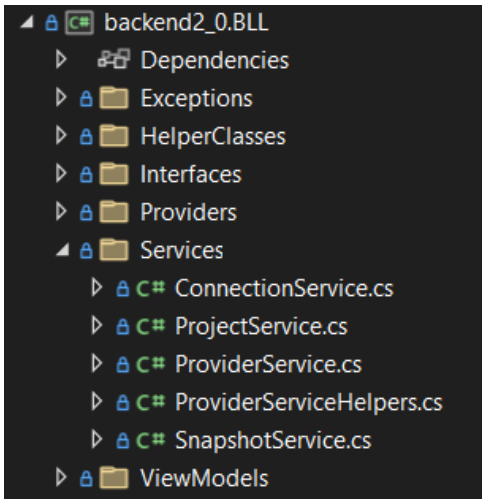



De relaties tussen deze layers moeten zo worden opgebouwd dat een layer enkel maar een depency kan hebben op een layer die er onder ligt. In dit voorbeeld kan de DAL layer dus geen dependency hebben op BLL layer, maar omgekeerd wel.

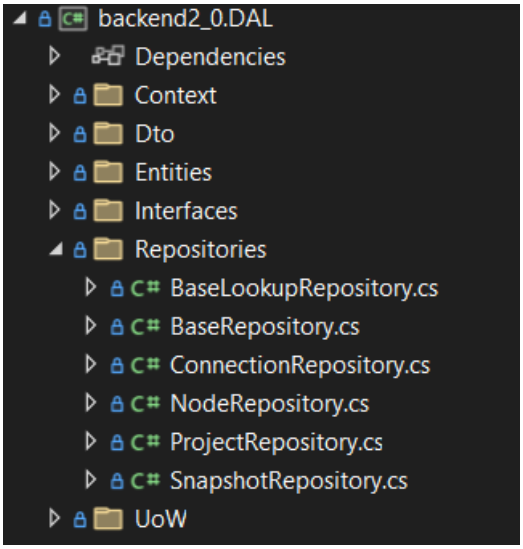
- UI Layer (API): Deze layer is verantwoordelijk voor de communicatie met de eindgebruiker. In mijn project is dit een REST API die json data uitwisselt met de frontend. Hier vind je voornamelijk de controllers terug waarmee de frontend kan communiceren.




- Business Logic Layer: Deze layer is verantwoordelijk voor de verwerking van de inkomende en uitgaande data. Wanneer deze layer data van de gebruiker ontvangt is het de taak van deze layer om de data te converteren naar een formaat waarmee de onderste layer aan de slag kan. Daarnaast worden business regels hier afgedwongen. De core componenten van deze layer zijn de verschillende services, daarrond draaien de interfaces, viewmodels, …



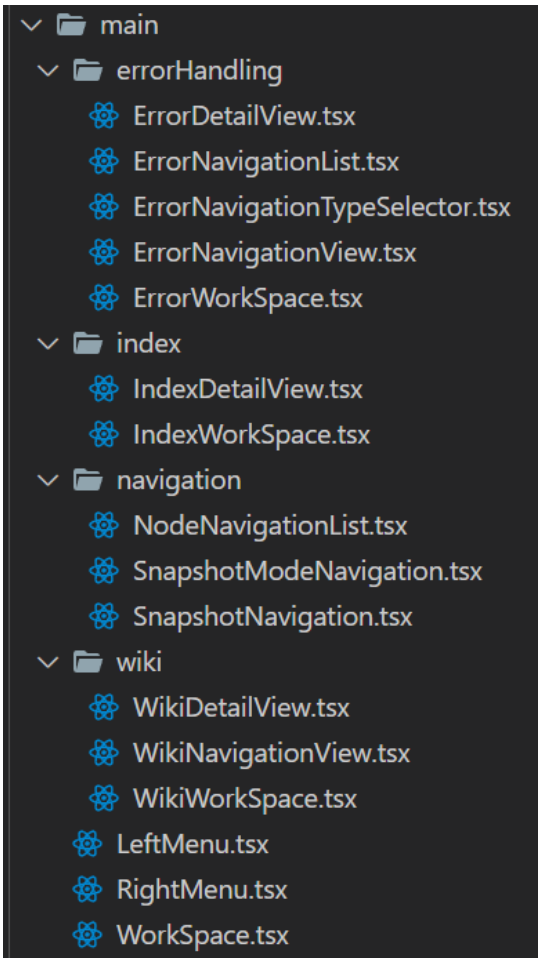

- Data Access Layer: Deze layer is verantwoordelijk voor de communicatie met de gepersisteerde data die bijvoorbeeld afkomstig is uit een database. In mijn project praat deze layer met de MongoDB database. Om de communicatie met de bovenstaande layers eenvoudig te houden gebruik ik hiervoor het repository pattern.



**Frontend**

Naar mijn eigen ervaring heb je bij het vormgeven van de frontend meer vrijheid en is dit sterk afhankelijk van je project. Voor de frontend ben ik eigenlijk niet echt op onderzoek uitgetrokken omdat dit vrij natuurlijk tot mij kwam.

Zo heb ik de componenten voor de main functionaliteiten gestructureerd per functionaliteit en binnen elke functionaliteit per deelvenster.




### Projecten managen

Het doel van de project management schermen is om de verschillende snapshots die al gemaakt zijn en nog gemaakt moet worden gestructureerd onder te brengen en dat de gebruiker op een gebruiksvriendelijke manier snapshot kan maken. Daarbij speelt de provider een belangrijke rol. Zo kan de gebruiker bij het aanmaken van een nieuwe connectie zelf kiezen welke provider er achter die connectie moet zitten.

**Dynamische provider**

Om dit mogelijk te maken is er in de backend een IProvider interface in het leven geroepen waarvan concrete providers kunnen overerven. Op die manier kan de code makkelijk worden uitgebreid met nieuwe providers. De rest van de code is ook zo rond de IProvider opgebouwd dat wanneer een provider de interface correct implementeert de rest van de code ook niet hoeft worden aangepast.

```
5 references | AzureAD\ArnoVandenBergh, 20 days ago | 1 author, 2 changes
public interface IProvider
{
    4 references | AzureAD\ArnoVandenBergh, 45 days ago | 1 author, 1 change
    void SetProviderConfiguration(Dictionary<string, object> configuration);
    3 references | AzureAD\ArnoVandenBergh, 45 days ago | 1 author, 1 change
    Task<IDictionary<string, string>> TestConnectionAsync(CreateConnectionModel connection, IDictionary<string, string> messages);
    3 references | AzureAD\ArnoVandenBergh, 20 days ago | 1 author, 1 change
    Task CreateSnapshotAsync(Dictionary<string, object> configuration, string connectionId);
    3 references | AzureAD\ArnoVandenBergh, 20 days ago | 1 author, 1 change
    IDictionary<string, bool> GetProviderConfigurationListWithIsRequired();
}
```

De belangrijkste twee methodes die worden overgeërfd van de interface zijn de “CreateSnapshotAsync” en “TestConnectionAsync” methode. De eerste staat in voor het creëren van de snapshot en wordt verder in dit document nog uitvoerig besproken, en de tweede methode gaat controleren of de provider connectie kan maken met de externe data source. Die laatste methode wordt dieper besproken in volgende deeltje. De andere methodes worden door de frontend gebruikt om de configuratie info van de provider door te geven aan de backend zodat de concrete provider klasse voldoende info heeft om snapshots te maken.

De communicatie tussen backend en frontend omtrent de provider is ook zo opgebouwd dat de frontend zijn configuratie formulier voor de connectie dynamisch kan aanpassen op basis van de gekozen provider (de backend stuurt de configuratie velden telkens mee naar de frontend: “GetProviderConfigurationListWithIsRequiered”)

```
2 references | AzureAD\ArnoVandenBergh, 20 days ago | 1 author, 1 change
public IDictionary<string, bool> GetProviderConfigurationListWithIsRequired()
{
    //Al de configuratie properties van de Humqu provider worden meegegeven
    //aan de frontend in de vorm van een Dictionary.
    var dictionary = new Dictionary<string, bool>
    {
        { nameof(ConnectionString), true },
        { nameof(Username), false },
        { nameof(Password), false },
        { nameof(HumquModel.Database), true },
        { nameof(HumquModel.FilterKey), false },
    };
    return dictionary;
}
```

*Implementatie van de Humqu provider*

Wanneer de frontend dan de ingevulde configuratievelden terug stuurt wordt in de backend de “SetProviderConfiguration” aangesproken.

```
3 references | AzureAD\ArnoVandenBergh, 20 days ago | 1 author, 1 change
public void SetProviderConfiguration(Dictionary<string, object> configuration)
{
    //De configuratie Dictionary die wordt meegegeven vanuit de frontend
    //wordt hier gemapt aan de configuratie properties van de Humqu provider.
    ConnectionString = configuration[nameof(ConnectionString)].ToString();
    Username = configuration[nameof(Username)].ToString();
    Password = configuration[nameof(Password)].ToString();
    var properties = typeof(HumquModel).GetProperties();
    ModelConfiguration = new HumquModel();
    foreach (var property in properties)
    {
        typeof(HumquModel).GetProperty(property.Name)
            .SetValue(ModelConfiguration, configuration[property.Name]
            .ToString());
        var test = typeof(HumquModel).GetProperty(property.Name)
            .GetValue(ModelConfiguration);
    }
}
```

*Implementatie van de Humqu provider*

**Connecties testen**

Wanneer een gebruiker een nieuwe connectie wilt toevoegen aan een project, moet steeds gecontroleerd worden of er met de meegegeven configuratie gegevens voor de provider een connectie tot stand kan worden gebracht met de externe data source. Wanneer dit niet het geval is moet de gebruiker de juiste feedback krijgen om de configuratie gegevens te corrigeren.

De “TestConnectionAsync” methode is verantwoordelijk voor dit gedrag. Deze methode wordt opgeroepen bij het aanmaken en wijzigen van een connectie en elke keer er een snapshot wordt gemaakt voor die connectie.

```
2 references | AzureAD\ArnoVandenBergh, 20 days ago | 1 author, 1 change
public async Task<IDictionary<string, string>> TestConnectionAsync(CreateConnectionModel connection, IDictionary<string, string> messages)
{

    try
    {
        //De connectie string wordt opgebouwd met de meegeleverde configuratie gegevens.
        var connectionstring = GetConnectionString();
        //Er wordt geprobeerd een connectie op te zetten
        var testClient = new MongoClient(GetConnectionString());
        var task = testClient.ListDatabaseNamesAsync();
        var completedTask = await Task.WhenAny(task, Task.Delay(TimeSpan.FromSeconds(6)));

        //Mogelijke fouten in de configuratie worden opgevangen en teruggestuurd naar de gebruiker
        if (completedTask != task)
        {
            throw new Exception();
        }
        else if (completedTask.IsFaulted)
        {
            if (!connectionstring.Contains('@'))
            {
                throw new MongoAuthRequiredException();
            }
            else
            {
                throw new MongoAuthNotRequiredException();
            }
        }
        var cursor = await task;
        var databases = await cursor.ToListAsync();
        if (!databases.Contains(ModelConfiguration.Database))
        {
            throw new MongoCantFindDatabaseException();
        }
    }
    catch (MongoConfigurationException ex)
    {
        messages.Add(nameof(ConnectionString), "The connection string is invalid");
    }
    catch (MongoCantFindDatabaseException ex)
    {
        messages.Add(nameof(HumquModel.Database), "Can't find the given database name on the mongodb server.");
    }
```

*Implementatie van de Humqu provider*

Wanneer de gebruiker een foute configuratie meegeeft wordt dit opgevangen door de backend en geeft deze feedback terug aan de frontend. Vervolgens wordt de feedback op een gepast manier aan de eindgebruiker getoont.

```
ConnectionsApi.postConnection(connection)
  .then(() => {
    submissionHandler(connection);
    setRequestLoadingStatus(false);
  })
  .catch((error) => {
    providerConfigItems.setErrorMessages(error.response.data);
    setRequestLoadingStatus(false);
  });
```

*ConnectionForm.tsx*

De providerConfigItems variabele is een object dat wordt teruggegeven door de custom “useInputValidatorForAnyObjectWithState” hook. Dit object is verantwoordelijk voor de validatie en state van de configuratie velden van de connectie, zo ook het verwerken van de feedback van de ingevulde configuratie velden. Hierover later meer (Custom value validation hooks).

### Main functionaliteit

Naast het managen van de snapshots in de besproken projecten structuur, kan een gebruiker in interactie gaan met de snapshots door gebruik te maken van de drie main functionaliteiten: Wiki, Index, Fout opsporing. De flow tussen backend en frontend is voor elk van de functionaliteiten erg gelijkaardig vandaar dat ik het voorbeeld van de Wiki hieronder even ga toelichten.

**Dataflow**

Zoals in het gedeelte van de frontend architectuur al besproken zijn de componenten van de verschillende functionaliteiten verdeeld over 3 deelvensters: een navigatie venster, een werk venster en een detail venster.

*Navigatie venster*

Voor de Wiki begint het verhaal bij het navigatie venster dat aan de backend vraagt om alle modellen (collecties en views) waarbinnen de gebruiker kan navigeren in REST formaat te versturen:

```tsx
useEffect(() => {
  SnapshotsApi.getAllCollectionsOfSnapshot(snapshotId!).then((result) => {
    const tempCollections = result.data as Collection[];
    setCollections(tempCollections);
    setCurrentCollections(
      tempCollections.slice(
        0,
        getIndexArray(tempCollections, COLLECTIONS_PER_PAGE, {
          id: (item: any) => item.id,
          name: (item: any) => item.name,
        })[0]
      )
    );
  });
}, [snapshotId]);
```

*WikiNavigationView.tsx*

Wanneer het navigatie venster van de wiki functionaliteit in de DOM wordt getrokken, gaat de useEffect hook een request sturen naar de backend om een lijst met alle collecties te verkrijgen. Deze collecties worden in state opgeslagen zodat de rest van het component hier gebruik van kan maken.

In de backend komt deze request aan in de Snapshot Controller bij het juiste endpoint. Daar wordt er eerst gekeken of de meegegeven snapshotId bestaat. Als dat het geval is worden alle collectie nodes terug gegeven. De snapshot service is verantwoordelijk voor het teruggeven van de juist nodes.

```cs
[HttpGet("collection/{snapshotId}")]
0 references | AzureAD\ArnoVandenBergh, 21 days ago | 1 author, 1 change
public async Task<ActionResult<IEnumerable<NodeEntity>>> GetSnapshotCollectionsAsync(string snapshotId)
{
    var snapshot = await _snapshotService.GetByIdAsync(snapshotId);

    if (snapshot == null)
    {
        return NotFound();
    }

    var snapshotNodes = await _snapshotService.GetAllCollectionsOfSnapshotAsync(snapshotId);


    return Ok(snapshotNodes);
}
```

*SnapshotController.cs*

De snapshot service spreekt op zijn beurt weer de node repository aan. Deze drie instanties (controller, service en repository) vertegenwoordigen meteen ook de drie layers in de backend structuur.

```cs
2 references | AzureAD\ArnoVandenBergh, 21 days ago | 1 author, 1 change
public async Task<IEnumerable<NodeEntity>> GetAllCollectionsOfSnapshotAsync(string snapshotId)
{
    return await _nodeRepository.GetAllCollectionsOfSnapshotAsync(snapshotId);
}
```

*SnapshotService.cs*

```
2 references | AzureAD\ArnoVandenBergh, 21 days ago | 1 author, 1 change
public async Task<IEnumerable<NodeEntity>> GetAllCollectionsOfSnapshotAsync(string snapshotId)
{
    var all = DbSet.Find(Builders<NodeEntity>.Filter.Eq(s => s.SnapshotId, snapshotId)
        & Builders<NodeEntity>.Filter.In("NodeType", new List<int>() { 0, 1 }))
        .SortBy(n => n.Name);
    return await all.ToListAsync();
}
```

*NodeRepository.cs*

Wanneer deze collecties en views dan worden teruggegeven aan de frontend leg ik de verantwoordelijkheid bij de frontend om daar paginering en een zoek actie voor te steken.

```
import { useEffect, useMemo, useState } from "react";
import InputText from "../../../ui/inputs/InputText";
import NavList from "../../../ui/inputs/NavList";
import Collection from "../../../models/Collection";
import SnapshotsApi from "../../../helpers/SnapshotsApi";
import { useLoaderData, useParams } from "react-router-dom";
import NavButtons from "../../../ui/inputs/NavButtons";
import PageNavigation from "../../../ui/inputs/PageNavigation";
import { getIndexArray } from "../../helpers/FieldRenderHelpers";
import IBaseComponent from "../../../models/IBaseComponent";

const COLLECTIONS_PER_PAGE = 17;

const WikiNavigationView = (props: IBaseComponent) => {
  const { snapshotId } = useParams();

  const [collections, setCollections] = useState<Collection[]>([]);
  const [filter, setFilter] = useState("");
  const [index, setIndex] = useState<number>(1);

  const filteredCollections = useMemo(
    () => collections?.filter((c) => c.name.includes(filter)),
    [filter, collections]
  );

  const indexArray = useMemo(
    () =>
      getIndexArray(filteredCollections, COLLECTIONS_PER_PAGE, {
        id: (item: any) => item.id,
        name: (item: any) => item.name,
      }),
    [filteredCollections, COLLECTIONS_PER_PAGE]
  );
  const [currentCollections, setCurrentCollections] = useState<
    Collection[] | undefined
  >(collections?.slice(0, indexArray[0]));
```

```
useEffect(() => {
  SnapshotsApi.getAllCollectionsOfSnapshot(snapshotId!).then((result) => {
    const tempCollections = result.data as Collection[];
    setCollections(tempCollections);
    setCurrentCollections(
      tempCollections.slice(
        0,
        getIndexArray(tempCollections, COLLECTIONS_PER_PAGE, {
          id: (item: any) => item.id,
          name: (item: any) => item.name,
        })[0]
      )
    );
  });
}, [snapshotId]);

useEffect(() => {
  setCurrentCollections(
    filteredCollections?.slice(
      index === 1 ? 0 : indexArray[index - 2],
      indexArray[index - 1]
    )
  );
}, [filter, index]);

const filterHandler = (event: React.ChangeEvent<HTMLInputElement>) => {
  if (index !== 1) {
    setIndex(1);
  }
  setFilter(event.currentTarget.value);
};

return (
  <div className={props.className + " flex flex-col"}>
    <InputText
      search={true}
      placeholder="filter..."
      isValid={true}
      value={filter}
      onChange={(e) => filterHandler(e)}
    />
    <div className="flex flex-col flex-grow">
      <NavList
        className="mt-4 w-full flex-grow"
        items={currentCollections}
        id={(item: any) => item.id}
        name={(item: any) => item.name}
      />
```

```
        <PageNavigation
          maxNumber={indexArray.length}
          number={index}
          setNumber={setIndex}
        />
      </div>
    </div>
  );
};

export default WikiNavigationView;
```

*WikiNavigationView.tsx*

*Werk venster*

De gebruiker kan het navigatie venster nu gebruiken om een model (view of collectie) aan te klikken. De frontend steekt de id van dit model (colId) vervolgens in de url als een query parameter. Dit gebeurt aan de hand van de custom useQueryState hook, waarover later meer:

```
...

const NavList = <TValue,>(props: NavListProps<TValue>) => {
...

  const [colCoordinatesQueryState,setColCoordinatesQueryState] =
useQueryState("colId","");

  const navigationHandler = (col: TValue) => {
    if(props.clickEventHandler){
      props.clickEventHandler();
    }
    setColCoordinatesQueryState(props.id(col));
  }

  return (
    <div className={props.className}>
      <table className="w-full">
        <tbody>
          {props.items &&
            props.items.map((col) => (
              <Fragment key={props.id(col)}>
                {isNewHeaderRequired(props.name(col)!) && (
                  <Tr>
                    <Th>{props.name(col)!.charAt(0).toUpperCase()}</Th>
                  </Tr>
                )}
                <Tr onClick={()=>{navigationHandler(col)}}>
                  <Td className="text-ellipsis overflow-hidden whitespace-
nowrap max-w-[100px] pr-5 hover:overflow-visible hover:whitespace-normal
hover:h-auto hover:break-words">{props.name(col)}</Td>
                </Tr>
              </Fragment>
            ))}
        </tbody>
      </table>
    </div>
  );
};

export default NavList;
```

*NavList.tsx deel component van WikiNavigationView.tsx*

Het wiki werk venster component gaat dan op zijn beurt weer gebruik maken van de useQueryState om de query parameter voor de model id (colId) uit te lezen en deze vervolgens op te nemen in de request link naar de backend om al de nodes op te vragen die behoren tot dat model (alle velden).

```
...

const WikiWorkSpace = (props: WikiWorkSpaceProps) => {
  const [colIdQueryState, setColIdQueryState] = useQueryState("colId", "");
  const [nodeIdQueryState, setNodeIdQueryState] = useQueryState("nodeId", "");

  const [nodes, setNodes] = useState<SnapshotNode[]>([]);

  const directChildNodes = nodes.filter((n) => n.tier == 2);
  const collectionNode = nodes.find((n) => n.tier == 1);

  useEffect(() => {
    if (colIdQueryState !== "") {
      SnapshotsApi.getAllSnapshotNodesOfParentCollectionNode(
        props.snapshotId,
        colIdQueryState
      ).then((result) => {
        return setNodes(result.data as SnapshotNode[]);
      });
    }
  }, [colIdQueryState, props.refresherStatus]);

  return (
    <div>
      {nodes.length !== 0 && colIdQueryState !== "" && (
        <>
          <H2 className="mb-4">{collectionNode?.name}</H2>
          <Table>
            ...
            <tbody>
              {directChildNodes.map((node) => (
                <WikiFieldRow
                  key={node.id}
                  node={node}
                  nodes={nodes}
                  setCollectionQuery={setColIdQueryState}
                  setNodeQuery={setNodeIdQueryState}
                />
              ))}
            </tbody>
          </Table>
        </>
      )}
```

```
    </div>
  );
};

export default WikiWorkSpace;
```

*WikiWorkSpace.tsx*

De useEffect hook is opnieuw verantwoordelijk voor het versturen van het request naar de backend. Telkens wanneer de gebruiker een andere model id in de query parameter duwt of de gebruiker een andere wijziging doorvoert (later meer) wordt de useEffect hook opnieuw uitgevoerd.

In de backend komt deze request opnieuw aan in de Snapshot Controller. De verwerking van dat request in de backend verloopt haast volledig analoog: van controller naar service, naar repository. Vandaar dat er meteen wordt ingezoomd op de repository.

```
3 references | AzureAD\ArnoVandenBergh, 31 days ago | 1 author, 1 change
public async Task<IEnumerable<NodeEntity>> GetAllSnapshotNodesOfCollectionAsync(string snapshotId, int leftId, int rightId)
{
    var builder = Builders<NodeEntity>.Filter;
    var all = await DbSet.FindAsync(builder.Eq(s => s.SnapshotId, snapshotId)
                                  & builder.Gte(s => s.Left, leftId)
                                  & builder.Lte(s => s.Right, rightId));
    return all.ToList();
}
```

*NodeRepository.cs*

Zoals besproken in het deeltje “Model Tree Structure” wordt de relatie tussen de verschillende nodes in de database bijgehouden in de vorm van een index pad. Zo heeft elke node een linker en rechter index. Op die manier kunnen de velden nodes van de gekozen collectie node makkelijk opgehaald worden en teruggegeven aan de frontend.

De frontend krijgt deze data dan binnen maar moet die nog mooi formatteren zodat de gebruiker hier iets van kan maken. Dat is de verantwoordelijkheid van het WikiFieldRow component.

```
export const WikiFieldRow = (props: FieldRowProps) => {
  let needsLink: boolean;
  let isLookup = true;
  let collectionId = "";

  switch (getLookupType(props.node, props.nodes)) {
    case LOOKUP_TYPE.COLLECTION:
      needsLink = true;
      const colLookup = props.nodes.find(
        (n) => n.left > props.node.left && n.right < props.node.right
      );
      collectionId = colLookup?.lookupCollectionId
        ? colLookup.lookupCollectionId
        : "";
      break;
```

```
    case LOOKUP_TYPE.BUSINESS_LOOKUP:
      needsLink = true;
      collectionId = props.node.lookupCollectionId;
      break;
    case LOOKUP_TYPE.MULTI_BUSINESS_LOOKUP:
      needsLink = true;
      const lookup = props.nodes.find(
        (n) => n.left > props.node.left && n.right < props.node.right
      );
      collectionId = lookup?.lookupCollectionId
        ? lookup.lookupCollectionId
        : "ERROR";
      break;
    case LOOKUP_TYPE.VALUE_LOOKUP:
      needsLink = false;
      break;
    case LOOKUP_TYPE.MULTI_VALUE_LOOKUP:
      needsLink = false;
      break;
    default:
      isLookup = false;
      needsLink = false;
  }

  const navigateHandler = (
    e: React.MouseEvent<HTMLSpanElement, MouseEvent>
  ) => {
    e.stopPropagation();
    props.setCollectionQuery(collectionId);
  };

  return (
    <Tr key={props.node.id} onClick={() =>
props.setNodeQuery(props.node.id!)}>
      <Td className="px-4">{props.node.name}</Td>
      <Td className="px-4">{props.node.caption}</Td>
      <Td className="px-4">
        {!needsLink && !isLookup && props.node.valueType}
        {!needsLink && isLookup && (
          <span className="text-green-400">{props.node.valueType}</span>
        )}
        {needsLink && (
          <span
            className="text-red-600 cursor-pointer"
            onClick={navigateHandler}
          >
            {props.node.valueType}
          </span>
        )}
```

```
      </Td>
      <Td className="px-4 break-words">{props.node.description}</Td>
    </Tr>
  );
};
```

*FieldRenderHelpers.tsx*

Dit component kunnen we in 2 delen bekijken. Eerst heb je de logica de voor elk veld gaat bekijken hoe de rendering er moet uitzien. Dat gebeurt aan de hand van de “getLookupType” methode. Belangrijk daarbij is dat afhankelijk van het lookup type de gebruiker moet kunnen navigeren naar de wiki pagina van een model waar het betreffende veld een referentie naar heeft. Onder deze logica heb je de code die mag worden geprint in de browser.

*Detail venster*

Wanneer de gebruiker op één van de velden klikt, krijgt die de detail weergave van dat veld te zien. De frontend steekt id van dit veld (nodeId) opnieuw in de url als een query parameter zodat het wiki detail venster hier mee verder kan.

In het wiki detail venster wordt dan een request gemaakt naar de backend, gebruikmakend van de nodeId die het component uit de query parameter haalt.

```
  const [nodeIdQueryState, setNodeIdQueryState] = useQueryState("nodeId", "");
  const [colIdQueryState, setColIdQueryState] = useQueryState("colId", "");

...

  useEffect(() => {
    if (nodeIdQueryState !== "") {
      SnapshotsApi.getAllSnapshotNodesOfParentFieldNode(
        snapshotId!,
        nodeIdQueryState
      ).then((result) => {
        const resultNodes = result.data as SnapshotNode[];
        setNodes(resultNodes);
        setDescriptions(
          resultNodes.map((n) => (n.description ? n.description : ""))
        );
      });
    }
  }, [nodeIdQueryState]);
```

*WikiDetailView.tsx*

De communicatie met de backend verloopt volledig hetzelfde als bij het werk venster. Beide componenten gaan in interactie met hetzelfde endpoint.

De rendering van de velden in het detail venster verloopt analoog met de velden in het werkvenster. Er zal daarom ook niet op worden ingezoomd. Wat wel nieuw is, is de mogelijkheid om beschrijvingen te kunnen gaan toevoegen:

```
...

  const editHandler = () => {
    setFormMode(FORM_MODE.EDIT);
  };

  const changeHandler = (
    arrayIndex: number,
    e: React.ChangeEvent<HTMLInputElement>
  ) => {
    setDescriptions((oldDescriptions) => {
      const newDescriptions = [...oldDescriptions];
      newDescriptions[arrayIndex] = e.target.value;
      return newDescriptions;
    });
  };

  const changeTextAreaHandler = (
    arrayIndex: number,
    e: React.ChangeEvent<HTMLTextAreaElement>
  ) => {
    setDescriptions((oldDescriptions) => {
      const newDescriptions = [...oldDescriptions];
      newDescriptions[arrayIndex] = e.target.value;
      return newDescriptions;
    });
  };

  const saveHandler = () => {
    const updateNodes = [...nodes];
    updateNodes.forEach((n, i) => {
      n.description = descriptions[i];
    });
    SnapshotsApi.updateNodes(updateNodes).then((result) => {
      setFormMode(FORM_MODE.READ);
      dispatch(forceReload());
    });
  };

...
```

*WikiDetailView.tsx*

Dit component bevat vier event handlers die verantwoordelijk zijn voor het toevoegen of bewerken van een beschrijving:

- De editHandler verandert de formulier naar de editeer modus zodat de input velden zichtbaar worden.

- De changeHandler en changeTextAreaHandler zijn dan weer verantwoordleijk voor het regristreren van de wijzigingen die de gebruiker doorvoert.
- Als de gebruiker tevreden is met de wijzigingen kan die deze opslaan en worden de wijzigingen verstuurt naar de backend. Wanneer dit succesvol is wordt het werkvenster ook opnieuw geladen zodat de beschrijving ook daar verschijnt (forceReload).

# Frontend

### Hooks

Hooks zijn een belangrijk deel van React als het gaat om functional components. Hiermee is het mogelijk om binnen je componenten gebruik te maken van verschillende features die React aanbiedt zoals state, effects, routing, …

In React is het ook mogelijk om zelf hooks te schrijven door reeds bestaande hooks te gebruiken. Deze custom hooks kan je dan net als de standaard hooks ook gebruiken in je eigen componenten en op die manier houd je de code proper doordat componenten zich enkel bezighouden met hun core.

**Custom value validation hooks**

De useInputValidatorWithState hook is in het leven geroepen om de logica binnen een formulier component te vereenvoudigen. Het doel van deze hook is om de verantwoordelijkheid voor het managen van de state en validatie van een bepaald veld op zich te nemen, zodat het component deze hook gewoon kan implementeren:

```
...

  const name = useInputValidatorWithState(
    (val) => val !== "",
    props.connection?.name
  );

...

  const formValidity =
    name.isValid &&
    provider.isValid &&
    providerConfigValidationArray.length == 0
      ? true
      : providerConfigValidationArray.reduce((pv, cv) => pv && cv);

  const submitConnectionAndValidationHandler = (
...
    let formValid = true;
    formValid = name.validate() && formValid;
...
    if (formValid) {
      const connection: Connection = {
        id: props.formMode == FORM_MODE.EDIT ? props.connection?.id :
undefined,
        name: name.inputValue,
...

  return (
...
          <InputText
            search={false}
            placeholder="Fill in a connection name"
            isValid={name.isValid}
            onChange={name.changeAndValidateHandler}
            value={name.inputValue}
            errorMessage={name.errorMessage}
          />
...
```

*ConnectionForm.tsx*

De hook neemt als input de validatie regel waaraan de waarde van het veld moet voldoen en de start waarde voor dat veld. Als output krijg je dan een object met een aantal properties waar het component dan mee aan de slag kan:

- inputValue: De huidige waarde van het veld.
- isValid: Een boolean die weergeeft of het veld voldoet aan de validatie regel.
- validate: Een methode die gaat controleren of de huidige waarde valid is. Deze wordt gebruikt vlak voor het submitten van een formulier.
- changeAndValidateHandler: Een methode die de waarde van de inputValue en isValid waarden gaat beheren.

```
const useInputValidatorWithState = (validateValue: (val: any) =>
boolean,initialValue: string = "") => {
  const [isValid, setIsValid] = useState(true);
  const [inputValue,setInputValue] = useState(initialValue);

  const changeAndValidateHandler = (event:
React.ChangeEvent<HTMLInputElement>) => {
    setInputValue(event.currentTarget.value);
    if (!isValid) {
      if (validateValue(event.currentTarget.value)) {
        setIsValid(true);
      }
    } else {
      if (!validateValue(event.currentTarget.value)) {
        setIsValid(false);
      }
    }
  };

  const validate = (): boolean => {
    if (!validateValue(inputValue)) {
      setIsValid(false);
      return false;
    }
    return true;
  };

  return {
    validate,
    changeAndValidateHandler,
    isValid,
    errorMessage,
    inputValue
  };
};

export default useInputValidatorWithState;
```

*useInputValidatorWithState.tsx*

De uitwerking van deze hook is vrij voor de hand liggend. Twee useState calls die verantwoordelijk zijn voor de waarde van een veld en de validatie ervan. Daarnaast twee event handlers die met deze staat aan de slag gaan.

Initieel pakte deze hook alle problemen binnen het connectie formulier aan, maar met dat de nood voor dynamische providers duidelijker werd en de frontend vooraf daarom niet weet met hoeveel velden die te maken gaat hebben was er de nood voor een hook die overweg kan met een variabel aantal velden: useInputValidatorForAnyObjectWithState. Deze neemt in plaats van een enkele waarde een heel object als start waarde:

```
const useInputValidatorForAnyObjectWithState = (
  validateValue: (val: any) => boolean,
  initialValue: any
) => {
  const [isValid, setIsValid] = useState<boolean[]>(
    Array(Object.values(initialValue).length).fill(true)
  );
  const [isRequired, setIsRequired] = useState<boolean[]>(
    Array(Object.values(initialValue).length).fill(true)
  );
  const [inputValues, setInputValue] = useState<string[]>(
    Object.values(initialValue)
  );
  const [inputKeys, setInputKeys] = useState<string[]>(
    Object.keys(initialValue)
  );
  const [errorMessage, setErrorMessage] = useState<string[]>(
    Array(Object.values(initialValue).length).fill("")
  );

  //Set value and check for validity when invalid
  const changeAndValidateHandler = (
    event: React.ChangeEvent<HTMLInputElement>,
    index: number
  ) => {
    setInputValue((v) => {
      const nv = [...v];
      nv[index] = event.target.value;
      return nv;
    });
    if (!isValid![index]) {
      if (validateValue(event.target.value)) {
        setIsValid((v) => {
          const nv = [...v];
          nv[index] = true;
          return nv;
        });
      }
    } else {
      if (isRequired[index] && !validateValue(event.target.value)) {
        setIsValid((v) => {
          const nv = [...v];
```

```
          nv[index] = false;
          return nv;
        });
      }
    }
    setErrorMessage((v) => {
      const nv = [...v];
       nv[index] = "";
      return nv;
    });
  };

  const validate = (index: number): boolean => {
    if (isRequired[index] && !validateValue(inputValues[index])) {
      setIsValid((v) => {
        const nv = [...v];
        nv[index] = false;
        return nv;
      });
      return false;
    }
    return true;
  };

  const validateHttpResponse = (message: string, index: number) => {
    setIsValid((v) => {
      const nv = [...v];
      nv[index] = false;
      return nv;
    });
    setErrorMessage((v) => {
      const nv = [...v];
      nv[index] = message;
      return nv;
    });
  };

  const setInputValidationState = (providerConf: any) => {
    setIsRequired(Object.values(providerConf));
    setIsValid(Array(Object.values(providerConf).length).fill(true));
    const initialVal = (Object.keys(initialValue).length ===
0)?Array(Object.values(providerConf).length).fill(""):Object.values(initialVal
ue);
    setInputValue(initialVal);
    setInputKeys(Object.keys(providerConf));
    setErrorMessage(Array(Object.values(providerConf).length).fill(""));
  };

  const setErrorMessages = (errorMessages: any) => {
```

```
    const errArray = Object.keys(errorMessages);
    errArray.forEach((errKey,i) => {
      const j = inputKeys.indexOf(errKey);
      const errVal = Object.values(errorMessages)[i] as string;
      validateHttpResponse(errVal,j)
    })
  }

  const getInputObject = () => {
    return Object.fromEntries(inputKeys.map((v,i) => [v,inputValues[i]]));
  }

  const getInputValidationArray = (): InputValidor[] => {
    return Object.values(inputValues).map((configItem, index) => {
      return {
        isValid: isValid[index],
        isRequired: isRequired[index],
        errorMessage: errorMessage[index],
        inputValue: inputValues[index],
        inputKey: inputKeys[index],
        validate: () => validate(index),
        validateHttpResponse: (message: string) =>
          validateHttpResponse(message, index),
        changeAndValidateHandler: (
          event: React.ChangeEvent<HTMLInputElement>
        ) => changeAndValidateHandler(event, index),
      };
    });
  };

  return {
    getInputValidationArray,
    setInputValidationState,
    getInputObject,
    setErrorMessages
  };
};

export default useInputValidatorForAnyObjectWithState;
```

useInputValidatorForAnyObjectWithState.*tsx*

De interne werking van deze hook ziet er toch wat anders uit dan de versie voor een enkel veld. Dat wordt al meteen duidelijk in de state binnen de hook. Zo bestaat de state uit 5 useState calls die elks een array vertegenwoordigen. Er werd gekozen voor een array omdat de frontend op voorhand niet weet hoeveel velden een provider nodig heeft om configuratie tot stand te brengen. De states zijn verantwoordelijk voor het bijhouden van de validatie, vereiste, invoer waarde, captions en error berichten.

Naast de states vind je in deze hook ook een aantal methodes die worden vrij gegeven. Deze methodes maken het mogelijk om de verschillende states te beheren, op deze manier kan het ConnectionForm component op een efficiënte manier in interactie gaan met de verschillende states van de hook.

**Custom modal hook**

Binnen Quala wordt op verschillende plekken gebruik gemaakt van modals om de gebruiker bijvoorbeeld te voorzien van extra informatie wanneer nodig of om te vragen achter een extra bevestiging. Dit was een interessante use case voor het uitschrijven van een custom hook:

```
export const useModal = () => {
  const [modalVisibility, setModalVisibility] = useState(false);
  const [submissionHandler, setSubmissionHandler] = useState<() => void>(
    () => {}
  );
  const [prompt, setPrompt] = useState<React.ReactNode>(<></>);
  const [modalState, setModalState] = useState<ModalState>(ModalState.YesNo);

  const openModalHandler = (
    subHandler: () => void,
    promptP: React.ReactNode = <></>,
    modalStateP: ModalState = ModalState.YesNo
  ) => {
    //Bij het ingeven van een functie probeert react vorige waarde
    //van de state mee te geven als een variabele vandaar dat we de
    //callback nog eens extra inwikkelen zodat dit geen error geeft.
    setSubmissionHandler(() => subHandler);
    setPrompt(promptP);
    setModalState(modalStateP);
    setModalVisibility(true);
  };

  const updateModalHandler = (
    promptP: React.ReactNode
  ) => {
    setPrompt(promptP);
  }

  const closeModalHandler = () => setModalVisibility(false);

  const submissionAndCloseHandler = async () => {
    await submissionHandler();
    closeModalHandler();
  };
```

```
  const Modal = (props: ModalProps) => {
    return (
      <>
        {modalVisibility && (
          <>
            <div
              className="fixed top-0 left-0 flex justify-center items-center
w-screen h-screen z-20"
              onClick={closeModalHandler}
            >
              <div
                className="bg-white p-5 border-t-2 border-red-600 shadow-md
relative"
                onClick={(e) => {
                  e.stopPropagation();
                }}
              >
                <div
                  className="absolute top-2 right-2"
                  onClick={closeModalHandler}
                >
                  <Dismiss20Regular className="text-red-600 cursor-pointer
hover:rotate-90 transition-transform" />
                </div>
                {prompt}
                {props.children}
                {modalState === ModalState.YesNo && (
                  <>
                    <Button
                      className="!w-24 mt-4 mr-4"
                      onClick={submissionAndCloseHandler}
                    >
                      Yes
                    </Button>
                    <Button className="!w-24 mt-4"
onClick={closeModalHandler}>
                      No
                    </Button>
                  </>
                )}
                {modalState === ModalState.Ok && (
                  <>
                    <Button className="!w-24 mt-4"
onClick={submissionAndCloseHandler}>
                      Ok
                    </Button>
                  </>
                )}
              </div>
```

```
            </div>
            <div className="fixed w-screen h-screen opacity-25 bg-gray-500
top-0 left-0 z-10"></div>
          </>
        )}
      </>
    );
  };

  return {
    Modal,
    openModalHandler,
    updateModalHandler
  };
};

interface ModalProps {
  children: React.ReactNode;
}
```

useModal.*tsx*

De useModal hook heeft geen invoer parameters en levert een object aan met drie properties:

- Het Modal component: Deze plaats je in de root van de pagina waar je de Modal wil gaan gebruiken.
- De openModalHandler methode: Deze methode neemt als parameters een arrow functie die moet worden uitgevoerd wanneer de gebruiker het bericht in de modal bevestigt, de prompt die moet worden getoont en de modus van de Modal. Wanneer deze methode wordt opgeroepen wordt de modal geopend.
- De updateModalHandler methode: Deze kan  gebruikt worden om de prompt te wijzigen wanneer de modal al geopent is.

```
...

const {Modal,openModalHandler,updateModalHandler} = useModal();

  return (
    <>
    <Modal>
    </Modal>
...
```

Main.*tsx*

```
  const prompt = (id: number, snapshots: Snapshot[]) => (
    <div className="w-80">
      <H3>Connections</H3>
      <table className="w-full">
        <tbody>
          {connections.map((c, i) => (
            <Tr
              key={c.id}
              className={i === id ? "text-red-600" : ""}
              onClick={() => {
                setCurrentConnectionIndex(i);
                props.updateModalHandler!(prompt(i, snapshots));
              }}
            >
              <Td>
              {c.name}
              </Td>
            </Tr>
          ))}
        </tbody>
...
      </table>
    </div>
  );

  return (
    <Button
      Icon={Connected20Regular}
      onClick={() =>
        props.openModalHandler!(
          () => {},
          prompt(currentConnectionIndex, snapshots),
          ModalState.Ok
        )
      }
    >
      Snapshot
    </Button>
  );
};

export default SnapshotNavigation;
```

SnapshotNavigation.tsx

In bovenstaande code voorbeelden wordt getoond hoe de useModal hook gebruikt kan worden.

### State

State wordt binnen React gebruikt om de DOM te controleren. Wanneer de state wijzigt wordt het component opnieuw gerenderd binnen de DOM. Het gebruiken van state binnen React heeft zijn beperkingen. Zo kan state enkel gebruikt worden binnen het component waarin het wordt gedefinieerd. Natuurlijk kan je het van component tot component doorgeven, maar dit is niet altijd praktisch. Daarnaast wordt de state ook gereset telkens wanneer de pagina wordt herladen.

Omdat dit twee beperkingen zijn die het coderen en de user experience bemoeilijken is er voor gekozen om deze aan te pakken.

**Query state**

Standaard wanneer een gebruiker zijn pagina herlaad in een React applicatie wordt de state gereset naar zijn start waarde. Een mogelijkheid om hier mee om te gaan is door de state bij te houden in query parameters. Op die manier wanneer de gebruiker de pagina herlaad kan de state opnieuw worden uitgelezen uit de query parameters.

De React Router package heeft een methode useSearchParams waarmee je deze kunt uitlezen. Omdat deze hook niet even intuïtief is als de gekende useState hook is er voor gekozen om daar een hook omheen te bouwen, zodat deze gelijkaardig werkt als de useState hook:

```tsx
import { useSearchParams } from "react-router-dom";

export function useQueryState(
  searchParamName: string,
  defaultValue: string
): readonly [
  searchParamsState: string,
  setSearchParamsState: (newState: string) => void
] {
  const [searchParams, setSearchParams] = useSearchParams();

  const acquiredSearchParam = searchParams.get(searchParamName);
  const searchParamsState = acquiredSearchParam ?? defaultValue;

  const setSearchParamsState = (newState: string) =>
    setSearchParams((searchParams) => {
      searchParams.set(searchParamName, newState);
      return searchParams;
    });

  return [searchParamsState, setSearchParamsState];
}
```

useQueryState.tsx

Als invoer parameters neemt deze hook de naam van de query parameter die je wil beheren en de default waarde die mag worden meegegeven. Net zoals bij de useState hook, krijg je een array terug met de waarde die je kan raadplegen en een methode om de waarde te veranderen. De code achter deze hook wijst zich zelf uit.

**Global state**

Om er voor te zorgen dat je een bepaalde state van over heel de applicatie kunt gebruiken kan je werken met een global state management library. Binnen dit project werd er daarom gebruik gemaakt van Redux Toolkit.

# Backend

### Snapshot pipeline

Het belangrijkste stuk code achter heel de applicatie is de methode die een snapshot neemt van een bepaalde database. Al de andere functionaliteiten steunen op de goede werking van deze methode.

De “CreateSnapshotAsync” methode bestaat uit een aantal stadia die elks hun eigen verantwoordelijkheid hebben:

- **Data input**: In het eerste deel van deze methode worden alle modellen vanuit de gewenste MongoDB database ingelezen zodat de rest van de pipeline hier mee aan de slag kan.

```
public async Task CreateSnapshotAsync(Dictionary<string, object> configura-
tion, string connectionId)
{
    SetProviderConfiguration(configuration);
    var appConfig = new ConfigurationBuilder().AddJsonFile("appset-
tings.json").Build();

    //GET ALL MODELS
    var client = new MongoClient(GetConnectionString());

    var database = client.GetDatabase(ModelConfiguration.Database);
    var modelCollection = database.GetCollection<BsonDocument>("_models");
    var viewCollection = database.GetCollection<BsonDocument>("_views");

    var filter = Builders<BsonDocument>.Filter.Empty;

    var models = modelCollection.Find(filter).ToList();
    var views = viewCollection.Find(filter).ToList();
```

- **Data formatting**: Binnen de Humqu database (waar de data uit wordt geïmporteerd) bestaat er tussen de verschillende modellen een boom structuur. Wat wil zeggen dat alle model collecties die één of meerdere parents (eventueel uit hogere niveau's) hebben de velden van deze parents overerven. Binnen Quala is er de keuze gemaakt dat alle child model collecties de velden van de parent collecties zelf gewoon kopiëren en dus niet overerven.
  In dit deel van de methode worden daarom de opgekuiste modellen in een formaat gestoken dat kan gebruikt worden om de velden van de parent model collecties te kopiëren naar hun child model collecties op een recursieve manier. Het idee hier is dat elk van de modellen in een shell object wordt gestoken die referentie houdt naar parent en child shell objecten. Deze shell objecten worden dan gebruikt om via recursie voor de onderliggende modellen de velden van de parent modellen naar de child modellen te kopiëren.

```
    //ADD PARENTFIELDS TO CHILDFIELDS
    var shells = Shell.CreateRootShells(models);
    shells.ForEach(shell =>
    {
        shell.AddParentFieldsToThis();
    });

    models.ForEach(model =>
    {
        Console.WriteLine(model.ToJson(new JsonWriterSettings { Indent = true
}));
        Console.WriteLine();
    });

    shells.ForEach(shell =>
    {
        shell.GetAllBsonDocs(ref models);
    });
```

- **Data filtering**: Na het formateren van de data is er ruimte om modellen die niet van toepassing zijn binnen Quala weg te filteren. Momenteel staat deze code in commentaar omdat er voor gekozen is om alle modellen te tonen.

```
   //FILTER TO ONLY GET MODELS THAT ARE N0T A PARENT

    //var parents = models.Aggregate(new HashSet<string>(), (a, b) =>
    //{
    //    if (b.Contains("parentobject"))
    //    { a.Add((string)b.GetElement("parentobject").Value); }
    //    return a;
    //}, a => a).ToList();

    //documents = documents.FindAll(d =>
!parents.Contains((string)d.GetElement("name").Value));
```

- **Data projecting**: Na een eventuele filtering is de data klaar om deze om te vormen naar de Node object waarover steeds gesproken is. Deze code is heel provider afhankelijk aangezien elke provider zijn eigen manier heeft om modellen weg te schrijven. Er is geprobeerd om de mapping naar de Nodes zo algemeen mogelijk te doen, maar omdat het gaat om een proof of concept is er hier en daar toch meer de focus gelegd op het mappen van de Humqu modellen naar de Quala nodes.

```
    //MAP MODELS TO NODES
    List<NodeEntity> nodes = new List<NodeEntity>();

    var pathIndex = 2;
    var snapshot = new SnapshotEntity()
    {
        Id = ObjectId.GenerateNewId().ToString(),
        CreatedAt = new DateTime(DateTime.Now.Ticks,
DateTimeKind.Utc).ToString("o"),
        ConnectionId = connectionId,
    };
    var snapShotId = snapshot.Id;
    models.ForEach(model =>
    {
        var collectionNodes = HumquNodeHelpers.createCollectionNode(model,
snapShotId, ref pathIndex);
        nodes.AddRange(collectionNodes);
    });

    views.ForEach(view =>
    {
        var viewNodes = HumquNodeHelpers.createViewNode(view, snapShotId, ref
pathIndex);
        nodes.AddRange(viewNodes);
    });
```

- **Data linking**: Sommige velden van de modellen kunnen een referentie houden (link of lookup) naar andere modellen. In deze fase worden deze referenties gelegd in de nodes die deze velden vertegenwoordigen. Er wordt als het ware een foreign key bijgehouden naar de betreffende modellen.

```
    //LOOKUP VERWIJZINGEN ZETTEN NAAR DE JUISTE NODE
    var collections = nodes.FindAll(node => node.NodeType ==
NodeType.Collection
                                          || node.NodeType ==
NodeType.View);
    nodes.ForEach(node =>
    {
        if (node.LookupCollection != null)
        {
            var collection = collections.Find(col =>
col.Name.Equals(node.LookupCollection));
            node.LookupCollectionId = (collection != null) ? collection.Id :
"NOT AVAILABLE";

            if (collection != null)
            {
                var idField = nodes.Find(idfield => idfield.Left >
collection.Left
                && idfield.Right < collection.Right
                && (idfield.Name != null) ?
idfield.Name.Equals(node.LookupField) : false);
                node.LookupFieldId = (idField != null) ? idField.Id : "NOT
AVAILABLE";
            }
        }
    });
```

- **Catch errors in data**: Vanuit de communicatie met de stagegever en ook tijdens het debuggen van deze methode is het duidelijk geworden dat er in de modellen van Humqu fouten kunnen zitten. De fouten die tijdens het debuggen zo naar boven zijn gekomen worden in dit deel van de methode opgevangen en mee aan de betreffende node objecten toegevoegd zodat hier later op kan worden ingezet. De bedoeling is dat er op dit deel in de toekomst nog op wordt uitgebreid.

```
    //INSERT ERRORS WHERE NECESARRY
    nodes.ForEach(node =>
    {
        var errors = new List<string>();
        var properties = typeof(NodeEntity).GetProperties();
        foreach (var property in properties)
        {
            if (property.GetValue(node) != null ?
            Regex.IsMatch(property.GetValue(node).ToString(), "NOT AVAILABLE")
: false)
            {
                errors.Add(property.Name);
            }
        }
        if (errors.Count > 0)
        {
            node.Errors = errors;
        }
    });
```

- **Data output**: In dit deel van de code worden de aangemaakt node weggeschreven naar de Quala database.

```
    //INSERT NODES INTO DATABASE
    var insertianClient = new
MongoClient(appConfig.GetValue<string>("MongoSettings:Connection"));

    var insertianDatabase = insertianClient.
    GetDatabase(appConfig.GetValue<string>("MongoSettings:DatabaseName"));
    var nodesCollection = insertianDatabase.
    GetCollection<NodeEntity>(appConfig.GetValue<string>("MongoSettings:Collec
tions:" +
    typeof(NodeEntity).Name));
    var snapshotCollection = insertianDatabase.
    GetCollection<SnapshotEntity>(appConfig.GetValue<string>("MongoSettings:Co
llections:" +
    typeof(SnapshotEntity).Name));

    await nodesCollection.InsertManyAsync(nodes);
    await snapshotCollection.InsertOneAsync(snapshot);
}
```

Humqu.cs