

Scaffold Fabric: “Het visualiseren van de configuratie”

Stepp right into the development process

The logo for 'hoppinger' is displayed on a dark grey rectangular background. The word 'hoppinger' is written in a white, lowercase, sans-serif font. To the right of the text is a solid green equilateral triangle pointing upwards.

Steven Koerts
090486
22 juni 2020

Informatie

Hogeschool Rotterdam

Studie	Informatica
Technisch docent	Jan Kroon
Email	j.kroon@hr.nl
Skills docent	Pascale Klein Hegeman
Email	p.klein.hegeman@hr.nl
Adres	Wijnhaven 107, 3011 WN Rotterdam
Website	www.hogeschoolrotterdam.nl/

Hoppinger

Bedrijfsbegeleider	Francesco Di Giacomo
Email	francesco@hoppinger.com
Functie	Systems & software architect
Opdrachtgever	Guiseppe Maggiore
Email	guiseppe@hoppinger.nl
Functie	CTO
Adres	Loydstraat 138, 3024 EA Rotterdam
Website	www.hoppinger.com

Student

Naam	Steven Koerts
Studentnummer	0904861
Email	0904861@hr.nl of steven@hoppinger.nl

Versie

Datum	22 juni 2020
Versie	1.0

Voorwoord

Voor u ligt mijn afstudeerscriptie over het visualiseren van het ontwikkelproces bij Hoppinger. Ik heb deze opdracht als zeer uitdagend ervaren. Het bouwen van goedwerkende software is altijd een uitdaging. Normaliter spreken we over software die door consumenten gebruikt gaat worden. Het bouwen van software voor softwareontwikkelaars heeft een extra dimensie, programmeurs denken meestal na over hoe de software gebouwd is. Als tools worden gebruikt voor het bouwen van software dan zijn er hoge verwachtingen voor het eindproduct.

Ik heb de afgelopen maanden een visuele editor gebouwd voor een interne tool bij Hoppinger genaamd de Scaffold, ik heb de editor de naam 'Scaffolder Fabric' gegeven.

Ik wil graag Hoppinger bedanken voor het geven van de kans en de vrijheid om aan deze opdracht te werken. Niet veel bedrijven bouwen hun eigen tools om hun producten en websites mee te bouwen en zijn hiermee afhankelijk van software van derden. Het unieke aan Hoppinger is dat ze veel tijd en geld investeren in het ontwikkelen van eigen frameworks en tooling die helpen bij het ontwikkelingsproces.

Verder wil ik Giuseppe Maggiore bedanken voor zijn tips en visie op het eindproduct in de twee wekelijkse intervisie gesprekken en Francesco Di Giacomo voor zijn feedback en supervisie op code niveau op het eindproduct.

Tot slot, ik heb de afgelopen maanden met plezier gewerkt aan Scaffold Fabric en heb vele technieken toe kunnen passen die ik in de afgelopen vier jaar bij de opleiding informatica geleerd heb.

Steven Koerts, Schiedam

Samenvatting

Dit onderzoek is uitgevoerd in opdracht van Hoppinger. Hoppinger is een full service web development agency, die zich voornamelijk focust op dotnet(.NET) applicaties en React. Ze werken voornamelijk met eigen gebouwde frameworks en tools die het ontwikkelproces ondersteunen. Zo hebben ze een eigen Scaffolder gebouwd, waarmee een webapplicatie kan worden opgezet. De output van de Scaffolder is een dotnet backend en een React frontend. De Scaffolder wordt niet alleen gebruikt om een applicatie snel te kunnen op zetten, maar ook om een applicatie te onderhouden. Zo helpt de Scaffolder om nieuwe functionaliteiten toe te voegen. Met de Scaffolder worden entiteiten, database relaties en permissies gegenereerd. De huidige manier waarmee de Scaffolder wordt aangestuurd is een JSON-bestand, dit bestand bevat alle specificaties van een applicatie. Intern wordt het configuratiebestand de Spec genoemd.

Het probleem wat Hoppinger heeft met de huidige situatie is dat specificaties al gauw duizenden regels code bevatten en behoorlijk complex kunnen worden. Ook is het zo dat JSON geschreven wordt in plain tekst, dus geen foutmeldingen bevat. De programmeur komt er pas achter dat hij een fout heeft gemaakt als hij de Scaffolder uitvoert, of erger nog wanneer hij al bezig is met bouwen. Het is ook moeilijk te bepalen of dit wordt veroorzaakt door een typefout, logicafout of een fout in de Scaffolder. Ten slotte is het voor nieuwe programmeurs een steile leer curve om inzicht te krijgen over welke functionaliteiten de Scaffolder beschikt.

Als oplossing om het ontwikkelproces met de Scaffolder te verbeteren zal er een userinterface gebouwd worden. Aan de interface zijn de volgende eisen gekoppeld:

- De interface zal informatie verschaffen over alle beschikbare opties van de Scaffolder.
- In de interface is het mogelijk om entiteiten te kunnen creëren.
- Entiteiten kunnen via de interface verbonden worden door middel van relaties.
- Permissies en permissiefilters zullen visueel gemaakt worden.
- Bestaande applicaties moeten makkelijk te importeren zijn in de user interface.

De volgende onderzoeksvraag zal worden beantwoord:

“Hoe kan een visualisatie van een applicatie specificatie ingezet worden om het ontwikkelproces te verbeteren?”

Hiervoor is een studie gedaan naar de geavanceerde types van Typescript en canvas applicaties. In het praktijkonderzoek is aangetoond dat een userinterface het ontwikkelproces kan verbeteren, de developer heeft op deze manier beter inzicht hoe bestaande applicaties werken. Een developer hoeft dan niet meer duizenden regels aan specificatiebestanden door te scrollen om te zien hoe de interne structuur van een applicatie eruit ziet. Ook alle opties en functionaliteiten zitten in de tool verwerkt, op deze manier zal de documentatie minder geraadpleegd hoeven te worden en is het leren werken met de Scaffolder eenvoudiger. Tenslotte is het opstarten van een nieuwe applicatie ook eenvoudiger, de developer heeft in een paar muisklikken al meer dan honderd regels aan specificatiecode. Het tekenen van de specificatie is ook een logische stap, omdat rekening is gehouden met een standaard van het ER-model. Een veel gebruikt model voor het ontwerpen van een database, iets wat programmeurs al gewend waren om te doen. Op deze manier komen de ontwerpfase en de opzetfase van een applicatie samen en kan meer tijd worden gependend aan de ontwikkelfase.

De gebouwde tekentool is geen vervanging, maar een verbetering en aanvulling op de huidige situatie. Specificaties kunnen altijd nog via een IDE bewerkt worden en vervolgens weer in de visuele editor worden uitgelezen.

Summary

The research in this report has been executed on behalf of Hoppering. Hoppering is a full service web development agency, who focusses mainly on dotnet(.NET) applications and React. They work mainly with frameworks and tools they build their self, to improve the development process. One of such a tool is the Scaffolder, this tool can help to quickly setup a web application . The output of the Scaffolder is a dotnet backend and a React frontend. The Scaffolder is not only being used to quickly setup an application, but also to maintain an application. The Scaffolder will help to add new functionality. With the Scaffolder you can generate entities, database relations and permissions. Right now the Scaffolder receives as input a JSON-file, this file contains all specifications of an application. Internally this configuration file is being called the Spec.

The problem Hoppering deals with in the current situation is that a specification can soon grow to over thousand lines of code and become pretty complex. Another thing is that JSON is being written in plain text, so doesn't contain any error messaging. The programmer will discover that he made a mistake when he runs the Scaffolder, or even worse when he is already busy with working on the application. It is also difficult to determine if he made a typo, logical error or if the error is being caused by a bug in the Scaffolder. Finally, for a new programmer it is a very steep learning curve to get insights for all functionalities in the Scaffolder.

As a solution to improve the development process with the Scaffolder, there will be build a user interface. The interface will have the following requirements:

- The interface will make information available about all functionalities of the Scaffolder.
- In the interface it must be possible to generate entities.
- Entities can be connected by relations in the interface.
- Permissions and permission filters will be made visual in the interface.
- In the interface it must be possible to import existing applications.

The following research question will be answered:

"How can a visualization of an application specification being used to improve the development process?"

This has been accomplished with research on advanced types and canvas applications. Practical research has proven that the development process can be improved with a user interface, the developer gets positive insights on how existing applications work. A developer does not have to scan thousands of lines of specification files to get an idea of the internal structure from an application. Also are all the options and functionalities implemented in the tool, this way you have to look less into the documentation and is learning made easier. Finally, setting up a new application has become easier, within a few mouse clicks you have more than a hundred lines of Spec code. Drawing specifications is a logical step into the development process. Since the rules of the ER-model are being respected and developers are already used to the notation for designing a database. This way the design phase and the setup phase come together and more time can be spend to the development phase.

The drawing tool that has been build is not a replacement, but an improvement on the current situation. Specifications can also be edited with help of an IDE and then being imported into the visual editor.

Inhoud

Voorwoord	4
Samenvatting.....	5
Summary	6
Begrippenlijst.....	9
1. Inleiding	10
2. Theoretisch kader	12
2.1. Eerder onderzoek	13
2.1.1. Scriptie Barld Boot.....	13
2.2. Eigen inzichten.....	14
3. Requirements analyse	15
4. Minimale typesafety.....	16
4.1. Een statisch getypeerde automatische formulier generator	16
4.2. Methodes van de formuliergenerator.....	18
4.2.1. Select	18
4.2.2. Children	19
4.2.3. Assign.....	19
4.2.4 AssignAny	20
4.2.5. ChildrenObject.....	20
4.3. Toepassing.....	21
4.4. Deelconclusie.....	23
5. Het visualiseren van de configuratie	24
5.1. UML regels.....	25
5.1.1. ER-model standaarden	27
5.2. Lay-out algoritmes.....	29
5.3. Toepassing.....	33
5.4. Deelconclusie.....	34
6. Integratie & gebruik	35
7. Foutafhandeling	37
8. Conclusie	40
9. Aanbeveling	43
10. Eigenreflectie.....	45
11. Competenties	46
12. Nawoord.....	48
13. Literatuurlijst	49
14. Bijlagen	50

14.1.	Voorbeeld Spec.....	51
14.2.	Ontwerp applicatie	52
14.3.	Algoritme implementatie	53
14.4.	Test importeren GrandeOmega	54
14.5.	Testplan	55
14.6.	Globale planning.....	56
14.7.	Beoordelingsformulier bedrijfsbegeleider	57

Begrippenlijst

Scaffolding: Een proces in software engineering waarbij aan de hand van een bepaalde configuratie een standaard programma wordt gegenereerd om mee te beginnen. Letterlijke vertaling is steiger, een Scaffolder is je hulpmiddel die dient voor het bouwen van software.

Typescript: Een statisch getypeerde programmeertaal, ontwikkeld door Microsoft. Typescript wordt gecompileerd naar Javascript.

JSON: Afkorting van Javascript Object Notatie, een bestandsformaat die veel wordt gebruikt voor configuratiebestanden.

IDE: Integrated Development Environment. Een editor die gebruikt kan worden om code te bewerken, testen, uitvoeren of deployen.

Spec: Een interne term bij Hoppinger waarmee het JSON-specificatiebestand wordt bedoeld. In de Spec staat alle informatie met betrekking tot de applicatie.

Permissie: Een toestemming om aan te geven welke entiteit een andere entiteit, relatie of attribuut mag wijzigen, aanmaken, toevoegen, inzien of verwijderen. Voor elk van deze acties moet apart worden aangegeven wie dit mag doen.

Permissiefilters: Een door Hoppinger bedacht systeem om de permissies uit te breiden, met een permissiefilter kan worden aangegeven dat een bepaalde entiteit alleen een andere entiteit mag bewerken als er een bepaalde relatie bestaat. Net als bij de permissies bestaan er permissiefilters voor het aanmaken, inzien, toevoegen, bewerken of verwijderen.

ERD: Entiteiten Relatie diagram. Dit wordt gebruikt om een schema te maken van een relationele database. Om volgens een bepaalde standaard de database structuur te kunnen modelleren. Wordt ook wel ER-model genoemd.

Entiteit: Hiermee wordt bedoeld een object in de database, dit kan een persoon, rol, proces, of een procesonderdeel zijn in een database. In deze scriptie wordt dit ook wel model genoemd.

1. Inleiding

Bij het maken van software komt heel veel kijken, vooral bij het opzetten van een nieuw project. Er moet een database model gemaakt worden, migraties worden uitgevoerd, keuze voor technieken, rechten van verschillende gebruikers etc. Dit is allemaal vrij herhaaldelijk werk. Hoppering heeft dit proces voor een groot deel geautomatiseerd, maar een automatisch proces moet ook worden onderhouden.

Voor het bouwen aan software worden veel andere tools en frameworks gebruikt. Al die tools zijn ook weer gebouwd door andere programmeurs voor programmeurs om een bepaald probleem op te lossen en software ontwikkeling makkelijker te maken.

Aanleiding

Hoppering heeft een automatische code generator genaamd de Scaffolder. Deze tool maakt backends, front-end views en permissies voor webapplicaties. De Scaffolder wordt gebruikt om een nieuw project op te starten en om features toe te voegen aan bestaande projecten. Dit wordt gedaan aan de hand van een configuratiebestand, geschreven in JSON, intern de Spec genoemd. Dit is erg foutgevoelig omdat een JSON-bestand gewoon platte tekst is zonder foutmeldingen, je komt hier pas achter als je de Scaffolder uitvoert. Zoals met veel codegeneratoren is het zo dat die alleen aan het begin van het ontwikkelingsproces worden gebruikt. Het bijzondere aan de Scaffolder is dat deze ook wordt toegepast om projecten te onderhouden en functionaliteiten toe te voegen.

Doel

Wat Hoppering wil is een visualisatietool en visuele editor om de Spec te bewerken, om relaties, entiteiten, permissies en fouten in de Spec inzichtelijk te maken. Het belangrijkste van de tool is dat het onderhoudbaar en schaalbaar gebouwd wordt. Verder is dit een mooi project om mijn in de afgelopen jaren opgedane kennis over algoritmes, datastructuren en software engineering in de praktijk toe te passen.

Het doel van de tool is dat het een vervanging wordt van de JSON-spec. Het moet qua gebruik vergelijkbaar zijn met een bestaande IDE. Problemen die deze tool moet gaan oplossen zijn:

- Betere documentatie, meer duidelijkheid over welke functionaliteiten de Scaffolder bezit.
- Visueel inzicht in eerder gemaakte projecten, zodat een nieuwe programmeur niet eerst duizenden regels aan specificatiebestanden moet analyseren.
- Beter inzicht in wat wel en niet kan met de Scaffolder.

Hoofdvraag

“Hoe kan een visualisatie van een applicatie specificatie ingezet worden om het ontwikkelproces te verbeteren?”

Deelvragen:

1. Hoe maak je de tool zo dat de gebruiker niet hoeft na te denken over het bijhouden van documentatie?
2. Hoe integreer je de tool met de huidige Scaffolder?
3. Hoe ga je de huidige Spec verwerken zonder dat er informatie verloren gaat?
4. Wat is de prioritering voor het implementeren van alle functionaliteiten van de Scaffolder in de editor?
5. Wat is er al gedaan aan eerder onderzoek, bestaan er al visualisatietools?

De deelvragen zijn er om beter te specificeren wat wordt gezien als het verbeteren van het ontwikkelproces. Het niet meer hoeven nadenken over documentatie of het beter beschikbaar maken van alle functionaliteiten van de Scaffolder is een verbetering op het ontwikkelproces. Bovendien is het altijd belangrijk om te kijken naar wat er al bestaat aan tools voordat er iets nieuws gebouwd gaat worden.

Werkwijze

Het onderzoek zal voornamelijk bestaan uit een literatuurstudie naar de geavanceerde types van Typescript en hoe dit ingezet kan worden om een typesafety binnen de applicatie te realiseren. Het praktijk onderzoek zal voornamelijk gaan over het tekenen van specificaties, met behulp van wiskundige principes en algoritmes. Verder zal ook aandacht worden besteedt aan de gebruiksvriendelijkheid, aangezien er ook een interface gekoppeld wordt aan dit project. Tot slot zal er ook onderzoek gedaan worden naar bestaande tools die kunnen worden ingezet om het doel te bereiken. Zoals bestaande tekenprogramma's en code editors.

Er zal worden gewerkt aan de hand van de volgende stappen:

- Eerste is het analyseren van alle verschillende functionaliteiten van de Scaffolder.
- Vervolgens zal een eerste formulier structuur gemaakt worden om de Spec te kunnen bewerken, op een dynamische manier.
- Daarna zal ik me gaan verdiepen in het visualiseren van alle entiteiten, relaties, permissies en permissie filters. Door middel van tekenprogramma's.
- Vervolgens zal nagedacht worden over fout afhandeling, om ervoor te zorgen dat fouten al in een vroeg stadium van het ontwikkelproces opgespoord kunnen worden.
- Tenslotte zal gewerkt worden aan het gebruik van de tool, hoe zorg je ervoor dat andere programmeurs Scaffolder Fabric gaan gebruiken.

Randvoorwaarden

Minimale eisen voor de tool zijn de creatie en visualisatie van entiteiten, met hun relaties, permissies en permissiefilters. Deze afbakening is belangrijk om het project te kunnen starten, omdat de Scaffolder een vrij groot project is. Daarom spreken we van een minimale implementatie en extra features. Als alles gaat volgens plan dan kunnen er altijd extra features bij gebouwd worden.

Leeswijzer

Gedurende het project heeft de tool de naam Scaffolder Fabric gekregen, omdat deze naam pas later bedacht is wordt op veel plekken in dit document nog de naam tool of tekentool gebruikt.

Deze scriptie is als volgt opgebouwd, voor de inleiding staat een samenvatting en een begrippenlijst. Omdat de naam Scaffolder Fabric pas in een later stadium is bedacht, zal op sommige plekken in dit document nog de termen tool of tekentool worden gebruikt.

In hoofdstuk 2 staat kort het resultaat van een eerder onderzoek beschreven en mijn eigen inzichten, wat neem ik mee van het eerdere onderzoek? (Deelvraag 5)

In hoofdstuk 3 begint de eerste iteratie van het bouwproces, hoe kan een automatische statisch getypeerde formulier generator helpen bij de onder houdbaarheid en schaalbaarheid van de applicatie? (Deelvraag 3)

Hoofdstuk 4 gaat over het visualiseren van de Spec (Deelvraag 3), in hoofdstuk 5 wordt dieper ingegaan op foutopsporingstechnieken (Deelvraag 1 en 4). In hoofdstuk 6 zal het gebruik worden bestudeerd (Deelvraag 2). Daarna volgt de conclusie en aanbeveling, afsluitend met een eigenreflectie en de behaalde competenties.

2. Theoretisch kader

In dit hoofdstuk worden de projecttermen verder uitgelegd en wat een Scaffolder precies is. Scaffolder komt van het Engelse woord Scaffolding, wat steiger betekent. Een Scaffolder is een algemene naam voor een tool dat aan de hand van een bepaalde invoer of configuratie code genereert en dus een project in de steigers zet. Zodat de programmeur meteen kan beginnen met bouwen.

Dit bespaart in het ontwikkelingsproces veel tijd en geld omdat herhaaldelijke handelingen door de Scaffolder worden uitgevoerd. De configuratie van de Scaffolder van Hoppinger wordt bijgehouden in een JSON bestand, voor grote projecten kan zo een bestand al gauw duizenden regels code bevatten en met behoorlijk complexe logica.

Voor nieuwe programmeurs is er een redelijk steile leer curve voor het gebruik van de Scaffolder, de documentatie wordt niet altijd even goed bijgehouden. Dus wat de tool sowieso moet hebben is goede inzichten in alle opties en functionaliteiten, plus dat er makkelijk nieuwe functies toegevoegd kunnen worden.

Wat onder andere mogelijk is in de Scaffolder is het aanmaken van entiteiten, relaties en permissies. Permissies geven aan wie wat mag bewerken, verwijderen, aanmaken of inzien. Dit wordt in de configuratie voor elke actie apart gespecificeerd.

Stel we hebben de entiteiten User en Admin.

- Een Admin kan een User bewerken en verwijderen.
- Iedereen kan een User inzien en aanmaken.

In de Spec ziet de permissie voor een User er dan als volgt uit:

```
permissions: { create: ['*'], view: ['*'], edit: ['User', 'Admin'], delete: ['User', 'Admin'] },
```

Nu is het zo dat in het systeem elke User elke User kan bewerken en verwijderen, maar het is realistisch om het zo te bouwen dat alleen een ingelogde gebruiker zijn eigen account mag verwijderen en bewerken. Daarvoor heeft Hoppinger de permissiefilters bedacht en dit ziet er als volgt uit:

```
permission_filters: { delete: [['User']], edit: [['User']] }
```

Dit is een simpel voorbeeld van permissiefilters, maar het concept is nog krachtiger. Je kan als het ware een pad creëren aan validatiepunten waar een entiteit langs moet om een actie te mogen uitvoeren. Bijvoorbeeld: Een gebruiker mag alleen een BlogPost bewerken als hij ingelogd is, hij zelf de Author is en het van zijn Redactie komt.

Zo een permissiefilter ziet er als volgt uit:

```
Permission_filters: {edit: [['Author', 'Author_Redactie', 'Redactie', 'Redactie_BlogPost', 'BlogPost']]}
```

Dit is slechts een klein stuk van de configuratie die heel snel, heel complex kan worden en die ook heel veel herhaald wordt. Een mooie oplossing hiervoor zou zijn dat de programmeur het pad langs de verschillende entiteiten kan tekenen in de interface.

Tenslotte kent de Spec drie niveaus waarop eigenschappen kunnen worden ingevuld:

1. Root-niveau: Hier staan alle globale eigenschappen die gelden voor de hele applicatie.
2. Model-niveau: Hier staan alle eigenschappen per model, zoals can-login en abstract.
3. Relatie-niveau: Hier staan alle eigenschappen per relatie zoals de source, target en het type relatie (one-to-many, many-many, one-to-one).

2.1. Eerder onderzoek

Er is binnen Hoppinger al een eerder onderzoek gedaan om het ontwikkelproces te verbeteren van de Scaffolder. Omdat een JSON-bestand geen foutmeldingen heeft en de programmeur daar dan pas achter komt op het moment dat hij de Scaffolder uitvoert, of pas als hij al aan het bouwen is, zo kwam het idee om een statisch getypeerde configuratie te maken voor de Scaffolder. Op deze manier krijgt de developer dus al in een vroeg stadium foutmeldingen te zien.

2.1.1. Scriptie Barld Boot

Barld Boot is afgestudeerd bij Hoppinger in 2019 [1], met het project om een statisch getypeerd framework te bouwen voor de Scaffolder. Het doel was om het ontwikkelproces te verbeteren door middel van een statisch getypeerde configuratie. Hij heeft hiervoor een case studie gedaan naar de geavanceerde type definities van Typescript. Met behulp van deze definities kan je dus eigen types aanmaken afhankelijk van gebruikersinvoer en meerdere variabelen. Er bestaat ook iets als recursieve types om bijvoorbeeld een unieke lijst aan waarden af te dwingen, of conditionele types om de type variabele te maken gebaseerd op bepaalde invoer.

Het gebouwde framework heeft de volgende voordelen:

- Het vinden van de juiste optie die je nodig hebt.
- Weten wat geldige waarden zijn.
- Weten wanneer je opties wel en niet mag inschakelen.
- Alle opties hebben het juiste type.
- Opties kunnen een beperkte set aan strings bevatten.
- Afdwingen unieke waarden.
- Afdwingen juiste volgorde van invoer.
- Eerdere invoer gebruiken als mogelijke invoer voor andere opties.
- Compleet nieuwe datatypes genereren op basis van invoer.

Het framework is echter niet in gebruik genomen. Niet omdat de logica niet klopte, maar om de volgende redenen:

- De performance was erg traag door de vele recursieve types.
- Het is een extra onderhoud last aan de Scaffolder.
- Het framework heeft echt de grenzen van Typescript opgezocht en de compiler wordt erg traag.
- Het is niet mogelijk een bepaald invoer formaat af te dwingen, zo mag een model naam niet met een cijfer beginnen.

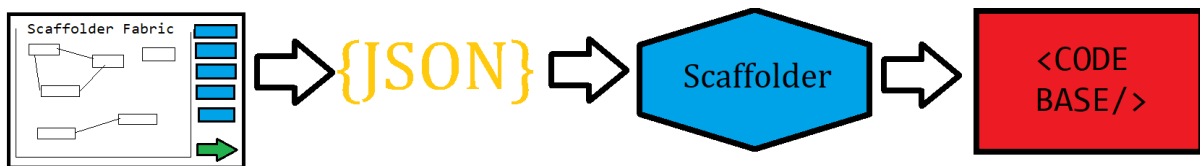
Dat de tool niet in gebruik genomen is wil niet zeggen dat het slechte code is, het is erg waardevol om de grenzen van een programmeertaal op te zoeken. Het is uiteraard handig om te weten wat wel en niet kan in een taal, zo zal ik van het geavanceerde type systeem van Typescript ook veel functionaliteiten gebruiken in mijn project.

2.2. Eigen inzichten

Na het lezen van de scriptie van Barld Boot ben ik zelf al tot een aantal inzichten gekomen, waarmee ik rekening zal gaan houden tijdens het bouwen van de visuele editor. Eén van de belangrijkste eisen is onderhoudbare software, als er een nieuwe functie aan de Scaffolder wordt toegevoegd moet de tool ook mee aanpassen of een duidelijke melding geven waar iets aangepast moet worden. Het wegnemen van de extra onderhoudslast. Als het gaat om performance en type safety zal ik niet gaan voor het bouwen van een volledig typesafe framework, aangezien we te maken hebben met een gebruikersinterface is dit ook een goede manier om invoer af te dwingen. Het gebruik van Typescript maakt het al typesafe.

Verder neem ik de belangrijkste voordelen van het eerder gebouwde framework mee, omdat daar al veel regels in staan over verplichte waardes, invoerformaat en invoervolgorde. Dus aan die functionaliteiten zal mijn tool ook moeten voldoen, voor een volwaardig stuk software dat in productie genomen kan worden.

In onderstaande afbeelding staat de huidige flow van de Scaffolder en waar mijn project komt te staan. Voor de eerste pijl staat het prototype wat gebouwd gaat worden en daar achter is de huidige flow van de Scaffolder weergegeven. De tool die gebouwd gaat worden genereert een Spec in JSON formaat, de Spec wordt doorgegeven aan de Scaffolder en de output is een code base waarmee de programmeur aan de slag kan gaan.



Figuur 1: Flow van de Scaffolder

Zoals in de afbeelding te zien is werk voornamelijk in een eigen project, met de logica van de Scaffolder zelf heb ik niet heel veel te maken. Het belangrijkste is dat er een valide JSON bestand wordt gegenereerd die in de Scaffolder kan worden uitgevoerd. In bijlage 2 wordt het ontwerp van de applicatie verder toegelicht.

Belangrijkste inzichten

- Volgorde van invoer is belangrijk.
- Voorafgaande invoer bepaalt de invoer van andere invoer velden.
- Er wordt een duidelijk onderscheidt gemaakt in de Scaffolder tussen optionele en verplichte invoer.
- Wegnemen van de onderhoudslast, als iemand een nieuwe functie of feature aan de Scaffolder toevoegt moet dit ook automatisch of eenvoudig aan te passen zijn in de tool.
- Aan zoveel mogelijk functionaliteiten van de Scaffolder voldoen.

3. Requirements analyse

Voor de eisen van het te bouwen prototype worden de eisen van mijn voorganger Barld Boot in acht genomen en aangevuld met eigen eisen.

Het belangrijkste is het vinden van de juiste optie in de Scaffolder die de gebruiker nodig heeft, zodat de gebruiker dit niet meer in de documentatie hoeft op te zoeken. Alleen de juiste datatype accepteren, de Scaffolder checkt ook al op types, dus het is een vereiste om binnen de visuele editor ook op datatypes te controleren. Weergeven wanneer een optie wel of niet ingeschakeld mag worden, zo kan inheritance alleen worden toegepast met abstracte entiteiten en permissies alleen op entiteiten die kunnen inloggen. Voor een volledige lijst met bestaande eisen zie paragraaf 2.1.

Vervolgens worden er nog eisen toegevoegd aan het prototype. We spreken hier van een minimaal op te leveren product. De eisen worden ingedeeld in must have's en nice to have's.

Must have

- De editor moet hetzelfde aanvoelen als een IDE om de drempel naar het gebruik zo laag mogelijk te maken.
- Voor het visualiseren van de Spec moeten minimaal entiteiten en relaties worden weergegeven.
- De statische eigenschappen van de Spec, op root niveau, moeten gemakkelijk aan- of uitgezet kunnen worden.
- Het diagram moet een voor het menselijk oog aantrekkelijke en leesbare weergave zijn.
- Het importeren van bestaande specificaties, zodat developers verder kunnen gaan met hun werk dat ze eerder gedaan hebben.

Nice to have

- Het visualiseren van de permissies, zonder dat het diagram daardoor onoverzichtelijk wordt.
- Het visualiseren van het pad van de permissiefilters, zonder dat het diagram daardoor onoverzichtelijk wordt. Hiervoor geldt dat die dit geen officieel onderdeel is van een ERD, dus zal hier een eigen notatie voor moeten worden ontworpen.
- Het tekenen van deel Spec's in het diagram, om ook de relaties met de hoofd Spec inzichtelijk te maken. Op zo een manier dat ook onderscheid wordt gemaakt in wat een hoofd Spec is en wat een deel Spec is.
- Het opslaan van de Spec in lokale opslag zodat de developer op een later moment weer verder kan werken en dat er geen gegevens verloren gaan.
- Het downloaden van de Spec vanuit de editor zodat een developer zijn werk kan testen in de Scaffolder.
- Het uitvoeren van de Scaffolder vanuit de editor, in plaats vanaf de command line.

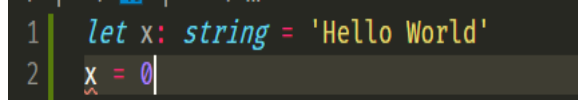
Aan deze eisen zal gedurende de stageperiode gewerkt worden, niet in exact de volgorde waarin ze nu staan. De must have requirements zijn het belangrijkste omdat dit de minimale eisen zijn voor het succes en gebruik van het prototype.

4. Minimale typesafety

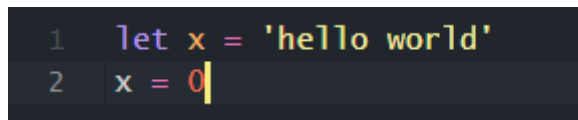
In dit hoofdstuk beschrijf ik hoe ik het project begonnen ben, met een statisch getypeerde automatische formulier generator. Voor het project kreeg ik twee oplossingen voorgeschoteld, één was een uitgebreid formulier in Typescript en React en twee was een grafische tekentool, waarbij de entiteiten en hun relaties op het scherm getekend kunnen worden. Ik ben dus begonnen met het formulier.

Voor de editor die gebouwd gaat worden is een minimale typesafety vereist. Dit gebeurt grotendeels al automatisch door het gebruik van Typescript in plaats van Javascript.

Zo geeft figuur 2 in Typescript wel een type error en figuur 3 in Javascript niet.



Figuur 2: Type error in Typescript



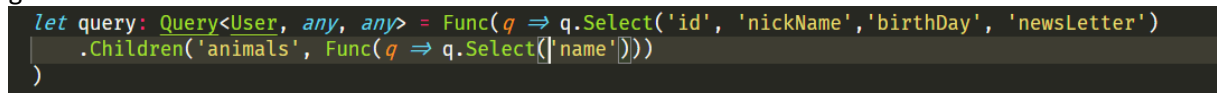
Figuur 3: Geen error in Javascript

Door gebruik te maken van het type systeem van Typescript krijg je een groot deel van het veilig programmeren er gratis bij. [2]

4.1. Een statisch getypeerde automatische formulier generator

Wat meteen al opvalt is hoe groot het object is dat de Spec representeert, om hier een formulier van te maken is dus veel type werk voor nodig. Tenzij je een automatische formulierbouwer gebruikt. Niet alleen is een formulier in HTML schrijven erg saai en herhaaldelijk werk, het is ook niet onderhoudbaar en foutgevoelig.

Bij Hoppinger gebruiken ze veel frameworks met een LINQ-achtige notatie, dit gebruiken ze om entiteiten en attributen te selecteren en om te filteren. Deze manier van selecteren lijkt erg veel op SQL. Dus als eerste ontwerp voor de formulier generator ben ik naar onderstaande structuur toe gaan werken.



Figuur 4: Formbuilder query

De query in figuur 4 selecteert van 'User' de 'nickName', 'gender' en 'birthDay' en heeft meerdere 'animals' en selecteert daarvan de 'name'.

Dit is equivalent aan de volgende SQL-syntax:

```
SELECT id, nickName, birthDay, newsLetter, animals.name
FROM Users
JOIN animals ON animals.id = user.id
```


Het resultaat van de query wordt gebruikt om een formulier te renderen op het scherm, waarbij de types van de eigenschappen van 'User' overeenkomen met het type van het formulierveld. Zo worden de naam en id een string, birthDay een datum picker en newsLetter een checkbox. De Children van het formulier resulteert in een genesteld formulier in User, met alle velden van 'animals'. Het formulier ziet er als volgt uit, als in figuur 5. In figuur 6 staat het gebruikte datamodel.

Dit principe wordt in de applicatie uiteraard toegepast op het datamodel van de Spec.

Figuur 5: Resultaat formulier door query

```
type Gender = 'he' | 'she' | 'it'
type User = {
  id: string
  nickName: string
  birthDay: Date
  newsLetter: boolean
  gender: Gender
  animals: Animal[]
  colors: {
    favoriteColor: string
    leastFavoriteColor: string
  },
  todo: string[],
  permissions: Permissions,
  permission_filters: PermissionFilters
  my_numbers: number[]
}
type Animal = {
  kind: 'dog' | 'cat' | 'goldfish'
  name: string
}
```

Figuur 6: Datamodel user

De formulier generator bestaat uit twee componenten, één is een datacontainer die geselecteerde velden opslaat en de andere een query builder die de eigenschappen van de data selecteert. Vervolgens wordt het resultaat doorgegeven aan een renderer, dit is een React component die aan de hand van het datatype het juiste formulierveld op het scherm zet.

In het formulier in figuur 5 wordt alleen gebruik gemaakt van primitieve datatypes, string, integer en boolean, maar in de Spec zitten veel ingewikkeldere datatypes verwerkt. Het wordt complexer als je te maken hebt met een array van strings, Object, array van Objecten, tweedimensionale array en union types. Hier moet dus onderscheid in gemaakt worden, vooral het datatype Array lijkt heel erg op dat van Object.

Voor de formulierbouwer heb ik de volgende eisen opgesteld, deze eisen zijn zo opgesteld dat een minimale type safety gerealiseerd kan worden:

- De query builder moet een keten van aan elkaar gekoppelde functies vormen, die in meerdere volgordes uitgevoerd kan worden, zodat de query op elk punt kan worden bijgewerkt.
- Er moeten alleen eigenschappen van het datamodel geselecteerd kunnen worden, om te voorkomen dat een eigenschap die niet bestaat geselecteerd kan worden.
- Er moeten genestelde objecten geselecteerd kunnen worden, om ook binnen in een object eigenschappen te selecteren.
- Een eerder geselecteerde eigenschap mag niet nog een keer geselecteerd kunnen worden, om te voorkomen dat iets dubbel geselecteerd wordt.
- Binnen in de builder moeten standaard waarden toegekend kunnen worden, om het formulier al een waarden mee te kunnen geven.
- Er moet de functionaliteit bestaan om het datatype in de query te veranderen van een eigenschap, om meerdere opties te kunnen selecteren. Dit wordt onder ander gebruikt voor support van dropdowns met union types.

4.2. Methodes van de formuliergenerator

In deze paragraaf worden de verschillende methodes beschreven van de formuliergenerator. Wat maakt de methodes typesafe? Waarvoor wordt het gebruikt? En hoe is de functie opgebouwd? De codesnippets komen uit een overkoepelende interface en de data is in de vorm van een paar van arrays, waar in de linkerzijde de lijst staat met entiteiten en eigenschappen waar uit geselecteerd kan worden en in de rechterkant alle entiteiten met de al geselecteerde eigenschappen. Bij elke functie waarbij iets geselecteerd wordt, worden de eigenschappen van het linker object overgeheveld naar het rechter object. Zo voldoet het aan de eis dat een eigenschap niet dubbel geselecteerd kan worden.

De initiële waarde uit het voorbeeld van paragraaf 4.1 is dus een paar van een lijst van Users links en rechts een leeg object genoemd Unit, zie figuur 7.

```
type Unit = {}
```

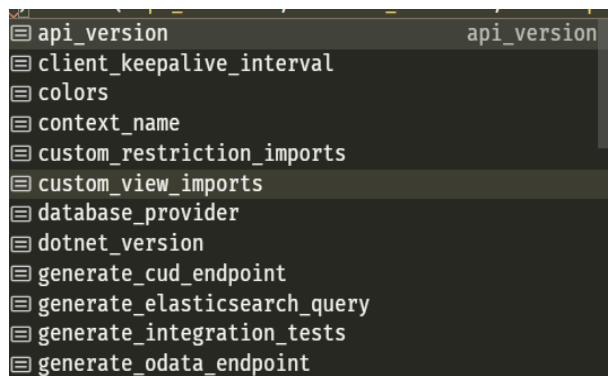
Figuur 7: Unit, het kleinst mogelijk object

4.2.1. Select

```
Select: function <K extends keyof T>( ... properties: K[]): FormSelector<Omit<T, K>, Pick<T, K> & U> {
  return FormSelector(this.data.map(
    fst => fst.map(entry => omitMany(entry, properties)),
    snd => zip(snd, this.data.First.map(entry => pickMany(entry, properties))).map(p => ({ ... p.First, ... p.Second })))
  ),
}
```

Figuur 8: Implementatie van de Select

De Select functie werkt als volgt, aan de hand van een lijst van eigenschappen worden eigenschappen uit het linker object gehaald (Omitted) en toegevoegd aan het rechter object (Picked). Door gebruik te maken van de keyof operator geef je een beperking mee aan het datatype van properties. De generieke type variabele T bevat de linkerkant van het paar en U de rechterkant van het paar, T bevat dus alle nog niet geselecteerde waarden. Dit resulteert dus in een lijst met opties van alle toegestane waarden, zoals in figuur 9. De drie puntjes voor properties worden geïnterpreteerd als een array.



```
api_version
client_keepalive_interval
colors
context_name
custom_restriction_imports
custom_view_imports
database_provider
dotnet_version
generate_cud_endpoint
generate_elasticsearch_query
generate_integration_tests
generate_odata_endpoint
```

Figuur 9: Opties van de select

```
q.Select('project_name').Select('api_version', 'dotnet_version', 'namespace')
```

Figuur 10: Voorbeeld invoer Select

4.2.2. Children

```
Children: function <K extends FilterType<T, Array<Object>>, t, TResult>(
  child: K,
  q: Func<InitialFormSelector<ArrayType<T[K]>>, FormSelector<t, TResult>>
): FormSelector<Omit<T, K>, U & { [key in K]: Array<TResult> }> {
  return FormSelector(this.data.map([
    fst => fst.map(entry => omitOne(entry, child)),
    snd => zip(snd, this.data.First.map(entry =>
      ({ [child]: q.f(InitialFormSelector(entry[child])).data.Second })))
      .reverse()
      .map(p => ({ ...p.First, ...p.Second }))) as any
  ]))
},
```

Figuur 11: Implementatie Children

Om genestelde objecten te kunnen selecteren kan ook de Select methode gebruikt worden, maar met behulp van Children is het ook mogelijk om een query uit te voeren op het genestelde object. De methode heeft als invoer een string waarvan de waarde alleen een eigenschap mag zijn die nog niet eerder geselecteerd is en die als datatype een Array van Objecten heeft. Vervolgens heeft de functie een query die wordt uitgevoerd op de lijst van genestelde objecten.

```
q.Select('project_name')
  .Children('models', Func(q => q.Select('name'))))
```

Figuur 12: Voorbeeld van Children

4.2.3. Assign

```
Assign: function <K extends keyof U>(property: K, new_value: U[K]): FormSelector<T, U> {
  return FormSelector(this.data.mapRight(snd => snd.map(e => (<U>{ ...e, [property]: new_value }))))
},
```

Figuur 13: Implementatie Assign

Met Select en Children kan een geheel formulier samengesteld worden, voor sommige gevallen is het nodig om de data van het formulier in tact te laten en niet te bewerken. Dan moeten de mutaties op de data indirect via de query gedaan worden. Dat is precies wat Assign doet, de functie neemt een eigenschap die al geselecteerd is en kent een nieuwe waarde toe met hetzelfde datatype als de eigenschap.

```
q.Select('project_name', 'dotnet_version').Assign('project_name', 'SomeProject'))
```

Figuur 14: Voorbeeld Assign

Op deze manier wordt de data van het formulier niet alleen gebruikt als structuurbeschrijving voor het formulier, maar ook als uitvoerdata van de gebruiker. Voor het gebruik van de formulier generator moet altijd de afweging worden gemaakt welke methode wordt gebruikt voor het bewerken van de data. Als de data met Assign wordt aangepast dan is het niet mogelijk dropdowns in het formulier te hebben. Dus afhankelijk van het gebruik van het formulier kan een verschillende update methode worden gebruikt.

4.2.4 AssignAny

```
AssignAny: function <K extends keyof U, A>(property: K, new_value: A): FormSelector<I, ChangeType<U, A, K>> {
  return FormSelector(this.data.mapRight(snd => snd.map(e => (<ChangeType<U, A, K>>{ ...e, [property]: new_value }))))
},
```

Figuur 15: Implementatie AssignAny

Naast dat je een andere waarde moet kunnen toekennen via de query, wil je soms ook het datatype kunnen veranderen van een eigenschap in de entiteit. Dit kun je gebruiken voor de zo geheten union types. In Typescript kun je variabelen verschillende types meegeven, dit wordt in de spec toegepast voor onder andere dotnet_version en database_provider. De waarde is een string, maar mag alleen één van de waardes in de union bevatten.

```
type DotnetVersion = "dotnet1" | "dotnet2" | "dotnet3"
type DatabaseProvider = "sqlite" | "postgresql" | "mssql"
```

Figuur 16: Voorbeeld union type

In het formulier moet dit uiteindelijk een dropdown lijstje worden waar de gebruiker verschillende opties kan selecteren. Dit doen we door met behulp van AssignAny een nieuwe waarde toe te kennen in de vorm van een speciaal object dat een array van verschillende waarden bevat.

```
q.Select('project_name', 'dotnet_version')
  .AssignAny('dotnet_version', new SelectOption('dotnet1', ['dotnet1', 'dotnet2', 'dotnet3'])))
```

Figuur 17: Voorbeeld AssignAny

Na het uitvoeren van de AssignAny is het type van dotnet_version een SelectOption object en dit wordt door de builder geïnterpreteerd als een dropdown list. Het nadeel van deze methode is dat je dus de structuur van de data wijzigt en de data in builder alleen nog maar kunt gebruiken als structuurbeschrijving voor het formulier en niet meer als de uitvoerdata van het formulier.

4.2.5. ChildrenObject

```
ChildrenObject: function <K extends FilterType<I, Object>, t2, TResult>(
  child: K,
  q: Func<InitialFormSelector<I[K]>, FormSelector<t2, TResult>>
): FormSelector<Omit<I, K>, U & { [key in K]: TResult }> {
  return FormSelector(this.data.map(
    fst => fst.map(entry => omitOne(entry, child)),
    snd => zip(snd, this.data.First.map(entry =>
      ({ [child]: q.f(InitialFormSelector([entry[child])).data.Second[0] })))
      .reverse()
      .map(p => ({ ...p.First, ...p.Second })) as any
  ))
},
```

Figuur 18: Implementatie ChildrenObject

De ChildrenObject methode lijkt qua implementatie heel erg op die van Children, het enige verschil is dat deze functie niet filtert op Arrays van Objecten, maar op alleen losse Objecten. Het kan in een datamodel namelijk ook voorkomen dat een eigenschap alleen een enkel object is. Deze functie maakt het dus mogelijk om op enkele objecten een query uit te voeren.

```
q.Select('project_name', 'dotnet_version')
  .ChildrenObject('colors', Func(q => q.Select('main_color')))))
```

Figuur 19: Voorbeeld ChildrenObject

4.3. Toepassing

Deze automatische formuliergenerator heb ik gebouwd in de eerste iteratie van het project. Met behulp van dit framework kan vrij snel een formulier gebouwd worden die alle attributen en eigenschappen van de spec bevat. Dit was een goede keuze om als eerste te implementeren, om een inzicht te krijgen in alle functionaliteiten van de Scaffolder. Dit geeft ook een goed antwoord op deelvraag 1, zo minmogelijk nadenken over de documentatie.

Zo gebruik ik onderstaande query in figuur 20, om alle verplichte velden van de Spec op het scherm te krijgen, dit eerste formulier is een mooi begin van het starten van nieuwe applicaties. Zo heb je in veel code editors een begin scherm waar de gebruiker eerst een serie gegevens moet invoeren en aan de hand van die gegevens wordt een eerste boilerplate aan code geproduceerd om mee te beginnen.

```
q.Select('project_name')
  .Select('namespace', 'context_name', 'dotnet_version', 'database_provider', 'api_version', 'url_prefix')
  .Select('custom_view_imports', 'custom_restriction_imports')
  .AssignAny('database_provider', new SelectOption('sqlite', databaseProviders))
  .AssignAny('dotnet_version', new SelectOption('dotnet1', dotnetVersions))),
```

Figuur 20: Query om alle basis informatie in een formulier te krijgen

Scaffolder Fabric

Create a new project

project_name
MyProject

namespace

context_name

dotnet_version
dotnet1

database_provider
mssql

api_version
v1

url_prefix
/

custom_view_imports

custom_restriction_imports

Get started

Figuur 21: Het resultaat van de query

Het grootste voordeel is dat als er een nieuwe functie wordt toegevoegd aan de Scaffolder, dat een volgende programmeur simpelweg een nieuwe Select kan toevoegen aan de query. Ook de volgorde van de formulervelden is hetzelfde als die van de query.

Kortom, de formuliergenerator kan gebruikt worden om een aantal statische delen van de Spec in te vullen en om een indruk te krijgen van alle functionaliteiten van de Scaffolder.

De formuliergenerator is gebouwd voor React. Het resultaat van een query is dus een React component. Om de data in een component te updaten moet dit gedaan worden door middel van een onChange event handler en de setState methode van React. Omdat je met variabelen formulervelden zit heb je dus ook voor elk formulierveld een andere functie nodig die de nieuwe waarden door geeft aan de state. Die functies kunnen ter plekke gegenereerd worden. Bij het bewerken van een formulierveld geeft die methode drie parameters mee: Een key, een nieuwe waarde en een index. De key bevat de naam van het veld dat bewerkt is en zo kan de state worden geüpdatet met de nieuwe waarden. De index wordt gebruikt om genestelde velden en lijsten te updaten, zo weet de component welk element in een lijst veranderd is.

De state van een component kan op twee manieren worden geüpdatet, één is door de standaardwaarden te overschrijven. Aan de hand van de key en de waarde die je mee krijgt van de update functie. Een andere manier is door gebruik te maken van de query. Door elke keer dat een veld bewerkt wordt een Assign functie toe te voegen aan de formulier query. De standaardwaarden blijven zo constant en worden nooit aangepast. Je kunt dan altijd makkelijk het formulier resetten. Om de huidige state van het formulier op te vragen voer je de query uit op de standaarddata en dan compileert de query de data naar de huidige state.

Zie onderstaande afbeeldingen voor twee voorbeelden over het updaten van de state van het formulier:

```
<FormMaster<Customer> id_prefix='my-form'
  defaultData={[this.state.customer]}
  query={Func(q => q.Select('FirstName', 'LastName', 'birthDay', 'gender', 'terms')
    .Select('favoriteColor')
    .Select('lists', 'todo')
    .Children('products', Func(q => q.Select('productName'))))
    .AssignAny('gender', new DropDownOptions('male', [Pair('M', 'male'), Pair('V', 'female')]))
  )}
  onChange={({key, newValue, index}) => {
    console.log(`Edited ${key}:${index} to ${newValue}`)
    this.setState({ ...this.state, customer: { ...this.state.customer, [key]: newValue } })
  }}
/>
```

Figuur 22: Het direct updaten van de data in de state

```
<FormMaster id_prefix='my-form-Immutable'
  defaultData={[this.state.immutableCustomer]}
  query={this.state.formQuery}
  onChange={({key, newValue, index}) => {
    console.log(`Edited ${key}:${index} to ${newValue}`)
    this.setState({ ...this.state, formQuery: this.state.formQuery.then(Func(q => q.Assign(key, newValue))) })
  }}
/>
```

Figuur 23: Het updaten van de data via een query

Bij beide methoden is de state van de React component de single source of truth.

Het updaten van een de state in React gebeurt via de onChange methode van een component¹. In figuur 22 wordt direct de data overschreven met de nieuwe waarde uit het formulier, in figuur 23 wordt een Assign toegevoegd aan de query, de data wordt nooit aangeraakt.

Beide methodes worden gebruikt in de tool om de Spec te updaten, voor de root eigenschappen is het handiger om direct de Spec aan te passen in de state door elke keer de huidige waarden te overschrijven.

Voor de eigenschappen van modellen en relaties wordt een query gebruikt om die te updaten. Dit omdat een Spec een lijst van meerdere modellen en relaties heeft en je niet voor elk model een standaardwaarde kunt opslaan. Op deze manier sla je één object op die alle waarden van een model of relatie bevat en door dat object aan de query mee te geven kan het juiste model worden geproduceerd.

¹ React update state: <https://reactjs.org/docs/forms.html>

Beide methode hebben voor- en nadelen, zo kun je bij de eerste methode geen “ga-terug”-functie gebruiken omdat je niet alle veranderingen bijhoudt. Wel kun je de Assign methode gebruiken om de waarden te overschrijven door bijvoorbeeld een dropdown lijst.

Bij de tweede methode is het niet mogelijk om dropdowns te gebruiken omdat bij de eerste keer dat de dropdown geopend wordt de waarden veranderd in een gewone string en vervolgens ook het type van het tekstveld veranderd. Dit resulteert in het verdwijnen van de dropdown, naar een tekstveld.

Bij het maken van een keuze welke methode gebruikt wordt om het formulier te updaten moet altijd eerst goed nagedacht worden wat het doel is van het formulier. Is het data die echt gebruikt wordt, of slechts placeholder data om later aan een ander object toe te voegen?

4.4. Deelconclusie

De automatische formuliergenerator is een goede oplossing om de Spec automatisch in te vullen, terugkomend op het doel van project. Een nieuwe developer krijgt dan meteen feedback van het systeem over alle functionaliteiten van de Scaffolder en documentatie wordt dan voor een groot deel automatisch bijgehouden. Dit voldoet overigens nog niet aan alle eisen van de visuele editor, want je kan hiermee nog geen nieuwe entiteiten, relaties en permissies toevoegen. Dit geeft antwoord op deelvraag 1, de gebruiker hoeft minder na te denken over het bij houden van documentatie.

Een nadeel van de form builder is dat hij erg traag wordt als het formulier te groot wordt, omdat het updaten van de data gebeurt bij elke letter die getypt wordt en het formulier opnieuw gebouwd wordt. Ook als er meer dan 40 modellen en relaties in de Spec staan wordt het typen traag, de builder itereert over alle eigenschappen van alle modellen. Dus bij 40 modellen met 60 eigenschappen zijn dat al 2400 iteraties en dus ook 2400 invoervelden. Het is ook niet heel gebruiksvriendelijk om meer dan 1000 invoervelden te moeten invoeren.

Dus het aantal invoervelden zal tot een minimum beperkt moeten worden, dit is een oplossing om de specificaties van de Scaffolder te genereren, maar nog geen volledige oplossing.

De form builder moet niet volledig verantwoordelijk zijn voor het genereren van de JSON-Spec, het is overigens wel een goede oplossing om alle statische velden in te vullen op root niveau van de Spec.

Wat ik tenslotte heb opgemerkt is dat de Spec in eerste instantie heel erg groot lijkt, maar als je alle optionele velden weghaalt en ook van zowel de modellen en relaties blijven er maar een aantal eigenschappen over. In figuur 20 staan dus alle verplichte waarden van de Spec, een model of entiteit heeft niet meer dan een naam, attributen (genesteld object met een naam en type) en permissies. Een relatie heeft niet meer dan een source, target en type.

Dus als je alle extra opties weghaalt blijft er een overzichtelijke minimale specificatie over die altijd nog bijgewerkt kan worden in een later stadium van het ontwikkelproces.

5. Het visualiseren van de configuratie

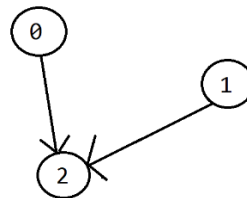
Een andere oplossing voor het verbeteren van het ontwikkelproces is het visualiseren van de Spec. Voor het invullen van alle root eigenschappen is de formuliergenerator heel geschikt. Dit kan ook toegepast worden voor het bewerken van modellen en relaties. Nu is het zo dat relaties en modellen vastgelegd kunnen worden in een ERD. Hoe mooi zou het zijn als je van een Spec een ERD kunt maken en andersom. Dan is de programmeur in staat om simpelweg de entiteiten op het scherm te tekenen en wordt aan de hand van het diagram een correcte Spec gegenereerd. De relaties kunnen dan ook tussen de entiteiten worden getekend, zoals developers gewend zijn in andere UML tekenprogramma's.

Tools die nodig zijn voor het realiseren van deze feature zijn een datastructuur om de relaties en entiteiten op te slaan. Een implementatie van de UML regels van toegestane relaties, de juiste symbolen en een front-end framework om vervolgens de modellen en relaties te kunnen tekenen.

Een geschikte datastructuur om de entiteiten en relaties op te slaan is een graph, dat is een non-lineaire datastructuur die bestaat uit nodes, soms ook vertices genoemd, en edges. De edges representeren lijnen die de nodes met elkaar verbinden. Dit is een ideale datastructuur om de entiteiten en hun relaties te representeren. Je kunt dan een transformatie functie maken van een Spec object naar een graph. Een methode om een graph uit te drukken in code is met een adjacency matrix, dit is een tweedimensionale array waarbij de lengte gelijk is aan het aantal nodes. Zie figuur 24. Uit onderstaande matrix is af te lezen dat er drie nodes zijn en twee edges. De edges worden uitgedrukt door een "1" te plaatsen op een kruispunt tussen twee nodes. De eerste edge kan als volgt geïndexeerd worden: *graph[0][2]*, en de tweede edge staat op rij 1 en kolom 2: *graph[1][2]*. Dit betekent dat node 0 is verbonden met node 2 en node 1 is verbonden met node 2. Het diagram ziet er uit zoals in figuur 25.

```
let graph = [  
  [0, 0, 1],  
  [0, 0, 1],  
  [-1, -1, 0]  
]
```

figuur 24: Adjacencymatrix



Figuur 25: Voorbeeld graph

Om de entiteiten-relatie-graph te tekenen wordt een HTML5 canvas gebruikt, in combinatie met een React applicatie. Omdat het tekenen op canvas in webapplicaties nog vrij primitief is wordt een canvas teken framework gebruikt. Voor het prototype is gebruik gemaakt van KonvaJs. Dit framework heeft ook een React variant waarbij je voorgeprogrammeerde figuren kunt toevoegen aan de DOM². Over de werking van dit framework zal niet heel veel verder worden ingegaan, omdat de focus meer moet liggen op het visualiseren van de Spec.

Je kan dus makkelijk een transformatie maken van de Spec naar een adjacency matrix, waarbij alle modellen nodes worden en alle relaties worden edges.

² Document Object Model: Het model wat alle elementen op een webpagina bevat.

Naast een adjacency matrix heb je ook nog andere manieren om een graph uit te drukken, dit kan met een adjacency list waar alle nodes worden opgeslagen als key-value pairs met als key de naam van de node en de value is een lijst aan waardes die alle edges representeren. Verder is er ook nog een object georiënteerde representatie, maar daar ben je te veel afhankelijk van object referenties. Dit is vrij traag in het opzoeken van alle nodes met behorende edges.

De reden voor de keuze van een adjacency matrix is vanwege performance en simpelheid. Het aantal nodes is altijd gelijk aan de lengte van de array. Het verwijderen, uitlezen of updaten van een edge kan gedaan met behulp van de indexen. Op het moment dat een relatie omgedraaid wordt is het kwestie van het wisselen van een 1 en 0 (Figuur 26). Dit is vooral handig als de programmeur een relatie wil omdraaien. Het nadeel van deze structuur is dat het verwijderen van een specifieke node vrij traag is (complexiteit $O(n^2)$). Omdat je dan over elke horizontale en verticale kolom moet itereren om de juiste index te verwijderen.

```
//Swap edge  
graph[0][2] = 0  
graph[2][0] = 1
```

Figuur 26: Het omdraaien van een edge, complexiteit $O(1)$

Dit hoofdstuk zal verder ingaan op de UML regels, de regels van de Scaffolder, wat wel en niet moet kunnen in de tekentool en hoe je ervoor kunt zorgen dat de lay-out van het diagram er netjes uit ziet. Hoe maak je de omzetting naar een graph zo dat er geen informatie verloren gaat (Deelvraag 3). Aan het eind wordt een deelconclusie getrokken of dit bruikbaar is voor de toekomst.

5.1. UML regels

Op het moment dat een programmeur een nieuw project start zit daar ongetwijfeld een database achter. Het kan gaan om een webshop of een internsysteem om alle verkoopcijfers bij te houden. Wat de developer ook zal bouwen het eerste wat hij zal doen is een ontwerp maken van de database. Om dit te modeleren wordt meestal UML³ en een ERD⁴ gebruikt, het ER-model geeft de mogelijkheid om een real-world-scenario te modelleren naar objecten met relaties.[3] Entiteiten kunnen personen, voorwerpen en processen zijn. De relaties geven aan hoe al die processen zich met elkaar verhouden. Verder kunnen de entiteiten nog een set attributen bevatten die meer informatie bevatten over het object. In de Scaffolder heeft een attribuut de vorm van een naam met een datatype. Verder kunnen er nog meer eigenschappen aan mee worden gegeven als attribuutpermissies, wie mag welke eigenschap van welke entiteit bewerken. De tekentool zal zich niet focussen op het toevoegen van attributen, dit zal gedaan worden via het formulier of een IDE.

Relaties binnen het ER-model kunnen verschillende vormen aannemen. We onderscheiden drie verschillende soorten relaties: one-to-one, one-to-many en many-to-many. Een relatie heeft altijd een bron en een bestemming. One-to-many geeft aan dat een entiteit vanaf de bron veel heeft van zijn bestemming. Bijvoorbeeld een Employee kan werken in meer Departments.

Belangrijke punten om rekening mee te houden in de implementatie zijn:

- Circulaire relaties zijn niet toegestaan.
- Er kunnen geen twee relaties van bron naar bestemming lopen.
- Het is mogelijk om een relatie te hebben waarbij de bron en de bestemming hetzelfde zijn, zelfrelatie.

³ UML: Unified modeling languages

⁴ ERD: Entity Relationship Diagram

In de Scaffolder is het ook mogelijk om bij relaties permissies toe te voegen, om aan te geven wie een relatie kan inzien, bewerken of toevoegen. Een mogelijke toepassing hiervoor is als een User een one-to-many relatie heeft met entiteit BlogPost, want een gebruiker kan heel veel blogartikelen posten. Dan zou je kunnen zeggen dat andere gebruikers kunnen zien welke blogs er door die gebruiker zijn gepost. Ook dit zal buiten de implementatie vallen en zal gewoon via een IDE aan de relatie toegevoegd moeten worden. De permissies zijn een veel complexer dan een relatie. Voordat gewerkt wordt aan een manier om de permissies en permissiefilters toe te voegen, zal eerst een eigen notatie ontwikkelt moeten worden om permissies en permissiefilters te visualiseren.

Tenslotte is er volgens het ER-model ook nog de mogelijkheid om een class hiërarchie te definiëren, door bepaalde entiteiten als abstract te markeren. Andere entiteiten kunnen dan doormiddel van inheritance alle eigenschappen erven van een abstracte entiteit. We kunnen dus zeggen dat we de entiteiten Employee en Manager hebben, maar dat beide objecten hun gedrag erven van Person. In de Spec is dit simpel te realiseren door het model voor Person de eigenschap abstract is gelijk aan true mee te geven en bij Employee en Manager geef je de eigenschap inherit mee met de naam van de abstract class.

Je zegt dus als het ware dat Manager en Employee hun gedrag en eigenschappen erven van Person.

Alle objecten hebben dan dezelfde eigenschappen als de parent class. In de UML notatie wordt inheritance, of generalization vaak uitgedrukt met een driehoek, pijl of ruit aan het einde van de lijn. [4]

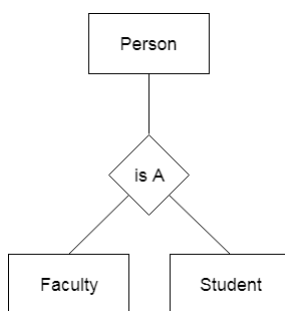
Belangrijke punten om rekening mee te houden bij de implementatie van inheritance zijn:

- Een object kan alleen maar erven van een abstract object.
- Abstracte objecten kunnen ook erven van andere abstracte objecten.
- Een abstract object kan niet van zichzelf erven.
- Er kunnen oneindig veel objecten van een abstract object erven.
- Het officiële teken voor inheritance is een driehoek, waarbij vanaf de top een lijn loopt naar de abstract class en vanaf de onderkant lijnen naar alle child classes. In de driehoek of ruit staat het woord ISA (*staat voor is a*).
- In veel andere diagrammen zie je vaak ook een lijn met aan de kant van de abstracte class een ruit.
- Er zal een afweging gemaakt moeten worden welke notatie makkelijker te tekenen is.

5.1.1. ER-model standaarden

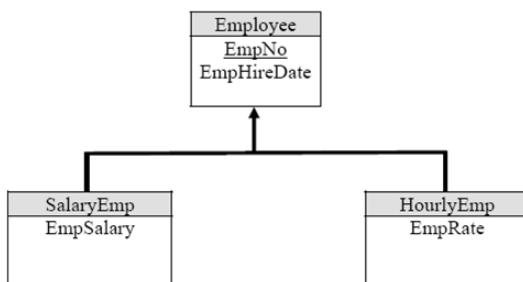
Voor het modeleren van een database in een ERD zijn verschillende notaties en standaarden. Er zal een keuze gemaakt moeten worden welke standaard gebruikt gaat worden in de applicatie. Afwegingen voor de keuze zijn complexiteit van de lijnen en figuren. Als er heel veel details en lijnen over de figuren heen lopen wordt het tekenen ook ingewikkelder. Er wordt een keuze gemaakt uit de volgende standaarden[5][6]:

Chen notatie: Deze notatie is ontworpen door Peter Chen in 1976, zijn idee van het ER-model beschrijft vier lagen. Entiteiten, relaties, informatie structuur en afhankelijkheden. De relaties worden getekend als lijnen met ruiten ertussen waar de naam van de relaties wordt beschreven. Het relatietype staat bij de lijn. Inheritance wordt met een driehoek weergegeven. (zie figuur 27)



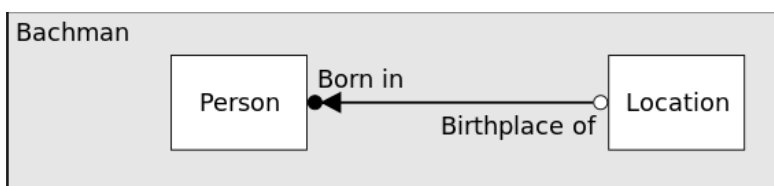
Figuur 27: Voorbeeld 1 inheritance Chen notatie

IDEF1X notatie: Deze notatie wordt vooral gebruikt om de details goed in kaart te brengen. In alle entiteiten worden alle attributen weergegeven. De relaties zijn lijnen met aan de uiteinde een bol of niks om het type aan te geven. Inheritance wordt weergegeven met een pijl. (zie figuur 28)

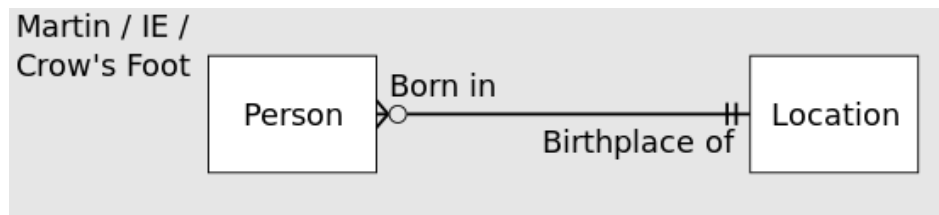


Figuur 28: Inheritanc voorbeeld 2 IDEF1X notatie

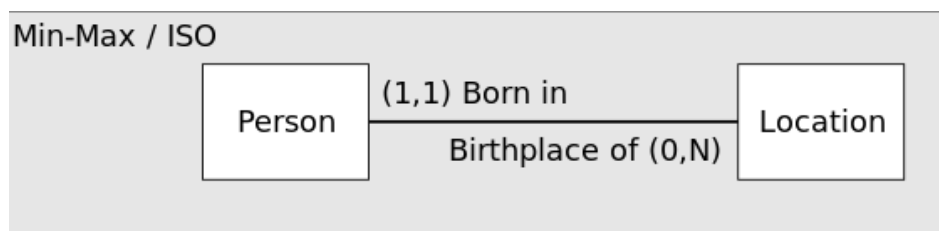
Bachman notatie: Deze notatie is wat simpeler, de attributen worden weggelaten. De relaties worden uitgedrukt door middel van pijlen en inheritance met een ruit.



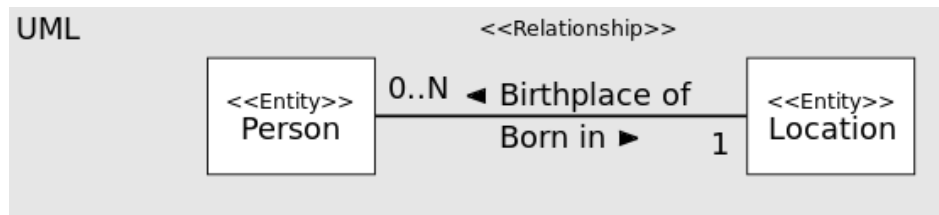
Martin of crow's foot notatie: De notatie zit hem hier al in de naam, om aan te geven dat een relatie 1-N of N-N is worden aan de N kant van de lijn drie schuine lijnen naar de entiteit toe getekend. Dit lijkt op een kraaienpoot, crow's foot. Deze notatie is vrij lastig om te tekenen omdat voor elke kraaienpoot de invalshoek moet worden berekend afhankelijk van de richting van de lijn.



Min-max ISO: Deze standaard wordt vrij veel gebruikt omdat er minder kans is op misinterpretatie, aan de uiteinde van de lijn wordt het relatietype geschreven. Zo hoeft je niet de betekenis van symbolen op te zoeken, maar lees je het af in het diagram. Inheritance wordt uitgedrukt met een pijl.



UML: Een ERD of ER-Model is officieel geen UML, hoewel daar over te twisten valt. Deze notatie wordt vaak gebruikt om databases te ontwerpen en komt overeen met een UML class diagram.

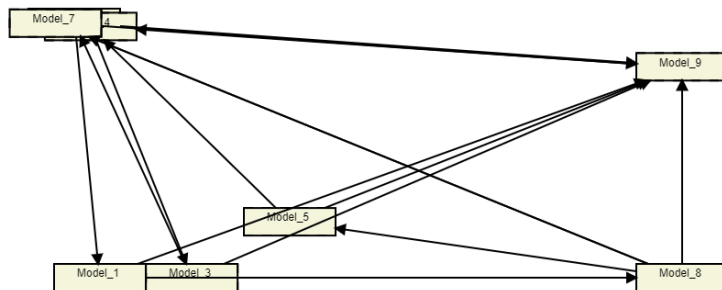


Keuze voor standaard

De keuze voor de standaard van het ER-model binnen de applicatie wordt de min-max-ISO notatie, dit vanwege zijn simpelheid en immuniteit voor misinterpretatie. Het tekenen van lijnen en pijlen met een stuk tekst erbij is ook veel minder complex dan het tekenen van driehoeken op een lijn met kraaienpotten. Hierbij moet rekening gehouden worden dat het diagram in verschillende richtingen getekend kan worden en dus voor elke lijn de invalshoek berekend moet worden ten opzichte van de entiteit. Dus hoe minder lijnen, des te makkelijker de implementatie.

5.2. Lay-out algoritmes

Een van de problemen die je al snel tegenkomt bij het tekenen of plotten van een graph is de positie van alle nodes ten opzichte van hun relaties. De computer weet uiteraard niet waar hij alle elementen moet neer zetten. Als je een bestaande Spec importeert en omzet naar de matrix, kun je willekeurige posities toekennen aan alle nodes. Dan komt de ERD eruit te zien zoals in figuur 25.[10]



Figuur 29: Willekeurige lay-out

Als alle nodes op willekeurige plekken worden gezet is er totaal geen controle over de lay-out. De lijnen gaan over elkaar heen en de kans is groot dat nodes ook op en over elkaar gezet worden.

Een bestaande oplossing voor dit probleem voor het tekenen en plotten van graphs is: Graph layouting algoritmes. Voor de lay-out van de graph is een force-directed drawing algorithm gebruikt, dit behoort tot de basis van graph-layouting[7]. Het doel van het algoritme is om een aantrekkelijke lay-out te produceren op basis van de structuur van de graph en niet op basis van domein specifieke kennis. Onder aantrekkelijke lay-out wordt verstaan een symmetrisch figuur waar de nodes op gelijke afstand van elkaar staan en er zo min mogelijk lijnen door elkaar heen lopen.

De eerste algoritmes komen uit 1963, *The algorithm of Tute*, verder bestaan er nog formules als *The barycentric representation*, *Method of Eades* en *The algorithm of Fruchterman and Reingold*. De laatste is gebruikt om een mooie lay-out te produceren in de tekentool.

Elk van deze algoritmes is gebaseerd op het natuurkundige principe van veerkracht, de graph kun je zien als een groot netwerk van objecten die met veren aan elkaar verbonden zijn. Vergelijkbaar hoe atomen binnen moleculen met elkaar zijn verbonden. Elke node is een metalen ring en elke edge is een veer die de ring met een andere ring verbindt. Als de nodes in vergelijking met de rest ver uit elkaar staan is de veer ver uitgerekt en is er een kracht die die de ringen naar elkaar toe trekt. Als de ringen dicht bij elkaar staan is de veer ingedrukt en is er een kracht die ringen naar buiten duwt. Verder is er ook nog een neutrale fase waarbij de veer in rust is en er geen kracht is die ringen weg duwt of aantrekt. Het algoritme probeert door middel van het verplaatsten van elke ring een gelijke kracht te krijgen op elke edge. Als dit voor alles gelijk is zullen de afstanden tussen alle nodes gelijk zijn.

Deze methode werkt goed voor kleine graphs met een maximum van 100 nodes, worden de graphs groter met meer edges dan wordt het moeilijker om een symmetrische lay-out te genereren en edges niet te laten kruisen. Dit heeft er mee te maken dat het algoritme is gebaseerd op een fysiek model en geen informatie heeft of gebruikt over de richting of de lengte van een edge. Er zijn complexere formules die ook gebruik maken van de richting en de lengte om een betere initiële lay-out te produceren.

Het verloop van het Fruchterman and Reingold algoritme is als volgt, alle nodes worden op een initiële positie geplaatst. Dit wordt willekeurig gedaan. Vervolgens wordt voor elke node de afstotende kracht berekend ten opzichte van de andere nodes en voor elke edge de aantrekkingskracht berekend. Tenslotte worden alle nodes opgeschoven om de kracht en energie tussen alle nodes te beperken. De verplaatsing gaat volgens een logaritme:

$$c_1 * \log(d/c_2)$$

d = De lengte van de veer.

c_1, c_2 = Zijn de veerconstanten

Als d gelijk is aan c_2 dan is de kracht 0, want $\log(1) = 0$.

De eerste formule wordt gebruikt om nodes naar elkaar toe te laten bewegen, om de nodes uit elkaar te laten bewegen wordt onderstaande formule gebruikt:

$$c_3/d$$

Dit alles samengevat naar onderstaand schema:

```

algorithm SPRING( $G$ :graph);
place vertices of  $G$  in random locations;
repeat  $M$  times
    calculate the force on each vertex;
    move the vertex  $c_4 * (\text{force on vertex})$ 
draw graph on CRT or plotter.

```

Figuur 30: Layouting algoritme

Het algoritme wordt M keer herhaald, voor kleine graphs wordt de ideale lay-out al bereikt bij 100 iteraties. Als het vaker wordt uitgevoerd zal de lay-out niet heel veel meer veranderen.

Het originele design van dit algoritme is gebaseerd op Eades algoritme en had twee eisen:

- Alle edges moeten dezelfde lengte hebben.
- Alles moet zoveel mogelijk symmetrisch zijn.

Het Fruchterman and Reingold algoritme uit 1991 voegt daar nog een eis aan toe:

- Gelijke verdeling van nodes.

De nodes worden in dit algoritme behandeld als atomische deeltjes, die tussen elkaar ook krachten bevatten. Hoe het eerste algoritme alleen keek naar de kracht op een edge, wordt bij hier ook gekeken naar de krachten tussen nodes onderling. Ook als er helemaal geen edge tussen de nodes zit.

De aantrekkingskracht f_a en afstotingskracht f_r wordt als volgt berekend:

$$f_a(d) = d^2/k$$

$$f_r(d) = -k^2/d$$

Waarbij d de werkelijke afstand is tussen de nodes en k de gewenste afstand tussen de nodes. Voor k kan een constante waarden worden meegegeven of er kan ideale waarde voor k worden berekend aan de hand van de oppervlakte van het tekenveld.

$$k = \sqrt{\frac{(\text{width} * \text{height})}{\text{number of nodes}}}$$

Voor kleinere graphs voldoet een constante waarde van 100 pixels voor k , maar op het moment dat de graph groter wordt is het beter om de gewenste lengte k te laten berekenen.

Tenslotte voegt het algoritme nog een *temperature* t toe en wordt gebruikt als volgt. De temperatuur begint op een initiële waarde, meestal tien procent van de breedte, en neemt elke iteratie af om aan het einde op nul uit te komen. Het idee erachter is dat de temperature wordt gebruikt om elke iteratie de verplaatsing van elke node kleiner te maken, omdat elke iteratie van het algoritme dichterbij de ideale lay-out komt. Als de temperatuur te snel afneemt dan verplaatst de graph niks meer en neemt de temperatuur te langzaam af, dan zal de ideale lay-out weer worden afgebroken.

In figuur 31 staat de pseudocode van het *Freuchterman and Reingold algoritme*, de complexiteit⁵ van het algoritme is $O(|E| + |V|^2)$ voor elke iteratie. Waar $|E|$ staat voor aantal edges en $|V|$ voor aantal nodes. [8]

```

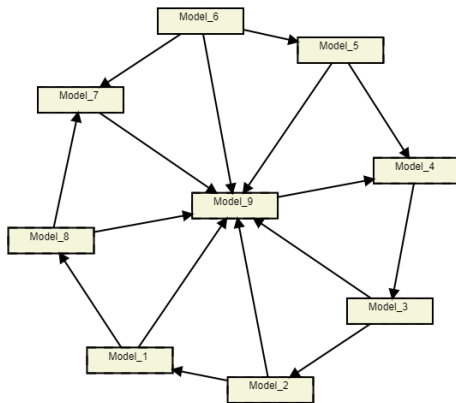
area := W * L; {W and L are the width and length of the frame}
G := (V, E); {the vertices are assigned random initial positions}
k :=  $\sqrt{\text{area}/|V|}$ ;
function  $f_a(x)$  := begin return  $x^2/k$  end;
function  $f_r(x)$  := begin return  $k^2/x$  end;
for i := 1 to iterations do begin
    {calculate repulsive forces}
    for v in V do begin
        {each vertex has two vectors: .pos and .disp}
        v.disp := 0;
        for u in V do
            if (u  $\neq$  v) then begin
                { $\delta$  is the difference vector between the positions of the two vertices}
                 $\delta := v.pos - u.pos$ ;
                v.disp := v.disp + ( $\delta/|\delta|$ ) *  $f_r(|\delta|)$ 
            end
        end
    end
    {calculate attractive forces}
    for e in E do begin
        {each edges is an ordered pair of vertices .v and .u}
         $\delta := e.v.pos - e.u.pos$ ;
        e.v.disp := e.v.disp - ( $\delta/|\delta|$ ) *  $f_a(|\delta|)$ ;
        e.u.disp := e.u.disp + ( $\delta/|\delta|$ ) *  $f_a(|\delta|)$ 
    end
    {limit max displacement to temperature t and prevent from displacement
    outside frame}
    for v in V do begin
        v.pos := v.pos + (v.disp/|v.disp|) * min(v.disp, t);
        v.pos.x := min(W/2, max(-W/2, v.pos.x));
        v.pos.y := min(L/2, max(-L/2, v.pos.y))
    end
    {reduce the temperature as the layout approaches a better configuration}
    t := cool(t)
end
end

```

Figuur 31: Pseudocode Freuchterman and Reingold

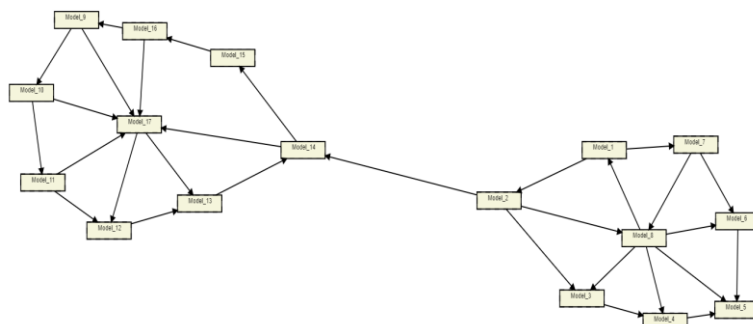
⁵ Big-O notation: Een wiskundige notatie om de complexiteit van een algoritme aan te geven.

Na het succesvol uitvoeren van het algoritme ziet de lay-out eruit zoals in figuur 32 uit. Dit is hetzelfde diagram als in figuur 29. Je kunt zien dat dit een maximale symmetrie heeft en alle edges op gelijke afstand staan.



Figuur 32: Lay-out na algoritme

Op het moment dat we nog een diagram toevoegen aan de graph die met elkaar verbonden zijn (figuur 33) zien we dat twee cirkels niet meer symmetrisch zijn, dit komt omdat de edge van het ene diagram trekt aan de buitenste nodes van het andere diagram.



Figuur 33: Twee gekoppelde diagrammen

Tenslotte kan er nog een scenario zijn waar de diagrammen niet met elkaar verbonden zijn. Wat dan opvalt is dat links en rechts een rechte lijn aan nodes ontstaat langs rand van het tekenoppervlak. Dit komt omdat het algoritme niet om kan gaan met twee graphs en behandelt dit als één diagram.



Figuur 34: Split lay-out

5.3. Toepassing

Voor het tekenen van de Spec moet er een transformatie functie worden gemaakt van de Spec naar een adjacency matrix, dus de graph. De transformatie van Spec naar Matrix is eenvoudiger dan die van adjacency matrix naar een Spec object. Dit heeft er mee te maken dat de matrix een eenvoudige datastructuur is met weinig extra informatie. De matrix bevat alleen informatie over hoe elk model verbonden is met andere modellen.

Het omgekeerde, van matrix naar Spec, is een stuk ingewikkelder omdat je in de matrix veel informatie hebt weggelaten die in de Spec nodig is. Men kan afvragen of het zo handig is om deze operatie te gebruiken, of helemaal niet mogelijk is.

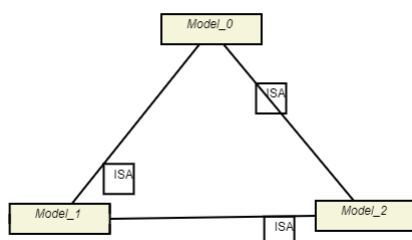
Voor de omzetting worden alle relaties getransformeerd naar edges en entiteiten getransformeerd naar nodes.

Op het moment dat je van de Spec een matrix hebt gemaakt kun je de graph tekenen op een canvas in de browser. Vervolgens wil je het diagram kunnen bewerken. Dat de gebruiker entiteiten kan toevoegen door op het tekenoppervlak te klikken en relaties kan tekenen tussen de entiteiten. Dit is met het canvas framework KonvaJs eenvoudig te realiseren. Wat ingewikkelder is, is dat terwijl je het diagram aan het bewerken bent ook de Spec mee bewerkt. Dit gaat over de volgende acties:

- Het toevoegen van een entiteit.
- Het verwijderen van een entiteit.
- Het aanpassen van de naam.
- Toevoegen van een relatie.
- Verwijderen van een relatie.
- Het aanpassen van het relatietype.
- Relaties omdraaien.

Voor deze set van acties zijn al behoorlijk wat regels over wat wel en niet is toegestaan. Zo zijn circulaire relaties niet toegestaan, dat houdt in dat als er een relatie van User naar Blog is er geen relatie van Blog naar User mag zijn. Dan moet of het relatie type worden aangepast of de relatie moeten worden omgedraaid. Op het moment dat de naam van een model wijzigt moet in de Spec ook de namen van alle relaties van dat model veranderd worden. Anders krijg je relaties die verwijzen naar een model dat niet bestaat, of bestond in dit geval. Dit geldt ook voor het verwijderen van een entiteit, dan moeten alle relaties die daarmee geassocieerd worden ook verwijderd worden. Dit is de basis functionaliteit van het ERD, tot slot is er ook nog de mogelijkheid om inheritance toe te voegen. Hierbij kunnen entiteiten alle eigenschappen erven van een abstracte entiteit. Abstracte entiteiten kunnen ook van elkaar erven. Het is niet toegestaan om een generalisatie loop te creëren. Waarbij een entiteit direct of indirect van zichzelf erft (zie figuur 31).

Om al deze checks op de UML regels en ook die van de Scaffolder uit te voeren wordt per check een functie uitgevoerd die aan elkaar gekoppeld kunnen worden om één functie te produceren.



Figuur 35: Generalisatie loop

5.4. Deelconclusie

Wat het tekenen van het ER-model van de JSON-Spec oplost is dat een developer nu simpelweg de overgang van ontwerp naar code kan overslaan. Dit is een stap die veel tijd kost in het ontwikkelproces. Specificaties van bestaande projecten bij Hoppinger zijn vaak meer dan duizend regels lang en bevatten wel zestig modellen met allemaal onderlinge relaties. Om een ERD over te typen en te vertalen naar JSON-Spec is een herhaaldelijk proces, dat dus geautomatiseerd kan worden. Je kunt de Spec bewerken door modellen toe te voegen aan de graph en op het canvas te tekenen. Vervolgens kun je relaties toevoegen door de getekende modellen te verbinden met lijnen en pijlen.

Dit is ook minder fout gevoelig omdat je geen relaties kunt tekenen tussen modellen die niet bestaan, zo heb je al een minimale foutopsporingstechniek. Meer over foutopsporing in hoofdstuk 6.

Ook veranderingen in de Spec zijn makkelijker te managen. Op het moment dat een modelnaam verandert, of een entiteit wordt verwijderd kan dit nagesynchroniseerd worden met de hoofd specificatie.

Tenslotte kan een nieuwe programmeur snel inzicht krijgen in de structuur van applicaties, aan de hand van een bestaande specificaties. Door bestaande specificaties te importeren in de editor en vervolgens automatisch het ERD te genereren.

De belangrijkste voordelen zijn:

- Snel inzicht in de structuur van bestaande applicaties.
- Makkelijke overstap van de ontwerpfase naar de opzetfase.
- Onderhoudbaar voor alle entiteiten met hun relaties.
- Minimale foutopsporing.

Nadelen en verbeterpunten zijn:

- Er moet rekening gehouden worden in het programma dat elke veranderactie meerdere componenten moet updaten.
- Het toevoegen van een model is een zware operatie op de graph (complexiteit $O(n^2)$)
- Als de Spec groter wordt, wordt het diagram al snel rommelig en onoverzichtelijk. Ook na het uitvoeren van het Fruchterman en Reingold algoritme.
- Gesplitste diagrammen worden nog niet correct weergegeven. Een mogelijke oplossing hiervoor is het splitsen van de graph, het layout algoritme op de losse diagrammen uitvoeren en vervolgens weer samenvoegen. (Divide en conquer approach).

Wat hieruit de conclusie is, is dat een automatische ERD tool met teken functionaliteit zeker potentie heeft om in gebruik genomen te worden. Om volwassen genoeg te kunnen functioneren en om te zorgen dat alles goed werkt moet ook de lay-out van grote specificaties er mooi uit zien op het scherm. Om zo een goed overzicht te krijgen van een applicatie.

Dit is een mooi voorbeeld van schaalbaarheid. Een systeem, algoritme of programma kan werken met een kleine dataset, maar op het moment dat je honderd modellen en vijftig relaties hebt mag je verwachten dat dat geen problemen veroorzaakt. Daarom is het belangrijk om na te denken over performance en complexiteit van algoritmes en zo nodig zuinig om te gaan met zware handelingen of zelfs volledig weg te halen. Want de snelste code is de code die niet wordt uitgevoerd.

6. Integratie & gebruik

In dit hoofdstuk worden de stappen besproken om richting het gebruik van de tool te gaan. Wat er voor nodig is om te zorgen dat dit geen extra stap in het ontwikkelproces wordt, maar echt een verbetering op het ontwikkelproces wordt.

Tot zover is er een tool waar je de Spec mee kunt bewerken op twee niveaus. Eerst op root niveau, dit wordt gedaan via de formuliergenerator. Omdat dit gaat over eigenschappen die in het begin van een project worden ingevuld, zal dit niet zo vaak veranderen. Deze eigenschappen worden als statisch omschreven. Het tweede niveau waarop de Spec bewerkt kan worden is door middel van de canvas tool. Door de entiteiten te tekenen en te verbinden via relaties.

Nu is het de belangrijkste taak om alle losse componenten samen te laten werken om een correcte Spec aan de Scaffolder te kunnen geven. Hiervoor moeten alle eigenschappen als even belangrijk worden beschouwd. Het is niet juist om te beweren dat de eigenschap *make_solution* minder belangrijk is dan *permission_filters*. Omdat de één toevallig meer wordt gebruikt dan de ander. Je mag overigens verwachten dat voor bestaande specificaties alle eigenschappen worden uitgelezen op een correcte manier. (Deelvraag 4)

Naast de tekeninterface en het formulier is er uiteraard nog een derde manier om de Spec te bewerken: Via een IDE. Dit is hoe alle programmeurs gewend zijn om code te schrijven en om specificaties te schrijven. Daarom is het handig om naast het formulier en de grafische interface ook een IDE in de browser te hebben. Zo kan de developer direct de Spec bewerken en ook direct veranderingen zien zodra de Spec wordt aangepast via het formulier of de tekentool. Er zijn bestaande browser editors die hiervoor gebruikt kunnen worden zoals Microsoft Monaco, of Ace editor. De Ace editor is een veel gebruikte en eenvoudig te installeren code editor voor in de browser. Deze wordt gebruikt door veel grote bedrijven als Amazon en Code academy. Ook de code editors en playgrounds van de JSFiddle⁶ website maken gebruik van deze Ace editor. Dus als je een online code editor ergens tegenkomt dan is de kans groot dat dit een Ace editor is. Microsoft Monaco is de code editor die gebruikt wordt voor Visual Studio Code, ook binnen Hoppinger een veel gebruikte IDE. De Monaco Editor kom je onder andere tegen op de Typescript playground website⁷.

Door het gebruik van bestaande herkenbare editors zal de drempel naar het gebruik lager worden, mensen houden overigens niet van verandering.[9]

Beide editors hebben de volgende functionaliteiten al ingebouwd:

- Syntax highlighting.
- Programmeertaal validatie.
- Verschillende thema's.
- Meer dan 45 verschillende programmeertalen beschikbaar.

Om naar het gebruik toe te werken is het een belangrijke feature om bestaande Spec's te kunnen importeren en uit te lezen. Dit is vrij eenvoudig te realiseren aangezien alles in JSON formaat staat en dit makkelijk uit te lezen is door Javascript of Typescript.

Er zit hier overigens één moeilijkheid in, de meeste applicaties bestaan uit meer dan één Spec. Zo kan een hoofdspecificatie op root niveau een eigenschap hebben genaamd *include*. Dit bevat een Array van andere specificatiebestanden waar een lijst met modellen en relaties in staan.

⁶ Lijst met alle Ace editor gebruikers: <https://ace.c9.io/#nav=production>

⁷ Typescript Playground met Monaco Editor: <https://www.typescriptlang.org/play/index.html#>

Om correct om te gaan met de include moet er een aparte feature worden ingebouwd die een lijst van kleinere specificaties toevoegt aan de hoofd Spec. Dit kan handmatig door alle bestanden in de hoofd Spec te kopiëren en te plakken, maar uiteraard kan ook deze handeling geautomatiseerd worden.

Het correct verwerken van de includes moet zo gebeuren dat de eindgebruiker dit niet door heeft. Bij het importeren van de verschillende Spec's moeten alle includes in een aparte lijst gezet worden, alles bij elkaar wordt door het programma als één grote Spec behandeld. Op het moment dat de developer de Spec weer exporteert moeten alle includes weer als losse bestanden gedownload kunnen worden.

De applicatie mag dus niet de structuur van de specificatie bestanden aanpassen. Het moet alleen instaat zijn om mogelijke fouten op te sporen en functionaliteiten toe te kunnen voegen.

De includes werken dus als volgt: In de root van de Spec staat een lijst aan namen die verwijzen naar een andere Spec waar extra relaties of modellen in staan. Deze bestanden staan in dezelfde bestandsmap als de hoofd Spec. Er kunnen dus geen twee bestanden in de lijst dezelfde naam hebben. Bij het uitlezen van de include parameter worden de overige Spec's gelezen en toegevoegd bij de relaties en modellen van de hoofd Spec, op die manier produceert de computer één grote Spec. Dit is hoe de Scaffolder om gaat met de includes, het voordeel bij de Scaffolder is dat die niet terug hoeft te gaan van één grote applicatie specificatie naar losse specificaties.

Dit moet de tool wel kunnen omdat elke eigenschap van de Scaffolder gerespecteerd moet worden en niet zomaar de aanname gemaakt kan worden dat die minder gebruikt zal worden of in de toekomst zal verdwijnen.

Een manier om dit te realiseren is door middel van een dictionary. Dit is een lineaire datastructuur waarbij alle waarden gekoppeld zitten aan een unieke key. Die key wordt gegenereerd aan de hand van een hash algoritme.

Om de key te produceren gebruik je in dit geval de naam van het bestand dat in de include staat en de waarde is de genestelde Spec. Zo kan kun je de juiste Spec op vragen aan de hand van de naam van het bestand. Het voordeel is ook dat de naam dan uniek moet zijn en dat is ook een eis voor alle namen in de include. Op deze manier houd je de includes gescheiden van de hoofd Spec, het samenvoegen gebeurt dan gewoon in memory. Bij het exporteren worden dan de hoofd Spec samen met zijn includes gedownload.

Het implementeren van de includes voor de Spec is de laatste stap voor de integratie met de Scaffolder. Deze parameter is dus net zo belangrijk als alle andere eigenschappen, maar vereist wel een aparte behandeling om te zorgen dat de applicatie er correct mee overweg kan gaan. Op het moment dat deze feature correct geïmplementeerd is, is de stap naar het gebruik van de tool kleiner. Zo kan elk development team zijn bestaande specificaties in de applicatie importeren en er direct mee aan de slag gaan. Als een developer toch nog handmatig een Spec wil bewerken, dan kan hij die downloaden, bewerken en vervolgens opnieuw importeren.

In het laatste hoofdstuk volgt de laatste stap voor het produceren van correcte specificaties die direct aan de Scaffolder kunnen worden meegegeven. Hier zal dieper worden ingegaan op alle mogelijkheden van de Scaffolder, alle regels wat wel en niet is toegestaan in de Scaffolder en hoe hier een goede foutopsporing op toegepast kan worden.

7. Foutafhandeling

Met de formuliergenerator, de IDE in de browser en de tekentool is er een goede oplossing om de Spec voor de Scaffolder te bewerken. Er wordt inzichtelijk gemaakt hoe relaties en entiteiten met elkaar samenhangen en de developer krijgt feedback van alle beschikbare functionaliteiten van de Scaffolder. Wat nu nog niet kan is het opsporen van fouten, of het wel mogelijk is om een bepaalde relatie aan te maken, het opsporen van generalisatieloops en regels voor het toekennen van permissies. Dit hoofdstuk is het laatste technische hoofdstuk van deze scriptie en zal gaan over hoe fouten door de tool worden opgespoord en uit de Specs worden gefilterd.

Onder foutopsporing wordt verstaan het valideren van de juiste keywords, eigenschappen en interne logica van de Spec. Er is geen typesafe framework gebouwd, op de formbuilder na, die voor honderd procent kan garanderen dat alle eigenschappen volledig typesafe zijn. In principe is er al een minimale typesafety gerealiseerd door het gebruik van Typescript. Dit zorgt er al voor dat alle eigenschappen van de Spec de juiste datatypes hebben.

De foutopsporing gebeurt op drie niveaus. Ten eerste wordt op syntax gecheckt. Het formaat van de Spec is JSON dat is een gestandaardiseerd format. In de editor die in de browser geïnstalleerd is zit al een syntax checker voor verschillende programmeertalen en configuratiescripts. Zo ook voor JSON, deze syntax validator checkt of er geen komma's ontbreken, dat alle haakjes netjes zijn afgesloten, dat de aanhalingstekens voor strings niet ontbreken etc. Dit is een minimale check waar de tool aan moet voldoen om ooit in gebruik genomen te gaan worden, want in elke IDE zit deze functionaliteit ook.

Als tweede check kan je nog een JSON schema validator installeren, een schema validator checkt niet de syntax op komma's, maar controleert alle eigenschappen binnen het JSON bestand. In Visual Studio Code zitten ook meerdere schemavalidators geïnstalleerd, voor configuratiebestanden als tslint, tsconfig en package.json van npm⁸. Er is ook de mogelijkheid om eigen schema's aan te maken, op basis van een model voor een object.

In onderstaand figuur (figuur 37), staat een voorbeeld van een schema. Dit schema geeft aan dat het JSON bestand van type object moet zijn met eigenschappen p1 en p2 en de waarden van p1 is of "v1" of "v2". Op het moment dat je in de IDE een fout maakt krijg je een handige foutmelding, want het bestand moet voldoen aan het schema. [11]

```
schema: {  
  type: "object",  
  properties: {  
    p1: {  
      enum: ["v1", "v2"]  
    },  
    p2: {  
      type: "object",  
      properties: {  
        q1: {  
          enum: ["x1", "x2"]  
        }  
      }  
    }  
  }  
}
```

Figuur 37: Voorbeeld schema

```
"p1": "v3",  
"p2": false
```

Incorrect type. Expected "object".

Peek Problem No quick fixes available

Figuur 36: Voorbeeld foutmelding schema

⁸ NPM: Node Package Manager

Dit is slechts een klein voorbeeld van een simpel schema, maar op het moment dat je van de hele Spec een schema maakt met validatieregels heeft de programmeur al een extra controlestap bovenop syntaxvalidatie. Dit is overigens niet alleen een goede manier voor foutopsporing, maar ook het bijhouden van documentatie. Zo kan je binnen het validatieschema niet alleen regels voor vaste waarden en datatype aangeven, maar ook een beschrijving die vertelt wat een bepaald keyword of eigenschap doet. Zo heeft de programmeur de softwaredocumentatie rechtstreeks in zijn editor.

De schemavalidator heeft de volgende eigenschappen: Typesafety, verplichte eigenschappen, beperkte waarden voor input, vast input patroon en unieke lijsten. Er zijn nog veel meer eigenschappen die op de documentatie website staan, maar dit zijn de belangrijkste die direct gebruikt worden door het validatieschema.

Ten slotte is er nog een derde vorm van validatie, die zelf gebouwd zal moeten worden. Dat is de validatie op interne logica van de Scaffolder en JSON Spec's. Er zijn veel regels waar een Spec aan moet voldoen. Veel van de regels staan al beschreven in hoofdstuk 4.1 die komen overeen met de UML standaarden. Aan die regels kan voldaan worden door de bestaande datastructuren toe te passen. Verder zijn er nog een aantal regels voor eigenschappen die niet met datastructuren kunnen worden opgelost en er dus eigen validatiefuncties geschreven moeten worden. Elke functie is verantwoordelijk voor het controleren van één specifieke eigenschap van de Spec en moet een foutmelding teruggeven als het niet aan de eisen voldoet. De reden dat hiervoor gekozen is, is vanwege onderhoud, op het moment dat er een regel bijkomt of een eigenschap bijkomt dan kan er makkelijk een nieuwe functie worden bijgeschreven of worden aangepast.

Voorbeelden van een aantal validatieregels zijn: Een permissie of permissiefilter kan alleen entiteiten bevatten die kunnen inloggen. Een entiteit kan zijn eigenschappen alleen erven van een abstracte entiteit, het is niet toegestaan dat de parent class tegelijkertijd van zijn child erft. (Generalisatieloop). Ook relaties kunnen geen loops bevatten. Een modelnaam moet uniek zijn. Een relatie moet uniek zijn. De lijst met permissiefilters geeft een pad aan van een in te loggen entiteit naar zijn target entiteit.

Dit zijn een aantal voorbeelden van validatie regels een aantal daarvan is al op te lossen met eerder genoemde methodes. Zoals de unieke lijst aan permissies en dat een modelnaam niet met een cijfer mag beginnen. Het garanderen van een unieke lijst aan modellen en relaties werkt niet in de schema validator, want die kan alleen maar kijken op unieke waardes van primitieve datatypen zoals boolean, string en number, maar niet voor objecten. Deze vergelijking moet dus in code worden uitgevoerd.

Het filteren van generalisatieloops is ook een interessante casus, dit is iets wat in veel object georiënteerde programmeertalen niet is toegestaan en al de nodige foutmeldingen oplevert. Zoals in talen als C#, Java en ook Typescript. Wat die precies inhoudt is dat het niet is toegestaan dat een class erft van andere class terwijl die andere class ook van de ene erft. Je krijgt dan een foutmelding dat de class direct of indirect van zichzelf erft. Zie het voorbeeld zoals in figuur 38.

```
25 |
26 | class Parent
27 | 'Parent' is referenced directly or indirectly in its own base expression. ts(2506)
28 |
29 | class Parent extends Child {
30 |
31 | }
32 |
33 |
34 | class Child extends Parent {
35 |
36 | }
37 |
```

Figuur 38: Inheritance loop (direct)

Er bestaat de mogelijkheid om te zeggen dat Child zijn eigenschappen erft van Parent, wat een logische bewering is. Op het moment dat Parent tegelijkertijd zijn eigenschappen erft van Child, wat in een praktijkscenario ook onmogelijk is, krijg je dus een inheritance loop. Dit wordt veroorzaakt doordat de compiler eerst in Child opzoek gaat naar de Parent class en vervolgens in Parent weer naar Child en in Child weer naar... Een oneindige loop dus.

Dit is een voorbeeld van een fout die de Scaffolder er niet uithaalt en waar je dus pas achter komt als je een project opent.

Met twee classes is het nog simpel om de fout op te sporen, zie hoofdstuk 4, omdat in de adjacency matrix dit wordt weergegeven als twee objecten die naar elkaar toe verwijzen, dit kan worden opgelost door de een 1 en een 0 om te draaien. Met meer dan twee objecten wordt het al ingewikkelder, want je kan ook een generalisatieloop hebben met meer objecten. Zoals in figuur 39.

```
18 class GrandGrandParent
19 'GrandGrandParent' is referenced directly or indirectly in its own base expression. ts(2506)
20
21 Peek Problem (Alt+F8) No quick fixes available
22 class GrandGrandParent extends Child {
23
24 }
25
26 class GrandParent extends GrandGrandParent {
27
28 }
29
30 class Parent extends GrandParent {
31
32 }
33
34 class Child extends Parent {
35
36 }
37 }
```

Figuur 39: Indirecte generalisatie loop

Dit is in een graph ook moeilijker op te sporen, om deze fout te vinden moet je echt in de context van elk object kijken en controleren waar naar wordt verwezen. Je onthoudt de startpositie en kijkt elke keer naar de context van de parent class. Op het moment dat de parent hetzelfde is als de start class is er dus een generalisatieloop.

Dit is een oplossing voor de ingewikkeldste foutopsporing van de tool. Het is nooit mogelijk om voor honderd procent alle fouten uit de Spec te halen en in een later stadium zul je altijd weer nieuwe fouten tegenkomen en bepaalde edge cases waar nog geen rekening mee is gehouden. Door verschillende methoden toe te passen van foutopsporing kun je toch een relatief correcte Spec produceren. In dit geval wordt er gebruik gemaakt van twee bestaande technieken die al een aantal jaren van ontwikkeling op hun naam hebben staan uitgebreid met een eigen methode, specifiek voor de specificatie van de Scaffolder. Het doel is ook niet om nu al een volledig dichtgetimmerde foutenanalyse te hebben, maar om de basis te leggen die later nog uitgebreid kan worden. Wat hier ook zal worden meegenomen is dat de Scaffolder alleen wordt gebruikt door beroepsprofessionals, dus niet alle fouten hoeven eruit gehaald te worden. We gaan er vanuit dat iedereen die met de Scaffolder werkt al achtergrondkennis heeft van databases en applicatieontwikkeling.

8. Conclusie

Het doel van het gebouwde prototype was het visualiseren van het ontwikkelproces, om een beter inzicht te krijgen in specificaties en eigenschappen van grote applicaties. Zonder dat hier extra tools aan te pas komen die een grote extra onderhoudslast veroorzaken, of het bijhouden van extra documentatie. Om te bepalen in hoeverre het doel behaald is zal eerst teruggekeken worden op de gestelde deelvragen.

De eerste deelvraag was hoe je Scaffolder Fabric zo ontwerpt dat de gebruiker niet hoeft na te denken over documentatie. Documentatie bijhouden en onderhoud moet altijd gedaan worden voor wat voor software dan ook. De vraag is hier dan ook hoe je de onderhoudslast zo klein mogelijk maakt voor de programmeur. Dit wordt voor een groot deel opgelost door de FormBuilder, die bevat één query die de verplichte Spec eigenschappen selecteert. Als hier nieuwe eigenschappen bijkomen geeft de query een foutmelding dat er eigenschappen ontbreken, door slim gebruik te maken van het type systeem van Typescript. Verder zijn de optionele eigenschappen opgeslagen in een apart object waarvan het type is afgeleid van de originele Spec. De optionele eigenschappen worden als checkboxes weergegeven, zodat de gebruiker feedback krijgt over functionaliteiten waarover de Scaffolder beschikt. Zo hoeft een nieuwe developer niet meer door de readme te scrollen op zoek naar de juiste feature. Tot slot is het JSON-schema een goede plek om in de toekomst documentatie bij te houden, door onder elke eigenschap de beschrijving in te vullen. Zo ziet de developer de documentatie vanuit zijn eigen IDE, ook kan dit schema op een centrale plek gehost worden zodat iedereen altijd de laatste versie heeft.

Een andere belangrijke deelvraag is of er transformaties naar andere datastructuren nodig zijn en zo ja, hoe zorg je ervoor dat de Spec intact blijft en er geen gegevens verloren gaan. Om de Spec te visualiseren is een Graph datastructuur gebruikt. Bij het omzetten van een Spec naar Graph gaat veel interne informatie verloren. Het enige wat bewaard blijft zijn de relaties en entiteiten, alle overige gegevens zijn niet meer terug te halen vanuit de Graph. Ook alle eigenschappen van individuele modellen en relaties gaan verloren. Er wordt een index bijgehouden van alle nodes die corresponderen met de juiste modellen van de Spec. Het is in dit geval noodzaak dat de volgorde van de modellen niet veranderd, zo wel moet de graph opnieuw gemaakt worden. Een andere datastructuur die gebruikt is, is een dictionary. Deze wordt gebruikt om de deel Spec's op te slaan, bij deze datastructuur gaat er geen informatie verloren, want in de dictionary worden de keys direct gekoppeld aan de volledige deel Spec. Die vervolgens weer bijgevoegd kan worden in de hoofdspec. Ook het grote voordeel is dat de keys uniek zijn en er dus geen dubbele namen kunnen voorkomen in de lijst met deel Specs.

Vervolgens heb ik mij afgevraagd wat de belangrijkste prioritering is voor de verschillende eigenschappen van een Spec. Elke eigenschap heeft een eigen datatype, de simpele primitieve datatypes als string, number en boolean zijn gemakkelijk te verwerken. Voor zowel het vergelijken als het bewerken van de eigenschappen. Verder zijn er ook veel complexere datatypes zoals de zogenaamde union types. Deze worden door elke andere programmeertaal behandeld als een string, maar in Typescript hebben ze de speciale eigenschap dat het een specifieke string moet zijn die in de union staat. Verder is het lastig om onderscheid te maken tussen de datatypes: array, array van objecten en object. Deze lijken qua structuur vrij veel op elkaar, omdat array ook een object is. Hier zijn dus extra checks voor gebouwd, om onderscheid te kunnen maken tussen deze verschillende datatypes.

Dan is er nog de specifieke logica van verschillende eigenschappen. Ik zal niet meer uitgebreid ingegaan op alle verschillende context gebonden eigenschappen. Het komt er op neer dat sommige aspecten van de Spec afhankelijkheden met andere eigenschappen hebben terwijl andere eigenschappen puur los van de rest kunnen functioneren. Zo zijn eigenschappen als *make_solution* en *hot_reloadable* puur op zichzelf en hebben *permissions* en *permission_filters* een lijst met alleen bestaande model namen.

Voor dit probleem kun je als oplossing houden dat alle eigenschappen even belangrijk zijn en er geen aannames gemaakt worden of iets weg gelaten kan worden, of welke eigenschap vaker gebruikt gaat worden. Door verschillen in afhankelijkheden krijg je wel dat het bewerken en toevoegen van verschillende eigenschappen verschillend geïmplementeerd wordt. In sommige gevallen en voor performance redenen is het dan handiger om toch pas in de Scaffolder te checken of een model wel bestaat in een relatie. Het kost ook extra rekenkracht om voor elke relatie, permissie en permissiefilter te controleren of alle waarden wel zijn toegestaan.

Vervolgens de integratie met de huidige Scaffolder, voordat ik ga beantwoorden of dit mogelijk is, gaan we eerst naar het gebruik toe van de applicatie. De huidige situatie bij Hoppinger voor developers is dat ze specificaties configureren door middel van een JSON bestand en dit bestand bewerken ze met een IDE. De meeste developers gebruiken hiervoor Visual Studio Code. Dus om de drempel voor het gebruik van Scaffolder Fabric zo laag mogelijk te maken, moet de tool ook zo aanvoelen als van een IDE. Zie hoofdstuk 5 over integratie en gebruik. Dus naast het formulier en de tekentool komt dan ook nog een editor om de Spec te bewerken. Als die editor er dan ook nog hetzelfde uitziet als de IDE die de meeste programmeurs gebruiken, kunnen de programmeurs in ieder geval de Spec zo bewerken op een manier die ze kennen en die bekend aanvoelt.

Wat de integratie met de huidige Scaffolder betreft is iets wat in een later stadium gedaan kan worden. Het idee is dat er in Scaffolder Fabric een run knop komt, waar de gebruiker de Scaffolder mee kan uitvoeren. De focus bij dit onderzoek ligt meer op het visualiseren van de specificatie dan op het uitvoeren ervan. Op dit moment bestaat er al een goede manier om de Scaffolder uit te voeren, namelijk de command line. Dus als een developer in staat is om een vanuit Scaffolder Fabric een specificatie te downloaden, kan die altijd via de command line uitgevoerd worden door de Scaffolder. De manier om in de toekomst de editor uit te voeren vanaf Scaffolder Fabric is door de Scaffolder te verplaatsen naar de cloud, samen met de interface. Op het moment dat dat zo ver is moet er ook gedacht worden aan autorisatie, hoe worden Spec's opgeslagen, wie mag het bewerken? Etc. Dat is een heel ander project en iets voor de toekomst, wellicht dat de Scaffolder in staat is om daarvoor zijn eigen backend te bouwen.

Tot slot, wat voor bestaande tools zijn er gebruikt om dit project te realiseren? De formulier generator die gebouwd is, is een volledig eigen project gebaseerd op het type systeem van Typescript. Ook het onderzoek van Barld Boot is gebruikt om inzichten uit te halen voor het prototype. In zijn onderzoek was het doel om een volledig typesafe framework te bouwen voor de Scaffolder, dit was ook gelukt. Het enige probleem was de performance van het framework en om die reden toch niet in gebruik genomen. Verder is het voor het uittekenen van de specificaties van applicaties een HTML canvas teken framework gebruikt (KonvaJS). Omdat het tekenen op canvas vrij low-level is en alle figuren en lijnen apart gedefinieerd moeten worden is het slimmer om een bestaand framework te nemen waarbij al die figuren al bestaan. Zo kan je je meer focussen op de logica van de applicatie en het werkelijke doel en niet op het tekenen van rechthoeken, lijnen en pijlen. Voor de code editor is Microsoft Monaco gebruikt omdat deze qua gebruik heel veel aanvoelt als Visual Studio Code. De IDE die het meest gebruikt wordt binnen Hoppinger. Het is bij het bouwen van elk software product van belang om goed de doelen te inventariseren en af te vragen voor wie iets is en waarvoor het gebruikt gaat worden. Dan kan ook veel beter bepaald worden wat voor bestaande tools en technieken gebruikt kunnen worden en wat zelf gebouwd gaat worden. Dit bespaart in de praktijk veel tijd en geld.

De hoofdvraag van dit onderzoek is: *“Hoe kan een visualisatie van een applicatie specificatie ingezet worden om het ontwikkelproces te verbeteren?”*.

Er is dus een visualisatie tool gebouwd genaamd Scaffolder Fabric, die de mogelijkheden heeft om alle relaties en entiteiten weer te geven. De tool kan ook als IDE gebruikt worden en de programmeur kan dus specificaties bewerken zoals die dat voorheen ook gedaan heeft. Om het gebruik van de editor beter te maken is er een JSON schema validator geconfigureerd die controleert op alle toegestane waardes inclusief datatypes. De programmeur krijgt dus zo suggesties over de mogelijke eigenschappen en alle functionaliteiten van de Scaffolder. Op deze manier worden veel fouten er al uitgehaald.

Vervolgens is er nog het diagram waar de entiteiten getekend worden met de verschillende relaties en inheritance. Omdat de Spec naar een zogeheten graph datastructuur getransformeerd wordt zijn een aantal fouten ook al makkelijk op te sporen, door de regels van de datastructuur te volgen. Zo zijn circulaire relaties en generalisatie loops niet toegestaan. Het belangrijkste van de foutopsporing is het detecteren van een set regels, in dit geval die van UML en de Scaffolder. Vervolgens kunnen die regels vertaald worden naar bestaande datastructuren met bestaande algoritmes. Als er dan in de Spec iets niet klopt kom je daar achter omdat het formaat niet past binnen de regels van de bestaande datastructuur. Zo worden verwijzingen naar entiteiten die niet bestaan binnen relaties er uitgefilterd. Omdat de datastructuur geen object kan verbinden met een ander object dat niet bestaat. In sommige gevallen is het nodig om extra regels hieraan toe te voegen, omdat dit gaat over context specifieke logica. Zo is het in een graph wel toegestaan om een object met zichzelf te verbinden, dit kan voor relaties wel, maar niet voor inheritance. Er zijn nog meer voorbeelden op te noemen van dit soort gevallen, maar het principe van context specifieke logica transformeren naar een bestaande datastructuur blijft hetzelfde.

Het ontwikkelproces kan dus verbeterd worden met een visualisatie van de Spec door rekening te houden met alle bestaande tools en te analyseren wat de huidige situatie is voor software ontwikkelaars op dit moment binnen het bedrijf. Om een nieuwe tool te introduceren moet rekening gehouden worden met de gewoontes van de mens. Mensen houden overigens niet van verandering, op het moment dat het leren van een nieuwe tool een te hoge drempel is zijn mensen waarschijnlijk ook niet geneigd om het in gebruik te nemen. Je moet mensen het gevoel geven bij de huidige situatie te blijven en dat door de nieuwe ontwikkelingsmethode het proces eenvoudiger wordt. Het idee van de visuele editor is ook dat in een paar muisklikken al meer dan honderd regels code worden gegenereerd, die de developer normaal gesproken had moeten kopiëren en plakken, of had moeten typen.

Dus door het introduceren van Scaffolder Fabric wordt niet de huidige situatie vervangen, maar aangevuld en verbeterd.

9. Aanbeveling

Tijdens het bouwen van de tool Scaffolder Fabric kwam ik veel uitdagingen tegen die stuk voor stuk een slimme oplossing vereiste. In dit hoofdstuk zal teruggeblikt worden op het ontwikkelproces van de tool en een aanbeveling worden gemaakt voor de verschillende oplossingen, welke het beste werkt en hoe dit het beste kan worden toegepast.

Terug naar het begin van het ontwikkelproces, één van de eerste dingen die gedaan is, is het analyseren van alle functies van de Scaffolder en alle eigenschappen van de Spec. Hieruit bleek dat veel eigenschappen bestaan uit primitieve datatypen, zoals integers, strings en booleans. Deze waarden kunnen worden ingevuld door een formulier, dan gaat dit over eigenschappen op root niveau. Het gemak bij de root eigenschappen is dat hier geen herhalingen in voor kunnen komen en die bovenaan het JSON-bestand komen te staan. Bij de modellen en relaties zit veel herhaling door het feit dat het gaat over een lijst van objecten. De Spec heeft op het moment van schrijven 31 eigenschappen op root niveau, elk model heeft 37 optionele eigenschappen en de relaties nog eens 19 eigenschappen. Als je 30 modellen hebt met 20 relaties en je voor elke eigenschap een formulier veld maakt dan kom je op meer dan honderd velden uit die de gebruiker moet invullen. Mijn aanbeveling hierop is dan ook om de formuliergenerator alleen te gebruiken voor de eigenschappen op root niveau. Zo hou je het maximale aantal velden constant. De formuliergenerator is zo generiek gebouwd dat het op elk datamodel toepasbaar is en zo ook gebruikt kan worden in andere applicaties. De formbuilder gaat de onderhoudbaarheid van de software ten goede, omdat er geen grote formulieren meer in de applicatie staan in HTML. Je kan dus een apart bestand maken met alle queries voor de verschillende formulieren. Als er ergens een veld bij moet komen, kan die toegevoegd worden aan het datamodel en de query.

Tot aan hier is het dus mogelijk om de root van de Spec te bewerken op een Typesafe manier die ook nog eens onderhoudbaar is. Omdat het niet erg gebruiksvriendelijk is om een developer meer dan honderd formuliervelden in te laten vullen zal voor het bewerken van modellen en relaties een andere oplossing moeten komen. Op dit moment wordt de Spec door alle programmeurs bewerkt vanuit hun eigen IDE, een IDE is tegenwoordig een geavanceerd stuk software met allemaal mogelijkheden. Zoals syntax high lighting, auto complete en schema validators. Deze manier van het bewerken van Spec's en code is zo generiek en veel gebruikt dat alle programmeurs hieraan gewend zijn. Microsoft besteed veel tijd en geld aan de ontwikkeling van deze tools, dus waarom zou je het wiel opnieuw uitvinden en deze tools willen vervangen. Door een IDE te integreren in de browser, in Scaffolder Fabric, blijft de manier van code bewerken hetzelfde. In dit geval is de Microsoft Monaco editor gebruikt, dezelfde editor als voor Visual Studio Code. De IDE is de basis voor het bewerken van Spec's, deze wordt aangevuld met de nieuwe oplossingen die het ontwikkelproces eenvoudiger moeten maken. Om modellen en relaties toe te voegen wordt dus alleen het diagram gebruikt in combinatie met de editor. Dit diagram helpt om inzicht te krijgen in bestaande Spec's en in een paar muisklikken heb je twintig modellen toegevoegd gelijk aan meer dan honderd regels code. Verder kan per model wel een apart formulier worden gemaakt, die niet altijd zichtbaar is op het scherm. Dit formulier kan gebruikt worden om de verplichte eigenschappen te bewerken. Over het algemeen is het in dit geval het makkelijkst om de entiteiten te produceren via het diagram en vervolgens de details te veranderen via de IDE.

Tenslotte, hoe sluit het gebouwde product, Scaffolder Fabric, aan bij de doelstelling van het onderzoek: Het verbeteren van het ontwikkelproces bij Hoppering?

Het is een feit dat door het produceren van snippets van code met een muisklik het ontwikkelproces eenvoudiger wordt. Door het gebruik van Microsoft Monaco lijkt het ontwerp van de editor veel op die van Visual Studio Code en is de drempel voor het gebruik lager. Door de relaties en entiteiten visueel te maken krijgt de developer meteen inzicht in de structuur van een applicatie. De code editor is wat mij betreft klaar voor gebruik, samen met het gebouwde JSON validatie schema. Dit verandert ook niks aan de huidige situatie, alleen dat de Spec nu vanuit de browser bewerkt wordt. De formulier generator kan ook in een productie omgeving gebruikt worden, niet alleen voor Scaffolder Fabric, maar kan ook breder worden ingezet. Je kan zelfs alle formulieren hiermee produceren. Hier zou je een nieuw onderzoek op kunnen baseren.

De ERD editor is op dit moment nog in de ontwikkelfase en bevat alle basis functionaliteiten, het toevoegen van relaties, inheritance en entiteiten. In het begin van het onderzoek was er vraag naar om ook permissies te tekenen in het diagram. Hier zitten twee keerzijdes aan de eerste is dat er dan wordt afgeweken van de standaarden van het ER-model aangezien permissies meestal geen onderdeel zijn van database design. Ten tweede, door alle permissies te tekenen (create, edit, view en delete) krijg je veel extra lijnen op het scherm die het diagram onleesbaar maken. Je zou hier dan een optie voor kunnen inbouwen om dit wel of niet weer te geven. Verder is het managen van de state van het diagram ook vrij complex. Elke verandering aan het diagram moet worden doorgevoerd in de Spec om herhaaldelijk werk te voorkomen. Zo heeft een naamswijziging van een entiteit vele gevolgen voor de Spec. Want deze naam wordt gebruikt in al zijn relaties, permissies, permissiefilters en inheritance. Dan ben ik er in dit rijtje vast nog wel één vergeten, om aan te geven dat bij zo een verandering veel afhankelijkheden zitten en er snel één vergeten wordt.

Eindaanbeveling

Tenslotte, mijn aanbeveling voor het gebruik van Scaffolder Fabric. Op dit moment is het een handig hulpmiddel dat kan dienen om inzicht te krijgen in de structuur van bestaande applicaties en om snel van start te kunnen gaan met een nieuw project. De software is nog niet betrouwbaar genoeg om volledig een Spec te kunnen tekenen, ook is het nog niet mogelijk om alle opties te kunnen bewerken vanuit de tekentool. Men kan wel verschillende opties voor modellen en relaties aanvinken, maar hiervoor worden de standaard waarden gebruikt. Op het moment dat er specifieke waarde nodig zijn ontkomt de programmeur er niet aan om de specificatie handmatig te bewerken.

10. Eigenreflectie

In dit hoofdstuk zal ik terugblikken op hoe ik het ontwikkelproces heb ervaren en heb aangepakt, wat ik beter heb gedaan ten opzichte van andere schoolprojecten en waarom ik bij Hoppinger ben gaan werken.

Een jaar voor mijn stage volgde ik een minor software engineering, waar Guiseppe Magiore de theorie van verzorgt en Francesco Di Giacomo onderdeel is van het praktijkteam. De minor is een introductie tot categorie theorie en probeert deze principes toe te passen op moderne software engineering methodes als functors, monoids en monads. Ik zal niet te diep ingaan op de inhoud van de minor. Bij de minor deed ik ook een project: “Bouw een volledig typesafe LINQ framework in Typescript”. Tijdens dit project heb de theorie kunnen toepassen en heb ik een studie gedaan naar het geavanceerde type systeem van Typescript. Aan het eind van de minor gaven de docenten aan dat ze de kennis van de minor ook toepassen bij Hoppinger, het bedrijf waar een aantal Hogeschool docenten tevens werkzaam zijn. De reden dat ik voor Hoppinger heb gekozen om mijn afstudeerstage te doen is omdat ik een vervolg wilde op de concepten van de minor. Om dieper in te gaan op de stof en dit toepassen op een ‘real world scenario’.

Toen ik begon aan mijn opdracht heb ik het anders aangepakt dan de meeste schoolprojecten. Wat ik me herinner van de eerste- en tweedejaarsprojecten is dat we meestal meteen begonnen met bouwen. Dit leidde op het eind vaak tot dilemma’s met de eisen omdat we niet altijd goed keken naar wat het minimaal acceptabele prototype was. Ik heb nu beter geleerd om de belangrijkste eisen van te voren vast te stellen. Ook als het gaat voor de keuze van technologie, talen en tools om te gebruiken. In het eerste jaar ben ik vaak van de eerste de beste optie uit gegaan, terwijl ik nu eerst meerdere alternatieven onderzoek voordat ik het implementeer in het project.

Zo heb ik in het prototype meerdere mogelijkheden uitgezocht om figuren te tekenen op het scherm. Dit kan uiteraard met een HTML5 canvas, maar dat is vrij low-level. Hiervoor vond ik twee frameworks de eerste was fabricJS die veel voor gedefinieerde figuren bevat die je simpelweg kunt aanroepen. Het nadeel hiervan is dat dit framework niet goed samenwerkte met React, daarvoor vond ik later KonvaJS. Dit is een voorbeeld dat je niet altijd de eerste de beste optie hoeft te kiezen, hetzelfde geldt voor de keuze van browser IDE. Door mijn afstudeerstage zie ik nu ook het nut in van een goed vooronderzoek, door eerst goed te zoeken naar wat er al bestaat bespaar je veel tijd die je kunt besteden aan het bouwen van de daadwerkelijke applicatie en ontbrekende logica. Er is overigens geen reden om het wiel opnieuw uit te vinden.

Het ontwikkelproces is vrij gestructureerd verlopen, van te voren had ik een globale planning gemaakt waar de stappen en verschillende competenties staan beschreven. Elke twee weken sprak ik met Guiseppe voor een demo van het project en om de vervolgstappen te bespreken. Dit is ook hoe het er aan toe gaat als je in gesprek gaat met externe klanten. Aan het eind van elke sprint komt de klant weer langs om te kijken naar de voortgang van het project en komt met nieuwe eisen of vertelt welke eisen nog niet voldoen.

Tot slot was dit een mooi project als voorbereiding op de praktijk, waar ik mijn in de afgelopen vier jaar opgedane kennis kon toepassen.

11. Competenties

Analyseren

Bij deze competentie is het belangrijkste dat er een requirements analyse is gemaakt, aan de hand van de wensen van de opdrachtgever. Hiervoor heb ik in het begin eisen opgesteld, software eisen veranderen gedurende het proces daarom sprak ik elke twee weken met de opdrachtgever Guiseppe Maggiore om de voortgang te bespreken en nieuwe eisen op te stellen. Zo bleek het achteraf toch niet handig te zijn permissiefilters in het diagram te tekenen aangezien dit het diagram groot en onoverzichtelijk maakt. De eisen om aan de standaard van het ER-model te voldoen bleken daarvoor te belangrijk, kijkend naar het gebruik van de tool. Om iets te bouwen wat programmeurs kunnen herkennen. Voor de implementatie van permissiefilters in het diagram moet een eigen notatie ontwikkeld worden en hiermee wordt dan afgeweken van het ER-model. Uiteindelijk is er toch een visuele weergave gemaakt van de permissies en permissiefilters.

De belangrijkste eis voor het eindproduct is een tool die hetzelfde aanvoelt als een IDE, die extra functionaliteit toegevoegd specifiek aan de Scaffolder. Deze competentie heb ik gedurende het hele proces gerealiseerd. Door elke keer de eisen bij te stellen naar de nieuwe wensen van de opdrachtgever. Ook als het gaat om het uitzoeken van bestaande tools en frameworks heb ik voor alles meerdere oplossingen gezocht en uiteindelijk een keuze gemaakt die het meest geschikt was voor het project.

Adviseren

De Scaffolder is nog een redelijk jong software project. Hoppinger gebruikt het nu ruim vijf jaar. De Scaffolder is, zoals elke software, verre van perfect. Het werken aan een tool die nog niet zo oud is geeft meer mogelijkheden om een kritische blik te werpen op de software en mogelijke veranderingen. Dit kan gaan over veranderingen in ontwerp, documentatie of gebruik. Wat meteen al opviel bij de repository van de Scaffolder is dat de documentatie niet up-to-date is. Er missen eigenschappen, dus niet alle functionaliteiten staan hier in. Ook de volgorde van de eigenschappen bevat geen logica, root eigenschappen en model eigenschappen staan door elkaar in één grote tabel. Een concreet advies voor verandering hiervan is door de documentatie te vervangen door een JSON schema die de programmeur kan importeren in zijn eigen IDE. Dan krijgt de developer feedback over alle mogelijke eigenschappen en opties die hij kan gebruiken, met een beschrijving erbij wat het doet. Op het moment dat een developer op zoek is naar een bepaalde functie kan hij door een lijstje scrollen waar de juiste eigenschap in staat. Een ander voorbeeld is het gebruik van een tekenprogramma en framework om het diagram op te tekenen. Hoppinger gaf aan dat een HTML5 canvas een handige oplossing hiervoor zou zijn, dit is een optie die alleen vrij low-level is. Alle figuren van lijnen, tot vierkanten en pijlen moeten apart gedefinieerd worden. Hiervoor heb ik twee frameworks uitgezocht die deze figuren al bevatten, om te kunnen focussen op applicatie logica. Uiteindelijk is de keuze komen te liggen op KonvaJS omdat dit ook goed samenwerkt met React.

Tot slot nog het advies van het gebruik van het gebouwde prototype in het algemeen, wat ik heb geadviseerd voor het gebruik van de tool is om het uitsluitend te gebruiken als hulpmiddel. Om beter inzicht te krijgen in bestaande applicaties en om snel een groot stuk code te produceren in een paar muisklikken.

Ontwerpen

Het eerste ontwerp wat ik heb gemaakt voor de applicatie staat in hoofdstuk 2, deze flowchart heb ik gebruikt gedurende het hele proces. Door de tool te bouwen als een losstaande applicatie creëer je onafhankelijkheid van de Scaffolder. Het resultaat van de tool kan vervolgens worden doorgegeven aan de Scaffolder om een codebase te produceren. Naast dit ontwerp zit er ook een front-end design achter de tool. Hiervoor heb ik eerst een aantal ontwerpen gemaakt voordat ik begon met bouwen.

Realiseren

Onder deze competentie valt het bouwen en beschikbaar stellen van het eindproduct. De tool is gebouwd met Typescript en volgens de principes van functioneel programmeren. Omdat dit een principe is die bij het hele bedrijf wordt toegepast. De keuze voor Typescript was eenvoudig, aangezien de Scaffolder ook in Typescript is gebouwd, dit maakt integratie een stuk eenvoudiger. Functies en code snippets van de Spec zitten in één bestand, dezelfde als het type object van de Spec. De snippets die standaardwaarden bevatten zijn allemaal gebaseerd op het type van de Spec, dit maakt de software onderhoudbaar. Op het moment dat een eigenschap aan de Spec wordt toegevoegd verschijnt er een type error bij het object met de standaardwaarden, dat die functie ook toegevoegd moet worden. Zo verschijnt de optie ook automatisch in het formulier, als verplichtveld of optionele parameter. De software wordt beschikbaar gemaakt in de Scaffolder zelf, door een commando in te tikken wordt de GUI⁹ opgestart in de browser en kan de programmeur beginnen met het maken van de applicatie specificatie.

Manage en Control

Deze competentie omschrijft het waarborgen van kwaliteitseisen van het te bouwen prototype en het verstrekken van informatie over wat al geïmplementeerd is en wat nog gebouwd gaat worden. Bij Hoppinger werken we met scrum, door elke ochtend in de stand-up de voortgang van het project te bespreken waarborg je de kwaliteit van de projecteisen. Er is voor mijn afstuderen een planning opgesteld waarin wordt vermeld wanneer welke competentie behaald is en wanneer welke eis is behaald. Voor dit project zal na mijn afstuderen ook een nieuwe planning worden gemaakt met wat gebouwd gaat worden en hoe de tool verder doorontwikkeld kan worden in de toekomst.

⁹ GUI: Grafische User Interface

12. Nawoord

Toen ik op zoek ging naar een stageplek, zocht ik een bedrijf waar ik de kennis die ik de afgelopen vier jaar heb opgedaan kon toepassen in de praktijk. Hoppinger bleek hiervoor de geschikte plek. Ook omdat hier meerdere docenten van de Hogeschool Rotterdam werkzaam zijn sluiten de technieken die Hoppinger gebruikt naadloos aan bij het curriculum van de Hogeschool.

In de afgelopen vijf maanden heb ik gezien hoe een web development agency zich kan onderscheiden door veel tijd en geld te investeren in innovatie en nieuwe technieken. Dit is het tegenovergestelde van alleen focussen op het bouwen van een werkend eindproduct. Het gaat overigens ook om de kwaliteit van de software en niet alleen hoe het er uit ziet.

Ten slotte, wil ik graag Hoppinger bedanken voor de vrijheid die ik kreeg om aan het project Scaffolder Fabric te werken. Ook na mijn afstuderen zal hier nog verder aan worden gebouwd.

Steven Koerts, Schiedam

13. Literatuurlijst

- [1] Hogeschool Rotterdam. Barld Boot (2019). Verbeter het ontwikkelproces door een configuratie “statisch getypeerd” te maken.
- [2] Typescript Programming language (2020). Advanced types.
<https://www.typescriptlang.org/docs/handbook/advanced-types.html>. Geraadpleegd op: 4 maart 2020
- [3] Raghu Ramakrishnan, Johannes Gehrke (2003) Database management systems third edition. University of Wisconsin.
- [4] Russ Miles & Kim Hamilton (2006). Learning UML 2.0: A pragmatic Introduction to UML First edition. O’Reilly.
- [5] Massachusetts Institute of Technology. (1997). Peter Pin-Shan Chen. The Entity-Relationship Model-Toward a Unified View of Data.
- [6] Louisiana State University (1997) Peter P. Chen. Entity-Relationship Modeling: Historical Events, Future Trends, and Lessons Learned. Computer Science Department.
- [7] Stephen G. Kobourov. (2000). Force-Directed Drawing algorithms. University of Arizona. Chapter 12, page: 383-404.
- [8] University of Woiiiongong. Naeem Sajjad (2013). Force directed graphs: An algorithm for drawing planar graphs.
- [9] Theo Hendriks (2017) Change the script derde druk. Mulder van Meurs Amsterdam.
- [10] KonvaJS (2020) Connected Objects. https://konvajs.org/docs/sandbox/Connected_Objects.html. Geraadpleegd op: 15 april 2020.
- [11] JSON Schema (2020). <http://json-schema.org/> Geraadpleegd op: 7 mei 2020.

14. Bijlagen

In de bijlage vindt u de bewijslast bij de verschillende competenties, code snippets, implementatie van algoritmes en het feedbackformulier ingevuld door de bedrijfsbegeleider.

Bijlage nummer	Competentie(s)
1. Voorbeeld Spec	Analyseren
2. Ontwerp applicatie	Ontwerpen, adviseren
3. Algoritme implementatie	Adviseren, realiseren
4. Test importeren GrandeOmega	Analyseren
5. Testplan	Analyseren
6. Globale planning	Manage & control
7. Beoordeling begeleider	Manage & control, adviseren

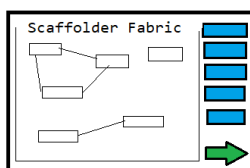
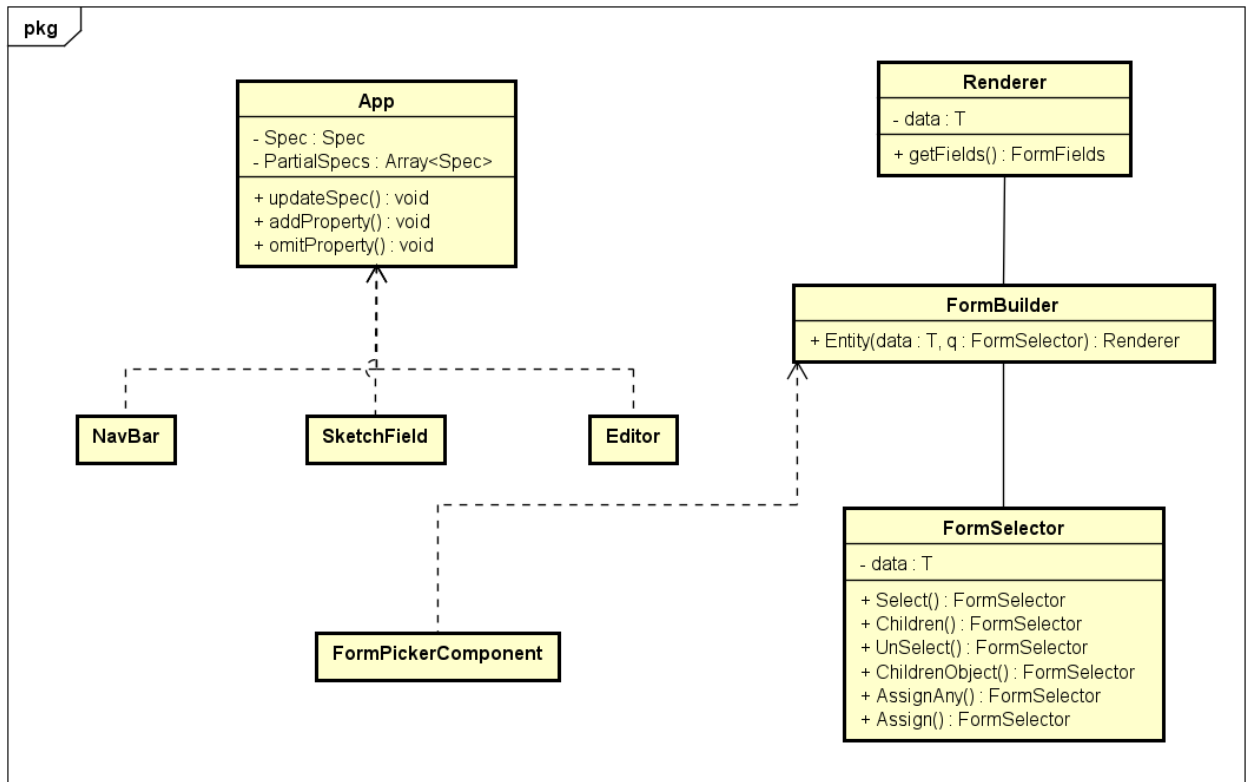
14.1. Voorbeeld Spec

Dit is een specificatie voor de GrandeOmega website. Deze Spec is gebruikt om de applicatie te testen op schaalbaarheid.

```
{
  "namespace": "GrandeOmega",
  "context_name": "GrandeOmegaContext",
  "api_version": "v1",
  "skip_forced_id_increment": false,
  "url_prefix": "admin",
  "database_provider": "postgresql",
  "custom_view_imports": "import * as CustomViews from '../custom_views'",
  "languages": ["en", "nl"],
  "custom_restriction_imports": "",
  "include": [
    "spec_teaching_context",
    "spec_user",
    "spec_activity",
    "spec_exam",
    "spec_site",
    "spec_blog"
  ],
  "dotnet_version": "dotnet2",
  "models": [
    {
      "name": "Application",
      "type": "homepage",
      "allow_maximisation": true,
      "attributes": [
        { "type": "string", "name": "Name" },
        { "type": "string", "name": "Description" }
      ],
      "permissions": {
        "view": ["*"],
        "edit": ["Admin"],
        "delete": [],
        "create": []
      },
      "seeds": [
        { "Id": 1, "Name": "\"GrandeOmega\"", "Description": "\"The ultimate interactive CS teaching tool.\"" }
      ]
    },
    {
      "name": "Reset",
      "allow_maximisation": true,
      "attributes": [
        { "hidden_when_minimised": true,
          "type": "null", "name": "ResetAllSchools",
          "custom_rendering": "CustomViews.ResetAllSchools" }
      ],
      "seeds": [
        { "Id": 1 }
      ],
      "permissions": {
        "view": ["Admin"],
        "edit": ["Admin"],
        "delete": ["Admin"],
        "create": ["Admin"]
      }
    }
  ]
}
```

14.2. Ontwerp applicatie

Hier staat een ontwerp van de structuur van de applicatie. Aangezien dit project geen backend heeft staan hier uitsluitend de hoofdcomponenten van de applicatie. De namen komen overeen met React componenten. De hoofdcomponent is de App, hierin staat de state van de hele applicatie, de belangrijkste informatie is de Spec. Deze is opgeslagen als object. Alle component die daar aan vast zitten hebben interactie met de Spec. De componenten SketchField, NavBar en Editor roepen één voor één de methodes van de App component aan om de Spec te bewerken. De FormBuilder wordt gebruikt in meerdere componenten en is gescheiden in drie objecten. De FormPickerComponent is een React component die het juiste invoerveld op het scherm zet.

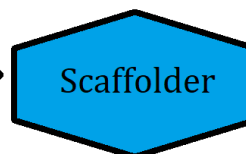


Scaffolder Fabric is het gebouwde prototype. In de interface kunnen diagrammen worden gemaakt en via een formulier bepaalde eigenschappen worden toegevoegd of gewijzigd.



{JSON}

De output van Scaffolder Fabric is een JSON bestand die de hele specificatie van een webapplicatie bevat.



De gegenereerde JSON wordt vervolgens meegegeven aan de Scaffolder van Hoppering.



De Scaffolder geeft als een output een code base, die bestaat uit een webapplicatie met een dotnet backend en een React frontend. Vanaf hier kan de developer verder bouwen.

14.3. Algoritme implementatie

Hieronder staat de implementatie van het Fruchterman and Reingold layouting algoritme. De input van de functie is een adjacency matrix en als output geeft de functie een nieuwe lay-out terug. De lay-out heeft de vorm van een lijst aan vectoren, die de x- en y-coördinaten van alle nodes bevatten. Verder is het algoritme te tunen door de hoogte en breedte van het tekenoppervlak te veranderen, hoe groter het oppervlak hoe meer nodes je kwijt kunt en des te minder lijnen elkaar snijden. Het aantal iteraties geeft aan hoelang het algoritme erover mag doen om tot de juiste oplossing te komen voor een ideale lay-out en de edge lengte geeft aan hoever de nodes maximaal uit elkaar mogen staan. Indien niet gedefinieerd wordt dit berekend aan de hand van het tekenoppervlak. Als het diagram niet al te groot is ziet het er beter uit om een edge lengte van 50 pixels mee te geven.

```
// fruchtermanReingold algorithm
let fruchtermanReingold = (G: AdjacencyMatrix, W = 1000, H = 1000, iterations = 50, edge_length?: number): GraphLayout => {
  let area = W * H
  let edges = getEdges.f(G)
  let k = edge_length == undefined ? Math.sqrt(area / G.length) : edge_length //maximum distance of the nodes
  let fa = (x: number): number => x ** 2 / k // Formula to calculate attractive forces
  let fr = (x: number): number => k ** 2 / x // Formula to calculate repulsive forces

  // initial positions
  let positions = createLayout(W, H).f(G)
  let displacements = G.map(_ => Vector2D(0, 0))

  let t = W / 10
  let dt = t / (iterations + 1)

  //console.log(`area: ${area}`)
  //console.log(`k: ${k}`)
  //console.log(`t: ${t}, dt: ${dt}`)

  for (let i = 1; i <= iterations; i++) {
    //console.log(`Iteration: ${i}`)

    // Calculate repulsive forces
    G.forEach((v, indexV) => {
      displacements[indexV] = Vector2D(0, 0)
      v.forEach((u, indexU) => { // It doesn't matter if you iterate over v or G
        if (indexU !== indexV) {
          let delta = positions[indexV].Min(positions[indexU])
          if (delta.length !== 0) {
            displacements[indexV] = displacements[indexV].Plus(delta.divide(delta.length).times(fr(delta.length)))
          }
        }
      })
    })

    // Calculate attractive forces
    edges.forEach(edge => {
      let delta = positions[edge.to].Min(positions[edge.from])
      if (delta.length !== 0) {
        displacements[edge.to] = displacements[edge.to].Min(delta.divide(delta.length).times(fa(delta.length)))
        displacements[edge.from] = displacements[edge.from].Plus(delta.divide(delta.length).times(fa(delta.length)))
      }
    })

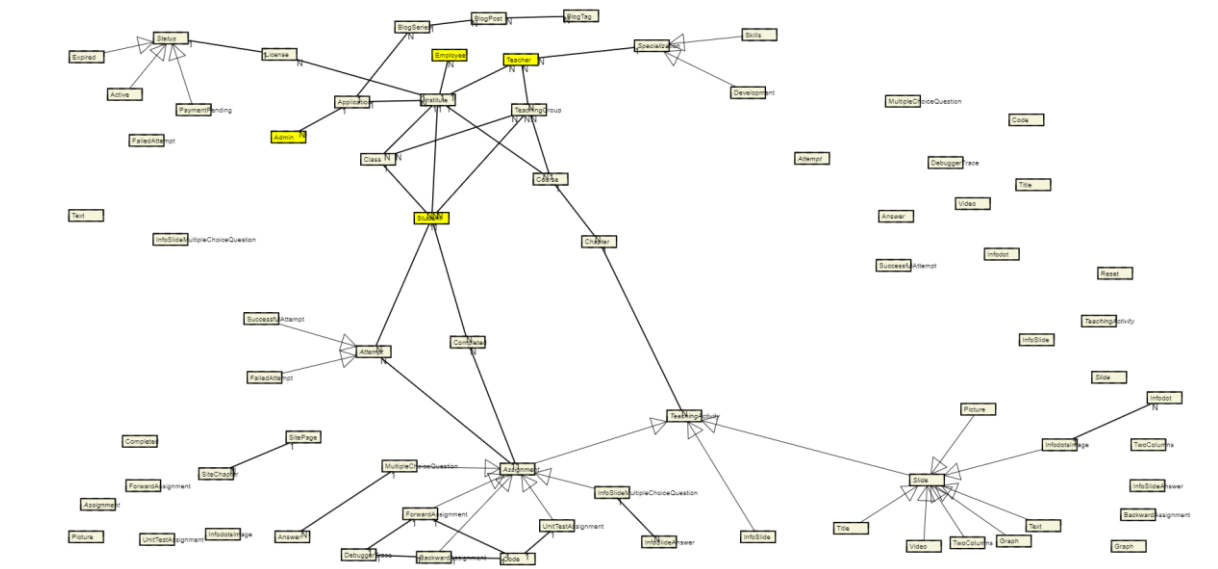
    // limit max displacement
    G.forEach((node, index) => {
      if (displacements[index].length !== 0) {
        positions[index] = positions[index].Plus(displacements[index].divide(displacements[index].length).times(Math.min(displacements.length, 1)))
        positions[index].x = Math.min(W / 2, Math.max(-W / 2, positions[index].x))
        positions[index].y = Math.min(H / 2, Math.max(-H / 2, positions[index].y))
      }
    })

    // reduce the temperature as the layout approaches a better configuration
    t -= dt
  }

  //console.log('Done ... ')

  // Still some nodes appear outside of the screen, so just move the position by fixed number
  return positions.map(vector => vector.Plus(Vector2D(200, 200)))
}
```

In onderstaande afbeelding is het ER-model te zien van de GrandeOmega website, deze specificatie is ook gebruikt om de applicatie te testen. De gele entiteiten zijn zogeheten loggable entities, dat houdt in dat dit rollen zijn in het systeem die kunnen inloggen. Zoals: Admin, Student, Teacher en Employee. Deze rollen hebben ook ieder zijn eigen permissies om andere entiteiten toe te voegen, aan te maken, te bewerken, te verwijderen of te bekijken.



14.5. Testplan

Inleiding

Het testen van Scaffolder Fabric gaat iets anders dan bij andere softwareproducten. Normaliter zou je alle functies van de software testen met bepaalde input en de werkelijke output vergelijken met de verwachte output. In dit geval is dat niet de meest handige manier van testen, aangezien de applicatie zelf een input genereert voor de Scaffolder en de Scaffolder een output geeft in de vorm van een codebase. Er zijn geen frameworks die kunnen controleren of de gegenereerde Spec correct is, de enige die dat kan controleren is de Scaffolder zelf.

Testmethode

Er zal getest worden volgens twee methode:

- De eerste is een integratie test met de Scaffolder, is het mogelijk om de gegenereerde Spec uit te voeren op Scaffolder en kan je een bestaande Spec importeren in Scaffolder Fabric.
- De tweede techniek is een geautomatiseerde testtechniek, om de user interface te testen. Met behulp van CypressJS kun je automatisch interacties op het scherm laten uitvoeren. Als een interactie in de toekomst verandert zal de test een foutmelding geven. Deze tests zijn nuttig om duidelijk te maken aan andere developers hoe een bepaalde interactie of handling origineel bedoeld is.¹⁰

Uitvoering & resultaten

Tijdens het uitvoeren van de integratie test bleek dat er verkeerde aannames waren gemaakt over de verplichte eigenschappen van de Spec. In de applicatie was de aanname gebaseerd op de type declaratie van de Spec in de repository van de Scaffolder. Zo zijn eigenschappen als `project_name` en `searchable_by` niet verplicht in de Scaffolder, maar wel in de type declaratie. Verder bleek dat gedurende verschillende versies van de Scaffolder het datatype van een eigenschap wel eens wil veranderen. Zo is `presentation_structure` veranderd van een object naar een array van objecten. Het is belangrijk om met dit soort dingen rekening te houden, voor backwards compatibility.

```
1 describe('Generate Spec', () => {
2   it('Test the interface', () => {
3     cy.visit('http://localhost:3000/')
4     cy.contains('Menu').click()
5
6     cy.contains('Download').click()
7     cy.contains('Load').click()
8     cy.contains('Save').click()
9   })
10 })
11 })
```

Voorbeeld Cypress test automatisch door de interface klikken

Conclusie

Uit de eerste test is gebleken dat er een aantal verkeerde aannames waren gemaakt voor de vorm en verplichte aanwezigheid van bepaalde attributen. Hiervoor zal het JSON-schema worden aangepast om te zorgen dat de tool voor alle Spec's toepasbaar is. Uit de tweede test ging het voornamelijk om de usability. Het ging voornamelijk over hoe makkelijk het is om bepaalde opties te vinden en Spec's te analyseren. Hieruit bleek dat er extra invoervelden en klik mogelijkheden moesten komen om de gebruiker beter wegwijs te maken in het systeem.

¹⁰ CypressJS: <https://www.cypress.io/>

14.6. Globale planning

Onderstaande tabel bevat een globale planning die aan het begin van het onderzoek is opgesteld, elke twee weken staat een voortgangsgesprek met Guiseppe Magiore en code review met Francesco Di Giacommo gepland.

Vorbereiding (OP2 8 januari t/m 8 februari)

Schoolweek	Wat	Status afstuderen
Week 6	Start met werken bij Hoppinger	Afstudeervoorstel goedgekeurd
Week 7	Experimenteren met de Scaffolder en lezen documentatie	
Week 8	Opstart voorbeeld code	
Week 9	Starten project, React app	Start plan van aanpak
Week 10	Verder bouwen aan eerste concept.	Goedgekeurd plan van aanpak

Afstudeertraject (OP3, OP4 10 februari t/m 19 juni)

Schoolweek	Wat	Status afstuderen
Week 1	Start fulltime bij Hoppinger	Start schrijven scriptie, inleiding
Week 2		
Week 3		
Week 4	Bedrijfsbezoek begeleider	
Week 5		
Week 6		
Week 7		
Week 8		
Week 9	Mijlpaalsessie	Eerste versie scriptie af
Week 10	Minimale eisen af	
Week 11	Revisie product, verbeterpunten minimale eisen. Extra features	
Week 12		
Week 13		
Week 14		Concept scriptie af
Week 15		
Week 16		Definitieve scriptie af
Week 17		Presentatie af
Week 18	Concept prototype af	Scriptie af
Week 19		(Afstudeerzitting)
Week 20		(Afstudeerzitting)

14.7. Beoordelingsformulier bedrijfsbegeleider

**VI. Bijlage Observatie van bedrijfsbegeleider over de afstudeerder**

Naam student:	Steven Koerts
Studentnummer:	0904861
Opleiding:	Informatica
Titel eindverslag:	Scaffolder: Het visualiseren van applicaties

Inleiding
 Met dit formulier wordt naar uw mening gevraagd over het functioneren van de afstudeerstudent. Hierbij komen aan bod:

- Personal skills
- De vijf ICT- competenties
- Een vrij veld waar u zelf aanvullende observaties kan vermelden

Het formulier hoeft niet door één persoon ingevuld te worden: meningen van opdrachtgever, begeleider, projectleider en collega's kunnen meegenomen worden (**eventueel in bijlagen**). Uiteindelijk moet het formulier door de begeleider ondertekend worden. Per onderdeel wordt gevraagd de sterke punten en de verbeterpunten aan te geven. Wees a.u.b. zo **expliciet** mogelijk, zodat een goed beeld over het functioneren van de student gevormd kan worden.

Professional skills
 Aandachtspunten hierbij zijn: communicatie en samenwerking.

Sterke punten
Steven was really punctual with his communication: we had a number of meetings where he explained clearly the progress and the problems that were arising during the development of his application.

Verbeterpunten

Manage & control
 Aandachtspunten hierbij zijn: planmatig werken, projectvoortgang bewaken, principes toepassen om een softwareontwikkelp proces te managen en te bewaken.

Sterke punten
The planning was most of the times respected and the communication about the progress was clear.

Verbeterpunten
 (minor issue) Sometimes Steven tends to be a little dispersive (but always in the boundaries of the project scope) and to prioritize low-priority features over high-priority ones. With some course correction, he managed to go back on track.

<p>Analyseren</p> <p>Aandachtspunten hierbij zijn: doelgerichtheid, keuze van de juiste deelvragen die beantwoord moeten worden om tot een goed beroepstechnische probleemstelling te komen, diepgang van de analyses, kritisch gebruik bronnen, helder vaststellen van de problemen.</p> <p><u>Sterke punten</u></p> <p>Steven did not exitate to dive into compex algorithmical solutions to improve the application with well-documented references from literature (i.e. the graph displacement algorithm to re-arrange the visualization of the model).</p> <p><u>Verbeterpunten</u></p>
<p>Adviseren</p> <p>Aandachtspunten hierbij zijn: gemaakte keuzes en onderbouwing hiervan, volledigheid (bijvoorbeeld kwaliteitsaspecten, security, schaalbaarheid, performance, privacy), presentatie van de adviezen.</p> <p><u>Sterke punten</u></p> <p>Steven managed to identify performance issues related to the rendering, benchmark the application, and find a solution to overcome them.</p> <p><u>Verbeterpunten</u></p>
<p>Ontwerpen</p> <p>Aandachtspunten hierbij zijn: keuze van relevante ontwerptechnieken (FO, TO, UI, DB), kwaliteit van de ontwerpen, architectuur, teststrategie, toetsing van de ontwerpen (prototyping, voorleggen aan experts).</p> <p><u>Sterke punten</u></p> <p>Steven managed to design the application and, with some input, to improve the initial design to accomodate the requirements of the application over time.</p> <p><u>Verbeterpunten</u></p>

Realiseren

Aandachtspunten hierbij zijn: kwaliteit van het beroepstechnische en kwaliteit van het ontwikkelproces. Clean code, documenteren, testen, integreren, continious integration & deployment, gebruik frameworks.

Sterke punten

Steven proved to apply state-of-the-art development techniques in the complex domain of React framework and canvas.

Verbeterpunten**Aanvullende observaties**

Handtekening/signature:

The image shows two handwritten signatures. The top signature is in blue ink and appears to be 'Steven Koerts' written in a cursive style. The bottom signature is in black ink and is the name 'Steven' written in a bold, slightly stylized font. Both signatures are written over a horizontal dashed line.

