Technische verslag

Het CMS voor iedereen

Dante Klijn

Samenvatting

Hier komt de nieuwe samenvatting

Contactgegevens

Student

Naam	Dante Klijn
Studentnummer	4565908
Academisch jaar	2023/2024
E-mail	dante.klijn@student.nhlstenden.com
Telefoonnummer	+31 (0)6 24 76 59 74

Onderwijsinstelling

Naam	NHL Stenden University of Applied Sciences
Course	HBO-ICT
Locatie	Rengerslaan 8-10, 8917 DD, Leeuwarden
Telefoonnummer	+31 (0)88 991 7000

Docentbegeleider

Naam	Stefan Rolink
Email	stefan.rolink@nhlstenden.com
Telefoonnummer	+31 (0)6 42 28 30 77

Afstudeercommissie

Email	afstuderenschoolofict@nhlstenden.com

Examencommissie

Email	examencommissiehboict@nhlstenden.com

Organisatie

Naam	Snakeware New Media B.V.
Locatie	Veemarktplein 1, 8601 DA, Sneek
Telefoonnummer	+31 (0)515 431 895

Bedrijfsbegeleider

Naam	Thom Koenders
Email	thom@snakeware.com
Telefoonnummer	+31 (0)6 13 09 18 51
Rol	Senior software developer

Versiebeheer

Versie	Datum	Veranderingen
0.1	TBD	Eerste hoofdstukken

Woordenlijst

beheerder Een beheerder is een klant van Snakeware die een website onderhoud via het CMS. Een voorbeeld hiervan is de klant Poiesz.

Contentmanagementsysteem Een contentmanagementsysteem is een softwaretoepassing, meestal een webapplicatie, die het mogelijk maakt dat mensen eenvoudig, zonder veel technische kennis, documenten en gegevens op internet kunnen publiceren (contentmanagement). Als afkorting wordt ook wel CMS gebruikt.

Eindgebruiker Een eindgebruiker is de bezoeker van een website die onderhouden wordt door een beheerder. Het zijn de klanten van de beheerder, een voorbeeld hiervan zijn de klanten van Poiesz supermarkten.

Graphical user interface Een graphical user interface (GUI), is een manier van interacteren met een computer waarbij grafische beelden, widgets en tekst gebruikt worden.

SDK Een Software Development Kit (SDK), is een verzameling hulpmiddelen, libaries die wordt verstrekt door softwareontwikkelaars om andere ontwikkelaars te helpen bij het maken van softwaretoepassingen voor een specifiek platform, framework, programmeertaal of service.

Search engine optimization Search Engine Optimisation (SEO), zijn alle processen en verbeteringen die als doel hebben een website hoger in Google te laten verschijnen.

Software development life cycle De software development life cycle (SDLC) is een procesmatige manier van werken met als doel goede kwaliteit software te produceren met lage kosten in een korte tijd. De SDLC bestaat uit 5 fases: *Requirements analysis*, *Design*, *Implementation*, *Testing*, en *Evolution* (Zuci Systems, g.d.).

Lijst van figuren

Inhoudsopgave

Hoofdstuk 1

Inleiding

Dit is het technische verslag voor het "Het CMS voor iedereen" project. het verslag is een onderdeel van de afstudeerperiode binnen NHL Stenden Hogeschool. Dit document dient als verantwoording voor de gebruikte processen en het eindproduct dat gerealiseerd is in de afstudeerperiode. In de volgende sectie wordt de organisatiestructuur beschreven, samen met de aanleiding en de context van de afstudeeropdracht.

1.1 Organisatieomschrijving

Snakeware New Media B.V. (Snakeware) is een E-business bureau gevestigd in Sneek. Haar aangeboden diensten omvatten het adviseren, bouwen en onderhouden van digitale producties, met een focus op websites, webshops en mobiele apps (Snakeware, 2022b). Op het moment van schrijven telt Snakeware meer dan 60 werknemers, elk met verschillende specialiteiten. Ze leveren services aan organisaties zoals DPG Media, DekaMarkt en Poiesz supermarkten (Snakeware, 2022a).

1.2 Context

Snakeware heeft een platform genaamd "Snakeware Cloud" dit is een contentmanagementsysteem (CMS) waarmee digitale content kunnen voorzien voor haar (grotere) klanten. Snakeware Cloud is een applicatie waarmee Snakeware en haar klanten webapplicaties kan inrichten en voorzien van content.

De klant van Snakeware kan zijn of haar website zelf inrichten door middel van het specificeren van de content op de verschillende pagina's. Dit wordt gedaan door middel van artikelen die door het CMS gebruikt kunnen worden. De content van het artikel kan verschillen tussen simpele tekst, vragenlijst, webshop producten, etc. Hiernaast zijn er ook search engine optimization (SEO) opties binnen Snakeware Cloud om de site goed te kunnen vinden op het internet. Hierbij zijn er opties zoals de mogelijkheid om de title tags en zoekwoorden toe te kunnen voegen in de head (Mozilla, 2023c)

Hierom heeft Snakeware Cloud veel features en configuratie stappen wat het complex en duur maakt om een relatief kleine webapplicatie te maken voor kleinere klanten. Dit zorgt ervoor dat Snakeware zich niet kan vestigen in een markt met veel kleinere klanten, en hierdoor omzet misloopt.

1.3 Aanleiding

Het huidige platform is 21 jaar oud en er is veel functionaliteit in de loop der jaren aan toegevoegd. Omdat Snakeware Cloud een oud platform is zijn er veel technieken en best practices gebruikt die nu niet meer als optimaal worden beschouwd. Deze technieken waren erg geïntegreerd in Snakeware Cloud en er is in het verleden gekozen om niet de code te herschrijven om het aan de huidige standaarden te voldoen van andere projecten. Een voorbeeld hiervan is het plaatsen van afkortingen voor de tabelnaam als prefix bij elke kolom, of gigantische C# (Microsoft, 2022) files van 10 000 regels met verschillende functies. Deze functies houden zich niet aan de *Single Responsibility Principle* van de SOLID ontwerpmethode (Watts, 2020) wat het moeilijk maakt om het huidige CMS te onderhouden.

Ook zijn er technieken toegepast die nu niet meer relevant zijn. Een voorbeeld hiervan is dat het CMS gebruikmaakt van JavaScript (Mozilla, 2023b) en toen ze er mee begonnen bestonden JavaScript classes (Mozilla, 2023a) nog niet, dus hebben ze die zelf geïmplementeerd. Deze oudere technieken en standaarden zorgen ervoor dat het meer tijd kost om het CMS te onderhouden vanwege de extra code. Dit zorgt ervoor dat het meer tijd en geld kost om het Snakeware Cloud uit te breiden.

Een van de voornaamste uitdaging met Snakeware Cloud betreft de verouderde datastructuur van de applicatie. Deze veroudering is het gevolg van een initïele ontwikkeling waarbij onvoldoende rekening werd gehouden met toekomstige functionaliteitsuitbreidingen in het systeem. Als gevolg daarvan is de onderliggende datastructuur niet aangepast, maar zijn er elementen aan toegevoegd. Dit heeft geresulteerd in database query's van duizenden regels en complexe relaties tussen tabellen in de database. Dit huidige scenario bemoeilijkt aanzienlijk het toevoegen van nieuwe functionaliteiten, wat resulteert in aanzienlijke tijd en kosten investeringen.

Hierom wil Snakeware een nieuw systeem met een nieuwe datastructuur. Door het gebruiken van een nieuwe softwarearchitectuur zouden er velen problemen opgelost kunnen worden die nu voor komen. Omdat er een nieuwe datastructuur moet komen en de logica van het oude systeem nauw verbonden is met de datastructuur is het niet mogelijk om de oude code opnieuw te gebruiken.

1.4 Opdrachtomschrijving

De opdracht is om een proof of concept CMS-API te ontwikkelen die gebruikt maakt van een datamodel en systeemarchitectuur dat flexibeler, onderhoudbaarder is en gebruik maakt van moderne best practices. Tijdens de afstudeeropdracht wordt er primair op het datamodel en de systeemarchitectuur gefocust. Omdat er nog geen concreet datamodel en systeemarchitectuur is zal dit onderzocht en ontworpen moeten worden.

De opdracht omvat het achterhalen van de requirements, ontwerpen en ontwikkelen van het proof of concept met als focus een nieuw datamodel, met de essentiële functionaliteiten.

Het huidige Snakeware Cloud platform bestaat uit 2 verschillende graphical user interfaces (GUI):

- Snakeware Cloud GUI
- De beheerder zijn webapplicatie

Met de Snakeware Cloud GUI kan de beheerder de content van de website aanpassen. Door middel van de webapplicatie van de beheerder kan de eindgebruiker de content bekijken en er mee interacteren. Er is voor gekozen om niet de Snakeware Cloud GUI te realiseren om de afstudeeropdracht in scope te houden. Er is wel voor gekozen om de beheerder zijn webapplicatie in zijn minimale vorm uit te werken.

Het doel van het proof of concept is dat er aangetoond kan worden dat door het gebruiken van een nieuw datamodel en systeemarchitectuur ook services verleend kunnen worden aan kleinere klanten. Dit zou eventueel ook een startpunt zijn om op verder te bouwen.

1.5 Leeswijzer

als laatste

Hoofdstuk 2

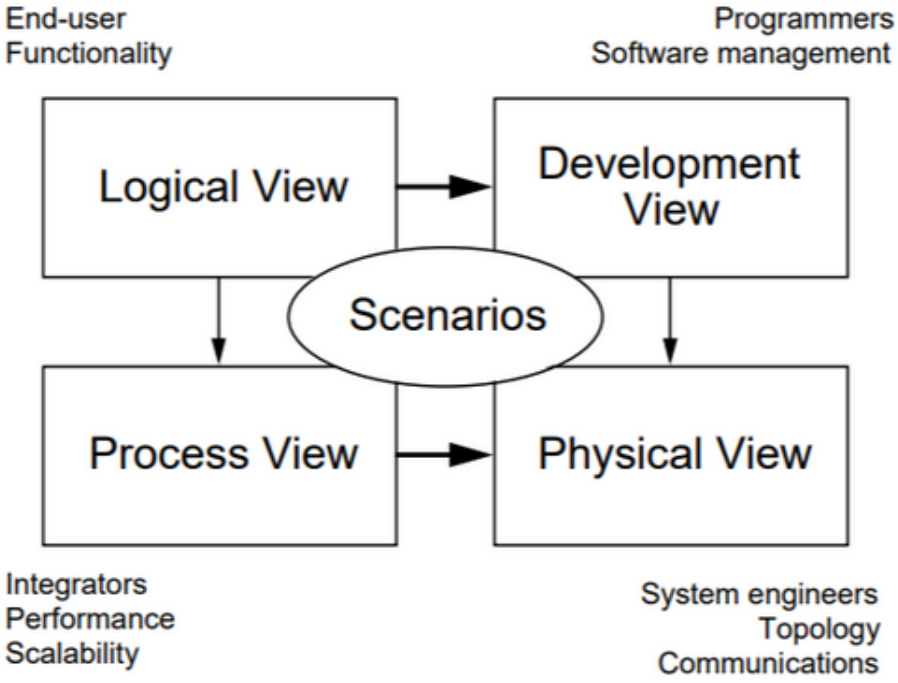
Ontwerp

In dit hoofdstuk wordt het ontwerp van de softwareproducten beschreven. Tijdens de eerste fase van de afstudeerperiode is er een onderzoek gedaan naar de requirements van het systeem (Klijn, 2023). Het resultaat van dit onderzoek is een lijst van geprioriteerde requirements en randvoorwaarden waar het systeem aan moet voldoen. Deze resultaten zijn gebruikt om het softwareproduct te ontwerpen.

Voor het ontwerpen van het systeem is er gebruikgemaakt van een ontwerp framework genaamd het 4 + 1 view model (Kruchten, 1995). Er is voor het 4 + 1 view model gekozen omdat dit een bekende methode is om het software ontwerp in beeld te krijgen. Dit helpt met het vastleggen van technische specificaties en keuzes van het ontwerp.

Het 4 + 1 view model maakt gebruik van 5 verschillende perspectieven om de software inbeeldt te krijgen. Deze perspectieven zijn scenarios, logical, process, development en physical view (visualisatie is te zien in figuur 2.1). In de volgende secties worden de perspectieven uitgelegd en ingevuld.

Figuur 2.1: 4 + 1 Model view model (Kruchten, 1995)

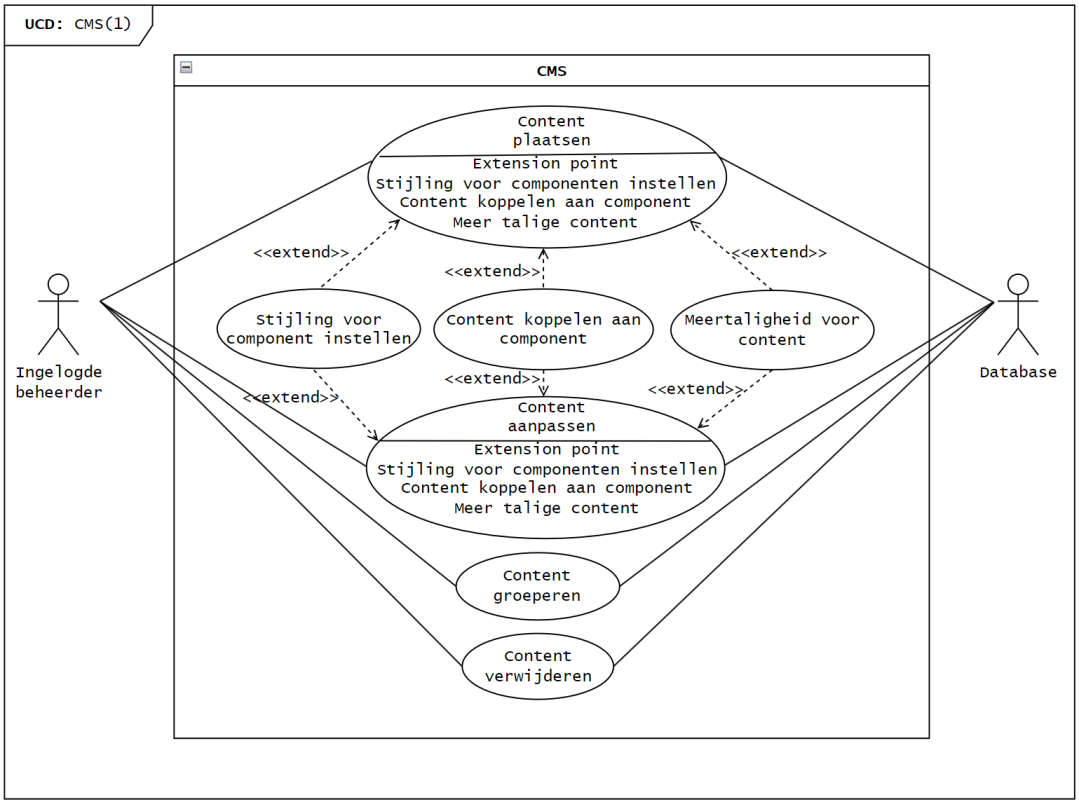


2.1 Scenario's

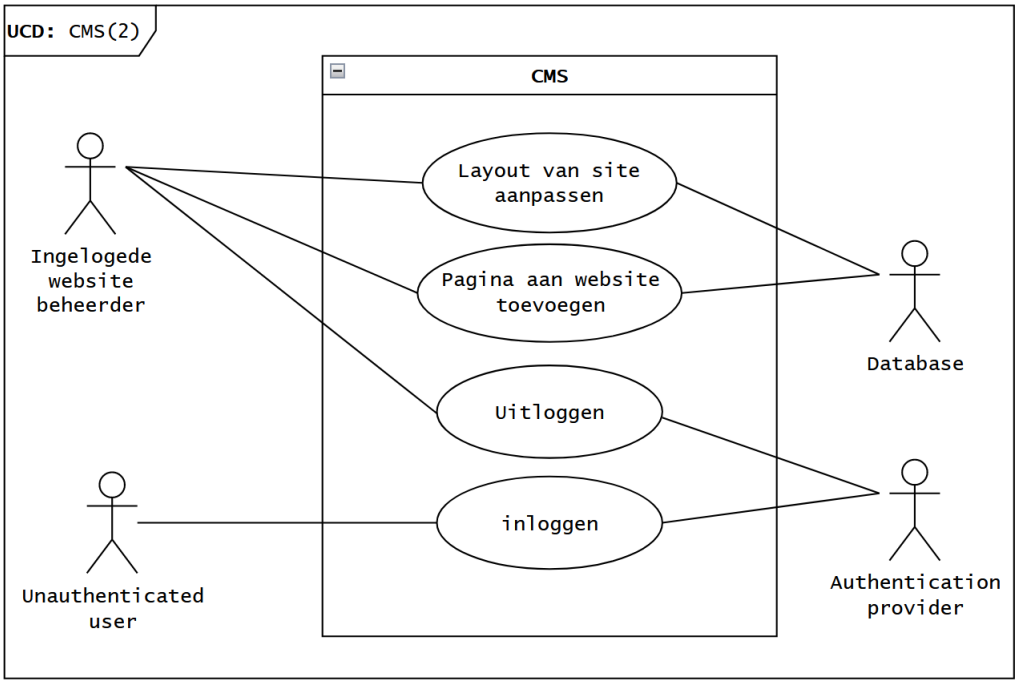
De scenario view is een representatie van de belangrijkste use cases van het systeem (Kruchten, 1995). De use cases zijn opgesteld door middel van de geprioriteerde lijst van requirements van het onderzoek (Klijn, 2023). Om de verschillende use cases en interactie met andere actoren in het systeem in beeld te krijgen wordt er gebruik gemaakt van een use case diagram (lucidchart, 2023). In figuur 2.2, 2.3 en 2.4 zijn de verschillende use case diagramen te zien.

In figuur 2.2 zijn de verschillende use cases te zien die te maken hebben met de content binnen het CMS. De overige functionaliteiten van het CMS-platform zijn te vinden in figuur 2.3 Dit use case diagram toont het authenticatie en algemene site functionaliteit. Het laatste diagram (figuur 2.4) is gemaakt voor de beheerder zijn site. Dit is de locatie waar de (website) bezoeker de content kan bekijken.

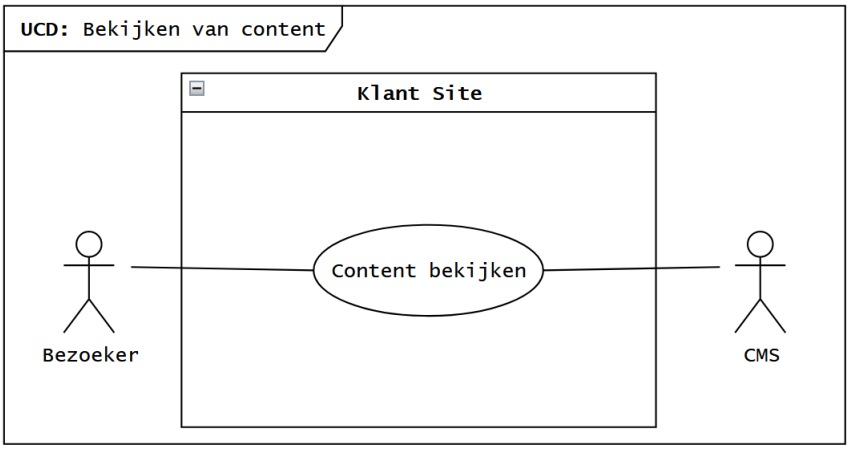
Figuur 2.2: Use case diagram CMS (1)



Figuur 2.3: Use case diagram CMS (2)



Figuur 2.4: Use case diagram beheerder site



2.2 Logical View

In de volgende sectie wordt de logical view van het 4 + 1 model toegelicht. Het doel van de logical view is de functionaliteiten van het systeem in beeld te brengen (Kruchten, 1995). Dit wordt gedaan door op een abstract niveau naar de structuur en datamodel van het systeem te kijken zonder implementatie details. De volgende sub secties gaan dieper in op het datamodel en de softwarearchitectuur van het systeem.

2.2.1 Datamodel

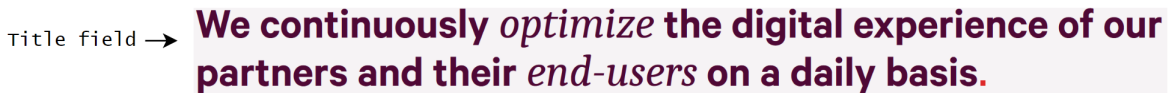
Een van de belangrijkste doelen van de afstudeeropdracht is om een nieuw datamodel te maken voor een CMS-systeem. Dit datamodel moet veel verschillende soorten datastructuren kunnen ondersteunen en deze kunnen presenteren op een beheerder zijn website. Het huidige CMS van Snakeware doet dit door middel van een complexe structuur. Deze structuur zorgt ervoor dat het moeilijk is om nieuwe functionaliteit toe te voegen en dat het systeem lastig is te onderhouden.

Daarom heeft het nieuwe datamodel twee belangrijke uitgangspunten om de pijnpunten van het oude CMS te voorkomen. Het datamodel moet generiek blijven, zodat er veel verschillende datastructuren in opgeslagen kunnen worden. Door het datamodel generiek en flexibel te houden hoeft het systeem niet uitgebreid te worden om een nieuwe datastructuren te ondersteunen. Verder moet de structuur van het datamodel simpel blijven, zodat het makkelijk te onderhouden is. Om deze doelen te bereiken is er gebruikgemaakt van de volgende concepten.

Fields

Content op websites bestaan uit kleinere stukken data/content. Fields representeren de kleinste laag content op een website. Hierbij kan gedacht worden aan een stukje tekst of een prijs op een pagina. Om bij fields een beter beeld te schetsen is er een stuk van de Snakeware website (snakeware.com) gebruikt als voorbeeld. In figuur 2.5 wordt weergegeven waar de fields zich bevinden in de content.

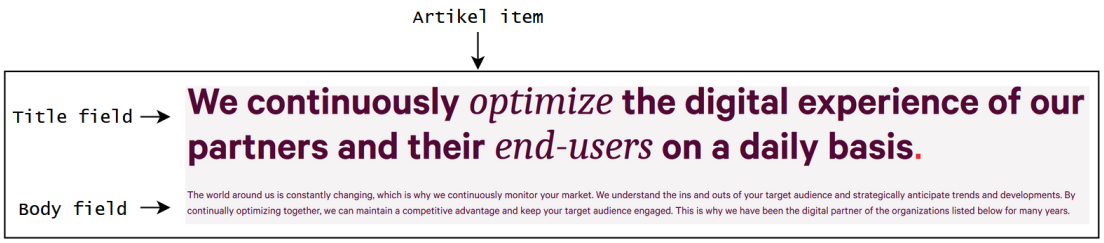
Figuur 2.5: Visualisatie van fields



Items

Om een samenhangend stuk content op te stellen wordt er gebruikgemaakt van items. Een item is een container dat meerdere fields kan bevatten. Om hier een beter beeld bij te schetsen is hetzelfde voorbeeld gebruikt voor de items (zie figuur 2.6).

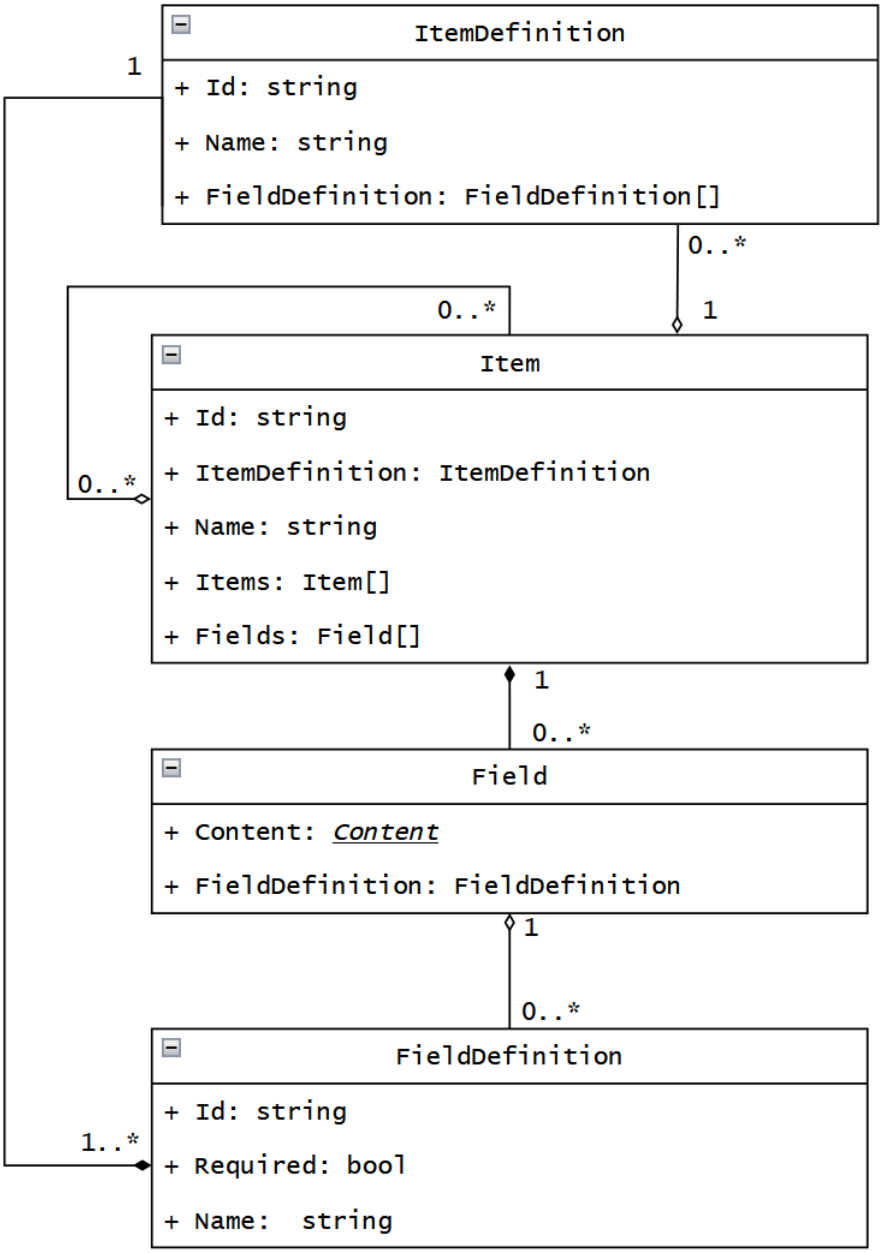
Figuur 2.6: Visualisatie van een item



In het voorbeeld (figuur 2.6) is te zien dat het item “Artikel” 2 fields heeft. Deze fields zijn “Title” en “Body” en bevatten de teksten voor de gedefinieerde gedeeltes van het item. Een ander functionaliteit van een item is dat het meerdere items kan bevatten. Door dit te doen is het mogelijk om complexe structuren op te bouwen.

Om ervoor te zorgen dat items hergebruikt kunnen worden, wordt er gebruikgemaakt van templating. Hierom zijn er item en field definities gemaakt die aangeven welke fields verplicht zijn op een item en welke optioneel. In het ontwerp worden deze definities ItemDefinition en FieldDefinition genoemd. Een klassendiagram met het geïntegreerde templating systeem is te zien in figuur 2.7. Het datatype “Content” bevat verschillende primaire types zoals integer, string, boolean ect.

Figuur 2.7: Klassendiagram datastructuur afstudeeropdracht



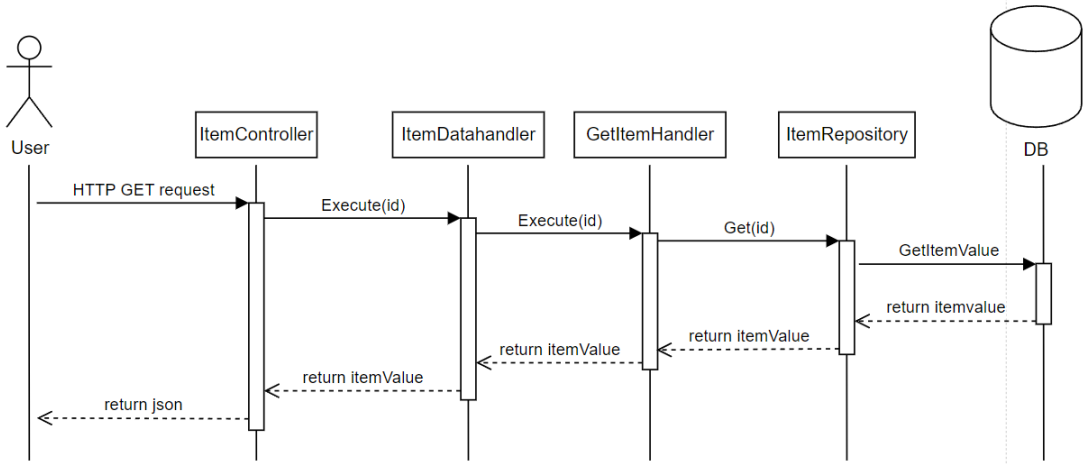
2.2.2 Software architectuur

Om in het afstudeerproject niet dezelfde fouten te maken als het huidige CMS-systeem is er ook nagedacht over de development principes waar de code aan moet voldoen. Hiervoor is er gekeken hoe de architectuur zich houdt aan de verschillende SOLID principes (Watts, 2020). SOLID is een acroniem dat 5 verschillende principes inhoudt. Deze principes zorgen ervoor dat de code beter te onderhoudbaar is en makkelijker te begrijpen is. SOLID bestaat uit de volgende principes:

1. **Single-responsibility principle** Dit betekent dat een class/module, maar 1 verantwoordelijkheid mag hebben. Als een class/module te veel verantwoordelijkheden heeft dan wordt het moeilijker om de code te begrijpen en aan te passen.
2. **Open-closed principle** Dit houdt in als class of functie of een andere software entiteit uitgebreid moet worden dat het wordt gedaan door middel van een extensie(open) in plaats van modificatie(closed). Hierdoor hou je oude code intact en heb je geen risico dat je bestaande code stuk gaat vanwege de nieuwe functionaliteit die geschreven is.
3. **Liskov substitution principle** Een class die afgeleid is van een super class zou moeten vervangen kunnen worden van een andere afgeleiden variant van die superclass Dit moet gedaan kunnen worden zonder de validiteit (corectness) van het programma te beïnvloeden. Door het gebruik van het Liskov substitution principle verhoog je de consistentie en de verwachte uitkomst van het programma.
4. **Interface segregation** Een interface moet alleen de methodes geven die nodig zijn voor de client. Geen client moet geforceerd zijn om methodes te implementeren waar die geen gebruik van maakt. Dit kan verminderd worden door meerdere kleinere interfaces te maken in plaats van een grote interface. Deze interfaces behandelen specifiekere use-cases in plaats van gegeneraliseerde use-cases.
5. **Dependency inversion** Een class of module zou niet moeten afhangen van implementaties maar van abstracties. Hierdoor de koppelingen van de modules/classes verminderd, en verhoog je de code onder houdbaarheid. Om aan dit principe te voldoen wordt er vaak gebruikgemaakt van dependency injection.

Om de verschillende aspecten van SOLID te implementeren is er gekozen om gebruik te maken van handlers. Hierbij heeft elke handler één taak. Daarnaast wordt er ook gebruikgemaakt van een repository pattern om met de database te communiceren. Er is gekozen om gebruik te maken van een repository pattern zodat er geen afhankelijkheid is van de database. Een sequencediagram van deze architectuur is te zien in figuur 2.8

Figuur 2.8: Sequencediagram Handler structuur



De verschillende handlers zorgen ervoor dat het single-responsibility principle nagevolgd wordt. Omdat elke handler maar 1 verantwoordelijk heeft. Verder worden er kleine interfaces gebruikt om aan het Interface segregation principle te volgen. En als laatste worden de verschillende dependencies geïnjecteerd (dependency injection pattern) om harde koppeling te voorkomen.

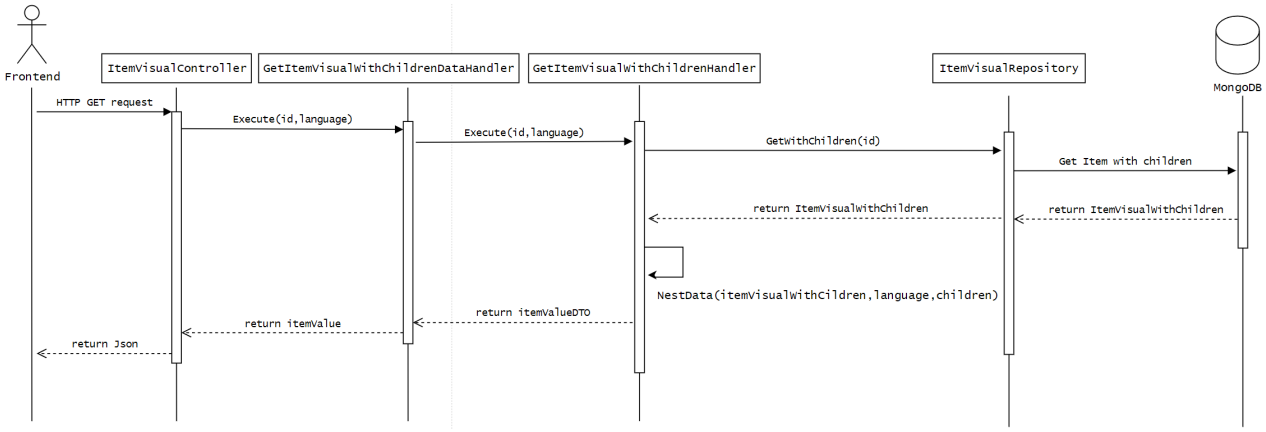
2.3 Process View

De process view van het 4 + 1 model dient om het run-time gedrag van het systeem in beeld te brengen (Kruchten, 1995). In deze sectie wordt er gekeken hoe het ophalen van de data vanuit de backend werkt. Verder wordt er ook gekeken hoe de data gerenderd wordt in de frontend.

2.3.1 Backend

Om het proces van het ophalen van de data in beeld te krijgen is er gebruikgemaakt van een sequencediagram. In figuur 2.9 (vergrote afbeelding is te zien in A.1) is te zien dat er 4 lagen zijn. Dit zijn de controller, datahandler, handler en repository. De logica om te bepalen of het een succesvolle request was zit in de datahandler. De verschillende handlers handelen de buisness logica af. En als laatste is er de repository die de data uit de database haalt.

Figuur 2.9: Sequencediagram ItemValue



2.3.2 Frontend

Om de geneste structuur van het datamodel te renderen wordt er gebruikgemaakt van 2 verschillende component types. Deze component types worden appart toegelicht om een beter beeld te schetsen.

Type 1: Implementatie componenten

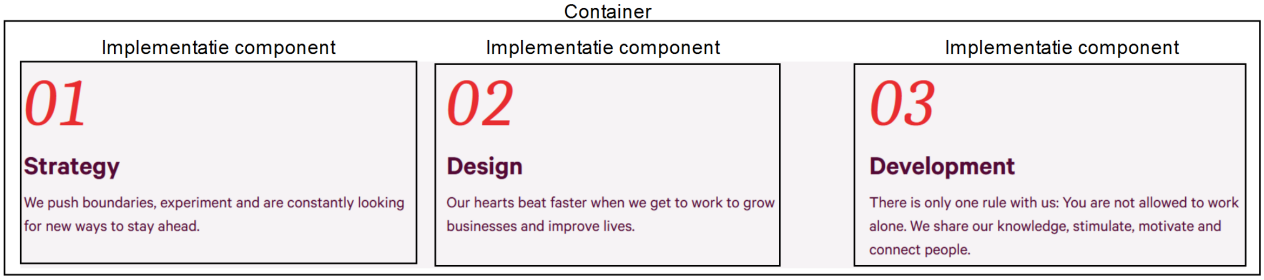
Het eerste type componenten is het component dat daadwerkelijk de data rendered. Deze componenten pakken de *fields* van het *item* en renderen dit op de frontend. Als voorbeeld is er een stukje van de Snakeware site gepakt om dit beter toe te lichten (zie figuur 2.10) In dit voorbeeld wordt het titel field gebruikt om de titel van het component te renderen. En hetzelfde wordt ook gedaan voor het nummer en de body van het component.

Figuur 2.10: Visualisatie implementatie component

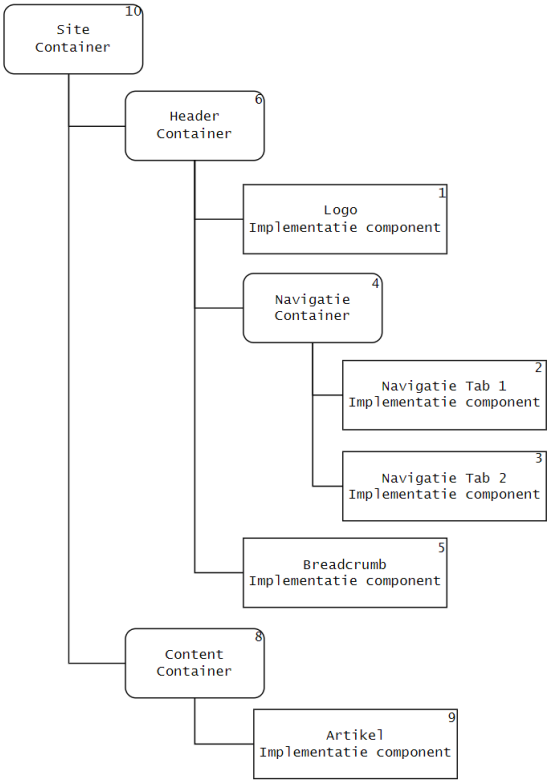
Type 2: Containers

Omdat de datastructuur meerder componenten moet kunnen groeperen wordt er gebruikgemaakt van containers. Containers zijn generieke componenten die een of meerdere implementatie componenten of meerdere containers kan renderen kan renderen. Door het gebruik van een generieke structuur is het mogelijk om elke complexe datastructuur te renderen zonderen enige aanpassingen te maken aan het model zelf. Om hier een beter beeld bij te schetsen is het vorige voorbeeld (zie figuur 2.10) uitgebreid. De uitgebreide versie is te zien in figuur 2.11. Verder is er in figuur 2.12 een voorbeeld gemaakt van een versimpelde site architectuur. Normaliter bevatten de *content container* ook meerdere containers en implementatie componenten. Deze zijn weggelaten om mee voor meer duidelijkheid te geven aan het figuur. Verder is het figuur in de rechterkant van elke container en implementatie component een nummer te zien. Deze nummers laten zien in welke volgorde de componenten gerenderd worden.

Figuur 2.11: Visualisatie containers

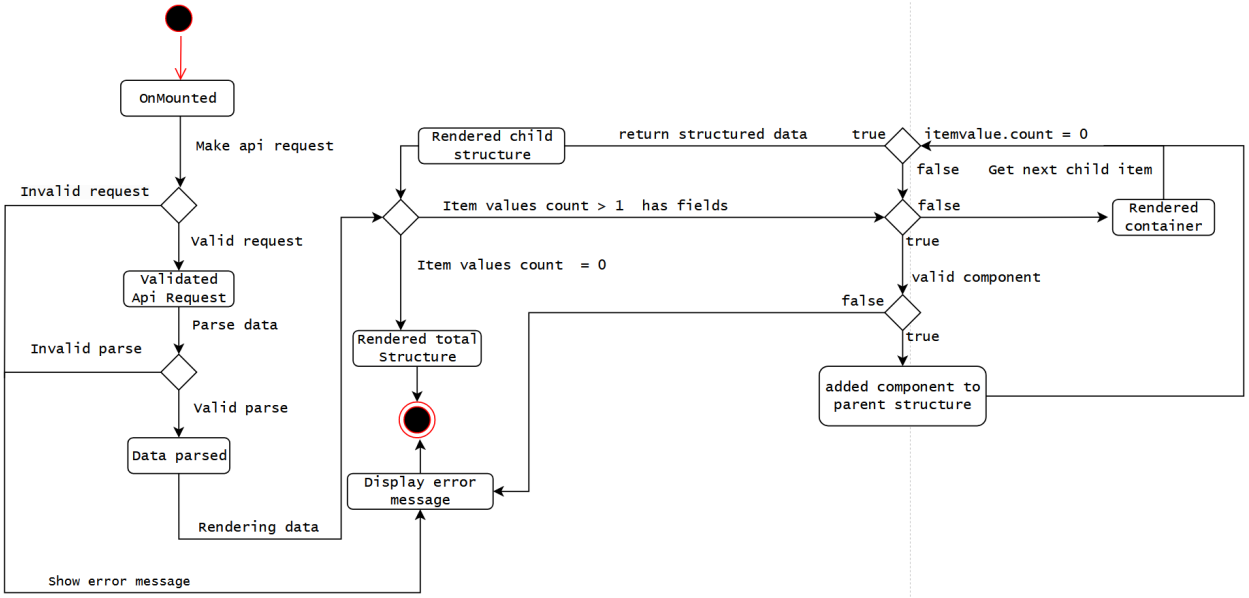


Figuur 2.12: Visualisatie containers



Figuur 2.13 laat zien hoe de frontend omgaat met deze containerstructuur. Na het ophalen van de data van de api wordt de root container gerendered. Eerst wordt er gekeken of het huidige object 1 of meer items heeft. Daarna wordt er gecontroleerd of het item een valide implementatie component heeft. Als het item een container is dan wordt deze recursief gerenderd. Dit wordt gedaan tot dat alle items gerenderd zijn.

Figuur 2.13: flowchart diagram frontend

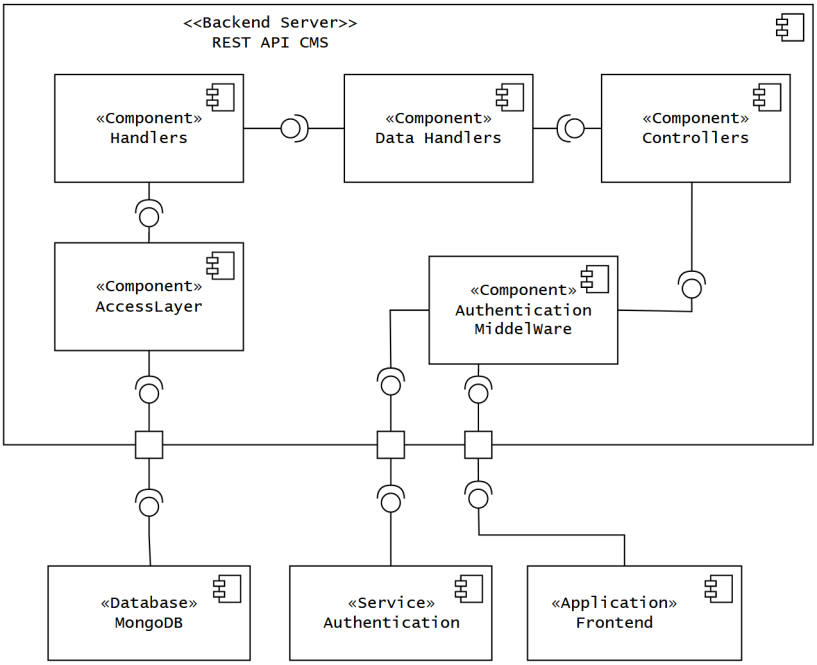


2.4 Development view

De development view is gefocust op het in beeld brengen van de organisatie van software modules (Kruchten, 1995). Om dit in beeld te brengen is er gebruikgemaakt van een component diagram.

In figuur 2.14 is een component diagram gemaakt. In het diagram is te zien dat er gebruikgemaakt wordt van een externe authenticatie provider. Verder is de SOLID implementatie terug te vinden in het diagram (zie 2.2.2 voor meer informatie).

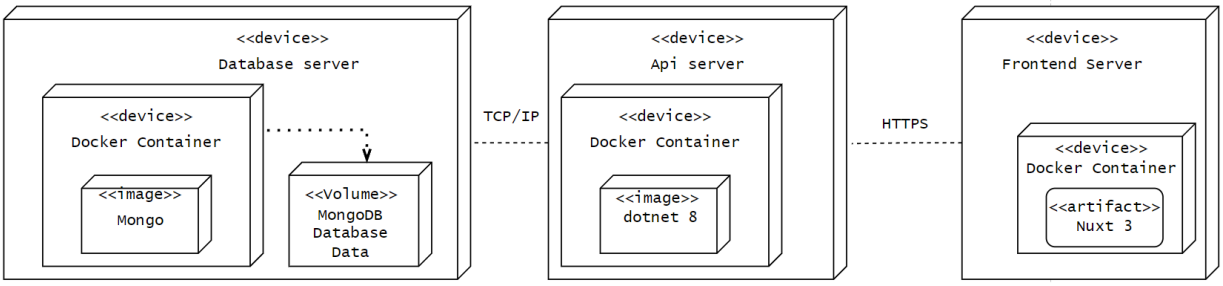
Figuur 2.14: Deployment diagram van het afstudeerproduct



2.5 Physical view

In de physical view wordt er gekeken naar hoe het systeem gedeployed moet worden en waar. Om dit in beeld te brengen is er gebruikgemaakt van een deployment diagram 2.15. Er is gekozen om gebruik te maken van Docker (Docker.com, g.d.). Docker is een software waarmee software gedeployed kan worden op elke machine op een lichte en efficiënte manier (Docker.com, g.d.). Door gebruik te maken van docker kan het systeem gemakkelijk gedeployed worden op cloud hosting platformen en dynamisch schalen.

Figuur 2.15: Deployment diagram van het afstudeerproduct



2.6 Database Keuze

Omdat dit een nieuw datamodel is, is er ook nagedacht over welke database gebruikt zal worden. De meeste gebruikte databases binnen Snakeware zijn SQL-databases (Amazon, g.d.). De specifieke database die het huidige CMS gebruikt is Microsoft SQL Server (Microsoft, g.d.-c). Voor de afstudeeropdracht is er gekozen om van dit pad af te stappen en een NoSQL (MongoDB, g.d.-e) database te gebruiken.

Er is voor een NoSQL-database oplossing gekozen omdat het belangrijk is dat het systeem zo flexibel mogelijk blijft. Omdat het datamodel semigestructureerde data opslaat bij definitie, veroorzaakt het gebruik van een SQL-database extra overhead. Verder geeft een NoSQL-database meer vrijheid in het uitbreiden van het systeem.

Voor de specifieke database keuzes is er gekeken naar een paar kwantitatieve punten die gebruikt worden om verschillende tegenover elkaar te zetten. Deze punten zijn datamodel, licentiekosten, C# sofware development kit (SDK), schaalbaarheid en licentiekosten. Omdat er gebruik gemaakt wordt van een NoSQL-database zijn er alleen NoSQL-database overwogen. De vergelijkingen van de databases zijn te zien in tabel 2.1.

Een van de criteria die gebruikt wordt is ACID (MongoDB, g.d.-d) wat staat voor atomicity, consistency, isolation en durability.

1. **Atomicity:** Alle veranderingen worden uit gevoerd als een operatie. Als een van de veranderingen faalt alle veranderingen
2. **Consistency:** De data in de database is consistent aan het begin van de transactie en aan het einde.
3. **Isolation:** De stand van een transactie heeft geen invloed op andere gelijktijdige transacties
4. **Durability:** Als de transactie succesvol is afgerond zijn permanent en gaan niet verloren, zelfs niet bij systeemfouten

In essentie biedt ACID een set van eigenschappen die ervoor zorgen dat transacties op een betrouwbare en consistente manier worden uitgevoerd in een databasesysteem, wat cruciaal is voor gegevensintegriteit en betrouwbaarheid.

Tabel 2.1: Database providers vergelijking

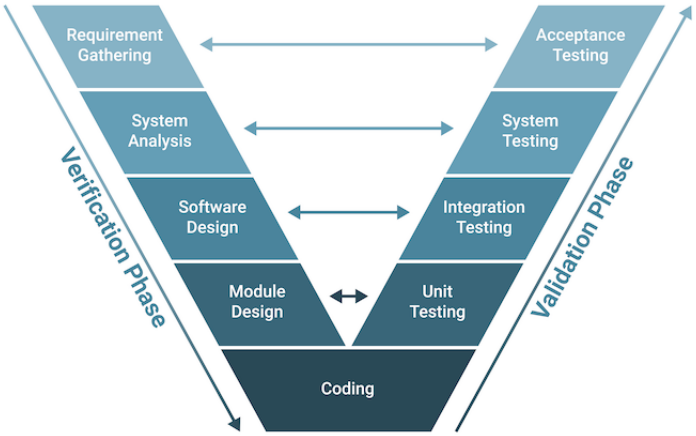
Database providers					
Criteria	MongoDB	SurrealDB	Redis	CouchBase	Cassandra
Datamodel	Document	Document	key-value	Document	Wide column
Licentiekosten	Nee	Nee	Nee	Ja	Nee
C# SDK	Ja	Ja	Ja	Ja	Ja
Schaalbaarheid	Horizontaal/ verticaal	Horizontaal	Horizontaal/ verticaal	Horizontaal/ verticaal	Horizontaal/ verticaal
ACID	Ja	Ja	Nee	Nee	Nee

Voor de specifieke database is er gekozen voor MongoDB (MongoDB, g.d.-a). Deze keuze is gemaakt op basis van de verschillende criteria die gesteld zijn in tabel 2.1 MongoDB is een open-source document database, waarbij de data wordt opgeslagen in een json achtige format (Bson (MongoDB, g.d.-c)). Verder is MongoDB de meest gebruikte NoSQL-database waardoor er veel informatie over te vinden is.

2.7 Teststrategie

Om de kwaliteit van het product te waarborgen is er een test strategie opgesteld. De testen worden gebruikt om de technische en functionele correctheid van het programma aan te tonen. Dit wordt gedaan door gebruik te maken van het V-Model (Oppermann, 2023) (zie figuur 2.16). Het V-Model is een variatie van de SDLC waar bij er bij elke stap van het proces testen worden uitgevoerd om de kwaliteit van het product te waarborgen.

Figuur 2.16: V-Model (Oppermann, 2023)



Bij elke stap van de validatie fase van het realiseren worden de verschillende testen uitgevoerd. De Unit en intergration testen worden automatisch uitgevoerd. De systeem en acceptatie testen worden met de hand uitgevoerd.

Hoofdstuk 3

Realisatie

In dit hoofdstuk wordt de realisatie van het eindproduct in beeld gebracht. Dit wordt gedaan door de bouwomgeving in beeld te brengen. Daarna wordt de implementatie van de SOLID principes besproken. En als laatste wordt er focus gelegd op het ontwikkelde eindproduct.

3.1 Bouwomgeving

Tijdens de laatste fase van het afstudeerproces is er verschillende software gebruikt. Deze software omvat libraries, frameworks, database tools en modelleringssoftware. In deze sectie wordt de software besproken en toegelicht waarom deze keuze gemaakt.

Proces en documentatie

Voor het maken van documenten tijdens het afstudeerproces (Plan van aanpak, Onderzoek en Technisch verslag) is er gebruikgemaakt van **Latex** (latex-project, g.d.). Er is voor Latex gekozen omdat het vergeleken met andere documenten programma's het professioneelste resultaat geeft. De latex teksten zijn geschreven in de code-editor **Neovim** (Neovim-team, g.d.). Om de verschillende diagrammen te maken is er gebruikgemaakt van **Draw.io** (draw.io, g.d.). Voor het noteren van het proces is er gebruikgemaakt van de notitie applicatie **Obsidian** (Obsidian, g.d.). Hier werden de verschillende sprints bijgehouden en genoteerd. Er is voor Obsidian gekozen omdat dit een simpele manier om de sprints bij te houden. Binnen Snakeware wordt er gebruikgemaakt van het platform Jira (Atlassian, g.d.). Jira is een erg extensief pakket wat voor de scope van de opdracht te groot is.

Frontend

Voor het realiseren van de frontend applicatie is er gebruikgemaakt van **Vue 3** (Vuejs.org, 2023) en **Nuxt 3** (Nuxt, g.d.). Vue 3 is een Javascript (Mozilla, 2023b) / Typescript (Typescript.org, 2023) framework dat het makkelijker maakt om webapplicaties te maken. Nuxt 3 is een Vue framework wat het makkelijker maakt om Vue applicaties te maken en meerdere rendering opties aanbiedt. Er is voor Vue 3 met **Typescript** en Nuxt 3 gekozen omdat de gebruikte standaard is binnen Snakeware. Hierdoor ontstaat er meer draagvlak binnen de frontend developers. Voor het schrijven van de frontend code is gebruikgemaakt van **Webstorm** (Jetbrains, g.d.).

Backend

Voor de backend code is er gebruikgemaakt van **C# 12** (Microsoft, 2022) en het **.NET 8** framework (Microsoft, g.d.-a). Er is voor deze taal en framework gekozen omdat het een van de randvoorwaarden is van de afstudeeropdracht. Verder wordt er gebruikgemaakt van **xUnit** (Foundation, g.d.) om de unit testen te maken. Er is voor xUnit gekozen omdat dit framework het beste is in het isoleren van de testen (Sheth, 2021). Verder is xUnit ook het meest gebruikte test-framework binnen snakeware. Om de code te schrijven wordt er gebruikgemaakt van **Visual Studio 2022** (Microsoft, g.d.-b).

Overig

Zoals besproken in 2.6 wordt er gebruikgemaakt van MongoDB. Voor het managen van de database is er gebruikgemaakt van **MongoDB Compas** (MongoDB, g.d.-b). Met deze tool is het mogelijk om querries uit te testen en de data makkelijk te bekijken. Voor de frontend en backend is er gebruikgemaakt van versiebeheer. De tools die hier voor zijn gebruikt zijn **Git** (Git-scm, g.d.) en als repository platform **Bitbucket** (Bitbucket, g.d.). Verder is er voor het beheren van de docker omgeving gebruikgemaakt van **Docker desktop** (Docker, g.d.).

3.2 SOLID implementatie

Tijdens het realiseren zijn de ontwerpprincipes van SOLID (zie 2.2.2 voor meer informatie) mee genomen. In deze sectie wordt er meer ingezoomd naar de implementatie van de SOLID principes en hoe dat tot uiting is gekomen.

De softwarearchitectuur maakt gebruik van handlers. Een handler is een encapsulatie van de business logica voor een bepaalde context. Deze handler heeft maar één taak waardoor het aan het *Single-responsibility principle* voldoet. In het voorbeeld (figuur 3.1) is te zien dat deze handlers als taak heeft het upserten van Items. In figuur 3.2 is te zien dat hij 2 methodes heeft, maar beide methodes hebben dezelfde "taak", het upserten van items.

Figuur 3.1: UpsertItemHandler Implementatie

```
namespace CMS_Api.ControllersLayer.Item.Handlers.UpsertItemHandler
{
    1 reference | dante, 20 days ago | 1 author, 1 change
    public class UpsertItemHandler(IItemRepository itemRepository, IValidateItemHandler validateItemHandler) : IUpsertItemHandler
    {
        2 references | dante, 20 days ago | 1 author, 1 change
        public string? Execute(ItemValueEntity itemValue)
        {
            bool isValid = validateItemHandler.Execute(itemValue);

            if (!isValid)
            {
                return null;
            }
            string upsertedId = itemRepository.Upsert(itemValue);
            return upsertedId;
        }

        2 references | dante, 20 days ago | 1 author, 1 change
        public long Execute(IEnumerable<ItemValueEntity> itemValues)
        {
            List<ItemValueEntity> resultItemDefinitions = itemValues
               .Where(itmDef => validateItemHandler.Execute(itmDef, itemValues))
               .ToList();

            long resultModifiedCount = itemRepository.Upsert(itemValues);
            return resultModifiedCount;
        }
    }
}
```

Figuur 3.2: UpsertItemHandler interface

```
using CMS_Api.Models.ItemVisual;

namespace CMS_Api.ControllersLayer.Item.Handlers.UpsertItemHandler
{
    3 references | dante, 54 days ago | 2 authors, 2 changes
    public interface IUpsertItemHandler
    {
        2 references | dante, 54 days ago | 1 author, 1 change
        public string? Execute(ItemValueEntity itemValue);
        2 references | dante, 54 days ago | 1 author, 1 change
        public long Execute(IEnumerable<ItemValueEntity> itemValues);
    }
}
```

In de verschillende handlers wordt er gebruikgemaakt van het *Open/Closed principle*. Doordat de handler gebruik maakt van de interface *IUpsertItemHandler* zou er makkelijk een nieuwe implementatie toegevoegt kunnen worden. Omdat de handlers maar één taak hebben blijft hun interface erg klein (meestal 1 of 2 methoden). Dit zorgt dat de interfaces erg gefocust blijven op hun taak en er daarmee voldaan wordt aan het *Interface segrigation principle*. De injectie van de concrete implementatie wordt gedaan door middel van *Dependency injection*. Door gebruik te maken van *depencency injection* wordt er ook voldaan aan het *Depencency inversion principle*.

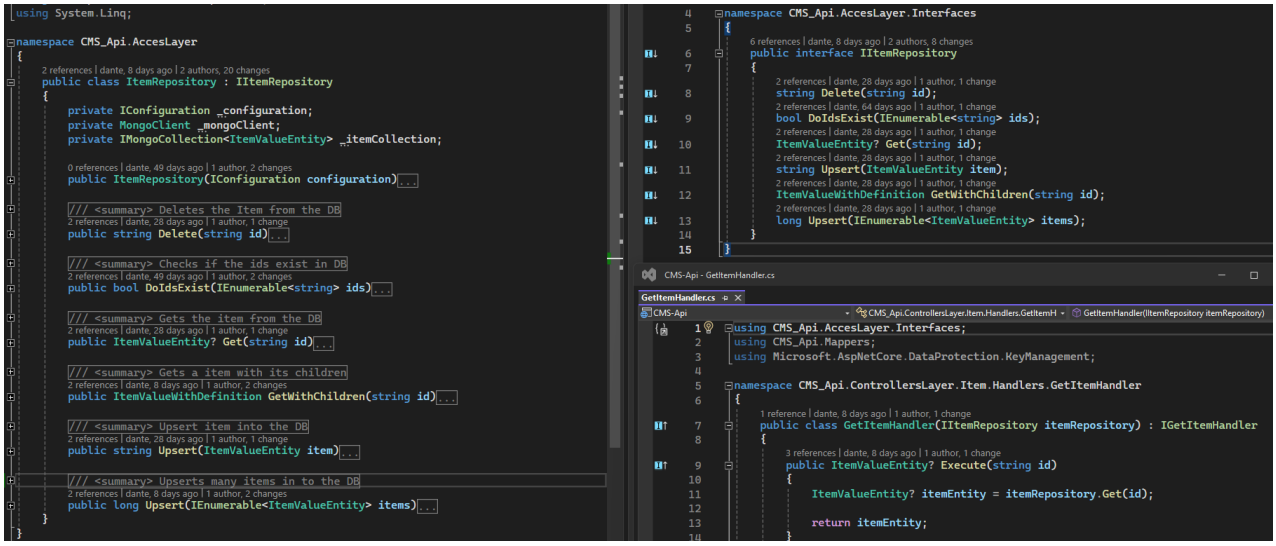
3.2.1 Repository pattern

Om de data laag en de logica van elkaar te scheiden is er gebruikt van een repository pattern (DevIQ, 2023). Het repository pattern is een structureel design pattern dat de data laag en logica van de applicatie scheidt. Dit wordt bereikt door middel van een abstracte tussenlaag die communicatie verzorgt tussen de logica en de data laag.

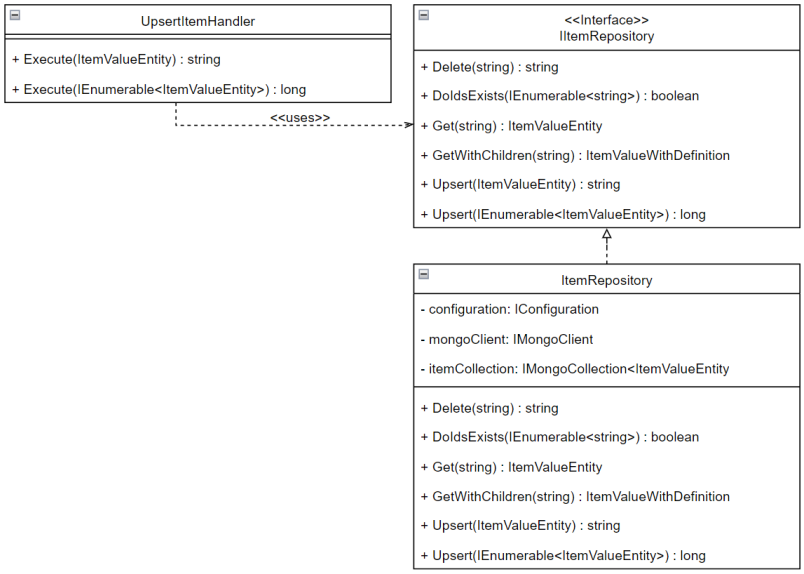
Dit leidt tot een scheiding tussen de logica en de data, waardoor voldaan wordt aan het *single responsibility principle*. Dit resulteert in beter testbare code, wat de betrouwbaarheid van het programma verhoogt. Bovendien voorkomt dit dat de codebase afhankelijk is van één specifieke databaseprovider.

Dit is geïmplementeerd door de logica af te laten hangen van een interface in plaats van een concrete implementatie. Vervolgens wordt bij initialisatie van de applicatie een concrete versie geïnjecteerd. Een visualisatie van de geïmplementeerde versie is te zien in figuur 3.3. Er is ook een klassen diagram gemaakt van de implementatie die is te vinden in 3.4.

Figuur 3.3: Repository pattern implementatie



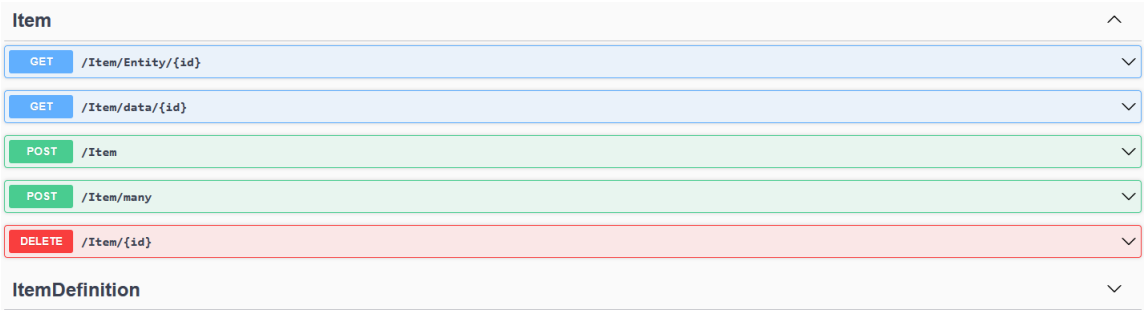
Figuur 3.4: Repository pattern implementatie UML



3.3 Eindproduct

Het doel van de afstudeeropdracht is het maken van een proof of conceptversie van een nieuw CMS. Hierbij was bepaald om de CMS beheerder GUI weg te laten om de scope van de opdracht realistisch te houden. Er is wel gebruikgemaakt van een developer interface genaamd swagger (Swagger, g.d.). Swagger is een softwareoplossing dat een UI geeft aan een API zodat er makkelijker mee geïnteracteerd kan worden. In figuur 3.5 is de swagger UI te zien van de CMS API.

Figuur 3.5: Swagger interface

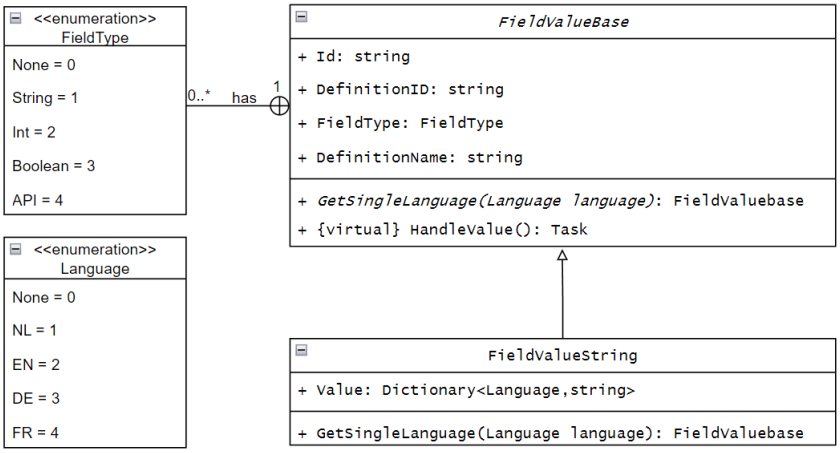


3.3.1 Implementatie Datamodel

Het ontwerp dat is gemaakt, mist een specifieke implementatie voor het gegevenstype "content", zoals te zien is in figuur 2.7 (raadpleeg sectie 2.2.1 voor meer details). Dit gegevenstype zou verschillende vormen van gegevens moeten kunnen bevatten, zoals strings, getallen en booleans. In programmeertalen zoals Python en JavaScript is dit geen probleem, omdat ze dynamisch getypeerd zijn. Dit betekent dat het gegevenstype van variabelen niet van tevoren gespecificeerd hoeft te worden. Een van de randvoorwaarden is dat het afstudeerproduct gerealiseerd wordt in C# wat een statisch getypeerde taal.

Dit is opgelost door gebruik te maken van een abstracte baseclass *FieldValueBase*. Deze baseclass heeft verschillende implementaties die de verschillende datatypes kunnen ondersteunen. Een klassendiagram hiervan is te zien in figuur 3.6. In het klassendiagram is er één implementatie getoond van *FieldValueBase* om het voorbeeld simpel te houden.

Figuur 3.6: Implementaite Fields



Een van de kenmerken van *FieldValueString* is dat de data wordt opgeslagen in een dictionary met de taal als key. Door vertalingen op te slaan in de *fields*, kunnen vertalingen eenvoudig

worden toegevoegd. Dit resulteert in het onderhouden van slechts één site in plaats van een aparte site voor elke taal.

Het nadeel van een abstracte klasse als datatype gebruiken is dat de standaard C# oplossing van *Json Deserialisation* niet goed werkt. Het probleem ligt in het interpreteren van de abstracte types naar de concrete implementaties. Hierdoor kunnen controller endpoints niet de json data verwerken omdat er niet gedeserialiseerd mag worden van een abstracte klas. Hierom wordt er gebruikgemaakt van een .NET 8 oplossing genaamd *Model binders*. Model binders zorgen ervoor dat controllers direct met het model kunnen werken in plaats van het HTTP request (Larkin, 2022). De geïmplementeerde versie van de model binder is te zien in figuur 3.7. Verder zijn de mappers te zien die gebruikt worden in figuur 3.8.

Figuur 3.7: Item Modelbinder

```
3 references | dante, 12 days ago | 2 authors, 6 changes
public class ItemModelBinder : IModelBinder
{
    0 references | dante, 12 days ago | 2 authors, 3 changes
    public async Task BindModelAsync(ModelBindingContext bindingContext)
    {
        using StreamReader stringReader = new StreamReader(bindingContext.HttpContext.Request.Body);
        string json = await stringReader.ReadToEndAsync();
        JToken jObject = JToken.Parse(json);
        if (jObject != null && jObject is JArray)
        {
            List<ItemValueEntity?> res = jObject.Select(item =>
                ItemValueMapper.FromJson(item.ToString())
            ).ToList();

            bindingContext.Result = res.Contains(null) ? ModelBindingResult.Success(res) : ModelBindingResult.Failed();
            return;
        }

        ItemValueEntity? parsedItem = ItemValueMapper.FromJson(json);
        bindingContext.Result = parsedItem == null ? ModelBindingResult.Success(parsedItem) : ModelBindingResult.Failed();
    }
}
```

Figuur 3.8: Gebruikte mappers voor de modelbinder

```
5 references | ● 3/3 passing | dante, 12 days ago | 1 author, 1 change
public static ItemValueEntity? FromJson(string jsonString)
{
    List<FieldValueEntityBase?>? res = null;

    try
    {
        var settings = new JsonSerializerSettings
        {
            Formatting = Formatting.Indented,
            TypeNameHandling = TypeNameHandling.Objects,
            MissingMemberHandling = MissingMemberHandling.Error,
        };

        JObject jObject = JObject.Parse(jsonString);
        JToken? fieldValues = jObject["fieldValues"];

        if (fieldValues != null && fieldValues is JArray)
        {
            res = fieldValues.Select(item =>
                FieldValueEntityMapper.FromJson(item.ToString())
            ).ToList();

            if (res.Contains(null))
            {
                return null;
            }

            jObject.Remove("fieldValues", out JToken? _);
        }
        ItemValueEntity? parsedItem = JsonConvert.DeserializeObject<ItemValueEntity>(jObject.ToString(), settings);
        if (parsedItem == null)
        {
            return null;
        }
        ItemValueEntity parsedItemWithFieldValues = parsedItem with { FieldValues = res };

        return parsedItemWithFieldValues;
    }
    catch
    {
        return null;
    }
}
```

```
2 references | dante, 62 days ago | 1 author, 1 change
public class FieldValueBaseMapper
{
    2 references | 0 changes | 0 authors, 0 changes
    public static FieldValueBase? FromJson(string jsonString)
    {

        FieldType? type = JObject.Parse(jsonString)["fieldType"]?.ToObject<FieldType>();

        FieldValueBase? fieldBaseConcreteType = type switch
        {
            FieldType.Bool => JsonConvert.DeserializeObject<FieldValueBool>(jsonString),
            FieldType.String => JsonConvert.DeserializeObject<FieldValueString>(jsonString),
            FieldType.Int => JsonConvert.DeserializeObject<FieldValueInt>(jsonString),
            FieldType.API => JsonConvert.DeserializeObject<FieldValueApi>(jsonString),
            FieldType.None => null,
            _ => null
        };
        return fieldBaseConcreteType;
    }
}
```

3.3.2 Implementatie Frontend

Het laatste deel van de afstudeerproduct is de webapplicatie voor de eindgebruiker. Deze website moet de data van de CMS-API kunnen renderen. Er is gekozen om een site na te bouwen, de site die hiervoor gekozen is de Snakeware site (Snakeware.com). Dit is gedaan omdat de developers van de site makkelijk aangesproken kunnen worden voor mogelijke ondersteuning.

Na dat de site was uitgekozen is het ontwerp geraliseerd voor de frontend (zie sectie 2.3.2). Het eerste type is de simpelste variant van de frontend onderdelen. Het voorbeeld dat gebruikt is rendered een stuk tekst met een titel en optioneel een button.

Door middel van de *fields* worden de verschillende waardes gevuld in het component. De implementatie hiervan is te zien in figuur 3.9. Verder is er ook te zien dat de "Link" field gebruikt wordt om een redirect uit te voeren zodra er opgeklikt wordt.

Figuur 3.9: Implementatie van Vue component

```
const { itemValue } = defineProps<{
  itemValue: ItemValue
}>();

const text = getFieldValue(itemValue, field: "Text")
const title = getFieldValue(itemValue, field: "Title")
const link = getFieldValue(itemValue, field: "Link")
const buttonText = getFieldValue(itemValue, field: "ButtonText")

</script>

<template>
    <section class="title-with-text" >
      <h2 class="with-dot" v-html="title" />
      <div class="title-with-text--container">
        <p>{{text}}</p>
        <nuxt-link v-if="link && buttonText" :to="link" class="button large">
          {{buttonText}}
        </nuxt-link>
      </div>
    </section>
</template>
```

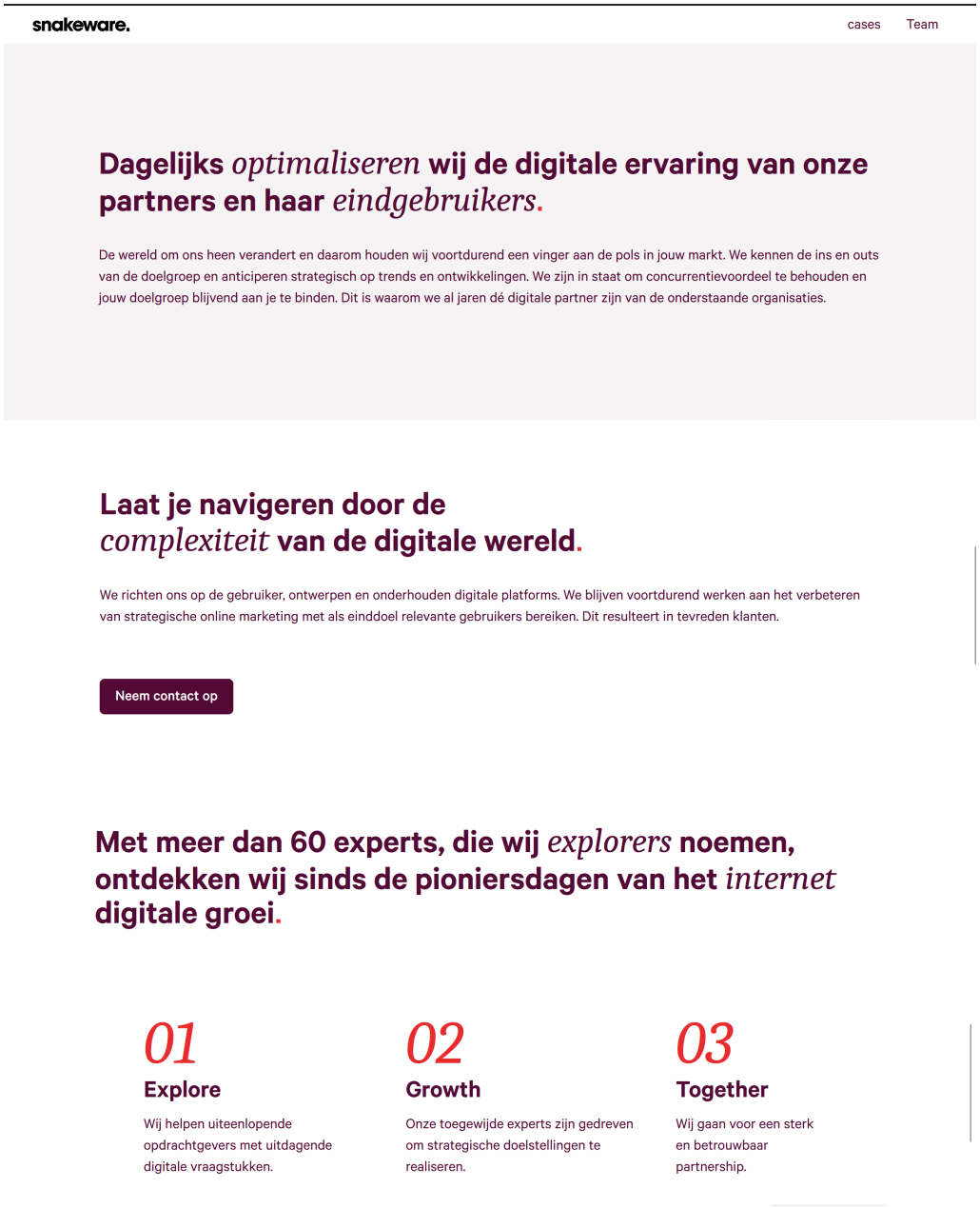
Het tweede type component is een Container component die te zien is in figuur 3.10. De containers renderen alle "implementatie componenten" dit wordt gedaan door gebruik te maken van het component "component". Het "component" wordt ook wel v-component genoemd om de verwarringen te voorkomen. Het v-component kan gerenderd worden als elk component dat mee gegeven wordt. Hierdoor is het makkelijk om op run time te bepalen welk component gerenderd moet worden. Verder wordt het Container zelf ook gebruikt als het item geen fields heeft. Hierdoor kan de content recursief gerenderd worden.

Figuur 3.10: Implementatie container component

```
const { itemValue } = defineProps<{
  itemValue: ItemValue
}>();

const components = {
  TextComponent,
  TitleComponent,
  ButtonComponent,
  FooterComponent,
  OEmbedComponent,
  ImageComponent,
  SummaryComponent,
  NumberColumn,
  TitleWithTextComponent,
  NavTabComponent,
  SVGComponent
}

</script>
<template>
  <section :class="style" style="display: flex;">
    <div v-if="itemValue.itemValues != null" v-for="(val,index) in itemValue.itemValues">
      <component
          v-if="val.fieldValues?.length != 0 && val.visualComponent != null"
          :key="index + val.id"
          :is="components[val.visualComponent?.name]"
          v-bind="{itemValue: val}"
      />
        <Container v-if="val.fieldValues?.length == 0 || val.fieldValues == null" :itemValue="val" />
    </div>
  </section>
</template>
```

Door het gebruik te maken van deze twee type componenten is het mogelijk om de snakeware site te renderen. De volledige rendering van de site is te zien in figuur 3.11 en figuur 3.12.

Figuur 3.11: Frontend(1)



Figuur 3.12: Frontend(2)

Hoofdstuk 4

Beheer, Validatie en Verificatie

In dit hoofdstuk wordt beschreven hoe de kwaliteit van het eindproduct is bewaard. Eerst wordt behandeld de afstudeer voortgang gesprekken zijn gedaan samen met de codereviews. Daar na wordt de Versiebeheer en Codestandaarden inbeeldt gebracht. Vervolgens wordt het proces en de bewaking daarvan besproken En als laatste wordt het testrapport besproken

4.1 Codereviews en Afstudeerstage voortgang

Tijdens de afstudeerperiode is er een keer per week een gesprek ingepland met de afstudeer begeleider Thom Koenders (senior software engineer). Deze gesprekken gingen over de huidige stand van zaken en wat de moglijke problemen die er waren. Verder is het ook vaak over de opzet van de architectuur en de code gegaan.

Verder werden er ook codereviews gebruikt voor dat de code gemerged werd op de main branch. Dit werd gedaan door de verschillende specialisten in hun vak. Voor de backend code is Kevin Sneider (backend software engineer) benaderd voor deze reviews. En voor de frontend is Maks Tennebroek (frontend software engineer) benaderd.

4.2 Versiebeheer

Voor het versiebeheer van de broncode is er gebruikgemaakt van git en Bitbucket als git repository platform. Er is voor git en Bitbucket gekozen omdat dat de standaarden zijn binnen Snakeware. Hierdoor kan het makkelijk opgepakt worden door de andere developers binnen Snakeware. Voor elke nieuwe functionaliteit is er een nieuwe branche aangemaakt waar op ontwikkeld werd. Wanneer de functionaliteit klaar was werd er een pull request gemaakt die vervolgens gemerged werd. Door dit te doen wordt de code kwaliteit gewaarborgd.

4.3 Codestandaarden

Tijdens het ontwikkelen van de afstudeeropdracht is er gecontroleerd op de code standaarden die gebruikt zijn. Deze controles werden tijden de code reviews en stagevoortgang besproken (Zie sectie 4.1).

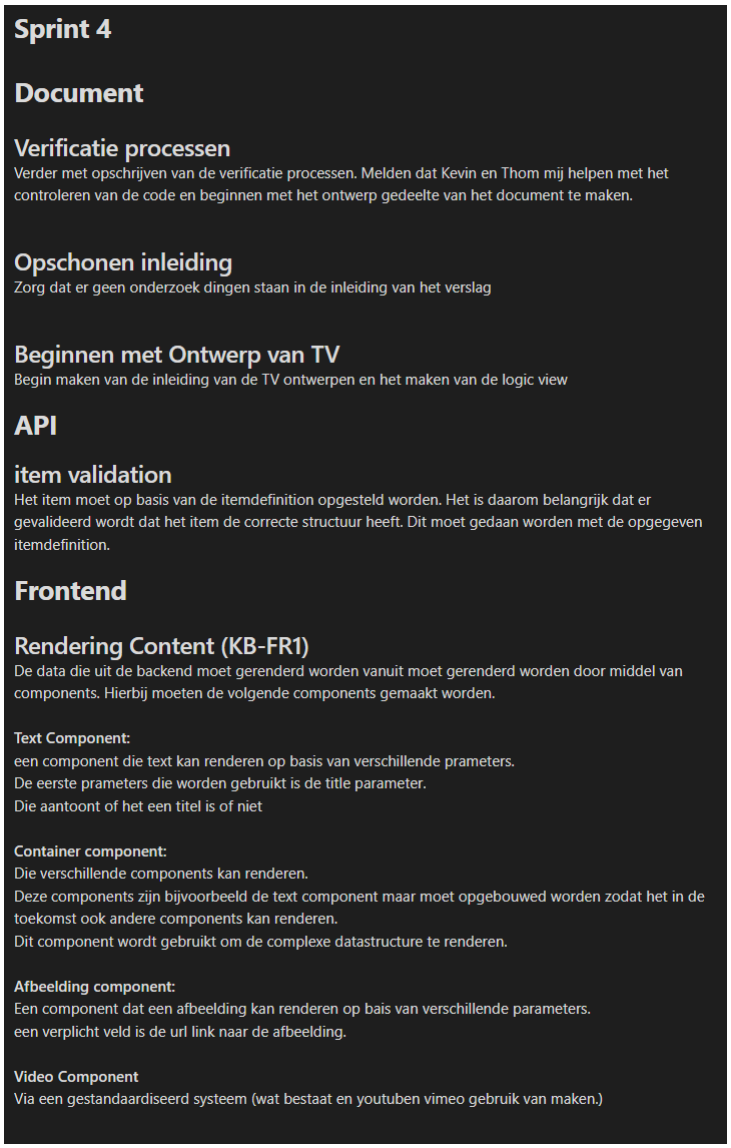
Voor de C# code is er van grotendeels gebruik gemaakt van de standaard van C# (Microsoft, 2023). Er zijn afwijking gemaakt van deze principes deze staan genoteerd in bijlage B. Voorbeelden van deze standaarden zijn het gebruikmaken van expliciete types en variabele

benaming conventies. Verder wordt er ook gecontroleerd op het gebruik van de verschillende SOLID principes (zie sectie 2.2.2).

4.4 Process en bewaking

Tijdens de afstudeerperiode is er gewerkt met een Agile (LeanSixSigmaGroup, g.d.) methode genaamd Scrum (Vennix, g.d.). Er is gewerkt met sprints van 2 weken waar bij de requirements opgedeeld werden in kleine stukjes. Dit is gedaan om de taken behapbaar te maken en om snel veranderingen te kunnen aanbrengen. Elke week werd er met de afstudeerbegeleider de progressie gesproken van de week waar bij mogelijk bijgestuurd werd. De verschillende sprints werden bijgehouden in een notitie programma **Obsidian** hierbij is elke sprint opgedeeld in een apparte note. Vervolgens is er gebruikgemaakt van een scrumboard om de progressie bij te houden dit werd ook gedaan in Obsidian. In figuur 4.1 is te zien hoe zo'n sprint ingepland werdt.

Figuur 4.1: Sprint 4 van het realisatie process



4.5 Testrapport

Tijdens het ontwikkelen van het eindproduct zijn er verschillende testen uitgevoerd. In deze sectie worden de verschillende testen die gemaakt zijn beschreven. Er is gebruik gemaakt van het V-model de teststrategie die gebruikt wordt is beschreven in sectie 2.7.

4.5.1 Unittesten

Voor het unit testen van de API is er gebruikgemaakt van xUnit (voor meer informatie zie sectie 3.1). Unit testen worden gebruikt om de verschillende logica van het systeem te testen. Hierom zijn testen geschreven voor de handlers en de data mappers in de applicatie. Een voorbeeld van zo'n unit test is te zien in figuur 4.2. Verder is er een overzicht van de verschillende unittesten in figuur 4.4.

Figuur 4.2: Geïmplementeerde unit test

```
public class ItemValueMapperTests
{
    [Fact]
    | 0 references
    public void FromJson_Valid()
    {
        // arrange
        string itemId = "65af7493c97f9e0ada822829";
        string definitionId = "65af7307c97f9e0ada82281d";
        string name = "EmployeImage";

        string fieldId = "65c6305ef81b7bb1658708d3";
        string fieldDefinitionId = "65af7307c97f9e0ada82281e";

        Dictionary<Language, string> fieldValues = new Dictionary<Language, string> {
                {Language.NL,"https://snakeware.nl/_ipx/f_webp&s_2280x2280/img/home/Sander.jpg" },
                {Language.EN,"https://snakeware.nl/_ipx/f_webp&s_2280x2280/img/home/Bradley.jpg" }
        };

        string json = $$"""
                {
                    "id": "{{itemId}}",
                    "definitionId": "{{definitionId}}",
                    "name": "{{name}}",
                    "fieldValues": [
                        {
                            "id": "{{fieldId}}",
                            "definitionId": "{{fieldDefinitionId}}",
                            "fieldType": "{{FieldType.String}}",
                            "value": {
                                "NL": "https://snakeware.nl/_ipx/f_webp&s_2280x2280/img/home/Sander.jpg" ,
                                "EN": "https://snakeware.nl/_ipx/f_webp&s_2280x2280/img/home/Bradley.jpg" ,
                            }
                        }
                    ]
                }
            """;

        List<FieldValueEntityBase> fields = new() {
            new FieldValueStringEntity(id: fieldId, definitionId: fieldDefinitionId, fieldType: FieldType.String, fieldValues)
        };

        ItemValueEntity expectedResult = new ItemValueEntity(
            id: itemId,
            name: name,
            definitionID: definitionId,
            fieldValues: fields,
            itemValues: null
        );

        // act
        ItemValueEntity? result = ItemValueMapper.FromJson(json.ToString());

        // assert
        result.Should().BeEquivalentTo(expectedResult);
    }
```

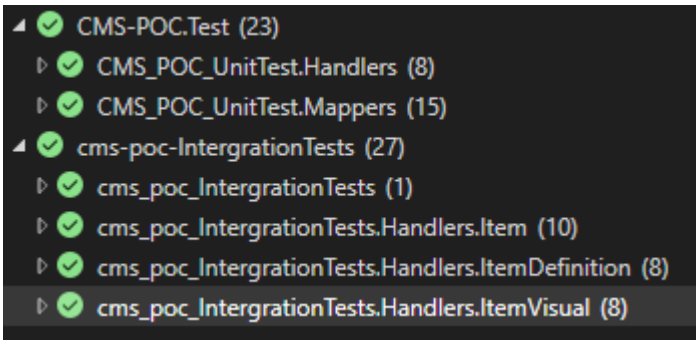
4.5.2 Integratie testen

Alle belangrijke interacties tussen de componenten zijn getest door middel van integratie testen. Deze testen zijn ook gemaakt in xUnit een van de geïmplementeerde testen is te zien in figuur 4.3. In het voorbeeld is te zien dat er gebruik wordt gemaakt van het mocken van de repository. Mocken is het gebruiken van een "nep" implementatie van de gebruikte interface. Hierdoor is het mogelijk om aan te geven welke specifieke methode terugverwacht wordt. Omdat er gebruikgemaakt wordt van het repository pattern is het makkelijk om de database te mocken. Verder is er een overzicht te zien van alle testen in figuur 4.4.

Figuur 4.3: Geïmplementeerde integratie test

```
4 references | 2/4 passing
private static UpsertItemDataHandler SetupUpsertItemDataHandler(IItemRepository itemRepository)
{
    ILoggerFactory factory = LoggerFactory.Create(l => l.AddDebug());
    ILogger<ValidateItemHandler> logger = factory.CreateLogger<ValidateItemHandler>();

    GetItemHandler getItemHandler = new(itemRepository);

    ValidateItemHandler validateItemHandler = new(itemRepository, logger);

    UpsertItemHandler upsertItem = new(itemRepository, validateItemHandler);

    UpsertItemDataHandler upsertItemDataHandler = new(upsertItem, getItemHandler);

    return upsertItemDataHandler;
}

[Fact]
| 0 references
public void UpsertOne_SendBackObject_Valid()
{
    // arange
    ItemValueEntity itemValue = CreateDummyObject();

    Mock<IItemRepository> mockItemRepository = new();
    mockItemRepository.Setup(mock => mock.Upsert(It.IsAny<ItemValueEntity>())).Returns("65af7493c97f9e0ada822829");
    mockItemRepository.Setup(mock => mock.Get(It.IsAny<string>())).Returns(itemValue);
    mockItemRepository.Setup(mock => mock.DoIdsExist(It.IsAny<IEnumerable<string>>())).Returns(true);

    UpsertItemDataHandler upsertItemDataHandler = SetupUpsertItemDataHandler(mockItemRepository.Object);

    // act
    ActionResult<string> result = upsertItemDataHandler.Execute(itemValue, true);

    // assert
    ActionResult<string> expectedResult = new OkObjectResult(itemValue);
    result.Should().BeEquivalentTo(expectedResult);
}
```

Figuur 4.4: Overzicht integratie en unittesten



4.5.3 Systeemtesten

Voor de systeemtesten zijn de webapplicatie en API handmatig getest. Tijdens deze testen werd er gekeken of alle functionele en niet-functionele requirements van de applicatie werkte. Hierbij wordt er gecontroleerd of de nieuwe functionaliteit voldoet aan de vastgesteld applicatiecriteria.

Als een functionaliteit klaar was werd deze apart nog een keer getest. Eerst werd de API getest via swagger en daarna werd er gekeken of de functionaliteit ook goed werkte op de frontend.

4.5.4 Acceptatie testen

Een van de randvoorwaarden was dat er geen GUI wordt gemaakt voor het beheer van het CMS. Dit is gedaan om de scope van de opdracht reëel te houden en binnen de afstudeerperiode. Het nadeel hiervan is dat het lastig is om acceptatie uit te voeren is die betrekkingen hebben tot de beheerder. Daarom is er besloten om geen acceptatie testen uit te voeren.

4.6 Originele eisen en wensen

In deze sectie wordt er nagegaan welke eisen en wensen niet behaald zijn. In sectie 3.3 wordt het gerealiseerde eindproduct besproken. De meeste requirements zijn behaald, met uitzondering van de volgende requirements:

KB-FR6, Authenticatie en identificeren

Het authenticeren en identificeren van de beheerder is een requirement wat besloten is om te laten vallen voor het proof of concept. De requirement was origineel verkeerd ingeschat aan hoeveel waarde het zou opleveren aan het project. Samen met de product owner is er toen besloten om de realisatie van deze requirement een lagere prioriteit te geven. Helaas was er geen tijd over om deze functionaliteit te implementeren.

KB-FR10, Formulieren

Formulieren zijn een belangrijk onderdeel voor verschillende webapplicaties. Verder is het een belangrijk deel van het huidige CMS. Tijdens het ontwerpen van het datamodel kwam al naar voren dat formulieren implementeren veel werk zou kosten. Hierom is er besloten dat formulieren niet tijdens de afstudeerperiode te realiseren.

Hoofdstuk 5

Conclusie

Het doel van de afstudeeropdracht was het maken van een proof of concept CMS. Dit CMS moet het mogelijk maken dat er ook services verleend kunnen worden aan kleinere klanten. Hiervoor moet een nieuw datamodel en systeemarchitectuur komen om het systeem flexibel en onderhoudbaar te maken. Tijdens de afstudeerperiode is de software development life cycle (SDLC) een keer doorlopen.

Requirement analysis fase

Tijdens deze fase is er een onderzoek gedaan naar de requirements van het proof of concept (Klijn, 2023). Uit dit onderzoek is er een lijst van geprioriteerde requirements gekomen die gebruikt zijn om het product te realiseren.

Designfase

Tijdens de ontwerpfase is er een nieuw datamodel ontworpen wat de gedefinieerde requirements kan ondersteunen. Het datamodel is ontworpen zodat het complexe structuren kan bevatten zonder extra code toe te voegen. Hierdoor is het generiek en kan het voor veel doeleinden gebruikt worden. Er is ook een ontwerp gemaakt voor de systeemarchitectuur van het proof of concept. Deze architectuur zorgt ervoor dat het systeem beter te onderhouden is.

Implementatiefase

Na dat het ontwerp gemaakt was, is het geïmplementeerd in 2 producten, een Nuxt 3 frontend applicatie en een .NET 8 API. Voor het realiseren van de API is de gekozen softwarearchitectuur geïmplementeerd. Tijdens het realiseren van het eindproduct is er gebruikgemaakt van de agile methode Scrum.

Testfase

De testfase liep tegelijk met de implementatiefase van het project. Voor het maken van de testen is er een testplan gemaakt die tijdens de testfase doorlopen is. Het testplan maakt gebruik van het V-Model waarbij elke stap doorlopen is behalve de acceptatie testen. Voor het unit testen van de API is er gebruikgemaakt van xUnit. Er is voor xUnit gekozen omdat het beste geïsoleerde testen kan runnen vergeleken met de andere frameworks.

Het proof of concept voldoet bijna aan alle eisen en wensen van de stakeholders op 2 na. Dit is gedaan omdat deze niet veel zouden bewijzen voor het proof of concept.

Het gerealiseerde CMS heeft de doelen gehaald van het proof of concept. Door middel van de datastructuur hoeft er minder maatwerk gedaan te worden voor de eindgebruiker webapplicatie. Verder is het systeem onderhoudbaar door middel van de softwarearchitectuur die ontworpen is.

5.1 Aanbevelingen

In deze sectie worden de mogelijke aanbevelingen besproken die in een vervolg stadium gemaakt kunnen worden. Naast de niet geïmplementeerde requirements die in een volgend stadium mee genomen kunnen worden.

Een van de grotere missende componenten van het proof of concept is de beheerder interface. Door deze applicatie te realiseren is er een compleet proof of concept van een CMS-applicatie.

Verder is het verlenen van webshops niet mee genomen vanwege de scope van de afstudeeropdracht. Dit zou een goede vervolgstap zijn na het realiseren van de beheerder interface.

Bibliografie

Amazon. (g.d.). *What is SQL (Structured Query Language)?* Verkregen 11 maart 2024, van https://aws.amazon.com/what-is/sql/

Atlassian. (g.d.). *Samen snel werken, op een lijn blijven en beter bouwen.* Verkregen 11 maart 2024, van https://www.atlassian.com/nl/software/jira

Bitbucket. (g.d.). Verkregen 12 oktober 2023, van https://bitbucket.org/

DevIQ. (2023). *RepositoryPattern*. Verkregen 11 maart 2024, van https://deviq.com/design-patterns/repository-pattern

Docker. (g.d.). *The 1 containerization software for developers and teams*. Verkregen 21 maart 2024, van https://www.docker.com/products/docker-desktop/

Docker.com. (g.d.). *What is Docker*. Verkregen 28 februari 2024, van https://www.docker.com/

draw.io. (g.d.). *Draw.io homepage*. Verkregen 5 maart 2024, van https://www.drawio.com/

Foundation, . (g.d.). *About xUnit*. Verkregen 5 februari 2024, van https://xunit.net/

Git-scm. (g.d.). *About*. Verkregen 12 oktober 2023, van https://git-scm.com/about

Jetbrains. (g.d.). *The JavaScript and TypeScript IDE*. Verkregen 5 februari 2024, van https://www.jetbrains.com/webstorm/

Klijn, D. (2023, augustus). *Onderzoeksverslag* (PDF) (Verkregen febuari 2024). NHL Stendejn Hogeschool.

Kruchten, P. (1995, november). *Architectural Blueprints–The "4+1" View Model of Software Architecture* (report). Rational Software Corp.

Larkin, K. (2022). *Custom Model Binding in ASP.NET Core*. Verkregen 14 maart 2024, van https://learn.microsoft.com/en-us/aspnet/core/mvc/advanced/custom-model-binding?view=aspnetcore-8.0

latex-project. (g.d.). *Latex - A document preparation system*. Verkregen 5 februari 2024, van https://www.latex-project.org/

LeanSixSigmaGroup. (g.d.). *Wat is Agile*. Verkregen 21 maart 2024, van https://leansixsigmagroep.nl/lean-agile-en-six-sigma/wat-is-agile/

lucidchart. (2023). *UML Use Case Diagram Tutorial*. Verkregen 15 februari 2024, van https://www.typescriptlang.org/

Microsoft. (g.d.-a). *Build. Test. Deploy.* Verkregen 5 februari 2024, van https://dotnet.microsoft.com/en-us/

Microsoft. (g.d.-b). *Github Copilot and Visual Studio 2022.* Verkregen 5 februari 2024, van https://visualstudio.microsoft.com/vs/

Microsoft. (g.d.-c). *Maak kennis met SQL Server 2022.* Verkregen 11 maart 2024, van https://www.microsoft.com/nl-nl/sql-server

Microsoft. (2022). *Een rondleiding door de C#-taal.* Verkregen 10 oktober 2023, van https://learn.microsoft.com/nl-nl/dotnet/csharp/tour-of-csharp/

Microsoft. (2023, januari). *Common C# code conventions.* Verkregen 21 maart 2024, van https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions

MongoDB. (g.d.-a). *Build faster. Build smarter.* Verkregen 11 maart 2024, van https://www.mongodb.com

MongoDB. (g.d.-b). *Compass. The GUI for MongoDB.* Verkregen 5 februari 2024, van https://www.mongodb.com/products/tools/compass

MongoDB. (g.d.-c). *Json and Bson.* Verkregen 15 maart 2024, van https://www.mongodb.com/json-and-bson

MongoDB. (g.d.-d). *What are ACID Properties in Database Management Systems?* Verkregen 18 maart 2024, van https://www.mongodb.com/basics/acid-transactions

MongoDB. (g.d.-e). *What is NoSQL?* Verkregen 11 maart 2024, van https://www.mongodb.com/nosql-explained

Mozilla. (2023a). *Classes.* Verkregen 11 oktober 2023, van https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

Mozilla. (2023b). *JavaScript.* Verkregen 10 oktober 2023, van https://www.javascript.com/

Mozilla. (2023c). *What's in the head? Metadata in HTML.* Verkregen 11 oktober 2023, van https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/The_head_metadata_in_HTML

Neovim-team. (g.d.). *Vision.* Verkregen 5 februari 2024, van https://neovim.io/charter/

Nuxt. (g.d.). *The Intuitive Vue Framework.* Verkregen 5 februari 2024, van https://nuxt.com/

Obsidian. (g.d.). *Sharpen your thinking.* Verkregen 5 februari 2024, van https://obsidian.md/

Oppermann, A. (2023). Verkregen 5 oktober 2023, van https://builtin.com/software-engineering-perspectives/v-model

Sheth, H. (2021, maart). *NUnit vs. XUnit vs. MSTest: Unit Testing Frameworks.* Verkregen 21 maart 2024, van https://www.lambdatest.com/blog/nunit-vs-xunit-vs-mstest/

Snakeware. (2022a). *Cases.* Verkregen 10 oktober 2023, van https://www.snakeware.nl/cases

Snakeware. (2022b). *What we do*. Verkregen 10 oktober 2023, van https://www.snakeware.com/what-we-do

Swagger. (g.d.). *API Development for Everyone*. Verkregen 18 maart 2024, van https://swagger.io/

Typescript.org. (2023). *Typescript is JavaScript with syntax for types*. Verkregen 16 november 2023, van https://www.typescriptlang.org/

Vennix, M. (g.d.). *Wat is Scrum?* Verkregen 12 oktober 2023, van https://leanagilegroep.nl/wat-is-scrum/

Vuejs.org. (2023). *The Progressive JavaScript Framework*. Verkregen 30 oktober 2023, van https://vuejs.org/

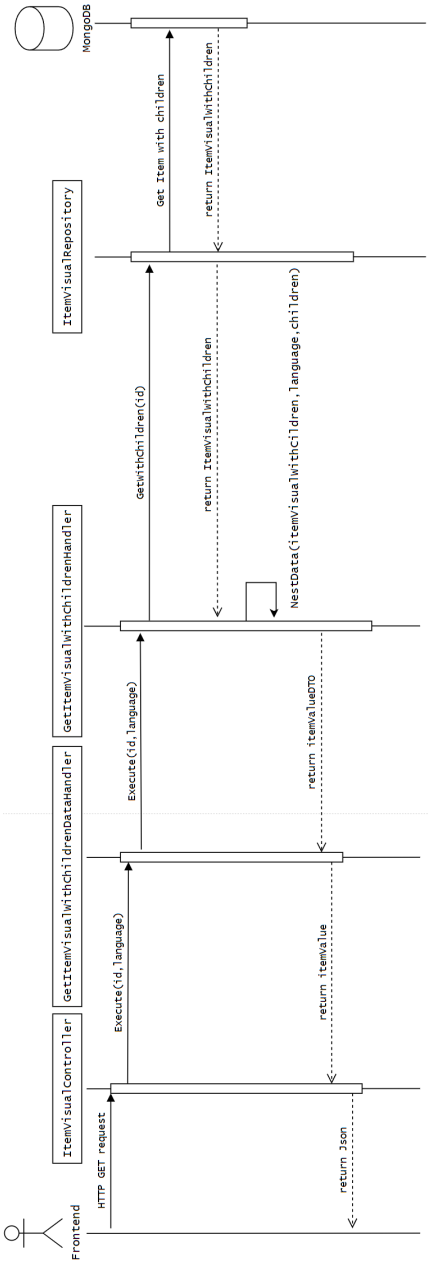
Watts, S. (2020). *The importance of SOLID Design Principles*. Verkregen 11 oktober 2023, van https://www.bmc.com/blogs/solid-design-principles/

Zuci Systems. (g.d.). *Levenscyclus van softwareontwikkeling — Wat is SDLC*. Verkregen 10 oktober 2023, van https://www.zucisystems.com/nl/diensten/levenscyclus-van-softwareontwikkeling-wat-is-sdlc/

Bijlage A

Figuren UML

Figuur A.1: Sequencediagram ItemValue



Bijlage B

Code conventies

Documentation (frontend en backend)

Write the documentation in english so that more people can understand it.
Further more most programming terms dont have a clear definition in the dutch dictionary.

C# code conventions

Explicit typing

Make use of explicit typing, do this so that you can descern the type that is being used at a glance.
This inproves readeblilty of the code.

```
int orderNumber = 10
string orderName = "Order pizza magarita"

CustomerOrder? order = GetOrder(ordernumber);
decimal customerBalance = GetCustomerBalance();

// Don't
var customerOrder = new CustomerOrder();

// do
CustomerOrder customerOrder = new CustomerOrder();

// allowed
CustomerOrder customerOrder = new();
```

No naked ifs

Naked ifs are the practice of not using curly brackets when you are only executing one line of code after the if statement (see example).

Diff unfriendly
When you are commit changes and have people look at pull request, you want to introduce as little clutter as possible. When you add an extra line to the naked, if you introduce a lot respective changes in the file.

Accidental logic changes
You can accidentally add a new line to a naked if and logic is not executed the correct way.
This can lead to unneccery bugs.

```
bool condition = true

// Don't
if(condition)
        Console.Writeline("Don't")

// Do
if(condition)
{
        Console.Writeline("Do")
}
```

Naming bools

When naming booleans don't use the true state as perspective, so name IsX and not IsNotX.
This increases clarity of the function of the variable.

```
// Don't
bool isNotLoading = false
// Don't
bool loading = true
// Do
bool isLoading = true
```

Don't return direct into return statement.

Don't return the result of a function or a statement direct into the return statement.
This increases readability of the function and the possibility to debug the function.

```
// Don't
public string ExampleFunction()
{
        return GetCoolString();
}

// Do
public string CreateCoolString()
{
     string result = GetCoolString();
     return result;
}
```

The most complex variable left

This for increasing readability.

```
public void TestFunc()
{
        List<int> ids = new List<int>() {0,1,2,4};

        // Don't
        if(0 =≠ ids.Count ) {
                // code ..
        }

        // Do
        if(ids.Count =≠ 0) {
                // code ..
        }
}
```

Try to reduce nesting

Try to reduce the amount of nesting in your code.
If there is a lot of nesting in your code, it is difficult to see what is going on.
Make use of Extraction and inversion to reduce nesting

"If you need more than 3 levels of **indentation**,
you're **screwed** anyway,
and you should **fix** your program." - Linux kernel style guide

Don't:

```
public int Calcutate(int bottom, int top)
{
        if(top > bottom)
        {
                int sum = 0
                for (int number = bottom; number ≤ top; number++)
                {
                        if (number & 2 == 0)
                        {
                                sum += number;
                        }
                }

                return sum;
        }
        else
        {
                 return 0
        }

}
```

Do:

```
public int Calcutate(int bottom, int top)
{
        if(top < bottom)
        {
                return 0;
        }

        int sum = 0
        for (int number = bottom; number ≤ top; number++)
        {
                sum += filterNumber(number);
        }

        return sum;
}

private int FilterNumber(int number)
{
        if (number & 2 == 0)
        {
                return number;
        }

        return 0;
}
```

Use IEnumbreble in public interface methods

Reduces the amount of conversions that need to take place if people want to make use of the functions.

```
// Don't
public void WithoutIEnumerable(List<string> collection) {
        // do something with collection
}

// Do
public void WithIEnumerable(IEnumerable<string> collection) {
        // do something with collection
}
```

Frontend conventions

Dont use the "Any" type

This makes the code less save and less predictable

Try to reduce the amount arrow functions

Use arrow functions only for annomynus functions.
The function key word uses less characters and is more clearer than the arrow variant.

```
// Don't
const calculate = (input1: number, input2: number): number => {
        // code here..
}
// Do
function calculate(input1: number, input2: number) -> number {
        // code here
}
```

Always specify the return type

This improves readability and allows you to see what kind of type a function returns at a glance.

```
// Don't
function calcutale(input1: number, input2: number) {
        // code here
        return 42
}

// Do

function calcutale(input1: number, input2: number) -> number {
        // code here
        return 42
}
```