Technisch Ontwerp

Use cases:

UC1 t/m UC19

Niveau 2
Versie: 1.5
Datum: 24-10-2025

Team SE
2025

Versiebeheer

Versie	Datum	Wijzigingen
0.1	9-6-2025	Initiële versie
1.0	11-9-2025	Uc2 en UC3 bijgewerk: lijst en overzicht
1.1.1	18-09-2025	Inloggen geïmplementeerd
1.2.0	26-092025	Filteren/zoeken van producten en verwijderen
1.3.1	03-10-2025	UC10,11 en 13 toegevoegd
1.4.0	10-10-2025	UC12, 14 en 15 toegevoegd
1.5.0	24-10-2025	UC17, 18 en 19 toegevoegd

Distributie

Versie	Datum	Ontvangers
1.0	19-09-2025	Christian Nyqvist
1.1	26-09-2025	Rob Kaesehagen
1.2	03-10-25	Rob Kaesehagen / Christian Nyqvist
1.3	10-10-2025	Rob Kaesehagen
1.4	24-10-2025	Rob Kaesehagen / Christian Nyqvist

Inhoud

Inleiding

In dit technisch ontwerp wordt een systeem beschreven dat zich richt op het ontwerp en de implementatie van de use case die is beschreven in het Functioneel Ontwerp. Het doel van dit document is om nieuwe developers in het team een overzicht te geven van de architectuur van het systeem en de technische aspecten. Het ontwerp omvat een gedetailleerde weergave van de systeemcontext, container- en componentdiagrammen.

Team SE

Setup

In dit hoofdstuk een beschrijving hoe het systeem lokaal te runnen is.

Repositories

Om het systeem te kunnen runnen is het noodzakelijk om de bijbehorende repositories te clonen. Deze zijn te vinden in de studentenhandleiding.

Runnen project

Open de solution in Visual Studio 2022 of een soortgelijke tool. Voor het runnen van het project heb je .NET 8 nodig en apart daarvan MAUI. Vervolgens is het project direct te starten, zonder verdere configuratie.

Technieken

In dit hoofdstuk wordt een overzicht gegeven van de gebruikte tools en standaarden.

Tools

In deze paragraaf een overzicht van de gebruikte tools.

Visual Studio

Visual Studio is een IDE die de ontwikkelaar ondersteunt bij het schrijven van code, onder andere door middel van auto-completion

GitHub
GitHub wordt gebruikt voor de opslag en het beheer van de Git repository. Ook kan gebruikgemaakt worden van de CI/CD om code te deployen naar Cloudflare en Skylab.

Programmeertalen

Bij het ontwikkelen van het systeem is een programmeer taal gebruikt. Hieronder volgen de talen die gebruikt zijn.

C#
De .NET applicatie is geschreven in C#.

Frameworks

Bij het ontwikkelen van het systeem is gebruik gemaakt van een framework. Deze zijn hieronder beschreven.

.NET (Microsoft, 2023)

Voor het ontwikkelen van de back-end is gebruik gemaakt van .NET Maui.

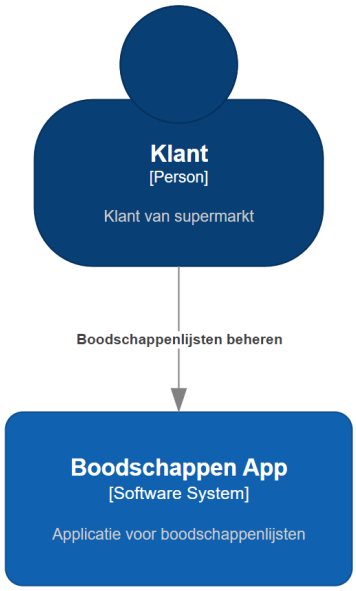
Definition of Done

In dit hoofdstuk is de definition of done uitgewerkt. Dit zijn de eisen, waar nieuwe functionaliteit (technisch) aan moet voldoen, voordat deze kan worden afgerond. Zoals vastgelegd in het projectplan zijn deze eisen voorgelegd aan de opdrachtgever. Deze DOD zou in plaats van in het TO opgenomen kunnen worden in andere projectdocumentatie.

Design	
1.	Het FO en TO weerspiegelen de gerealiseerde functionaliteit
2.	De documentatie heeft een consistente opmaak en duidelijk gestructureerde hoofdstukken.
3.	De documentatie is opgesteld volgens de Windesheim standaarden.
Development and Testing	
4.	De gerealiseerde functionaliteit voldoet aan alle acceptatiecriteria (vastgelegd in het functioneel ontwerp)
5.	De gerealiseerde functionaliteit voldoet aan de eisen gesteld in het issue (zie issue templates)
6.	De testen van de gerealiseerde functionaliteit slagen allen
Deployment	
7.	De pipeline van de feature branch slaagt
8.	Alle secure parameters zijn opgenomen als environment variabelen

Systeem Context (C4 niveau 1)

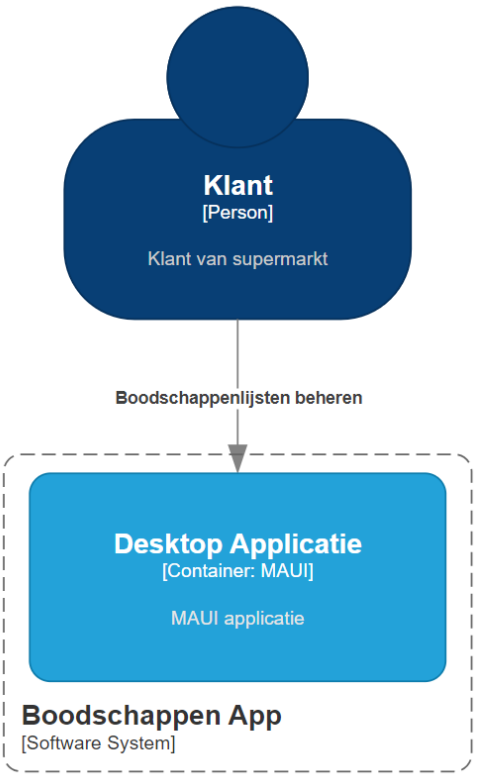
Dit hoofdstuk geeft een overzicht van het systeem. De klant is iemand die boodschappenlijsten wilt maken en beheren. De klant is een persoon in de context van een bezoeker van een supermarkt.



Figuur 1 C4 level 1 van de Boodschappen App

Container Context (C4 niveau 2)

Dit hoofdstuk beschrijft de containers waaruit het systeem bestaat. Voor de Boodschappen App is er één container, de desktop applicatie. Dit is een desktop applicatie gemaakt in het framework van MAUI. Dit is te zien in Figuur 2.



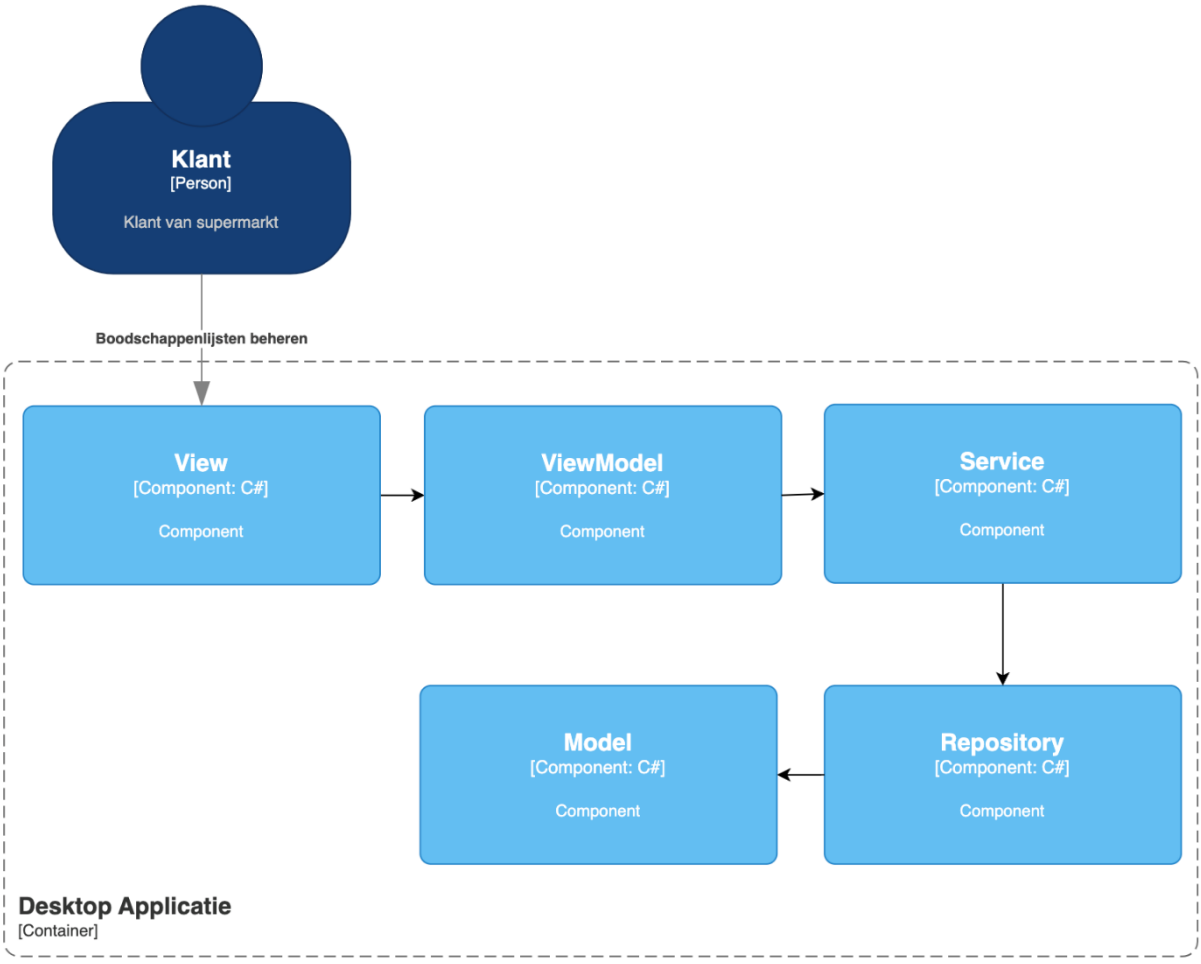
Figuur 2 C4 level 2 van de Boodschappen App

Componenten Context (C4 niveau 3)

In dit hoofdstuk is voor de componenten van de containers van de Boodschappen app getoond hoe de architectuur is vormgegeven.

Desktop applicatie

Voor de desktop applicatie is gebruik gemaakt van de architectuur MVVM (Model-View-ViewModel). Zoals te zien is in Figuur 3 heeft de klant alleen interactie met de *views* van de applicatie. De *views* zijn verantwoordelijk voor de opmaak en weergave van de gebruikersinterface. Aan de hand van databinding wordt een *view* gekoppeld aan een *viewmodel*. Hiermee heeft de *view* geen logica nodig en kan dat op andere plekken worden afgehandeld. De *viewmodel* fungeert als de tussenlaag tussen de *view* en de *model*. In de *viewmodel* zit alle logica die nodig is voor de *view,* zoals het ophalen van gegevens en afhandelen van gebruikersacties. De *models* bevatten de dataclasses (entiteiten), waar geen logica in is opgenomen, maar wel de data in wordt opgeslagen.



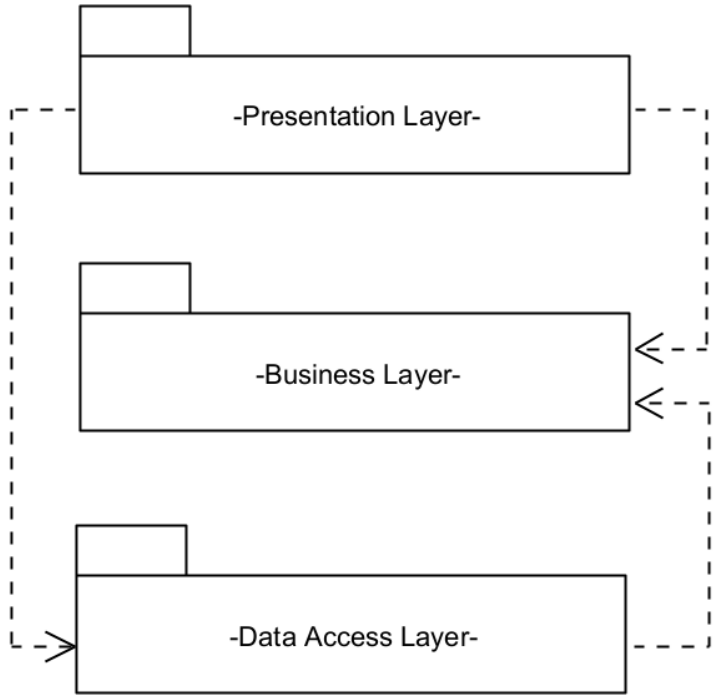
Figuur 3 C4 level 3 van de Boodschappen App

Securitymaatregelen

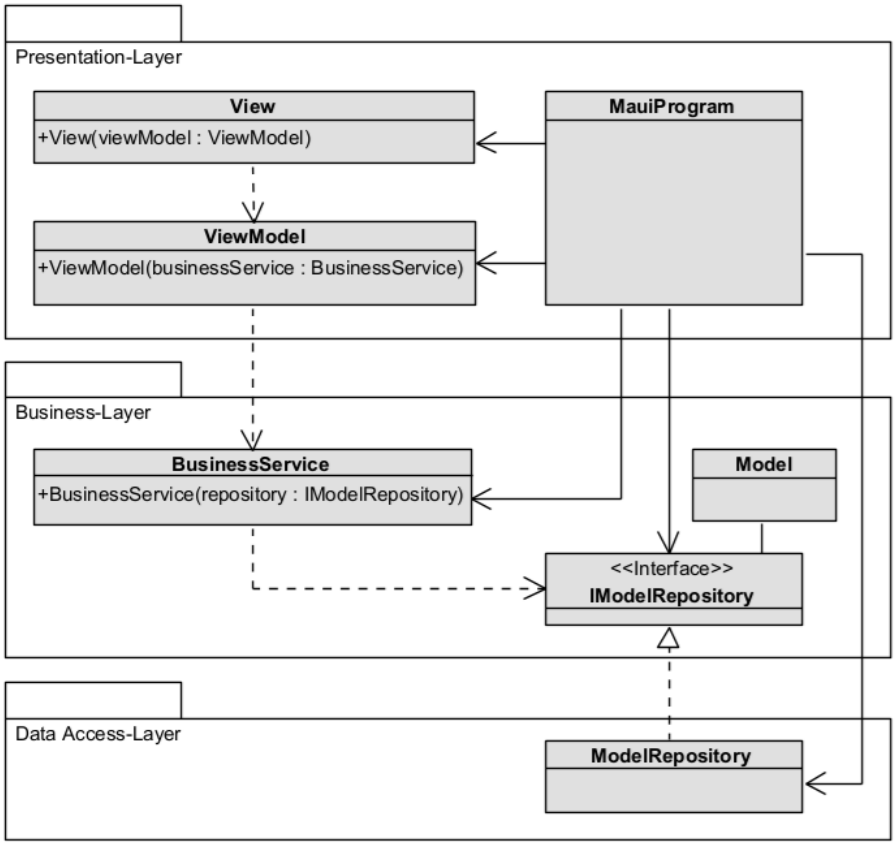
Omdat geen Threat Model is uitgevoerd zijn ook geen aanvullende securitymaatregelen genomen. Geen nadere actie nodig.

Project architectuur (C4 niveau 4)

Voor de desktop applicatie is een klassendiagram gemaakt, te zien in Figuur 5, **Fout! Verwijzingsbron niet gevonden.** en **Fout! Verwijzingsbron niet gevonden.**. De GroceryApp bestaat uit drie verschillende projecten: GroceryApp, GroceryApp.Business en GroceryApp.Data. Deze projecten zijn zo opgesteld zodat de juiste scheiding van verantwoordelijkheden wordt aangehouden volgens Clean Architecture architectuur.



Figuur 4 Afhankelijkheden tussen de lagen



Figuur 5 Klassendiagram van GroceryApp

Packages

Naam	Versie	Beschrijving
MAUI.Controls	8.0.100	.NET Multi-platform App UI (.NET MAUI) is een cross-platform framework om native mobiele en desktop applicaties met C# en XAML te maken. Door .NET MAUI te gebruiken, kan je apps developen die op Android, iOS, macOS, Tizen en Windows kunnen runnen op één codebase.
CommunityToolKit.MVVM	8.4.0	Helper package om te werken met MVVM.

Configuraties

Om een nieuwe repository toe te voegen aan de applicatie, moet er in de *MauiProgram.cs* een line toegevoegd worden, waarbij de *Model* en *Repository* vervangen moeten worden met de correct klasse namen.

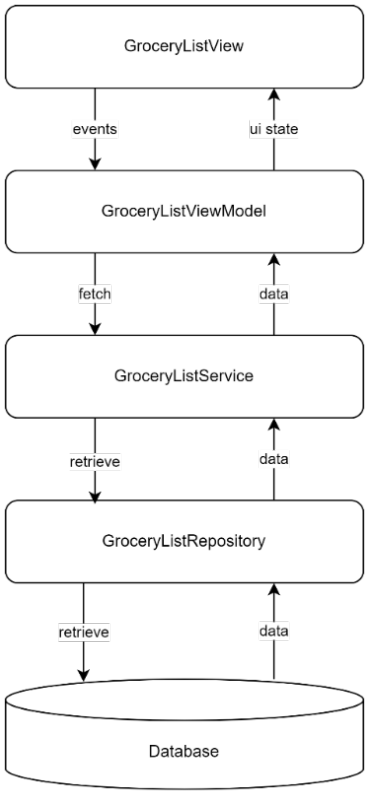
```
builder.Services.AddSingleton<IRepository<Model>, Repository>();
```

Dataflow

In applicaties is het belangrijk om duidelijk te begrijpen hoe gegevens (data) door de verschillende lagen van de applicatie stromen. Dit wordt ook wel de *dataflow* genoemd. Door deze datastroom gestructureerd op te bouwen, kunnen we de applicatie overzichtelijk, onderhoudbaar en uitbreidbaar houden.

In dit hoofdstuk bekijken we stap voor stap hoe data zich verplaatst van de opslag (de database) naar het scherm dat de gebruiker ziet (de view), en weer terug. We volgen de route die een gebruikersactie aflegt – zoals het openen van een boodschappenlijst – en hoe die via verschillende lagen, zoals de *ViewMode*l, *Service* en *Repository,* leidt tot het ophalen en tonen van gegevens.

Aan de hand van het lagenmodel van Figuur 6 maken we inzichtelijk welke laag welke rol speelt in dit proces en hoe ze samenwerken om de juiste data op het juiste moment beschikbaar te maken.



Figuur 6: Visualisatie van dataflow

Het diagram laat zien hoe een applicatie is opgebouwd in lagen, waarbij elke laag een eigen verantwoordelijkheid heeft. Dit principe, dat vaak wordt gebruikt in softwareontwikkeling, zorgt ervoor dat de code overzichtelijk en goed onderhoudbaar blijft.

In deze structuur is de bovenste laag de *GroceryListView*, oftewel de gebruikersinterface. Dit is wat de gebruiker ziet en waarmee hij of zij kan interacteren, zoals knoppen, lijsten of invoervelden. De *GroceryListView* ontvangt de gebruikersinterface-status van de *ViewModel* en stuurt gebruikersacties, zoals klikken of invoer, terug als gebeurtenissen (events).

GroceryListViewModel

De volgende laag is de *GroceryListViewModel*. Deze laag vangt de gebeurtenissen van de View op en bepaalt vervolgens wat er moet gebeuren. Hij verwerkt gebruikersacties, beheert de status van het scherm en haalt zo nodig gegevens op via de onderliggende servicelaag. De *ViewModel* is dus verantwoordelijk voor de logica die bepaalt hoe het scherm eruit moet zien op basis van de beschikbare gegevens en gebruikersacties.

GroceryListService

Onder de *GroceryListViewModel* bevindt zich de *GroceryListService*, ook wel de servicelaag of domeinlaag genoemd. Hierin zit de kernlogica van de toepassing. Deze laag bepaalt wat er precies moet gebeuren wanneer de *ViewModel* bijvoorbeeld om een boodschappenlijst vraagt. De service weet welke acties nodig zijn om aan de informatie te komen en schakelt hiervoor de repositorylaag in.

GroceryListRepository

De *GroceryListRepository* is de laag die verantwoordelijk is voor het daadwerkelijk ophalen van gegevens. Deze laag weet waar de gegevens vandaan moeten komen, bijvoorbeeld uit een lokale database of een externe API. De service roept de repository aan om data op te halen, en de repository levert deze gegevens terug aan de service.

Database

Tot slot is er de *Database*, waarin de gegevens uiteindelijk zijn opgeslagen. De repository maakt verbinding met deze database om gegevens op te halen of op te slaan. De database bevat bijvoorbeeld de boodschappenlijstjes die de gebruiker eerder heeft ingevoerd.

Samengevat zorgt deze gelaagde opbouw ervoor dat de verschillende onderdelen van de applicatie gescheiden blijven: de gebruikersinterface, de logica die bepaalt wat er moet gebeuren, en de opslag van gegevens. Dit maakt het geheel niet alleen beter te begrijpen, maar ook gemakkelijker te testen en aan te passen.

Auth

In dit hoofdstuk bekijken we hoe het inlogproces technisch is opgebouwd. We starten met een toelichting op het sequence diagram dat de stappen van het inlogproces weergeeft. Daarna bespreken we hoe databinding ervoor zorgt dat gebruikersinvoer gekoppeld wordt aan de logica in de view. Tot slot komt de password helper aan bod, die ondersteuning biedt bij het controleren en beveiligen van wachtwoorden.

Authenticatieflow

In dit hoofdstuk bekijken we hoe het inloggen met een gebruikersnaam en wachtwoord werkt. We laten zien wat er gebeurt als een gebruiker zijn gegevens invult en hoe die gecontroleerd worden. In Bijlage 1: Sequentie diagram inloggen zie je stap voor stap welke onderdelen meewerken en de onderlinge communicatie.

Verifiëren van het wachtwoord met de PasswordHelper

De methode *VerifyPassword* controleert of een ingevoerd wachtwoord overeenkomt met een eerder opgeslagen wachtwoord.

Wanneer een wachtwoord eerder is opgeslagen, is dat niet als platte tekst gebeurd, maar als een combinatie van een unieke "zoutwaarde" (salt) en een gehashte versie van het wachtwoord. Deze twee onderdelen zijn samengevoegd tot één string. Een *salt* is een willekeurige waarde die aan een wachtwoord wordt toegevoegd voordat het wordt gehasht, om identieke wachtwoorden unieke hashes te geven en bescherming te geven tegen aanvallen.

Bij het controleren van een wachtwoord wordt deze string eerst opgesplitst in de zoutwaarde en de originele hash. Daarna wordt het nieuwe, ingevoerde wachtwoord opnieuw gehashed met exact dezelfde zoutwaarde en instellingen. Ten slotte wordt de nieuwe hash vergeleken met de opgeslagen hash. Dit gebeurt op een veilige manier, zodat het systeem niet sneller reageert bij een fout wachtwoord (wat een beveiligingsrisico zou kunnen zijn).

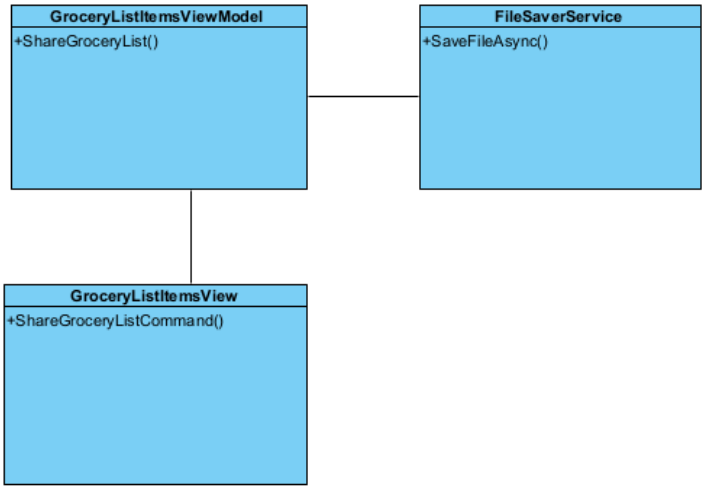
Als beide hashes exact gelijk zijn, betekent dit dat het ingevoerde wachtwoord klopt en wordt de gebruiker geverifieerd. Zo niet, dan wordt de toegang geweigerd.

Delen boodschappenlijst

Het delen van een boodschappenlijst is het opslaan van een boodschappenlijst als een tekstbestand.

Betrokken klassen

De *GroceryListView* ontvangt een *GroceryListViewModel*. De *GroceryListViewModel* heeft een methode *ShareGroceryList*. Deze methode heeft een annotatie *[RelayCommand]*. Dit zorgt ervoor dat deze methode beschikbaar is in de view als *ShareGroceryList**Command***. Maui genereert op basis van de annotatie deze methode.



Figuur 7 Betrokken klassen bij het delen van een boodschappenlijst

Bestanden opslaan

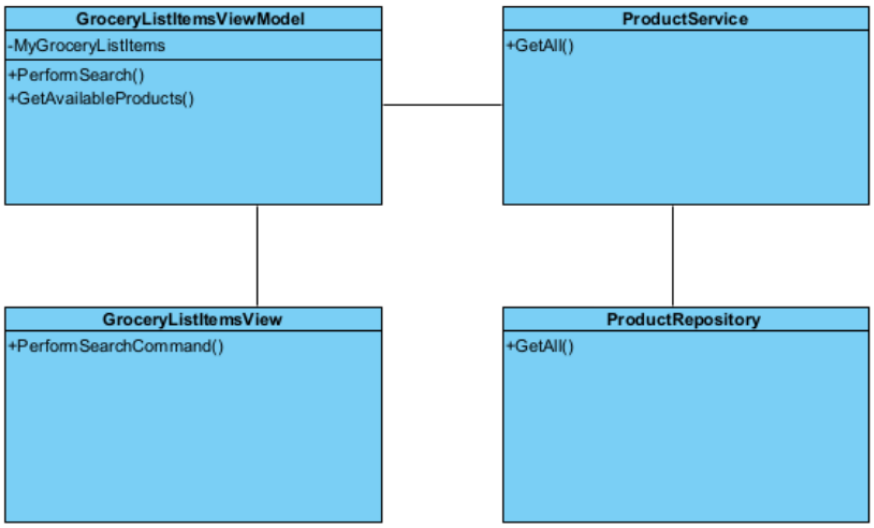
De *FileSaverService* is verantwoordelijk voor het opslaan van de content (String) in een bestand. Hiervoor wordt van de String een Byte array gemaakt die met behulp van een MemoryStream op de harde schijf wordt opgeslagen.

Operating system

Mac en Windows gaan niet hetzelfde om met het opslaan van bestanden. Daarom wordt er gecontroleerd op welk OS de methode *FileSaverService.SaveFileAsync* wordt aangeroepen. Dit wordt gecontroleerd door middel van *#if...*

Zoeken producten

Voor het zoeken van zijn een aantal klassen betrokken. Het is een standaard flow die gebruik maakt
van een View, ViewModel, Service en Repository. De betrokken klassen zijn te zien in Figuur 8.



Figuur 8 Betrokken klassen bij het zoeken van GroceryListItems

Betrokken klassen

Voor het zoeken van een producten maakt de *ProductService* gebruik van een LINQ query. Dit maakt het zoeken erg eenvoudig.

In de view is de methode *PerformSearchCommand* beschikbaar geworden door in het *ViewModel* een annotatie *RelayCommand* toe te voegen aan de methode *PerformSearch*.

State update

ViewModel zijn verantwoordelijk voor de state van de data. Het *GroceryListItemViewModel* heeft onder andere een lijst met *MyGroceryListItems*:

```
public ObservableCollection<GroceryListItem> MyGroceryListItems { get; set; } = [];
```

Door gebruik te maken van een *ObservableCollection* worden de elementen in de View die gebruik maken van deze data uit het *ViewModel* automatisch geupdate.

UC12 Categorieën

Deze use case breidt de bestaande GroceryApp uit met de mogelijkheid om producten te groeperen in categorieën.
Door middel van een koppeling tussen **Product** en **Category** kunnen gebruikers productcategorieën beheren en de bijbehorende producten raadplegen.
Deze functionaliteit ondersteunt de verdere uitbreiding van het productbeheer en maakt het mogelijk om gegevens gestructureerd te presenteren binnen de app.

De realisatie van UC12 bestaat uit drie hoofdonderdelen:

1. Het **model** en de **relatie** tussen Product en Category.
2. De benodigde **repositories** en **services** voor data-toegang.
3. De **views** en **viewmodels** voor weergave en interactie.

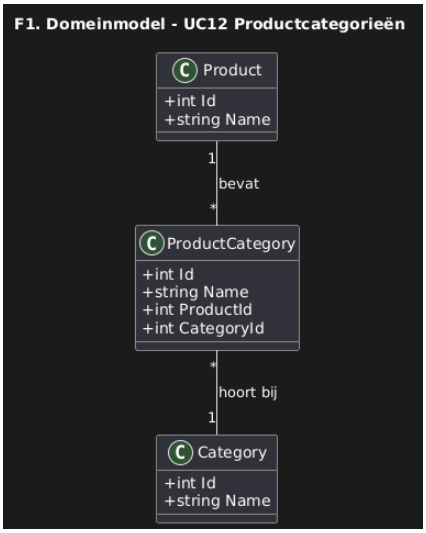
De architectuur bestaat uit de volgende lagen:

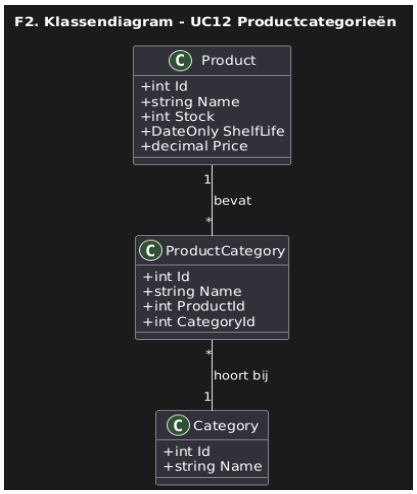
Laag	Functie	Betrokken componenten
Model (Core.Models)	Definieert de domeinstructuur van categorieën en de koppeling tussen producten en categorieën.	Product, Category, ProductCategory
Data (Core.Data.Repositories)	Bevat in-memory repositories die de gegevens leveren.	CategoryRepository, ProductCategoryRepository
Service (Core.Services)	Verzorgt de business-logica en validatie.	CategoryService, ProductCategoryService
ViewModel (App.ViewModels)	Beheert de state en interacties tussen UI en services.	CategoriesViewModel, ProductCategoriesViewModel
View (App.Views)	Presenteert de informatie aan de gebruiker.	CategoriesView, ProductCategoriesView

Navigatie verloopt via **Shell-routes**.
Wanneer een gebruiker in de CategoriesView op een categorie tikt, wordt de ProductCategoriesView geopend met de bijbehorende CategoryId als parameter.

UML

Figuur 1 laat zien welke onderdelen in het systeem horen bij deze use case.
Het model bevat de klassen *Product*, *Category* en *ProductCategory* en toont hoe ze met elkaar verbonden zijn.
De klasse *Product* is hier simpel gehouden, met alleen de belangrijkste eigenschappen (*Id* en *Name*).
Dit model is bedoeld om te laten zien **wat er in het systeem bestaat**, niet hoe het technisch werkt.
De koppeling *ProductCategory* laat zien dat één product bij meerdere categorieën kan horen en andersom.



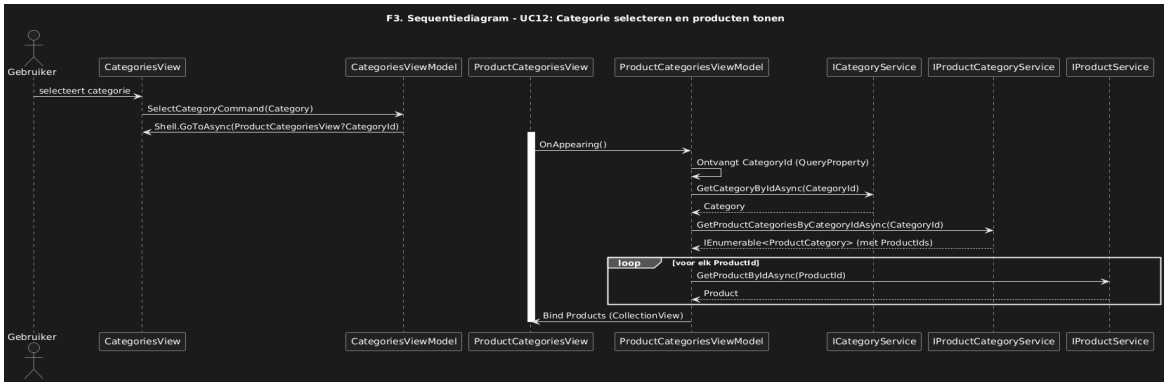


Klassendiagram

Figuur 2 laat zien hoe de klassen uit het domeinmodel in de code zijn uitgewerkt.
De klasse *Product* is hier uitgebreider, met velden zoals *Stock*, *ShelfLife* en *Price*.
Het klassendiagram gaat dus meer over **hoe** het systeem is gebouwd.
Hier zijn ook de datatypes en exacte relaties te zien tussen de klassen.
Zo geeft het klassendiagram een duidelijk beeld van de technische structuur van de GroceryApp.

Seuqentie diagram:

Figuur 3 laat stap voor stap zien wat er gebeurt wanneer de gebruiker een categorie kiest in de app.
Eerst kiest de gebruiker een categorie in het scherm *CategoriesView*.
De *CategoriesViewModel* reageert hierop en opent het scherm *ProductCategoriesView*, waarbij het ID van de gekozen categorie wordt meegegeven.
Daarna haalt de *ProductCategoriesViewModel* de bijbehorende categorie op via de *CategoryService* en zoekt alle producten die aan deze categorie zijn gekoppeld via de *ProductCategoryService*. Voor elk gevonden product wordt de informatie opgehaald met de *ProductService*.
Uiteindelijk toont de app de producten die bij de gekozen categorie horen in de lijst.



UC14 Prijzen Tonen

Deze use case breidt de bestaande GroceryApp uit met de mogelijkheid om **prijzen per product weer te geven** in het productoverzicht.
De gebruiker ziet in de lijstweergave (ProductView) voor elk product de naam, prijs, voorraad en THT-datum.

Doel

De gebruiker direct inzicht geven in de prijs van producten binnen de applicatie.

Betrokken klassen

De architectuur volgt het MVVM-patroon:

Laag	Functie	Componenten
Model (Core.Models)	Definieert producteigenschappen inclusief prijs	Product
Data (Core.Data.Repositories)	Levert data met prijzen	ProductRepository
ViewModel (App.ViewModels)	Beheert productlijst en data-binding naar de UI	ProductViewModel
View (App.Views)	Toont lijst van producten met prijzen in euro's	ProductView

Belangrijkste wijzigingen

- In de **Product-klasse** is een nieuw veld UnitPrice toegevoegd.
  De prijs wordt opgeslagen in **centen** om afrondingsfouten te voorkomen.
- De **PriceDisplay-property** converteert de prijs naar een **euro-string**, die in de UI getoond wordt.
- De **ProductRepository** vult de unitPrice in bij elk product.
- De **ProductView** bevat een extra kolom voor de prijs.

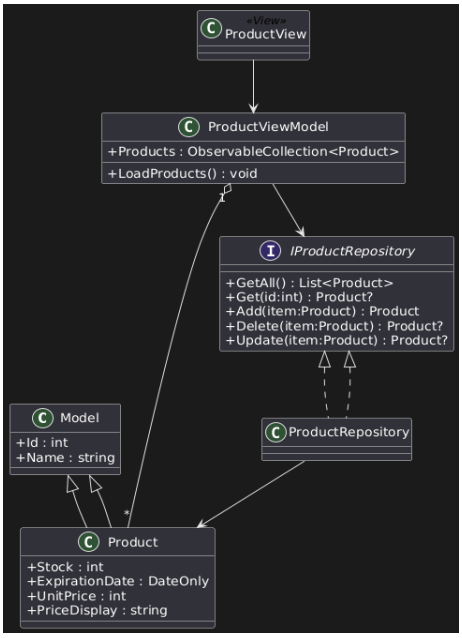
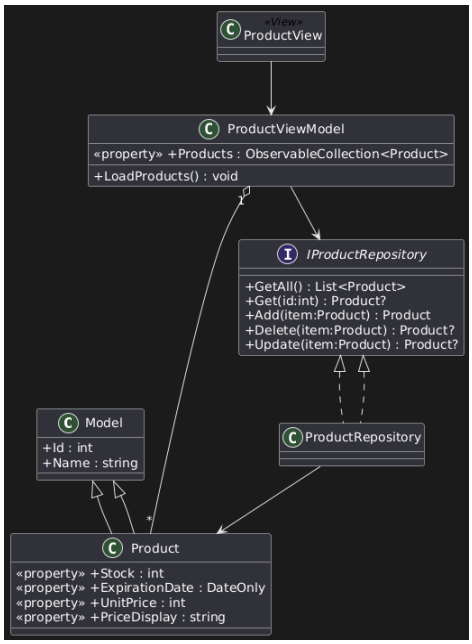
UML

Deze UML laat zien hoe de prijsfunctionaliteit technisch is opgebouwd binnen de GroceryApp.
De architectuur volgt het **MVVM-patroon**, waarin de View communiceert met de ViewModel en Repository om data op te halen.

- **ProductView** toont de productlijst inclusief prijs.
- **ProductViewModel** beheert de lijst van producten (ObservableCollection<Product>) en haalt data op via de repository.
- **IProductRepository** definieert de methoden voor data-toegang.
- **ProductRepository** implementeert deze interface en levert de producten met hun prijs.
- **Product** bevat de eigenschappen van een product, waaronder UnitPrice (in centen) en PriceDisplay (in euro's voor de UI).
- **Model** is de basisklasse met algemene velden zoals Id en Name.

Feedback Ismael (klasgenoot):

"Hey Ramazan, Je UML ziet er al goed uit, maar ik mis nog het label *property* bij je C#-eigenschappen. In C# gebruik je geen losse variabelen, maar properties met get en set. Volgens de HBO-ICT richtlijnen is het handig om dat ook in je diagram te laten zien, zodat duidelijk is dat het om properties gaat en niet om gewone velden. Voeg dus bij je attributen zoals *Stock*, *ExpirationDate*, *UnitPrice* en *PriceDisplay* het label *property* toe. Dan sluit je UML beter aan op je C# code en is meteen te zien hoe de klasse is opgebouwd."

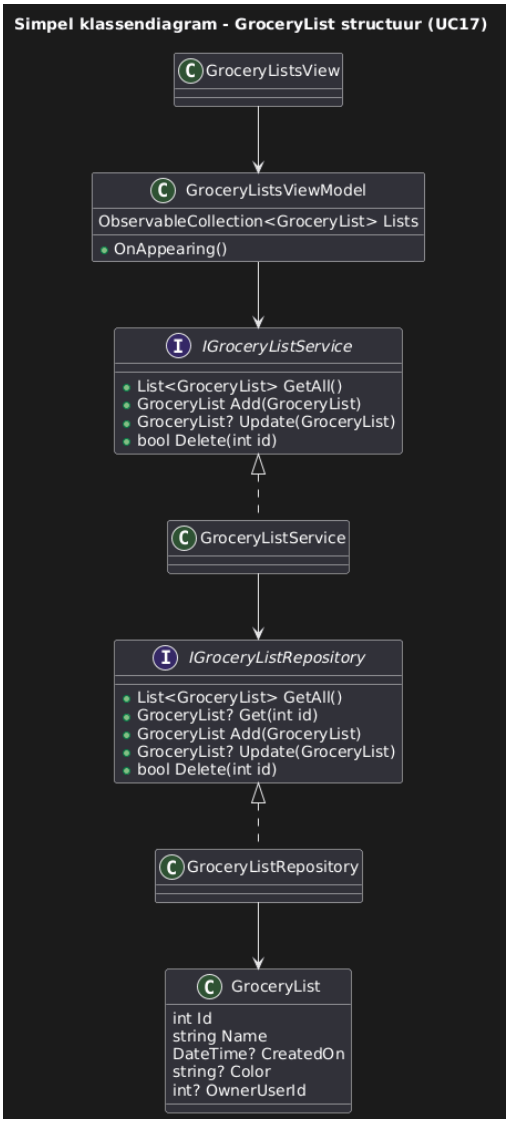
Voor	na
	

UC 17-18-19

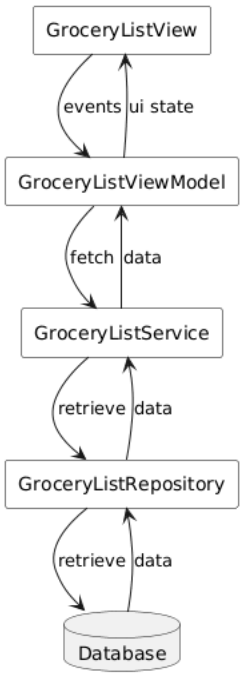
De diagrammen laten zien hoe de applicatie technisch is opgebouwd en hoe gegevens door de lagen heen stromen tijdens het opslaan van boodschappenlijsten (UC17–UC19).

Het **klassendiagram** toont de gelaagde structuur volgens het *MVVM- en Service-Repository-patroon*. Elke laag heeft een duidelijke verantwoordelijkheid:

- **GroceryListView** vormt de gebruikersinterface waarin lijsten worden weergegeven.
- **GroceryListsViewModel** verwerkt gebruikersacties en communiceert met de servicelaag.
- **GroceryListService** bevat de logica voor het aanmaken, bijwerken en ophalen van lijsten.
- **GroceryListRepository** verzorgt de communicatie met de database.
- **GroceryList** is het domeinmodel dat de gegevensstructuur van een boodschappenlijst weergeeft.



Figuur 6: Visualisatie van dataflow (UC17 - UC19)



Het **dataflowdiagram** laat de richting van de gegevens zien:

• Gebruikersacties worden vanuit de *View* als **events** naar de *ViewModel* gestuurd.
• De *ViewModel* vraagt via de *Service* om data (**fetch/retrieve**) en ontvangt vervolgens de resultaten (**data**).
• De *Repository* haalt de gevraagde informatie op uit de **Database**, waarin alle boodschappenlijsten en bijbehorende items centraal zijn opgeslagen.
• Ten slotte geeft de *ViewModel* de verwerkte gegevens terug aan de *View* als **UI-state**, zodat de gebruiker direct het resultaat ziet.

Persistentie

Vanaf UC17 wordt data opgeslagen via een database. Dit zorgt ervoor dat de data, ook na het afsluiten van de applicatie, blijft bestaan. Hiervoor wordt een SQLite-databasebestand gebruikt dat binnen je project wordt opgeslagen. SQLite is een databaseformaat waarmee je geen aparte databaseserver hoeft te starten. Alle database-acties worden direct uitgevoerd op dit bestand.

Om te kunnen verbinden aan deze database, moet het systeem weten waar het de database kan vinden. Dit gebeurt aan de hand van een *connectionstring*, die is opgenomen in het *appsettings.json* document. Zodra de applicatie gaat builden, zal de applicatie deze waarde ophalen en daarna de files (database en *appsettings.json*) in de juiste build-mappen plaatsen.

De interactie met de database wordt gedaan door in de *repositories* de models aan te roepen. De models weten, aan de hand van de eerdergenoemde *connectionstring* hoe ze met de database moeten verbinden. Door de models te gebruiken, hoef je niet handmatige SQL-queries uit te voeren.

Operating systems

Voor MacOS is er een andere opslaglocatie vereist en is de manier van opslag anders dan op andere platformen. Hiervoor is gebruik gemaakt van #if MACCATALYST om de juiste scheiding aan te geven binnen de applicatie in zowel *DatabaseConnection.cs* en *ConnectionHelper.cs*

Andere database

Als je toch een andere database wilt gebruiken, kan dit gedaan worden door de *connectionstring* in *appsettings.json* aan te passen naar de database type van jouw keuze.

Figuren

Bibliografie

Microsoft. (2023). *.NET documentation*. Opgehaald van Microsoft: https://learn.microsoft.com/en-us/dotnet/

NUnit. (2023). *NUnit*. Opgehaald van NUnit Documentation Site: https://docs.nunit.org/

Bijlage 1: Sequentie diagram inloggen