Examentraining 1

Realiseren

hoofdstuk

3

Veel op veel relaties

Algemene informatie

Onderwerp	Veel op veel relaties in MVC
Leerdoel(en)	1. De student kan een model bouwen wat een veel op veel relatie implementeert in code 2. De student kan een controller bouwen wat voor een model met een veel op veel relatie de CRUD correct uitvoert. 3. De student kan een view bouwen die een model met een veel op veel relatie correct weergeeft en laat bewerken door de gebruiker.
Vereiste voorkennis	1. De student heeft een voorkennis van MVC 2. De student heeft een voorkennis van SQL en kent de begrippen 1 op veel en veel op veel relatie.
Kwalificatiedossier	☐ B1-K1-W1: Plant werkzaamheden en bewaakt de voortgang ☐ B1-K1-W2: Ontwerpt software ☒ B1-K1-W3: Realiseert (onderdelen van) software ☐ B1-K1-W4: Test software ☐ B1-K1-W5: Doet verbetervoorstellen voor de software ☒ B1-K2-W1: Voert overleg ☐ B1-K2-W2: Presenteert het opgeleverde werk ☐ B1-K2-W3: Reflecteert op het werk

Inhoudsopgave

Introductie

Je hebt in alle voorgaande realiseren lessen geleerd hoe je een goedwerkende applicatie kunt bouwen waarin de gebruiker alle beschreven functionaliteiten met een op veel relaties kan uitvoeren.

In sommige gevallen zit er in je data analyse en dus in je database een veel op veel relaties. Denk bijvoorbeeld aan modules en klassen. Een module wordt gevolgd door meerdere klassen en een klas volgt meerdere modules.

Inhoud

1. Model

Bij het implementeren van je modellen moet je goed kijken wat je wilt weergeven. Alleen die relatie ga je in het model inbouwen. Als je namelijk in allebei de entiteiten je veel op veel relatie gaat maken krijg je te maken met "*circular referencing*".

Circular referencing kan problemen veroorzaken, zoals een oneindige lus of vastlopen van het programma. Als entiteit 1 entiteit 2 ophaalt, entiteit 2 entiteit 1 ophaalt enzovoort, kan het programma blijven verwijzen zonder te stoppen.

Dit kan problemen veroorzaken, zoals:

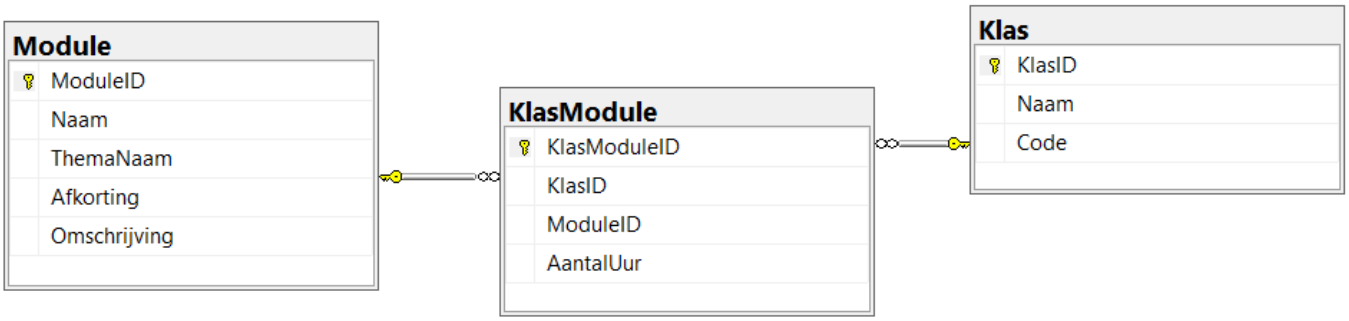
- Oneindige lus: Het programma kan vastlopen in een herhaalde cyclus zonder verder te gaan. Dit kan de prestaties van het programma vertragen of het laten vastlopen.
- Geheugenlekken: Circular referencing kan geheugenlekken veroorzaken als referenties niet op de juiste manier worden vrijgegeven. Dit kan resulteren in onnodig geheugengebruik.
- Onvoorspelbaar gedrag: Circular referencing kan leiden tot inconsistent of incorrect gedrag van het programma. Hierdoor kunnen resultaten onbetrouwbaar worden.

Om deze problemen te voorkomen, moet je zorgvuldig nadenken over de manier waarop je de gegevens organiseert. Soms moet je de relatie tussen de entiteiten herstructureren, bijvoorbeeld door een extra entiteit toe te voegen of de relatie te vereenvoudigen om circular referencing te vermijden.

Voor nu is het van belang dat we dus geen verwijzing in alle twee de entiteiten bouwen! Is dit toch nodig, kies er dan voor om twee aparte modellen van iedere entiteit te maken!

Voorbeeld:

Neem het volgende ERD.



Die zou resulteren in de volgende modellen:

```
internal class ModuleModel
{
    public int ModuleID { get; set; }
    public string Naam { get; set; }
    public string Themanaam { get; set; }
    public string Afkorting { get; set; }
    public string Omschrijving { get; set; }
}
```

```
internal class KlasModuleModel
{
    public int KlasModuleID { get; set; }
    public ModuleModel Module { get; set; }
    public int AantalUur { get; set; }
}
```

```
internal class KlasModel
{
    public int KlasID { get; set; }
    public string Naam { get; set; }
    public string Code { get; set; }
    public List<KlasModuleModel> Modules { get; set; }
}
```

Zoals je ziet is in deze oplossing er alleen een link van KlasModel via KlasModuleModel naar ModuleModel. In ModuleModel zit niet dezelfde List<KlassenModuleModel> opgenomen. Als je dit wel doet krijg je namelijk last van "Circular referencing" zoals hierboven beschreven. Bij het inladen van een klas laat je dan namelijk alle bijbehorende *KlasModuleModel* objecten in, die op hun buurt weer het bijbehorende *ModuleModel* object inladen en die zou dan weer alle bijbehorende *KlasModuleModel* objecten in, die op hun buurt weer het bijbehorende *KlasModel* object inladen en die zou dan weer alle bijbehorende *ModuleModel* object inladen en die zou dan weer alle bijbehorende *KlasModuleModel* objecten in, die op hun buurt weer het bijbehorende *KlasModel* object inladen en die zou dan weer alle … enzovoort. Je snapt dat dit maar door en door gaat.

Heb je wel van beide kanten een lijst nodig (dus wil je in je applicatie van een module laten zien welke klassen deze module volgen EN wil je van een klas laten zien welke modules de klas volgt), dan moet je zorgen dat je middels aparte Models de "Circular referencing" oplost.

Dat zou als volgt kunnen:

Je maakt eerst twee klassen die geen verwijzing bevatten.

```
internal class ModuleModel_NoRef
{
    public int ModuleID { get; set; }
    public string Naam { get; set; }
    public string Themanaam { get; set; }
    public string Afkorting { get; set; }
    public string Omschrijving { get; set; }
}
```

```
internal class KlasModel_NoRef
{
    public int KlasID { get; set; }
    public string Naam { get; set; }
    public string Code { get; set; }
}
```

Zoals je ziet is deze klasse gelijk aan die hierboven alleen ziet er geen attribuut List in!

Vervolgens pas je het koppelmodel aan, zodat deze verwijst naar de modellen zonder verwijzing:

```
internal class KlasModuleModel
{
    public int KlasModuleID { get; set; }
    public ModuleModel_NoRef Module { get; set; }
    public KlasModel_NoRef Klas { get; set; }
    public int AantalUur { get; set; }
}
```

En als laatste pas je de 'standaard' KlasModel en ModuleModel aan:

```
internal class ModuleModel
{
    public int ModuleID { get; set; }
    public string Naam { get; set; }
    public string Themanaam { get; set; }
    public string Afkorting { get; set; }
    public string Omschrijving { get; set; }
    public List<KlasModuleModel> Klassen { get; set; }
}
```

```
internal class KlasModel
{
    public int KlasID { get; set; }
    public string Naam { get; set; }
    public string Code { get; set; }
    public List<KlasModuleModel> Modules { get; set; }
}
```

Zoals je ziet kunnen ze beide een List<> attribuut bevatten, zonder dat we bang hoeven zijn voor een "Circular referencing".

Echter maken we nu wel een ander probleem. Want als een Klas of Module een extra attribuut zou krijgen, of een attribuut veranderd of verdwijnt, dan moeten we dat voortaan op twee plekken aanpassen. Namelijk in het 'volledige' model en in de _NoRef variant. Dat is niet wenselijk, want is feitelijk vragen om problemen! Daarom gaan we gebruik maken van een OOP eigenschap genaamd overerving.

Het principe van overerving is een belangrijk concept in objectgeoriënteerd programmeren (OOP) dat helpt bij het organiseren en hergebruiken van code. In OOP is overerving een manier om nieuwe klassen te creëren op basis van bestaande klassen, waarbij de nieuwe klassen automatisch eigenschappen en gedrag erven van de bestaande klassen.

Stel je voor dat je een programma moet schrijven voor een dierentuin. Je kunt beginnen met het definiëren van een algemene klasse genaamd "Dier", die gemeenschappelijke eigenschappen en gedragingen van dieren bevat, zoals naam, leeftijd en geluid maken. Deze klasse kan er bijvoorbeeld zo uitzien:

```
public class Dier
{
    private String naam;
    private int leeftijd;

    public Dier(String naam, int leeftijd)
    {
        this.naam = naam;
        this.leeftijd = leeftijd;
    }

    public void maakGeluid()
    {
        MessageBox.Show("Het dier maakt geluid.");
    }
}
```

Nu wil je specifieke dieren, zoals een "Leeuw" en een "Olifant", vertegenwoordigen in je programma. Deze dieren hebben unieke eigenschappen en gedragingen, maar delen ook veel kenmerken met het algemene dier. In plaats van de klasse voor elk specifiek dier afzonderlijk te schrijven, kun je overerving gebruiken om nieuwe klassen te maken op basis van de "Dier"-klasse.

Hier is een voorbeeld van een subklasse genaamd "Leeuw" die de klasse "Dier" erft:

```
public class Leeuw : Dier
{
    private int manen;

    public Leeuw(string naam, int leeftijd, int manen)
        : base(naam, leeftijd)
    {
        this.manen = manen;
    }

    public void Brul()
    {
        Console.WriteLine("De leeuw brult!");
    }

    // Extra methodes en eigenschappen specifiek voor de leeuw
}
```

De klasse "Leeuw" erft alle eigenschappen en methoden van de klasse "Dier". In C# wordt de erfenis aangegeven met het sleutelwoord :, gevolgd door de naam van de basisklasse. De constructor van de basisklasse wordt aangeroepen met het sleutelwoord base, gevolgd door de argumenten die aan de constructor worden doorgegeven. Hierdoor kan een "Leeuw"-object dezelfde eigenschappen hebben als een "Dier"-object, zoals naam en leeftijd, en kan het ook de methode maakGeluid() van de "Dier"-klasse gebruiken. Daarnaast heeft de "Leeuw"-klasse ook zijn eigen specifieke eigenschappen, zoals het aantal manen, en zijn eigen specifieke methode brul().

Met overerving kunnen er meerdere niveaus van hiërarchie worden gecreëerd. Bijvoorbeeld, je kunt een andere subklasse maken genaamd "Olifant" die ook de klasse "Dier" erft, maar met zijn eigen specifieke eigenschappen en methoden.

Door overerving te gebruiken, kun je de code hergebruiken, de structuur van je programma organiseren en de relaties tussen klassen op een logische manier weergeven. Het stelt je in staat om algemene eigenschappen en gedragingen te definiëren in een basisklasse en specifieke kenmerken toe te voegen in afgeleide klassen.

Als wij in ons voorbeeld gebruik gaan maken van overerving om dubbele code te voorkomen zien de KlasModel en ModuleModel er als volgt uit:

```
internal class ModuleModel: ModuleModel_NoRef
{
    public List<KlasModuleModel> Klassen { get; set; }
}
```

```
internal class KlasModel: KlasModel_NoRef
{
    public List<KlasModuleModel> Modules { get; set; }
}
```

Zoals je ziet zijn de attribute die in de _NoRef varianten zitten verdwenen en is aangegeven (middels de :) dat onze modellen deze attributen erven van hun _NoRef varianten.

2. Controller

Voor het implementeren van een veel op veel relatie in je controller moet je wat aanpassingen doen. We gaan er hiervan uit dat je de standaard methodes in een controller kunt maken en beschrijven hier dus alleen de aanpassingen nodig voor een veel op veel relatie.

Read

Als je de "read" methode aanpast, betekent dit dat je het attribuut "List<>" van een object ook moet vullen. Voor elk object wat je maakt (elke keer dat je "hoofdquery" een resultaat rij oplevert), moet je een tweede query naar de database sturen om de entiteiten voor de lijst op te halen. Standaard kan dit niet, want onze connectie met de SQL server staat slechts 1 uitgevoerde query tegelijk toe. Dus … moeten we onze connectie aanpassen. Die doe je door het volgende toe te voegen aan je connectiestring: `MultipleActiveResultSets=true`

Je maakt voert nu natuurlijk een tweede query uit, middels een tweede SqlCommand en maakt ook een tweede SqlDataReader middels een ExecuteReader methode van dit tweede SqlCommand object.

Je krijgt dus een loop in een loop.

Create

Voor een CREATE methode moet je nu natuurlijk ook in twee tabellen nieuwe data invoegen. Allereerst in de tabel van je hoofdentiteit, namelijk de entiteit die het List<> attribuut bevat. Dus bijvoorbeeld in de Klas tabel. Vervolgen moet je de List<> objecten aanmaken, ofwel de rijen in je koppeltabel toe te voegen. Het probleem waar je hierbij tegenaan gaat lopen is dat je in je code nog geen waarde hebt voor het Id van je hoofdentiteit.

Stel je maakt een nieuwe klas aan waaraan je modules toevoegt. Dan heb je alle info van je klas (behalve je KlasID, want die bepaald de database door de IDENTITY functie op deze kolom) en je weet welke KlasModulesModel objecten (Module en ook aantal uren zijn bekend) hieraan toegevoerd moeten worden. Om dit echter in de database op te kunnen slaan heb ik in mijn koppeltabel ook een KlasID nodig… Dit ID kun je uit de database ophalen met de volgende query:

```
SELECT Ident_Current('<naam tabel>') AS <zelf te kiezen naam>
```

Dus in ons voorbeeld met de volgende query:

```
SELECT Ident_Current('Klant') AS KlantID
```

Deze query levert je de laatst uitdeelde waarde van je identity kolom. Je moet dus eerst een nieuwe klant maken en deze waarde daarna uitlezen. Met deze waarde kun je de KlasModulesModel object nu verder vullen (je kan het KlantID invullen) en vervolgens kun je ze opslaan zoals je al vaker gedaan hebt (met een INSERT statement).

Delete

Voor een DELETE methode moet je nu natuurlijk ook in twee tabellen data verwijderen. Als je het item van een hoofdentiteit verwijderd, moeten alle koppeling naar deze entiteit natuurlijk ook verwijderd worden.

Gelukkig is dat heel makkelijk te realiseren! Daar kan onze database namelijk mee helpen. Dit doe je door bij het aanmaken van de relatie tussen de tabel van de entiteit en de koppeltabel (de Foreign key) de optie ON DELETE CASCADE toe te voegen.

In ons voorbeeld van klassen en modules zouden we de koppeltabel dan als volgt aanmaken:

```
CREATE TABLE KlasModule (
  KlasModuleID INT PRIMARY KEY,
  KlasID INT,
  ModuleID INT,
  AantalUur INT,
  FOREIGN KEY (KlasID) REFERENCES Klas (KlasID) ON DELETE CASCADE,
  FOREIGN KEY (ModuleID) REFERENCES Module (ModuleID) ON DELETE CASCADE
);
```

Hierdoor verdwijnen de koppelingen naar een klas als we een klas weggooien en ook verwijzingen naar een module als we een module weggooien.

Update

Het aanpassen van de UPDATE is het lastigste. Je hebt immers een bestaande entiteit waar je verwijzingen naar hebt. Deze verwijzingen kunnen zelf veranderen (in ons voorbeeld zou je bijvoorbeeld het AantalUur kunnen aanpassen), er kunnen nieuwe verwijzingen bijkomen (in ons voorbeeld kan je een nieuwe module aan een klas koppelen) of de gebruiker kan een verwijzing weghalen (in ons voorbeeld haal je een module bij een klas helemaal weg).

Dit betekent concreet dat je Update methode in de controller van je hoofdentiteit, namelijk de entiteit die het List<> attribuut bevat, zowel nieuwe verwijzing moet kunnen toevoegen, bestaande moet kunnen aanpassen en nieuwe moet kunnen maken. Maar hoe weet hij nu wat hij met welke verwijzing moet doen?

Dit kunnen we op een aantal manieren oplossen. Een manier is om aan het object van de verwijzing een status toe te voegen. In ons voorbeeld zou er dat zo uit kunnen zien:

```
internal class KlasModuleModel
{
    public int KlasModuleID { get; set; }
    public ModuleModel_NoRef Module { get; set; }
    public KlasModel_NoRef Klas { get; set; }
    public int AantalUur { get; set; }
    public byte Status { get; set; }
}
```

Je kan dan afspreken (en desnoods in commentaar erbij vermelden) dat status 0 betekent dat het een nieuw object is, status 1 dat het een aangepast object is en status 3 dat het een verwijderd object is. Heel duidelijk is dat natuurlijk niet.

Je kan het ook implementeren met een zogeheten enum variabele type. In C# is een enum (afkorting voor enumeratie) een datatype waarmee je een set van benoemde constanten kunt definiëren. Een enum kan handig zijn wanneer je een beperkt aantal mogelijke waarden wilt vertegenwoordigen, zoals dagen van de week, maanden, kleuren, enzovoort.

Hier is een voorbeeld van een enum voor dagen van de week:

```
public enum DagVanDeWeek
{
    Maandag,
    Dinsdag,
    Woensdag,
    Donderdag,
    Vrijdag,
    Zaterdag,
    Zondag
}
```

In dit voorbeeld definieert de enum genaamd DagVanDeWeek de mogelijke waarden voor dagen van de week. Elke waarde wordt gescheiden door een komma en staat standaard op volgorde, beginnend met 0 voor de eerste waarde (in dit geval "Maandag"). Je kunt deze waarden gebruiken als constanten in je programma.

Bijvoorbeeld, je kunt een variabele van het enum-type maken en toewijzen aan een van de mogelijke waarden:

```
DagVanDeWeek dag = DagVanDeWeek.Dinsdag;
```

Je kunt de enum-waarden vergelijken, doorgeven als parameters aan methodes en switch-statements gebruiken om verschillende acties uit te voeren op basis van de waarde van het enum.

Enums bieden een gestructureerde en betekenisvolle manier om een set van gerelateerde constanten te definiëren. Ze helpen om de leesbaarheid van de code te vergroten en mogelijke fouten te verminderen door alleen de toegestane waarden toe te staan.

In ons voorbeeld zouden we de enum als volgt kunnen inzetten:

```
enum StatusObject {Nieuw, Aangepast, Verwijderd };

internal class KlasModuleModel
{
    public int KlasModuleID { get; set; }
    public ModuleModel_NoRef Module { get; set; }
    public KlasModel_NoRef Klas { get; set; }
    public int AantalUur { get; set; }
    public StatusObject Status { get; set; }

    public KlasModuleModel()
    {
        // Default krijgt een object de status Nieuw
        Status = StatusObject.Nieuw;
    }
}
```

 Je kunt nu **oefening 3.1** maken.

3. View

Om in je view een veel op veel relatie te kunnen weergeven moet je in het detail scherm van je hoofdentiteit, namelijk de entiteit die het List<> attribuut bevat, een listview moet opnemen. Deze listview toont de inhoud van de lijst die in de entiteit zit.

Inhoud van een List in een Listview plaatsen is bekende stof. Die gaan we hier niet opnieuw behandelen. Zie daarvoor de readers van thema 7 en thema 8. Er is hier natuurlijk 1 uitzondering. Je vult de listview alleen met elementen met de status Nieuw of Aangepast. Verwijderde items tonen we niet!

Naast de listview (bij voorkeur rechts) neem je drie knoppen op. Een knop om een item toe te voegen, een knop om een item aan te passen en een knop om een item weg te halen.

Knop toevoegen opent een nieuw scherm waarin de gebruiker de waarde van het nieuwe object kan invullen. Dit object maakt je aan met status Nieuw.

Knop aanpassen opent een nieuw scherm waarin de gebruiker de waarde van het bestaande object kan aanpassen. Dit object geef je vervolgens status Aangepast.

Knop verwijderen past de status van het geselecteerde object aan naar status Verwijderd.

Je kunt nu **oefening 3.2** maken.