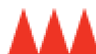
Basis standalone

Realiseren

hoofdstuk

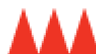
8

Toevoegen en verwijderen van gegevens



Algemene informatie

Onderwerp	Toevoegen en verwijderen van gegevens
Leerdoel(en)	1. De student kent het SQL DELETE commando en kan het toepassen. 2. De student kent het SQL INSERT commando en kan het toepassen. 3. De student kan de onderdelen Delete en Create programmeren in een Controller 4. De student kan de onderdelen Verwijderen en toevoegen programmeren in de View, gebruikmakend van het Model en de controller.
Vereiste voorkennis	1. Student kent de stof uit readers 1 t/m 7 van thema 7.
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

Algemene informatie 2
Inhoudsopgave 3
Introductie 4
Inhoud 4
Gegevens verwijderen 4
- Functionaliteit Verwijderen knop 4
- Verwijderen van gegevens 6
- Controller aanvullen 7
  - Using System.Data.SqlClient 8
  - Connectionstring via ConfigurationManager 8
  - SQLConnection / Using 8
  - SQLCommand 9
  - ExecuteNonQuery 10
- Foutafhandeling 10
  - Try catch 11

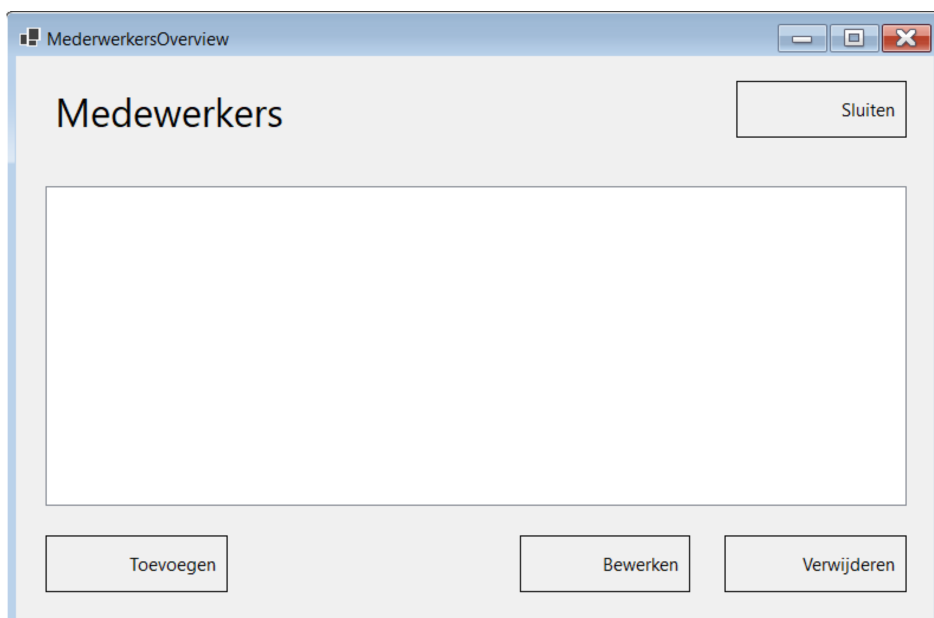
Gegevens toevoegen 12
- Functionaliteit Toevoegen knop 12
- View bouwen 12
- Toevoegen van gegevens 15
- Controller aanvullen 16
  - Using System.Data.SqlClient 16
  - Connectionstring via ConfigurationManager 16
  - SQLConnection / Using 16
  - SQLCommand 17
  - ExecuteNonQuery 18
- View bouwen – deel 2 19

Introductie

Binnen een applicatie maak je gebruik van gegevens die je wilt weergeven, toevoegen, aanpassen en/of verwijderen. In de praktijk wordt deze data opgeslagen in een database. De data die in de database zit in niet statisch, maar verandert. Er kan data bijkomen en data weggegooid worden. Hoe je dat vanuit onze applicatie moet bouwen behandelen we in deze reader.

Inhoud

In de reader van hoofdstuk 6 kun je lezen hoe je een eerste opzet van een view (ofwel een weergave/scherm) maakt. Dit scherm heb je opgebouwd volgens onderstaande lay-out en je hebt geleerd hoe je ervoor kunt zorgen dat de listview gevuld wordt met gegevens uit de database. In deze reader gaan we stap voor stap kijken hoe we de functionaliteit achter de 'Verwijderen' en 'Toevoegen' knop moeten realiseren.

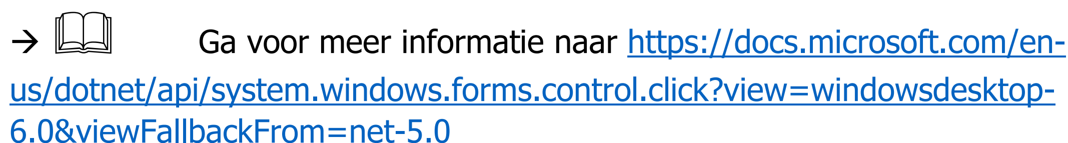


Gegevens verwijderen

Functionaliteit Verwijderen knop

Nadat we in onze applicatie het overzichtsscherm voor een entiteit hebben geopend, moeten alle records natuurlijk zichtbaar worden. Vervolgens moet een gebruiker een record/entiteit kunnen selecteren en verwijderen. Dat doet hij door op de verwijderknop te drukken.

In Visual studio kunnen we code die uitgevoegd moet worden als we bijvoorbeeld op een knop klikken programmeren door in het ontwerp van het formulier dubbel te klikken op de knop. Technisch gezien maakt Visual Studio voor jouw een methode die hij direct koppelt aan het Click event van de knop.



→ Ga voor meer informatie naar https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.control.click?view=windowsdesktop-6.0&viewFallbackFrom=net-5.0

Als je de naamgeving van de knop gedaan hebt zoals in reader 6 is uitgelegd maakt hij de volgende methode voor je aan:

```
private void btnDelete_Click(object sender, EventArgs e)
{

}
```

In deze methode moeten we nu de code gaan plaatsen die het geselecteerde element weggooit.

Het geselecteerde element kunnen we uit de listview halen. Een Listview heeft namelijk een argument genaamd SelectedItems. Dit argument bevat een lijst (in de beschrijving in Visual studio noemen ze het een Collection) van alle geselecteerde items.

Let op! De lijst met geselecteerde items kan GEEN, 1 of zelfs meerdere item(s) bevatten! Meerdere items kan alleen als je de instelling `MultiSelect` op `true` staat. Standaard staat deze niet op true en in onze applicatie zetten we hem ook niet op true. Bij ons kan er dus alleen geen of 1 element in de lijst staan.

We weten nu dus nog niet of er een element is geselecteerd. Dat moeten we controleren, want alleen als er één element is geselecteerd gaan we aangeven dat we dit element willen verwijderen. Controleren of er een element in een lijst zit, kun je heel simpel door op te vragen hoeveel elementen er in de lijst zitten. Dat doe je met het attribuut `Count`. Dit argument geeft het aantal elementen in de lijst terug. Geen elementen? Dan is het resultaat 0. Dus:

```
if (lvEmployees.SelectedItems.Count == 1)
{

}
```

Natuurlijk kan je een andere check inbouwen (>0, <> 0 etc.). Maar in ieder geval stel je hiermee vast dat de gebruiker iets geselecteerd heeft.

Nu gaan we op zoek naar de entiteit die we willen verwijderen. In het geval van het voorbeeld, welk EmployeeModel moet nu worden weggegooid? Dit model kunnen we ophalen, want bij het vullen van de listview hebben we het model, ofwel het object, de variabele opgeslagen in het Tag attribuut. Zie reader hoofdstuk 7 om dit nog eens na te lezen.

Het Tag attribuut is van het type object. Het datatype object is een zogenaamde baseclass. Andere objecten zijn afgeleid (inherited) van deze baseclass. Meer over inheritence bij Object Oriented Programming (OOP) kun je hier lezen.

→ Ga voor meer informatie naar https://www.theitstuff.com/what-is-inheritance-in-oop

Voor nu is het van belang om te weten dat je het object wat in Tag zit kunt 'omzetten' naar ons model. We zetten feitelijk een andere bril op en kijken op een andere manier naar de opgeslagen variabele dit in het Tag attribuut zit. Daardoor zien we ineens dat het ons model is 😜. Dat heet in C# type casten. Dat doen we in code als volgt:

```
EmployeeModel employee =
(EmployeeModel)lvEmployees.SelectedItems[0].Tag;
```

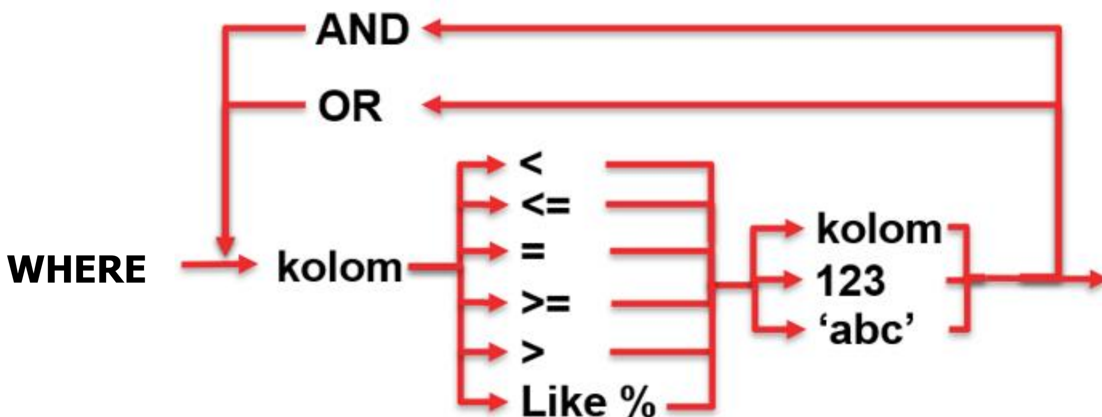
We weten nu dus dat de gebruiker een item geselecteerd heeft en we hebben het object hiervan opgehaald uit de listview. Dan moeten we het nu gaan verwijderen. Verwijderen betekent natuurlijk dat we het geselecteerde item uit de database moeten verwijderen. Interactie met de database loop via de controller. Dat betekent dus dat we in onze controller een methode moeten maken om een element uit de database te verwijderen. Als we deze methode gemaakt hebben kunnen we die als volgt aanroepen:

```
employeeController.Delete(employee);
```

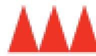
Verwijderen van gegevens

In thema 4 hebben jullie voor het eerst kennis gemaakt met SQL-commando's. Een van de commando's die jullie daar hebben leren gebruiken is het DELETE statement. Dit (DML-statement) wordt gebruikt voor het verwijderen van gegevens uit een SQL-database. Elk statement heeft een syntax. Dit is de schrijfwijze van het statement. Hieronder staat de vereenvoudigde syntax van het delete-statement weergegeven.

DELETE → **FROM** → **Tabel**



Wees met deze query wel heel voorzichtig: een weggegooid record komt niet zomaar weer terug. Natuurlijk zijn er mogelijkheden als er een goede backup van een database is, maar dit levert sowieso heel veel problemen op!



Als je al iets wil weggooien (in praktijk krijgen 'afgeschreven records' vaak gewoon een andere status of komen in een andere tabel) dan zul je een `WHERE` moeten gebruiken.

Voor:

196	2540	Sweet	Melodie	Hayward	1996-01-11	15
197	2541	Edwards	Gage	Redondo Beach	1995-10-12	15
198	2542	Small	Hollee	Hot Springs	1991-12-11	3
199	2543	Perez	Isabella	Temecula	1993-11-28	13
200	2544	Edwards	Chloe	Columbus	1992-04-16	10

Query executed successfully. | DESKTOP-PDT5EKK (15.0 RTM) | DESKTOP-PDT5EKK\robwes... | Leerlingen | 00:00:00 | 200 rows

```
-- leerling nr. 2543 weggooien.
DELETE
FROM leerling
WHERE ll_nr = 2543;
```

Na:

196	2540	Sweet	Melodie	Hayward	1996-01-11	15
197	2541	Edwards	Gage	Redondo Beach	1995-10-12	15
198	2542	Small	Hollee	Hot Springs	1991-12-11	3
199	2544	Edwards	Chloe	Columbus	1992-04-16	10

Query executed successfully. | DESKTOP-PDT5EKK (15.0 RTM) | DESKTOP-PDT5EKK\robwes... | Leerlingen | 00:00:00 | 199 rows

Je ziet dat record met `ll_nr` 2543 nu weg is. Het laatste record op positie 199 is het record dat zojuist nog op positie 200 stond.

Je kunt nu **oefening 8.1** maken.

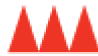
Controller aanvullen

Een controller is, zoals je inmiddels al weet, de verbinding van je code naar je datalaag. In de vorige reader is uitgelegd wat je moet doen als deze datalaag bestaat uit een SQL Server en is een eerste methode (ReadAll) gemaakt.

In onze controllers zorgen ervoor dat minimaal de CRUD functies (zie afbeelding) zijn geïmplementeerd. In deze reader, dit hoofdstuk gaan we aan de slag met de Delete, ofwel het verwijderen van gegevens alsook de Insert, ofwel het aanmaken van gegevens. In onze controller kunnen we dus een methode aanmaken met de naam Delete en de naam Create. Deze methode heeft als resultaat een getal wat aangeeft hoeveel rijen er verwijderd of toegevoegd zijn aan de database.

Vandaar dat de methoden als volgt gedefinieerd worden:

```
public int Create(EmployeeModel employee)

public int Delete(EmployeeModel employee)
```



Net als in de vorige reader, werken ook voor deze functionaliteit met ADO.NET. Meer informatie hierover vind je dus in reader 7.

Om een DELETE statement (query) uit te voeren op een database, dien je de volgende zaken (in volgorde) te programmeren:

1. Using System.Data.SqlClient
2. **ConnectionString** definiëren.
3. **SqlConnection** definiëren en starten (via **.Open()**).
4. **SqlCommand** definiëren (query komt hier), inclusief opgave **SqlParameters**
5. Query uitvoeren via **.ExecuteNonQuery()** en resultaat (aantal aangepast rijen) opvragen.

We gaan deze stappen één voor één hieronder kort behandelen. Daarna ga je aan de hand van een voorbeeld deze theorie omzetten naar een werkende applicatie.

Using System.Data.SqlClient

Alle functionaliteiten rondom het werken met ADO.NET zit verpakt in de library *System.Data.SqlClient*. Deze zit standaard meegeleverd met .NET en kunnen we daardoor gebruiken.

Connectionstring via ConfigurationManager

→ Een beschrijving van de connectionstring en van de app.config vind je in reader 7.

We gebruiken dezelfde connectionString als dat we gebruiken bij de ReadAll() methode. Dus ook nu halen we de connectionstring weer op uit de app.config op deze manier:

```
string connectionString =
ConfigurationManager.ConnectionStrings["connectionStringNorthwind"].ConnectionString;
```

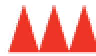
SQLConnection / Using

De connectionstring gebruiken we om, de naam zegt het al, een connectie met een database te maken. Deze connectie maken we met de C# class SQLConnection. Als je een nieuw object aanmaakt van de class SQLConnection geeft je als parameter namelijk de Connection string mee:

```
SqlConnection con = new SqlConnection(connectionString))
```

Ook in deze methode gebruiken we het C# commando using (lees hier meer). We kunnen bovenstaand commando dus aanpassen naar:

```
// Creeer een nieuw SqlConnection Object met de connectionString
using (SqlConnection con = new SqlConnection(connectionString))
{
}
```



SQLCommand

Nu is het tijd om een commando naar de database te gaan sturen. Een commando voor een database noemen we een query. Dit doen we middels een **SqlCommand.**

Een voorbeeld van een command ofwel query zou kunnen zijn:

```
DELETE FROM Employee WHERE EmployeeID = 12
```

Dit kunnen we vervolgens op de volgende manier doorgeven richting de database:

```
string sqlQuery = " DELETE FROM Employee WHERE EmployeeID = 12"

SqlCommand command = new SqlCommand(sqlQuery, connection))
```

Net als met het connection object is het netjes om aan te geven tot wanneer je het object gebruikt, dus is het goed om ook bij het command een using commando te gebruiken.

```
using (SqlConnection con = new SqlConnection(connectionString))
{
      //Sql statement (select, update en insert)
      string sqlQuery = " DELETE FROM Employee WHERE EmployeeID = 12";
      using (SqlCommand command = new SqlCommand(sqlQuery, con))
      {
      }
}
```

Let op dat je het using commando van je SqlCommand object binnen de using van de SqlConnection object plaatst. Het SqlCommand heeft deze SqlConnection namelijk nodig. Binnen de tweede accolades gaan we nu de code plaatsen waarmee we de gegevens uit de database gaan halen.

De query zoals we die gemaakt hebben is natuurlijk geen handige query. Op deze manier kunnen we alleen een employee met employeeID 12 verwijderen uit de database. Dat willen we natuurlijk niet, we willen zelf kunnen aangeven welke employee we moeten verwijderen. We willen, met andere woorden, onze query flexibel maken. Dat kunnen we doen met behulp van een zogenaamde parameter. Een parameter in een query voer je in door een @ te plaatsen met de naam van de parameter er achter. Dus in ons voorbeeld wordt het dan:

```
string sqlQuery = " DELETE FROM Employee WHERE EmployeeID = @EmployeeIDValue"
```

Zoals je ziet hebben we als naam van de parameter gekozen voor de naam van het veld (EmployeeID) gevolgd door Value. Zodoende is duidelijk dat we hier de waarde voor het veld in gaan stoppen. De parameter geven we vervolgens via de volgende codeerregel een waarde:

```
command.Parameters.AddWithValue("EmployeeIDValue", 12);
```

Natuurlijk kunnen we de waarde van de parameter ook invullen op basis van het object wat we willen verwijderen. Dat object zit in de variabele (feitelijk parameter van de methode) met de naam employee. Ons commando wordt dan:

```
command.Parameters.AddWithValue("EmployeeIDValue", employee.EmployeeID);
```

ExecuteNonQuery

We hebben inmiddels een object wat de verbinding naar de database verzorgd, een object wat een query naar een database kan sturen en SQL parameters die we voorzien hebben van een waarde. Het wordt tijd dat we de query daadwerkelijk gaan uitvoeren op de database.

Om te beginnen gaan we zorgen dat het object wat de verbinding verzorgd ook daadwerkelijk die verbinding naar de database opent. Dat doen we met de methode Open(). Dus bijvoorbeeld zo:

```
con.Open();
```

Vervolgens willen we een DELETE statement uitvoeren. Een DELETE statement geeft, in tegenstelling tot een SELECT statement, geen lijst met resultaten terug. In plaats daarvan geeft zo'n statement een getal als resultaat wat aangeeft hoeveel record er zijn aangepast, ofwel in dit geval hoeveel records er uit de database zijn verwijderd.

Daarom voeren we deze query uit met een ander statement als we gebruiken voor een SELECT. Bij een DELETE roepen we de methode ExecuteNonQuery() van het SqlCommand object aan. De methode geeft als resultaat het aantal aangepaste record in de vorm van een int terug. Dus maken voor een variabele van dit datatype aan en zetten we daar het resultaat in. Dus bijvoorbeeld zo:

```
// Voer het SqlCommand uit
affectedRows = command.ExecuteNonQuery();
```

Onze Delete methode hebben we gedefinieerd dat deze als resultaat het aantal aangepast rijen heeft. Dus moeten we dit nog even als resultaat van de functie teruggeven. Dat doen we als volgt:

```
return affectedRows;
```

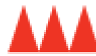
Foutafhandeling

Werken met een database betekent dat je werkt met iets waar je applicatie geen controle over heeft. De database draait (in de praktijk) op een andere server binnen het bedrijf of in de cloud. Deze server kan een storing hebben waardoor er geen verbinding mogelijk is. Maar verder zijn er nog tal van andere gevoeligheden voor fouten in je applicatie.

Natuurlijk wil je voorkomen dat je applicatie een ongecontroleerde foutmelding aan de gebruiker laat zien. Dit schrikt natuurlijk ontzettend als hij eens zo'n foutmelding te zien krijgt:

```
Msg 547, Level 16, State 0, Line 1

The DELETE statement conflicted with the REFERENCE constraint
"FK_Orders_Employees". The conflict occurred in database "Northwind", table
"dbo.Orders", column 'EmployeeID'.
```

Om dit te voorkomen bouwen we in onze applicaties foutafhandeling in. We zorgen met andere woorden dat we ervoor zorgen dat een gebruiker van onze applicatie een duidelijke melding krijgt als er iets niet goed gaat wat de applicatie niet zelf kan oplossen. We zorgen dus dat de applicatie (en dus feitelijk de programmeur) bepaald welke melding een gebruiker te zien krijgt.



Try catch

Het afvangen van een fout waardoor je applicatie uiteindelijk kan crashen doe je met een try catch constructie. Vrij vertaald staat er … probeer en vang (af). Ofwel, probeer een stukje code uit te voeren en vang een eventuele fout af. Dit afvangen doe je om zelf een duidelijke melding naar de gebruiker van je applicatie te kunnen sturen.

In de tekst hierboven lees je terug dat we een try… catch inzetten om richting de gebruiker van je applicatie duidelijk te kunnen communiceren. De communicatie met een gebruiker vindt plaats via de view. Dat betekent dus dat we in de code van je view wat moeten aanpassen.

Als we gaan kijken naar de code in je view en in dit geval specifiek naar de code die wordt uitgevoerd als we op de Verwijder knop drukken, dan zijn er twee plekken waarop er iets fout KAN gaan.

Als eerste (maar die check hebben we al), zou het mis kunnen gaan als de gebruiker in de listview niets geselecteerd heeft. Dit vangen we in de code al af met een simpele if controle.

Als tweede kan het misgaan als we iets met de database willen doen. Dat gebeurt met de volgende code:

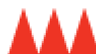
```
employeeController.Delete(employee);
```

Hier kan het dus mis gaan. Daarom plaatsen we deze regel in een try…catch constructie. Dat doen we als volgt:

```
try
{
    employeeController.Delete(employee);
}
catch (Exception exception)
{
    MessageBox.Show("Er is een fout opgetreden tijdens het verwijderen.");
}
```

→ Ga voor meer informatie naar https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch

Je kunt nu **oefening 8.2** maken.

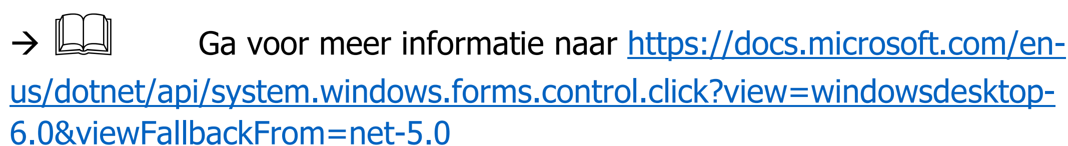


Gegevens toevoegen

Functionaliteit Toevoegen knop

Inmiddels hebben we een applicatie die gegevens uit een database kan laten zien en gegevens kan verwijderen. Natuurlijk moet een gebruiker van je applicatie ook gegevens kunnen toevoegen. Dat doet hij door op de knop 'Toevoegen' te drukken.

In Visual studio kunnen we code die uitgevoegd moet worden als we bijvoorbeeld op een knop klikken programmeren door in het ontwerp van het formulier dubbel te klikken op de knop. Technisch gezien maakt Visual Studio voor jouw een methode die hij direct koppelt aan het Click event van de knop.



→ Ga voor meer informatie naar https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.control.click?view=windowsdesktop-6.0&viewFallbackFrom=net-5.0

Als je de naamgeving van de knop gedaan hebt zoals in reader 6 is uitgelegd maakt hij de volgende methode voor je aan:

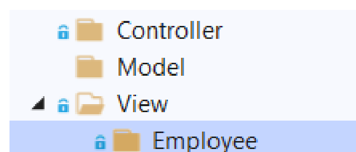
```
private void btnAdd_Click(object sender, EventArgs e)
{

}
```

In deze methode moeten we nu de code gaan plaatsen om een nieuw element toe te kunnen voegen. Dit werkt wat anders dan het verwijderen van een element. Immer bij het toevoegen van een element kan de gebruiker de inhoud van de verschillende attributen zelf bepalen/invullen.

Dit betekent dat, in tegenstelling tot bij het verwijderen van gegevens, we nu een extra view/scherm zullen moeten maken. Een formulier waar de gebruiker de gegevens van het element in kan invullen.

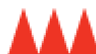
View bouwen

In het vorige hoofdstuk heb je al een scherm gemaakt voor medewerkers. Daar gaan we nu een scherm aan toevoegen. Maak dus onder het mapje view een mapje met de naam van het model aan (mocht dit nog niet gebeurd zijn). Dan kun je alle schermen die met een specifiek model te maken hebben 'groeperen'.



In deze map maak je vervolgens een nieuw Form (Windows Forms) aan. Geef dit formulier de naam frm<modelnaam>Add. Dus in ons voorbeeld zou het worden frmEmployeeAdd.

We gaan het scherm wederom maken volgens de huisstijl van deze module. Deze ga je in de komende hoofdstukken steeds beter leren kennen. Voor nu gaan we er vanuit dat een toevoeg scherm er op de volgende manier uitziet (de rode cijfers staan natuurlijk niet in je scherm maar worden gebruikt om hieronder instellingen toe te lichten):



frmEmployeeAdd 1

Medewerker toevoegen 2

Achternaam

Voornaam

Aanhef

3 Geboortedatum 4 04/04/2022 12:35

Datum in dienst 04/04/2022 12:35

Adres

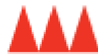
Woonplaats

5 Toevoegen

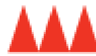
6 Sluiten

Hieronder vind je per nummer de naam van het control wat gebruikt is en de instellingen die zijn aangepast in de standaard waarde van dit control. Je ziet dat we een Panel gebruiken om de invoervelden (met bijbehorende labels) te 'groeperen'.

Nummer	Opmerkingen
1	De naam van het formulier. Geef hier duidelijk aan dat het een toevoegen (Add) scherm betreft en wat je gaat toevoegen (in dit geval medewerkers).
2	Control: Panel Text: <Naam vd entiteit> gevolgd door toevoegen Font: Segoe UI; 9 pt (die rond hij vaak wat af…)
3	Control: Label Name: Standaard naam mag je laten staan. Text: Naam van het attribuut wat je wil invoeren.
4	Controls die je gebruikt om gegevens in te laten voeren. Dit is natuurlijk afhankelijk van het type attribuut wat je hiermee vult. Dus kunnen het verschillende controls zijn: Control(s): Textbox (txt) of Combobox (cmb) of DateTimePicker (dtp) of NumericUpDown (num) Name: <drie letters (zie hierboven tussen haakjes)><attribuutnaam> bijvoorbeeld txtAchternaam, cmbAanhef, dtpGeboortedatum



5	Control: Button Name: btnAdd Flatsyle: Flast Tekst: Toevoegen Textalign: MiddleRight Width: 164 Height: 48
6	Control: Button Name: btnClose Flatsyle: Flast Tekst: Sluiten Textalign: MiddleRight Width: 144 Height: 48



Toevoegen van gegevens

In thema 4 hebben jullie voor het eerst kennis gemaakt met SQL-commando's. Een van de commando's die jullie daar hebben leren gebruiken is het INSERT statement. Dit (DML-statement) wordt gebruikt voor het toevoegen van gegevens aan een SQL-database. Elk statement heeft een syntax. Dit is de schrijfwijze van het statement. Hieronder staat de vereenvoudigde syntax van het insert-statement weergegeven.

```
INSERT → INTO → Tabel → (kolom1,kolom2,…)
VALUES (waarde1, waarde2,…) ;
```

Het is ook mogelijk om in een commando meerdere rijen in te voeren. Dat doe je dan als volgt:

```
INSERT → INTO → Tabel → (kolom1,kolom2,…)
VALUES (waarde1, waarde2,…) ,
(waarde1, waarde2,…) ,
…
(waarde1, waarde2,…) ;
```

De grijze tekst in de INSERT commando's hierboven is optioneel. Ofwel het is niet verplicht deze mee te geven. Als je de kolomnamen echter weglaat verwacht de SQL server dat de volgende van de kolommen in een tabel hetzelfde is als de volgorde waarin je de waardes aanbied. In de praktijk zal dit vaak zo zijn, vandaar dat in de praktijk de kolomnamen achter tabel vaak worden weggelaten.

Is de waarde van een veld (nog) niet bekend, dan kan de waarde NULL worden gebruikt. Velden die niet in het statement worden opgenomen, worden automatisch met de waarde NULL gevuld. Gegevens van het teksttype (o.a. Varchar, Char, Nvarchar enz.) en datums worden tussen enkele aanhalingstekens geplaatst, numerieke gegevens niet.

TIP: Let op met het invoeren van een datum. Hierbij moet je de structuur van een datum goed in je string zetten. Daar is '03-06-1999' voor SQL iets anders als '1995-06-03'. Als je wilt weten in welke volgorde je de waarde voor dag, maand en jaar moet plaatsen kun je deze query uitvoeren:

```
1  SELECT GETDATE();
```

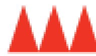
146 %

Results | Messages

	(No column name)
1	2022-04-05 02:25:47.677

Het resultaat laat de huidige datum/tijd zien, maar hieraan kun je ook de door SQL gewenste volgorde van dag, maand en jaar aflezen!

Je kunt nu **oefening 8.3** maken.



Controller aanvullen

Een controller is, zoals je inmiddels al weet, de verbinding van je code naar je datalaag. Inmiddels heb je in je controler twee methodes (ReadAll, Delete) gemaakt. Nu gaan we de controller uitbreiden en zorgen dat de de Insert, ofwel het aanmaken van gegevens wordt toegevoegd. Hierboven hebben we gezien dat deze methode als volgt gedefinieerd wordt:

```
public int Create(EmployeeModel employee)
```

Net als in de vorige methodes, werken ook voor deze functionaliteit met ADO.NET. Meer informatie hierover vind je dus in reader 7.

Om een INSERT statement (query) uit te voeren op een database, dien je de volgende zaken (in volgorde) te programmeren:

1. Using System.Data.SqlClient
2. **ConnectionString** definiëren.
3. **SqlConnection** definiëren en starten (via **.Open()**).
4. **SqlCommand** definiëren (query komt hier), inclusief opgave **SqlParameters**
5. Query uitvoeren via **.ExecuteNonQuery()** en resultaat (aantal aangepast rijen) opvragen.

We gaan deze stappen wederom één voor één hieronder kort behandelen. Daarna ga je aan de hand van een voorbeeld deze theorie omzetten naar een werkende applicatie.

Using System.Data.SqlClient

Alle functionaliteiten rondom het werken met ADO.NET zit verpakt in de library *System.Data.SqlClient***.** Deze zit standaard meegeleverd met .NET en kunnen we daardoor gebruiken.

Connectionstring via ConfigurationManager

→ Een beschrijving van de connectionstring en van de app.config vind je in reader 7.

We gebruiken dezelfde connectionString als dat we gebruiken bij de ReadAll() en de Delete() methode. Dus ook nu halen we de connectionstring weer op uit de app.config op deze manier:

```
string connectionString =
ConfigurationManager.ConnectionStrings["connectionStringNorthwind"].ConnectionString;
```

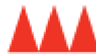
SQLConnection / Using

De connectionstring gebruiken we om, de naam zegt het al, een connectie met een database te maken. Deze connectie maken we met de C# class SQLConnection. Als je een nieuw object aanmaakt van de class SQLConnection geeft je als parameter namelijk de Connection string mee:

```
SqlConnection con = new SqlConnection(connectionString))
```

Ook in deze methode gebruiken we het C# commando using (lees hier meer). We kunnen bovenstaand commando dus aanpassen naar:

```
// Creeer een nieuw SqlConnection Object met de connectionString
using (SqlConnection con = new SqlConnection(connectionString))
{
}
```



SQLCommand

Nu is het tijd om een commando naar de database te gaan sturen. Een commando voor een database noemen we een query. Dit doen we middels een **SqlCommand.**

Een voorbeeld van een command ofwel query zou kunnen zijn:

```
INSERT INTO Employee (LastName, FirstName, BirthDate) VALUES ('Puk',
'Pietje','2001-03-15');
```

Dit kunnen we vervolgens op de volgende manier doorgeven richting de database:

```
string sqlQuery = " INSERT INTO Employee (LastName, FirstName, BirthDate)
VALUES ('Puk', 'Pietje','2001-03-15');"

SqlCommand command = new SqlCommand(sqlQuery, connection))
```

Net als met het connection object is het netjes om aan te geven tot wanneer je het object gebruikt, dus is het goed om ook bij het command een using commando te gebruiken.

```
using (SqlConnection con = new SqlConnection(connectionString))
{
      //Sql statement (select, update en insert)
      string sqlQuery = "INSERT INTO Employee (LastName, FirstName, BirthDate)
VALUES ('Puk', 'Pietje','2001-03-15');";
      using (SqlCommand command = new SqlCommand(sqlQuery, con))
      {
      }
}
```

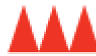
Let op dat je het using commando van je SqlCommand object binnen de using van de SqlConnection object plaatst. Het SqlCommand heeft deze SqlConnection namelijk nodig. Binnen de tweede accolades gaan we nu de code plaatsen waarmee we de gegevens uit de database gaan halen.

Ook deze query willen we natuurlijk flexibel maken. Dat doen we ook hier met behulp van een zogenaamde parameter. Een parameter in een query voer je in door een @ te plaatsen met de naam van de parameter er achter. Dus in ons voorbeeld wordt het dan:

```
string sqlQuery = " INSERT INTO Employee (LastName, FirstName, BirthDate)
VALUES (@LastNameValue, @FirstNameValue,@BirtDateValue);"
```

Zoals je ziet hebben we als naam van de parameter gekozen voor de naam van het veld (LastName) gevolgd door Value. Zodoende is duidelijk dat we hier de waarde voor het veld in gaan stoppen. De parameter geven we vervolgens via de volgende codeerregel een waarde:

```
command.Parameters.AddWithValue("LastNameValue", "Puk");

command.Parameters.AddWithValue("FirstNameValue", "Pietje");

command.Parameters.AddWithValue("BirtDateValue", "2001-03-15");
```



Natuurlijk kunnen we de waarde van de parameter ook invullen op basis van het object wat we willen toevoegen. Dat object zit in de variabele (feitelijk parameter van de methode) met de naam employee. Ons commando wordt dan:

```
command.Parameters.AddWithValue("EmployeeIDValue", employee.EmployeeID);

command.Parameters.AddWithValue("LastNameValue", employee.LastName);

command.Parameters.AddWithValue("FirstNameValue", employee.FirstName);

command.Parameters.AddWithValue("BirtDateValue", employee.BirthDate);
```

Natuurlijk zorg je ervoor dat de query ALLE velden die een gebruiker invult ook daadwerkelijk in de database plaatst!

ExecuteNonQuery

We hebben inmiddels een object wat de verbinding naar de database verzorgd, een object wat een query naar een database kan sturen en SQL parameters die we voorzien hebben van een waarde. Het wordt tijd dat we de query daadwerkelijk gaan uitvoeren op de database.

Om te beginnen gaan we zorgen dat het object wat de verbinding verzorgd ook daadwerkelijk die verbinding naar de database opent. Dat doen we met de methode Open(). Dus bijvoorbeeld zo:

```
con.Open();
```

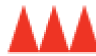
Vervolgens willen we een INSERT statement uitvoeren. Een INSERT statement geeft, in tegenstelling tot een SELECT statement, geen lijst met resultaten terug. In plaats daarvan geeft zo'n statement een getal als resultaat wat aangeeft hoeveel record er zijn aangepast, ofwel in dit geval hoeveel records er uit de database zijn toegevoegd.

Daarom voeren we deze query uit met een ander statement als we gebruiken voor een SELECT. Net als bij een DELETE roepen we bij een INSERT de methode ExecuteNonQuery() van het SqlCommand object aan. De methode geeft als resultaat het aantal aangepaste record in de vorm van een int terug. Dus maken voor een variabele van dit datatype aan en zetten we daar het resultaat in. Dus bijvoorbeeld zo:

```
// Voer het SqlCommand uit
affectedRows = command.ExecuteNonQuery();
```

Onze Create methode hebben we gedefinieerd dat deze als resultaat het aantal aangepast rijen heeft. Dus moeten we dit nog even als resultaat van de functie teruggeven. Dat doen we als volgt:

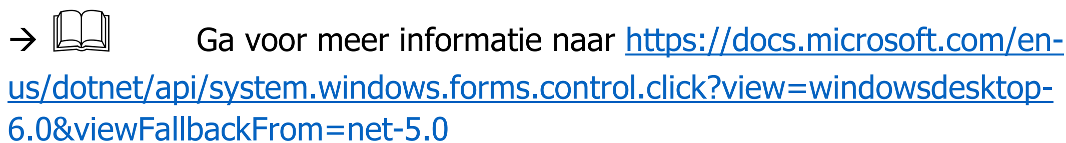
```
return affectedRows;
```



View bouwen – deel 2

Inmiddels hebben we een view gemaakt om een nieuwe entiteit toe te kunnen voegen en hebben we onze controller uitgebreid zodat deze ook een nieuw record van deze entiteit in de database kan toevoegen. Nu wordt het zaak om deze twee onderdelen aan elkaar te koppelen. Dat doen we natuurlijk door deze code te laten uitvoeren als iemand op de Toevoegen knop drukt in het <Entiteit>Add scherm.

In Visual studio kunnen we code die uitgevoegd moet worden als we bijvoorbeeld op een knop klikken programmeren door in het ontwerp van het formulier dubbel te klikken op de knop. Technisch gezien maakt Visual Studio voor jouw een methode die hij direct koppelt aan het Click event van de knop.



→ Ga voor meer informatie naar https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.control.click?view=windowsdesktop-6.0&viewFallbackFrom=net-5.0

Als je de naamgeving van de knop gedaan hebt zoals in reader 6 is uitgelegd maakt hij de volgende methode voor je aan:

```
private void btnAdd_Click(object sender, EventArgs e)
{

}
```

In deze methode moeten we nu de code gaan plaatsen die een nieuw element aanmaakt en deze (via de controller) toevoegt aan de database.

Allereerst moeten we natuurlijk een variabele aanmaken van het <Entiteit>Model en deze vervolgens vullen met de, door de gebruiker, ingevoerde gegevens. In ons voorbeeld maken we dus een nieuw object van een employeeModel wat we vervolgens vullen met de ingevulde informatie.

Een nieuw object van het employeeModel maken doen we als volgt:

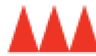
```
EmployeeModel employee = new EmployeeModel();
```

Vervolgens gaan we dit model, deze *employee* dus, vullen met de gegevens die de gebruiker in ons <Entiteit>Add scherm heeft ingevuld. Als je gebruik gemaakt hebt van een Textbox control dan kun je de inhoud hiervan ophalen via het .Text attribuut. Dus bijvoorbeeld:

```
employee.LastName = txtLastName.Text;
```

Hieronder volgt een overzicht van verschillende controls en het attribuut wat je gebruikt om de ingevoerde waarde op te halen.

Control	**Attribuut**
Textbox	Text
DateTimePicker	Value
NumericUpDown	Value
Combobox	Tekst (in een later hoofdstuk leer je een andere manier om de waarde van een combobox op te halen).



Als je alle attributen van je object gevuld hebt wordt het tijd om dit op te slaan. Dit doen we via de controller (de controller zorgt immers voor de communicatie met de database). In de vorige paragraaf hebben we deze methode in onze controller aangemaakt. Nu is het dus alleen nog een controller object aanmaken en de Create methode daarvan aanroepen.

In ons voorbeeld zou het dan als volgt gaan:

```
EmployeeController controller = new EmployeeController();
controller.Create(employee);
```

Via het controller object maken we verbinding met de database. Dat is wederom een potentiele bron voor fouten. We zijn immers afhankelijk van een extern systeem. Daarom is het ook hier verstandig om de aanroep van de Create() methode tussen in try…catch constructie te plaatsen.

```
try
{
    controller.Create(employee);
}
catch (Exception exception)
{
    MessageBox.Show("Er is iets fout gegaan tijdens het aanmaken van de medewerker.");
}
```

Tip: Probeer code waarvan je weet dat het mogelijk een applicatiefout oplevert in een try…catch te plaatsen.

Je kunt nu **oefening 8.4** maken.