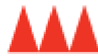
Gevorderd standalone

Realiseren

hoofdstuk

1

Bewerken van gegevens



Algemene informatie

Onderwerp	Bewerken van gegevens
Leerdoel(en)	1. De student <werkwoord> … 2. De student <werkwoord> … 3. De student <werkwoord> … 4. De student <werkwoord> …
Vereiste voorkennis	1. De student … 2. De student … 3. De student …
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

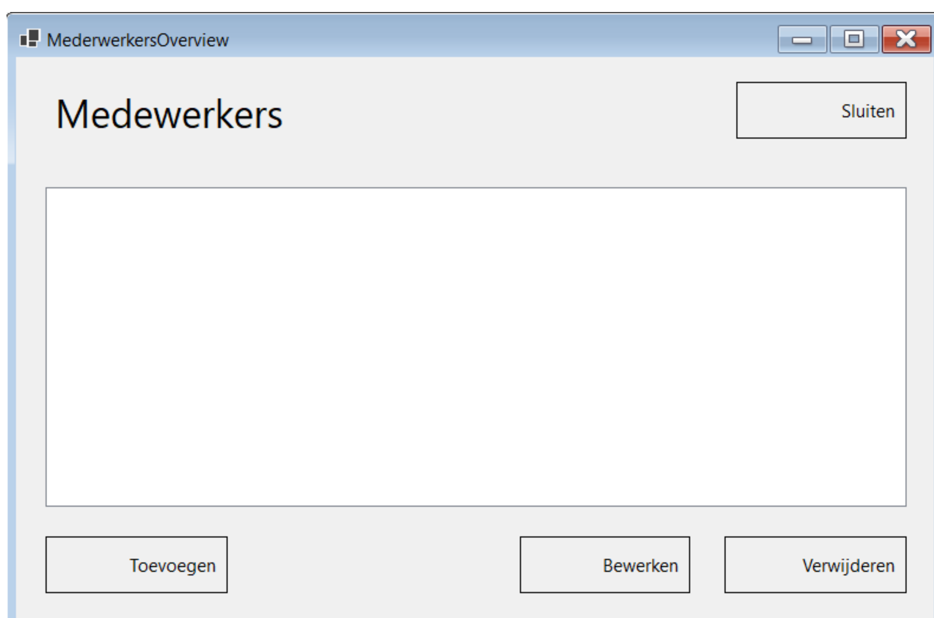
Algemene informatie 2
Inhoudsopgave 3
Introductie 4
Inhoud 4
Bewerken van gegevens 4
Functionaliteit Bewerken knop 4

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. Naast het ophalen, toevoegen en verwijderen van gegevens moet een gebruiker natuurlijk ook gegevens kunnen aanpassen. Hoe je dat vanuit onze applicatie moet bouwen behandelen we in deze reader.

Inhoud

In de reader van hoofdstuk 7 heb je geleerd 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 en welke functionaliteit je programmeert achter de 'Verwijderen' en 'Toevoegen' knop. In dit hoofdstuk ga je leren hoe je de functionaliteit achter de 'Bewerken' knop moet realiseren.

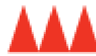


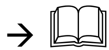
Bewerken van gegevens

Functionaliteit Bewerken 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 kunnen bewerken. Dat doet hij door op de 'Bewerken' knop 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 btnEdit_Click(object sender, EventArgs e)
{

}
```

In deze methode moeten we nu de code gaan plaatsen die het geselecteerde element moet laten bewerken ofwel aanpassen door de gebruiker van de applicatie.

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:

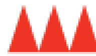
```
if (lvEmployees.SelectedItems.Count == 1)
{

}
```

Natuurlijk kan je een andere check inbouwen (>0, <> 0 etc.).

Nu gaan we op zoek naar de entiteit die we willen bewerken. In het geval van het voorbeeld, welk EmployeeModel moet nu worden bewerkt? 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

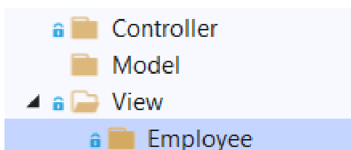


Voor nu is het van belang om te weten dat je het object wat in Tag zit kunt 'omzetten' naar ons model. We zetten feitelijk een andere bril op en kijken op een andere manier naar de opgeslagen variabele dit in het Tag attribuut zit. Daardoor zien we ineens dat het ons model is 😉. Dat heet in C# type casten. 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 aan laten passen door de gebruiker. Aanpassen betekent natuurlijk dat we een scherm (view) moeten maken waarin we de gegevens uit het object plaatsen en de gebruiker de kans geven om wijzigingen door te voeren.

In het vorige hoofdstuk heb je al schermen 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>Edit. Dus in ons voorbeeld zou het worden frmEmployeeEdit.

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 bewerk 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):

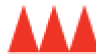
frmEmployeeEdit 1
Medewerker bewerken 2
Achternaam van Breukelen
Voornaam Hans
Aanhef Dhr.
Geboortedatum 3 4-10-1956 4
Datum in dienst 1- 1-1984
Adres Philipslaan 1
Woonplaats Eindhoven
5 Bewerken
6 Sluiten

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 bewerken (Edit) scherm betreft en wat je gaat bewerken (in dit geval medewerkers).
2	Control: Panel Text: <Naam vd entiteit> gevolgd door bewerken 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 de gebruiker laat aanpassen.
4	Controls die je gebruikt om gegevens aan te laten passen. 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: btnEdit Flatsyle: Flast Tekst: Bewerken Textalign: MiddleRight Width: 164 Height: 48
6	Control: Button Name: btnClose Flatsyle: Flast Tekst: Sluiten Textalign: MiddleRight Width: 144 Height: 48

Natuurlijk moet het scherm de gegevens weten van de entiteit die moet worden aangepast. De gegevens moeten immers bij het tonen van het scherm reeds ingevuld zijn. Dit doen we door bij het aanmaken van het scherm deze entiteit reeds mee te nemen. Dit doen we door het aanpassen van de Constructor van het scherm. Standaard ziet deze functie er in de code als volgt uit:

```
public frmEmployeeEdit()
{
    InitializeComponent();
}
```



We passen dit aan naar de volgende code:

```
private EmployeeModel employeeItem;

0 references | 0 changes | 0 authors, 0 changes
public frmEmployeeEdit(EmployeeModel employee)
{
    InitializeComponent();
    employeeItem = employee;
}
```

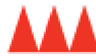
Je ziet dat we een private variabele employeeItem gemaakt hebben van het type EmployeeModel. Daarin bewaren we de entiteit die we moeten aanpassen. Deze employee geven we als parameter mee aan de constructor. In de code van de constructor zetten we vervolgens de meegegeven entiteit in de private variabele.

Vervolgens gaan we de labels die we hebben aangemaakt gaan vullen met de gegevens uit de entiteit. Dit KAN natuurlijk ook in de contructor, maar wij kiezen ervoor om dit in de het OnLoad event van het formulier te doen. Daar plaatsen we code om alle controls te vullen. Zoiets als dit:

```
private void frmEmployeeEdit_Load(object sender, EventArgs e)
{
    txtLastName.Text = employeeItem.LastName;
    txtFirstName.Text = employeeItem.FirstName;
    dtpBirthDate.Value = employeeItem.BirthDate;
}
```

Als de gebruiker op de knop btnEdit klikt moeten we de, door de gebruiker, gewijzigde gegevens opslaan in de database. Communicatie met de database verloopt via de Controller, dus dat betekent dus dat we in onze controller een methode moeten maken om een element uit de database aan te passen. Als we deze methode gemaakt hebben kunnen we die als volgt aanroepen:

```
employeeController.Update(employeeItem);
```



Aanpassen 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 UPDATE statement. Dit (DML-statement) wordt gebruikt voor het aanpassen 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 update-statement weergegeven.

UPDATE → **Tabel**

SET → **kolom1 = waarde1**,

→ kolom2 = waarde2.

etc

WHERE → **kolomID = waardeID ;**

De grijze tekst in het UPDATE commando hierboven is optioneel. Ofwel het is niet verplicht deze mee te geven. Je moet dus minimaal 1 kolom een nieuwe waarde geven. In onze applicatie zetten we hier alle kolommen neer die een gebruiker KAN aanpassen.

Heel belangrijk in het UPDATE statement is de WHERE clausule. Daarmee geef je aan welke rijen aangepast moeten worden. Hier wil je in principe (zo werken we vanuit de applicatie) maar 1 rij selecteren. Daarom staat er in de syntax dat je daar een ID kolom gebruikt. Daarmee kun je namelijk 1 rij selecteren. Het is echter ook mogelijk om hier meerdere rijen te selecteren. Zo past het volgende UPDATE statement in de Northwind database 5 records aan (namelijk alle employees die als Country de waarde 'USA' hebben).

```
UPDATE Employees
SET Country = 'US'
WHERE Country = 'USA'
```

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();
```

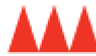
146 %

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 1.1** 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 drie methodes (ReadAll, Delete en Insert) gemaakt. Nu gaan we de controller uitbreiden en zorgen dat de de Update, ofwel het bewerken van gegevens wordt toegevoegd. Hierboven hebben we gezien dat deze methode als volgt gedefinieerd wordt:

```
public int Update(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 UPDATE 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;
```

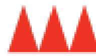
SQLConnection / Using

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:

```
UPDATE Employee

SET   LastName='Puk',

      FirstName='Pietje',

      BirthDate='2001-03-15'

WHERE EmployeeID = 1;
```

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

```
string sqlQuery = "UPDATE Employee SET LastName='Puk', FirstName='Pietje',
BirthDate='2001-03-15' WHERE EmployeeID = 1;"

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.

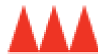
```
using (SqlConnection con = new SqlConnection(connectionString))
{
      //Sql statement (select, update en insert)
      string sqlQuery = "UPDATE Employee SET LastName='Puk',
FirstName='Pietje', BirthDate='2001-03-15' WHERE EmployeeID = 1;";
      using (SqlCommand command = new SqlCommand(sqlQuery, 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 = "UPDATE Employee SET LastName=@LastNameValue,
FirstName=@FirstNameValue, BirthDate=@BirtDateValue WHERE EmployeeID =
@EmployeeIDValue;"
```

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");

command.Parameters.AddWithValue("EmployeeIDValue", 1);
```

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("LastNameValue", employee.LastName);

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

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

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

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

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();
```

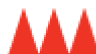
Vervolgens willen we een UPDATE statement uitvoeren. Een UPDATE statement geeft, in tegenstelling tot een SELECT statement, geen lijst met resultaten terug. In plaats daarvan geeft zo'n statement een getal als resultaat wat aangeeft hoeveel records er zijn aangepast, ofwel in dit geval hoeveel records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

Daarom voeren we deze query uit met een ander statement als we gebruiken voor een SELECT. Net als bij een DELETE en de INSERT roepen we bij een UPDATE de methode ExecuteNonQuery() van het SqlCommand object aan. De methode geeft als resultaat het aantal aangepaste records 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 Update 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:

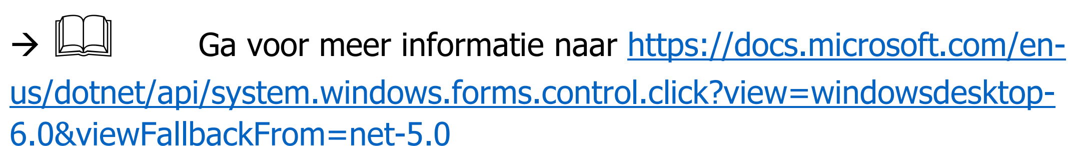
```
return affectedRows;
```



View bouwen – deel 2

Inmiddels hebben we een view gemaakt om de gekozen entiteit aan te kunnen passen en hebben we onze controller uitgebreid zodat deze het record van deze entiteit in de database kan aanpassen. 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 Bewerken knop drukt in het <Entiteit>Edit 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 btnEdit_Click(object sender, EventArgs e)
{

}
```

In deze methode moeten we nu de code gaan plaatsen die de gewijzigde gegevens van de entiteit (via de controller) doorvoert in de database.

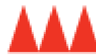
Allereerst moeten we natuurlijk de private variabele van het <Entiteit>Model vullen met de, door de gebruiker, ingevoerde/aangepaste gegevens. In ons voorbeeld vullen we dus het object van een employeeModel met de ingevulde informatie.

Vervolgens gaan we dit model, deze *employee* dus, vullen met de gegevens die de gebruiker in ons <Entiteit>Edit 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:

```
employeeItem.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 Update methode daarvan aanroepen.

In ons voorbeeld zou het dan als volgt gaan:

```
EmployeeController controller = new EmployeeController();
controller.Update(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 Update() methode tussen in try…catch constructie te plaatsen.

```
try
{
    controller.Update(employeeItem);
}
catch
{
    MessageBox.Show("Er is iets fout gegaan bij het bewerken 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 1.2** maken.