ADO.Net

Inhoudstafel

1. System.Data of ADO.NET

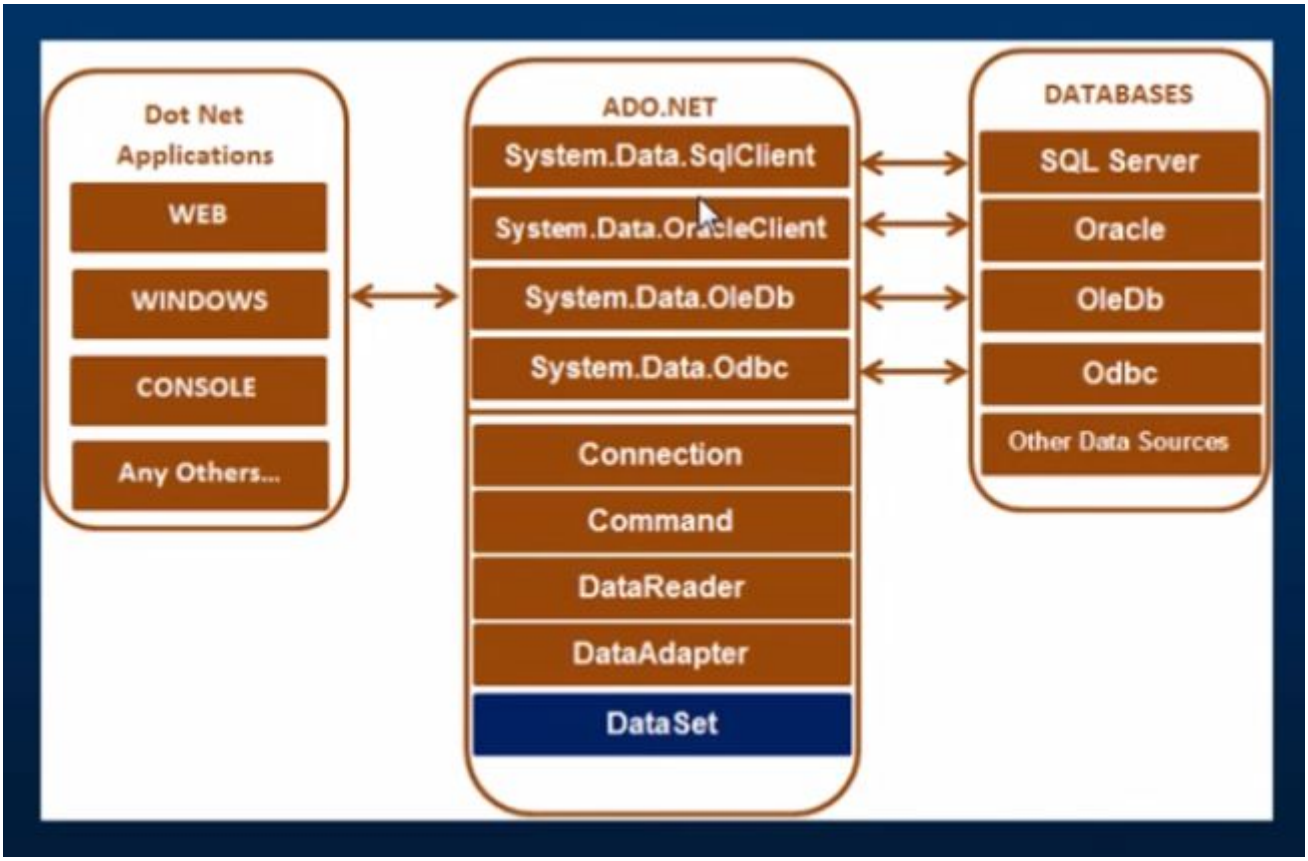
ActiveX Data Objects for .NET, afgekort naar ADO.NET is verbonden aan het .NET-framework en biedt een programmeerinterface om applicaties in staat te stellen met willekeurige databases te communiceren.

ADO.NET is bedoeld om een standaard interface voor data access te definiëren.

ADO.NET bevat onder andere interfaces als Connection en Command.

Iedere technologie (SQL Server, Oracle, XML) kan zijn eigen implementatie geven aan deze interfaces. Met de komst van ADO.NET kan de keuze voor een data access technologie configureerbaar gemaakt worden.

Via de NuGet package manager zullen we de dependencies voor Microsoft.Data.SqlClient toevoegen: dit is de data provider voor SQL Server.

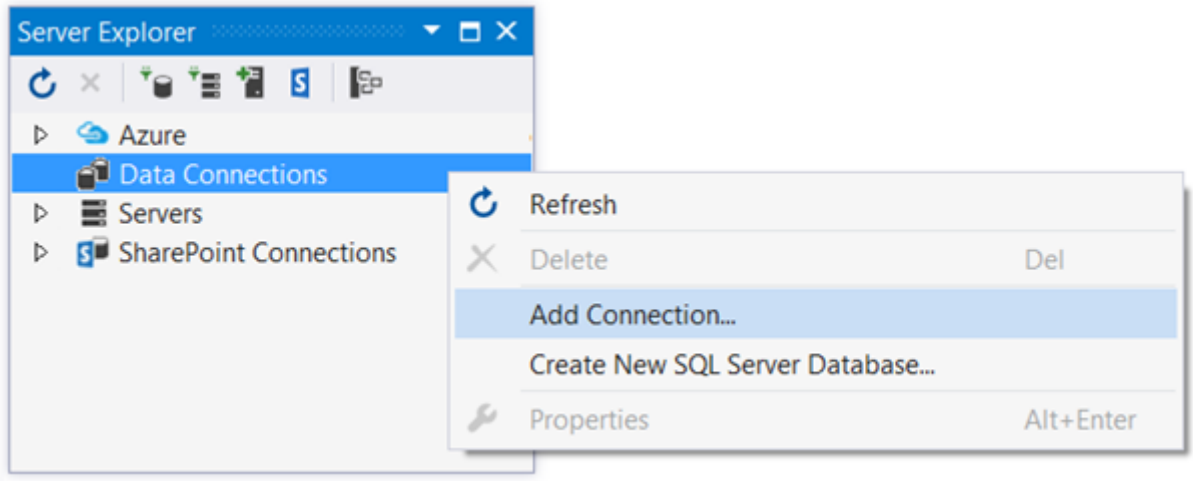


2. Visual Studio Tooling

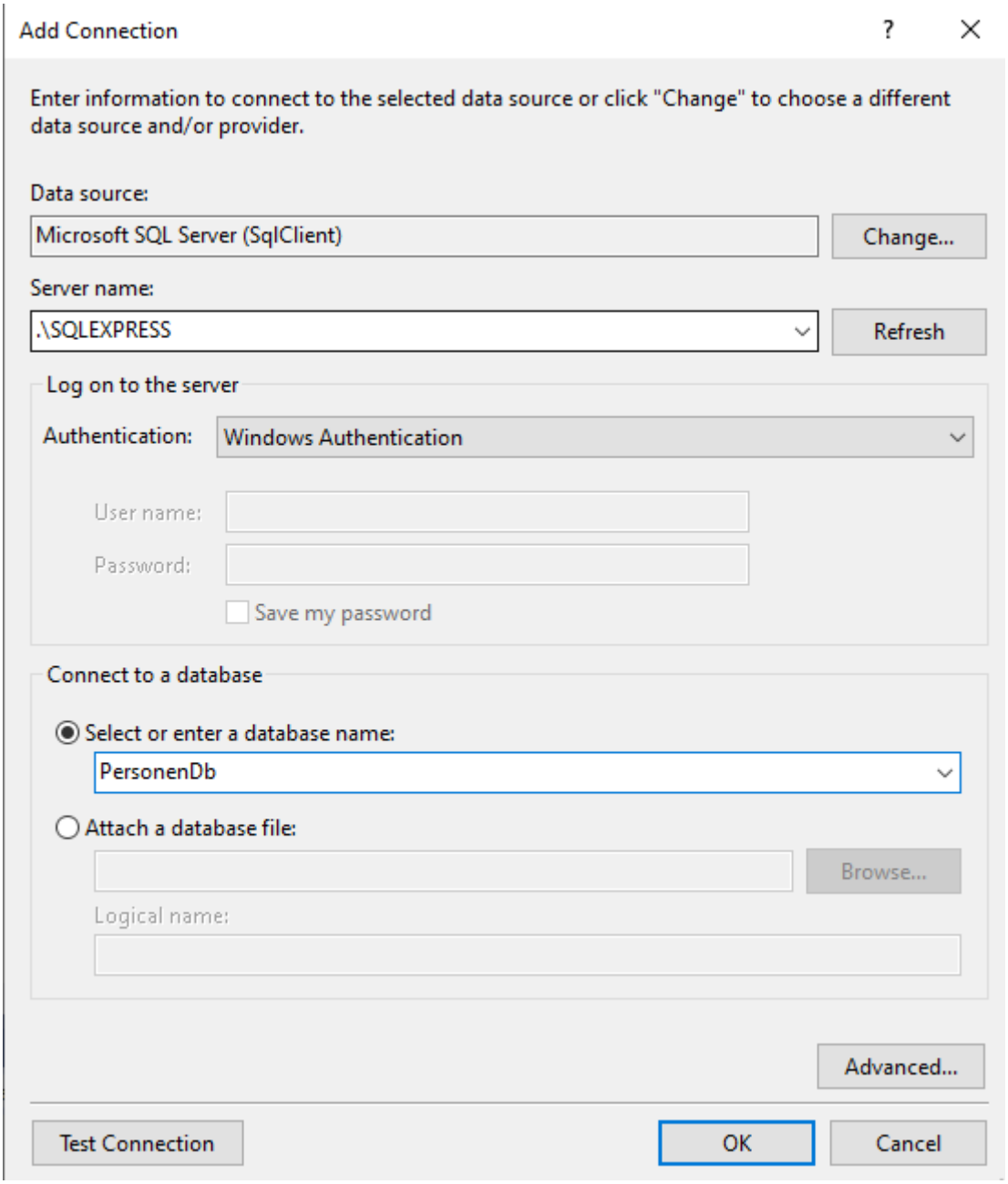
2.1. Databank bestand creëren

Na het installeren van SQL Express hebben we een SQL server draaien waarmee we kunnen connecteren.

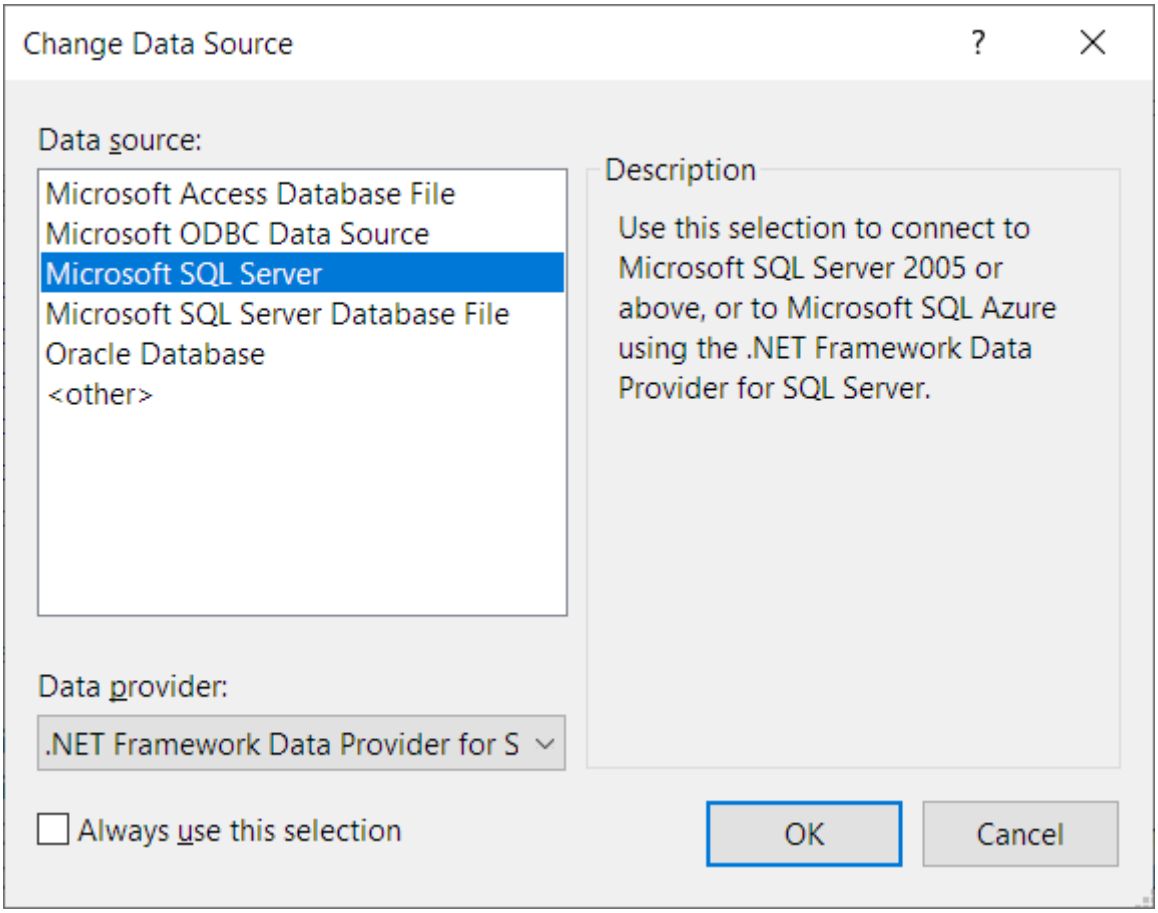
Open het *Server Explorer* toolbox venster door in *Visual Studio* in de *View* menu te kiezen voor *Server Explorer*. Rechterklik op de *Data Connections* node en kies voor *Add Connection...*.



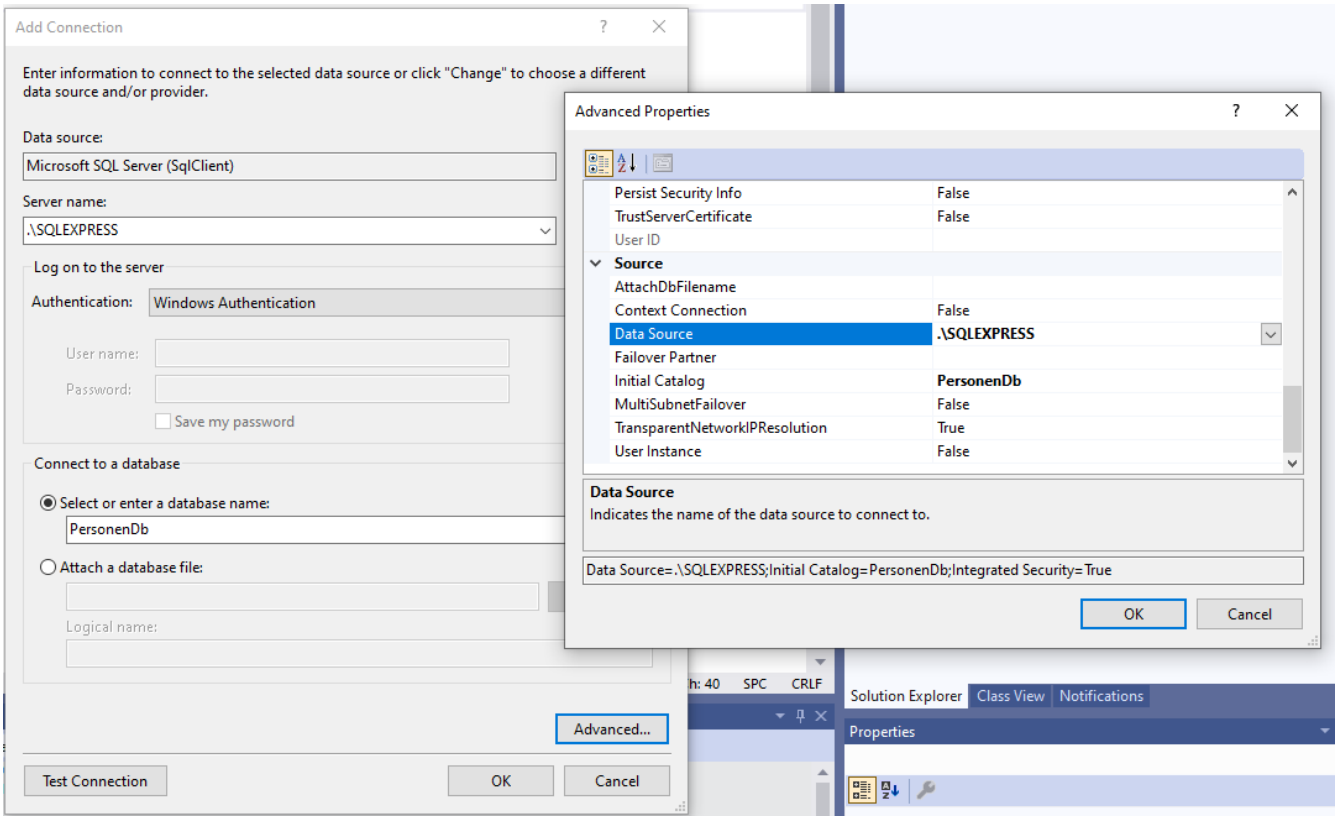
Verifieer of het *Data source* veld staat ingesteld op *Microsoft SQL Server (SqlClient)*.



Indien dit niet zo is, klik je op de *Change…* knop om deze instelling aan te passen.

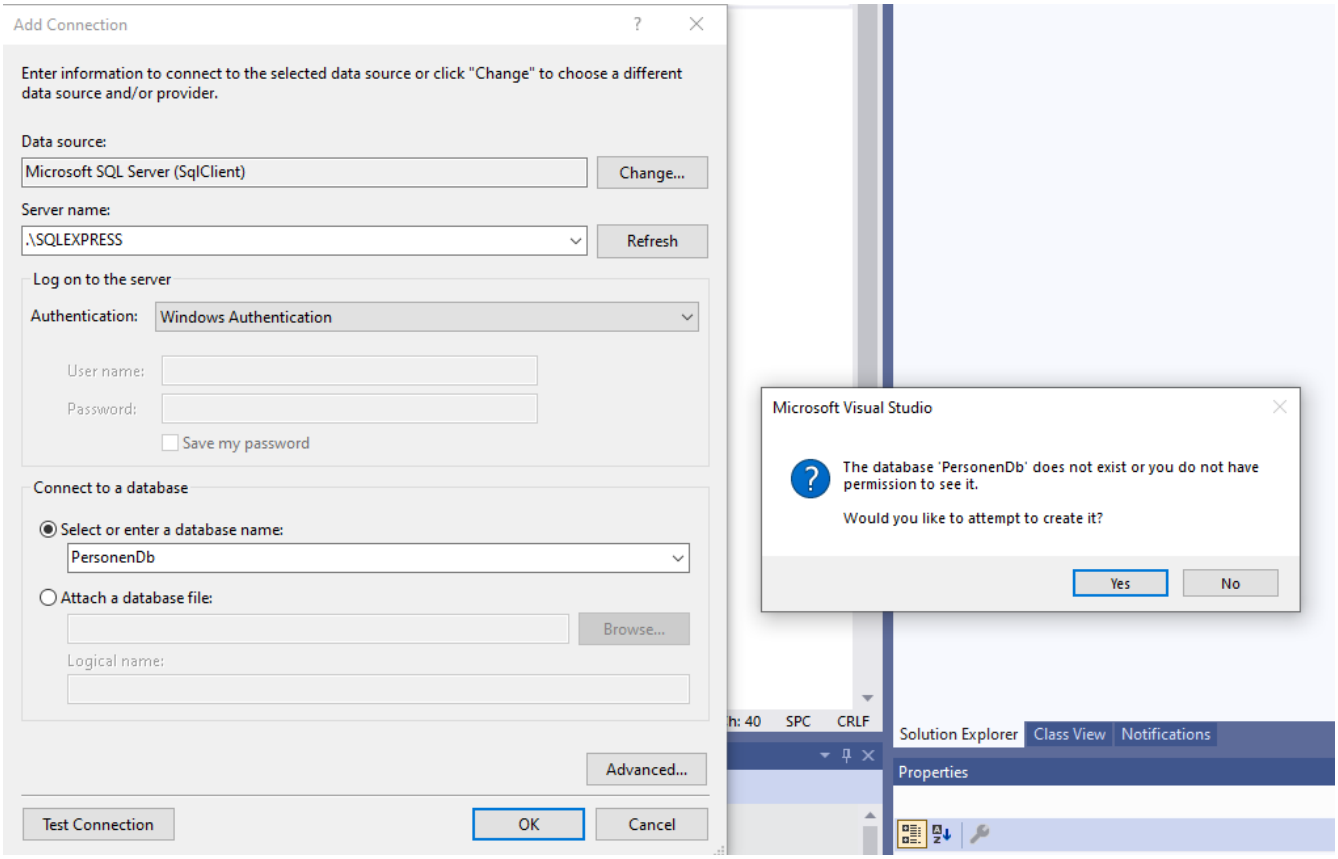


Dit venster biedt je eventueel ook de mogelijkheid de hiermee samengestelde *connection string* op te vragen, via de knop ``Advanced...". Deze waarde (onderaan de dialoog) zou je kunnen kopiëren en straks gebruiken in onze broncode.

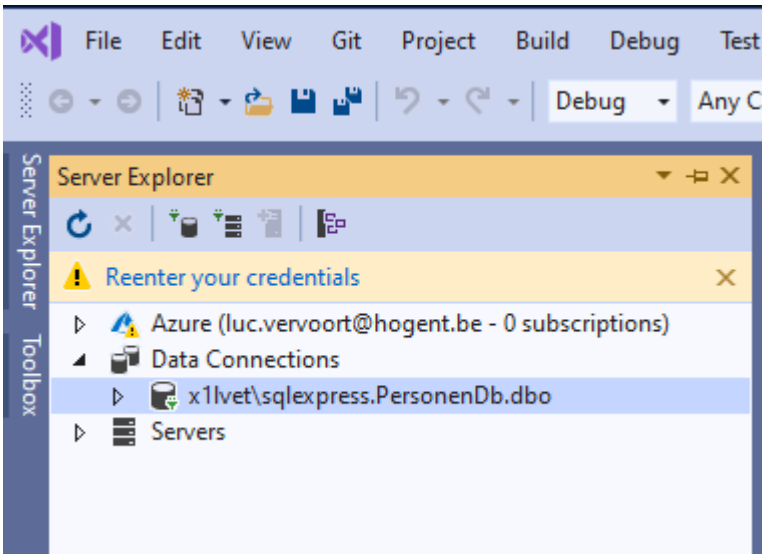


Klik je in het *Add Connection* venster op *OK* dan wordt een node toegevoegd aan het *Server Explorer*

toolvenster. Tussendoor wordt om bevestiging van creatie gevraagd.

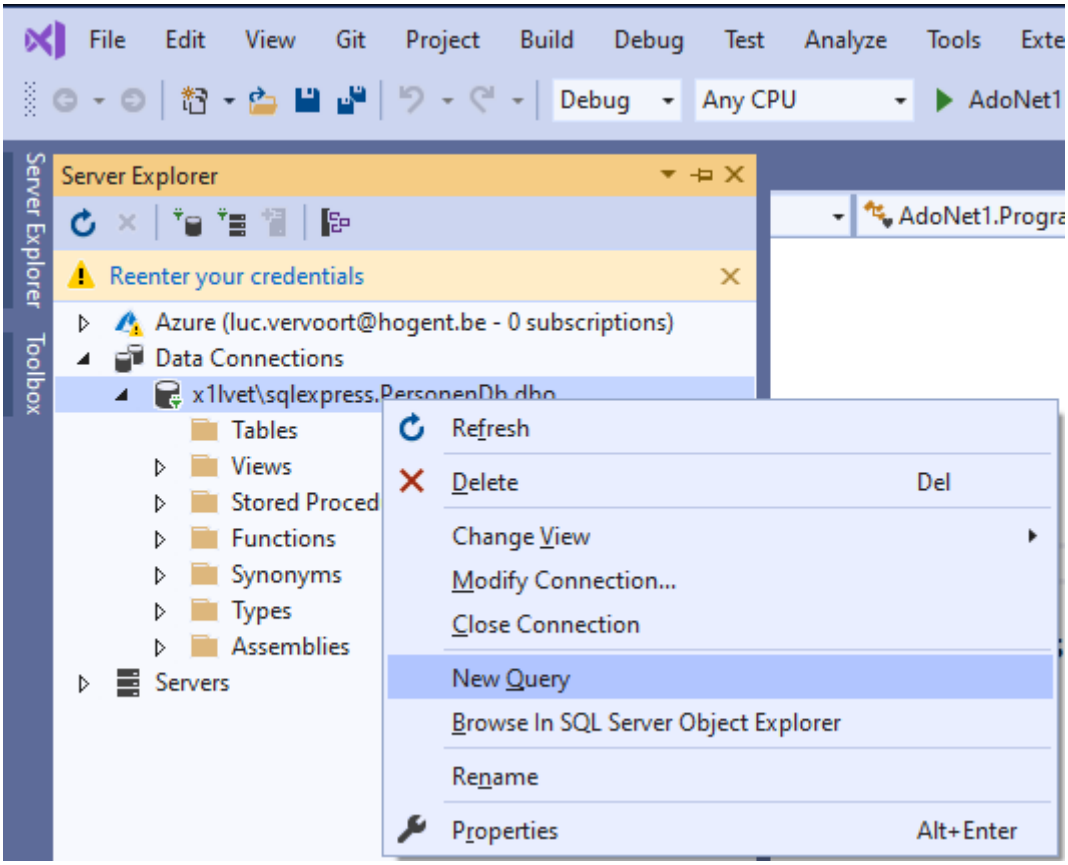


Resultaat:

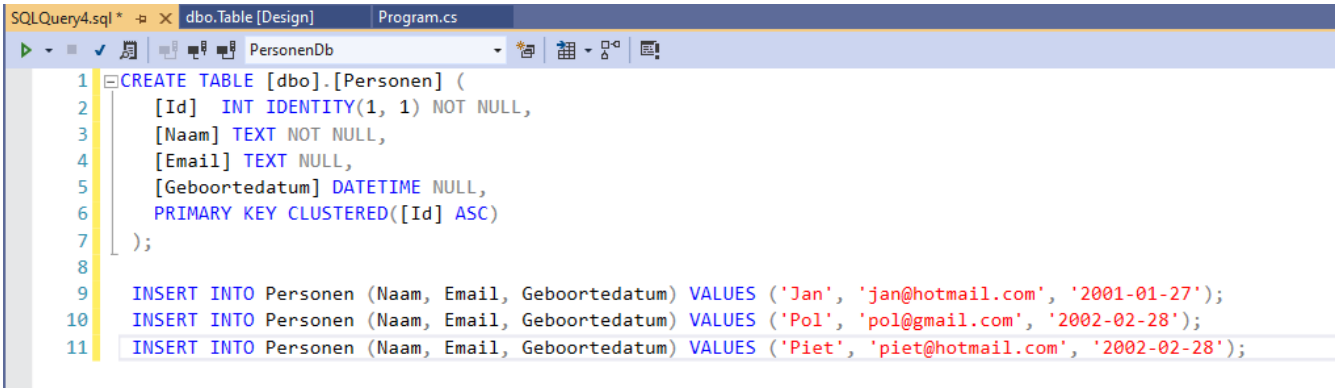


2.2. Tabel creëren en opvullen

Om vlot de nodige databank tabel te creëren maken we gebruik van *SQL creational language*. Rechterklik op de PersonenDb.dbo node en kies voor New Query:



Neem volgende *SQL* over en kies in de *SQL* menu van *Visual Studio* voor *Execute* (in het venster bovenaan links de groene pijl):



Copieer de queries hier:

```
CREATE TABLE [dbo].[Personen] (
   [Id]  INT IDENTITY(1, 1) NOT NULL,
   [Naam] TEXT NOT NULL,
   [Email] TEXT NULL,
   [Geboortedatum] DATETIME NULL,
   PRIMARY KEY CLUSTERED([Id] ASC)
);

 INSERT INTO Personen (Naam, Email, Geboortedatum) VALUES ('Jan',
'jan@hotmail.com', '2001-01-27');
 INSERT INTO Personen (Naam, Email, Geboortedatum) VALUES ('Pol', 'pol@gmail.com',
'2002-02-28');
```

```
11  INSERT INTO Personen (Naam, Email, Geboortedatum) VALUES ('Piet',
    'piet@hotmail.com', '2002-02-28');
```

Als alles goed loopt krijg je onderaan het scherm te zien dat de gebruikte *SQL manipulation language* (de *insert* statements) drie rijen heeft toegevoegd.

```
1 (1 row(s) affected)
2 (1 row(s) affected)
3 (1 row(s) affected)
```

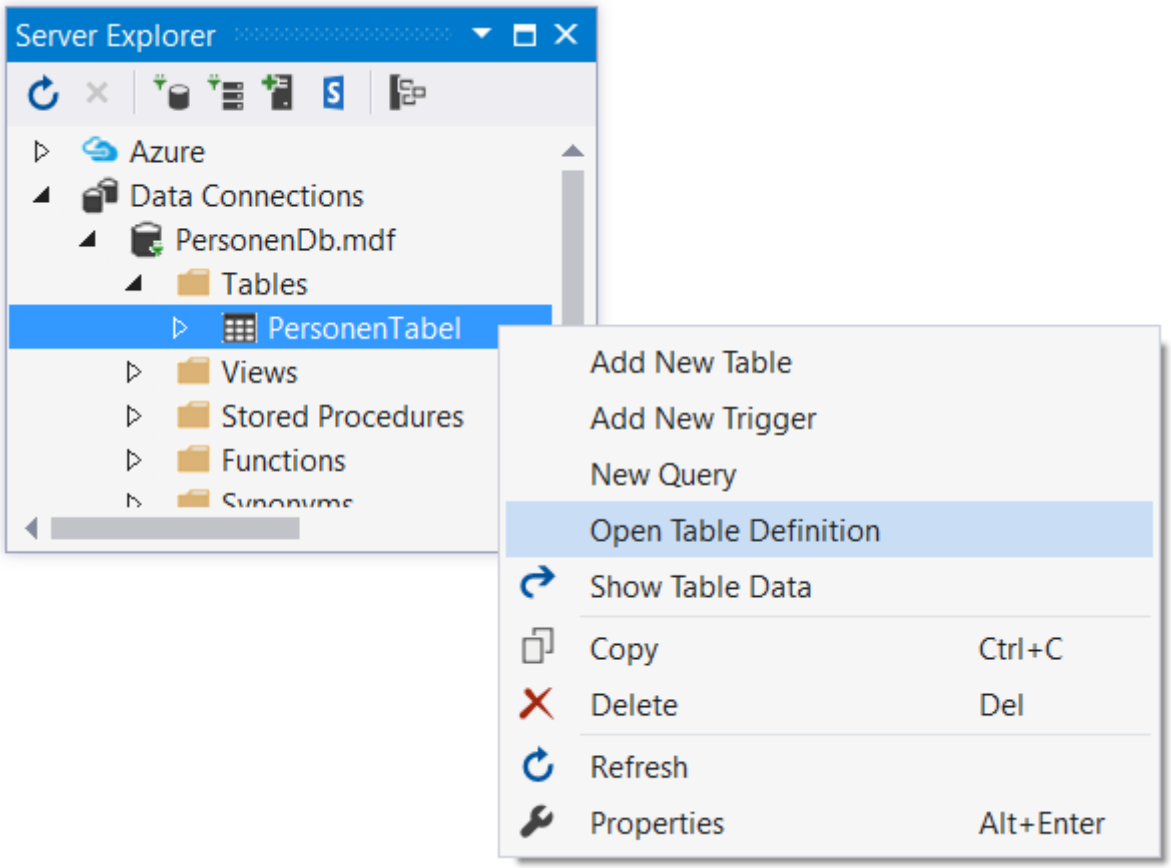
- Wikipedia: Data definiton language: en.wikipedia.org/wiki/Data_definition_language-
- Wikipedia: Insert (SQL): en.wikipedia.org/wiki/Insert_(SQL)

Aan de hand van deze werkwijze kan je ook andere *SQL* statements, zowel ter creatie (*creational language*), manipulatie (*manipulational language*) of raapleging (*selection language*) uitvoeren.

Het in *Visual Studio* gecreëerde tabblad (*SQLQuery1.sql*) mag je sluiten en hoef je niet te bewaren.

2.3. Tabel definitie of informatie verkennen

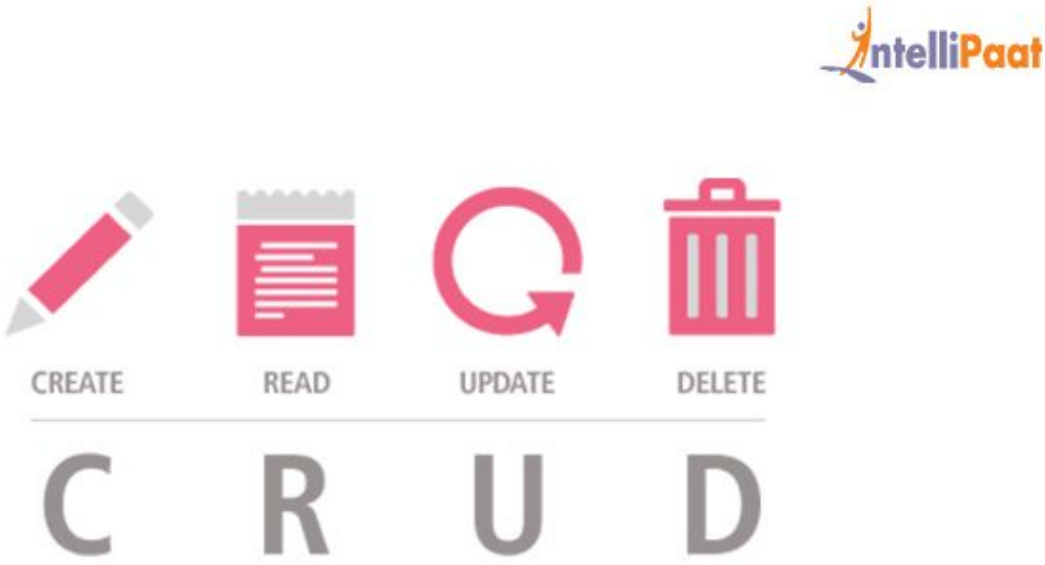
Indien je in de *Server Explorer* de *PersonenDb.dbo* en *Tables* node uitvouwt vindt je een node terug voor de gecreëerde tabel. Rechterklik eventueel op de node en kies voor *Open Table Definition* of *Show Table Data* indien je het dataschema of de opgevulde informatie wil verkennen of zelfs aanpassen:



3. CRUD

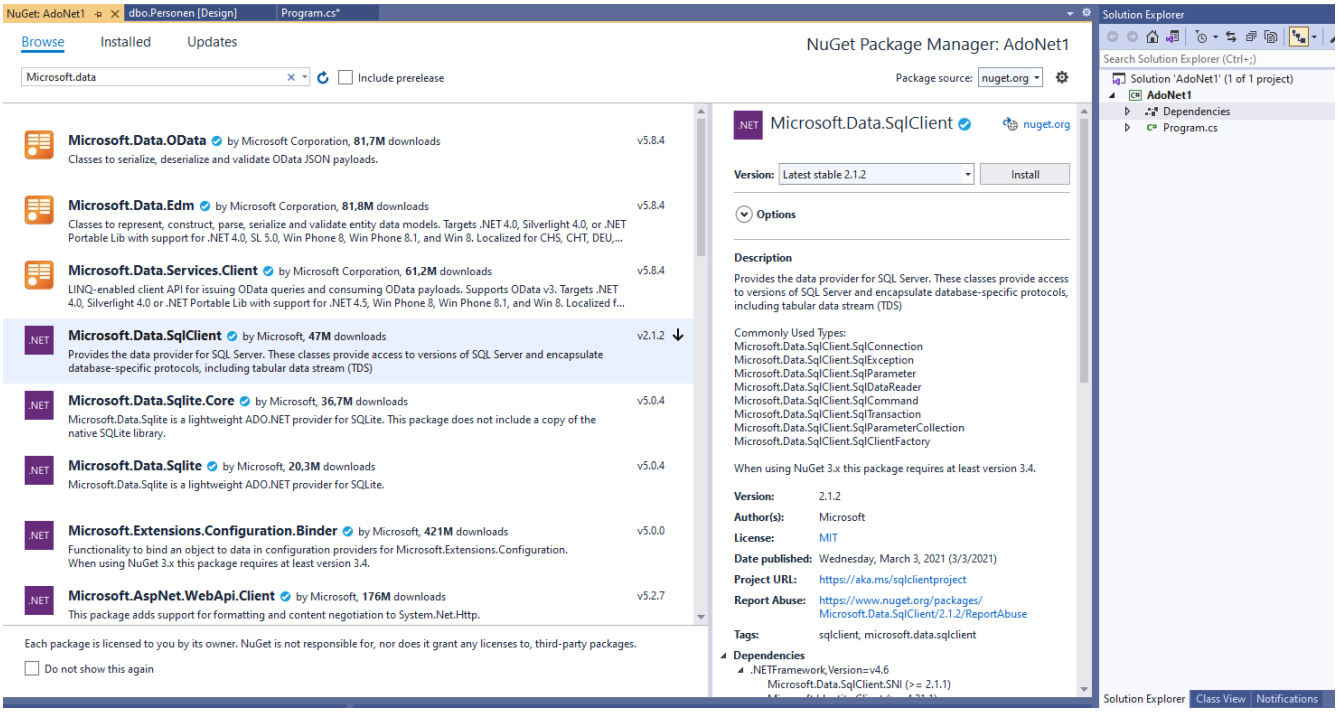
Zie **wikipedia**.

- *C*REATE: het insert statement in geval van SQL
- *R*EAD: het select statement
- *U*PDATE: het update statement
- *D*ELETE: het delete statement



4. ADO.NET gebruiken

Gebruik **NuGet** package manager om de dependency voor de SQL provider toe te voegen aan een project: klik rechts op project, selecteer *Manage Nuget Packages...* en kies onder *Browse*: *Microsoft.Data.SqlClient*. Bevestig de licentie en zie hoe de noodzakelijke packages onder *Dependencies* toegevoegd worden.



> What is the use of NuGet package manager?
>
> NuGet provides the tools developers need for creating, publishing, and consuming packages. Most importantly, NuGet maintains a reference list of packages used in a project and the ability to restore and update those packages from that list.
>
> docs.microsoft.com/en-us/nuget/what-is-nuget
>
> — Microsoft

Vergeet in onderstaande voorbeelden niet de connection string aan te passen naar deze welke je kan vinden onder: **Server Explorer** > **Data Connections:** rechterklik naar *Properties* en dan **Connections** > **Connection string**.

4.1. Connecteren

Om informatie uit een database te kunnen benaderen gebruik je een *verbinding* (*connectie*) met de database. Deze verbinding kan tot stand worden gebracht aan de hand van een *connection object*.

Deze verbinding blijft open gedurende de verwerking van de gegevens. In het geheugen zit telkens slechts één record dat later kan bewerkt worden via specifieke commando's.

Afhankelijk van het type database management server (*Microsoft SQL Server*, *Oracle*, *MySQL*, *Microsoft Access*, ...) zal je gebruik moeten maken van een connection object van een bepaald datatype:

- System.Data.SqlClient.SqlConnection voor Microsoft *SQL Server* databases.

- System.Data.OracleClient.OracleConnection voor *Oracle* databases.
- MySql.Data.MySqlClient.MySqlConnection voor *MySQL* databases.
- System.Data.Odbc.OdbcConnection of System.Data.OleDb.OleDbConnection voor *Microsoft Access* databases.

Zoals je ziet zijn vele van deze types vervat in de System.Data namespace van de klassenbibliotheek van het .NET Framework.

Voor SQL Server maken we gebruik van volgende klassen:

- System.Data.SQLClient.SqlConnection: dit is de klasse om de verbinding met de database te openen en te sluiten:
- System.Data.SQLClient.SqlCommand: dit is de klasse om gegevens op te vragen of aan te passen. Meestal is dit een SQL-instructie die uitgevoerd wordt bij het oproepen van de SQLCommand.

4.2. Klasse SQLConnection

Bij creatie van een connection object kan je aan de constructor doorgeven welke datasource wordt geconnecteerd. Wat de locatie is van waarop de datasource wordt geconnecteerd, wat de authenticatie (gebruikersnaam en wachtwoord) is voor het connecteren, … .

Al deze informatie wordt samen in één string ondergebracht, de *connectionstring*, die je kan doorgeven aan de constructor van het *connection object*:

```
1 string connectionString = @"Data Source=.\SQLEXPRESS; Initial Catalog=PersonenDB;
  Integrated Security=True";
2 SqlConnection conn = new SqlConnection(connectionString);
```

Een connectie string is een tekstuele representatie van de informatie die nodig is om verbinding te maken met een database in SQLExpress. Het bevat informatie zoals de naam van de server, de naam van de database, de gebruikersnaam en het wachtwoord.

Visual Studio beschikt over de mogelijkheid om op basis van enkele configuratiestappen een connectionstring voor ons samen te stellen.

4.3. Openen en sluiten van een connectie

Een *connection object* stelt de verbinding voor maar is pas bruikbaar wanneer de connectie geopend wordt. Om de connectie te openen gebruik je de Open() method:

```
1 SqlConnection conn = new SqlConnection(connectionString);
2 conn.Open();
```

Een connectie is een resource en wordt gedeeld met andere gebruikers. We gaan dus de verbinding

naar deze resource zo snel mogelijk terug vrijgeven indien we ze niet meer nodig hebben.

Om de connectie af te sluiten gebruik je de `Close()` method indien deze connectie later nog opnieuw wordt geopend (via de `Open()` method), of `Dispose()` indien deze connectie verderop niet meer wordt gebruikt.

De klasse SqlConnection implementeert de interface `IDisposable`: de resource kan dus automatisch gesloten worden als we het keyword `using` gebruiken.

Het afsluiten van de resource via `using` of in een finally gedeelte van een try statement wordt beschouwd als *good practice*.

```
1  using (SqlConnection conn = new SqlConnection(connectionString))
2  {
3    conn.Open();
4    //...
5  }
```

Een `using` statement garandeert het *disposen* (aanroepen van de Dispose() method) van de resource, hier het database connection object, aan het eind van het `using` statement.

Zelfs al zou er een exception optreden in de body van het statement, dan nog zal de resource *gedisposed* worden nog vooraleer de call stack wordt afgedaald om te zoeken naar afhandeling. Hierbij wordt dus hetzelfde effect bekomen als bovenstaand try statement.

Indien de connectie verderop nog wordt gebruikt, en dus opnieuw wordt geopend, moet je in plaats van Dispose() de Close() method te gebruiken.

```
1  SqlConnection conn;
2  using (conn = new SqlConnection(connectionString))
3  {
4      conn.Open();
5      //...
6      conn.Close();
7      //...
8      conn.Open()
9      //...
10     conn.Close();
11     //...
12     conn.Open()
13     //...
14     conn.Dispose()
15 }
```

4.4. Klasse SqlCommand

Een SqlCommand object laat toe om een query uit te voeren op een database of om er commando's naar toe te sturen. Het object is verbonden met een SqlConnection object.

Er worden twee methodes voorzien:

- ExecuteReader: voor het uitvoeren van een query
- ExecuteNonQuery: voor het uitvoeren van een commando (insert, update en delete commando's)

4.4.1. ExecuteReader - Lezen uit de database

Indien je record-informatie van een datasource wil uitlezen kan je gebruik maken van een *data reader* (System.Data.SqlClient.SqlDataReader) object.

Een *data reader* is een *connected* (*verbonden*/*life*) browser object om door bepaalde record informatie te navigeren.

Aan de hand van een *command* (SqlCommand) object, die zowel op de hoogte is van de te gebruiken verbinding als de selectie query, kan je een *data reader* instantiëren. Dit via de `ExecuteReader()` method.

```
1         public static void PrintPersonsTable(string connString)
2         {
3             using (SqlConnection connection = new(connString))
4             {
5                 connection.Open();
6
7                 SqlCommand command = new("SELECT Id, Naam, Email, Geboortedatum
  FROM Personen;", connection);
8
9                 using (SqlDataReader dataReader = command.ExecuteReader())
10                {
11                    Console.WriteLine();
12                    if (dataReader.HasRows)
13                    {
14                        while (dataReader.Read())
15                        {
16                            int id = (int)dataReader["Id"];
17                            string naam = (string)dataReader["Naam"];
18                            string email = null;
19
20                            if (!dataReader.IsDBNull(dataReader.GetOrdinal(
  "Email")))
21                            {
22                                email = (string)dataReader["Email"];
```

```
23                          }
24
25                          DateTime geboortedatum =
(DateTime)dataReader["Geboortedatum"];
26                          Console.WriteLine($"{id} | {naam} | {(email is null ?
"N/A" : email)} | {geboortedatum.ToShortDateString()}");
27                      }
28                  }
29                  else { Console.WriteLine("Geen personen gevonden."); }
30              }
31
32              connection.Close();
33          }
34      }
```

Uitvoer:

```
1 1 | Jan | jan @hotmail.com | 27/01/2001
2 2 | Pol | pol @gmail.com | 28/02/2002
3 3 | Piet | piet @hotmail.com | 29/03/2003
```

- Elke `Read()` opdracht zal binnen de *reader* de *cursor* laten wijzen naar het volgende record. *False* wordt opgeleverd indien geen verdere record informatie wordt gevonden.
- Een opzet als while (dataReader.Read()) wordt gebruikt om alle record informatie te benaderen. De HasRows property wordt gebruikt om na te gaan of er wel records werden gevonden.
- Tussen vierkante haakjes kan je verwijzen naar de uit te lezen *kolom*. De in object vorm opgeleverde waarde wordt in het voorbeeld telkens gecast naar het gewenste datatype.

Ook een *data reader* is een resource en dient afgesloten te worden.

Gebruik van DBNull

Indien een waarde in de database `null` kan zijn, dien je hier rekening mee te houden bij het inlezen.

DBNull is een object in .NET dat wordt gebruikt om aan te geven dat een waarde in een database kolom null is. Als een kolom null is, zal de SqlDataReader de waarde `DBNull.Value` teruggeven. Je kunt deze waarde controleren met de `IsDBNull` methode van de SqlDataReader object.

In volgend voorbeeld zou het Email adress `null` kunnen zijn, je vangt dit als volgt op in de code:

```
1 if (dataReader.HasRows)
2 {
3     while (dataReader.Read())
4     {
```

```
5           int id = (int)dataReader["Id"];
6           string naam = (string)dataReader["Naam"];
7
8           string email = null;
9
10          if (!dataReader.IsDBNull(dataReader.GetOrdinal("Email"))) {
11              email = (string) dataReader["Email"];
12          }
13
14          DateTime geboortedatum =
    (DateTime)dataReader["Geboortedatum"];
15          Console.WriteLine($"{id} | {naam} | {(email is null ? "N/A"
    : email)} | geboortedatum.ToShortDateString()}");
16      }
17  }
```

De methode `GetOrdinal` geeft de index terug van de kolom met als titel *Email*.

4.4.2. ExecuteNonQuery

Command objecten kunnen ook gebruikt worden om *SQL data manipulation language* (*delete*, *insert* en *update* instructies) mee uit te voeren. In dat geval gebruik je de **ExecuteNonQuery()** method...

```
1           public static void InsertRow(string connString)
2           {
3               using (SqlConnection connection = new(connString))
4               {
5                   connection.Open();
6                   Console.Write("\nInvoegen nieuwe persoon met naam John,
    geboortedatum 10 oktober 1979...");
7                   string insertSql = $"INSERT INTO Personen (Naam, Email,
    Geboortedatum) VALUES (@Name, @Email, @BirthDate);";
8                   SqlCommand insertCommand = new(insertSql, connection);
9
10                  insertCommand.Parameters.Add("@Name", SqlDbType.VarChar);
11                  insertCommand.Parameters["@Name"].Value = "John";
12
13                  insertCommand.Parameters.Add("@Email", SqlDbType.VarChar);
14                  insertCommand.Parameters["@Email"].Value = "john@mail.be";
15
16                  insertCommand.Parameters.Add("@BirthDate", SqlDbType.DateTime);
17                  insertCommand.Parameters["@BirthDate"].Value = DateTime.Now;
18
19                  insertCommand.ExecuteNonQuery();
20                  connection.Close();
21              }
22          }
23
24          public static void DeleteRow(string connString)
```

```
        {
            using (SqlConnection connection = new(connString))
            {
                connection.Open();
                Console.Write("\nVerwijderen persoon met id 3: ");
                string deleteSql = $"DELETE FROM Personen WHERE Id = {3};";
                SqlCommand deleteCommand = new(deleteSql, connection);
                deleteCommand.ExecuteNonQuery();
                connection.Close();
            }
        }

        public static void UpdateRow(string connString)
        {
            using (SqlConnection connection = new(connString))
            {
                connection.Open();
                Console.Write("\nAanpassen naam in James van persoon met id 2: ");
                string updateSql = $"UPDATE Personen SET Naam = @Name WHERE Id =
@Id; ";
                SqlCommand updateCommand = new(updateSql, connection);

                updateCommand.Parameters.AddWithValue("@Name", "James");
                updateCommand.Parameters.AddWithValue("@Id", 2);

                updateCommand.ExecuteNonQuery();
                connection.Close();
            }
        }
```

Uitvoer:

```
1 | Jan | jan@hotmail.com | 27/01/2001
2 | Pol | pol@gmail.com | 28/02/2002
3 | Piet | piet@hotmail.com | 28/02/2002

Aanpassen naam in James van persoon met id 2:
1 | Jan | jan@hotmail.com | 27/01/2001
2 | James | pol@gmail.com | 28/02/2002
3 | Piet | piet@hotmail.com | 28/02/2002

Invoegen nieuwe persoon met naam John, geboortedatum 10 oktober 1979...
1 | Jan | jan@hotmail.com | 27/01/2001
2 | James | pol@gmail.com | 28/02/2002
3 | Piet | piet@hotmail.com | 28/02/2002
4 | John | john@mail.be | 31/10/1979

Verwijderen persoon met id 3:
1 | Jan | jan@hotmail.com | 27/01/2001
2 | James | pol@gmail.com | 28/02/2002
```

```
19 4 | John | john@mail.be | 31/10/1979
```

Gebruik van DBNull

Indien een waarde in de database `null` kan zijn, dien je hier rekening mee te houden bij manipulatie van de database.

In volgend voorbeeld zou het Email adress `null` kunnen zijn, je vangt dit als volgt op in de code:

```
using (SqlConnection connection = new(connString))
{
    String newEmail = null;

    connection.Open();
    Console.Write("\nAanpassen naam in James van persoon met id 2: ");
    string updateSql = $"UPDATE Personen SET Email = @Email WHERE Id = @Id; ";
    SqlCommand updateCommand = new(updateSql, connection);

    if (newEmail is null)
        updateCommand.Parameters.AddWithValue("@Email", DBNull.Value);
    else
        updateCommand.Parameters.AddWithValue("@Email", newEmail);

    updateCommand.Parameters.AddWithValue("@Id", 2);

    updateCommand.ExecuteNonQuery();
    connection.Close();
}
```

Gebruik van Parameters: door een SqlCommand object te gebruiken en parameters toe te voegen aan de `Parameters` collectie van het SqlCommand object worden SQL-injectie-aanvallen voorkomen en de prestaties van het uitvoeren van de query verbeterd!

4.5. ExecuteScalar en parameters

De `ExecuteScalar` methode wordt gebruikt om één enkele waarde (een enkele rij en een enkele kolom) van een database op te halen *als resultaat* van een SQL-query. De waarde kan elk SQL-gegevenstype zijn, zoals een getal, tekenreeks, datum, enz.

Het gebruik van `ExecuteScalar` is handig wanneer je slechts één enkele waarde uit de database wilt ophalen, zoals de som van een kolom, het aantal rijen in een tabel, het hoogste of laagste getal in een kolom, enz. na het uitvoeren van een

query.

De ExecuteScalar methode retourneert een object, dat de enkele waarde uit de database bevat. Dit object kan van elk gegevenstype zijn, dus je moet dit object casten naar het juiste gegevenstype dat je in de query hebt gebruikt.

ExecuteScalar is interessant om een identity waarde terug te krijgen bij een insert query of delete query of update query. Voor een insert query kan dat op een paar manieren:

```
1         public static int AddPerson(string newName, string email, string
  connString)
2         {
3             Int32 newProdID = 0;
4
5             string sql =     "INSERT INTO Personen (Naam, Email, Geboortedatum) " +
6                              "OUTPUT inserted.Id " +
7                              "VALUES (@Name, @Email, @BirthDate); ";
8
9             using (SqlConnection conn = new(connString))
10            {
11                SqlCommand cmd = new(sql, conn);
12
13                cmd.Parameters.Add("@Name", SqlDbType.VarChar);
14                cmd.Parameters["@Name"].Value = newName;
15
16                cmd.Parameters.Add("@Email", SqlDbType.VarChar);
17                if (email is null)
18                    cmd.Parameters["@Email"].Value = DBNull.Value;
19                else
20                    cmd.Parameters["@Email"].Value = email;
21
22                cmd.Parameters.Add("@BirthDate", SqlDbType.DateTime);
23                cmd.Parameters["@BirthDate"].Value = DateTime.Now;
24
25                try
26                {
27                    conn.Open();
28                    newProdID = (Int32)cmd.ExecuteScalar();
29                }
30                catch (Exception ex)
31                {
32                    Console.WriteLine(ex.Message);
33                }
34            }
35            return (int)newProdID;
36        }
```

5. Inleiding Transactions

Transacties in een database zijn een mechanisme om ervoor te zorgen dat meerdere databasebewerkingen als één enkele, atomische eenheid worden uitgevoerd.

Een transactie wordt gebruikt om ervoor te zorgen dat als een van de bewerkingen mislukt, alle bewerkingen in de transactie worden teruggedraaid, zodat de database weer in de oorspronkelijke staat wordt hersteld.

In databasesystemen zoals SQL Server worden transacties vaak gebruikt om de integriteit van de database te waarborgen. Als een transactie wordt uitgevoerd en een van de bewerkingen mislukt, wordt de transactie teruggedraaid en worden alle bewerkingen in de transactie ongedaan gemaakt, zodat de database in zijn oorspronkelijke staat wordt hersteld. Dit helpt om inconsistenties in de gegevens te voorkomen en de integriteit van de database te waarborgen.

Een transactie bestaat uit een reeks databasebewerkingen die allemaal ofwel moeten slagen ofwel allemaal moeten worden teruggedraaid. Dit wordt bereikt door de transactie te beginnen met de transactie-startopdracht en deze te eindigen met een commit of rollback-opdracht.

Kortom, transacties zijn een belangrijk mechanisme in een database om ervoor te zorgen dat meerdere bewerkingen als één enkele, atomische eenheid worden uitgevoerd en om de integriteit van de database te waarborgen.

5.1. Data consistentie

Bekijk volgende diagram. We hebben een `Accounts` tabel hebben met twee Accounts.

AccountNumber	CustomerName	Balance
Account1	James	1000
Account2	Smith	1000

Om 500 over te maken van Account1 naar Account2 moeten we twee update statements schrijven zoals hieronder getoond. Het eerste update statement trekt 500 af van Account1 en het tweede update statement voegt 500 toe aan Account2:

```
1 UPDATE Accounts SET Saldo = Saldo - 500 WHERE AccountNumber = 'Account1';
2 UPDATE Accounts SET Saldo = Saldo + 500 WHERE AccountNumber = 'Account2';
```

Onze bedoeling is dat de gegevens consistent zijn. Zodra de update statements zijn uitgevoerd, moeten de gegevens in een consistente staat zijn:

Geval 1

Het eerste update statement is succesvol uitgevoerd, maar het tweede update statement is mislukt. In dat geval is er 500 afgetrokken van Account1, maar dat bedrag is niet toegevoegd aan

Account2, wat resulteert in inconsistentie van de gegevens.

Geval 2

Het eerste update statement is mislukt maar het tweede update statement is met succes uitgevoerd. In dat geval wordt 500 niet afgetrokken van Account1, maar wordt 500 toegevoegd aan Account2, waardoor er inconsistentie in de gegevens ontstaat.

Geval 3

Wanneer beide update statements mislukken, dan zijn de gegevens in een consistente staat.

Geval4

Wanneer beide update statements correct uitgevoerd worden, dan zijn de gegevens ook in een consistente staat. Dat wil zeggen 500 wordt afgetrokken van Account1 en 500 wordt toegevoegd aan Account2.

Van de vier hierboven besproken gevallen, hebben we geen problemen in geval 3 en 4. Tegelijkertijd kunnen we ook niet de garantie geven dat elke keer beide update statements mislukken en slagen. Dat betekent dat we iets speciaals moeten doen om geval 1 en 2 af te handelen, zodat de gegevens in een consistente toestand blijven en daarvoor moeten we transacties gebruiken.

5.2. Transacties

Een transactie is een reeks operaties die ervoor zorgen dat alle databasebewerkingen slagen of dat ze allemaal mislukken om de consistentie van de gegevens te waarborgen. Dit betekent dat het werk nooit half gedaan is, of alles is gedaan of er is niets gedaan.

5.2.1. BeginTransaction

In volgend voorbeeld voeren we twee update-statements uit met behulp van ADO.NET `Transaction`. We starten met volgende gegevens:

AccountNumber	CustomerName	Balance
Account1	James	1000
Account2	Smith	1000

Begin met een nieuwe database *AccountDB* aan te maken en maak een tabel Accounts aan:

```
1 CREATE TABLE Accounts
2 (
3       AccountNumber VARCHAR(60) PRIMARY KEY,
4       CustomerName VARCHAR(60),
5       Balance int
6 );
7 GO
```

```
INSERT INTO Accounts VALUES('Account1', 'James', 1000);
INSERT INTO Accounts VALUES('Account2', 'Smith', 1000);
GO
```

```
    public class DB_Transacties
    {
        private const String ConnectionString = @"Data Source =.\SQLEXPRESS;Initial Catalog = AccountDB; Integrated Security = True";
        public static void Main(string[] args)
        {
            try
            {
                Console.WriteLine("Before Transaction");
                GetAccountsData();
                MoneyTransfer();
                Console.WriteLine("After Transaction");
                GetAccountsData();
            }
            catch (Exception e)
            {
                Console.WriteLine("OOPs, something went wrong" + e.Message);
            }
        }
        private static void MoneyTransfer()
        {
            using (SqlConnection connection = new(ConnectionString))
            {
                // The connection needs to be open before we begin a transaction
                connection.Open();
                // Create the transaction object by calling the BeginTransaction method on connection object
                SqlTransaction transaction = connection.BeginTransaction();
                try
                {
                    // Associate the first update command with the transaction
                    SqlCommand cmd = new SqlCommand("UPDATE Accounts SET Balance = Balance - 500 WHERE AccountNumber = 'Account1'",
                        connection,
                        transaction);
                    cmd.ExecuteNonQuery();

                    // Associate the second update command with the transaction
                    /*
                    cmd = new SqlCommand("UPDATE Accounts SET Balance = Balance + 500 WHERE AccountNumber = 'Account2'",
                        connection,
                        transaction);
                    /*/
                    cmd = new SqlCommand("UPDATE MyAccounts SET Balance = Balance + 500 WHERE AccountNumber = 'Account2'",
```

```
                        connection,
                        transaction);
                    //*/

                    cmd.ExecuteNonQuery();

                    // If everythinhg goes well then commit the transaction
                    transaction.Commit();
                    Console.WriteLine("Transaction Committed");
                }
                catch
                {
                    // If anything goes wrong, rollback the transaction
                    transaction.Rollback();
                    Console.WriteLine("Transaction Rollback");
                }
            }
        }
        private static void GetAccountsData()
        {
            using (SqlConnection connection = new SqlConnection(ConnectionString))
            {
                connection.Open();
                SqlCommand cmd = new("Select * from Accounts", connection);
                SqlDataReader sdr = cmd.ExecuteReader();
                while (sdr.Read())
                {
                    Console.WriteLine(sdr["AccountNumber"] + ",  " +
sdr["CustomerName"] + ",  " + sdr["Balance"]);
                }
            }
        }
    }
```

Uitvoer:

```
Before Transaction
Account1,  James,  1000
Account2,  Smith,  1000
Transaction Committed
After Transaction
Account1,  James,  500
Account2,  Smith,  1500
```

5.2.2. Verifieer de consistentie

Laten we het programma wijzigen door aan het tweede command de verkeerde tabel mee te geven. Dit is een wijziging die de applicatie zou laten crashen at runtime na het uitvoeren van het eerste update statement.

Je ziet dat een volledige rollback wordt uitgevoerd van de transactie:

```
1 Before Transaction
2 Account1,  James,  1000
3 Account2,  Smith,  1000
4 Transaction Rollback
5 After Transaction
6 Account1,  James,  1000
7 Account2,  Smith,  1000
```

Opgelet: zolang een transactie open staat, bijvoorbeeld in de debugger, worden andere operaties op de betrokken tabel(len) geblokkeerd.