Gegevensbeheer en beveiliging – Les 1 – SQLite en Entity Framework

In deze les gaan we aan de slag met een eenvoudige SQL database: SQLite. SQLite is een minimalistische database die de gegevens van de tabellen opslaat in één bestand. Daardoor is de database gemakkelijk te verplaatsen en is het een handige manier om gestructureerde data snel en eenvoudig op te slaan. Het voordeel is dat je voor SQLite geen apart server-proces moet hebben draaien zoals bij MySQL of MariaDB. Alle gegevens zitten bij SQLite gewoon in één bestand.

Vermits SQLite een *light* database engine is zijn er een aantal zaken weggelaten: gebruikersrollen bestaan er niet in de SQLite wereld en er zijn maar 5 types voor de velden:

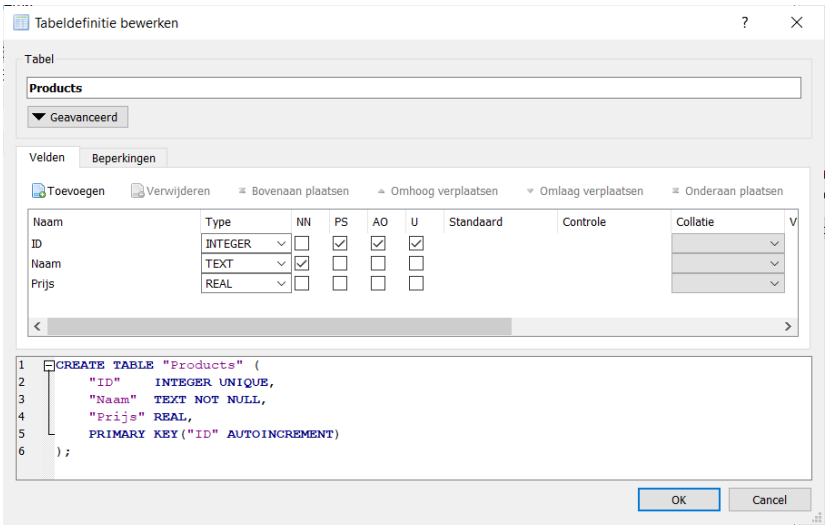
- NULL
- INTEGER (= geheel getal)
- REAL (= kommagetal)
- TEXT
- BLOB (= voor willekeurige gegevens)

https://www.sqlite.org/datatype3.html

Als je meer opties wil, zoals gebruikers met verschillende rechten en andere veldtypes, kies je beter toch voor MySQL of MariaDB.

Database aanmaken

We gaan een kleine database aanmaken met 1 tabel. We gebruiken hiervoor het programma https://sqlitebrowser.org/ (= open source). Download het programma en maak een database met daarin een tabel Products met een integer **ID** (= primaire sleutel die automatisch verhoogt), een tekstveld **Naam** en **Prijs** waar een komma-getal in past:



Bewaar het bestand als **database.db**. Let er op dat je het database bestand op de juiste plaats zet. Als je het programma nog aan het ontwikkelen bent is dat in de map *bin\Debug\netcoreapp3.1*

Class Product aanmaken

Maak in Visual Studio een class Product aan met dezelfde gegevens (als properties) als de tabel Products.

```
public class Product
{
    public int ID { get; set; }
    public string Naam { get; set; }
    public float Prijs { get; set; }
}
```

C# en SQLite laten communiceren

Het communiceren tussen een database en C# wordt mogelijk gemaakt door het Entity Framework van Microsoft. Om het te installeren (dat moet per project) ga je naar *Tools > NuGet Package Manager > Package Manager Console*. In het venstertje onderaan type (of plak) je:

```
Install-Package Microsoft.EntityFrameworkCore.Sqlite
```

Vergeet niet af te sluiten met de enter-toets.

We hebben dit using statement ook nog nodig:

```
using Microsoft.EntityFrameworkCore;
```

Om onze class Product met de database te laten spreken voegen we ook nog onverstaande class toe, hier wordt de naam van de database-file vermeld. *Products* is de naam van de tabel. Let goed op of je de bestandsnaam en de tabelnaam juist geschreven hebt .

```
public class ShopContext : DbContext
{
    public DbSet<Product> Products { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseSqlite("Data Source=database.db");
}
```

Tabel aanpassen

De tabel in de database is nu gekoppeld. We kunnen met enkele commando's de tabel gaan manipuleren. De typische CRUD (create, read, update, delete) operaties staan hieronder.

Create

Omdat we met een database aan het werken zijn zet je best een **using**-statement rond de code die de gegevens wijzigt of opvraagt. Na het using-statement wordt de connectie van de database gesloten zodat andere processen de database kunnen bereiken.

```
using (var db = new ShopContext())
{
    db.Add(new Product { Naam = "Emmertje" });
    db.Add(new Product { Naam = "Spons" });
    db.Add(new Product { Naam = "Borstel" });
    db.SaveChanges();
}
```

Read

Deze code zet je ook in het using-blok. Met FirstOrDefault vraag je het eerste element op dat aan een voorwaarde voldoet. First bestaat ook, maar dat geeft een foutmelding als er niet gevonden wordt, FirstOrDefault geeft een *null* waarde.

```
var p = db.Products
    .FirstOrDefault(p => p.Naam == "Spons");
```

Met *Where* kan je een lijst opvragen van elementen die aan een bepaalde voorwaarde voldoen.

```
Console.WriteLine(p.Naam);

var ps = db.Products.Where(p => p.Naam.Contains("s"));
foreach(var p in ps)
{
    Console.WriteLine(p.Naam);
}
```

Je kan ook al de producten in een tabel opvragen en omzetten naar een lijst. Zo kan je al de methodes voor een lijst gaan gebruiken zoals je die reeds kent van de vorige module.

```
var products = db.Products.ToList();

foreach(var p in products)
{
    Console.WriteLine(p.Naam);
}
```

Meer informatie vind je hier:

https://docs.microsoft.com/en-us/ef/core/querying/

Veel van de LINQ methodes van Queryable zijn ook toe te passen op DbSets:

https://docs.microsoft.com/en-us/dotnet/api/system.linq.queryable?view=net-5.0

Update

Eens je een element hebt opgevraagd kan je het ook eenvoudig wijzigen…

```
var p1 = db.Products
    .FirstOrDefault(p => p.Naam == "Spons");

Console.WriteLine($"vorige naam: {p1.Naam}");
p1.Naam = "Vod";
db.SaveChanges();
```

```
Console.WriteLine($"nieuwe naam: {p1.Naam}");
```

Delete

… of verwijderen.

```
var p1 = db.Products
    .FirstOrDefault(p => p.Naam == "Spons");

db.Products.Remove(p1);
```

https://docs.microsoft.com/en-us/ef/core/saving/basic

Oefening: ToDo List

Maak een nieuw project en een nieuwe database en koppel ze.

Maak een programma waarmee je een lijst van taken kan **toevoegen**, **wijzigen** en **verwijderen** in een database. Zorg voor een veld waarmee je de taken ook kan markeren als

- Nog niet begonnen
- In progress
- Afgehandeld