SQL SERVER: ENTITY FRAMEWORK

Softwarové inžinierstvo

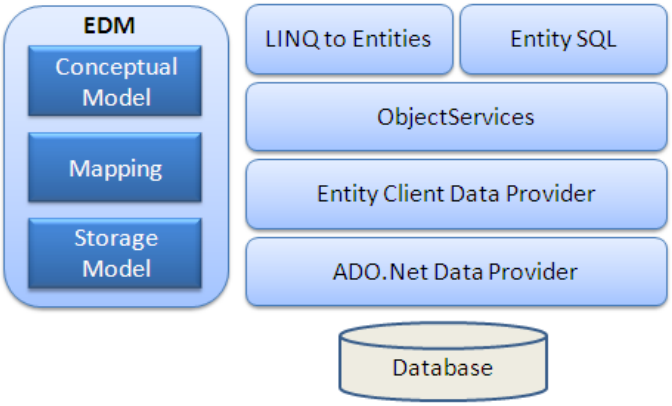
MARTIN TIMOTHY TIMKO

17.9. 2017

1 ÚVOD

Predtým než neexistoval *Entity Framework* programátori používali rôzne verzie ADO.NET, predtým ADO (OleDb), ODBC alebo v začiatkoch 90.rokov aj surové dáta prostredníctvom nejakého priameho programového rozhrania API. Postupom času a najmä s rozvojom objektovo-orientovanej paradigmy sa začali do povedomia dostávať tzv. *objektové databázy*, ktoré čiastočne nahradzujú pôvodné tzv. *relačné databázy*.

Objektovo-orientovaná databáza (*ORM*, Object-Relational Mapping) je systém správy databázy, v ktorom je informácia predstavená vo forme **objektov** podobne ako v objektovo-orientovanom programovaní. S rozvojom objektovo-orientovaného vývoja aplikácií vzišla určitá idea transformovať základné objektové princípy na často používané relačné databázy. Aj napriek tejto vznešenej myšlienke, nie je možné povedať, že objektové databázy plne nahradia relačné databázy, pretože to ani nebol pôvodný zámer.



Obrázok 1: Architektúra Entity Frameworku

Koncepcia objektových databáz začala už v prvopočiatkoch 80.rokov, takže sa nejedná primárne o nejakú špecialnu vymoženosť. Základné koncepcie medzi relačnými a objektovými databázami sú značne odlišné a z toho dôvodu je potrebné vytvárať rôzne medzikroky, ktoré jednotlivé databázy spájajú. A práve konštrukcia týchto medzikrokov je na príčine zníženia výkonu objektových databáz a zvyšovania potencionálych chýb. Teda, objektové databázy je nutné považovať za určitú **alternatívu** k relačným databázam.

2 VÝHODY A NEVÝHODY JEDNOTLIVÝCH DATABÁZ

Výhoda relačných databáz je v jednoduchosti a pochopiteľnosti ich použitia, ktorá je postavená na matematickej teórii, teda logicky organizované dáta v databáze usporiadané do matematických relácií podľa jeho zakladateľa *Edgar F. Codd*. Tieto databázy sú veľmi rozšírené, obsahujú stabilné štandardy a používajú štandardizované dopytovacie jazyky. Na druhej strane takáto jednoduchosť použitia sa stáva problémom pri efektívnom modelovaní zložitejších objektov. V rámci navrhovania tak výrazne zjednodušujú problematiku. Navyše poskytujú len jendoduché dátové typy.

Z tohto dôvodu je zrejmé, že relačné databázy sú vhodné pre správu síce **veľkého objemu dát**, avšak jednoduchej povahy. Obsahujú veľmi dobré mechanizmy pre selekciu dát, ale konkrétna manipulácia s týmito dátami je komplikovanejšia. Navyše je problém s kontrolou napísaných SQL selektov, čo je v Entity Frameworku aspoň čiastočne zabezpečené tzv. *LINQ* a napokon v nám programový kód založený na relačných databázach neposkytuje žiadny *IntelliSense*. V podstate je potrebné na manipuláciu s databázou ovládať ďalší jazyk, teda SQL jazyk, kde v metodike Entity Frameworku nemusíte vedieť jazyk SQL, aspoň v základných operáciách. Ale samozrejme znalosť SQL jazyka bude určite použiteľná aj v prostredí Entity Frameworku a teda LINQ.

Výhodou objektových databáz je možnosť vyjadrenia zložitosti daného problému priamo objektovo-orientovanou metodikou. Vzhľadom k tomu, že súčasťou v takto koncipovaných objektoch je aj ich správanie, zjednodušuje sa **štruktúra aplikácie**. Odstraňujú sa medzikroky pri prevode objektov do tabuliek relačnej databázy pred uložením každého objektu (a vice versa). Objektové triedy, ktoré používame pri programovaní aplikácie sú triedy priamo používané v objektových databázach, teda model databázy a programovacieho jazyka je plne konzistentný. A z toho dôvodu nie je potrebná transformácia programového objektového modelu do špecifického databázového modelu.

Z toho vyplýva, že návrh objektovej databázy je do veľkej miery aj časťou návrhu celej aplikácie, keďže objektový model databázy pozostáva z klasického objektového modelu, ktorý je zároveň používaný v objektovo-orientovanej metodike návrhu informačných systémov. Teda, jednoduchosť objektového modelu pomocou ktorého môžme vytvárať zložitejšie štruktúry je v relačných databázach problémom, pretože je nutné vytvárať pre jednoduchú štruktúru uloženú v databáze objekt, s ktorým následne aplikácia dokáže nejakým spôsobom manipulovať. Všetky tieto medzikroky sú v objektových databázach **zautomatizované**.

Ako už bolo spomínané práve realizácia automatizovaných medzikrokov má za následok jednak **zníženie výkonu** objektových databáz a jednak **zvýšenie potencionálych chýb** vo forme rôznych duplicít a v rozsiahlejších programových systémov aj neprehľadnosť.

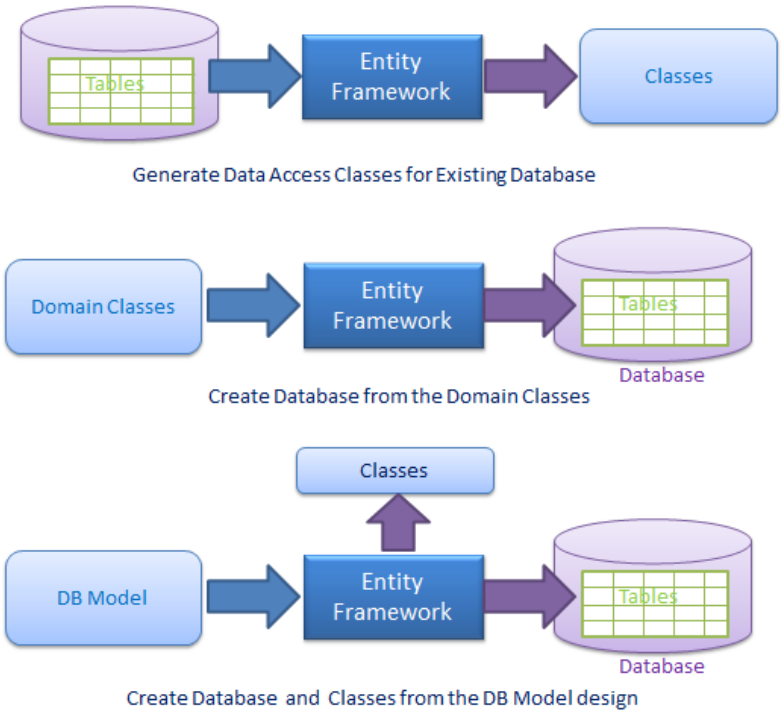
3 VÝKONNOSŤ OBJEKTOVÝCH DATABÁZ

Častým problémom objektových databáz, konkrétne Entity Frameworku, je výkonnosť. Avšak za uplynulý čas sa vývoj Entity Frameworku posunul aj v tejto oblasti na vyššiu úroveň. Aktuálne Entity Framework obsahuje rôzne mechanizmy pomocou ktorých je možné dosiahnuť **veľmi dobré výkonnostné výsledky**. V každom prípade veľká časť problémov je na strane programátorského tímu, ktorý stojí pred voľbou komfortu s určitým kompromisom alebo využitie plného výkonu s väčším úsilím a ťažkopádnosťou. Pre ladenie výkonu a nachádzanie problematických miest existujú rôzne nástroje, tzv. **profilovacie nástroje**, napr. *Entity Framework Profiler*.

4 DATABASE FIRST/MODEL FIRST/CODE FIRST

Entity Framework používa niekoľko prístupov k tvorbe objektového modelu databázy. Jedná sa o určitú logickú reprezentáciu návrhových modelov, ktorú je možné rozdeliť do 3 častí:

- Database First (databázový prístup)
- Model First (modelový prístup)
- Code First (kódový prístup)



Obrázok 2: Prístupy k tvorbe objektových modelov

Databázový prístup je založený na myšlienke, kde prvotný návrh sa realizuje na strane klasickej databázy. To znamená, že sa vytvorí databáza a z nej Entity Framework vygeneruje objektové entity pre ďalšie použitie. Navrhnutú databázu je možné neskôr upraviť a aktualizovať daný objektový model. Entity Framework vytvorí programové triedy založené na tabuľkách z databázy.

Modelový prístup je založený na tvorbe modelu pomocou ktorého sa generuje databáza. Tento prístup je určený skôr pre menšie projekty, pretože nie je možné primárne vykonať manuálne zmeny na strane databázy, keďže nedisponujeme priamym príkazom na databázu, ale na druhej strane je možné vykonať zmeny v modelových triedach. Takže pomocou grafického návrhára modelov je možné nakresliť modely tried a na základe týchto modelových tried sa vygeneruje databáza.

Kódový prístup je založený na tvorbe programových tried z ktorých Entity Framework vygeneruje danú databázu. Pomocou tohto prístupu je možné vykonávať kompletné zmeny priamo z programového kódu. Tento prístup sa v novších verziách a vôbec v celkovej koncepcii Entity Frameworku presadzuje stále viac, pretože vychádza v podstate zo základnej koncepcie objektových databáz a teda, odvrátiť programátora od návrhu fyzickej databázy.

Ono, je možné navrhnúť databázu a aj napriek tomu použiť kódový prístup zhodný z databázovými entitami a k danej databáze sa pripojiť. V každom prípade v ďalších častiach budú prakticky popísané prístupy založené na kóde a databáze, pričom prístup založený na modely je len medzikrokom prístupu založenom na týchto dvoch prístupoch, avšak je možné použiť len prístup založený na modely vo forme grafického rozhrania. V každom prípade sa tento prístup používa len ako nejaká prebežná šablóna návrhu, teda nie je určený pre rozsiahle projekty.

4.1 Databázový prístup (Database First)

Databázový prístup je jednoduchší spôsob návrhu ako neskôr uvidíme, pretože každý krok je do veľkej miery automatizovaný, teda zásah používateľa do pôvodných štrukúr je minimálna.

Ako už z názvu vyplýva, predtým než začneme manipulovať s objektami je potrebné vytvoriť databázu. Konkrétny postup tvorby databázy ako aj tabuľky s nejakým preddefinovaným obsahom bol obsahom publikácie *SQL Server: ADO.NET*. Výsledkom bola jednoduchá tabuľka.

	Id	Name	Age
▶	0	Peter	17
	1	Fero	28
	2	Vlado	47
	3	Jozef	35
	100	Stano	59
	101	Juraj	24
*	NULL	NULL	NULL

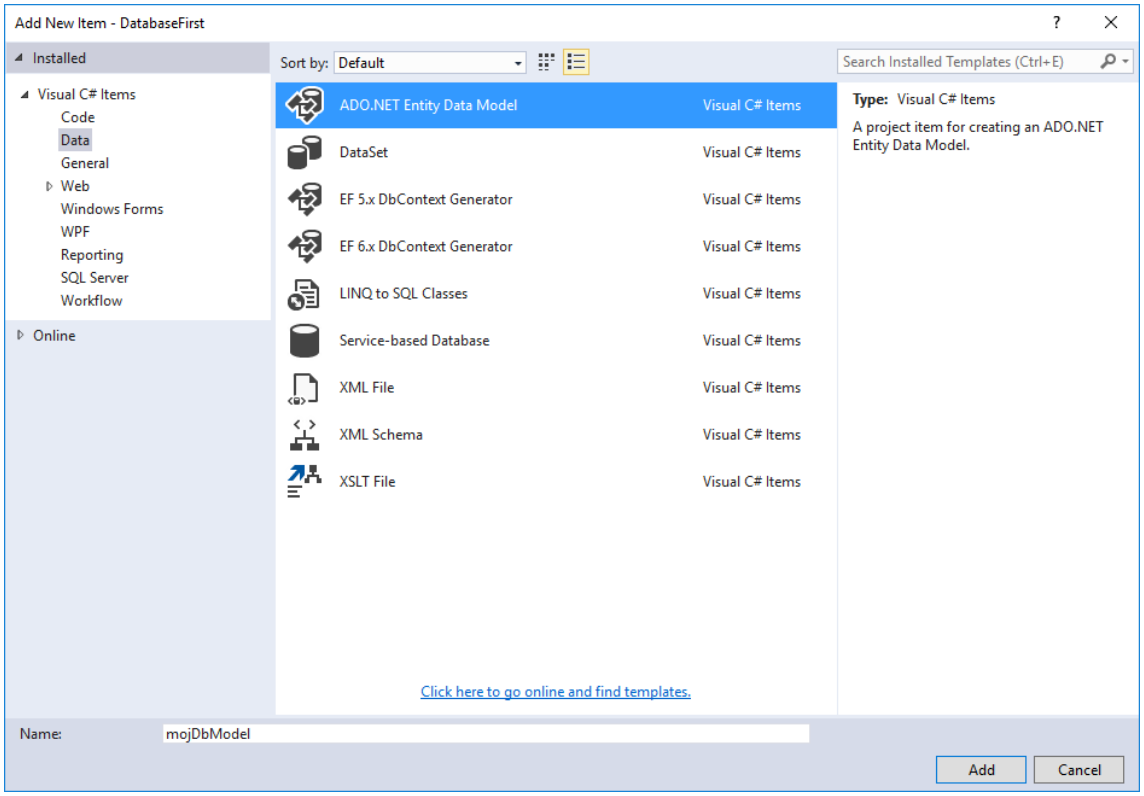
Z tejto tabuľky vygenerujeme objektový model.

Vytvoríme si klasickú prázdnu konzolovú aplikáciu s nasledujúcim kódom:

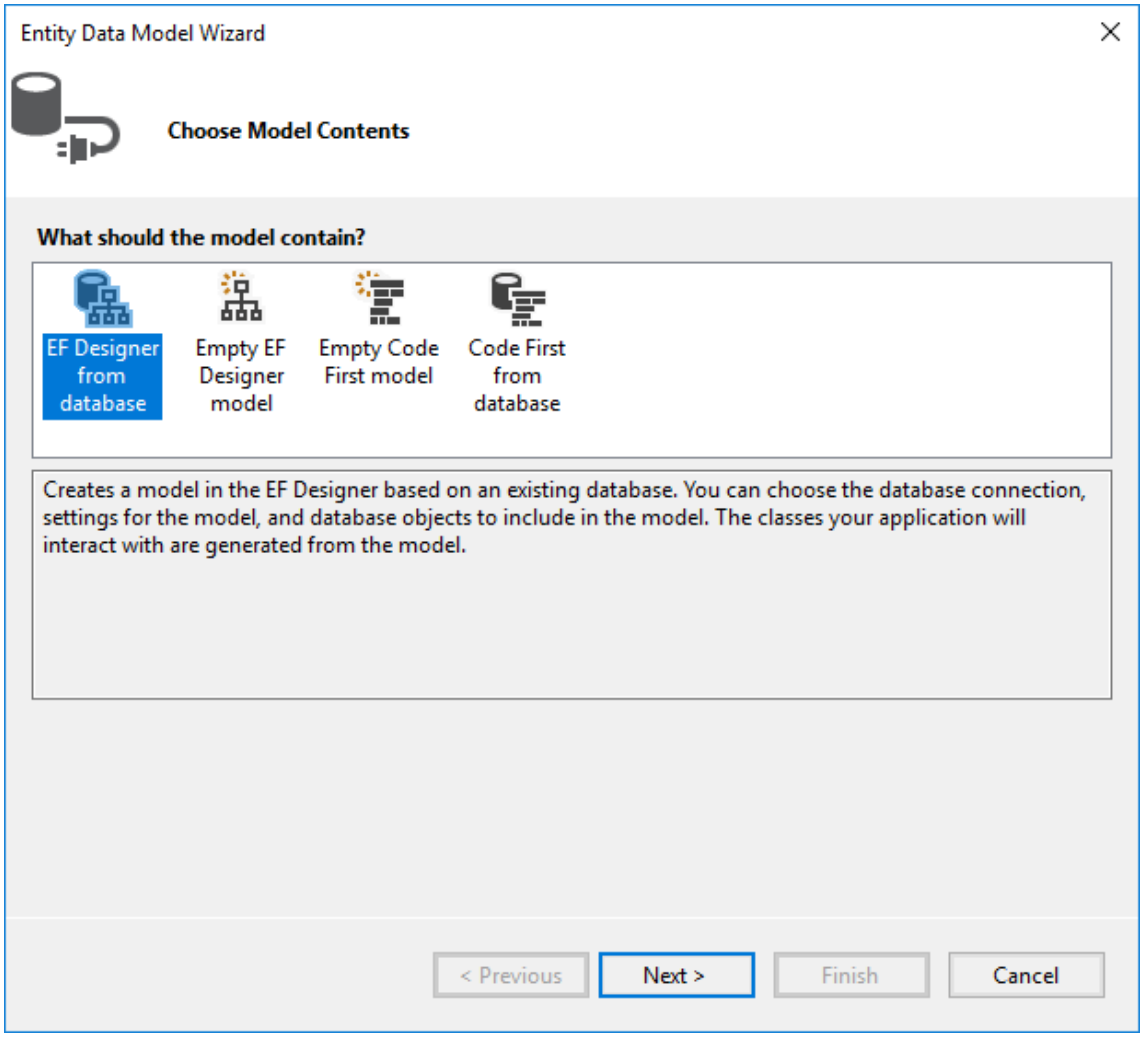
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DatabaseFirst
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Poznámka: Ak nie je uvedené inak, názvy tried, atribútov a podobne by sa mali písať v anglickom jazyku, ale na tomto mieste použijeme kvôli prehľadnosti a viditeľnosti vlastného kódu od automaticky generovaných názvov slovenské názvy.

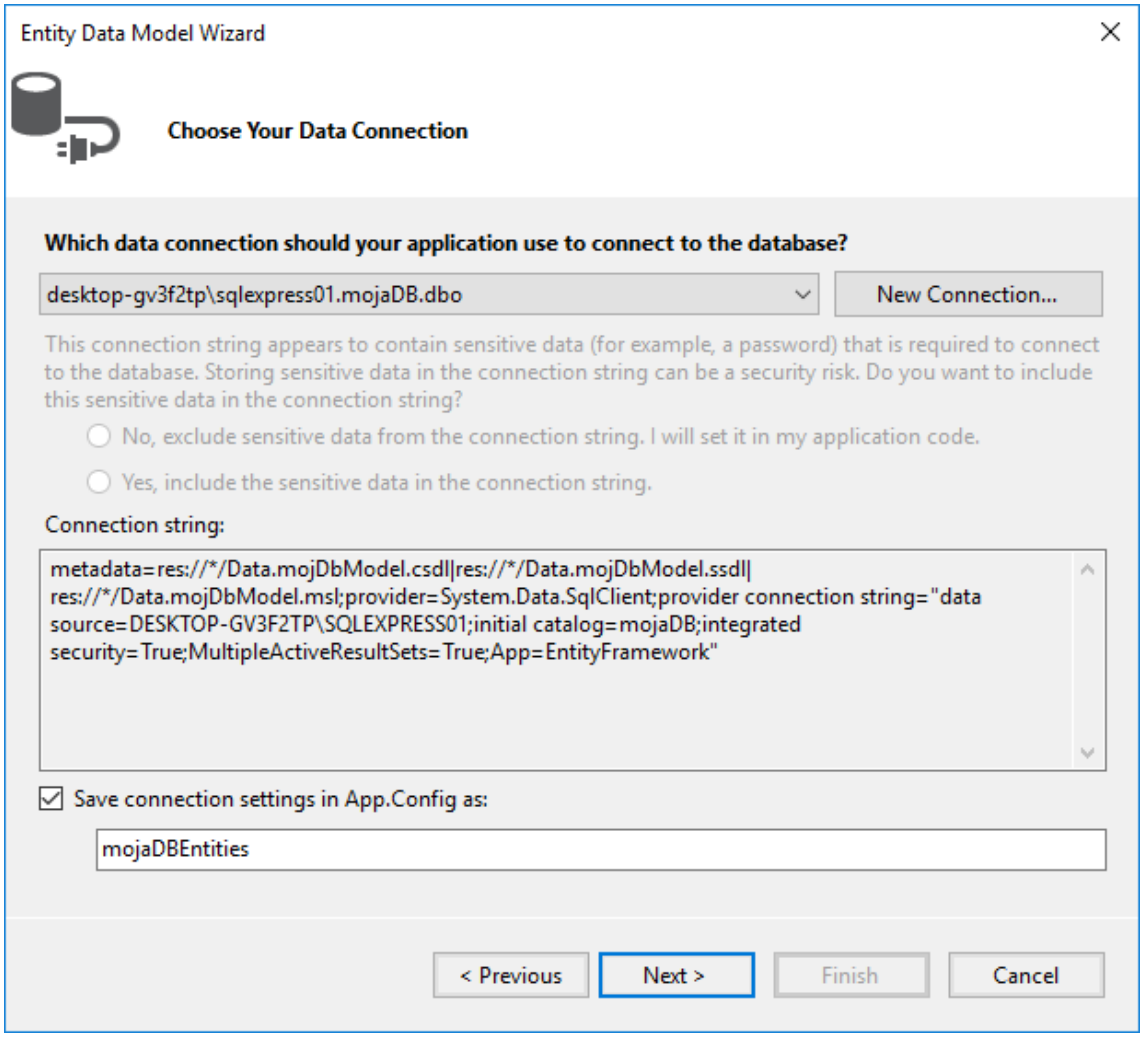
V `Server Explorer` vytvoríme adresár, napr. `"Data"` a pridáme nový objekt `ADO.NET Entity Data Model`.



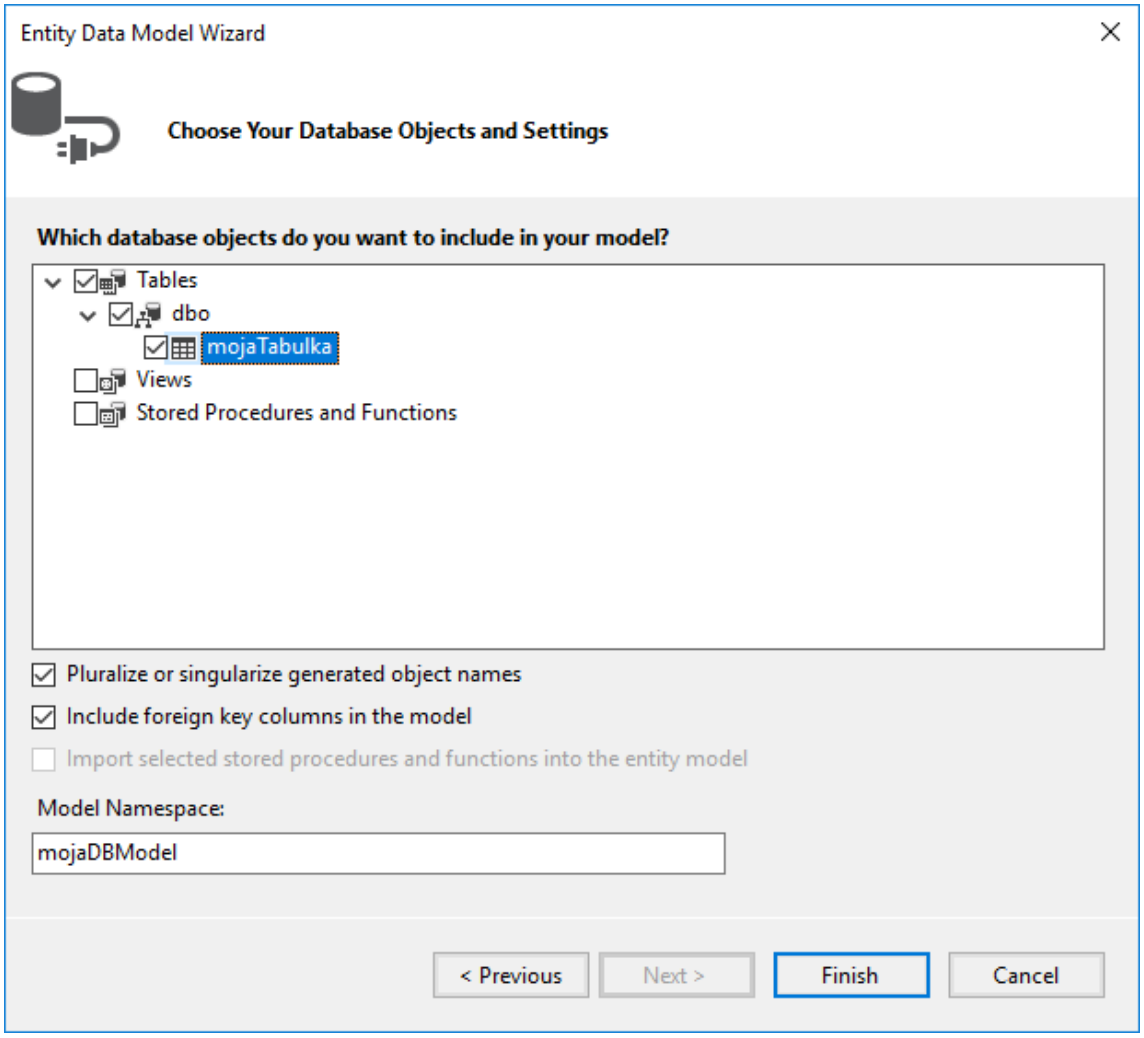
V ďalšom kroku zvolíme spôsob generovania modelu z databázy.



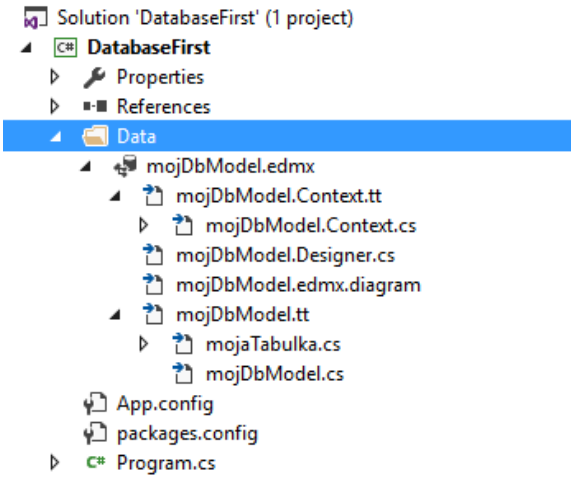
Následne zvolíme databázu a ponecháme údaje v predpísanom tvare, kde nám generátor automaticky pridá pripojovací reťazec do `App.config`, pričom zvolíme verziu Entity Framework 6.x.



Na záver zvolíme ktoré databázové entity chceme zahrnúť do objektového modelu.



Po krátkom čase nám *Visual Studio* vygeneruje pomocou Entity Frameworku model a automaticky pridá referencie do programu.



Súbor s príponou `*.edmx` je obyčajný XML dokument, čitateľný v textovom editore:

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="3.0" xmlns:edmx="http://schemas.microsoft.com/ado/2009/11/edmx">
  <!-- EF Runtime content -->
  <edmx:Runtime>
    <!-- SSDL content -->
    <edmx:StorageModels>
      <Schema Namespace="mojaDBModel.Store" Provider="System.Data.SqlClient" ProviderManifestToken="
        <EntityType Name="mojaTabulka">
          <Key>
            <PropertyRef Name="Id" />
          </Key>
          <Property Name="Id" Type="int" Nullable="false" />
          <Property Name="Name" Type="nvarchar" MaxLength="50" />
          <Property Name="Age" Type="int" />
        </EntityType>
        <EntityContainer Name="mojaDBModelStoreContainer">
          <EntitySet Name="mojaTabulka" EntityType="Self.mojaTabulka" Schema="dbo" store:Type="Table
        </EntityContainer>
      </Schema>
    </edmx:StorageModels>
    <!-- CSDL content -->
    <edmx:ConceptualModels>
      <Schema Namespace="mojaDBModel" Alias="Self" annotation:UseStrongSpatialTypes="false" xmlns:an
        <EntityType Name="mojaTabulka">
          <Key>
            <PropertyRef Name="Id" />
          </Key>
          <Property Name="Id" Type="Int32" Nullable="false" />
          <Property Name="Name" Type="String" MaxLength="50" FixedLength="false" Unicode="true" />
          <Property Name="Age" Type="Int32" />
```

```
        </EntityType>
        <EntityContainer Name="mojaDBEntities" annotation:LazyLoadingEnabled="true">
          <EntitySet Name="mojaTabulkas" EntityType="Self.mojaTabulka" />
        </EntityContainer>
      </Schema>
    </edmx:ConceptualModels>
    <!-- C-S mapping content -->
    <edmx:Mappings>
      <Mapping Space="C-S" xmlns="http://schemas.microsoft.com/ado/2009/11/mapping/cs">
        <EntityContainerMapping StorageEntityContainer="mojaDBModelStoreContainer" CdmEntityContaine
          <EntitySetMapping Name="mojaTabulkas">
            <EntityTypeMapping TypeName="mojaDBModel.mojaTabulka">
              <MappingFragment StoreEntitySet="mojaTabulka">
                <ScalarProperty Name="Id" ColumnName="Id" />
                <ScalarProperty Name="Name" ColumnName="Name" />
                <ScalarProperty Name="Age" ColumnName="Age" />
              </MappingFragment>
            </EntityTypeMapping>
          </EntitySetMapping>
        </EntityContainerMapping>
      </Mapping>
    </edmx:Mappings>
  </edmx:Runtime>
  <!-- EF Designer content (DO NOT EDIT MANUALLY BELOW HERE) -->
  <Designer xmlns="http://schemas.microsoft.com/ado/2009/11/edmx">
    <Connection>
      <DesignerInfoPropertySet>
        <DesignerProperty Name="MetadataArtifactProcessing" Value="EmbedInOutputAssembly" />
      </DesignerInfoPropertySet>
    </Connection>
    <Options>
      <DesignerInfoPropertySet>
        <DesignerProperty Name="ValidateOnBuild" Value="true" />
        <DesignerProperty Name="EnablePluralization" Value="true" />
        <DesignerProperty Name="IncludeForeignKeysInModel" Value="true" />
        <DesignerProperty Name="UseLegacyProvider" Value="false" />
        <DesignerProperty Name="CodeGenerationStrategy" Value="None" />
      </DesignerInfoPropertySet>
    </Options>
    <!-- Diagram content (shape and connector positions) -->
    <Diagrams></Diagrams>
  </Designer>
</edmx:Edmx>
```

Obsahuje kontextuálne modely a rôzne mapovania z nami vytvorenej databázy. Zaujímavý je súbor s príponou `*.tt`, konkrétne v našom prípade `mojModel.tt`, ktorý obsahuje tzv. *T4 šablónu*, ktorá automaticky generuje naše objektové triedy z databázy.

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated from a template.
//
//     Manual changes to this file may cause unexpected behavior in your application.
//     Manual changes to this file will be overwritten if the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace DatabaseFirst.Data
{
    using System;
    using System.Collections.Generic;

    public partial class mojaTabulka
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public Nullable<int> Age { get; set; }
    }
}
```

Samotný programový kód bude vyzerať nasledovne:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using DatabaseFirst.Data;

namespace DatabaseFirst
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var dc = new mojaDBEntities())
            {
                var dopyt = from osoba in dc.mojaTabulkas
                            select osoba;

                foreach (var osoba in dopyt)
                {
                    Console.WriteLine(osoba.Name + "␣" + osoba.Age);
                }

                Console.ReadKey();
```

```
            }
        }
    }
}
```

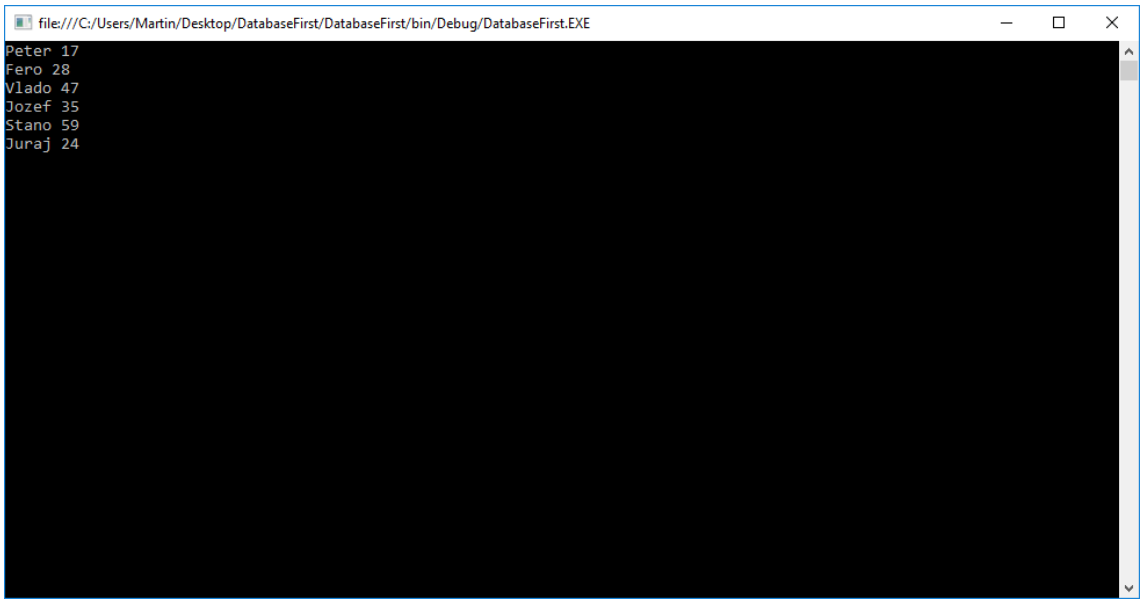
V prvom kroku je nutné vytvoriť inštanciu triedy `mojaDBEntities.tt`, tzv. **dátový kontext**, ktorý bude reprezentovať databázový model. Jedná sa v podstate o ekvivalent `SqlConnection`. Selekcia dát bude realizovaná prostredníctvom LINQ dopytu:

```
from osoba in dc.mojaTabulkas
select osoba;
```

Vyzerá to podobne ako klasický SQL selekt, teda:

```
select * from mojaTabulkas
```

Týmto je možné vypísať všetky položky tabuľky.



Pre takéto jednoduché dopyty nemusíme písať selekt v kompletnom rozpoložení, ale stačí napísať:

```
var osoby = dc.mojaTabulkas;
```

Dôvod je ten, že `mojaTabulka` je celá tabuľka vo forme objektu a teda je možné priamo pristupovať k objektu bez použitia selektu, pretože sa v podstate jedná o ten istý dopyt.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Threading.Tasks;
using DatabaseFirst.Data;

namespace DatabaseFirst
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var dc = new mojaDBEntities())
            {
                var osoby = dc.mojaTabulkas;

                foreach (var osoba in osoby)
                {
                    Console.WriteLine(osoba.Name + "␣" + osoba.Age);
                }

                Console.ReadKey();
            }
        }
    }
}
```

Výsledok bude rovnaký.

V prípade, že by sme chceli pridať novú osobu do tabuľky, napíšeme nasledujúci kód:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using DatabaseFirst.Data;

namespace DatabaseFirst
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var dc = new mojaDBEntities())
            {
                mojaTabulka osoba = new mojaTabulka();
                osoba.Name = "Peter";
                osoba.Age = 17;

                dc.mojaTabulkas.Add(osoba);
                dc.SaveChanges();
```

```
                Console.ReadKey();
            }
        }
    }
}
```

Teda, vytvoríme si inštanciu tabuľky, ktorá obsahuje všetky osoby a priradíme nové atribúty danému objektu. Tento objekt pridáme do dátového kontextu, čím sa zaradí objekt do nejakej fronty a nad týmto kontextom zavoláme funkciu `SaveChanges()`, ktorá vykoná dané zmeny. V tabuľke následne uvidíme nový záznam.

	Id	Name	Age
▶	0	Peter	17
	1	Fero	28
	2	Vlado	47
	3	Jozef	35
	100	Stano	59
	101	Juraj	24
	102	Janko	42
*	NULL	NULL	NULL

Takže takýmto spôsobom sme schopní realizovať operácie nad objektovým modelom v Entity Frameworku prístupom návrhu databázovou metódou.

4.2 Kódový prístup (Code First)

Kódový prístup je v zásade akýsi pravý prístup k tvorbe návrhu, pretože poskytuje do veľkej miery plnú kontrolu nad celým procesom vývoja. Ono, všeobecne platí, že čím menší výskyt automatizovaných funckií, tým väčšiu kontrolu získame nad celkovým prístupom. Okrem toho je tento prístup v podstate intuitívnejší, pretože z daných programových tried nám Entity Framework autoamaticky vytvorí tabuľky v databáze, takže sme separovaný od zásahu do databázy.

Na začiatku je potrebné nainštalovať explicitne Entity Framework prostredníctvom *NuGet* priamo vo Visual Studio alebo nejakom inom vývojom prostredí, pričom je možné použiť aj príkaz z konzoly: `install-package EntityFramework`.

Takže začneme s prázdnym konzolovým projektom, kde použijeme rovnakú databázu, avšak nebudeme vytvárať tabuľky v tejto databáze, ale vytvoríme adresár `"Data"`. V tomto adresári vytvoríme obyčajnú triedu s niekoľkými property položkami.

Poznámka: Ak nie je uvedené inak, názvy tried, atribútov a podobne by sa mali písať v anglickom jazyku, ale na tomto mieste použijeme kvôli prehľadnosti a viditeľnosti vlastného kódu od automaticky generovaných názvov názvy v slovenskom jazyku.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CodeFirst.Data
{
    public class Osoba
    {
        public int OsobaId { get; set; }
        public string Meno { get; set; }
        public int Vek { get; set; }
    }
}
```

Z toho budeme požadovať vygenerovať tabuľku v databáze. Za týmto účelom bude potrebné vytvoriť dátový kontext, tzn. novú triedu, ktorá bude dediť od *DbContext* a ktorej obsahom bude property položka s názvom našej pôvodnej triedy.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Data.Entity;

namespace CodeFirst.Data
{
    class MojContext : DbContext
    {
        public DbSet<Osoba> Osoby { get; set; }
```

```
    }
}
```

Vzhľadom k tomu, že naša tabuľka bude prázdna, tak do takto vytvorenej tabľky pridáme nejaké dáta spôsobom veľmi podobným z predchádzajúcej časti ohľadne databázového prístupu.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using CodeFirst.Data;

namespace CodeFirst
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var dc = new MojContext())
            {
                Osoba osoba = new Osoba();
                osoba.Meno = "Zuzka";
                osoba.Vek = 20;

                dc.Osoby.Add(osoba);
                dc.SaveChanges();

                Console.ReadKey();
            }
        }
    }
}
```

V tomto okamihu, keď to spustíme, tak zistíme, že nám buď vytvorí nejakú novú databázu s názvom dátového kontextu a tabuľky alebo nám nevytvorí žiadnu databázu a tabuľku. Z toho dôvodu je vždy vhodné nespoliehať sa na Entity Framework, ale dodatočne explicitne uviesť v pripojovacom reťazci o akú databázu sa jedná. Takže v súbore `App.config` pridáme pripojovací reťazec:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlin
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSect
  </configSections>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.2" />
  </startup>
```

```
  <connectionStrings>
    <add name="MojPripojovaciRetazec" providerName ="System.Data.SqlClient" connectionString="Server
  </connectionStrings>
  <entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlConnectionFactory, EntityFr
    <providers>
      <provider invariantName="System.Data.SqlClient" type="System.Data.Entity.SqlServer.SqlProvider
    </providers>
  </entityFramework>
</configuration>
```

Týmto presne špecifikujeme dátový zdroj na ktorom zamýšľame vytvoriť dané tabuľky. V programovom kóde bude potrebné vytvoriť konštruktor dátového kontextu a zavolať funkciu v základnej triede pre nastavenie pripojovacieho reťazca:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Data.Entity;

namespace CodeFirst.Data
{
    class MojContext : DbContext
    {
        public MojContext() : base (nameOrConnectionString: "MojPripojovaciRetazec")
        public DbSet<Osoba> Osoby { get; set; }
    }
}
```

V takto nastavenom pripojovacom reťazci spustíme aplikáciu a v `Server Explorer` uvidíme novú tabuľku.

Ako sme si mohli všimnúť pri definovaní tabuľky, nešpecifikovali sme bližšie o aké atribúty tabuľky sa jedná. To znamená, že Entity Framework automaticky pridelí veľkosť daného atribútu na maximum, prípade na inak prednastavenú hodnotu.

Ak chceme špecifikovať vlastné atribúty pre tento účel je potrebné definovať tzv. *Data Annotation Attributes* s požadovanými hodnotami:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CodeFirst.Data
{
    public class Osoba
```

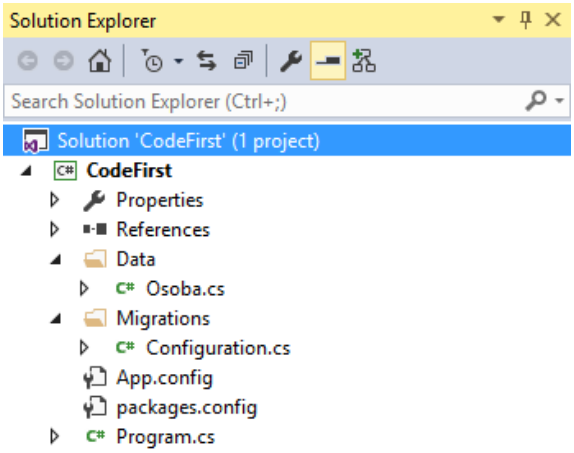
```
    {
        [Key]
        public int OsobaId { get; set; }
        [Required, StringLength(50)]
        public string Meno { get; set; }
        [MaxLength(100)]
        public int Vek { get; set; }
    }
}
```

Primárne kľúče nie je potrebné definovať, ale kvôli prehľadnosti je to vhodný postup. Ak danú aplikáciu v tomto okamihu spustíme, tak obdržíme chybové hlásenie, vyvolanie výnimky. To znamená, že model, ktorý vyžaduje naša aplikácia, nezodpovedá údajom, ktoré sa nachádzajú v už existujúcej databáze. Jedno z riešení by bolo také, že by sme odstránili existujúcu databázu a spustili aplikáciu ešte raz, čím by sme vygenerovali novú tabuľku so zmenenými atribútmi. Ovšem v praxi nie je prípustné kvôli jednoduchým zmenám v modely odstraňovať kompletnú databázu, resp. niektoré tabuľky. A toto je práve záležitosť, ktorá **nie je automatická** oproti databázovému prístupu, kde akákoľvek zmena môžme jednoducho synchronizovať s pôvodnou databázou. Takže tento medzikrok je potrebné explicitne špecifikovať tzv. **verzovaním databázovej štruktúry** (**Code First Migrations**).

Code First Migrations poskytuje správu verzií databázy z programového kódu. Samotné ovládanie funguje pomocou príkazového riadku v *Package Manager Console*. Každú zmenu, ktorú vykonáme pošleme na preskúmanie, spustí sa porovnanie aktuálneho a nového stavu databázy a v prípade nejakých zmien sa vygeneruje nová trieda s metódami **Up()** a **Down()**. Spustením metódy `Up()` migrujeme na nejakú novú tabuľku a naopak spustením metódy `Down()` danú tabuľku odstránime. Zároveň na každej novej vytvorenej tabuľke existuje metóda `Seed()`, ktorá sa spúšťa automaticky a ktorá prednastavuje hodnoty danej tabuľky.

Takže v *Package Manager Console* napíšeme príkaz `enable-migrations`.

V `Solution Explorer` vznikne nový adresár `Migrations`:



Súbor `Configuration.cs` obsahuje metódu `Seed()` a konštruktor s prednastavenou hodnotou `AutomaticMigrationsEnabled = false;`. Aj napriek tomu, že by nám to mohlo generovať automaticky, zmena tejto hodnoty na `true` sa neodporúča.

```
namespace CodeFirst.Migrations
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Migrations;
    using System.Linq;

    internal sealed class Configuration : DbMigrationsConfiguration<CodeFirst.Data.Mo
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = false;
            ContextKey = "CodeFirst.Data.MojContext";
        }

        protected override void Seed(CodeFirst.Data.MojContext context)
        {
            //  This method will be called after migrating to the latest version.

            //  You can use the DbSet<T>.AddOrUpdate() helper extension method
            //  to avoid creating duplicate seed data. E.g.
            //
            //    context.People.AddOrUpdate(
            //      p => p.FullName,
            //      new Person { FullName = "Andrew Peters" },
            //      new Person { FullName = "Brice Lambson" },
            //      new Person { FullName = "Rowan Miller" }
            //    );
            //
        }
    }
}
```

Ďalší súbor `201709171249066_InitialCreate.cs` obsahuje práve spomínané metódy `Up()` a `Down()`, pričom metóda `Up()` aplikuje uvedené zmeny a metóda `Down()` danú tabuľku odstráni.

```
namespace CodeFirst.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class InitialCreate : DbMigration
    {
        public override void Up()
        {
            CreateTable(
                "dbo.Osobas",
                c => new
```

```
                    {
                        OsobaId = c.Int(nullable: false, identity: true),
                        Meno = c.String(),
                        Vek = c.Int(nullable: false),
                    })
                .PrimaryKey(t => t.OsobaId);

        }

        public override void Down()
        {
            DropTable("dbo.Osobas");
        }
    }
}
```

V tomto okamihu spustíme v konzole príkaz `add-migration zmena_osoba`.

Týmto príkazom nám vytvorilo nový súbor v adresári `Migrations`, konkrétne `201709171455069_zmena_osoba.cs` s nasledovným obsahom:

```
namespace CodeFirst.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class zmena_osoba : DbMigration
    {
        public override void Up()
        {
            AlterColumn("dbo.Osobas", "Meno", c => c.String(nullable: false, maxLength
        }

        public override void Down()
        {
            AlterColumn("dbo.Osobas", "Meno", c => c.String());
        }
    }
}
```

Z toho je zrejmé, že pri v metóde `Up()` sa zmení v pôvodnej tabuľke špecifikácia stĺpca a naopak v metóde `Down()` sa tieto zmeny vrátia do pôvodného stavu. Dané metódy **je možné** manuálne upravovať, rovanko ako aj metódu `Seed()`. Následne je potrebné spustiť príkaz `update-database -verbose` (*zoznam ostatných prepínačov*).

Aktualizovali sme stav databázy, kde je možné vidieť konkrétny SQL príkaz a nejakú internú serializovanú informáciu, ktorá mapuje danú zmenu. Následne, ak sa pozrieme do databázovej štruktúry, tak bude možné vidieť vykonanú zmenu. Atribút `Meno` obsahuje maximálnu hodnotu `50` znakov a daná položka je požadovateľná, teda `not null` namiesto pôvodnej hodnoty `null`.

V prípade, že by sme z nejakých dôvodov potrebovali zmigrovať na nižšiu verziu, tak by sme napísali do konzoly `update-database -targetmigration initialcreate`. Týmto sa databázová štrukúra migruje do pôvodného stavu.

Takže takýmto spôsobom funguje *Code First Migrations* a kódový prístup návrhu databázy. Ako bolo možné vidieť kódový prístup vyžaduje viac úsilia a rozhodovania v procese návrhu oproti databázovému prístupu. Všeobecne sa odporúča používať kódový prístup, okrem iného z toho dôvodu, ako už bolo spomínané, že akékoľvek automaticky generované postupy zneprehľadňujú situáciu navrhovania a samotný súbor s príponou `*.edmx` môže spôsobovať pri častejších zmenách databázovej štruktúry nezrovnalosti medzi modelom a databázou. Napriek tomu je možné všetky spôsoby do určitej miery kombinovať a navyše v určitých špecifických prípadoch použiť aj klasický relačný spôsob dopytovania. Ostatné náležitosti ohľadne manipulácie dát platia pre všetky spôsoby prístupov.

5 NEVHODNOSŤ POUŽITIA ENTITY FRAMEWORK

Entity Framework disponuje veľkými výhodami oproti klasickému relačnému prístupu tvorby databáz. Mohlo by sa zdať, že jedinou prekážkou je výkonnosť Entity Frameworku, ale pri mnohých výkonnostných porovnaniach klasického prístupu a objektového prístupu tvorby databáz sú napokon rozdiely takmer zanedbateľné a navyše budúcnosť Entity Frameworku má tendenciu tieto nedostatky zlepšovať. Avšak v niektorých prípadoch je použitie Entity Frameworku nevhodné z hľadiska štruktúry budúcej alebo existujúcej databázy a procesov, ktoré sú spojené s manipuláciou databázy.

V týchto prípadoch nie je vhodné používať Entity Framework:

- Databáza obsahuje nejaké špecifickosti (používateľské definované typy/funkcie, *Sparse Columns*, fulltextové vyhľadávanie, *SQL CLR*)
- Zvláštne uložené procedúry (Entity Framework predpokladá relačnú databázu ako primárny zdroj dát nad ktoru je možné prehľadne použiť *CRUD* logiku, teda nepripúšťajú sa nejaké skryté objekty, resp. odkazované a podobne)
- Databázová štruktúra je príliš dynamická (Entity Framework predpokladá do istej miery statickú štrukúru)
- Hromadné príkazy Update/Delete
- Výkon databázy

Napokon to, že na daných miestach nie je vhodné použiť Entity Framework neznamená, že nie je možné ho použiť vôbec. Teda, vždy je nejaká možnosť kombinácie klasického a objektového prístupu, resp. použiť nejaké osvedčené postupy, ktorým dané kritické miesto prekryjete, ale bude to už akési hybridné riešenie s možnosťou výskytu potencionálnych chýb. V každom prípade sú objektové databázy v súčinnosti s moderným objektovo-orientovaným programovaním aj napriek niektorým obmedzeniam pravdepodobne vhodnou voľbou.