

Entity Framework

Relacione baze podataka

Relaciona baza podataka je poseban tip baze podataka kod kojeg se organizacija podataka zasniva na **relacionom modelu**. Podaci se u ovakvim bazama organizuju u skup relacija između kojih se definišu određene veze. Relacija se definiše kao skup n-torki sa istim atributima, definisanih nad istim domenima iz kojih mogu da uzimaju vrednosti. U relacionim bazama podataka, svaka relacija mora da ima definisan **primarni ključ**, koji predstavlja atribut pomoću kojeg se jedinstveno identifikuje svaka n-torka. Relacija opciono može da poseduje i **strani ključ**, preko kojeg ostvaruje vezu sa drugim relacijama. Svaka relacija može se predstaviti u tabelarnom obliku, pa se ti termini često koriste kao sinonimi (iako strogo gledano to nisu). Naredna slika prikazuje jednostavnu bazu podataka sa dve relacije: knjige i pisci.

Books

Id	Title	Year	Rating	AuthorId
1	Za kim zvona zvone	1940	3.97	1
2	Plodovi gneva	1939	3.96	2
3	Istočno od raja	1952	4.37	2
4	2666	2004	4.21	3
5	Divlji detektivi	1998	4.12	3
6	Kvaka 22	1961	3.98	4
7	Hari Poter 1-7	2007	4.74	5
8	Autostoperski vodič kroz galaksiju	1996	4.37	6
9	Travnička hronika	1945	4.26	7
10	Znakovi pored puta	1976	4.47	7

Authors

Id	Name	Gender	Born	Died
1	Ernest Hemingway	Male	21/07/1899	02/07/1961
2	John Steinbeck	Male	27/02/1902	20/12/1968
3	Roberto Bolano	Male	28/04/1953	15/07/2003
4	Joseph Heller	Male	01/05/1923	12/12/1999
5	J.K. Rowling	Female	31/07/1965	
6	Douglas Adams	Male	11/03/1952	11/05/2001
7	Ivo Andric	Male	09/10/1892	13/03/1975

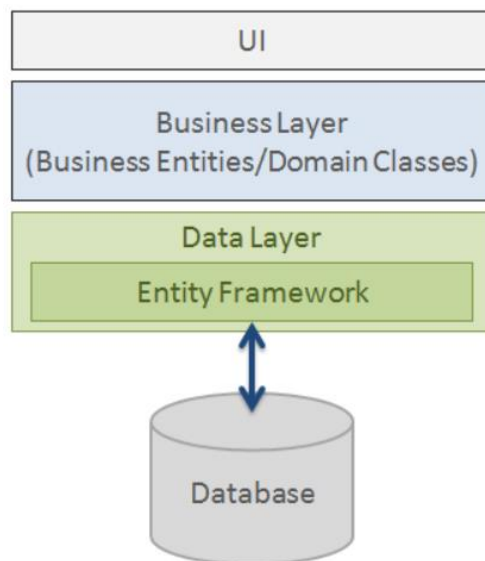
Primarni ključevi ovih tabela su atributi Id, jer su oni jedinstveni za svaku n-torku relacija. Obično je praksa da se za primarni ključ kreira novi, numerički atribut koji će biti jedinstven za svaku n-torku date relacije (nije pravilo, može se koristiti i neki postojeći atribut ukoliko je „prirodno“ da bude ključ – npr. JMBG, bar-kod i slično). Tabela Books sadrži i strani ključ kojim je povezana sa tabelom Authors – AuthorId. Strani ključ je, kao što mu i ime kaže, atribut neke relacije dobijen „prenošenjem“ primarnog ključa iz druge relacije. U ovom slučaju, strani ključ AuthorId je posledica 1-N veze (jedan na više) između knjiga i autora: svaku knjigu je napisao jedan autor, dok autor može da napiše više knjiga. Bitno ograničenje jeste da je domen vrednosti atributa AuthorId iz relacije Books podskup domena atributa Id iz relacije Authors. Drugim rečima, atribut

AuthorId može da sadrži samo vrednosti koje se već pojavljuju u koloni Id tabele Authors. Napomena: ovo je ograničenje na nivou baze, nije nešto što vi implementirate.

Upravljanje relacionim bazama podataka realizuje se preko sistema za upravljanje bazama podataka (SUBP). Neki od najpopularnijih SUBP-ova su: Microsoft SQL Server, Oracle Database, MySQL, itd.

Entity Framework

Entity Framework je *open-source ORM (Object-relational mapping) framework* za .NET aplikacije. *Entity Framework* omogućava rad sa podacima korišćenjem objekata određenih klasa umesto fokusiranja na detalje baze podataka i tabela u kojima su ti podaci skladišteni. Ovim je eliminisana potreba za pisanjem koda zaduženog za sam pristup podacima. Sledeća slika ilustruje poziciju *Entity Framework*-a u tipičnoj aplikaciji. Može se primetiti da se *Entity Framework* nalazi između baze podataka i biznis logike (klase sistema), sa kojima je u dvosmernoj „komunikaciji“: čuva podatke skladištene u svojstvima (*properties*) klasa i dobavlja podatke iz baze i „prepakuje“ ih u objekte odgovarajućih klasa.



Dva suštinski različita pristupa u korišćenju *Entity Framework*-a i projektovanju biznis logike (klasa) neke aplikacije su:

- *Code First* - implementacija klasa pa potom njihovo prevođenje u tabele baze podataka,
- *Database First* – projektovanje baze podataka pa potom prevođenje u klase.

Entity Framework podržava oba navedena načina, koji su detaljnije objašnjeni u nastavku. Da biste mogli da odradite sve što je prikazano u nastavku neophodno je da imate *Visual Studio > 2010*. Ko koristi VS 2010 mora da instalira i *NuGet*.

Code First pristup

Code First pristup omogućava realizaciju baze podataka kroz implementaciju klasa. Klase koje se implementiraju su gotovo identične „normalnim“ klasama, s tim da je potrebno izvršiti manje

izmene da bismo maksimalno iskoristili *Entity Framework*. U nastavku je dat izgled klasa (samo svojstva) **Book** i **Author**.

```
public class Author
{
    [Key]
    public int Id { get; set; }
    public string Name { get; set; }
    public Gender Gender { get; set; }
    public DateTime Born { get; set; }
    public DateTime? Died { get; set; }

    public virtual List<Book> Books { get; set; }
}

public class Book
{
    [Key]
    public int Id { get; set; }
    public string Title { get; set; }
    public int Year { get; set; }
    public double Rating { get; set; }

    public int AuthorId { get; set; }
    public virtual Author Author { get; set; }
}
```

Prva izmena u odnosu na do sada implementirane klase je upotreba anotacija. Anotacije nisu isključivo vezane za *Entity Framework* i postoji znatno veći broj različitih anotacija i njihovih upotreba u .NET-u. Ovde je korišćena jedino *Key* anotacija, koja govori *Entity Framework*-u da od svojstva *Id* želimo da napravimo primarni ključ (za obe tabele, naravno ne mora da se zove *Id*). Bitno je napomenuti da postoji i anotacija *ForeignKey*, koja se koristi da se naglasi da je svojstvo strani ključ, dobijen prenošenjem primarnog ključa iz neke druge tabele. U prethodnom primeru anotacija *ForeignKey* nije korišćena jer je svojstvo koje predstavlja strani ključ nazvano u adekvatnom formatu (*AuthorId*). U slučaju da želimo drugačije da nazovemo strani ključ, neophodno je korišćenje *ForeignKey* anotacije u kojoj će se navesti ime tabele iz koje se prenosi ključ, iznad svojstva koje predstavlja strani ključ.

Druga bitna izmena jeste upotreba virtuelnih svojstava prilikom kompozicije (*virtual Author* i *virtual List<Book>*). Ovim se omogućava jedna izuzetno korisna funkcionalnost *Entity Framework*-a nazvana *Lazy Loading*. Uloga *Lazy Loading*-a je automatsko učitavanje određenih podataka iz baze tek u trenutku prvog pristupa svojstvima. U ovom slučaju to znači da će se lista knjiga za svakog autora popuniti automatski iz tabele knjiga tek prilikom prvom pristupu *Books* svojstvu. Ovim se takođe smanjuje potreba za eksplicitnim spajanjem tabela (*join*), jer je moguće automatsko učitavanje (i dalje je ponekad neophodno).

Entity Framework prevodi klase *Author* i *Book* u adekvatan SQL kod, pomoću kog se kreiraju dve tabele u bazi. U nastavku je dat izgled SQL koda kojim su kreirane tabele *Authors* i *Books*.

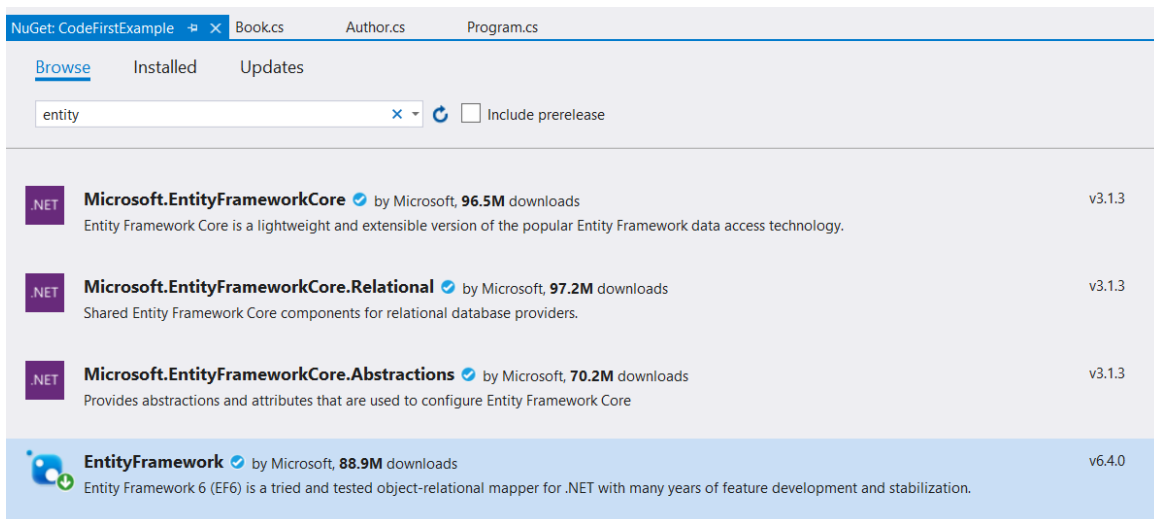
```

CREATE TABLE [dbo].[Authors] (
    [Id] INT IDENTITY (1, 1) NOT NULL,
    [Name] NVARCHAR (MAX) NULL,
    [Gender] INT NOT NULL,
    [Born] DATETIME NOT NULL,
    [Died] DATETIME NULL,
    CONSTRAINT [PK_dbo.Authors] PRIMARY KEY CLUSTERED ([Id] ASC)
);

CREATE TABLE [dbo].[Books] (
    [Id] INT IDENTITY (1, 1) NOT NULL,
    [Title] NVARCHAR (MAX) NULL,
    [Year] INT NOT NULL,
    [Rating] FLOAT (53) NOT NULL,
    [AuthorId] INT NOT NULL,
    CONSTRAINT [PK_dbo.Books] PRIMARY KEY CLUSTERED ([Id] ASC),
    CONSTRAINT [FK_dbo.Books_dbo.Authors_AuthorId] FOREIGN KEY ([AuthorId]) REFERENCES
[dbo].[Authors] ([Id]) ON DELETE CASCADE
);

```

Da bi *Entity Framework* mogao da mapira klase na tabele, neophodno je implementirati *Context* klasu. *Context* klasa mora da nasledi *DbContext* klasu iz *System.Data.Entity namespace*-a, i ona reprezentuje sesiju sa bazom podataka, tj. omogućava dobavljanje i čuvanje podataka u bazu. Pre implementacije (da biste uopšte mogli da koristite navedeni *namespace*), neophodno je instalirati *Entity Framework*. To možete da uradite na sledeći način: desni klik na projekat u *Solution Explorer*-u, iz padajućeg menija izaberete opciju *Manage NuGet Packages*, potom u kartici *Browse* ukucate *Entity Framework* i instalirate ga. Sledeća slika prikazuje izgled prozora prilikom instalacije *Entity Framework*-a.



Implementacija *Context* klase je veoma jednostavna: potrebno je napraviti po jedan *DbSet<>* za svaku od klasa modela. Preko ove klase se pristupa tabelama baze podataka, koje će biti kreirane za svaki *DbContext* klase. Sam pristup podacima iz tabela ne razlikuje se od rada sa bilo kojom

kolekcijom, npr. listom (*Add*, *Remove* i sve ostale metode, LINQ upiti, itd.). Sledeća slika prikazuje izgled *Context* klase za primer sa knjigama i autorima.

```
public class LibraryContext : DbContext
{
    public DbSet<Author> Authors { get; set; }
    public DbSet<Book> Books { get; set; }
}
```

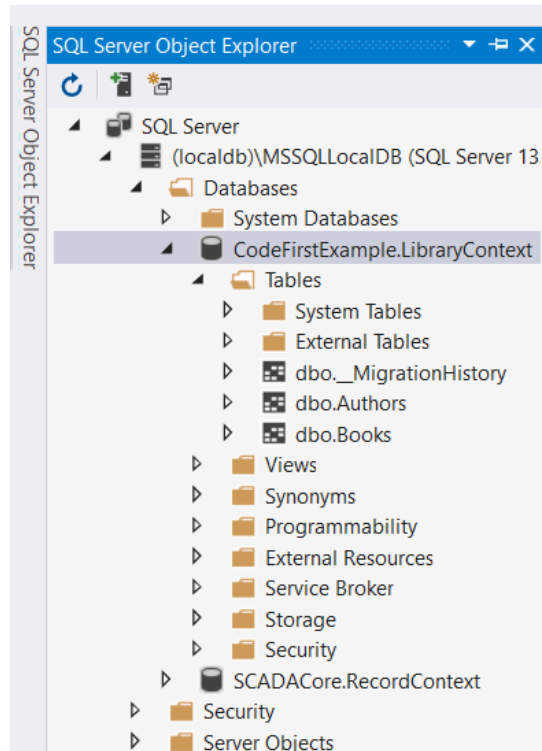
Ovim je urađeno sve što je potrebno da bismo mogli da vršimo CRUD (*Create Read Update Delete*) operacije nad bazom. Pošto su u ovom primeru implementirane dve klase modela, u *Context* klasi dodata su dva *DbSet* svojstva, za svaku od klasa. Time će se u bazi *CodeFirstExaple.LibraryContext* (ovo je default za davanje imena *Namespace.ContextKlasa*) kreirati dve tabele: *Authors* i *Books*, sa kolonama ekvivalentnim svojstvima tih klasa. Naredna slika prikazuje rad sa *Context* klasom u *Main*-u. U primeru je urađeno samo dodavanje dva nova autora u bazu.

```
using (var db = new LibraryContext())
{
    Console.WriteLine("Adding some authors into database...");

    Author author1 = new Author("Roberto Bolano", Gender.MALE, new DateTime(1953, 4,
28), new DateTime(2003, 7, 15));
    Author author2 = new Author("J.K. Rowling", Gender.FEMALE, new DateTime(1965, 7,
31), null);

    db.Authors.Add(author1);
    db.Authors.Add(author2);
    db.SaveChanges();
}
```

Using blok se koristi da bi se automatski zatvorila konekcija ka bazi (slično kao i kod čitanja/upisa u fajl). Nakon kreiranja objekta *LibraryContext*, jednostavno se pristupa odgovarajućim *DbSet*-ovima, preko kojih je moguće vršiti sve CRUD operacije kao da radimo sa bilo kojom drugom kolekcijom. Nakon određenih promena u okviru *DbSet*-ova, neophodno je pozvati metodu *SaveChanges* nad objektom *Context* klase, da bi se sve promene sačuvale i u bazi! Nakon prvog izvršavanja ovog koda kreirana je baza podataka. Da biste videli bazu podataka potrebno je da uradite sledeće: *View -> SQL Server Object Explorer*, gde se u okviru neke od lokalnih baza nalazi i ova upravo kreirana baza (verovatno *MSSQLLocalDB*). Sledeća slika prikazuje lokaciju na kojoj ćete (najverovatnije) pronaći bazu.



U okviru ovog prozora možete videti tabele (*dbo.Authors*, *dbo.Books*), izvorni SQL kod za kreiranje tabela, sadržaj svake od tabela itd. Napomena: kada dodajete podatke u bazu i hoćete da ih vidite kroz ovaj prozor, neophodno je da uradite *Refresh* (plava kružna strelica u gornjem levom uglu ili desni klik pa *refresh*).

Modelovanje N-N (više na više) veze

U prethodnom primeru ilustrovano je modelovanje 1-N veze: knjigu je napisao tačno jedan autor, a jedan autor može da napiše više knjiga. Ukoliko je potrebno modelovati N-N vezu, neophodno je uvođenje povezne tabele, tj. u slučaju Code First pristupa, pomoćne klase. U nastavku je dat izgled klase kojima je ilustrovan primer modelovanja N-N veze na jednostavnom primeru radnika i projekata (radnik radi na više projekata, a na jednom projektu može da radi više radnika).

```
public class Worker
{
    [Key]
    public string Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public uint Age { get; set; }
    public DateTime BirthDay { get; set; }
    public double Salary { get; set; }
    public string PhoneNumber { get; set; }

    public virtual List<WorkerProject> WorkerProjects { get; set; }
}
```

```

public class Project
{
    [Key]
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime StartDate { get; set; }
    public DateTime? FinishDate { get; set; }

    [ForeignKey("Leader")]
    public string LeaderId { get; set; }
    public virtual Worker Leader { get; set; }

    public virtual List<WorkerProject> WorkerProjects { get; set; }
}

public class WorkerProject
{
    [Key, Column(Order = 1)]
    [ForeignKey("Worker")]
    public string WorkerId { get; set; }
    public virtual Worker Worker { get; set; }

    [Key, Column(Order = 2)]
    [ForeignKey("Project")]
    public int ProjectId { get; set; }
    public virtual Project Project { get; set; }

    public int WorkHours { get; set; }
}

```

U primeru iznad, radnik i projekat predstavljani su pomoću *Worker* i *Project* klasa. *WorkerProject* je povezna klasa, koja sadrži strane ključne klase koje povezuje (*Worker* i *Project*), kao i dodatno svojstvo koje predstavlja broj radnih sati datog radnika na odgovarajućem projektu. U datom primeru postoji i jedna 1-N veza, između klasa radnik i projekat: projekat sadrži svojstvo koje predstavlja vođu projekta (stran ključ *LeaderId*).

Database First pristup

Database First pristup je inverzan *Code First* pristupu: upotrebom *Entity Framework*-a vrši se mapiranje postojeće baze podataka na klase u C# kodu.

Upotreba *Entity Framework*-a za potrebe *Database First* pristupa podrazumeva više koraka: dodavanje konekcije ka željenoj bazi podataka i kreiranje projekta sa *ADO.NET Entity Data Model*-om, u okviru kog se upotrebom *Entity Framework Designer*-a kreira model iz postojeće baze podataka.

Na sledećoj slici prikazano je dodavanje konekcije ka postojećoj bazi podataka. Za otvaranje ovog menija potrebno je kliknuti na *View -> Server Explorer*, a potom u meniju koji se otvori desni klik na *Data Connections* i klik na opciju *Add Connection*. U okviru *Server Name* polja potrebno je uneti ime SQL servera, tj. verovatno *(localdb)\MSSQLLocalDB*.

The screenshot shows the 'Add Connection' dialog box. It has a title bar with a question mark and a close button. The main text says: 'Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.'

The 'Data source:' section has a dropdown menu showing 'Microsoft SQL Server (SqlClient)' and a 'Change...' button.

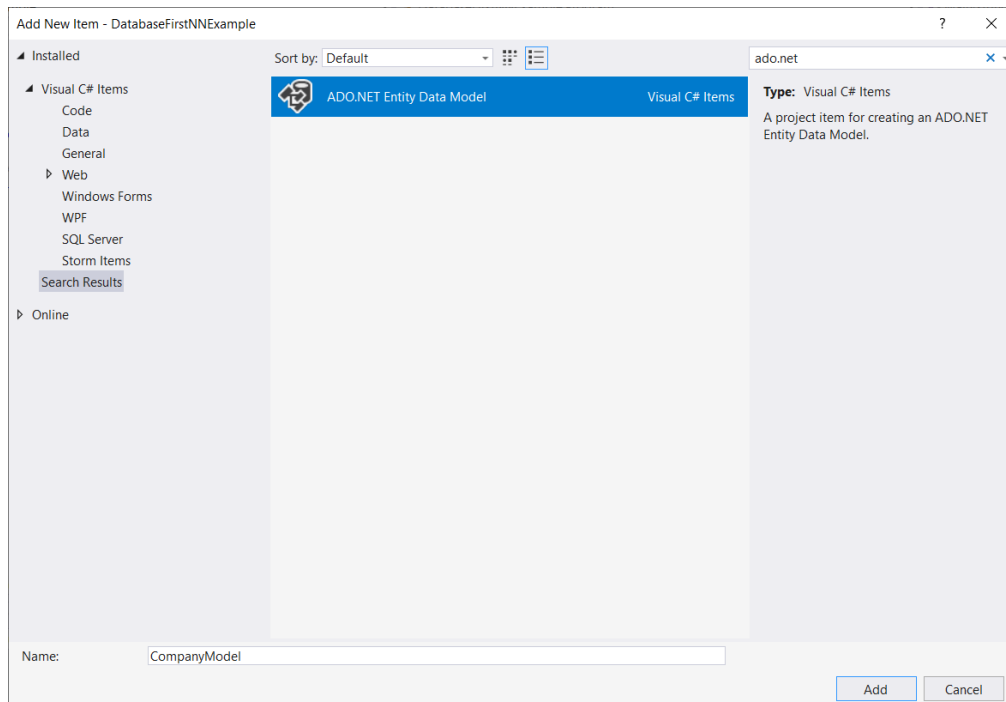
The 'Server name:' section has a dropdown menu showing '(localdb)\MSSQLLocalDB' and a 'Refresh' button.

The 'Log on to the server' section has a dropdown menu for 'Authentication' set to 'Windows Authentication'. Below it are fields for 'User name:' and 'Password:', and a checkbox for 'Save my password'.

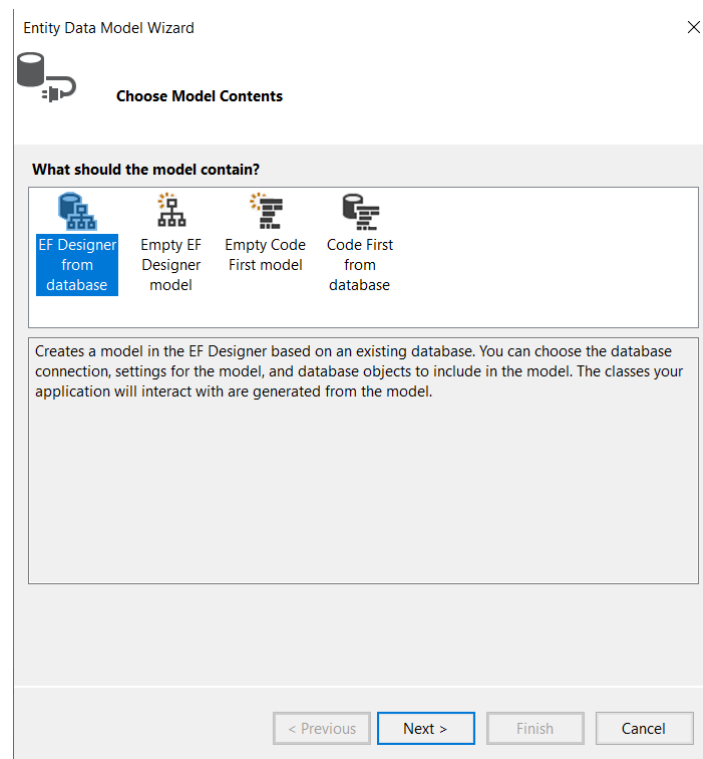
The 'Connect to a database' section has two radio buttons. The first, 'Select or enter a database name:', is selected and has a dropdown menu showing 'CodeFirstNNExample.CompanyContext'. The second, 'Attach a database file:', is unselected and has a 'Browse...' button. Below these are fields for 'Logical name:'.

At the bottom right is an 'Advanced...' button. At the bottom are three buttons: 'Test Connection' (highlighted with a blue border), 'OK', and 'Cancel'.

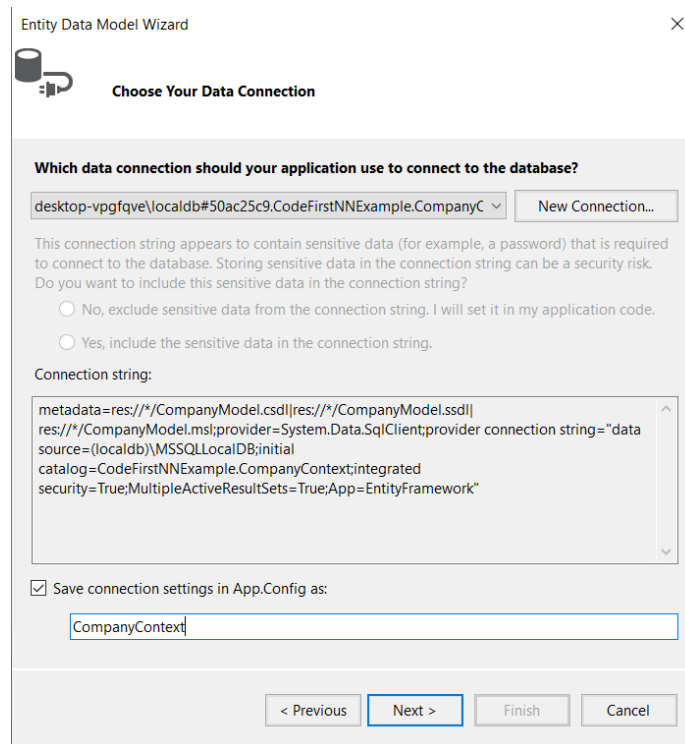
Nakon uspešnog dodavanja konekcije ka već postojećoj bazi podataka, potrebno je napraviti novi projekat (*Console Application*) i u okviru njega dodati *ADO.NET Entity Data Model*, što je prikazano na sledećoj slici.



Nakon dodavanja *Entity Data* modela, otvara se meni u okviru kog je potrebno izabrati sadržaj modela. U našem slučaju, to je *EF Designer from database*.

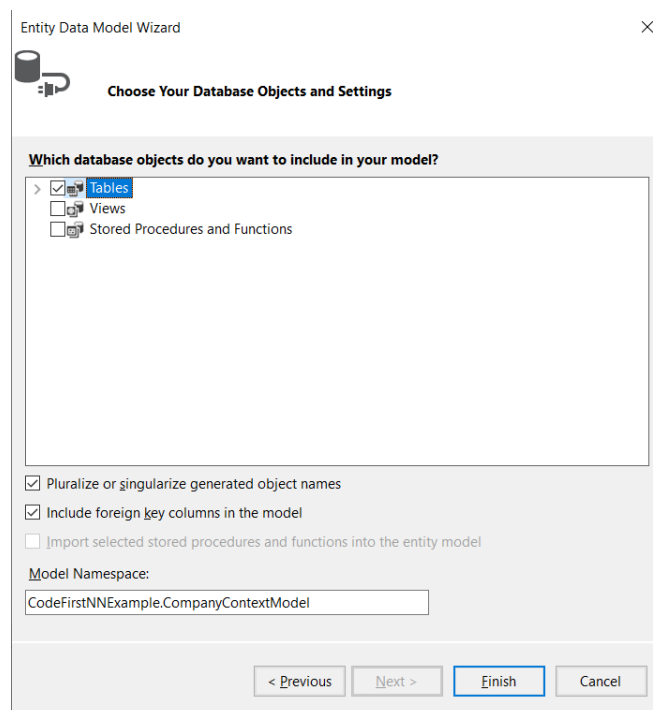


Klikom na *Next* dugme prelazi se na sledeći korak, u kom je potrebno izabrati prethodno kreiranu konekciju ka bazi i uneti naziv *connection string*-a.



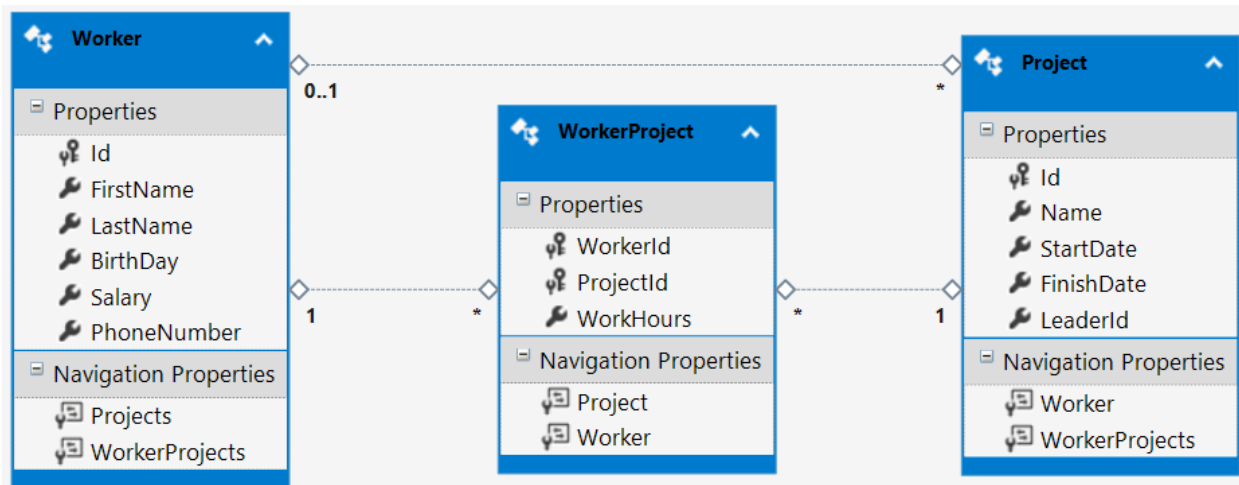
The screenshot shows the 'Entity Data Model Wizard' window, specifically the 'Choose Your Data Connection' step. The window title is 'Entity Data Model Wizard' with a close button (X) in the top right corner. Below the title bar is a header area with a database icon and the text 'Choose Your Data Connection'. The main content area is titled 'Which data connection should your application use to connect to the database?'. It features a dropdown menu showing 'desktop-vpgfqve\localdb#50ac25c9.CodeFirstNNExample.CompanyC' and a 'New Connection...' button. Below this, a warning message states: 'This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?'. There are two radio buttons: 'No, exclude sensitive data from the connection string. I will set it in my application code.' (selected) and 'Yes, include the sensitive data in the connection string.'. Below the radio buttons is a text box labeled 'Connection string:' containing the following text: 'metadata=res://*/CompanyModel.csdl|res://*/CompanyModel.ssdl|res://*/CompanyModel.msl;provider=System.Data.SqlClient;provider connection string="data source=(localdb)\MSSQLLocalDB;initial catalog=CodeFirstNNExample.CompanyContext;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework"'. Below the text box is a checkbox labeled 'Save connection settings in App.Config as:' which is checked. Below the checkbox is a text box containing 'CompanyContext'. At the bottom of the window are four buttons: '< Previous', 'Next >' (highlighted with a blue border), 'Finish', and 'Cancel'.

Sledeći korak je izbor objekata baze podataka koje uključujemo u model i davanje naziva *namespace*-u modela. Za potrebe ovog predmeta dovoljno je izabrati samo *Tables* opciju, pošto ne radimo sa pogledima, procedurama i funkcijama nad bazom.



The screenshot shows the 'Entity Data Model Wizard' window, specifically the 'Choose Your Database Objects and Settings' step. The window title is 'Entity Data Model Wizard' with a close button (X) in the top right corner. Below the title bar is a header area with a database icon and the text 'Choose Your Database Objects and Settings'. The main content area is titled 'Which database objects do you want to include in your model?'. It features a list of database objects with checkboxes: 'Tables' (checked), 'Views' (unchecked), and 'Stored Procedures and Functions' (unchecked). Below the list are three checkboxes: 'Pluralize or singularize generated object names' (checked), 'Include foreign key columns in the model' (checked), and 'Import selected stored procedures and functions into the entity model' (unchecked). Below these checkboxes is a text box labeled 'Model Namespace:' containing 'CodeFirstNNExample.CompanyContextModel'. At the bottom of the window are four buttons: '< Previous', 'Next >', 'Finish' (highlighted with a blue border), and 'Cancel'.

Pritiskom na *Finish* dugme izvršava se mapiranje tabela baze podataka na klase, koje se dodaju u izabrani *namespace* kreiranog projekta. Takođe, otvara se i dijagram kojim je grafički predstavljen relacioni model baze podataka, koji je moguće po potrebi menjati, uz ažuriranje modela.



Migracije

Migracije obezbeđuju skup alata koji omogućavaju:

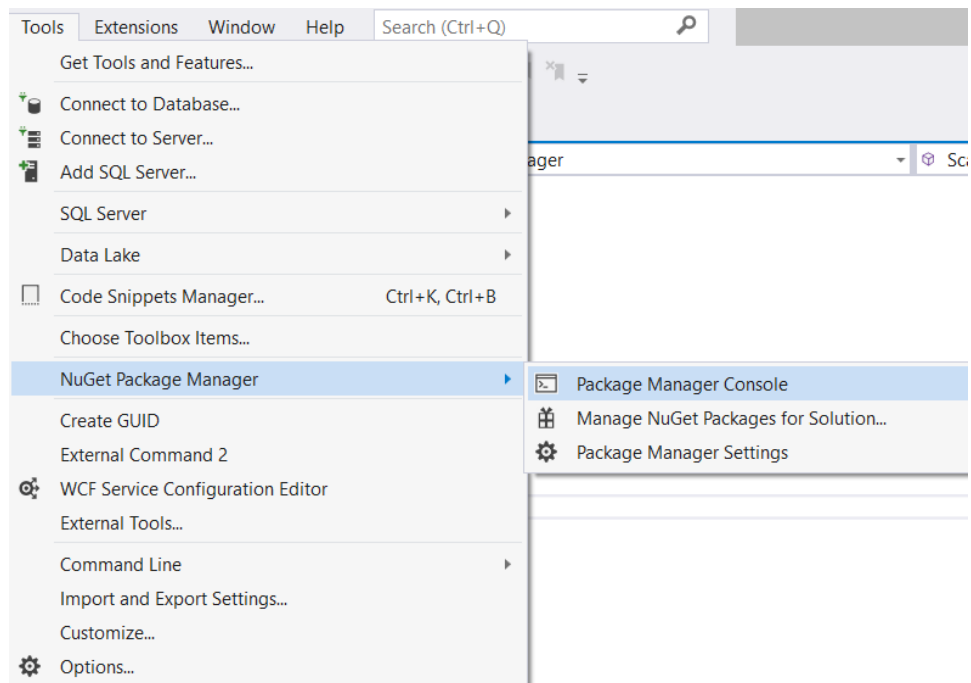
1. Kreiranje inicijalne baze podataka koja radi sa EF modelom
2. Generisanje migracija da biste pratili promene koje ste uneli u svoj EF model
3. Redovno ažuriranje šeme baze podataka u skladu sa tim promenama

U *Code First* pristupu, migracije se najčešće koriste kada je potrebno mapirati izmenu modela podataka na već kreiranu bazu podataka. Na primer, ukoliko želimo da dodamo svojstvo u klasu koja je već mapirana na tabelu u bazi podataka, prilikom prvog sledećeg pokretanja programa desiće se greška, zbog toga što se model podataka i šema baze podataka razlikuju. Rešenje je u korišćenju migracija, pomoću kojih se promene modela podataka mapiraju na promene šeme baze podataka.

Rad sa migracijama zahteva izvršavanje *enable-migrations* komande u *Package Manager Console*-i. Nakon izvršavanja navedene komande, u projektu će se pojaviti *Migrations* folder, koji će sadržati konfiguracionu klasu i sve migracije koje će se izvršiti tokom razvoja i održavanja projekta.

Generisanje i izvršavanje migracija vrši se u dva koraka:

1. Izvršavanje naredbe *add-migration ImeMigracije* u *Package Manager Console*-i.
2. *update-database* izvršava sve migracije na čekanju nad bazom podataka.



Da bi se otvorila *Package Manager* konzola, potrebno je u *Tools* padajućem meniju izabrati *NuGet Package Manager* i kliknuti na prvu ponuđenu stavku.

Validacija

Validacija u *EF Code First* uvek se javlja prilikom čuvanja entiteta, što je obično proces pokrenut **SaveChanges** metodom. Ovo je automatsko ponašanje koje se može sprečiti postavljanjem **ValidateOnSaveEnabled** svojstva objekta **Context** klase na **false** vrednost.

```
// disable validation upon saving
db.Configuration.ValidateOnSaveEnabled = false;
```

Provera da li su unosi koje kontekst trenutno prati validni može se izvršiti eksplicitnim pozivanjem **GetValidationErrors** metode.

```
// all validation errors
var allErrors = db.GetValidationErrors();

// validation errors for a given entity
var errorsInEntity = db.Entry(p).GetValidationResult();
```

Rezultat validacije sastoji se od kolekcije objekata klase **DbEntityValidationResult**, gde postoji po jedan objekat za svaki nevalidan entitet. Ova klasa sadrži sledeća svojstva:

Svojstvo	Namena
Entry	Entitet na koji se validacija odnosi.
IsValid	Indikator da li je entitet validan ili ne.
ValidationErrors	Kolekcija pojedinačnih greški.

ValidationErrors svojstvo sadrži kolekciju objekata klase **DbValidationError**, gde svaki objekat sadrži poruku o grešci (**ErrorMessage** svojstvo) i ime svojstva koje je nevalidno (**PropertyName** svojstvo, može biti prazno ukoliko je ceo entitet nevalidan).

Ukoliko pokušamo da sačuvamo entitet sa nevalidnim svojstvima, biće bačen **DbEntityValidationException** izuzetak, u okviru kog se nalazi **EntityValidationErrors** kolekcija sa svim **DbEntityValidationResult** objektima.

```
try
{
    // try to save all changes
    db.SaveChanges();
}
catch(DbEntityValidationException ex)
{
    // validation errors were found that prevented saving changes
    var errors = ex.EntityValidationErrors.ToList();
}
```

Validacioni atributi

Atribut (anotacije) smo do sada koristili za definisanje opcija mapiranja klasa na tabele u relacionoj bazi podataka. Na sličan način, atributi se mogu koristiti za definisanje validacionih pravila. Validacioni atribut mora da nasledi **ValidationAttribute** klasu iz **System.ComponentModel.DataAnnotations** namespace-a i da nadjača (*override*) neku od **IsValid** metoda. Postoji veliki broj jednostavnih validacionih atributa koje možemo odmah

koristiti i koji ni na koji način nisu vezani za Entity Framework. U nastavku je dat tabelarni pregled nekih od korisnih validacionih atributa.

Validacioni atribut	Namena
CompareAttribute	Poredi dva svojstva i ne prolazi ukoliko su različita.
CustomValidationAttribute	Izvršava custom validacionu metodu i vraća njenu povratnu vrednost.
MaxLengthAttribute	Proverava da li string svojstvo sadrži više od prosleđenog broja karaktera.
MinLengthAttribute	Proverava da li string svojstvo sadrži manje od prosleđenog broja karaktera.
RangeAttribute	Proverava da li je vrednost svojstva u prosleđenom opsegu.
RegularExpressionAttribute	Proverava da li string odgovara prosleđenom regularnom izrazu.
RequiredAttribute	Proverava da li svojstvo ima vrednost. Ako se radi o string svojstvu, proverava i da li je prazno.
StringLengthAttribute	Proverava da li je dužina stringa u prosleđenim granicama.
MembershipPasswordAttribute	Proverava da li string svojstvo (tipično lozinka) odgovara zahtevima podrazumevanog Membership Provider-a.

Svi atributi pišu se iznad svojstva/klase/metode koju validiraju, sa 0 ili više obaveznih parametara i listom opcionih imenovanih parametara. Na primer, atribut **StringLength** sadrži jedan obavezan parametar **MaximumLength**, uz nekoliko opcionih imenovanih parametara kao što su **MinimumLength** i **ErrorMessage**. **Required** je jedan od atributa koji nema obaveznih parametara, i veoma često se koristi da naglasi da se vrednost određenog svojstva mora definisati prilikom dodavanja objekta u bazu podataka. Naredni primer ilustruje primenu navedene dve anotacije za svojstvo **FirstName**.

```
[Required]
[StringLength(30, MinimumLength = 5, ErrorMessage = "Use 5-30 characters")]
public string FirstName { get; set; }
```

U nastavku je dat primer implementacije custom validacionog atributa, koji proverava da li je broj paran.

```

[AttributeUsage(AttributeTargets.Property, AllowMultiple = false, Inherited = true)]
public sealed class IsEvenAttribute : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        // check if the value is not null or empty
        if((value != null) && !String.IsNullOrEmpty(value.ToString()))
        {
            TypeConverter converter = TypeDescriptor.GetConverter(value);
            // check if the value can be converted to a long
            if(converter.CanConvertTo(typeof(Int64)))
            {
                Int64 number = (Int64)converter.ConvertTo(value,
                    typeof(Int64));

                // fail if the number is odd
                if(number % 2 != 0)
                {
                    return new ValidationResult(this.ErrorMessage,
                        new String[] { validationContext.MemberName });
                }
            }
            return ValidationResult.Success;
        }
    }
}

```

Implementirani validacioni atribut može da se primeni na bilo koje svojstvo čija vrednost se može konvertovati u long.

```

[IsEven(ErrorMessage = "Number must be even")]
public int Year { get; set; }

```

Drugi način za obezbeđivanje specifične validacione metode je primenom CustomValidationAttribute. U nastavku je prikazana implementacija iste validacije za parne brojeve na ovaj način. Prvo, potrebno je koristiti sledeću deklaraciju atributa.

```

[CustomValidation(typeof(CustomValidationRules), "IsEven", ErrorMessage = "Number must be even")]
public int Year { get; set; }

```

Nakon toga, potrebno je implementirati validaciono pravilo. U ovom primeru napravljena je statička metoda, što nije obavezno. U slučaju da validaciona metoda nije statička, neophodno je da klasa bude bezbedna za instanciranje (ne sme biti apstraktna, mora imati javan konstruktor bez parametara).

```

public class CustomValidationRules
{
    public static ValidationResult IsEven(object value,
        ValidationContext validationContext)
    {
        // check if the value is not null or empty
        if ((value != null) && !String.IsNullOrEmpty(value.ToString()))
        {
            TypeConverter converter = TypeDescriptor.GetConverter(value);
            // check if the value can be converted to a long
            if (converter.CanConvertTo(typeof(Int64)))
            {
                Int64 number = (Int64)converter.ConvertTo(value,
                    typeof(Int64));

                // fail if the number is odd
                if (number % 2 != 0)
                {
                    return new ValidationResult("Number must be even",
                        new String[] { validationContext.MemberName });
                }
            }
        }
        return ValidationResult.Success;
    }
}

```

Još jedna mogućnost za implementaciju specifične validacije jeste upotreba IValidatableObject interfejsa. Implementacijom ovog interfejsa klasa sadrži logiku za validiranje same sebe (*self-validatable*).

```

public class Book : IValidatableObject
{
    // other members go here
    public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
    {
        if (String.IsNullOrEmpty(Title))
        {
            yield return new ValidationResult("Title is required!");
        }
        if (Year % 2 != 0)
        {
            yield return new ValidationResult("Number must be even");
        }
        if (Rating < 1 || Rating > 5)
        {
            yield return new ValidationResult("Rating out of range");
        }
    }
}

```