

Razvoj primijenjene programske potpore

5. Rad s bazom podataka

Zajednički SQL server

- SQL Server: fantom.fer.hr,3000
- Ogledna baza podataka: Firma
 - SQL Server Authentication: rppp/zaporka (navedeno u uputama u repozitoriju na FER webu)
 - Moguće mijenjati podatke u svrhu testiranja
- Baze podataka oblika RPPPxx:
 - Baza podataka za grupu XX
 - Podaci za spajanje dostavljeni pojedinoj grupi

Server type:	Database Engine
Server name:	fantom.fer.hr,3000
Authentication:	SQL Server Authentication
Login:	rpppXX
Password:

Lokalna instalacija SQL Servera

- Nije nužna, ali može biti praktična (cca 1.6 GB)
 - Samostalna instalacija
 - Docker
 - https://hub.docker.com/_/microsoft-mssql-server

```
docker run -it -e "ACCEPT_EULA=Y" -e "SA_PASSWORD=nesto-  
ne.trivijalno" -p 1433:1433 --name sql-server-2019  
mcr.microsoft.com/mssql/server:2019-latest
```

Server type:

Database Engine

Server name:

localhost,1433

Authentication:

SQL Server Authentication

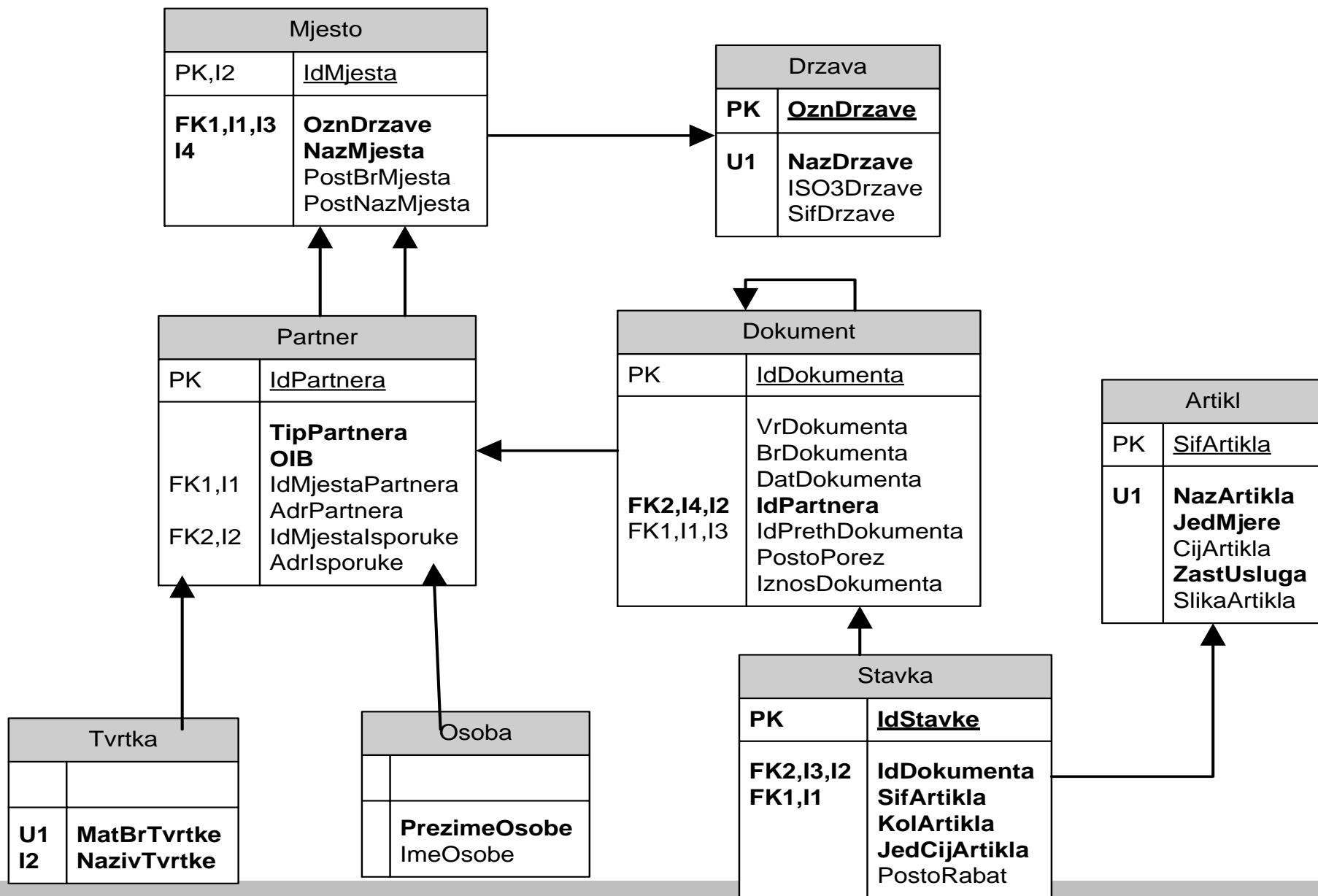
Login:

sa

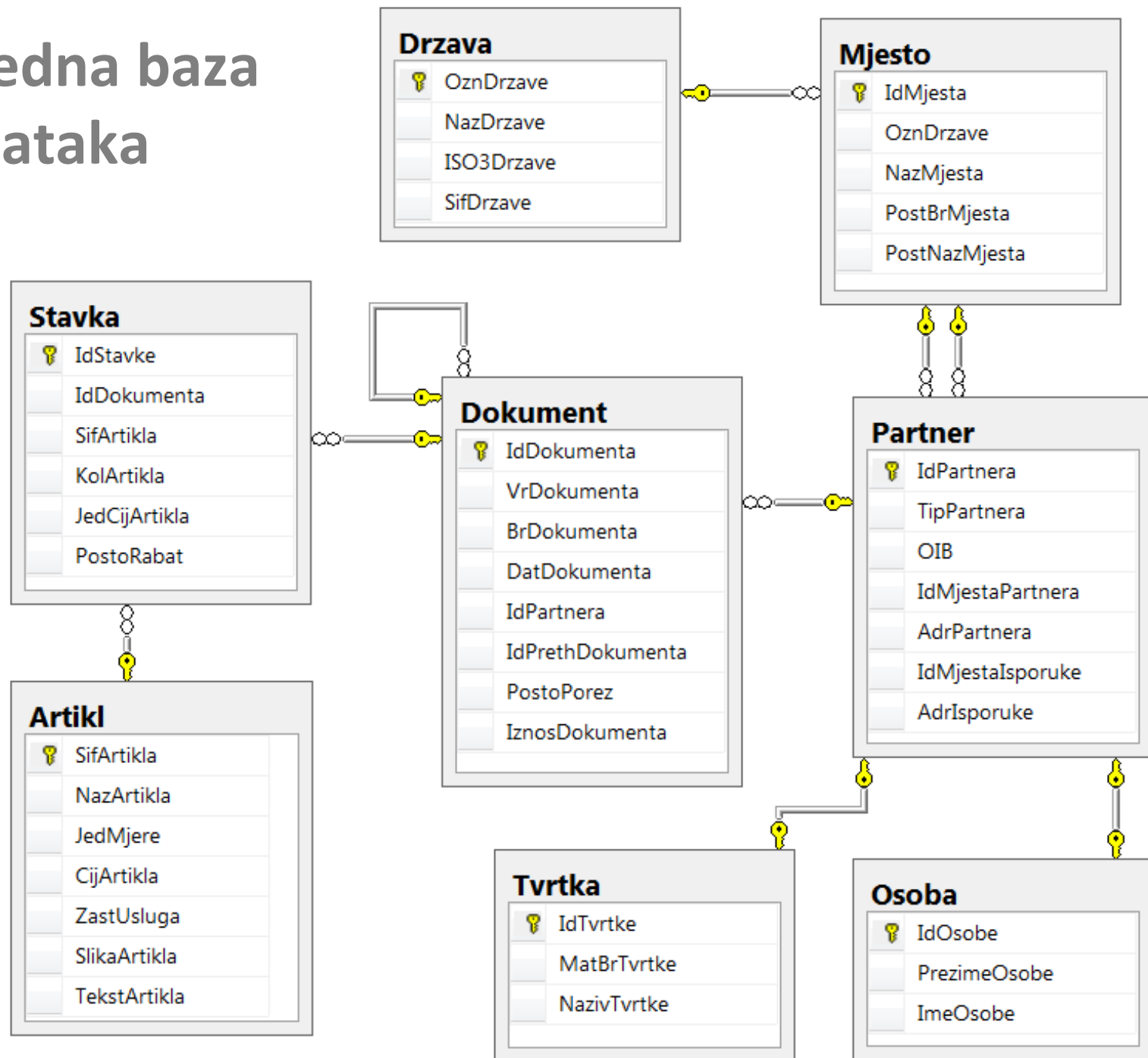
Password:

.....

Ogledna baza podataka (dijagram - MS Visio)

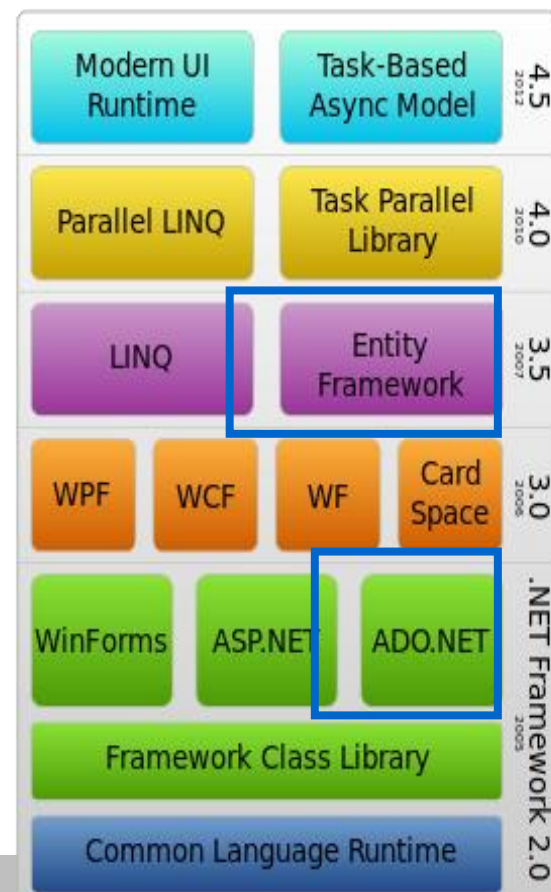


Ogledna baza podataka



.NET Framework i ADO.NET

- ActiveX Data Objects .NET (ADO.NET) je tehnologija za rukovanje podacima unutar .NET Frameworka koja omogućuje pristup bazama podataka, ali i drugim spremištima podataka, za koje postoji odgovarajući opskrbljivač podacima (provider)
 - sinonimi za opskrbljivač: davatelj, pružatelj, poslužitelj
- Podrška različitim tipovima spremišta
 - Strukturirani, nehijerarhijski podaci
 - Comma Separated Value (CSV) datoteke,
 - Microsoft Excel proračunske tablice, ...
 - Hijerarhijski podaci (npr. XML dokumenti)
 - Relacijske baze podataka
 - SQL Server, Oracle, MS Access, ...
- Entity Framework za objektno-relacijsko preslikavanje
 - Izvorno dio .NET-a, kasnije Open Source paket
 - U .NET Coreu razdvojeno u manje pakete



Opskrbljivači (davatelji) podataka

- Davatelji za različite tehnologije (SQL Server, PostgreSQL, SQLite, MongoDB, ...)
 - direktni pristup ili tehnologije s određenom razinom apstrakcije kao što su ORM, npr. Entity Framework Core
 - <https://docs.microsoft.com/hr-hr/ef/core/providers/?tabs=dotnet-core-cli>
 - <https://devblogs.microsoft.com/dotnet/net-core-data-access/>
- Microsoft.Data.SqlClient
 - optimiran za rad s MS SQL Server-om
 - razredi: *SqlCommand*, *SqlConnection*, *SqlDataReader*, ...
- Za ostale relacijske baze podataka razredi sličnih naziva
 - npr. *NpgsqlConnection*, *NpgsqlCommand*, *SQLiteConnection*, ...
- Navedeni razredi implementiraju zajednička sučelja pa imaju članove jednakih naziva
 - neovisnost aplikacije o fizičkom smještaju podataka

Osnovni pojmovi u pristupu bazi podataka

- `Connection`
 - Priključak (veza) s izvorom podataka
- `Command`
 - naredba nad izvorom podataka
 - izvršava se nad nekim otvorenim priključkom
- `DataReader`
 - Rezultat upita nad podacima
 - (forward-only, read-only connected result set)
- `ParameterCollection`
 - Parametri `Command` objekta
- `Parameter`
 - Parametar parametrizirane SQL naredbe ili pohranjene procedure
- `Transaction`
 - Nedjeljiva grupa naredbi nad podacima

Priključak na bazu podataka

- Priključak, veza (Connection)
 - otvara i zatvara vezu s fizičkim izvorom podataka
 - omogućuje transakcije i izvršavanje upita nad bazom podataka
- Sučelje *System.Data.IDbConnection*
i apstraktni razred *System.Data.DbConnection*
- Implementacije: *NpgsqlConnection*, *SqlConnection*, ...
- Važnija svojstva
 - *ConnectionString* – string koji se sastoji od parova postavki oblika naziv=vrijednost odvojenih točka-zarezom
 - *State* – oznaka stanja priključka (enumeracija *ConnectionState*)
 - *Broken*, *Closed*, *Connecting*, *Executing*, *Fetching*, *Open*
- Važniji postupci
 - *Open* – prikapčanje na izvor podataka
 - *Close* - otkapčanje s izvora podataka

Primjeri postavki priključka na bazu

- Microsoft SQL Server

- `Data Source=.;Initial Catalog=Firma;Integrated Security=True`
- `Data Source=rppp.fer.hr,3000;Initial Catalog=Firma;User Id=rppp;Password=šifra`

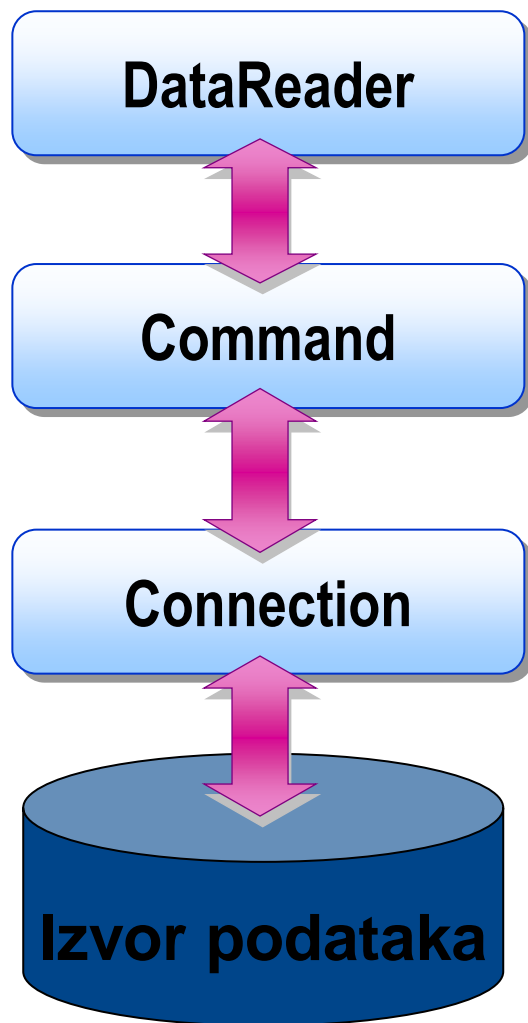
- PostgreSQL

- `User ID=rppp;Password=**;Host=localhost;Port=5432;Database=firma;Pooling=true;`

- Više primjera na <https://www.connectionstrings.com>

Izravna obrada podataka





Izravna obrada podataka na poslužitelju



1. Otvori priključak
 2. Izvrši naredbu
 3. Obradi podatke u čitaču
 4. Zatvori čitač
 5. Zatvori priključak
- Za vrijeme obrade (čitanja) podataka priključak na izvor podataka je otvoren!

Dodavanje NuGet paketa

- U primjerima koji slijede potrebno uključiti dodatne biblioteke
 - *Manage NuGet Packages* ili ručno ažurirati csproj datoteke
- U primjerima s izravnom obradom ti paketi su
 - Microsoft.Data.SqlClient
 - Microsoft.Extensions.Configuration.UserSecrets


	Microsoft.Data.SqlClient by Microsoft	5.0.1
	Provides the data provider for SQL Server. These classes provide access to versions of SQL Server and encapsulate database-specific protocols, including tabular data stream (TDS)	
	Microsoft.Extensions.Configuration.UserSecrets by Microsoft	6.0.1
	User secrets configuration provider implementation for Microsoft.Extensions.Configuration.	

Skica rješenja izravne obrade podataka

- Kostur rješenja s izravnom obradom, ali bez obrade iznimki

```
string connString = ... ;
IDbConnection conn = new SqlConnection(connString);
IDbCommand command = new SqlCommand();
command.CommandText = "SELECT TOP 3 * FROM Artikl ORDER BY ...";
command.Connection = conn;
conn.Open();
IDataReader reader = command.ExecuteReader();
while (reader.Read())
{
    object NazivArtikla = reader["NazArtikla"];
    ...
}
reader.Close();
conn.Close();
```

Postavke priključka na bazu podataka


- Postavke staviti u konfiguracijsku datoteku pod *ConnectionStrings*
 - Nije nužno koristiti taj naziv, ali postoje postupci (pokrati) koje očekuju postavke upravo pod tim ključem
 - Voditi računa o tome je li izvorni kod javno dostupan
- Primjer:  DataAccess / DataReader / appsettings.json
 - Desni klik → Copy To Output Directory : Copy if newer

```
{  
  "ConnectionStrings": {  
    "Firma": "Data Source=.;Initial Catalog=Firma;Integrated  
Security=true"  
    ...  
  }  
}
```

- Primjer:  ...UserSecrets/Firma/secrets..json

```
{  
  "ConnectionStrings": {  
    "Firma": "Data Source=fantom.fer.hr,3000;Initial Catalog=Firma;User  
Id=rpppp;Password=prava_šifra"    ...  
  }  
}
```

Dohvat postavki priključka na bazu podataka


- Dohvatljivo iz koda pomoću razreda *ConfigurationBuilder*
- Primjer:  DataAccess / DataReader / Program.cs
 - Metoda proširenja *GetConnectionString* koja u JSON datoteci traži vrijednost ispod elementa *ConnectionStrings*

```
IConfiguration configuration = new ConfigurationBuilder()  
                                .AddJsonFile("appsettings.json")  
                                .AddUserSecrets<Program>()  
                                .Build();  
  
string connString = configuration  
                    .GetConnectionString("Firma");
```

- Mapa za tajne vrijednosti određena u csproj datoteci projekta kojem pripada klasa Program

```
<PropertyGroup>                                 DataAccess / DataReader / DataReader.csproj  
  <UserSecretsId>Firma</UserSecretsId>  
</PropertyGroup>
```


Zatvaranje priključka

- Svaku otvorenu vezu prema bazi podataka treba zatvoriti!
 - Što ako se dogodi iznimka u prethodnom primjeru?
 - `Conn.Close()` nije izvršen – veza ostaje otvorena i ne može se ponovo iskoristiti → Staviti `Conn.Close()` unutar `finally` blocka?
 - Priključak implementira sučelje *IDisposable*.
 - *Dispose* je (u ovom slučaju) ekvivalentan *Close* → koristiti `using`
 - Primjer  `.DataAccess \ DataReader \ Programs.cs`

```
using (var conn = new SqlConnection(connString)){  
    using (var command = conn.CreateCommand()){  
        ...  
        using (var reader = command.ExecuteReader()){ ...
```

- Napomena: rješenje s *using* nije uvijek moguće (npr. ako je čitanje preko *readera* izvan metode u kojoj *reader* nastaje)
 - Automatsko zatvaranje priključka kad se zatvori čitač

```
command.ExecuteReader(System.Data.CommandBehavior.CloseConnection)
```

Sučelje *IDbCommand*

- Reprezentira SQL naredbe koje se obavljaju nad izvorom podataka
 - upit može biti SQL naredba ili pohranjena procedura
- Važnija svojstva
 - *Connection*: priključak na izvor podataka
 - *CommandText*: SQL naredba, ime pohranjene procedure ili ime tablice
 - *CommandType*: tumačenje teksta naredbe, standardno *Text*

```
enum CommandType { Text, StoredProcedure, TableDirect }
```
- Važniji postupci
 - *ExecuteReader* – izvršava naredbu i vraća *DataReader*
 - *ExecuteNonQuery* – izvršava naredbu koja vraća broj obrađenih zapisa, npr. neka od naredbi UPDATE, DELETE ili INSERT.
 - *ExecuteScalar* – izvršava naredbu koja vraća jednu vrijednost, npr. rezultat agregatne funkcije


Sučelje *IDataReader*

- Sučelje za iteriranje nad rezultatom upita.
- Važnija svojstva
 - *Item* – vrijednost stupca u izvornom obliku
 - `public virtual object this[int] {get;}`
 - `public virtual object this[string] {get;}`
 - *FieldCount* - broj stupaca u rezultatu upita
- Važniji postupci
 - *Read* – prelazi na sljedeći redak rezultata i vraća true ako takav postoji
 - *Close* – zatvara *DataReader* objekt (ne nužno i priključak s kojeg čita)
 - *GetName* – vraća naziv za zadani redni broj stupca
 - *GetOrdinal* – vraća redni broj za zadano ime stupca
 - *GetValue* – dohvaća vrijednost zadanog stupca za aktualni redak
 - `public virtual object GetValue(int ordinal);`
 - *GetValues* – dohvaća aktualni redak i sprema u polje objekata
 - `public virtual int GetValues(object[] values);`
 - *GetXX* (*GetInt32*, *GetChar*, ...) – dohvaća vrijednost zadanog stupca pretpostavljajući određeni tip

Neovisnost o konkretnoj implementaciji

- Primjeri se mogu poopćiti na način da se za tip reference umjesto konkretnih implementacija koriste sučelja ili apstraktni razredi
 - *IDbConnection* ili *DbConnection*
 - *IDbCommand* ili *DbCommand*
 - *IDataReader* ili *DbDataReader*
- Alternativno definirati reference s ključnom riječi *var*.
- *DBProviderFactory* kao „tvornica“
 - Omogućava stvaranje priključaka i naredbi bez navođenja konkretnih implementacija
 - Postupci *CreateConnection*, *CreateCommand*, ... kao rezultat vraćaju instance konkretnih implementacija, ali promatrane kroz odgovarajuće apstraktne razrede
 - Primjer slijedi uskoro


DbProviderFactory / DbProviderFactories

- .NET Framework:
 - Statički postupak GetFactory u razredu DbProviderFactories
- .NET Core:
 - [*]ClientFactory.Instance
 - Ili prethodno registrirati s DbProviderFactories.RegisterFactory("System.Data.SqlClient", SqlConnectionFactory.Instance);
- Primjer:  DataAccess / ParamsAndProc / Program.cs

```
DbProviderFactory factory = SqlConnectionFactory.Instance;

using (DbConnection conn = factory.CreateConnection()) {
    conn.ConnectionString = ...
    using (DbCommand command = factory.CreateCommand()){
        command.Connection = conn;
        ...
    }
}
```

Parametrizirani upiti

- Dijelovi upita s parametrima oblika @NazivParametra
 - Olakšava pisanje upita
 - Brže izvršavanje u slučaju višestrukih izvršavanja
 - Zaštita od SQL injection napada
 - Parametar se kreira s new [Sql]Parameter ili pozivom postupka CreateParameter na nekoj naredbi
- Primjer:  DataAccess \ ParamsAndProc \ Program.cs – ParametrizedQueryDemo

```
command.CommandText = "SELECT TOP 3 * FROM Artikl WHERE JedMjere =  
    @JedMjere ORDER BY CijArtikla DESC;" +  
    "SELECT TOP 3 * FROM Artikl WHERE JedMjere = @JedMjere AND  
    CijArtikla > @Cijena ORDER BY CijArtikla";
```


```
DbParameter param = command.CreateParameter();  
param.ParameterName = "JedMjere"; param.DbType = DbType.String;  
param.Value = "kom"; command.Parameters.Add(param);
```

```
param = command.CreateParameter();  
param.ParameterName = "Cijena"; param.DbType = DbType.Decimal;  
param.Value = 100m; command.Parameters.Add(param);
```

Svojstva parametra


- *DbType* – vrijednost iz enumeracije *System.Data.DbType*
 - Predstavlja tip podatka koji se prenosi parametrom.
- *Direction* – vrijednost iz enumeracije *System.Data.ParameterDirection*
 - Određuje da li je parametar ulazni, izlazni, ulazno-izlazni ili rezultat poziva pohranjene procedure. Ako se ne navede, pretpostavlja se da je ulazni.
- *IsNullable* – određuje može li parametar imati null vrijednost
- *ParameterName* – naziv parametra
- *Size* – maksimalna veličina parametra u bajtovima
 - Upotrebljava se kod prijenosa tekstualnih podataka.
- *Value* – vrijednost parametra
 - Vrijednost izlaznog argumenta se može dobiti i preko instance naredbe `command.Parameters["Naziv parametra"].Value`

Upit s više skupova rezultata

- U slučaju da rezultat upita vraća više skupova rezultata, svaki sljedeći dohvaća se postupkom `NextResult` na čitaču podataka
 - Primjer:  `.DataAccess \ ParamsAndProc \ Program.cs` – `ParametrizedQueryDemo`

```
command.CommandText = "SELECT TOP 3 * FROM Artikl WHERE JedMjere =  
    @JedMjere ORDER BY CijArtikla DESC;" +  
    "SELECT TOP 3 * FROM Artikl WHERE JedMjere = @JedMjere AND  
    CijArtikla > @Cijena ORDER BY CijArtikla";  
  
...  
using (DbDataReader reader = command.ExecuteReader()){  
    do{  
        while (reader.Read()){  
            ...  
        }  
    } while (reader.NextResult());  
}
```


Pozivi pohranjenih procedura

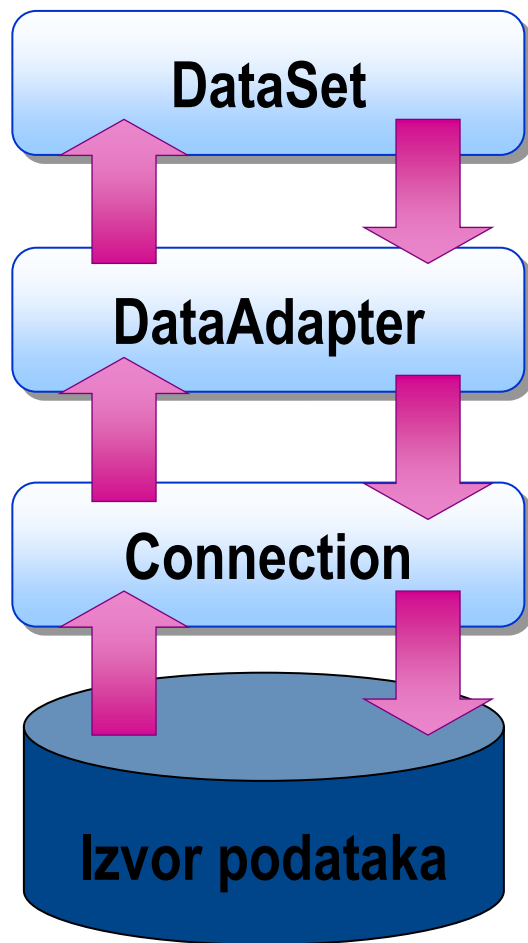
- Primjer:  DataAccess \ ParamsAndProc \ Program.cs – ProcedureDemo
 - Parametri procedure navode se kao i kod parametriziranih upita
 - Svojstvo *CommandType* na naredbi potrebno je postaviti na *System.Data.CommandType.StoredProcedure*
 - Ako procedura ne vraća skup podataka, koristi se postupak *ExecuteNonQuery*
 - Očekuje li se skup podataka kao rezultat koristi se *ExecuteReader*.
 - Vrijednosti izlaznih parametara mogu se dobiti **tek po zatvaranju čitača**

```
command.CommandText = "ap_ArtikliSkupljajOd";
command.CommandType = System.Data.CommandType.StoredProcedure;
...
param = command.CreateParameter();
param.ParameterName = "BrojJeftinijih"; param.DbType = DbType.Int32;
param.Direction = System.Data.ParameterDirection.Output;
command.Parameters.Add(param);
...
using (DbDataReader reader = command.ExecuteReader()){ ... }
int brJef = command.Parameters["BrojJeftinijih"].Value
```

Lokalna obrada podataka

Entity Framework Core

Lokalna obrada podataka



- Podaci se obrađuju lokalno
 - *DataSet* reprezentira stvarne podatke pohranjene u memoriju
- 1. Otvori priključak
- 2. Napuni DataSet
- 3. Zatvori priključak
- 4. Izmijeni DataSet
- 5. Otvori priključak
- 6. Ažuriraj izvor podataka
- 7. Zatvori priključak
 - Ideja lokalne obrade podataka *DataSetom* „prenesena” na EntityFramework

Načini kreiranja EF modela

- *Database First*
 - Baza podataka već postoji i model nastaje reverznim inženjerstvom BP
- *Model First*
 - Model se dizajnira kroz grafičko sučelje, a BP nastaje na osnovu modela.
- *Code First*
 - Model opisan kroz ručno napisane razrede te nema vizualnog modela
 - BP se stvara temeljem napisanih razreda. Izgled određen nazivima razreda, nazivima i vrstama asocijacija između razreda te dodatnim atributima.
- *Code First from existing database*
 - Slično kao Code First, ali za postojeću bazu podataka
 - Baza podataka opisuje se razredima, ali se ne stvara nova baza podataka.
 - Razredi se mogu stvoriti ručno ili nekim od generatora.
- *Code First with Migrations*
 - Izvršavaju se posebno definirani postupci (migracije) - Up/Down
- *Entity Framework Core podržava samo Code First varijante*
 - **U primjerima koristimo Code First from existing database**

Stvaranje modela na osnovu postojeće BP

1. Instalirati dotnet-ef na računalu

```
dotnet tool install --global dotnet-ef
```

2. U mapi ciljanog projekta izvršiti sljedeće naredbe

```
dotnet add package Microsoft.EntityFrameworkCore.Design  
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

1. ili dodati koristeći opciju Manage NuGet Packages

3. U naredbenom retku izvršiti sljedeće dvije naredbe

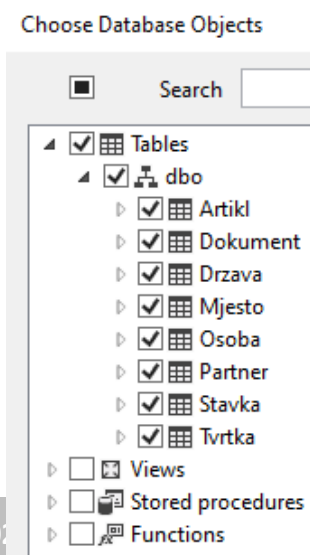
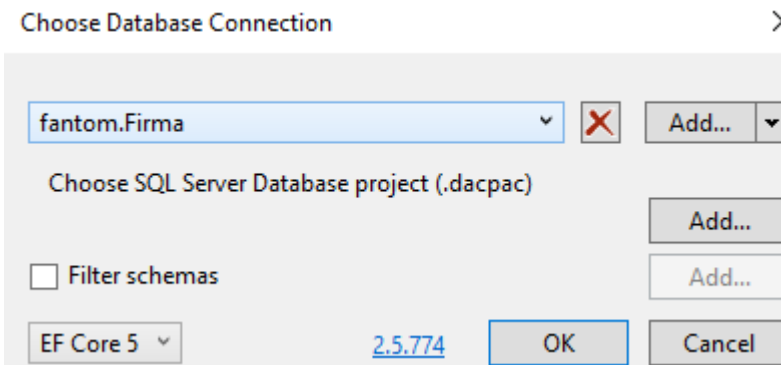
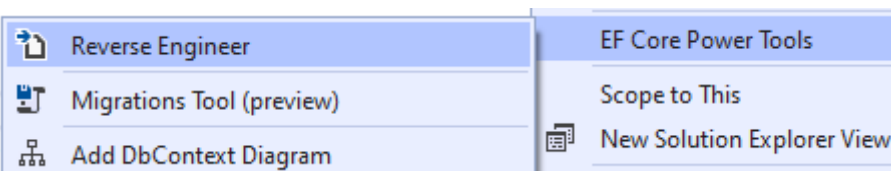
```
dotnet restore
```

```
dotnet ef dbcontext scaffold
```

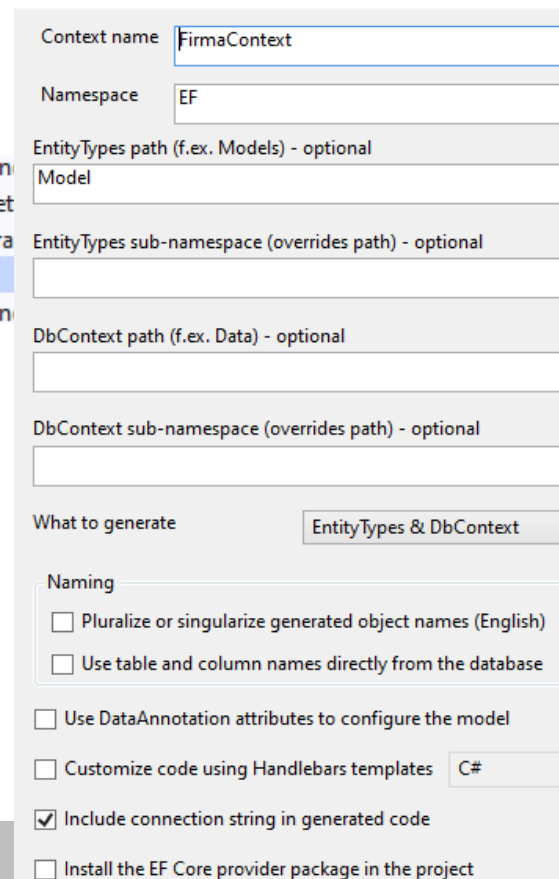
```
"Server=fantom.fer.hr,3000;Database=Firma;User  
Id=rpppp;Password=*" Microsoft.EntityFrameworkCore.SqlServer -  
o Model -t Artikel -t Dokument -t Drzava -t Mjesto -t Osoba -t  
Partner -t Stavka -t Tvrtka --no-pluralize
```

EF Core Power Tools

- EF Core Power Tools (dodatak za Visual Studio)
 - <https://marketplace.visualstudio.com/items?itemName=ErikEJ.EFCorePowerTools>
 - praktičniji način za stvaranje (i naknadno ažuriranje modela)
 - nudi mogućnost uvoza i procedura i funkcija iz SQL servera
 - Desni klik na projekt →
EF Core Power Tools → Reverse Engineer

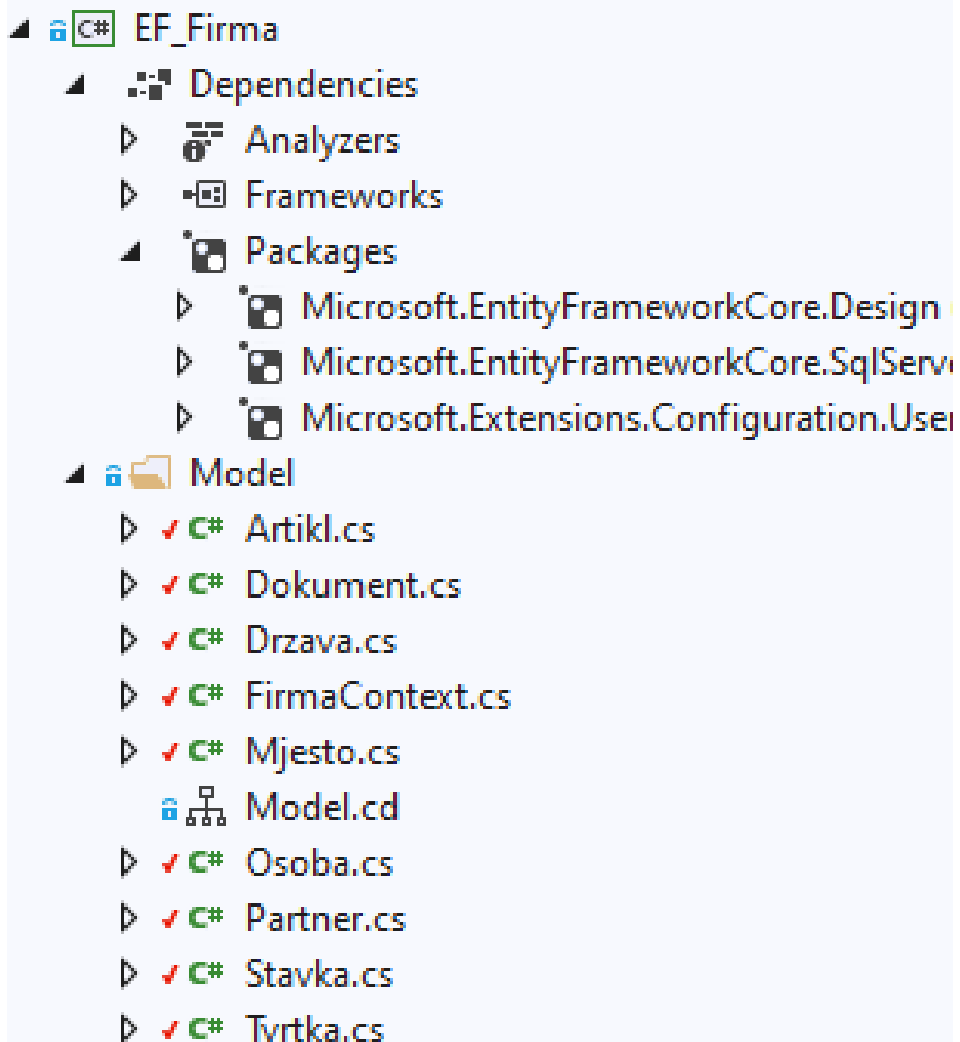


Generate EF Core Model in Project EF




Model nastao temeljem postojeće BP

- Na osnovu postojećih stranih ključeva EF automatski stvara asocijacije između stvorenih
- razreda
- Za naš primjer stvaraju se:
 - Firma.Context.cs
 - Po jedna cs datoteka za svaku tablicu
- Postavke spajanja inicijalno tvrdo kodirane u FirmaContext.cs
 - **Potrebno ukloniti i prebaciti u konfiguracijsku datoteku**



Postavke spajanja na BP korištenjem EF-a (1)

- Generirani model sadrži tvrdo kodirane postavke za spajanje na BP
 - Primjer:  Bilo koji [Naziv]Context.cs stvoren prema prethodnim uputama


```
void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {  
    optionsBuilder.UseSqlServer(@"Server=...sword=*");  
}
```

- Može se zamijeniti kodom za dohvat konfiguracijskih postavki

```
void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {  
    var config = new ConfigurationBuilder()  
        ...  
        .Build();  
    string connString = config.GetConnectionString("Firma");  
    optionsBuilder.UseSqlServer(connString);  
}
```

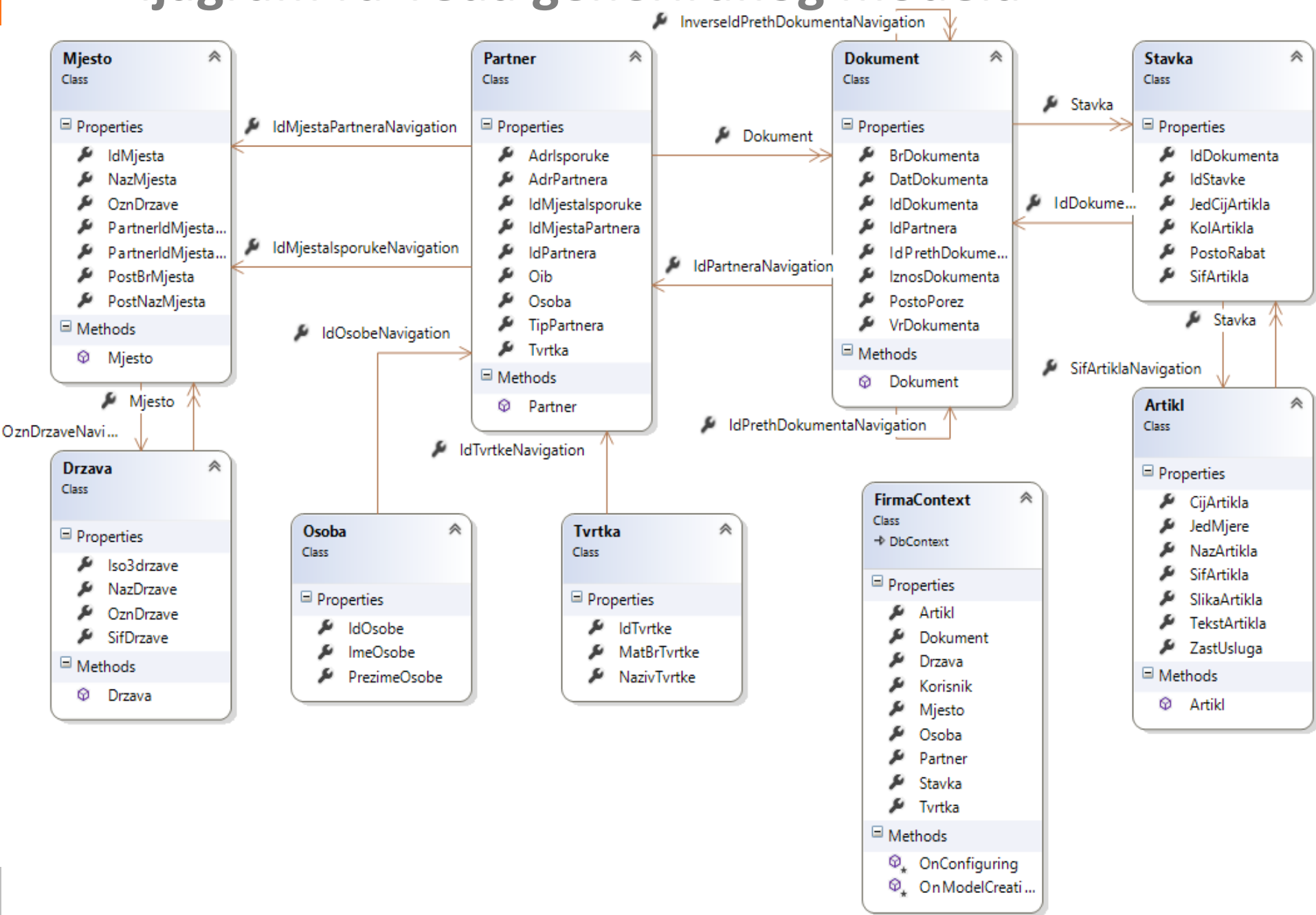
- Nije idealno rješenje, jer model određuje naziv konfiguracijske datoteke i naziv ključa
- Bolje rješenje na sljedećem slajdu

Postavke spajanja na BP korištenjem EF-a (2)


- Onemogućiti stvaranje *FirmaContexta* direktno
 - Obrisati konstruktor bez argumenata
 - *Connection string* kao argument? → Nije praktično. Potrebno svaki put prije instanciranja dohvatiti *connection string*
 - Posljedično može se obrisati *OnConfiguring*
- Uspostaviti lanac ovisnosti potreban za stvaranje objekta tipa *FirmaContext* koristeći objekt tipa *ServiceProvider*
 - Svaki put potrebno stvoriti novi kontekst (Transient)
 - Primjer:  DataAccess \ EF \ Model \ Program.cs : BuildDI

```
IServiceCollection services = new ServiceCollection();  
var provider = services  
    .AddDbContext<FirmaContext>(options=>{  
        options.UseSqlServer(  
            configuration.GetConnectionString("Firma"));  
        }, contextLifetime: ServiceLifetime.Transient)  
    .BuildServiceProvider();
```

Dijagram razreda generiranog modela



Elementi EF modela

- *FirmaContext* – naslijeđen iz razreda *DbContext*
 - predstavlja kontekst za pristup bazi podataka
 - podaci pohranjeni unutar konteksta u skupu entiteta tipa `DbSet<T>`, gdje je T tip entiteta
 - Definiran u  `.DataAccess \ EF \ Model \ FirmaContext.cs`
- Svaki entitet predstavljen parcijalnim razredom
 - Asocijacije kao virtualna svojstva (*ICollection<T>* za agregacije)
 - Omogućava stvaranje proxy razreda koji pruža vlastitu implementaciju virtualnih svojstava
- „Korisnički” definiran dio parcijalnih razreda smješta se unutar projekta po volji
 - Generirani razredi su parcijalni, pa se njihova definicija može nalaziti u više datoteka
 - Generirani kontekst sadrži i parcijalne metode

Preslikavanje između EF modela i BP (1)

- Primjer:  DataAccess \ EF \ Model \ FirmaContext.cs

```
protected override void OnModelCreating(  
    modelBuilder modelBuilder)  
{  
    modelBuilder.Entity<Artikl>(entity =>  
    {  
        entity.HasKey(e => e.SifArtikla).HasName("pk_Artikl");  
        entity.HasIndex(e => e.NazArtikla)  
            .HasName("ix_Artikl_NazArtikla")  
            .IsUnique();  
        entity.Property(e => e.SifArtikla)  
            .HasDefaultValueSql("0");  
        entity.Property(e => e.CijArtikla)  
            .HasColumnType("money")  
            .HasDefaultValueSql("0");  
    });  
}
```

Preslikavanje između EF modela i BP (2)

- Entiteti ostaju „čisti“, a preslikavanje je u jednom postupku
- Olakšava promjenu naziva atributa u bazi podataka
- Npr. PostgreSQL ima drugačiji stil imenovanja i tipove, pa bi tada isječak prilagođen za PostgreSQL bio

```
protected override void OnModelCreating(  
    modelBuilder modelBuilder)  
{  
    modelBuilder.Entity<Artikl>(entity =>  
    {  
        entity.Property(e => e.JedMjere)  
            .IsRequired()  
            .HasColumnName("jed_mjere")  
            .HasColumnType("varchar")  
            .HasMaxLength(5)  
            .HasDefaultValueSql("'kom'::character varying");  
    });  
}
```


Važnija svojstva razreda DbContext

- *SaveChanges[Async]*
 - spremanje promjena u bazi podataka
- *Database*
 - svojstvo koje omogućava direktni rad s BP (npr. kreiranje i brisanje BP, izvršavanje vlastitih SQL upita i procedura)
- *ChangeTracker*
 - pristup do razreda koji prati promjene na objektima u kontekstu
- *Set* i *Set<T>*
 - vraćaju DbSet za konkretni tip entiteta (Koristi se ako se želi napisati općeniti postupak, inače je svaki entitet već sadržan u kontekstu kao svojstvo)
- *Entry* i *Entry<T>*
 - služi za dohvat informacije o nekom entitetu u kontekstu i promjenu njegovog stanja (npr. otkazivanje promjena)

Važnija svojstva razreda *DbSet*


- *Add*
 - dodavanje objekta u skup
- *Remove*
 - označavanje objekta za brisanje
- *Local*
 - kolekcija svih trenutno učitanih podataka (koristi se za povezivanje na forme)
- *Find [Async]*
 - Dohvat objekta unutar konteksta na osnovu primarnog ključa
- *AsNoTracking*
 - Dohvat podataka za koje se ne evidentiraju promjene

Dodavanje novog zapisa

- Primjer:  DataAccess \ EF \ Program.cs - AddProduct
 - Stvoriti novi objekt konstruktorom te ga dodati u kolekciju nekom od mogućih varijanti
 - `context.Artikl.Add(artikl);`
 - `context.Add(artikl);`
 - `context.Set<Artikl>().Add(artikl);`
 - Pohraniti promjene u kontekstu (jednom za sve promjene)


```
using (var context = new FirmaContext())  
using (var context =  
    serviceProvider.GetService<FirmaContext>()) {  
    Artikl artikl = new Artikl() { ... };  
    context.Artikl.Add(artikl);  
    context.SaveChanges();  
}
```

Ažuriranje postojećeg zapisa

- Primjer:  DataAccess \ EF \ Program.cs – ChangeProductPrice
 - Dohvatiti entitet
 - korištenjem postupka Find ili FindAsync na DbSetu – traži zapis na osnovu vrijednosti primarnog ključa
 - Pretražuje unutar već učitano konteksta, a ako ga ne pronađe obavlja se upit na bazu. Vraća null ako traženi zapis ne postoji
 - Ili postavljanjem Linq upita
 - Promijeniti željena svojstva i pohraniti promjene u kontekstu


```
using (var context = ...) {  
    Artikl artikl = context.Artikl.Find(sifraArtikla);  
    //moglo je i context.Find<Artikl>(sifraArtikla);  
    artikl.CijArtikla = 750m;  
    context.SaveChanges();  
}
```

Brisanje zapisa

- Primjer:  DataAccess \ EF \ Program.cs – DeleteProduct
 - Dohvatiti entitet
 - Izbaciti ga iz konkretnog *DbSeta* ili označiti ga za brisanje pomoću *context.Entry*
 - Pohraniti promjene u kontekstu

```
using (var context = ...) {  
    Artikl artikl = context.Artikl.Find(sifraArtikla);  
    context.Artikl.Remove(artikl);  
    //ili context.Entry(artikl).State = EntityState.Deleted;  
    context.SaveChanges();  
}
```

Upiti nad EF modelom

- *Where, OrderBy, OrderByDescending, ThenBy, First, Skip, Take, Select, ...*
 - Davatelj usluge pretvara Linq upit u SQL upit
 - Nije uvijek moguće sve pretvoriti u SQL upit
- Upit se izvršava u trenutku dohvata prvog podataka ili eksplicitnim pozivom postupka *Load (LoadAsync)*
 - Moguće ulančavanje upita (rezultat upita najčešće *IQueryable<T>*)
 - Podaci iz vezane tablice se učitavaju pri svakom dohvatu ili eksplicitno korištenjem postupka *Include* (kreira join upit u sql-u)
- Primjer  `.DataAccess \ EF \ Program.cs` – `PrintMostExpensives`
 - Primjer upita za dohvat prvih n najskupljih artikala


```
var query = context.Artikl.Include(a => a.Stavka)
                        .AsNoTracking()
                        .OrderByDescending(a => a.CijArtikla)
                        .Take(n);
foreach (Artikl artikl in query) { ... }
```

SQL nastao upitom kroz EF

- Za prethodni primjer na SQL serveru će se izvršiti sljedeći upit
 - Trebalo li sve podatke?
 - Možda samo trebamo ispisati koliko artikl ima stavki?

```
exec sp_executesql N'SELECT [s].[IdStavke],  
[s].[IdDokumenta], [s].[JedCijArtikla], [s].[KolArtikla],  
[s].[PostoRabat], [s].[SifArtikla]  
FROM [Stavka] AS [s]  
INNER JOIN (  
    SELECT DISTINCT TOP(@__p_0) [a].[CijArtikla],  
[a].[SifArtikla]  
    FROM [Artikl] AS [a]  
    ORDER BY [a].[CijArtikla] DESC, [a].[SifArtikla]  
) AS [a0] ON [s].[SifArtikla] = [a0].[SifArtikla]  
ORDER BY [a0].[CijArtikla] DESC, [a0].[SifArtikla]',N'@__p_0  
int',@__p_0=10
```

Anonimni razredi kao rezultati upita

- Rezultat upita ne mora biti neki od postojećih entiteta, već podskup ili agregacija više njih
- Rezultat je anonimni razred sa svojstvima navedenim u upitu
 - Može se dati novo ime za pojedino svojstvo
 - Primjer  DataAccess \ EF \ Program.cs – PrintMostExpensivesAnonymous

```
var query = context.Artikl
    .OrderByDescending(a => a.CijArtikla)
    .Select(a => new {
        a.NazArtikla, a.CijArtikla,
        Sales = a.Stavka.Count
    })
    .Take(n);
foreach (var product in query) { ... }
```

SQL nastao modificiranim upitom

- SQL za prethodni EF upit je jednostavniji

```
exec sp_executesql N'SELECT TOP(@__p_0) [a].[NazArtikla],  
[a].[CijArtikla], (  
    SELECT COUNT(*)  
    FROM [Stavka] AS [s0]  
    WHERE [a].[SifArtikla] = [s0].[SifArtikla]  
)  
FROM [Artikl] AS [a]  
ORDER BY [a].[CijArtikla] DESC',N'@__p_0 int',@__p_0=10
```

Ostale mogućnosti EF-a

- Nakon uspješnog upita EF automatski vrši dohvat primarnog ključa koji je definiran kao tip *identity*
- U trenutku pisanja ovih materijala EF Core ne podržava dodavanje procedura u model, ali EF Core Power Tools može generirati kod koji to omogućava
- Moguće je samostalno napisati upit, ali rezultat je (trenutno) moguće samo pohraniti u neki od postojećih entiteta
- Poglede je moguće dodati u model, što će biti prikazano u poglavlju s web-aplikacijama

Migracije (1)

- Generirani kod vrši promjene u bazi temeljem promjena u modelu
 - Početna migracija ili promjena u odnosu na predhodno stanje
 - Migracije se mogu koristiti neovisno o načinu dobivanja modela (*code first* ili *code first from existing database*)
- Praktično nužno u slučaju da direktni pristup bazi podataka nije moguć, ili ako svaki korisnik ima svoju bazu
- Može biti praktično u slučaju kontinuirane isporuke, jer rješava raskorak između promjene baze podataka i usklađivanja modela
 - Potencijalni problem s više istovremenih instanci
- Migracije se mogu pokretati ručno (*dotnet ef* u naredbenom retku ili *Add-Migration* u *Package Manager Console*) ili kroz kod.

Migracije (2)

- Promotriti što bi se generiralo s

```
dotnet ef migrations add CreateDb -c FirmaContext -o  
Migrations
```

- Može se promijeniti *connection string* i unijeti postavke za lokalno instalirani SQL server i izvršiti

```
dotnet ef database update -c FirmaContext
```

- Što bi se dogodilo da u EF model dodamo nova svojstva i kreiramo novu migraciju?