

1

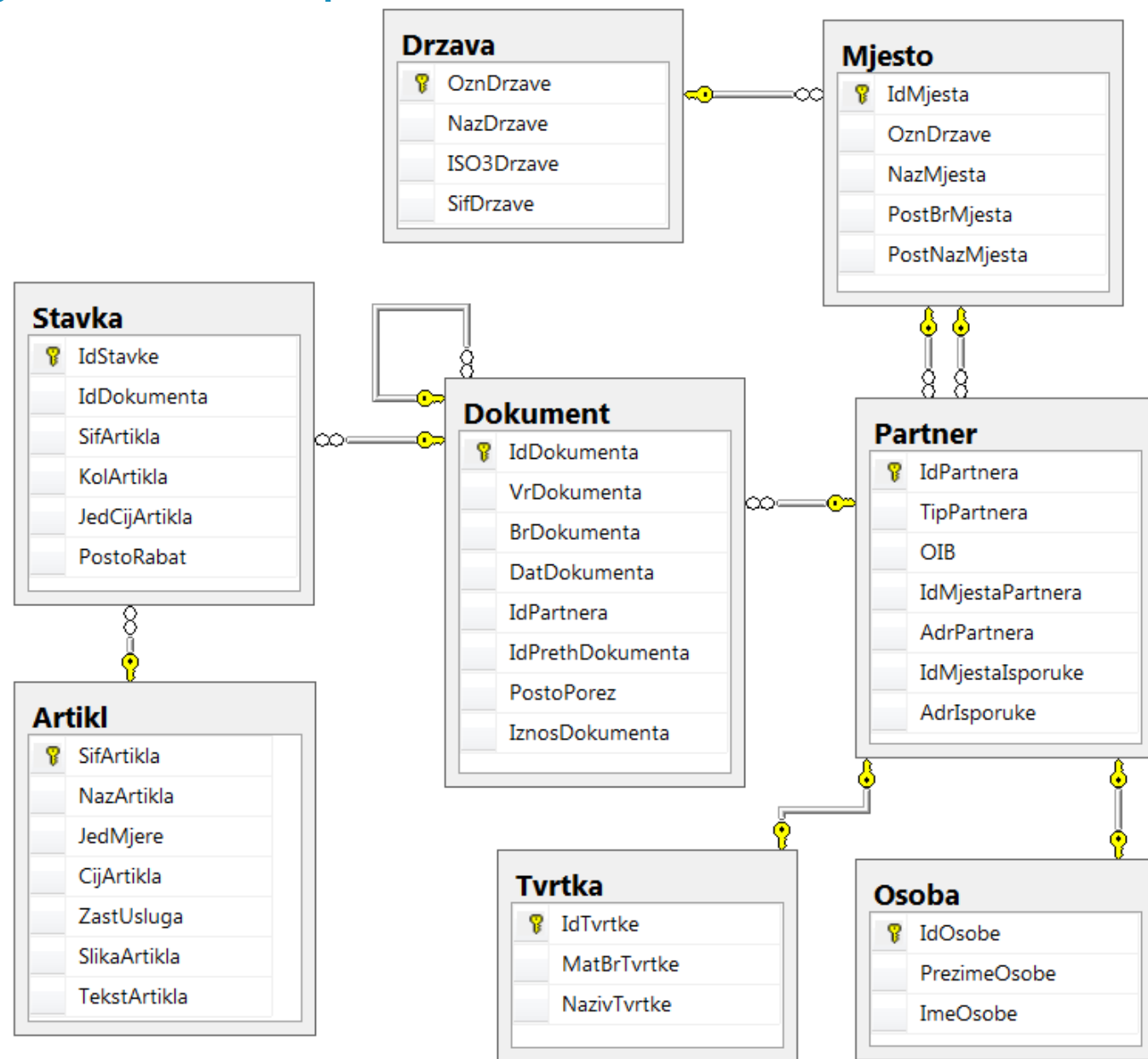
Rad s bazom podataka

2016/17.07

Ogledna baza podataka

2

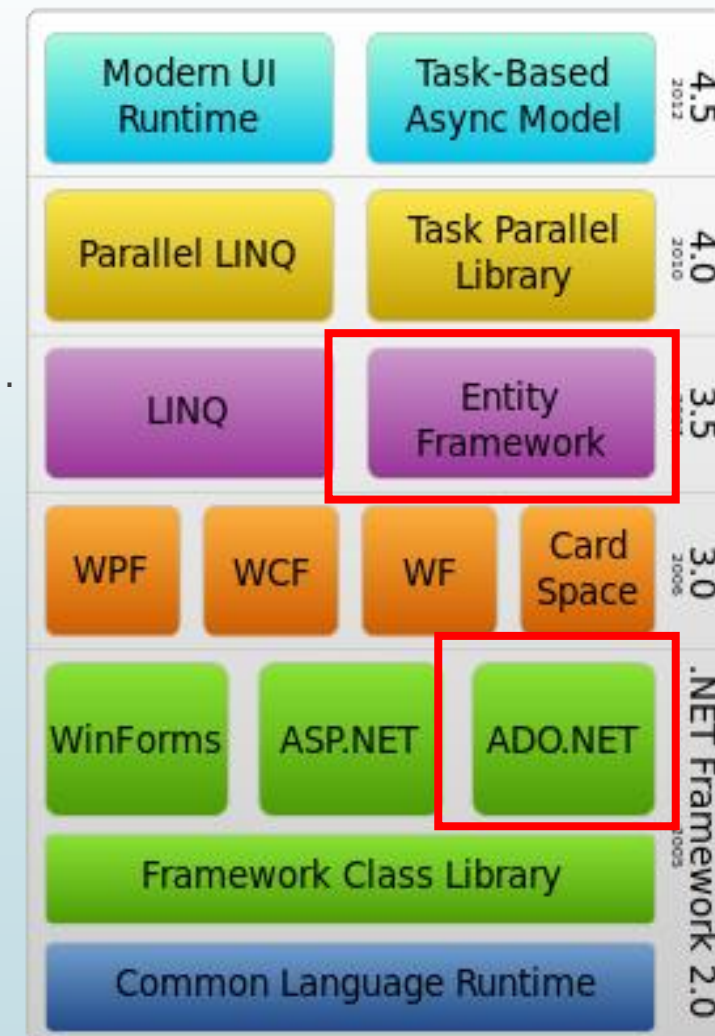
- SQL Server:
rppp.fer.hr,3000
- Baza podataka: Firma
 - SQL Server Authentication:
rppp/lozinka se nalazi u
popisima u privatnom
repozitoriju
 - Moguće mijenjati podatke u
svrhu testiranja



.NET Framework i ADO.NET

3

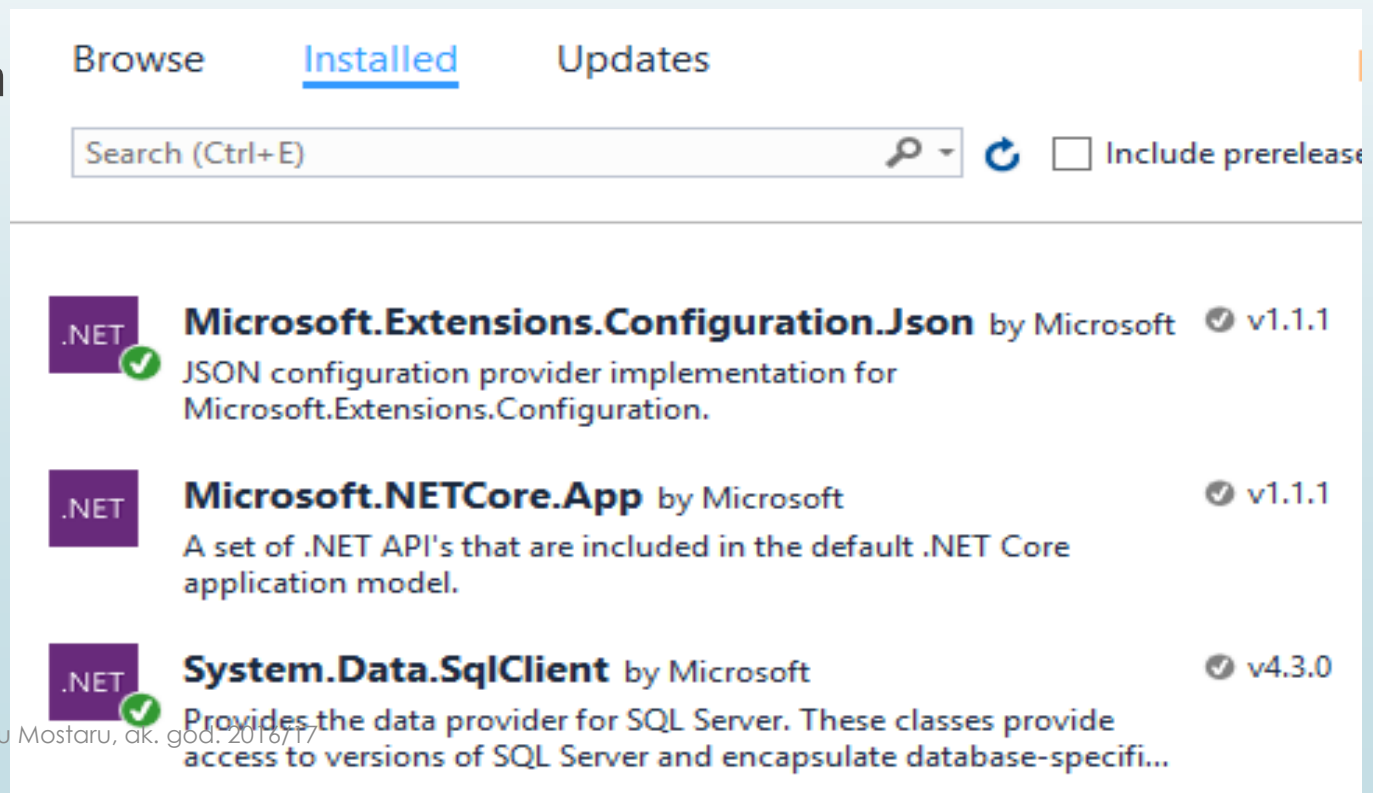
- ActiveX Data Objects .NET (ADO.NET) je tehnologija za rukovanje podacima
 - Omogućuje pristup bazama podataka, ali i drugim spremištima podataka, za koje postoji odgovarajući opskrbljivač podacima (*provider*)
 - sinonimi za opskrbljivač: davatelj, pružatelj, poslužitelj
- Podrška različitim tipovima spremišta
 - Strukturirani, nehijerarhijski podaci
 - Comma Separated Value (CSV) datoteke, Microsoft Excel 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**



NuGet paketi korišteni u primjerima

4

- U primjerima koji slijede bit će potrebno uključiti dodatne biblioteke
 - naredbom *dotnet add package*, ručnim ažuriranjem *csproj* datoteke ili odabirom opcije *Manage NuGet Packages*
- U primjeru koji slijedi ti paketi su
 - *Microsoft.Extensions.Configuration.Json* (za korištenjem JSON konfiguracijskih datoteka)
 - *System.Data.SqlClient* za rad s Microsoft SQL Serverom



Opskrbljivači (davatelji) podataka

5

- Davatelji za različite tehnologije (SQL Server, PostgreSQL, SQLite, MongoDB, ...)
 - <https://blogs.msdn.microsoft.com/dotnet/2016/11/09/net-core-data-access/>
 - Direktni pristup ili tehnologije s određenom razinom apstrakcije (npr. ORM alati)
- **System.Data.SqlClient**
 - optimiran za rad s MS SQL Server-om
 - Razredi: *SqlCommand*, *SqlConnection*, *SqlDataReader*, *SqlDataAdapter*
- Za ostale relacijske baze podataka razredi sličnih naziva
 - Npr. *NpgsqlConnection*, *NpgsqlCommand*, *SqlConnection*, ...
- Navedeni razredi implementiraju zajednička sučelja pa imaju članove jednakih naziva
 - neovisnost aplikacije o fizičkom smještaju podataka

Osnovni pojmovi u pristupu bazi podataka

6

- **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

7

- 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* – prikapćanje na izvor podataka
 - *Close* - otkapćanje s izvora podataka

Primjeri postavki priključka na bazu

8

➤ Microsoft SQL Server

➤ `Data Source=.;Initial Catalog=Firma;Integrated Security=True`

➤ `Data Source=rppp.fer.hr,3000;Initial Catalog=Firma;UserId=rppp;Password=šifra`


➤ PostgreSQL

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

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

Konfiguracijska datoteka za .NET Core

9

- Izbjegavati pisanje postavki priključka unutar koda
 - promjena postavki bi zahtijevala ponovnu izgradnju programa
 - nova verzija programa?
 - potencijalni sigurnosni problem
 - Može se pojaviti kao informaciju u tragu stoga neke iznimke
- Postavke staviti u konfiguracijsku datoteku – JSON datoteka proizvoljnog imena
 - Primjer:  DataAccess / DataReader / appsettings.json

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

Dohvat podataka iz konfiguracijske datoteke

10

- Dohvatljivo iz koda pomoću razreda *ConfigurationBuilder*
 - Nalazi se u NuGet paketu *Microsoft.Extensions.Configuration.Json*
- Primjer:  DataAccess / DataReader / Program.cs

```
var config = new ConfigurationBuilder()  
    .SetBasePath(Directory.GetCurrentDirectory())  
    .AddJsonFile("appsettings.json")  
    .Build();  
string connString = config["ConnectionStrings:Firma"];
```

Zaštita postavki za spajanje na bazu

11

- Što ako prilikom razvoja treba javno dijeliti izvorni kod?
- Mehanizam „korisničkih tajni” (engl. user secrets)
- U projektnim datotekama stoji ključ (<userSecretsId>), a vrijednost spremljena u korisnikovom profilu
 - Windows: %APPDATA%\microsoft\UserSecrets\<userSecretsId>\secrets.json
 - Linux: ~/.microsoft/usersecrets/<userSecretsId>/secrets.json
 - Mac: ~/.microsoft/usersecrets/<userSecretsId>/secrets.json
- Samo za razvoj! Datoteke nisu kriptirane

Postavljanje datoteke s „tajnim” vrijednostima

12

- Izmijeniti projektnu datoteku i dodati odgovarajuće pakete (CLI paketi nisu podržani kroz grafičko sučelje)
- Primjer:  DataAccess / DataReader / DataReader.csproj

```
<Project Sdk="Microsoft.NET.Sdk"> ...  
  <PropertyGroup>  
    <UserSecretsId>Firma</UserSecretsId>  
  </PropertyGroup>  
  <ItemGroup>  
    <DotNetCliToolReference  
      Include="Microsoft.Extensions.SecretManager.Tools"  
      Version="1.0.0" />  
    <PackageReference  
      Include="Microsoft.Extensions.Configuration.UserSecrets"  
      Version="1.1.1" />  
  </ItemGroup>
```

- U naredbenom retku u mapi projekta pokrenuti
dotnet user-secrets set FirmaSqlPassword šifra
- Stvara .../Firma/secrets.json s ključem FirmaSqlPassword

Dohvat „tajnih” vrijednosti

13

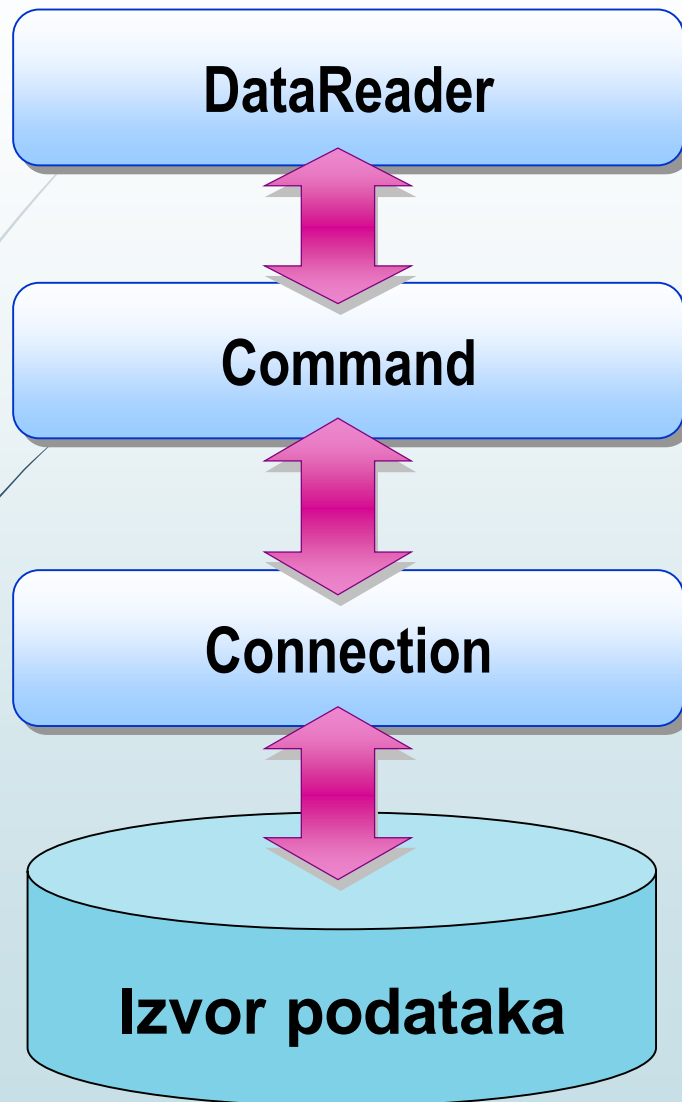
➡ Primjer:  DataAccess / DataReader / Program.cs

- ➡ U *appsettings.json* zapisano sve osim prave lozinke korisnika
- ➡ *AddUserSecrets("Firma")* uključuje datoteku .../Firma/secrets.json koja sadrži ključ *FirmaSqlPassword*

```
var config = new ConfigurationBuilder()  
    .AddUserSecrets("Firma")  
    .SetBasePath(Directory.GetCurrentDirectory())  
    .AddJsonFile("appsettings.json")  
    .Build();  
  
string connString = config["ConnectionStrings:Firma"];  
  
connString = connString.Replace("sifra",  
    config["FirmaSqlPassword"]);
```

Izravna obrada podataka na poslužitelju

14



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!


Skica rješenja izravne obrade podataka

15

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

Zatvaranje priključka

16

- Svaku otvorenu vezu prema bazi podataka treba zatvoriti!
- Što ako se dogodi iznimka u prethodnoj skici?
 - Conn.Close() ne bi bio 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 / Program.cs

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

- Napomena: Ako rješenje s using nije izvedivo (npr. postupak vraća IDataReader koji se naknadno koristi) može se prilikom izvršavanja upita postaviti automatsko zatvaranje priključka prilikom zatvaranja čitača

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


Neovisnost o konkretnoj implementaciji

17

- 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

Sučelje IDbCommand

18

- 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

19

➤ Sučelje za rezultat upita nad podacima (forward-only, read-only connected result set).

➤ Važnija svojstva

➤ `Item` – vrijednost stupca u izvornom obliku

➤ `public virtual object this[int] {get;}`

➤ `public virtual object this[string] {get;}`

➤ Važniji postupci

➤ `Read` – čita sljedeći zapis u *DataReader*

➤ vraća `true` ako postoji još zapisa (potrebno za inicijalno čitanje)

➤ `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 kao polje objekata

➤ `public virtual int GetValues(object[] values);`

DbProviderFactory / DbProviderFactories

20

➤ .NET Framework:

➤ Statički postupak GetFactory u razredu DbProviderFactories

➤ .NET Core:

➤ [*]ClientFactory.Instance


➤ Primjer:  DataAccess / ProcedureParametri / Program.cs

```
DbProviderFactory factory = SqlClientFactory.Instance;

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

Parametrizirani upiti

21

- Dijelovi upita s parametrima oblika @NazivParametra ili ?
 - Olakšava pisanje upita i ubrzava 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 \ Parametri \ Program.cs

```
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

22

- *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


23

- ➡ U slučaju da rezultat upita vraća više skupova rezultata, svaki sljedeći dohvaća se postupkom *NextResult* na čitaču podataka
- ➡ Primjer:  DataAccess \ Parametri \ Program.cs

```
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());  
}
```

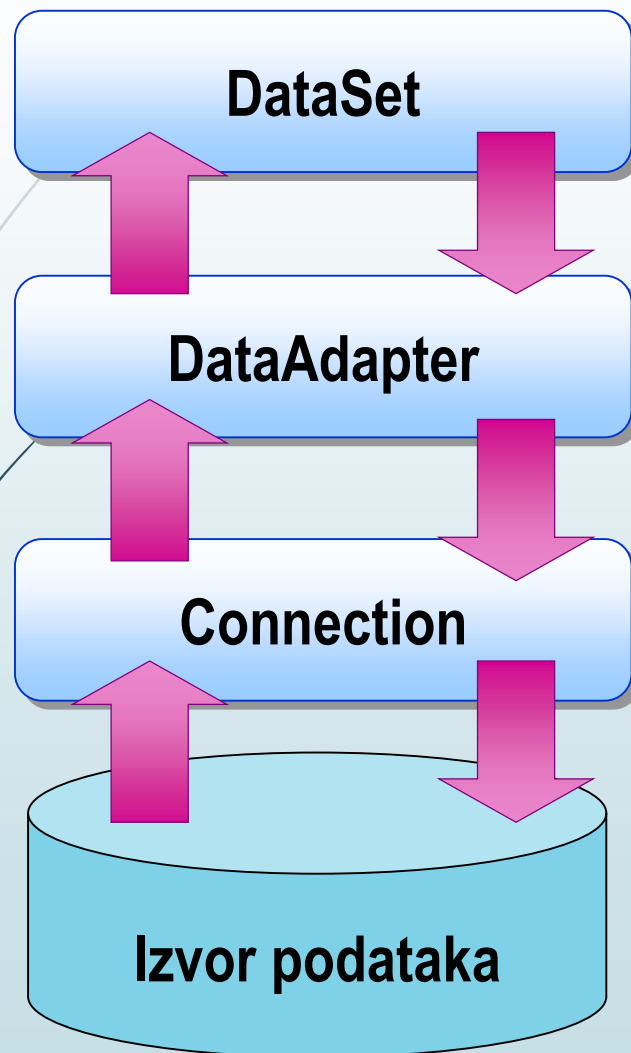
Pozivi pohranjenih procedura

24

- 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
- Primjer:  `DataAccess \ Procedure \ Program.cs`

```
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



- Podaci se obrađuju lokalno, DataSet reprezentira stvarne podatke pohranjene u memoriju

1. Otvori priključak
2. Napuni DataSet
3. Zatvori priključak
4. Obradi DataSet
5. Otvori priključak
6. Ažuriraj izvor podataka
7. Zatvori priključak

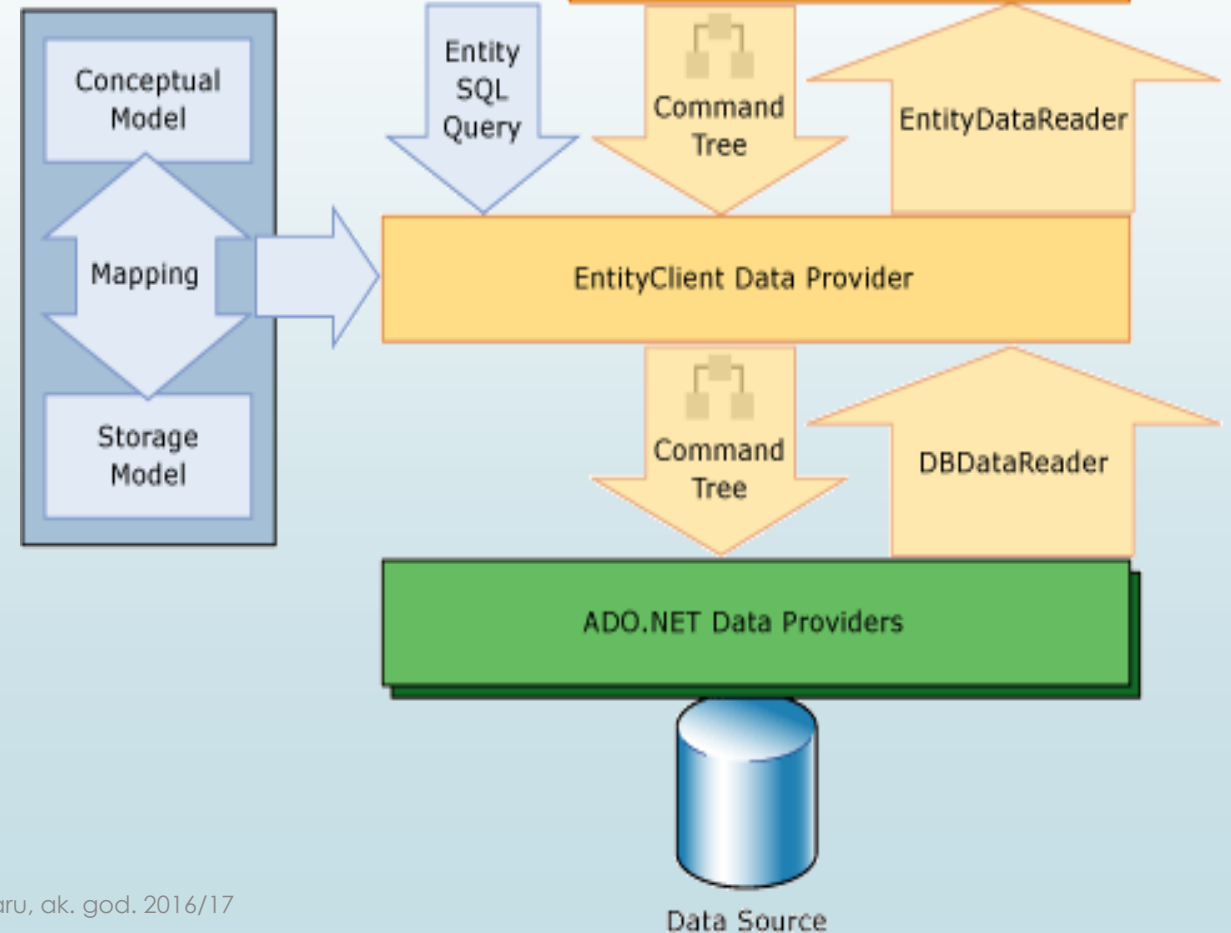
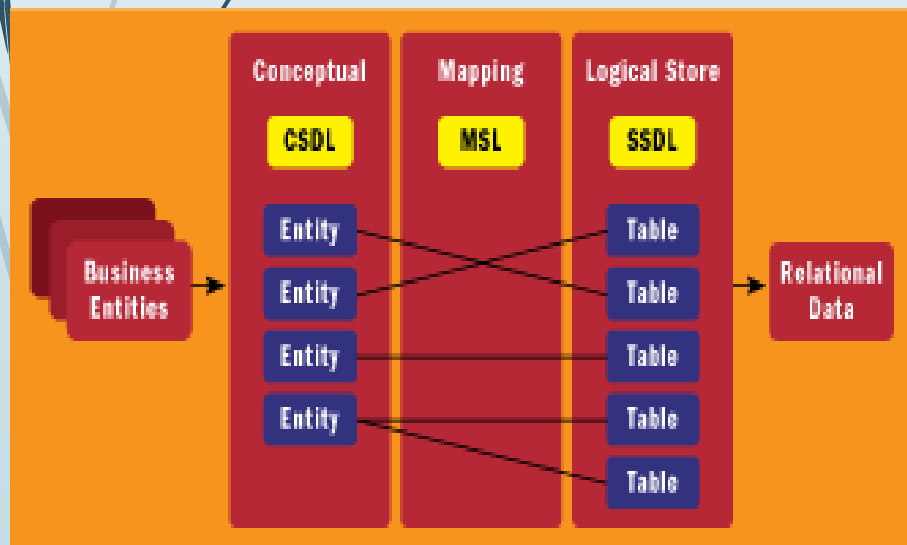
- .NET sadržavao razred DataSet koji je bio preslika relacijske baze podataka (nije ORM)

- Ideja lokalne obrade podataka DataSetom „prenesena” na EntityFramework

Inicijalna ideja Entity Frameworka

26

- Nadgradnja nad ADO.NET-om
- Rad s BP na višoj razini putem objektnog modela
 - preslikavanje između modela i podataka u BP
- Evidentiranje promjena
- Automatsko stvaranje odgovarajućih SQL upita



Entity Framework Core

Načini kreiranja EF modela

28

- *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 na osnovu napisanih razreda. Izgled BP 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 ne uzrokuje stvaranje nove baze podataka.
 - Razredi se mogu stvoriti ručno ili nekim od generatora.
- .NET Core, odnosno *Entity Framework Core* podržava samo *Code First* varijante
 - U primjerima koristimo *Code First from existing database*

Stvaranje modela na osnovu postojeće BP (1)

29

1. U mapi ciljanog projekta izvršiti sljedeće naredbe

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

➡ ili dodati koristeći opciju Manage NuGet Packages

2. U csproj datoteku pod ItemGroup dodati:

```
<DotNetCliToolReference  
  Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="1.0.0"  
>
```

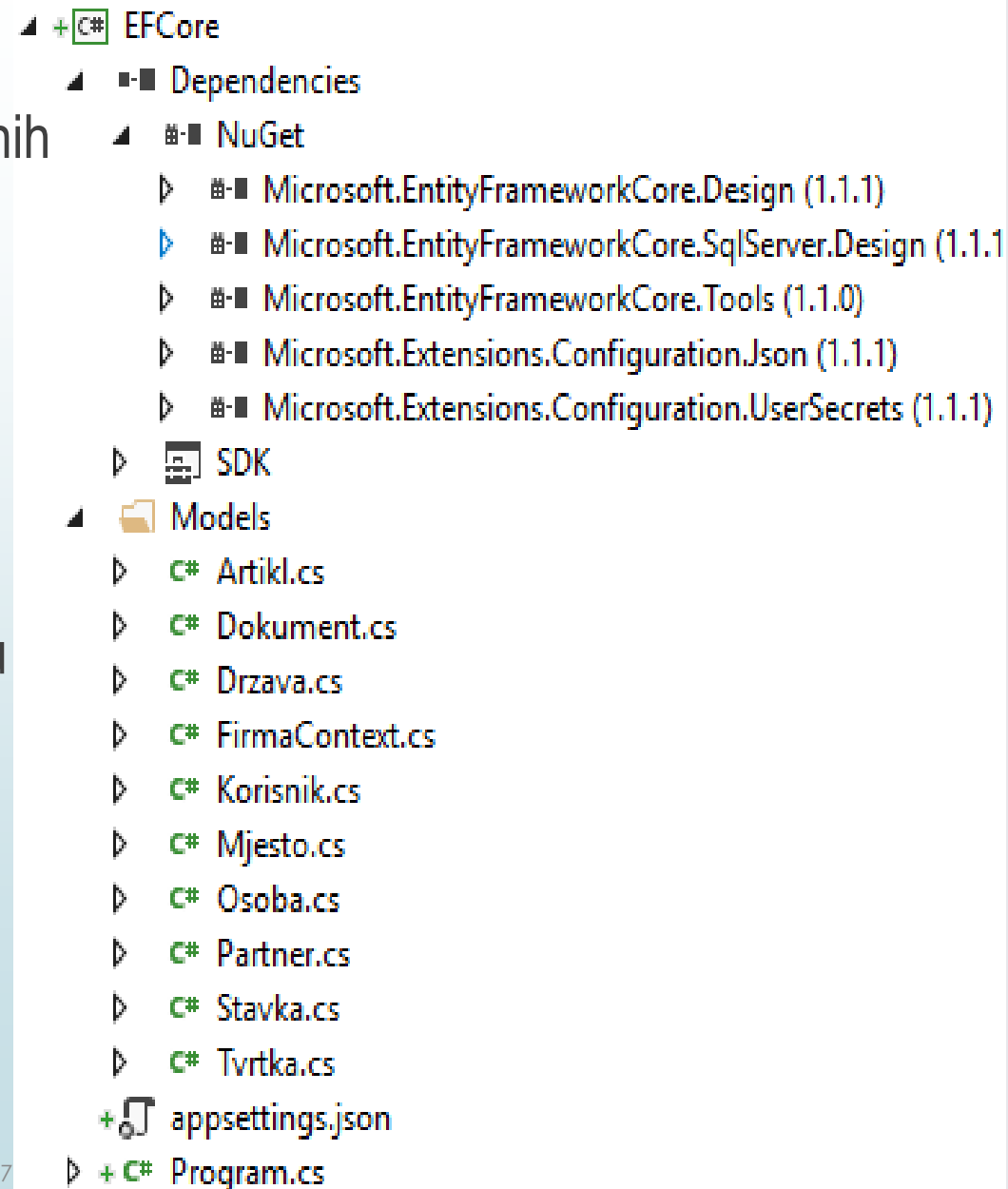
3. U naredbenom retku izvršiti

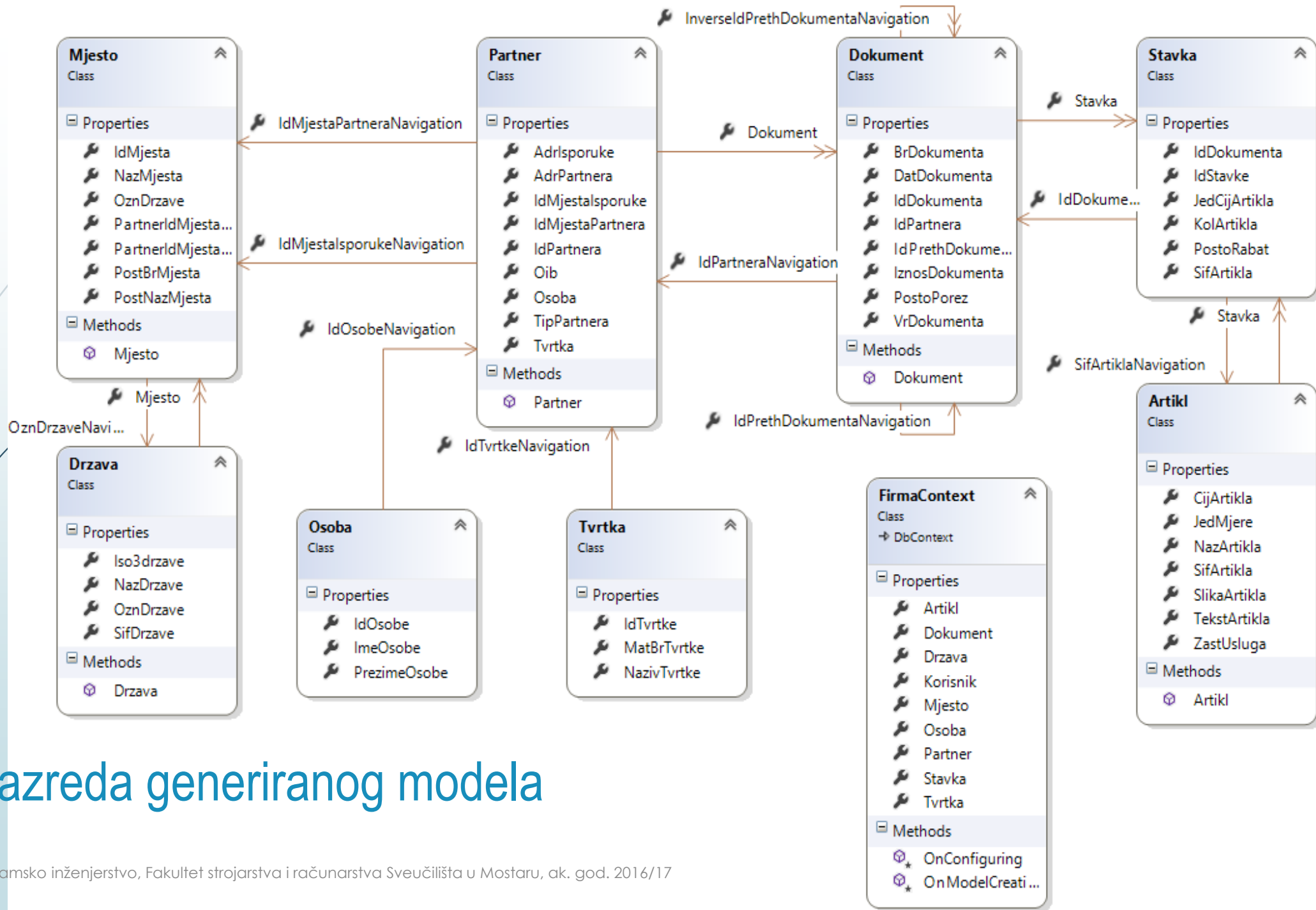
```
dotnet restore  
  
dotnet ef dbcontext scaffold  
"Server=rppp.fer.hr,3000;Database=Firma;User Id=rppp;Password=sifra"  
Microsoft.EntityFrameworkCore.SqlServer -o Models
```

Stvaranje modela na osnovu postojeće BP (2)

30

- Na osnovu postojećih stranih ključeva EF automatski stvara asocijacije između stvorenih razreda
- Za primjer s oglednom bazom podataka 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**





Dijagram razreda generiranog modela

Postavke spajanja na BP korištenjem EF-a

32

- Generirani model inicijalno 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=...password=*");  
}
```

- Može se zamijeniti odsječkom koji bi se dohvatile postavke iz konfiguracijske datoteke


- Nije dobro rješenje (model određuje naziv konfiguracijske datoteke i naziv ključa), ali trenutno bolje od postojećeg – bit će unaprijeđeno naknadno

- Primjer  DataAccess \ EFCore\ Models \ FirmaContext.cs

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {  
    var config = new ConfigurationBuilder().AddUserSecrets("Firma")  
        .SetBasePath(Directory.GetCurrentDirectory())  
        .AddJsonFile("appsettings.json") .Build();  
    string connString = config["ConnectionStrings:Firma"];  
    connString = connString.Replace("sifra", config["FirmaSqlPassword"]);  
    optionsBuilder.UseSqlServer(connString);  
}
```


Elementi EF modela (2)

33

- *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 \ EFCore \ Models \ 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
 - Parcijalni razredi moraju biti definirani unutar istog prostora imena (npr. namespace `EFCore.Models`)

Preslikavanje između EF modela i BP (1)

34

► Primjer:  DataAccess \ EFCore \ Models \ 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)

35

- 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 i postupci razreda *DbContext* i *DbSet*

36

➤ *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)

➤ *DbSet<T>*

- *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

37

➤ Primjer:  DataAccess \ EFCore \ Program.cs - DodajArtikl

➤ 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()) {  
    Artikl artikl = new Artikl{ // (1)  
        SifArtikla = 12345678, CijArtikla = 10m,  
        JedMjere = "kom", NazArtikla = "Burek sa sirom"  
    };  
    context.Artikl.Add(artikl);  
    context.SaveChanges(); // pohrani sve promjene  
}
```

Ažuriranje postojećeg zapisa

38

➤ Dohvatiti entitet

- korištenjem postupka *Find[Async]* 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


➤ Pohraniti promjene u kontekstu

➤ Primjer: DataAccess \ EFCore \ Program.cs - IzmijeniCijenuArtikla

```
using (var context = new FirmaContext()) {  
    Artikl artikl = context.Artikl.Find(12345678) ;  
    //moglo je i context.Find<Artikl>(sifraArtikla) ;  
    artikl.CijArtikla = 11m;  
    context.SaveChanges() ;  
}
```

Brisanje zapisa


39

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

```
using (var context = new FirmaContext())  
{  
    Artikl artikl = context.Artikl.Find(12345678);  
    context.Artikl.Remove(artikl);  
    //ili context.Entry(artikl).State = EntityState.Deleted;  
    context.SaveChanges();  
}
```

Upiti nad EF modelom

40

- 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 dohvatom prvog podataka ili eksplicitnim pozivom postupka *Load*
 - 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)
-  DataAccess \ EFCore \ Program.cs – DohvatiNajskuplje
 - Primjer upita za dohvat prvih n najskupljih artikala

```
var upit = context.Artikl.Include(a => a.Stavka)
                        .AsNoTracking()
                        .OrderByDescending(a => a.CijArtikla)
                        .Take(broj);

foreach (Artikl artikl in upit) { ... }
```


SQL nastao upitom kroz EF

41

➡ 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

42

- 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
-  DataAccess \ EFCore \ Program.cs – DohvatiNajskuplje_v2

```
var upit = context.Artikl
    .OrderByDescending(a => a.CijArtikla)
    .Select(a => new
        {
            a.NazArtikla, a.CijArtikla,
            BrojProdanih = a.Stavka.Count
        })
    .Take(n);

foreach (var artikl in upit)
    Console.WriteLine("...", artikl.NazArtikla,
        artikl.CijArtikla, artikl.BrojProdanih);
```

SQL nastao modificiranim upitom

43

➡ 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

44

- 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
 - Može se pozvati procedura koja za rezultat ima neki od postojećih entiteta
- 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


Poboljšanje rada s postavkama za spajanje na BP iz EF (1)

45

- Prethodno tvrdo kodirane postavke za spajanje na BP zamijenjen s odsječkom koji učitava postavke iz konfiguracijske datoteke
- Mane:
 - Modelom se propisuje da konfiguracijska datoteka mora biti upravo appsettings.json
 - Mora se koristiti isključivo navedeni ključ
- Alternativno može se definirati da konstruktor konteksta prima postavke
- Mana: Prilikom svakog poziva mora se poslati navedeni argument i napisati kod za dohvat vrijednosti iz konfiguracijske datoteke

Poboljšanje rada s postavkama za spajanje na BP iz EF (2)

46

- ➡ (Naoko zasad ne tako dobra) ideja:
- ➡ Definirati sučelje s metodom koja vraća postavke te u konstruktoru primiti primjerak sučelja
 - ➡  DataAccess \ EFCore_DI \ Models \ FirmaContext.cs

```
private IConnectionStringTool tool;

public FirmaContext(IConnectionStringTool tool) {
    this.connectionStringTool = tool;
}

override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {
    optionsBuilder.UseSqlServer(tool.GetConnectionString());
}
```

Poboljšanje rada s postavkama za spajanje na BP iz EF (3)

47

➡ Implementacija sučelja vrši dohvat podataka iz konfiguracijske datoteke

➡  DataAccess \ EFCore_DI \ ConnectionStringTool.cs

```
public class ConnectionStringTool : IConnectionStringTool {  
    private string connectionString;  
    public ConnectionStringTool() {  
        var config = new ConfigurationBuilder().AddUserSecrets("Firma")  
            .SetBasePath(Directory.GetCurrentDirectory())  
            .AddJsonFile("appsettings.json")  
            .Build();  
        connectionString = config["ConnectionStrings:Firma"];  
        connectionString = connectionString.Replace(  
            "sifra", config["FirmaSqlPassword"]);  
    }  
    public string GetConnectionString() { //metoda sučelja  
        return connectionString;  
    }  
}
```


Poboljšanje rada s postavkama za spajanje na BP iz EF (4)

48

- Ovakvo rješenje zahtjeva da se prilikom svakog instanciranja konteksta pošalje instanca vlastitog razreda
- Naoko izgleda kao korak nazad u odnosu na varijantu u kojoj se šalje samo *connection string*, ali...
-NET Core ima ugrađenu podršku za *Dependency Injection*
- Umjesto direktnog stvaranja objekta konteksta s *new* koristit će se primjerak sučelja *IServiceProvider*
 - osigurava da se prilikom stvaranja nekog objekta automatski stvore i odgovarajući objekti potrebni u njegovom konstruktoru

.NET Core i Dependency Injection

49

- Primjer  DataAccess \ EFCore_DI \ Program.cs
- Inicijaliziraju se postavke za DI te se postavljaju sljedeće postavke
 - Svaki put kada se korištenjem *Service Providera* stvara objekt koji u konstruktoru kao parametar treba primjerak sučelja *IConnectionStringTool* neka mu se pošalje objekt tipa *ConnectionStringTool*
 - *AddSingleton* označava da će se za cijeli program objekt tipa *ConnectionStringTool* stvoriti samo jednom
 - Ako je potrebno stvoriti objekt tipa *FirmaContext*, neka se svaki put stvori novi (*AddTransient*) objekt tipa *FirmaContext*

```
private static IServiceProvider serviceProvider;  
public static void Main(string[] args) {  
    //Inicijaliziraj postavke za DI  
    serviceProvider = new ServiceCollection()  
        .AddSingleton<IConnectionStringTool,  
            ConnectionStringTool>()  
        .AddTransient<FirmaContext,  
            FirmaContext>()  
        .BuildServiceProvider();  
}
```

.NET Core, DI na primjeru EF konteksta

50

- Primjer  DataAccess \ EFCore_DI \ Program.cs

```
context = serviceProvider.GetService<FirmaContext>()
```

- *Service Provider* treba vratiti objekt koji je pridružen razredu/sučelju *FirmaContext*, a to je upravo *FirmaContext*.
- Postavkama je određeno da to mora biti novi objekt
- Konstruktor razreda *FirmaContext* ima argument tipa *IConnectionStringTool*
- *ServiceProvider* za *IConnectionStringTool* ima pridružen razred *ConnectionStringTool* i to s opcijom *singleton*.
 - Ako objekt još nije stvoren, stvara se
 - Postojeći objekt tipa *IConnectionStringTool* se predaje konstruktoru razreda *FirmaContext*
- Navedeno rješenja nam omogućava da jednostavno zamijenimo način pohrane i dohvata postavki za spajanje na BP, a da pritom naš kontekst ovisi samo o jednoj metodi sučelja
 - Intenzivno se koristi unutar ASP.NET Core-a

➤ Overview of ADO.NET

➤ <http://msdn.microsoft.com/en-us/library/h43ks021.aspx>

➤ Entity Framework Core

➤ <https://docs.microsoft.com/en-us/ef/core/index>

➤ „Tajne” vrijednosti (user/app secrets)

➤ <https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets>

➤ .NET Core i Dependency Injection

➤ <https://andrewlock.net/using-dependency-injection-in-a-net-core-console-application/>