

1

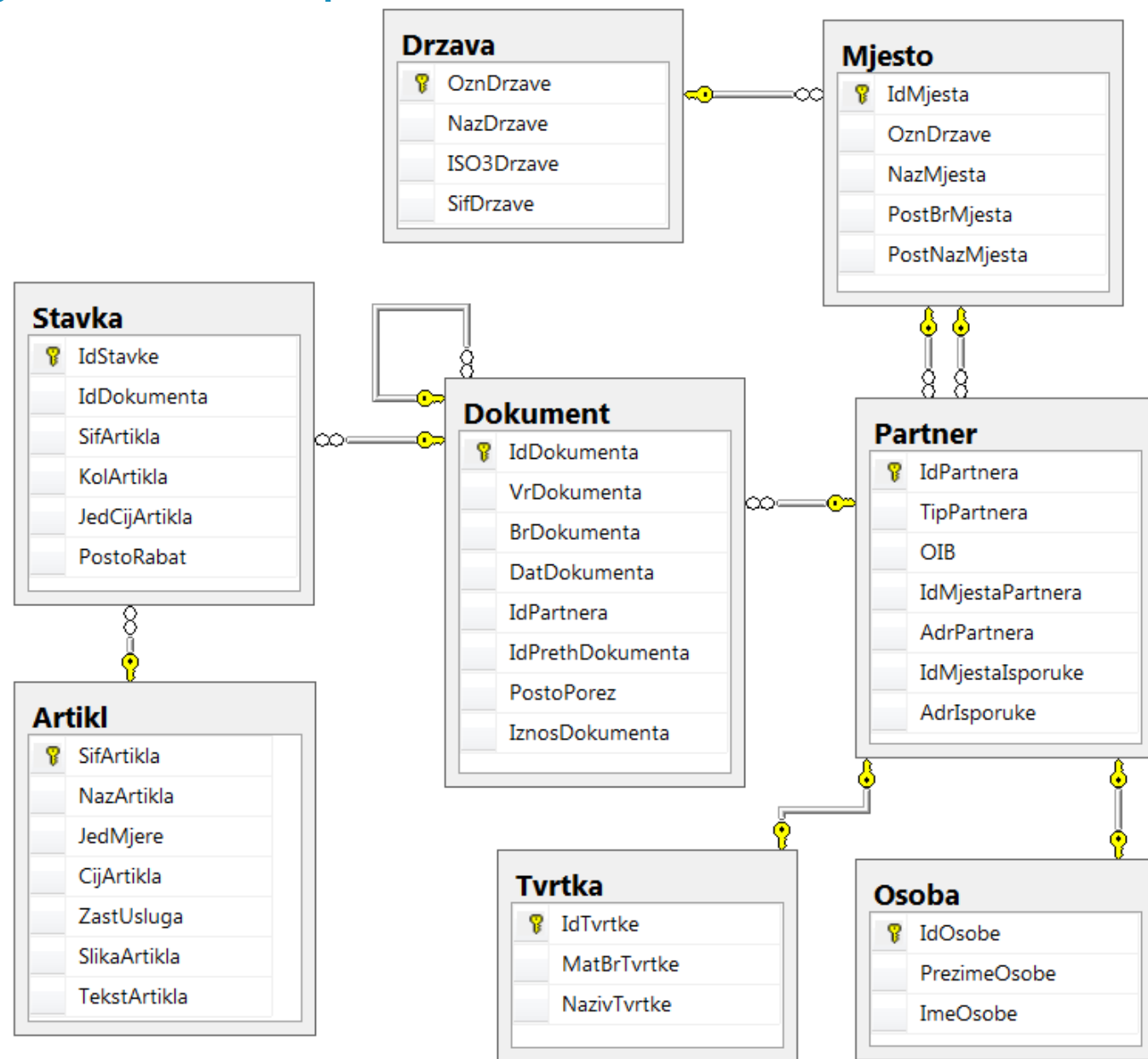
Rad s bazom podataka

2020/21.06

Ogledna baza podataka

2

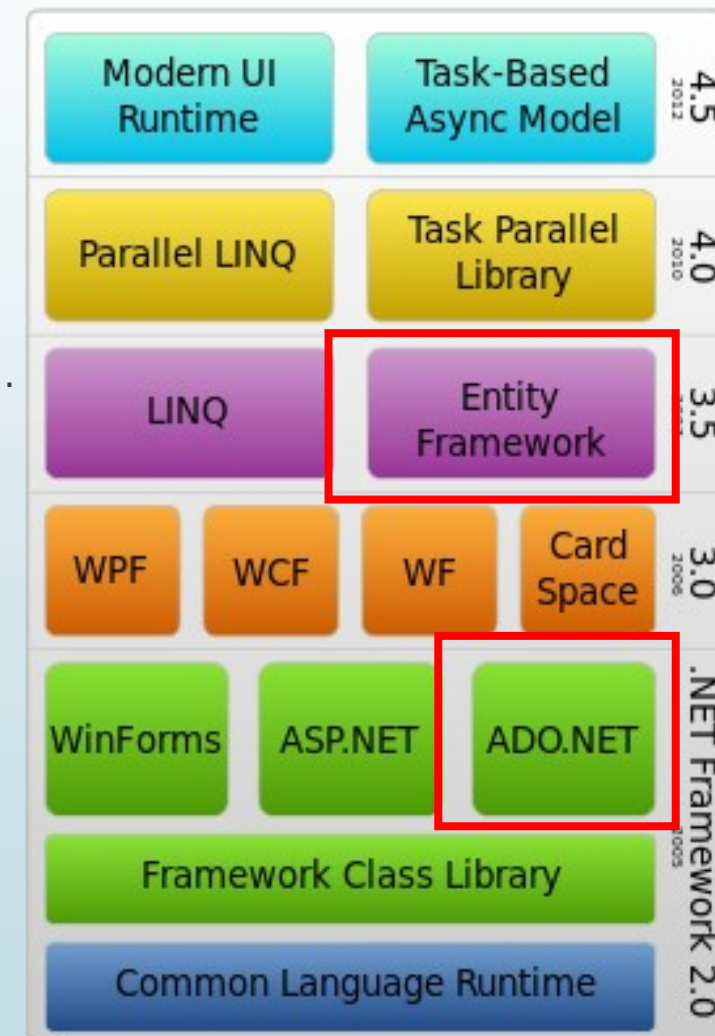
- SQL Server: rppp.fer.hr,3000
- Baza podataka: Firma
 - SQL Server Authentication: rppp/lozinka se nalazi u popisima u privatnom repozitoriju
 - Moguće mijenjati podatke u svrhu testiranja
- BazePodataka.zip :
 - Firma.bak
 - backup baze podataka Firma
 - Firma.vst
 - MS Visio dijagram baze podataka



.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**



Opskrbljivači (davatelji) podataka

4

- Davatelji za različite tehnologije (SQL Server, PostgreSQL, SQLite, MongoDB, ...)
 - <https://devblogs.microsoft.com/dotnet/net-core-data-access/>
 - direktni pristup ili tehnologije s određenom razinom apstrakcije (npr. ORM alati)
<https://docs.microsoft.com/hr-hr/ef/core/providers/?tabs=dotnet-core-cli>
- System.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

5

- **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

6

- 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

7

➤ Microsoft SQL Server

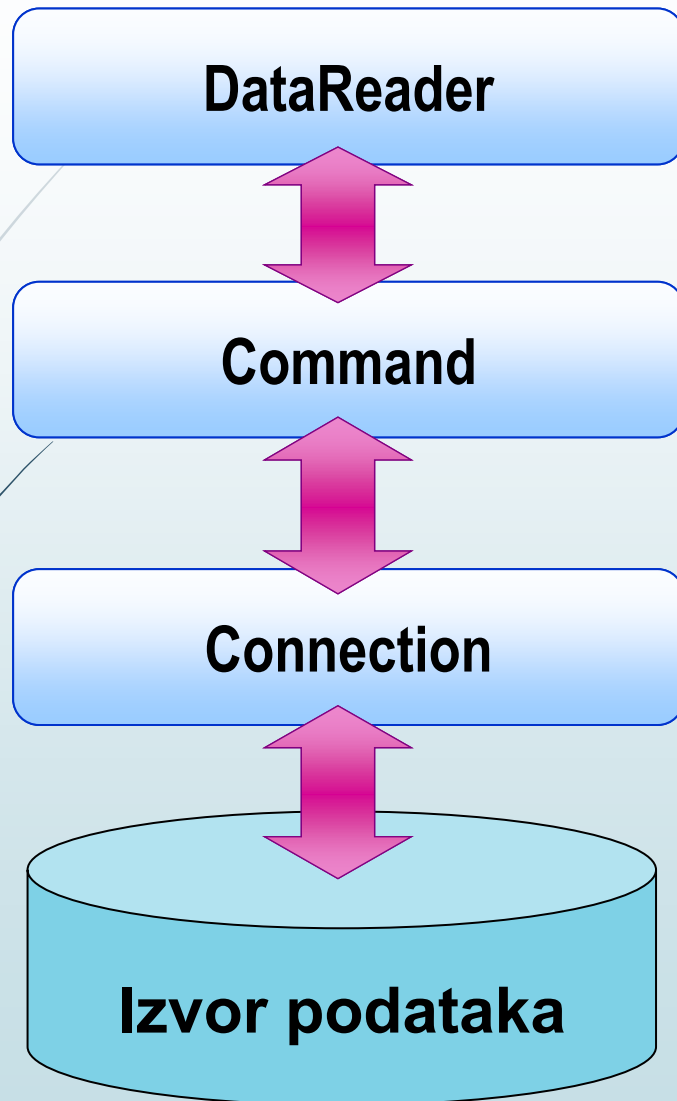
- `Data Source=.;Initial Catalog=Firma;Integrated Security=True`
- `Data Source=rppp.fer.hr,3000;Initial Catalog=Firma;UserId=rppp;Password=šifra`

➤ PostgreSQL

- `User ID=rppp;Password=**;Host=localhost;Port=5432;Database=firma;Pooling=true;`
- Više primjera na <https://www.connectionstrings.com>

Izravna obrada podataka na poslužitelju

8



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

9

- U primjeru koji slijedi 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.Data.SqlClient
 - Microsoft.Extensions.Configuration.Json
 - Microsoft.Extensions.Configuration.UserSecrets



Microsoft.Data.SqlClient by Microsoft

v2.1.2

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.Json by Microsoft

v5.0.0

JSON configuration provider implementation for Microsoft.Extensions.Configuration. When using NuGet 3.x this package requires at least version 3.4.



Microsoft.Extensions.Configuration.UserSecrets by Microsoft

v5.0.0

User secrets configuration provider implementation for Microsoft.Extensions.Configuration. When using NuGet 3.x this package requires at least version 3.4.

Skica rješenja izravne obrade podataka


10

► Primjer:  DataAccess / DataReader / Program.cs : ShowProducts

```
string connString = ...;
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

11

- 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 \ Programs.cs - *BetterVersion

```
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

12

- 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

13

- 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

14

- 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

Postavke priključka na bazu podataka (1)


15

- Izbjegavati pisanje postavki priključka unutar koda
 - promjena postavki zahtijeva ponovnu izgradnju programa (nova verzija?)
 - potencijalni sigurnosni problem (npr. ispis u tragu stoga kod iznimke)
- Postavke staviti u konfiguracijsku datoteku
 - U .NET Core-u uobičajeno appsettings.json
 - može i nešto drugo (ne mora biti u JSON formatu)
- Primjer:  DataAccess / DataReader / appsettings.json
 - Desni klik → Copy To Output Directory : Copy if newer

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

Postavke priključka na bazu podataka (2)

16

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

```
var configBuilder = new ConfigurationBuilder()
    .AddJsonFile("appsettings.json");
var config = configBuilder.Build();

//string connString = config["ConnectionStrings:Firma"];
string connString = config.GetConnectionString("Firma");
```


Višestruke konfiguracijske datoteke

17

- Moguće imati više konfiguracijskih datoteka
 - Kasnije navedeni nadjačavaju parametri iz prethodno uključenih
- Često se koristi s varijablama okruženja
 - *Desni klik na projekt → Properties → Debug → Environment var.*

```
var configBuilder = new ConfigurationBuilder()  
    .AddJsonFile("appsettings.json");  
  
var environmentName = Environment.GetEnvironmentVariable("ENVIRONMENT");  
configBuilder = configBuilder  
    .AddJsonFile($"appsettings.{environmentName}.json",  
        optional: true);  
  
var config = configBuilder.Build();
```

Zaštita postavki za spajanje na bazu

18

- 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**
 - Prilikom kontinuirane isporuke osmisлити mehanizam zamjene šifre s pravom šifrom na produkciji

Postavljanje datoteke s „tajnim” vrijednostima

19

- Izmijeniti projektnu datoteku tako da sadrži *UserSecretsId* i dodati odgovarajuće pakete (*Microsoft.Extensions.Configuration.UserSecrets*)

➤ Primjer:  DataAccess / DataReader / DataReader.csproj

```
<Project Sdk="Microsoft.NET.Sdk"> ...  
  <PropertyGroup>  
    <UserSecretsId>Firma</UserSecretsId>  
  </PropertyGroup>
```

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

Dohvat tajnih vrijednosti

20

➡ 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 configBuilder = new ConfigurationBuilder()
    .AddJsonFile("appsettings.json");

if (useSecretFile)
    configBuilder = configBuilder.AddUserSecrets("Firma");
var config = configBuilder.Build();
string connString = config.GetConnectionString("Firma");

if (useSecretFile)
    connString = connString.Replace("sifra",
                                config["FirmaSqlPassword"]);
```

DbProviderFactory / DbProviderFactories

21

➤ .NET Framework:

- Statički postupak GetFactory u razredu DbProviderFactories

➤ .NET Core:

- [*]ClientFactory.Instance

- Ili prethodno registrirati s

DbProviderFactories.RegisterFactory("System.Data.SqlClient", SqlClientFactory.Instance);

➤ Primjer: DataAccess / ParamsAndProc / Program.cs

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

Parametrizirani upiti

22

- 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 \ 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

23

- *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


24

- 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());  
}
```

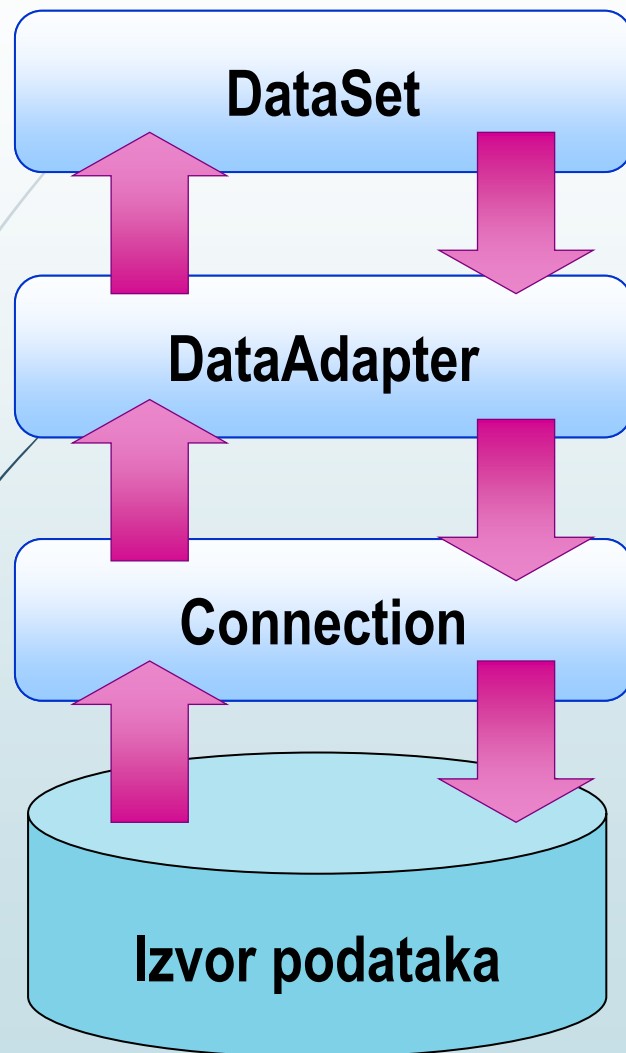

Pozivi pohranjenih procedura

25

- 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



- 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

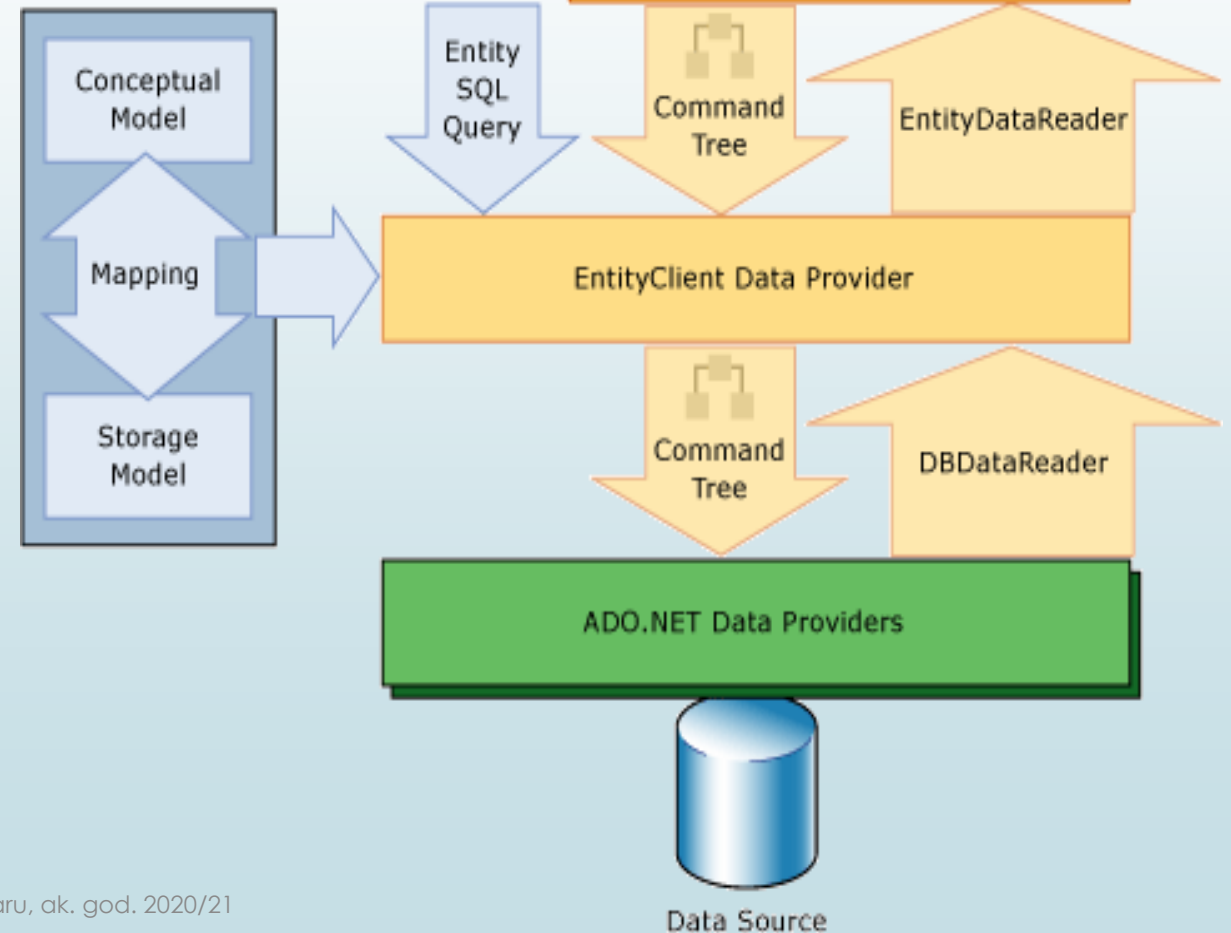
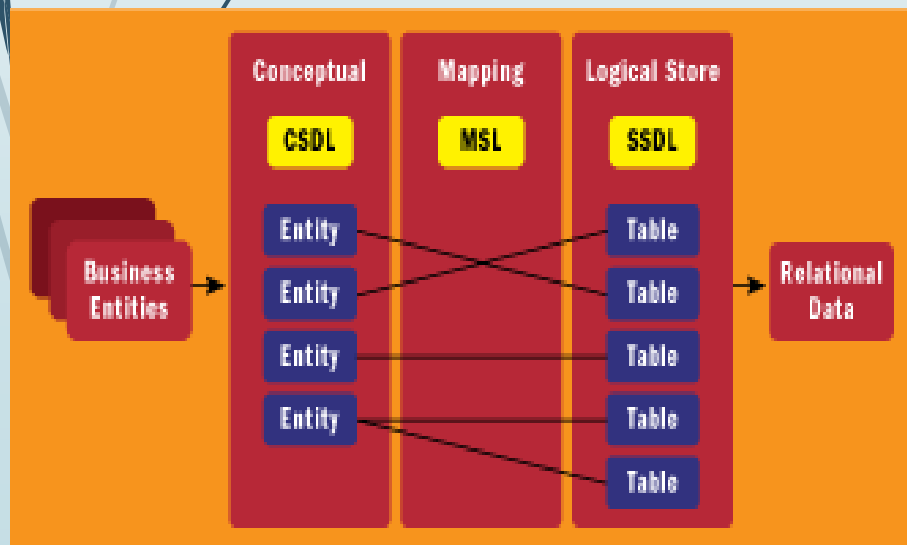
- .NET sadržavao razred DataSet koji je bio preslika relacijske baze podataka (nije ORM)

- Ideja lokalne obrade podataka DataSetom „prenesena” na EntityFramework

Inicijalna ideja Entity Frameworka

27

- 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



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.
- *Code First with Migrations*
 - Izvršava skup radnji (migracija) definiranih u posebnim postupcima (Up/Down)
- .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. 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
```

➡ ili dodati koristeći opciju Manage NuGet Packages

3. U naredbenom retku izvršiti

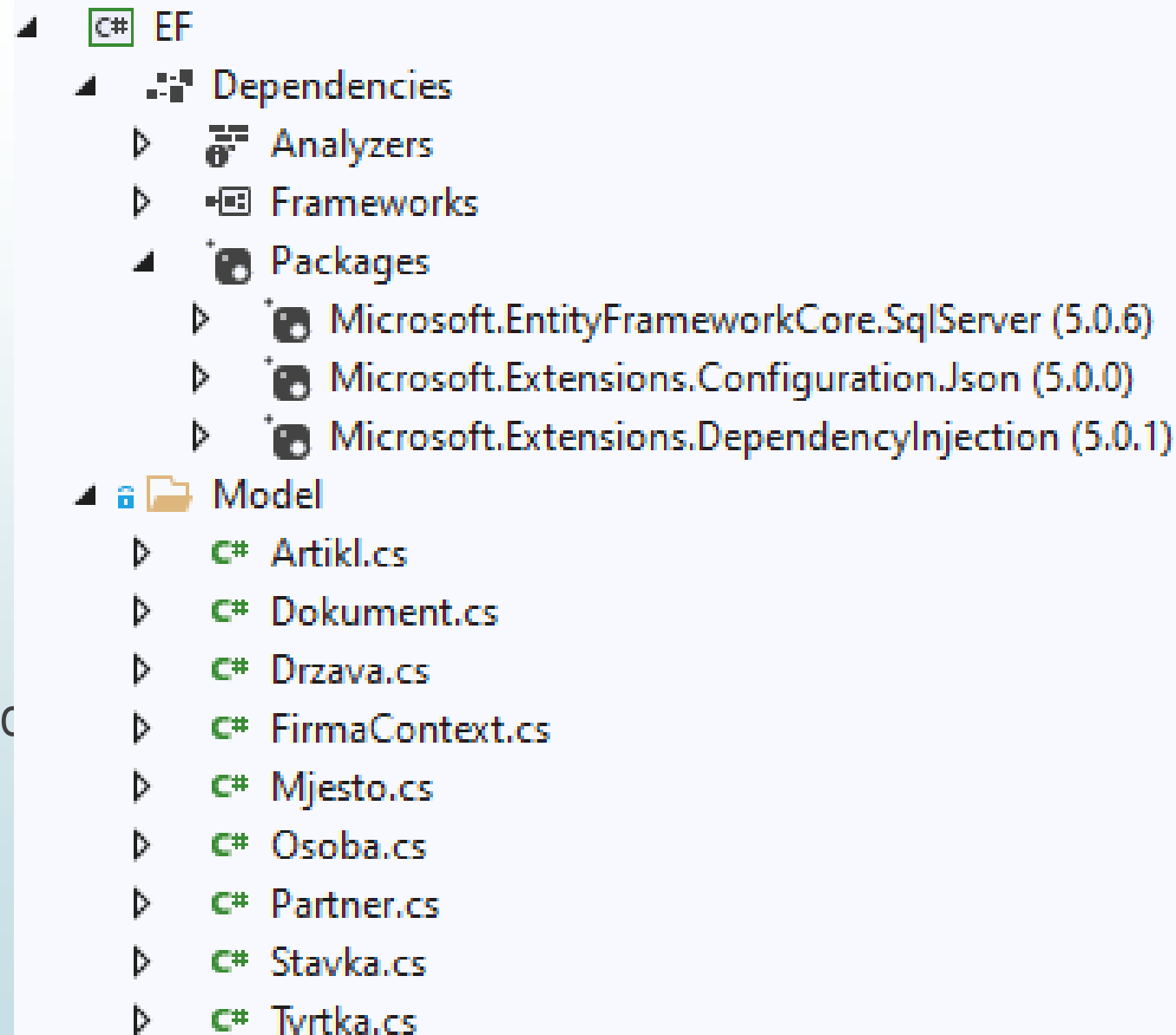
```
dotnet restore
```

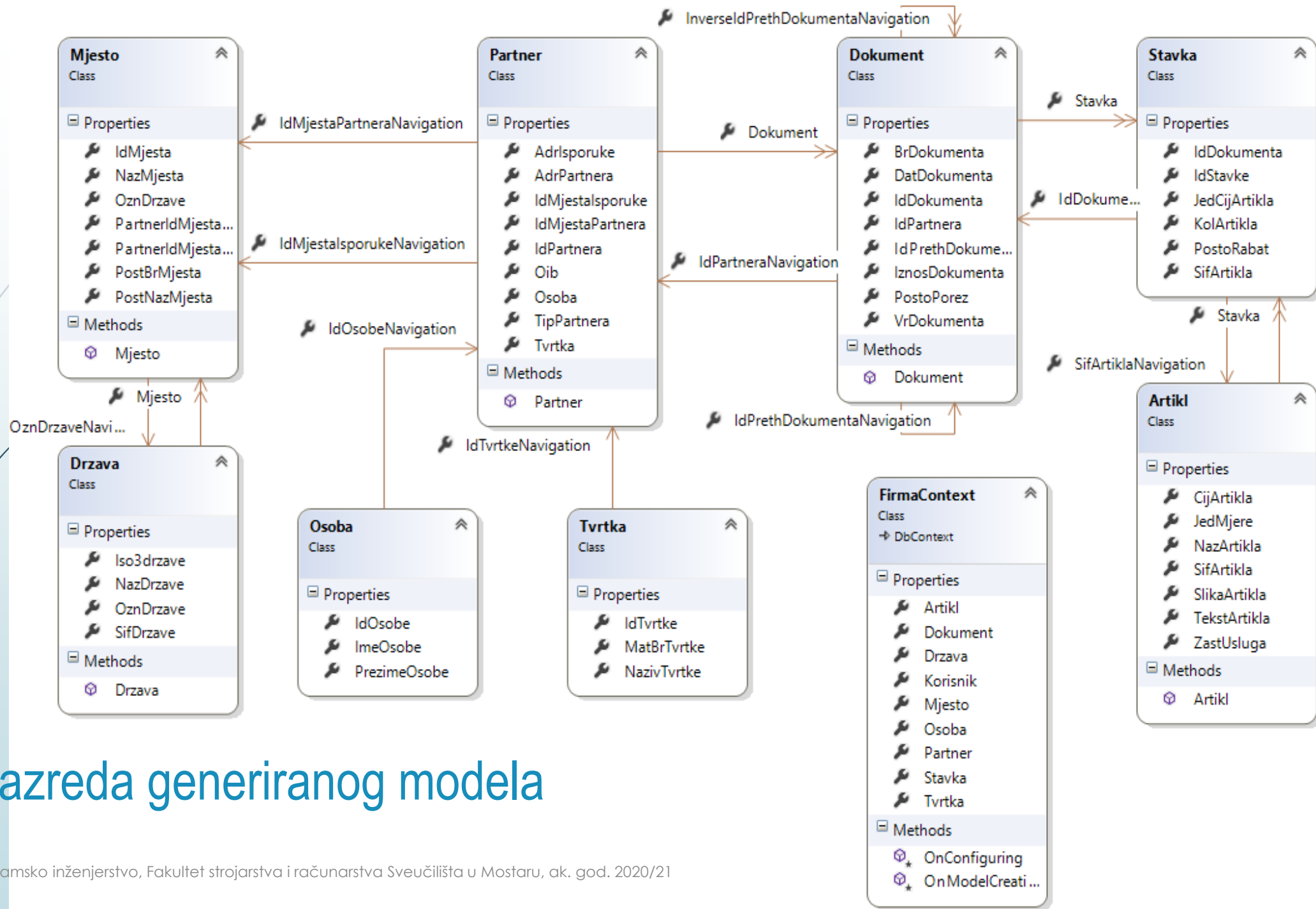
```
dotnet-ef dbcontext scaffold  
"Server=rppp.fer.hr,3000;Database=Firma;User  
Id=rppp;Password=*" Microsoft.EntityFrameworkCore.SqlServer -o  
Model -t Artikal -t Dokument -t Drzava -t Mjesto -t Osoba -t  
Partner -t Stavka -t Tvrtka
```

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 tvrdokodirane u FirmaContext.cs
 - **Potrebno ukloniti i prebaciti u konfiguracijsku datoteku**





Dijagram razreda generiranog modela

Postavke spajanja na BP korištenjem EF-a (1)

32

➡ 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=...password=*");  
}
```

➡ Može se zamijeniti odsječkom za dohvat postavki iz konfiguracijske datoteke.


➡ Nije idealno rješenje, jer model određuje naziv konfiguracijske datoteke i naziv ključa

➡ Bolje rješenje na sljedećem slajdu


```
void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {  
    var config = new ConfigurationBuilder()  
        .AddJsonFile("appsettings.json").Build();  
    string connString = config["ConnectionStrings:Firma"];  
    optionsBuilder.UseSqlServer(connString);  
}
```


Postavke spajanja na BP korištenjem EF-a (2)

33

- 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*
- Upostaviti 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();
```

- *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
 - Strani ključ urokuje asocijaciju sufiksa Navigation
 - Obrnuto stvara se svojstvo tipa *ICollection<T>* (za agregacije)
- „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`)
 - Generirani kontekst sadrži i parcijalne metode

Preslikavanje između EF modela i BP (1)

35

► 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)

36

- 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

37

➤ *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*

38

➤ *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

39

➡ 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

40

➡ 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

41


➡ 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

42

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

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

SQL nastao upitom kroz EF

43

➡ 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

44

- 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

45

➡ 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

46

- 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
 - Može se definirati entitet bez ključa
- 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

➤ Overview of ADO.NET

➤ <https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview>

➤ Entity Framework Core

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

➤ „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/>

➤ Varijable okruženja u .NET konzolnoj aplikaciji

➤ https://medium.com/@craigcampbell_54461/working-with-user-secrets-and-environment-variables-in-net-core-console-applications-f7c3f9ec46bb