Entity Framework: Úvod

Martin Macák

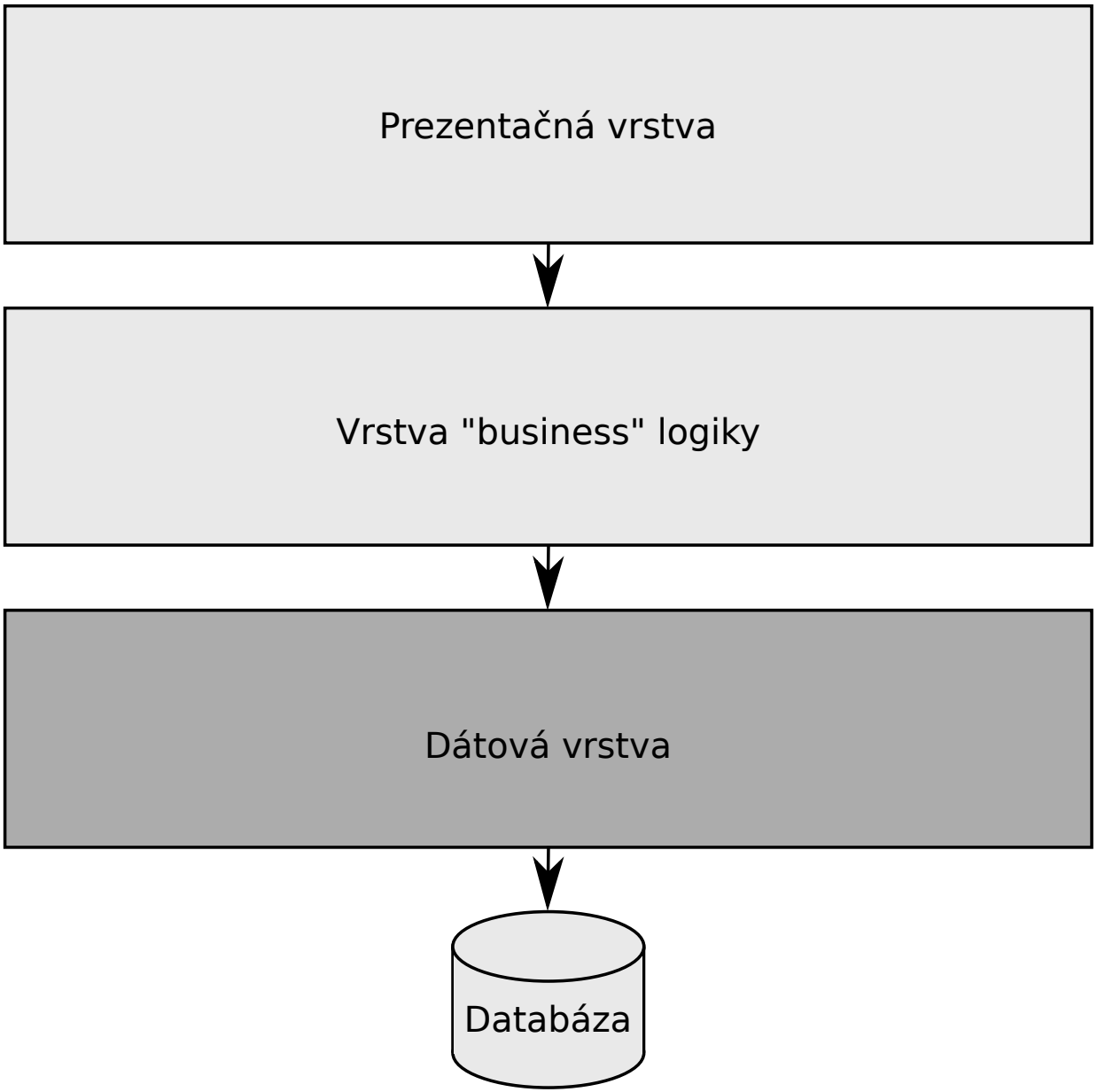
Fakulta informatiky, Masarykova univerzita, Brno

29. 9. 2016

Osnova prednášky

1. Základy Entity Frameworku
2. Návrh databázy (detailnejšie Code First prístup)
3. Migrácie
4. Dotazovanie a modifikácia dát

Vrstvy aplikácie



Vlastná implementácia DAL

- Trieda, čo zapúzdri logiku prístupu k DB
  - pre každú tabuľku *Select*, *Insert*, *Update*, *Delete*, ...
  - využitie **SqlConnection** a **SqlCommand**
- Výhody:
  - plná kontrola nad operáciami
  - nižšia réžia
- Nevýhoda: **veľa roboty**

ORM (Object-Relational Mapping)

- Je to technika na konverziu dát medzi nekompatibilným typovým systémom relačných databáz a objektovo orientovaných programovacích jazykov
- Výhody:
  - podstatne znížené náklady na tvorbu DAL
- Nevýhody:
  - treba dávať pozor na cenu dotazov
  - ťažšia manuálna optimalizácia dotazov
- Príklady .NET ORM frameworkov:
  - LINQ to SQL
  - NHibernate
  - Entity Framework
  - ...

Entity Framework

- Aktuálna verzia 6.1.3
- Microsoft
- https://github.com/aspnet/EntityFramework6
- Mimo jadra .NETu, treba ho nainštalovať cez NuGet
- Entity Framework Core

Práca s Entity Frameworkom

1. Model First prístup
   - nemám databázu a chcem si ju vyklikať v designéry
   - vytvorím model
   - z modelu sa vytvorí DB a vygenerujú triedy na prácu s DB
2. Database First prístup
   - mám databázu a chcem vygenerovať model
   - model sa vygeneruje podľa existujúcej DB
   - z modelu sa vygenerujú triedy na prácu s DB
3. Code First prístup
   - nechcem používať designér
   - vytvorím si triedy na prácu s DB
   - z nich sa vytvorí DB
   - tento prístup sa dá aplikovať aj na existujúcu databázu, na jej základe sa vytvoria priamo triedy

Code First prístup

1. Vytvoríme si triedy reprezentujúce riadky tabuľky – **entity**
2. Vytvoríme si triedu reprezentujúcu **DbContext**
   - vytvára inštancie entít pri načítaní dát z DB
   - zaisťuje, že opakované načítanie riadku vráti rovnakú inštanciu entity (návrhový vzor *Identity Map*)
   - sleduje zmeny v načítaných entitách a následne ich ukladá do DB (návrhový vzor *Unit of Work*)
3. Do nášho DbContextu pridáme pre každú tabuľku triedu **DbSet**, ktorá umožňuje CRUD operácie nad množinou daných entít (návrhový vzor *Repository*)

Vytvorenie DB v Code First

- Cestu pre pripojenie k DB určíme parametrom, ktorý predáme rodičovskému DbContextu v konštruktore
  1. Názov databázy
     - pripojenie **(localdb)\MSSQLLocalDB**, prípadne **.\SQLExpress**
     - názov DB podľa parametru, defaultne *Namespace.ClassName*
  2. Názov kľúča v konfiguračnom súbore, kde je uložený connection string k databáze

PetShopContext

```
public PetShopContext() : base("name=PetShopConnectionString") { }
```

App.config alebo Web.config

```
<connectionStrings>
  <add name="PetShopConnectionString"
       connectionString="Data Source=localhost;Initial Catalog=PetShop;
                         Integrated Security=true"
       providerName="System.Data.SqlClient"/>
</connectionStrings>
```

Inicializácia dát

- Typicky sa vytvára vlastný inicializátor
- Je to trieda, ktorá rozširuje jednu z týchto stratégií:
  1. CreateDatabaseIfNotExists
  2. DropCreateDatabaseIfModelChanges
  3. DropCreateDatabaseAlways
- Prekryjeme metódu *Seed*
- V konštruktore nastavíme náš inicializátor metódou *SetInitializer*

Zabudované konvencie

1. Primárneho kľúča
   - ak má property názov ID alebo MenoTriedyID, tak sa usudzuje, že je primárnym kľúčom
   - ak je primárny kľúč číselného typu alebo GUID automaticky sa generuje
2. Vzťahov medzi entitami
   - ak v entite A uvedieme ako typ property entitu B, automaticky sa vytvorí navigation property a cudzí kľúč (doporučuje sa vždy uvádzať väzbu z oboch strán)
   - ak je v triede B property s rovnakým názvom a typom ako je primárny kľúč v A, usudzuje sa, že je cudzí kľúč
3. Objavovania typov
   - ak má A property typu B a v DbContexte je iba DbSet entity A, tak DbSet entity B sa automaticky zahrnie tiež

Dátové anotácie

- Konvenciám treba niekedy pomôcť
- Dajú sa využiť anotácie nad propertami, ktoré nejak obchádzajú konvencie
- Napríklad:
  - **[Key]**
  - **[ForeignKey("PlayerSupportId")]**
  - **[Required]**
  - **[MaxLength(20)]**
  - **[MinLength(5)]**
  - **[NotMapped]**
  - **[Index]**
  - ...
- Môžete vytvárať aj vlastné anotácie

Vlastné dátové anotácie

```
public class StringLengthRangeAttribute : ValidationAttribute
{
    public int Minimum { get; set; } = 0;
    public int Maximum { get; set; } = int.MaxValue;

    public override bool IsValid(object value)
    {
        var strValue = value as string;
        if (!string.IsNullOrEmpty(strValue))
        {
            return strValue.Length >= Minimum && strValue.Length <= Maximum;
        }
        return true;
    }
}
```

```
[Required]
[StringLengthRange(Minimum = 3, Maximum = 8)]
public string Name { get; set; }
```

Migrácie

- Umožňujú nám aktualizovať databázu bez straty dát
- V projekte môžeme vytvoriť aktuálny obraz dátového modelu
- Každý takýto obraz obsahuje informáciu o tom, ako bolo treba aktualizovať schému DB, aby sa z predchádzajúceho obrazu stal aktuálny. Taktiež obsahuje informáciu o tom, ako sa dá zmeniť schéma DB tak, aby sa zmenila do podoby predchádzajúceho obrazu.

Použitie migrácií

1. Využijeme **Tools** $\rightarrow$ **Library Package Manager** $\rightarrow$ **Package Manager Console**
2. Do konzoly napíšeme **enable-migrations**, čo nám vytvorí iniciálny obraz databázy
3. Po vykonaní zmien v entitách napíšeme do tejto konzoly **add-migration MenoMigracie**
4. Pre ručnú aplikáciu migrácie napíšeme príkaz **update-database** (je možné aktualizovať aj na konkrétnu verziu)

Dotazovanie a modifikácia dát

1. SQL
   - moc sa nepoužíva
   - nevýhody:
     - databázovo závislé
     - žiadna kontrola kompilátorom
2. Entity SQL
   - prekladá sa do SQL podľa použitej DB
   - výhoda: dá sa použiť v hocijakom .NET jazyku
   - nevýhoda: žiadna kontrola kompilátorom
3. LINQ to Entities
   - prekladá sa do Entity SQL
   - výhody:
     - podobný LINQ to Objects
     - IntelliSense
     - kontrola kompilátorom

Stavy entít

- Added: nie je v DB, sledovaná, má sa pridať
- Unchanged : je v DB, sledovaná, nezmenená
- Modified: je v DB, sledovaná, má sa zmeniť
- Deleted: je v DB, sledovaná, má sa zmazať
- Detached: nie je sledovaná

CRUD operácie v LINQ to Entities

- Select: metódy *ToList*, *First*, *Last*, ...
- Insert: metóda *Add* alebo nastaviť stav *Added*
- Update: zmeniť danú property v tom istom DbContexte alebo nastaviť stav *Modified*
- Delete: metóda *Remove* alebo nastaviť stav *Deleted*

Práca s navigation properties

Predpokladajme, že máme entitu A, ktorá má ako navigačnú property List<B> (je s B vo vzťahu 1 : N).

- Insert
  - ak pridávame do databázy entitu B s nastavenou property typu A, pridá sa stav *Added* na entitu B, ale taktiež automaticky aj na entitu A
  - toto nie je vždy to, čo chceme!
- Select
  - pri vrátení entity A bude B štandardne **null**
  - pre získanie B sa využívajú tieto techniky:
    - Eager Loading – vráti všetky A priamo aj so všetkými B (metóda *Include*)
    - Explicit Loading – pre daný objekt A načíta všetky B (metóda *Load*)
    - Lazy Loading – ak sa niekto dotáže na List<B> v danom objekte A, automaticky sa načíta (virtual property)

Práca s navigation properties

Predpokladajme, že máme entitu A, ktorá má ako navigačnú property List<B> (je s B vo vzťahu 1 : N).

- Update
  - ak nechcem entite A zmeniť property B, ale nejakú inú, musím stále načítať aj B pretože by ju inak editovalo!
- Delete
  - ak chcem zmazať entitu A, musím načítať aj entitu B, aby jej mohlo zmeniť cudzí kľúč (prípadne ju zmazať)

Vyhodnotenie dotazu

- Na strane servera – dáta nie sú v pamäti, filtrovanie prebieha v databáze, typicky práca s **IQueryable**
- Na strane klienta – dáta sú v pamäti, filtrovanie prebieha na klientovi, typicky práca s **IEnumerable**

Ďakujem za pozornosť