

1. Entity Framework Core	2
1.1 Entity Framework Core - 01 - Architektura aplikace a frameworku	2
1.1.1 Entity Framework Core - 01 - NuGet balíčky	3
1.2 Entity Framework Core - 02 - Model	4
1.2.1 Entity Framework Core - 02 - Kolekce s filtrováním smazaných záznamů	8
1.3 Entity Framework Core - 03 - Entity	9
1.4 Entity Framework Core - 04 - DataLayer & Patterns	12
1.4.1 Entity Framework Core - 04 - Cachování	12
1.4.2 Entity Framework Core - 04 - Data Sources	17
1.4.3 Entity Framework Core - 04 - DataEntries	18
1.4.4 Entity Framework Core - 04 - DataLoader	19
1.4.5 Entity Framework Core - 04 - Generovaná metadata do modelu	21
1.4.6 Entity Framework Core - 04 - Generátor kódu	22
1.4.7 Entity Framework Core - 04 - Unit of Work (BeforeCommit, AfterCommit, validate)	24
1.4.8 Entity Framework Core - 04 - LookupServices	27
1.4.9 Entity Framework Core - 04 - Lokalizace	29
1.4.10 Entity Framework Core - 04 - Repositories	30
1.5 Entity Framework Core - 05 - Seedování dat	31
1.6 Entity Framework Core - 30 - Práce s entitami reprezentující dekomponovaný vztah ManyToMany (M:N)	37
1.7 Entity Framework Core - 70 - Generování seedů z existující databáze	39
1.8 Entity Framework Core - 80 - Model Extensions	39
1.8.1 Entity Framework Core - 80 - Stored Procedures	41
1.8.2 Entity Framework Core - 80 - Pridanie novej funkcionality	46
1.8.3 Entity Framework Core - 80 - Infrastruktúra Model Extensions	47
1.9 Entity Framework Core - 99 - Použití vlastních služeb z dependency injection	49
1.10 Entity Framework Core - 99 - Soft Deletes - Hledání cesty	51
1.11 Entity Framework Core - 10 - Know-How - Výchozí hodnoty v databázi vs. uložené hodnoty	54
1.12 Entity Framework Core - 10 - Know-How - Výchozí hodnoty v databázi vs. datové typy	55
1.13 Entity Framework Core - instalace dotnet ef	57
1.14 Entity Framework Core - 06 - Registrace do DI containeru	58

Entity Framework Core

- [Business Layer - Database Migrations](#)
 - [Business Layer - Database Migrations - Extended properties](#)
 - [Business Layer - Database Migrations - Konvence modelu](#)
 - [Business Layer - Database Migrations - Model Extensions - Stored Procedures](#)
- [Entity Framework Core - 01 - Architektura aplikace a frameworku](#)
 - [Entity Framework Core - 01 - NuGet balíčky](#)
- [Entity Framework Core - 02 - Model](#)
 - [Entity Framework Core - 02 - Kolekce s filtrováním smazaných záznamů](#)
- [Entity Framework Core - 03 - Entity](#)
- [Entity Framework Core - 04 - DataLayer & Patterns](#)
 - [Entity Framework Core - 04 - Cachování](#)
 - [Entity Framework Core - 04 - DataEntries](#)
 - [Entity Framework Core - 04 - DataLoader](#)
 - [Entity Framework Core - 04 - Data Sources](#)
 - [Entity Framework Core - 04 - Generátor kódu](#)
 - [Entity Framework Core - 04 - Generovaná metadata do modelu](#)
 - [Entity Framework Core - 04 - Lokalizace](#)
 - [Entity Framework Core - 04 - LookupServices](#)
 - [Entity Framework Core - 04 - Repositories](#)
 - [Entity Framework Core - 04 - Unit of Work \(BeforeCommit, AfterCommit, validate\)](#)
- [Entity Framework Core - 05 - Seedování dat](#)
- [Entity Framework Core - 06 - Registrace do DI containeru](#)
- [Entity Framework Core - 10 - Know-How - Výchozí hodnoty v databázi vs. datové typy](#)
- [Entity Framework Core - 10 - Know-How - Výchozí hodnoty v databázi vs. uložené hodnoty](#)
- [Entity Framework Core - 30 - Práce s entitami reprezentující dekomponovaný vztah ManyToMany \(M:N\)](#)
- [Entity Framework Core - 70 - Generování seedů z existující databáze](#)
- [Entity Framework Core - 80 - Model Extensions](#)
 - [Entity Framework Core - 80 - Infrastruktúra Model Extensions](#)
 - [Entity Framework Core - 80 - Pridanie novej funkcionality](#)
 - [Entity Framework Core - 80 - Stored Procedures](#)
- [Entity Framework Core - 99 - Použití vlastních služeb z dependency injection](#)
- [Entity Framework Core - 99 - Soft Deletes - Hledání cesty](#)
- [Entity Framework Core - instalace dotnet ef ...](#)

Entity Framework Core - 01 - Architektura aplikace a frameworku

Aplikační projekty

Projekt	Obvyklé závislosti	Význam	Příklad implementovaných tříd
Model	<ul style="list-style-type: none">• Havit.Model	Modelové třídy projektu (později odpovídají tabulkám v databázi, avšak samy o sobě nic o databázi neví). <i>Assembly je (nepřímým) vstupem pro generátor kódu.</i>	<ul style="list-style-type: none">• LoginAccount
Entity	<ul style="list-style-type: none">• Model• Havit.Data. EntityFrameworkCore• Microsoft. EntityFrameworkCore. SqlServer• Microsoft. EntityFrameworkCore.Tools• (a pár dalších)	Konfigurace EntityFrameworku a mapování modelu do databáze, obsahuje: <ul style="list-style-type: none">• aplikační DbContext• konfigurace entit• code migrations <i>Assembly je vstupem pro generátor kódu.</i>	<ul style="list-style-type: none">• ApplicationDbContext• ApplicationDesignTimeDbContextFactory• LoginAccountConfiguration
DataLayer	<ul style="list-style-type: none">• Havit.Data. EntityFrameworkCore. Patterns• Havit.Data. EntityFrameworkCore. CodeGenerator• Model• Entity	Implementace Havit.Data.Entity.Patterns v projektu, obsahuje: <ul style="list-style-type: none">• implementace repositories,• implementace datasources,• implementace dataentries <i>Obsahuje zejména generovaný kód.</i>	<ul style="list-style-type: none">• ILoginAccountDataSource,• DbLoginAccountDataSource,• FakeLoginAccountDataSource• ILoginAccountRepository,• DbLoginAccountRepository• ILoginAccountDataEntries,• DbLoginAccountDataEntries
Services	<ul style="list-style-type: none">• Model• DataLayer• Havit.Extensions. DependencyInjection. Abstractions	Služby (tj. business logika aplikace) poskytované dalším částem aplikace, např. pro uživatelské rozhraní, webové služby, atp.	<ul style="list-style-type: none">• AuthenticationService• UserStore• ...

Facades	<ul style="list-style-type: none"> • Model • DataLayer • Services • Havit.Data.Patterns 	Služby konzumované pro controllery, WebAPI, atp. Mají usnadnit lepší testovatelnost než logika v controllerech.	<ul style="list-style-type: none"> • LoginAccountFacade
WindsorInstallers	<ul style="list-style-type: none"> • Services • Havit.Data.EntityFrameworkCore.Patterns.Windsor • Castle.Windsor • Havit.Extensions.DependencyInjection.CastleWindsor • Havit.Extensions.DependencyInjection.CastleWindsor.AspNetCore 	Konfigurátor Castle Windsor containeru.	<ul style="list-style-type: none"> • WindsorInstaller
Web(API)	<ul style="list-style-type: none"> • Model • Facades • WindsorInstaller • (a spousta dalších) 	Webové uživatelské rozhraní aplikace / API.	<ul style="list-style-type: none"> • AccountController / ...
EntityTests DataLayerTests ServicesTests ...	<ul style="list-style-type: none"> • DataLayer • Entity • Facades • Model • Services • Microsoft.EntityFrameworkCore.InMemory • (a pár dalších) 	Unit testy testující služby aplikace.	<ul style="list-style-type: none"> • AuthenticationServiceTest • ...
TestsForLocalDebugging	<ul style="list-style-type: none"> • DataLayer • Entity • Facades • Model • Services • WindsorInstallers 	Testy (obvykle integrační) usnadňující vývoj a ladění stavebních bloků aplikace.	

NuGet balíčky

Balíčky jsou určeny pro .NET Standard 2.0.

Assembly (NuGet)	Význam
Havit.Data.EntityFrameworkCore	Rozšíření a customizace Entity Framework Core pro firemní použití (DbContext, apod.)
Havit.Data.EntityFrameworkCore.CodeGenerator	Generátor kódu pro DataLayer.
Havit.Data.EntityFrameworkCore.Patterns	Implementace Havit.Patterns nad Entity Framework Core.
Havit.Data.EntityFrameworkCore.Patterns.Windsor	Konfigurátor Castle Windsor, registruje služby Havit.Data.EntityFrameworkCore.Patterns a další do Castle Windsor containeru.
Havit.Data.Patterns	Definice tříd a interfaces pro "moderní" architekturu aplikace (IRepository, IUnitOfWork). Poznámka: Nuget balíček používá též implementace EF6.
Havit.Model	Bázové třídy (resp. interfaces) pro použití v modelu projektu. Poznámka: Nuget balíček používá též implementace EF6.

Entity Framework Core - 01 - NuGet balíčky

Balíčky jsou určeny pro .NET Standard 2.0.

Assembly (NuGet)	Význam
Havit.Data.EntityFrameworkCore	Rozšíření a customizace Entity Framework Core pro firemní použití (DbContext, apod.)
Havit.Data.EntityFrameworkCore.CodeGenerator	Generátor kódu pro DataLayer.

Havit.Data.EntityFrameworkCore.Patterns	Implementace Havit.Patterns nad Entity Framework Core.
Havit.Data.EntityFrameworkCore.Patterns.Windsor	Konfigurátor Castle Windsor, registruje služby Havit.Data.EntityFrameworkCore.Patterns a další do Castle Windsor containeru.
Havit.Data.Patterns	Definice tříd a interfaces pro "moderní" architekturu aplikace (IRepository, IUnitOfWork). Poznámka: Nuget balíček používá též implementace EF6.
Havit.Model	Bázové třídy (resp. interfaces) pro použití v modelu projektu. Poznámka: Nuget balíček používá též implementace EF6.

Entity Framework Core - 02 - Model

Konvence datového modelu a výchozí chování je velmi dobře popsáno v [dokumentaci EF Core](#), proto nemá význam zde věci z dokumentace opakovat.

Primární klíč

Používáme primární klíč typu int pojmenovaný Id.

```
public int Id { get; set; }
```

Přítomnost a pojmenování primárního klíče je kontrolována unit testem.

Délky stringů

U všech vlastností typu string je nutno uvést jejich maximální délku pomocí atributu `MaxLengthAttribute`. Pokud nemá být délka omezená, použijeme hodnotu `Int32.MaxValue`, resp. hodnotu nezadááme (default). Ze zadaných hodnot jsou vygenerována metadata, např. pro snadné omezení maximální délky textu v `<asp:TextBox />`, apod.

```
[MaxLength(128)]
public string PasswordHash { get; set; }

[MaxLength(8)]
public string PasswordSalt { get; set; }

...

[MaxLength] // pro maximální možnou délku
public string Poznamka { get; set; }
```

Přítomnost atributu `MaxLength` s kladnou hodnotou nebo bez hodnoty je kontrolována unit testem.

Výchozí hodnoty vlastností

Výchozí hodnoty vlastností definujeme v kódu, např.

```
public bool IsActive { get; set; } = true;
```

Reference / cizí klíče

Není-li jiná potřeba, definujeme v páru cizí klíč (vlastnost typu int nesoucí hodnotu cizího klíče) a navigation property (obvykle reference na cílový objekt). Pro pojmenování konvencí ObjektId a Objekt.

Důvodem jsou možnosti pro dotazování či možnosti podpory seedování dat.

```
public Pohlavi Parent { get; set; }
public int ParentId { get; set; }

public Language Language { get; set; }
public int LanguageId { get; set; }
```

Unit test kontroluje, že jsou vlastnosti v páru, tedy že každá navigation property má i foreign key property. Dále kontroluje pojmenování vlastností končících na Id - nepoužíváme ID.

Kolekce One-To-Many (1:N)

Obvykle používáme List<T>, ale striktně předepsáno to není.

Kolekce nepoužíváme v takovém rozsahu, jako v aplikacích založených na Havit Business Layer. Hlavním důvodem je nemožnost NAČÍST do kolekcí jen nesmazané záznamy (QueryFilters není možné smysluplně využít pro soft deletes - viz [issue](#) nebo [Entity Framework Core - 99 - Soft Deletes - Hledání cesty](#)). Význam však má na takové situace, jako je aggregate root (Order + OrderLines, lokalizace, ...), členství uživatelů v rolích, atp., pak je možné mít v modelu dvě kolekce.

Kolekce dáváme jako readonly a inicializujeme v konstruktoru (resp. auto-property initializers)

```
public List<PohlaviLocalization> Localizations { get; } = new
List<PohlaviLocalization>();
```

Kolekce Many-To-Many (M:N)

✖ Entity Framework Core 5.x přináší podporu pro vazby typu M:N (viz [dokumentace](#)), avšak HFW pro práci s kolekcemi nemá podporu.

Vazby typu M:N je doporučeno dekomponovat na vazby 1:N tak, jak jsme byli zvyklí pro EF Core 2.x a 3.x. Ve výchozím chování EF Core je třeba této entitě nakonfigurovat složený primární klíč (pomocí data annotations nelze definovat složený [primární klíč](#)), nám se klíč nastaví sám (pokud není ručně nastaven) konvencí. Pokud je to třeba, nastavíme pouze název databázové tabulky, do které je entita mapována.

Příklad

Pokud má mít User kolekci Roles, musíme zavést entity Membership se dvěma vlastnostmi. User pak bude mít kolekci nikoliv rolí, ale těchto Membershipů.

▼ [Click here to expand...](#)

```
// Model

public class User
{
    public int Id { get; set; }

    public List<Membership> Roles { get; } = new List<Membership>();

    ...
}

public class Role
{
    public int Id { get; set; }

    ...
}

public class Membership
{
    public User User { get; set; }
    public int UserId { get; set; }

    public Role Role { get; set; }
    public int RoleId { get; set; }
}
```

Kolekce s filtrováním smazaných záznamů

Viz [Entity Framework Core - 02 - Kolekce s filtrováním smazaných záznamů](#).

Mazání příznakem

Podpora mazání příznakem je na objektech, které obsahují vlastnost Deleted typu Nullable<DateTime>. Podpora není implementovatelná na dočítání kolekcí modelových objektů, tj. **při načítání kolekcí objektů jsou načítány i smazané objekty**.

```
public DateTime? Deleted { get; set; }
```

Lokalizace

V aplikaci je třeba definovat

- třídu Language implementující Havit.Model.Localizations.ILanguage
- interface ILocalized<TLocalizationEntity> dědící z Havit.Model.Localizations.ILocalized<TLocalizationEntity, Language> pro označení tříd, které jsou lokalizovány,
- interface ILocalization<TLocalizedEntity> dědící z Havit.Model.Localizations.ILocalization<TLocalizedEntity, Language> pro označení tříd lokalizujících základní třídu (předchozí bod)

Datové třídy pak definujeme s těmito interfaces.

```

public class Country : ILocalized<CountryLocalization>
{
    public int Id { get; set; }

    ...

    public List<CountryLocalization> Localizations { get; } = new
List<CountryLocalization>();
}

public class CountryLocalization : ILocalization<Country>
{
    public int Id { get; set; }

    public Country Parent { get; set; }
    public int ParentId { get; set; }

    public Language Language { get; set; }
    public int LanguageId { get; set; }

    public string Name { get; set; }
}

```

Entries / systémové záznamy (aka EnumClass)

Pokud má třída sloužit jako systémový číselník se známými hodnotami, použijeme vnořený veřejný enum Entry s hodnotami. Pokud mají mít záznamy v databázi stejné Id, což je obvyklé, je třeba uvést položkám hodnotu.

Na základě tohoto enumu pak generátor zakládá DataEntries.

TODO Links Více informací v [Entity Framework - 04 - Seedování dat](#) a [Entity Framework - 04 - Seedování Entries](#).

```

public class Role
{
    ...

    public enum Entry
    {
        Administrator = -1,
        CustomerAdministrator = -2,
        BookReader = -3,
        PublisherAdministrator = -4
    }
}

```

Entity Framework Core - 02 - Kolekce s filtrováním smazaných záznamů

Bohužel není možné NAČÍST jen nesmazané záznamy. Můžeme však načíst do paměti všechny záznamy a používat jen ty nesmazané, například vytvořením dvou kolekcí - persistentní (se všemi objekty) a nepersistentní (počítaná, filtruje jen nesmazané záznamy s persistentní kolekcí).

V následujících ukázkách budeme pracovat s třídou Child, kterou lze příznakem označit za smazanou a s třídou Master mající kolekci Children objektů Child.

Pro implementaci potřebujeme zajistit:

- [Použití kolekcí v modelu](#)
- [Mapování vlastností \(kolekcí\) v EF](#)
- [Kolekce filtrující smazané záznamy](#)

Není možné používat v query a v include. V DataLoaderu je to možné použít.

Použití kolekcí v modelu

```
public class Child
{
    public int Id { get; set; }

    public int MasterId { get; set; }
    public Master Master { get; set; }

    public DateTime? Deleted { get; set; }
}

public class Master
{
    public int Id { get; set; }
    public ICollection<Child> Children { get; } // nepersistentní
    public IList<Child> ChildrenWithDeleted { get; } = new
List<Child>(); // persistentní

    public Master()
    {
        // kolekce children je počítanou kolekcí
        Children = new FilteringCollection<Child>
(ChildrenWithDeleted, child => child.Deleted == null);
    }
}
```

Mapování vlastností (kolekcí) v EF


```
public class MasterConfiguration : IEntityTypeConfiguration<Master>
{
    public void Configure(EntityTypeBuilder<Master> builder)
    {
        builder.Ignore(c => c.Children);
        builder.HasMany(c => c.ChildrenWithDeleted);
    }
}
```

Kolekce filtrující smazané záznamy

Viz `Havit.Model.Collections.Generic.FilteringCollection<T>` - [dokumentace](#), [zdrojáky](#). Kolekce je v nuget balíčku `Havit.Model`.

```
public class FilteringCollection<T> : ICollection<T>
{
    private readonly ICollection<T> source;
    private readonly Func<T, bool> filter;

    public FilteringCollection(ICollection<T> source, Func<T, bool>
filter)
    {
        this.source = source;
        this.filter = filter;
    }

    public IEnumerator<T> GetEnumerator()
    {
        return source.Where(filter).GetEnumerator();
    }

    ...
}
```

Entity Framework Core - 03 - Entity

Definuje datový kontext, jeho vlastnosti a migrace.

DbContext

Nevyžadujeme vytvářet vlastnosti typu `DbSet` pro každou evidovanou entitu.

Obvyklá (a doporučená) struktura třídy DbContext

`ProjectNameDbContext` je připraven v `NewProjectTemplate`, nicméně kdyby bylo potřeba ručně:

Důležité je dědit z `Havit.Data.EntityFrameworkCore.DbContext`.

Obvykle se používají dva konstruktory:

- konstruktor přijímající `DbContextOptions` pro běžné použití (produkční běh aplikace)
- bezparametrický konstruktor pro použití v unit testech, proto je `internal`.

```

public class NewProjectTemplateDbContext : Havit.Data.
EntityFrameworkCore.DbContext
{
    internal NewProjectTemplateDbContext()
    {
        // NOOP
    }

    public NewProjectTemplateDbContext(DbContextOptions options) :
base(options)
    {
        // NOOP
    }

    protected override void CustomizeModelCreating(ModelBuilder
modelBuilder)
    {
        base.CustomizeModelCreating(modelBuilder);

        modelBuilder.RegisterModelFromAssembly(typeof(Havit.
NewProjectTemplate.Model.Localizations.Language).Assembly);
        modelBuilder.ApplyConfigurationsFromAssembly(this.
GetType().Assembly);
    }
}

```

ConnectionString

Není žádné výchozí nastavení, jaký connection string bude použit. Vše je řešeno až při použití DbContextu, např. v nastavení ve Castle Windsor instalerech.

Exception handling metod Save[Async]

V případě selhání uložení objektů je vyhozena výjimka DbUpdateException, ta je však "ošklivě formátovaná" a vyžaduje dohledávání, co se vlastně stalo v InnerException.

Proto v případě výskytu DbUpdateException tuto zachytáváme a vyhazujeme novou instanci DbUpdateException s trochu lépe formátovanou zprávou (Message). Původní výjimku DbUpdateException použijeme jako InnerException námi vyhozené výjimky.

DesignTimeDbContextFactory

Viz dokumentace [Design-time DbContext Creation](#).

Využívá jej tooling migrací a code generátor. Pro účely toolingu migrací musí db context používat SqlServer (nebo jinou relační databázi, nelze použít in-memory provider).

Registrace modelu a konfigurací

Abychom nemuseli registrovat entity ručně, je k dispozici extension metoda RegisterModelFromAssembly. Zaregistruje všechny třídy z dané assembly, které nemají žádný z atributů: NotMappedAttribute, ComplexTypeAttribute, OwnedAttribute.

Pro registraci konfigurací je k dispozici extension metoda ApplyConfigurationsFromAssembly.

Conventions

Výchozí konvence

- **ManyToManyEntityKeyDiscoveryConvention**
Konvence nastaví tabulkám, které reprezentují vazbu Many-To-Many složený primární klíč, pokud jej nemají nastaven. Index primárního klíče má sloupce v pořadí, v jakém byly definovány v kódu.
- **DataTypeAttributeConvention**
Pokud je vlastnost třídy modelu označena atributem `DataTypeAttribute` s hodnotou `DataType.Date` pak se použije v databázi datový typ `Date`.
- **CascadeDeleteToRestrictConvention**
Všem cizím klíčům s nastaví `DeleteBehavior` na `Restrict`, čímž zamezí kaskádnímu delete.
- **CacheAttributeToAnnotationConvention**
Hodnoty zadané v `CacheAttribute` předá do anotations.

Volitelné konvence

- **StringPropertiesDefaultValueConvention**
Pro všechny stringové vlastnosti, pokud nemají výchozí hodnotu, se použije výchozí hodnota `String.Empty`. Nastavuje vlastnosti výchozí hodnotu a `ValueGenerated` na `Never` dle [Entity Framework Core - 10 - Know-How - Výchozí hodnoty v databázi vs. uložené hodnoty](#).

Selektivní potlačení konvence

Pokud některá naše konvence nevyhovuje na určitém místě, lze ji potlačit. Dříve (EF Core 2.x) bylo možné je potlačit v konfiguraci entity, to již (EF Core 3.x) možné není, neboť v té době jsou již konvence aplikovány. Jediná šance je znečistit model informací, že se konvence nemá aplikovat.

Potlačení konvence lze vyjádřit umístěním `SuppressConventionAttribute` s uvedením konvence, kterou potlačujeme. Identifikátory konvencí jsou ve třídě `ConventionIdentifiers`. Atribut i třída s identifikátory jsou v nuget balíčku `Havit.Data.EntityFrameworkCore.Abstractions`.

Potlačit lze tyto konvence:

- `StringPropertiesDefaultValueConvention` (na modelové třídě pro všechny vlastnosti třídy, nebo jen na vlastnosti)
- `ManyToManyEntityKeyDiscoveryConvention` (na modelové třídě)

```
[SuppressConvention(ConventionIdentifiers.
ManyToManyEntityKeyDiscoveryConvention)]
public class SomeClass
{
    ...
}

public class OtherClass
{
    ...
    [SuppressConvention(ConventionIdentifiers.
StringPropertiesDefaultValueConvention)]
    public string SomeString { get; set; }
}
```

Konfigurace

Viz dobře napsaná [dokumentace EF Core](#).

Vztah M:N

✖ Entity Framework Core 5.x přináší podporu pro vazby typu M:N (viz [dokumentace](#)), avšak HFW pro práci s kolekcemi nemá podporu.

Příklad řešení v modelu a konfigurace je uveden v sekci [Entity Framework Core - 02 - Model](#).

Migrations

Viz dokumentace [migrations](#).

Spouštění Migrations

Migrations jsou spuštěny na požádání. Obvykle je spouštíme při startu aplikace.

```
context.Database.Migrate();
```

Entity Framework Core - 04 - DataLayer & Patterns

- [Entity Framework Core - 04 - Cachování](#)
- [Entity Framework Core - 04 - Data Sources](#)
- [Entity Framework Core - 04 - DataEntries](#)
- [Entity Framework Core - 04 - DataLoader](#)
- [Entity Framework Core - 04 - Generovaná metadata do modelu](#)
- [Entity Framework Core - 04 - Generátor kódu](#)
- [Entity Framework Core - 04 - Unit of Work \(BeforeCommit, AfterCommit, validate\)](#)
- [Entity Framework Core - 04 - LookupServices](#)
- [Entity Framework Core - 04 - Lokalizace](#)
- [Entity Framework Core - 04 - Repositories](#)

Entity Framework Core - 04 - Cachování

Jak to funguje

Implementace cachování je realizována na úrovni Repositories, DbDataLoader a UnitOfWork:

- `XyDbRepository.GetObject[Async]` - pokud nemá objekt v identity mapě, pokusí se ho najít v cache, pokud není ani v cache, načítá jej z databáze, poté jej uloží do cache
- `XyDbRepository.GetObjects[Async]` - objekty, které nemá v identity mapě se pokusí najít v cache, objekty, které nejsou ani v cache, načítá z databáze a uloží je do cache
- `XyDbRepository.GetAll()` - hledá v cache identifikátory objektů
- `DbDataLoader.Load` - při načítání referencí i kolekcí se pokusí najít objekty v cache, objekty, které nejsou v cache, načítá z databáze a uloží je do cache
- `DbUnitOfWork.Commit[Async]` - invaliduje položky v cache
- `XyEntries.Item` - pod pokličkou volá `XyDbRepository.GetObject()`

Implementaci zajišťuje `IEntityCacheManager` a jeho implementace.

Ve výchozí konfiguraci (viz dále) je použita `EntityCacheManager`, která realizuje cachování se závislostmi:

- `IEntityCacheSupportDecision` - rozhoduje, zda je daná entita cachovaná či nikoliv
- `IEntityCacheKeyGenerator` - definuje, pod jakým klíčem bude entita uložena do cache
- `IEntityCacheOptionsGenerator` - určuje další parametry položky v cache (priorita, sliding expirace)
- `IEntityCacheDependencyManager` - poskytuje klíč pro cache dependencies

❌ Cachování kolekcí

Cachování kolekcí funguje spolehlivě pro objekty, které nepřecházejí mezi různými parenty. Tj. cachování funguje v obvyklých typických scénářích - objekt s lokalizacemi, faktura s řádky faktur, atp.

Cachování kolekcí však nefunguje tam, kde mohou prvky kolekce přecházet mezi různými parenty, např. pokud budu přepínat zaměstnanci jeho nadřízeného zaměstnance a tento nadřízený zaměstnanec má kolekci svých podřízených, pak cachování této kolekce bude vykazovat chyby. Pokud má být v tomto scénáři nadřízený zaměstnanec cachován, nesmíme mu zapnout cachování kolekcí. Nesmíme ani použít "cachování všech entit se sliding expirací", jak je uvedeno níže.

(Důvod: Invalidace cache se provádí po uložení změn. Po uložení změn vidíme jen nový, aktuální stav objektů. Nejsme schopni tedy invalidovat cache pro původního nadřízeného zaměstnance, neboť nevíme, kdo to byl.)

Konfigurace

Výchozí konfigurace

Ve výchozí konfiguraci jsou:

- cachovány entity, které označeny atributem `Havit.Data.EntityFrameworkCore.Abstractions.Attributes.CacheAttribute` (zjednodušeně),
- v atributu lze nastavit prioritu položek v cache, sliding a absolute expiraci,
- v atributu lze zakázat cachování klíčů `GetAll`.
- kolekce jsou cachovány, pokud je cílový typ cachován (umožňuje cachovat entities)

i Je vyžadována registrace závislosti `ICacheService`, kterou HFW neřeší, je třeba ji zaregistrovat do DI containeru samostatně.

Ukázková situace: Reference

```
public class Auto
{
    public int Id { get; set; }
    public Barva Barva { get; set; }
    // ...
}

[Cache]
public class Barva
{
    public int Id { get; set; }
    // ...
}
```

- Barva je cachovaná, Auto nikoliv
- z cache se proto mohou odbavovat např.:
 - `barvaRepository.GetObject(...)`
 - `barvaRepository.GetAll()`
 - `barvaEntries.Black`
 - `dataLoader(auto, a => a.Barva)`

Ukázková situace: Kolekce 1:N

```

[Cache]
public class Stav : ILocalized<StavLocalization>
{
    public int Id { get; set; }
    public List<StavLocalization> Localizations { get; } = new
List<StavLocalization>();
}

[Cache]
public class StavLocalization : ILocalization<Stav>
{
    public int Id { get; set; }

    public Stav Parent { get; set; }
    public int ParentId { get; set; }
    public Language Language { get; set; }
    public int LanguageId { get; set; }

    // ...
}

```

- číselník stavů je cachovaný vč. svých lokalizací
- atribut Cache je třeba uvést na obou třídách, žádná předpoklad, "když X je cachované, tak XLocalization také" není uplatňován
- z cache se proto mohou odbavovat např.:
 - stavRepository.GetObject(...)
 - stavRepository.GetAll()
 - stavEntries.Aktivni
 - (obdobně stavLocalizationRepository, stavLocalizationEntries, avšak nemá valného významu takto použít)
 - dataLoader(stav, s => s.Localization)

Ukázková situace: Dekomponovaný vztah M:N do asociační třídy s kolekcí 1:N

```

public class LoginAccount
{
    public int Id { get; set; }

    public List<Membership> Memberships { get; } = new
List<Membership>();

    // ...
}

[Cache]
public class Membership
{
    public LoginAccount LoginAccount { get; set; }
    public int LoginAccountId { get; set; }

    public Role Role { get; set; }
    public int RoleId { get; set; }
}

[Cache]
public class Role
{
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int Id { get; set; }

    // ...
}

```

- LoginAccount není cachován, třídy Membership a Role jsou cachovány
- z cache se proto mohou odbavovat např.:
 - roleRepository.GetObject(...)
 - roleRepository.GetAll()
 - roleEntries.Administrator
 - dataLoader.Load(loginAccount, la => la.Membership).ThenLoad(m => m.Role)
- cachování Membership je pro daný scénář nutné, ale nemá jiného významu, neboť nemáme pro třídy reprezentující M:N vazbu nepoužíváme repository
- pokud nebude Membership označen jako cachovaný, nebude se loginAccountu cachovat kolekce Memberships

Cachování vypnuto

Pokud je nutné cachování vypnout (např. jednorázově běžící konzolovky, které jen sežerou paměť, ale data v cache nevyužijí), je možné toto řešit extension metodou:

```
// Service Collection
services.WithEntityPatternsInstaller(c => c.ConfigureNoCaching())
...

// Castle Windsor
container.WithEntityPatternsInstaller(c =>
{
    c.GeneralLifestyle = lf => ...;
    c.ConfigureNoCaching();
})
...
```

Není pak potřeba registrovat závislost `ICacheService`.

Cachování všech entit s použitím sliding expirace [Experimental]

Myšlenka: Na chvíli si do cache umístíme cokoliv, s čím pracujeme. Až s tím nebudeme pracovat, vypadne to z cache. Cachujeme tedy vše, ale zároveň všemu omezuje dobu expirace.

Na rozdíl od výchozí konfigurace:

- se neohlíží na `CacheAttribute`, cachováno je vše,
- v atributu nastavená priorita položek v cache, sliding a absolute expirace se respektuje (použije), pokud není uvedena sliding expirace, použije se výchozí.

```
// Service Collection
services.WithEntityPatternsInstaller(c => c.
ConfigureCacheAllEntitiesWithDefaultSlidingExpirationCaching(TimeSpan.
FromMinutes(5)))
...

// Castle Windsor
container.WithEntityPatternsInstaller(c =>
{
    c.GeneralLifestyle = lf => ...;
    c.ConfigureCacheAllEntitiesWithDefaultSlidingExpirationCaching
(TimeSpan.FromMinutes(5));
})
...
```

Cache dependencies

Pokud potřebujeme do cache uložit objekt, který bychom chtěli invalidovat v případě změněny nějakého konkrétního objektu v databázi, případně změny jakéhokoliv objektu v databázi, můžeme objekt do `ICacheService` registrovat se závislostmi.

Klíče závislostí lze získat ze služby `IEntityCacheDependencyManager`:

- `GetSaveCacheDependencyKey` - závislosti jsou vyhozeny při uložení entity daného typu s daným ID (např. pro invalidaci nějaké vlastnosti subjektu, pokud se subjekt změní).
- `GetAllSaveCacheDependencyKey` - závislosti jsou vyhozeny při uložení (a založení a smazání) jakékoliv entity daného typu (např. pro invalidaci součtu částek všech faktur).

Předpokládáme úpravu tohoto interface na základě dalších požadavků.

Invalidace

- invalidace provádí DbUnitOfWork v metodě Commit
- invaliduje se při uložení entity:
 - entita samotná
 - GetAll typu dle entity
 - kolekce, jichž je entita členem
- po invalidaci entity se uložená entita opět uloží do cache (tj. omezí se nutné načtení entity po její změně)
- je myšleno na distribuovanou invalidaci v lokálních caches

Nepodporované scénáře

- Uložení/vyzvednutí z cache
 - Owned Types (cachování entity s owned types není a nebude, při použití Owned Entity Types je třeba úplně vypnout cachování - viz [níže](#).)
 - Vazba 1:1 (v případě potřeby prověříme možnost doimplementování)
- Veškeré obejítí UnitOfWork
 - např. Cascade Deletes

Modely s Owned Entity Types

Problémy, které způsobuje použití Owned Entity Types:

- ChangeTracker sleduje změny na owned types samostatně, v pokud má Person vlastnost pro domácí adresu HomeAddress (owned) typu Address, pak při změně (např.) ulice ChangeTracker vidí změnu v owned entitě Address, nikoliv v Person. To ztěžuje invalidace. (Pozn: Ale ukládá to efektivně, takže musí jít nějak rozumně pospojovat entitu a jí použité owned typy).
- Současná implementace uložení Person do cache neukládá owned entity types, tj. při odbavení položky z cache nebudou hodnoty pro vlastnosti dobře nastaveny.

Entity Framework Core - 04 - Data Sources

Zprostředkovává přístup k datům jako IQueryable. Umožňuje snadné podstrčení dat v testech.

IEntityDataSource, IDataSource<Entity>

Poskytuje dvě vlastnosti: Data a DataIncludingDeleted. Pokud obsahuje třída příznak smazání (soft delete), pak vlastnost Data automaticky odfiltruje příznakem smazané záznamy.

Pro každou entitu vzniká jeden interface pojmenovaný *IEntityDataSource* (např. *ILanguageDataSource*).

EntityDbDataSource

Generované třídy implementující *IEntityDataSource*. Pro každou entitu vzniká jedna třída, třídy jsou pojmenované *EntityDbDataSource* (např. *LanguageDbDataSource*).

Data jsou čerpána z databáze (resp. z IDbContextu a jeho DbSetu).

FakeEntityDataSource

Jedná se rovněž o generované třídy implementující *IEntityDataSource* (rovněž je pro každou entitu jedna třída *FakeEntityDataSource*, např. *FakeLanguageDataSource*), avšak nejsou napojeny na databázi. Třídy jsou dekorovány atributem *FakeAttribute* a jsou vnořeny do namespace *Fakes*.

Data jsou čerpána z kolekce předané v konstruktoru. Určeno pro podstrčení dat v testech tam, kde je použita závislost *IEntityDataSource* (ev. ve frameworku *IDataSource<Entity>*),

Příklad použití *FakeEntityDataSource* v UnitTestu

```
// Arrange

// připravíme data source obsahující zadané záznamy
FakeUserDataSource fakeUserDataSource = new FakeUserDataSource(
    new User { Id = 1, Username = "...", ... },
    new User { Id = 2, Username = "...", ... },
    new User { Id = 3, Username = "...", ... });

// použijeme data source jako závislost v testované třídě
ITestedService service = new TestedService(..., fakeUserDataSource,
...);

// Act
...
```

Proč vůbec řešíme `IEntityDataSource`?

Q: Proč jako závislost nepoužíváme `(I)DbSet`?

A: Protože není dostatečně obecný.

Q: Proč tedy nepoužijeme `IQueryable<TEntity>`? Vždyť ten je přece dostatečně obecný a lze snadno mockovat, ne?

A: Ne. Není pravda, že by `IQueryable<TEntity>` při použití s Entity Frameworkem šel jednoduše mockovat pro všechny scénáře. Pro běžné situace funguje, ale asynchronní operace nelze nad běžně podhozeným `IQueryable` provést. Příklad: Pokud je v testovaném kódu (ve službě) `queryable.ToListAsync()`, nelze jednoduše napsat unit test, který by tento kód mohl volat - pokud jako `queryable` použijeme např. `new List<Entity>(...).AsQueryable()`, je při volání asynchronní operace vyhozena výjimka (*The source IQueryable doesn't implement IEnumerableAsync<...>. Only sources that implement IEnumerableAsync can be used for Entity Framework asynchronous operations.*). `DbDataSource` poskytuje asynchronní operace "z podstaty" (jde o entity frameworkové `IQueryable`), pro možnost podhození dat v testech je zde právě `FakeDataSource`.

Q: A to je vše?

A: Další myšlenkou `IDataSource` je automatické odfiltrování smazaných záznamů (ve vlastnosti `Data`), avšak s možností získat i záznamy smazané (ve vlastnosti `DataIncludingDeleted`).

Entity Framework Core - 04 - DataEntries

`DataEntries` zpřístupňují systémové záznamy v databázi dle `Entries` v modelu.

Příklad vygenerovaného interface pro `DataEntries`

(viz [Entries v Modelu](#))

```
public interface IRoleEntries : IDataEntries
{
    Role Administrator { get; }
    Role BookReader { get; }
    Role CustomerAdministrator { get; }
    Role PublisherAdministrator { get; }
}
```

Implementace vyzvedává objekty z příslušné repository (`IRepository<Entity>`) pomocí metody `GetObject`.

Příklady použití

```
IRoleEntries entries;
...
// máme strong-type k dispozici objekt, který reprezentuje konkrétní
záznam v databázi
bool userIsAdmin = userRoles.Contains(entries.Administrator);
```

```
INastaveniEntries nastaveni;
...
// máme strong-type k dispozici objekt, který reprezentuje nastavení
aplikace
string url = nastaveni.Current.ApplicationUrl
```

Párování záznamů v databázi

Pokud primární klíč cílové tabulky není autoincrement, páruje se Id záznamu s hodnotou enumů (`Role.Id == (int)Role.Entry.Administrator`).

Pokud je primární klíč cílové tabulky autoincrement, páruje se pomocí stringového sloupce Symbol, který je v takovém případě povinný (`Role.Symbol == Role.Entry.Administrator.ToString()`). Párování (Id, Symbol) se pro každou tabulku načítá jen jednou a drží se v paměti.

Entity Framework Core - 04 - DataLoader

Explicitní loader (`IDataloader`) dočítá objekty, které dosud nebyly načteny.

Činnost

- Explicitní loader - dočítá objekty, které dosud nebyly načteny.
- Objekty jedné vlastnosti jsou dočteny jedním dotazem.
- Při dotazování se nepoužívá join přes všechno načítané, vždy jde o dotaz do tabulky, ze které se načítá daná vlastnost (žádné joiny).
- Např. načtení *faktura* => *faktura.Dodavatel.Adresa.Zeme* spustí do databáze 3 dotazy - načtení dodavatelů, načtení adres a načtení zemí. Nevadí, pokud je některá z vlastností po cestě null.
- Spoléhá se na EF Change Tracker, objekty, ke kterým jsou dočítány "závislosti" musí být trackované. Tato podmínka je testována a v případě nesplnění je vyhozena výjimka.
- Objekty musí mít klíč Id typu `Int32`.
- není vyžadována dualita cizího klíče a navigační property (není tedy vyžadována existence obou sloupců `auto.Barva` a `auto.BarvaId`, stačí samotné `auto.Barva`).
- Neprovádí dotazy do databáze pro nově založené objekty (příklad: nově zakládanému a ještě neuloženému uživateli nemůžeme z databáze načítat role, když uživatel v databázi ještě není)
 - Instance kolekcí inicializuje na prázdné (pro `IList<>` a `List<>`), pokud jsou null.

Chování ohledně foreign keys

Při načítání referenci spoléhá na hodnoty cizích klíčů, potažmo jako shadow properties.

Mějme tedy příklad:

```
1 Auto auto = autoRepository.GetObject(1); // nate auto s Id 1, Barva bude null,
BarvaId eknme nap. 2.
auto.BarvaId = 5; // změníme BarvaId na jinou hodnotu
dataLoader.Load(auto, a => a.Barva); // pokusíme se doíst vlastnost Barva
```

Pod pokličkou se provede:

```
dbContext.Set<Barva>().Where(barva => barva.Id == 5).ToList();
```

Čímž se načte barva podle hodnoty cizího klíče objektu v paměti, nikoliv podle databáze (tam je aktuálně BarvaId == 2). Jinými slovy, po načtení bude mít auto přiřazenu do vlastnosti Barva instanci s Id 5.

Tímto chováním se DataLoader v EF Core liší od implementace DataLoader v EF 6.

Metody

- Load, LoadAsync - přijímá jeden objekt, ke kterému jsou dočteny požadované nenačtené vlastnosti
- LoadAll, LoadAllAsync - přijímá kolekci objektu, kterým jsou dočteny požadované nenačtené vlastnosti
- Podporuje fluent API pro načítání dalších objektů (...Load(...).ThenLoad(...).ThenLoad(...)), viz příklady.
- Není podporována syntaxe se .Select(...), která byla v EF6.

Příklady

```
dataLoader.Load(jednoAuto, auto => auto.Vyrobce).ThenLoad(vyrobce =>
    vyrobce.Kategorie);
dataLoader.Load(jednoAuto, auto => auto.Vyrobce.Kategorie); // funguje
také se zetzením
```

```
dataLoader.LoadAll(mnohoAut, auto => auto.Vyrobce.Kategorie); //
mnohoAut = kolekce, pole, ... (IEnumerable<Auto>)
dataLoader.LoadAll(mnohoAut, auto => auto.NahradniDily).ThenLoad
(nahradniDil => nahradniDil.Dodavatel); // načítání objekt v kolekci
```

Kolekce M:N

- ✖ Entity Framework Core 5.x přináší podporu pro vazby typu M:N (viz dokumentace), avšak HFW pro práci s kolekcemi nemá podporu. DataLoader při pokusu o načtení M:N kolekce neřízeně spadne.

Podpora kolekcí s filtrováním smazaných záznamů

- Kolekce s filtrováním smazaných záznamů jsou dataloaderem podporovány.
- Pokud model obsahuje kolekci Xyz (obvykle nepersistentní) a Xyz**IncludingDeleted** (obvykle persistentní), pak je použití kolekce Xyz automaticky nahrazeno načtením kolekce Xyz**IncludingDeleted**.
- Konvence je dána pojmenováním kolekcí (přípona **IncludingDeleted**), žádné další testy vůči konfiguraci EF nejsou prováděny.
- Pokud je dále použito ThenLoad(...), načtou se hodnoty jen nesmazaným záznamům, mají-li se načíst hodnoty i ke smazaným záznamům, je třeba použít kolekci XyzIncludingDeleted, lepší pochopení dá následující příklad.
- Je požadováno, aby filtrovaná kolekce dokázala během práce vrátit nesmazané objekty, např. **toto není podporováno**, neboť získání ChildGroup.Deleted vyvolá NullReferenceException.

Příklad

```

public class Master
{
    public int Id { get; set; }
    public ICollection<Child> Children { get; } // nepersistentní
    public IList<Child> ChildrenIncludingDeleted { get; } = new
List<Child>();// persistentní
    ...
}

dataLoader.Load(master, m => m.Children); // pod poklikou je
transformováno na natení ChildrenIncludingDeleted, nateny jsou proto
všechny Child (v. příznakem smazaných) k danému masteru
dataLoader.Load(master, m => m.Children).ThenLoad(c => c.Boss); //
nateny jsou všechny Children daného masteru, z nich se vyberou jen
nesmazané a k tm se nate vlastnost Boss
dataLoader.Load(master, m => m.ChildrenIncludingDeleted).ThenLoad(c =>
c.Boss); // vlastnost Boss je natená i smazaným Childm

```

DataLoader jako závislost v unit testech

K dispozici je FakeDataLoader, který nic nedělá. Lze tak použít v unit testech, které pracují s daty v paměti a nemají co dočítat.

```

// Arrange

// připravíme data source obsahující zadané záznamy
FakeUserDataSource userDataSource = new FakeUserDataSource(...);

// připravíme fake data loaderu
FakeDataLoader dataLoader = new FakeDataLoader();

// použijeme data loader jako závislost v testované tíď
ITestedService service = new TestedService(..., fakeUserDataSource,
fakeDataLoader, ...);

// Act
...

```

Entity Framework Core - 04 - Generovaná metadata do modelu

Metadata (maximální délky stringů)

Na základě modelu jsou pro všechny stringové vlastnosti generována metadata s definicí jejich maximálních délek dle atributu `MaxLengthAttribute` (viz [Entity Framework Core - 02 - Model](#)). Pro vlastnosti označované jako "maximální možná délka" se použije hodnota `Int32.MaxValue`, byť to není správně (nejde uložit tolik znaků, ale tolik byte). Jiná metadata negenerujeme.

Metadata jsou generována přímo do modelu a jsou určena pro definici maximálních délek např. ve view modelu. Změnou délky textu v modelu, se po přegenerování kódu změní vygenerované konstanty, které změní maximální velikosti viewmodelu...

```
public static class LanguageMetadata
{
    public const int CultureMaxLength = 10;
    public const int NameMaxLength = 200;
    public const int SymbolMaxLength = 50;
    public const int UiCultureMaxLength = 10;
}
```

Entity Framework Core - 04 - Generátor kódu

Implementace tříd popsaných v následujících kapitolách je automaticky generována s umožněním vlastního rozšíření.

Kód je generován pomocí [dotnet tool](#) `Havit.Data.EntityFrameworkCore.CodeGenerator.Tool` (musí být nainstalován), jenž spouští code generátor z NuGet balíčku `Havit.Data.EntityFrameworkCore.CodeGenerator`, který je zamýšlen pro použití v projektu `DataLayer`.

V tento okamžik není generátor jakkoliv konfigurovatelný.

Aktualizace dotnet toolu `Havit.Data.EntityFrameworkCore.CodeGenerator.Tool`

Aktualizace `Havit.Data.EntityFrameworkCore.CodeGenerator.Tool` se očekávají příležitostně, např. když se změní podporovaná verze .NET Core, atp. Změny (aktualizace) `Havit.Data.EntityFrameworkCore.CodeGenerator` jsou podle změn, které potřebujeme udělat do code generátoru.

✖ Tím, že `Havit.Data.EntityFrameworkCore.CodeGenerator.Tool` není "běžným nuget "balíčkem v projektu, ale jde o [dotnet tool](#), nezobrazují se jeho aktualizace v Package Manageru.

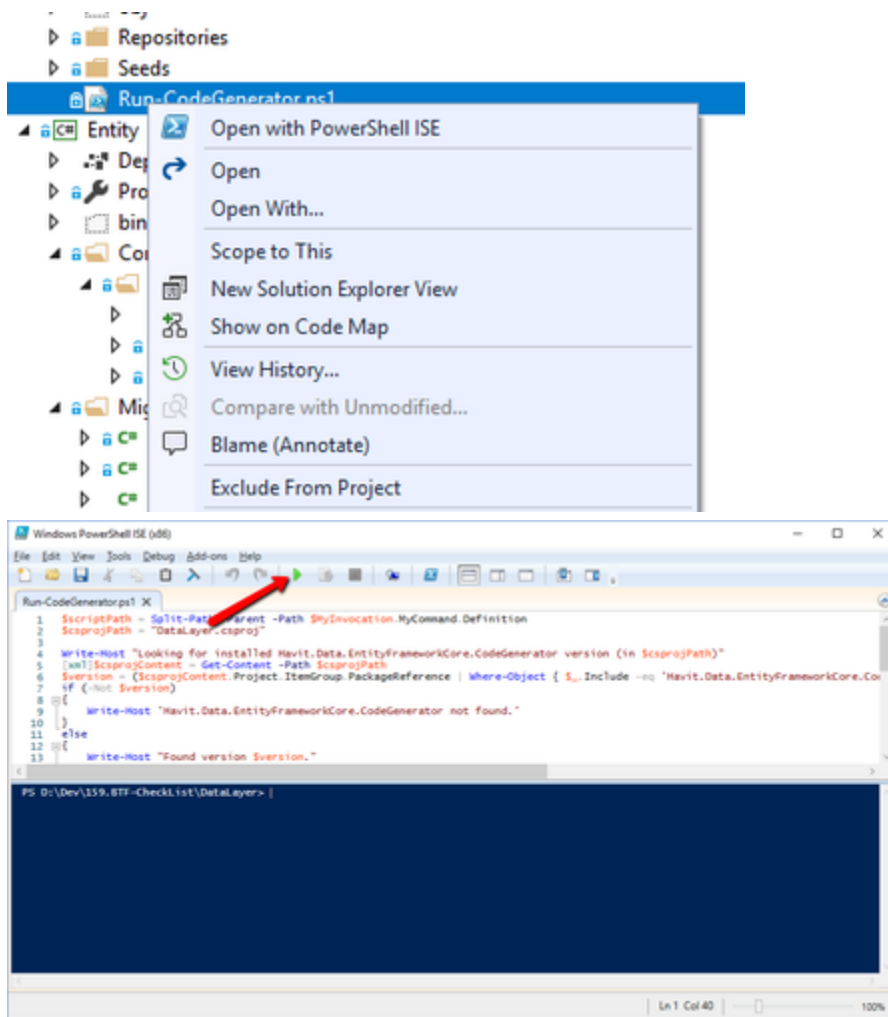
Pro aktualizaci je třeba z příkazové řádky spustit (aktuální složka ve složce se solution):

```
dotnet tool update Havit.Data.EntityFrameworkCore.CodeGenerator.Tool
```

Spuštění generátoru

Generátor lze spustit powershell skriptem `Run-CodeGenerator.ps1` v rootu projektu `DataLayer`. Spustit jej lze přímo z Visual Studio, přes pravé tlačítko na skript v Solution Exploreru a volbě na `Open with Powershell ISE`. Pak spustit přes `Run Script` (F5).

Běh generátoru je (oproti všem našim předchozím variantám) rychlý, generátor je obvykle hotov do jedné sekundy.



Princip generátoru (aneb kde bere generátor data)

Generátor získává data z Modelu DbContextu (`DbContext.Model`). DbContext se hledá v assembly projektu Entity, získává se přes `DbContextActivator`, čímž získáme instanci přes `IDesignTimeDbContextFactory<TContext>`, pokud existuje. Viz [Entity Framework Core - 03 - Entity](#).

Assembly pro Entity se hledá ve složce Entity\bin (a všech podsložkách), bere se poslední v čase, tj. nejaktuálnější. Tím řešíme případnou existenci více verzí assembly v případě existence lokálního buildu v Debugu i v Release.

Co generuje

Viz níže uvedené:

- DbDataSource, FakeDataSource
- DbRepository
- DataEntries
- Metadata

Generované soubory

V `DataLayeru` jsou generovány soubory:

- `_generated\DataEntries\Namespace\EntityEntries.cs`
- `_generated\DataSources\Namespace\EntityDataSource.cs`
- `_generated\DataSources\Namespace\EntityDbDataSource.cs`
- `_generated\DataSources\Namespace\Fakes\FakeEntityDataSource.cs`
- `_generated\Repositories\Namespace\EntityRepository.cs`
- `_generated\Repositories\Namespace\EntityDbRepository.cs`
- `_generated\Repositories\Namespace\EntityDbRepositoryBase.cs`

a dále jsou jednorázově vytvořeny soubory:

- `Repositories\Namespace\EntityRepository.cs`
- `Repositories\Namespace\EntityDbRepository.cs`

V **Modelu** jsou generovány soubory:

- `_generated\Metadata\Namespace\EntityMetadata.cs`

Poznámka ke složce `_generated`

V každém projektu (Model, Entity) jen jednu složku `_generated`. To umožňuje přehledné zobrazení pending changes (všechno generované lze snadno sbalit a přeskočit) nebo třeba [vynechání generovaných souborů z navigace ReSharperu](#).

Poznámka k entitám pro vztah M:N

Pro entity reprezentující vztah M:N (entity mající jen složený primární klíč ze dvou sloupců a nic víc) se žádný kód negeneruje.

Troubleshooting

Nepodaří se načíst `Microsoft.EntityFrameworkCore` (případně obdobnou)

Generátor kódu sám pod pokličkou používá `Microsoft.EntityFrameworkCore` a `Microsoft.EntityFrameworkCore.Design`.

Technický problém: Pokud používá projekt, kterému se generuje kód, musí generátor kódu načíst tyto novější assembly do paměti, což se nepodaří, protože jsou již načteny jiné verze (zjednodušeno). Použití aplikačních domén není možné (.NET Core neobsahuje podporu aplikačních domén).

Řešení: Aktualizovat generátor kódu na novější EFCore, která je použita v projektu, publikovat jej a použít jej v projektu (viz výše).

Poznámka: Od zavedení `Havit.Data.EntityFrameworkCore.CodeGenerator.Tool` již k tomuto problému nedochází.

Entity Framework Core - 04 - Unit of Work (BeforeCommit, AfterCommit, validate)

[Martin Fowler: PEAA: Unit Of Work](#)

`IUnitOfWork` / `IUnitOfWorkAsync`

Poskytují metody:

- `AddFor(Insert | Update | Delete)`
- `AddRangeFor(Insert | Update | Delete)`
- `Commit / CommitAsync`
- `RegisterAfterCommitAction` (umožňuje přidat zvenku nějakou akci k provedení po commitu (odeslání emailu, smazání cache, atp.).

Dokumentace [IUnitOfWork](#) a [IUnitOfWorkAsync](#).

`DbUnitOfWork`

Dokumentace [DbUnitOfWork](#). Implementuje `IUnitOfWork` i `IUnitOfWorkAsync`.

Navíc poskytuje protected metody k uzpůsobení si UoW v aplikaci:

- `PerformAddFor(Insert | Update | Delete)` - určeno k override, pokud se má změnit logika registrace entit. Standardně se ihned registrují do `IDbContextu` a zároveň se evidují v kolekci. Mazání objektů s příznakem smazání neprovede registraci k delete, ale nastaví jim příznak smazání a zavolá nad objekty `PerformAddForUpdate`.
- `BeforeCommit`
- `AfterCommit`
- `GetAllKnownChanges` - vrací známé změny v `changetrackeru`

RegisterAfterCommitAction

Umožňuje přidat zvenku nějakou akci k provedení po commitu (odeslání emailu, smazání cache, atp.). Akce je spuštěna metodou `AfterCommit` a je spuštěna jen jednou (i kdyby nad stejnou instancí `DbContextu` došlo k více voláním `SaveChanges`).

Příklad


```
private void ProcessPayment(Payment payment)
{
    ...
    // vbec nevíme, kde je unitOfWork.Commit(), ale víme, že po jeho
    spuštění dojde k odeslání notifikace
    unitOfWork.RegisterAfterCommitAction(() => SendNotification
(payment));
    ...
}
```

Koncept BeforeCommitProcessorů

DbUnitOfWork obsahuje koncept, který umožní při volání commitu spustit služby pro každou změněnou entitu ještě před uložením objektu. Je možné tak "na poslední chvíli" provést v entitách nějaké změny.

Pro implementaci nějakého vlastního BeforeCommitProcessoru je třeba implementovat interface `IBeforeCommitProcessor<TEntity>` a zaregistrovat službu do DI containeru (Castle Windsor). Díky použití s DI mohou mít beforecommitprocessorů vlastní závislosti.

Příklad

Viz např. implementace [SetCreatedToInsertingEntitiesBeforeCommitProcessor](#).

```
public class MyEntityBeforeCommitProcessor :
IBeforeCommitProcessor<MyEntity>
{
    public void Run(ChangeType changeType, MyEntity changingEntity)
    {
        if (changeType == ChangeType.Insert)
        {
            // do something
        }
    }
}
```

```
container.Register(Component.For<IBeforeCommitProcessor<MyEntity>>().
ImplementedBy<MyEntityBeforeCommitProcessor>().LifestyleSingleton());
// ev. jiný lifestyle
```

`SetCreatedToInsertingEntitiesBeforeCommitProcessor`

Pro nově založené objekty, které mají vlastnost `Created` typu `DateTime` a v této vlastnosti je hodnota `default(DateTime)` nastaví aktuální čas (z `ITimeService`). Tj. automaticky nastavuje hodnotu `Created` entitám, které ji nastavenou nemají.

Je použit automaticky (díky registraci do DI containeru).

Koncept EntityValidatorů

Před uložením objektů (a po spuštění BeforeCommitProcessorů) se spustí validátory entit, které umožňují kontrolovat jejich stav. Pokud je zjištěna nějaká validační chyba, je vyhozena výjimka typu `ValidationFailedException`.

Pro implementaci nějakého vlastního EntityValidatoru je třeba implementovat interface IEntityValidator<TEntity>. K implementaci je jediná metoda Validate, jež má na výstupu kolekci IEnumerable stringů - zjištěných chyb při validaci.

Dále je třeba službu zaregistrovat do DI containeru (Castle Windsor), díky použití s DI containerem může mít třída vlastní závislosti.

Příklad

```
public class MyEntityEntityValidator : IEntityValidator<MyEntity>
{
    IEnumerable<string> Validate(ChangeType changeType, MyEntity
changingEntity)
    {
        if (changingEntity.StartDate >= changingEntity.EndDate)
        {
            yield return "Poátení datum musí pedcházet koncovému datu.";
        }
    }
}
```

IValidatableObject.Validate()

Jednou ze specifických možností implementace EntityValidatoru je IValidatableObject.Validate() přímo entitě. Je to něco podobného, jako jsme měli BusinessObject.CheckConstraints():

```
public IEnumerable<ValidationResult> Validate(ValidationContext
validationContext)
{
    if ((this.Parent == null) && (this.Id != (int)Project.Entry.
Root))
    {
        yield return new ValidationResult($"Property {nameof
(Parent)} is allowed to be null only for Root project.");
    }
    if ((this.Depth == 0) && (this.Id != (int)Project.Entry.Root))
    {
        yield return new ValidationResult($"Value 0 of {nameof
(Depth)} property is allowed only for Root project.");
    }
}
```

Tyto validace lze pak do commit-sekvence zapojit zaregistrováním služby ValidatableObjectEntityValidator do dependency-injection containeru (řádek 13):

```
private static void InstallHavitEntityFramework(IServiceCollection
services, InstallConfiguration configuration)
{
    DbContextOptions options = configuration.UseInMemoryDb
? new DbContextOptionsBuilder<GoranG3DbContext>().
UseInMemoryDatabase(nameof(GoranG3DbContext)).Options
```

```

        : new DbContextOptionsBuilder<GoranG3DbContext>().
UseSqlServer(configuration.DatabaseConnectionString, c => c.MaxBatchSize
(30)).Options;

        services.WithEntityPatternsInstaller()
            .AddEntityPatterns()
            //.AddLocalizationServices<Language>()
            .AddDbContext<GoranG3DbContext>(options)
            .AddDataLayer(typeof(IApplicationSettingsDataSource).
Assembly);

        services.AddSingleton<IEntityValidator<object>,
ValidatableObjectEntityValidator>();
    }

```

...pokud se to osvědčí, uděláme tuto registraci automatickou součástí registrace `WithEntityPatternsInstaller()`.



POZOR

`ValidatableObjectEntityValidator` nezajišťuje validace dle `DataAnnotations` atributů, jako jsou např. `[Required]`, `[MaxLength]` apod.

Pořadí akcí v commitu

Během commitu dochází postupně k těmto akcím:

- zavolání metody `BeforeCommit`
- spuštění `BeforeCommitProcessorů`
- spuštění `EntityValidátorů`
- uložení změn do databáze (`DbContext.SaveChanges[Async]`).
- zavolání metody `AfterCommit` (zajišťuje volání akcí registrovaných metodou `RegisterAfterCommitAction`)

Entity Framework Core - 04 - LookupServices

Jde o třídy, které mají zajistit možnost rychlého vyhledávání entity podle klíče. Na rozdíl od ostatních (`Repository`, `DataSources`) nejsou generované - píšeme je ručně, pro jejich napsání je však připravena silná podpora.

Třída je určena k použití u neměnných či občasné měněných entit a u entit které se mění hromadně (naráz). Není garantována stoprocentní spolehlivost u entit, které se mění často (myšleno zejména paralelně) v různých transakcích - invalidace a aktualizace může proběhnout v jiném pořadí, než v jakém doběhly commity.

Rovněž z principu "out-of-the-box" nefunguje korektně invalidace při použití více instancí aplikace k aktualizaci dat aplikace (farma, web+webjoby, atp.), pro distribuovanou invalidaci je udělána příprava.

Implementace

Je potřeba dědit z třídy, viz tato ukázka kódu minimálního kódu.

```

public class UzivatelLookupService : LookupServiceBase<string,
Uzivatel>, IUzivatelLookupService
{
    public UzivatelLookupService(IEntityLookupDataStorage
lookupStorage, IRepository<Uzivatel> repository, IDataSource<Uzivatel>
dataSource, IEntityKeyAccessor entityKeyAccessor, ISoftDeleteManager
softDeleteManager) : base(lookupStorage, repository, dataSource,
entityKeyAccessor, softDeleteManager)
    {
    }
}

```

```

        public Uzivatel GetUzivatelByEmail(string email) =>
            GetEntityByLookupKey(email);

        protected override Expression

```

Implementována je zejména vlastnost - `LookupKeyExpression`, jejíž návratovou hodnotu je expression pro získání párovacího klíče. Zde tedy říkáme, že párujeme uživatele dle emailu. Druhou implementovanou vlastností je `OptimizationHints`, vysvětlení viz níže.

Metoda `GetUzivatelByEmail` je pak službou třídy samotné, kterou mohou její konzumenti používat. Pod pokličkou jen volá metody `GetEntityByLookupKey`.

IncludeDeleted

By default nejsou uvažovány (a vráceny) příznakem smazané záznamy. Pokud mají být použity, je třeba provést override vlastosti `IncludeDeleted` a vrátit `true`.

Filter

Pokud nás zajímají jen nějaké instance třídy (neprázdný párovací klíč, objekty v určitém stavu, atp.), lze volitelně provést override vlastnosti `Filter` a vrátit podmínku, kterou musí objekty splňovat.

ThrowExceptionWhenNotFound

Pokud není podle klíče objekt nalezen, je vyhozena výjimka `ObjectNotFoundException`. Pokud nemá být vyhozena výjimka a má být vrácena hodnota `null`, lze provést override této vlastnosti, aby vracela `false`.

OptimizationHints

Pro efektivnější fungování invalidací (viz níže) je možné zadat určité hinty, např., pokud je entita `readonly` a tedy nemůže být za běhu aplikace změněna, nemusí k žádné invalidaci docházet.

Dependency Injection

Třídy je nutno do DI containeru instalovat nejen pod sebe sama, ale ještě pod servisní interface, který zajistí možnost invalidace dat při uložení nějaké entity (viz dále).

Není tak možné pro lookup service použít automatickou registraci pomocí atributu `[Service]`.

```

services.WithEntityPatternsInstaller()
    ...
    .AddLookupService<IUserLookupService,
        UserLookupService>();

```

Invalidace

Pokud dojde k uložení entity, je potřeba lookup data nějakým způsobem invalidovat. S objektem se může stát spousta věcí - změna vyhledávacího klíče, smazání příznakem, změna jiných vlastností tak, aby objekt již neodpovídal filtru, atp. Je třeba zajistit, aby lookup data držaná službou, byla aktuální.

Zvolené řešení je efektivnější než prostá invalidace, data jsou rovnou aktualizována na nové hodnoty.

To lze omezit tam, kde jsou entity např. `readonly`, viz `OptimizationHints`.

ClearLookupData

Pokud chceme ručně vynutit odstranění dat z paměti, je k dispozici metoda `ClearLookupData`.

Užitečné to může být pro situace:

- Jednorázově jsme použili lookup service a víme, že ji dlouho nebudeme potřebovat - pak zavolání metody uvolní paměť alokovanou pro lookup data.
- Došlo k úpravě dat mimo `UnitOfWork` (třeba stored procedurou) a potřebujeme dát lookup službě vědět, že lookup data již nejsou aktuální.

Použití repository

Objekty jsou po nalezení klíče v lookup datech vyzvednuty z repository. Přínos tohoto chování je takový, že získaný objekt je trackovaný a mohl být získán z cache, bez dotazu do databáze.

Pozor na scénáře, kde se ptáme do lookup služby opakovaně pro necachované objekty (nebo cachované, které ještě v cache nejsou), každé volání pak může udělat dotaz do databáze právě pro získání instance z repository.

Pro tuto situaci je k dispozici metoda, která nevrátí instanci entity, ale jen její klíč - `GetEntityKeyByLookupKey`. Je pak možno implementačně získat klíče všech objektů, které můžeme ve vlastním kódu přehodit metodě `GetObjects` repository. Pokud máme problém poté objekty roztřídit znovu dle klíčů, můžeme uvažovat takto:

1. Nejprve získáme všechny klíče entit dle vyhledávané vlastnosti
2. Poté všechny entity načteme pomocí `repository.GetObjects(...)`, čímž dostaneme objekty do paměti (identity mapy, `DbContext`).
3. Nyní se můžeme do lookup služby (a metody `GetEntityByLookupKey`) ptát jeden objekt po druhém, vracení objektů z repository již nebude dělat dotazy do databáze, neboť jsou již načteny.

Použití složeného klíče

Pro vyhledávání je možno použít složený klíč, klíč musí mít vlastní třídu, která musí zajistit fungování porovnání v Dictionary, tedy předefinovat porovnání. S úspěchem lze použít v implementaci anonymní třídu, byť se trochu zhorší kvalita kódu tím, že jako typ klíče musíme uvést typ `object`.

Použití více entit pod jedním klíčem

Aktuálně není podporováno, je vyhozena výjimka.

Časová složitost

Snažíme se, aby složitost vyhledání byla $O(1)$.

Konstantní složitost samozřejmě neplatí pro první volání, které sestavuje vyhledávací slovník.


Implementační detail

Sestavení lookup dat provede jediný dotaz do databáze pro všechny objekty (s ohledem na `IncludeDeleted` a `Filter`). Nenačítají se celé instance entit, ale jen jejich projekce, tj. vrací se netrackované objekty, tj. nenaplní se identity mapa (`DbContext`) instancemi entit, ve kterých je vyhledáváno.

Entity Framework Core - 04 - Lokalizace

Pokud máme model s lokalizacemi (viz [Entity Framework Core - 02 - Model](#), kapitola Lokalizace), pak službou `ILocalizationService` získáváme pro entitu z kolekce `Localizations` hodnotu pro zvolený jazyk. Hodnotu můžeme získat pro "aktuální" nebo zvolený jazyk.

```
ContryLocalization countryLocalization = localizationService.  
GetCurrentLocalization(country);  
ContryLocalization countryLocalization = localizationService.  
GetLocalization(country, czechLanguage);
```

 Služba nezajišťuje načtení kolekce `Localizations` z databáze, zajišťuje jen výběr požadované hodnoty z této kolekce.

Logika hledání pro daný jazyk je postavena takto:

- Pokud existuje položka pro zadaný jazyk, je použita tato.
- Není-li nalezena, zkouší se hledat pro jazyk dle "neutrálnější culture" jazyka.
- Není-li nalezena, zkouší se hledat pro jazyk dle invariantní culture (prázdný `UICulture`).

Například pro uživatele pracující v češtině se hledá položka pro jazyky dle `UICulture` postupně pro "cs-cz", "cs", "".

Registrace DI

Použití služby je podmíněno registrací do IoC containeru, což můžeme udělat extension metodou `AddLocalizationServices`.

```
services.WithEntityPatternsInstaller( )
...
AddLocalizationServices<Language>( )
...;
```

Entity Framework Core - 04 - Repositories

Repositories jsou třídy s jednoduchými a opakovaně použitelnými metodami pro přístup k datům.

Repositories (navzdory 95% implementací nalezitelných na internetu) neobsahují metody pro CRUD operace.

IEntityRepository, *IRepository<Entity>*

Poskytuje metody:

- `GetAll(Async)`
- `GetObject(Async)`
- `GetObjects(Async)`

Pro každou entitu vzniká jeden interface pojmenovaný *IEntityRepository* (např. *ILanguageRepository*), který implementuje *IRepository<Entity>*.

EntityDbRepository

Generované třídy implementující *IEntityRepository*.

Poskytuje veřejné metody a vlastnosti (implementace *IRepository<Entity>*)

- `GetAll(Async)` - Vrací příznakem nesmazané záznamy, pokud je metoda nad jednou instancí volána opakovaně, nedochází k opakovaným dotazům do databáze.
- `GetObject(Async)` - Vrací objekt dle `Id`, pokud neexistuje záznam s takovým `Id`, je vyhozena výjimka.
- `GetObjects(Async)` - Vrací objekty dle kolekce `Id`, pokud neexistuje záznam pro alespoň jedno `Id`, je vyhozena výjimka. Při opakovaném volání metody jsou objekty vráceny z identity mapy (`IDbContextu`).

a protected vlastnosti

- `Data` a `DataIncludingDeleted` - viz [Data Sources](#), implementačně používají hodnoty ze závislosti `IDataSource<TEntity>`, čímž je lze snadno napsat test s mockem dat pro tyto vlastnosti.

Implementační instrukce

Není zvykem, aby se repository navzájem používaly jako závislosti v implementacích, protože by to mohlo vést až k nepřehlednému a neřešitelnému zauzlování repositories navzájem.

Pokud potřebuje jedna repository to samé, co jiná, což je samo o sobě nezvyklé, je doporučeno vyextrahovat kód do samostatné služby, např. jako `Query`.

Ghosts

Entity Framework Core nedisponuje aparátem [ghostů](#), na které jsme byli zvyklí v `Havit Business Layer`. Všechny objekty jsou načítány z databáze, ale bez referencí a kolekcí. Není-li zajištěno jinak, jsou reference null a kolekce null či prázdné (dle implementace).

Načítání závislých objektů

Jak je zmíněno výše, EF nedisponuje aparátem ghostů. Objekty jsou načítány z databáze, ale jejich reference jsou null, pokud referovaný objekt není v identity mapě. U kolekcí je to složitější, protože v kolekci se automaticky objeví jen ty objekty, které jsou v identity mapě (při vztahu 1:N).

Pokud chceme načíst referované objekty či kolekce, disponuje EF [třemi možnostmi načtení referovaných objektů](#). My máme navíc implementovaný [DbDataLoader](#).

Repository disponuje možnostmi načíst závislé objekty.

GetLoadReferences

Metoda je určena k override a definuje, jaké závislosti mají být s objektem načteny. Syntaxe viz [DbDataLoader](#).

Příklad:

```
protected override IEnumerable<Expression<Func<EmailTemplate, object>>>
GetLoadReferences( )
{
    yield return x => x.Localizations;
}
```

Návratového typu `IEnumerable<Expression<Func<Entity, object>>>` se není třeba bát 😊:

- `Func<Entity, object>` říká, že použijeme lambda výraz, kterým určíme z `Entity`, nějakou vlastnost vracející cokoliv
- `Expression` rozšiřuje `Func` o to, že se lambda výraz přeloží jako [expression tree](#)
- `IEnumerable` říká, že můžeme vrátit více takových výrazů.
- Viz ukázka, je to jednoduché.
- Aktuálně není možné touto metodou zajistit načtení objektů z kolekce (tedy `x => x.VlastnostA.VlastnostB` lze použít jen tehdy, pokud `VlastnostA` není kolekcí objektů).
 - použijte override `LoadReferences+Async`

LoadReferences(Async)

Načte závislosti definované v `GetLoadReferences`. Automaticky použito v metodách `GetAll`, `GetObject(Async)` a `GetObjects(Async)`. Pokud repository obsahuje vlastní metody vracející entity, je potřeba před navrácením dat provést dočtení závislostí touto metodou!

Načítání závislostí je provedeno pomocí [DbDataLoaderu](#), nikoliv pomocí `Include` (byť by to mohlo být někdy výhodnější). Možno overridovat (rozšířit) o další dočítání věcí, co nejsou přímo podporované skrze `GetLoadReferences` (např. prvky kolekce).

Příklad:

```
public EmailTemplate GetByXy(string xy) // vymyšleno pro ukázkou
{
    EmailTemplate template = Data.FirstOrDefault(item => item.XY ==
xy);
    LoadReferences(template);
    return template;
}
```

Entity Framework Core - 05 - Seedování dat

Seedování dat je automatické založení dat v databázi.

Definice dat k seedování

Seedování dat provádí třídy implementující interface `IDataSeed<>`. S jednoduchostí lze vytvořit třídu dědící ze třídy `DataSeed<>`, která tento interface poskytuje, je třeba jen implementovat abstraktní metodu `SeedData`.

Vytvořením instancí dat, metodou For a provedené konfigurace nad jejím výsledkem, se připraví data, která mají být v databázi. Připravená data se předhodí metodě Seed. (Poznámka: Metoda For vychází z otevřenosti pro další rozšíření, kdy se mohou data získávat z jiných zdrojů, např. ForCsv, ForExcel, ForResource. Není implementováno a budeme řešit, až bude potřeba.)

Párování pomocí sloupce Id (>99% případů)

Pokud jde o systémový číselník, do kterého nejsou **vkládány** hodnoty uživatelsky, pak můžeme na sloupci vypnout autoincrement a tím použít vlastní hodnoty pro Id.

Pokud jde o číselník, do kterého jsou vkládány hodnoty uživatelsky, můžeme namísto autoincrementu použít sekvenci, což nám umožní, abychom stále mohli použít vlastní hodnoty pro Id.

Seedovaná data s daty v databázi jsou pak párována pomocí Id.

Příklad bez autoincrementu

```
public class Role
{
    [DatabaseGenerated(DatabaseGeneratedOption.None)] // nepoužijeme
    autoincrement, čímž umožníme vkládat vlastní hodnoty do sloupce Id
    public int Id { get; set; }

    ...

    public enum Entry
    {
        Writer = -3,
        Reader = -2,
        Administrator = -1
    }
}
```

```
public class RoleSeed : DataSeed<CoreProfile>
{
    public override void SeedData()
    {
        Role[] roles = new[]
        {
            new Role
            {
                Id = (int)Role.Entry.Administrator, //
                Name = "Administrátor"
            },
            ... // Weader, Writer
        };
        Seed(For(roles).PairBy(item => item.Id)); // ekneme, že
```



```
se má párovat dle sloupce Id
    }
}
```

Příklad se sekvencí

```
public class User
{
    public int Id { get; set; }
    ...

    public enum Entry
    {
        SystemUser = -1
    }
}
```

```
public class UserConfiguration : IEntityTypeConfiguration<User>
{
    public void Configure(EntityTypeBuilder<User> builder)
    {
        builder.Property(user => user.Id).HasDefaultValueSql
("NEXT VALUE FOR UserSequence");
    }
}

// dále je teba na modelu (v DbContextu) zajistit existenci sekvence
// modelBuilder.HasSequence<int>("UserSequence");
```

```
public class UserSeed : DataSeed<CoreProfile>
{
    public override void SeedData()
    {
        User[] users = new[]
        {
            new User
            {
                Id = (int)User.Entry.SystemUser, //
nastavíme hodnotu pro sloupec Id
                Name = "(Systémový uživatel)"
            }
        };
        Seed(For(users).PairBy(item => item.Id)); // ekneme, že
```

```
se má párovat dle sloupce Id
    }
}
```

Párování pomocí sloupce Symbol (<1% případů)

i Toto řešení jsme implementovali a používali jako první verzi v Entity Framework 6. V Entity Framework Core nemá valného použití díky možnosti využití sekvence a řešení dle předchozího odstavce.

Pokud z nějakého důvodu potřebujeme autoincrement na číselníku, do něž potřebujeme seedovat data, např. pokud nemůžeme použít sekvenci, pak musíme párovat data v databázi s párovanými daty podle jiného sloupce než podle Id. V takovém případě používáme párování dle sloupce Symbol, který je (bohužel díky zpětné kompatibilitě) výchozím párováním, pokud sloupec existuje.

```
public class LanguageSeed : DataSeed<DefaultDataSeedProfile>
{
    public override void SeedData()
    {
        Language czechLanguage = new Language
        {
            Name = "esky",
            Culture = "cs-CZ",
            UiCulture = "",
            Symbol = Language.Entry.Czech.ToString()
        };

        Seed(For(czechLanguage)); // PairBy(item => item.
// Symbol) je default, který není třeba uvádět
    }
}
```

Konfigurace

Poskytuje fluidní API, následující metody lze řetěžit.

Párování seedovaných dat s daty v databázi

Metodami `PairBy` a `AndBy` lze určit sloupce, pomocí kterých budou seedovaná data párována s daty v databázi. Pro reference je nutno použít cizí klíče, nikoliv navigation property (jinými slovy: je nutno použít `LanguageId`, nikoliv `Language`).

```
Seed(For(...).PairBy(item => item.LanguageId).AndBy(item => item.
// ParentId));
Seed(For(...).PairBy(item => item.LanguageId, item => item.ParentId));
```

Aktualizace záznamů

Neexistující záznamy jsou standardně založeny. Existující záznamy aktualizovány.

Aktualizace existujících záznamů lze potlačit metodou `WithoutUpdate`:

```
Seed(For(...).WithoutUpdate());
```

Potlačení **aktualizace** pouze vybraných **vlastností** (sloupců) lze metodou `ExcludeUpdate`:

```
Seed(For(...).ExcludeUpdate(item => item.UserRank)); // sloupec  
UserRank nebude aktualizován
```

Závislé záznamy / Kolekce

Po uložení "parent" záznamů je možné zajistit uložení i jejich referencí či kolekci (například lokalizace číselníků). Jaké hodnoty budou ukládány je určeno metodami `AndFor` nebo `AndForAll`. Tyto metody dále umožňují provést nastavení seedování těchto záznamů.

Pro řešení "jak získat Id aktuálně uloženého záznamu" lze použít metodu `AfterSave`.

Dobrou ukázkou je níže ukázka výchozí konfigurace pro lokalizované tabulky.

Výchozí konfigurace

Symbol

Pokud třída obsahuje sloupec `Symbol`, je podle něj automaticky párováno (není-li určeno jinak):

```
Seed(For(...).PairBy(item => item.Symbol));
```

Lokalizované tabulky

Lokalizovaným třídám zajistí uložení lokalizací, lokalizacím zajistí párování dle `ParentId` a `LanguageId` (aktuálně hardcodováno v HFW). Že jde o lokalizované a lokalizační třídy se poznává podle implementace `ILocalization<>` a `ILocalized<>`.

```
// pseudokód popisující implementaci  
Seed(For(...)  
    // po uložení každého lokalizovaného záznamu nastavíme jeho  
    // lokalizací ParentId  
    .AfterSave(item => item.SeedEntity.Localization.ForEach  
        (localization => localization.ParentId = item.PersistedEntity.Id))  
    // po seedu lokalizovaných dat budeme seedovat lokalizace,  
    // které budeme párovat pomocí ParentId a LanguageId  
    .AndForAll(item => item.Localization, configuration =>  
        {  
            configuration.PairBy(item => item.ParentId,  
item => item.LanguageId);  
        }));
```

Ve skutečnosti je tato výchozí hodnota implementována mnohem komplikovaněji, efekt je takovýto.

Závislost na jiných seedovaných datech

Pokud chceme seedovat data, potřebujeme závislosti (například pro lokalizovaná data potřebujeme mít provedeno seedování jazyků).

Metodou `GetPrerequisiteDataSeeds` lze říct, na jakých seedech je tento závislý. V ukázce musí nejprve proběhnout `EmailTemplateSeed` a `LanguageSeed` než je spuštěn `EmailTemplateLocalizationSeed` (HFW řeší i detekci cyklů, atp.).

Návratovou hodnotou metody je `IEnumerable` **typů**, nikoliv instancí (instance dává Castle Windsor), implementace je tak náchylná na chybu - např. použití `typeof(Language)` namísto `typeof(LanguageSeed)`. Taková chyba je však v runtime detekována a je vyhozena výjimka.

```
public class EmailTemplateLocalizationSeed :
    DataSeed<DefaultDataSeedProfile>
{
    public override void SeedData()
    {
        ...
    }

    public override IEnumerable<Type> GetPrerequisiteDataSeeds()
    {
        yield return typeof(EmailTemplateSeed);
        yield return typeof(LanguageSeed);
    }
}
```

Profily

Data je možné seedovat pro různé účely - produkční data, testovací data pro testování funkcionality A, testovací data pro testování funkcionality B, atp. Pro tento scénář máme k dispozici profily pro seedování dat, které určují, které seedy se mají v jakém profilu spustit. Profil je třída implementující `IDataSeedProfile`, např. děděním z abstraktní třídy `DataSeedProfile`.

Jaká data patří do jakého profilu je určeno generickým parametrem u třídy `DataSeed`. Profily mohou mít závislosti na jiných profilem, jsou definovány pomocí metody `GetPrerequisiteProfiles()`.

```
public class TestDataProfile : DataSeedProfile
{
    public override IEnumerable<Type> GetPrerequisiteProfiles()
    {
        yield return typeof(DefaultDataSeedProfile);
    }
}
```

Jak se spustí jednotlivé seedování dat jednotlivých profilů je uvedeno dále.

Spuštění seedování

Spuštění seedování zajišťuje třída `DataSeedRunner`. Ta dostává závislosti: Kolekci data seedů, které se mají provést, persister seedovaných dat a strategii rozhodující, zda je třeba seedování pustit. S Castle Windsorem to funguje hezky dohromady. Profil, který se má spustit je určen generickým parametrem v metodě `SeedData`, viz ukázka.

Obvyklé spuštění je při startu aplikace (global.asax.cs, apod.):

```
IDataSeedRunner dataSeedRunner = container.Resolve<IDataSeedRunner>();
dataSeedRunner.SeedData<DefaultDataSeedProfile>();
```

```
// alternativn též dataSeedRunner.SeedData(typeof
(DefaultDataSeedProfile));
container.Release(dataSeedRunner);
```

Izolace jednotlivých seedů

Pro izolaci jednotlivých seedů se pod pokličkou vytváří pro každý seed nový DbContext. Přínosem v porovnání se sdíleným DbContextem je také výkonový zisk - počet objektů sledovaných ChangeTrackerem postupně nenarůstá, což při seedování většího objemu dat znamená podstatný přínos (při ladění jednoho z projektů se dostáváme na pětinašobné zrychlení). Přes tuto izolaci jednotlivé seedy sdílejí databázovou transakci (pro čtení i zápis) použitím TransactionScope.

Omezení spuštění seedování

Aby se nespouštělo seedování dat při každém startu aplikace, pamatujeme si v databázové tabulce __SeedData (zjednodušeně) verzi dataseedů, která byla spuštěna. V tabulce jsou záznamy pro jednotlivé profily, název profilu je primárním klíčem.

Pokud zjistíme, že daná verze již byla spuštěna, nebude se seedování spouštět.

Verze dataseedů se určí z názvu assembly, z file version a z data (datumu) posledního zápisu assembly. Díky datu poslední zápisu assembly nám funguje seedování i při vývoji, kde se nám jinak název a verze assembly nemění a bez data posledního zápisu assembly bychom při vývoji spustili seedování jen jedenkrát.

Toto je implementováno v OncePerVersionDataSeedRunDecision, která je zaregistrována do DI containeru pod IDataSeedRunDecision.

K dispozici je ještě strategie AlwaysRunDecision, která nic nekontroluje ale zajistí spuštění seedování vždy.

Entity Framework Core - 30 - Práce s entitami reprezentující dekomponovaný vztah ManyToMany (M:N)

Q: Pokud chci odebrat členství v M:N kolekci, musím entitu reprezentující vztah M:N zaregistrovat ke smazání nebo stačí odebrat z kolekce?

A: Stačí odebrat z kolekce (kolekce musí být pochopitelně předtím načtená).

▼ [Click here to expand...](#)

```
LoginAccount loginAccount = new LoginAccount();
loginAccount.Membership.Add(new Membership { LoginAccount =
loginAccount, Role = new Role() /* založíme i roli */ });
loginAccount.Membership.Add(new Membership { LoginAccount =
loginAccount, Role = new Role() /* založíme i roli */ });
dbContext.AddRange(loginAccount);
dbContext.SaveChanges();

loginAccount.Membership.Clear(); // co to udlá?
dbContext.SaveChanges(); // podíváme se!
```

Druhé volání SaveChanges způsobí smazání členství v kolekci:

```
exec sp_executesql N'SET NOCOUNT ON;
DELETE FROM [Membership]
WHERE [LoginAccountId] = @p0 AND [RoleId] = @p1;
SELECT @@ROWCOUNT;
',N'@p0 int,@p1 int',@p0=1,@p1=1
```

```
exec sp_executesql N'SET NOCOUNT ON;
DELETE FROM [Membership]
WHERE [LoginAccountId] = @p0 AND [RoleId] = @p1;
SELECT @@ROWCOUNT;
',N'@p0 int,@p1 int',@p0=1,@p1=2
```

Q: Pokud již v kolekci M:N nějaké prvky mám a chci toto členství aktualizovat. Musím provést "merge", tj. chtěné prvky nechat, nové založit existující ponechat?

A: Nikoliv, lze pohodlně vymazat kolekci a založit nové instance reprezentující vztah M:N. Entity Framework Core se postará o smazání odebraných prvků a vložení nových. Vazby (členství v kolekci), které má zůstat, nebude v databázi dotčeno přestože jsme pro vazbu založili novou instanci reprezentující tento vztah.

▼ [Click here to expand...](#)

```
Role role1 = new Role();
Role role2 = new Role();
Role role3 = new Role();
dbContext.AddRange(role1, role2, role3); // založíme role

LoginAccount loginAccount = new LoginAccount();
loginAccount.Membership.Add(new Membership { LoginAccount =
loginAccount, Role = role1 });
loginAccount.Membership.Add(new Membership { LoginAccount =
loginAccount, Role = role2 });
dbContext.AddRange(loginAccount); // založíme login account se
lenstvím v role1 a role2
dbContext.SaveChanges(); // uložíme

// role1 odebereme, role3 přidáme, role2 necháme, ale v nové instanci
Membership
loginAccount.Membership.Clear();
loginAccount.Membership.Add(new Membership { LoginAccount =
loginAccount, Role = role2 });
loginAccount.Membership.Add(new Membership { LoginAccount =
loginAccount, Role = role3 });

dbContext.SaveChanges(); // co to udlá?
```

Druhé volání SaveChanges způsobí smazání členství v kolekci prvku role1 a přidání prvku role3. To je skvělé.

```
exec sp_executesql N'SET NOCOUNT ON;
DELETE FROM [Membership]
WHERE [LoginAccountId] = @p0 AND [RoleId] = @p1;
SELECT @@ROWCOUNT;
',N'@p0 int,@p1 int',@p0=1,@p1=1

exec sp_executesql N'SET NOCOUNT ON;
INSERT INTO [Membership] ([LoginAccountId], [RoleId])
VALUES (@p0, @p1);
',N'@p0 int,@p1 int',@p0=1,@p1=3
```

Entity Framework Core - 70 - Generování seedů z existující databáze

Předpokládá se existující databáze postavená pro Business Layer a prvotní vytvoření seedů při migraci na Entity Framework Core. Kód je pro jedno použití, dále je třeba upravovat seedy ručně.

Nejprve je třeba vygenerovat z existující databáze model, ve kterém budou mít číselníky public property enum Entry. U číselníku se dále předpokládá, že primární klíč nemá autoincrement.

Vzorová ukázka je ve frameworku v projektu Havit.Business.EntityFrameworkSeedGenerator, kde je zpracován seed obecných číselníků. Pro specifické číselníky, nebo nějakou custom podobu je třeba kód modifikovat. Custom generování je dobré vytvořit si dvě textové šablony - jednu pro třídu, druhou pro jednotlivé řádky. Šablony si vložte do projektu jako embedded resource.

Entity Framework Core - 80 - Model Extensions

Model Extensions je všeobecná funkcionality, ako spravovať objekty, ktoré už nespravuje sám EF Core (tabuľky, sekvencie atď.), tak aby vývojár nemusel ručne aktualizovať tieto objekty v migrácii.

Objekty môžu byť napr. [uložené procedúry](#), pohľady, typy, apod.

Cieľom tejto funkcionality je:

- sledovať zmeny aj "stav" definície objektu naprieč migráciami
- uľahčiť prácu vývojárom (stačí upraviť definíciu objektu v SQL skripte a vytvoriť migráciu)

Na tejto stránke je popis (okrem základného postupu pre zapnutie tejto funkčnosti) hlavne toho, ako Model Extensions funguje a ako postaviť nad nimi ďalšiu funkcionality.

Základné pojmy

ModelExtension

Objekt reprezentujúci objekt spravovaný pomocou ModelExtensions funkcionality. Implementuje rozhranie [IModelExtension](#).

ModelExtender

Trieda, ktorá zapuzdruje ModelExtensions. ModelExtensions objekty sú návratovou hodnotou public metód v ModelExtenderi. ModelExtender implementuje marker rozhranie [IModelExtender](#).

Anotácie

[Interná funkcionality Entity Framework Core](#), jedná sa o súčasť infraštruktúry používaná DB providermi a rozšíreniami (ako napr. Model Extensions).

Použitie

Použitie závisí na konkrétnej funkcionalite, ktorá je postavená na infraštruktúre Model Extensions. Momentálne sú to iba [uložené procedúry](#). Na to, aby bolo možné použiť Model Extensions, je nutné iba aktivovať podporu Model Extensions v DbContexte (ktorá si sama zaktivuje [všeobecnú infraštruktúru](#) podľa potreby). Voliteľne je možné konfigurovať spôsob registrovania ModelExtenderov (triedy zapuzdrujúce samotné ModelExtension objekty).

DbContext.OnConfiguring

V prípade, že sa jedná o aplikáciu postavenú na BusinessLayer (a používa teda špeciálny BusinessLayerDbContext), tak stačí použiť BusinessLayerDbContext a podpora je automaticky zapnutá. Jedná sa však iba o niekoľko úprav DbContextu. V OnConfiguring je potrebné zavolať extension metódu **UseModelExtensions** a voliteľne nastaviť **ModelExtensionsAssembly**:

```
protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
```

```

{
    base.OnConfiguring(optionsBuilder);

    optionsBuilder.UseModelExtensions(builder => builder
        .ModelExtensionsAssembly(GetType().Assembly)
        .UseStoredProcedures()
        .UseViews());
}

```

Extension metóda **UseModelExtensions** umožňuje pomocou builder objektu zapnúť jednotlivé funkčné celky Model Extensions podľa potreby:

Metóda	Funkčnosť
UseStoredProcedures	Podpora pre uložené procedúry.
UseExtendedProperties	Podpora pre extended properties na ModelExtenderoch. Možnosť pridať ext. properties na akomkoľvek objekte, u ktorého to podporuje SQL Server.
UseBusinessLayerStored Procedures	Špeciálna podpora pre niektoré extended properties na uložených procedúrach, ktoré využíva Business Layer (+ generátor).
UseViews	WORK IN PROGRESS: podpora pre pohľady.

Registrácia ModelExtenderov

Registrácia prebehne automaticky, ako assembly sa vezme assembly, kde je definovaný DbContext, ktorý sa práve používa (viz. [ModelExtensionsAssembly](#) v HFW).

Je možné assembly aj explicitne nakonfigurovať pomocou metódy **ModelExtensionsAssembly** (viz. ukážka vyššie)

Princíp fungovania Model Extensions

Model Extensions sú založené na anotáciách EF Core, do ktorých sa ukladajú CREATE skripty pre vytvorenie daného objektu v DB. Pri scaffoldovaní migrácie sa porovnávajú existujúce anotácie v stávajúcom modeli a nové anotácie v novom modeli. V prípade rozdielu sa vygeneruje operácia AlterDatabaseOperation, na ktorú sa naviažu rozdielne anotácie.

Anotácie "miznú" respektíve pribúdajú v modeli práve pomocou ModelExtenderov. Princíp registrovania anotácií je nasledovný:

1. [ModelExtensionsAssembly](#) nájde všetky ModelExtendery v assembly
2. [Konvenca ModelExtensionRegistrationConvention](#) získava z ModelExtenderu inštancie tried ModelExtension
3. služba **IModelExtensionAnnotationProvider** ich transformuje na anotácie (Annotation objekty EF Core)
4. konvenca anotácie vloží do modelu

Pri spustení migrácie sa generujú SQL príkazy pre jednotlivé príkazy v migrácii. U Model Extensions sa najprv prevedú anotácie naspäť na ModelExtension a potom:

- v prípade nových anotácií sa vykoná pôvodný CREATE skript v anotácii
- v prípade existujúcich anotácií, ktoré sa zmenili, sa vygeneruje a spustí ALTER skript pre konkrétnu ModelExtension
- v prípade zmazaných anotácií sa vygeneruje a spustí DROP skript pre konkrétnu ModelExtension

ALTER a DROP skripty generuje služba **IModelExtensionSqlGenerator** (viď nižšie).



CREATE OR ALTER

Od SQL Serveru verzie 2016 SP1 [je možné použiť CREATE OR ALTER príkaz](#). Model Extensions tento use case momentálne nepodporuje (ale bude).

"Duplicitné" anotácie v kóde migrácie

Malo nevýhodou internej implementácie Model Extensions (ktorá je založená na anotáciách na úrovni modelu) je, že štandardný MigrationsModelDiffer (popis v [Business Layer - Database Migrations - Infraštruktúra](#)) generuje staré a aktuálne anotácie do AlterDatabaseOperation aj vtedy, ak hodnoty starých a nových anotácií sú rovnaké.

Dôsledkom je to, že vyscaffoldovaný kód novej migrácie obsahuje zbytočne veľa kódu (týkajúcich sa konfigurácie anotácii pre `AlterDatabaseOperation`).

Možným riešením je použiť upravenú implementáciu `MigrationsModelDiffer` komponenty [AlterOperationsFixUpMigrationsModelDiffer](#), ktorá zabezpečí odstránenie nadbytočných anotácií. Konfigurácia prebieha na **`ModelExtensionsExtension`**, pričom je možné túto funkčnosť konfigurovať aj na **`ModelExtensionsExtensionBuilder`** a v konečnom dôsledku aj v `UseModelExtensions` extension metóde:

```
optionsBuilder.UseModelExtensions(builder => builder
    .UseStoredProcedures()
    .UseExtendedProperties()
    .UseBusinessLayerStoredProcedures()
    .UseViews()
    .ConsolidateStatementsForMigrationsAnnotationsForModel(false));
// štandardne true, takto je to možné vypnúť
```

Toto nastavenie ovplyvní iba novovyscaffoldované migrácie, existujúce migrácie už nie (je nutné ich ručne upraviť späťne, čo za normálnych okolností nie je doporučované).

Entity Framework Core - 80 - Stored Procedures

Nad [ModelExtensions](#) je postavená podpora pre správu a verzovanie uložených procedúr v rámci migrácii EF Core.

Cieľom tejto funkcionality je:

- sledovať zmeny aj "stav" definície uloženej procedúry naprieč migráciami
- odstrániť použitie skriptov `<názov_SP>.v01.sql`, `<názov_SP>.v02.sql` atď. (viz. [ukážka na projekte 158.BTM](#))
- uľahčiť prácu vývojárom (stačí upraviť definíciu procedúry a vytvoriť migráciu)

Použitie

Definícia uloženej procedúry sa skladá z dvoch častí:

- metóda v triede `ModelExtender` (konkrétne potomok **`StoredProcedureModelExtender`**)
- CREATE skript uložený ako embedded resource v assembly Entity

Po definovaní `ModelExtender` triedy, jej metódy (a skriptu ako embedded resource) už stačí iba v priebehu vývoja upravovať definíciu uloženej procedúry a nechať vygenerovať scaffoldingom migráciu EF Core.

Triedy `ModelExtender` sú nájdené a spracované [všeobecnou logikou ModelExtensions](#).

Trieda a metóda pre SP

Potomok triedy **`StoredProcedureModelExtender`** umožňuje jednoducho pristupovať k embedded resources.

Metódy reprezentujú samotné uložené procedúry. Návratová hodnota je inštancia triedy **`StoredProcedureModelExtension`**, v ktorom sú uložené potrebné metadáta pre automatické verzovanie v rámci migrácii.

Nasledovná ukážka je z projektu 123.AEP (AccacePayroll) - popis jednotlivých častí je nižšie.

```
public class AttendancePeriodApprovalProcedures :
    StoredProcedureModelExtender
{
    public StoredProcedureModelExtension
    GetApprovalsForTimesheetCheck()
    {
        return Procedure("Havit.AccacePayroll.Entity.Sql.
        StoredProcedures.AttendancePeriodApproval_GetApprovalsForTimesheetCheck.
        sql");
    }
}
```

```

        public StoredProcedureModelExtension
GetPeriodApprovalForEmployee()
    {
        return Procedure("Havit.AccacePayroll.Entity.Sql.
StoredProcedures.AttendancePeriodApproval_GetPeriodApprovalForEmployee.
sql");
    }
}

```

Procedure metóda

Pomocná metóda Procedure vytvorí **StoredProcedureModelExtension** zo skriptu, ktorý je uložený v embedded resources assembly, kde sa nachádza trieda **AttendancePeriodApprovalProcedures** (Entity projekt)

```

protected StoredProcedureModelExtension Procedure(string
createScriptResourceName, Assembly resourceAssembly = null)

```

(Assembly, kde s bude hľadať reasource, sa dá prenastaviť cez voliteľný parameter.)

Tento skript by mal obsahovať príkaz CREATE PROCEDURE <názov procedúry>. Názov procedúry je dôležitý, nakoľko sa používa ako kľúč v internej implementácii.

Interná implementácia pri aktualizovaní uloženej procedúry automaticky vymení CREATE PROCEDURE za ALTER PROCEDURE príkaz (je to kvôli tomu, že CREATE PROCEDURE podporuje [OR ALTER variantu iba od SQL Serveru 2016 SP1 a vyššie](#)).



CREATE PROCEDURE skript

Tento skript by mal byť napísaný tak, aby bolo možné ho spustiť aj v prípade, že sa spúšťa modifikovaná verzia s ALTER PROCEDURE príkazom.

Ukážka CREATE skriptu:

```

CREATE PROCEDURE AttendancePeriodApproval_GetPeriodApprovalForEmployee
    -- Add the parameters for the stored procedure here
    @employee int,
    @date datetime
AS
BEGIN
    SET NOCOUNT ON;

    SELECT AttendancePeriodApprovalID FROM AttendancePeriodApproval
apa
    LEFT JOIN ClientPeriod cp ON apa.ClientPeriodID = cp.
ClientPeriodID
    WHERE apa.Deleted IS NULL AND cp.PeriodStart <= @date AND cp.
PeriodEnd >= @date AND apa.EmployeeID = @employee
END

```

Migrácie

Scenár - nová procedúra (nová ModelExtension)

Po pridaní novej SP (resp. vytvorení nového ModelExtenderu) vznikne v migrácii zhruba takýto kód:

```
migrationBuilder.AlterDatabase()  
    .Annotation("StoredProcedure:GetPeriodApprovalForEmployee:  
AttendancePeriodApproval_GetPeriodApprovalForEmployee", @"CREATE  
PROCEDURE AttendancePeriodApproval_GetPeriodApprovalForEmployee  
    -- Add the parameters for the stored procedure here  
    @employee int,  
    @date datetime  
  
AS  
BEGIN  
  
    SET NOCOUNT ON;  
  
    SELECT AttendancePeriodApprovalID FROM  
AttendancePeriodApproval apa  
        LEFT JOIN ClientPeriod cp ON apa.ClientPeriodID = cp.  
ClientPeriodID  
        WHERE apa.Deleted IS NULL AND cp.PeriodStart <= @date  
AND cp.PeriodEnd >= @date AND apa.EmployeeID = @employee  
END  
  
" )
```

Scenár - úprava procedúry (existujúca ModelExtension)

Pre úpravu SP stačí upraviť súbor s SQL skriptom - ModelExtensions infraštruktúra automaticky zdetekuje zmenu a pri vygenerovaní novej migrácie sa objaví tento kód:

```
migrationBuilder.AlterDatabase()  
    .Annotation("StoredProcedure:GetPeriodApprovalForEmployee:  
AttendancePeriodApproval_GetPeriodApprovalForEmployee", @"CREATE  
PROCEDURE AttendancePeriodApproval_GetPeriodApprovalForEmployee  
    -- Add the parameters for the stored procedure here  
    @employee int,  
    @date datetime  
  
AS  
BEGIN  
  
    SET NOCOUNT ON;  
  
    -- modified version of select  
    SELECT AttendancePeriodApprovalID FROM AttendancePeriodApproval  
apa  
        LEFT JOIN ClientPeriod cp ON apa.ClientPeriodID = cp.  
ClientPeriodID  
        WHERE apa.Deleted IS NULL  
        -- AND cp.PeriodStart <= @date
```

```

                AND cp.PeriodEnd >= @date AND apa.EmployeeID = @employee
END

")

        .OldAnnotation("StoredProcedure:GetPeriodApprovalForEmployee:
AttendancePeriodApproval_GetPeriodApprovalForEmployee", @"CREATE
PROCEDURE AttendancePeriodApproval_GetPeriodApprovalForEmployee
    -- Add the parameters for the stored procedure here
    @employee int,
    @date datetime
AS
BEGIN
    SET NOCOUNT ON;

    SELECT AttendancePeriodApprovalID FROM AttendancePeriodApproval
apa
    LEFT JOIN ClientPeriod cp ON apa.ClientPeriodID = cp.
ClientPeriodID
    WHERE apa.Deleted IS NULL AND cp.PeriodStart <= @date AND cp.
PeriodEnd >= @date AND apa.EmployeeID = @employee
END

");

```

(v anotáciach je vidieť starú verziu procedúry a novú)

Spustenie tejto migrácie vygeneruje toto SQL:

```

ALTER PROCEDURE AttendancePeriodApproval_GetPeriodApprovalForEmployee
    -- Add the parameters for the stored procedure here
    @employee int,
    @date datetime
AS
BEGIN
    SET NOCOUNT ON;

    -- modified version of select
    SELECT AttendancePeriodApprovalID FROM AttendancePeriodApproval apa
    LEFT JOIN ClientPeriod cp ON apa.ClientPeriodID = cp.ClientPeriodID
    WHERE apa.Deleted IS NULL
        -- AND cp.PeriodStart <= @date
        AND cp.PeriodEnd >= @date AND apa.EmployeeID = @employee
END

```

(zmena z CREATE PROCEDURE na ALTER PROCEDURE)

Scenár - zmazanie procedúry (zmazanie ModelExtension)

Pre zmazanie procedúry stačí zmazať metódu z Model Extenderu (pre poriadok by sa mal zmazať aj SQL skript s telom procedúry):

```

public class AttendancePeriodApprovalProcedures :
    StoredProcedureModelExtender
{
    public StoredProcedureModelExtension
    GetPeriodApprovalForEmployee()
    {
        return Procedure("Havit.AccacePayroll.Entity.Sql.
        StoredProcedures.AttendancePeriodApproval_GetPeriodApprovalForEmployee.
        sql");
    }
}

```

Nová migrácia síce obsahuje telo procedúry ale ako **OldAnnotation** na AlterDatabaseOperation:

```

migrationBuilder.AlterDatabase()
    .OldAnnotation("StoredProcedure:GetPeriodApprovalForEmployee:
AttendancePeriodApproval_GetPeriodApprovalForEmployee", @"CREATE
PROCEDURE AttendancePeriodApproval_GetPeriodApprovalForEmployee
-- Add the parameters for the stored procedure here
@employee int,
@date datetime
AS
BEGIN
    SET NOCOUNT ON;

    SELECT AttendancePeriodApprovalID FROM AttendancePeriodApproval
apa
    LEFT JOIN ClientPeriod cp ON apa.ClientPeriodID = cp.
ClientPeriodID
    WHERE apa.Deleted IS NULL AND cp.PeriodStart <= @date AND cp.
PeriodEnd >= @date AND apa.EmployeeID = @employee
END

");

```

Spustenie tejto migrácie vygeneruje toto SQL:

```

DROP PROCEDURE [AttendancePeriodApproval_GetPeriodApprovalForEmployee]

```

Internals - implementácia

Interná implementácia pre uložené procedúry je postavená na [infraštruktúre ModelExtensions](#). Pre funkčnosť uložených procedúr sú potrebné tieto tri triedy:

- StoredProcedureAnnotationProvider
- StoredProcedureModelExtender
- StoredProcedureSqlGenerator

(patria tu aj samozrejme triedy s atribútami pre extended properties, prípadne podpora pre prácu so súbormi s XML komentármi)

StoredProcedureAnnotationProvider

Využíva funkčnosť [ModelExtensionAnnotationProvider<T>](#), vytvára anotácie s prefixom "**StoredProcedure:**". Tieto anotácie reprezentujú aktuálny stav CREATE skriptu uloženej procedúry.

Implementácia tejto triedy zabezpečuje preklad z StoredProcedureModelExtension na IAnnotation a naopak.

StoredProcedureModelExtender

Popísané už vyššie. Je to базовая trieda pre všetky zastrešujúce triedy pre uložené procedúry. Interne sa používa [regulárny výraz pre naparsovanie](#) názvu procedúry z CREATE PROCEDURE príkazu.

StoredProcedureSqlGenerator

Najdôležitejšia trieda pre podporu uložených procedúr. Využíva funkčnosť базовой triedy [ModelExtensionSqlGenerator<T>](#) a určuje akým spôsobom sa bude generovať SQL príkaz [pre úpravu procedúry](#) a [DROPnutie procedúry](#).

Entity Framework Core - 80 - Pridanie novej funkcionality

Pre implementovanie novej funkcionality postavenej na ModelExtensions, je nutné naimplementovať niekoľko interfaceov a tie následne zaregistrovať v **ModelExtensionsExtension**.

IModelExtensionAnnotationProvider

Služba, ktorej úlohou je "konverzia" medzi ModelExtension objektom a anotáciami.

```
public interface IModelExtensionAnnotationProvider
{
    List<IAnnotation> GetAnnotations(IModelExtension dbAnnotation,
    MemberInfo memberInfo);

    List<IModelExtension> GetModelExtensions(List<IAnnotation>
    annotations);
}
```

GetAnnotations metóda dostane ako ďalší parameter **MethodInfo** objekt - metódu, ktorá vytvorila IModelExtension objekt.

GetModelExtensions metóda dostane zoznam všetkých anotácií definovaných na modeli. Je preto dôležité, aby ignorovala anotácie, ktoré nepozná (napr. môže rozpoznať "svoje" anotácie pomocou nejakého prefixu). Je vhodné, aby metóda zrekonštruovala ModelExtension objekt do rovnakej podoby ako vznikol v ModelExtenderi.

Tento provider môže vygenerovať viac anotácií pre jednu IModelExtension (ale potom ich musí následne vedieť zlúčiť do jednej IModelExtension). Zároveň však môže provider vygenerovať anotácie pre "cudzie" ModelExtensions - takýmto spôsobom fungujú [rozšírenia a pre uložené procedúry](#) pre BusinessLayer (extended properties).

ModelExtensionAnnotationProvider<T>

Bázová trieda ModelExtensionAnnotationProvider<T> uľahčuje implementáciu tohto interfaceu a to tak, že umožňuje pracovať so strong-type podobou ModelExtension (určenú cez typový parameter).

IModelExtensionSqlGenerator

Služba, ktorá generuje ALTER a DROP skripty pre ModelExtension. Bázová trieda ModelExtensionSqlGenerator<T> umožňuje pracovať so strong-type podobou ModelExtension.

```
public interface IModelExtensionSqlGenerator
{
    string GenerateAlterSql(IModelExtension modelExtension);
}
```

```

        string GenerateDropSql(IModelExtension modelExtension);
    }

```

Registrácia do ModelExtensionsExtension

Trieda ModelExtensionsExtension je navrhnutá (zhruba) podľa konvencií EF Core. Konfiguruje sa pomocou builder triedy založenej na **ModelExtensionsExtensionBuilderBase**. Na zaregistrovanie nových implementácií IModelExtensionSqlGenerator a IModelExtensionAnnotationProvider je odporúčané vytvoriť potomka **ModelExtensionsExtensionBuilderBase** podľa vzoru pôvodného builderu BusinessLayerModelExtensionsExtensionBuilder a následne použiť базовú metódu **WithOptions**:

```

public ModelExtensionsExtensionBuilder UseExtendedProperties() =>
    WithOption(e => e.
        WithAnnotationProvider<ExtendedPropertiesAnnotationProvider>());

```

Volanie buildera následne zaobaliť extension metódou podobnou **UseExtendedProperties**:

```

public static ModelExtensionsExtensionBuilder UseExtendedProperties
(this ModelExtensionsExtensionBuilder optionsBuilder)
{
    Contract.Requires<ArgumentNullException>(optionsBuilder !=
null);

    return new BusinessLayerModelExtensionsExtensionBuilder
(optionsBuilder)
        .UseExtendedProperties();
}

```

(prípadne metódy Use* na builderi volať priamo z tejto extension metódy)

Entity Framework Core - 80 - Infraštruktúra Model Extensions

Na tejto stránke je popis infraštruktúry pre Model Extensions ale aj DB Migrations (napr. extended properties).

Infraštruktúra je v zásade potrebná pre dve komponenty EF Core:

- IMigrationsAnnotationProvider
- IMigrationsSqlGenerator

IMigrationsAnnotationProvider je služba, ktorá poskytuje anotácie z rôznych častí modelu (IModel, IEntityType, IProperty, atp.) pre komponentu **MigrationsModelDiffer**. Tento differ vytvára diff medzi dvoma modelmi (používa sa pri scaffoldovaní migrácií).

IMigrationsSqlGenerator je služba, ktorá generuje MigrationCommandy z MigrationOperations. Túto službu využíva komponenta **Migrator** - vykonáva kód v migráciách. MigrationCommandy obsahujú už reálne SQL príkazy.

CompositeMigrationsAnnotationProvider

Implementácia rozhrania IMigrationsAnnotationProvider a zároveň patternu Composite - aby bolo možné používať viacero (nezávislých) implementácií IMigrationsAnnotationProvider.

CompositeMigrationsSqlGenerator

Implementácia rozhrania IMigrationsSqlGenerator a zároveň patternu Composite (aby bolo možné aby bolo možné používať viacero (nezávislých) služieb, ktoré generujú MigrationCommandy). Narozdiel od annotation provideru, tento SqlGenerator pracuje nad kolekciami implementácií IMigrationOperationSqlGenerator, pretože rozhranie IMigrationsSqlGenerator umožňuje implementovať iba túto metódu:

```
IReadOnlyList<MigrationCommand> Generate([NotNull]  
IReadOnlyList<MigrationOperation> operations, [CanBeNull] IModel model  
= null);
```

(a teda vygenerovať MigrationCommandy pre všetky MigrationOperations).

V prípade SqlGeneratoru je dôležité zachovať správne poradie MigrationCommandov pre rôzne MigrationOperations. Očakáva sa totiž, že pre jednu MigrationOperation je možné vytvoriť viacero MigrationCommandov z viacerých implementácií IMigrationOperationSqlGenerator.

TODO: príklad?

Registrácia do interného DI kontajnera - IDbContextOptionsExtension

Na to, aby EF Core (a jeho interné služby a komponenty) využíval tieto implementácie rozhraní, je nutné ich zaregistrovať do interného DI kontajnera. Toto je možné pomocou tejto extension metódy:

```
public static void UseCodeMigrationsInfrastructure(this  
DbContextOptionsBuilder optionsBuilder)
```

Interne táto metóda nepoužíva ReplaceService<TService, TImplementation> metódu, keďže funkcionality vyššie popísaných komponent je všeobecná a reprezentuje API pre ďalšiu funkcionality (ako napríklad podpora Model Extensions alebo extended properties).

Namiesto toho táto extension metóda používa zapúzdrenie rozšírení DbContextu - **IDbContextOptionsExtension**.

IDbContextOptionsExtension

IDbContextOptionsExtension síce je v Infrastructure namespace, ale je určený pre providerov (ako napr. SqlServer alebo InMemory) a iné rozšírenia. Funkčnosť ako Model Extensions alebo extended properties rozširuje funkčnosť SqlServer provideru.

Toto rozhranie definuje metódy ApplyServices, ktorá umožňuje pomocou IServiceCollection (takmer ľubovoľne) upraviť interný DI kontajner EF Core:

```
/// <summary>  
///     Adds the services required to make the selected options  
work. This is used when there  
///     is no external <see cref="T:System.IServiceProvider" /> and  
EF is maintaining its own service  
///     provider internally. This allows database providers (and  
other extensions) to register their  
///     required services when EF is creating an service provider.  
/// </summary>  
/// <param name="services"> The collection to add services to. <  
/param>  
/// <returns> True if a database provider and was registered; false  
otherwise. </returns>  
bool ApplyServices([NotNull] IServiceCollection services);
```

Registráciu oboch komponent (CompositeMigrationsAnnotationProvider a CompositeMigrationsSqlGenerator) zabezpečujú dve IDbContextOptionsExtension triedy:

- CompositeMigrationsAnnotationProviderExtension
- CompositeMigrationsSqlGeneratorExtension

Rozšíření funkcionality

Obidve IDbContextOptionsExtension třídy poskytují Fluent API na registraci dalších implementací IMigrationsAnnotationProvider nebo IMigrationOperationSqlGenerator.

✖ Důležité je, že obidve třídy sú navrhnuté podľa konvencií v EF Core a sú teda **immutable** - pridanie ďalšieho IMigrationsAnnotationProvider alebo IMigrationOperationSqlGenerator spôsobí vytvorenie ďalšej inšancie.

Konfigurácia preto musí vyzerať zhruba takto:

```
public static DbContextOptionsBuilder UseModelExtensions(this
DbContextOptionsBuilder optionsBuilder)
{
    // ...
    IDbContextOptionsBuilderInfrastructure builder = optionsBuilder;

    builder.AddOrUpdateExtension(optionsBuilder.Options
        .
        FindExtension<CompositeMigrationsAnnotationProviderExtension>()
        .
        WithAnnotationProvider<ModelExtensionsMigrationsAnnotationProvider>());

    // ...
    builder.AddOrUpdateExtension(optionsBuilder.Options
        .
        FindExtension<CompositeMigrationsSqlGeneratorExtension>()
        .
        WithGeneratorType<ModelExtensionMigrationOperationSqlGenerator>());
}
```

Entity Framework Core - 99 - Použití vlastních služeb z dependency injection

Pokud bychom v DbContextu (nebo jiné třídě) potřebovali instanci vlastní služby, řekněme ISoftDeleteManager, máme k dispozici několik možností:

Přidat závislost do konstruktoru

Pro:

- Technicky správné funkční řešení.
- Nejjednodušší na použití (při použití DI & containeru).

▼ [Click here to expand...](#)

```
public class MyDbContext : Havit.Data.Entity.DbContext
{
    public MyDbContext(ISoftDeleteManager softDeleteManager) :
    base(softDeleteManager) { }
    public MyDbContext(ISoftDeleteManager softDeleteManager,
    DbContextOptions<MyDbContext> options) : base(softDeleteManager,
    options) { }
    ...
}
```

Proti:

- Při přidání každé závislosti se změní signatura konstruktoru, což vynutí změny v konstruktoru (jako kdekoliv jinde při DI constructor injection). Proti:
- Ne vždy bychom chtěli získávat instanci z DI containeru, pak se použití komplikuje.
- Řekl bych, že jde proti myšlence DI v EF Core.

Rozšířit DbContext extension metodou .Use(...), která zaregistruje potřebné služby

Proti:

- Zbytečně komplikované.
- Dvojí konfigurace služeb - jedna pro aplikační DI container (Castle Windsor), jedna pro interní DI container EF Core.
- Ovlivňujeme interní DI container DbContextu (nastavujeme služby, které tam "nepatří")

Nastavit DbContext, aby sdílel svůj DI container s aplikačním

Proti:

- Ovlivňujeme interní DI container DbContextu (vlastně ho vypínáme)
- Vstupujeme na neznámé území, které úplně měníme.

▼ [Click here to expand...](#)

```
ServiceCollection serviceCollection = new ServiceCollection();
serviceCollection.AddEntityFrameworkSqlServer();
serviceCollection.AddDbContext<MyDbContext>((sp, options) => options.
UseInternalServiceProvider(sp));
serviceCollection.AddSingleton<ISoftDeleteManager,
SoftDeleteManager>();
serviceCollection.AddSingleton<ITimeService, ServerTimeService>();
ServiceProvider serviceProvider = serviceCollection.
BuildServiceProvider();

using (var dbContext = serviceProvider.GetService<MyDbContext>())
```

Vyzvedávat v DbContextu službu z aplikačního containeru (ala Service Locator)

EF Core používá dva DI containery - interní a aplikační (viz [zdrojáky](#))!

Pro:

- IMHO zapadá do filozofie EF Core.

Proti:

- Používá skryté závislosti!
- Vyžaduje použití s DI containerem (aneb no more new MyDbContext()).

▼ [Click here to expand...](#)

```
ServiceCollection serviceCollection = new ServiceCollection();
serviceCollection.AddEntityFrameworkSqlServer();
serviceCollection.AddDbContext<MyDbContext>();
serviceCollection.AddSingleton<ISoftDeleteManager,
SoftDeleteManager>();
serviceCollection.AddSingleton<ITimeService, ServerTimeService>();
ServiceProvider serviceProvider = serviceCollection.
BuildServiceProvider();
```

```

using (var dbContext = serviceProvider.GetService<MyDbContext>())
{
    // NOOP
}

// příklad použití s konstruktorem DbContextu (pro testování, hraní,
apod.)

ServiceCollection serviceCollection = new ServiceCollection();
serviceCollection.AddSingleton<ISoftDeleteManager,
SoftDeleteManager>();
serviceCollection.AddSingleton<ITimeService, ServerTimeService>();
ServiceProvider serviceProvider = serviceCollection.
BuildServiceProvider();

var optionsBuilder = new DbContextOptionsBuilder<MyDbContext>();
optionsBuilder.UseApplicationServiceProvider(serviceProvider);

using (var dbContext = new MyDbContext(optionsBuilder.Options))
{
    // NOOP
}

```

Vyzvednutí instance služby v DbContextu:

```

var softDeleteManager = this.GetService<ISoftDeleteManager>();
Contract.Assert(softDeleteManager != null, "softDeleteManager !=
null");

```

Entity Framework Core - 99 - Soft Deletes - Hledání cesty

Entity Framework Core má podporu pro [globální filtry](#) s možností potlačení filtru. To je vhodné a dobře použitelné pro naše scénáře soft deletů.

Pokud bychom chtěli vznést na architekturu aplikací nadstandardní požadavky (třeba multitenancy), narazíme. Zkusili jsme proto najít alternativu k tomu, jak se se soft deletes vypořádat.

Problém: `HasQueryFilter(...)` může nastavit jen jedinou podmínku

Pokud bychom chtěli mít více podmínek na entitu (třeba soft delete a multi-tenancy), je třeba, aby byly obě podmínky v jedné.

Není možné použít

```
modelBuilder.Entity<LoginAccount>().HasQueryFilter(loginAccount => !
loginAccount.IsDeleted);
modelBuilder.Entity<LoginAccount>().HasQueryFilter(loginAccount =>
loginAccount.TenantId == _tenantId);
```

Problémem je, že poslední vyhrává. Toto lze v modulu vyřešit pouze query filtrem:

```
modelBuilder.Entity<LoginAccount>().HasQueryFilter(loginAccount => !
loginAccount.IsDeleted && (loginAccount.TenantId == _tenantId));
```

Problém: Použití query filterů způsobí nenačtení root entity

Pokud je v následujícím kódu pro daný child master odfiltrován (třeba je příznakem smazán), pak není načten ani Child, byť Child s Id=1 existuje (je vrácen null). Pokud vynecháme Include, pak je Child normálně načten.

```
Child childIncludingMaster = dbContext.Set<Child>().Include(m => m.
Master /* has a query filter */).Where(c => c.Id == 1).FirstOrDefault();
```

Celá issue je trackována na githubu - <https://github.com/dotnet/efcore/issues/11691>

V podstatě nám [tímto příspěvkem vysvětlují](#), že na takovéto použití query filters nejsou zamýšleny, detaily na githubu.

Vize: Soft-deletes upravují dotazy pod pokličkou, potlačit je lze pomocí IgnoreSoftDeleteQueryFilter

a) Úprava dotazů pod pokličkou

Query Filtry registrované u entity se do dotazů přidávají ve třídě ModelExpressionApplyingExpressionVisitor, ve virtuální metodě VisitConstant.

Nabízí se možnost tvorby potomka, který tuto metodu rozšíří, avšak tvorba visitoru je napevno zadrátovaná v konstruktoru (abstraktní) třídy EntityQueryModelVisitor. To je škoda, je to jediná takto zadrátovaná závislost, která zásadním způsobem brání rozšiřitelnosti (a věřím, že to bude jednoho dne narovnáno a i tento visitor bude předán jako závislost třídy).

ModelExpressionApplyingExpressionVisitor je použit ve virtuální metodě OptimizeQueryModel (abstraktní) třídy EntityQueryModelVisitor, teoreticky by bylo možné doplnit ještě úpravu dotazu v potomkovi této třídy. To už je ale nesprávné drbání, třída má více potomků, které se vytvářejí přes factory. Implementace by tedy obnášela

- implementovat potomka (nebo zapouzdření) RelationalQueryModelVisitor
- implementovat náhradu za RelationalQueryModelVisitorFactory
- nahradit registraci IEntityQueryModelVisitorFactory za tuto factory

Řešení vidím, že by realizovat šlo, ale není správně. Proč bychom kvůli automatickým soft deletes měli implementovat potomky jistých tříd. Co kdybychom chtěli automatizovat něco jiného (třeba zmíněnou multi-tenancy)? Od čeho budeme dědit? Od RelationalQueryModelVisitor nebo od SoftDeleteRelationQueryModelVisitoru?

b) Označení dotazu IgnoreSoftDeleteQueryFilter()

Pokud se někdy podaří vyřešit bod a, tak toto je již triviální a lze v podstatě vyřešit zkopírováním kódu okolo metody IgnoreQueryFilter().

Následující kód umožňuje v IQueryable použít tuto metodu a zajistí, aby EF Code nepadal. Jinak to nic nedělá.

▼ [Click here to expand...](#)

```
public static class QueryableExtensions
{
```

```

        internal static readonly MethodInfo
IgnoreSoftDeleteQueryFiltersMethodInfo
        = typeof(QueryableExtensions)
            .GetTypeInfo().GetDeclaredMethod
(nameof(IgnoreSoftDeleteQueryFilters));

        public static IQueryable<TEntity>
IgnoreSoftDeleteQueryFilters<TEntity>(
            this IQueryable<TEntity> source)
            where TEntity : class
        {
            return
                source.Provider is EntityQueryProvider
                    ? source.Provider.
CreateQuery<TEntity>(
                                Expression.Call(
                                    instance:
null,
                                method:
IgnoreSoftDeleteQueryFiltersMethodInfo.MakeGenericMethod(typeof
(TEntity)),
                                arguments:
source.Expression))
                    : source;
        }
    }

    public class IgnoreSoftDeleteQueryFiltersExpressionNode :
ResultOperatorExpressionNodeBase
    {
        public IgnoreSoftDeleteQueryFiltersExpressionNode
(MethodCallExpressionParseInfo parseInfo, LambdaExpression
optionalPredicate, LambdaExpression optionalSelector) : base
(parseInfo, optionalPredicate, optionalSelector)
        {
        }

        public override Expression Resolve
(ParameterExpression inputParameter, Expression
expressionToBeResolved, ClauseGenerationContext
clauseGenerationContext)
        {
            return Source.Resolve(inputParameter,
expressionToBeResolved, clauseGenerationContext);
        }

        protected override ResultOperatorBase
CreateResultOperator(ClaueGenerationContext clauseGenerationContext)
        {

```

```

        return new
IgnoreSoftDeleteQueryFiltersResultOperator();
    }
}

public class IgnoreSoftDeleteQueryFiltersResultOperator :
SequenceTypePreservingResultOperatorBase, IQueryAnnotation
{
    public virtual IQuerySource QuerySource { get; set; }

    public virtual QueryModel QueryModel { get; set; }

    public override string ToString() =>
"IgnoreSoftDeleteQueryFilters()";

    public override ResultOperatorBase Clone(CloneContext
cloneContext)
        => new
IgnoreSoftDeleteQueryFiltersResultOperator();

    public override void TransformExpressions
(Func<Expression, Expression> transformation)
    {
    }

    public override StreamedSequence ExecuteInMemory<T>
(StreamedSequence input) => input;
}

```

Rozhodnutí

Máme na výběr tedy 2 možnosti:

- Neřešit soft deletes pomocí EF a použít stejné řešení nad EF, jako jsme použili s EF6 (FilteringCollections, Data /DataIncludingDeleted apod.). To mi sice přijde škoda se na začátku vzdát cesty, nicméně díky existenci [issue](#) volíme toto řešení.
- Řešit soft deletes pomocí EF a QueryFiltrů, tím aktuální implementaci znemožníme použití QueryFiltrů na něco jiného.

Entity Framework Core - 10 - Know-How - Výchozí hodnoty v databázi vs. uložené hodnoty

Model + Konfigurace

Mějme třídu FlagClass, jejíž vlastnost vlastnost MyFlag bude mít v databázi výchozí hodnotu true (resp. 1):

```

public class FlagClass
{
    public int Id { get; set; }
    public bool MyFlag { get; set; }
}

```

```
modelBuilder.Entity<FlagClass>().Property(fc => fc.MyFlag).
HasDefaultValue(true);
```

Neočekávané chování

Pak založení nové instance FlagClass s explicitně určenou hodnotou MyFlag = false **způsobí uložení hodnoty true**, přestože jsme nastavili false.

```
FlagClass flagClass = new FlagClass();
flagClass.MyFlag = false;
dbContext.Set<FlagClass>().AddRange(new FlagClass[] { myClass });
dbContext.SaveChanges();
```

způsobí

```
INSERT INTO [FlagClass]
DEFAULT VALUES;
SELECT [Id], [MyFlag]
FROM [FlagClass]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
```

Z pohledu C# má flagClass.MyFlag výchozí hodnotu (default bool == false) a zároveň má tato vlastnost definovanou výchozí hodnotu v databázi, takže ji nebude ukládat! Díky tomu dostane objekt výchozí hodnotu z databáze (true), ačkoliv jsme v kódu explicitně uvedli false.

Co s tím, jak to vyřešit?

Pokud chceme, aby Entity Framework definoval výchozí hodnotu v databázi, ale nespolehal na nastavení vlastnosti v databázi, lze triviálně upravit konfiguraci na

```
modelBuilder.Entity<FlagClass>().Property(fc => fc.MyFlag).
HasDefaultValue(true).ValueGeneratedNever();
```

Pak již uložení do databáze vypadá takto

```
INSERT INTO [FlagClass] ([MyFlag])
VALUES (@p0);
SELECT [Id]
FROM [FlagClass]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
```

Entity Framework Core - 10 - Know-How - Výchozí hodnoty v databázi vs. datové typy

Model + Konfigurácia

Majme triedu MyEntityClass, ktorej vlastnosť MyDate má v databáze nejakú špecifickú výchozí hodnotu (HasDefaultValue):

```
public class MyEntityClass
{
    public int Id { get; set; }
    public string Text { get; set; }
    public DateTime MyDate { get; set; }
}

modelBuilder.Entity<MyEntityClass>().Property(me => me.MyDate).
    HasDefaultValue(new DateTime(2019, 1, 1));
```

Nečakané chovanie

Entity Framework Core vygeneruje pre takýto model tento migračný skript:

```
CREATE TABLE [MyEntityClass] (
    [Id] int NOT NULL IDENTITY,
    [Text] nvarchar(max) NULL,
    [MyDate] datetime NOT NULL DEFAULT '2019-01-01T00:00:00.0000000',
    CONSTRAINT [PK_MyEntityClass] PRIMARY KEY ([Id])
);
```

Default hodnota pre stĺpec MyDate je nastavená na **2019-01-01T00:00:00.0000000**. Toto spôsobí chybu, ak sa pokúsime vložiť riadok do tejto tabuľky s default hodnotou, tak to spôsobí túto chybu:

Conversion failed when converting date and/or time from character string.

Podobná chyba nastane, ak takýto stĺpec dodatočne pridáme do existujúcej tabuľky, kde už nejaké záznamy existujú. Vtedy sa SQL Server pokúsi nastaviť default hodnotu pre existujúce záznamy a nepodarí sa mu to (a migrácia zlyhá).

Problém je presnosťou - EF Core vygeneroval default hodnotu pre **datetime2** stĺpec, aj keď tento stĺpec je typu **datetime**. Rovnaký problém je aj u **smalldatetime** a **date**.

Ako to vyriešiť?

Jedná sa o bug v EF Core, podrobnosti sú pri tejto issue v ADOS:

https://havit.visualstudio.com/DEV/_workitems/edit/44494/

Ako workaround sa dá nastaviť default hodnota cez DefaultValueSql:

```
modelBuilder.Entity<MyEntityClass>().Property(me => me.MyDate)
    .HasColumnType("datetime")
    .HasDefaultValueSql("'2019-01-01T00:00:00.000'");
```

Potom CREATE skript vyzerá takto:

```
CREATE TABLE [MyEntityClass] (
    [Id] int NOT NULL IDENTITY,
```



```
[Text] nvarchar(max) NULL,  
[MyDate] datetime NOT NULL DEFAULT ('2019-01-01T00:00:00.000'),  
CONSTRAINT [PK_MyEntityClass] PRIMARY KEY ([Id])  
);
```

Entity Framework Core - instalace dotnet ef ...

dotnet -ef mě psal:

```

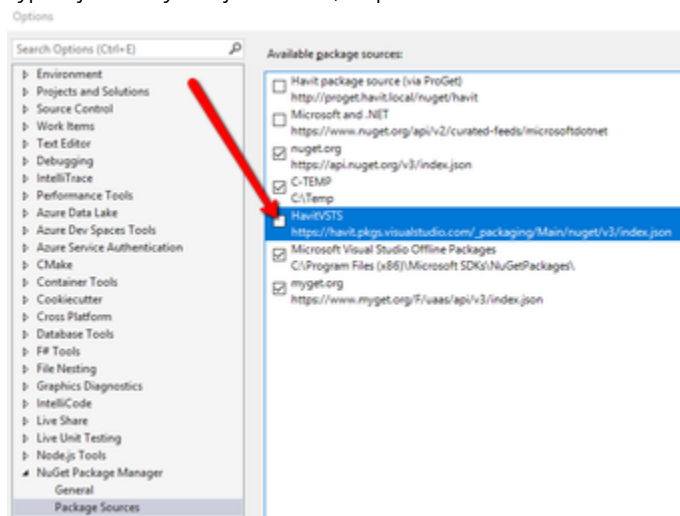
D:\dev\dotnet ef
Could not execute because the specified command or file was not found.
Possible reasons for this include:
 * You specified a built-in dotnet command.
 * You intended to execute a .NET Core program, but dotnet-ef does not exist.
 * You intended to run a global tool, but a dotnet-preload executable with this name could not be found on the PATH.

```

Musí se udělat instalace, napoprvé s chybou:

[illegible]

Vypnul jsem tedy zdroj HavitVSTS, resp. HavitADOS ve Visual studiu:



Následně se objevila následující chyba:

[illegible]

Vynutil jsem tedy instalace pro verzi 2.1:

```
D:\dev>dotnet tool install --global --version 2.1 dotnet-ef
You can invoke the tool using the following command: dotnet-ef
Tool 'dotnet-ef' (version '2.1.0') was successfully installed.
```

Takže nyní vše chodí:

```
D:\dev>dotnet ef
Entity Framework Core .NET Command-line Tools 2.1.0-rtn-30799
Usage: dotnet ef {options} [command]
```

```
Options:
--version      Show version information
-h|--help      Show help information
-v|--verbose   Show verbose output.
--no-color     Don't colorize output.
--prefix-output Prefix output with level.

Commands:
database       Commands to manage the database.
dbcontext      Commands to manage DbContext types.
migrations     Commands to manage migrations.

Use "dotnet ef [command] --help" for more information about a command.
```

Mimochodem, odinstalace se provede:

```
D:\dev>dotnet tool uninstall --global dotnet-ef
Tool 'dotnet-ef' (version '2.1.0') was successfully uninstalled.
```

Konec hlášení...

Entity Framework Core - 06 - Registrace do DI containeru

Registrace do DI containeru je podporována pro IServiceCollection a pro Windsor Castle.

Způsob použití je pro oba kontejnery stejný, liší se jen objekt, ke kterému se použije extension metoda WithEntityPatternsInstaller.

(I)ServiceCollection

```
services.WithEntityPatternsInstaller()
    .AddDataLayer(typeof(Havit.Application.DataLayer.Properties.
AssemblyInfo).Assembly)
    .AddDbContext<ApplicationDbContext>(options)
    .AddLocalizationServices<Language>() // volitelné
    .AddEntityPatterns();
```

Využíváme type forwarding a factories, které nejsou v ServiceCollection/ServiceProvider nativně podporovány.

Forwarding je řešení registrací třídy pod první interface a registrací dalších interface tak, že získávají hodnotu pomocí tohoto prvního interface ([kód](#)).

Factories jsou řešeny obdobně, jen je pro každý interface factory potřeba napsat implementaci, která se opakuje a opakuje ([kód](#)) (teoreticky bychom si vystačili z IServiceFactory a tedy jednou implementací, ale z historických důvodů je v našem kódu pro EF Core jinak).

Windsor Castle

```
container.WithEntityPatternsInstaller(c => c.GeneralLifestyle = ... /*
dle projektu */ )
    .AddDataLayer(typeof(Havit.Application.DataLayer.Properties.
AssemblyInfo).Assembly)
    .AddDbContext<ApplicationDbContext>(options)
    .AddLocalizationServices<Language>() // volitelné
    .AddEntityPatterns();
```

Je vyžadováno, aby u CollectionResolveru bylo povoleno použití prázdných kolekcí (pozor však při použití Castle.Windsor. MsDependencyInjection ("volosoft"), které toto zajišťuje samo, pak tento řádek nemusíme řešit, navíc by způsobil komplikace v podobě změny pořadí resolvování kolekcí):

```
container.Kernel.Resolver.AddSubResolver(new CollectionResolver  
(container.Kernel, allowEmptyCollections: true));
```