1. **Введение в Entity Framework Core**

Entity Framework Core (EF Core) представляет собой технологию от компании Microsoft для доступа к данным. В чём преимущества технологии:

1. Легковесная (как и все Core)
2. Расширяемая(open-source)
3. Кроссплатформенная и различные типы приложений (console, wpf…)
4. **Объектно-ориентированная**. Поэтому она использует такое свойство как **абстракция**. Мы работаем не с таблицами, индексами, первичными и внешними ключами, а с **объектами и их коллекциями (и есть сущность - entity)**. EFC является **ORM**-инструментом (object-relational mapping - отображения данных на реальные объекты).
5. Способна поддерживать **любую** базу данных (нужен лишь провайдер) имея при этом только **один API**. Т.е. если мы решим сменить целевую СУБД, то основные изменения в проекте будут касаться только настройки подключения к провайдерам. А **код** **прежний**!

Подключаемые библиотеки:

1. **Microsoft.EntityFrameworkCore.SqlServer** (либо не sql)
2. **Microsoft.EntityFrameworkCore.SqlServer.Design**
3. **Microsoft.EntityFrameworkCore.Tools** (если нужно создать бд)

Основные классы:

1. **DbContext**: определяет **контекст данных** (штука, где хранится группа таблиц), используемый для взаимодействия с базой данных (наш контекст обязательно наследуется от него)
2. **DbSet<TEntity>**: представляет набор объектов, которые хранятся в базе данных (и есть таблицы по сути)
3. **DbContextOptionsBuilder**: устанавливает параметры подключения к бд

Наш контекст поддерживает операции **CRUD** (Create, Read, Update, Delete):

1. Create: db.Users.**Add**(user1), db.Users.**AddRange**(user1, user2)
2. Read: user = db.Users.**FirstOrDefault**() / [i]
3. Update db.**SaveChanges**(),

db.Users.**Update**(user1)–если обновляем не в том же блоке кода, где редактируем

1. Delete: .**Remove**(user1), .**RemoveRange**(user1, user2)
2. **Модели**

Все сущности представлены в виде **моделей** (**models** = entity в бд ).

Три подхода к *сопоставлению* классов моделей с таблицами:

1. Условности (**conventions**) – уже за тебя заложено, что таблица называются по именам свойств например
2. Аннотации данных (**атрибуты**)
3. **Fluent API** – набор методов, которые определяют сопоставление между классами со свойствами и таблицами со столбцами.

*Сохранение изменений* моделей в базе данных:

1. Вручную (заходим и добавляем столбик например)
2. Database.EnsureDeleted и Database.EnsureCreated в конструкторе контекста - сначала удаляем базу данных со старой схемой, а потом создаем ее заново.
3. **Migrations** (миграции - представляют план перехода базы данных от старой схемы к новой). Для этого в Package Manager Console:

**Add-Migration** произвольное\_название\_миграции

**Update-Database**

В результате создаются 3 файла (миграция, метаданные, состояние модели). В миграции определяются два метода: Up() и Down(). В методе Up() с помощью вызова метода CreateTable() добавляется новое определение таблиц. В Down() – что будет при возврате к пред версии.

**Fluent API** методы (в методе *OnModelCreating(ModelBuilder mB)* ):

1)Включение сущностей в модель:

mb.Entity<Country>() - включает модель Country как таблицу в бд, если она не была создана (например не создали мы DbSet для него)

mb.Ignore<Company>() - исключает модель Company из бд

mb.Entity<Product>().Ignore(b => b.Rate) – исключает свойство Rate из таблицы Product

2) Сопоставление таблиц и столбцов:

mb.Entity<User>().ToTable("People") - сопоставляет модель User с таблицей People()

mb.Entity<User>().Property(u=>u.Id).HasColumnName("user\_id") – сопостовляет модель Id с user\_id

**Ключи**:

mb.Entity<User>().HasKey(u => u.Ident) - будет сопоставлять **ключ** с Ident (по умолчанию ключ сопоставляется с Id или UserId)

.HasKey(u => new { u.PassportSeria, u.PassportNumber}) – составной ключ из 2 свойств

.HasAlternativeKey(u => u.Ident) – доп ключ

**Индекс**:

.HasIndex(u => u.Passport) - для увеличения производительности поиска в базе данных применяются **индексы**. Установка индекса для свойства Passport (по умолчанию индекс создается для каждого свойства, которое используется в качестве внешнего ключа).

3) Генерация стандартных значений:

.Property(b => b.Id).ValueGeneratedNever() –отключение автогенерации (и так отключена у всех, кроме Id)

.Property(u => u.Age).HasDefaultValue(18) – стандартное значение

4) Ограничения

.Property(b => b.Name).IsRequired() – не может быть null, ты должен обяз задать его

.HasMaxLength(50)

5) Если огромный код:

Выносим настройки из метода *OnModelCreating()* в отдельный класс, реализующий интерфейс *EntityTypeConfiguration<T>.* А после этого в *OnModelCreating():* mb.**ApplyConfiguration**(new ProductConfiguration())

1. **Отношения между моделями**

**Навигационное свойство** – свойство в сущности, имеющий тип другой сущности [public Company CompanyProp { get; set;}] (тем самым **зависимая** сущность User *ссылается* на **главную** сущность Company)

**Внешний ключ** – свойство в зависимой сущности, предназначенное для хранения *ключа* главной сущности [public int CompanyId { get; set;}]. **Правило** названий внешних ключей:

[Имя\_**навигационного**\_свойства] **/** [Имя\_**класса**\_связанной\_сущности] + [Имя **ключа** из связанной сущности]

Настройка **ключа** с помощью **Fluent API**:

* **HasOne** и **HasMany** устанавливают навигационное свойство для сущности, для которой производится конфигурация
* **WithOne** и **WithMany** идентифицируют навигационное свойство на стороне связанной сущности
* HasOne/WithOne применяются для навигационного свойства, представляющего одиночный объект
* HasMany/WithMany используются для навигационных свойств, представляющих коллекции
* **HasForeignKey** устанавливается внешний ключ

mb.Entity<User>().**HasOne**(p => p.Company).**WithMany**(t => t.Users).**HasForeignKey**(p => p.CompanyInfoKey)

**Поведение** зависимой сущности в случае **удаления** главной сущности:

* **Cascade**: зависимая сущность удаляется вместе с главной
* **SetNull**: свойство-внешний ключ в зависимой сущности получает значение null
* **Restrict**: зависимая сущность никак не изменяется при удалении главной сущности
* По умолчания, если нав. свойство ссылочный тип(null может), то как SetNull, а если значимый тип(нет null), то Cascade (при этом Re: int?)

mb.Entity<User>()HasOne(p => p.Company).WithMany(t => t.Users).**OnDelete**(DeleteBehavior.**Cascade**)

**Загрузка связанных данных** (через навиг. Св-ва):

1. **Eager** loading (жадная)

// получаем пользователей

var users = db.Users

                   .**Include**(u=>u.Company)  // добавляем данные по компаниям

                   .ToList();

Если куча связанных данных:

// получаем пользователей

var users = db.Users

                    .Include(u=>u.Company)  // добавляем данные по компаниям

                        .**ThenInclude**(comp => comp.Country //к компании добавляем страну

                            .ThenInclude(count => count.Capital)//к стране добавляем столицу

                    .Include(u=>u.Position) // добавляем данные по должностям

                    .ToList();

1. **Explicit** loading (явная)

db.Users..***Load***() – абсолютно всех пользователей со всеми связ данными

db.Users.***Where***(p=>p.Id==company.Id).Load() – только кого нужно выбираем

*Отдельная* загрузка связ-х данных:

db.Entry(user).***Reference***(x => x.Company).Load() – св-во есть один-ый объект

db.Entry(company).***Collection***(t=>t.Users).Load() – св-во есть коллекция

1. **Lazy** loading

Загружаются при непосредственном обращении к сущности (напр. в цикле или .ToList() )

Но для этого надо:

* Добавить пакет Microsoft.EntityFrameworkCore.Proxies
* При конфигурации данных:

optionsBuilder.**UseLazyLoadingProxies()**.UseSqlServer(“…”)

* Все навигационные свойства – **virtual**, все классы *открыты для наследования*

No: использует синхронную загрузку. Поэтому для асинхронного нужны 1 и 2

**Отношения:**

1. **One-to-one**

-У какой модели явно прописано свойство Id (внешний ключ), та и является **зависимой**

-методы **HasOne**() и **WithOne**()

-можно хранить данные моделей в одной таблице – **ToTable**()

modelBuilder.Entity<User>().ToTable("Users");

modelBuilder.Entity<UserProfile>().ToTable("Users");

1. **One-to-many**

-Например, в одной компании может работать несколько сотрудников, а каждый сотрудник в свою очередь может официально работать только в одной компании

-методы **HasOne**(), **WithMany**() или HasMany(), WithOne()

3) **Many-to-many**

-Например, один студент может посещать сразу несколько курсов, и, в свою очередь, один курс может посещаться множеством студентов

- EF Core 3.0 **не** позволяет создавать отношение многие-ко-многим без связующей сущности. Поэтому фактически: **many-to-many = 2\*one-to-many**. Т.е. один Student - много StudentCourse, один Course - много StudentCourse (где StudentCourse – связующее, хранит курс и студента).

**No:** когда определенный класс существует **не сам по себе**, а несет некоторую дополнительную информацию по отношению к другой главной модели можно назначить одну модель **хозяином** другой (**собственные** типы). Тогда он будет дополнять, а хранится в одной таблице с хозяином.

modelBuilder.Entity<User>().**OwnsOne**(u => u.Profile);

В метод **OwnsOne**() указывается навигационное свойство, которое представляет зависимый тип.