ASP.NET Core. Веб-приложения

EF Core и структуры данных



# На этом уроке

1. Освоим обобщенность
2. Рассмотрим различные структуры данных
3. Изучим подход для работы с EF Core
4. Реализуем первый репозиторий на базе EF Core

# 

# Введение в обобщенность (generic)

Один из самых важных моментов в C# — это способность строго типизировать и обобщить заранее неизвестный тип. С одной стороны, это может показаться достаточно сложным механизмом, а с другой — обобщения открывают большую гибкость. Если посмотреть стандартные типы в сборках C#, то обобщенность будет как минимум в каждой из них. Но для понимания того, что такое обобщенность, стоит рассмотреть один из самых ярких примеров, где она демонстрирует свою гибкость.

Допустим, у нас есть несколько таблиц в некой базе данных: Users (пользователи), Cities (города), Orders (заказы). Каждая таблица представляет собой определённую сущность в нашем приложении: UserEntity, CityEntity и OrderEntity. Для снижения дублирования кода все эти сущности будут наследоваться от базового класса BaseEntity, в которой уже определены общие для всех наследников поля: Id в качестве целочисленного числа и Comment как обычный строковой комментарий.

|  |
| --- |
| public abstract class BaseEntity  {  public int Id { get; set; }    public string Comment { get; set; }  }    public sealed class UserEntity : BaseEntity  {  public string Name { get; set; }    //other properties goes here  }    public sealed class CityEntity : BaseEntity  {  public string Name { get; set; }    //other properties goes here  }    public sealed class OrderEntity : BaseEntity  {  public decimal Total { get; set; }    //other properties goes here  } |

В предыдущих уроках мы уже сталкивались с понятием паттерна IRepository. Его цель — абстрагировать некий источник данных, потому что конечных потребителей не интересует, какой именно этот источник (база данных или сервис). Соответственно, для каждой сущности у нас будет по одному репозиторию IUsersRepository, ICitiesRepository и IOrdersRepository. Как вы помните, репозиторий должен содержать как минимум четыре метода, которые закрывают все четыре операции из CRUD (create/read/update/delete).

Рассмотрим первый пример, когда у нас нет возможности использовать обобщенность:

|  |
| --- |
| public interface IUsersRepository  {  bool Add(UserEntity entity);    IEnumerable<UserEntity> Get();    bool Update(UserEntity entity);    bool Delete(int id);  }    public interface ICitiesRepository  {  bool Add(CityEntity entity);    IEnumerable<CityEntity> Get();    bool Update(CityEntity entity);    bool Delete(int id);  }    public interface IOrdersRepository  {  bool Add(OrderEntity entity);    IEnumerable<OrderEntity> Get();    bool Update(OrderEntity entity);    bool Delete(int id);  } |

Структура классов и наследования здесь достаточно топорная и неповоротливая. Нет базовой абстракции для всех репозиториев. Это не так уж и страшно, но гибкость будет потеряна, потому что мы не можем добавлять общий функционал для всех сущностей сразу.Обратите внимание на то, что у методов одинаковое название, но сигнатура аргументов различается только типом: в первом UserEntity, во втором CityEntity и в последнем OrderEntity.

А теперь разберем подход с обобщенностью:

|  |
| --- |
| public interface IEntitiesRepository<TEntity>  {  bool Add(TEntity entity);    IEnumerable<TEntity> Get();    bool Update(TEntity entity);    bool Delete(int id);  }    public interface IUsersRepository : IEntitiesRepository<UserEntity>  {    }    public interface ICitiesRepository : IEntitiesRepository<CityEntity>  {    }    public interface IOrdersRepository : IEntitiesRepository<OrderEntity>  {    } |

Как видно из кода, у нас есть один базовый интерфейс, у которого есть открытая обобщенность, и она задана в конце названия этого интерфейса – TEntity. Она позволяет вставить любой тип в дальнейшем, что и было сделано в последующих репозиториях – IUsersRepository, где мы указали в обобщенности тип UserEntity и т. д. Все наследуемые дальше методы смогут принимать именно тот тип, который мы указали при наследовании.

К примеру, в «пустых» уже IUsersRepository, ICititesRepository, IOrdersRepository нам больше не нужно дублировать методы с измененной сигнатурой аргументов – они уже там будут автоматически унаследованы с правильным типом аргументов.

|  |
| --- |
| public class SomeLogic  {  public SomeLogic(  IUsersRepository usersRepository,  ICitiesRepository citiesRepository,  IOrdersRepository ordersRepository)  {  usersRepository.Add(new UserEntity());  IEnumerable<UserEntity> userEntities = usersRepository.Get();    citiesRepository.Add(new CityEntity());  IEnumerable<CityEntity> cityEntities = citiesRepository.Get();    ordersRepository.Add(new OrderEntity());  IEnumerable<OrderEntity> orderEntities = ordersRepository.Get();  }  } |

Обобщенность дает разработчику гибкость в реализации общей логики для многих типов. Это хорошо просматривается, в списках – List<TItem>. Мы можем строго типизировать список к нужному типу, и соответственно, этот список будет работать только с указанным типом, хотя ничто не мешает сделать список других типов. Но работать оба списка будут одинаково.

|  |
| --- |
| List<int> myIntList = new List<int>();  List<string> myStringList = new List<string>();  myIntList.Add(1);  myStringList.Add("Foo"); |

# Обобщенные коллекции IList, IDictionary

Две самых часто используемых структур данных в C#: обычный список IList и ключ-значение IDictionary. На протяжении курса вы уже сталкивались с ними и знаете их принцип работы. Список используется, когда индексирование не важно, а справочник — когда мы чаще ищем по какому-либо ключу ( например, в качестве ключа идеально подходит уникальный идентификатор сущности).

|  |
| --- |
| IDictionary<int, CityEntity> citiesIndex = new Dictionary<int, CityEntity>();  citiesIndex.Add(1, new CityEntity() { Id = 1 });  citiesIndex.Add(2, new CityEntity() { Id = 2 });  CityEntity cityEntity = citiesIndex[2]; |

Обратите внимание на то, как здесь используется обобщенность: ключ и значение являются обобщениями и могут быть любыми типами, хотя поведение для всех типов будет одинаковое.

# Обобщенные коллекции IReadOnlyList, IReadOnlyDictionary

Рассмотрим тот же паттерн репозиторий. При чтении из источника данных у нас формируется некий список сущностей. Этот список является списком исключительно для отображения. То есть, если мы в него добавим новую сущность, то она не появится в источнике данных. Однако потребителю нашего репозитория функционал добавления в список не нужен и даже может привести к неправильному пониманию кода. Другой разработчик может по ошибке подумать, что добавление новой сущности в этот список приводит к добавлению в источник данных. Признак хорошего кода — возможность прочитать его и понять, как он работает, без лишнего погружения.

В C# помимо IList и IDictionary есть два интерфейса, которые являются отображением списка и справочника как структур только для чтения. Ими удобно указывать, что этот список не обновляется со стороны потребителя, а только со стороны поставщика.

|  |
| --- |
| Dictionary<int, CityEntity> citiesIndex = new Dictionary<int, CityEntity>();    IReadOnlyDictionary<int, CityEntity> readOnlyCitiesIndex = citiesIndex;    List<CityEntity> cityListEntities = new List<CityEntity>();    IReadOnlyList<CityEntity> readOnlyCityEntitiesList = cityListEntities; |

Соответственно, в репозиториях стоит всегда возвращать списки и коллекции только на чтение.

|  |
| --- |
| List<CityEntity> cityListEntities = new List<CityEntity>();    IReadOnlyList<CityEntity> readOnlyCityEntitiesList = cityListEntities;    // readOnlyCitiesIndex.Count == 0    cityListEntities.Add(new CityEntity());    // readOnlyCitiesIndex.Count == 1 |

В примере выше мы создали коллекцию городов. Она пустая и ее представление в виде списка на чтение тоже пустое. После этого в базовую коллекцию был добавлен один город, и ее представление на чтение автоматически тоже стало содержать этот город. Это происходит из-за того, что представления на чтение является той же коллекцией, что и на чтение. Просто ограничено интерфейсом на чтение – IReadOnlyList.

# Обобщенные коллекции ImmutableList, ImmutableDictionary

Исходя из примера выше, стоит отметить, что не всегда нам нужны представления коллекций, которые могут изменяться от поставщика. Бывает, что нужно отдавать потребителю структуру данных, которую ни поставщик, ни потребитель не может изменить, а может только пересоздать.

|  |
| --- |
| List<CityEntity> cityListEntities = new List<CityEntity>();    ImmutableList<CityEntity> immutableList\_One = cityListEntities.ToImmutableList();    // immutableList\_One.Count == 0    cityListEntities.Add(new CityEntity());    // immutableList\_One.Count == 0    ImmutableList<CityEntity> immutableList\_Two = cityListEntities.ToImmutableList();    // immutableList\_Two.Count == 1 |

Даже при том, что поставщик добавил новую сущность в исходный список, структура не изменилась: в ней все так же хранится список из 0 элементов. Просто поставщик вернул новую структуру данных.

Для упрощения понимания стоит изучить на практике поведение этих структур данных и запомнить таблицу ниже:

|  |  |
| --- | --- |
| Структура | Можно изменять список |
| List | Да |
| Dictionary | Да |
| IReadOnlyList | Напрямую нет, только через List |
| IReadOnlyDictionary | Напрямую нет, только через Dictionary |
| ImmutableList | Нет |
| ImmutbaleDictionary | Нет |

Главное правило — всегда возвращайте ту структуру данных, которая четко отражает задуманную логику. Если вы не планируете изменять коллекции извне, то только на чтение, а если не собираетесь изменять даже в источнике, то неизменяемую. В других случаях можно использовать стандартный тип.

# Что такое ORM

При работе с базой данных язык SQL может доставить очень много неудобств. В хороших компаниях есть четкое разделение на разработчиков баз данных и на обычных разработчиков. Но в целом знание SQL является хорошим тоном и увеличит вашу ценность на рынке.

Как мы уже говорили, язык SQL достаточно сложен, особенно при работе с большими таблицами и со сложной связкой. Для упрощения работы ввели такое понятие, как Object-Relational mapping. Если дать описание простыми словами, то это «автоматический мэппинг таблиц и запросов на классы в C#». На предыдущих курсах мы уже затрагивали ORM, а точнее micro ORM – пакет для работы с базами данных Dapper.

# Подключаем EF Core и провайдера для СУБД Postgres

Допустим, есть такая таблица в базе PostgreSQL:

|  |
| --- |
| create table "User"  (  "Id" integer,  "Comment" text,  "FirstName" text,  "LastName" text,  "MiddleName" text,  "IsDeleted" boolean  ); |

Это обычное представление сущности User. Скрипт этой таблицы упрощен для быстрого понимания. В таблице явно указано, что есть уникальный идентификатор, комментарий, ФИО и признак, удален ли данный пользователь или нет. В контексте C# данную таблицу можно трактовать таким классом:

|  |
| --- |
| public class BaseEntity<TUniqueId>  where TUniqueId : struct  {  public TUniqueId Id { get; set; }    public bool IsDeleted { get; set; }  }    [Table("User", Schema = "Test")]  public sealed class User : BaseEntity<int>  {  public string Comment { get; set; }    public string FirstName { get; set; }    public string LastName { get; set; }    public string MiddleName { get; set; }  } |

В базовом классе у нас есть открытый обобщенный тип уникального идентификатора. Далее он ограничен тем, что он может быть только структурой, соответственно, тип можем указать только примитивный – целочисленный. Это может быть тип int или long, в зависимости от того, какой тип указан в самой таблице. В таблице у нас указан integer, и, исходя из таблицы типов ниже, он маппится на обычный тип int в C#.

|  |  |
| --- | --- |
| **Тип Postgresql** | **Тип C#** |
| boolean | bool |
| smallint | short |
| integer | int |
| bigint | long |
| real | float |
| money | decimal |
| text | string |

Для чего был сделан обобщенный тип уникального идентификатора? Все достаточно просто. Иногда идентификаторы в разных сущностях имеют разные типы. Это вызвано тем, что некоторые сущности могут быть в огромном количестве и для них нужен большой диапазон значений. А для некоторых хватит и меньшего диапазона. Чтобы сэкономить память, т. к. long весит значительно больше как в базе, так и в памяти у C#, мы посредством этого обобщения можем указать конкретный тип для каждой сущности. К примеру, таблица пользователей будет не слишком большая и диапазона значений хватит integer. А вот если это была бы таблица заказов, то там логичнее было бы иметь тип long, т. к. на одного пользователя может прилагаться более одного заказа. Этот обобщенный идентификатор уменьшит количество дублированного кода в классах, потому что нам просто нужно указать его тип при наследовании в конечной сущности.

Если присмотреться, то можно увидеть, что у сущности User также определен атрибут – Table. Он указывает, какая таблица и какая схема, откуда брать данные для движка EF Core.

Но для начала нужно подключить нужные nuget пакеты:

|  |  |
| --- | --- |
| **Пакет** | **Функционал** |
| System.ComponentModel.Annotations | Пакет для аннотации. Здесь и находится атрибут Table |
| Microsoft.EntityFrameworkCore | Базовая сборка движка EF Core |
| Npgsql | Провайдер базы данных для PostgreSQL |
| Npgsql.EntityFrameworkCore.PostgreSQL | Поддержка провайдера баз данных PostgreSQL в EF Core |

Начало работы с EF Core лежит через определения моделей к таблицам. Этот метод называют Code First. Таблицы, классы и все остальное пишется не в автоматическом режиме с помощью дизайнера, а руками. В данном случае у разработчика больше контроля и над таблицами, и над кодом. Такой подход является довольно распространенным, и в дальнейшем мы будем придерживаться именно его. Второй подход – это Model First. Здесь используется дизайнер для составления моделей сущностей, а код создается автоматически при любом изменении схемы в дизайнере. Этот подход встречается реже, да и в некоторых IDE дизайнера из коробки нет – Jetbrains Rider или VS Code. Дизайнер схемы работает с edmx файлами (базируются на XML), в которых описана структура. Зачастую edmx файлы становятся очень запутанными и сложными.

Далее работа с EF Core состоит из простейших шагов по созданию контекста базы данных:

|  |
| --- |
| internal sealed class UserDbContext : DbContext  {  public DbSet<User> Users { get; set; }    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)  {  optionsBuilder .UseNpgsql("Host=192.168.1.72;Database=GeekBrains;Username=postgres;Password=qwe123;");  }    protected override void OnModelCreating(ModelBuilder modelBuilder)  {  modelBuilder.Entity<User>().Ignore(x => x.Comment);  }  } |

Внимательно рассмотрим этот контекст. Он представляет собой наследника класса DbContext, который указан в сборке движка EF Core. Это обязательное условие. Далее каждая таблица и ее сущность должны быть определены свойствами. В нашем случае свойство Users – это прямой доступ к таблице User в базе. При обращении к ней мы можем манипулировать данными. Обратите внимание, что тип данного свойства DbSet<TEntity>. В одном контексте может быть n+1 таких свойств по разным сущностям.

Дальше идут два метода, которые настраивают доступ и саму модель. В переопределённом методе OnConfiguring происходит настройка доступа и передача строки подключения к базе данных с помощью DbContextOptionsBuilder аргумента.

А в методе OnModelCreating модель настраивается уже посредством Fluent синтаксиса.

Fluent Syntax — то вызов цепочки extensions методов. К примеру, мы четко указали, что у нас есть модель User и что при выборке нужно игнорировать поле Comment.

Осталось только зарегистрировать данный контекст в контейнере для инъекции в другие классы. Например, в наш IUserRepository:

|  |
| --- |
| public static IServiceCollection AddSqlRepositories(this IServiceCollection container)  {  container.AddSingleton<UserDbContext>();    return container;  } |

Теперь все потребители, которым нужен доступ к базе данных, могут получить UserDbContext. Но лучше придерживаться того, чтобы только наследники IRepository требовали данный контекст и вся работа с источниками данных была именно через IRepository:

|  |
| --- |
| internal sealed class UserRepository : IUsersRepository  {  private readonly UserDbContext \_context;    public UserRepository(UserDbContext context)  {  \_context = context;  }    public bool Add(UserEntity entity)  {  try  {  \_context.Add(entity);  \_context.SaveChanges();  }  catch (Exception exception)  {  return false;  }    return true;  }    public IReadOnlyList<UserEntity> Get()  {  // skipped try/catch    return \_context.Users.Where(x => x.IsDeleted == false).ToList();  }    public bool Update(UserEntity entity)  {  // skipped try/catch    return Commit();  }    public bool Delete(int id)  {  // skipped try/catch    UserEntity entity = \_context.Users.Find(id);    entity.IsDeleted = true;    return Commit();  }    private bool Commit()  {  int count = \_context.SaveChanges();    return count > 0;  }  } |

Как можно заметить, манипуляция данными в EF Core происходит через Fluent Syntax. Многие из этих методов транслитерируются в запрос SQL.

Рассмотрим код метода выборки всех пользователей из таблицы, кроме тех, которые были удалены:

|  |
| --- |
| public IEnumerable<UserEntity> Get()  {  // skipped try/catch    return \_context.Users.Where(x => x.IsDeleted == false).ToList();  } |

Как мы уже говорили, свойство Users – это прямой доступ к таблице. Посредствам данного синтаксиса мы фильтруем данные на предмет маркера удаления. Таким образом, в конечной выборке у нас будут только те сущности, которые не были удалены.

Как это можно доказать? Просто выполнить данный метод и запросить с помощью специального запроса в базу, какие запросы выполнялись:

|  |
| --- |
| select pid,  usename as username,  datname as database\_name,  query,  application\_name,  backend\_start,  state,  state\_change  from pg\_stat\_activity |

В выведенном списке можно найти примерно такой запрос:

|  |
| --- |
| SELECT u."Id", u."FirstName", u."IsDeleted", u."LastName", u."MiddleName"  FROM "Test"."User" AS u  WHERE NOT (u."IsDeleted") |

Это и доказывает, что Where транслировалось в WHERE NOT.

Отдельно стоит рассмотреть приватный метод Commit. Он сохраняет все изменения, которые происходили с сущностями в базе данных. Что это значит? Наш метод Get возвращает список не удаленных пользователей. Он формируется посредством движка EF Core, и каждая сущность в этом списке встаёт на учет – т.е. любое изменение свойств будет зафиксировано самой EF Core. Чтобы сохранить это в базе, нужно просто вызвать у контекста метод SaveChanges. Именно из-за этого метод Update вызывает метод Commit, игнорируя аргумент.

Слежение за изменениями объектов — хороший функционал, но он не всегда нужен. Кроме того, он несет довольно сильную нагрузку. К примеру, если требуется только вывести на экран данные и они точно не будут редактироваться, то слежение за изменением объектов будет излишне. Как его отключить? Для этого нужно явно указать, что учет нам не нужен. Модифицируем код репозитория:

|  |
| --- |
| public IReadOnlyList<UserEntity> Get(bool track = false)  {  // skipped try/catch    IQueryable<UserEntity> userEntities = \_context.Users.Where(x => x.IsDeleted == false);    if (track)  {  return userEntities.ToList();  }    return userEntities.AsNoTracking().ToList();  } |

При работе с EF Core нужно следить за тем, чтобы случайно не забрать все данные из таблицы. Метод ToList является заключительным и физически возвращает объекты из таблицы. Он выполняет запрос немедленно, в то время как метод AsEnumerable и AsQueryable выполняется с задержкой – lazy loading.

|  |  |  |
| --- | --- | --- |
| ToList | AsEnumerable | AsQueryable |
| Выполнение запроса сразу летит в базу данных, формируется список объектов и возвращается потребителю. Стоит использовать тогда, когда нужно получить фиксированный список. | Выполняется с задержкой. Грузит каждую запись поочередно в память и затем проведет фильтрацию. Стоит использовать для фильтрации результата уже в коде, а не в базе. | Выполняется с задержкой. Транслитерируется в SQL код. Не грузит каждую запись в память, вся фильтрация происходит на стороне базы после формирования SQL запроса. Работает быстрее, чем AsEnumerable |

Пример использования AsQueryable:

|  |
| --- |
| public IReadOnlyList<UserEntity> Get(bool selectEmptyFirstName, bool skipFirstTenUsers, bool track = false)  {  // skipped try/catch    IQueryable<UserEntity> query = \_context.Users.Where(x => x.IsDeleted == false);    if (selectEmptyFirstName == false)  {  query = query.Where(x => x.FirstName != "");  }    if (skipFirstTenUsers)  {  query = query.Skip(10);  }    if (track)  {  return query.ToList();  }    return query.AsNoTracking().ToList();  } |

В этом примере строится запрос на выборку пользователей с возможностью пропустить первых 10 пользователей и/или пропустить всех пользователей с пустым именем. Метод Skip часто используется для пейджинга данных.

Запрос выполнится только после того, как принудительно вызовется метод ToList.

# Практическое задание

В вашем приложении из предыдущего урока необходимо подключить EF Core, создать свой DbContext и структуру базы для моделей. Нужно перевести репозитории на правильные структуры данных и использовать DbContext как источник данных из этих таблиц.

Примечание: реализовать CRUD для Users & Employees.

# Дополнительные материалы

1. Pro ASP.NET Core MVC 2, Adam Freeman
2. <https://docs.microsoft.com/ru-ru/aspnet/core/mvc/overview?view=aspnetcore-5.0#web-apis>
3. Ссылка на репозиторий проекта Timesheets: <https://github.com/evgshk/gb-ts-app>

# Используемые источники

1. <https://www.npgsql.org/doc/types/basic.html>
2. <https://docs.microsoft.com/ru-ru/ef/core/>
3. <https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable?view=net-5.0>
4. <https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ireadonlylist-1?view=net-5.0>
5. <https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ireadonlydictionary-2?view=net-5.0>