Темы урока

[**Вводная часть**](#_mkwp76bay9xx) **2**

[Планы на сегодня](#_teroujgn83w2) 2

[**Пишем код DAL, это самое главное :)**](#_977r9mbi36bm) **2**

[Reminder.Storage.SqlServer.EF:](#_eqed66gwteb) 2

[EntityFrameworkReminderStorage: IReminderStorage](#_fmzkkict8o3a) 2

[ReminderStorageContext : DbContext](#_6bjzbe101vy) 2

[ReminderItemDto](#_iql9wunfs7hr) 3

[Описание модели через Fluent API в OnModelCreating](#_bamuai3wcdll) 4

[Особенности при описании ReminderItemStatus](#_dh3oq57xxuu9) 5

[Связываем контекст БД с классом реализации интерфейса IReminderStrorage](#_h1chwfm8dgyw) 5

[int Count (проверка работоспособности идеи)](#_7gwcq231pzbl) 6

[Добавляем логирование](#_oq7vyhybv2ca) 6

[Guid Add(ReminderItemRestricted reminder)](#_1nkswlo753rw) 8

[ReminderItem Get(Guid id)](#_ncu6tkycay5p) 9

[List<ReminderItem> Get(int count = 0, int startPostion = 0)](#_h1g3e2fplm28) 10

[List<ReminderItem> Get(ReminderItemStatus status, int count, int startPostion)](#_gzmacosbu8ee) 11

[List<ReminderItem> Get(ReminderItemStatus status)](#_8ts7jpgicl3w) 11

[bool Remove(Guid id)](#_bm2rr9j2gd9) 12

[void UpdateStatus(IEnumerable<Guid> ids, ReminderItemStatus status)](#_i4c13z1zmvuv) 12

[void UpdateStatus(Guid id, ReminderItemStatus status)](#_og0g4skkh7ba) 12

[Reminder.Storage.SqlServer.EF.DbInit](#_rw45oqgoz1i2) 13

[appsettings.json](#_flpjv98a9nye) 13

[ConnectionStringFactory.cs](#_71yrwou7dxoo) 13

[DesignTimeReminderStorageDbContextFactory.cs](#_7u3epoujzy4t) 14

[**Создаём миграцию InitialCreate**](#_de4b3ga51zhg) **15**

[**Подключаем новый DAL к Web API**](#_riijgo575o2e) **15**

[Проект Reminder.Storage.WebApi](#_m8648j9s1asl) 15

[Reminder.Storage.WebApi/Startup.cs](#_j2jhbobgek03) 15

[Reminder.Storage.WebApi/appsettings.json](#_rbmvoov64u0s) 16

[**Запускаем чат-бот**](#_xeg481cdwml0) **16**

[**Домашнее задание**](#_xliovyhyekrq) **16**

На этом уроке мы **должны успеть** подготовить слой доступа к данным на базе MS SQL Server через Entity Framework Core и подключить новую реализацию DAL к нашему чат-боту.

# Вводная часть

## Планы на сегодня

1. Мы напишем две новых сборки
   1. Библиотеку классов .NET Standard Reminder.Storage.SqlServer.EF — собственно библиотека слоя доступа к данным. Мы реализуем все методы интерфейса IReminderStorage; вы увидите, насколько проще можно делать многие вещи, когда работаешь с БД через LINQ.
   2. Консольное приложение .NET Core Reminder.Storage.SqlServer.EF.DbInit, в котором мы будем работать с версиями БД с помощью EF Core:
      1. В этом проекте мы будем создавать и хранить файлы миграций,
      2. Через него мы будем накатывать миграции в среде разработки.
   3. Сборку для тестов мы писать не будем, так как времени на это у нас не останется, скорее всего, однако, вы здесь достаточно подкованы и, полагаю, **теперь сможете делать это самостоятельно** :) На протяжении последних двух месяцев мы уделяли внимание Unit-тестам в достаточной степени.
2. Мы подключим нашу сборку к чат-боту и убедимся, что всё работает в точности так, как мы задумали (ReminderItem-ы кладутся в базу данных MS SQL Server).

# Пишем код DAL, это самое главное[[1]](#footnote-0) :)

## Reminder.Storage.SqlServer.EF:

Создаём библиотеку классов.

### EntityFrameworkReminderStorage: IReminderStorage

Создаём основной класс EntityFrameworkReminderStorage, наследуем его от IReminderStorage и пока забываем о нём на время.

### ReminderStorageContext : DbContext

Обратимся к созданию модели БД. У нас будет *как бы code first*, потому что мы на самом деле уже знаем, что примерно нам нужно от хранилища, однако, давайте сделаем вид, что не знаем.

Наш контекст БД мы вместе с DTO-классами[[2]](#footnote-1) положим в папку Context в корне этого проекта. На это две причины, во-первых, мы не можем сделать контекст не public, так как **необходимо будет работать с миграциями из запускаемого приложения**, а мы находимся в библиотеке классов. Во-вторых, чтобы всё, что не касается напрямую единственно интересного реальным клиентам класса EntityFrameworkReminderStorage было скрыто в отдельном неймспейсе Context.

Здесь мы создаём класс ReminderStorageContext и определяем его DbSet’ы.

Я для данной конкретной задачи хочу избежать излишнего усложнения схемы и не буду работать лишь с одним DbSet-ом — ReminderItems.

public class ReminderStorageContext : DbContext

{

public DbSet<ReminderItemDto> ReminderItems { get; set; }

}

### ReminderItemDto

Теперь создадим класс нашего объекта ReminderItemDto:

public class ReminderItemDto

{

public Guid Id { get; set; }

public string ContactId { get; set; }

public DateTimeOffset TargetDate { get; set; }

public string Message { get; set; }

public ReminderItemStatus Status { get; set; }

public DateTimeOffset CreatedDate { get; set; }

public DateTimeOffset UpdatedDate { get; set; }

}

### 

### Описание модели через Fluent API в OnModelCreating

Я не хочу нагружать этот объект логикой, поэтому всё, что касается описания логики модели буду описывать в контексте через Fluent API:

public class ReminderStorageContext : DbContext

{

public DbSet<ReminderItemDto> ReminderItems { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<ReminderItemDto>(entity =>

{

entity

.ToTable("ReminderItem")

.HasIndex(e => e.Status);

entity.Property(e => e.Id)

.IsRequired()

.ValueGeneratedOnAdd();

entity.Property(e => e.ContactId)

.IsRequired()

.HasMaxLength(50)

.IsUnicode(false);

entity.Property(e => e.TargetDate)

.IsRequired()

.HasColumnType("datetimeoffset(7)");

entity.Property(e => e.Message)

.IsRequired()

.HasMaxLength(200)

.IsUnicode(true);

entity.Property(e => e.Status)

.HasColumnName("StatusId")

.IsRequired()

.HasConversion(

new EnumToNumberConverter<ReminderItemStatus, int>());

entity.Property(e => e.CreatedDate)

.HasColumnType("datetimeoffset(7)")

.HasDefaultValueSql("sysdatetimeoffset()")

.ValueGeneratedOnAdd();

entity.Property(e => e.UpdatedDate)

.HasColumnType("datetimeoffset(7)")

.HasDefaultValueSql("sysdatetimeoffset()")

.ValueGeneratedOnAddOrUpdate();

});

}

}

По ходу написания рассказываем, почему именно так :)

### Особенности при описании ReminderItemStatus

Особенно нужно уделить внимание конвертации данных для enum ReminderItemStatus.

Поскольку теперь нет необходимости лазить в базу руками, нет необходимости вручную и следить за целостностью базы. Можно переложить это на плечи EF Core. Таким образом, нам будет не нужна вторая таблица со словарём статусов — EF Core будет сам конвертировать значения энумерейшна в число (а ещё можно в строку). Мы выбираем вариант преобразования в число, так как работать это будет быстрее, меньше места под индекс, а индекс нам нужен, так как мы будем делать выборку с учётом статусов.

### Связываем контекст БД с классом реализации интерфейса IReminderStrorage

Теперь давайте посмотрим на зависимости.

Наш контекст ReminderStorageContext будет использоваться наследником интерфейса IReminderStorage классом EntityFrameworkReminderStorage.

Соответственно, определяться он должен уже в рантайме и хардкодить строку подключения к БД внутри ReminderStorageContext было бы неправильно. Она должна приходить из настроек приложения нашего Web API.

Таким образом, давайте сделаем у класс ReminderStorageContext конструктор с параметром:

public ReminderStorageContext(DbContextOptions<ReminderStorageContext> options)

: base(options)

{

}

Теперь при создании контекста мы всегда должны передавать ему уже настроенный типизированный экземпляр класса настроек (в котором уже будут заданы строка подключения и другие параметры).

А приходить они должны будут из класса EntityFrameworkReminderStorage. Создадим у него внутренний член билдера этих настроек для хранения:

private readonly DbContextOptionsBuilder<ReminderStorageContext> \_builder;

Мы объявили его как private readonly, так что инициализация предполагается в конструкторе. А что нам реально нужно в конструкторе для нашего класса EntityFrameworkReminderStorage? Конечно, строка подключения!

public EntityFrameworkReminderStorage(string connectionString)

{

\_builder = new DbContextOptionsBuilder<ReminderStorageContext>()

.UseSqlServer(connectionString);

}

### int Count (проверка работоспособности идеи)

Шаблон использования нашего билдера будет следующий:

1. Мы создаём внутри конструкции using экземпляр контекста,
2. Работаем с ним
3. В конце метода (по выходу из блока using) контекст освобождается.

Давайте опробуем наш подход на примере свойства Count:

public int Count

{

get

{

using (var context = new ReminderStorageContext(\_builder.Options))

{

return context.ReminderItems.Count();

}

}

}

### Добавляем логирование

Теперь хочется сказать пару слов про логирование.

К сожалению, использовать красивый способ через DI для веб-приложений тут не получится, так как по сути, мы не регистрируем в нашем веб-сервисе DbContext, мы изолируем его, прячем за фасадом класса EntityFrameworkReminderStorage. И это, к сожалению, означает, что нам (в текущей версии EF Core) будут недоступны фичи подключения логирования “из-коробки”. Поэтому мы добавим собственную (устаревшую, но работоспособную) имплементацию, как мы делали на прошлом уроке. Добавляем в наш класс EntityFrameworkReminderStorage статическую фабрику (помним, статические члены — в самом верху):

public static readonly LoggerFactory MyConsoleLoggerFactory =

new LoggerFactory(new[]

{

#pragma warning disable CS0618 // Type or member is obsolete

new ConsoleLoggerProvider(

(category, level) =>

category == DbLoggerCategory.Database.Command.Name

&& level == LogLevel.Information,

true)

#pragma warning restore CS0618 // Type or member is obsolete

});

И в конструкторе, где мы конструируем наш \_builder добавим следующие строки, чтобы включить логирование в консоли:

public EntityFrameworkReminderStorage(

string connectionString**,**

**bool enableLogging = false**)

{

\_builder = new DbContextOptionsBuilder<ReminderStorageContext>()

.UseSqlServer(connectionString);

**if (enableLogging)**

**{**

**\_builder**

**.UseLoggerFactory(MyConsoleLoggerFactory)**

**.EnableSensitiveDataLogging(true);**

**}**

}

В принципе, здесь осталось только реализовать интерфейса используя наш контекст:

### 

### Guid Add(ReminderItemRestricted reminder)

Видно, что в параметре он принимает объект класса ReminderItemRestricted, а мы оперируем сущностями класса ReminderItemDto. При этом мы должны создать экземпляр ReminderItemDto на базе ReminderItemRestricted.

Логично добавить в наш класс ReminderItemDto конструктор (кроме дефолтного) у которого аргументом будет ReminderItemRestricted:

public class ReminderItemDto

{

...

public ReminderItemDto() {}

public ReminderItemDto(ReminderItemRestricted restricted)

{

ContactId = restricted.ContactId;

TargetDate = restricted.Date;

Message = restricted.Message;

Status = restricted.Status;

}

}

Теперь, собственно, имплементация метода Add:

public Guid Add(ReminderItemRestricted reminder)

{

var dto = new ReminderItemDto(reminder);

using (var context = new ReminderStorageContext(\_builder.Options))

{

context.ReminderItems.Add(dto);

context.SaveChanges();

return dto.Id;

}

}

### 

### ReminderItem Get(Guid id)

Здесь у нас обратная проблема, в контексте БД мы оперируем классом ReminderItemDto, а на выходе метода от нас ожидается ReminderItem.

Добавим метод приведения к ReminderItem к нашему классу ReminderItemDto:

public class ReminderItemDto

{

...

public ReminderItem ToReminderItem()

{

return new ReminderItem

{

Id = Id,

ContactId = ContactId,

Date = TargetDate,

Message = Message,

Status = Status

};

}

}

Теперь, собственно, имплементация метода Get:

public ReminderItem Get(Guid id)

{

using (var context = new ReminderStorageContext(\_builder.Options))

{

return context.ReminderItems

.FirstOrDefault(r => r.Id == id)

?.ToReminderItem();

}

}

### 

### List<ReminderItem> Get(int count = 0, int startPostion = 0)

public List<ReminderItem> Get(int count = 0, int startPostion = 0)

{

using (var context = new ReminderStorageContext(\_builder.Options))

{

if (count == 0 && startPostion == 0)

{

return context.ReminderItems

.Select(r => r.ToReminderItem())

.ToList();

}

if (count == 0)

{

return context.ReminderItems

.OrderBy(ri => ri.Id)

.Skip(startPostion)

.Select(r => r.ToReminderItem())

.ToList();

}

return context.ReminderItems

.OrderBy(r => r.Id)

.Skip(startPostion)

.Take(count)

.Select(r => r.ToReminderItem())

.ToList();

}

}

### 

### List<ReminderItem> Get(ReminderItemStatus status, int count, int startPostion)

public List<ReminderItem> Get(ReminderItemStatus status, int count, int startPostion)

{

using (var context = new ReminderStorageContext(\_builder.Options))

{

if (count == 0 && startPostion == 0)

{

return context.ReminderItems

.Where(r => r.Status == status)

.Select(r => r.ToReminderItem())

.ToList();

}

if (count == 0)

{

return context.ReminderItems

.Where(r => r.Status == status)

.OrderBy(r => r.Id)

.Skip(startPostion)

.Select(r => r.ToReminderItem())

.ToList();

}

return context.ReminderItems

.Where(r => r.Status == status)

.Skip(startPostion)

.OrderBy(r => r.Id)

.Take(count)

.Select(r => r.ToReminderItem())

.ToList();

}

}

### List<ReminderItem> Get(ReminderItemStatus status)

public List<ReminderItem> Get(ReminderItemStatus status)

{

using (var context = new ReminderStorageContext(\_builder.Options))

{

return context.ReminderItems

.Where(r => r.Status == status)

.Select(r => r.ToReminderItem())

.ToList();

}

}

### 

### bool Remove(Guid id)

public bool Remove(Guid id)

{

using (var context = new ReminderStorageContext(\_builder.Options))

{

var dto = context.ReminderItems.FirstOrDefault(r => r.Id == id);

if (dto == null)

{

return false;

}

context.ReminderItems.Remove(dto);

context.SaveChanges();

return true;

}

}

### void UpdateStatus(IEnumerable<Guid> ids, ReminderItemStatus status)

public void UpdateStatus(IEnumerable<Guid> ids, ReminderItemStatus status)

{

using (var context = new ReminderStorageContext(\_builder.Options))

{

var dtos = context.ReminderItems

.Where(d => ids.Contains(d.Id))

.ToList();

foreach (var dto in dtos)

{

dto.Status = status;

}

context.SaveChanges();

}

}

### void UpdateStatus(Guid id, ReminderItemStatus status)

public void UpdateStatus(Guid id, ReminderItemStatus status)

{

using (var context = new ReminderStorageContext(\_builder.Options))

{

var dto = context.ReminderItems.Find(id);

dto.Status = status;

context.SaveChanges();

}

}

На этом с нашей сборкой DAL мы закончили. Её можно подключать к Reminder.Storage.WebApi и она будет работать, как только мы нацелим её на нужную базу в файле конфигурации.

Однако, БД ещё не создана, мы не создали миграций и не накатили их на реальную БД.

## Reminder.Storage.SqlServer.EF.DbInit

Мы будем делать это в отдельной сборке консольного приложения, так как мы не хотим, чтобы файлы миграций лежали в сборке Reminder.Storage.WebApi (а мы помним, что миграции не могут располагаться в сборке библиотеки классов, работа с миграциями должна происходить из сборки сборки запускаемого типа: либо ASP .NET Core, либо Console Application .NET Core).

Добавляем в проект консольное приложение .NET Core Reminder.Storage.SqlServer.EF.DbInit.

### appsettings.json

Для начала вынесем в конфигурацию приложения строку подключения. Создадим файл appsettings.json:

{

"ConnectionStrings": {

"DefaultConnection":

"Data Source=localhost\\SQLEXPRESS;Initial Catalog=ReminderEF;Integrated Security=true;"

}

}

**!!! Необходимо не забыть** выставить у этого файла свойству Copy to Output Directory значение Copy if newer.

### ConnectionStringFactory.cs

Теперь реализуем класс, который будет читать строку подключения из файла настроек: ConnectionStringFactory.

Поскольку мы будем работать с файлом конфигурации в формате JSON, нам потребуется явно установить NuGet-пакет для этого:

* Microsoft.Extensions.Configuration.Json (в зависимостях есть Microsoft.Extensions.Configuration)

Ну и поскольку мы планируем работать с миграциями EF Core здесь же, нам потребуются ещё 2 NuGet-пакета, сделаем это сразу:

* Microsoft.EntityFrameworkCore.SqlServer
* Microsoft.EntityFrameworkCore.Tools

Всё, реализуем наш класс фабрики строчек подключения:

public class ConnectionStringFactory

{

public const string DbConnectionName = @"DefaultConnection";

public const string SettingFileName = @"appsettings.json";

public static string GetDbConnectionString()

{

IConfiguration config = new ConfigurationBuilder()

.AddJsonFile(SettingFileName)

.Build();

return config.GetConnectionString(DbConnectionName);

}

}

### DesignTimeReminderStorageDbContextFactory.cs

Чтобы работать с миграциями из этого консольного приложения, нам нужно будет добавить немного магии EF Core :)

Нужно создать фабрику наших контекстов для дизайн-тайма, и внутри билдера свойств контекста указать, что миграции необходимо сохранять в этой сборке, не смотря на то, что сам класс контекста описан в другой (по умолчанию EF Core ругается, если контекст и миграции не в одной сборке).

Звучит страшно, выглядит, не лучше, если честно, однако, это последнее занятие, мы уже давно позволяем себе писать код корпоративного уровня:

public class DesignTimeReminderStorageDbContextFactory

: IDesignTimeDbContextFactory<ReminderStorageContext>

{

public ReminderStorageContext CreateDbContext(string[] args)

{

string connectionString = ConnectionStringFactory.GetDbConnectionString();

var migrationAssembly = typeof(Program).GetTypeInfo().Assembly.GetName().Name;

var builder = new DbContextOptionsBuilder<ReminderStorageContext>();

builder.UseSqlServer(

connectionString, ob => ob.MigrationsAssembly(migrationAssembly));

return new ReminderStorageContext(builder.Options);

}

}

# 

# Создаём миграцию InitialCreate

1. Выставляем проект Reminder.Storage.SqlServer.EF.DbInit в качестве запускаемого по-умолчанию (пункт контекстного меню проекта Set as StartUp Project.
2. Открываем Package Manager Console
3. Выбираем в качестве *Default project* в правом верхнем углу наш проект Reminder.Storage.SqlServer.EF.DbInit.
4. Пишем PS-команды для создания миграции (если забыли команды, можно начать с команды вывода справки get-help entityframeworkcore)

PM> Add-Migration InitialCreate

PM> Script-Migration

PM> Update-Database

Смотрим на созданную базу с одной табличкой. Радуемся.

# Подключаем новый DAL к Web API

## Проект Reminder.Storage.WebApi

Удаляем из зависимостей ссылку на проект Reminder.Storage.SqlServer.ADO, вместо этого добавляем ссылку на проект Reminder.Storage.SqlServer.EF.

## Reminder.Storage.WebApi/Startup.cs

Можно удалить более не нужный

using Reminder.Storage.Sql;

и добавить вместо него

using Reminder.Storage.SqlServer.EF;

Затем в методе ConfigureServices:

Заменяем подключение старого DAL новым:

string connectionString = Configuration.GetConnectionString("DefaultConnection");

services.AddSingleton<IReminderStorage>(

new EntityFrameworkReminderStorage(connectionString, true));

## 

## Reminder.Storage.WebApi/appsettings.json

Не забываем поменять имя БД, с которой будем работать в файле настроек нашего веб-приложения. У меня она будет называться ReminderEF.

# Запускаем чат-бот

На этом написание кода закончено.  
Стремиться нужно к результату тут: <https://github.com/ago-cs/cs-course-q4/tree/master/Lessons/36/ClassWork/Final>.

Радуемся или, если не работает, разбираемся в чём дело?

# Домашнее задание

—

1. <http://macode.ru> [↑](#footnote-ref-0)
2. Data Transfer Objects [↑](#footnote-ref-1)