Лекция №9. Базы данных

Базы данных играют ключевую роль в современных приложениях. Они необходимы для хранения, управления и быстрого доступа к большим объемам структурированных данных. В качестве примеров можно привести интернет-магазины, где хранятся данные о товарах и заказах, а также социальные сети, в которых фиксируются данные о пользователях и их постах.

Работа с базами данных предполагает выполнение целого ряда задач. Основными из них являются операции по созданию, чтению, обновлению и удалению данных (так называемые CRUD-операции). Также важно обеспечивать целостность данных, используя механизмы транзакций и различные ограничения.

Безопасность является ещё одной важной составляющей: необходимо реализовывать авторизацию и защищать систему от таких угроз, как SQL-инъекции. **SQL-инъекция** — это тип уязвимости, при котором злоумышленник внедряет вредоносный SQL-код в пользовательский ввод, чтобы получить несанкционированный доступ к данным или изменить их.

В языке программирования C# для работы с базами данных существует несколько технологий. Одной из них является **ADO.NET** — это низкоуровневая библиотека, позволяющая напрямую взаимодействовать с базой данных посредством SQL-запросов. Более современным и удобным решением считается **Entity Framework Core** — высокоуровневая ORM-платформа. Она позволяет работать с базой данных через привычные объекты C#, избавляя от необходимости напрямую писать SQL-код.

ORM (Object-Relational Mapping) — это технология, которая позволяет связать объекты в программном коде с таблицами в базе данных. Например, класс `User` в языке C# может быть напрямую связан с таблицей `Users` в БД.

Использование ORM имеет множество преимуществ.

- Во-первых, оно позволяет существенно сократить объём рутинного кода, так как отпадает необходимость вручную писать SQL-запросы для каждой операции.
- Во-вторых, ORM обеспечивает повышенную безопасность, автоматически параметризуя запросы и тем самым защищая от SQL-инъекций.
- В-третьих, такие системы поддерживают механизм миграций, позволяющий автоматически обновлять структуру базы данных при изменении модели данных в коде.

Однако, у ORM есть и свои недостатки. Основным из них являются накладные расходы на производительность — в некоторых случаях ручной SQL может работать значительно быстрее.

ADO.NET

ADO.NET может использоваться в двух режимах — подключенном и отключенном.

Подключенный режим (Connected Mode):

- Работа с данными через постоянное соединение с БД.
- Используется для операций, требующих мгновенной реакции (например, онлайн-транзакции).
  Его ключевыми компонентами являются:
- `SqlConnection` — управление подключением.
- `SqlCommand` — выполнение SQL-запросов.
- `SqlDataReader` — потоковое чтение данных.

Отключенный режим (Disconnected Mode):

- Загрузка данных в локальное хранилище (`DataSet` или `DataTable`) с последующим разрывом соединения.
- Подходит для работы с данными офлайн (например, мобильные приложения).
  Его ключевыми компонентами являются:
- `SqlDataAdapter` — мост между БД и `DataSet` .
- `DataSet` — контейнер для таблиц, отношений и ограничений.
- `DataTable` — аналог таблицы БД в памяти.

Сравнение режимов:

Критерий	Подключенный	Отключенный
Соединение	Активно во время операции	Закрывается после загрузки данных
Производительность	Высокая (меньше накладных расходов)	Ниже (из-за работы с памятью)
Использование	Частые короткие запросы	Редактирование данных офлайн

Рассмотрим примеры работы с основными классами ADO.NET.
Для установки и управления соединением с БД нужно использовать класс `SqlConnection` .

```
// Строка подключения к БД
string connectionString =
"Server=myServer;Database=myDB;UserId=user;Password=pass;";
// Реализует IDisposable
using (SqlConnection connection = new SqlConnection(connectionString))
```

```
{
    connection.Open();
    // ... операции с БД
} // Автоматическое закрытие соединения при выходе из using
```

Для создания и выполнения SQL-запросов и хранимых процедур нужно использовать класс `SqlCommand`.

```
    // Простой SELECT
    SqlCommand command = new SqlCommand("SELECT * FROM Users", connection);

    // INSERT с параметрами (защита от SQL-инъекций)
    command.CommandText = "INSERT INTO Users (Name, Email) VALUES (@Name,
@Email)";
    command.Parameters.AddWithValue("@Name", "Alice");
    command.Parameters.AddWithValue("@Email", "alice@example.com");
    command.ExecuteNonQuery();
```

Параметры позволяют защитить приложение от SQL-инъекций, они экранируют все спецсимволы в введенных данных (имя, e-mail), делая их безопасными для использования.
Команда может исполнятся с помощью разных методов:

- `ExecuteNonQuery()` — для INSERT, UPDATE, DELETE.
- `ExecuteScalar()` — возвращает одно значение (например, COUNT).
- `ExecuteReader()` — возвращает `SqlDataReader`.

`SqlDataReader` используется для чтения данных в подключенном режиме.

```
// Выполнение команды и получение SqlDataReader
using (SqlDataReader reader = command.ExecuteReader())
{
    // Чтение строк по одной
    while (reader.Read())
    {
        // Чтение значения первого столбца (Id)
        int id = reader.GetInt32(0);
        // Чтение значения второго столбца (Name)
        string name = reader.GetString(1);
        Console.WriteLine($"ID: {id}, Name: {name}");
    }
}
```

Теперь давайте научимся работать в отключенном режиме, с использованием классов `SqlDataAdapter` и `DataSet`

```
// SqlDataAdapter выполняет SQL-запрос и связывает данные между БД и DataSet
SqlDataAdapter adapter = new SqlDataAdapter("SELECT * FROM Products",
connection);
//  DataSet — это контейнер, в который можно загрузить одну или несколько
таблиц
DataSet dataSet = new DataSet();

// Заполняем DataSet таблицей "Products" из БД
// Метод Fill создаёт внутри DataSet объект DataTable с именем "Products" и
копирует туда данные
adapter.Fill(dataSet, "Products");

// Получаем ссылку на таблицу "Products" внутри DataSet
DataTable productsTable = dataSet.Tables["Products"];

// Вносим изменения в память: меняем значение в первом ряду столбца "Price"
productsTable.Rows[0]["Price"] = 100;

// SqlCommandBuilder автоматически создаёт SQL-команды (INSERT, UPDATE,
DELETE) для адаптера
// Это необходимо, чтобы можно было отправить изменения из памяти обратно в
базу
SqlCommandBuilder builder = new SqlCommandBuilder(adapter);

// Метод Update применяет изменения, сделанные в DataSet, к базе данных
// Он использует автоматически сгенерированные команды для обновления
записей
adapter.Update(dataSet, "Products");
```

Рассмотрим на практике, как выполнять основные операции с базой данных с помощью ADO.NET.

Пример 1: CRUD-операции

Для начала создаётся таблица `Users`, в которой будут храниться данные пользователей. Она включает в себя три поля: `Id` (целочисленный первичный ключ с автоинкрементом), `Name` (имя пользователя) и `Email` (электронная почта):

```
CREATE TABLE Users (
    Id INT PRIMARY KEY IDENTITY,
    Name NVARCHAR(50),
    Email NVARCHAR(100)
);
```

Чтобы добавить нового пользователя в эту таблицу из C#-кода, используется `SqlCommand` с параметризованным SQL-запросом. Это позволяет защититься от SQL-инъекций:

```
using (SqlCommand cmd = new SqlCommand(
    "INSERT INTO Users (Name, Email) VALUES (@Name, @Email)",
    connection))
{
    cmd.Parameters.AddWithValue("@Name", "Bob");
    cmd.Parameters.AddWithValue("@Email", "bob@example.com");
    cmd.ExecuteNonQuery();
}
```

Здесь создаётся команда для вставки новой строки, и через параметры передаются значения имени и электронной почты. Метод `ExecuteNonQuery()` выполняет запрос без возврата результата.

Пример 2: Транзакции

Транзакции используются, когда необходимо выполнить несколько операций как единое целое, чтобы гарантировать целостность данных. В следующем примере реализуется перевод средств между двумя аккаунтами:

```
using (SqlTransaction transaction = connection.BeginTransaction())
{
    try
    {
        SqlCommand cmd1 = new SqlCommand(
            "UPDATE Accounts SET Balance = Balance - 100 WHERE Id = 1",
            connection,
            transaction);
        cmd1.ExecuteNonQuery();

        SqlCommand cmd2 = new SqlCommand(
            "UPDATE Accounts SET Balance = Balance + 100 WHERE Id = 2",
            connection,
            transaction);
        cmd2.ExecuteNonQuery();

        transaction.Commit(); // Подтверждение изменений
    }
    catch
    {
        transaction.Rollback(); // Откат при ошибке
    }
}
```

Сначала открывается транзакция. Затем последовательно выполняются две команды: списание со счёта отправителя и зачисление на счёт получателя. Если обе операции проходят успешно, транзакция фиксируется методом `Commit()`. В случае любой ошибки происходит откат (`Rollback()`), и никакие изменения в базу не попадают.

Entity Framework Core

EF Core — это объектно-реляционный маппер (ORM) для C#, который позволяет работать с базой данных через объекты и LINQ, минимизируя необходимость в SQL. Для работы с ним нужно будет установить NuGet-пакет `Microsoft.EntityFrameworkCore`.

Преимущества по сравнению с ADO.NET:

- Меньше ручного кода — SQL-запросы формируются автоматически.
- Безопасность — параметризация запроса встроена по умолчанию.
- Миграции — изменения схемы БД отслеживаются и применяются через код.
- Кроссплатформенность — поддержка SQL Server, PostgreSQL, SQLite и др.

В EF Core можно использовать два подхода: **Code First** и **Database First**.

1. Code First
Создаются C#-классы, которые описывают структуру данных, а база формируется автоматически через миграции. Подходит для новых проектов, где структура БД создаётся «с нуля».

```
// Класс, который будет представлять таблицу Book в базе данных
public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int AuthorId { get; set; }
    public Author Author { get; set; }
}
```

2. Database First
Используется существующая база данных, из которой автоматически генерируются классы. Подходит для работы с готовыми или унаследованными базами.

```
// Команда для генерации классов из существующей базы данных
Scaffold-DbContext "Server=...;Database=Library;"
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

Рассмотрим основные компоненты EF Core.

Ключевой элемент — класс `DbContext`, который управляет подключением, отслеживанием изменений и выполнением операций с БД.

```
// Контекст базы данных, который управляет сущностями и соединением с БД
public class AppDbContext : DbContext
```

```
{
    // Представление таблицы Books и Authors в базе данных
    public DbSet<Book> Books { get; set; }
    public DbSet<Author> Authors { get; set; }

    // Метод для настройки подключения к базе данных
    protected override void OnConfiguring(DbContextOptionsBuilder options)
        // Используем SQL Server и указываем строку подключения
        => options.UseSqlServer("Server=...;Database=Library;");
}
```

Коллекции `DbSet<T>` представляют таблицы базы данных и позволяют выполнять CRUD-операции.

Если в ходе разработки нужно будет изменить структуру БД, но при этом требуется сохранить уже имеющиеся данные, то нам понадобится создать миграцию. **Миграции** фиксируют изменения моделей в виде кода, который обновляет схему БД.

Для создания и применения миграции нам понадобятся следующие команды:

- `Add-Migration InitialCreate` — создание миграции
- `Update-Database` — применение изменений к базе данных

Пример части миграции:

```
// Миграция для создания таблицы Authors
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        // Создаем таблицу Authors с полями Id и Name
        migrationBuilder.CreateTable(
            name: "Authors",
            columns: table => new
            {
                Id = table.Column<int>(nullable: false)
                    .Annotation("SqlServer:Identity", "1, 1"),  // Id с
автоинкрементом
                Name = table.Column<string>(nullable: true)  // Строковое
поле для имени автора
            },
            constraints: table => {
                // Добавляем первичный ключ
                table.PrimaryKey("PK_Authors", x => x.Id);
            });
    }
}
```

Рассмотрим CRUD-операции в EF Core.

1. Добавление (Create):

```
// Используем контекст для добавления нового автора в базу данных
using (var context = new AppDbContext())
{
    // Создаем новый объект автора
    var author = new Author { Name = "Достоевский" };
    // Добавляем его в DbSet Authors
    context.Authors.Add(author);
    // Сохраняем изменения в базе данных
    context.SaveChanges();
}
```

2. Чтение (Read):

```
// Простой запрос для получения всех книг, опубликованных после 2000 года
var books = context.Books
    .Where(b => b.Year > 2000)  // Фильтруем по году
    .ToList();  // Преобразуем в список

// Загрузка связанных данных (например, книг с их авторами)
var authorsWithBooks = context.Authors
    .Include(a => a.Books)  // Загружаем книги вместе с авторами
    .ToList();
```

3. Обновление (Update):

```
// Ищем книгу с Id = 1 и обновляем ее название
var book = context.Books.FirstOrDefault(b => b.Id == 1);
if (book != null)
{
    book.Title = "Новое название";  // Меняем название книги
    context.SaveChanges();  // Сохраняем изменения в базе данных
}
```

4. Удаление (Delete):

```
// Ищем книгу с Id = 1 и удаляем ее
var book = context.Books.FirstOrDefault(b => b.Id == 1);
if (book != null)
{
    context.Books.Remove(book);  // Удаляем книгу
    context.SaveChanges();  // Применяем изменения в базе данных
}
```

Рассмотрим настройку связей между сущностями, использованием атрибутов.

Один-ко-многим (один автор может иметь много книг):

```
// Модель для автора
[Table("Authors")] // Название таблицы
public class Author
{
    [Key] // первичный ключ
    public int Id { get; set; }

    [Required] // не null
    [MaxLength(200)] // длина строки максимум 200
    public string Name { get; set; }

    // Навигационное свойство
    // Список книг, которые принадлежат автору
    public List<Book> Books { get; set; }
}

// Модель для книги
[Table("Books")]
public class Book
{
    [Key]
    public int Id { get; set; }

    [Required]
    [MaxLength(500)]
    public string Title { get; set; }

    [Range(1450, 2100)] // год публикации в диапазоне
    public int Year { get; set; }

    // Внешний ключ для связи с автором
    public int AuthorId { get; set; }
    // Навигационное свойство для связи с автором
    public Author Author { get; set; }
}
```

Многие-ко-многим (много книг ↔ много жанров):

```
// Модель для книги
public class Book
{
    //...
    // Связь с жанрами
    public List<BookGenre> BookGenres { get; set; }
```

```
}

// Модель для жанра
[Table("Genres")]
public class Genre
{
    [Key]
    public int Id { get; set; }

    [Required]
    [MaxLength(100)]
    public string Name { get; set; }

    public List<BookGenre> BookGenres { get; set; } = new();
}

// Промежуточная таблица
[Table("BookGenres")]
public class BookGenre
{
    // составной ключ
    [Key, Column(Order = 0)]
    [ForeignKey(nameof(Book))]
    public int BookId { get; set; }
    public Book Book { get; set; }

    [Key, Column(Order = 1)]
    [ForeignKey(nameof(Genre))]
    public int GenreId { get; set; }
    public Genre Genre { get; set; }
}
```

Атрибуты (DataAnnotations) в C# — это специальные метки (аннотации) в квадратных скобках, которые «приписывают» классам или свойствам дополнительную информацию. EF Core читает эти атрибуты и на их основе:

1. Формирует схему БД (имена таблиц/столбцов, типы, ограничения).
2. Проводит валидацию данных при сохранении.
3. Настраивает связи между сущностями.

Несколько самых важных атрибутов:

- `[Table("Name")]`
  Задаёт явное имя таблицы в базе. Без него EF бы использовал имя класса.
- `[Key]`
  Помечает свойство как первичный ключ таблицы.

- `[Required]`
  Гарантирует, что в столбец нельзя записать `NULL`; при попытке сохранить `null` будет ошибка.
- `[MaxLength(n)]` / `[StringLength(n)]`
  Ограничивает максимальную длину строкового поля (VARCHAR(n)).
- `[Range(min, max)]`
  Валидирует числовое свойство: значение должно быть в указанном диапазоне.
- `[ForeignKey(nameof(Other))]`
  Устанавливает внешнее ключевое свойство, связывая с другой сущностью.
- `[InverseProperty("…")]`
  Явно связывает навигационные свойства двух классов, когда EF не может однозначно определить пару.

Fluent API в EF Core — это способ настройки моделей и их отношений между собой с помощью кода, а не через атрибуты. Он позволяет более гибко и точно управлять поведением сущностей и их связями, например, указывать типы внешних ключей, настраивать каскадное удаление, уникальные ограничения и другие параметры. Все эти настройки выполняются в методе `OnModelCreating` в контексте `DbContext`.

```
// Использование Fluent API для настройки отношений между сущностями
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // Настроим связь "Один-ко-многим" между Book и Author
    modelBuilder.Entity<Book>()
        .HasOne(b => b.Author)  // Книга имеет одного автора
        .WithMany(a => a.Books)  // Автор может иметь много книг
        .HasForeignKey(b => b.AuthorId);  // Внешний ключ для связи с
автором
}
```

Советы по работе с базами данных в EF Core

1. Используйте EF Core

EF Core — это мощный инструмент для работы с базами данных, который предоставляет удобный и безопасный способ взаимодействия с данными через объекты и LINQ, минимизируя количество кода, который необходимо писать вручную. Использование EF Core позволяет вам сосредоточиться на бизнес-логике, а не на реализации SQL-запросов.

2. Используйте асинхронные методы

Асинхронные операции позволяют улучшить производительность и избежать блокировки потоков, особенно при работе с большими объемами данных или в высоконагруженных приложениях.

```
public async Task<List<Book>> GetBooksAsync()
{
    return await _context.Books.ToListAsync();
}

public async Task AddBookAsync(Book book)
{
    await _context.Books.AddAsync(book);
    await _context.SaveChangesAsync();
}
```

3. Не создавайте слишком много контекстов

Создание нового экземпляра `DbContext` на каждый запрос или операцию может привести к утечкам памяти и снижению производительности. Вместо этого используйте **единственный экземпляр** контекста в рамках одного запроса или операции.

4. Используйте транзакции для обеспечения атомарности

Транзакции гарантируют, что все изменения, сделанные в базе данных, будут либо подтверждены (commit), либо отменены (rollback) в случае ошибки. Это особенно важно при выполнении нескольких операций, которые должны быть выполнены как единое целое.

```
using (var transaction = _context.Database.BeginTransaction())
{
    try
    {
        // Выполнение нескольких операций
        _context.Books.Add(new Book { Title = "New Book" });
        _context.SaveChanges();

        // Подтверждение транзакции
        transaction.Commit();
    }
    catch (Exception)
    {
        // Откат транзакции при ошибке
        transaction.Rollback();
    }
}
```

5. Пример класса CRUD-сервиса

Создание универсального CRUD-сервиса для работы с сущностями, например, с книгами:

```
public class BookService
{
    private readonly AppDbContext _context;

    public BookService(AppDbContext context)
    {
        _context = context;
    }

    // Create
    public async Task AddBookAsync(Book book)
    {
        await _context.Books.AddAsync(book);
        await _context.SaveChangesAsync();
    }

    // Read
    public async Task<List<Book>> GetAllBooksAsync()
    {
        return await _context.Books.ToListAsync();
    }

    // Update
    public async Task UpdateBookAsync(int id, string title)
    {
        var book = await _context.Books.FindAsync(id);
        if (book != null)
        {
            book.Title = title;
            await _context.SaveChangesAsync();
        }
    }

    // Delete
    public async Task DeleteBookAsync(int id)
    {
        var book = await _context.Books.FindAsync(id);
        if (book != null)
        {
            _context.Books.Remove(book);
            await _context.SaveChangesAsync();
        }
    }
}
```

---

6. Подключение к разным базам данных

EF Core поддерживает подключение к множеству СУБД, таких как SQL Server, PostgreSQL, SQLite и другие. Чтобы подключиться к разной базе данных, необходимо указать соответствующий провайдер в методе `OnConfiguring` или в файле конфигурации. Так же нужно будет установить необходимые NuGet-пакеты.

- Для SQL Server `Microsoft.EntityFrameworkCore.SqlServer`.
- Для PostgreSQL `Npgsql.EntityFrameworkCore.PostgreSQL`.
- Для SQLite `Microsoft.EntityFrameworkCore.Sqlite`.
- Для миграций и инструментов `Microsoft.EntityFrameworkCore.Tools`.

Пример подключения к SQL Server:

```
protected override void OnConfiguring(DbContextOptionsBuilder options)
{
    options.UseSqlServer("Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;");
}
```

Пример подключения к PostgreSQL:

```
protected override void OnConfiguring(DbContextOptionsBuilder options)
{

options.UseNpgsql("Host=myserver;Database=mydb;Username=mylogin;Password=myp
ass");
}
```

Для подключения к SQLite:

```
protected override void OnConfiguring(DbContextOptionsBuilder options)
{
    options.UseSqlite("Data Source=mydatabase.db");
}
```

При подключении к разным СУБД важно учитывать особенности их настройки и возможные ограничения, такие как различия в типах данных, индексах и поддержке транзакций.