

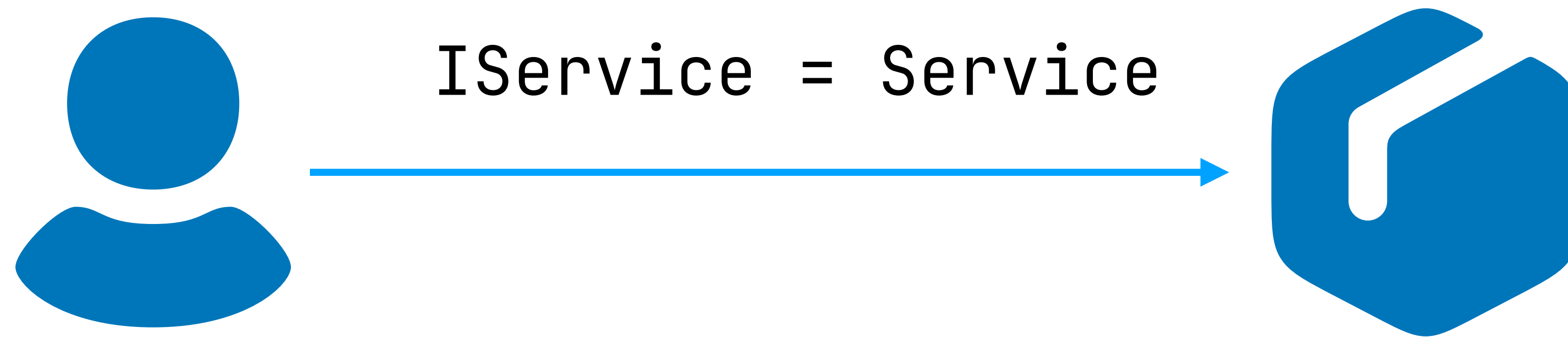
# управление данными в микросервисах на C#

инструменты работы с данными в .NET

dependency injection

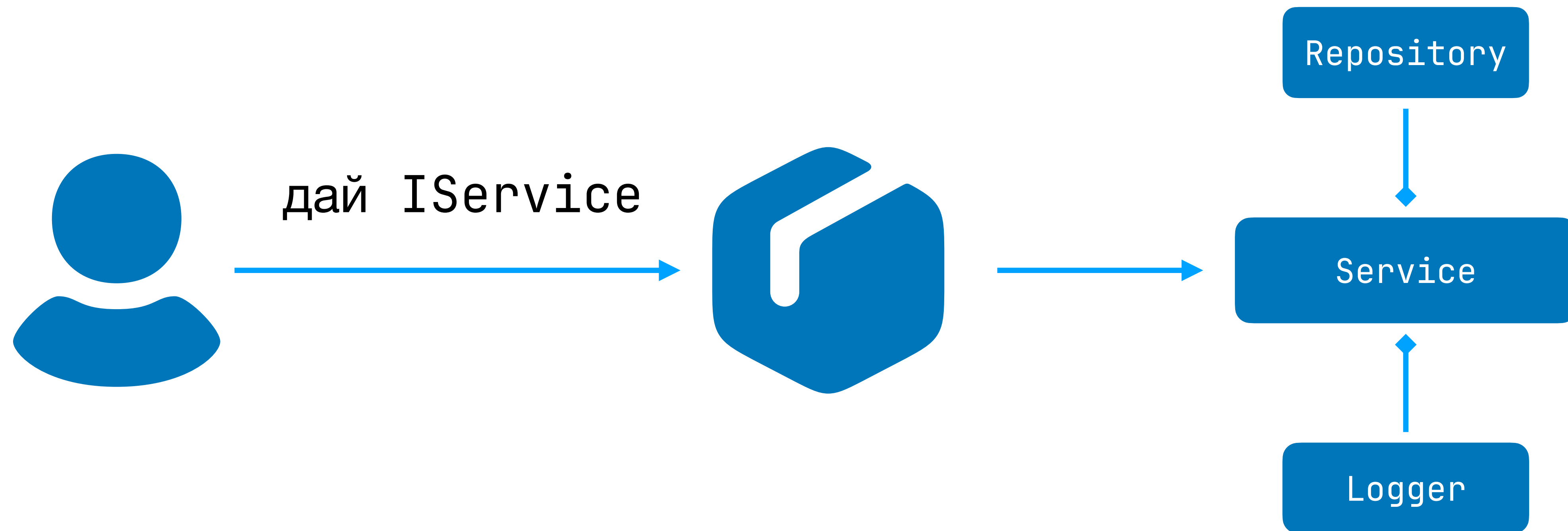
# dependency injection

концепция



# dependency injection

## концепция



# dependency injection

## Microsoft.Extensions.DependencyInjection

- стандартная реализация DI контейнера в .NET от Microsoft
- используется по умолчанию в ASP.NET
- сама по себе независима, может быть использована отдельно
- зачастую сокращают до MSDI

# dependency injection

## **IServiceProvider**

```
object? serviceObj = serviceProvider.GetService(typeof(IService));  
IService? serviceNullable = serviceProvider.GetService<IService>();  
IService service = serviceProvider.GetRequiredService<IService>();
```

# dependency injection

## service collection

- представлена типами `IServiceCollection` и `ServiceCollection`
- является коллекцией дескрипторов сервисов (`ServiceDescriptor`)
- выполняет роль билдера для сервис провайдера

# dependency injection

## service type

- определяет тип описываемой зависимости
- может быть как абстракцией, так и конкретным типом
- соответствует тому типу, который мы передаем в сервис провайдер, чтобы получить объект зависимости



# dependency injection

## implementation type

- определяет тип, реализующий зависимость
- должен отличаться от `ServiceType`, если `ServiceType` абстрактный
- если в дескрипторе указан `ImplementationType`, DI контейнер попытается создать объект данного типа

# dependency injection

## implementation factory

- определяет делегат с сигнатурой `Func<IServiceProvider, object>`
- делегат используется в качестве фабрики для создания реализации
- используется когда необходимо добавить какую-то кастомную логику создания объекта зависимости

# dependency injection

## implementation instance

- определяет объект, который будет использоваться в качестве зависимости

# dependency injection

## lifetimes

singleton

scoped

transient

# dependency injection

## singleton lifetime

- зависимости с таким лайфтаймом будут созданы DI контейнером единожды
- создаются лениво, при первом запросе
- все следующие запросы будут получать ранее созданный объект

# dependency injection

## transient lifetime

- transient – недолговечный, кратковременный
- на каждый запрос на получение зависимости с таким лайфтаймом DI контейнер будет создавать новый объект

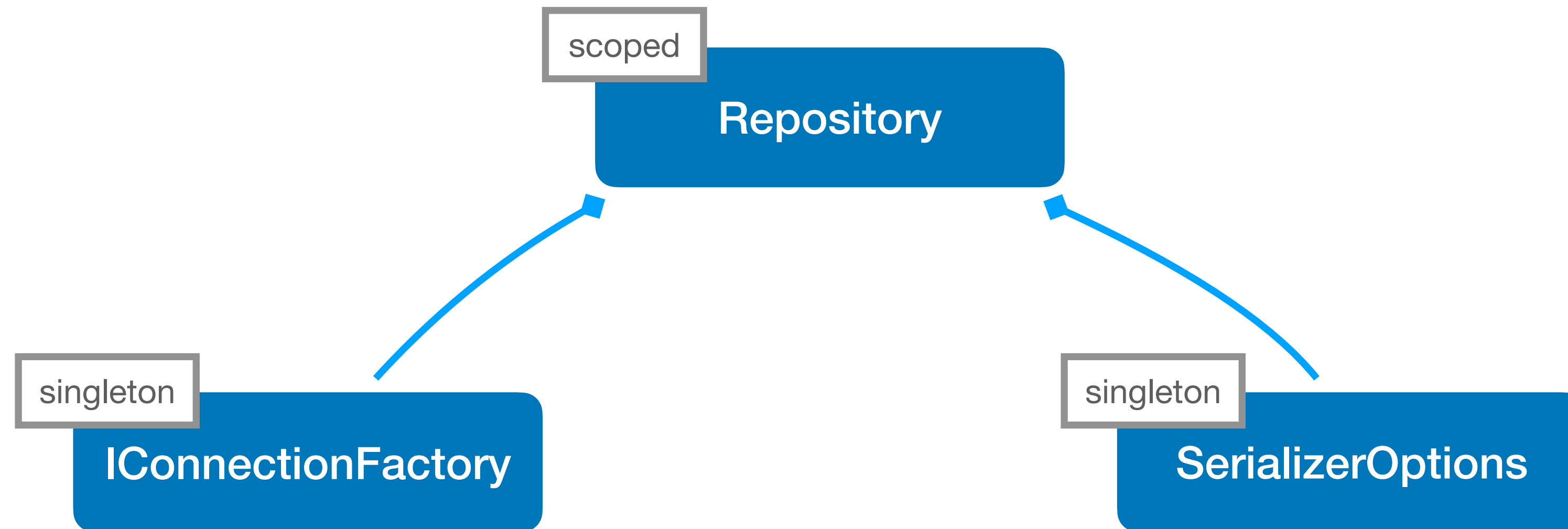
# dependency injection

## scopes

- зачастую в приложениях требуется изолированность данных между различными операциям
- имея только Singleton и Transient лайфтаймы задача этой изоляции перешла бы на наш код
- MSDI позволяет создавать контексты, имеющие свои “синглтоны”

# dependency injection

## scoped lifetime

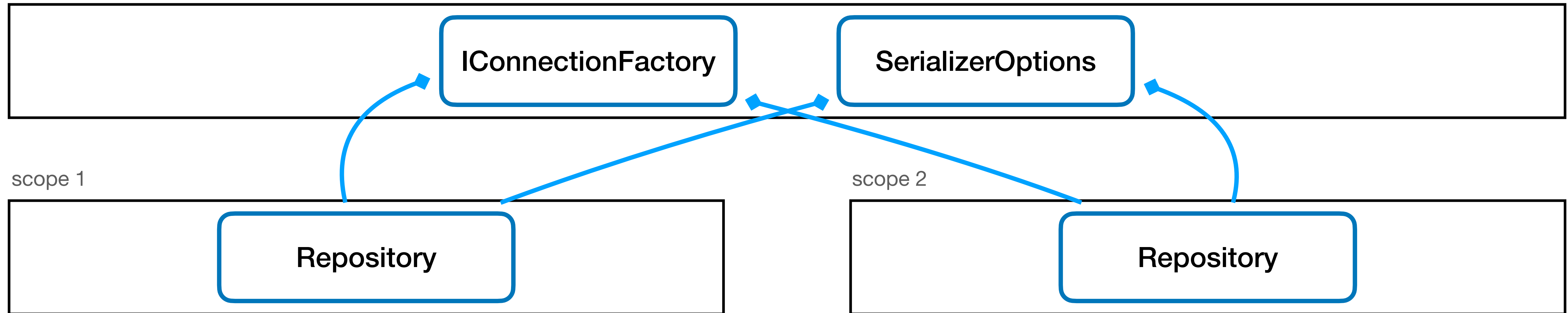




# dependency injection

## scoped lifetime

global service provider



# dependency injection

## scopes

```
async Task ExecuteScopedOperationAsync(IServiceProvider serviceProvider)
{
    await using AsyncServiceScope scope = serviceProvider.CreateAsyncScope();
    IScopedDependency dependency = scope.ServiceProvider.GetRequiredService<IScopedDependency>();

    await dependency.ExecuteOperationAsync();
}
```

# dependency injection

## регистрация зависимостей

```
serviceCollection.AddSingleton<ISingletonDependency, SingletonImplementation>();  
serviceCollection.AddScoped<IScopedDependency, ScopedImplementation>();  
serviceCollection.AddTransient<ITransientDependency, TransientImplementation>();
```

# dependency injection

## регистрация зависимостей

```
void ConfigureCollection(IServiceCollection collection)
{
    collection.AddScoped<IDependency, Dependency1>();
    collection.AddScoped<IDependency, Dependency2>();
}

void Execute(IServiceScope scope)
{
    IEnumerable<IDependency> dependencies = scope.ServiceProvider.GetRequiredService<IEnumerable<IDependency>>();

    foreach (IDependency dependency in dependencies)
    {
        dependency.DoSomething();
    }
}
```

# dependency injection

## работа с существующими зависимостями

заменяет уже существующую зависимость

```
serviceCollection.Replace(ServiceDescriptor.Singleton<ILibraryDependency, CustomImplementation>());
```

добавляет зависимость если она не была зарегистрирована ранее

```
serviceCollection.TryAddSingleton<ILibraryDependency, CustomImplementation>();
```

добавляет множественную зависимость если эта конкретная реализация не была зарегистрирована ранее

```
var descriptor = ServiceDescriptor.Singleton<ILibraryDependency, CustomImplementation>();
```

```
serviceCollection.TryAddEnumerable(descriptor);
```

```
serviceCollection.TryAddEnumerable(descriptor);
```

# dependency injection

## создание ServiceProvider

```
var serviceCollection = new ServiceCollection();
```

```
// ...
```

```
using ServiceProvider serviceProvider = serviceCollection.BuildServiceProvider();
```

```
using (IServiceScope scope = serviceProvider.CreateScope())
```

```
{
```

```
    // ...
```

```
}
```

# serialization

# serialization

## System.Text.Json


- представлен классом JsonSerializer
- `string JsonSerializer.Serialize(T)`
- `T? JsonSerializer.Deserialize<T>(string)`



# serialization

## JsonSerializer.Serialize

```
var data = new { id = 1, name = "aboba" };  
string serialized = JsonSerializer.Serialize(data);  
Console.WriteLine(serialized);
```



```
{  
  "id": 1,  
  "name": "aboba"  
}
```

# serialization

## JsonSerializer.Deserialize

```
var serialized = """  
{"Id":1,"Name":"aboba"}  
""";
```

```
Model? model = JsonSerializer.Deserialize<Model>(serialized);
```

```
Console.WriteLine(model);
```



Model { Id = 1, Name = aboba }

# serialization

## полиморфная сериализация

```
public record Base(int Id);  
  
public record Derived(int Id, string Name) : Base(Id);
```

```
Base model = new Derived(1, "aboba");
```

```
string serialized = JsonSerializer.Serialize(model);  
Console.WriteLine(serialized);
```



```
{  
  "Id": 1  
}
```

# serialization

## полиморфная сериализация

```
Base model = new Derived(1, "aboba");
```

```
string serialized = JsonSerializer.Serialize(model, model.GetType());  
Console.WriteLine(serialized);
```



```
{  
  "Name": "aboba",  
  "Id": 1  
}
```

# serialization

## полиморфная сериализация

```
[JsonDerivedType(typeof(Derived), typeDiscriminator: nameof(Derived))]  
public record Base(int Id);  
  
public record Derived(int Id, string Name) : Base(Id);
```

```
Base model = new Derived(1, "aboba");
```

```
string serialized = JsonSerializer.Serialize(model);  
Console.WriteLine(serialized);
```



```
{  
  "$type": "Derived",  
  "Name": "aboba",  
  "Id": 1  
}
```

# serialization

## полиморфная сериализация

```
var serialized = ""{"$type":"Derived","Name":"aboba","Id":1}"";  
Base? model = JsonSerializer.Deserialize<Base>(serialized);
```

```
Console.WriteLine(model?.GetType().Name);
```

—————→ Derived

# serialization

## конвертеры

- позволяют изменять реализацию записи сериализатором конкретных значений
- реализуются как наследник от класса `JsonConverter<T>`, где `T` – конвертируемый тип

# serialization

## конвертеры

```
public readonly record struct MyId(long Value);
```

```
public sealed record Entity(MyId Id);
```

```
var id = new MyId(1);  
var entity = new Entity(id);  
  
string serialized = JsonSerializer.Serialize(entity);  
Console.WriteLine(serialized);
```



```
{  
  "Id": {  
    "Value": 1  
  }  
}
```



# serialization

## конвертеры

```
public class MyIdConverter : JsonConverter<MyId>
{
    public override MyId Read(ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions options)
    {
        long value = reader.GetInt64();
        return new MyId(value);
    }

    public override void Write(Utf8JsonWriter writer, MyId value, JsonSerializerOptions options)
    {
        writer.WriteNumberValue(value.Value);
    }
}
```

# serialization

## конвертеры

```
[JsonConverter(typeof(MyIdConverter))]  
public readonly record struct MyId(long Value);
```

```
string serialized = JsonSerializer.Serialize(entity);  
Console.WriteLine(serialized);
```

```
var options = new JsonSerializerOptions  
{  
    Converters = { new MyIdConverter() },  
};
```

```
string serialized = JsonSerializer.Serialize(entity, options);  
Console.WriteLine(serialized);
```



{  
 "Id": 1  
}

работа с HTTP

# работа с HTTP

## HttpClient

```
using var client = new HttpClient
{
    BaseAddress = new Uri("https://api.github.com"),
    Timeout = TimeSpan.FromSeconds(2),
};

client.DefaultRequestHeaders.UserAgent.ParseAdd("PostmanRuntime/7.42.0");
```

# работа с HTTP

## HttpClient

- имеет методы для выполнения различных HTTP запросов (GET, POST, ...)
- эти методы возвращают HttpResponseMessage
  - код ответа
  - тело ответа
  - хедеры

```
HttpResponseMessage response = await client.GetAsync("users/ronimizy");
```

# работа с HTTP

## HttpContent

- представляет данные из тела ответа или запроса
- имеет методы для получения данных в различном виде
  - строка
  - массив байтов
  - ПОТОК

# работа с HTTP

## HttpContent

```
string contentString = await response.Content.ReadAsStringAsync();  
byte[] contentBytes = await response.Content.ReadAsByteArrayAsync();  
await using Stream contentStream = await response.Content.ReadAsStreamAsync();
```

```
using System.Net.Http.Json;  
GithubUser? user = await response.Content.ReadFromJsonAsync<GithubUser>();
```

# работа с HTTP

## HttpRequestMessage

```
using var message = new HttpRequestMessage(HttpMethod.Get, "users/ronimizy");  
message.Headers.Add("Accept", "application/xml");  
  
HttpResponseMessage response = await client.SendAsync(message);  
  
Console.WriteLine(await response.Content.ReadAsStringAsync());
```



```
{  
  "message": "Unsupported 'Accept' header: 'application/xml'. Must accept 'application/json'.",  
  "documentation_url": "https://docs.github.com/v3/media",  
  "status": "415"  
}
```



# работа с HTTP

## HttpClientFactory

- при создании `HttpClient` через конструктор, он занимает сокет системы, и возвращает его не сразу, даже после `Dispose`
- при частом создании `HttpClient`, эти сокеты могут закончиться
- при создании клиентов через `HttpClientFactory`, эти сокеты переиспользуются
- `Microsoft.Extensions.Http`

# работа с HTTP

## HttpClientFactory

```
var collection = new ServiceCollection();  
collection.AddHttpClient();
```

```
ServiceProvider provider = collection.BuildServiceProvider();  
IHttpClientFactory factory = provider.GetRequiredService<IHttpClientFactory>();
```

```
using HttpClient client = factory.CreateClient();
```

# работа с HTTP

## Refit

- написание HTTP-клиентов вручную – крайне рутинный процесс
- существует много библиотек упрощающих этот процесс
  - генерация по OpenAPI спецификации (большинство из них генерируют не особо дружелюбный код)
- библиотека Refit позволяет самостоятельно определять интерфейсы для HTTP-клиентов, реализуя сами HTTP вызовы

# работа с HTTP

## Refit

```
public sealed record GithubUser(long Id, string Login);

public interface IGithubClient
{
    [Get("users/{login}")]
    Task<GithubUser> GetUserAsync(string login, CancellationToken cancellationToken);
}
```

# работа с HTTP

## Refit.HttpClientFactory

```
collection
    .AddRefitClient<IGithubClient>()
    .ConfigureHttpClient(
        client => client.BaseAddress = new Uri("https://api.github.com"));
```

**Для меньшей связанности внешнего API  
и типов вашего приложения стоит делать  
отдельные обёртки над Refit клиентами**



**ВАЖНО**

# конфигурации в .NET

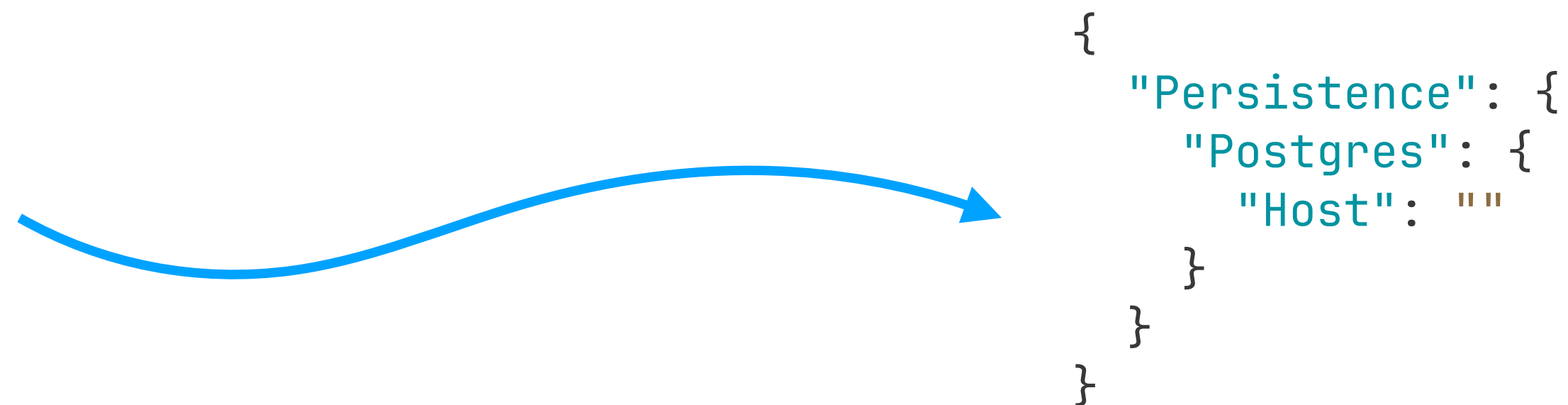
**Microsoft.Extensions.Configuration**

# Microsoft.Extensions.Configuration

## работа с конфигурациями

- конфигурации хранятся в виде набора пар ключ-значение
- библиотека интерпретирует эти данные в “объектной” форме
- если в ключе содержится символ “:”, библиотека будет интерпретировать его как разделитель пути до свойства какого-то “объекта”

Persistence:Postgres:Host





# Microsoft.Extensions.Configuration

## получение данных из конфигураций

```
IConfigurationSection postgresSection = configuration.GetSection("Persistence:Postgres");  
  
string? host = postgresSection.GetSection("Host").Value;  
int port = postgresSection.GetValue<int>("Port");
```

# Microsoft.Extensions.Configuration

## мапинг конфигураций на объекты

```
IConfigurationSection section = configuration.GetSection("Persistence:Postgres");

var postgresConfiguration = new PostgresConfiguration(
    Host: string.Empty,
    Port: 0,
    Username: string.Empty,
    Password: string.Empty);

section.Bind(postgresConfiguration);
```

При маппинге конфигураций на  
объекты не используются JSON  
сериализаторы

**ВАЖНО**



# Microsoft.Extensions.Configuration

## настройка конфигураций

- в метод Add у объекта IConfigurationBuilder добавляется объект IConfigurationSource
- IConfigurationSource является фабрикой для создания объекта IConfigurationProvider
- IConfigurationProvider это то, откуда библиотека будет получать пары ключ-значения составляющие конфигурацию

# Microsoft.Extensions.Configuration

## IConfigurationProvider

```
public interface IConfigurationProvider
{
    bool TryGet(string key, out string? value);

    void Set(string key, string? value);

    IChangeToken GetReloadToken();

    void Load();

    IEnumerable<string> GetChildKeys(
        IEnumerable<string> earlierKeys,
        string? parentPath);
}
```

# Microsoft.Extensions.Configuration

## реализация кастомного провайдера

```
public sealed class PostgresConfigurationProvider : ConfigurationProvider
{
    public void OnConfigurationUpdated(PostgresConfiguration configuration)
    {
        const string prefix = "Persistence:Postgres";

        Data["{prefix}:Host"] = configuration.Host;
        Data["{prefix}:Port"] = configuration.Port.ToString();
        Data["{prefix}:Username"] = configuration.Username;
        Data["{prefix}:Password"] = configuration.Password;

        OnReload();
    }
}
```

# Microsoft.Extensions.Configuration

## реализация кастомного провайдера

```
public sealed class PostgresConfigurationSource : IConfigurationSource
{
    private readonly PostgresConfigurationProvider _provider;

    public PostgresConfigurationSource(PostgresConfigurationProvider provider)
    {
        _provider = provider;
    }

    public IConfigurationProvider Build(IConfigurationBuilder builder) => _provider;
}
```

# Microsoft.Extensions.Configuration

## реализация кастомного провайдера

```
var provider = new PostgresConfigurationProvider();
configurationBuilder.Add(new PostgresConfigurationSource(provider));

var configuration = new PostgresConfiguration(string.Empty, 0, string.Empty, string.Empty);
configurationRoot.GetSection("Persistence:Postgres").Bind(configuration);
Console.WriteLine(configuration);

provider.OnConfigurationUpdated(new PostgresConfiguration("localhost", 5432, "postgres", "postgres"));

configurationRoot.GetSection("Persistence:Postgres").Bind(configuration);
Console.WriteLine(configuration);
```



```
PostgresConfiguration { Host = , Port = 0, Username = , Password = }
PostgresConfiguration { Host = localhost, Port = 5432, Username = postgres, Password = postgres }
```



# Microsoft.Extensions.Configuration

## ConfigurationManager

```
var configurationManager = new ConfigurationManager();  
  
IConfigurationBuilder configurationBuilder = configurationManager;  
IConfigurationRoot configurationRoot = configurationManager;
```

# существующие провайдеры

## переменные окружения

- добавляются через пакет `Microsoft.Extensions.Configuration.EnvironmentVariables`
- ключи в самих переменных окружения должны использовать символы “\_\_” вместо “:”

```
configurationManager.AddEnvironmentVariables();
```

# существующие провайдеры JSON файлы

- добавляются через пакет  
`Microsoft.Extensions.Configuration.Json`
- этот провайдер следит за обновлениями JSON файла и перезагружает конфигурацию

```
configurationManager.AddJsonFile("config.json");
```

# существующие провайдеры

## user secrets

- добавляются через пакет `Microsoft.Extensions.Configuration.UserSecrets`
- позволяет использовать конфигурации из user secrets
- полезны для локальной разработки

```
configurationManager.AddUserSecrets<Program>();
```

# Microsoft.Extensions.Configuration

приоритет провайдеров



# конфигурации в .NET

**Microsoft.Extensions.Options**

# Microsoft.Extensions.Options

## правильное использование конфигураций в коде

- библиотека позволяет автоматизировать маппинг конфигураций на объекты
- используется вместе с DI контейнером, регистрируется через вызов метода `AddOptions` над `IServiceCollection`
- к типам на которые мапятся конфигурации есть особые требования
  - наличие пустого конструктора
  - публичные сеттеры для свойств на которые будут мапиться значения

# Microsoft.Extensions.Options

## ТИПЫ ДЛЯ ОПШЕНОВ

```
public class PostgresOptions
{
    public string Host { get; set; } = string.Empty;

    public int Port { get; set; }

    public string Username { get; set; } = string.Empty;

    public string Password { get; set; } = string.Empty;
}
```

```
collection.AddOptions<PostgresOptions>().Configure(o => o.Port = 5432);
```



# Microsoft.Extensions.Options

## IOptions<>

- синглтон от мира опшенов
- значение получается единожды и сохраняется на протяжении выполнения программы

```
var options = serviceProvider.GetService<IOptions<PostgresOptions>>();  
PostgresOptions optionsValue = options.Value;
```

# Microsoft.Extensions.Options

## IOptionsSnapshot<>

- scoped от мира опшенов
- значение получается единожды и сохраняется в рамках одного скоупа

```
var options = serviceProvider.GetRequiredService<IOptionsSnapshot<PostgresOptions>>();  
PostgresOptions optionsValue = options.Value;
```

# Microsoft.Extensions.Options

## IOptionsMonitor<>

- transient от мира опшенов
- при каждом запросе значения опшенов оно собирается заново из конфигурации

```
var options = serviceProvider.GetRequiredService<IOptionsMonitor<PostgresOptions>>();  
PostgresOptions optionsValue = options.CurrentValue;
```

# Microsoft.Extensions.Options

## options + configuration

- `OptionsBuilder` позволяет привязать какую-либо секцию к конкретным опшенам
- можно упростить привязку, указывая только путь до секции используя пакет `Microsoft.Extensions.Options.ConfigurationExtensions`
  - в таком случае, в DI контейнере должен быть зарегистрирован `IConfigurationRoot`

```
collection.AddOptions<PostgresOptions>().Bind(configuration.GetSection("Persistence:Postgres"));  
collection.AddOptions<PostgresOptions>().BindConfiguration("Persistence:Postgres");
```

работа с  
базами данных в .NET

# работа с базами данных в .NET

## System.Data

- набор общих абстракций для работы с базами данных
  - подключения
  - запросы
  - параметры
  - транзакции

# работа с базами данных в .NET

## DbConnection

```
async Task DoSomething(DbConnection connectionParameter)
{
    await using DbConnection connection = connectionParameter;

    await connection.OpenAsync();
    await connection.CloseAsync();
}
```

# работа с базами данных в .NET

## DbCommand

- определяет запрос к базе данных
- хранит текст запроса
- хранит параметры запроса
- хранит коннекшен над которым будет выполнен запрос
- можно выполнить через `ExecuteReaderAsync` и `ExecuteNonQueryAsync`



# работа с базами данных в .NET

## DbCommand

```
async Task HandleCommandNonQueryAsync(  
    DbCommand command,  
    CancellationToken cancellationToken)  
{  
    int rows = await command.ExecuteNonQueryAsync(cancellationToken);  
    Console.WriteLine($"Command affected {rows} rows");  
}
```

# работа с базами данных в .NET

## DbDataReader

- однонаправленный итератор по строкам запроса
- переключается на следующую строку вызовом `ReadAsync`
- имеет методы получения данных текущей строки
  - `GetInt32`
  - `GetString`
  - ...
- реализует `IAsyncDisposable`

# работа с базами данных в .NET

## DbDataReader

```
async IEnumerable<Model> HandleCommandAsync(  
    DbCommand command,  
    [EnumeratorCancellation] CancellationToken cancellationToken)  
{  
    await using DbDataReader reader = await command.ExecuteReaderAsync(cancellationToken);  
  
    while (await reader.ReadAsync(cancellationToken))  
    {  
        yield return new Model(  
            Id: reader.GetInt32("id"),  
            Name: reader.GetString("name"));  
    }  
}
```

# работа с базами данных в .NET

## транзакции

```
async Task ExecuteTransactionalOperationAsync(DbConnection connection)
{
    await using DbTransaction transaction = connection.BeginTransaction(IsolationLevel.ReadCommitted);

    // ...

    await transaction.CommitAsync();
}
```

# работа с базами данных в .NET

## транзакции

```
async Task ExecuteTransactionalOperationAsync(DbConnection connection)
{
    using var transaction = new TransactionScope(
        TransactionScopeOption.Required,
        new TransactionOptions { IsolationLevel = IsolationLevel.ReadCommitted },
        TransactionScopeAsyncFlowOption.Enabled);

    // ...

    transaction.Complete();
}
```

# работа с базами данных в .NET

## npqsql

- Postgres драйвер для .NET
- реализует абстракции из System.Data для работы с Postgres

# работа с базами данных в .NET

## NpgsqlDataSource

- фабрика для соединений
- создаётся через NpgsqlDataSourceBuilder
  - конфигурирует маппинги
  - конфигурирует логгирование
  - ...

```
var connectionString = "Host=localhost;Username=postgres;Password=postgres;Database=postgres";  
var dataSourceBuilder = new NpgsqlDataSourceBuilder(connectionString);  
  
await using NpgsqlDataSource dataSource = dataSourceBuilder.Build();
```

# работа с базами данных в .NET

## **NpgsqlConnection и пулинг соединений**

- подключение к Postgres базе
- привязано к физическому соединению, но не наоборот
- Npgsql реализует пулинг физических соединений, несколько объектов `NpgsqlConnection`, созданных в разное время, могут использовать одно физическое соединение



# работа с базами данных в .NET

## NpgsqlCommand

```
const string sql = """
select *
from users
where age > :age
""";

await using NpgsqlCommand command = new NpgsqlCommand(sql, connection)
{
    Parameters =
    {
        new NpgsqlParameter("age", 18),
    },
};
```

# работа с базами данных в .NET

## NpgsqlCommand

```
const string sql = """
select *
from users
where age > :age
""";
```

```
await using NpgsqlCommand command = connection.CreateCommand();
command.CommandText = sql;
command.Parameters.Add(new NpgsqlParameter("age", 18));
```

# работа с базами данных в .NET

## МАППИНГ ТИПОВ

```
create type status as enum
(
    'pending',
    'processing',
    'succeeded',
    'failed'
);

create type work_item as
(
    name    text,
    status  status
)
```

```
public enum Status
{
    Pending,
    Processing,
    Succeeded,
    Failed,
}

public record WorkItem(
    string Name,
    Status Status);
```

# работа с базами данных в .NET

## МАППИНГ ТИПОВ

```
dataSourceBuilder.MapEnum<Status>(pgName: "status");  
dataSourceBuilder.MapComposite<WorkItem>(pgName: "work_item");
```

# работа с базами данных в .NET

## миграции

# работа с базами данных в .NET

## миграции

- обеспечивают соответствие схемы базы данных вашему коду
- представляют собой набор SQL скриптов отражающие историческое изменение схемы базы данных
- миграции не должны изменяться, любые изменения схемы данных требуют создания новых миграций

# FluentMigrator

## определение миграций

```
public class InitialMigration : Migration
{
    public override void Up() { }

    public override void Down() { }
}
```

# FluentMigrator

## определение миграций

```
public class InitialMigration : Migration
{
    public override void Up()
    {
        Create.Table("users")
            .WithColumn("user_id").AsInt64().PrimaryKey().Identity()
            .WithColumn("user_name").AsString().NotNullable();
    }

    public override void Down()
    {
        Delete.Table("users");
    }
}
```



# FluentMigrator

## определение миграций

```
public class InitialMigration : IMigration
{
    public void GetUpExpressions(IMigrationContext context)
    {
        context.Expressions.Add(new ExecuteSqlStatementExpression
        {
            SqlStatement = """
create table users
(
    user_id bigint primary key generated always as identity,
    user_name text not null
);
""",
        });
    }

    public void GetDownExpressions(IMigrationContext context)
    {
        context.Expressions.Add(new ExecuteSqlStatementExpression { SqlStatement = """drop table users;""" });
    }

    public string ConnectionString => throw new NotSupportedException();
}
```

# FluentMigrator

## определение миграций

```
[Migration(version: 1727972936, description: "Initial migration")]  
public class InitialMigration : IMigration
```

# FluentMigrator

## определение миграций

```
#pragma warning disable SA1649
```

```
[Migration(version: 1727972936, description: "Initial migration")]  
public class InitialMigration : IMigration
```

# FluentMigrator

## хранение миграций

```
create table "VersionInfo"  
(  
    "Version"      bigint not null,  
    "AppliedOn"    timestamp,  
    "Description"  varchar(1024)  
);
```

# FluentMigrator

## выполнение миграций

```
serviceCollection
    .AddFluentMigratorCore()
    .ConfigureRunner(runner => runner
        .AddPostgres()
        .WithGlobalConnectionString("Host=localhost;Username=postgres;Password=postgres")
        .WithMigrationsIn(typeof(IMigrationAssemblyMarker).Assembly));
```

# FluentMigrator

## выполнение миграций

```
await using (AsyncServiceScope scope = serviceProvider.CreateAsyncScope())
{
    IMigrationRunner runner = scope.ServiceProvider.GetRequiredService<IMigrationRunner>();
    runner.MigrateUp();
}
```

# FluentMigrator

## выполнение миграций

```
await using (AsyncServiceScope scope = serviceProvider.CreateAsyncScope())
{
    IMigrationRunner runner = scope.ServiceProvider.GetRequiredService<IMigrationRunner>();
    runner.MigrateDown(1727972936);
}
```

# работа с базами данных в .NET

реализация стандартных операций репозиториев



# репозитории

- тип, ответственный за реализацию операций с базами данных
- реализует
  - вставку
  - обновление
  - удаление
  - поиск данных

# репозитории

## множественная вставка/обновление данных

- множественная обработка данных помогает в оптимизации
- помогает снизить затраты на
  - сериализацию
  - обращения к сетевым ресурсам

# множественная вставка/обновление данных

## кастомные типы данных

```
create type user_insert_model as (name text, age int);
```

```
public readonly record struct UserInsertModel(string Name, int Age);
```

# множественная вставка/обновление данных

## кастомные типы данных

```
async Task InsertUsersAsync(IEnumerable<User> users, CancellationToken cancellationToken)
{
    UserInsertModel[] models = users.Select(x => new UserInsertModel(x.Name, x.Age)).ToArray();

    const string sql = """
insert into users (user_name, user_age)
select name, age from unnest(:users::user_insert_model[]);
""";







    await using NpgsqlConnection connection = await dataSource.OpenConnectionAsync(cancellationToken);

    await using var command = new NpgsqlCommand(sql, connection)
    {
        Parameters = { new NpgsqlParameter("users", models) }
    };






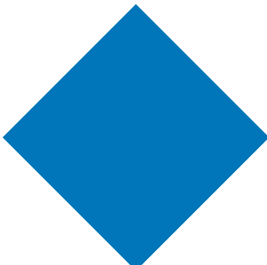
    await command.ExecuteNonQueryAsync(cancellationToken);
}
```

# множественная вставка/обновление данных

unnest

unnest([,  
[)



col1	col2
	
	
	

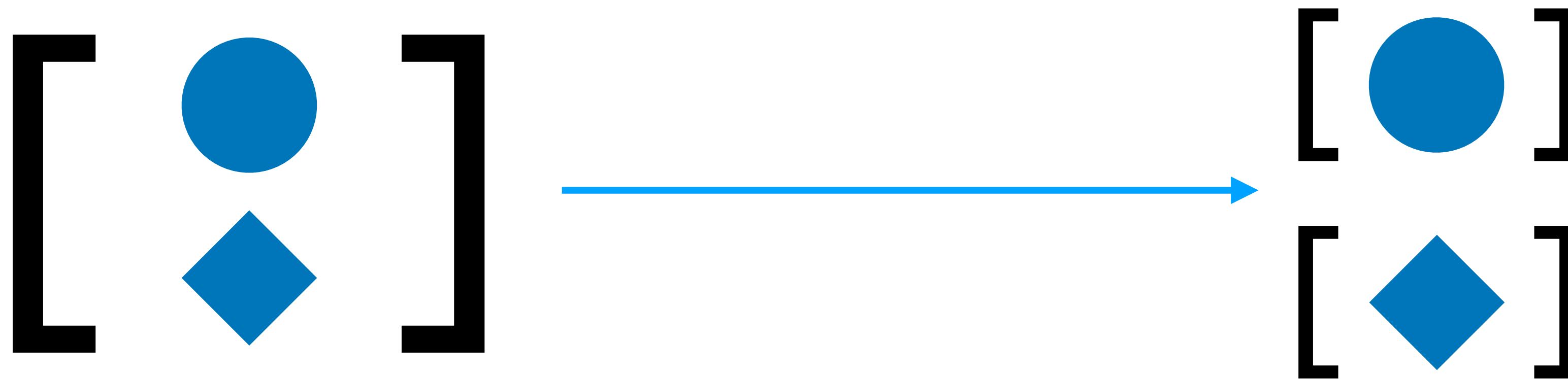
# **множественная вставка/обновление данных**

## **кастомные типы данных**

- требуют миграций для создания
- требуют миграций для изменения
- не гибкое решение

# множественная вставка/обновление данных

## пересборка объектов



# множественная вставка/обновление данных

## пересборка объектов

```
async Task InsertUsersAsync(IReadOnlyCollection<User> users, CancellationToken cancellationToken)
{
    const string sql = """
insert into users (user_name, user_age)
select name, age from unnest(:names, :ages) as source(name, age);
""";

    await using NpgsqlConnection connection = await dataSource.OpenConnectionAsync(cancellationToken);

    await using var command = new NpgsqlCommand(sql, connection)
    {
        Parameters =
        {
            new NpgsqlParameter("names", users.Select(x => x.Name).ToArray()),
            new NpgsqlParameter("ages", users.Select(x => x.Age).ToArray()),
        },
    };

    await command.ExecuteNonQueryAsync(cancellationToken);
}
```



# множественная вставка/обновление данных

## обновление данных

```
async Task UpdateUsersAsync(IReadOnlyCollection<User> users, CancellationToken cancellationToken)
{
    const string sql = """
update users
set user_name = source.name,
    user_age = source.age
from (select * from unnest(:ids, :names, :ages)) as source(id, name, age)
where user_id = source.id
""";

    await using NpgsqlConnection connection = await dataSource.OpenConnectionAsync(cancellationToken);

    await using var command = new NpgsqlCommand(sql, connection)
    {
        Parameters =
        {
            new NpgsqlParameter("ids", users.Select(x => x.Id).ToArray()),
            new NpgsqlParameter("names", users.Select(x => x.Name).ToArray()),
            new NpgsqlParameter("ages", users.Select(x => x.Age).ToArray()),
        },
    };

    await command.ExecuteNonQueryAsync(cancellationToken);
}
```

# ПОИСК ДАННЫХ

## модели запросов

```
public record UserQuery(  
    long[] Ids,  
    string? NamePattern,  
    int? MinAge,  
    int Cursor,  
    int PageSize);
```

```
public interface IUserRepository  
{  
    IEnumerable<User> QueryAsync(UserQuery query, CancellationToken cancellationToken);  
}
```

# ПОИСК ДАННЫХ

## фильтрации по модели запроса

```
select user_id, user_name, user_age
from users
where
    (user_id > :cursor)
    and (cardinality(:ids) = 0 or user_id = any (:ids))
    and (:name_pattern is null or user_name like :name_pattern)
    and (:min_age is null or user_age > :min_age)
order by user_id
limit :page_size;
```

# ПОИСК ДАННЫХ

## фильтрации по модели запроса

```
async IEnumerable<User> QueryUsersAsync(
    UserQuery query,
    [EnumeratorCancellation] CancellationToken cancellationToken)
{
    ...

    await using NpgsqlConnection connection = await dataSource.OpenConnectionAsync(cancellationToken);

    await using var command = new NpgsqlCommand(sql, connection)
    {
        Parameters =
        {
            new NpgsqlParameter("ids", query.Ids),
            new NpgsqlParameter("name_pattern", query.NamePattern),
            new NpgsqlParameter("min_age", query.MinAge),
            new NpgsqlParameter("cursor", query.Cursor),
            new NpgsqlParameter("page_size", query.PageSize),
        },
    };

    ...
}
```

# ПОИСК ДАННЫХ

## фильтрации по модели запроса

```
async IEnumerable<User> QueryUsersAsync(
    UserQuery query,
    [EnumeratorCancellation] CancellationToken cancellationToken)
{
    ...

    await using NpgsqlDataReader reader = await command.ExecuteReaderAsync(cancellationToken);

    while (await reader.ReadAsync(cancellationToken))
    {
        yield return new User(
            reader.GetInt64("user_id"),
            reader.GetString("user_name"),
            reader.GetInt32("user_age"));
    }
}
```