Распределенные информационно-аналитические системы

Практическое занятие № 13. «Работа с базой данных.

CORS и кросс-доменные запросы»

Профессор кафедры КБ-2: д.т.н. Шатовкин Р.Р.

Учебные вопросы:

1. Работа с базой данных.

1.1. Подключение Entity Framework Core.

1.2. Основные операции с данными в Entity Framework Core.

2. CORS и кросс-доменные запросы.

2.1. Подключение CORS в приложении.

2.2. Конфигурация CORS.

2.3. Политики CORS.

2.4. Глобальная и локальная настройка CORS.

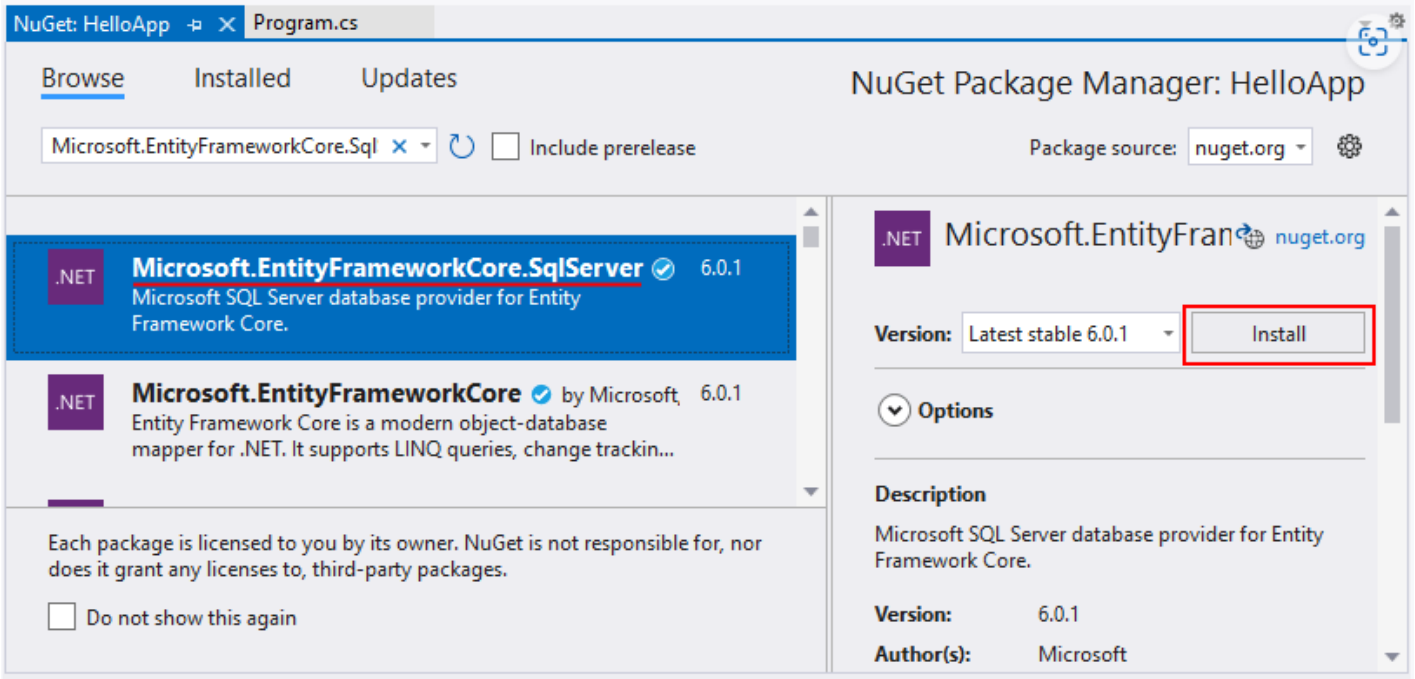
1. Работа с базой данных

1.1. Подключение Entity Framework Core

Entity Framework представляет ORM-решение, которое позволяет автоматически связать классы языка C# с таблицами в базе данных. **Entity Framework Core** поддерживает большинство популярных СУБД, таких как **MS SQL Server**, **SQLite**, **MySQL**, **Postres**. В данном случае мы будем работать через **Entity Framework Core** с базами данных в **MS SQL Server**.

Детально ознакомиться с работой с **Entity Framework Core** можно в соответствующем руководстве, здесь же мы сосредоточимся прежде всего на тех моментах, которые характерны для веб-приложения **ASP.NET Core**.

Для работы с базами данных **MS SQL Server** через **Entity Framework Core** необходим пакет **Microsoft.EntityFrameworkCore.SqlServer**. По умолчанию он отсутствует в проекте, поэтому его надо добавить, например, через пакетный менеджер **Nuget**:



Далее добавим в проект класс, которые будет представлять данные. Пусть он будет называться **User** и будет иметь следующий код:

```
public class User
{
    public int Id { get; set; }
    public string Name { get; set; } = ""; // имя пользователя
    public int Age { get; set; } // возраст пользователя
}
```

Этот класс представляет те объекты, которые будут храниться в базе данных.

Для взаимодействия с базой данных через **Entity Framework Core** необходим контекст данных – класс, унаследованный от класса **Microsoft.EntityFrameworkCore.DbContext**. Поэтому добавим в проект новый класс, который назовем **ApplicationContext** (название класса контекста произвольное):

```
using Microsoft.EntityFrameworkCore;
public class ApplicationContext : DbContext
{
    public DbSet<User> Users { get; set; } = null!;
    public ApplicationContext(DbContextOptions<ApplicationContext> options)
        : base(options)
    {
        Database.EnsureCreated();   // создаем базу данных при первом обращении
    }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<User>().HasData(
                new User { Id = 1, Name = "Tom", Age = 37 },
                new User { Id = 2, Name = "Bob", Age = 41 },
                new User { Id = 3, Name = "Sam", Age = 24 }
        );
    }
}
```

Свойство **DbSet** представляет собой коллекцию объектов, которая сопоставляется с определенной таблицей в базе данных. То есть свойство **Users** будет представлять таблицу, в которой будут храниться объекты **User.**

Класс **DbSet**, как и другие типы, является ссылочным. А, начиная с C# 10 и .NET 6 автоматически применяется функциональность ссылочных **nullable**-типов. И переменные/свойства тех типов, которые не являются **nullable**, следует инициализировать некотором значением перед их использованием. Чтобы выйти из этой ситуации мы можем инициализировать свойство с помощью выражения **null!**, которое говорит, что данное свойство в принципе не будет иметь значение **null**. Потому что в реальности конструктор базового класса **DbContext** гарантирует, что все свойства типа **DbSet** будут инициализированы и соответственно в принципе не будут иметь значение **null**.

В конструкторе класса **ApplicationContext** через параметр **options** будут передаваться настройки контекста данных. А в самом конструкторе с помощью вызова **Database.EnsureCreated()** по определению модели будет создаваться база данных (если она отсутствует).

Кроме того, в методе **OnModelCreating()** настраивается некоторая начальная конфигурация модели. В частности, с помощью метода **modelBuilder.Entity<User>().HasData()** в базу данных добавляются три начальных объекта

Конфигурация и строка подключения

Чтобы подключаться к базе данных, нам надо задать параметры подключения. Это можно сделать либо в коде C#, либо в файле конфигурации. Выберем второй способ. Для этого изменим файл **appsettings.json**. По умолчанию он содержит только настройки логгирования:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Теперь изменим его, добавив определение строки подключения:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=applicationdb;Trusted_Connection=True;"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

В данном случае мы будем использовать упрощенный движок базы данных **LocalDB**, который представляет легковесную версию **SQL Server Express**, предназначенную специально для разработки приложений. Об этом говорит параметр **Server=(localdb)\\mssqllocaldb**. Ну а сама база данных будет называться **applicationdb**.

Добавление контекста данных EF Core в качестве сервиса

Контекст данных, через который идет взаимодействие с базой данных, передается в приложение в качестве сервиса через механизм внедрения зависимостей. Поэтому для работы с **Entity Framework Core** необходимо добавить класс контекста данных в коллекцию сервисов приложения:

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder();

// получаем строку подключения из файла конфигурации
string connection = builder.Configuration.GetConnectionString("DefaultConnection");

// добавляем контекст ApplicationContext в качестве сервиса в приложение
builder.Services.AddDbContext<ApplicationContext>(options => options.UseSqlServer(connection));

var app = builder.Build();

app.Run();
```

Для использования контекста данных ему надо передать строку подключения, которая выше была определена в файл конфигурации **appsettings.json**. Поэтому сначала считываем строку подключения под названием **"DefaultConnection"**:

```
string connection = builder.Configuration.GetConnectionString("DefaultConnection");
```

Для получения строки конфигурации для объекта **IConfiguration**, который представляет конфигурацию, определен метод расширения **GetConnectionString()**, в который передается название строки подключения.

Для добавления контекста данных в качестве сервиса у объекта коллекции сервисов **IServiceCollection** определен метод **AddDbContext()**, который типизируется классом контекста данных:

```
builder.Services.AddDbContext<ApplicationContext>(options => options.UseSqlServer(connection));
```

Этот метод в качестве параметра принимает делегат, который настраивает подключение. В частности, с помощью вызова **options.UseSqlServer()** настраивается подключение к серверу **MS SQL Server**, а в качестве параметра в этот вызов передается строка подключения.

Обращение к базе данных

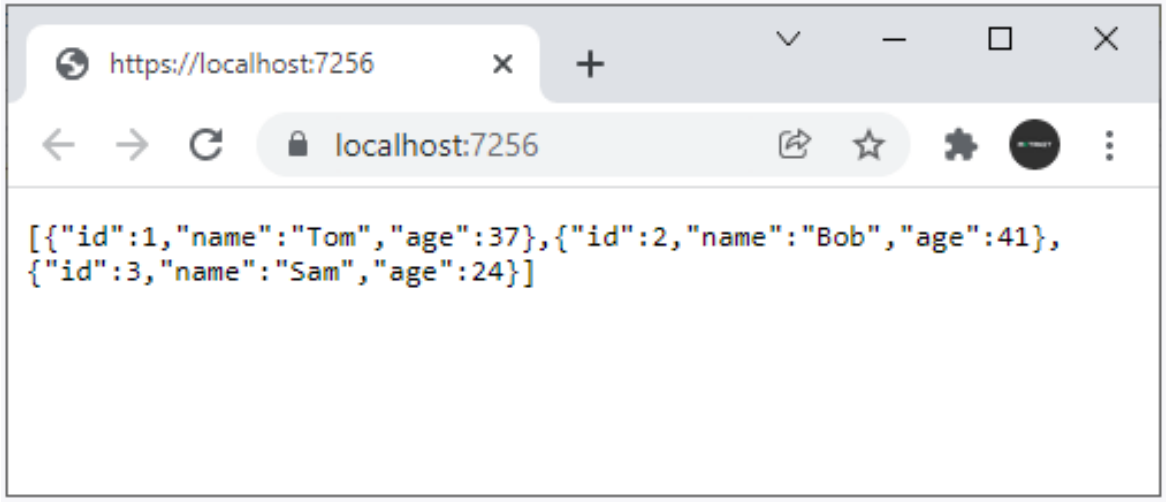
Подключив контекст данных, мы можем с его помощью обращаться к базе данных. Например, в качестве теста получим добавляемые по умолчанию данные:

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder();
string connection = builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationContext>(options => options.UseSqlServer(connection));

var app = builder.Build();

// получение данных
app.MapGet("/", (ApplicationContext db) => db.Users.ToList());

app.Run();
```

Поскольку контекст данных добавлен в сервисы, то мы можем его получить в обработчике конечной точки:

```
app.MapGet("/", (ApplicationContext db) => db.Users.ToList());
```

В данном случае клиенту отправляется список добавленных в БД по умолчанию объектов **User** в формате **JSON**:



1.2. Основные операции с данными в Entity Framework Core

Рассмотрим, как мы можем выполнять основные операции с данными в приложении **ASP.NET Core**. За основу возьмем проект **Web API**, описанный в статье Пример приложения **Web API**, но теперь добавим в него взаимодействие с базой данных.

Пусть у нас проекте определен класс **User**, который будет представлять данные:

```
public class User
{
    public int Id { get; set; }
    public string Name { get; set; } = ""; // имя пользователя
    public int Age { get; set; } // возраст пользователя
}
```

Для взаимодействия с базой данных **MS SQL Server** в качестве контекста данных определим следующий класс **ApplicationContext**:

```
using Microsoft.EntityFrameworkCore;
public class ApplicationContext : DbContext
{
    public DbSet<User> Users { get; set; } = null!;
    public ApplicationContext(DbContextOptions<ApplicationContext> options)
        : base(options)
    {
        Database.EnsureCreated();   // создаем базу данных при первом обращении
    }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<User>().HasData(
                new User { Id = 1, Name = "Tom", Age = 37 },
                new User { Id = 2, Name = "Bob", Age = 41 },
                new User { Id = 3, Name = "Sam", Age = 24 }
        );
    }
}
```

Далее в файле Program.cs определим основной код приложения, который будет обрабатывать запросы и подключаться к базе данных:

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder();
string connection = "Server=(localdb)\\mssqllocaldb;Database=applicationdb;Trusted_Connection=True;";
builder.Services.AddDbContext<ApplicationContext>(options => options.UseSqlServer(connection));

var app = builder.Build();

app.UseDefaultFiles();
app.UseStaticFiles();

app.MapGet("/api/users", async (ApplicationContext db) => await db.Users.ToListAsync());

app.MapGet("/api/users/{id:int}", async (int id, ApplicationContext db) =>
{
    // получаем пользователя по id
    User? user = await db.Users.FirstOrDefaultAsync(u => u.Id == id);

    // если не найден, отправляем статусный код и сообщение об ошибке
    if (user == null) return Results.NotFound(new { message = "Пользователь не найден" });

    // если пользователь найден, отправляем его
    return Results.Json(user);
});

app.MapDelete("/api/users/{id:int}", async (int id, ApplicationContext db) =>
{
    // получаем пользователя по id
    User? user = await db.Users.FirstOrDefaultAsync(u => u.Id == id);

    // если не найден, отправляем статусный код и сообщение об ошибке
    if (user == null) return Results.NotFound(new { message = "Пользователь не найден" });

```

```
    // если пользователь найден, удаляем его
    db.Users.Remove(user);
    await db.SaveChangesAsync();
    return Results.Json(user);
});

app.MapPost("/api/users", async (User user, ApplicationContext db) =>
{
    // добавляем пользователя в массив
    await db.Users.AddAsync(user);
    await db.SaveChangesAsync();
    return user;
});

app.MapPut("/api/users", async (User userData, ApplicationContext db) =>
{
    // получаем пользователя по id
    var user = await db.Users.FirstOrDefaultAsync(u => u.Id == userData.Id);

    // если не найден, отправляем статусный код и сообщение об ошибке
    if (user == null) return Results.NotFound(new { message = "Пользователь не найден" });

    // если пользователь найден, изменяем его данные и отправляем обратно клиенту
    user.Age = userData.Age;
    user.Name = userData.Name;
    await db.SaveChangesAsync();
    return Results.Json(user);
});

app.Run();
```

Вначале добавляем класс **ApplicationContext** в сервисы приложения:

```
var builder = WebApplication.CreateBuilder();
string connection = "Server=(localdb)\\mssqllocaldb;Database=applicationdb;Trusted_Connection=True;";
builder.Services.AddDbContext<ApplicationContext>(options => options.UseSqlServer(connection));
```

Далее после создания объекта **WebApplication** подключаем функциональность статических файлов:

```
app.UseDefaultFiles();
app.UseStaticFiles();
```

Затем с помощью методов **MapGet**/**MapPost**/**MapPut**/**MapDelete** определяется набор конечных точек, которые будут обрабатывать разные типы запросов.

Первая конечная точка обрабатывает запрос типа **GET** по маршруту "api/users":

```
app.MapGet("/api/users", async (ApplicationContext db) => await db.Users.ToListAsync());
```

Поскольку выше в коде контекст данных **ApplicationContext** был добавлен в качестве сервиса, то мы можем его получить через параметр обработчика конечной точки и через полученный контекст данных получить из БД список объектов **User** и отправить их клиенту.

Когда клиент обращается к приложению для получения одного объекта по **id** в запрос типа GET по адресу "api/users/{id}", то срабатывает другая конечная точка:

```
app.MapGet("/api/users/{id:int}", async (int id, ApplicationContext db) =>
{
    // получаем пользователя по id
    User? user = await db.Users.FirstOrDefaultAsync(u => u.Id == id);

    // если не найден, отправляем статусный код и сообщение об ошибке
    if (user == null) return Results.NotFound(new { message = "Пользователь не найден" });

    // если пользователь найден, отправляем его
    return Results.Json(user);
});
```

Здесь через параметр **id** получаем из пути запроса идентификатор объекта **User** и по этому идентификатору ищем нужный объект в базе данных, используя контекст данных **ApplicationContext**. Если объект по **Id** не был найден, то возвращаем с помощью метода **Results.NotFound()** статусный код 404 с некоторым сообщением в формате **JSON**. Если объект найден, то с помощью метода **Results.Json()** отправляет найденный объект клиенту.

При получении запроса типа **DELETE** по маршруту "/api/users/{id}" срабатывает другая конечная точка:

```
app.MapDelete("/api/users/{id:int}", async (int id, ApplicationContext db) =>
{
    // получаем пользователя по id
    User? user = await db.Users.FirstOrDefaultAsync(u => u.Id == id);

    // если не найден, отправляем статусный код и сообщение об ошибке
    if (user == null) return Results.NotFound(new { message = "Пользователь не найден" });

    // если пользователь найден, удаляем его
    db.Users.Remove(user);
    await db.SaveChangesAsync();
    return Results.Json(user);
});
```

Здесь если объект по **Id** не найден в базе данных, то отправляем статусный код 404. Если же объект найден, то с помощью вызова **db.Users.Remove(user)** указываем, что данный объект надо удалить из БД. А с помощью последующего вызова **db.SaveChangesAsync()** сохраняем изменения в базу данных (то есть удаляем объект). И в конце посылаем удаленный объект клиенту.

При получении запроса с методом POST по адресу "/api/users" срабатывает следующая конечная точка:

```
app.MapPost("/api/users", async (User user, ApplicationContext db) =>
{
    // добавляем пользователя в массив
    await db.Users.AddAsync(user);
    await db.SaveChangesAsync();
    return user;
});
```

Здесь мы ожидаем, что в запросе типа **POST** клиент будет передавать на сервер данные, которые соответствуют определению типа **User**. И поэтому инфраструктура **ASP.NET Core** сможет автоматически создать из них объект **User**. И этот объект мы сможем получить в качестве параметра в обработчике конечной точки вместе с сервисом **ApplicationContext**.

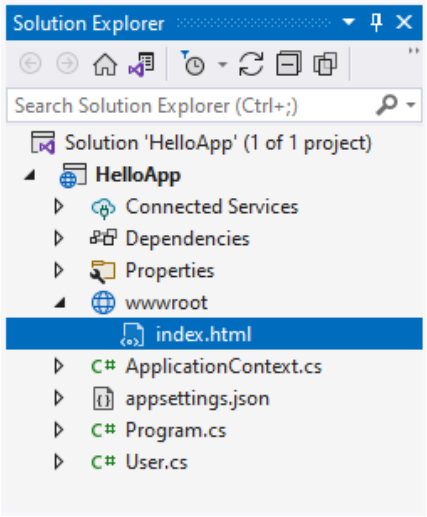
После получения объекта **User** с помощью метода **db.Users.AddAsync(user)** указываем, что данный объект надо добавить в БД. А с помощью последующего вызова **db.SaveChangesAsync()** сохраняем изменения в базу данных (то есть добавляем объект). После добавления отправляем объект **User** обратно клиенту.

Если приложению приходит **PUT**-запрос по адресу "/api/users", то запрос обрабатывает последняя конечная точка:

```
app.MapPut("/api/users", async (User userData, ApplicationContext db) =>
{
    // получаем пользователя по id
    var user = await db.Users.FirstOrDefaultAsync(u => u.Id == userData.Id);

    // если не найден, отправляем статусный код и сообщение об ошибке
    if (user == null) return Results.NotFound(new { message = "Пользователь не найден" });

    // если пользователь найден, изменяем его данные и отправляем обратно клиенту
    user.Age = userData.Age;
    user.Name = userData.Name;
    await db.SaveChangesAsync();
    return Results.Json(user);
});
```

Здесь аналогичным образом получаем отправленные клиентом данные в виде объекта **User** и сервис **ApplicationContext**. Затем пытаемся найти подобный объект в базе данных. Если объект не найден, отправляем статусный код 404. Если объект найден, то изменяем его данные, с помощью вызова **db.SaveChangesAsync()** сохраняем изменения в базу данных и отправляем измененный объект обратно клиенту

Теперь добавим код клиента. Для этого создадим в проекте новую папку wwwroot, в которую добавим новый файл index.html.



Определим в файле index.html следующим код для взаимодействия с веб-приложением **ASP.NET Core**:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>METANIT.COM</title>
<style>
td {padding:5px;}
button{margin: 5px;}
</style>
</head>
<body>
    <h2>Список пользователей</h2>
    <div>
        <input type="hidden" id="userId" />
        <p>
            Имя:<br/>
            <input id="userName" />
        </p>
        <p>
            Возраст:<br />
            <input id="userAge" type="number" />
        </p>
         <p>
            <button id="saveBtn">Сохранить</button>
            <button id="resetBtn">Сбросить</button>
        </p>
    </div>
    <table>
        <thead><tr><th>Имя</th><th>Возраст</th><th></th></tr></thead>
        <tbody>
        </tbody>
    </table>

```

```
    <script>
    // Получение всех пользователей
        async function getUsers() {
            // отправляет запрос и получаем ответ
            const response = await fetch("/api/users", {
                method: "GET",
                headers: { "Accept": "application/json" }
            });
            // если запрос прошел нормально
            if (response.ok === true) {
                // получаем данные
                const users = await response.json();
                const rows = document.querySelector("tbody");
                // добавляем полученные элементы в таблицу
                users.forEach(user => rows.append(row(user)));
            }
        }
        // Получение одного пользователя
        async function getUser(id) {
            const response = await fetch(`/api/users/${id}`, {
                method: "GET",
                headers: { "Accept": "application/json" }
            });
            if (response.ok === true) {
                const user = await response.json();
                document.getElementById("userId").value = user.id;
                document.getElementById("userName").value = user.name;
                document.getElementById("userAge").value = user.age;
            }
            else {
                // если произошла ошибка, получаем сообщение об ошибке
                const error = await response.json();
                console.log(error.message); // и выводим его на консоль
            }
        }
```

```
        // Добавление пользователя
        async function createUser(userName, userAge) {

            const response = await fetch("api/users", {
                method: "POST",
                headers: { "Accept": "application/json", "Content-Type": "application/json" },
                body: JSON.stringify({
                    name: userName,
                    age: parseInt(userAge, 10)
                })
            });
            if (response.ok === true) {
                const user = await response.json();
                document.querySelector("tbody").append(row(user));
            }
            else {
                const error = await response.json();
                console.log(error.message);
            }
        }
        // Изменение пользователя
        async function editUser(userId, userName, userAge) {
            const response = await fetch("api/users", {
                method: "PUT",
                headers: { "Accept": "application/json", "Content-Type": "application/json" },
                body: JSON.stringify({
                    id: userId,
                    name: userName,
                    age: parseInt(userAge, 10)
                })
            });
            if (response.ok === true) {
                const user = await response.json();
                document.querySelector(`tr[data-rowid='${user.id}']`).replaceWith(row(user));
            }
            else {
                const error = await response.json();
                console.log(error.message);
            }
        }
```

```
        // Удаление пользователя
        async function deleteUser(id) {
            const response = await fetch(`/api/users/${id}`, {
                method: "DELETE",
                headers: { "Accept": "application/json" }
            });
            if (response.ok === true) {
                const user = await response.json();
                document.querySelector(`tr[data-rowid='${user.id}']`).remove();
            }
            else {
                const error = await response.json();
                console.log(error.message);
            }
        }

        // сброс данных формы после отправки
        function reset() {
            document.getElementById("userId").value =
            document.getElementById("userName").value =
            document.getElementById("userAge").value = "";
        }
        // создание строки для таблицы
        function row(user) {

            const tr = document.createElement("tr");
            tr.setAttribute("data-rowid", user.id);

            const nameTd = document.createElement("td");
            nameTd.append(user.name);
            tr.append(nameTd);

            const ageTd = document.createElement("td");
            ageTd.append(user.age);
            tr.append(ageTd);

            const linksTd = document.createElement("td");

```

```
            const editLink = document.createElement("button");
            editLink.append("Изменить");
            editLink.addEventListener("click", async() => await getUser(user.id));
            linksTd.append(editLink);

            const removeLink = document.createElement("button");
            removeLink.append("Удалить");
            removeLink.addEventListener("click", async () => await deleteUser(user.id));

            linksTd.append(removeLink);
            tr.appendChild(linksTd);

            return tr;
        }
        // сброс значений формы
        document.getElementById("resetBtn").addEventListener("click", () =>  reset());

        // отправка формы
        document.getElementById("saveBtn").addEventListener("click", async () => {

            const id = document.getElementById("userId").value;
            const name = document.getElementById("userName").value;
            const age = document.getElementById("userAge").value;
            if (id === "")
                await createUser(name, age);
            else
                await editUser(id, name, age);
            reset();
        });

        // загрузка пользователей
        getUsers();
    </script>
</body>
</html>
```

Основная логика здесь заключена в коде **javascript**. При загрузке страницы в браузере получаем все объекты из БД с помощью функции **getUsers()**:

```
async function getUsers() {
    // отправляет запрос и получаем ответ
    const response = await fetch("/api/users", {
        method: "GET",
        headers: { "Accept": "application/json" }
    });
    // если запрос прошел нормально
    if (response.ok === true) {
        // получаем данные
        const users = await response.json();
        const rows = document.querySelector("tbody");
        // добавляем полученные элементы в таблицу
        users.forEach(user => rows.append(row(user)));
    }
}
```

Для добавления строк в таблицу используется функция **row()**, которая возвращает строку. В этой строке будут определены ссылки для изменения и удаления пользователя.

Ссылка для изменения пользователя с помощью функции **getUser()** получает с сервера выделенного пользователя:

```
async function getUser(id) {
    const response = await fetch(`/api/users/${id}`, {
        method: "GET",
        headers: { "Accept": "application/json" }
    });
    if (response.ok === true) {
        const user = await response.json();
        document.getElementById("userId").value = user.id;
        document.getElementById("userName").value = user.name;
        document.getElementById("userAge").value = user.age;
    }
    else {
        // если произошла ошибка, получаем сообщение об ошибке
        const error = await response.json();
        console.log(error.message); // и выводим его на консоль
    }
}
```

И выделенный пользователь добавляется в форму над таблицей. Эта же форма применяется и для добавления объекта. С помощью скрытого поля, которое хранит **id** пользователя, мы можем узнать, какое действие выполняется – добавление или редактирование. Если **id** не установлен (равен пустой строке), то выполняется функция **createUser**, которая отправляет данные в **POST**-запросе:

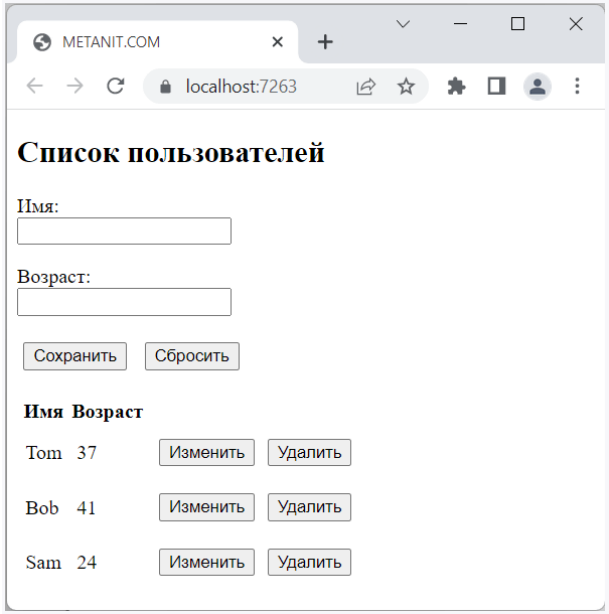
```
async function createUser(userName, userAge) {

    const response = await fetch("api/users", {
        method: "POST",
        headers: { "Accept": "application/json", "Content-Type": "application/json" },
        body: JSON.stringify({
            name: userName,
            age: parseInt(userAge, 10)
        })
    });
    if (response.ok === true) {
        const user = await response.json();
        document.querySelector("tbody").append(row(user));
    }
    else {
        const error = await response.json();
        console.log(error.message);
    }
}
```

Если же ранее пользователь был загружен на форму, и в скрытом поле сохранился его **id**, то выполняется функция **editUser**, которая отправляет **PUT**-запрос:

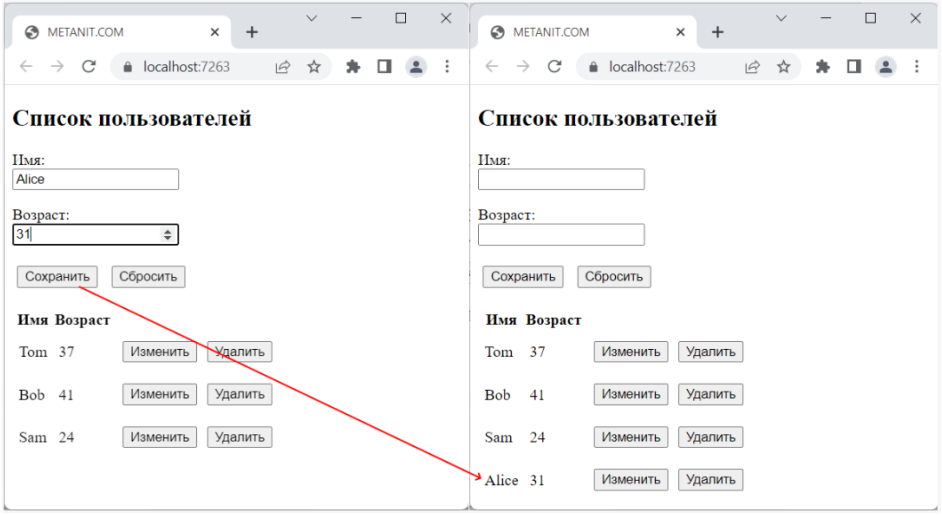
```
async function editUser(userId, userName, userAge) {
    const response = await fetch("api/users", {
        method: "PUT",
        headers: { "Accept": "application/json", "Content-Type": "application/json" },
        body: JSON.stringify({
            id: userId,
            name: userName,
            age: parseInt(userAge, 10)
        })
    });
    if (response.ok === true) {
        const user = await response.json();
        document.querySelector(`tr[data-rowid='${user.id}']`).replaceWith(row(user));
    }
    else {
        const error = await response.json();
        console.log(error.message);
    }
}
```

И функция **deleteUser()** посылает приложению **ASP.NET Core** запрос типа **DELETE** на удаление пользователя, и при успешном удалении на сервере удаляет пользователя по **id** из таблицы пользователей.

Теперь запустим проект, и по умолчанию приложение отправит браузеру веб-страницу index.html, которая загрузит список объектов:



После этого мы сможем выполнять все базовые операции с пользователями – получение, добавление, изменение, удаление. Например, добавим нового пользователя:



2. CORS и кросс-доменные запросы

2.1. Подключение CORS в приложении

По умолчанию веб-браузеры в целях безопасности ограничивают **ajax**-запросы между различными доменами. Однако нередко возникает ситуация, когда необходимо выполнять запросы из приложения с одного адреса (или домена) к приложению, которое размещено по другому адресу. Для этого нам надо использовать технику, которая называется **CORS (Cross Origin Resource Sharing)**.

Добавление и настройка сервисов CORS

Возьмем простейший проект **ASP.NET Core** по типу **Empty** и в файле Program.cs определим следующий код:

```
var builder = WebApplication.CreateBuilder();

builder.Services.AddCors(); // добавляем сервисы CORS

var app = builder.Build();

// настраиваем CORS
app.UseCors(builder => builder.AllowAnyOrigin());

app.Map("/", async context => await context.Response.WriteAsync("Hello METANIT.COM!"));

app.Run();
```

Для подключения сервисов **CORS** в приложении вызывается метод **builder.Services.AddCors()**.

Чтобы задействовать CORS для обработки запроса вызывается метод **app.UseCors()**. Для конфигурации параметров CORS этот метод использует делегат, в который передается объект **CorsPolicyBuilder**. И с помощью этого объекта можно выполнить настройку **CORS**.

```
app.UseCors(builder => builder.AllowAnyOrigin());
```

С помощью метода **AllowAnyOrigin()** мы указываем, что приложение может обрабатывать запросы с любых доменов/адресов. В качестве ответа клиенту приложение будет отправлять строку "Hello METANIT.COM!".

Тестирование CORS

Для тестирования создадим второй проект также **ASP.NET Core** по типу **Empty** и определим в этом проекте в файле Program.cs следующий код:

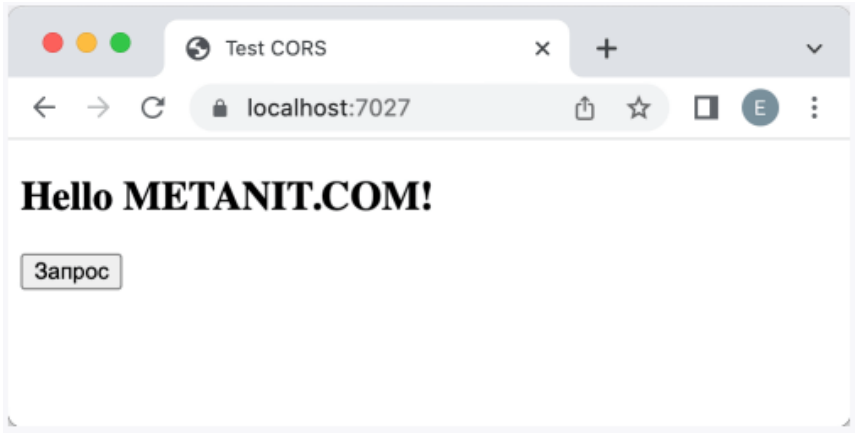
```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.UseDefaultFiles();
app.UseStaticFiles();

app.Run();
```

Здесь мы будем работать только со статическими файлами. В частности, определим в этом проекте в папке wwwroot html-страницу index.html, которая будет загружаться по умолчанию:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Test CORS</title>
</head>
<body>
    <h2 id="result"></h2>
    <button id="btn" value="Запрос">Запрос</button>

    <script>
        const btn = document.getElementById("btn");
        const result = document.getElementById("result");
        btn.addEventListener("click", async () => {

        try {
            const response = await fetch("https://localhost:7199/");
            if (response.ok) result.innerText = await response.text();
        }
        catch (e) {
            result.innerText = e.message;
        }
        });
    </script>
</body>
</html>
```

В данном случае по нажатию на кнопку будет выполняться **ajax**-запрос к первому приложению, который, в моем случае, будет запускаться по адресу "https://localhost:7199/". Полученный от первого приложения ответ будет загружаться в элемент заголовка <h2 id="result">.

Запустим оба проекта и во втором приложении нажмем на кнопку, чтобы получить ответ от первого приложения:



2.2. Конфигурация CORS

Для обработки кроссдоменных запросов и работы **CORS** ранее код приложения выглядел следующим образом:

```
var builder = WebApplication.CreateBuilder();

builder.Services.AddCors(); // добавляем сервисы CORS

var app = builder.Build();

// настраиваем CORS
app.UseCors(builder => builder.AllowAnyOrigin());

app.Map("/", async context => await context.Response.WriteAsync("Hello METANIT.COM!"));

app.Run();
```

В вызове **app.UseCors()** с помощью методов объекта **CorsPolicyBuilder** можно настроить конфигурацию **CORS**:

AllowAnyOrigin(): принимаются запросы с любого адреса.

AllowAnyHeader(): принимаются запросы с любыми заголовками.

AllowAnyMethod(): принимаются запросы любого типа (**GET**/**POST**).

AllowCredentials(): разрешается принимать идентификационные данные от клиента (например, куки).

WithHeaders(): принимаются только те запросы, которые используют содержат определенные заголовки.

WithMethods(): принимаются запросы только определенного типа.

WithOrigins(): принимаются запросы только с определенных адресов.

WithExposedHeaders(): позволяет серверу отправлять на сторону клиента свои заголовки.

Определение адреса

Метод **AllowAnyOrigin()** позволяет установить взаимодействие с любым приложением по любому адресу. Однако подобное поведение может быть нежелательным. В этом случае мы можем ограничить круг адресов с помощью метода **WithOrigins()**:

```
app.UseCors(builder => builder.WithOrigins("http://example.com", "http://google.com"));
```

При чем, что важно, в конце названия домена не должно быть конечного слеша.

Определение метода запроса

Метод **AllowAnyMethod()** позволяет принимать запросы любого типа (**GET**/**POST**). Также можно настроить принятие только определенного типа запросов:

```
app.UseCors(builder => builder.WithOrigins("https://localhost:7027").WithMethods("GET"));
```

Определение заголовков

Для разрешения запросов с любыми заголовками применяется метод **AllowAnyHeader()**. Следует отметить, что вместе с этим методом лучше также указывать и метод **AllowAnyMethod()** или **WithMethods()** для указания типа запроса:

```
app.UseCors(builder => builder.WithOrigins("https://localhost:7027")
                              .AllowAnyHeader()
                              .AllowAnyMethod());
```

Если необходимо принимать запросы только с определенными заголовками, то все требуемые заголовки надо передать в метод **WithHeaders()**:

```
app.UseCors(builder => builder.WithOrigins("https://localhost:7027")
                              .AllowAnyMethod()
                              .WithHeaders("custom-header"));
```

В данном случае необходимо, чтобы клиент отправлял в запросе заголовок **"custom-header"**. Например, отправка данного заголовка в коде **javascript** с помощью функции **fetch**:

```
<h2 id="result"></h2>
<button id="btn" value="Запрос">Запрос</button>

<script>
    const btn = document.getElementById("btn");
    const result = document.getElementById("result");
    btn.addEventListener("click", async () => {
        try {
            const response = await fetch("https://localhost:7199/", { headers: { "custom-header": "test" } });
            if (response.ok) result.innerText = await response.text();
        }
        catch (e) {
            result.innerText = e.message;
        }
    });
</script>
```

Получение заголовков на клиенте

Если сервер отправляет какие-то свои заголовки, то по умолчанию клиент их не получает. Чтобы на стороне сервера указать, какие заголовки может получать клиент, следует использовать метод **WithExposedHeaders()**:

```
var builder = WebApplication.CreateBuilder();

builder.Services.AddCors(); // добавляем сервисы CORS

var app = builder.Build();

// настраиваем CORS
app.UseCors(builder => builder.WithOrigins("https://localhost:7027")
                              .AllowAnyMethod()
                              .AllowAnyHeader()
                              .WithExposedHeaders("custom-header"));

app.Run(async (context) =>
{
    context.Response.Headers.Add("custom-header", "5678");
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

Сервер устанавливает заголовок **custom-header** и отправляет его клиенту. Чтобы клиент получил этот заголовок, он передается в метод **WithExposedHeaders**.

Затем на стороне клиента можно получить значение этого заголовка. Например, получение в коде **JavaScript** с помощью функции **fetch**:

```html
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8" />
   <title>Test CORS</title>
</head>
<body>
   <h2 id="result"></h2>
   <button id="btn" value="Запрос">Запрос</button>

   <script>
        const btn = document.getElementById("btn");
        const result = document.getElementById("result");
        btn.addEventListener("click", async () => {
            try {
                const response = await fetch("https://localhost:7199/");
                if (response.ok) {
                    const headerTitle = "custom-header"; // название заголовка
                    result.innerText = await response.text();
                    if (response.headers.has(headerTitle)) {  // если заголовок есть
                        console.log(response.headers.get(headerTitle));     // получаем его значение
                    }
                }
            }
            catch (e) {
                result.innerText = e.message;
            }
        });
   </script>
</body>
</html>
```

Альтернативное получение через **XMLHttpRequest**:

```
const btn = document.getElementById("btn");
const result = document.getElementById("result");
const headers = document.getElementById("headers");
const request = new XMLHttpRequest();

btn.addEventListener("click", function (e) {
    request.open("GET", "https://localhost:44313/");
    request.onreadystatechange = reqReadyStateChange;
    request.send();
});

function reqReadyStateChange() {
    if (request.readyState == 4) {
        if (request.status == 200){
            result.innerText = request.responseText;
            // получаем заголовок
            headers.innerText = request.getResponseHeader("custom-header");
        }
    }
}
```

Передача идентификационных данных

По умолчанию браузер не посылает никаких идентификационных данных. Подобные данные включают куки, а также данные HTTP-аутентификации. Для отправки идентификационных данных в кроссдоменном запросе на стороне клиента у объекта **XMLHttpRequest** необходимо установить свойство **withCredentials** равным **true**.

```
const request = new XMLHttpRequest();
request.open("GET", "https://localhost:44313/");
request.withCredentials = true;
```

Для получения данных на стороне сервера применяется метод **AllowCredentials()**. Этот метод устанавливает заголовок **Access-Control-Allow-Credentials**, который говорит браузеру, что сервер разрешает отправку идентификационных данных. При этом данный метод не может использоваться с методом **AllowAllOrigin**, то есть обязательно нужно указать набор адресов, с которыми будет взаимодействовать сервер. Например:

```
var builder = WebApplication.CreateBuilder();

builder.Services.AddCors(); // добавляем сервисы CORS

var app = builder.Build();

// настраиваем CORS
app.UseCors(builder => builder.WithOrigins("https://localhost:7027")
                              .AllowCredentials());

app.Run(async (context) =>
{
    var login = context.Request.Cookies["login"];  // получаем отправленные куки
    await context.Response.WriteAsync($"Hello {login}!");
});

app.Run();
```

При отправке запроса с помощью функции **fetch** ей необходимо передать опцию **credentials** со значением **include**.

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8" />
   <title>Test CORS</title>
</head>
<body>
   <h2 id="result"></h2>
   <button id="btn" value="Запрос">Запрос</button>

   <script>
        const btn = document.getElementById("btn");
        const result = document.getElementById("result");
        document.cookie = "login=tom32"; // куки, которые будут отправляться
        btn.addEventListener("click", async () => {
            try {
                const response = await fetch("https://localhost:7199/", { credentials: "include"});
                if (response.ok)  result.innerText = await response.text();

            }
            catch (e) {
                result.innerText = e.message;
            }
        });
   </script>
</body>
</html>
```

Альтернативный вариант с помощью **XMLHttpRequest**:

```
const btn = document.getElementById("btn");

const request = new XMLHttpRequest();
document.cookie = "login=tom32;";   // куки, которые будут отправляться

btn.addEventListener("click", function () {
    request.open("GET", "https://localhost:44313/");
    request.onreadystatechange = reqReadyStateChange;
    request.withCredentials = true; //  устанавливаем отправку
    request.send();
});
function reqReadyStateChange() {
    if (request.readyState == 4) {
        if (request.status == 200)
            console.log(request.responseText);
    }
}
```

2.3. Политики CORS

Для упрощения комплексной конфигурации **CORS ASP.NET Core** позволяет определять политики **CORS**. Политики представляют набор правил взаимодействия сервера и клиента. А чтобы задействовать нужную политику, в метод **app.UseCors** передается ее название:

```
var builder = WebApplication.CreateBuilder();

builder.Services.AddCors(options => options.AddPolicy("AllowLocalhost7027", builder => builder
                    .WithOrigins("https://localhost:7027")
                    .AllowAnyHeader()
                    .AllowAnyMethod())
                );

var app = builder.Build();

app.UseCors("AllowLocalhost7027");

app.Run(async context => await context.Response.WriteAsync("Hello client!"));

app.Run();
```

В метод **AddCors** передается делегат, который принимает объект **CorsOptions** – настройки **CORS**. У этого объекта вызывается метод **AddPolicy()**, который устанавливает политику. Первый параметр метода - произвольное название политики (в данном случае **AllowLocalhost7027**). Второй параметр – делегат, который для создания политики использует объект **CorsPolicyBuilder**. То есть это тот же самый объект, методы которого были рассмотрены в прошлой теме. Поэтому для него можно использовать все те же самые методы:

```
builder.Services.AddCors(options => options.AddPolicy("AllowLocalhost7027", builder => builder
                    .WithOrigins("https://localhost:7027")
                    .AllowAnyHeader()
                    .AllowAnyMethod())
);
```

Чтобы применить эту политику в метод **app.UseCors** передается ее название:

```
app.UseCors("AllowLocalhost7027");
```

При этом можно определить набор политик и применять одну из них, меняя ее по мере необходимости:

```
builder.Services..AddCors(options =>
{
    options.AddPolicy("TestPolicy", builder => builder
                        .WithOrigins("https://localhost:7027")
                        .AllowAnyHeader()
                        .AllowAnyMethod());
    options.AddPolicy("ProductionPolicy", builder => builder
                        .WithOrigins("http://localhost.com")
                        .AllowAnyHeader()
                        .AllowAnyMethod());
});
```

2.4. Глобальная и локальная настройка CORS

Если для обработки запросов в приложении применяются конечные точки, то мы можем для каждой конечной точки указать свои настройки **CORS**. Таким образом, мы можем установить глобально для всех ресурсов одни и те же настройки **CORS**, либо конкретизировать для каждого ресурса свои настройки.

Ранее использовалась глобальная настройка **CORS**, которая предполагает, что настройки **CORS** передаются в метод **app.UseCors()**:

```
var builder = WebApplication.CreateBuilder();

builder.Services.AddCors(); // добавляем сервисы CORS

var app = builder.Build();

// глобальная настройка CORS для всех ресурсов
app.UseCors(builder => builder.AllowAnyOrigin());

app.MapGet("/", async context => await context.Response.WriteAsync("Hello World!"));
app.MapGet("/home", async context => await context.Response.WriteAsync("Home Page!"));

app.Run();
```

Здесь вне зависимости, обращаемся ли мы к ресурсу "/" или к ресурсу "/home", к обоим ресурсам могут обращаться пользователи с любых адресов.

С помощью метода **RequireCors()** у объекта **IEndpointConventionBuuilder** мы можем установить настройки **CORS** для каждого конкретного маршрута. Данный метод в качестве параметра принимает либо название политики **CORS**, либо делегат с параметром **CorsPolicyBuilder**, с помощью который устанавливается конфигурацию **CORS**.

```
var builder = WebApplication.CreateBuilder();

builder.Services.AddCors(options => {

    options.AddPolicy("AllowTestSite", builder => builder
        .WithOrigins("https://localhost:7027")
        .AllowAnyHeader()
        .AllowAnyMethod());

    options.AddPolicy("AllowNormSite", builder => builder
            .WithOrigins("https://localhost.com")
            .AllowAnyHeader()
            .AllowAnyMethod());
});

var app = builder.Build();

app.UseCors();

app.MapGet("/", async context => await context.Response.WriteAsync("Hello World!"))
    .RequireCors(options => options.AllowAnyOrigin());

app.MapGet("/home", async context => await context.Response.WriteAsync("Home Page!"))
    .RequireCors("AllowNormSite");

app.MapGet("/about", async context => await context.Response.WriteAsync("About Page!"))
    .RequireCors("AllowTestSite");

app.Run();
```

Таким образом, мы можем определить для каждого ресурса в приложении свою конфигурацию **CORS**.