**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**



**МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

***Факультет Информационных технологий***

***Кафедра Информатики и информационных технологий***

**направление подготовки**

**09.03.02 «Информационные системы и технологии»**

**ЛАБОРАТОРНАЯ РАБОТА №** 12

**Дисциплина: «Backend»**

**Тема:** *Работа с базами данных в приложении на основе ASP.NET Core*

**Выполнил: студент группы: 231-339**

Карапетян Нвер Каренович\_\_\_\_\_\_\_\_\_\_\_

(Фамилия И.О.)



**Дата, подпись:** 28.02.25 \_\_\_\_\_\_\_\_\_\_\_\_

(Дата)(Подпись)

**Проверил:** \_\_*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_*

(Фамилия И.О., степень, звание) (Оценка)

**Дата, подпись** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ *\_\_\_\_\_\_\_\_\_\_\_\_\_*

(Дата)(Подпись)

**Москва**

**2025**

Цель:

Ознакомиться с работой с кросс-доменными запросами (CORS) в веб-приложениях на платформе ASP.NET Core для обеспечения безопасности и разрешения запросов с других источников.

Задачи:

* Создать модель данных используя ORM Entity Framework.
* Описать модель и контекст данных.
* Подключить модель к приложению, используя внедренные зависимости.
* Провести инициализацию базы данных начальными данными.
* Выбрать данные в одном из методов контроллера и вернуть их как результат метода.

Ход работы

Создание модели данных

Для начала создадим модель данных Product в предварительно созданной папке Models, представляющая товар в интернет-магазине. Модель включает следующие свойства:

* **Id** — уникальный идентификатор товара.
* **Name** — название товара.
* **Description** — описание товара.
* **Price** — цена товара.
* **Stock** — количество товара на складе.

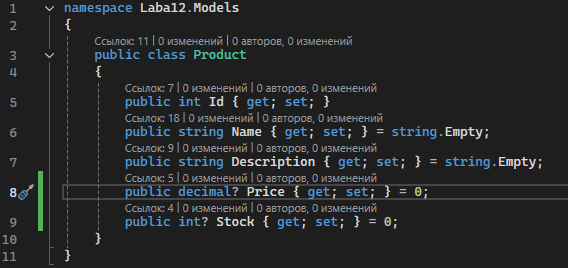


Рисунок 1. Листинг скрипта «Product.cs».

Описание контекста данных

Следующим шагом является создание контекста данных ProductDbContext, который наследуется от DbContext. В контексте данных определяем набор сущностей Products для работы с таблицей товаров. В методе OnModelCreating настроим начальные данные для таблицы Products с использованием метода HasData для базовой инициализации базы данных начальными данными.

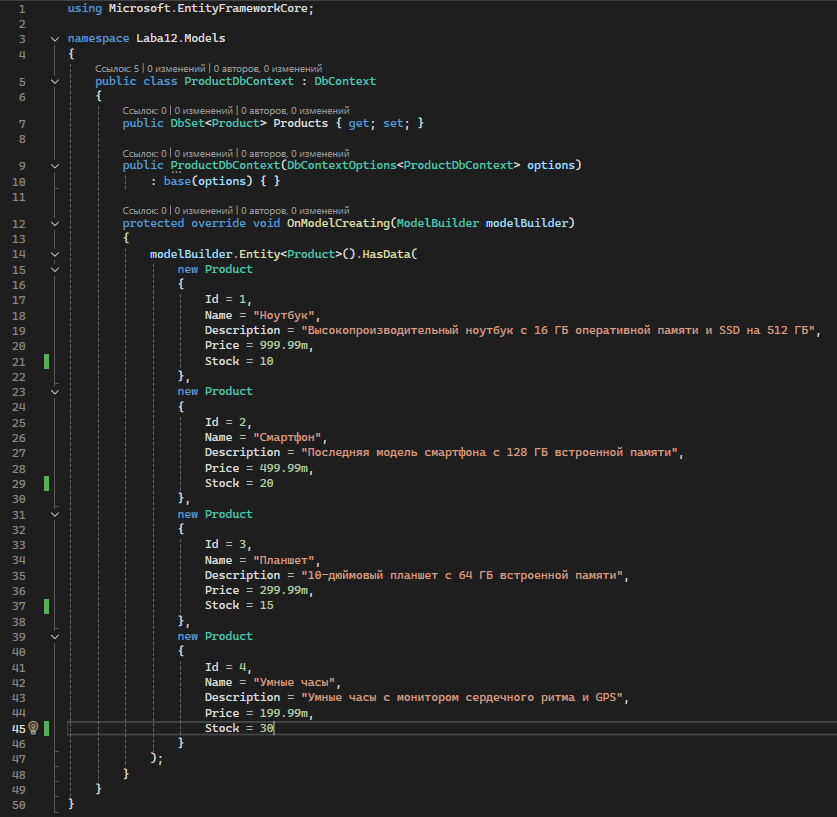


Рисунок 2. Листинг скрипта «ProductDbContext.cs».

Подключение модели к приложению

Для того, чтобы подключить контекст данных ProductDbContext к приложению, воспользуемся механизмом внедрения зависимостей (Dependency Injection, DI). Для этого добавим следующее подключение в файле «Program.cs»:



Рисунок 3. Подключение модели к приложению.

Контекст данных зарегистрирован с использованием метода AddDbContext. В качестве провайдера базы данных используется PostgreSQL (UseNpgsql).

Для подключения к базе данных PostgreSQL добавим строку подключения в файле «appsettings.json», в котором указаны хост, порт, имя базы данных, имя пользователя и пароль.



Рисунок 4. Строка подключения к базе данных.

Создание и применение миграций

Для инициализации базы данных и создания таблиц на основе модели Product в консоли диспетчера пакетов напишем команду «Add-Migration Init», где «Init» — название миграции. После этой команды в папке проекта автоматически создастся папка «Migrations», где и будет файл нашей первой миграции, включая SQL-скрипты для создания таблиц:

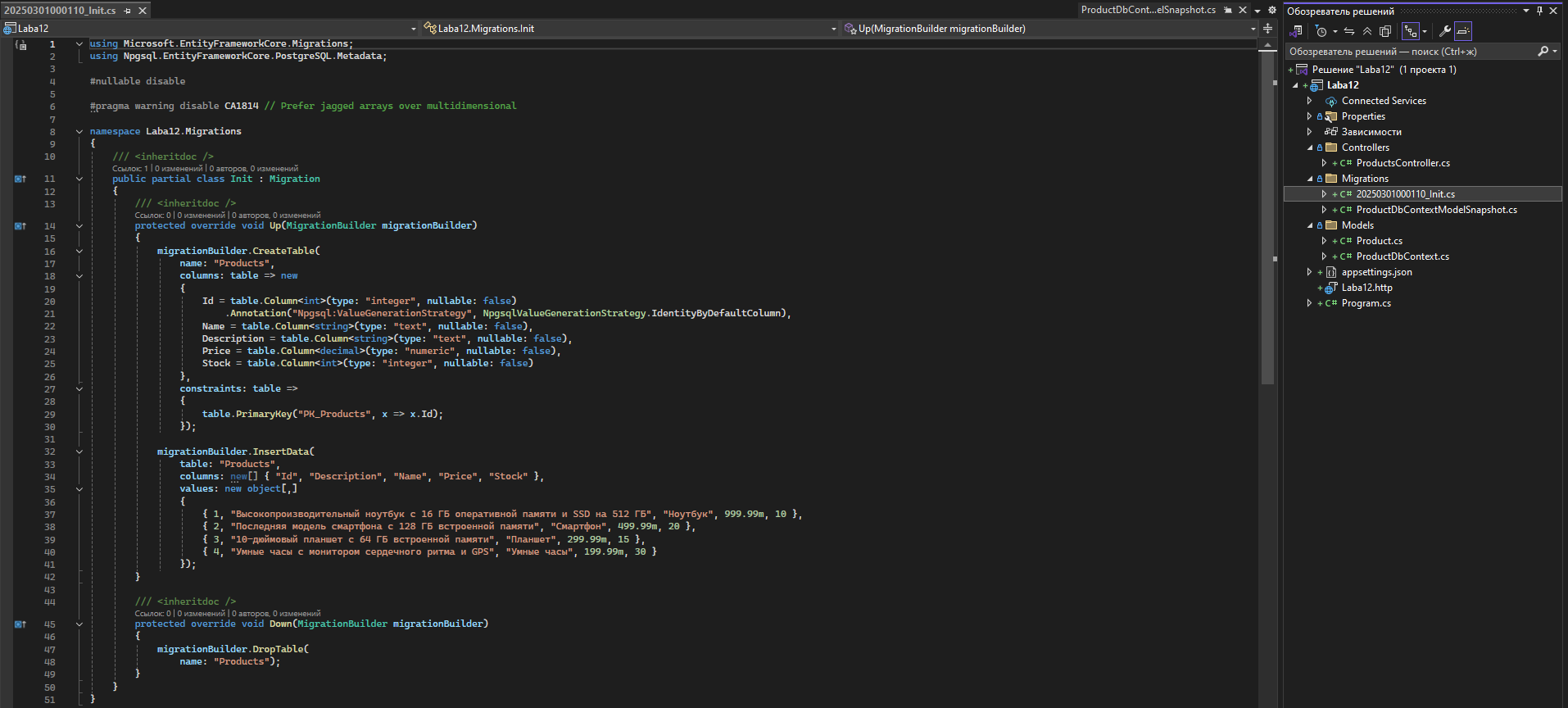


Рисунок 5. Результат создания миграции.

Следующим шагом необходимо применить созданную ранее миграцию. Для этого во все ту же консоль диспетчера пакетов нужно написать команду «Update-Database». После этого в pgAdmin мы сможем наблюдать созданную базу данных с таблицей, которая содержит те же поля, что и наша модель в C#:

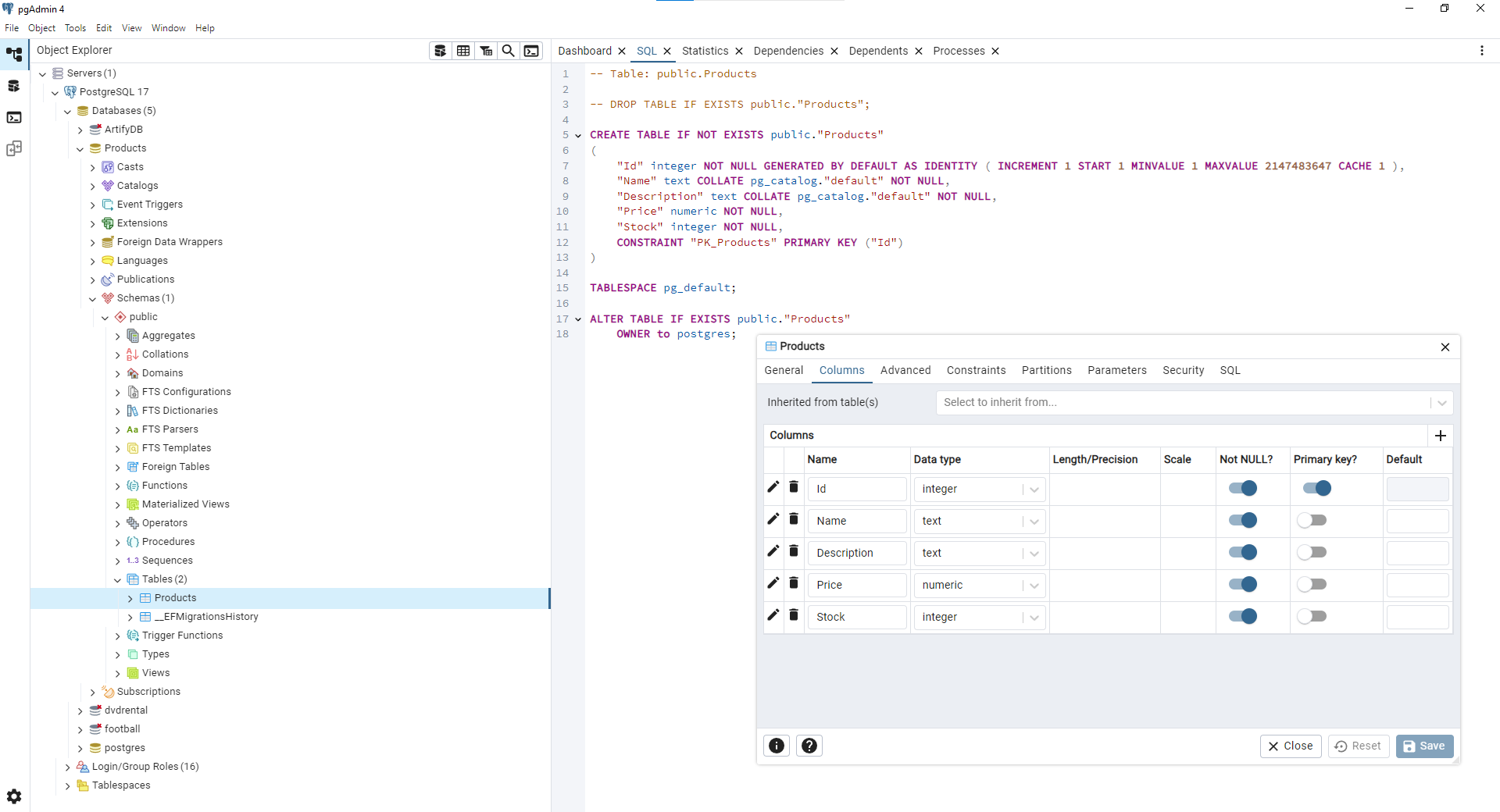


Рисунок 6. Созданная база данных «Products».

Выполним SQL-запрос SELECT для выборки всех данных из базы данных:

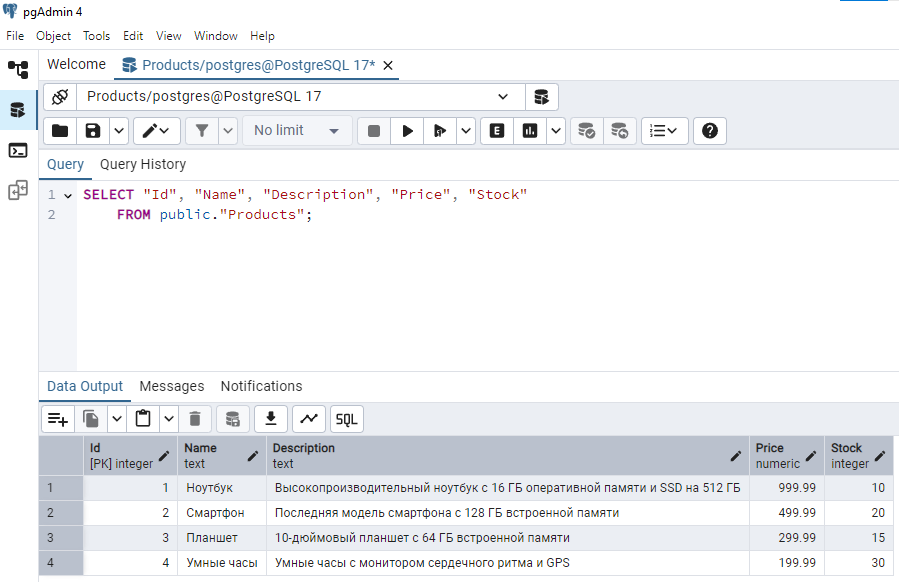


Рисунок 7. На выходе получаем те данные, которыми мы инициализировали базу данных.

Написание простейших CRUD-запросов

Метод GET, который возвращает все записи из таблицы «Products»:

// GET: api/products

[HttpGet]

public async Task<ActionResult<List<Product>>> GetProducts()

{

    var products = await \_context.Products.ToListAsync();

    return Ok(products);

}

Метод GET, который возвращает одну запись по его уникальному идентификатору (ID) из таблицы «Products». В случае, если товар с указанным ID не найден, метод возвращает статус 404 (Not Found):

// GET: api/products/{id}

[HttpGet("{id}")]

public async Task<ActionResult<Product>> GetProductById(int id)

{

    var product = await \_context.Products.FirstOrDefaultAsync(p => p.Id == id);

    if (product is null)

        return NotFound("Продукт с указанным ID не найден в базе данных");

    return Ok(product);

}

Метод POST, с помощью которого можно создавать новый товар и добавлять его данные в таблицу «Products». В ответ возвращается созданный товар и статус 201 (Created):

// POST: /api/products

[HttpPost]

public async Task<ActionResult<Product>> CreateProduct(Product product)

{

    \_context.Products.Add(product);

    await \_context.SaveChangesAsync();

    return CreatedAtAction(nameof(GetProductById), new { id = product.Id }, product);

}

Метод PUT, с помощью которого можно обновлять данные товара с указанным ID. Если товар не найден, возвращается статус 404 (Not Found). В случае успешного обновления возвращается статус 204 (No Content):

// PUT: /api/products/{id}

[HttpPut("{id}")]

public async Task<IActionResult> UpdateProduct(int id, Product updatedProduct)

{

    var product = await \_context.Products.FirstOrDefaultAsync(p => p.Id == id);

    if (product is null)

        return NotFound("Продукт с указанным ID не найден в базе данных");

    if (updatedProduct.Name is not null && product.Name != updatedProduct.Name)

        product.Name = updatedProduct.Name;

    if (updatedProduct.Description is not null && product.Description != updatedProduct.Description)

        product.Description = updatedProduct.Description;

    if (updatedProduct.Price is not null && product.Price != updatedProduct.Price)

        product.Price = updatedProduct.Price;

    if (updatedProduct.Stock is not null && product.Stock != updatedProduct.Stock)

        product.Stock = updatedProduct.Stock;

    await \_context.SaveChangesAsync();

    return NoContent();

}

Метод DELETE, с помощью которого можно удалить из таблицы «Products» запись с указанным ID. В случае успешного удаления возвращается статус 204 (No Content):

// DELETE: /api/products/{id}

[HttpDelete("{id}")]

public async Task<IActionResult> DeleteProduct(int id)

{

    var product = await \_context.Products.FirstOrDefaultAsync(p => p.Id == id);

    if (product is null)

        return NotFound("Продукт с указанным ID не найден в базе данных");

    \_context.Products.Remove(product);

    await \_context.SaveChangesAsync();

    return NoContent();

}

Настройка и применение CORS-политик

Cross-Origin Resource Sharing (CORS) — механизм, использующий дополнительные HTTP-заголовки, чтобы дать возможность агенту пользователя получать разрешения на доступ к выбранным ресурсам с сервера на источнике (домене), **отличном** от того, что сайт использует в данный момент.

В целях безопасности браузеры ограничивают cross-origin запросы, инициируемые скриптами. Например, XMLHttpRequest и Fetch API следуют политике одного источника (Same-Origin Policy). Это значит, что WEB-приложения, использующие такие API, могут запрашивать HTTP-ресурсы только с того домена, с которого были загружены, пока не будут использованы CORS-заголовки.

Для примера создадим небольшой проект на React, напишем простенький React-компонент, который будет представлять из себя таблицу, аналогичную таблице «Products» из нашей базы данных. С помощью fetch-запроса в хуке «useEffect» обратимся к нашей API, а именно к эндпоинту «api/products» и получим записи продуктов «data» из базы данных:

useEffect(() => {

        fetch("https://localhost:7039/api/products")

            .then(response => {

                if (!response.ok)

                    throw new Error('Ошибка загрузки данных')

                return response.json()

            })

            .then(data => {

                setProducts(data)

            })

            .catch(err => {

                throw new Error(err)

            })

    }, []);

Теперь, если мы запустим наше веб-приложение, то обнаружим на странице пустую таблицу и следующие ошибки в консоли:

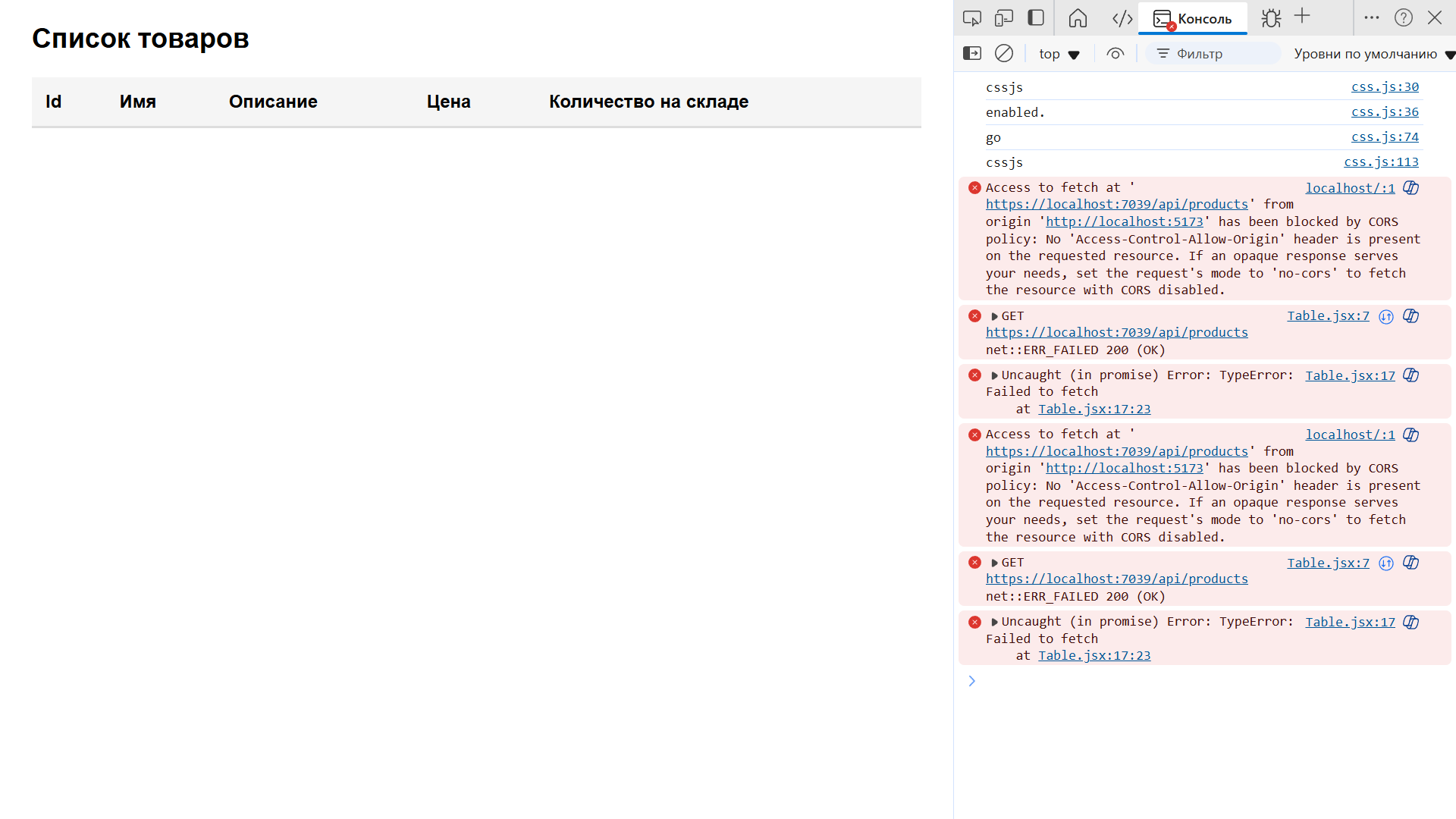


Рисунок 8. Ошибки при обращении к нашему API.

Ошибка возникает из-за того, что API и веб-приложение располагаются на разных портах — 7039 и 5173 соответственно. И по умолчанию такие запросы блокируются.

Для того, чтобы разрешить кросс-доменные запросы, необходимо добавить в «Program.cs» следующую настройку CORS:



Рисунок 9. Настройка и подключение CORS.

Перезагрузим страницу веб-приложения:

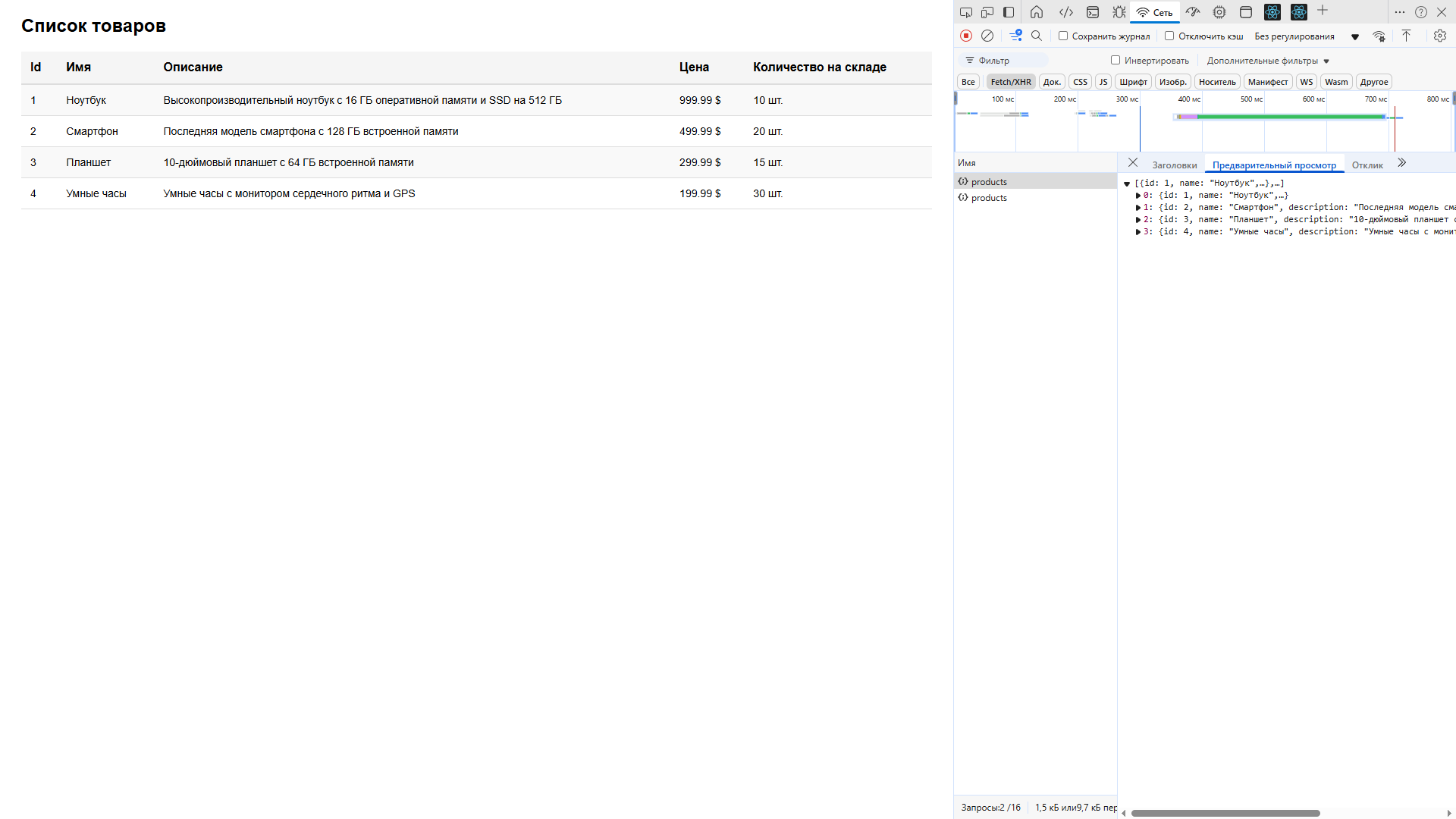


Рисунок 10. Успешные результаты запроса.