УДК 004.273

# ИСПОЛЬЗОВАНИЕ DAPPER В ПРОЕКТАХ ASP.NET CORE WEB API

Хидиров А.А, студент гр. ИТб-202, IV-курс
Сыркин И. С., доцент (к.н.) кафедры ИиАПС
Кузбасский государственный технический университет
имени Т.Ф. Горбачева
г. Кемерово

ASP.NET Core Web API - это мощный фреймворк для создания веб-приложений и API на платформе .NET Core. При разработке таких приложений важно выбрать подходящий инструмент для работы с базой данных. Один из таких инструментов - Dapper, который представляет собой легковесную ORM для .NET. В этой статье мы рассмотрим, как использовать Dapper в проектах ASP.NET Core Web API, его преимущества и недостатки.

Dapper - это легковесная ORM (Object-Relational Mapping) для .NET, разработанная с целью предоставления эффективного и производительного способа взаимодействия с базой данных. В отличие от некоторых других ORM, Dapper не скрывает сложности SQL и предоставляет разработчикам прямой доступ к низкоуровневым операциям базы данных. В проектах ASP.NET Core Web API Dapper может быть весьма полезным инструментом для работы с данными, особенно в случаях, когда требуется максимальная производительность и контроль над запросами к базе данных.

Одним из ключевых преимуществ Dapper является его высокая производительность. Это достигается за счет того, что Dapper минимизирует накладные расходы при выполнении операций базы данных. Он использует низкоуровневые методы доступа к данным и минимально вмешивается в процесс выполнения запросов. Это особенно полезно для проектов, где требуется обработка больших объемов данных или высокая производительность при обработке запросов от множества клиентов. В отличие от некоторых тяжеловесных ORM, Dapper предоставляет простой и понятный API, который легко освоить и использовать. Разработчики могут писать SQL-запросы напрямую и маппить результаты на объекты .NET без необходимости создания

сложных маппингов или конфигураций. Это позволяет сократить время разработки и упростить процесс работы с данными. Dapper обладает высокой степенью гибкости, что позволяет разработчикам полностью контролировать выполнение запросов к базе данных. Он поддерживает различные типы запросов, включая запросы с параметрами, хранимые процедуры и даже выполнение нескольких запросов в рамках одной транзакции. Это делает Dapper идеальным инструментом для работы с самыми разнообразными сценариями использования данных.

В отличие от некоторых ORM, которые могут скрывать сложность и механизмы работы с базой данных, Dapper позволяет разработчикам иметь полный контроль над выполнением запросов. Это позволяет более точно контролировать производительность и оптимизировать запросы для конкретных требований проекта. Dapper поддерживает широкий спектр различных систем управления базами данных (СУБД), включая SQL Server, MySQL, PostgreSQL, SQLite и другие. Это обеспечивает максимальную гибкость в выборе технологий и интеграции с уже существующей инфраструктурой проекта. Dapper имеет активное сообщество пользователей и разработчиков, которые обеспечивают поддержку и помощь в случае возникновения проблем или вопросов. Существует множество ресурсов, форумов и документации, которые помогают разработчикам получить необходимую информацию и решить любые трудности.

В целом, использование Dapper в проектах ASP.NET Core Web API предоставляет множество преимуществ, включая высокую производительность, гибкость, простоту использования и поддержку различных СУБД. Это делает Dapper привлекательным выбором для разработчиков, стремящихся к эффективной и гибкой работе с данными в своих проектах. Dapper расширяет интерфейс IDbConnection следующими методами:

**Execute** – это метод расширения, который мы используем для выполнения команды один или несколько раз и возвращает количество затронутых строк. **Query** – с помощью этого метода расширения мы можем выполнить запрос и отобразить результат. **QueryFirst** – он выполняет запрос и отображает первый результат. **QueryFirstOrDefault** – мы используем этот метод для выполнения запроса и отображения первого результата или значения по умолчанию, если последовательность не содержит элементов.

**QuerySingle** – метод расширения, который выполняет запрос и отображает результат. Он генерирует исключение, если в последовательности нет ровно одного элемента. **QuerySingleOrDefault** – выполняет запрос и отображает результат или значение по умолчанию, если последовательность пуста. Он генерирует исключение, если в последовательности больше одного элемента. **QueryMultiple** – метод расширения, который выполняет несколько запросов в одной команде и отображает результаты. Dapper предоставляет асинхронную версию для всех этих методов (ExecuteAsync, QueryAsync, QueryFirstAsync, QueryFirstOrDefaultAsync, QuerySingleAsync, QuerySingleOrDefaultAsync, QueryMultipleAsync).

Одним из наиболее заметных недостатков Dapper является отсутствие встроенной поддержки автоматического маппинга отношений между таблицами базы данных и объектами .NET. В отличие от некоторых других ORM, которые автоматически создают связи между объектами на основе структуры базы данных, при использовании Dapper разработчику придется вручную управлять этим процессом. Это может быть неудобно и привести к повышенной сложности кода, особенно в случае работы с большими и сложными моделями данных. Dapper позволяет разработчикам писать SQL-запросы напрямую, что может быть как преимуществом, так и недостатком в зависимости от ситуации. Хотя это предоставляет большой контроль над запросами к базе данных, некоторым разработчикам может быть неудобно работать с SQL, особенно если они привыкли к использованию более высокоуровневых методов доступа к данным, таких как LINQ в Entity Framework. Написание и поддержка сложных SQL-запросов также может потребовать дополнительных усилий и времени. Dapper не предоставляет встроенной поддержки отслеживания изменений и управления состоянием объектов, что может затруднить реализацию шаблонов проектирования, таких как Unit of Work и Repository Pattern. Это может быть проблематично при работе с большими и сложными моделями данных, особенно если требуется сохранение изменений в базу данных и управление транзакциями. Dapper обладает ограниченной поддержкой сложных отображений данных, таких как многие-ко-многим и наследование. В некоторых случаях это может ограничить возможности разработчика при моделировании и работе с данными, особенно если требуется обработка сложных структур данных. Использование Dapper требует от разработчика хорошего уровня знаний

SQL, так как все запросы выполняются напрямую. Это может быть проблематично для разработчиков, которые не имеют достаточного опыта или знаний в области SQL, и может привести к написанию неэффективных запросов или даже к уязвимостям безопасности.

Для создания системы учета транспортных средств на базе ASP.NET Core Web API с использованием Dapper и PostgreSQL мы можем разработать CRUD-операции (Create, Read, Update, Delete) для управления данными о транспортных средствах.
Сначала необходимо настроить ASP.NET Core проект и подключиться к базе данных PostgreSQL. Для этого необходимо установить необходимые пакеты NuGet (Рис. 1), настроить строку подключения в файле appsettings.json (Рис. 2) и добавить сервисы Dapper в контейнер зависимостей. (Рис. 3)

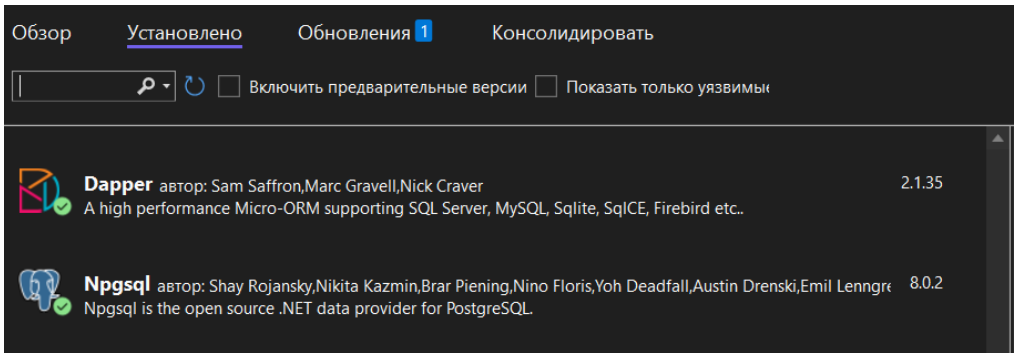



Рис. 1. Пакеты Dapper, Npgsql

```
"ConnectionStrings": {
  "TransportServiceConnection": "Host=localhost;Port=5432;Database=transportdb;Username=postgres;Password=malik98"
},
"AllowedHosts": "*"
```

Рис. 2. Строка подключения

```
builder.Services.AddScoped<IDbConnection>((sp) =>
{
    var configuration = sp.GetRequiredService<IConfiguration>();
    var connectionString = configuration.GetConnectionString("TransportServiceConnection");
    return new NpgsqlConnection(connectionString);
});
```

Рис. 3. Настройка сервисов Dapper в контейнер зависимостей

Добавляем модели.

```
namespace TransportService.Models;

Ссылок: 3
public class Vehicle
{
    Ссылок: 0
    public int Id { get; set; }
    Ссылок: 1
    public string Model { get; set; } = null!;
    Ссылок: 1
    public string Year { get; set; } = null!;
    Ссылок: 1
    public string? Color { get; set; }
    Ссылок: 1
    public string? Country { get; set; }
}
```

Рис. 4. Модель “Транспорт”

Добавляем контроллеры.

```
private readonly IDbConnection _dbConnection;
Ссылок: 0
public VehicleController(IDbConnection dbConnection)
{
    _dbConnection = dbConnection;
}

// GET: api/<VehicleController>
[HttpGet]
Ссылок: 0
public async Task<IActionResult> GetAllVehicles()
{
    const string sql = @"SELECT * FROM Vehicles";
    var vehicles = await _dbConnection.QueryAsync<Vehicle>(sql);
    return Ok(vehicles);
}
```

Рис. 5. Получение всех транспортов

```
// GET api/<VehicleController>/5
[HttpGet("{id}")]
Ссылок: 0
public async Task<IActionResult> GetById(int id)
{
    const string sql = @"SELECT * FROM Vehicles WHERE Id = @id";
    var vehicle = await _dbConnection.QueryFirstAsync(sql, new { Id = id });
    return Ok(vehicle);
}
```

Рис. 6. Получение по Id

```
// POST api/<VehicleController>
[HttpPost]
Ссылок: 0
public async Task<IActionResult> AddVehicle(Vehicle vehicle)
{
    const string sql = @"INSERT INTO Vehicles (Id, Model, Year, Color, Country)
                         VALUES (@id, @color, @model, @year, @country);";
    var id = await _dbConnection.ExecuteScalarAsync<int>(sql, vehicle);
    return Ok(id);
}
```

Рис. 7. Добавление транспорта

```
// PUT api/<VehicleController>/5
[HttpPut("{id}")]
Ссылок: 0
public async Task<IActionResult> UpdateVehicle(int id, Vehicle vehicle)
{
    const string sql = @"UPDATE Vehicles
                         SET Color = @color, Model = @model, Year = @year, Country = @country
                         WHERE Id = @id";
    var queryArg = new
    {
        Id = id,
        Model = vehicle.Model,
        Year = vehicle.Year,
        Color = vehicle.Color,
        Country = vehicle.Country,
    };
    var success = await _dbConnection.ExecuteAsync(sql, queryArg);
    return Ok(success);
}
```

Рис. 8. Редактирование транспорта

```
// DELETE api/<VehicleController>/5
[HttpDelete("{id}")]
Ссылок: 0
public async Task<IActionResult> DeleteVehicle(int id)
{
    const string sql = @"DELETE FROM Vehicles WHERE Id = @Id";
    var success = await _dbConnection.ExecuteAsync(sql, new { Id = id });
    return Ok(success);
}
```

Рис. 9. Удаление транспорта


Рис. 10. Конечные точки API.

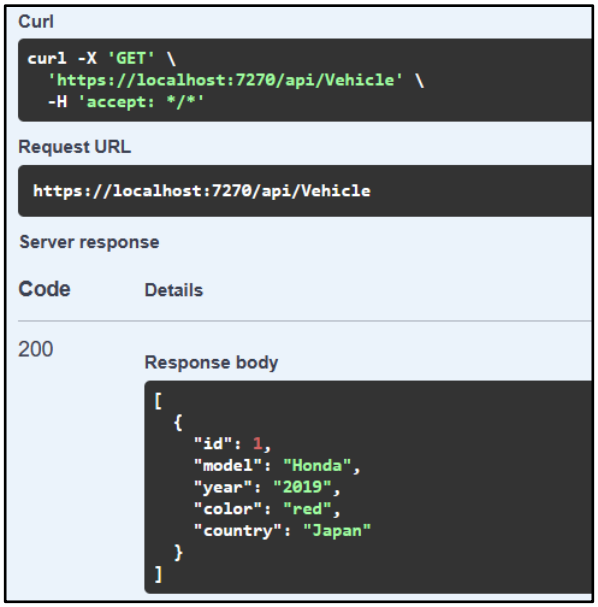



Рис. 11. Запрос на получение данных транспорта

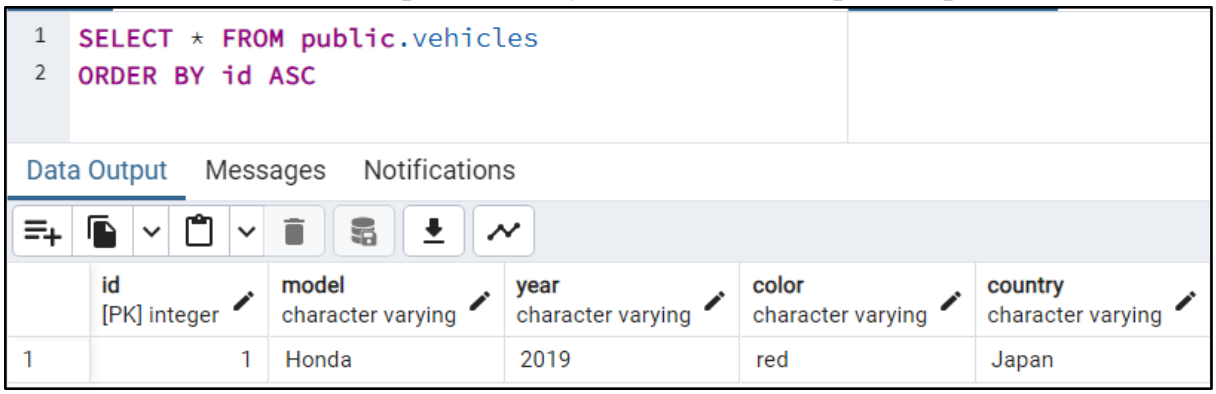



Рис. 12. Данные в БД

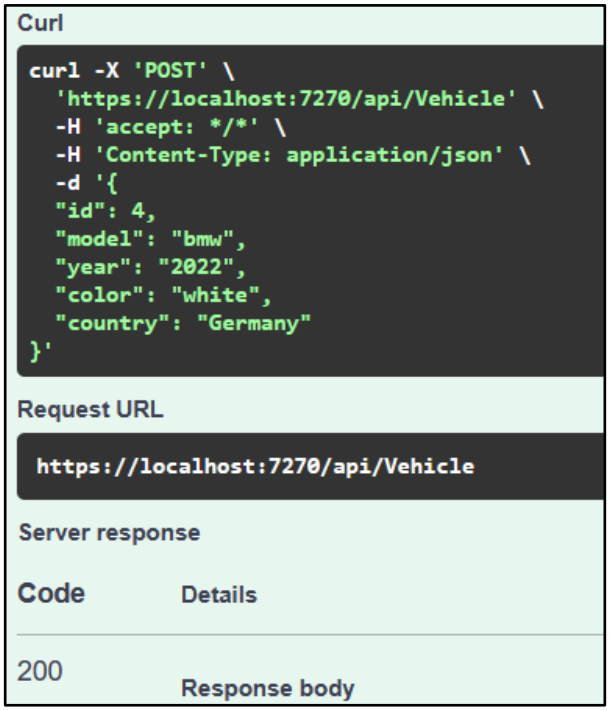
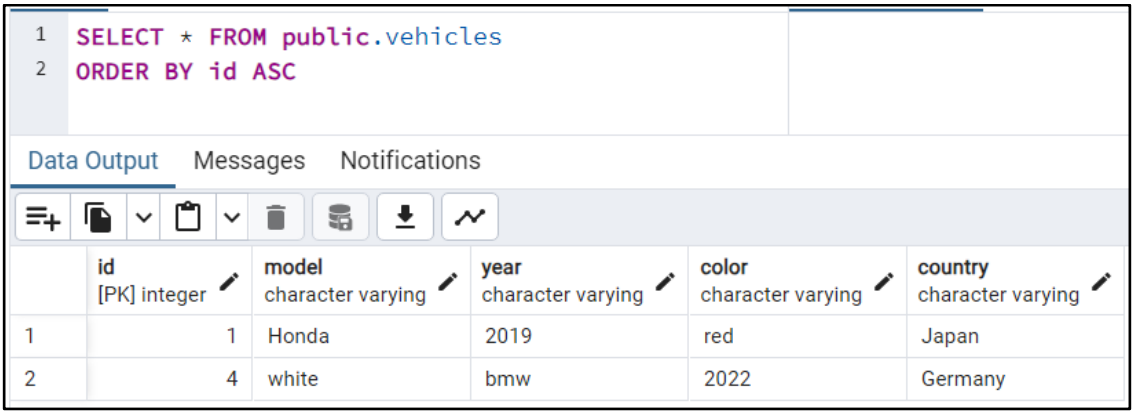



Рис. 13. Запрос на добавление данных транспорта




| | id [PK] integer | model character varying | year character varying | color character varying | country character varying |
|---|---|---|---|---|---|
| 1 | 1 | Honda | 2019 | red | Japan |
| 2 | 4 | white | bmw | 2022 | Germany |

Рис. 14. Данные в БД (после добавление)

**Список литературы**:

1. Официальный репозиторий Dapper на GitHub: [Электронный ресурс]. – Режим доступа: https://github.com/DapperLib/Dapper - Здесь вы найдете документацию, примеры кода и исходный код библиотеки Dapper. Свободный. (Дата обращения 29.03.2024)

2. Документация ASP.NET Core: [Электронный ресурс]. – Режим доступа: https://docs.microsoft.com/en-us/aspnet/core - Здесь вы найдете официальную документацию по ASP.NET Core, включая разделы о работе

с базами данных и интеграции с ORM. Свободный. (Дата обращения 29.03.2024)

3. Статья "Upgrade Your Data Layer With Dapper (a .NET Micro ORM)" на сайте DZone: [Электронный ресурс]. – Режим доступа: https://dzone.com/articles/fasten-your-data-layer-with-dapper-a-net-micro-orm, свободный. (Дата обращения 30.03.2024)

5. Этот сайт предназначен для разработчиков, которые хотят научиться использовать Dapper: [Электронный ресурс]. – Режим доступа: https://www.learndapper.com/, свободный. (Дата обращения 30.03.2024)