ASP.Net Core Minimal APIs using Dapper Day 6

1. What is Dapper?

Dapper is a "micro ORM" for .NET that makes ADO.NET easier and cleaner, while still staying very close to SQL.

- It runs on top of **SqlConnection / SqlCommand**
- You still write SQL manually
- It maps query results → C# classes automatically
- Very lightweight, very fast

You can tell students:

"Dapper = ADO.NET + Less Typing + Auto Mapping"

2. Why Use Dapper with Minimal APIs?

Compared to raw ADO.NET in Minimal API:

Aspect	**ADO.NET (Manual)**	**Dapper**
Code length	Long (reader, loops, mapping)	Very short (Query, Execute)
Mapping	Manual (reader.GetInt32, etc.)	Automatic object mapping
Control	Full control	Still full control (you write SQL)
Performance	Excellent	Excellent
Readability	Lower	Much higher

Conclusion for students:
Use ADO.NET to understand the fundamentals, then use Dapper in real code.

3. Setup: Install Dapper Package

From **Package Manager Console**:

Install-Package Dapper

4. Common Model: Student

We'll reuse the same table and model as in ADO.NET examples.

```
CREATE TABLE Students (
   Id INT IDENTITY(1,1) PRIMARY KEY,
   Name NVARCHAR(100),
   Age INT
);
```

C# Model:

```
public class Student
{
    public int Id { get; set; }   // identity in DB
    public string Name { get; set; }
    public int Age { get; set; }
}
```

5. Connection String and Basic Program Setup

```
{
  "ConnectionStrings": {
    "Default": "Server=.;Database=MyDb;Trusted_Connection=True;TrustServerCertificate=True"
  }
}
```

Program.cs (base):

```
using System.Data;
using System.Data.SqlClient;
using Dapper;

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

var connStr = builder.Configuration.GetConnectionString("Default");
```

For now, we'll use using var conn = new SqlConnection(connStr); inside handlers.

6. READ ALL – GET /students (Dapper Query)

```
app.MapGet("/students", async () =>
{
    using var conn = new SqlConnection(connStr);
    var sql = "SELECT Id, Name, Age FROM Students";

    var students = await conn.QueryAsync<Student>(sql);
    return Results.Ok(students);
});
```

Key points:

- QueryAsync<Student> → runs SELECT and maps each row to Student.
- No manual SqlCommand, SqlDataReader, while(reader.Read()), etc.
- Dapper uses property names to match columns.

7. READ ONE – GET /students/{id}

```
app.MapGet("/students/{id}", async (int id) =>
{
    using var conn = new SqlConnection(connStr);
    var sql = "SELECT Id, Name, Age FROM Students WHERE Id = @Id";

    var student = await conn.QueryFirstOrDefaultAsync<Student>(sql, new { Id = id });

    return student is null
        ? Results.NotFound()
        : Results.Ok(student);
});
```

Explanation:

- Parameter object: new { Id = id } → Dapper builds parameter @Id.
- QueryFirstOrDefaultAsync returns:
    - First row → Student object
    - Or null if no row
- Use Results.NotFound() for HTTP 404.

8. CREATE – POST /students (Insert with Dapper Execute)

```
app.MapPost("/students", async (Student s) =>
{
   using var conn = new SqlConnection(connStr);
   var sql = @"INSERT INTO Students (Name, Age)
            VALUES (@Name, @Age);
            SELECT CAST(SCOPE_IDENTITY() as int);";

   var newId = await conn.ExecuteScalarAsync<int>(sql, new { s.Name, s.Age });
   s.Id = newId;

   return Results.Created($"/students/{newId}", s);
});
```

Teach:

- ExecuteScalarAsync<int> → returns single value (new Id).
- We return 201 Created with location header and object.
- Dapper automatically maps @Name and @Age from anonymous object.

9. UPDATE – PUT /students/{id}

```
app.MapPut("/students/{id}", async (int id, Student s) =>
{
    using var conn = new SqlConnection(connStr);

    var sql = @"UPDATE Students
                SET Name = @Name, Age = @Age
                WHERE Id = @Id";

    var rows = await conn.ExecuteAsync(sql, new
    {
        Id = id,
        s.Name,
        s.Age
    });

    if (rows == 0)
        return Results.NotFound();

    s.Id = id;
    return Results.Ok(s);
});
```

10. DELETE – DELETE /students/{id}

```
app.MapDelete("/students/{id}", async (int id) =>
{
    using var conn = new SqlConnection(connStr);
    var sql = "DELETE FROM Students WHERE Id = @Id";

    var rows = await conn.ExecuteAsync(sql, new { Id = id });

    return rows == 0
        ? Results.NotFound()
        : Results.Ok("Deleted");
});
```

11. Summary: CRUD Endpoints with Dapper (Minimal API)

Program.cs structure:

```
using System.Data;
using System.Data.SqlClient;
using Dapper;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
var connStr = builder.Configuration.GetConnectionString("Default");

// GET all
app.MapGet("/students", async () => { ... });

// GET by id
app.MapGet("/students/{id}", async (int id) => { ... });

// POST
app.MapPost("/students", async (Student s) => { ... });

// PUT
app.MapPut("/students/{id}", async (int id, Student s) => { ... });

// DELETE
app.MapDelete("/students/{id}", async (int id) => { ... });

app.Run();
```

12. Optional (But Recommended): Move to Repository Class

To show better architecture, you can refactor Dapper code into a repository.

12.1 Define Interface

```
public interface IStudentRepository
{
    Task<IEnumerable<Student>> GetAllAsync();
    Task<Student?> GetByIdAsync(int id);
    Task<Student> InsertAsync(Student student);
    Task<bool> UpdateAsync(int id, Student student);
    Task<bool> DeleteAsync(int id);
}
```

12.2 Implement with Dapper

```
public class StudentRepository : IStudentRepository
{
    private readonly string _connStr;
    public StudentRepository(IConfiguration cfg)
    {
        _connStr = cfg.GetConnectionString("Default");
    }

    private IDbConnection CreateConnection() => new SqlConnection(_connStr);

    public async Task<IEnumerable<Student>> GetAllAsync()
    {
        using var conn = CreateConnection();
        var sql = "SELECT Id, Name, Age FROM Students";
        return await conn.QueryAsync<Student>(sql);
    }

    public async Task<Student?> GetByIdAsync(int id)
    {
        using var conn = CreateConnection();
        var sql = "SELECT Id, Name, Age FROM Students WHERE Id = @Id";
        return await conn.QueryFirstOrDefaultAsync<Student>(sql, new { Id = id });
    }

    public async Task<Student> InsertAsync(Student s)
    {
        using var conn = CreateConnection();
        var sql = @"INSERT INTO Students (Name, Age)
                    VALUES (@Name, @Age);
                    SELECT CAST(SCOPE_IDENTITY() as int);";
```

```
        var newId = await conn.ExecuteScalarAsync<int>(sql, new { s.Name, s.Age });
        s.Id = newId;
        return s;
    }

    public async Task<bool> UpdateAsync(int id, Student s)
    {
        using var conn = CreateConnection();
        var sql = @"UPDATE Students
                    SET Name = @Name, Age = @Age
                    WHERE Id = @Id";

        var rows = await conn.ExecuteAsync(sql, new { Id = id, s.Name, s.Age });
        return rows > 0;
    }

    public async Task<bool> DeleteAsync(int id)
    {
        using var conn = CreateConnection();
        var sql = "DELETE FROM Students WHERE Id = @Id";
        var rows = await conn.ExecuteAsync(sql, new { Id = id });
        return rows > 0;
    }
}
```

12.3 Register in DI and Use in Minimal APIs

```
builder.Services.AddScoped<IStudentRepository, StudentRepository>();

var app = builder.Build();

app.MapGet("/students", async (IStudentRepository repo) =>
{
    var data = await repo.GetAllAsync();
    return Results.Ok(data);
});

app.MapGet("/students/{id}", async (int id, IStudentRepository repo) =>
{
    var s = await repo.GetByIdAsync(id);
    return s is null ? Results.NotFound() : Results.Ok(s);
});

app.MapPost("/students", async (Student s, IStudentRepository repo) =>
{
    var created = await repo.InsertAsync(s);
    return Results.Created($"/students/{created.Id}", created);
});
```

```
app.MapPut("/students/{id}", async (int id, Student s, IStudentRepository repo) =>
{
   var ok = await repo.UpdateAsync(id, s);
   return ok ? Results.Ok(s) : Results.NotFound();
});

app.MapDelete("/students/{id}", async (int id, IStudentRepository repo) =>
{
   var ok = await repo.DeleteAsync(id);
   return ok ? Results.Ok("Deleted") : Results.NotFound();
});

app.Run();
```

Now you know:

- Minimal API
- Dapper
- Repository pattern
- DI in minimal style

This is extremely close to how you'd do it in production.

13. Exercises for Students

1. Add Email column to Students table and update Dapper queries.
2. Add endpoint /students/search?name=ra → returns students where Name LIKE '%ra%'.
3. Add validation: if Age <= 0 return 400 Bad Request.
4. Create new entity Course and do full CRUD using Dapper + Minimal API.
5. Convert your earlier ADO.NET CRUD Minimal API to Dapper.