**Minimal API:**

**Technologies and libraries:**

1. Docker.
2. MSSQL using docker.
3. AutoMapper.Extensions.Microsoft.DependencyInjection. Version="11.0.0"
4. Microsoft.EntityFrameworkCore. Version="6.0.7"
5. Microsoft.EntityFrameworkCore.Design Version="6.0.7"
6. Swashbuckle.AspNetCore Version="6.3.2"
7. Microsoft.EntityFrameworkCore.SqlServer Version="6.0.7"

**Commands:**

Setting database connection string credential as secret commands (The username and password will not appear inside configuration appsetting.json file):

> dotnet user-secrets init

> dotnet user-secrets set "UserId" "sa"

> dotnet user-secrets set "Password" "pa55w0rd!"

initiate migration file:

> dotnet tool install --global dotnet-ef

> dotnet ef migrations add initialmigration

Reflect your database that you defined inside setting file as database inside database server and its tables that you defined inside AppDbContext:

> dotnet ef database update

**Create Minimal API Steps:**

1. Create new project (ASP.NET Core Web API).
2. Make sure to uncheck this option 🡺 Use controllers (uncheck to use minimal APIs).
3. The project structure is:

Data folder: contains our Db Context and Repository class.

Models folder: contains our Db tables representation classes folder.

Dtos folder: classes that carry the data that should be passed from or retrieved from the user.

Profiles folder: Mapper classes

Docker-compose.yaml file: contains the MSSQL image information that we will use in the project.

**Classes Comments:**

1. DbContext Example (Inside Data File):

public class AppDbContext:DbContext

{

public AppDbContext(DbContextOptions<AppDbContext>options):base(options)

{

}

public DbSet<Command> Commands => Set<Command>();

}

After that you have to add this line inside Program.cs class:

builder.Services.AddScoped<ICommandRepo, CommandRepo>();

1. It preferred to has Repository class that implements Repository Interface

public interface ICommandRepo{}

You have to inject DbContext in your Repository class

public class CommandRepo: ICommandRepo {

public CommandRepo(AppDbContext context) {

\_context = context;

} }

1. Auto Mapping class example:

public class CommandsProfile:Profile

{

public CommandsProfile()

{

// Source -> Target

CreateMap<Command, CommandReadDto>();

CreateMap<CommandCreateDto, Command>();

CreateMap<CommandUpdateDto, Command>();

}

}

After that you have to add this line inside Program.cs class:

builder.Services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());

Program.cs:

You have to link Your Db with your initiated credential (The user name and password that you have created by command):

var sqlConBuilder = new SqlConnectionStringBuilder();

sqlConBuilder.ConnectionString = builder.Configuration.GetConnectionString("SQLDbConnection");

sqlConBuilder.UserID = builder.Configuration["UserId"];

sqlConBuilder.Password = builder.Configuration["Password"];

builder.Services.AddDbContext<AppDbContext>(opt => opt.UseSqlServer(sqlConBuilder.ConnectionString));

In Minimal API we don’t have controller, so instead we have can add our end points in Program.cs file:

app.UseHttpsRedirection();

app.MapGet("api/v1/commands", async (ICommandRepo repo, IMapper mapper) =>

{

var commands = await repo.GetAllCommand();

return Results.Ok(mapper.Map<IEnumerable<CommandReadDto>>(commands));

});

app.MapGet("api/v1/commands/{id}", async (ICommandRepo repo, IMapper mapper,int id) => {

var command=await repo.GetCommandById(id);

if(command != null)

{

return Results.Ok(mapper.Map<CommandReadDto>(command));

}

return Results.NotFound();

});

app.MapPost("api/v1/commands", async (ICommandRepo repo, IMapper mapper,CommandCreateDto cmdCreateDto) => {

var commandModel = mapper.Map<Command>(cmdCreateDto);

await repo.CreateCommand(commandModel);

await repo.SaveChanges();

var cmdReadDto = mapper.Map<CommandReadDto>(commandModel);

return Results.Created($"api/v1/commands/{cmdReadDto.Id}", cmdReadDto);

});

app.MapPut("api/v1/commands/{id}", async (ICommandRepo repo, IMapper mapper,int id, CommandUpdateDto cmdUpdateDto) =>

{

var command = await repo.GetCommandById(id);

if (command == null)

{

return Results.NotFound();

}

mapper.Map(cmdUpdateDto,command);

await repo.SaveChanges();

return Results.NoContent();

});

app.MapDelete("api/v1/commands/{id}", async (ICommandRepo repo, IMapper mapper, int id) =>

{

var command = await repo.GetCommandById(id);

if (command == null)

{

return Results.NotFound();

}

repo.DeleteCommand(command);

await repo.SaveChanges();

return Results.NoContent();

});

app.Run();

Polly (try another time if the request failed)

Package: Microsoft.Extensions.Http.Polly Version="6.0.7"

public class ClientPolicy

{

public AsyncRetryPolicy<HttpResponseMessage> ImmediateHttpRetry { get; }

public AsyncRetryPolicy<HttpResponseMessage> LinearHttpRetry { get; }

public AsyncRetryPolicy<HttpResponseMessage> ExponentialHttpRetry { get; }

public ClientPolicy()

{

// If fails do another request immediately and keep trying for 5 times as maximum.

ImmediateHttpRetry = Policy.HandleResult<HttpResponseMessage>(res=> !res.IsSuccessStatusCode).RetryAsync(5);

// If fails do another request after 3 seconds and keep trying for 5 times as maximum.

LinearHttpRetry = Policy.HandleResult<HttpResponseMessage>(res => !res.IsSuccessStatusCode).WaitAndRetryAsync (5,retryAttempt=> TimeSpan.FromSeconds(3));

// If fails do another request after 1, 3 , 7 , 15 ... seconds and keep trying for 5 times as maximum.

ExponentialHttpRetry = Policy.HandleResult<HttpResponseMessage>(res => !res.IsSuccessStatusCode).WaitAndRetryAsync(5, retryAttempt => TimeSpan.FromSeconds(Math.Pow(2,retryAttempt)));

}

}

Request Controller:

[Route("api/[controller]")]

[ApiController]

public class RequestController:ControllerBase

{

private readonly ClientPolicy \_clientPolicy;

private readonly IHttpClientFactory \_httpClientFactory;

public RequestController(/\*ClientPolicy clientPolicy,\*/IHttpClientFactory httpClientFactory)

{

/\*\_clientPolicy = clientPolicy;\*/

\_httpClientFactory=httpClientFactory;

}

//Get api/request

[HttpGet]

public async Task<ActionResult> MakeRequest()

{

// var client =new HttpClient();

var client = \_httpClientFactory.CreateClient("TestClientFactory");

var response = await client.GetAsync("http://localhost:5209/api/response/25");

//I have commneted this becase there is another way to use Polly by Program.cs

//

// var response = await \_clientPolicy.ExponentialHttpRetry.ExecuteAsync(()=> client.GetAsync("http://localhost:5209/api/response/25"));

if (response.IsSuccessStatusCode)

{

Console.WriteLine("--> ResponseService return Success");

return Ok();

}

Console.WriteLine("--> ResponseService return FALURE");

return StatusCode(StatusCodes.Status500InternalServerError);

return Ok();

}

}

Program.cs

// Add services to the container.

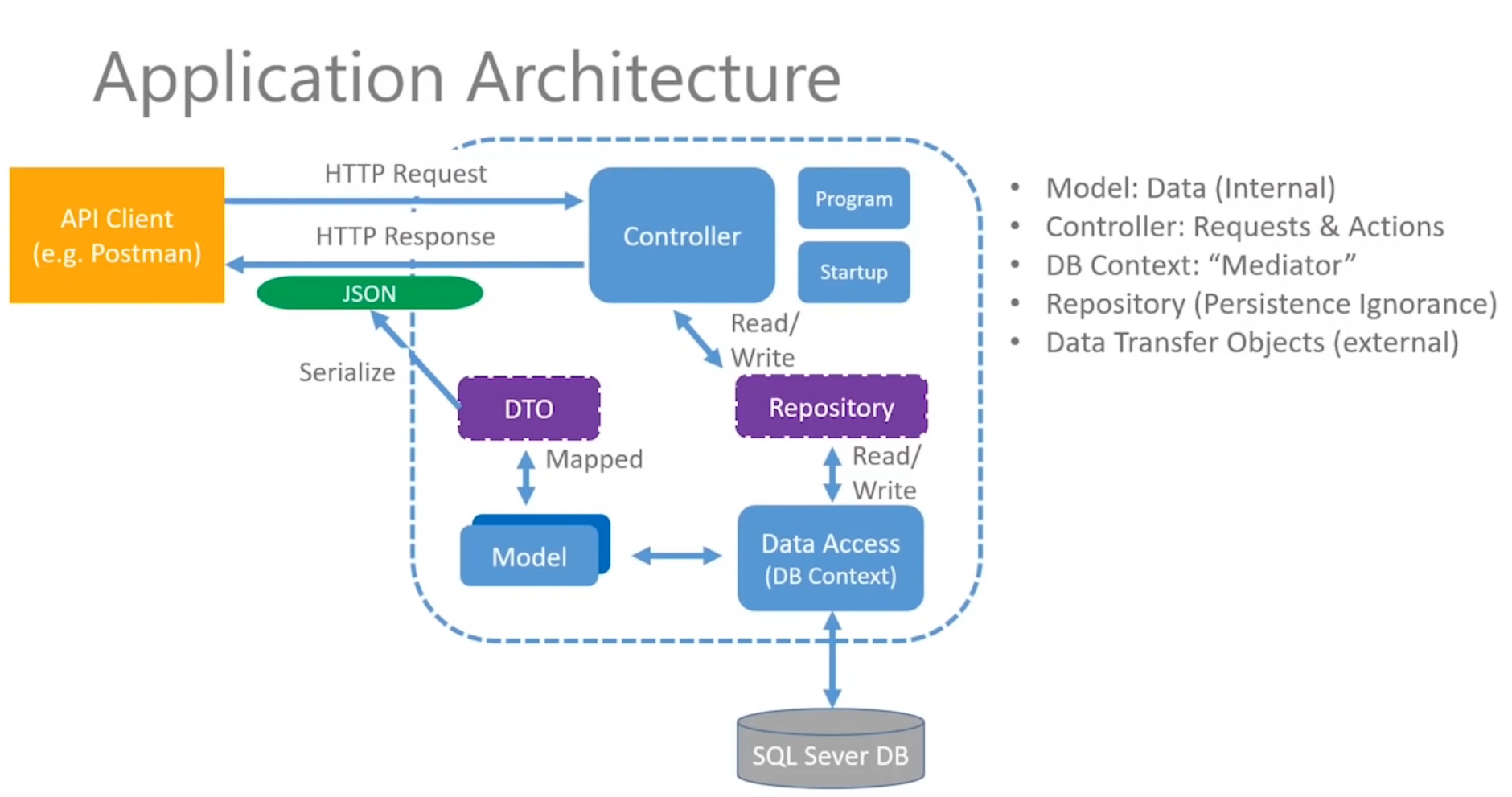
//in below way we don’t need to use dependency injection

builder.Services.AddHttpClient("TestClientFactory").AddPolicyHandler(

request=>request.Method == HttpMethod.Get ? new ClientPolicy().ImmediateHttpRetry : new ClientPolicy().ImmediateHttpRetry);

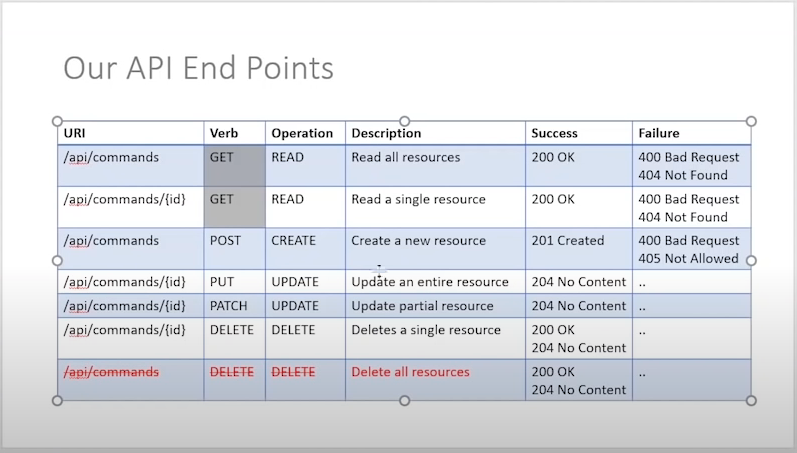
builder.Services.AddSingleton<ClientPolicy>(new ClientPolicy());

MVC REST API:

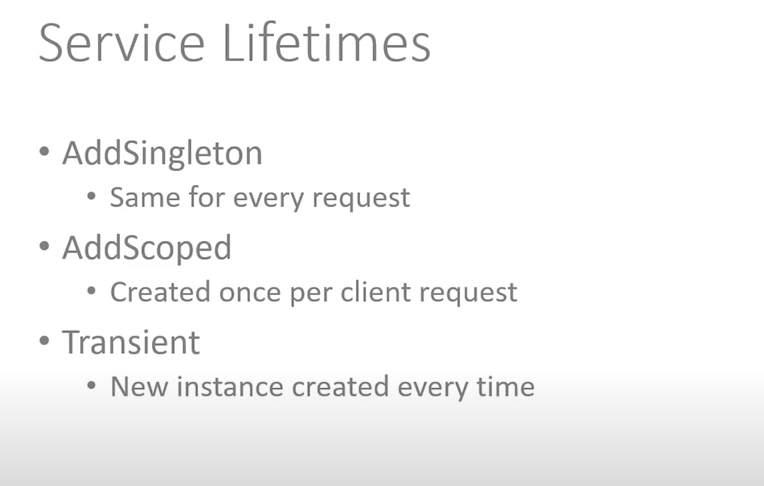
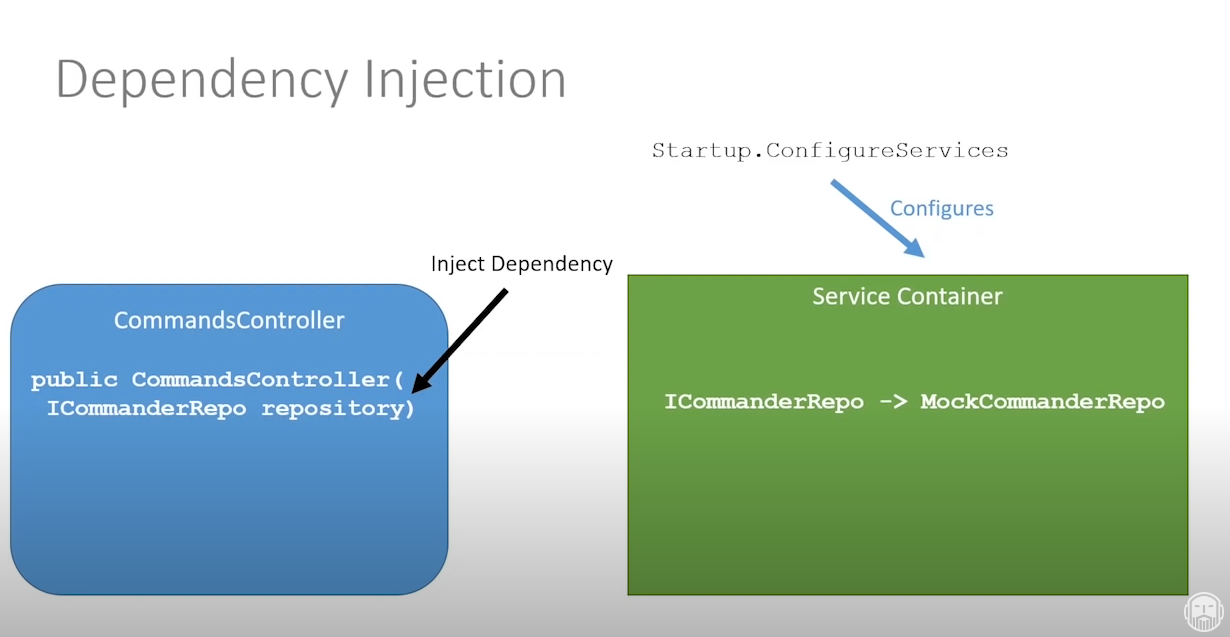


1. **ASP.NET Core Web API** project

API Documentation for this project

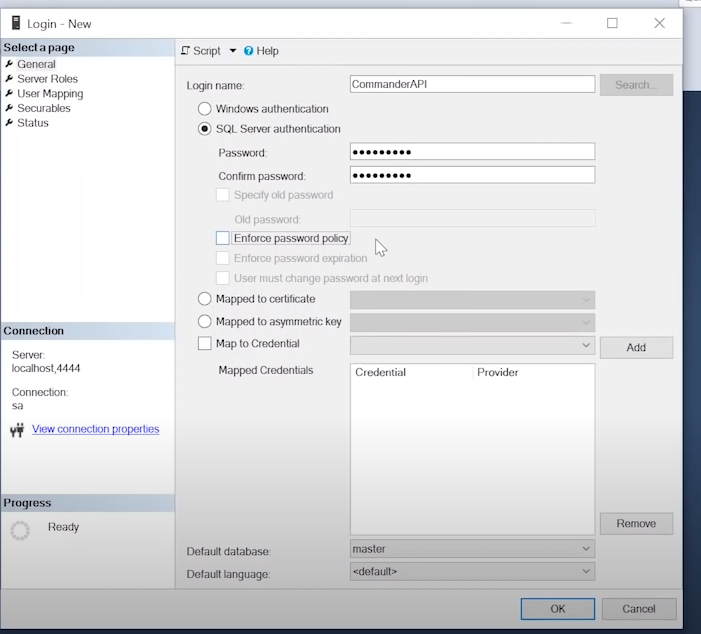


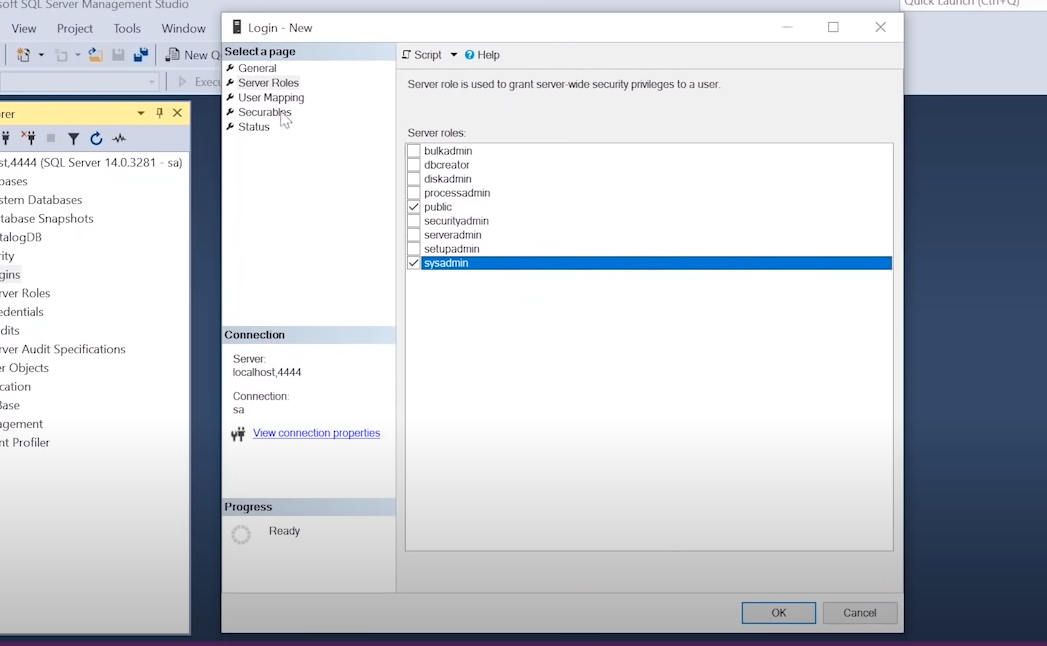
Your Repository has to be added to Startup.ConfigureServices

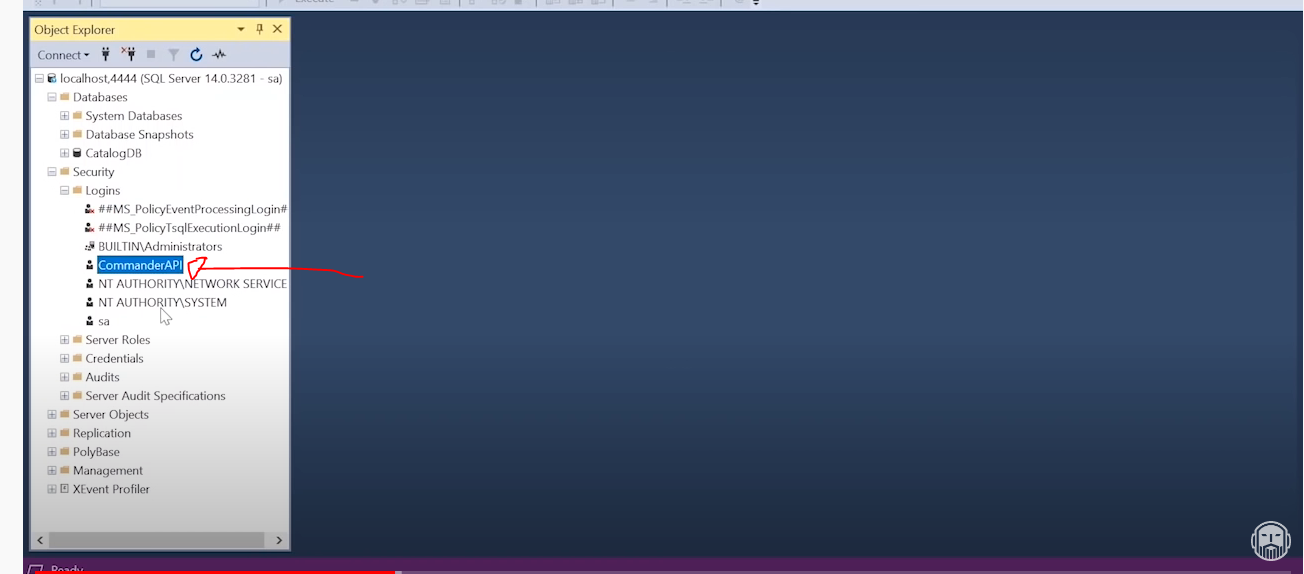


Database Configuration for our project:









Now disconnect and reconnect

The packages that we are using:

Microsoft.EntityFrameworkCore.SqlServer Version="6.0.7"

Microsoft.EntityFrameworkCore.Design Version="6.0.7"

Microsoft.EntityFrameworkCore Version="6.0.7"

AutoMapper.Extensions.Microsoft.DependencyInjection Version="11.0.0"

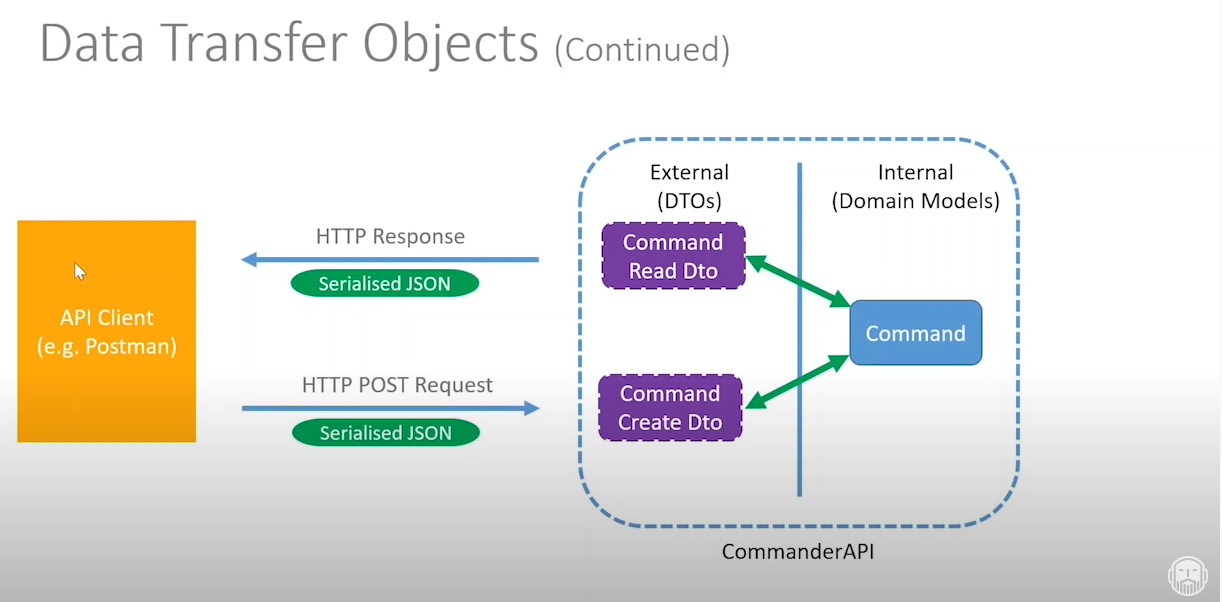
Microsoft.AspNetCore.JsonPatch Version="6.0.7"

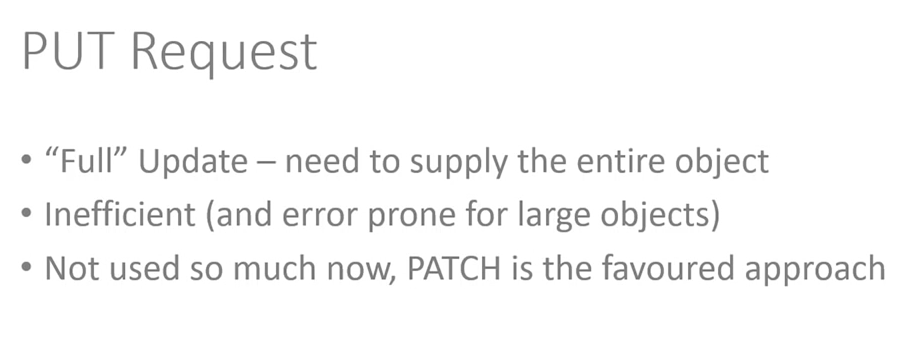
Microsoft.AspNetCore.Mvc.NewtonsoftJson Version="6.0.7"

In case you want to remove migrations that haven’t been implemented yet you can use below command:

>Dotnet ef migrations remove

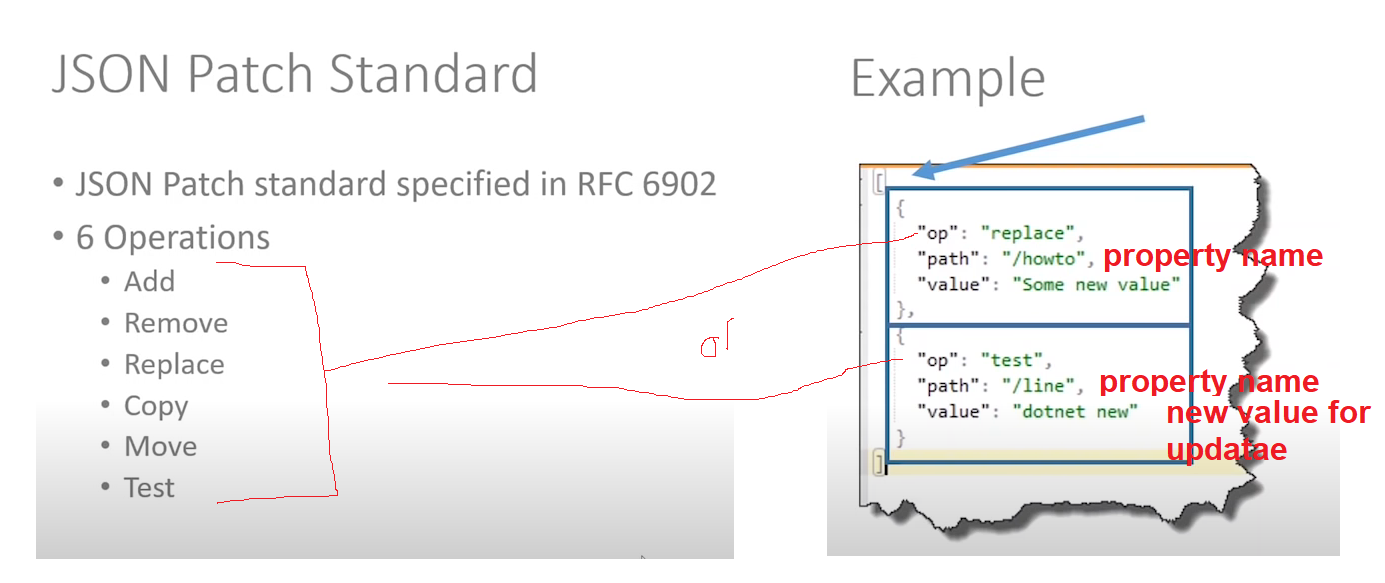
DTO Overview and the purpose of using DTO





Patch Request allow you to update the record partially and there are standards to do that:

1. You have to follow below pattern of request



If you want to use Patch request you need to install couple of packages:

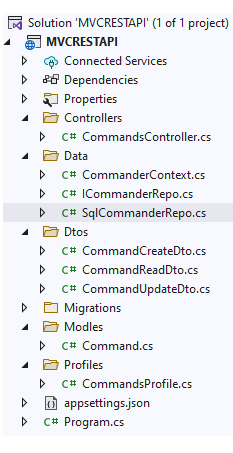
Microsoft.AspNetCore.JsonPatch Version="6.0.7"

Microsoft.AspNetCore.Mvc.NewtonsoftJson Version="6.0.7"

And add this line to the configuration

builder.Services.AddControllers().AddNewtonsoftJson(s=> { s.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver(); });

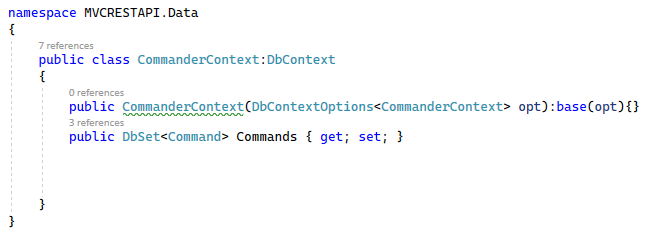
**Project Structure:**



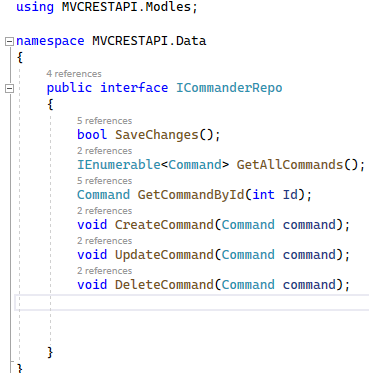
Controller:



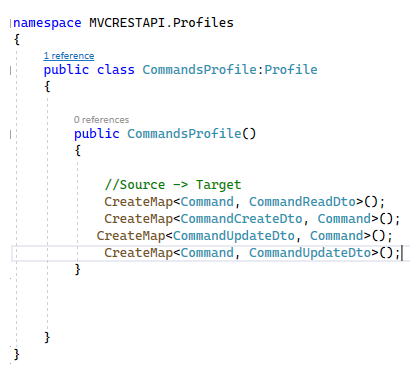
Project DbContext:



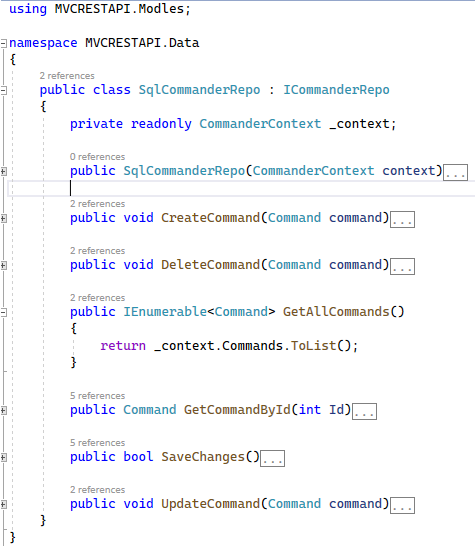
Repository Interface:



Auto Mapper Profile:



Repository:



Project Configuration (Program.cs class)

