Create a Solution and Projects

Create an ASP.NET Core Web API project and two separate projects for commands and queries.

Create a Model For Product

namespace MicroservicesWithCQRSDesignPattern.Model

{

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

}

}

Create a Model for Query with data Filters

namespace MicroservicesWithCQRSDesignPattern.Model

{

public class GetProductsQuery

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

public int PageNumber { get; set; }

public int PageSize { get; set; }

public string SearchTerm { get; set; }

public decimal? MinPrice { get; set; }

public decimal? MaxPrice { get; set; }

}

}

Command and Query Models for Create Product

Create models for commands and queries. For instance.

namespace MicroservicesWithCQRSDesignPattern.Quries.CommandModel

{

public class CreateProductCommand

{

public string Name { get; set; }

public decimal Price { get; set; }

}

}

Command and Query Models for Delete Product

namespace MicroservicesWithCQRSDesignPattern.Quries.QueryModel

{

public class DeleteProductCommand

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

}

}

 Command and Query Models for Update Product

namespace MicroservicesWithCQRSDesignPattern.Quries.QueryModel

{

public class UpdateProductCommand

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

}

}

Command and Query Models for Get All Product

namespace MicroservicesWithCQRSDesignPattern.Quries.QueryModel

{

public class GetAllProductCommand

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

}

}

 Create Interface for ICommandHandler<TCommand>

namespace MicroservicesWithCQRSDesignPattern.Interfaces

{

public interface ICommandHandler<TCommand>

{

Task Handle(TCommand command);

}

}

Create Interface for IQueryHandler<TQuery, TResult>

namespace MicroservicesWithCQRSDesignPattern.Interfaces

{

public interface IQueryHandler<TQuery, TResult>

{

Task<TResult> Handle(TQuery query);

}

}

Create an Interface for IRepository

namespace MicroservicesWithCQRSDesignPattern.Interfaces

{

public interface IRepository<T>

{

Task<T> GetByIdAsync(int id);

Task<IEnumerable<T>> GetAllAsync();

Task AddAsync(T entity);

Task UpdateAsync(T entity);

Task DeleteAsync(T entity);

Task SaveAsync();

}

}

Implement the Repository Pattern for Products

using MicroservicesWithCQRSDesignPattern.AppDbContext;

using MicroservicesWithCQRSDesignPattern.Interfaces;

using MicroservicesWithCQRSDesignPattern.Model;

using Microsoft.EntityFrameworkCore;

namespace MicroservicesWithCQRSDesignPattern.Repository

{

public class ProductRepository : IRepository<Product>

{

private readonly ApplicationDbContext \_dbContext;

public ProductRepository(ApplicationDbContext dbContext)

{

\_dbContext = dbContext;

}

public async Task<Product> GetByIdAsync(int id)

{

return await \_dbContext.Set<Product>().FindAsync(id);

}

public async Task<IEnumerable<Product>> GetAllAsync()

{

return await \_dbContext.Set<Product>().ToListAsync();

}

public async Task AddAsync(Product entity)

{

await \_dbContext.Set<Product>().AddAsync(entity);

}

public async Task UpdateAsync(Product entity)

{

\_dbContext.Set<Product>().Update(entity);

}

public async Task DeleteAsync(Product entity)

{

\_dbContext.Set<Product>().Remove(entity);

}

public async Task SaveAsync()

{

await \_dbContext.SaveChangesAsync();

}

}

}

**Create Handlers for Create, Update, Delete, GetAllProducts**

CreateProductCommandHandler

using MicroservicesWithCQRSDesignPattern.Interfaces;

using MicroservicesWithCQRSDesignPattern.Model;

using MicroservicesWithCQRSDesignPattern.Quries.CommandModel;

namespace MicroservicesWithCQRSDesignPattern.Handlers

{

public class CreateProductCommandHandler : ICommandHandler<CreateProductCommand>

{

private readonly IRepository<Product> \_repository;

public CreateProductCommandHandler(IRepository<Product> repository)

{

\_repository = repository;

}

public async Task Handle(CreateProductCommand command)

{

var product = new Product

{

Name = command.Name,

Price = command.Price

};

await \_repository.AddAsync(product);

await \_repository.SaveAsync();

}

}

}

DeleteProductCommandHandler:

using MicroservicesWithCQRSDesignPattern.Interfaces;

using MicroservicesWithCQRSDesignPattern.Quries.CommandModel;

using MicroservicesWithCQRSDesignPattern.Model;

using MicroservicesWithCQRSDesignPattern.Quries.QueryModel;

namespace MicroservicesWithCQRSDesignPattern.Handlers

{

public class DeleteProductCommandHandler : ICommandHandler<DeleteProductCommand>

{

private readonly IRepository<Product> \_repository;

public DeleteProductCommandHandler(IRepository<Product> repository)

{

\_repository = repository;

}

public async Task Handle(DeleteProductCommand command)

{

var productToDelete = await \_repository.GetByIdAsync(command.Id);

if(productToDelete != null)

{

await \_repository.DeleteAsync(productToDelete);

}

else

{

throw new Exception("Product not found"); // Handle product not found scenario

}

}

}

}

GetProductsQueryHandler:

using MicroservicesWithCQRSDesignPattern.Interfaces;

using MicroservicesWithCQRSDesignPattern.Model;

using MicroservicesWithCQRSDesignPattern.Quries.QueryModel;

namespace MicroservicesWithCQRSDesignPattern.Handlers

{

public class GetProductsQueryHandler : IQueryHandler<GetProductsQuery, IEnumerable<GetAllProductCommand>>

{

private readonly IRepository<Product> \_repository; // Inject repository or database context

public GetProductsQueryHandler(IRepository<Product> repository)

{

\_repository = repository;

}

public async Task<IEnumerable<GetAllProductCommand>> Handle(GetProductsQuery query)

{

var products = await \_repository.GetAllAsync(); // Implement repository method

// Map products to ProductViewModel

return products.Select(p => new GetAllProductCommand

{

Id = p.Id,

Name = p.Name,

Price = p.Price

});

}

}

}

UpdateProductCommandHandler:

using MicroservicesWithCQRSDesignPattern.Interfaces;

using MicroservicesWithCQRSDesignPattern.Quries.CommandModel;

using MicroservicesWithCQRSDesignPattern.Model;

using MicroservicesWithCQRSDesignPattern.Quries.QueryModel;

namespace MicroservicesWithCQRSDesignPattern.Handlers

{

public class UpdateProductCommandHandler : ICommandHandler<UpdateProductCommand>

{

private readonly IRepository<Product> \_repository;

public UpdateProductCommandHandler(IRepository<Product> repository)

{

\_repository = repository;

}

public async Task Handle(UpdateProductCommand command)

{

// Fetch the product to update from the repository

var productToUpdate = await \_repository.GetByIdAsync(command.Id);

if(productToUpdate != null)

{

// Update the product properties

productToUpdate.Name = command.Name;

// Update other properties

// Save changes to the repository

await \_repository.UpdateAsync(productToUpdate);

}

else

{

throw new Exception("Product not found"); // Handle product not found scenario

}

}

}

}

Dependency Injection of Services and Handler in IOC Container:

using MicroservicesWithCQRSDesignPattern.AppDbContext;

using MicroservicesWithCQRSDesignPattern.Handlers;

using MicroservicesWithCQRSDesignPattern.Interfaces;

using MicroservicesWithCQRSDesignPattern.Model;

using MicroservicesWithCQRSDesignPattern.Quries.CommandModel;

using MicroservicesWithCQRSDesignPattern.Quries.QueryModel;

using MicroservicesWithCQRSDesignPattern.Repository;

using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

var configuration = builder.Configuration;

// Add services to the container.

builder.Services.AddDbContext<ApplicationDbContext>(options =>

{

options.UseSqlServer(configuration.GetConnectionString("DefaultConnection")); // Replace with your database provider and connection string

});

builder.Services.AddScoped<IRepository<Product>, ProductRepository>();

builder.Services.AddTransient<ICommandHandler<CreateProductCommand>, CreateProductCommandHandler>();

builder.Services.AddTransient<IQueryHandler<GetProductsQuery, IEnumerable<GetAllProductCommand>>, GetProductsQueryHandler>();

builder.Services.AddTransient<ICommandHandler<UpdateProductCommand>, UpdateProductCommandHandler>();

builder.Services.AddTransient<ICommandHandler<DeleteProductCommand>, DeleteProductCommandHandler>();

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.

if(app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

Application Database Context Class

using MicroservicesWithCQRSDesignPattern.Model;

using Microsoft.EntityFrameworkCore;

namespace MicroservicesWithCQRSDesignPattern.AppDbContext

{

public class ApplicationDbContext : DbContext

{

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

{

}

public DbSet<Product> Products { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

//modelBuilder.Entity<Entity>().HasKey(e => e.Id);

}

}

}

Create a Controller For Handling the HTTP Requests

using MicroservicesWithCQRSDesignPattern.Interfaces;

using MicroservicesWithCQRSDesignPattern.Model;

using MicroservicesWithCQRSDesignPattern.Quries.CommandModel;

using MicroservicesWithCQRSDesignPattern.Quries.QueryModel;

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

namespace MicroservicesWithCQRSDesignPattern.Controllers

{

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

[ApiController]

public class ProductController : ControllerBase

{

private readonly ICommandHandler<CreateProductCommand> \_createProductCommandHandler;

private readonly IQueryHandler<GetProductsQuery, IEnumerable<GetAllProductCommand>> \_getProductsQueryHandler;

private readonly ICommandHandler<UpdateProductCommand> \_updateProductCommandHandler;

private readonly ICommandHandler<DeleteProductCommand> \_deleteProductCommandHandler;

public ProductController(

ICommandHandler<CreateProductCommand> createProductCommandHandler,

IQueryHandler<GetProductsQuery, IEnumerable<GetAllProductCommand>> getProductsQueryHandler,

ICommandHandler<UpdateProductCommand> updateProductCommandHandler)

{

\_createProductCommandHandler = createProductCommandHandler;

\_getProductsQueryHandler = getProductsQueryHandler;

\_updateProductCommandHandler = updateProductCommandHandler;

}

[HttpPost(nameof(CreateProduct))]

public async Task<IActionResult> CreateProduct(CreateProductCommand command)

{

await \_createProductCommandHandler.Handle(command);

return Ok();

}

[HttpGet(nameof(GetProducts))]

public async Task<IActionResult> GetProducts()

{

var products = await \_getProductsQueryHandler.Handle(new GetProductsQuery());

return Ok(products);

}

[HttpPut(nameof(UpdateProduct))]

public async Task<IActionResult> UpdateProduct(UpdateProductCommand command)

{

try

{

await \_updateProductCommandHandler.Handle(command);

return Ok("Product updated successfully");

}

catch(Exception ex)

{

return StatusCode(StatusCodes.Status500InternalServerError, $"Error updating product: {ex.Message}");

}

}

[HttpDelete(nameof(DeleteProduct))]

public async Task<IActionResult> DeleteProduct(int productId)

{

try

{

var command = new DeleteProductCommand { Id = productId };

await \_deleteProductCommandHandler.Handle(command);

return Ok("Product deleted successfully");

}

catch(Exception ex)

{

return StatusCode(StatusCodes.Status500InternalServerError, $"Error deleting product: {ex.Message}");

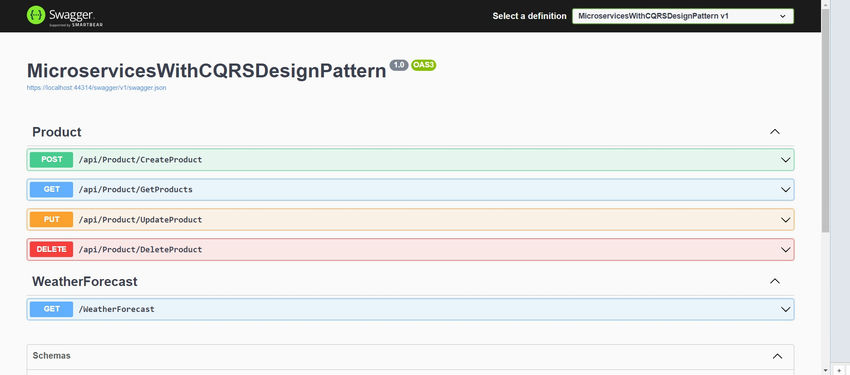
}

}

}

}

**Output.**Output of the Microservice with CQRS Design Pattern.



Conclusion

The Command Query Responsibility Segregation (CQRS) pattern is an architectural principle that separates the responsibility for handling commands (write operations that change state) from queries (read operations that retrieve state). It advocates having separate models for reading and writing data.

Components of CQRS

1. **Command**: Represents an action that mutates the system's state.
2. **Query**: Represents a request for data retrieval without changing the system's state.
3. **Command Handler**: Responsible for executing commands and updating the system's state.
4. **Query Handler**: Responsible for handling read requests and returning data in response to queries.
5. **Command Model**: Contains the logic and rules necessary to process commands and update the data store.
6. **Query Model**: Optimized for querying and presenting data to users, often involving denormalized or optimized data structures tailored for specific queries.

Key Principles

1. **Separation of Concerns**: Splitting the responsibilities of reading and writing data helps in maintaining simpler, more focused models for each task.
2. **Performance Optimization**: Enables independent scaling of read and write operations. The read model can be optimized for query performance without affecting the write model.
3. **Flexibility**: Allows for different models to be used for reading and writing, which can cater to specific requirements and optimizations for each use case.
4. **Complex Domain Logic**: Particularly beneficial in domains where read and write logic significantly differ, allowing tailored models for each type of operation.

Benefits

* **Scalability**: CQRS enables scaling read and write operations independently, optimizing performance.
* **Flexibility and Optimization**: Tailoring models for specific tasks allows for better optimization of the system.
* **Complexity Management**: Separating concerns can make the system easier to understand and maintain.

Challenges

* **Increased Complexity**: Introducing separate models for reading and writing can add complexity to the system.
* **Synchronization**: Keeping the read and write models synchronized can pose challenges, potentially requiring mechanisms like eventual consistency.

CQRS is not a one-size-fits-all solution and is typically employed in systems with complex business logic or where read and write operations vastly differ in terms of frequency, complexity, or optimization requirements. Its application should be carefully considered based on the specific needs and trade-offs of a given system.