# Building Microservices Architecture Using CQRS and ASP.NET Core

In today's fast-paced business landscape, we often build scalable, secure, high-performing, and maintainable applications. A plethora of design patterns and architectural approaches can help in this regard. Command Query Reponsibility Segregation, or CQRS, is a proven architectural pattern that can help build scalable applications in complex scenarios. It does this by splitting responsibilities among read and write models. This article discusses the CQRS pattern, why it’s important, and shows how you can implement the CQRS pattern in Microservices-based applications.

If you’re to work with the code examples discussed in this article, you need the following installed in your system:

* Visual Studio 2022
* .NET 9.0
* ASP.NET 9.0 Runtime

If you don’t already have Visual Studio 2022 installed on your computer, you can download it from here: <https://visualstudio.microsoft.com/downloads/>.

## Understanding the Problem

Consider an enterprise application built in ASP.NET Core that needs to handle big data or massive amounts of data. For example, the application might be handling millions of complex transactions such as retrieving product details, updating stock, processing orders, etc. Over time, as the application attempts to scale to handle more concurrent requests, things can get complicated because you might be using the same models for data read and write operations. As a result, you might often observe inconsistences in your application's data.

The CQRS pattern can help you isolate these operations into commands (i.e., create, update, and delete data) and queries (i.e. retrieve data from the database). This helps you optimize and scale the command and query components of your application independently, enabling your application to be high performant, scalable, and reliable. **Figure 1** shows typical application-enabling queries and updates from and to an underlying database.



**Figure 1:** A typical application with queries and updates from and to a database

Why do you need a mediator? The mediator is the component in your application that’s responsible for routing each request to the appropriate component. The result? Your application's code will be more lean, clean, decoupled, and manageable.

## An Introduction to the CQRS Pattern

In most applications, the same model is used both for read and write/update operations. When you're using simple CRUD operations (create, read, update, and delete), you're good to use the same model to query data as well as save/update data. Over time, as the application grows and you have more and more data in the database, things become complicated. You might often observe anomalies in read and write operations because you have certain properties that must be persisted or updated in the database but you don’t want them to be returned in queries.

For example, you might need the ProductId of the Product model to be stored in the database but not returned when the same model is queried. This might lead to data loss and inconsistencies in data. Enter the CQRS pattern. CQRS, an acronym for Command and Query Responsibility Segregation, is an architectural pattern in which the data read and data write operations are isolated from one another, as shown in **Figure 2**.



**Figure 2:** CQRS at work with the data read and write operations isolated

You can even isolate the read data and write data by using separate databases for each. In this case, you'll have two databases, one for reading or querying data optimized for queries and one write database that’s optimized for create, update, and delete operations. Hence, your read database can be a document database and the write database can be a relational database. By isolating the read and write data stores, you can achieve enhanced scalability to handle increased loads.

For instance, you may want to optimize your read database to withstand significantly greater loads than your write database. There are more read operations in an application than writes and updates. **Figure 3** shows an implementation of the CQRS pattern.



**Figure 3:** An implementation of the CQRS pattern with separate databases for read and write operations

Here’s how the entire process works:

1. The client communicates with an application by sending commands using an API as an interface.
2. The application receives the command and processes it.
3. The application writes the data associated with the command into the write (or command) repository.
4. Once the command is saved to the write database, events are fired in the read (query) repository to update the data.
5. In the read (or query) database, the data persists after processing.
6. By communicating with the APIs used for receiving data, the client sends queries to the query side of the program.
7. The application processes the read request to retrieve the appropriate data from the read database.

In CQRS, a command should always be task-based and not data-centric. For example, "Reserve a train ticket" is an example of a command. However, "Change train reservation status to Reserved" is not a command.

### Benefits

Here are the key benefits of the CQRS pattern:

* **Separation of concerns:** The CQRS pattern separates the query (i.e., data retrieval without modifying the state) and command (i.e., create, update, and delete operations) components of the application, enabling you to optimize both independently. Although you can optimize your queries to be more efficient and fast, you can implement strict validation, transaction, and security logic in the command components.
* **Optimization:** Because the read and write operations are isolated into two different models, the CQRS pattern can help in optimizing performance of data access operations in your application. Although it can help you optimize query performance by improving the speed of data retrieval in read operations, the CQRS pattern can help you preserve transactional integrity and domain logic, and optimize write or update operations in the database.
* **Scalability:** Because the CQRS pattern splits the data access components into read and write components, it enables you to scale each of these components independently of one another. Typically, write operations in an application are fewer compared to read operations. Hence, if the application experiences heavy read traffic, the read models can be scaled horizontally.
* **Security:** The CQRS pattern helps you to implement different security strategies for read and write operations in your application. For example, you might want to secure certain operations that write or update sensitive data in the database. On the other hand, you may want most of the read operations in the application to allow data to the made available.
* **Maintainability:** Because it isolates the read and write operations in an application, the CQRS pattern facilitates maintainability due to separation of concerns. For example, you can change the query components of your application without affecting the command side of your application that’s responsible for updating or persisting data.

## Key Components of the CQRS Design Pattern

The CQRS pattern encompasses several key components:

* **Commands:** These are components that help you change the application's state. For example, you can take advantage of commands to create new data, update data, or delete data.
* **Queries:** These components don't change the application’s state; instead, they help you in data retrieval from the data store.
* **Command handlers:** These are components that accept some incoming commands, perform the actions per these commands and consequently alter the application’s state.
* **Query handlers:** These are components that can help you build queries, execute those queries to retrieve data out of a data store, and subsequently return this data to the invoker.

## Challenges of the CQRS Design Pattern

There are several challenges of the CQRS pattern:

* **Increased complexity:** CQRS introduces additional complexity in your application because of the need for different paths for reading and writing/updating data. Additionally, applications that take advantage of CQRS pattern should have different components for reading and writing operations, which can eventually become an overhead.
* **Consistency:** It’s difficult to ensure that your application's data is consistent—updates to the data in your data store must be reflected in the query results. Hence, you need proper data synchronization mechanisms to ensure that the data pertaining to read and write operations is in sync. Ensure that there’s no data duplication between command and query components. It’s quite challenging to ensure data integrity across multiple data stores, particularly when there’s a system or network failure.
* **Operational overhead:** The CQRS pattern can introduce operational overhead because of the need to deploy services for both query and write operations in separate servers. Deploying, scaling, monitoring, and debugging applications that take advantage of the CQRS pattern can also be challenging.
* **Testing and debugging:** It’s quite challenging to test and debug applications that leverage the CQRS pattern. You might need to adopt specific testing strategies because of the asynchronous nature of the CQRS pattern. Additionally, because the commands and events are processed in an isolated manner and asynchronously, detecting issues in distributed applications can be challenging.

### Use Cases

Typically, the CQRS pattern is used in large and complex projects where performance and scalability are important. Here are the key use cases of the CQRS pattern:

* E-commerce applications
* Healthcare applications
* Financial applications
* IoT applications
* High-traffic systems
* Supply chain management systems

## Introduction to Microservices Architecture

Microservices architecture encompasses a conglomeration of loosely coupled components that can be built using a collection of homogenous or heterogenous technologies. Microservices can be used to build, deploy, and scale components individually and independently of each other. This architectural approach represents a great leap forward in software development by providing organizations with the requisite agility and flexibility to operate efficiently in today's digital world. Because microservices architecture is a scalable architecture, it provides enterprises with the ability to scale existing services as needed.

There’s a plethora of benefits of microservices architecture, such as the following:

* Fault tolerance
* Modularity
* Improved scalability
* Reduced coupling
* Better ROI
* Faster releases
* Faster development

## Building a Microservices-Based Application Using CQRS

When building microservices-based applications, you can take advantage of the CQRS design pattern and the MediatR library to manage the command and query responsibilities of your application efficiently. This approach fosters separation of concerns, which in turn enables you to build an application that contains scalable, efficient, and maintainable source code.

In this section, you’ll build a microservices-based application using CQRS. Let’s now examine how to build a simple ASP.NET Core 9 Web API application using CQRS. You’ll implement a simple order processing application that demonstrates how you can use CQRS in ASP.NET Core. A typical Order Processing System is composed of several entities, such as Supplier, Order, Product, Customer, etc. For the sake of simplicity and brevity, you’ll build the Product module of the application in this example.

In the next section, let’s examine how to create an ASP.NET Core 9 project in Visual Studio 2022.

Create a New ASP.NET Core 9 Project in Visual Studio 2022

You can create a project in Visual Studio 2022 in several ways, such as, from the Visual Studio 2022 Developer Command Prompt or by launching the Visual Studio 2022 IDE. When you launch Visual Studio 2022, you'll see the Start window. You can choose "Continue without code" to launch the main screen of the Visual Studio 2022 IDE.

Now that you know the basics, let’s start setting up the project. To create a new ASP.NET Core 8 Project in Visual Studio 2022:

1. Start the Visual Studio 2022 IDE.
2. In the **Create a new project** window, select “ASP.NET Core Web API” and click Next to move on.
3. Specify the project name as **ShoppingCartSystem** and the path where it should be created in the **Configure your new project** window.
4. If you want the solution file and project to be created in the same directory, you can optionally check the **Place solution and project in the same directory** checkbox. Click Next to move on.
5. In the next screen, specify the target framework and authentication type as well. Ensure that the "Configure for HTTPS," "Enable Docker Support," “Do not use top-level statements”, and the “Enable OpenAPI support” checkboxes are unchecked because you won’t use any of these in this example.
6. Remember to leave the **Use controllers** checkbox checked because you won’t use minimal API in this example.
7. Click Create to complete the process.

A new ASP.NET Core Web API project is created. You’ll use this project to implement the CQRS pattern in ASP.NET Core and C#.

### Install Entity Framework Core

So far so good. The next step is to install the necessary NuGet Package(s) for working with Entity Framework Core and SQL Server. To install these packages into your project, right-click on the solution and the select **Manage NuGet Packages for Solution….**

Now search for the NuGet packages named Microsoft.EntityFrameworkCore and Microsoft.EntityFrameworkCore.InMemory packages in the search box and install them one after the other. Alternatively, you can type the commands shown below at the NuGet Package Manager Command Prompt:

PM> Install-Package

Microsoft.EntityFrameworkCore

PM> Install-Package

 Microsoft.EntityFrameworkCore.

SqlServer

Alternatively, you can install these packages by executing the following commands at the Windows Shell:

dotnet add package

Microsoft.EntityFrameworkCore

dotnet add package

 Microsoft.EntityFrameworkCore.

SqlServer

### Introducing the Mediator Pattern

The mediator design pattern is a behavioral pattern that helps decrease dependencies among objects and regulates how they can interact with each other effectively. This pattern prevents objects from communicating directly, instead requiring them to communicate through a mediator object. As a result, it helps build applications that are loosely coupled, and easier to manage and maintain. **Figure 4** illustrates the Mediator pattern.



**Figure 4:** Demonstrating the Mediator pattern

### Introducing MediatR

To implement the mediator design pattern, you can take advantage of the open-source library called MediatR. This library enables you to implement the CQRS with ease and manage the command and query handlers effectively. It allows easy implementation of CQRS by offering an effective way of dealing with command and query handlers. In essence, MediatR acts as a mediator that directs commands and queries to the appropriate handlers.

The key benefits of MediatR include the following:

* Promotes loose coupling
* Facilitates easy maintainability and testability
* Helps adhere to the single responsibility principle (SRP)
* Enables clear communication between objects

**Figure 5** shows how MediatR works by delegating the request to the respective handlers.



**Figure 5:** Demonstrating how MediatR delegates the requests to the respective handlers

### Install MediatR in ASP.NET Core

You can install the MediatR library from NuGet. To do that, right-click on the solution and the select Manage NuGet Packages for Solution….

Now search for the NuGet packages named MediatR, and MediatR.Extensions.Microsoft.DependencyInjection packages in the search box and install them one after the other. Alternatively, you can write the commands given below at the NuGet Package Manager Command Prompt:

PM> Install-Package MediatR

PM> Install-Package

 MediatR.Extensions.Microsoft.

DependencyInjection

Alternatively, you can install these packages by executing the following commands at the Windows Shell:

dotnet add package MediatR

dotnet add package

 MediatR.Extensions.Microsoft.

DependencyInjection

The MediatR.Extensions.Microsoft.DependencyInjection helps you register the MediatR handlers in your ASP.NET Core application automatically.

### Register MediatR in ASP.NET Core

You should register the MediatR handlers with the services collection of your application. To do this, write the following line of code in the Program.cs file:

builder.Services.AddMediatR

(cfg => cfg.

RegisterServicesFromAssembly

(Assembly.

GetExecutingAssembly()));

### Create a Request in MediatR

In MediatR, messages can be of two types. These include requests (commands/queries) and notifications (for events). You can define a request in MediatR by implementing the IRequest<TResponse> interface, where TResponse is the type of response, i.e., Order, Product, Customer, Supplier, etc. The following code snippet illustrates how you can define a request in MediatR:

public class GetCustomerQuery

: IRequest<Customer>

{

 public Guid CustomerId { get; set; }

 public GetCustomerQuery

 (Guid CustomerId)

 {

 this.CustomerId = CustomerId;

 }

}

### Create a Request Handler in MediatR

You need a handler to handle the request you just created. Essentially, a handler contains the necessary logic that should be executed to handle an incoming request in MediatR. To create a handler in MediatR, create an interface that implements the IRequestHandler<TRequest, TResponse> interface as shown below:

public class GetCustomerQueryHandler

: IRequestHandler

<GetCustomerQuery, Customer>

{

 private readonly

 ICustomerRepository \_repository;

 public GetCustomerQueryHandler

 (ICustomerRepository repository)

 {

 \_repository = repository;

 }

 public async Task<User> Handle

 (GetCustomerQuery request,

 CancellationToken token)

 {

 return await \_repository.

 GetCustomerById

 (request.CustomerId);

 }

}

### Create a Notification in MediatR

You can also create notifications and notification handlers using MediatR. Assume that you want to send notifications when a customer record is deleted from the database. To create a notification in MediatR, implement the INotification interface, as shown below.

public class

CustomerDeletedNotification :

INotification

{

 public Guid CustomerId { get; set; }

 public string FirstName { get; set; }

 public string LastName { get; set; }

}

You’ll create a handler for the CustomerDeletedNotification in the next section.

### Create a Notification Handler in MediatR

To create a notification handler in MediatR, implement the INotificationHandler<TNotification> interface, as shown in the code snippet given below.

public class

CustomerDeletedNotificationlHandler :

INotificationHandler

<CustomerDeletedNotification>

{

 public async Task Handle(

CustomerDeletedNotification

notification,

CancellationToken token)

 {

 //Write your code here

 //to send notification(s)

 //when an existing customer record

 // is deleted from the data store

 }

}

Now that you know how to work with MediatR, in the sections that follow, you’ll implement a simple microservices-based application that leverages the MediatR library.

### Create the Shopping Cart System Database

Create a new database called ShoppingCartSystem using the following script:

Create database ShoppingCartSystem

Next, create the Product, Customer, Order, OrderItem, and the Cart database tables inside the ShoppingCartSystem database using the script given in **Listing 1**.

**Figure 6** demonstrates the database diagram of the ShoppingCartSystem database.



**Figure 6:** The database design of the ShoppingCartSystem database

### Create the Solution Structure

As evident from the database design, the ShoppingCartSystem application is comprised of the Product, Customer, Cart, Order, and OrderItem microservices. So, you should create five WebAPI projects for each of them in the solution you created earlier. You’ll also create solution folders to organize the files in each of the projects. **Figure 7** shows how the solution structure looks.



**Figure 7:** The Solution structure of the ShoppingCartSystem

In this example, for the sake of simplicity and brevity, you’ll create only the **Product** microservice. In the sections that follow, you’ll create classes and interfaces pertaining to the Product microservices-based application.

## Create the Product Microservice

In this example, you’ll build the Product microservice application or the Product API. The product microservice application is comprised of the following files:

* **Product.cs:** This represents the product model that contains domain-specific data and (optionally) business logic.
* **IProductRepository.cs:** This represents the IProductRepository interface that contains the declaration of the operations supported by the product repository.
* **ProductRepository.cs:** This represents the product repository class that implements the members of the IProductRepository interface.
* **ProductDbContext.cs:** This represents the product data context used to perform CRUD operations for the Product table in the database.
* **GetProductByIdQuery.cs:** This represents the query for retrieving a product record based on its ID.
* **GetProductByIdQueryHandler.cs:** This represents the query handler for the GetProductByIdQuery that contains the logic for returning the product record.
* **GetAllProductsQuery.cs:** This represents the query for retrieving all product records.
* **GetAllProductsQueryHandler.cs:** This represents the query handler for the GetAllProductsQuery that contains the logic for returning all product records.
* **CreateProductCommand.cs:** This represents the required operations to create a product record in the database.
* **CreateProductCommandHandler.cs:** This represents the command handler that contains the implementation of the CreateProductCommand operation.
* **UpdateProductCommand.cs:** This represents the operations to be executed to update an existing product record in the database.
* **UpdateProductCommandHandler.cs:** This represents the command handler that contains the implementation of the UpdateProductCommand operation.
* **DeleteProductCommand.cs:** This represents the operations to be executed to delete a product record in the database.
* **DeleteProductCommandHandler.cs:** This represents the command handler that contains the implementation of the DeleteProductCommand operation.
* **appsettings.json:** This represents the application’s settings file where you can configure the database connection string, logging metadata, etc.
* **Program.cs:** Any ASP.NET Core application contains a file where the startup code required by the application resides. This file is named Program.cs where the services required by your application are configured. You can specify dependency injection (DI), configuration, middleware, and much more information in this file.

### Specify the Database Connection String

Your application requires a connection string to establish a connection to the database which, in turn, contains the necessary information about the database connection and any initialization parameters sent by a data provider to a data source. Typically, a connection string contains the name of the database to connect to, the instance name of the database server where the database resides, and some other settings pertaining to security of the database.

In ASP.NET Core, the application's settings are stored in a file known as appsettings.json. This file is created by default when you create a new ASP.NET Core project. You can take advantage of the ConnectionString property to retrieve or store the connection string for a database. You can specify the connection string in the appsettings.json file, as shown below:

{

 "Logging": {

 "LogLevel": {

 "Default": "Information",

 "Microsoft.AspNetCore": "Warning"

 }

 },

 "ConnectionStrings": {

 "SCSDbSettings":

 "Write your connection string here."

 },

 "AllowedHosts": "\*"

}

You’ll use this connection string to enable the application to connect to the database in a section later in this article.

### Create the Model Classes

First off, create two solution folders, one named Models and the other DataAccess. The former will contain one or more model classes, and the latter will have the data context and repository interfaces and classes. It should be noted that you can always create multiple data context classes in the same project. If your data context class contains many entity references, it’s good practice to split the data context among multiple data context classes rather than having one large data context class.

Create a new class called Product in a file named Product.cs inside the Models folder and write the following code in there:

namespace SCS.Product.Models

{

 public record Product

 {

 public Guid Product\_Id

 { get; set; }

 public string Product\_Name

 { get; set; } = default!;

 public string Product\_Description

 { get; set; } = default!;

 public string Product\_Category

 { get; set; } = default!;

 public decimal Product\_Price

 { get; set; } = default!;

 public int Product\_Quantity

 { get; set; } = default!;

 public DateTime Created\_At

 { get; set; } = DateTime.Now;

 public DateTime Modified\_At

 { get; set; } = DateTime.Now;

 }

}

In this implementation, you’ll use only one model class: Product.

### Create the Data Context

In Entity Framework Core (EF Core), a data context is a component used by an application to interact with the database and manage database connections, and to query and persist data in the database. Let’s now create the data context class to enable the application to interact with the database to perform CRUD (Create, Read, Update, and Delete) operations.

To do this, create a new class named ProductDbContext that extends the DbContext class of EF Core and write the following code in there.

public class ProductDbContext :

DbContext

{

 public DbSet <Models.Product>

 Products { get; set; }

 protected override void

 OnConfiguring

 (DbContextOptionsBuilder

 optionsBuilder)

 {

 base.OnConfiguring

 (optionsBuilder);

 }

}

In the preceding piece of code, the statement base.OnConfiguring(optionsBuilder) calls the OnConfiguring method of the base class of your ProductDbContext. Because the base class of the ProductDbContext class is DbContext, the call does nothing in particular.

You can specify your database connection string in the OnConfiguring overloaded method of the ProductDbContext class. However, in this implementation, you’ll store the database connection settings in the AppSettings.json file and read it in the Program.cs file to establish a database connection.

Note that your custom data context class (the ProductDbContext class in this example), must expose a public constructor that accepts an instance of type DbContextOptions<ApplicationDbContext> as an argument. This is needed to enable the runtime to pass the context configuration using a call to the AddDbContext() method to your custom DbContext class. The following code snippet illustrates how you can define a public constructor for your data context class.

public ProductDbContext

(DbContextOptions

<ProductDbContext> options,

IConfiguration configuration) :

base(options)

{

 \_configuration = configuration;

}

### Seed the Database

You might often want to work with data seeding when using Entity Framework Core (EF Core) to populate a blank database with an initial or minimal data set. Data seeding is a one-time process of loading data into a database. The EF Core framework provides an easy way to seed the data using the OnModelCreating() method of the DbContext class.

To generate fake data in your ASP.NET Core application, you can take advantage of the Bogus open-source library. It helps you to seed your database by taking advantage of randomly generated but realistic data. To use this library, you should install the [Bogus](https://www.nuget.org/packages/bogus) (<https://www.nuget.org/packages/bogus>) library from NuGet into your project.

The following code snippet illustrates how you can generate data using random data from the Bogus library:

private Models.Product[]

GenerateProductData()

{

 var productFaker = new Faker <

 SCS.Product.Models.Product> ()

 .RuleFor(x => x.Product\_Id,

 f => Guid.NewGuid())

 .RuleFor(x =>

 x.Product\_Name, f =>

 f.Commerce.ProductName())

 .RuleFor(x => x.Product\_Description,

 f => f.Commerce.ProductDescription())

 .RuleFor(x => x.Product\_Category,

 f => f.Commerce.ProductMaterial())

 .RuleFor(x => x.Product\_Price,

 f => Math.Round(f.Random.Decimal

 (1000, 5000), 2));

 return productFaker.

 Generate(count: 5).ToArray();

}

Invoke the GenerateProductData method in the OnModelCreating method to populate the database with randomly generated data, as shown in the following piece of code:

protected override

void OnModelCreating

(ModelBuilder modelBuilder)

{

 modelBuilder.Entity

 <Models.Product> ().

 ToTable("Product");

 modelBuilder.Entity

 <Models.Product> ().

 HasKey(p => p.Product\_Id);

 var products =

 GenerateProductData();

 modelBuilder.Entity

 <Models.Product> ().

 HasData(products);

}

The complete source code of the ProductDbContext class is given in **Listing 2**.

Register the ProductDb data context instance as a service with the services container of ASP.NET Core using the following piece of code in the Program.cs file.

builder.Services.AddDbContext

<ProductDbContext>(options =>

options.UseSqlServer(@"Data Source=

<<Specify the data source here>>;

Initial Catalog=

<<Specify the initial catalog here>>;

Trusted\_Connection=True;

TrustServerCertificate=True;

Integrated Security=True;"));

If your application needs to perform multiple units of work, it’s advisable to use a DbContext factory instead. To do this, register a factory by calling the AddDbContextFactory method in the Program.cs file of your project, as shown in the following code example:

builder.Services.AddDbContextFactory

<ProductDbContext>(options =>

{

 options.UseSqlServer

(builder.Configuration

["ConnectionStrings:SCSDbSettings"]);

});

## Query Product Data from the Database

In this example, you'll create the following commands and queries:

* Commands
	+ Create
	+ Update
	+ Delete
* Queries
	+ Get
	+ GetAll

Note that any command class should extend the IRequest<T> interface pertaining to the MediatR library.

The DbContext class in the EF Core library is not thread safe. Refrain from sharing data context between threads.

### The GetProductById Query

To read data from the database, take advantage of queries and query handers. In this example, you'll implement two types of queries and query handlers, one of them to read a product record by ID and the other to retrieve a list of all product records from the data store.

The following code snippet illustrates how you can define a query named GetProductByIdQuery inside the /Queries solution folder of the project to retrieve a product record from the database based on the product ID.

using MediatR;

namespace SCS.Product.Queries

{

 public record GetProductByIdQuery :

 IRequest<Models.Product>

 {

 public Guid Id { get; set; }

 }

}

Next, create a new .cs file named GetProductByIdHandler.cs inside the /Queries solution folder that contains a class named GetProductByIdQueryHandler that, in turn, encapsulates the logic required to retrieve a product record from the data store, as shown in **Listing 3**.

### The GetAllProductsQuery

Next, create a file named GetAllProductsQuery in a file having the same name with a .cs extension inside the /Queries folder and write the following code in there:

using MediatR;

namespace SCS.Product.Queries

{

 public record GetAllProductsQuery :

 IRequest<List<Models.Product>>;

}

The GetAllProductsQueryHandler class encapsulates the logic for retrieving a list of all product records from the database, as shown in **Listing 4**.

## Create, Update, and Delete Products

Now that you know how to query data from the Product database table, let’s understand how you can create a new product record, update an existing product record, and delete a product record from the database. To do this, you need to create commands to handle each of the Create, Update, and Delete operations.

### Create the CreateProductCommand

Create a new .cs file named CreateProductCommand.cs inside the /Commands folder of the project and replace the default generated source code with the code snippet given below:

using MediatR;

namespace SCS.Product.Commands

{

 public record CreateProductCommand :

 IRequest<Models.Product>

 {

 public Guid Id { get; set; }

 public string Product\_Name

 { get; set; } = default!;

 public string Product\_Description

 { get; set; } = default!;

 public string Product\_Category

 { get; set; } = default!;

 public decimal Product\_Price

 { get; set; } = default!;

 public DateTime Created\_At

 { get; set; } = DateTime.Now;

 public DateTime Modified\_At

 { get; set; } = DateTime.Now;

 }

}

The CreateProductCommand should have a corresponding handler to persist the product data into the data store. To do this, create another .cs file named CreateProductCommandHandler.cs and replace the default generated source code with the code given in **Listing 5**.

### Code Explanation

The following series of steps explain how the CreateProductHandler works:

* The CreateProductCommandHandler contains an asynchronous method named Handle that accepts an instance of CreateProductCommand and a CancellationToken object as a parameter.
* Inside the Handle method, a new Product object is created.
* This new object is populated with the data retrieved from the CreateProductCommand instance passed to the Handle method as a parameter.
* The new Product instance is finally saved into the database by making a call to the SaveChangesAsync method on the data context instance.
* The product ID of the newly created record is then returned.

### Create the UpdateProductCommand

Now, create a command named UpdateProductCommand under the /Commands solution folder to update a product record in the database, as shown in the code snippet given below:

using MediatR;

namespace SCS.Product.Commands

{

 public record UpdateProductCommand :

 IRequest<Product>

 {

 public Guid Product\_Id

 { get; set; }

 public string

 Product\_Name

 { get; set; } = default!;

 public string

 Product\_Description

 { get; set; } = default!;

 public string

 Product\_Category

 { get; set; } = default!;

 public decimal Product\_Price

 { get; set; } = default!;

 public DateTime Created\_At

 { get; set; } = DateTime.Now;

 public DateTime Modified\_At

 { get; set; } = DateTime.Now;

 }

}

The UpdateProductCommand should have a corresponding handler to update a product record based on the product ID. To do this, create another .cs file named UpdateProductCommandHandler.cs and write the source code given in **Listing 6** in there.

### Code Explanation

The following series of steps explain how the UpdateProductHandler works:

* The UpdateProductCommandHandler contains an asynchronous method named Handle that accepts an instance of UpdateProductCommand and a CancellationToken object as a parameter.
* Inside the Handle method, the Product record based on its ID is retrieved from the database.
* This Product object is updated with the data retrieved from the UpdateProductCommand instance passed to the Handle method as a parameter.
* The updated Product instance is finally saved into the database by making a call to the SaveChangesAsync method on the data context instance.
* The Product instance is then returned.

### Create the DeleteProductCommand

Next, create another command named DeleteProductCommand under the /Commands solution folder to delete a product record in the database, as shown in the code snippet given below.

using MediatR;

namespace SCS.Product.Commands

{

 public record

 DeleteProductCommand(Guid Id) :

 IRequest;

}

The DeleteProductCommand should also have a corresponding handler to delete a product record based on the product ID. To do this, create another .cs file named DeleteProductCommandHandler.cs and write the source code given in **Listing 7** in there.

### Code Explanation

The following series of steps explain how the DeleteProductHandler works:

* The DeleteProductCommandHandler contains an asynchronous method named Handle that accepts an instance of DeleteProductCommand and a CancellationToken object as a parameter.
* Inside the Handle method, the Product record to be deleted based on its ID is retrieved from the database.
* The Product instance is finally deleted by making a call to the SaveChangesAsync method on the data context instance.
* An appropriate exception message is thrown if the ProductId pertaining to the product record to be deleted doesn’t exist in the database.

Create the ProductRepository Class

A repository class is an implementation of the Repository design pattern and is one that manages data access. The application takes advantage of the repository instance to perform CRUD operations against the database. Now, create a new class named ProductRepository in a file having the same name with a .cs extension. Then write the following code in there:

public class ProductRepository :

IProductRepository

{

}

The ProductRepository class, illustrated in the code snippet below, implements the methods of the IProductRepository interface. Here is how the IProductRepository interface should look:

namespace SCS.Product.DataAccess

{

 public interface IProductRepository

 {

 public Task<Ienumerable

 <Models.Product>> GetAllAsync();

 public Task<Models.Product>

 GetByIdAsync(Guid id);

 public Task CreateAsync

 (Models.Product product);

 public Task UpdateAsync

 (Models.Product product);

 public Task DeleteAsync

 (Models.Product product);

 }

}

In the Product model class, you’ll observe the usage of Models.Product when referring to the Product class. This is needed because the names of the project and the model class are identical. You can avoid this by using different names anyway.

The complete source code of the ProductRepository class is given in **Listing 8**.

Create the ProductController Class

Now, create a new controller named ProductController in the Controllers folder of the project. The following code snippet shows how you can take advantage of constructor injection to pass an instance of the query handler using the constructor and then use it to retrieve all product records from the database.

private readonly IRequestHandler

 <GetAllProductsQuery,

 IEnumerable

 <Models.Product>>

 \_getAllProductQueryHandler;

public ProductController

(IRequestHandler

<GetAllProductsQuery, IEnumerable

<Models.Product>>

getAllProductsQueryHandler)

{

 \_getAllProductQueryHandler =

 getAllProductsQueryHandler;

}

[HttpGet("GetAllProducts")]

public async Task <IEnumerable

<Models.Product>> GetAllProducts()

{

 return await

 \_getAllProductQueryHandler.Handle

 (new GetAllProductsQuery(),

 new CancellationToken());

}

The following code snippet shows the action methods for creating, updating, and deleting product records. The first code snippet shows the CreateProduct action method that creates a new product record in the database.

[HttpPost(nameof(CreateProduct))]

public async Task<IActionResult>

CreateProduct(CreateProductCommand command)

{

 try

 {

 await \_createProductCommandHandler.Handle

 (command, new CancellationToken());

 return Ok

 ("Product record added successfully");

 }

 catch (Exception ex)

 {

 return StatusCode

 (StatusCodes.Status500InternalServerError,

 $"Error adding product: {ex.Message}");

 }

}

Similarly, the UpdateProduct action method shown in the code example below is used to update or alter an existing product record in the database.

[HttpPut(nameof(UpdateProduct))]

public async Task<IActionResult>

UpdateProduct(UpdateProductCommand command)

{

 try

 {

 await \_updateProductCommandHandler.Handle

 (command, new CancellationToken());

 return Ok("Product record updated

 successfully");

 }

 catch (Exception ex)

 {

 return StatusCode(StatusCodes.

 Status500InternalServerError,

 $"Error updating product: {ex.Message}");

 }

}

Finally, the DeleteProduct action method given below is responsible for deleting an existing product record in the database.

[HttpDelete("DeleteProduct")]

public async Task<IActionResult>

DeleteProduct(DeleteProductCommand

deleteProductCommand)

{

 try

 {

 await \_deleteProductCommandHandler.Handle

 (deleteProductCommand,

 new CancellationToken());

 return Ok

 ("Product record deleted successfully");

 }

 catch (Exception ex)

 {

 return StatusCode

 (StatusCodes.Status500InternalServerError,

 $"Error deleting product: {ex.Message}");

 }

}

The action verbs HttpGet, HttpPost, HttpPut, HttpDelete, and HttpPatch are specified using attributes in the action methods in ASP.NET Core to handle various types of requests. If you've not specified any action verb in an action method, the runtime will consider the request as a HttpGet request by default.

**Listing 9** shows the complete source of the ProductController class.

## Register the Service Instances with IServiceCollection

The following code snippet illustrates how you can register the IRequestHandler instances added as a transient service to the IServiceCollection.

builder.Services.AddScoped

<IProductRepository,

ProductRepository>();

builder.Services.AddTransient

<IRequestHandler

<GetProductByIdQuery,

Models.Product>> ();

builder.Services.AddTransient

<IRequestHandler

<GetAllProductsQuery,

IEnumerable<Models.Product>>,

GetAllProductsQueryHandler>();

builder.Services.AddTransient

<IRequestHandler

<CreateProductCommand,

Models.Product>,

CreateProductCommandHandler>();

builder.Services.AddTransient

<IRequestHandler

<UpdateProductCommand,

Models.Product>,

UpdateProductCommandHandler>();

builder.Services.AddTransient

<IRequestHandler

<DeleteProductCommand>,

DeleteProductCommandHandler>();

In addition, register the IproductRepository instance with the service collection, as shown below:

builder.Services.AddScoped

<IProductRepository, ProductRepository>();

The complete source code of the Program.cs file is given in **Listing 10**.

## Sequence Diagram of the GetAllProducts Flow

The sequence diagram of the GetAllProducts HttpGet flow is shown in **Figure 8**:



**Figure 8:** The sequence diagram of the GetAllProducts HttpGet flow

## Execute the Application

Finally, run the application and launch the popular API tool Postman. **Figure 9** shows the output upon execution of the getproductbyid endpoint.



**Figure 9:** The getproductbyid endpoint in action

When you run the createproduct endpoint and specify the details of the new product to be added to the database in the body of the request, a new product record is added to the Product table and the text “Product record added successfully” is returned as part of the response, as shown in **Figure 10**.



**Figure 10:** The createproduct endpoint in action

## Best Practices

Here are some of the best practices to follow when working with CQRS:

* Define distinct Commands and Queries.
* Commands in a CQRS implementation should be task-based (i.e., CreateOrder, UpdateUser) and not data-centric.
* Queries in a typical CQRS implementation should only return data without changing the state.
* Your write model should be normalized to ensure data integrity and consistency.
* Take advantage of MediatR for mediating commands and queries.
* You can leverage AutoMapper to transfer domain models to data transfer objects.

## Where Do We Go from Here?

This article taught you about the CQRS design pattern, its benefits, components, challenges, and how to use it in microservices-based applications. You also learned how to take advantage of this design pattern to implement microservices-based applications that are scalable, efficient, and maintainable. By splitting the application's CRUD operations into two sections, namely, the command and query sides, the CQRS pattern promotes flexibility in designs, enhances security, improves performance, and helps scale your application easily. I'll discuss more design patterns related to microservices architecture in future articles.

Sponsored Sidebar

CODE Is Hiring!

**CODE Staffing** is accepting resumes for various open positions ranging from junior to senior roles. We have **multiple openings** and will consider candidates who seek full-time employment or contracting opportunities.

**For more information:** [www.codestaffing.com](http://www.codestaffing.com).

Listing 1: Create the database tables

CREATE TABLE Product (

 [Product\_Id]

 UniqueIdentifier PRIMARY KEY,

 [Name] varchar(255) NOT NULL,

 [Description] Text NOT NULL,

 [Category] varchar(50) NOT NULL,

 [Price] DECIMAL(10, 2),

 [Quantity] INT,

 [Created\_At] DATETIME,

 [Modified\_At] DATETIME

);

CREATE TABLE Customer (

 [Customer\_Id]

 UniqueIdentifier PRIMARY KEY,

 [FirstName] VARCHAR(50),

 [LastName] VARCHAR(50),

 Email VARCHAR(100),

 [Address] VARCHAR(255),

 [Phone] VARCHAR(15),

 [Created\_At] DATETIME,

 [Modified\_At] DATETIME

);

CREATE TABLE [Order] (

 [Order\_Id]

 UniqueIdentifier PRIMARY KEY,

 [Customer\_Id] UniqueIdentifier,

 [OrderDate] TIMESTAMP,

 [TotalAmount] DECIMAL(10, 2),

 [Created\_At] DATETIME,

 [Modified\_At] DATETIME,

 FOREIGN KEY (Customer\_Id)

 REFERENCES Customer(Customer\_Id)

);

CREATE TABLE OrderItem (

 [OrderItem\_Id]

 UniqueIdentifier PRIMARY KEY,

 [Order\_Id] UniqueIdentifier,

 [Product\_Id] UniqueIdentifier,

 [Quantity] INT,

 [Price] DECIMAL(10, 2),

 [Created\_At] DATETIME,

 [Modified\_At] DATETIME,

 FOREIGN KEY (Order\_Id)

 REFERENCES [Order](Order\_Id),

 FOREIGN KEY (Product\_Id)

 REFERENCES Product(Product\_Id)

);

CREATE TABLE Cart (

 [Cart\_Id]

 UniqueIdentifier PRIMARY KEY,

 [Customer\_Id] UniqueIdentifier,

 [Created\_At] DATETIME,

 [Modified\_At] DATETIME,

 FOREIGN KEY (Customer\_Id)

 REFERENCES Customer(Customer\_Id)

);

Listing 2: The ProductDbContext class

using Bogus;

using Microsoft.EntityFrameworkCore;

namespace SCS.Product.DataAccess

{

 public class ProductDbContext : DbContext

 {

 private readonly IConfiguration \_configuration;

 public ProductDbContext

 (DbContextOptions<ProductDbContext> options,

 IConfiguration configuration) : base(options)

 {

 \_configuration = configuration;

 }

 protected override void OnConfiguring

 (DbContextOptionsBuilder optionsBuilder)

 {

 \_ = optionsBuilder.UseSqlServer(\_configuration.

 GetConnectionString("SCSDbSettings")).

 EnableSensitiveDataLogging();

 }

 public DbSet<Models.Product>

 Products { get; set; }

 protected override void OnModelCreating

 (ModelBuilder modelBuilder)

 {

 modelBuilder.Entity< Models.Product>().

 ToTable("Product");

 modelBuilder.Entity< Models.Product>().

 HasKey (p => p.Product\_Id);

 var products = GenerateProductData();

 modelBuilder.Entity<Models.Product>().

 HasData(products);

 }

 private Models.Product[] GenerateProductData()

 {

 var productFaker = new

 Faker<SCS.Product.Models.Product>()

 .RuleFor(x => x.Product\_Id,

 f => Guid.NewGuid())

 .RuleFor(x => x.Product\_Name,

 f => f.Commerce.ProductName())

 .RuleFor(x => x.Product\_Description,

 f => f.Commerce.ProductDescription())

 .RuleFor(x => x.Product\_Category,

 f => f.Commerce.ProductMaterial())

 .RuleFor(x => x.Product\_Price,

 f => Math.Round(f.Random.Decimal

 (1000, 5000), 2));

 return productFaker.Generate

 (count: 5).ToArray();

 }

 }

}

Listing 3: The GetProductByIdQueryHandler class

using MediatR;

using SCS.Product.DataAccess;

namespace SCS.Product.Queries

{

 public class GetProductByIdQueryHandler :

 IRequestHandler<GetProductByIdQuery,

 Models.Product>

 {

 private readonly

 IProductRepository \_repository;

 public GetProductByIdQueryHandler

 (IProductRepository repository)

 {

 \_repository = repository;

 }

 public async Task<Models.Product>

 Handle(GetProductByIdQuery productRequest,

 CancellationToken cancellationToken)

 {

 return await

 \_repository.GetByIdAsync

 (productRequest.Id);

 }

 }

}

Listing 4: The GetAllProductsQueryHandler

using MediatR;

using SCS.Product.DataAccess;

namespace SCS.Product.Queries

{

 public class GetAllProductsQueryHandler :

 IRequestHandler<GetAllProductsQuery,

 IEnumerable<Models.Product>>

 {

 private readonly IProductRepository \_repository;

 public GetAllProductsQueryHandler

 (IProductRepository repository)

 {

 \_repository = repository;

 }

 public async Task<IEnumerable<Models.Product>>

 Handle(GetAllProductsQuery request,

 CancellationToken cancellationToken)

 => await \_repository.GetAllAsync();

 }

}

Listing 5: The CreateProductCommandHandler class

using MediatR;

using SCS.Product.DataAccess;

namespace SCS.Product.Commands

{

 public class CreateProductCommandHandler

 (IProductRepository productRepository) :

 IRequestHandler<CreateProductCommand,

 Models.Product>

 {

 public async Task<Models.Product>

 Handle(CreateProductCommand productCommand,

 CancellationToken cancellationToken)

 {

 var product = new Models.Product

 {

 Product\_Name = productCommand.Product\_Name,

 Product\_Description =

 productCommand.Product\_Description,

 Product\_Category =

 productCommand.Product\_Category,

 Product\_Price = productCommand.Product\_Price,

 Created\_At = DateTime.UtcNow,

 Modified\_At = DateTime.UtcNow

 };

 await productRepository.CreateAsync(product);

 return product;

 }

 }

}

Listing 6: The UpdateProductCommandHandler class

using MediatR;

using SCS.Product.DataAccess;

namespace SCS.Product.Commands

{

 public class UpdateProductCommandHandler

 (IProductRepository productRepository) :

 IRequestHandler<UpdateProductCommand,

 Models.Product>

 {

 public async Task<Models.Product> Handle

 (UpdateProductCommand productRequest,

 CancellationToken cancellationToken)

 {

 var product = await

 productRepository.GetByIdAsync

 (productRequest.Id);

 if (product != null)

 {

 product.Product\_Name =

 productRequest.Product\_Name;

 product.Product\_Description =

 productRequest.Product\_Description;

 product.Product\_Category =

 productRequest.Product\_Category;

 product.Product\_Price =

 productRequest.Product\_Price;

 product.Modified\_At = DateTime.UtcNow;

 await productRepository.UpdateAsync(product);

 return product;

 }

 return default;

 }

 }

}

Listing 7: The DeleteProductCommandHandler class

using MediatR;

using SCS.Product.DataAccess;

namespace SCS.Product.Commands

{

 public class DeleteProductCommandHandler

 (IProductRepository productRepository) :

 IRequestHandler<DeleteProductCommand>

 {

 public async Task Handle

 (DeleteProductCommand productRequest,

 CancellationToken cancellationToken)

 {

 var product = await

 productRepository.

 GetByIdAsync(productRequest.Id);

 if (product != null)

 {

 await productRepository.

 DeleteAsync(product);

 }

 else

 {

 throw new Exception

 ($"Product with Id

 {productRequest.Id} not found.");

 }

 }

 }

}

Listing 8: The ProductRepository Class

using Microsoft.EntityFrameworkCore;

namespace SCS.Product.DataAccess

{

 public class ProductRepository : IProductRepository

 {

 private readonly ProductDbContext \_dbContext;

 public ProductRepository

 (ProductDbContext dbContext)

 {

 \_dbContext = dbContext;

 \_dbContext.Database.EnsureCreated();

 }

 public async Task<Models.Product>

 GetByIdAsync(Guid id)

 {

 return await

 \_dbContext.Set

 <Models.Product>().FindAsync(id);

 }

 public async Task<Ienumerable

 <Models.Product>> GetAllAsync()

 {

 return await

 \_dbContext.Set

 <Models.Product>().ToListAsync();

 }

 public async Task CreateAsync

 (Models.Product entity)

 {

 await \_dbContext.Set<Models.Product>().

 AddAsync(entity);

 await \_dbContext.SaveChangesAsync();

 }

 public async Task UpdateAsync

 (Models.Product entity)

 {

 \_dbContext.Set<Models.Product>().

 Update(entity);

 await \_dbContext.SaveChangesAsync();

 }

 public async Task DeleteAsync(Models.Product entity)

 {

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

 await \_dbContext.SaveChangesAsync();

 }

 }

}

Listing 9: The ProductController class

using MediatR;

using Microsoft.AspNetCore.Mvc;

using SCS.Product.Commands;

using SCS.Product.Queries;

namespace SCS.Product.Controllers

{

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

 [ApiController]

 public class ProductController: Controller

 {

 private readonly IRequestHandler

 <GetProductByIdQuery, Models.Product>

 \_getProductByIdQueryHandler;

 private readonly IRequestHandler

 <GetAllProductsQuery , IEnumerable

 <Models.Product>>

 \_getAllProductsQueryHandler;

 private readonly IRequestHandler

 <CreateProductCommand , Models.Product>

 \_createProductCommandHandler;

 private readonly IRequestHandler

 <UpdateProductCommand , Models.Product>

 \_updateProductCommandHandler;

 private readonly IRequestHandler

 <DeleteProductCommand>

 \_deleteProductCommandHandler;

 public ProductController(

 IRequestHandler <GetProductByIdQuery,

 Models.Product>

 getProductByIdQueryHandler,

 IRequestHandler <CreateProductCommand,

 Models.Product>

 createProductCommandHandler,

 IRequestHandler

 <GetAllProductsQuery,

 IEnumerable <Models.Product>>

 getAllProductsQueryHandler,

 IRequestHandler

 <UpdateProductCommand,

 Models.Product>

 updateProductCommandHandler,

 IRequestHandler

 <DeleteProductCommand >

 deleteProductCommandHandler)

 {

 \_getProductByIdQueryHandler = =

 getProductByIdQueryHandler;

 \_getAllProductsQueryHandler = =

 getAllProductsQueryHandler;

 \_createProductCommandHandler = =

 createProductCommandHandler;

 \_updateProductCommandHandler = =

 updateProductCommandHandler;

 \_deleteProductCommandHandler = =

 deleteProductCommandHandler;

 }

 [HttpGet("GetProductById")]

 public async Task <Models.Product>

 GetProductById

 (GetProductByIdQuery getProductByIdQuery)

 {

 return await

 \_getProductByIdQueryHandler.Handle

 (getProductByIdQuery,

 new CancellationToken());

 }

 [HttpGet(nameof(GetAllProducts))]

 public async Task <IEnumerable

 <Models.Product>>

 GetAllProducts()

 {

 return await

 \_getAllProductsQueryHandler.

 Handle

 (new GetAllProductsQuery(),

 new CancellationToken());

 }

 [HttpPost(nameof(CreateProduct))]

 public async Task <IActionResult>

 CreateProduct(CreateProductCommand command)

 {

 try

 {

 await

 \_createProductCommandHandler.

 Handle

 (command,

 new CancellationToken());

 return Ok

 ("Product record added successfully");

 }

 catch (Exception ex)

 {

 return StatusCode(StatusCodes.

 Status500InternalServerError, $

 "Error adding

 product: {ex.Message}");

 }

 }

 [HttpPut(nameof(UpdateProduct))]

 public async Task <IActionResult>

 UpdateProduct(UpdateProductCommand command)

 {

 try

 {

 await \_updateProductCommandHandler.

 Handle(command,

 new CancellationToken());

 return Ok

 ("Product record updated successfully");

 }

 catch (Exception ex)

 {

 return StatusCode(StatusCodes.

 Status500InternalServerError,

 $ "Error updating

 product: {ex.Message}");

 }

 }

 [HttpDelete("DeleteProduct")]

 public async Task <IActionResult>

 DeleteProduct(DeleteProductCommand

 deleteProductCommand)

 {

 try

 {

 await\_deleteProductCommandHandler.Handle

 (deleteProductCommand,

 new CancellationToken());

 return Ok

 ("Product record deleted successfully");

 }

 catch (Exception ex)

 {

 return StatusCode(StatusCodes.

 Status500InternalServerError,

 $ "Error deleting

 product: {ex.Message}");

 }

 }

 }

}

Listing 10: The Complete Program.cs file

global using Models = SCS.Product.Models;

using MediatR;

using Microsoft.EntityFrameworkCore;

using SCS.Product.Commands;

using SCS.Product.DataAccess;

using SCS.Product.Queries;

using System.Reflection;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddMediatR

(cfg => cfg.RegisterServicesFromAssembly

(Assembly.GetExecutingAssembly()));

builder.Services.AddScoped

<IProductRepository, ProductRepository>();

builder.Services.AddControllers();

builder.Services.AddDbContext

<ProductDbContext>(options =>

{

options.UseSqlServer

(builder.Configuration

["ConnectionStrings:SCSDbSettings"]);

});

var app = builder.Build();

// Configure the HTTP request pipeline.

app.UseAuthorization();

app.MapControllers();

app.Run();