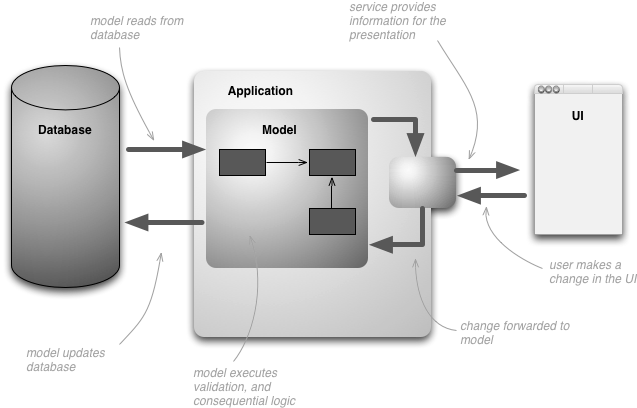
**Command Query Responsibility Segregation (CQRS)**

**The Traditional CRUD Approach**

The traditional way of system design is creating **one model** which is being updated by UI events. We have one database schema and corresponding application model. The same model for reading and writing.

For example, all CRUD operations are applied to the same representation of the entity. A data transfer object (DTO) representing a customer is retrieved from the data store by the data access layer (DAL) and displayed on the screen. A user updates some fields of the DTO (perhaps through data binding) and the DTO is then saved back in the data store by the DAL. The same DTO is used for both the read and write operations



**The traditional CRUD approach has some disadvantages:**

* There is a **mismatch** between the read and write representations of the data, such as **additional** **columns** **or properties** that must be updated correctly even though they are not required as part of an operation.
* It risks **data conflict** when records are locked in the data store in a collaborative domain, where multiple actors operate in parallel on the same set of data. Or update conflicts caused by concurrent updates when **optimistic locking** is used. These risks increase as the complexity and throughput of the system grows. In addition, the traditional approach can have a **negative effect on performance** due to load on the data store and data access layer and the complexity of queries required to retrieve information.
* It can make **managing security and permissions** more complex because each entity is subject to both read and write operations, which might expose data in the wrong context.

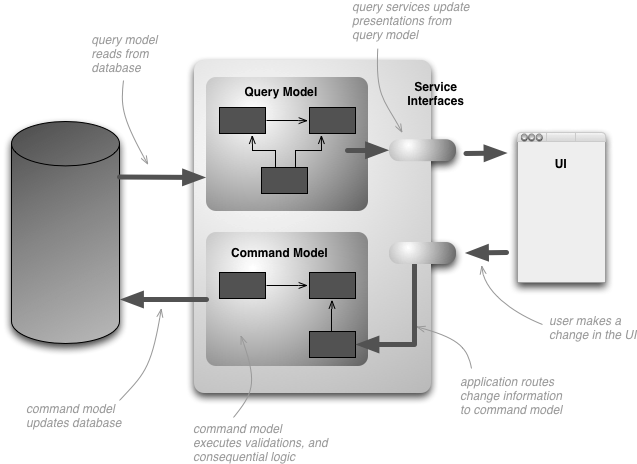
**Understanding CQRS – Command Query Responsibility Segregation:**

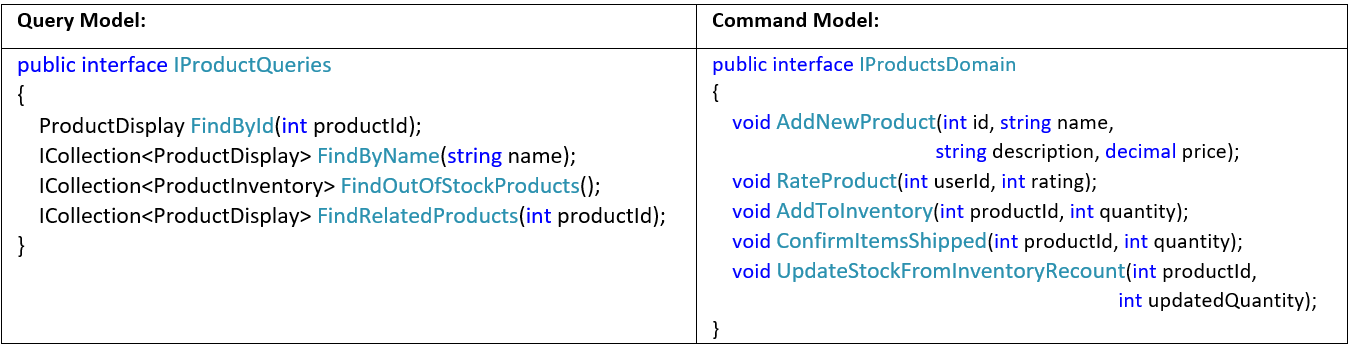
The CQRS pattern provides a guideline for identifying services and distributing various business aspects. The basic idea is to divide the operations that act on a **domain object** into two distinct categories:

* **Queries**—methods that return a result and do not change the system state.
* **Commands**—methods that change the system state but do not return values.

**Command Query Separation (CQS)** is a simple concept—it is about methods within the **same object** being either queries or commands. Each method either returns state or mutates state, but not both.

* In CQRS, the **data model** used for querying and model for updates are different.
* The separation aspect of CQRS is achieved by grouping **query operations in one layer** and **commands in another layer**.
* Each layer has its own data model (note that we say model, not necessarily a different database).
* The two layers can be within the **same tier or microservice** or they could be implemented on **different microservices or processes** so they can be optimized and scaled out separately without affecting one another.

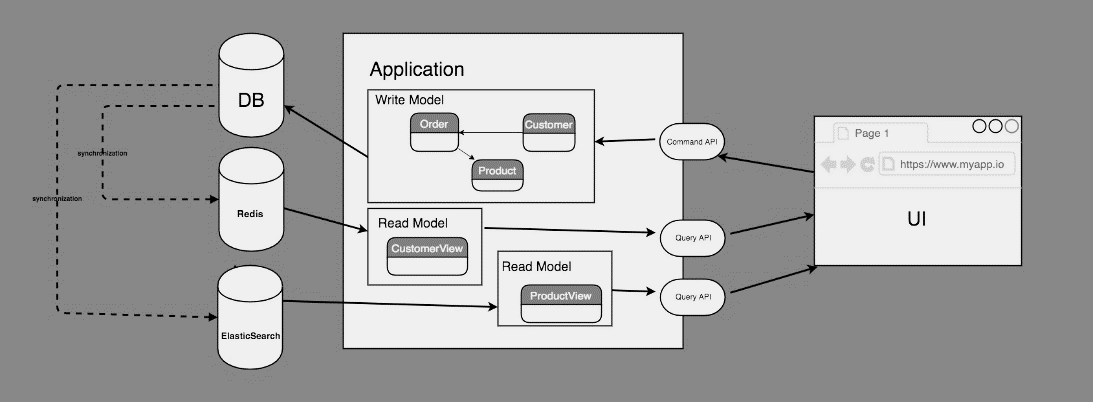




Notice how the IProductsDomain interface contains methods that have a meaning in the domain. Typically, in a CRUD environment these methods would have generic names such as **Save** or **Update**, and have a DTO as the only argument. The CQRS approach can be designed to meet the needs of this organization's business and inventory management systems.

**Same or Different Physical Store:**

The query model for reading data and the update model for writing data can access the **same physical store**, perhaps by using SQL views (or by generating projections on the fly). However, it is common to separate the data into **different physical stores** to **maximize performance, scalability and security**.



* The read store can be a **read-only replica** of the write store or the read and write stores can have a **different structure** altogether.
* Using **multiple read-only replicas** of the read store can greatly increase query performance and application UI responsiveness, especially in distributed scenarios where read-only replicas are **located close to the application instances**.
* Separation of the read and write stores also allows **each to be** **scaled appropriately to match the load**. Usually read stores typically encounter a much higher load than write stores. For example: In an ecommerce application, items details and images are uploaded once but are read as many items as the item is searched by the customers.

**Benefits of CQRS include:**

* **Independent scaling**. CQRS allows the read and write workloads to scale independently, and may result in fewer lock contentions.
* **Optimized data schemas**. The read side can use a schema that is optimized for queries, while the write side uses a schema that is optimized for updates.
* **Security**. It's easier to ensure that only the right domain entities are performing writes on the data.
* **Separation of concerns**. Segregating the read and write sides can result in models that are more maintainable and flexible. Most of the complex business logic goes into the write model. The read model can be relatively simple.
* **Simpler queries**. By storing a materialized view in the read database, the application can avoid complex joins when querying.

**Some challenges of implementing this pattern include:**

* **Complexity**. The basic idea of CQRS is simple. But it can lead to a more complex application design, especially if they include the Event Sourcing pattern.
* **Messaging**. Although CQRS does not require messaging, it's common to use messaging to process commands and publish update events. In that case, the application must handle message failures or duplicate messages. See the guidance on Priority Queues for dealing with commands having different priorities.
* **Eventual consistency**. If you separate the read and write databases, the read data may be stale. The read model store must be updated to reflect changes to the write model store, and it can be difficult to detect when a user has issued a request based on stale read data.

**When to use CQRS pattern**

* Collaborative domains where multiple operations are performed in parallel on the same data. CQRS allows you to define **commands with enough granularity** to minimize merge conflicts at the domain level. Eg: AddNewProductCommand, RateProductCommand, AddToInventoryCommand, ConfirmItemShippedCommand
* Task-based user interfaces where users are guided through a **complex process** as a series of steps or with complex domain models. The write model has a full command-processing stack with **business logic, input validation and business validation** to ensure that everything is always consistent in the write model. The read model has no business logic or validation stack and just returns a DTO for use in a view model. The read model is **eventually consistent** with the write model.
* Scenarios where **performance** of data reads must be fine-tuned separately from performance of data writes, especially when the read/write ratio is very high and when horizontal scaling is required.
* Scenarios where one team of **developers can focus** on the complex domain model that is part of the write model and another team can focus on the read model and the user interfaces.
* Scenarios where the system is expected to **evolve** over time and might contain multiple versions of the model or where **business rules change regularly**.
* Integration with other systems, especially in combination with event sourcing, where the temporal failure of one subsystem shouldn't affect the availability of the others.

**This pattern is not recommended in the following situations:**

* Where the domain or the business rules are simple.
* Where a simple CRUD-style user interface and the related data access operations are sufficient.
* There are specific components of an overall data management scenario where CQRS can be useful, but it can add considerable and unnecessary complexity when it is not required.

**Implementing reads/queries in a CQRS microservice**

**Product ReadModel Microservice**

The following code shows the read model definition. Note how the **names of the methods** are defined.

// Query interface

namespace **ReadModel**

{

public interface **IProductQueries**

{

ProductDisplay FindById(int productId);

ICollection<ProductDisplay> FindByName(string name);

ICollection<ProductInventory> FindOutOfStockProducts();

ICollection<ProductDisplay> FindRelatedProducts(int productId);

}

public class **ProductDisplay**

{

public int Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public decimal UnitPrice { get; set; }

public bool IsOutOfStock { get; set; }

public double UserRating { get; set; }

}

public class **ProductInventory**

{

public int Id { get; set; }

public string Name { get; set; }

public int CurrentStock { get; set; }

}

}

**Implementing Update Model in a CQRS microservice**

The system allows users to rate products. The application code does this using the RateProduct command shown in the following code.

public interface ICommand

{

Guid Id { get; }

}

public class RateProductCommand : ICommand

{

public RateProductCommand()

{

this.Id = Guid.NewGuid();

}

public int ProductId { get; set; }

public int Rating { get; set; }

public int UserId {get; set; }

}

The system uses the **ProductsCommandHandler** class to handle commands sent by the application. Clients typically send commands to the domain through a messaging system such as a queue. The command handler accepts these commands and invokes methods of the domain interface. **The granularity of each command is designed to reduce the chance of conflicting requests**.

The following code shows an outline of the ProductsCommandHandler class.

public class ProductsCommandHandler:

ICommandHandler<AddNewProductCommand>,

ICommandHandler<RateProductCommand>,

ICommandHandler<AddToInventoryCommand>,

ICommandHandler<ConfirmItemShippedCommand>,

ICommandHandler<UpdateStockFromInventoryRecountCommand>

{

private readonly IRepository<Product> repository;

public ProductsCommandHandler (IRepository<Product> repository)

{

this.repository = repository;

}

void Handle (**AddNewProductCommand** command)

{ ... }

void **Handle** (**RateProductCommand** command)

{

**//Get Product Entity from the Domain Driven Layer**

**IProductDomain** product = repository.Find(command.ProductId);

if (product != null)

{

**//Invoke the DDD specific behavior if any**

product.RateProduct(command.UserId, command.Rating);

**//Use the Infrastructure Layer repository to save**

repository.Save(product);

}

}

void Handle (**AddToInventoryCommand** command)

{ ... }

void Handle (**ConfirmItemsShippedCommand** command)

{ ... }

void Handle (**UpdateStockFromInventoryRecountCommand** command)

{ ... }

}

The following code shows the IProductsDomain interface from the write model.

public interface **IProductsDomain**

{

void AddNewProduct(int id, string name, string description, decimal price);

void RateProduct(int userId, int rating);

void AddToInventory(int productId, int quantity);

void ConfirmItemsShipped(int productId, int quantity);

void UpdateStockFromInventoryRecount(int productId, int updatedQuantity);

}