CQRS Pattern: -

**What is CQRS?**

**==============================**

CQRS (Command Query Responsibility Segregation) is a software design pattern that separates **read (query)** and **write (command)** operations into different models. This means that instead of using a single model for both retrieving and updating data, you have **separate models** optimized for each.

* **Command Model**: Handles **write** operations (Create, Update, Delete).
* **Query Model**: Handles **read** operations (Retrieve data).

**Why Do We Need CQRS?**

CQRS is useful in complex applications where performance, scalability, and maintainability are critical. Here’s why you might need it:

1. **Improved Performance & Scalability**
   * Reads and writes often have different performance and scaling requirements.
   * By separating them, you can **scale** reads and writes independently.
   * Example: A high-traffic e-commerce app where reads happen much more frequently than writes.
2. **Optimized Data Models**
   * The **read model** can be denormalized and optimized for fast queries.
   * The **write model** can be normalized to ensure consistency and business rules.
3. **Better Maintainability & Flexibility**
   * Separation of concerns makes the system **easier to understand and modify**.
   * Different teams can work on the **read** and **write** sides independently.
4. **Event Sourcing Compatibility**
   * CQRS works well with **Event Sourcing**, where every change is stored as an event.
   * This allows **rebuilding state**, **audit logs**, and **replaying history**.
5. **Security & Fault Tolerance**
   * Commands can be strictly validated before processing.
   * Read operations can be optimized with **caching** or **read replicas**.

**When to Use CQRS?**

* Large-scale applications with **high read and write workloads**.
* Applications requiring **event-driven architecture**.
* Systems with **complex domain logic** (e.g., banking, e-commerce, IoT).
* When **performance bottlenecks** are caused by a shared data model.

**When NOT to Use CQRS?**

* **Small applications** where a single database model is enough.
* Systems that don’t require **high scalability or complex business logic**

**Components of CQRS Pattern: -**

**================================**

**1. Command Side (Write Model)**

🔹 Commands

🔹 Command Handlers

🔹 Domain Model (Aggregates & Entities)

🔹 Event Store (Optional - Used with Event Sourcing)

**2️. Query Side (Read Model)**

🔹 Queries

🔹 Query Handlers

🔹 Read Model (DTOs)

**3️. Infrastructure Components**

🔹 MediatR (Command & Query Dispatcher)

🔹 Separate Databases (Optional)

🔹 Event Bus (Optional)

**Is Dispatcher a Component in CQRS Pattern?**

Yes, a **Dispatcher** is an optional but commonly used component in the CQRS pattern. It acts as a **mediator** that routes **commands and queries** to their respective handlers, ensuring **loose coupling** between different parts of the system.

**Dispatcher in MediatR (Commonly Used in .NET CQRS)**

In .NET, **MediatR** is widely used as a **Dispatcher** to send commands and queries.

**Alternatives to Dispatcher in CQRS**

* **MediatR (for .NET)**
* **Simple Command Bus or Query Bus**
* **Custom Dispatcher Service**
* **Event Bus (Kafka, RabbitMQ) for Distributed Systems**

**How is Mapping Done Between Command and Command Handler in MediatR?**

In **MediatR**, the mapping between a **Command** and its corresponding **Command Handler** is done automatically by the MediatR library using **dependency injection**. MediatR scans the assembly for **handler implementations** and associates them with their respective **commands or queries**.

✅ **How it Works?**

* MediatR **scans the assembly** for classes implementing IRequestHandler<TRequest, TResponse>.
* It **automatically maps** CreateOrderCommand → CreateOrderHandler.

**Difference Between Dispose() and Finalize () in C# ?**

**Key Differences**

| **Feature** | **Dispose()** | **Finalize()** |
| --- | --- | --- |
| **Trigger** | **Called manually by the user** | **Called automatically by GC** |
| **Interface** | **Implemented via IDisposable** | **Implemented as a destructor (~ClassName)** |
| **Performance** | **Fast (immediate resource release)** | **Slow (waits for GC cycle)** |
| **Resource Cleanup** | **Used for both managed & unmanaged resources** | **Used mostly for unmanaged resources** |
| **Calling Mechanism** | **Called using using block or explicitly** | **Called implicitly by GC** |
| **Reliability** | **Guaranteed execution** | **Execution timing is unpredictable** |
| **Explicit Call** | **Yes (by user)** | **No (only GC can call)** |
| **Best Practice** | **Preferred for cleanup** | **Fallback if Dispose() is not used** |