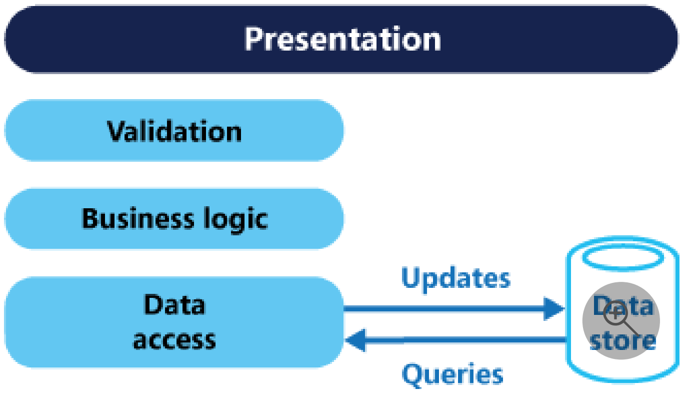
CQRS pattern

Article • 02/22/2025

Command Query Responsibility Segregation (CQRS) is a design pattern that segregates read and write operations for a data store into separate data models. This approach allows each model to be optimized independently and can improve the performance, scalability, and security of an application.

Context and problem

In a traditional architecture, a single data model is often used for both read and write operations. This approach is straightforward and is suited for basic create, read, update, and delete (CRUD) operations.



As applications grow, it can become increasingly difficult to optimize read and write operations on a single data model. Read and write operations often have different performance and scaling requirements. A traditional CRUD architecture doesn't take this asymmetry into account, which can result in the following challenges:

- **Data mismatch:** The read and write representations of data often differ. Some fields that are required during updates might be unnecessary during read operations.
- **Lock contention:** Parallel operations on the same data set can cause lock contention.
- **Performance problems:** The traditional approach can have a negative effect on performance because of load on the data store and data access layer, and the complexity of queries required to retrieve information.
- **Security challenges:** It can be difficult to manage security when entities are subject to read and write operations. This overlap can expose data in unintended contexts.

Combining these responsibilities can result in an overly complicated model.

Solution

Use the CQRS pattern to separate write operations, or *commands*, from read operations, or *queries*. Commands update data. Queries retrieve data. The CQRS pattern is useful in scenarios that require a clear separation between commands and reads.

- **Understand commands.** Commands should represent specific business tasks instead of low-level data updates. For example, in a hotel-booking app, use the command "Book hotel room" instead of "Set ReservationStatus to Reserved." This approach better captures the intent of the user and aligns commands with business processes. To help ensure that commands are successful, you might need to refine the user interaction flow and server-side logic and consider asynchronous processing.

Expand table

Area of refinement	Recommendation
Client-side validation	Validate specific conditions before you send the command to prevent obvious failures. For example, if no rooms are available, disable the "Book" button and provide a clear, user-friendly message in the UI that explains why booking isn't possible. This setup reduces unnecessary server requests and provides immediate feedback to users, which enhances their experience.
Server-side logic	Enhance the business logic to handle edge cases and failures gracefully. For example, to address race conditions such as multiple users attempting to book the last available room, consider adding users to a waiting list or suggesting alternatives.
Asynchronous processing	Process commands asynchronously by placing them in a queue, instead of handling them synchronously.

- **Understand queries.** Queries never alter data. Instead, they return data transfer objects (DTOs) that present the required data in a convenient format, without any domain logic. This distinct separation of responsibilities simplifies the design and implementation of the system.

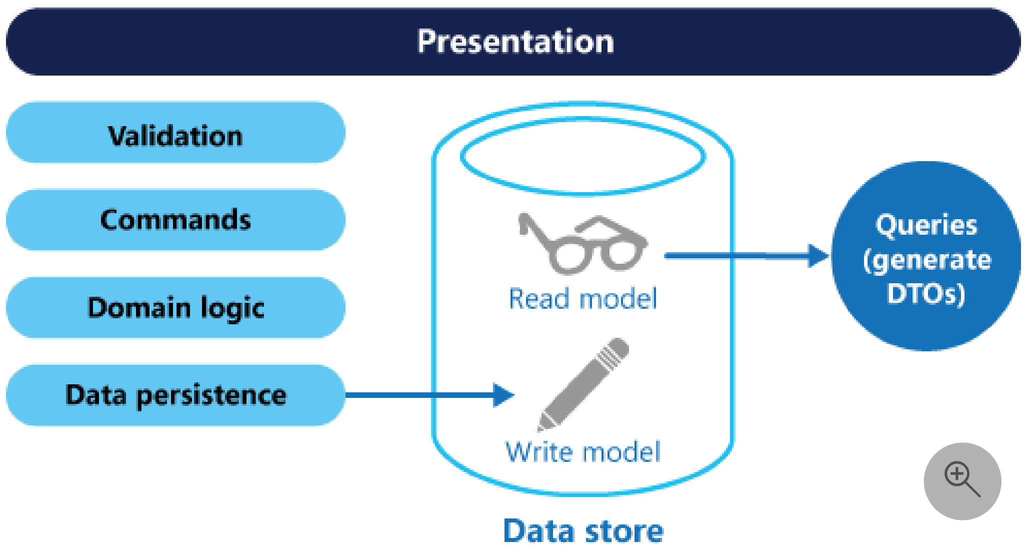
Separate read models and write models

Separating the read model from the write model simplifies system design and implementation by addressing specific concerns for data writes and data reads. This separation improves clarity, scalability, and performance but introduces trade-offs. For example, scaffolding tools like object-relational mapping (O/RM) frameworks can't automatically generate CQRS code from a database schema, so you need custom logic to bridge the gap.

The following sections describe two primary approaches to implement read model and write model separation in CQRS. Each approach has unique benefits and challenges, such as synchronization and consistency management.

Separate models in a single data store

This approach represents the foundational level of CQRS, where both the read and write models share a single underlying database but maintain distinct logic for their operations. A basic CQRS architecture allows you to delineate the write model from the read model while relying on a shared data store.

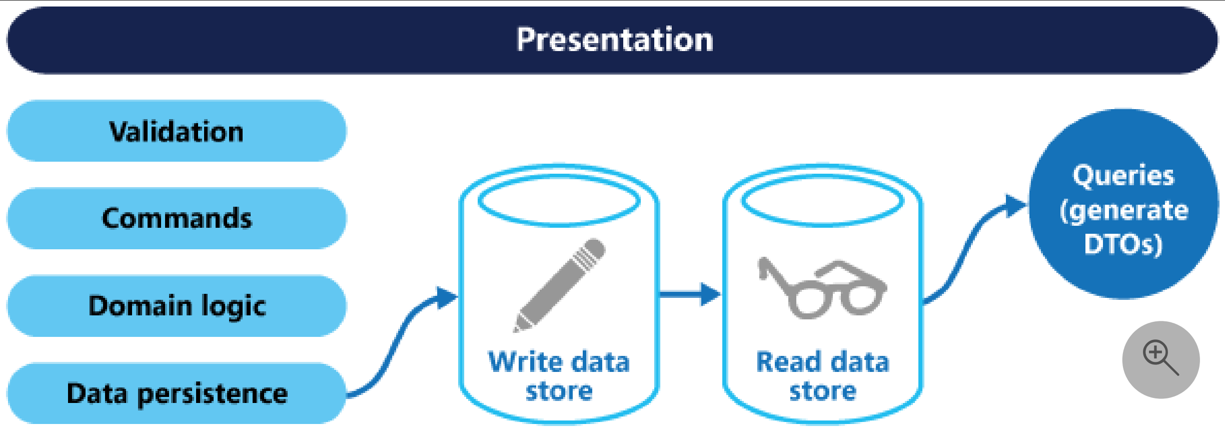


This approach improves clarity, performance, and scalability by defining distinct models for handling read and write concerns.

- **A write model** is designed to handle commands that update or persist data. It includes validation and domain logic, and helps ensure data consistency by optimizing for transactional integrity and business processes.
- **A read model** is designed to serve queries for retrieving data. It focuses on generating DTOs or projections that are optimized for the presentation layer. It enhances query performance and responsiveness by avoiding domain logic.

Separate models in different data stores

A more advanced CQRS implementation uses distinct data stores for the read and write models. Separation of the read and write data stores allows you to scale each model to match the load. It also enables you to use a different storage technology for each data store. You can use a document database for the read data store and a relational database for the write data store.



When you use separate data stores, you must ensure that both remain synchronized. A common pattern is to have the write model publish events when it updates the database, which the read model uses to refresh its data. For more information about how to use events, see Event-driven architecture style. Because you usually can't enlist message brokers and databases into a single distributed transaction, challenges in consistency can occur when you update the database and publishing events. For more information, see Idempotent message processing.

The read data store can use its own data schema that's optimized for queries. For example, it can store a materialized view of the data to avoid complex joins or O/RM mappings. The read data store can be a read-only replica of the write store or have a different structure. Deploying multiple read-only replicas can improve performance by reducing latency and increasing availability, especially in distributed scenarios.

Benefits of CQRS

- **Independent scaling.** CQRS enables the read models and write models to scale independently. This approach can help minimize lock contention and improve system performance under load.
- **Optimized data schemas.** Read operations can use a schema that's optimized for queries. Write operations use a schema that's optimized for updates.
- **Security.** By separating reads and writes, you can ensure that only the appropriate domain entities or operations have permission to perform write actions on the data.
- **Separation of concerns.** Separating the read and write responsibilities results in cleaner, more maintainable models. The write side typically handles complex business logic. The read side can remain simple and focused on query efficiency.
- **Simpler queries.** When you store a materialized view in the read database, the application can avoid complex joins when it queries.

Problems and considerations

Consider the following points as you decide how to implement this pattern:

- **Increased complexity.** The core concept of CQRS is straightforward, but it can introduce significant complexity into the application design, specifically when combined with the Event Sourcing pattern.

- **Messaging challenges.** Messaging isn't a requirement for CQRS, but you often use it to process commands and publish update events. When messaging is included, the system must account for potential problems such as message failures, duplicates, and retries. For more information about strategies to handle commands that have varying priorities, see Priority queues.

- **Eventual consistency.** When the read databases and write databases are separated, the read data might not show the most recent changes immediately. This delay results in stale data. Ensuring that the read model store stays up-to-date with changes in the write model store can be challenging. Also, detecting and handling scenarios where a user acts on stale data requires careful consideration.

When to use this pattern

Use this pattern when:

- **You work in collaborative environments.** In environments where multiple users access and modify the same data simultaneously, CQRS helps reduce merge conflicts. Commands can include enough granularity to prevent conflicts, and the system can resolve any conflicts that occur within the command logic.

- **You have task-based user interfaces.** Applications that guide users through complex processes as a series of steps or with complex domain models benefit from CQRS.

  - The write model has a full command-processing stack with business logic, input validation, and business validation. The write model might treat a set of associated objects as a single unit for data changes, which is known as an *aggregate* in domain-driven design terminology. The write model might also help ensure that these objects are always in a consistent state.

  - The read model has no business logic or validation stack. It returns a DTO for use in a view model. The read model is eventually consistent with the write model.

- **You need performance tuning.** Systems where the performance of data reads must be fine-tuned separately from performance of data writes benefit from CQRS. This pattern is especially beneficial when the number of reads is greater than the number of writes. The read model scales horizontally to handle large query volumes. The write model runs on fewer instances to minimize merge conflicts and maintain consistency.

- **You have separation of development concerns.** CQRS allows teams to work independently. One team implements the complex business logic in the write model, and another team develops the read model and user interface components.

- **You have evolving systems.** CQRS supports systems that evolve over time. It accommodates new model versions, frequent changes to business rules, or other modifications without affecting existing functionality.

- **You need system integration:** Systems that integrate with other subsystems, especially systems that use the Event Sourcing pattern, remain available even if a subsystem temporarily fails. CQRS isolates failures, which prevents a single component from affecting the entire system.

This pattern might not be suitable when:

- The domain or the business rules are simple.

- A simple CRUD-style user interface and data access operations are sufficient.

Workload design

Evaluate how to use the CQRS pattern in a workload's design to address the goals and principles covered in the Azure Well-Architected Framework pillars. The following table provides guidance about how this pattern supports the goals of the Performance Efficiency pillar.

Expand table

Pillar	How this pattern supports pillar goals
Performance Efficiency helps your workload efficiently meet demands through optimizations in scaling, data, and code.	The separation of read operations and write operations in high read-to-write workloads enables targeted performance and scaling optimizations for each operation's specific purpose. - PE:05 Scaling and partitioning - PE:08 Data performance

Consider any trade-offs against the goals of the other pillars that this pattern might introduce.

Combine the Event Sourcing and CQRS patterns

Some implementations of CQRS incorporate the Event Sourcing pattern. This pattern stores the system's state as a chronological series of events. Each event captures the changes made to the data at a specific time. To determine the current state, the system replays these events in order. In this setup:

- The event store is the *write model* and the single source of truth.
- The *read model* generates materialized views from these events, typically in a highly denormalized form. These views optimize data retrieval by tailoring structures to query and display requirements.

Benefits of combining the Event Sourcing and CQRS patterns

The same events that update the write model can serve as inputs to the read model. The read model can then build a real-time snapshot of the current state. These snapshots optimize queries by providing efficient and precomputed views of the data.

Instead of directly storing the current state, the system uses a stream of events as the write store. This approach reduces update conflicts on aggregates and enhances performance and scalability. The system can process these events asynchronously to build or update materialized views for the read data store.

Because the event store acts as the single source of truth, you can easily regenerate materialized views or adapt to changes in the read model by replaying historical events. Basically, materialized views function as a durable, read-only cache that's optimized for fast and efficient queries.

Considerations for how to combine the Event Sourcing and CQRS patterns

Before you combine the CQRS pattern with the Event Sourcing pattern, evaluate the following considerations:

- **Eventual consistency:** Because the write and read data stores are separate, updates to the read data store might lag behind event generation. This delay results in eventual consistency.
- **Increased complexity:** Combining the CQRS pattern with the Event Sourcing pattern requires a different design approach, which can make a successful implementation more challenging. You must write code to generate, process, and handle events, and assemble or update views for the read model. However, the Event Sourcing pattern

simplifies domain modeling and allows you to rebuild or create new views easily by preserving the history and intent of all data changes.

- **Performance of view generation:** Generating materialized views for the read model can consume significant time and resources. The same applies to projecting data by replaying and processing events for specific entities or collections. Complexity increases when calculations involve analyzing or summing values over long periods because all related events must be examined. Implement snapshots of the data at regular intervals. For example, store the current state of an entity or periodic snapshots of aggregated totals, which is the number of times a specific action occurs. Snapshots reduce the need to process the full event history repeatedly, which improves performance.

Example

The following code shows extracts from an example of a CQRS implementation that uses different definitions for the read models and the write models. The model interfaces don't dictate features of the underlying data stores, and they can evolve and be fine-tuned independently because these interfaces are separate.

The following code shows the read model definition.

C#

```csharp
// Query interface
namespace ReadModel
{
  public interface ProductsDao
  {
    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; }
  }
}
```

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

C#

```
public interface ICommand
{
  Guid Id { get; }
}

public class RateProduct : ICommand
{
  public RateProduct()
  {
    this.Id = Guid.NewGuid();
  }
  public Guid Id { get; set; }
  public int ProductId { get; set; }
  public int Rating { get; set; }
  public int UserId {get; set; }
}
```

The system uses the `ProductsCommandHandler` class to handle commands that the application sends. 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.

C#

```
public class ProductsCommandHandler :
    ICommandHandler<AddNewProduct>,
    ICommandHandler<RateProduct>,
    ICommandHandler<AddToInventory>,
    ICommandHandler<ConfirmItemShipped>,
    ICommandHandler<UpdateStockFromInventoryRecount>
{
  private readonly IRepository<Product> repository;

  public ProductsCommandHandler (IRepository<Product> repository)
  {
    this.repository = repository;
  }

  void Handle (AddNewProduct command)
  {
    ...
  }

  void Handle (RateProduct command)
  {
    var product = repository.Find(command.ProductId);
    if (product != null)
    {
      product.RateProduct(command.UserId, command.Rating);
      repository.Save(product);
    }
  }

  void Handle (AddToInventory command)
  {
    ...
  }

  void Handle (ConfirmItemsShipped command)
  {
    ...
  }

  void Handle (UpdateStockFromInventoryRecount command)
  {
    ...
  }
}
```

Next step

The following information might be relevant when you implement this pattern:

- Data partitioning guidance describes best practices for how to divide data into partitions that you can manage and access separately to improve scalability, reduce contention, and optimize performance.

Related resources

- Event Sourcing pattern. This pattern describes how to simplify tasks in complex domains and improve performance, scalability, and responsiveness. It also explains how to provide consistency for transactional data while maintaining full audit trails and history that can enable compensating actions.

- Materialized View pattern. This pattern creates prepopulated views, known as *materialized views*, for efficient querying and data extraction from one or more data stores. The read model of a CQRS implementation can contain materialized views of the write model data, or the read model can be used to generate materialized views.

Feedback

Was this page helpful? Yes No