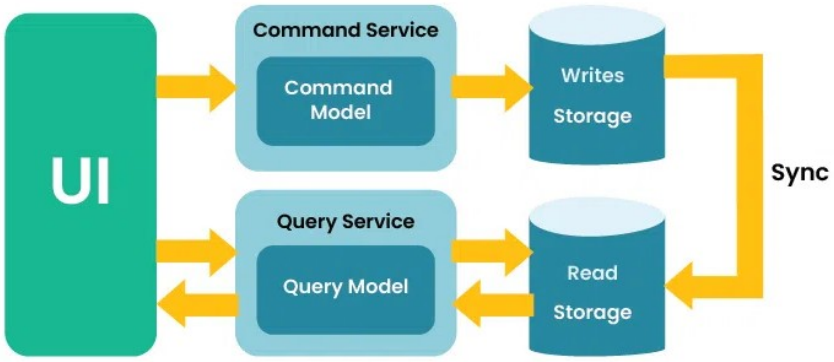
CQRS Architecture and Its Comparison with MVVM Architecture

YUMNA TASNEEM & RAIAN RASHID

September 2024

The *Command Query Responsibility Segregation* (CQRS) pattern is a software architectural approach that separates read and write operations for a data store. The key idea is to divide the responsibilities of handling commands (write operations) and queries (read operations) into distinct models. This separation allows for tailored optimizations for each model, enabling better performance and scalability. By employing CQRS, developers can also implement different technologies or data storage solutions for reading and writing, which can further enhance the application's efficiency and adaptability. Additionally, this pattern supports more straightforward implementation of complex business logic, resulting in a clearer separation of concerns and improved maintainability.



Here's a breakdown of CQRS pattern:

1 Concept:

Commands: These represent actions that change the state of the system, such as Create, Update, or Delete operations. For instance, when a user updates their profile, that action is processed as a command. Commands often contain the necessary data to perform the action and may include validation logic to ensure that the operation adheres to business rules.

Queries: These are used to fetch data from the system without modifying its state, essentially performing Read operations. For example, displaying user profile information is handled by a query. Queries can be optimized for performance, allowing for quick retrieval of specific data formats or aggregates, and they may leverage specialized read models to enhance efficiency.

2 Separation of Models:

In CQRS, the write model (used for commands) and the read model (used for queries) are distinct, each optimized for its specific task. The write model focuses on maintaining consistency and enforcing business rules, ensuring that all changes to the system's state are valid and adhere to the defined logic. This model typically handles complex validation and may incorporate mechanisms for transactions.

Conversely, the read model is often optimized for fast data retrieval, designed to serve the specific needs of various queries efficiently. It can be denormalized and structured to minimize the overhead associated with data fetching, allowing for quicker access to the information users need. This separation not only enhances performance but also allows for independent evolution of each model, accommodating changing requirements over time.

3 How CQRS Works:

Write Side (Commands): When a user issues a command (e.g., creating an order), it is handled by the command model. This model is responsible for updating the database or the state of the system, often involving complex business logic and validation processes. The command model ensures that all changes adhere to the business rules and maintain data integrity. Once the command is processed, it may also generate events to reflect the changes made, which can be used for further processing or updating the read model.

Read Side (Queries): Queries, in contrast, access a separate model that may utilize denormalized data for improved speed or even a completely different storage system based on specific requirements. This separation allows queries to retrieve information quickly and efficiently, without altering any data. The read model can be tailored to the needs of various use cases.

4 Benefits:

- **Separation of Concerns:** Allows for independent optimization of reads and writes, making the system more scalable and easier to maintain.
- **Scalability:** Enables independent scaling of read and write workloads, optimizing resource allocation based on usage patterns.
- **Flexibility:** Supports the use of different databases or data models for each side, allowing tailored optimizations.
- **Clearer Responsibility:** Provides a distinct separation between code that manages business rules (commands) and code that retrieves data (queries), simplifying the development process.

5 Challanges:

- **Complexity:**
  - Requires maintaining two separate models; one for commands and one for queries.
  - Introduces additional infrastructure needs and careful coordination to ensure consistency.
- **Eventual Consistency:**
  - Read and write sides are often eventually consistent, leading to potential delays in data updates.
  - Users may encounter outdated information, which can be problematic for applications requiring real-time data accuracy.
- **Event Sourcing:**
  - Often paired with CQRS, where state changes are stored as a series of events.
  - Adds complexity in terms of ensuring data integrity and handling event replay.
  - Requires management of event data growth, complicating system maintenance.

6 Use Cases:

- **High-Performance Systems:**
  - Ideal for applications with high read throughput, such as online retail platforms or news websites.

  - Allows for independent scaling of the read side, efficiently handling large volumes of queries while the write side maintains data integrity.

- **Complex Business Logic:**
  - Beneficial for applications with intricate domain logic, like financial systems or healthcare applications.
  - Separating reads and writes simplifies code structures, making it easier to implement and maintain business rules and validations.

- **Collaborative Applications:**
  - Suited for systems with multiple users editing the same data, such as project management tools or collaborative document editing platforms.
  - Manages complexities of concurrent modifications and conflict resolution by allowing commands to handle updates while queries provide users with the most current data without sacrificing performance.

7 Example:

- **Command:**
  - When a user places an order, the system's state changes by adding the order to the database.
  - This process includes:
    * Updating inventory levels.
    * Processing payments.
    * Validating order details to ensure compliance with business rules.

- **Query:**
  - When the user views their order history, the system retrieves data from a separate read model (e.g., a view-optimized version of the database).
  - This query displays the order history without altering any underlying data, ensuring users receive accurate and up-to-date information efficiently

CQRS in Android Project Development:

Implementing CQRS (Command Query Responsibility Segregation) in an Android project can enhance the structure, scalability, and maintainability of the application.
Here's how to approach it:

Key Concepts:

- Seperation of Concerns:
  - *Commands:* Handle state changes (create, update, delete).
  - *Queries:* Retrieve data without side effects.

Benefits:

- Scalability: Independent scaling for reads and writes.
- Maintainability: Clearer code structure for easier modifications.
- Performance: Optimized read models for faster data retrieval.
- Collaboration: Teams can work on commands and queries separately.

Implementation Steps:

- Define the Domain Model: Identify entities and core business logic.
- Create Command Handlers: Implement command classes and use background threads (Kotlin Coroutines or RxJava).
- Set Up Query Handlers: Create optimized query classes, using Room for data access.
- Integrate with UI: Use ViewModel and LiveData or StateFlow for data observation and UI updates.
- Testing: Unit test command and query handlers separately.

Tools and Libraries:

- Room: For local database management.
- Kotlin Coroutines: For asynchronous programming.
- ViewModel: To manage UI-related data.
- LiveData/StateFlow: For reactive data observation.

Example Use Case: Shopping Application

- Commands: Place orders, update inventory.
- Queries: Fetch order history, product listings.

Advantage and Disadvantages of CQRS:

Advantages of CQRS	Disadvantages of CQRS
1. **Scalability:** Allows independent scaling of read and write sides.	1. **Complexity:** Introduces added complexity with separate models for commands and queries.
2. **Performance Optimization:** Read models can be optimized for specific queries.	2. **Eventual Consistency:** Often relies on eventual consistency.
3. **Improved Maintainability:** Clearly separates business logic for commands and queries.	3. **Increased Development Effort:** Requires more initial setup and ongoing management.
4. **Enhanced Collaboration:** Different teams can work on command and query models independently.	4. **Testing Complexity:** Testing can become more complex due to the need to ensure consistency.
5. **Supports Complex Domain Logic:** Well-suited for applications with intricate business rules.	5. **Overhead:** Can introduce additional infrastructure overhead.

Comparing MVVM and CQRS in Android Development:

Aspect	MVVM	CQRS
Purpose	Separates UI from business logic	Segregates read and write operations
Components	Model, View, ViewModel	Commands, Queries, (often) Event Sourcing
Concerns	Focuses on UI and logic separation	Distinguishes between read and write models
Use Cases	Best for straightforward data flow	Ideal for high throughput and complex logic
Performance	Optimized for typical CRUD operations	Optimized for independent scaling of reads/writes
Testing	Easier unit testing of UI logic	More extensive testing required for consistency
Complexity	Generally simpler to implement	Introduces additional complexity

Why should we use MVVM instead of CQRS for an Android Project:

- **Simplicity:** MVVM is generally simpler to implement and understand, making it ideal for typical Android applications. The clear structure of Model, View, and ViewModel helps new developers grasp the architecture quickly.
- **Faster Development:** With MVVM, you can focus on straightforward data binding and UI updates without the additional complexity of separating commands and queries. This leads to quicker prototyping and faster time to market.
- **Better for Small to Medium Applications:** For projects with less complex business logic and data interactions, MVVM is often sufficient. CQRS may introduce unnecessary complexity for simpler use cases.
- **Ease of Testing:** MVVM facilitates unit testing of the ViewModel without relying on UI components. The separation of concerns makes it easier to write isolated tests for business logic.
- **Community Support and Resources:** MVVM has strong community support, with extensive resources, libraries, and best practices readily available for Android development. This can ease the learning curve and accelerate implementation.
- **Integrates Well with Android Architecture Components:** MVVM is designed to work seamlessly with Android Architecture Components like LiveData, Room, and Navigation, providing a cohesive development experience.
- **Flexibility:** MVVM allows for gradual scaling. You can start with a simple implementation and gradually introduce more complex features as needed, without having to completely overhaul the architecture.

Conclusion:

Using MVVM for Android projects offers a straightforward, efficient architecture that is easy to implement and understand, making it ideal for developers, especially in small to medium applications. Its focus on data binding, lifecycle awareness, and unit testing enhances responsiveness and maintainability. With strong community support and seamless integration with Android Architecture Components, MVVM allows for quick prototyping and gradual scaling.