**CQRS Command and Query Responsibility Segregation pattern**

CQRS stands for Command and Query Responsibility Segregation, a pattern that separates read and update operations for a data store.

In case of complex application Read and write workloads are often asymmetrical, with very different performance and scale requirements.

**Problem Statement**

1. There is often 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 aren't required as part of an operation.
2. Data contention can occur when operations are performed in parallel on the same set of data.
3. 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.
4. Managing security and permissions can become complex, because each entity is subject to both read and write operations, which might expose data in the wrong context.

**Solution**

CQRS separates reads and writes into different models, using **commands** to update data, and **queries** to read data.

1. Commands should be task-based, rather than data centric. ("Book hotel room", not "set ReservationStatus to Reserved").
2. Commands may be placed on a queue for asynchronous processing, rather than being processed synchronously.
3. Queries never modify the database. A query returns a DTO that does not encapsulate any domain knowledge.

Text

Description automatically generated with medium confidence

**Various ways to achieve**

1. You can maintain various materialized views in read databases.
2. You can fire events to update the read database after the successful update in write database.
3. You can take help of database syncing operation of the underlying database.
4. You can use Spring feature @ConditionalOnProperty for certain Command controller and for QueryController.

## **When to use CQRS pattern**

1. Collaborative domains **where many users access the same data in parallel**. CQRS allows you to define commands with enough granularity to minimize merge conflicts at the domain level, and conflicts that do arise can be merged by the command.
2. **The write model has a full command-processing stack with business logic, input validation, and business validation**. **The write model may treat a set of associated objects as a single unit for data changes** (an aggregate, in DDD terminology) and ensure that these objects are always in a consistent state. **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.
3. **Scenarios where performance of data reads must be fine-tuned separately from performance of data writes**, **especially when the number of reads is much greater than the number of writes**
4. 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.
5. **Scenarios where the system is expected to evolve over time and might contain multiple versions of the model,** or where business rules change regularly.
6. 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.

**What is 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.

Benefits of CQRS:

1. **Independent scaling**. CQRS allows the read and write workloads to scale independently, and may result in fewer lock contentions.
2. **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.
3. **Security**. It's easier to ensure that only the right domain entities are performing writes on the data.
4. **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.
5. **Simpler queries**. By storing a materialized view in the read database, the application can avoid complex joins when querying.

## **Implementation issues/Challenges and considerations**

* **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.
* **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 using CQRS combined with the Event Sourcing pattern, consider the following:**

* As with any system where the write and read stores are separate, systems based on this pattern are only eventually consistent. There will be some delay between the event being generated and the data store being updated.
* The pattern adds complexity because code must be created to initiate and handle events, and assemble or update the appropriate views or objects required by queries or a read model. The complexity of the CQRS pattern when used with the Event Sourcing pattern can make a successful implementation more difficult, and requires a different approach to designing systems.
* Generating materialized views for use in the read model or projections of the data by replaying and handling the events for specific entities or collections of entities can require significant processing time and resource usage. This is especially true if it requires summation or analysis of values over long periods, because all the associated events might need to be examined. Resolve this by implementing snapshots of the data at scheduled intervals, such as a total count of the number of a specific action that has occurred, or the current state of an entity.

**What is the difference between event sourcing and event-driven?**

**Event Sourcing**

* Event Sourcing a persistence strategy and doesn't intrinsically say anything about communication.
* Event Sourcing is keeping a log of events so you don't forget.
* Event Sourcing is when the components themselves treat the history of events which they have persisted as the ultimate source of truth for their state (or at least the state that they actually care about).

**Event Driven**

* Event Driven Architecture is about communicating what happened to others.
* Event Driven Architecture is about components communicating via publishing events rather than making calls against. It's a communication strategy .

Event Sourcing doesn't intrinsically have anything to do with CQRS (though event sourced systems do often find CQRS useful as a means of allowing queries which would have to replay a non-trivial subset of events to use a data model more optimized for that use case; likewise, there are some advantages to using an event sourcing persistence model for processing commands in a CQRS).

**Event Sourcing**

* Event sourcing is **an architectural approach** for (typically but not necessarily always) **synchronously receiving and subsequently asynchronously distributing data** (events) within an architecture. **Events, once received, are persisted in a data store unique to the receiving service**.
* The event source is the name of the software that logs the event.

The persisted events being the ultimate source of truth implies that an event replay recovers the state (by definition, if an event replay could not recover the state, then the events were not the ultimate source of truth).

**How it Works**

The fundamental idea of Event Sourcing is that of ensuring every change to the state of an application is captured in an event object, and that these event objects are themselves stored in the sequence they were applied for the same lifetime as the application state itself.

Event Sourcing pattern

Instead of storing just the current state of the data in a domain, use an append-only store to record the full series of actions taken on that data. The store acts as the system of record and can be used to materialize the domain objects.

**Problem Statement**

Most applications work with data, and the typical approach is for the application to maintain the current state of the data by updating it as users work with it. For example, in the traditional create, read, update, and delete (CRUD) model a typical data process is to read data from the store, make some modifications to it, and update the current state of the data with the new values—often by using transactions that lock the data.

**The CRUD approach has some limitations**:

1. **CRUD systems perform update operations directly against a data store**, which can slow down performance and responsiveness, and limit scalability, due to the processing overhead it requires.
2. **In a collaborative domain with many concurrent users, data update conflicts are more likely** because the update operations take place on a single item of data.
3. Unless there's an additional auditing mechanism that records the details of each operation in a separate log, **history is lost**.

**Solution**

* The Event Sourcing pattern defines an approach to handling operations on data that's driven by a sequence of events, **each event is recorded in an append-only store**. **Application code sends a series of events that imperatively describe each action that has occurred on the data to the event store**, where they're persisted. **Each event represents a set of changes to the data** (such as AddedItemToOrder).
* **The events are persisted in an event store that acts as the system of record** (the authoritative data source) about the current state of the data. **The event store typically publishes these events so that consumers can be notified and can handle them if needed**. Consumers could, for example, initiate tasks that apply the operations in the events to other systems, or perform any other associated action that's required to complete the operation. Notice that the application code that generates the events is decoupled from the systems that subscribe to the events.

Typical uses of the events published by the event store are to maintain materialized views of entities as actions in the application change them, and for integration with external systems.  For example, a system can maintain a materialized view of all customer orders that's used to populate parts of the UI. As the application adds new orders, adds or removes items on the order, and adds shipping information, the events that describe these changes can be handled and used to update the [materialized view](https://docs.microsoft.com/en-us/azure/architecture/patterns/materialized-view).

**The Event Sourcing pattern provides the following advantages:**

1. Events are immutable and can be stored using an append-only operation. The user interface, workflow, or process that initiated an event can continue, and tasks that handle the events can run in the background. This, combined with the fact that there's no contention during the processing of transactions, can vastly improve performance and scalability for applications, especially for the presentation level or user interface.
2. Events are simple objects that describe some action that occurred, together with any associated data required to describe the action represented by the event. **Events don't directly update a data store**. They're simply recorded for handling at the appropriate time. This can simplify implementation and management.
3. **Events typically have meaning for a domain expert**, whereas object-relational impedance mismatch can make complex database tables hard to understand. Tables are artificial constructs that represent the current state of the system, not the events that occurred.
4. **Event sourcing can help prevent concurrent updates from causing conflicts** because it avoids the requirement to directly update objects in the data store. However, the domain model must still be designed to protect itself from requests that might result in an inconsistent state.
5. **The append-only storage of events provides an audit trail that can be used to monitor actions taken against a data store**, **regenerate the current state as materialized views or projections** by replaying the events at any time, and assist in testing and debugging the system. In addition, the requirement to use compensating events to cancel changes provides a history of changes that were reversed, which wouldn't be the case if the model simply stored the current state. **The list of events can also be used to analyze application performance and detect user behavior trends, or to obtain other useful business information**.
6. The event store raises events, and tasks perform operations in response to those events. This decoupling of the tasks from the events provides flexibility and extensibility. Tasks know about the type of event and the event data, but not about the operation that triggered the event. In addition, multiple tasks can handle each event. **This enables easy integration with other services and systems that only listen for new events raised by the event store**. However, the event sourcing events tend to be very low level, and it might be necessary to generate specific integration events instead.

Event sourcing is commonly combined with the CQRS pattern by performing the data management tasks in response to the events, and by materializing views from the stored events.

**Issues and considerations/Complexity**

* The system will only be **eventually consistent when creating materialized views** or generating projections of data by replaying events. **There's some delay between an application adding events to the event store as the result of handling a request, the events being published, and consumers of the events handling them**. **During this period, new events that describe further changes to entities might have arrived at the event store**.
* The **event store is the permanent source of information, and so the event data should never be updated. The only way to update an entity to undo a change is to add a compensating event to the event store**. **If the format (rather than the data) of the persisted events needs to change, perhaps during a migration, it can be difficult to combine existing events in the store with the new version**. It might be necessary to iterate through all the events making changes so they're compliant with the new format, or add new events that use the new format. Consider using a version stamp on each version of the event schema to maintain both the old and the new event formats.
* **Multi-threaded applications and multiple instances of applications might be storing events in the event store. The consistency of events in the event store is vital, as is the order of events that affect a specific entity (the order that changes occur to an entity affects its current state). Adding a timestamp to every event can help to avoid issues**. **Another common practice is to annotate each event resulting from a request with an incremental identifier**. If two actions attempt to add events for the same entity at the same time, the event store can reject an event that matches an existing entity identifier and event identifier. There's no standard approach, or existing mechanisms such as SQL queries, for reading the events to obtain information. The only data that can be extracted is a stream of events using an event identifier as the criteria. The event ID typically maps to individual entities. The current state of an entity can be determined only by replaying all of the events that relate to it against the original state of that entity.
* **The length of each event stream affects managing and updating the system. If the streams are large**, consider creating snapshots at specific intervals such as a specified number of events. The current state of the entity can be obtained from the snapshot and by replaying any events that occurred after that point in time.
* Even though event sourcing minimizes the chance of conflicting updates to the data, **the application must still be able to deal with inconsistencies that result from eventual consistency** and the lack of transactions. For example, an event that indicates a reduction in stock inventory might arrive in the data store while an order for that item is being placed, resulting in a requirement to reconcile the two operations either by advising the customer or creating a back order.
* Event publication might be at least once, and so consumers of the events must be idempotent. They must not reapply the update described in an event if the event is handled more than once

**When to use this pattern**

1. When you want to capture intent, purpose, or reason in the data. For example, changes to a customer entity can be captured as a series of specific event types, such as Moved home, Closed account, or Deceased.
2. When it's vital to minimize or completely avoid the occurrence of conflicting updates to data.
3. For example, when a task involves multiple steps you might need to execute actions to revert updates and then replay some steps to bring the data back into a consistent state.
4. When using events is a natural feature of the operation of the application, and requires little additional development or implementation effort.
5. When you need to decouple the process of inputting or updating data from the tasks required to apply these actions. This might be to improve UI performance, or to distribute events to other listeners that take action when the events occur. For example, integrating a payroll system with an expense submission website so that events raised by the event store in response to data updates made in the website are consumed by both the website and the payroll system.
6. When you want flexibility to be able to change the format of materialized models and entity data if requirements change, or—when used in conjunction with CQRS—you need to adapt a read model or the views that expose the data.
7. When used in conjunction with CQRS, and eventual consistency is acceptable while a read model is updated, or the performance impact of rehydrating entities and data from an event stream is acceptable.

**Story behind Event Sourcing**

Let us develop an application which has the following functionalities.

* User can add new stock
* User can remove stock after selling it
* User can find the current stock of a particular item

As you noticed the above app just stores the **current state**of a stock item in a database.

At some particular point of time, there is a requirement that what was the stock on a particular day or previous day.

We can’t do as per the above design as we do not store the sequence of entities. This situation necessitates the concept of Event Sourcing.

**How to implement CQRS actually in Spring**

A very simple and naïve way of implementation is to maintain the same codebase and internally separate into functional units of Command and Query. Completely segregate the Controllers like Command Controller and Query Controller. Single application but make two different deployments with different ports which should behave as one application working on write side and another application in the read side. Example given below.

**First Deployment (As Command)**

@RestController

@RequestMapping(path = "/command")

@ConditionalOnProperty(prefix = "command", name= "enabled",havingValue = "true")

**public** **class** CommandController {

@PostMapping("/order")

**public** ResponseEntity<String> createOrder(OrderEntity order) {

System.out.println("Order: "+order);

**return** **new** ResponseEntity<String>("Order created ...", HttpStatus.CREATED);

}

}

**application.properties**

server.port=8081

**command.enabled=true 🡸 Command, for writing only**

query.enabled=false

**Second Deployment (As Query)**

@RestController

@RequestMapping(path = "/query")

@ConditionalOnProperty(prefix = "query", name= "enabled",havingValue = "true")

**public** **class** QueryController {

@GetMapping("/order/{id}")

**public** ResponseEntity<String> createOrder(@PathVariable("id") **long** id) {

System.***out***.println("Order Id: " + id);

**return** **new** ResponseEntity<String>("Order Info ...", HttpStatus.***OK***);

}

}

**application.properties**

server.port=8082

**command.enabled=false**

**query.enabled=true** **🡸 Query, for reading only**

**Code available in github**

<https://github.com/debjava/spring-command-conditional-controllers-2022>

<https://github.com/debjava/spring-query-conditional-controllers-2022>