Command Query Responsibility Segregation (CQRS) pattern

For complex data access needs, implement the CQRS pattern to segregate the APIs for accessing data, the models for managing data, or the database itself.

A domain model encapsulates domain data and maintains the correctness of that data as it is modified. The model structures the data based on how it is stored in the database. It also facilitates the management of the data. Multiple clients can independently update the data concurrently. Different clients might not use the data in the way that the domain model structures it and might not agree about how to structure it.

A domain model can become overburdened with managing complex aggregate objects, concurrent updates, and many cross-cutting views. When a model is overburdened, how can you refactor it to separate different aspects of how the data is used?

An application accesses data both to read and to modify it. The primitive data tasks are often expressed as create, read, update, and delete. Application code often doesn't make much distinction between the tasks; individual operations might mix reading the data with changing the data as needed.

This simple approach works well when all clients of the data can use the same structure and contention is low. A single domain model can manage the data, make it accessible as domain objects, and ensure that updates maintain its consistency. However, this approach can become inadequate in the following cases:

- Different clients want different views across sets of data
- The data is too widely used
- Different clients that are updating the data unknowingly conflict

For example, in a microservices architecture, each microservice must store and manage its own data. However, a user interface might need to display data from several microservices. A query that gathers bits of data from sources can be inefficient. Consider the time and bandwidth that are consumed to access data sources, the CPU that is consumed to transform data, or the memory that is consumed by intermediate objects. As a result, the query must be repeated each time that the data is accessed.

Another example is an enterprise database of record that manages the data that is required by multiple applications. The database can become overloaded with too many clients that need too many connections to run too many threads that are completing too many transactions. Eventually, the database becomes a performance bottleneck and can even fail.

A third example is maintaining the consistency of the data while clients concurrently update the data. While each update might be consistent, they can conflict with each other. Database locking ensures that the updates don't change the same data concurrently, but it doesn't ensure that multiple independent changes result in a consistent data model.

When data usage is more complex than a single domain model can support, you need a more sophisticated approach.

The Command Query Responsibility Segregation (CQRS) pattern was introduced by Greg Young (https://www.youtube.com/watch?v=JHGkaShoyNs) and is described in Martin Fowler's work on microservices (https://martinfowler.com/bliki/CQRS.html).

Refactor your domain model

The solution is to refactor a domain model into separate operations for querying data and for updating data so that the operations can be handled independently. The CQRS pattern strictly segregates operations that read data from operations that update data. An operation can read data or can write data, but not both.

This separation can make data much more manageable in several respects. The read operations and the write operations are simpler to implement because their functions are more finely focused. The operations can be developed independently, potentially by separate teams. The operations can be optimized independently, evolve independently, and follow changing user requirements more easily. These optimized operations can scale better and run better and you can apply security more precisely.

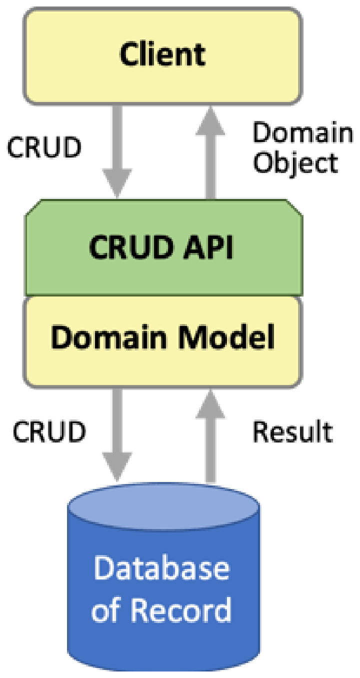
The full CQRS pattern uses separate read and write databases. In doing so, the pattern segregates the APIs for accessing data and the models for managing data. It even segregates the database itself into a read/write database that is effectively write only and one or more read-only databases.

The adoption of the pattern can be applied in phases incrementally from your code. To illustrate this pattern, these four stages can be used incrementally or a developer can go directly from stage 0 to 3 without considering the others:

1. Stage 0: Typical application data access (/cloud/architecture/architectures/event-driven-cqrs-pattern/#typicalapplicationdataaccess)
2. Stage 1: Separate read and write APIs (/cloud/architecture/architectures/event-driven-cqrs-pattern/#separatereadandwriteapis)
3. Stage 2: Separate read and write models (/cloud/architecture/architectures/event-driven-cqrs-pattern/#separatereadandwritemodels)
4. Stage 3: Separate read and write databases (/cloud/architecture/architectures/event-driven-cqrs-pattern/#separatereadandwritedatabases)

Typical application data access

Before you apply the pattern, consider the typical app design for accessing data. This diagram shows an app with a domain model for accessing data that is persisted in a database of record; that is, a single source of truth for that data. The domain model has an API that at a minimum enables clients to complete create, read, update, delete tasks on domain objects within the model.



The domain model is an object representation of the database documents or records. It comprises domain objects that represent individual documents or records and the business logic to manage and use them. Domain-Driven Design models these domain objects as entities and aggregates. *Entities* are objects that have a distinct identity that runs through time and different representations. *Aggregates* are a cluster of domain objects that can be treated as a single unit. The aggregate root maintains the integrity of the aggregate as a whole.

Ideally, the domain model's API is more domain-specific than creating, reading, updating, and deleting data. Instead, it must expose higher-level operations that represent business functions such as `findCustomer()`, `placeOrder()`, and `transferFunds()`. Those operations read and update data as needed, sometimes doing both in a single operation. They're correct if they fit the way that the business works.

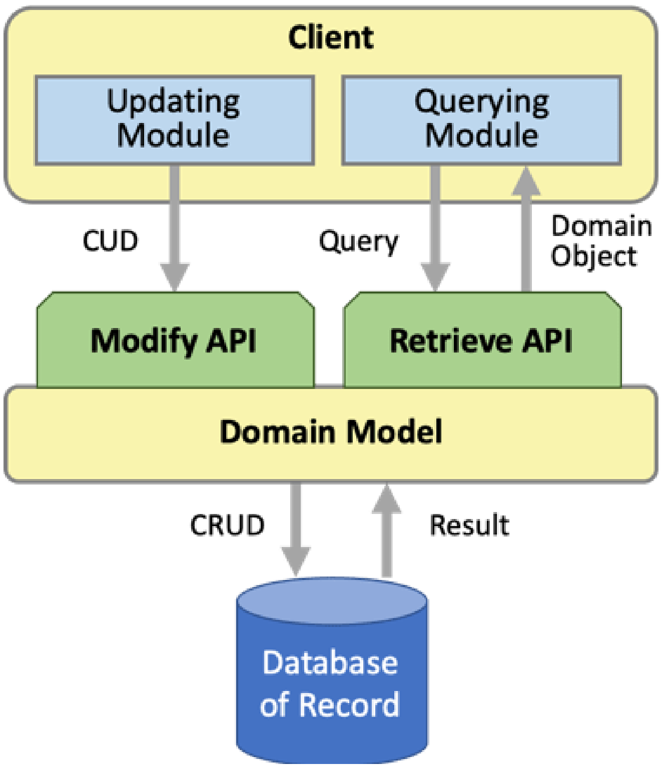
Read Apply Domain-Driven Design to microservices architecture (https://www.ibm.com/garage/method/practices/code/domain-driven-design/overview), an IBM Garage Method for Cloud practice.

Read more about domain concepts:

- Domain model (https://www.martinfowler.com/eaaCatalog/domainModel.html)
- Domain-Driven Design (https://dddcommunity.org/learning-ddd/what_is_ddd/)
- Entities (https://martinfowler.com/bliki/EvansClassification.html)
- Aggregates (https://martinfowler.com/bliki/DDD_Aggregate.html)

Separate read and write APIs

The first and most visible step in applying the CQRS pattern is splitting the CRUD API into separate read and write APIs. This diagram shows the same domain model as before, but its single CRUD API is split into retrieve and modify APIs.



The two APIs share the domain model but split the behavior:

- Read: The retrieve API is used to read the state of the objects in the domain model without changing that state. The API treats the domain state as read only.
- Write: The modify API is used to change the objects in the domain model. Changes are made by using create, update, and delete tasks: create domain objects, update the state in existing objects, and delete objects that are no longer needed. The operations in this API don't return result values. They return success, such as `ack` or `void`, or failure, such as `nak` or an exception. The create operation might return the primary of key of the entity, which can be generated either by the domain model or in the data source.

This separation of APIs is an application of the Command Query Separation (CQS) pattern, which separates methods that change state from the methods that don't. To do so, each of an object's methods can be in one of these categories, but not both:

- Query: Returns a result. Doesn't change the system's state or cause any side effects that change the state.
- Command (also known as modifiers or mutators): Changes the state of a system. Doesn't return a value, only an indication of success or failure.

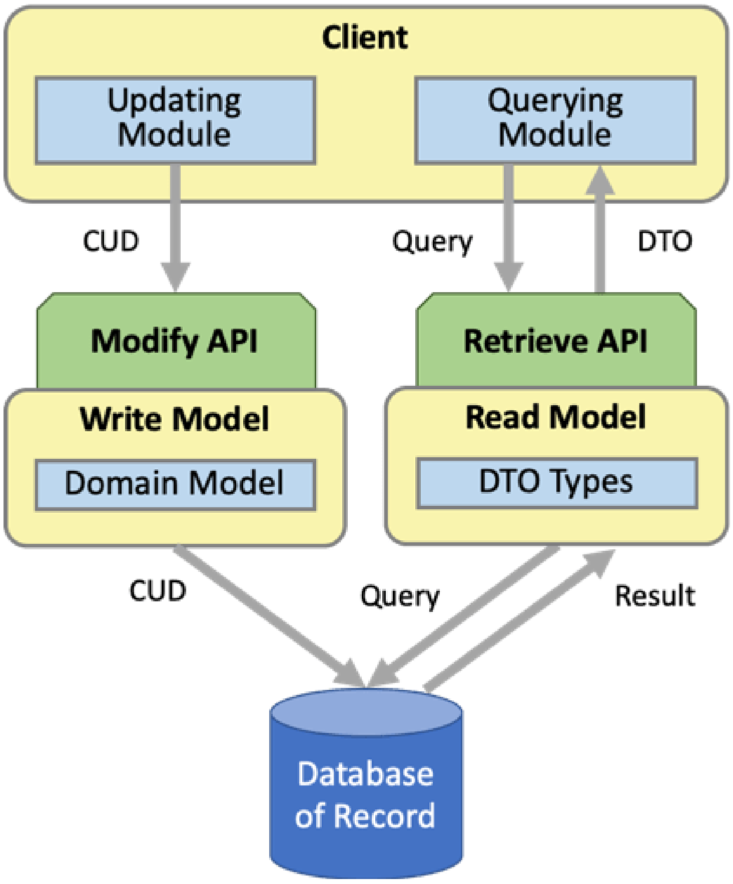
With this approach, the domain model works the same and provides access to the data the same as before. What changed is the API for the domain model. While a higher-level operation might change the state of the application and return a part of that state, now each such operation is redesigned to do only one or the other.

When the domain model splits its API into read and write operations, clients that use the API must likewise split their functions into querying and updating functions. Most new web-based applications are based in the single-page application, with components and services that use and encapsulate remote API. This separation of backend APIs fits well with modern web applications.

This stage depends on the domain model's ability to implement both the retrieve and modify APIs. A single domain model requires the retrieve and modify behavior to have similar, corresponding implementations. For them to evolve independently, the two APIs need to be implemented with separate read and write models.

Separate read and write models

The second step in applying the CQRS pattern is to split the domain model into separate read and write models. This step doesn't change only the API for accessing domain functions. It also changes the design of how that function is structured and implemented. As shown in this diagram, the domain model becomes the basis for a write model that handles changes to the domain objects and a read model that accesses the state of the app.



Naturally, the read model implements the retrieve API and the write model implements the modify API. Now the application consists not only of separate APIs for querying and updating the domain objects, but also separate business functions for doing so. Both the read business function and the write business function share a database.

The write model is implemented by specializing the domain model to focus solely on maintaining the valid structure of domain objects when they change their state and by applying any business rules.

Meanwhile, responsibility for returning domain objects is shifted to a separate read model. The read model defines data transfer objects (DTOs). Those objects are designed specifically for the model to return only the data that the client wants in a structure that is convenient for the client. The read model knows how to gather the data that is used to populate the DTOs. DTOs encapsulate little if any domain functions. They bundle data into a convenient package that can be transmitted by using a single method call, especially between processes.

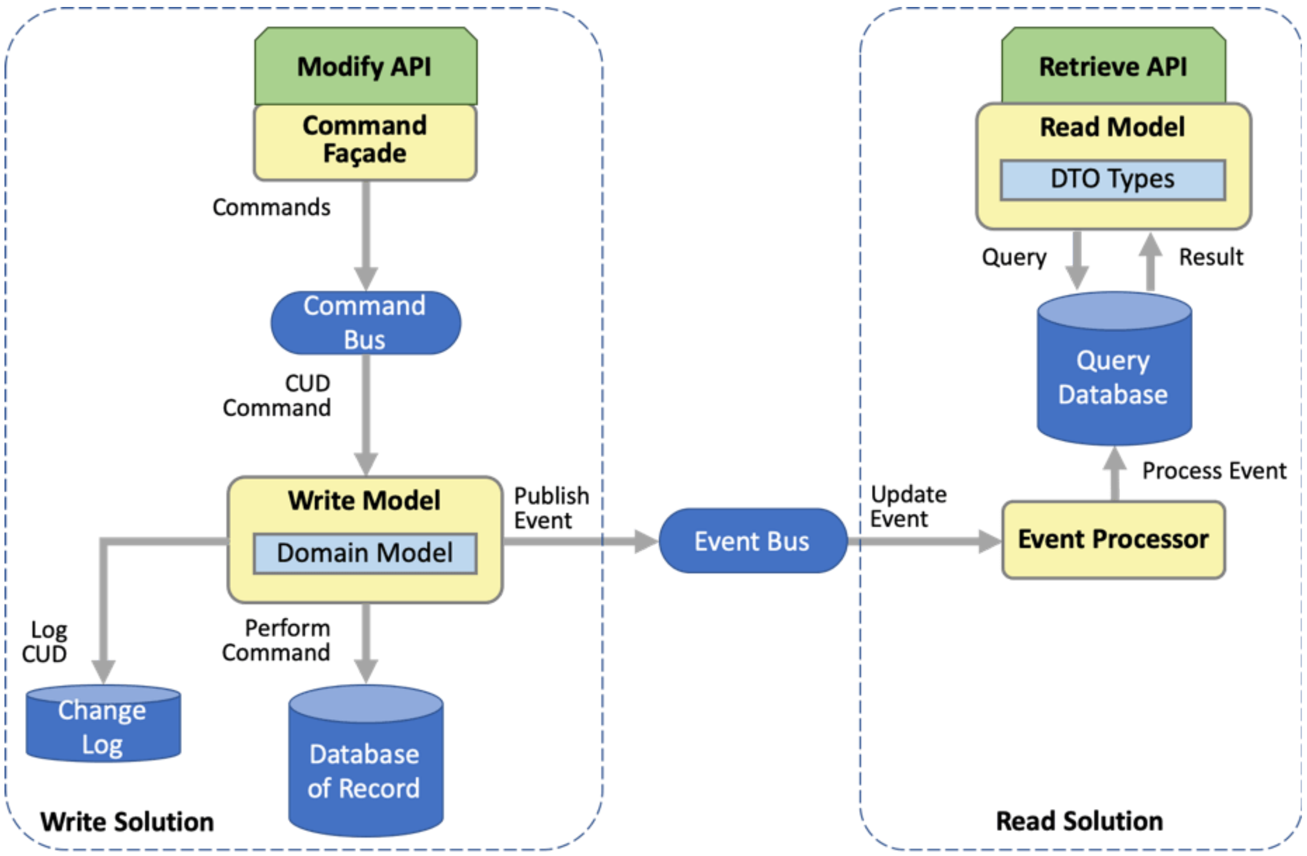
The read model can implement the retrieve API by implementing the necessary queries and running them. If the retrieve API is built to return domain objects as results, the read model can continue to do so, or better yet, it can be updated to implement DTO types that are compatible with the domain objects and return those types instead. Likewise, the modify API was implemented by using the domain model, so the write model can preserve that. The write model can enhance the implementation to more explicitly implement a command interface or use command objects.

This phase assumes that the read and write models can both be implemented by using the same database of record that the domain model uses. If that assumption is correct, the implementations of reading and writing can evolve independently and be optimized independently. This independence might become increasingly limited because they're both bound to the same database with a single schema or data model. To enable the read and write models to evolve independently, they might each need their own database.

Understand data transfer objects (https://martinfowler.com/eaaCatalog/dataTransferObject.html).

Separate read and write databases

The third step in applying the CQRS pattern, which implements the complete CQRS pattern solution, is splitting the database of record into separate read and write databases. This diagram shows the write model and read models, which are each supported by their own databases. The overall solution consists of two main parts: the write solution that supports updating the data and the read solution that supports querying the data. The two parts are connected by the event bus.



The write model has its own read/write database and the read model has its own read-only database. The read/write database still serves as the database of record—the single source of truth for the data—but is mostly used as write only. It is mostly written to and rarely read. Reading is offloaded onto a separate read database that contains the same data but is used read-only.

The query database is effectively a cache of the database of record with all of the inherit benefits and complexity of the caching pattern. The query database contains a copy of the data in the database of record with the copy structured and staged for access by clients that use the retrieve API. As a copy, extra work is needed to keep the copy synchronized with changes in the original. Latency in this synchronization process creates eventual consistency, during which the data copy is stale.

The separate databases enable the separate read and write models and their respective retrieve and modify APIs to evolve independently. Not only can the read model or write model's implementation change without changing the other, but how each stores its data can be changed independently.

This solution offers the following advantages:

- Scaling: The query load is moved from the write database to the read database. If the database of record is a scalability bottleneck and much of the load on it is caused by queries, you can unload those query responsibilities to improve the scalability of the combined data access.
- Performance: The schemas of the two databases can be different, enabling them to be designed and optimized independently for better performance. The write database can be optimized for data consistency and correctness with capabilities such as stored procedures that fit the write model and help with data updates. The read database can store the data in units that better fit the read model and are better optimized for querying, with larger rows that need fewer joins.

Notice that the design for this stage is much more complex than the design for the previous stage. Separate databases with copies of the same data can make data modeling and use easier. However, they require significant work to synchronize the data and keep the copies consistent.

CQRS employs these design features that support keeping the databases synchronized:

- Command bus for queuing commands (optional): A more subtle and optional design decision is to queue the commands that are produced by the modify API, which is shown in the diagram as the command bus. This approach can increase the throughput of multiple apps that are updating the database and serialize updates to help avoid, or at least detect, merge conflicts. With the bus, a client that makes an update doesn't block synchronously while the change is written to the database. Rather, the request to change the database is captured as a command and put on a message queue. Then, the client can continue with other work. Asynchronously in the background, the write model processes the commands at the maximum sustainable rate that the database can handle without the database ever becoming overloaded. If the database becomes temporarily unavailable, the commands queue and are processed when the database becomes available again.
- Event bus for publishing update events (required): Whenever the write database is updated, a change notification is published as an event on the event bus. Interested parties can subscribe to the event bus to be notified when the database is updated. One such party is an event processor for the query database, which receives update events and processes them by updating the query database. In this way, every time that the write database is updated, a corresponding update is made to the read database to keep it synchronized.

The connection between the command bus and the event bus is supported by an application of the Event sourcing pattern (/cloud/architecture/architectures/event-driven-event-sourcing-pattern), which keeps a change log that is suitable for publishing. Event sourcing maintains not only the current state of the data but also the history of how that current state was reached. For each command on the command bus, the write model completes these tasks to process the command:

- Logs the change
- Updates the database with the change
- Creates an update event that describes the change and publishes it to the event bus

The changes that are logged can be the commands from the command bus or the update events that are published to the event bus.

Learn more about command patterns (https://en.wikipedia.org/wiki/Command_pattern) and design patterns (https://www.pearson.com/us/higher-education/program/Gamma-Design-Patterns-Elements-of-Reusable-Object-Oriented-Software/PGM14333.html).

Consider the impact of using the pattern

When you apply this pattern, keep these considerations in mind.

Client impact

Applying CQRS changes how data is stored and accessed. It also changes the APIs that clients use to access data, which means that each client must be redesigned to use the new APIs.

Riskiness

Much of the complexity of the pattern solution involves duplicating the data in two databases and keeping them synchronized. Risk comes from querying data that is stale or wrong because of problems with the synchronization.

Eventual consistency

Clients that query data must expect updates to have latency. In a microservices architecture, eventual data consistency is a given and acceptable in many cases.

Command queuing

Using a command bus as part of the write solution to queue the commands is optional but powerful. In addition to the benefits of queuing, the command objects can be stored in the change log and be converted into notification events. For more information about how to use an event bus to queue commands, see Combining event sourcing and CQRS (/cloud/architecture/architectures/event-driven-cqrs-pattern/#combiningeventsourcingandcqrs).

Change log

The log of changes to the database of record can be either the list of commands from the command bus or the list of event notifications that is published on the event bus. The Event sourcing pattern (/cloud/architecture/architectures/event-driven-event-sourcing-pattern) assumes that it’s a log of events, but that pattern doesn’t include the command bus. An event list might be easier to scan as a history, while a command list is easier to replay.

Create keys

The strict interpretation of the CQS pattern says that command operations don't have return types. A possible exception is commands that create data. An operation that creates a record or document typically returns the key for accessing the new data, which is convenient for the client. However, if the create operation is started asynchronously by a command on a command bus, the write model must perform a callback on the client to return the key.

Messaging queues and topics

While messaging is used to implement both the command bus and event bus, the two buses use messaging differently. The command bus guarantees exactly one delivery. The event bus broadcasts each event to all interested event processors.

Query database persistence

The database of record is always persistent. The query database is a cache that can be a persistent cache or an in-memory cache. If the cache is in-memory and is lost, it must be rebuilt completely from the database of record.

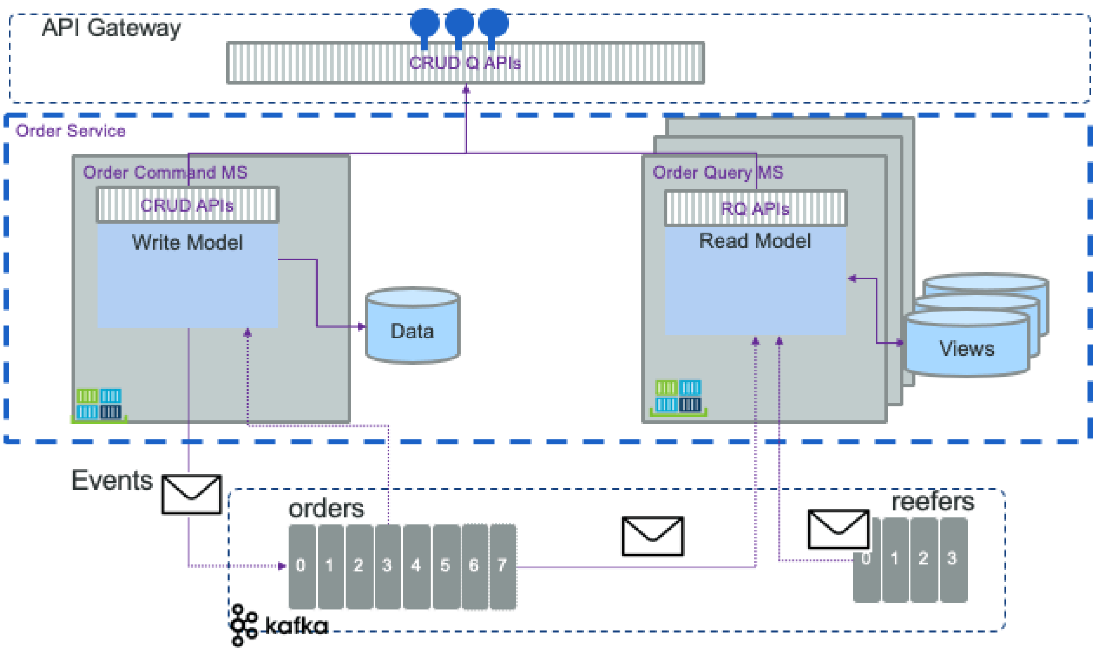
Security

You can apply controls on reading data and updating data separately by using the two parts of the solution.

Combining event sourcing and CQRS

The CQRS application pattern is frequently associated with event sourcing. When you practice event sourcing and Domain-Driven Design, you event-source the aggregates or root entities. Aggregates create events that are persisted. Beyond the simple create, update, and read by ID operations, the business requirements might be to run complex queries that can't be answered by a single aggregate. By using event sourcing to respond to a query such as "what are the orders of a customer", you must rebuild the history of all orders and filter per customer. It's a lot of computation. This issue is linked to the problem of having conflicting domain models between query and persistence.

Creations and updates are done as state notification events (change of state) and are persisted in the event log or store. The following figure presents a microservice that supports the write model and many other microservices that support the queries:



The query part is separate process that consumes change log events and builds a projection for future queries. The write part can persist in SQL while the read can use a document-oriented database with strong indexing and query capabilities. You can also use in-memory databases or a distributed cache. They don't need to be in the same language. With CQRS and event sourcing, the projections are retroactive. New query equals implementing a new projection and reading the events from the beginning of time or the recent committed state and snapshot. Read and write models are decoupled and can evolve independently. The command part can still handle simple queries that are primary-key based, such as `get order by id`, or queries that don't involve joins.

The event backbone uses a publish/subscribe model. Kafka is a good candidate as an implementation technology.

With this structure, the Read Model microservice most likely consumes events from topics to build the data projection based on joining those data streams. In the example implementation for a container shipment solution (https://ibm-cloud-architecture.github.io/refarch-kc/), you might have a query to assess if the cold-chain was respected on the fresh food order shipment. The query accesses the voyage, container metrics, and order microservices to be able to answer this question. This capability is where CQRS shines. You can separate the API definition and management in an API gateway.

Consider these implementation items:

- Consistency (ensure that the data constraints are respected for each data transaction): CQRS without event sourcing has the same consistency guarantees as the database that is used to persist data and events. With event sourcing, the consistency can be different, one for the write model and one for the read model. On the write model, strong consistency is important to ensure that the current state of the system is correct. Therefore, it uses transactions, locks, and sharding. On the read side, you need less consistency, as consumers of the query APIs mostly work on stale data. Locking data on the read operation isn't reasonable.
- Scalability: Separating read and write as two microservices allows for high availability. Caching at the read level can be used to increase performance response time, and can be deployed as multiple stand-alone instances (Pods in Kubernetes). You can also separate the query implementations between different services. Functions as service or serverless computing are good technology choices to implement complex queries.
- Availability: The write model sacrifices consistency for availability. The read model is eventually consistent, so high availability is possible. If a failure occurs, the system disables the writing of data but can read it because the models are served by different databases and services.

With CQRS, the write model can evolve over time without impacting the read model if the event model doesn't change. The read model requires more tables, but they're often simpler.

CQRS results in an increased number of objects, with commands, operations, events, and packaging in deployable components or containers. It adds potentially different type of data sources. It's more complex.

Some challenges to consider:

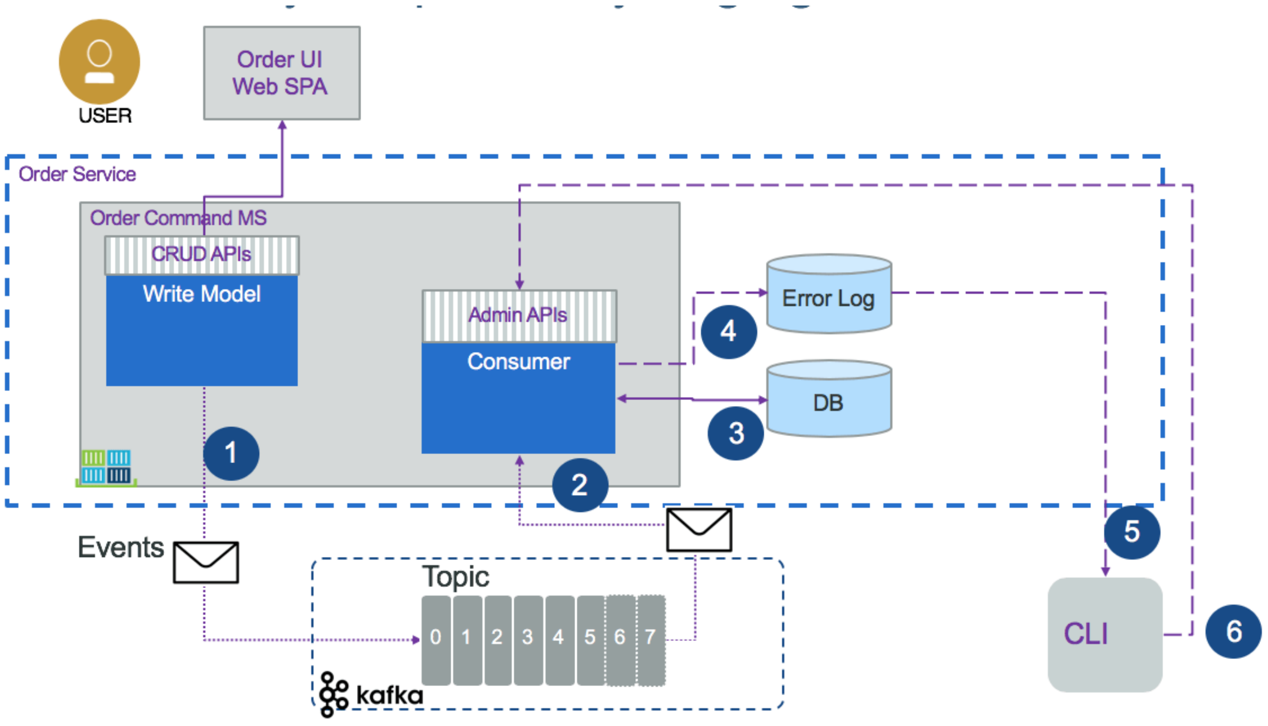
- How to support event structure version management
- How much data to keep in the event store (history)
- How to adopt data duplication, which results to eventual data consistency

As you can see in the figure, as soon as you see two arrows from the same component, you might ask yourself, "how does it work?" The write model must persist Order in its own database and then sends an OrderCreated event to the topic. Should those operations be atomic and controlled with transactions?

For more information, see Order Management - Command (https://ibm-cloud-architecture.github.io/refarch-kc/microservices/order-command/) and Order Management - Query (https://ibm-cloud-architecture.github.io/refarch-kc/microservices/order-query/).

The consistency challenge

A potential problem of data inconsistency exists. After a command saves changes into the database, the consumers don't see the new or updated data until event notification completes processing. With a traditional Java™ service that uses JPA and JMS, the save and send operations can be part of the same XA transaction and both succeed or fail. With the event sourcing pattern, the source of trust is the event source, which acts as a version control system as shown in the diagram.



Synchronizing changes to the data consists of these steps:

1. The write model creates the event and publishes it.
2. The consumer receives the event and extracts its payload.
3. The consumer updates its local data source with the payload data.
4. If the consumer fails to process the update, it can persist the event to an error log.
5. Each error in the log can be replayed.
6. A command-line interface replays an event by using an admin API, which searches in the topic by using this order ID to replay the save operation.

This implementation causes a problem for the `createOrder(order): string` operation. The Order service is supposed to return the new order with the order ID that is a unique key that was most likely created by the database. If a database update fails, no new order exists yet, so no database key exists to use as the order ID. To avoid this problem, if the underlying technology supports assigning the new order's key, the service can generate the order ID and use it as the order's key in the database.

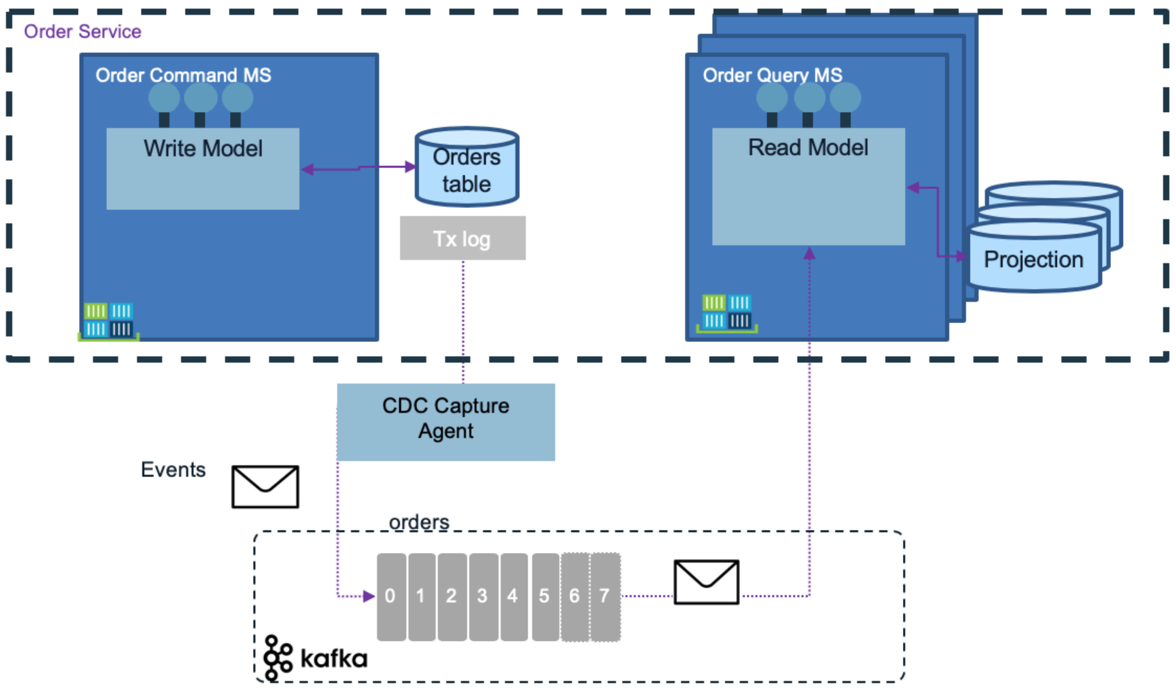
Study the Kafka consumer API and the different parameters for supporting the read offset. See these implementation practices (https://ibm-cloud-architecture.github.io/refarch-eda/technology/kafka-producers-consumers/#kafka-consumers).

CQRS and change data capture (CDC)

Other ways to support this dual operations level are as follows:

- When you use Kafka, Kafka Connect can subscribe to databases by using JDBC so that you can poll tables for updates and then produce events to Kafka.
- Use Debezium, which is an open-source change data capture solution that is based on extracting change events from database transaction logs. It helps to respond to insert, update, and delete operations on databases and to generate events. It supports databases such as MySQL, Postgres, MongoDB, and others.
- Write the order to the database and in the same transaction write to an event table ("outbox pattern"). Then, use polling to get the events to send to Kafka from this event table and delete the row in the table after the event is sent.
- Use the CDC from the database transaction log and generate events from this log. IBM® Infosphere CDC can help to implement this pattern.

The following diagram shows the CQRS implementation by using CDC:



The event must be flexible on the data payload. An event model (https://ibm-cloud-architecture.github.io/refarch-kc/implementation/consume-transform-produce/) is presented in the reference implementation.

On the view side, updates to the view part must be idempotent.

Read more about these topics:

- Kafka Connect (https://kafka.apache.org/documentation/#connect)
- Debezium (https://debezium.io/)
- Outbox event router (https://debezium.io/documentation/reference/0.10/configuration/outbox-event-router.html)

- IBM Infosphere CDC (https://www.ibm.com/support/knowledgecenter/SSTRGZ_11.4.0/com.ibm.idr.frontend.doc/pv_welcome.html)
- IBM Infosphere CDC product tour (https://www.ibm.com/cloud/garage/dte/producttour/ibm-infosphere-data-replication-product-tour)

Delay in the view

A delay exists between the data persistence and the availability of the data that is in the read model. For most business applications, the delay is acceptable. In web-based data access, most of the data is stale.

If the client that is calling the query operation needs to know whether the data is up to date, the service can define a versioning strategy. When the order data is entered in a form within a single-page application, the "create order" operation returns the order with its unique key freshly created. The single-page application will have the latest data. This example shows such an operation:

```
@POST
public Response create(OrderCreate dto) {
    Order order = new Order(UUID.randomUUID().toString(), dto.getProductID(),...);
    // ...
    return Response.ok().entity(order).build()
}
```

Schema change

What to do you do when you need to add an attribute to an event? You need to create a versioning schema for the event structure. You need to use a flexible schema, such as a JSON schema, and you might add an event adapter as a function to communicate between the different event structures.

Apache Avro (https://avro.apache.org/docs/current/) or a protocol buffer (https://developers.google.com/protocol-buffers) can also be used.

What's next

- Review the Saga pattern (/cloud/architecture/architectures/event-driven-saga-pattern).
- The Reefer Container Shipment Order Management (https://ibm-cloud-architecture.github.io/refarch-kc/) project includes two submodules. Each is deployable as a microservice to illustrate the command and query part.
- Read more about CQRS:
  - Introduction to CQRS (https://www.codeproject.com/Articles/555855/Introduction-to-CQRS)
  - CQRS (https://martinfowler.com/bliki/CQRS.html) by Martin Fowler
  - Pattern: Command Query Responsibility Segregation (CQRS) (https://microservices.io/patterns/data/cqrs.html) by Chris Richardson.

- When to use CQRS? (https://community.risingstack.com/when-to-use-cqrs)
- Concepts of CQRS (https://dzone.com/articles/concepts-of-cqrs)
- Command Query Separation (https://martinfowler.com/bliki/CommandQuerySeparation.html) by Martin Fowler