**Event sourcing & CQRS interview Q&As**

Posted on [July 4, 2018](https://www.java-success.com/event-sourcing-cqrs-interview-qas/)

Most feasible way to handle consistency across microservices is via **eventual consistency**. This model doesn’t enforce distributed ACID transactions across microservices. A better approach is to use **event sourcing**, which is an event-centric approach to business logic design and persistence. It favours to use some mechanisms of ensuring that the system would be eventually consistent at some point in the future. It reconstructs the current state of an aggregate by loading the events and replaying them. In functional programming terminology it is aggregated by a fold/reduce function over the events.

Q1. What is Event Sourcing (i.e. ES)?  
A1. In a standard table driven system you’re only storing the current state of an item (e.g. status=SENT or status=CANCELLED, etc). This approach gives you the “**current state of the world**“. However, there are times you not only just want to see **where you are**, but also you want to know **how you got there**. For example, in big data analytics you may want to know –

How many people cancelled the subscription within 7 days? or  
How many people added an item to their cart, then removed it, then bought that item a week later”?

In a standard table driven system you use the same model for both reads and writes. You design the tables write centric, and then perform various joins to cater for various read scenarios. This approach could be problematic for large applications as it is impossible to optimise the data structures for both read & write.

Event Sourcing (ES) is a pattern that focuses on **how you got there** as opposed to on the current state. It focuses on the changes that have occurred over time as Events. It is the practice of modelling your system as a sequence of events. Event Sourcing ensures that all changes to application state are stored as a sequence of events.

For example: ORDER PLACED, ORDER READY, ORDER SHIPPED, ORDER INVOICED, etc

In Event Sourcing you write the events to a persistent storage (E.g. SQL or NoSQL database), and read data for different scenarios by replaying the events. You project the data differently for different read scenarios.

Q2. Wouldn’t this be a slow process if you have many events to be applied from a blank or initial state?  
A2. Instead of replaying all the events on the fly when queried, you build **intermediate results** (aka snapshot) in a database in the background. In essence you are caching the intermediate results to be used later for a better read performance. Applying smaller subset of events to intermediary results locally can be faster than making SQL calls that may entail multiple tables and complex joins. You can have different snapshots based on the use cases.

This not only gives you complete freedom to structure & build your **read models** via intermediate results in any way you want, but also the ability to discard the models that are no longer needed.

Q3. How do you apply business rules or constraints like “order must be ready before it can be shipped”?  
A3. You just need to check if the event “ORDER READY” has happened.

Q4. What are the pros and cons of Event Sourcing?  
A4.

**Pros:**

**1)** Better performance as writes will only have **appends** (i.e. no updates), and reads can be optimised for different scenarios with different snapshots.

**2)** Since saving an event is a single operation, it is inherently atomic. The application reconstructs an entity’s current state by replaying the events.

**3)** No messy migration scripts.

**4)** Easier to communicate the process requirements as events as opposed to state changes. Event Sourcing forces you to model events as first class objects.

**5)** You can generate complex business/analytics reports. E.g. How many users removed the items from the cart? How many users added the items again within a week? etc.

**6)** Services can be integrated via events in a loosely coupled environment.

**Cons:**

**1)** Eventual Consistency where whenever an event occurs, other systems won’t know about it immediately as there will be a short time lag. This means you can’t guarantee that the data you query is immediately up-to-date.

**Note**: In NoSQL (i.e. Not only SQL) world the term consistency does not mean “C” in ACID. NoSQL consistency is know as “**eventual**” consistency. For example

DAY 1: A share market update program writes a record to the database “Avg selling price for XYZ = $35.50 on 15/Jun/2015”.  
DAY 2: The next day, share market update program updates the database so that “Avg selling price for XYZ = $32.00 on 16/Jun/2015”.

If someone queries the NoSQL database immediately after day 2, then there is a chance that a user will see that the average selling price on 16/Jun/2015 was $32.50, and NOT $32.00, which represents the latest data. However, “**eventually**” the data will, in fact, propagate such that everyone eventually sees $32.00.

**2)** High disk space usage to store events. This is not really an issue as HDFS (i.e. Hadoop Distributed File Systems), AWS S3 buckets, and NoSQL databases can handle big data volumes, and also you can schedule a clean up every few weeks.

**3)** Handling changes to events can be tricky as you need to upgrade your old events.

Q5. What is CQRS?  
A5. CQRS stands for **C**ommand **Q**uery **R**esponsibility **S**egregation. It’s a small pattern inspired by the CQS (i.e. Command Query Separation) pattern which states that “A method should either change state of an object, or return a result, but not both. This means querying should not change the state.

In a standard table driven system you have the data models that are common for both reads and writes. Having the same model for read and write side leads to a more complex model that could be very difficult to be maintained and optimised. By dividing methods into these two categories:

**Query** that does not change the state  
**Command** that changes the state

you will have a better understanding of what does, and what does not, change the state of your system. Command Query Responsibility Separation, or CQRS, takes this principle one step further by creating two objects. The read side can use a data store well-suited for queries, while the write side can use a data store well-suited for transactional updates. This can be accomplished with commands (i.e. perform an action to mutate state), events (i.e. events are how you keep the read object & write object synchronised) and queries (i.e read data).

CQRS is applicable where read & write performance need to be separately optimised and maintained. The write side is known as the domain. The domain contains all the behaviour. The read side is specialised for reporting needs.

CQRS + ES can simplify complex software systems by having rich data models, log of events for tracking, better scalability, and better visibility into concurrency issues.

1. [Language Fundamentals](https://www.java-success.com/250-core-java-interview-questions-answers/) (**LF**)
2. [Specification Fundamentals](https://www.java-success.com/java-interview-quick-prep-faqs-course/) (**SF**)
3. [Platform Fundamentals](https://www.java-success.com/java-interview-quick-prep-faqs-course/) (**PF**)
4. [Design Considerations](https://www.java-success.com/category/java-success-com/quick-java-prep/0003-16-key-areas/03-design-concepts/) (**DC**)
5. [Design Patterns](https://www.java-success.com/category/java-success-com-key-areas/010-16-key-areas/design-patterns/) (**DP**)
6. [Concurrency Management](https://www.java-success.com/category/java-engineers/0003-16-key-areas/05-concurrency/) (**CM**)
7. [Performance Considerations](https://www.java-success.com/category/java-success-com-key-areas/010-16-key-areas/performance/) (**PC**)
8. [Memory/Resource Considerations](https://www.java-success.com/category/java-success-com-key-areas/010-16-key-areas/memory-management/) (**MC**)
9. [Transaction Management](https://www.java-success.com/category/java-success-com-key-areas/010-16-key-areas/transaction-management/) (**TM**)
10. [Security](https://www.java-success.com/category/java-success-com-key-areas/010-16-key-areas/security/) (**SE**)
11. [Scalability](https://www.java-success.com/category/java-success-com-key-areas/010-16-key-areas/scalability/) (**SC**)
12. [Best Practices](https://www.java-success.com/category/java-engineers/0003-16-key-areas/04-best-practice/) (**BP**)
13. [Coding](https://www.java-success.com/category/java-success-com-800-qas/written-tests/) (**CO**)
14. [Exception Handling](https://www.java-success.com/category/java-success-com-key-areas/010-16-key-areas/exception-handling/) (**EH**)
15. [Software Development Processes](https://www.java-success.com/category/java-success-com-key-areas/010-16-key-areas/sdlc/) (**SDP**)
16. [Quality of Service](https://www.java-success.com/category/java-success-com-key-areas/010-16-key-areas/y-qos/) (**QoS**)
17. [Low Latency](https://www.java-success.com/category/java-engineers/0003-16-key-areas/09-low-latency/)
18. Debugging – [Core Java](https://www.java-success.com/category/java-success-com-key-areas/010-16-key-areas/debugging-java/) | [JEE](https://www.java-success.com/category/java-success-com-key-areas/010-16-key-areas/debugging-java/) | [Hadoop & Spark](https://www.java-success.com/category/java-success-com-key-areas/010-16-key-areas/debugging-spark/) | [Spring](https://www.java-success.com/category/java-success-com-key-areas/010-16-key-areas/debugging-spring/) | [Hibernate](https://www.java-success.com/category/java-success-com-key-areas/010-16-key-areas/debugging-hibernate/)
19. [CI/CD & DevOps](https://www.java-success.com/category/java-success-com-key-areas/010-16-key-areas/cicd-devops/)