Mastering Event Sourcing: A Game-Changing Design Pattern for Distributed Systems

InfoQ Webinar Q&A Transcript

1. Query Size for Decision-Making

Q: **You've stated that query size should not be used for decision-making. What if I need to make a decision on multiple aggregates or find an aggregate by filtering an internal field? Without the query, it will be problematic to load all the in-store memory.**

A: The typical answer is that it depends, but there are a couple of approaches. If you need to make a decision based on fully consistent data, you can create a new type of entity that subscribes to events from other types of entities. When it handles those events, it can maintain its own internal state, allowing that entity to be used for decision-making. Alternatively, you can base a decision on a view if you are comfortable with the possibility of using slightly stale data.

2. Distributed Components Writing to Event Log

Q: **In a distributed environment, how many distributed components should be allowed to write to the event log?**

A: There's a high-level answer and a more detailed one. If you're using the Akka SDK, calling `persist` ensures that the event is written to the event log, with Akka handling sequencing, timestamps, and replication across all regions where your service is running.

For example, in an application, events like `PendingMatchCreated` are persisted. If a pending match remains unjoined for more than five minutes, Akka automatically expires it. This workflow ensures proper event tracking without requiring manual intervention.

3. Rehydrating Expired Actors

Q: **When rehydrating an actor from event logs due to a service restart, is there a pattern that prevents bringing back expired actors to reduce memory usage?**

How do I control or can I control the rehydration of passivated actors? And what are the consequences of that?

A: If you're using the Akka 3 SDK, you don't interact with actors directly; you work with entities. The SDK determines when to instantiate an entity and feeds it the most current event stream. You don't have to write code that explicitly checks for restarts, as Akka handles it automatically.

4. Handling Async Side Effects

Q: **How do you handle async side effects if my system needs to trigger external actions that can fail and require compensation actions and crash recovery?**

A: One way is to publish messages to a topic. For example, in an Akka project, events from a `LobbyEntity` are published to a topic called `ChessEvents`, which is configured as a broker in the project. This allows for interactions with external systems without directly modeling side effects within Akka. Once the event is published, downstream consumers handle the necessary side effects.

5. Migrating Data Without an Event Log

Q: **How would you migrate an existing application's data without a reproducible event log into a new distributed system?**

A: If your existing system has no event log, one way to migrate data is to publish it to a Kafka topic and consume it in Akka. Another approach is to write a script that sends commands to Akka entities, which then generate events. The only way to introduce data into an event-sourced system is by issuing commands to entities.

6. Time Traveling for Financial Transactions

Q: **How can we time travel to view financial transactions from the past month to produce month-end balances?**

A: A good approach is to use views that aggregate data over time. As transactions come in, a consumer processes them and maintains a rolling summary. The view can be structured per month or queried dynamically based on the required time range.

7. Read-After-Write Consistency

Q: **If my system is event-sourced, but I require a facade that ensures read-after-write consistency, what can I do?**

A: If you need consistent data, query the entity directly. If your command is received successfully, the entity's state updates before your query. If eventual consistency is acceptable, query a view instead.

8. Event Sourcing in Different Domains

Q: **What are the primary domains where you've applied event sourcing? Are there any domains where this approach is not recommended?**

A: Event sourcing has been used in chess games, shopping carts, and banking. One interesting example involved thousands of edge devices running event-sourced systems, with selected events being sent to the cloud for processing. There isn't a domain where event sourcing can't be applied; it depends on how you model the events.