

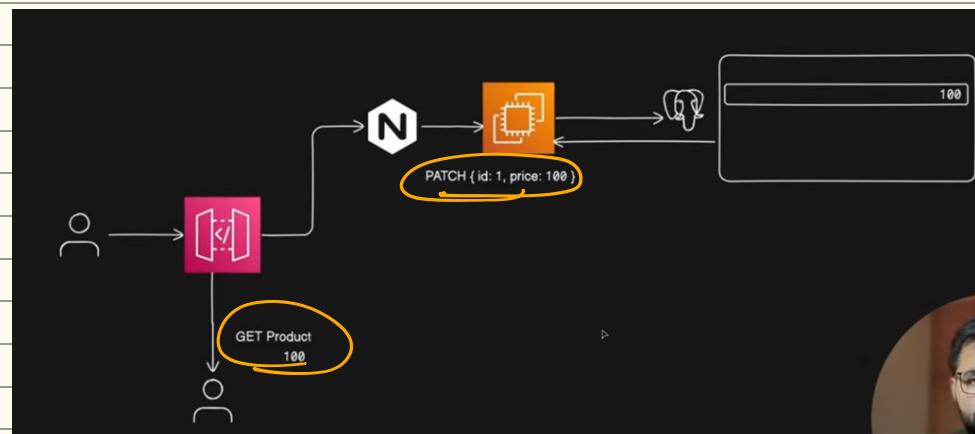
SYSTEM DESIGN

→ Event Sourcing.

→ Events :-

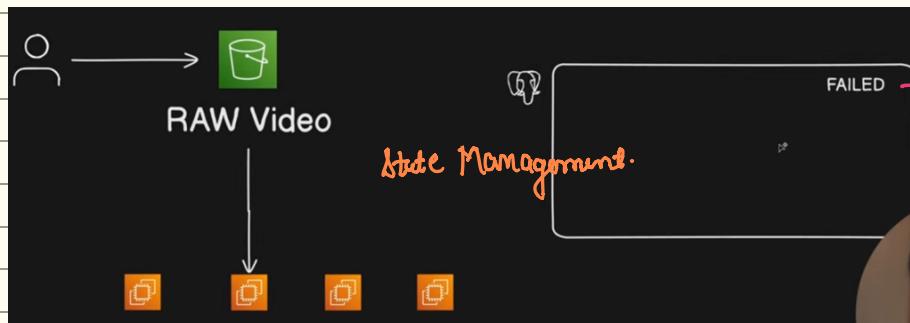
- (i). Add Item to Cart
- (ii). Checkout Item
- (iii). Product Update.

Example :-



If we update a row in a DB, but the server gets a GET request just a few ms before the PATCH updation command it becomes very difficult for state management and data updation and process.

→ Video Processing pipeline using Event sourcing.



ROUGH PSEUDO CODE:-

- i). USER uploads video
db update status "UPLOADED"
- ii). when worker picks video
db update status "PENDING"
- iii). When worker is done re-upload processed video
db update status "SUCCESS"

∴ If the query for DB fails for updating the state, because

- (i). Query was ran wrong.
- (ii). DB was busy.

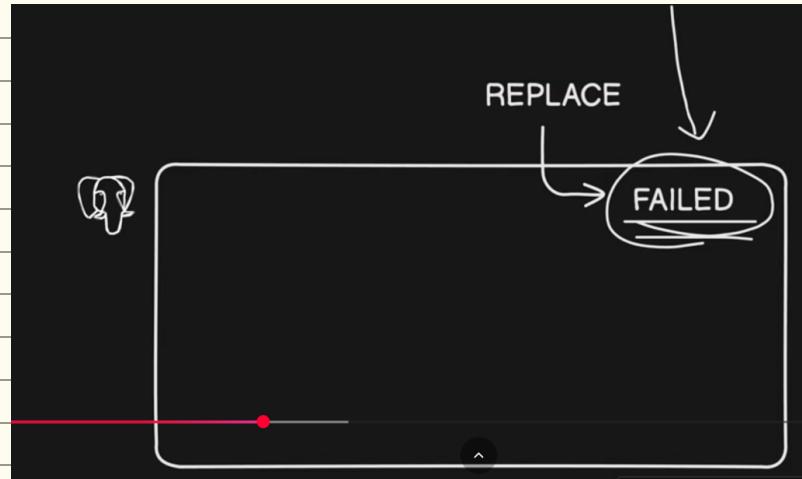
then it becomes very difficult to maintain the state of the server.

Here's where event sourcing comes into the picture.

∴ Event Sourcing states that our source of truth are events.

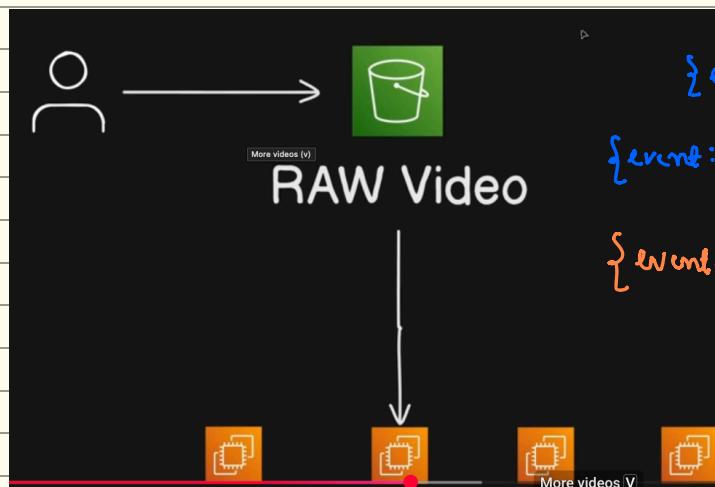
→ In event-sourcing,

- we do not update in the database, in DB we do atomic operations (we directly replace the status).



∴ Always add event log.

Event-log only works on append only logs.



{event: "Video Upload", data: {path: "1"},
+ timestamp}

{event: "Video Processing Init", data: {path: "1"},
+ timestamp}

{event: "Video Processing Success", data: {path: "1"},
+ timestamp}

→ **Hydration** :- Using events it re-constructs the state.

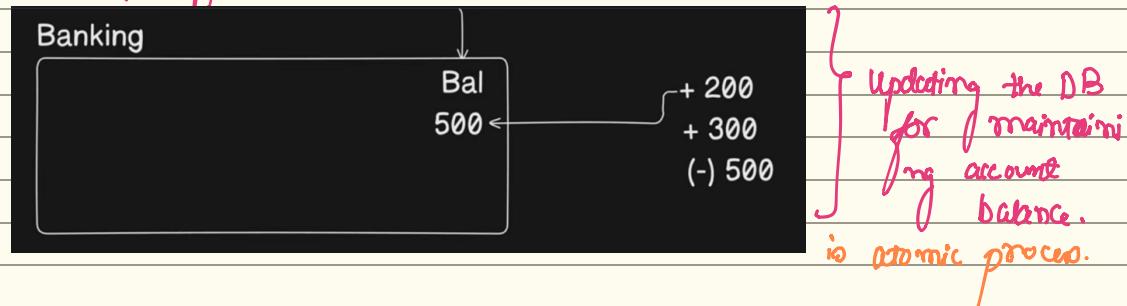
↳ Uploaded

↳ processing

↳ Success

```
{ event: "VideoUpload", data:{ path: " " }, timestamp }  
{ event: "VideoProcessingInit", data:{ path: " " }, timestamp }  
{ event: "VideoProcessingSuccess", data:{ path: " " }, timestamp }
```

→ In a banking application:-



INSTEAD maintain event logs.

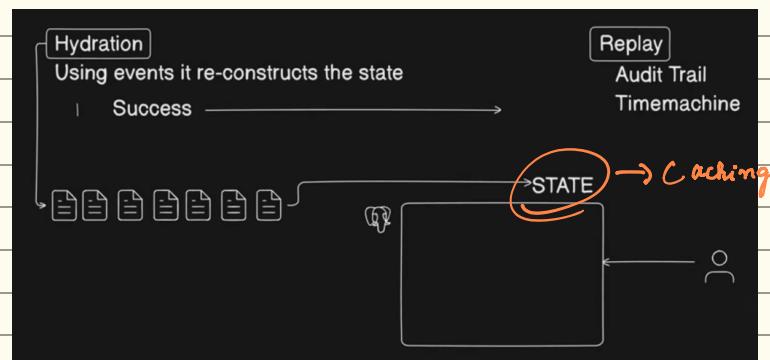


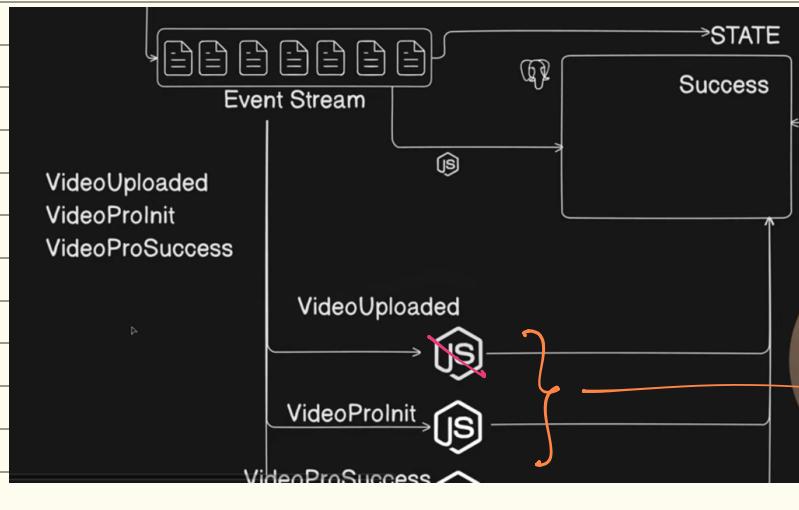
→ **Replay** :- Audit Trail., reconciliation of events if something goes wrong.

→ **Timemachine** :- Previous or plus logs of different time stamp :-

e.g. - 1 month, 1 yr, 5 yr. can be fetched using event logs.

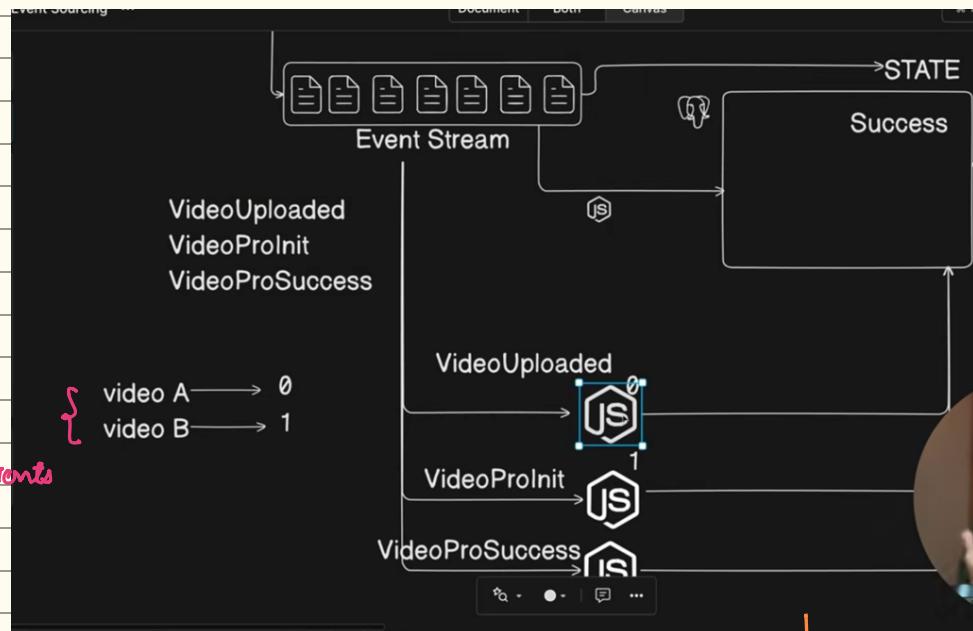
→ Hydration is a very slow and time-consuming process, hence we can cache the state.



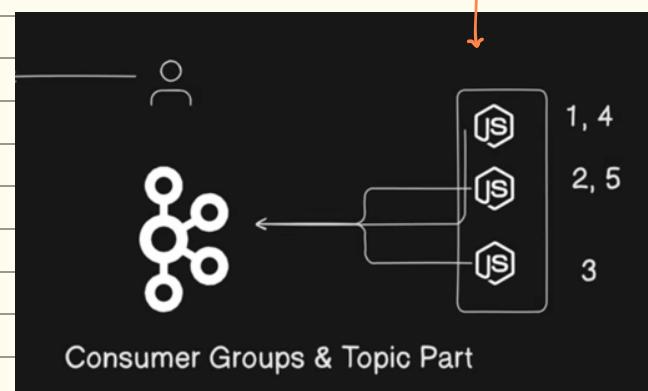


Now, if any worker is slow and consumed and other workers have took up and updated the events, but now the crashed worker has again updated the event wrongly, so in this case we use

Consumer Groups and Topic partitions.



So now, we partition the events



→ Event Sourcing :-

Event sourcing is a software design pattern where changes to an application's state are stored as a sequence of events, rather than just the current state. Instead of updating a database directly, each state-changing action is recorded as an event in an append-only log. This allows for reconstructing the application's state at any point in time by replaying these events.

Learn / Azure / Architecture Center /

Ask Learn

Event Sourcing pattern

Azure

Instead of storing just the current state of the data in a relational database, store the full series of actions taken on an object in an append-only store. The store acts as the system of record and can be used to materialize the domain objects. This approach can improve performance, scalability, and auditability in complex systems.

① Important

Event sourcing is a complex pattern that permeates through the entire architecture and introduces trade-offs to achieve increased performance, scalability, and auditability. Once your system becomes an event sourcing system, all future design decisions are constrained by the fact that this is an event sourcing system. There is a high cost to migrate to or from an event sourcing system. This pattern is best suited for systems where performance, scalability, and auditability are top requirements. The complexity that event sourcing adds to a system is not justified for simple systems.

Solution

The Event Sourcing pattern defines an approach to handling operations on data that's driven by a sequence of events, each of which is recorded in an append-only store. Application code raises events that imperatively describe the action taken on the object. The events are generally sent to a queue where a separate process, an event handler, listens to the queue and persists the events in an event store. Each event represents a logical change to the object, such as `AddedItemToOrder` or `OrderCanceled`.

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. Additional event handlers can listen for events they are interested in and take an appropriate action. 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.

At any point, it's possible for applications to read the history of events. You can then use the events to materialize the current state of an entity by playing back and consuming all the events that are related to that entity. This process can occur on demand to materialize a domain object when handling a request.

Because it is relatively expensive to read and replay events, applications typically implement materialized views, read-only projections of the event store that are optimized for querying. For example, a system can maintain a materialized