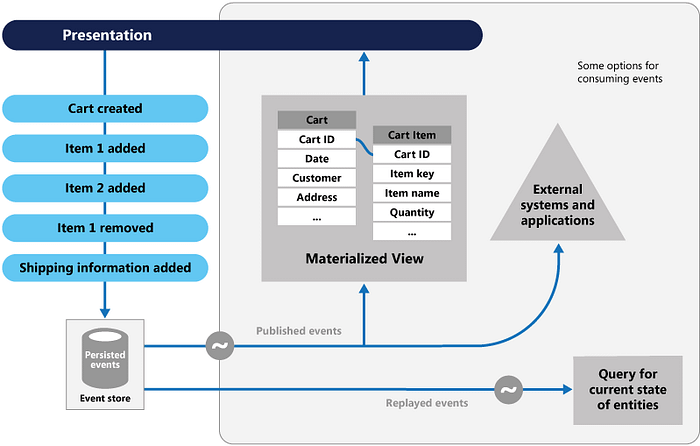
**Event Sourcing Pattern**

Most apps operate with data, and the common method is for the program to keep the data in its present state by updating it when users interact with it. In the classic create, read, update, and delete (CRUD) architecture, for example, a typical data operation is to receive data from the store, make some changes to it, and then update the current state of the data with the new values— often by utilizing transactions that lock the data.

The Event Sourcing design defines a method for handling data activities that are triggered by a series of events, each of which is recorded in an append-only store. Application code delivers a series of events to the event store, where they are persisted, that must describe each action that has occurred on the data. Each event describes a collection of data changes (for example,*"AddedItemToOrder"*).



The events are saved in an event store, which serves as the system of record (the official data source) for the data’s present state. These events are often published by the event retailer so that consumers are aware and can handle them if necessary. Consumers may, for example, start tasks that apply the operations in the events to other systems, or they could execute any other associated action required to finish the process. It’s worth noting that the application code that generates the events is separate from the systems that subscribe to them.

Event Sourcing is a design pattern used in software architecture to store and manage the state of an application by capturing and persisting all changes to the application's data as a sequence of events. Instead of storing the current state of the data, Event Sourcing focuses on recording the history of state-changing events, which can be replayed to reconstruct the current state at any point in time.

In the Event Sourcing pattern, the following concepts are involved:

Events: Events represent state-changing actions that have occurred in the system. Each event is an immutable record that contains all the necessary data to describe the change. For example, if a user account is created, an "AccountCreated" event is generated, capturing relevant information such as the user's name, email, and creation timestamp.

Event Store: The event store is a durable and append-only storage mechanism used to persist events. Events are added to the store in the order they occur and can be retrieved later for historical analysis or state reconstruction.

Aggregate: An aggregate is a cohesive unit that represents a domain object or an entity within the application. In Event Sourcing, aggregates are responsible for processing commands and generating events in response to those commands.

Command: A command is a message that represents the intention to perform a state-changing operation. When a command is received, the aggregate processes it and generates corresponding events, reflecting the changes.

Event Handler: Event handlers are responsible for updating the read models (query models) of the application. They process events and update the query-side data store, allowing efficient read operations on the denormalized data.

Key benefits and characteristics of Event Sourcing include:

Full Audit Trail: Event Sourcing provides a complete and auditable history of all state changes, which can be valuable for debugging, compliance, and regulatory requirements.

Temporal Querying: Since events are stored in the order they occurred, it is possible to query the state of the system at any specific point in time, facilitating temporal queries.

Resilience and Recovery: Event Sourcing provides resilience against system failures. The current state can be reconstructed by replaying the events from the event store.

Domain-Driven Design (DDD) alignment: Event Sourcing aligns well with DDD principles, as aggregates represent domain entities and their behavior.

Flexibility and Evolution: Event Sourcing allows the system to evolve over time as new business requirements emerge, making it easier to introduce new features and modify existing ones.

However, Event Sourcing introduces some complexities, such as managing event versioning, handling eventual consistency, and dealing with temporal queries efficiently. It is best suited for applications with complex business domains, high audit requirements, and a need for historical analysis and state reconstruction.

Frameworks and tools like Axon Framework, EventStore, and Kafka can simplify the implementation of Event Sourcing in your application. When considering Event Sourcing, carefully evaluate the requirements and complexity of your domain to determine if it is the right fit for your project.