

Event Sourcing pattern

06/23/2017 • 14 minutes to read • Contributors      [all](#)

In this article

[Context and problem](#)

[Solution](#)

[Issues and considerations](#)

[When to use this pattern](#)

[Example](#)

[Related patterns and guidance](#)

Instead of storing just the current state of the data in a domain, use an append-only store to record the full series of actions taken on that data. The store acts as the system of record and can be used to materialize the domain objects. This can simplify tasks in complex domains, by avoiding the need to synchronize the data model and the business domain, while improving performance, scalability, and responsiveness. It can also provide consistency for transactional data, and maintain full audit trails and history that can enable compensating actions.

Context and problem

Most applications work with data, and the typical approach is for the application to maintain the current state of the data by updating it as users work with it. For example, in the traditional create, read, update, and delete (CRUD) model a typical data process is to read data from the store, make some modifications to it, and update the current state of the data with the new values—often by using transactions that lock the data.

The CRUD approach has some limitations:

- CRUD systems perform update operations directly against a data store, which can slow down performance and responsiveness, and limit scalability, due to the processing overhead it requires.
- In a collaborative domain with many concurrent users, data update conflicts are more likely because the update operations take place on a single item of data.
- Unless there's an additional auditing mechanism that records the details of each operation in a separate log, history is lost.

For a deeper understanding of the limits of the CRUD approach see [CRUD, Only When You Can Afford It](#).

Solution

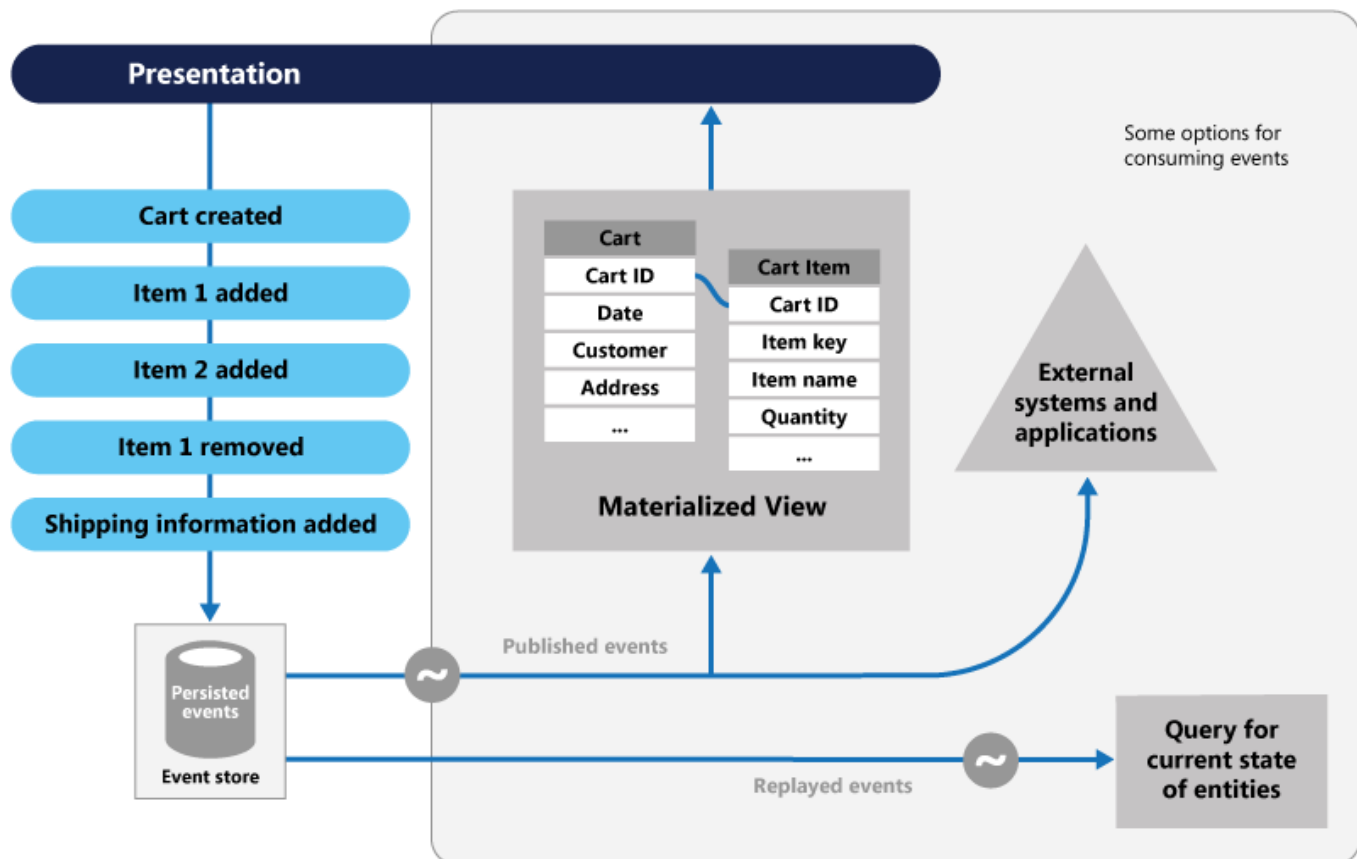
The Event Sourcing pattern defines an approach to handling operations on data that's driven by a sequence of events, each of which is recorded in an append-only store. Application code sends a series of events that imperatively describe each action that has occurred on the data to the event store, where they're persisted. Each event represents a set of changes to the data (such as `AddedItemToOrder`).

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. The event store typically publishes these events so that consumers can be notified and can handle them if needed. 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.

Typical uses of the events published by the event store are to maintain materialized views of entities as actions in the application change them, and for integration with external systems. For example, a system can maintain a materialized view of all customer orders that's used to populate parts of the UI. As the application adds new orders, adds or removes items on the order, and adds shipping information, the events that describe these changes can be handled and used to update the [materialized view](#).

In addition, at any point it's possible for applications to read the history of events, and use it to materialize the current state of an entity by playing back and consuming all the events related to that entity. This can occur on demand to materialize a domain object when handling a request, or through a scheduled task so that the state of the entity can be stored as a materialized view to support the presentation layer.

The figure shows an overview of the pattern, including some of the options for using the event stream such as creating a materialized view, integrating events with external applications and systems, and replaying events to create projections of the current state of specific entities.



The Event Sourcing pattern provides the following advantages:

- Events are immutable and can be stored using an append-only operation. The user interface, workflow, or process that initiated an event can continue, and tasks that handle the events can run in the background. This, combined with the fact that there's no contention during the processing of transactions, can vastly improve performance and scalability for applications, especially for the presentation level or user interface.
- Events are simple objects that describe some action that occurred, together with any associated data required to describe the action represented by the event. Events don't directly update a data store. They're simply recorded for handling at the appropriate time. This can simplify implementation and management.
- Events typically have meaning for a domain expert, whereas [object-relational impedance mismatch](#) can make complex database tables hard to understand. Tables are artificial constructs that represent the current state of the system, not the events that occurred.
- Event sourcing can help prevent concurrent updates from causing conflicts because it avoids the requirement to directly update objects in the data store. However, the domain model must still be designed to protect itself from requests that might result in an inconsistent state.

- The append-only storage of events provides an audit trail that can be used to monitor actions taken against a data store, regenerate the current state as materialized views or projections by replaying the events at any time, and assist in testing and debugging the system. In addition, the requirement to use compensating events to cancel changes provides a history of changes that were reversed, which wouldn't be the case if the model simply stored the current state. The list of events can also be used to analyze application performance and detect user behavior trends, or to obtain other useful business information.
- The event store raises events, and tasks perform operations in response to those events. This decoupling of the tasks from the events provides flexibility and extensibility. Tasks know about the type of event and the event data, but not about the operation that triggered the event. In addition, multiple tasks can handle each event. This enables easy integration with other services and systems that only listen for new events raised by the event store. However, the event sourcing events tend to be very low level, and it might be necessary to generate specific integration events instead.

Event sourcing is commonly combined with the CQRS pattern by performing the data management tasks in response to the events, and by materializing views from the stored events.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

The system will only be eventually consistent when creating materialized views or generating projections of data by replaying events. There's some delay between an application adding events to the event store as the result of handling a request, the events being published, and consumers of the events handling them. During this period, new events that describe further changes to entities might have arrived at the event store.

⚠ Note

See the [Data Consistency Primer](#) for information about eventual consistency.

The event store is the permanent source of information, and so the event data should never be updated. The only way to update an entity to undo a change is to add a compensating event to the event store. If the format (rather than the data) of the persisted events needs to change, perhaps during a migration, it can be difficult to combine existing events in the store with the new version. It might be necessary to iterate through all the events making changes so they're compliant with the new format, or add new events that use the new format. Consider using a version stamp on each version of the event schema to maintain both the old and the new event formats.

Multi-threaded applications and multiple instances of applications might be storing events in the event store. The consistency of events in the event store is vital, as is the order of events that affect a specific entity (the order that changes occur to an entity affects its current state). Adding a timestamp to every event can help to avoid issues. Another common practice is to annotate each event resulting from a request with an incremental identifier. If two actions attempt to add events for the same entity at the same time, the event store can reject an event that matches an existing entity identifier and event identifier.

There's no standard approach, or existing mechanisms such as SQL queries, for reading the events to obtain information. The only data that can be extracted is a stream of events using an event identifier as the criteria. The event ID typically maps to individual entities. The current state of an entity can be determined only by replaying all of the events that relate to it against the original state of that entity.

The length of each event stream affects managing and updating the system. If the streams are large, consider creating snapshots at specific intervals such as a specified number of events. The current state of the entity can be obtained from the snapshot and by replaying any events that occurred after that point in time. For more information about creating snapshots of data, see [Snapshot on Martin Fowler's Enterprise Application Architecture website](#) and [Master-Subordinate Snapshot Replication](#).

Even though event sourcing minimizes the chance of conflicting updates to the data, the application must still be able to deal with inconsistencies that result from eventual consistency and the lack of transactions. For example, an event that indicates a reduction in stock inventory might arrive in the data store while an order for that item is being placed, resulting in a requirement to reconcile the two operations either by advising the customer or creating a back order.

Event publication might be "at least once," and so consumers of the events must be idempotent. They must not reapply the update described in an event if the event is handled more than once. For example, if multiple instances of a consumer maintain an aggregate an entity's property, such as the total number of orders placed, only one must succeed in incrementing the aggregate when an order placed event occurs. While this isn't a key characteristic of event sourcing, it's the usual implementation decision.

When to use this pattern

Use this pattern in the following scenarios:

- When you want to capture intent, purpose, or reason in the data. For example, changes to a customer entity can be captured as a series of specific event types such as *Moved home*, *Closed account*, or *Deceased*.
- When it's vital to minimize or completely avoid the occurrence of conflicting updates to data.
- When you want to record events that occur, and be able to replay them to restore the state of a system, roll back changes, or keep a history and audit log. For example, when a task involves multiple steps you might need to execute actions to revert updates and then replay some steps to bring the data back into a consistent state.
- When using events is a natural feature of the operation of the application, and requires little additional development or implementation effort.
- When you need to decouple the process of inputting or updating data from the tasks required to apply these actions. This might be to improve UI performance, or to distribute events to other listeners that take action when the events occur. For example, integrating a payroll system with an expense submission website so that events raised by the event store in response to data updates made in the website are consumed by both the website and the payroll system.
- When you want flexibility to be able to change the format of materialized models and entity data if requirements change, or—when used in conjunction with CQRS—you need to adapt a read model or the views that expose the data.
- When used in conjunction with CQRS, and eventual consistency is acceptable while a read model is updated, or the performance impact of rehydrating entities and data from an event stream is acceptable.

This pattern might not be useful in the following situations:

- Small or simple domains, systems that have little or no business logic, or nondomain systems that naturally work well with traditional CRUD data management mechanisms.
- Systems where consistency and real-time updates to the views of the data are required.
- Systems where audit trails, history, and capabilities to roll back and replay actions are not required.
- Systems where there's only a very low occurrence of conflicting updates to the underlying data. For example, systems that predominantly add data rather than updating it.

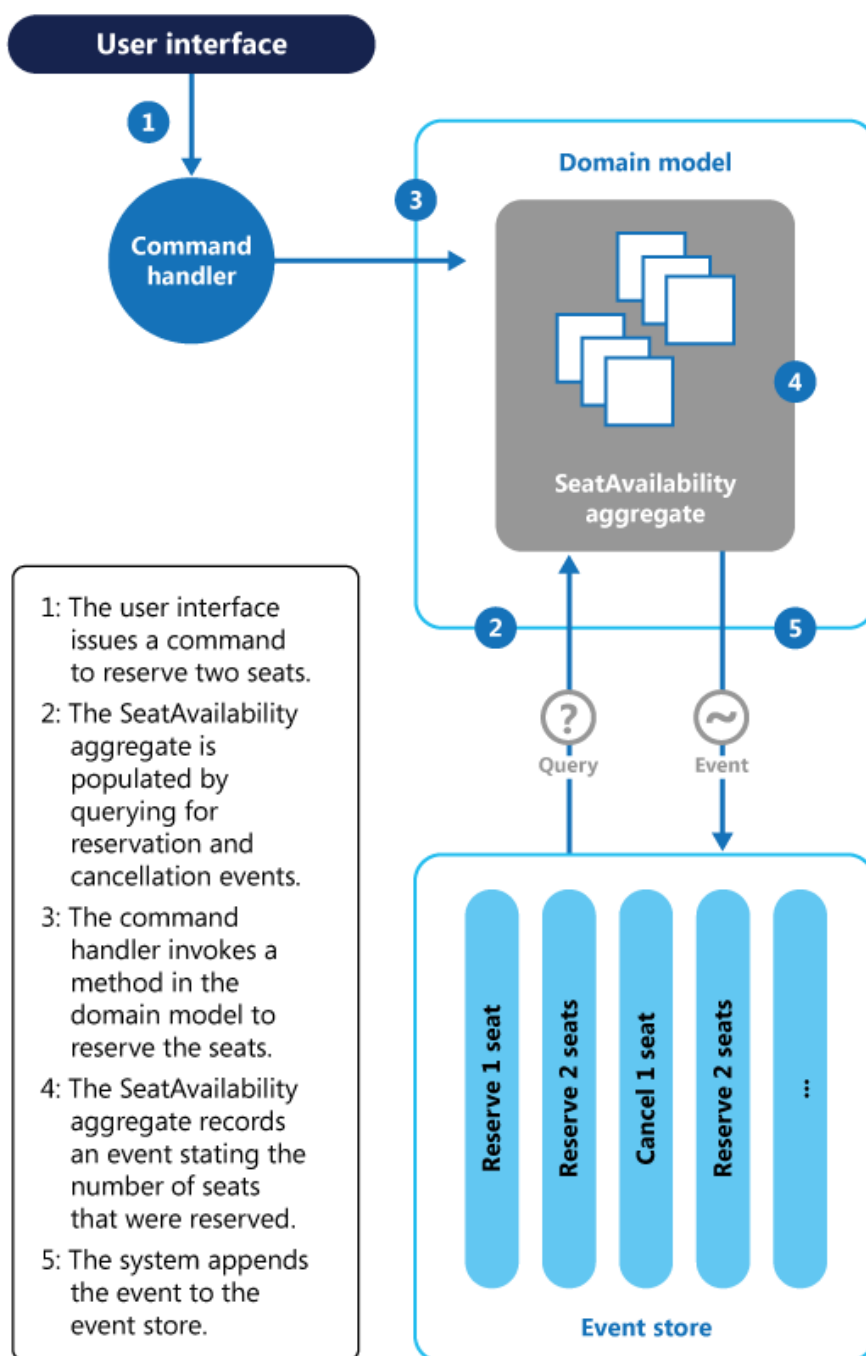
Example

A conference management system needs to track the number of completed bookings for a conference so that it can check whether there are seats still available when a potential attendee tries to make a booking. The system could store

the total number of bookings for a conference in at least two ways:

- The system could store the information about the total number of bookings as a separate entity in a database that holds booking information. As bookings are made or canceled, the system could increment or decrement this number as appropriate. This approach is simple in theory, but can cause scalability issues if a large number of attendees are attempting to book seats during a short period of time. For example, in the last day or so prior to the booking period closing.
- The system could store information about bookings and cancellations as events held in an event store. It could then calculate the number of seats available by replaying these events. This approach can be more scalable due to the immutability of events. The system only needs to be able to read data from the event store, or append data to the event store. Event information about bookings and cancellations is never modified.

The following diagram illustrates how the seat reservation subsystem of the conference management system might be implemented using event sourcing.



The sequence of actions for reserving two seats is as follows:

1. The user interface issues a command to reserve seats for two attendees. The command is handled by a separate command handler. A piece of logic that is decoupled from the user interface and is responsible for handling requests posted as commands.

2. An aggregate containing information about all reservations for the conference is constructed by querying the events that describe bookings and cancellations. This aggregate is called `SeatAvailability`, and is contained within a domain model that exposes methods for querying and modifying the data in the aggregate.

Some optimizations to consider are using snapshots (so that you don't need to query and replay the full list of events to obtain the current state of the aggregate), and maintaining a cached copy of the aggregate in memory.

3. The command handler invokes a method exposed by the domain model to make the reservations.
4. The `SeatAvailability` aggregate records an event containing the number of seats that were reserved. The next time the aggregate applies events, all the reservations will be used to compute how many seats remain.
5. The system appends the new event to the list of events in the event store.

If a user cancels a seat, the system follows a similar process except the command handler issues a command that generates a seat cancellation event and appends it to the event store.

As well as providing more scope for scalability, using an event store also provides a complete history, or audit trail, of the bookings and cancellations for a conference. The events in the event store are the accurate record. There is no need to persist aggregates in any other way because the system can easily replay the events and restore the state to any point in time.

You can find more information about this example in [Introducing Event Sourcing](#).

Related patterns and guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- [Command and Query Responsibility Segregation \(CQRS\) pattern](#). The write store that provides the permanent source of information for a CQRS implementation is often based on an implementation of the Event Sourcing pattern. Describes how to segregate the operations that read data in an application from the operations that update data by using separate interfaces.
- [Materialized View pattern](#). The data store used in a system based on event sourcing is typically not well suited to efficient querying. Instead, a common approach is to generate prepopulated views of the data at regular intervals, or when the data changes. Shows how this can be done.
- [Compensating Transaction pattern](#). The existing data in an event sourcing store is not updated, instead new entries are added that transition the state of entities to the new values. To reverse a change, compensating entries are used because it isn't possible to simply reverse the previous change. Describes how to undo the work that was performed by a previous operation.
- [Data Consistency Primer](#). When using event sourcing with a separate read store or materialized views, the read data won't be immediately consistent, instead it'll be only eventually consistent. Summarizes the issues surrounding maintaining consistency over distributed data.
- [Data Partitioning Guidance](#). Data is often partitioned when using event sourcing to improve scalability, reduce contention, and optimize performance. Describes how to divide data into discrete partitions, and the issues that can arise.