

SSENSE-TECH · [Follow publication](#)

Event Sourcing: A Practical Guide to Actually Getting It Done

10 min read · Mar 22, 2024



Sam-Nicolai Johnston

[Follow](#)

Listen



Share

Event Sourcing — the pattern for storing data as events in a log and using them to reconstruct its state — has been used for many years across various industries. From a general accounting ledger to a financial bank statement, a bill amendment, or a contract addendum. Despite its long history, event sourcing remains relatively uncommon in software development. Our Principal Software Architect, Mario Bittencourt, provided a great overview of the topic in this informative [series](#).

In 2020, I was the technical lead of the team that implemented the first event-sourced system at SSENSE. Although I had read a lot about event sourcing and was sold on its benefits for our project, I found it **very** challenging to jump in and implement a production event-sourced system. I had so many questions! This guide is for my past self, and anyone else in the same boat, to help make the jumping-off point less stressful and avoid common pitfalls.

NOTE: This article assumes basic knowledge of Event Sourcing and CQRS.

What Is the Value-Add of Event Sourcing?

Think of the value that event sourcing brings: it provides an append-only list of things that happened to your aggregates (if you're not sure what an "aggregate" is, check out Pablo Martinez's insightful [article](#) on domain-driven design). Depending on your use cases, this might:

- be a requirement, i.e. if you're working in a financial institution,
- bring no business value to your organization,

- lie somewhere in the middle, where knowing each change that occurred brings value to the customers and/or the business.

Creating an event-sourced system from the get-go can help change your perspective on the problem and solution spaces and how to tackle them. However since it's somewhat uncommon in software development, you must consider the overhead, documentation, and training requirements before jumping in.

What Does an Event Look Like?

Events are first-class citizens in event-sourced systems, so it is important to understand what they should contain.

To help you follow along and implement something similar in your production environment, let's use a common example: an e-commerce order. An order might look like this to a customer:

MENSWEARWOMENSWEAREVERYTHING ELSESEARCH

SSENSE

ENGLISHACCOUNTWISHLISTSHOPPING BAG (0)

ORDER MyOrderId
Date Placed: January 3, 2024

Print

Account

[Order History](#)
SSENSE+
Account Details
Email Preferences
Addresses
Appointments
Logout

SHIPPING METHOD
Standard


PAYMENT METHOD
VISA *****

SHIPPING ADDRESS
Sam-Nicolai Johnston
1234 Somewhere
SomeCity, SomeState
Canada
111-123-1234

BILLING ADDRESS
Sam-Nicolai Johnston
1234 Somewhere
SomeCity, SomeState
Canada
111-123-1234

SHIPPED

TRACK ORDER



THE NORTH FACE
Black Nuptse Down Coat
Size: S
SKU: 232802F06116801

\$300.00
~~\$600.00~~

TOTAL ORDER SUMMARY

Subtotal	\$300.00
Shipping total	\$0.00
GST (5 %)	\$15.00
PST (9.975 %)	\$29.93
ORDER TOTAL	\$344.93 CAD

RETURN ITEMS

COUNTRY/REGION: CANADA

NEWSLETTER SIGNUP

CUSTOMER CARE

BOOK AN APPOINTMENT

SSENSE MONTRÉAL

EDITORIAL ARCHIVE

CAREERS

AFFILIATES

SITEMAP

The order goes through multiple stages in its lifecycle, but to keep it simple, let's imagine there are only three types of events that can happen:

1. The order is placed
2. The payment is captured
3. The package is shipped

Notice that all these events are past tense, representing something that has **already** happened. These events should only contain the **minimum** amount of data, not the entire aggregate. This means each of these events will capture different details.

Event type	Event data
Order Placed	<pre>{ shippingAddress: { ... }, billingAddress: { ... }, items: [{ sku: 'sku1', quantity: 1 }], shippingMethod: 'fedex-express' }</pre>
Payment Captured	<pre>{ transactionId: 'someTransactionId' }</pre>
Package Shipped	<pre>{ trackingId: 'someTrackingId', items: [{ sku: 'sku1', quantity: 1}] }</pre>

But since we might persist multiple types of aggregates, such as orders, customers, products, saved addresses, etc., it's important to add some sort of unique domain name or namespace to the events.

Event type	Event data
Order/OrderPlaced	{ shippingAddress: { ... }, billingAddress: { ... }, items: [{ sku: 'sku1', quantity: 1 }], shippingMethod: 'fedex-express' }
Order/PaymentCaptured	{ transactionId: 'someTransactionId' }
Order/PackageShipped	{ trackingId: 'someTrackingId', items: [{ sku: 'sku1', quantity: 1 }] }

This way, we can avoid name clashes between a package shipped for an order (*Order.PackageShipped*) and a package shipped for a stock transfer between 2 fulfillment centres (*StockTransfer.PackageShipped*).

Next, we need to keep track of the specific identifier of the aggregate and the date the event happened.

Aggregate Id	Event type	Event data	OccurredOn
VSI98005	Order/OrderPlaced	{ shippingAddress: { ... }, billingAddress: { ... }, items: [{ sku: 'sku1', quantity: 1 }], shippingMethod: 'fedex-express' }	2024-01-03T18:18:18Z
VSI98005	Order/PaymentCaptured	{ transactionId: 'someTransactionId' }	2024-01-03T19:18:18Z
VSI98005	Order/PackageShipped	{ trackingId: 'someTrackingId', items: [{ sku: 'sku1', quantity: 1 }] }	2024-01-04T18:18:18Z

Lastly, we will need to add two more elements that will make our lives easier down the line:

1. Event Version

Event versions allow our system to grow and evolve over time. Each new event version will change the data that is persisted with the event. It should also be processed differently when rebuilding the aggregate or projections, which we will discuss in a later section. For example, if we decide down the line to allow coupons to be added when placing an order, this change won't affect past events, as they are immutable.

Aggregate Id	Event type	Event data	OccurredOn	Metadata
VSI98005	Order/OrderPlaced	{ shippingAddress: { ... }, billingAddress: { ... }, items: [{ sku: 'skul', quantity: 1 }], shippingMethod: 'fedex-express' }	2024-01-03T18:18:18Z	{ eventVersion: '1.0.0' }
VSI98005	Order/PaymentCaptured	{ transactionId: 'someTransactionId' }	2024-01-03T19:18:18Z	{ eventVersion: '1.0.0' }
VSI98005	Order/PackageShipped	{ trackingId: 'someTrackingId', items: [{ sku: 'skul', quantity: 1 }] }	2024-01-04T18:18:18Z	{ eventVersion: '1.0.0' }
KOI88672	Order/OrderPlaced	{ shippingAddress: { ... }, billingAddress: { ... }, items: [{ sku: 'skul', quantity: 1 }], shippingMethod: 'fedex-express', coupons: ['black-friday'] }	2024-01-03T18:18:18Z	{ eventVersion: '1.1.0' }

Now you can see that the new *Order/OrderPlaced* events have a 1.1.0 version, which includes the new *coupons* field.

*TIP: Use SEMVER to easily determine how you will need to adjust your aggregate and projectors to support the multiple versions that exist. Your code must be able to support **all** existing versions. In the example above, we added a new field, resulting in a minor version getting bumped, as it is a new feature without breaking changes. Removing or renaming a field would be a breaking change, requiring a major version change.*

So, there is no need for SQL migration, but it comes at the cost of having to maintain different event versions as our system evolves over time.

2. Sequence Number

Next, let's add a sequence number. I've fallen in love with sequence numbers while working in an event-driven architecture (EDA); they make everyone's lives much easier and are simple to implement. They can also be added to non-event-sourced systems, so *please* use them in your EDA!

The objective is to have a number that increases by one for each new event in a *single* aggregate:

Aggregate Id	Sequence Number	Event type	Event data	OccurredOn	Metadata
VSI98005	1	Order/OrderPlaced	{ shippingAddress: { ... }, billingAddress: { ... }, items: [{ sku: 'skul', quantity: 1 }], shippingMethod: 'fedex-express' }	2024-01-03T18:18:18Z	{ eventVersion: '1.0.0' }
VSI98005	2	Order/PaymentCaptured	{ transactionId: 'someTransactionId' }	2024-01-03T19:18:18Z	{ eventVersion: '1.0.0' }
VSI98005	3	Order/PackageShipped	{ trackingId: 'someTrackingId', items: [{ sku: 'skul', quantity: 1 }] }	2024-01-04T18:18:18Z	{ eventVersion: '1.0.0' }
KOI88672	1	Order/OrderPlaced	{ shippingAddress: { ... }, billingAddress: { ... }, items: [{ sku: 'skul', quantity: 1 }], shippingMethod: 'fedex-express' coupons: ['black-friday'] }	2024-01-03T18:18:18Z	{ eventVersion: '1.1.0' }

This will facilitate the creation of optimistic locking and the handling of failures when they happen. To implement this, you can use a unique constraint on the combination of aggregate id and sequence number if you're using an SQL database, or the "Condition Expression" if you're using DynamoDB, for example.

The sequence number will also make it easy for consumers of these events to determine if they have:

1. Missed an event
2. Received the same event multiple times
3. Received the events out-of-order — although not applicable in this simple example, processing the events in order might be critical for certain use cases

So, how do we efficiently persist in all of this?

How Do I Persist All This Data?

Considering that these systems are online transaction processing (OLTP) systems, their main use case is to fetch an aggregate from persistence, perform a command/action on it i.e. *order.ship(trackingNumber)* and save the new event.

Sometimes, it's also necessary to publish "integration" events that contain the full state of the aggregate for other services to use. In these cases, it's important to be able to query by a specific sequence number as well. We'll cover that use case in the "*How do I Publish Events?*" section below.

Get Sam-Nicolai Johnston's stories in your inbox

Join Medium for free to get updates from this writer.

[Subscribe](#)

So, the *only* things that really need to be queried/filtered are the aggregate id and the sequence number. Other access patterns can be handled by projections using the CQRS pattern; it's relatively easy when you have an event-sourced system. You can read more about it [here](#).

TIP: Remember, this is an OLTP system, not an online analytical processing (OLAP) system. It's meant to process transactions, not answer complex business questions. We'll cover creating projections for analytical use cases in the next section.

Most databases can meet these needs, so weigh the pros and cons of each solution. We went forward with DynamoDB because it provided the necessary capabilities for us and because it's fully managed, fast, highly reliable, and can easily scale in terms of throughput and storage. Something to keep in mind is that DynamoDB has a maximum item size of 400KB, which may not be enough if your events are really big.

Depending on your needs, a dedicated solution like [Event Store DB](#) could be required as it supports more complex cases.

What Are Projections and How Do I Build Them?

A projection is a representation of the data in a different format. It could be anything you can think of that can be built based on the underlying events.

It serves two main purposes:

1. Analyzing your data (an OLAP system), such as viewing all orders placed between dates X & Y, with at least 2 items but no more than 5, and a subtotal of \$500 to \$600.
2. Providing eventually-consistent data for the customer that is optimized for the use case(s), like pre-computing complicated or expensive queries that are often needed.

In our example, we might want to have a very optimized way to get a list of orders for a customer that returns only the necessary fields for the order history page:

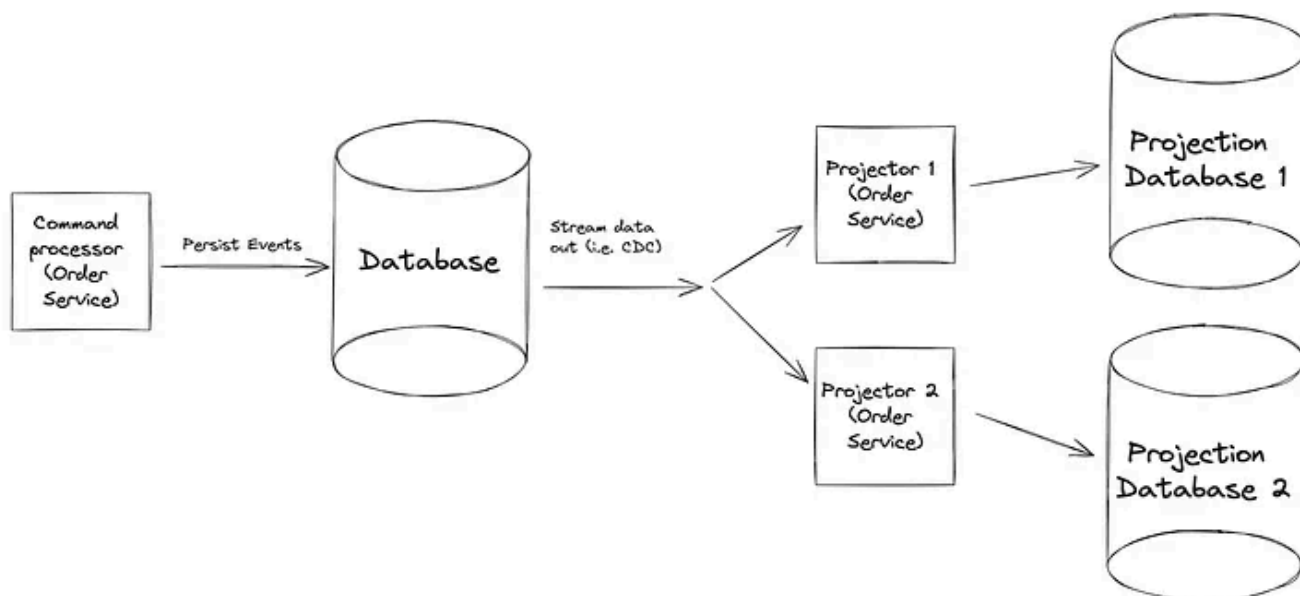


This projection could look something like this:

Customer Id	order Id	Status	Subtotal	Tracking Number
CU1923	VSI98005	Placed	175.89	UPS123123
CU1923	KOI88672	Shipped	257.23	FED155543
CU1923	BAT12311	Paid	18.34	TRK123123
CU19AF	CAN1244	Shipped	75.22	USPS12132

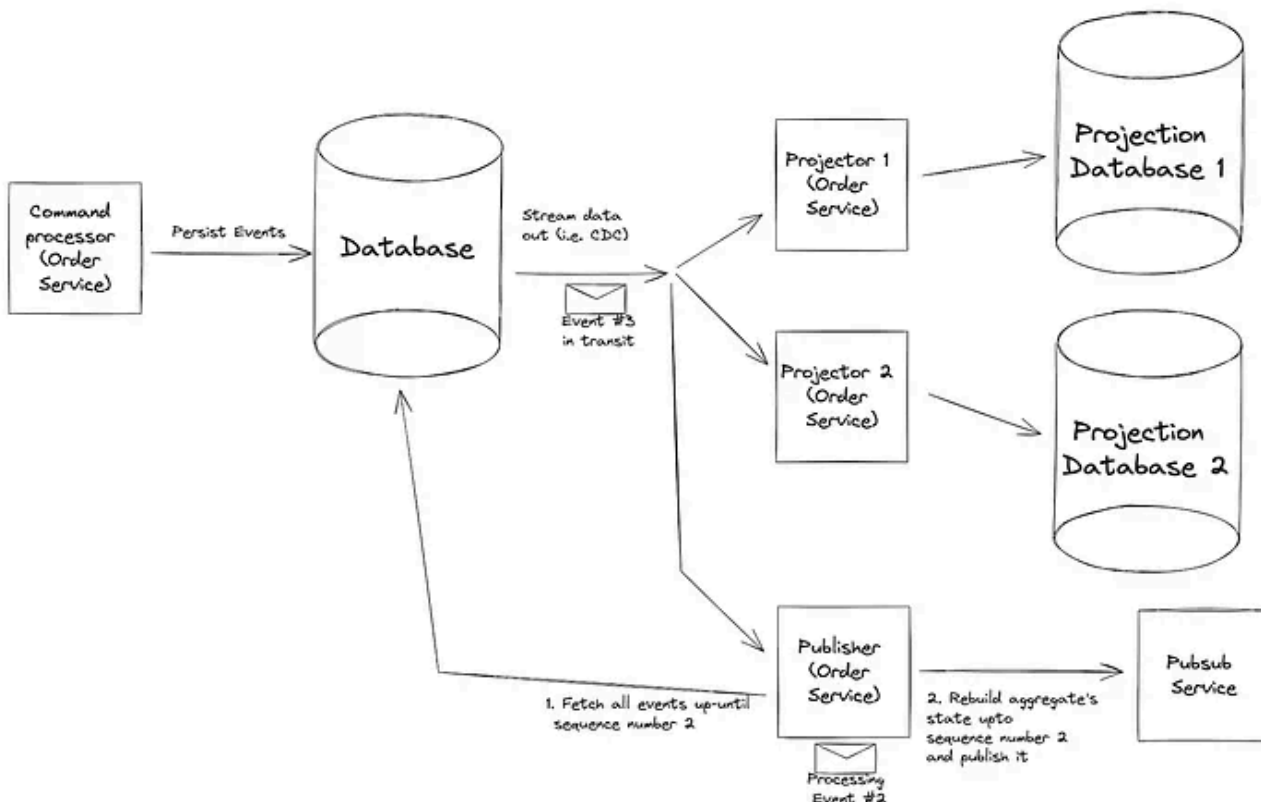
We would only fetch the full order when the customer clicks on the *View Details* button.

To build these projections, the database events are extracted, using for example change data capture (CDC), and streamed, using for example Kinesis, to be processed by one or more “projectors” that will build the necessary projection(s).



How Do I Publish Events?

Sometimes it's useful to publish your entire aggregate each time it changes. For this use case, it's very helpful to have sequence numbers in order to be able to rebuild the aggregate at that specific point in time. Sequence numbers can prevent race conditions when publishing your aggregates while there are in-flight events.



What Does the Code Look Like?

Now that we know what we need to persist, let's take the time to explore one of many ways in which event sourcing can be implemented. This is not production code, of course, but it

can give you a good idea if you like to learn through code.

Let's start with the repository, which is meant to save and retrieve only the events. This implies that the aggregate must provide, at a minimum, a way to:

1. get all pending events to persist
2. rebuild its state based on the events (the *when* method)

Next, let's look at the aggregate, which needs to provide a way to perform commands on the aggregate i.e. *place(...)*, in addition to what the repository requires. Notice that the properties of the aggregate are only changed in the *whenXYZ* methods, which are used both when performing commands and when rebuilding the aggregate from persistence:

This aggregate functionality is coded in one class, but my recommendation would be to extract a base class that would handle most of the heavy lifting and could be reused by all your aggregates.

Now, what does a projector look like? Its job is to receive the events and build a specific projection. So if I wanted to save the current state after each event, it would look something like this:

How Do I Make Sure Developers Have the Necessary Knowledge?

The value of event sourcing is significant. It makes investigations a breeze, helps our employees understand what happened and in what order, and guides business decisions with precise analytical information.

As mentioned in the introduction, event sourcing still isn't common in software development compared to other industries where it's the norm. It's simply a different way of thinking, and anything that is unfamiliar will be harder to grasp initially. So, you need to invest the time and resources into training, workshops, good documentation, and good diagrams to ensure that your development team can maintain it and your company can reap the benefits.

Recommendations

1. **Invest in documentation** — It doesn't need to be anything too detailed or fancy: a good README goes a long way. Include links to introductory information about event sourcing and CQRS (if applicable). Include high-level diagrams to help newcomers get a lay of the land (i.e. the high levels of the C4 model).
2. **Build runbooks** — In addition to documentation, having easy-to-follow runbooks for your system is essential, in case things break and a developer outside your team needs to quickly fix the issue. This is especially important for systems using architecture patterns that might be unfamiliar to the audience.
3. **Reuse projections (if possible)** — Initially, it might be tempting to create a new projection for different use cases that might require different data for various reasons. However, projections cost money for database resources and are also an overhead for maintenance. Take your time to determine if the new projections are worth it: is your

current projection database under too much strain because it's trying to answer too many unrelated, specific and complex use cases?

4. **Monitor your system** — CQRS systems generally have more moving parts, so make sure to monitor all of them closely i.e. events are consumed by the projectors and publishers.
5. **Ensure you can get the events, not just the full aggregate** — Events are very useful to debug what happened to complex aggregates in complex systems that are in a strange state. Usually, stakeholders are also interested in these events, so including them in a log tab of your UI is doubly useful.
6. **Use sequence numbers** — It'll make out-of-order events and duplicate events a breeze to process.
7. **Invest early in a simple system to rebuild projections** — Projections aren't the source of truth. They are meant to be entirely replaceable if the needs change. Being able to rebuild projections (i.e. in a new database instance) can be useful if your projection becomes outdated or broken for some reason. Ideally, you only want to build a new projector, create an empty projection, then run a CI/CD pipeline which will re-emit all historical events to this new projector.

What Now?

Adopting an event sourcing solution may seem daunting at first due to its novel aspect for many developers. However, it doesn't have to be overwhelming. As we have seen, it is possible to have a good starting point for implementing your own event sourced system while avoiding some common pitfalls.

It is important to be mindful of what should be included in events, as changing the event structure later on can impact the downstream consumers, including yourself, when handling projections.

While your mileage may vary, it is possible to address many use cases without the need for complicated frameworks or single-purpose persistence as your event store. Just remember what is the minimum amount of information that needs to be persisted and how CQRS and projections can help you answer all business questions.

With this in mind, you're ready to experiment with event sourcing in your next application.

. . .

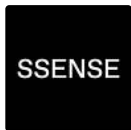
Editorial reviews by [Catherine Heim](#), [Luba Mikhnovsky](#) & [Mario Bittencourt](#).

Want to work with us? Click [here](#) to see all open positions at SSENSE!

Cqrs

Event Sourcing

Event Driven Architecture



Follow

Published in SSENSE-TECH

1.5K followers · Last published Mar 21, 2025

Ideas and research from the software, data & product teams behind the global fashion platform SSENSE.



Follow

Written by Sam-Nicolai Johnston

40 followers · 3 following