Whitepaper

Getting started with event sourcing in 2025

Bruce Hopkins

Developer Relations

Table of contents

1. Executive summary

Traditional software systems have long relied on relational databases to store and manage data. These databases are designed to keep the current state of an application's data, updating records as changes occur. While this method is straightforward and efficient for many purposes, it inherently lacks the ability to provide insights into how or why a particular state was reached. This limitation poses significant challenges in today's world, where transparency, traceability, and the ability to audit changes are increasingly important.

Today's software systems face challenges in maintaining an accurate and comprehensive history of data changes. Traditional architectures often rely on methods that capture only the current state of data, losing valuable historical context. This approach can complicate debugging, auditing, and the implementation of features such as the undo function for users.

Additionally, in recent years, message brokers have been used as an essential component for enabling real-time event streaming and communication between microservices within modern software architectures. While they excel at delivering messages rapidly between services, they typically do not incorporate a robust mechanism for data persistence. In other words, traditional message brokers are designed primarily to dispatch messages efficiently rather than preserve event history. This limitation often leads developers to invest heavily in custom solutions—such as integrating with external databases or building sophisticated persistence layers—just to capture a persistent record of events.

This white paper sheds light on the limitations of traditional data persistence methods and standalone event streaming tools. We will introduce event sourcing as a powerful, intuitive alternative that makes sense. Event sourcing is a software design pattern that records important changes in the system as an event, allowing developers to reconstruct any past state of the application. More importantly, we'll demonstrate how AxonIQ's solutions, namely Axon Framework, Axon Server, and AxonIQ Console, offer a unified and simplified approach to implementing event sourcing at scale.

2. Target audience

This document is tailored for technical decision-makers, developers, and architects eager to modernize their applications and simplify their infrastructure for messaging and data persistence. Whether building a new system or refactoring an existing one, understanding how to make event sourcing work for you is essential for creating robust and maintainable software.

3. The problems and challenges with traditional data persistence

The data persistence layer is the foundation of any application. While application developers are responsible for the business logic of an application, they rely on their data persistence layer to provide a means of preserving the application's data.

For decades, the de facto standard for data persistence has been the relational database and, more recently, the NoSQL database. However, traditional databases have serious limitations when providing a complete view of what's happening (and what has happened) in your application.

One of the most significant challenges of snapshot-based storage is the loss of historical records. Without a trace of previous states or changes, it becomes nearly impossible to understand the evolution of data or the reasons behind specific modifications. This lack of history poses severe limitations for debugging and auditing purposes. When an error occurs, developers are left without the necessary context to trace the root cause or to understand the sequence of events leading to the issue.

3.1. The real-world impact on various industries

In industries where regulatory compliance is crucial, the inability to provide a comprehensive audit trail can result in non-compliance and potential legal repercussions. Auditors and regulators often require detailed records of data changes, including who made a change, when it was made, and why. Therefore, traditional databases that only focus on preserving the current state of the data within your system are significantly hindered in providing this level of detail.

These challenges have significant real-world implications. For instance, in the financial sector, the inability to reconstruct events leading to a particular account state can delay fraud detection and risk management. Similarly, understanding the historical changes to a patient's record in healthcare is vital for accurate treatment and diagnosis.

Furthermore, as business requirements evolve, systems built on traditional data persistence methods face difficulties adapting to new demands, such as integrating advanced analytics and machine learning models that rely on historical data for training and prediction.

Consider the scenario where a product manager for an e-commerce application wants to know, "One week ago, what was the inventory count for this item?" Unfortunately, this question is impossible to

answer if you only maintain the current state in a traditional database. Your tables will have the current inventory state, but there is no record of the inventory level in the past.

Now, let's delve a little deeper. To ensure that the e-commerce site always has a high-revenue product in stock, she wants to know, "When, during the past year, was the inventory of this item below 5 units?" Once again, this question is impossible to answer with traditional databases and persistence frameworks. While the table shows the current state of the inventory, there is no record of the changes to the inventory level.

3.2. Databases can't keep up with the transaction volume needed for scalable microservices

CTOs and product architects have understood for decades now that to build scalable distributed systems, relying solely on a single database to function as the primary communication hub can bottleneck your microservices architecture. As transactional demands surge from multiple users and services, all vying for data consistency, the database becomes a traffic jam of read (and especially write) operations. This design quickly leads to performance slowdowns as each query competes for time and resources. When all your microservices require near-instant communication, a single database with tight coupling to the business logic often crumbles under pressure.

Beyond performance constraints alone, the sheer act of coordinating concurrency across numerous microservices reveals how fragile a database-centric approach can be. If two or more services start updating the same records simultaneously, locking conflicts or lengthy transactions are inevitable. Consider, for example, the e-commerce scenario described above. What happens if five customers are trying to buy the same shoe, handbag, or coat on your site simultaneously? These collisions of locking conflicts will ripple through your infrastructure and create an orchestration nightmare. For decision-makers and product managers, it's essential to realize that when your business logic depends too heavily on the scalability of a database, your ability to grow can grind to a halt at the exact moment your system needs to scale up.

Therefore, modern architects use an event-driven approach instead of allowing the database to act as the backbone of every interaction and communication within your system. Microservices gain a more fluid exchange by orchestrating communication through a dedicated message broker, such as a message queue. Services broadcast and subscribe to events in real-time, enabling high throughput and robust decoupling of business logic. This model aligns with continuous integration and deployment goals, offering a foundation for reliable scaling without forging heavy dependencies on a single data store.

Equally critical is reducing the “blast radius” of service failures. A database by itself can become a single point of failure if it experiences load spikes or maintenance downtime. Conversely, a message-based infrastructure thrives on distributing both data and control logic across many nodes. Such resiliency keeps microservices humming, even if parts of the system stall or go offline.

An event-driven communication layer empowers you to extend your business capabilities in ways not possible with a database-driven mindset. You can introduce new services that listen to specific events and emit their outputs without fear of consuming too many connections to an overloaded database. This approach fosters innovation and scalability: new features can come online faster, and product teams can pivot quickly to meet market demands.

By fostering free and open microservice communication, you balance transactional loads and position your organization to handle the unpredictable growth that tomorrow’s markets demand. However, enterprises still need to utilize some form of persistent storage for their applications and services to function properly. How is this accomplished within systems that only rely on event streaming?

4. The problems and challenges with event streaming and message brokers

To explain the challenges involved with event streaming, let's first define what it is. At its core, event streaming is an architectural pattern focusing on the continuous flow of data (called “events”) from various sources, allowing systems to process and react to these events in real-time. This approach is optimized for real-time data flow, but is not built for data persistence. While event streaming is invaluable for enabling real-time analytics, monitoring, and immediate feedback loops, it falls short when retaining and analyzing historical data over extended periods is needed. As a result, oftentimes, event streaming systems are used in combination with a traditional database to persist the current state of the system. However, as we already know, this approach has significant downsides.

Event streaming solutions also face the same critical limitation of traditional databases, where they cannot maintain coherent and comprehensive audit trails. In industries such as finance, healthcare, and legal services, where regulatory compliance is paramount, the need for transparent and traceable data practices is non-negotiable. As a result, event streaming often results in fragmented audit trails, making it difficult to reconstruct past states or actions within a system. This fragmentation arises because events are typically scattered across multiple streams and nodes, complicating efforts to piece together a cohesive historical record.

This lack of integration complicates the architecture and increases maintenance overhead. Developers and operations teams must implement additional layers of logic and monitoring to ensure that both the messaging and persistence platforms remain in sync. This added complexity can lead to higher costs and increased chances of errors as more systems need to be managed and coordinated.

4.1. The case for a unified approach

The limitations of event streaming highlight the urgent need for a unified approach to data persistence and processing. Today's businesses require solutions that can seamlessly merge real-time messaging with persistent storage, ensuring that current and historical data are readily accessible and usable. A unified approach would provide the best of both worlds: the immediacy and responsiveness of event streaming coupled with the durability and reliability of long-term storage solutions.

Such a solution would simplify architectural complexities and enhance the overall efficiency and effectiveness of data-driven applications. By integrating real-time processing with persistent storage, businesses can comprehensively view their data, enabling more accurate analyses, better

decision-making, and improved operational efficiency. Moreover, a unified approach would facilitate easier maintenance of audit trails and enhance the ability to integrate diverse systems, ultimately leading to more robust and scalable business solutions.

5. Introducing event sourcing: the new paradigm for data persistence

Event Sourcing is a new approach to data persistence that just makes sense. While developers have long embraced the relational database despite its shortcomings, the industry is now adopting this new approach to persisting data. Simply put, instead of storing the current state of your data (like a relational database does), with Event Sourcing, you store every significant change to your data. This provides developers with rich information to answer any business question.

5.1. The basics of event sourcing: a better way to record every change

At its core, event sourcing represents a fundamental shift in how we approach data persistence. Instead of focusing on the end result or the current state of data, event sourcing emphasizes the importance of every change and the transition that your data undergoes. This approach advocates for recording each significant change as a distinct event, capturing the full lifecycle of data within a system.

Event sourcing provides a comprehensive and chronological history of all modifications by storing a sequence of events. This ensures that you have a complete record of data transformations and allows for the reconstruction of any past state. The ability to replay events and rebuild previous states offers unparalleled flexibility and insight, making it easy to answer complex business questions and perform deep analyses.

For example, in an e-commerce scenario, an Order might be associated with the following events, such as "OrderCreated," "ItemAdded," "OrderShipped," and "OrderCancelled." A dedicated stream of events for each simplifies the management and retrieval of relevant data and enhances the system's scalability and performance. By grouping relevant events together (the industry term is called “aggregating"), the system can efficiently process and store vast amounts of data without becoming bogged down by dependencies or irrelevant information.

Furthermore, maintaining event streams in this manner facilitates targeted debugging and auditing. When an issue arises, developers can quickly access its specific event stream to trace the sequence of events and identify potential problems.

5.2. Easily rebuild application state

One of the most powerful features of event sourcing is its ability to rebuild the application state at any point in time. Since the system maintains a complete record of all events, it can reconstruct the application's state as it was at any given moment. This capability is invaluable for debugging, auditing, and historical analysis tasks.

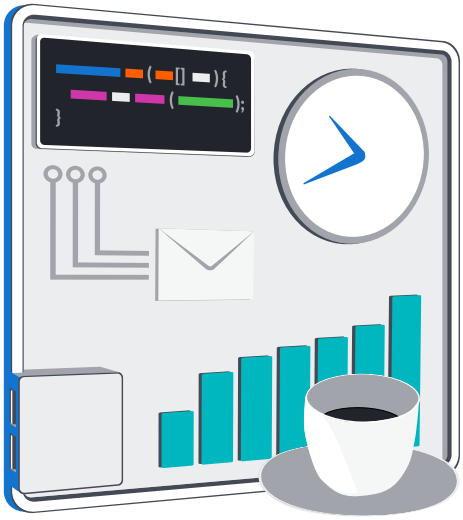
For instance, if a bug is reported within your application, your developers can simply replay the relevant events to see exactly how the system reached a particular state. This makes it easier to identify the root cause of the issue and implement a fix. Similarly, for auditing purposes, event sourcing allows organizations to provide definitive answers to questions about past states and actions within the system, enhancing transparency and compliance.

The ability to rebuild the application state supports advanced analytical techniques, such as time-based data projections and simulations. Businesses can explore "what-if" scenarios by modifying or injecting events and observing the resulting state changes, enabling more informed decision-making and strategic planning.

6. Axon Framework: event sourcing in a single package

With over 3.4k stars on Github, Axon Framework is the #1 rated open-source Java framework that's purpose-built for event sourcing. It's specifically designed to simplify creating and implementing scalable microservices within event sourced architectures. One of the key challenges developers face when adopting event sourcing is the steep learning curve and the boilerplate code often required to get started. Axon Framework addresses these issues by providing tools and libraries that streamline the development process, making it easier for developers to build robust, event-driven applications.

Axon Framework supports annotation-based configurations. Therefore, developers can use simple annotations to declare all the components necessary for an event-sourced system, such as Command handlers, Event handlers, and other essential components. In placing these annotations, Axon Framework can automatically wire the handlers to the command bus and event store, respectively. As a result, your codebase becomes more readable and maintainable, and developers can focus on writing business logic rather than plumbing code.



Axon Framework promotes the use of CQRS (Command Query Responsibility Separation), a pattern that separates command processing (write operations) from query handling (read operations). This separation enhances application scalability and creates optimized read models tailored to specific query requirements. With Axon Framework, developers can focus on implementing business logic while leveraging the framework's built-in capabilities to handle the intricacies of event sourcing and CQRS.

As a library with few dependencies, Axon Framework is flexible enough to be integrated into existing architectures. This means that teams with legacy systems can gradually adopt event sourcing and the CQRS design pattern without requiring a complete rewrite. The framework's modular design allows developers to implement parts of the architecture as needed, facilitating a smoother transition to a fully event-driven model.

6.1. Simplification of event sourcing

One of the most significant benefits of Axon Framework is that it dramatically simplifies the process of implementing event sourcing in your applications. The framework provides built-in support for defining and managing aggregates, which are the root entities of your domain model. Aggregates are responsible for encapsulating the state and behavior of a specific part of the system, and they generate events to represent state changes. With Axon Framework, you can easily define aggregates and their associated event handlers, reducing the manual coding required for event handling.

Axon Framework dramatically simplifies the process of event storage and retrieval. When your system generates an event, the framework takes care of persisting the event to the event store, Axon Server. This eliminates the need for developers to write custom code to handle event persistence, making the development process more efficient. Additionally, Axon Framework provides tools for event replay, allowing you to rebuild the state of your application by replaying events, which is essential for debugging and auditing purposes.

6.2. Write less code for your microservices

When working with a traditional database, developers either need to leverage a persistence framework or write the code necessary for Create, Read, Update, and Delete statements. All developers who have database experience are familiar with this process; however, it adds layers of code that needs to be created and maintained. As a result, challenges arise when dealing with complex business requirements.

The most notable challenge is the object hydration and dehydration process itself. Simply stated, dehydration is taking an in-memory object (such as an instance of a Java class) and converting it into a format suitable for storage in a database, typically a table row. Unless a 3rd party framework is utilized, this often involves writing extensive SQL INSERT or UPDATE statements to persist the object's state. On the flip side, object hydration is the opposite; it is the reconstructing an in-memory object from its stored representation in the database. Again, this requires either a 3rd party framework running SQL SELECT statements and mapping the result set back into the object's fields, which can be complex if the object has relationships with other objects.

With Axon Framework, this is no longer a concern because it automatically manages the conversion of Java objects and their events for storage in Axon Server and the reconstruction of Java objects from events. This means that your Java POJOs (Plain Old Java Objects) are instantiated and persisted when you need them. However, another key benefit of Axon Framework is homogeneity. This simply means that when your microservice receives an event for something that has just

happened (i.e., sent from another microservice via the message bus) or processes an event that has happened in the past (retrieved from the persistent storage of Axon Server), the Java object is the same object.

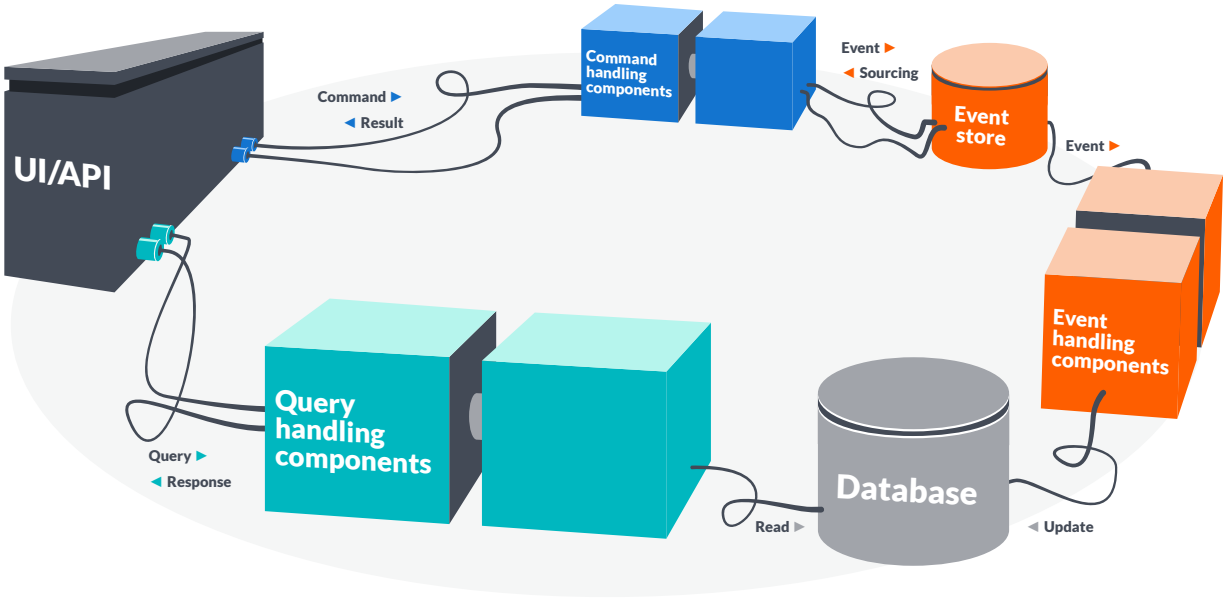
To illustrate this point, consider a microservices architecture for an e-commerce site. When the end user places an order on the website, the website dispatches the "OrderCreated" event. The message bus within Axon Server sends the event to the microservice responsible for processing the orders and receives that event as a Java object. If a manager wants to see a report of past orders, the list of orders needs to be retrieved from the persistent storage within Axon Server. The benefit here is that the Java object for the order is the same object. This is the power of homogeneity.

This is not necessarily the case when developers have to work with a patchwork of frameworks and services for persistence and message routing. This means that the Java object reconstructed from the message router is not the same as the Java object hydrated from the database. This can result in developers creating unnecessary layers of code to translate between their messaging and persistence platforms.

As a result, Axon Framework developers can focus on implementing business logic and domain-specific functionality rather than worrying about the plumbing code to persist and retrieve the application state. The end result is that the codebase becomes cleaner and more maintainable. Moreover, the simplified object lifecycle management offered by Axon Framework enhances the codebase's maintainability. Long-term maintenance becomes less burdensome when it's easier to understand and modify the code. This is especially important for large-scale applications that evolve over time. Teams can onboard new developers more efficiently, and changes can be implemented with confidence, knowing that the underlying infrastructure is robust and reliable.

7. Axon Server: the event store for your events

The Event Store is arguably the most critical component in an event-sourced system. It holds the persistent log of all events, serving as the single source of truth for the application's state. While developers could theoretically use a traditional relational or NoSQL database to store events, as explained previously, these general-purpose solutions are not optimized for the specific needs of event sourcing. An Event Store is efficient because all events are append-only, which is not the case for traditional databases.



Axon Server is an event store designed for developers of any programming language to create event-sourced architectures. If a developer decides to use Java as their programming language, then Axon Server works best with Axon Framework to build applications. However, Axon Server is not limited to Java-based clients and services. Therefore, developers of any programming language (such as Python, JavaScript, Go, C#, Rust, and others) are able to utilize either the HTTP or RSocket interfaces in order to send and receive events (as well as other message types) from the event store.

Axon Server is built with scalability in mind and can handle the demands of modern, distributed applications. It offers features for access control, ensuring that the event store is scalable and secure. These built-in capabilities make it easier for organizations to deploy event-sourced systems in production environments, confident that their data is both safe and accessible. Working hand-in-hand with Axon Framework, it provides the necessary infrastructure to support your event-sourced application.

While Axon Framework takes care of creating and publishing the events for each of your Aggregates (considered a collection of events in your system), Axon Server takes care of receiving and persisting

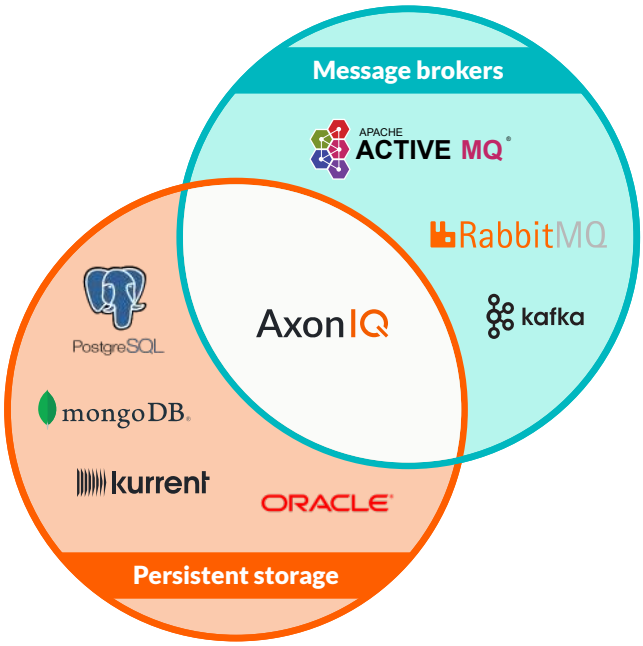
all the events for all the microservices in your entire application. Axon Server functions as a centralized Event Store, ensuring that all events are safely stored and readily accessible for replay and analysis. Its robust architecture supports high availability and scalability, making it suitable for modern enterprise-level applications.

Moreover, Axon Server facilitates seamless communication between microservices in an Axon-based system, simplifying the management of inter-service dependencies and event-driven interactions. By using Axon Server together with Axon Framework, you will have everything you need for an event-sourced application.

The Axon Server advantage: the best of both worlds with dual functionality

As you can see, Axon Server provides dual functionality by acting as a persistent event store and a real-time message router. This dual capability addresses the shortcomings of traditional event-streaming approaches by ensuring that all events are immediately available for real-time processing and securely stored for future reference and analysis.

When you choose a database (like PostgreSQL, MariaDB, or MongoDB) to persist your data, you'll **only** get persistence. As stated before, these databases are great at storing data, but they aren't built for event streaming or real-time messaging. Conversely, if you choose a message broker (like Apache Kafka, RabbitMQ, or ActiveMQ), then you'll **only** get routing – and nothing else. These tools excel at message routing and event streaming but don't offer persistent storage solutions. The intersection in the middle is where Axon Server fits in—uniquely combining both capabilities.



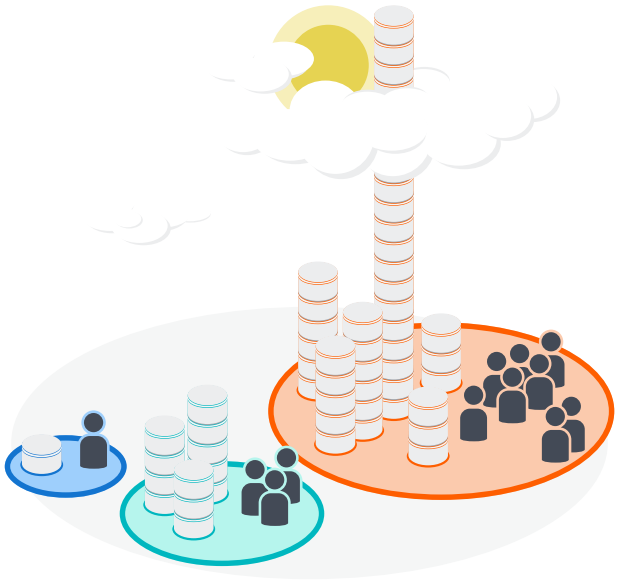
The key benefits of using Axon Server are numerous and compelling. By integrating storage and messaging capabilities into a single platform, Axon Server reduces the complexity of system architectures. This simplification makes it easier for developers to design and build applications and reduces maintenance overhead and potential points of failure, leading to more robust and reliable systems.

Axon Server enhances traceability and debugging processes. In any event-driven architecture, tracing the flow of events and understanding the sequence of actions is crucial for debugging and ensuring system integrity. With Axon Server's comprehensive event logging and persistent storage, developers have a clear and accessible audit trail that facilitates easier debugging and improves

overall system transparency. This feature is particularly beneficial for industries with strict compliance and auditing requirements.

Axon Server contributes to operational efficiencies through unified event management. By handling both the storage of events and their real-time routing, Axon Server streamlines operations and reduces the need for additional tools and integration layers. This unified approach not only cuts operational costs but also enhances the scalability of applications, enabling businesses to manage growing volumes of data efficiently and increasing levels of system interaction.

But is it really feasible to store every event with Axon Server?



For many database and system administrators, one of the first questions that arises when hearing about Event Sourcing is: "Won't we run out of disk space if we store every single event?" At first glance, this concern seems valid. The mere notion of holding an ever-growing collection of events and data may conjure visions of astronomical storage costs or unwieldy data volumes. However, real-world experience with Axon Server and Event Sourcing paints a much more manageable picture.

In practice, the storage consumption for event data is surprisingly modest. In the automotive industry, a global car manufacturer, for example, has deployed Axon Server as an Event Store for 10 separate microservices running in their production computing environment. Over the course of five years, they've accumulated only around 4 TB of event data. Another customer, with fewer overall transactions, has stored no more than 200 GB of event data over a similar five-year period. While exact storage needs will vary based on transaction volume and event size, these statistics underscore that storing every event from the beginning of time for an event-sourced application is feasible and practical.

A key insight is that events themselves tend to be much smaller than their relational counterparts. Instead of multiple columns and dozens of indices, events capture only the critical pieces of information—what changed, when it changed, and sometimes why it changed. Because events are intentionally designed to be as compact as possible, the overall storage footprint remains reasonable, even as history grows. This structure also makes queries more efficient in many cases, as you're not repeatedly writing large data sets for each minor state update.

Additionally, Axon Server employs an optimized architecture for storing and retrieving event data.

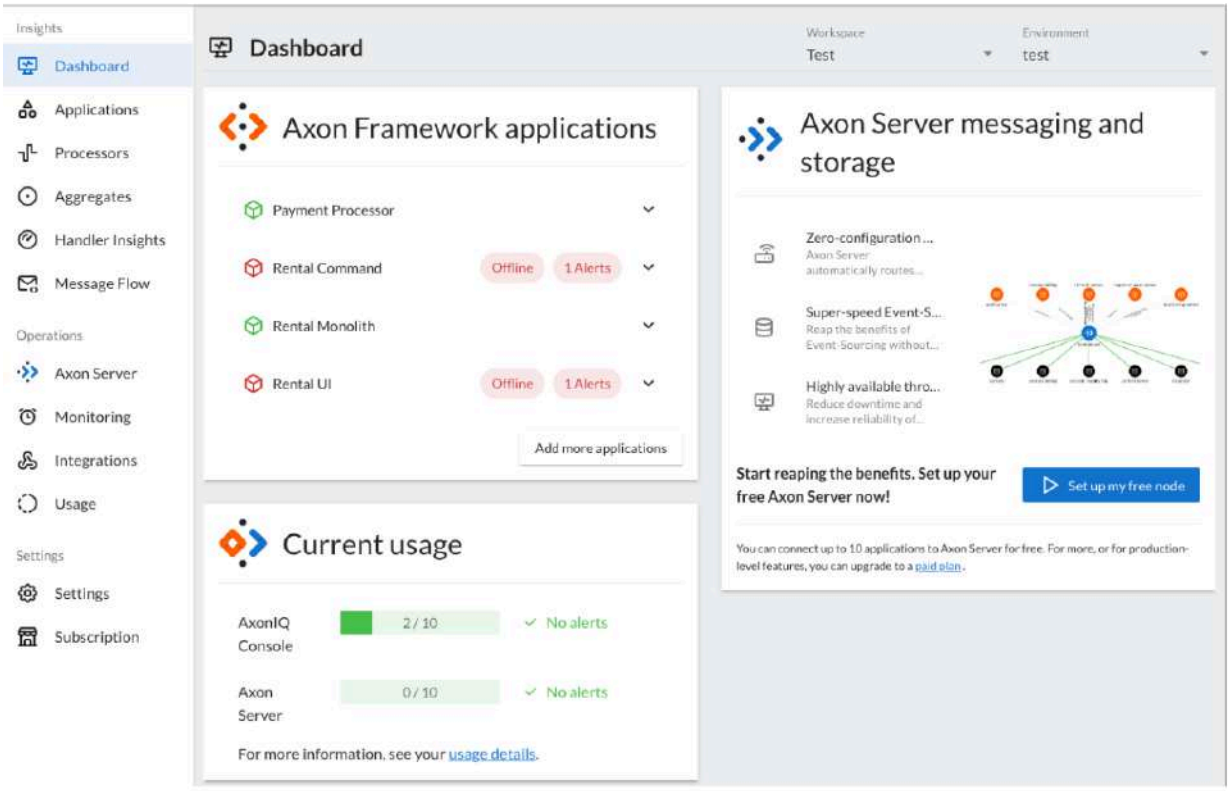
Unlike a general-purpose relational database, Axon Server structures and indexes event streams in a way that favors sequential reads and writes, ensuring that capacity is utilized efficiently. This emphasis on efficient write operations and read optimization helps keep hardware costs lower and performance higher.

Furthermore, we provide our customers with practical strategies to handle fast-growing event stores if and when they occur. Event snapshots, for instance, can store the current state of a system at given intervals, allowing older events to be archived. While the essence of event sourcing encourages keeping your entire event history for maximum insight and traceability, archival solutions remain available.

As a result, storing all events is no longer a daunting, resource-intensive gamble—instead, it's a forward-looking strategy that offers unparalleled visibility into your data's lifecycle. By understanding historical data changes event by event, teams can debug issues more effectively, perform audits with unprecedented accuracy, and gain deeper business insights. Thanks to Axon Server's specialized design and the continuing evolution of storage technologies, your organization can confidently embrace Event Sourcing, secure in the knowledge that disk space constraints need not be a showstopper.

8. Managing your event-sourced system with AxonIQ Console

AxonIQ Console is your central command hub for administering and visualizing the entire lifecycle of an event-sourced architecture. As previously stated, Axon Framework helps alleviate the complexities of boilerplate code and steep learning curves often associated with event-sourced application development. However, once your event-driven microservices run smoothly and feed events into Axon Server, you'll want an easy way to see and manage everything in real time. That's precisely where AxonIQ Console comes in, offering a clear window into your system's inner workings.



In essence, AxonIQ Console ties together the strengths of Axon Framework and Axon Server—two key components of the AxonIQ ecosystem. Axon Framework gives you annotation-based simplicity for building event-sourced microservices. Axon Server serves as the powerful event store and message router where events live and travel. AxonIQ Console then adds an intuitive layer for overseeing event streams, detecting bottlenecks, and ensuring high availability, all without wading through command-line tools or custom scripts.

For developers who appreciate Axon Framework's ability to manage aggregates seamlessly, AxonIQ Console extends that visibility one step further. Instead of diving into logs or writing elaborate

queries, you can inspect and see how your messages and events are routed through a user-friendly interface and navigate through the states they've transitioned between.

Beyond monitoring, AxonIQ Console simplifies the operational side of an event-sourced environment. An event-sourced system's greatest strength—keeping a reliable record of everything that has ever happened—also means you can become buried in a deluge of information if you don't have the right tools. AxonIQ Console helps prevent data overload by providing filters and search features that let you zoom in on specific event types, timestamps, or even user-defined metadata. This focused lens becomes crucial for large-scale applications where hundreds of thousands of events may stream through the system every hour.

AxonIQ Console's dashboards provide a high-level overview of application health, enabling teams to catch anomalies before they spiral into major incidents. Users have a comprehensive view of the entire event-driven pipeline. You can see precisely how Axon Server's dual nature as a persistent event store and real-time router works in concert with Axon Framework's domain-centric approach. Whether you are launching a new microservice, investigating a bug, or doing a routine health check, AxonIQ Console serves as your one-stop shop to ensure every event in your system is accounted for, routed to the right destination, and stored safely for future replays.

9. Event sourcing sounds a little challenging; what is the best way to quickly start my project?

AxonIQ is the company known for “making event sourcing easy”. We have a simple, straightforward, and repeatable process that allows companies to quickly get started with event sourcing. For example, for developers, we have tutorials, guides, and example code located at our Documentation website, which greatly reduces the learning curve in order to make the learning process easy.

We also understand that when starting any new project, modeling the system itself is one of the first steps involved. This is where the product owners and stakeholders specify the features and functionalities that they need within the desired software system. After gathering your requirements, you need to create an “Event Model”. If you're unfamiliar with the term, then think of it as a "data model" for your event-sourced application. The idea here is that you want to list all the events in your system, as well as the processes that change the data in your system (called Commands), and the components that show the status of your data (called Queries).

The best way to learn the process about creating an event model is to checkout AxonIQ Playbook. AxonIQ Playbook shows developers the hand-on patterns and best-practices needed for modeling, implementing, and operating event-sourced systems. Proper system design relies on a solid understanding of your business domain. AxonIQ Playbook also shows you how to take your model and implement it in code with Axon Framework and scale your system with Axon Server.

10. Embrace the future of data persistence today

Event sourcing is a better way to persist data, and AxonIQ provides the tools to make it work at scale. The combined power of Axon Framework and Axon Server offers a unified, simplified approach to building robust, scalable, and maintainable applications. Together, they eliminate the challenges of traditional data persistence and standalone event streaming, providing a seamless solution for modern architectures.

If you're ready to dive in and get started with event sourcing, we have multiple options for you to choose from:

Developers can log in and get a free account with AxonIQ Console. Here, you can deploy a sample project with Axon Framework and use the free "Developer Plan" to get an instance of Axon Server to deploy and test their applications and services.

Our documentation portal provides comprehensive guides, tutorials, and reference materials to help you understand and use Axon Framework and Axon Server effectively.

For those who prefer a more interactive learning experience, we regularly host webinars and training sessions, and we have hours of training courses available within AxonIQ Academy. These events are a great way to learn from the experts and get hands-on experience with the Axon ecosystem.

We created AxonIQ Playbook to show developers how to easily model, code, and deploy an event-sourced application with AxonIQ technologies.

Finally, don't forget to explore the AxonIQ community. Whether it's on GitHub, Stack Overflow, or the AxonIQ forums, you'll find a vibrant community of developers ready to share their knowledge and help you on your AxonIQ journey. Happy coding!

AxonIQ

Making Event Sourcing easy

Axoniq.io

youtube.com/c/AxonIQ

github.com/AxonIQ