Kolumne: EnterpriseTales

Was ich schon immer zu Event Sourcing wissen wollte ...

von *Andreas Willems*

… mich aber nicht zu fragen getraut habe. Der traditionelle Weg für Persistenz erlaubt das Speichern des letzten Zustands einer Entity. Wenn ich aber daran interessiert bin zu wissen, wie vorherige Zustände ausgesehen haben, reicht das Vorhandensein des letzten Zustands nicht aus. Event Sourcing löst dieses Problem und erlaubt es, theoretisch beliebig weit in die Vergangenheit einer Entity zu schauen. Das geht jedoch mit einer erhöhten Komplexität der Software einher und erfordert ein Umdenken bei verschiedenen Aspekten, die ich im Folgenden beleuchte.

Was ist Event Sourcing?

Die für mich übliche Herangehensweise an Persistenz ist, eine Instanz eines Objekts zu erzeugen und dann mit Hilfe eines ORM in eine Datenbank zu schreiben. Bei Bedarf wird die Instanz genauso wieder geladen, ihr Zustand verändert und der geänderte Stand zurückgeschrieben. Das reicht mir, wenn mich immer nur der letzte Zustand interessiert.

Wenn ich aber so etwas brauche wie eine Versionierung oder eine Historie meiner Entity, also die Möglichkeit, vergangene Zustände zu betrachten, komme ich mit diesem Ansatz nicht weiter. Diese Funktionalität lässt sich zwar auch im traditionellen Aufbau erreichen, etwa indem ein Versionsfeld eingefügt wird; das hat dann aber meist den Beigeschmack einer gebastelten und nur unzulänglichen Lösung.

Auftritt Event Sourcing: Event Sourcing ist ein Muster, das jede Zustandsänderung in Form von Ereignisobjekten bzw. Events kapselt und in einer Liste speichert [1]. Nicht der Zustand einer Entity zu einem Zeitpunkt wird gespeichert, sondern durch das „Abspielen“ aller bisherigen Events wird der Zustand berechnet. Dieses Vorgehen wird auch Rehydrieren genannt. Wird jedes Event mit einer eindeutigen ID, einer Sequenznummer und einem Zeitstempel versehen, lässt sich so ein exakter Zeitpunkt auswählen, zu dem der Zustand bestimmt werden soll.

Welche Anforderungen ergeben sich hierdurch konkret an meine Software? Zur korrekten Darstellung des Zustands ist es erforderlich, dass jedes Ereignis zu einer Entität unveränderlich gespeichert wird. Die Ereignisobjekte werden in einem sog. Append-only-Log gespeichert. Das ist im Grunde eine Liste, an die neue Objekte immer hinten angehängt werden, wobei die vorherigen Einträge nicht geändert werden können. Das Zusammenspiel von unveränderlichen Events und dem Append-only-Log ermöglicht es, den Zustand meiner Entität von jedem Zeitpunkt wieder zu errechnen.

Dieses Konzept führt aber gleichzeitig zu unserem ersten Problem: Was passiert, wenn diese Liste immer größer wird und die Anzahl der Ereignisse für meine Entity immer weiter steigt? Dauert die Berechnung des aktuellen Zustands dann nicht potenziell sehr lange?

Snapshots in Event Sourcing

Es ist leicht nachvollziehbar, dass das Laden und Verarbeiten einer beliebig großen Anzahl von Ereignissen prinzipiell aufwendiger ist als das Laden eines einzelnen Zustands, weil erst alle Einträge aus der Liste geladen werden müssen, um sie dann der Reihe nach auf die Entity anzuwenden. Um die Anzahl der Ereignisse, die für meine Berechnung notwendig sind, klein zu halten, gibt es beim Event Sourcing verschiedenen Optionen. Die auf den ersten Blick naheliegende Lösung ist, den Zustand meiner Entität ebenfalls zu speichern. Das kann so weit gehen, dass nach jedem Event der aktualisierte Zustand meiner Entität ebenfalls persistiert wird. Diese separat gespeicherten Zustände werden auch als Snapshots bezeichnet.

Das Speichern von Snapshots stellt jedoch lediglich eine Optimierung in Bezug auf die Performance beim Lesen der Entity dar und sollte nicht voreilig in Betracht gezogen werden. Ein Snapshot sollte nur dafür angelegt werden, um bei eintreffenden Operationen den letzten Zustand der Entity schnell zur Verfügung zu haben und hierauf Geschäftsregeln anwenden zu können. Er dient nicht dazu, für andere Zwecke missbraucht zu werden, etwa als Lese-Model (siehe unten). Speichere ich den Zustand in einem Cache, sei es in-memory oder extern, so muss ich mich um Dinge wie Cache-Invalidation kümmern. Verwende ich eine eigene Datenbank für die Snapshots, muss ich mir Gedanken um die entsprechenden Mechanismen, das Deployment sowie um Schemamigration machen. Snapshots sollten daher wirklich nur dann zum Einsatz kommen, wenn es die Anforderungen an die Performance unumgänglich machen. Der verfrühte Einsatz kann eher auf ein Problem im Design des Models hindeuten.

Mit der heutzutage verfügbaren Rechenleistung und Speicherkapazität bedarf es schon komplexerer Set-ups, in denen das Abspielen einer Vielzahl von Ereignissen wirklich zum Flaschenhals einer Anwendung wird. Die Datenbanken, die für das Speichern der Events verwendet werden, werden auch als Event Store bezeichnet. Die Technologien dahinter können sowohl „traditionelle“ RDBMS wie Postgres sein, die sich seit Jahrzehnten in der Praxis bewährt haben, als auch dedizierte Technologien wie EventStoreDB. Besonders Mutige zweckentfremden auch Kafka als Event Store ... Seufz ...

Dadurch, dass die Events meist relativ kleine Datensätze ohne Relationen sind, können sie sehr schnell von den Event Stores geladen werden, sodass der Overhead im Vergleich zum Auslesen einer Entität nicht sonderlich groß ist. Eine Frage, die sich hierbei stellt, ist: Was passiert mit den Events, die als Teil des Snapshots verarbeitet werden? Muss ich sie für immer behalten?

Im Grundsatz sind Events unveränderlich und müssen vollständig sein. Das gilt aber nur während des konkreten Einsatzes in der Anwendung. Solange Events eine Aussagekraft für den aktuellen Zustand in meiner Anwendung haben, muss ich sie auf jeden Fall behalten. Erst, wenn die dazugehörige Entity nicht mehr relevant ist, weil etwa ihr Lebenszyklus sein Ende erreicht hat, kann man sich Gedanken über das Aufräumen von Events machen. Das kann z. B. dadurch passieren, dass ich das Lebensende durch ein sogenanntes Tombstone Event markiere, das abhängige Komponenten dazu veranlasst, ihre Datenbestände anzupassen. Je nach Anwendung können die entsprechenden Events archiviert und schließlich gelöscht werden.

Eine vermutlich bessere Alternative zu Snapshots ist es, die Anzahl der Ereignisse, die für die Ermittlung des Zustands erforderlich sind, auf einen Zeitraum zu beschränken. Hierzu ist es sinnvoll, bei der Betrachtung der Ereignisse einen Schritt zurückzugehen und zu untersuchen, ob es in der Domäne natürliche Start- und Endpunkte für meine Liste von Ereignissen gibt. Bei einer Registerkasse zum Beispiel wird am Ende eines Tages der Bestand an Bargeld gezählt, mit dem Bestand im Kassensystem abgeglichen und schließlich als ein Stand aufgezeichnet.

Dieser Stand gilt dann für den kommenden Tag als der Beginn der Aufzeichnung. Hierbei ist es denkbar, den Kassenabschluss ebenfalls als ein Ereignis zu definieren, mit dem Kassenbestand als Payload. Dieses Ereignis bildet dann das letzte Ereignis in der Liste für den vergangenen Tag und gleichzeitig das erste Ereignis für die separate Liste des folgenden Tages.

Der Unterschied zwischen Snapshots und mehreren Listen von Events (auch Streams genannt) ist in Abbildung 1 dargestellt.

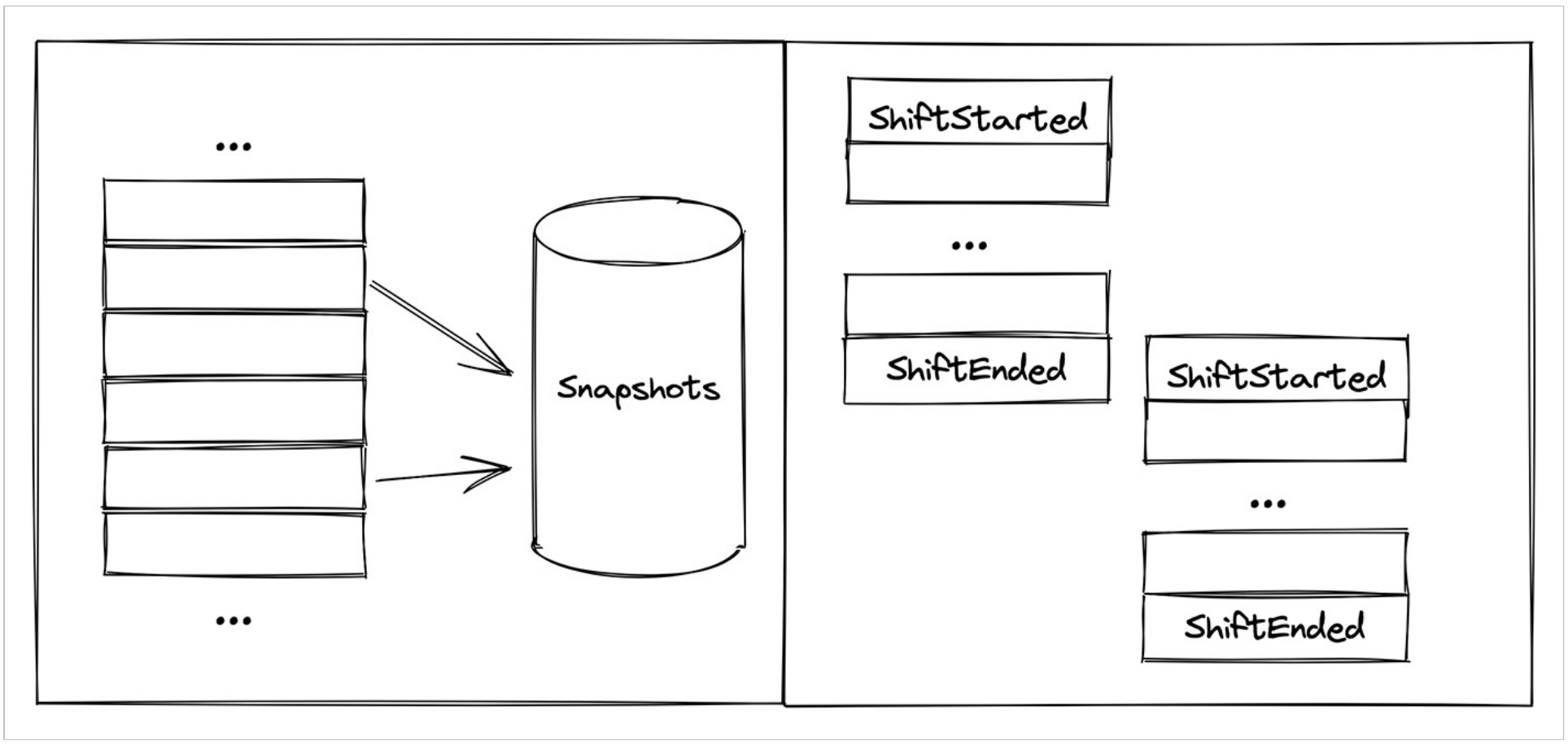

Abb. 1: Unterscheidung zwischen Snapshots und mehreren Streams

Durch dieses Vorgehen wird die Anzahl der Ereignisse, die für die Ermittlung eines Zustands notwendig sind, auf die Ereignisse für einen definierten Abschnitt (hier einen Arbeitstag) begrenzt. Den validierten Kassenbestand ebenfalls als Ereignis zu beschreiben, hilft, auf weitere Mechanismen zur Speicherung zu verzichten und stattdessen in einem Muster zu bleiben.

Queries in Event Sourcing

Das zweite Problem, das mir bei der Beschäftigung mit Event Sourcing bewusst geworden ist, hat mit Querys zu tun. Wie mache ich Querys beim Einsatz von Event Sourcing? Gar nicht! Zumindest nicht ohne Umwege. Bei reinem Event Sourcing ist der Zustand einer Entität nicht einfach irgendwo gespeichert, sondern muss erst durch die Anwendung der Events auf die Entität errechnet werden. Möchte ich z. B. die Summe aller Bestellungen in meinem System erfahren, müsste ich zuerst alle gespeicherten Events verarbeiten, daraus die Summen ermitteln und anschließend zusammenrechnen. Das kann offensichtlich nicht die beste Lösung sein.

Eine in der Praxis verbreitete Herangehensweise beim Event Sourcing ist es, für die lesenden Zugriffe (Queries) ein eigenes Datenmodell zu pflegen, das sog. Read Model. Um die Daten für das Read Model zu speichern, registriert sich die Anwendung auf die neu geschriebenen Events und wertet diese für Zwecke des lesenden Zugriffs aus. Das hat den Nachteil, dass jetzt mehrere Modelle gepflegt werden müssen, während man im traditionellen Ansatz mit nur einem Modell auskommt. Allerdings macht diese Trennung von schreibenden und lesenden Modellen die unterschiedlichen Verantwortlichkeiten und Funktionalitäten der Modelle explizit.

Im traditionellen Ansatz wird das Modell bzw. die repräsentierende Klasse für die verschiedenen Funktionalitäten zurechtgebogen und arbeitet ggf. auf einem nur dürftig normalisierten Datenmodell, um alle Anforderungen zu erfüllen. Mit der Trennung der Modelle ist es möglich, die schreibenden Operationen auf einem weitgehend normalisierten Modell auszuführen, während die lesenden Operationen auf dem für die jeweiligen Zwecke denormalisierten Modell arbeiten können. In Abbildung 2 ist die Trennung der Modelle graphisch dargestellt.

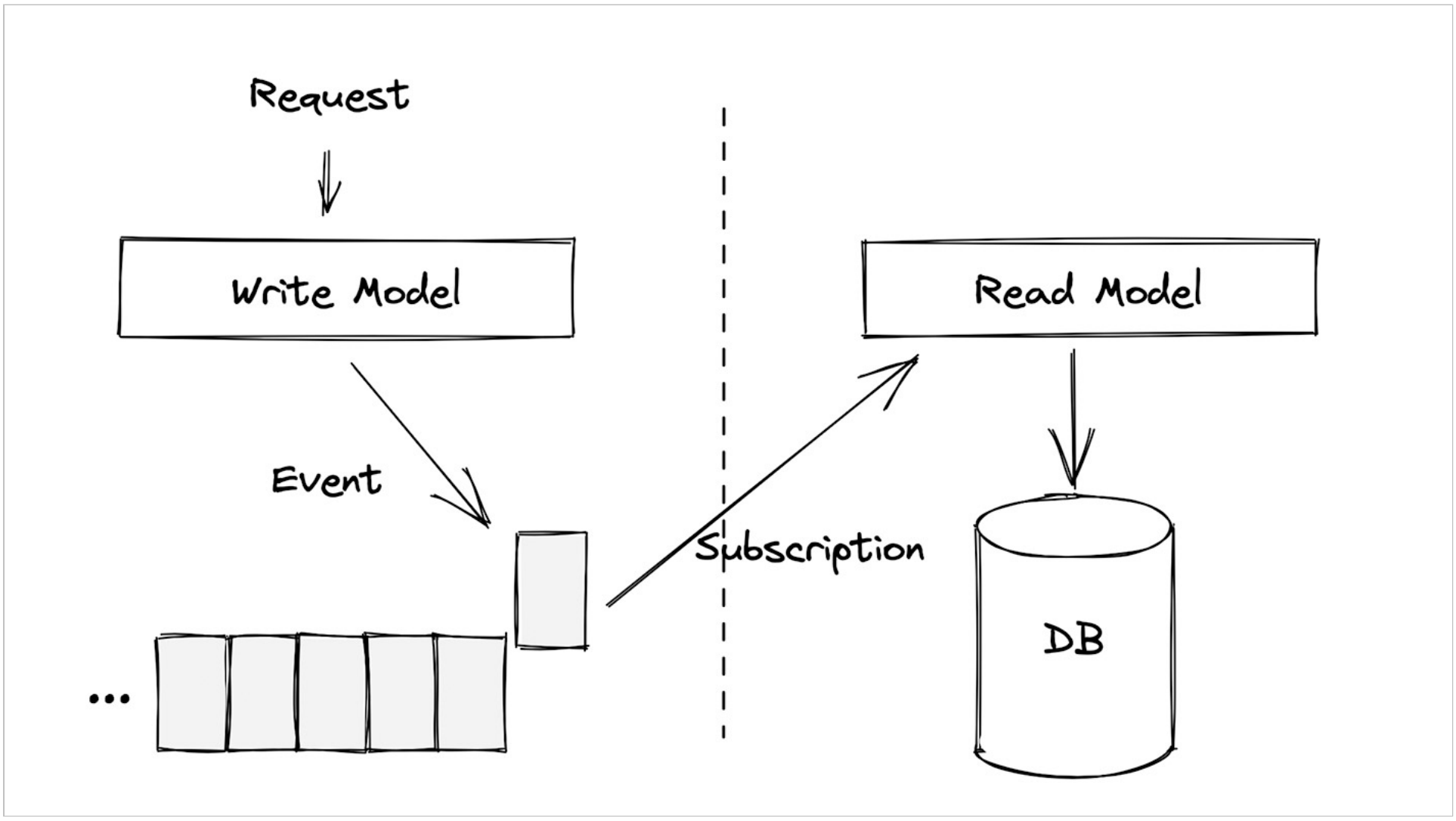

Abb. 2: Trennung von schreibendem und lesendem Modell

Die Abbildung zeigt auf der linken Seite das schreibende Modell (Write Model). Geht eine Operation, hier in Form eines Request, ein, wird diese vom Write Model verarbeitet und führt zu einem Event, das in den Event Store geschrieben wird. Das lesende Modell (Read Model) wird über das Hinzufügen der Events informiert und kann die Events entsprechend dem eigenen Modell verarbeiten und je nach Bedarf Änderungen im Datenbestand durchführen.

Die Trennung der Modelle bietet weiter die Möglichkeit, die Anwendung in zwei Teilsysteme aufzuteilen und für sich zu deployen. Das kann dabei helfen, Probleme in der Skalierbarkeit zu beheben, wenn z. B. die schreibenden Operationen einen wesentlichen größeren Anteil an der Nutzung haben als die lesenden Operationen. Diese Trennung bis hin zu unterschiedlichen Persistenztechnologien muss dann natürlich mit Eventual Consistency zurechtkommen, womit ganz eigene Probleme einhergehen.

Die grundlegende Trennung von schreibenden und lesenden Modellen ist auch als Command Query Responsibility Segregation (CQRS) bekannt. Dieses Konzept näher zu beschreiben, führt aber an dieser Stelle zu weit, weswegen ich hier nicht weiter darauf eingehe.

Koordinierung von Schreibzugriffen in Event Sourcing

Eine weitere Frage, die sich bei der Beschäftigung ergeben hat, hat mit Nebenläufigkeit zu tun: Was passiert eigentlich, wenn zwei Operationen zeitgleich auf eine Entity treffen? Die Situation kann z. B. dann eintreten, wenn ein Request eintrifft, während die Entity noch mit der Arbeit einer anderen Operation beschäftigt ist. Oder noch schlimmer: Die Anfragen kommen wirklich zeitgleich an.

Der übliche Ablauf beim Verarbeiten eines Request ist, den eingehenden Request in die Anwendung weiterzuleiten, die betreffende Entity zu laden und anhand der bisherigen Events den aktuellen Zustand zu errechnen. Anschließend wird die Operation auf der Entity durchgeführt. Entsprechen die Änderungen den Geschäftsregeln, werden die Änderungen in Form von Events gespeichert und der Vorgang ist beendet. In dieser naiven Form kann es passieren, dass eine Operation auf eine Entity trifft, deren aktueller Zustand eigentlich gar nicht dem Regelwerk entspricht (z. B. die Operation „Geld abheben“ auf einem Kontostand, der durch eine zeitgleiche Operation auf null Euro reduziert wurde.)

Um Inkonsistenzen zu vermeiden, müssen geeignete Mechanismen eingerichtet werden. Eine Möglichkeit ist dabei der Einsatz von Locking-Mechanismen, wie z. B. Optimistic Locking. In dem o. g. Ablauf wird beim Laden und Rehydrieren der Entity die aktuelle Version zwischengespeichert. Anschließend werden die Änderungen anhand der Operation durchgeführt und die resultierenden Events gesammelt. Bevor diese Events gespeichert werden, erfolgt eine Prüfung, ob die aktuelle Version noch mit der Version vor Beginn der Operationen übereinstimmt. Ist die Version gleich, werden die Events persistiert. Hat sich die Version jedoch in der Zwischenzeit geändert, weil ein anderer Request verarbeitet wurde, so werden die gesammelten Events verworfen, ggf. eine Exception geworfen und die blockierte Operation erneut versucht.

Event Sourcing: Fazit

Event Sourcing ist eine Möglichkeit, um Eigenschaften wie die Versionierung von Entities oder die Trennung von lesenden und schreibenden Modellen in einer Anwendung zu erlangen. Das erfordert jedoch ein Umdenken, wenn man bislang eher den traditionellen Weg gegangen ist. Gerade für den Anfang erfordert die Implementierung von Event Sourcing, meist in Verbindungen mit CQRS, einiges mehr an konzeptioneller Arbeit und Code. Mit zunehmender Größe einer Anwendung gewinnt man hierdurch aber an Deutlichkeit und Struktur. Wenn man es daher schafft, die hier beschriebenen und auch weitere Probleme von Event Sourcing in den Griff zu kriegen, ist es ein probates Mittel, seine Anwendung aufzubauen.

Links & Literature

[1] https://de.wikipedia.org/wiki/Event_Sourcing