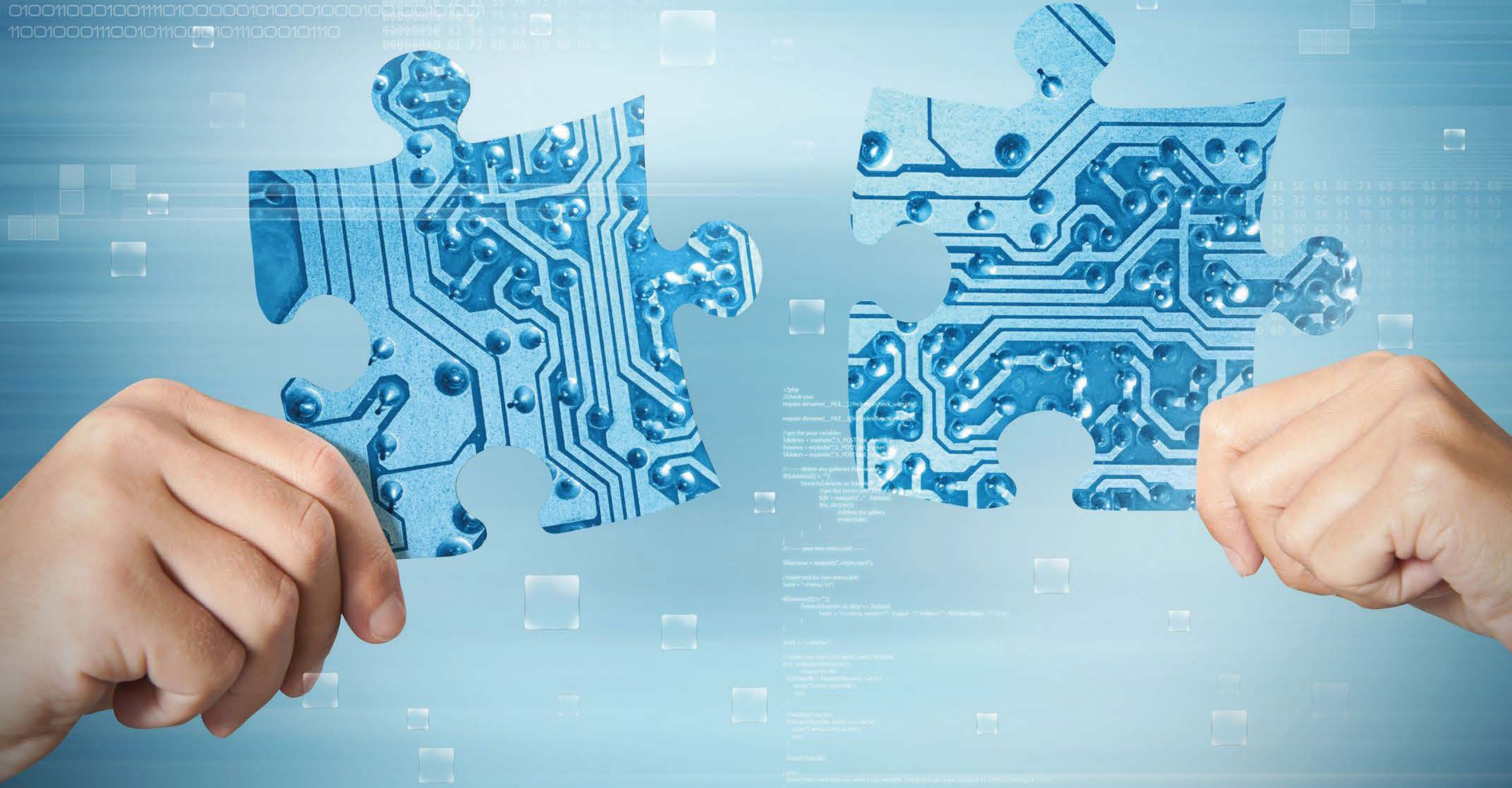

CQRS und Event Sourcing mit dem Axon Framework

Christian Iwanzik, tarent solutions GmbH

Klassische CRUD-Modelle kommen in puncto Erweiterbarkeit schnell an ihre Grenzen. CQRS gepaart mit Event Sourcing kann in solchen Fällen ein gutes Werkzeug sein, um aus der Komplexitätsfalle zu kommen. Exemplarisch mit dem Axon Framework umgesetzt, zeigt der Artikel neben der Theorie auch ein praktisches Beispiel.

Naiver Ansatz mit CRUD

Moderne Software basiert oft auf iterativen Modellen, um auf sich ändernde Anforderungen schnell und kurzfristig reagieren zu können. Daher sind die ersten Umsetzungen von Software oft mit einer einfachen CRUD-Architektur versehen, die schnell Resultate bringt. CRUD (Create, Read, Update, Delete) beschränkt sich auf eine zentrale Datenstruktur, für die die vier Operationen in einer Serviceklasse implementiert werden. Zusätzliche Controller und Repositories sorgen für die Serialisierung und Persistierung.
In *Abbildung 1* sehen wir eine klassische Kundenverwaltung. Die Customer-Entität wird für alle Belange der Software herangezogen. Customer, inklusive Adressen, werden immer ganz gelesen und bei Änderungen komplett überschrieben.

Allerdings erweitern sich die Anforderungen erwartungsgemäß sehr schnell und die vier Operationen reichen bald nicht mehr aus. So kann das System durch eine Zweifaktor-Registrierung per Mail erweitert werden, um Fake-Registrierungen entgegenzuwirken. Eine Analystin hätte zusätzlich gerne eine Abfrage über noch nicht bestätigte Kunden, um die Conversion Rate auszurechnen. Allein damit haben wir nun aber bereits einen ersten Prozess etabliert, dem unsere Architektur nicht mehr gewachsen ist.

1. Wer verschickt die E-Mail bei der Registrierung?
2. Wie muss eine Customer-Entität aussehen, die einmal einen Zustand eines nicht bestätigten Accounts darstellt und später den eines bestätigten Accounts?
3. Wo wird ein Bestätigungstoken gespeichert?
4. Ist eine E-Mail-Bestätigung ein Update der gesamten Entität?

So kann man sich schnell immer mehr Erweiterungen vorstellen. Kunden sollen ihre E-Mail-Adresse über denselben Zweifaktor-Mechanismus ändern können. Gleichzeitig sollen jedoch während des Prozesses etwaige E-Mails noch an die alte Adresse gesendet werden. Die Entität bläht sich schnell mit optionalen und Null-

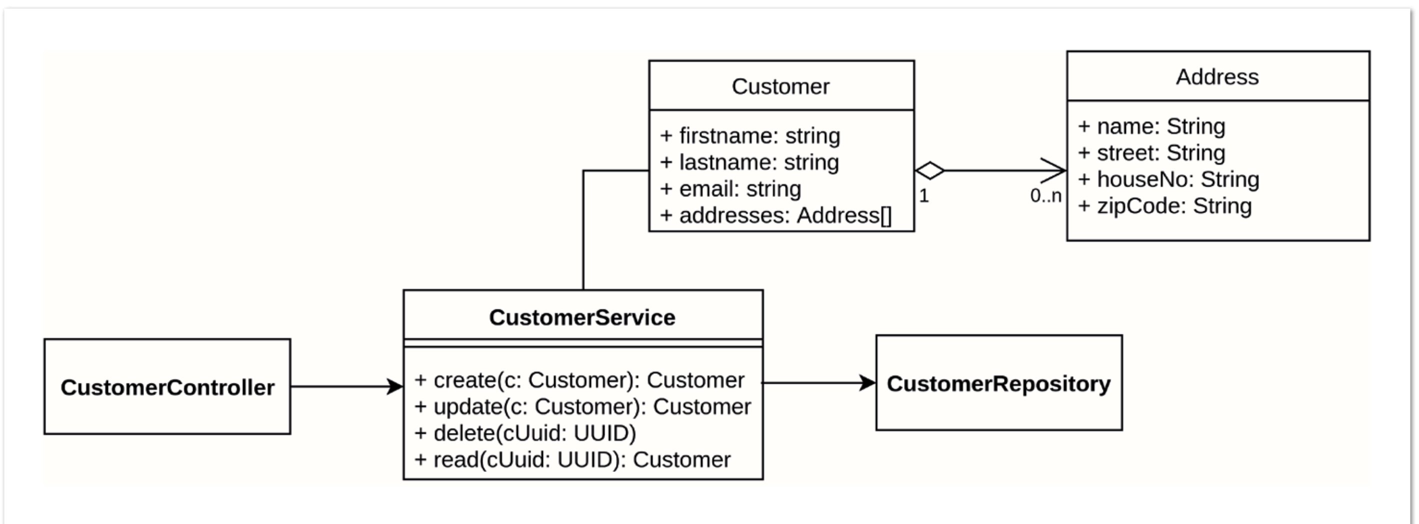

Abbildung 1: Simple CRUD-Architektur

Attributen auf, die eventuell genau einmal in ihrem Lebenszyklus für einen speziellen Prozess ausgewählt werden.

Zu allem Überfluss lagern sich Prozesse und Businesslogik in verschiedene Serviceklassen aus und haben nicht mehr den Zusammenhalt mit der eigentlichen Entität. Das Ergebnis eines solch unkontrollierten Wachstums ist der sogenannte „Big Ball of Mud" *[1]*. Also ein System chaotischer Beziehungen. *Abbildung 2* zeigt ein reales Beispiel einer Komponente, die mit CRUD gestartet ist und sich mit immer weiteren Anforderungen konfrontiert sah. Ein Refactoring auf CQRS konnte hier im Nachhinein einen Großteil der Komplexität eliminieren.

CQRS und Axon

CQRS steht für „Command Query Responsibility Segregation" *[2]*. Indem das System ändernde und lesende Zugriffe auf Daten hart voneinander trennt, können diese besser auf die Anforderungen reagieren und sind dabei leichter erweiterbar. Auch werden per Commands nicht mehr alle Daten überschrieben, sondern Änderungswünsche sauber von der eigentlichen Verarbeitung getrennt.

Abbildung 3 beschreibt eine grobe Skizze der Architektur. Im Zentrum steht immer das Domain Model. Es ist der eigentliche Kern der Software und beschreibt alle für die Software geschäftsrelevanten Daten und Methoden. Also den Grund, aus dem die Software überhaupt entwickelt wird. Das Model bietet nach außen Methoden an, über die Änderungsbefehle, sogenannte Commands, übergeben werden können. Diese Commands können von den Domain-Methoden entweder angenommen oder abgelehnt werden. Angenommene Commands erzeugen erst dann eine Zustandsänderung der Daten.

Commands selbst sind kleine Datenpakete, die so einen Befehl repräsentieren. Sie werden durch den Command Handler angenommen und in Aufrufe für das Domain Model übersetzt.

Ist innerhalb des Domain Model eine Zustandsänderung passiert, so schickt die Domain ein Business-Event in eine Event Queue. Diese Events wiederum werden von einem Event Handler in ein Read Model überführt, das einen speziellen Aspekt der Domain repräsentiert und für Abfragen bereitsteht.

Das Axon Framework

Das Axon Framework ist eine in Java geschriebene Bibliothek für die JVM, welches das CQRS Pattern umsetzt. Die folgenden Beispiele beschreiben das Framework in der Version 3.4 in Kombination mit Spring Boot 2.1.2. Als Sprache wurde Kotlin gewählt.

Die Einbindung in den eigenen Java-Code ist simpel über Maven oder Gradle Dependencies möglich. Eine Interoperabilität mit Spring Boot ist über die „axon-spring-boot-starter"-Dependency bereits vom Framework vorgesehen.

Commands

Am Anfang einer Datenveränderung stehen immer Commands. Dies sind kleine Datenobjekte, die ein bestimmtes Änderungsprädikat darstellen. In *Listing 2* sehen wir ein „CreateCustomer"-Command. Dieses beinhaltet alle Daten, um einen Customer initial anzulegen. Aber eben auch nicht mehr.

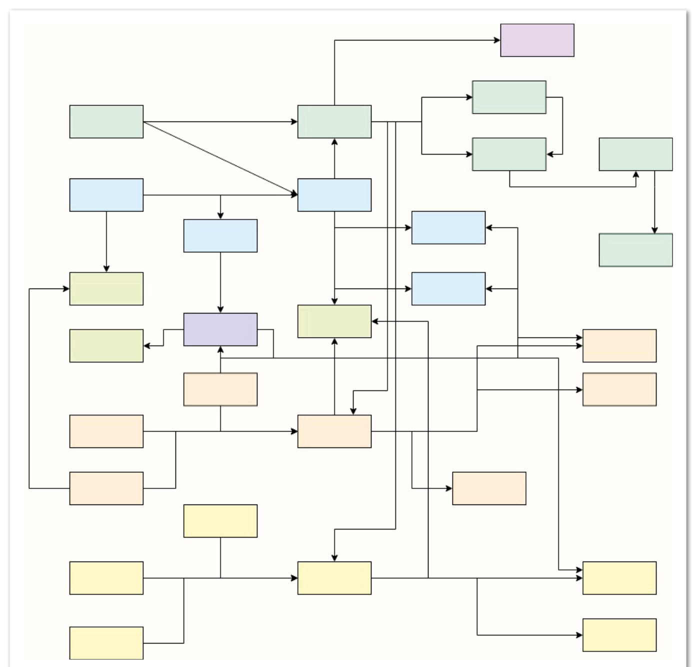

Abbildung 2: Reales Beispiel eines Big Ball of Mud

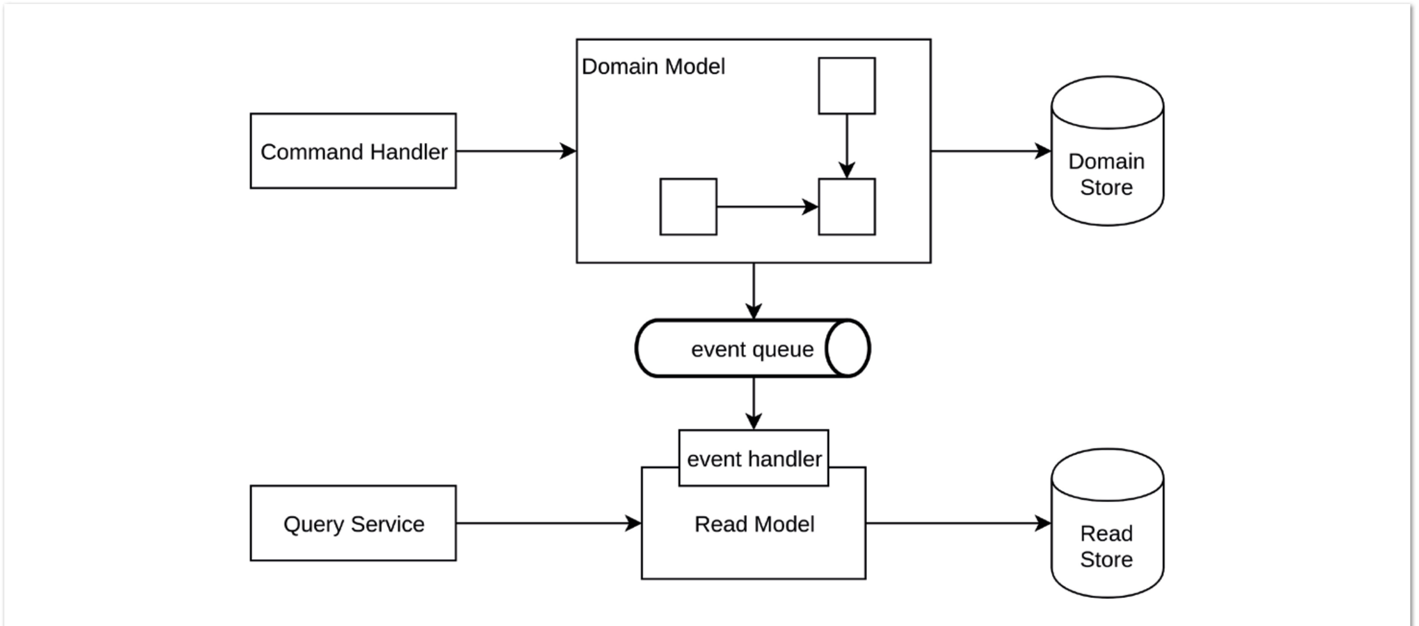

Abbildung 3: Übersicht CQRS

Anders als bei CRUD modellieren wir hier nicht die komplette Entität, die überschrieben werden soll, sondern nur die zu ändernden Elemente. Der angelegte Customer könnte später noch eine Adresse hinzubekommen. Die Adresse würde dann aber mit einem AddAddress-Command hinzugefügt werden, das nur die Daten der Adresse beinhaltet. Commands sind also keine direkten Zustandsbeschreibungen, sondern erst mal nur Absichtserklärungen an das Domain Model. Um diesem Sachverhalt mehr Ausdruck zu verleihen, sollten Commands im Namen immer ein Prädikat gefolgt vom Namen der zu ändernden Entität beinhalten. Als Quelle von Commands können alle eingehenden Datenquellen auftreten. Entweder ein UI oder REST Controller oder ein Message Subscriber.

```
@RequestMapping(value = ["/customer"], method = arrayOf(POST))
fun createCustomer(@RequestBody command: CreateCustomer): ... {
  val result = DeferredResult<Any>()

  // Sending the command into the CQRS system.
  val future: CompletableFuture<UUID> =
     commandGateway.send<UUID>(createCustomerCommand)

  // Return the uuid, when the command handling is finished.
  future.exceptionally { err → … }
     .thenAccept { customerUuid -> ...}

  return result
}
```

Listing 3: Ein REST-Endpunkt

```
@CommandHandler
fun createCustomer(customer: CreateCustomer): UUID {
  return repository
      .newInstance {
         Customer(
            firstname = FirstName( customer.firstname),
            lastname = LastName(customer.lastname),
            email = UnverifiedEmail(customer.email, ...)
         )
      }
      .identifier() as UUID
}
```

Listing 4: Ein Command Handler

Listing 3 zeigt als Beispiel einen Spring-MVC-REST-Endpunkt. Das Command kann hier direkt zur JSON-Deserialisierung benutzt werden. Der Einstieg in das Axon Framework geschieht über das „command-Gateway", das als Spring Bean direkt über die Axon Dependencies erzeugt wurde. Das Command Gateway übernimmt alle Commands und vermittelt sie an Command Handler weiter. Diese sind simple Spring Beans, die aber Methoden besitzen, die mit @CommandHandler annotiert sind. Die Registrierung geschieht automatisch.

Listing 4 zeigt den Command Handler, der das Anlegen eines neuen Customer übernimmt. Das hier erwähnte „repository" ist ebenfalls eine Dependency, die von Axon gestellt wird, um das Domain Model zu persistieren. Im weiteren Verlauf erzeugt der Handler ein neues Domain Model und übergibt es dem „repository". Andere Handler würden zunächst das Model mithilfe des „repository" laden und darauf eine Methode anwenden. Listing 5 zeigt dies anhand einer Adresse.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.2.RELEASE</version>
</parent>

<dependencies>
  <dependency>
     <groupId>org.axonframework</groupId>
     <artifactId>axon-spring-boot-starter</artifactId>
     <version>3.4.2</version>
  </dependency>
  …
```

Listing 1: Maven Dependencies für Axon

```
data class CreateCustomer (
  val name: String,
  val surname: String,
  val email: String
)
```

Listing 2: Ein simples Command

Das Domain Model

Die Domain ist das eigentliche Herz der Software. Sie beschreibt unabhängig von Datenbankschema oder technischen Abhängigkeiten die fachliche Gesamtheit des zu lösenden Problems. Also Daten und Methoden, um diese zu ändern. Wichtig ist hier, dass Daten stark in der Domain gekapselt sein müssen. Änderungen werden lediglich durch Methoden der Domain durchgeführt. Diese haben die Aufgabe, von außen eingegebene Änderungswünsche zu validieren und gegebenenfalls mit einem Fehler abzuweisen. Änderungswünsche, die den Anforderungen entsprechen, wie zum Beispiel Pre- und Postconditions innerhalb der Domain, führen dann zu einer Zustandsänderung der Domain. Wichtig ist aber, dass nur die Domain darüber entscheidet, ob sie die Änderung durchführt oder nicht.

```
@CommandHandler
fun addAddress(addrCmd: AddAddress) {
  repository
     .load(addrCmd.customerUuid.toString())
     .invoke { customer ->
        val address = Address.fromCommand(addrCmd)
        customer.addAddress(address)
     }
}
```

Listing 5: Ein Command Handler auf einem bestehenden Model

```
@Aggregate
class Customer() {
  @AggregateIdentifier
  private var customerUuid: UUID = UUID.randomUUID()
  private var addresses: List<Address> = listOf()
  private lateinit var firstname: FirstName
  private lateinit var lastname: LastName
  private lateinit var email: Email

  constructor(...) {
     this.firstname = firstname
     this.lastname = lastname
     this.email = unverifiedEmail
     AggregateLifecycle.apply(CustomerCreated(...))
  }

  fun addAddress(address: Address) {
     this.addresses = addresses + address
     AggregateLifecycle.apply(AddressAdded(...))
  }
...
```

Listing 6: Das Customer Domain Model – in Axon: Das Aggregat

```
data class CustomerCreated(
  val customerUuid: UUID,
  val name: String,
  val surname: String,
  val email: String
)
```

Listing 7: Das CustomerCreated-Eventobjekt

```
@Component
class AddressAggregator() {
  private val allAddresses: MutableList<Address> = ...
  @EventHandler
  fun handleAddressAdded(event: AddressAdded) {
     allAddresses + event.address
  }
  fun getAllAddresses(): List<Address> {
     return allAddresses.toList()
  }
}
```

Listing 8: Ein Event Handler am Beispiel eines Address-Aggregators

Listing 6 zeigt nun das Customer Domain Model. Wichtig zu wissen ist, dass das Domain Model bei Axon „Aggregate" genannt wird. Das Aggregat existiert nicht als Bean im System, sondern wird immer vom Repository geladen und instanziiert beziehungsweise von einem Command Handler erzeugt. Zur Serialisierung durch Axon werden lediglich die beiden Annotationen „@Aggregate" und „@AggregateIdentifier" benötigt.

Domain Events

Sich ändernde Domains beziehungsweise Aggregate werden in der Folge ein Domain Event emittieren, um der „restlichen Welt" zu sagen, dass sich etwas geändert hat. Domain Events lassen sich auf zwei grundlegende Weisen beschreiben. Als idempotente Objekte, die den gesamten augenblicklichen Zustand der Domain beschreiben, oder als Deltaobjekte, die nur die Änderung der Domain darstellen.

Auf Code-Ebene unterscheiden sich Commands und Events nicht sonderlich. Allerdings haben sie eine grundlegend andere Semantik. Während ein Command eine Änderung herbeiführen will, beschreibt ein Event eine bereits vergangene Änderung innerhalb des Model. Daher wird für Eventnamen auch immer die Vergangenheitsform gewählt, um diesen Aspekt zu unterstreichen. *Listing 7* zeigt als Beispiel das CreatedCustomer-Event, das bei der Generierung eines neuen Customer erzeugt wird. *Listing 6* zeigt auch, wie Events aus dem Model emittiert werden können. Über das statische „AggregateLifecycle"-Objekt können neue Events erzeugt und in das Axon-System gebracht werden.

Event Handler

Die emittierten Events werden nun durch registrierte Event Handler weiterverarbeitet. Auch hier macht es die Spring Integration wieder sehr einfach. Handler-Klassen müssen lediglich als Spring Bean im Context vorhanden sein. Methoden, die Events verarbeiten, werden dann mit einer „@EventHandler"-Annotation versehen. *Listing 8* zeigt als Beispiel einen Adress-Sammler, der sich auf „AddressAdded"-Events registriert. Axon ermittelt den passenden Eventtyp für den Handler dabei per Reflection.

Read Model und Queries

CQRS sieht nicht vor, das Domain Model bei der Verarbeitung eines Command direkt an einen Aufrufer zurückzugeben. Für diese Fälle bietet es die Read-Modelle. Dabei muss das Read-Modell nicht eins zu eins das Domain-Modell abbilden, sondern kann und soll immer das Bedürfnis der Abfrage erfüllen. Daher ist es durchaus üblich, mehrere Read-Modelle in einem System zu haben. Die mögliche Datenredundanz wird hier gegenüber der besseren und unabhängigen Entwicklung und Wartung in Kauf genommen.

In großen Systemen kann das Read-Modell auch aus einer Kombination von Events unterschiedlicher Domain-Modelle bestehen, um z.B. KPIs oder andere Analysen um funktionsübergreifende Daten zu erheben. In *Listing 8* haben wir bereits ein erstes Beispiel für ein Read-Modell. Hier interessiert sich die Abfrage nicht für einzelne Kunden, sondern für die Gesamtheit aller Adressen.

Queries sind nun gezielte Abfragen auf ein Read-Modell. Im Beispiel des Listings 8 könnte dies beispielsweise die Anzahl aller Adressen sein. Oder eine disjunkte Menge aller Städte. Andere Abfragen wür-

```
@Aggregate
class Customer() {
...
  @EventHandler
  fun handleAddressAdded(event: AddressAdded) {
      this.addresses = addresses + event.address
  }

  fun addAddress(address: Address) {
      this.addresses = addresses + address
      AggregateLifecycle.apply(AddressAdded(...))
  }
...
}
```

Listing 9: Das Domain Model konsumiert die eigenen Events

den auf andere Aspekte des Systems abzielen und dafür gegebenenfalls andere Read-Modelle benötigen.

Event Sourcing

CQRS kann man per se auch ohne Events betreiben. Wichtig ist allein, dass Commands und Queries getrennt sind und ein zentrales Domain Model die Hoheit über die Daten hat. Allerdings bietet eine eventbasierte Architektur einige Vorteile. Dadurch, dass jede Zustandsänderung des Domain Model in einem entsprechenden Event resultiert, ist es möglich, den aktuellen Zustand des Domain Model allein durch die Zusammenführung aller Events wiederherzustellen. Ein Event Handler könnte so eine exakte Kopie des Domain Model erzeugen, ohne über das eigentliche Domain Model Bescheid zu wissen.

Speichert man dabei noch alle Events mit einem Timestamp, so lassen sich Read-Modelle auch in die Vergangenheit projizieren oder Simulationen mit alten Ständen betreiben. Auch ein Rollback im Falle eines Fehlers oder einer Simulation ist hier möglich.

Axon bietet Event Sourcing als festen Bestandteil und sogar in der Standardkonfiguration an. Jedes Event, das von der Domain emittiert wird, wird nicht nur an die Event Handler weitergeleitet, sondern auch in einem Event Store persistiert. In *Listing 5* wird die „load"-Methode des Repository aufgerufen, um das Aggregate, also das Domain Model, zu laden. Technisch wird hier direkt auf den Event Store zugegriffen und das Domain Model aus allen bisherigen Events aggregiert. Daher auch der Axon-Begriff „Aggregate" für das Domain Model. Damit nun aber das Domain Model aus den Events zusammengebaut werden kann, muss es seine eigenen Events konsumieren, um entscheiden zu können, wie es sich selbst aufbaut. *Listing 9* zeigt die Command-Methode „addAddress" und den Event Handler „handleAddressAdded".

Das Event Sourcing erklärt auch, warum in *Listing 5* das geänderte Aggregat nicht gespeichert wird. Durch den Aufruf von „addAddress" wird ein Event „AdressAdded" emittiert, das im Event Store gespeichert wird. Beim erneuten Laden wird dieses Event auf das Aggregat angewendet, um den aktuellen Zustand wiederherzustellen. Um nicht jedes Mal alle Events zu laden und zu verarbeiten, bietet Axon hier Abhilfe in Form von Snapshots.

Vor- und Nachteile

CQRS als Ganzes muss immer als ein Werkzeug in einem Werkzeugkasten verstanden werden. Nicht alle Probleme lassen sich mit CQRS lösen und selbstverständlich hat auch CRUD seine Daseinsberechtigung in einfachen Domänen ohne viel Fachlogik.

Den Aufwand der harten Trennung zwischen Domain Model und Read Model ist die größte Hürde, da man bei Änderungen gegebenenfalls zwei Modelle anpassen muss oder sogar zwei Persistenzschichten einbezieht.

Aber gerade wenn es darum geht, verschiedene Sichten auf eine Domäne zu haben, hat es seine Vorteile. Durch die sehr lose Kopplung forcieren Änderungen in der Domäne nur selten eine Query, sofern die Events abwärtskompatibel bleiben. Auch bietet Event Sourcing eine interessante Möglichkeit, Datenbestände zu versionieren und diese abfragbar zu machen. Mit Axon 4 bietet das Framework darüber hinaus auch einen externen EventBus, um verteilte Systeme aufzubauen.

Die Trennung von Domäne und Commands bietet ebenfalls eine interessante Entkopplung, da man „Nutzerwünsche" als solche im System abbilden kann, ohne direkt das eigentliche Datenmodell zu verändern und dieses in sich geschlossen zu halten.

Weitere und nicht behandelte Themen

Axon bietet über Command- und Eventhandler noch Unterstützung zu eventbasierten Queries, um die Abfragen noch weiter von den Modellen zu trennen.

Über Sagas können zusammenhängende Events betrachtet werden. Im Normalfall stehen Events immer für sich. In komplexen Geschäftstransaktionen können Events jedoch aufeinander aufbauen. Hier helfen Sagas als spezielle Form von Eventhandlern.

Weiterführende Quellen

Beispielprojekt: *https://gitlab.com/Iwanzik/axon-spring*
http://www.laputan.org/mud/
https://martinfowler.com/bliki/CQRS.html
https://docs.axoniq.io/reference-guide/v/3.4/
https://axoniq.io/

[1] *Vgl. http://www.laputan.org/mud/*
[2] *Vgl. https://martinfowler.com/bliki/CQRS.html*

Christian Iwanzik
christian.iwanzik@gmail.com

Christian Iwanzik, 33 Jahre aus Siegburg, ist nun seit 2010 bei der tarent Solutions GmbH in Bonn tätig. Hauptsächlich auf der JVM arbeitet er mit Java, Kotlin und Scala und entwickelt moderne Microservices in Spring oder Play. Gerade die innere Architektur der Microservices liegt dabei mit Domain Driven Design mit CQRS oder hexagonaler Architektur im Fokus. Außerhalb des Codes arbeitet er beireits seit einigen Jahren in modernen Microservicearchitekturen und kümmert sich um die Intergration in CI/CD Systeme.