**Event-Sourcing+CQRS example application**

This example application is the money transfer application described in my talk [Building and deploying microservices with event sourcing, CQRS and Docker](http://plainoldobjects.com/presentations/building-and-deploying-microservices-with-event-sourcing-cqrs-and-docker/). This talk describe a way of architecting highly scalable and available applications that is based on microservices, polyglot persistence, event sourcing (ES) and command query responsibility segregation (CQRS). Applications consist of loosely coupled components that communicate using events. These components can be deployed either as separate services or packaged as a monolithic application for simplified development and testing.

**Built using the Eventuate platform**

[](http://eventuate.io/)

Read the [overview](http://eventuate.io/) or look at the other [example applications](http://eventuate.io/exampleapps.html).

**Big ideas**

This example illustrates several important concepts:

* How to decompose an application into microservices - as described below the application consists of several services. For example, bank accounts are managed by one service, money transfers by another service.
* Using an event-driven architecture to achieve data consistency - rather than using traditional distributed transaction to maintain database consistency this application uses an eventually consistent, event-driven approach.
* Using event sourcing to implement the event-driven architecture - the domain logic consists of Domain-Driven Design (DDD) aggregates that using event sourcing.
* Using Command Query Responsibility Segregation (CQRS) - update requests (HTTP POSTs and PUTs) and view requests (HTTP GETs) are handled by separate services.
* How event sourcing enables deployment flexibility - the application can either be deployed as a monolith or as microservices.

**About the example application**

This example application provides a REST API for creating and viewing bank accounts and transferring money between them.

The following diagram shows the architecture:

There are four logical services:

* Accounts (command-side) - REST API for creating accounts
* Money transfers (command-side) - REST API for transferring money
* Account view updater (query-side) - subscribes to events and updates a MongoDB View
* Account view reader (query-side) - REST API for retrieving accounts

One of the neat things about the modular architecture is that there are two ways to deploy these four services:

* monolithic-service - all services are packaged as a single Spring Boot executable JAR
* Microservices - three separate Spring Boot executable JARs
  + accounts-command-side-service - command-side accounts
  + transactions-command-side-service - command-side money transfers
  + accounts-query-side-service - Account View Updater and Account View Reader

**About the examples**

There are currently the following versions of the example application:

* java-spring - a Java and Spring Boot example
* scala-spring - a Scala and Spring Boot example

Other examples will be added shortly including a Scala/Play example.

For more information, please see the [wiki](https://github.com/cer/event-sourcing-examples/wiki)

**About the Event Store**

The application uses one of two event stores:

* Embedded SQL-based event store, which is great for integration tests. It is also used when running the monolithic version of the application.
* Event Store server - this is a full featured event store. See this [wiki page](https://github.com/cer/event-sourcing-examples/wiki/AboutTheEventStoreServer) for more details.

**Building the application (and running the tests)**

Both versions of the application use Gradle. To build an application, execute this command in the application's top-level directory:

./gradlew assemble

Note: you do not need to install Gradle. It will be automatically downloaded by ./gradlew.

This will build a Spring Boot jar in each of the \*-service directories.

You can also run the tests using gradle build. However, you must set some environment variables.

First, you need to tell the query side code how to connect to MongoDB:

export SPRING\_DATA\_MONGODB\_URI=mongodb://192.168.59.103/yourdb

[Docker Compose](https://docs.docker.com/compose/) is a great way to run MongoDB. You can run the docker-compose up -d mongodb to run MongoDB.

Second, some of the tests in accounts-command-side-service, transactions-command-side-service, accounts-query-side-service and e2e-test need you need to set some environment variables that tell them how to connect to the Event Store server. But don't worry. The build is configured to ignore failures for those projects.

**Running the application**

To run the application, you must to set the SPRING\_DATA\_MONGODB\_URI environment variable, which tells the query services how to connect to MongoDB.

There are a couple of different ways of running the application.

**Running the monolithic application**

One option is to run the self-contained monolithic application. It uses the embedded event store.

Simply use this command:

java -jar monolithic-service/build/libs/monolithic-service.jar

**Running the microservices**

The other option is to run the services separately. However, in order to do this you need to [get credentials for the Event Store](https://github.com/cer/event-sourcing-examples/wiki/AboutTheEventStoreServer).

One way to run the services is to use the scripts run-all-services.sh, which runs the services, and kill-all-services.sh, which kills the processes.

A much better way, however, is to use Docker Compose. Simply run the command docker-compose up to launch the services. This will create containers for MongoDB and each of the services.

You can now, for example, use the curl commands in handy-curl-commands.sh to interact with the server.

You can also use the Swagger UI exposed by each service http://host:port/swagger-ui.html.

## [Does each microservice really need its own database?](https://plainoldobjects.com/2015/09/02/does-each-microservice-really-need-its-own-database-2/)

Posted on [September 2, 2015](https://plainoldobjects.com/2015/09/02/does-each-microservice-really-need-its-own-database-2/)by [ceracm](https://plainoldobjects.com/author/ceracm/" \o "View all posts by ceracm)

The short answer is yes. However, before you start hyperventilating about the cost of all those extra Oracle licenses, lets first explore why it is essential to do this and then discuss what is meant by the term ‘database’.

The main benefit of the [microservice architecture](http://microservices.io/patterns/microservices.html) is that it dramatically improves agility and velocity. That’s because when you  correctly decompose a system into microservices, you can develop and deploy each microservice independently and in parallel with the other services. In order to be able to independently develop microservices , they must be loosely coupled. Each microservice’s persistent data must be private to that service and only accessible via it’s API . If two or more microservices were to share persistent data then you need to carefully coordinate changes to the data’s schema, which would slow down development.

There are a few different ways to keep a service’s persistent data private. You do not need to provision a database server for each service. For example,  if you are using a relational database then the options are:

* Private-tables-per-service – each service owns a set of tables that must only be accessed by that service
* Schema-per-service – each service has a database schema that’s private to that service
* Database-server-per-service – each service has it’s own database server.

Private-tables-per-service and schema-per-service have the lowest overhead.  Using a schema per service is appealing since it makes ownership clearer. For some applications, it might make sense for database intensive services to have their own database server.

It is a good idea to create barriers that enforce this modularity. You could, for example, assign a different database user id to each service and use a database access control mechanism such as grants. Without some kind of barrier to enforce encapsulation, developers will always be tempted to bypass a service’s API and access it’s data directly.

It might also make sense to have a [polyglot persistence](http://martinfowler.com/bliki/PolyglotPersistence.html) architecture. For each service you choose the type of database that is best suited to that service’s requirements. For example, a service that does text searches could use ElasticSearch. A service that manipulates a social graph could use Neo4j. It might not make sense to use a relational database for every service.

There are some downsides to keeping a service’s persistent data private. Most notably, it can be challenging to implement business transactions that update data owned by multiple services. Rather than using distributed transaction, you typically must use an [eventually consistent, event-driven approach](https://github.com/cer/event-sourcing-examples/wiki/WhyEventDrivenArch) to maintain database consistency.

Another problem, is that it is difficult to implement some queries because you can’t do database joins across the data owned by multiple services. Sometimes, you can join the data within a service. In other situations, you will need to use [Command Query Responsibility Segregation (CQRS)](https://github.com/cer/event-sourcing-examples/wiki/WhyEventSourcing) and maintain denormalizes views.

Another challenge is that  services sometimes need to share data. For example, let’s imagine that several services need access to user profile data. One option is to encapsulate the user profile data with a service, that’s then called by other services. Another option is to use an event-driven mechanism to replicate data to each service that needs it.

In summary,  it is important that each service’s persistent data is private. There are, however,  a few different ways to accomplish this such as a schema-per-service. Some applications benefit from a polyglot persistence architecture that uses a mixture of database types.  A downside of not sharing databases is that maintaining data consistency and implementing queries is more challenging.

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# Introduction to Event Sourcing and Command Query Responsibility Separation (CQRS)

For slides and videos about building microservices using event sourcing and CQRS please see[Building and deploying microservices with event sourcing, CQRS and Docker](http://plainoldobjects.com/presentations/building-and-deploying-microservices-with-event-sourcing-cqrs-and-docker/).

# About event sourcing

As described [earlier](https://github.com/cer/event-sourcing-examples/wiki/WhyEventDrivenArch), event sourcing simplifies the development of event-driven applications. The key idea in event sourcing is that rather than persist the current state of a business entity, a service persists the entity’s state changing events. For example, the state-changing events for a bank account include AccountOpened, AccountDebited, and AccountCredited. When a service recreates an entity’s current state it simply replays the events.

## About the event store

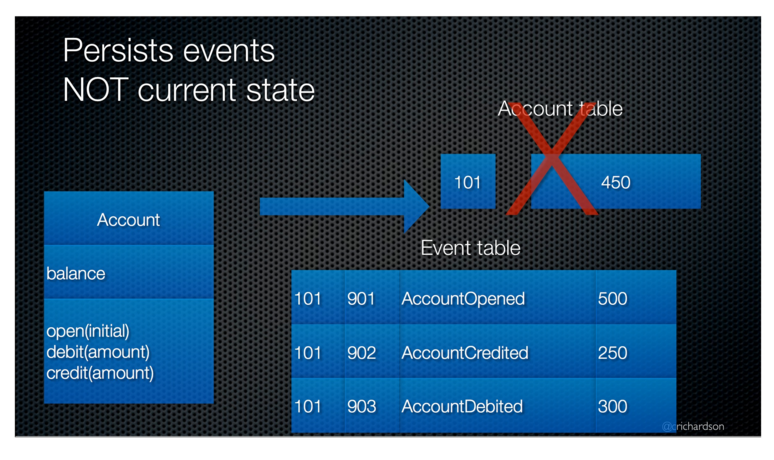
The application persists events in an event store. The event store behaves like both an database and a message broker. It's like a database since the application persists and retrieves events for an entity by primary key. The event store also behaves like a message broker since it publishes newly persisted events to interested consumers.

## Benefits of event sourcing

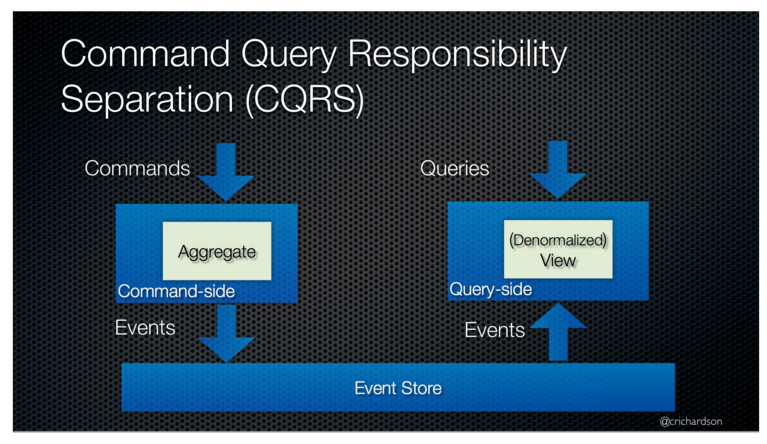
Event sourcing has several benefits. It’s a great way to implement an event-driven system because when a service persists a new event in the event store, that event is simultaneously published. You don’t need to implement a mechanism to atomically update state and publish an event. Another benefit of event sourcing is that the events published by a system are an accurate audit log of all user updates. You no longer need to sprinkle error-prone auditing logging code throughout your business logic. Event sourcing also eliminates the O/R mapping problem since events are trivial to persist.

# About Command Query Responsibility Separation (CQRS)

Event sourcing is great but for most applications it’s insufficient. That’s because an event store only supports the retrieval of events by an entity’s primary key. In order to implement more complex queries that do joins, such as those typically required by the UI, you need to use another pattern called Command-Query-Responsibility-Separation (CQRS). CQRS separates query processing from command processing, which updates entities using business logic that's implemented using event sourcing. The follow diagram shows the structure of a system that uses CQRS:



The query side consumes the events published by the command side when it updates aggregates and updates views. The views are designed specifically to support queries and often implemented using NoSQL databases. For example, one service might use a Neo4J –based view to handle queries about a social graph and another service might implement text search using ElasticSearch.



In addition to addressing the limitations of event sourcing, CQRS has other benefits. The separation of concerns means that you no longer implement a single model that handles both updates and queries. Instead, you have two narrowly focused models, one for updates and another for queries. In addition, CQRS let’s you scale the command side separately from the query side. This is especially relevant since many applications are very read intensive. This pattern also improves performance and scalability because the query side can use (denormalized) views that optimized for specific queries.

# Developing applications that use event sourcing

For more information on how to develop applications that use event sourcing please with the[developer guide](https://github.com/cer/event-sourcing-examples/wiki/DeveloperGuide).

**Developing applications with ES+CQRS**

**Introduction**

[**https://github.com/cer/event-sourcing-examples**](https://github.com/cer/event-sourcing-examples)

The [ES+CQRS](https://github.com/cer/event-sourcing-examples/wiki/WhyEventSourcing) pattern splits your application into two different kinds of modules. There are command modules, which handle update requests and contain your application’s business logic. The other kind of module is a query module, which handles read-only UI-oriented queries. The complexity of your application determines how your application’s modules are packaged for deployment. Simple applications might have a monolithic architecture, which packages all command and query modules into a single deployment unit, such as a WAR file. More complex applications typically have a [microservice architecture](http://microservices.io/patterns/microservices.html) consisting of multiple services, each of which contains one or more command and/or query modules.

**Command side**

The command modules implement the application’s business logic using the Domain Model pattern. ES+CQRS, however, imposes some restrictions on the structure of the domain model. Entities cannot reference each other without restriction. Instead, the domain model is broken up into Aggregates. Each Aggregate consists of one or more entities along with some value objects. One entity is the aggregate root from which all objects can be reached. An Aggregate does not reference another Aggregate directly – it only has the primary key. Furthermore, a transaction updates only a single Aggregate.

The approach of updating one aggregate per transaction has some important benefits. It works extremely well with aggregate-oriented NoSQL databases that do not support arbitrary transactions. It also works well with sharded SQL databases since transactions are limited to single shard. This approach is also consistent with the ideas about aggregate design in domain driven design including [this](http://martinfowler.com/bliki/DDD_Aggregate.html) and [this](http://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_1.pdf) (pdf).

Consider, for example, a banking domain model consisting of Accounts and MoneyTransfers, which represent a transfer of money from one account to another. In a traditional style domain model there are bidirectional relationships between Accounts and MoneyTransfer. Also, the application could transfer money by creating a money transfer and updating the corresponding accounts in a single ACID transaction.

In the ES+CQRS version of the application, those relationships are broken. For example, a MoneyTransfer stores the ids of the from/to Accounts. Here is what the ES+CQRS version of the domain model looks like.

This domain model consists of two independent aggregates: Account and MoneyTransfer. As described earlier, this version of the application transfers money uses a series of transactions, each one of which updates a single aggregate. The follow diagram shows series of events that transfer money between two accounts.

The MoneyTransferService creates a MoneyTransfer aggregate, which results in the publishing of a MoneyTransferCreatedEvent. The AccountService consumes that event, debits the ‘from’ Account, and publishes an AccountDebitedEvent. The MoneyTransferService consumes the AccountDebitedEvent, updates the MoneyTransfer to record the debit, and publishes a DebitRecordedEvent, and so on.

A command side module consists of an aggregate along with the adapters that handle inbound update requests. There are two types of inbound requests: external requests, such as HTTP requests and messages delivered by a message broker, and events published by other aggregates. The following diagram shows the structure of a command module.

Adapters convert external requests and events published by the event store into commands. The processing of a command either creates a new aggregate or updates an existing one. The new or update aggregate is persisted by saving events in the event store.

The following diagram shows how a request that updates an existing aggregate is handled.

The flow is as follows:

1. The aggregate’s existing events are loaded from the event store
2. An aggregate is instantiated using its default constructor
3. The current state is reconstituted by replaying the aggregate’s existing events
4. The command derived from the request/event is processed by the aggregate resulting in new events
5. The new events are saved in the event store and published, possibly triggering the update of additional aggregates or query side views

Let’s now look at how to implement an aggregate

**Defining Aggregates**

An aggregate consists of one or more entities along with their associated value objects. Event sourcing-based aggregates define two public methods. The first is processCommand(), which takes a command object derived from an update request, and returns a sequence of events. The second method is applyEvent(), which takes an event, and returns an updated aggregate. The applyEvent() method is called at two different times. It is invoked when processing an update request to update the state to reflect the new events. It is also called when an entity’s current state reconstituted by loading it’s events from the event store and replaying them.

**Defining aggregates in Scala**

Here is an example of an Account aggregate implemented in Scala

case class Account(balance : BigDecimal)

extends PatternMatchingCommandProcessingAggregate[Account, AccountCommand] {

def this() = this(null)

import net.chrisrichardson.eventstore.examples.bank.accounts.AccountCommands.\_

def processCommand = {

case OpenAccountCommand(initialBalance) =>

Seq(AccountOpenedEvent(initialBalance))

case CreditAccountCommand(amount, transactionId) =>

Seq(AccountCreditedEvent(amount, transactionId))

case DebitAccountCommand(amount, transactionId) if amount <= balance =>

Seq(AccountDebitedEvent(amount, transactionId))

case DebitAccountCommand(amount, transactionId) =>

Seq(AccountDebitFailedDueToInsufficientFundsEvent(amount, transactionId))

}

def applyEvent = {

case AccountOpenedEvent(initialBalance) => copy(balance = initialBalance)

case AccountDebitedEvent(amount, \_) => copy(balance = balance - amount)

case AccountCreditedEvent(amount, \_) =>

copy(balance = balance + amount)

case AccountDebitFailedDueToInsufficientFundsEvent(amount, \_) =>

this

}

}

The Account class is an immutable case class. It extend the trait CommandProcessingAggregate and defines two methods processCommand() and applyEvent(). The processCommand() method returns a PartialFunction from command to sequence of events. It leverages Scala’s pattern matching.

The applyEvent() method returns a PartialFunction from Event to Account. It uses pattern matching to dispatch on the Event and returns an updated copy of the account.

**Defining an aggregate in Java**

Here is the Java version of the Account class.

public class Account

extends ReflectiveMutableCommandProcessingAggregate<Account, AccountCommand> {

private BigDecimal balance;

public List<Event> process(OpenAccountCommand cmd) {

return EventUtil.events(new AccountOpenedEvent(cmd.getInitialBalance()));

}

public List<Event> process(DebitAccountCommand cmd) {

if (balance.compareTo(cmd.getAmount()) < 0)

return EventUtil.events(

new AccountDebitFailedDueToInsufficientFundsEvent(cmd.getTransactionId()));

else

return EventUtil.events(

new AccountDebitedEvent(cmd.getAmount(), cmd.getTransactionId()));

}

public List<Event> process(CreditAccountCommand cmd) {

return EventUtil.events(

new AccountCreditedEvent(cmd.getAmount(), cmd.getTransactionId()));

}

public void apply(AccountOpenedEvent event) {

balance = event.getInitialBalance();

}

public void apply(AccountDebitedEvent event) {

balance = balance.subtract(event.getAmount());

}

public void apply(AccountDebitFailedDueToInsufficientFundsEvent event) {

}

public void apply(AccountCreditedEvent event) {

balance = balance.add(event.getAmount());

}

public BigDecimal getBalance() {

return balance;

}

}

It extends the abstract class ReflectiveMutableCommandProcessingAggregate, which implements reflection-based logic to dispatch commands and events to the appropriate handler method. There are several overloaded process() method, one for each command type. Each one returns a java.util.List of events. There are also several overloaded apply() methods. Each one mutates the state of the object.

**Defining command classes**

Each aggregate has one or more command classes that must all extend an Aggregate-specific Command interface/trait. That interface/trait must extend the marker interface Command. For example, here is the Scala version of the OpenAccountCommand

sealed trait AccountCommand extends Command

case class OpenAccountCommand(initialBalance : BigDecimal) extends AccountCommand

The AccountCommand trait is the base trait for all of the Account’s commands.

Here is the Java version:

interface AccountCommand extends Command {

}

public class OpenAccountCommand implements AccountCommand {

private BigDecimal initialBalance;

public OpenAccountCommand(BigDecimal initialBalance) {

this.initialBalance = initialBalance;

}

public BigDecimal getInitialBalance() {

return initialBalance;

}

}

**Defining event classes**

In addition to have one or more command classes, an aggregate also has one or more Event classes. All event classes must extend the Event trait/interface. For example, here is the Scala version of the AccountOpenedEvent class

case class AccountOpenedEvent(initialBalance : BigDecimal) extends Event

Here is the Java version:

public class AccountOpenedEvent implements Event {

private BigDecimal initialBalance;

private AccountOpenedEvent() {

}

public AccountOpenedEvent(BigDecimal initialBalance) {

this.initialBalance = initialBalance;

}

public BigDecimal getInitialBalance() {

return initialBalance;

}

}

An event class must be capable of being serialized/deserialized to/from JSON using Jackson JSON.

All event classes or their containing package must have an @EventEntity annotation, which specifies the event’s Aggregate class. For example, here is the package-level annotation for the Account events.

@net.chrisrichardson.eventstore.EventEntity(entity="net.chrisrichardson.eventstore.javaexamples.banking.backend.commandside.accounts.Account")

package net.chrisrichardson.eventstore.javaexamples.banking.backend.commandside.accounts;

Currently, you need to specify the entity class name rather than the class object. That’s because these event classes are also used by the query side where the entity class is not available.

Now that we have looked at how to define aggregates let’s look at how to persist them using the event store. First we will look at using the EventStore interface directly. Later on we will look at a higher-level API that simplifies common tasks.

**Using the EventStore API**

The EventStore interface lets applications persist and load entities using event sourcing. Entities are versioned and the event store uses optimistic locking. To support reactive applications, the event store’s methods are asynchronous. The methods defined by the Java API return RxJava Observables and the Scala version’s method return Scala futures.

**Java version of the EventStore interface**

The Java version EventStore interface defines the following methods: save(), which persists a new entity’s events, update(), which persists an existing entity’s events, and find(), which loads an entity. Here is the Java version of the interface (Confusing = it’s written in Scala. LOL.):

trait EventStore {

def save[T <: javaapi.Aggregate[T]](entityClass: Class[T], events:

java.util.List[Event]) : Observable[EntityIdAndVersion]

def update[T <: javaapi.Aggregate[T]](entityClass: Class[T], entityIdAndVersion :

EntityIdAndVersion, events: java.util.List[Event]) : Observable[EntityIdAndVersion]

def find[T <: javaapi.Aggregate[T]](entityClass: Class[T], entityId: EntityId):

Observable[javaapi.EntityWithMetadata[T]]

…

}

The save() method takes two parameters, the class of the new entity and a list of events. It generates a unique id for the entity and persists the events in the event store. It returns an Observable[EntityIdAndVersion], which contains the entity’s assigned id, and it’s version.

The update() method, which updates an existing entity, takes three parameters: the entity’s class, EntityIdAndVersion, which contains the entity’s id and version, and a list of events. It persists the events in the event store. Like save(), it returns an Observable[EntityIdAndVersion]. It will throw an OptimisticLockingException if supplied version is older than the version in the event store.

The find() method, which loads an entity, takes two parameters, the class and id of the entity to load. It returns an Observable[EntityWithMetadata], which contains the entity’s id, it’s current version, and the entity reconstituted from it’s events.

**Scala version of the EventStore interface**

The Scala version is similar to the Java version. The main difference is that leverages some Scala features to simplify the method signatures (and it’s not written in Java ☺ ).

trait EventStore {

def save[T <: Aggregate[T] : ClassTag](events: Seq[Event], assignedId: Option[EntityId] = None, triggeringEvent: Option[ReceiptHandle] = None): Future[EntityIdAndVersion]

def update[T <: Aggregate[T] : ClassTag](entityIdAndVersion: EntityIdAndVersion, events: Seq[Event], triggeringEvent: Option[ReceiptHandle] = None): Future[EntityIdAndVersion]

def find[T <: Aggregate[T] : ClassTag](entityId: EntityId): Future[EntityWithMetadata[T]]

def findOptional[T <: Aggregate[T] : ClassTag](entityId: EntityId): Future[Option[EntityWithMetadata[T]]]

}

TODO: • Save: assignedId, • triggeringEvent

**Using the higher-level API**

Although the command side code can use the EventStore interface directly, it’s usually easier to use one of the higher-level APIs.

**Scala version**

The Scala API consists of a small DSL. Here is an example of a service written using the DSL.

class AccountService(implicit eventStore : EventStore) {

def openAccount(initialBalance : BigDecimal) =

newEntity[Account] <== OpenAccountCommand(initialBalance)

}

Behind the scenes, an Account is created, the command processed and the events persisted in the event store.

**Java version**

In your Java application you can use the AggregateRepository class, which defines methods that hide boilerplate:

public class AccountService {

private final AggregateRepository<Account, JavaAccountCommand> accountRepository;

public AccountService(AggregateRepository<Account, JavaAccountCommand>

accountRepository) {

this.accountRepository = accountRepository;

}

public rx.Observable<EntityWithMetadata<Account>>

openAccount(BigDecimal initialBalance) {

return accountRepository.save(new JavaOpenAccountCommand(initialBalance));

}

}

Here is how you can inject an AccountRepository into the AccountService using Spring Java configuration.

@Configuration

public class AccountConfiguration {

@Bean

public AccountService

accountService(AggregateRepository<Account, AccountCommand> accountRepository) {

return new AccountService(accountRepository);

}

@Bean

public AggregateRepository<Account, AccountCommand>

accountRepository(EventStore eventStore) {

return new AggregateRepository<Account, AccountCommand>(Account.class, eventStore);

}

}

**Writing Event handlers**

Entities typically subscribe to events published by other entities. For example, Account and MoneyTransfer subscribe to each other’s events. There are two ways to subscribe to events. One option is to use the EventStoreSubscriptionManagement interface directly. The other option is to use the higher-level event consumer API, which hides the boilerplate code.

**Using the EventStoreSubscriptionManagement interface in Java**

The EventStoreSubscriptionManagement interface defines a subscribeForObservable() method, which creates a durable named subscription to one or more event types:

trait EventStoreSubscriptionManagement {

def subscribeForObservable(subscriptionId: SubscriptionId):

Observable[AcknowledgableEventStream]

}

The subscriptionId parameter specifies the name of the subscription and the events to subscribe to. It returns an Observable[AcknowledgableEventStream] since subscribing is asynchronous. An AcknowledgableEventStream consists of an Observable containing the published events and an Acknowledger that provides API to acknowledge that an event has been processed.

**Using the EventStoreSubscriptionManagement interface in Scala**

The EventStoreSubscriptionManagement interface defines a subscribe() method, which creates a durable named subscription to one or more event types:

trait EventStoreSubscriptionManagement {

def subscribe(subscriptionId: SubscriptionId): Future[AcknowledgableEventStream]

}

This method returns a Scala Future since subscribing is asynchronous. An AcknowledgableEventStream consists of an Observable containing the published events and an Acknowledger that provides API to acknowledge that an event has been processed.

**Higher-level consumer API for Java**

The high-level consumer API is an easier to use interface that hides a lot of the boilerplate. To use this API you define one or more event handlers and register them as Spring beans. The framework takes care of the subscribing and dispatches events to the appropriate event handlers.

**Java version of the consumer API**

Here is an example of how the Account entity subscribes to MoneyTransferCreatedEvent, which is published by the MoneyTransfer entity.

@EventSubscriber(id="accountEventHandlers")

public class AccountWorkflow implements CompoundEventHandler {

@EventHandlerMethod

public Observable<?> debitAccount(EventHandlerContext<MoneyTransferCreatedEvent> ctx) {

MoneyTransferCreatedEvent event = ctx.getEvent();

BigDecimal amount = event.getDetails().getAmount();

Aggregate.EntityId transactionId = ctx.getEntityId();

Aggregate.EntityId fromAccountId = event.getDetails().getFromAccountId();

return ctx.update(Account.class, fromAccountId,

new DebitAccountCommand(amount, transactionId));

}

The event handlers are methods of a class that extends CompoundEventHandler. The @EventSubscriber annotation specifies the name of the subscription. An event handler method is invoked with an EventHandlerContext. The method obtains information about the event from the EventHandlerContext and invokes it’s update()/save() to update/create an entity.

To configure event handlers using Spring JavaConfig, you define EventHandler beans in a JavaConfig @Configuration class that has an @EnableEventHandlers annotation.

@Configuration

@EnableEventHandlers

public class AccountConfiguration {

@Bean

public AccountWorkflow accountWorkflow() {

return new AccountWorkflow();

}

The @EnableEventHandlers annotation takes care of calling EventStore.subscribe() for all CompoundEventHandlers defined in the application context.

**Scala version of the consumer API**

The Scala API is similar to the Java interface. The primary difference is that event handler methods have a different signature. Here

@EventSubscriber(id = "accountEventHandlers")

class TransferWorkflowAccountHandlers(eventStore: EventStore)

extends CompoundEventHandler {

implicit val es = eventStore

@EventHandlerMethod

val performDebit =

handlerForEvent[MoneyTransferCreatedEvent] { de =>

existingEntity[Account](de.event.details.fromAccountId) <==

DebitAccountCommand(de.event.details.amount, de.entityId)

}

Each event handler is a Scala val that is annotated with @EventHandlerMethod. Each event handler is defined using a simple DSL.

Now that we have looked at how to implement the command components, let’s look at the query side.

**Query-side**

Query side components subscribe to events and update views. You can accomplish by called EventStore.subscribe() directly. However, it’s much easier to use the high-level API, albeit with event handler methods with a slightly different signature.

**Writing Java event consumers**

Here is an example of a query-side event consumer that calls the AccountInfoUpdateService to update MongoDB in response to an event.

@EventSubscriber(id="querySideEventHandlers")

public class AccountQueryWorkflow implements CompoundEventHandler {

private AccountInfoUpdateService accountInfoUpdateService;

public AccountQueryWorkflow(AccountInfoUpdateService accountInfoUpdateService) {

this.accountInfoUpdateService = accountInfoUpdateService;

}

@EventHandlerMethod

public Observable<Object> create(DispatchedEvent<AccountOpenedEvent> de) {

AccountOpenedEvent event = de.event();

String id = de.entityId().id();

String eventId = de.eventId().asString();

BigDecimal initialBalance = event.getInitialBalance();

accountInfoUpdateService.create(id, initialBalance, eventId);

return Observable.just(null);

}

Each @EventHandlerMethod method has a DispatchedEvent parameter, which contains the event, the eventId, and the id and type of the entity that published the event.

Here is the corresponding Java Config class:

@Configuration

@EnableMongoRepositories

@EnableEventHandlers

public class QuerySideAccountConfiguration {

@Bean

public AccountQueryWorkflow

accountQueryWorkflow(AccountInfoUpdateService accountInfoUpdateService) {

return new AccountQueryWorkflow(accountInfoUpdateService);

}

Like the command side equivalent, the @Configuration class has an @EnableEventHandlers annotation.

**Writing Scala event consumers**

Here is the Scala version for the query side event handler for the AccountOpenedEvent:

@EventSubscriber (id = "querySideEventHandlers")

class AccountInfoUpdateService

(accountInfoRepository : AccountInfoRepository, mongoTemplate : MongoTemplate)

extends CompoundEventHandler with Logging {

@EventHandlerMethod

def created(de: DispatchedEvent[AccountOpenedEvent]) =

Future {

accountInfoRepository.save(

AccountInfo(de.entityId.id, toIntegerRepr(de.event.initialBalance),

Seq(), Seq(), de.eventId.asString))

…

}

In response to an AccountOpenedEvent, the created() method saves an AccountInfo in MongoDB using the Spring Data for Mongo-based AccountInfoRepository.

**Testing**

You can write integration tests for your ES+CQRS based code using the embedded JDBC-based event store. The JDBC event store uses an embedded, in-memory H2 database.

To use the JDBC event store in your integration tests simply import the JdbcEventStoreConfiguration:

@Configuration

@Import({AccountConfiguration.class, MoneyTransferConfiguration.class, JdbcEventStoreConfiguration.class})

public class BankingTestConfiguration {

}

This makes EventStore and EventStoreSubscriptionManagement beans available to be injected into your application components:

@RunWith(SpringJUnit4ClassRunner.class)

@SpringApplicationConfiguration(classes=BankingTestConfiguration.class)

@IntegrationTest

public class MoneyTransferIntegrationTest {

@Autowired

private AccountService accountService;

@Autowired

private MoneyTransferService moneyTransferService;

@Autowired

private EventStore eventStore;