Reference Guide

Axon Framework 2.0.9

Table of Contents

1. Introduction

Axon is a lightweight framework that helps developers build scalable and extensible applications by addressing these concerns directly in the architecture. This reference guide explains what Axon is, how it can help you and how you can use it.

If you want to know more about Axon and its background, continue reading in Section 1.1, “Axon Framework Background”. If you're eager to get started building your own application using Axon, go quickly to Section 1.2, “Getting started”. If you're interested in helping out building the Axon Framework, Section 1.3, “Contributing to Axon Framework” will contain the information you require. All help is welcome. Finally, this chapter covers some legal concerns in Section 1.5, “License information”.

1.1. Axon Framework Background

1.1.1. A brief history

The demands on software projects increase rapidly as time progresses. Companies no longer accept a brochure-like homepage to promote their business; they want their (web)applications to evolve together with their business. That means that not only projects and code bases become more complex, it also means that functionality is constantly added, changed and (unfortunately not enough) removed. It can be frustrating to find out that a seemingly easy-to-implement feature can require development teams to take apart an entire application. Furthermore, today's webapplications target an audience of potentially billions of people, making scalability an indisputable requirement.

Although there are many applications and frameworks around that deal with scalability issues, such as GigaSpaces and Terracotta, they share one fundamental flaw. These stacks try to solve the scalability issues while letting developers develop applications using the layered architecture they are used to. In some cases, they even prevent or severely limit the use of a real domain model, forcing all domain logic into services. Although that is faster to start building an application, eventually this approach will cause complexity to increase and development to slow down.

The Command Query Responsiblity Segregation (CQRS) pattern addresses these issues by drastically changing the way applications are architected. Instead of separating logic into separate layers, logic is separated based on whether it is changing an application's state or querying it. That means that executing commands (actions that potentially change an application's state) are executed by different components than those that query for the application's state. The most important reason for this separation is the fact that there are different technical and non-technical requirements for each of them. When commands are executed, the query components are (a)synchronously updated using events. This mechanism of updates through events, is what makes this architecture is extensible, scalable and ultimately more maintainable.

Note
A full explanation of CQRS is not within the scope of this document. If you would like to have more background information about CQRS,

visit the Axon Framework website: www.axonframework.org. It contains links to background information.

Since CQRS is so fundamentally different than the layered-architecture which dominates the software landscape nowadays, it is quite hard to grasp. It is not uncommon for developers to walk into a few traps while trying to find their way around this architecture. That's why Axon Framework was conceived: to help developers implement CQRS applications while focussing on the business logic.

1.1.2. What is Axon?

Axon Framework helps build scalable, extensible and maintainable applications by supporting developers apply the Command Query Responsibility Segregation (CQRS) architectural pattern. It does so by providing implementations of the most important building blocks, such as aggregates, repositories and event buses (the dispatching mechanism for events). Furthermore, Axon provides annotation support, which allows you to build aggregates and event listeners withouth tying your code to Axon specific logic. This allows you to focus on your business logic, instead of the plumbing, and helps you to make your code easier to test in isolation.

Axon does not, in any way, try to hide the CQRS architecture or any of its components from developers. Therefore, depending on team size, it is still advisable to have one or more developers with a thorough understanding of CQRS on each team. However, Axon does help when it comes to guaranteeing delivering events to the right event listeners and processing them concurrently and in the correct order. These multi-threading concerns are typically hard to deal with, leading to hard-to-trace bugs and sometimes complete application failure. When you have a tight deadline, you probably don't even want to care about these concerns. Axon's code is thoroughly tested to prevent these types of bugs.

The Axon Framework consists of a number of modules (jars) that provide the tools and components to build a scalable infrastructure. The Axon Core module provides the basic API's for the different components, and simple implemenatinos that provide solutions for single-JVM applications. The other modules address scalability or high-performance issues, by providing specialized building blocks.

1.1.3. When to use Axon?

Will each application benefit from Axon? Unfortunately not. Simple CRUD (Create, Read, Update, Delete) applications which are not expected to scale will probably not benefit from CQRS or Axon. Fortunately, there is a wide variety of applications that does benefit from Axon.

Applications that will most likely benefit from CQRS and Axon are those that show one or more of the following characteristics:

- The application is likely to be extended with new functionality during a long period of time. For example, an online store might start off with a system that tracks progress of Orders. At a later stage, this could be extended with Inventory information, to make sure stocks are updated when items are sold. Even later, accounting can require financial statistics of sales to be recorded, etc. Although it is hard to predict how software projects will evolve in the future, the majority of this type of application is clearly presented as such.

- The application has a high read-to-write ratio. That means data is only written a few times, and read many times more. Since data sources for queries are different to those that are used for command validation, it is possible to optimize these data sources for fast querying. Duplicate data is no longer an issue, since events are published when data changes.

- The application presents data in many different formats. Many applications nowadays don't stop when showing information on a web page. Some applications, for example, send monthly emails to notify users of changes that occured that might be relevant to them. Search engines are another example. They use the same data your application does, but in a way that is optimized for quick searching. Reporting tools aggregate information into reports that show data evolution over time. This, again, is a different format of the same data. Using Axon, each data source can be updated independently of each other on a real-time or scheduled basis.

- When an application has clearly separated components with different audiences, it can benefit from Axon, too. An example of such application is the online store. Employees will update product information and availability on the website, while customers place orders and query for their order status. With Axon, these components can be deployed on separate machines and scaled using different policies. They are kept up-to-date using the events, which Axon will dispatch to all subscribed components, regardles of the machine they are deployed on.

- Integration with other applications can be cumbersome work. The strict definition of an application's API using commands and events makes it easier to integrate with external applications. Any application can send commands or listen to events generated by the application.

1.2. Getting started

This section will explain how you can obtain the binaries for Axon to get started. There are currently two ways: either download the binaries from our website or configure a repository for your build system (Maven, Gradle, etc).

1.2.1. Download Axon

You can download the Axon Framework from our downloads page: axonframework.org/download.

This page offers a number of downloads. Typically, you would want to use the latest stable release. However, if you're eager to get started using the latest and greatest features, you could consider using the snapshot releases instead. The downloads page contains a number of assemblies for you to download. Some of them only provide the Axon library itself, while others also provide the libraries that Axon depends on. There is also a "full" zip file, which contains Axon, its dependencies, the sources and the documentation, all in a single download.

If you really want to stay on the bleeding edge of development, you can clone the Git repository: `git://github.com/AxonFramework/AxonFramework.git`, or visit `https://github.com/AxonFramework/AxonFramework` to browse the sources online.

1.2.2. Configure Maven

If you use maven as your build tool, you need to configure the correct dependencies for your project. Add the following code in your dependencies section:

```
<dependency>
    <groupId>org.axonframework</groupId>
    <artifactId>axon-core</artifactId>
    <version>2.0</version>
</dependency>
```

Most of the features provided by the Axon Framework are optional and require additional dependencies. We have chosen not to add these dependencies by default, as they would potentially clutter your project with artifacts you don't need.

1.2.3. Infrastructure requirements

Axon Framework doesn't impose many requirements on the infrastructure. It has been built and tested against Java 6, making that more or less the only requirement.

Since Axon doesn't create any connections or threads by itself, it is safe to run on an Application Server. Axon abstracts all asynchronous behavior by using `Executor`s, meaning that you can easily pass a container managed Thread Pool, for example. If you don't use an Application Server (e.g. Tomcat, Jetty or a stand-alone app), you can use the `Executors` class or the Spring Framework to create and configure Thread Pools.

1.2.4. When you're stuck

While implementing your application, you might run into problems, wonder about why certain things are the way they are, or have some questions that need an answer. The Axon Users mailinglist is there to help. Just send an email to axonframework@googlegroups.com. Other users as well as contributors to the Axon Framework are there to help with your issues.

If you find a bug, you can report them at axonframework.org/issues. When reporting an issue, please make sure you clearly describe the problem. Explain what you did, what the result was and what you expected to happen instead. If possible, please provide a very simple Unit Test (JUnit) that shows the problem. That makes fixing it a lot simpler.

1.3. Contributing to Axon Framework

Development on the Axon Framework is never finished. There will always be more features that we like to include in our framework to continue making development of scalabale and extensible application easier. This means we are constantly looking for help in developing our framework.

There are a number of ways in which you can contribute to the Axon Framework:

- You can report any bugs, feature requests or ideas about improvemens on our issue page: axonframework.org/issues. All ideas are welcome. Please be as exact as possible when reporting bugs. This will help us reproduce and thus solve the problem faster.

- If you have created a component for your own application that you think might be useful to include in the framework, send us a patch or a zip containing the source code. We will evaluate it and try to fit it in the framework. Please make sure code is properly documented using javadoc. This helps us to understand what is going on.
- If you know of any other way you think you can help us, do not hesitate to send a message to the Axon Framework mailinglist.

1.4. Commercial Support

Axon Framework is open source and freely available for anyone to use. However, if you have very specific requirements, or just want to be assured of someone to be standby to help you in case of trouble, Trifork provides several commercial support services for Axon Framework. These services include Training, Consultancy and Operational Support and are provided by the people that know Axon more than anyone else.

For more information about Trifork and its services, visit www.trifork.nl.

1.5. License information

The Axon Framework and its documentation are licensed under the Apache License, Version 2.0. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0.

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

2. Technical Overview

2.1. Architectural Overview

CQRS on itself is a very simple pattern. It only describes that the component of an application that processes commands should be separated from the component that processes queries. Although this separation is very simple on itself, it provides a number of very powerful features when combined with other patterns. Axon provides the building block that make it easier to implement the different patterns that can be used in combination with CQRS.

The diagram below shows an example of an extended layout of a CQRS-based event driven architecture. The UI component, displayed on the left, interacts with the rest of the application in two ways: it sends commands to the application (shown in the top section), and it queries the application for information (shown in the bottom section).

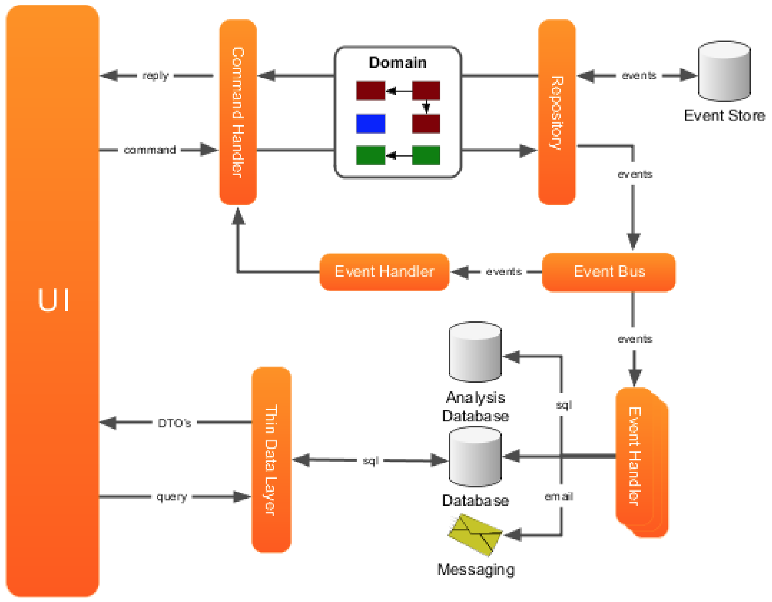

Figure 2.1. Architecture overview of a CQRS application

Command Handling

Commands are typically represented by simple and straightforward objects that contain all data necessary for a command handler to execute it. A command expresses its intent by its name. In Java terms, that means the class name is used to figure out what needs to be done, and the fields of the command provide the information required to do it.

The Command Bus receives commands and routes them to the Command Handlers. Each command handler responds to a specific type of command and executes logic based on the contents of the command. In some cases, however, you would also want to execute logic regardless of the actual type of command, such as validation, logging or authorization.

Axon provides building blocks to help you implement a command handling infrastructure with these features. These building blocks are thoroughly described in Chapter 3, *Command Handling*.

Domain Modeling

The command handler retrieves domain objects (Aggregates) from a repository and executes methods on them to change their state. These aggregates typically contain the actual business logic and are therefore responsible for guarding their own invariants. The state changes of aggregates result in the generation of Domain Events. Both the Domain Events and the Aggregates form the domain model. Axon provides supporting classes to help you build a domain model. They are described in Chapter 4, *Domain Modeling*.

Repositories and Event Stores

Repositories are responsible for providing access to aggregates. Typically, these repositories are optimized for lookup of an aggregate by its unique identifier only. Some repositories will store the state of the aggregate itself (using Object Relational Mapping, for example), while other store the state changes that the aggregate has gone through in an Event Store. The repository is also responsible for persisting the changes made to aggregates in its backing storage.

Axon provides support for both the direct way of persisting aggregates (using object-relational-mapping, for example) and for event sourcing. More about repositories and event stores can be found in Chapter 5, *Repositories and Event Stores*.

Event Processing

The event bus dispatches events to all interested event listeners. This can either be done synchronously or asynchronously. Asynchronous event dispatching allows the command execution to return and hand over control to the user, while the events are being dispatched and processed in the background. Not having to wait for event processing to complete makes an application more responsive. Synchronous event processing, on the other hand, is simpler and is a sensible default. Synchronous processing also allows several event listeners to process events within the same transaction.

Event listeners receive events and handle them. Some handlers will update data sources used for querying while others send messages to external systems. As you might notice, the command handlers are completely unaware of the components that are interested in the changes they make. This means that it is very non-intrusive to extend the application with new functionality. All you need to do is add another event listener. The events loosely couple all components in your application together.

In some cases, event processing requires new commands to be sent to the application. An example of this is when an order is received. This could mean the customer's account should be debited with the amount of the purchase, and shipping must be told to prepare a shipment of the purchased goods. In many applications, logic will become more complicated than this: what if the customer didn't pay in time? Will you send the shipment right away, or await payment first? The saga is the CQRS concept responsible for managing these complex business transactions.

The building blocks related to event handling and dispatching are explained in Chapter 6, *Event Processing*. Sagas are thoroughly explained in Chapter 7, *Managing complex business transactions*.

Querying for data

The thin data layer in between the user interface and the data sources provides a clearly defined interface to the actual query implementation used. This data layer typically returns read-only DTO objects containing query results. The contents of these DTO's are typically driven by the needs of the User Interface. In most cases, they map directly to a specific view in the UI (also referred to as table-per-view).

Axon does not provide any building blocks for this part of the application. The main reason is that this is very straightforward and doesn't differ from the layered architecture.

2.2. Axon Module Structure

Axon Framework consists of a number of modules that target specific problem areas of CQRS. Depending on the exact needs of your project, you will need to include one or more of these modules.

2.2.1. Main modules

Axon's main modules are the modules that have been thorouhgly tested and are robust enough to use in demanding production environments. The maven groupId of all these modules is `org.axonframework`.

Axon Core

The Core module contains, as the name suggests, the Core components of Axon. If you use a single-node setup, this module is likely to provide all the components you need. All other Axon modules depend on this module, so it must always be available on the classpath.

Axon Test

The Test modules contains test fixtures that you can use to test Axon based components, such as your Command Handlers, Aggregates and Sagas. You typically do not need this module at runtime and will only need to be added to the classpath during tests.

Axon Distributed CommandBus

The Distributed CommandBus modules contains a CommandBus implementation that can be used to distribute commands over multiple nodes. It comes with a JGroupsConnector that is used to connect DistributedCommandBus implementation on these nodes.

Axon Disruptor CommandBus

The SimpleCommandBus implementation provides pretty good performance, but the Disruptor CommandBus module allows you to squeeze more juice out of the same CPU. It provides an implementation that uses the Disruptor (a lightweight high-performance computing framework) to speed up processing.

Axon AMQP

The AMQP module provides components that allow you to build up an EventBus using an AMQP-based message broker as distribution mechanism. This allows for guaranteed-delivery, even when the Event Handler node is temporarily unavailable.

Axon Integration

The Integration module allows Axon components to be connected to Spring Integration channels. It provides an implementation for the EventBus that uses Spring Integration channels to distribute events as well as a connector to connect another EventBus to a Spring Integration channel.

Axon Mongo

MongoDB is a document based NoSQL database. The Mongo module provides an EventStore implementation that stores event streams in a MongoDB database.

Axon Monitoring JMX

Several AxonFramework components provide monitoring information. This module publishes that information over JMX. There is no configuration involved. If this module is on the classpath, statistics and monitoring informatin is automatically published over JMX.

2.2.2. Incubator modules

Incubator modules have not undergone the same amount of testing as the main modules, are not as well documented, and may therefore not be suitable for demanding production environments. They are typically work-in-progress and may have some "rough edges". Use these modules at your own peril.

The maven groupId for incubator modules is `org.axonframework.incubator`.

Axon Cassandra

This module aims to provide an implementation of the EventStore that uses Cassandra as backing storage.

Axon Google App Engine

The Google App Engine modules provides building blocks that use specific API's provided by GAE, such as an Event Store that uses the DatastoreService and a special XStream based serializer that works around some of the limitations of the GAE platform.

Axon Redis

Redis is a key-value based NoSQL store with very high performance characteristics. This module aims to provide an implementation of the EventStore that uses Redis as backing storage.

2.3. Working with Axon API's

To ensure maximum customizability, all Axon components are defined using interfaces. Abstract and concrete implementations are provided to help you on your way, but will nowhere be required by the framework. It is always possible to build a completely custom implementation of any building block using that interface.

2.4. Spring Support

Axon Framework provides extensive support for Spring, but does not require you to use Spring in order to use Axon. All components can be configured programmatically and do not require Spring on the classpath. However, if you do use Spring, much of the configuration is made easier with the use of Spring's namespace support. Building blocks, such as the Command Bus, Event Bus and Saga Managers can be configured using a single XML element in your Spring Application Context.

Check out the documentation of each building block for more information on how to configure it using Spring.

3. Command Handling

A state change within an application starts with a Command. A Command is a combination of expressed intent (which describes what you want done) as well as the information required to undertake action based on that intent. A Command Handler is responsible for handling commands of a certain type and taking action based on the information contained inside it.

The use of an explicit command dispatching mechanism has a number of advantages. First of all, there is a single object that clearly describes the intent of the client. By logging the command, you store both the intent and related data for future reference. Command handling also makes it easy to expose your command processing components to remote clients, via web services for example. Testing also becomes a lot easier, you could define test scripts by just defining the starting situation (given), command to execute (when) and expected results (then) by listing a number of events and commands (see Chapter 8, *Testing*). The last major advantage is that it is very easy to switch between synchronous and asynchronous command processing.

This doesn't mean Command dispatching using explicit command object is the only right way to do it. The goal of Axon is not to prescribe a specific way of working, but to support you doing it your way. It is still possible to use a Service layer that you can invoke to execute commands. The method will just need to start a unit of work (see Section 3.4, “Unit of Work”) and perform a commit or rollback on it when the method is finished.

The next sections provide an overview of the tasks related to creating a Command Handling infrastructure with the Axon Framework.

3.1. The Command Gateway

The Command Gateway is a convenient interface towards the Command dispatching mechanism. While you are not required to use a Gateway to dispatch Commands, it is generally the easiest option to do so.

There are two ways to use a Command Gateway. The first is to use the `CommandGateway` interface and `DefaultCommandGateway` implementation provided by Axon. The command gateway provides a number of methods that allow you to send a command and wait for a result either synchronously, with a timeout or asynchronously.

The other option is perhaps the most flexible of all. You can turn almost any interface into a Command Gateway using the `GatewayProxyFactory`. This allows you to define your application's interface using strong typing and declaring your own (checked) business exceptions. Axon will automatically generate an implementation for that interface at runtime.

3.1.1. Configuring the Command Gateway

Both your custom gateway and the one provided by Axon need to be configured with at least access to the Command Bus. In addition, the Command Gateway can be configured with a `RetryScheduler` and any number of `CommandDispatchInterceptor`s.

The `RetryScheduler` is capable of scheduling retries when command execution has failed. The `IntervalRetryScheduler` is an implementation that will retry a given command at set intervals until it succeeds, or a maximum number of retries is done. When a command fails due to an exception that is explicitly non-transient, no retries are done at all. Note that the retry scheduler is only invoked when a command fails due to a `RuntimeException`. Checked exceptions are regarded "business exception" and will never trigger a retry.

`CommandDispatchInterceptor`s allow modification of `CommandMessage`s prior to dispatching them to the Command Bus. In contrast to `CommandDispatchInterceptor`s configured on the CommandBus, these interceptors are only invoked when messages are sent through this gateway. The interceptors can be used to attach meta data to a command or do validation, for example.

3.1.2. Creating a Custom Command Gateway

The `GatewayProxyFactory` creates an instance of a Command Gateway based on an interface class. The behavior of each method is based on the parameter types, return type and declared exception. Using this gateway is not only convenient, it makes testing a lot easier by allowing you to mock your interface where needed.

This is how parameter affect the behavior of the CommandGateway:

- The first parameter is expected to be the actual command object to dispatch.
- Parameters annotated with `@MetaData` will have their value assigned to the meta data field with the identifier passed as annotation parameter
- Parameters of type `CommandCallback` will have their `onSuccess` or `onFailure` invoked after the Command is handled. You may pass in more than one callback, and it may be combined with a return value. In that case, the invocations of the callback will always match with the return value (or exception).
- The last two parameters may be of types `long` (or `int`) and `TimeUnit`. In that case the method will block at most as long as these parameters indicate. How the method reacts on a timeout depends on the exceptions declared on the method (see below). Note that if other properties of the method prevent blocking altogether, a timeout will never occur.

The declared return value of a method will also affect its behavior:

- A `void` return type will cause the method to return immediately, unless there are other indications on the method that one would want to wait, such as a timeout or declared exceptions.
- A Future return type will cause the method to return immediately. You can access the result of the Command Handler using the Future instance returned from the method. Exceptions and timeouts declared on the method are ignored.
- Any other return type will cause the method to block until a result is available. The result is cast to the return type (causing a ClassCastException if the types don't match).

Exceptions have the following effect:

- Any declared checked exception will be thrown if the Command Handler (or an interceptor) threw an exceptions of that type. If a checked exception is thrown that has not been declared, it is wrapped in a `CommandExecutionException`, which is a `RuntimeException`.
- When a timeout occurs, the default behavior is to return `null` from the method. This can be changed by declaring a `TimeoutException`. If this exception is declared, a `TimeoutException` is thrown instead.
- When a Thread is interrupted while waiting for a result, the default behavior is to return null. In that case, the interrupted flag is set back on the Thread. By declaring an `InterruptedException` on the method, this behavior is changed to throw that exception instead. The interrupt flag is removed when the exception is thrown, consistent with the java specification.
- Other Runtime Exceptions may be declared on the method, but will not have any effect other than clarification to the API user.

Finaly, there is the possiblity to use annotations:

- As specified in the parameter section, the `@MetaData` annotation on a parameter will have the value of that parameter added as meta data value. The key of the meta data entry is provided as parameter to the annotation.
- Methods annotated with `@Timeout` will block at most the indicated amount of time. This annotation is ignored if the method declares timeout parameters.
- Classes annotated with `@Timeout` will cause all methods declared in that class to block at most the indicated amount of time, unless they are annotated with their own `@Timeout` annotation or specify timeout parameters.

```
public interface MyGateway {

    // fire and forget
    void sendCommand(MyPayloadType command);
    // method that attaches meta data and will wait for a result for 10 seconds
    @Timeout(value = 10, unit = TimeUnit.SECONDS)
    ReturnValue sendCommandAndWaitForAResult(MyPayloadType command,
                                             @MetaData("userId") String userId);

    // alternative that throws exceptions on timeout
    @Timeout(value = 20, unit = TimeUnit.SECONDS)
    ReturnValue sendCommandAndWaitForAResult(MyPayloadType command)
                         throws TimeoutException, InterruptedException;

    // this method will also wait, caller decides how long
    void sendCommandAndWait(MyPayloadType command, long timeout, TimeUnit unit)
                         throws TimeoutException, InterruptedException;
}
```

Configuring a custom Command Gateway Spring

When using Spring, the easiest way to create a custom Command Gateway is by using the `CommandGatewayFactoryBean`. It uses setter injection, making it easier to configure. Only the "commandBus" and "gatewayInterface" properties are mandatory.

3.2. The Command Bus

The Command Bus is the mechanism that dispatches commands to their respective Command Handler. Commands are always sent to only one (and exactly one) command handler. If no command handler is available for a dispatched command, an exception (`NoHandlerForCommandException`) is thrown. Subscribing multiple command handlers to the same command type will result in subscriptions replacing each other. In that case, the last subscription wins.

3.2.1. Dispatching commands

The CommandBus provides two methods to dispatch commands to their respective handler: `dispatch(commandMessage, callback)` and `dispatch(commandMessage)`. The first parameter is a message containing the actual command to dispatch. The optional second parameter takes a callback that allows the dispatching component to be notified when command handling is completed. This callback has two methods: `onSuccess()` and `onFailure()`, which are called when command handling returned normally, or when it threw an exception, respectively.

The calling component may not assume that the callback is invoked in the same thread that dispatched the command. If the calling thread depends on the result before continuing, you can use the `FutureCallback`. It is a combination of a `Future` (as defined in the java.concurrent package) and Axon's `CommandCallback`. Alternatively, consider using a Command Gateway.

Best scalability is achieved when your application is not interested in the result of a dispatched command at all. In that case, you should use the single-parameter version of the `dispatch` method. If the `CommandBus` is fully asynchronous, it will return immediately after the command has been successfully dispatched. Your application will just have to guarantee that the command is processed and with "positive outcome", sooner or later...

3.2.2. SimpleCommandBus

The `SimpleCommandBus` is, as the name suggests, the simplest implementation. It does straightforward processing of commands in the thread that dispatches them. After a command is processed, the modified aggregate(s) are saved and generated events are published in that same thread. In most scenario's, such as web applications, this implementation will suit your needs.

The SimpleCommandBus allows interceptors to be configured. `CommandDispatchInterceptor`s are invoked when a command is dispatched on the Command Bus. The `CommandHandlerInterceptor`s are invoked before the actual command handler method is, allowing you to do modify or block the command. See Section 3.5, “Command Interceptors” for more information.

The SimpleCommandBus maintains a Unit of Work for each command published. This Unit of Work is created by a factory: the UnitOfWorkFactory. To suit any specifc needs your application might have, you can supply your own factory to change the Unit of Work implementation used.

Since all command processing is done in the same thread, this implementation is limited to the JVM's boundaries. The performance of this implementation is good, but not extraordinary. To cross JVM boundaries, or to get the most out of your CPU cycles, check out the other CommandBus implementations.

3.2.3. DisruptorCommandBus

The SimpleCommandBus has reasonable performance characteristics, especially when you've gone through the performance tips in Chapter 10, *Performance Tuning*. The fact that the SimpleCommandBus needs locking to prevent multiple threads from concurrently accessing the same aggregate causes processing overhead and lock contention.

The `DisruptorCommandBus` takes a different approach to multithreaded processing. Instead of having multiple threads each doing the same process, there are multiple threads, each taking care of a piece of the process. The `DisruptorCommandBus` uses the Disruptor (http://code.google.com/p/disruptor), a small framework for concurrent programming, to achieve much better performance, by just taking a different approach to multithreading. Instead of doing the processing in the calling thread, the tasks are handed of to two groups of threads, that each take care of a part of the processing. The first group of threads will execute the command handler, changing an aggregate's state. The second group will store and publish the events to the Event Store and Event Bus.

While the `DisruptorCommandBus` easily outperforms the `SimpleCommandBus` by a factor 4(!), there are a few limitations:

- The DisruptorCommandBus only supports Event Sourced Aggregates. This Command Bus also acts as a Repository for the aggreates processed by the Disruptor. To get a reference to the Repository, use `createRepository(AggregateFactory)`.
- A Command can only result in a state change in a single aggregate instance.
- Commands should generally not cause a failure that requires a rollback of the Unit of Work. When a rollback occurs, the DisruptorCommandBus cannot guarantee that Commands are processed in the order they were dispatched. Furthermore, it requires a retry of a number of other commands, causing unnecessary computations.
- When creating a new Aggregate Instance, commands updating that created instance may not all happen in the exact order as provided. Once the aggregate is created, all commands will be executed exactly in the order they were dispatched. To ensure the order, use a callback on the creating command to wait for the aggregate being created. It shouldn't take more than a few milliseconds.

Configuring the DisruptorCommandBus

To construct a `DisruptorCommandBus` instance, you need an `AggregateFactory`, an `EventBus` and `EventStore`. These components are explained in Chapter 5, *Repositories and Event Stores* and Section 6.1, "Event Bus". Finally, you also need a `CommandTargetResolver`. This is a mechanism that tells the disruptor which aggregate is the target of a specific Command. There are two implementations provided by Axon: the `AnnotationCommandTargetResolver`, which uses annotations to describe the target, or the `MetaDataCommandTargetResolver`, which uses the Command's Meta Data fields.

Optionally, you can provide a `DisruptorConfiguration` instance, which allows you to tweak the configuration to optimize performance for your specific environment. Spring users can use the <axon:disruptor-command-bus> element for easier configuration of the `DisruptorCommandBus`.

- ClaimStrategy: Sets the ClaimStrategy instance, which defines how a Thread dispatching a command can claim a "position" in the DisruptorCommandBus. Defaults to a `MultiThreadedClaimStrategy` with 4096 positions, which is safe to use when multiple threads can publish to the CommandBus.
- WaitStrategy: The strategy to use when the processor threads (the three threads taking care of the actual processing) need to wait for eachother. The best WaitStrategy depends on the number of cores available in the machine, and the number of other processes running. If low latency is crucial, and the DisruptorCommandBus may claim cores for itself, you can use the `BusySpinWaitStrategy`. To make the Command Bus claim less of the CPU and allow other threads to do processing, use the `YieldingWaitStrategy`. Finally, you can use the `SleepingWaitStrategy` and `BlockingWaitStrategy` to allow other processes a fair share of CPU. The latter is suitable if the Command Bus is not expected to be processing full-time. Defaults to the `BlockingWaitStrategy`.
- Executor: Sets the Executor that provides the Threads for the DisruptorCommandBus. This executor must be able to provide at least 4 threads. 3 threads are claimed by the processing components of the DisruptorCommandBus. Extra threads are used to invoke callbacks and to schedule retries in case an Aggregate's state is detected to be corrupt. Defaults to a CachedThreadPool that provides threads from a thread group called "DisruptorCommandBus".
- TransactionManager: Defines the Transaction Manager that should ensure the storage of events and their publication are executed transactionally.
- InvokerInterceptors: Defines the `CommandHandlerInterceptor`s that are to be used in the invocation process. This is the process that calls the actual Command Handler method.
- PublisherInterceptors: Defines the `CommandHandlerInterceptor`s that are to be used in the publication process. This is the process that stores and publishes the generated events.

- RollbackConfiguration: Defines on which Exceptions a Unit of Work should be rolled back. Defaults to a configuration that rolls back on unchecked exceptions.
- RescheduleCommandsOnCorruptState: Indicates whether Commands that have been executed against an Aggregate that has been corrupted (e.g. because a Unit of Work was rolled back) should be rescheduled. If `false` the callback's `onFailure()` method will be invoked. If `true` (the default), the command will be rescheduled instead.
- CoolingDownPeriod: Sets the number of seconds to wait to make sure all commands are processed. During the cooling down period, no new commands are accepted, but existing commands are processed, and rescheduled when necessary. The cooling down period ensures that threads are available for rescheduling the commands and calling callbacks. Defaults to 1000 (1 second).
- Cache: Sets the cache that stores aggregate instances that have been reconstructed from the Event Store. The cache is used to store aggregate instances that are not in active use by the disruptor.
- InvokerThreadCount: The number of threads to assign to the invocation of command handlers. A good starting point is half the number of cores in the machine.
- PublisherThreadCount: The number of threads to use to publish events. A good starting point is half the number of cores, and could be increased if a lot of time is spent on IO.
- SerializerThreadCount: The number of threads to use to pre-serialize events. This defaults to 1, but is ignored if no serializer is configured.
- Serializer: The serializer to perform pre-serialization with. When a serializer is configured, the DisruptorCommandBus will wrap all generated events in a SerializationAware message. The serialized form of the payload and meta data is attached before they are published to the Event Store or Event Bus. This can drastically improve performance when the same serializer is used to store and publish events to a remote destination.

3.3. Command Handlers

The Command Handler is the object that receives a Command of a pre-defined type and takes action based on its contents. In Axon, a Command may be any object. There is no predefined type that needs to be implemented.

3.3.1. Creating a Command Handler

A Command Handler must implement the `CommandHandler` interface. This interface declares only a single method: `Object handle(CommandMessage<T> command, UnitOfWork uow)`, where T is the type of Command this Handler can process. The concept of the UnitOfWork is explained in Section 3.4, “Unit of Work”. Be weary when using return values. Typically, it is a bad idea to use return values to return server-generated identifiers. Consider using client-generated (random) identifiers, such as UUIDs. They allow for a "fire and forget" style of command handlers, where a client does not have to wait for a response. As return value in such a case, you are recommended to simply return `null`.

3.3.2. Subscribing to a Command Bus

You can subscribe and unsubscribe command handlers using the `subscribe` and `unsubscribe` methods on `CommandBus`, respectively. They both take two parameters: the type of command to (un)subscribe the handler to, and the handler to (un)subscribe. An unsubscription will only be successful if the handler passed as the second parameter was currently assigned to handle that type of command. If another command was subscribed to that type of command, nothing happens.

```
CommandBus commandBus = new SimpleCommandBus();

// to subscribe a handler:
commandBus.subscribe(MyPayloadType.class.getName(), myCommandHandler);

// we can subscribe the same handler to different command types
commandBus.subscribe(MyOtherPayload.class.getName(), myCommandHandler);

// we can also unsubscribe the handler from one of these types:
commandBus.unsubscribe(MyOtherPayload.class.getName(), myCommandHandler);

// we don't have to use the payload to identifier the command type (but it's a good default)
commandBus.subscribe("MyCustomCommand", myCommandHandler);
```

3.3.3. Annotation based handlers

More often than not, a command handler will need to process several types of closely related commands. With Axon's annotation support you can use any POJO as command handler. Just add the `@CommandHandler` annotation to your methods to turn them into a command handler. These methods should declare the command to process as the first parameter. They may take optional extra parameters, such as the `UnitOfWork` for that command (see Section 3.4, "Unit of Work"). Note that for each command type, there may only be one handler! This restriction counts for all handlers registered to the same command bus.

Note

If you use Spring, you can add the `<axon:annotation-config/>` element to your application context. It will turn any bean with `@CommandHandler` annotated methods into a command handler. They will also be automatically subscribed to the `CommandBus`. In combination with Spring's classpath scanning (`@Component`), this will automatically subscribe any command handler in your application.

`AggregateAnnotationCommandHandler`

It is not unlikely that most command handler operations have an identical structure: they load an Aggregate from a repository and call a method on the returned aggregate using values from the command as parameter. If that is the case, you might benefit from a generic command handler: the `AggregateAnnotationCommandHandler`. This command handler uses `@CommandHandler` annotations on the aggregate's methods to identify which methods need to be invoked for an incoming command. If the `@CommandHandler` annotation is placed on a constructor, that command will cause a new Aggregate instance to be created.

The `AggregateAnnotationCommandHandler` still needs to know which aggregate instance (identified by it's unique Aggregate Identifier) to load and which version to expect. By default, the `AggregateAnnotationCommandHandler` uses annotations on the command object to find this information. The `@TargetAggregateIdentifier` annotation must be put on a field or getter method to indicate where the identifier of the target Aggregate can be found. Similarly, the `@TargetAggregateVersion` may be used to indicate the expected version.

The `@TargetAggregateIdentifier` annotation can be placed on a field or a method. The latter case will use the return value of a method invocation (without parameters) as the value to use.

If you prefer not to use annotations, the behavior can be overridden by supplying a custom `CommandTargetResolver`. This class should return the Aggregate Identifier and expected version (if any) based on a given command.

Creating new Aggregate Instances

When the `@CommandHandler` annotation is placed on an Aggregate's constructor, the respective command will create a new instance of that aggregte and add it to the repository. Those commands do not require to target a specific aggregate instance. That wouldn't make sense, since the instance is yet to be created. Therefore, those commands do not require any `@TargetAggregateIdentifier` or `@TargetAggregateVersion` annotations, nor will a custom `CommandTargetResolver` be invoked for these commands.

```
public class MyAggregate extends AbstractAnnotatedAggregateRoot {

    @AggregateIdentifier
    private String id;

    @CommandHandler
    public MyAggregate(CreateMyAggregateCommand command) {
        apply(new MyAggregateCreatedEvent(IdentifierFactory.getInstance().generateIdentifier()));
    }

    // no-arg constructor for Axon
    MyAggregate() {
    }

    @CommandHandler
    public void doSomething(DoSomethingCommand command) {
        // do something...
    }

    // code omitted for brevity. The event handler for MyAggregateCreatedEvent must set the id field
}

public class DoSomethingCommand {

    @TargetAggregateIdentifier
    private String aggregateId;

    // code omitted for brevity

}

// to generate the command handlers for this aggregate:
AggregateAnnotationCommandHandler handler = AggregateAnnotationCommandHandler.subscribe(MyAggregate.class, repository, commandBus);
// or when using another type of CommandTargetResolver:
AggregateAnnotationCommandHandler handler = AggregateAnnotationCommandHandler.subscribe(MyAggregate.class, repository, commandBus, myOwnCommandTargetResol

// to unsubscribe:
handler.unsubscribe();
```

3.4. Unit of Work

The Unit of Work is an important concept in the Axon Framework. The processing of a command can be seen as a single unit. Each time a command handler performs an action, it is tracked in the current Unit of Work. When command handling is finished, the Unit of Work is committed and all actions are finalized. This means that any repositores are notified of state changes in their aggregates and events scheduled for publication are sent to the Event Bus.

The Unit of Work serves two purposes. First, it makes the interface towards repositories a lot easier, since you do not have to explicitly save your changes. Secondly, it is an important hook-point for interceptors to find out what a command handler has done.

In most cases, you are unlikely to need access to the Unit of Work. It is mainly used by the building blocks that Axon provides. If you do need access to it, for whatever reason, there are a few ways to obtain it. The Command Handler receives the Unit Of Work through a parameter in the handle method. If you use annotation support, you may add a parameter of type `UnitOfWork` to your annotated method. In other locations, you can retrieve the Unit of Work bound to the current thread by calling `CurrentUnitOfWork.get()`. Note that this method will throw an exception if there is no Unit of Work bound to the current thread. Use `CurrentUnitOfWork.isStarted()` to find out if one is available.

One reason to require access to the current Unit of Work is to dispatch Events as part of a transaction, but those Events do not originate from an Aggregate. For example, you might want to publish an Event from a Command Handler, but don't require that Event to be stored as state in an Event Store. In such case, you can call `CurrentUnitOfWork.get().publishEvent(event, eventBus)`, where `event` is the Event (Message) to publish and `EventBus` the Bus to publish it on. The actual publication of the message is postponed unit the Unit of Work is committed, respecting the order in which Events have been registered.

> **Note**
> Note that the Unit of Work is merely a buffer of changes, not a replacement for Transactions. Although all staged changes are only committed when the Unit of Work is committed, its commit is not atomic. That means that when a commit fails, some changes might have been persisted, while other are not. Best practices dictate that a Command should never contain more than one action. If you stick to that practice, a Unit of Work will contain a single action, making it safe to use as-is. If you have more actions in your Unit of Work, then you could consider attaching a transaction to the Unit of Work's commit. See the section called “Binding the Unit of Work to a Transaction”.

UnitOfWork and Exceptions

Your command handlers may throw an Exception as a result of command processing. By default, unchecked exceptions will cause the UnitOfWork to roll back all changes. As a result, no Events are stored or published. In some cases, however, you might want to commit the Unif of Work and still notify the dispatcher of the command of an exception through the callback. The `SimpleCommandBus` allows you to provide a `RollbackConfiguration`. The `RollbackConfiguration` instance indicates whether an exception should perform a rollback on the Unit of Work, or a commit. Axon provides two implementation, which should cover most of the cases.

The `RollbackOnAllExceptionsConfiguration` will cause a rollback on any exception (or error). The default configuration, the `RollbackOnUncheckedExceptionConfiguration`, will commit the Unit of Work on checked exceptions (those not extending `RuntimeException`) while still performing a rollback on Errors and Runtime Exceptions.

Programatically managing a Unit of Work

When using a Command Bus, the lifeycle of the Unit of Work will be automatically managed for you. If you choose not to use explicit command objects and a Command Bus, but a Service Layer instead, you will need to programatically start and commit (or roll back) a Unit of Work instead.

In most cases, the DefaultUnitOfWork will provide you with the functionality you need. It expects Command processing to happen within a single thread. To start a new Unit Of Work, simply call `DefaultUnitOfWork.startAndGet();`. This will start a Unit of Work, bind it to the current thread (making it accessible via `CurrentUnitOfWork.get()`), and return it. When processing is done, either invoke `unitOfWork.commit();` or `unitOfWork.rollback(optionalException)`.

Typical usage is as follows:

```
UnitOfWork uow = DefaultUnitOfWork.startAndGet();
try {
    // business logic comes here
    uow.commit();
} catch (Exception e) {
    uow.rollback(e);
    // maybe rethrow...
}
```

Unit of Work phases

A Unit of Work knows several phases. Each time it progresses to another phase, the UnitOfWork Listeners are notified.

- Active phase: this is where the Unit of Work starts. Each time an event is registered with the Unit of Work, the `onEventRegistered` method is called. This method may alter the event message, for example to attach meta data to it. The return value of the method is the new EventMessage instance to use.
- Commit phase: before a Unit of Work is committed, the listeners' `onPrepareCommit` methods are invoked. This method is provided with the set of aggregates and list of Event Messages being stored. If a Unit of Work is bound to a transaction, the `onPrepareTransactionCommit` method is invoked. When the commit succeeded, the `afterCommit` method is invoked. If a commit failed, the `onRollback` is used. This method has a parameter which defines the cause of the failure, if available.
- Cleanup phase: This is the phase where any of the resources held by this Unit of Work (such as locks) are to be released. If multiple Units Of Work are nested, the cleanup phase is postponed until the outer unit of work is ready to clean up.

Binding the Unit of Work to a Transaction

The command handling process can be considered an atomic procedure; it should either be processed entirely, or not at all. Axon Framework uses the Unit Of Work to track actions performed by the command handlers. After the command handler completed, Axon will try to commit the actions registered with the Unit Of Work. This involves storing modified aggregates (see Chapter 4, *Domain Modeling*) in their respective repository (see Chapter 5, *Repositories and Event Stores*) and publishing events on the Event Bus (see Chapter 6, *Event Processing*).

The Unit Of Work, however, it is not a replacement for a transaction. The Unit Of Work only ensures that changes made to aggregates are stored upon successful execution of a command handler. If an error occurs while storing an aggregate, any aggregates already stored are not rolled back.

It is posssible to bind a transaction to a Unit of Work. Many CommandBus implementations, like the SimpleCommandBus and DisruptorCommandBus, allow you to configure a Transaction Manager. This Transaction Manager will then be used to create the transactions to bind to the Unit of Work that is used to manage the process of a Command. When a Unit of Work is bound to a transaction, it will ensure the bound transaction is committed at the right point in time. It also allows you to perform actions just before the transaction is committed, through the `UnitOfWorkListener`'s `onPrepareTransactionCommit` method.

When creating Unit of Work programmatically, you can use the `DefaultUnitOfWork.startAndGet(TransactionManager)` method to create a Unit of Work that is bound to a transaction. Alternatively, you can initialize the `DefaultUnitOfWorkFactory` with a `TransactionManager` to allow it to create Transaction-bound Unit of Work.

3.5. Command Interceptors

One of the advantages of using a command bus is the ability to undertake action based on all incoming commands. Examples are logging or authentication, which you might want to do regardless of the type of command. This is done using Interceptors.

There are two types of interceptors: Command Dispatch Interceptors and Command Handler Interceptors. The former are invoked before a command is dispatched to a Command Handler. At that point, it may not even be sure that any handler exists for that command. The latter are invoked just before the Command Handler is invoked.

3.5.1. Command Dispatch Interceptors

Command Dispatch Interceptors are invoked when a command is dispatched on a Command Bus. They have the ability to alter the Command Message, by adding Meta Data, for example, or block the command by throwing an Exception. These interceptors are always invoked on the thread that dispatches the Command.

3.5.1.1. Structural validation

There is no point in processing a command if it does not contain all required information in the correct format. In fact, a command that lacks information should be blocked as early as possible, preferably even before any transaction is started. Therefore, an interceptor should check all incoming commands for the availability of such information. This is called structural validation.

Axon Framework has support for JSR 303 Bean Validation based validation. This allows you to annotate the fields on commands with annotations like `@NotEmpty` and `@Pattern`. You need to include a JSR 303 implementation (such as Hibernate-Validator) on your classpath. Then, configure a `BeanValidationInterceptor` on your Command Bus, and it will automatically find and configure your validator implementation. While it uses sensible defaults, you can fine-tune it to your specific needs.

Tip
You want to spend as less resources on an invalid command as possible. Therefore, this interceptor is generally placed in the very front of the interceptor chain. In some cases, a Logging or Auditing interceptor might need to be placed in front, with the validating interceptor immediately following it.

The BeanValidationInterceptor also implements `CommandHandlerInterceptor`, allowing you to configure it as a Handler Interceptor as well.

3.5.2. Command Handler Interceptors

Command Handler Interceptors can take action both before and after command processing. Interceptors can even block command processing altogether, for example for security reasons.

Interceptors must implement the `CommandHandlerInterceptor` interface. This interface declares one method, `handle`, that takes three parameters: the command message, the current `UnitOfWork` and an `InterceptorChain`. The `InterceptorChain` is used to continue the dispatching process.

3.5.2.1. Auditing

Well designed events will give clear insight in what has happened, when and why. To use the event store as an Audit Trail, which provides insight in the exact history of changes in the system, this information might not be enough. In some cases, you might want to know which user caused the change, using what command, from which machine, etc.

The `AuditingInterceptor` is an interceptor that allows you to attach arbitrary information to events just before they are stored or published. The `AuditingInterceptor` uses an `AuditingDataProvider` to retrieve the information to attach to these events. You need to provide the implementation of the `AuditingDataProvider` yourself.

An Audit Logger may be configured to write to an audit log. To do so, you can implement the `AuditLogger` interface and configure it in the `AuditingInterceptor`. The audit logger is notified both on succesful execution of the command, as well as when execution fails. If you use event sourcing, you should be aware that the event log already contains the exact details of each event. In that case, it could suffice to just log the event identifier or aggregate identifier and sequence number combination.

Note
Note that the log method is called in the same thread as the command processing. This means that logging to slow sources may result in higher response times for the client. When important, make sure logging is done asynchronously from the command handling thread.

3.6. Distributing the Command Bus

The CommandBus implementations described in [Section 3.2, “The Command Bus”]() only allow Command Messages to be dispatched within a single JVM. Sometimes, you want multiple instances of Command Buses in different JVM's to act as one. Commands dispatched on one JVM's Command Bus should be seamlessly transported to a Command Handler in another JVM while sending back any results.

That's where the `DistributedCommandBus` comes in. Unlike the other `CommandBus` implementations, the `DistributedCommandBus` does not invoke any handlers at all. All it does is form a "bridge" between Command Bus implementations on different JVM's. Each instance of the `DistributedCommandBus` on each JVM is called a "Segment".

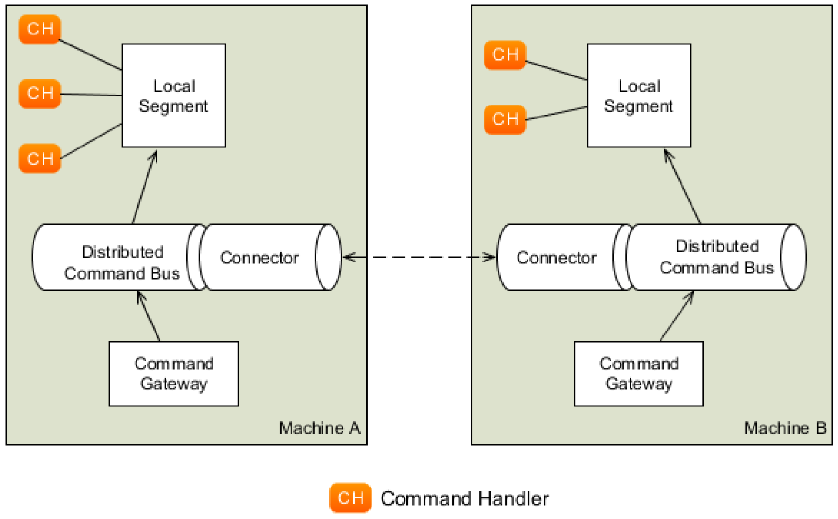

Figure 3.1. Structure of the Distributed Command Bus

Dependencies
The distributed command bus is not part of the Axon Framework Core module, but in the *axon-distributed-commandbus* module. If you use Maven, make sure you have the appropriate dependencies set. The groupId and version are identical to those of the Core module.

The `DistributedCommandBus` relies on two components: a `CommandBusConnector`, which implements the communication protocol between the JVM's, and the `RoutingStrategy`, which provides a Routing Key for each incoming Command. This Routing Key defines which segment of the Distributed Command Bus should be given a Command. Two commands with the same routing key will always be routed to the same segment, as long as there is no change in the number and configuration of the segments. Generally, the identifier of the targeted aggregate is used as a routing key.

Two implementations of the RoutingStrategy are provided: the `MetaDataRoutingStrategy`, which uses a Meta Data property in the Command Message to find the routing key, and the `AnnotationRoutingStrategy`, which uses the `@TargetAggregateIdentifier` annotation on the Command Messages payload to extract the Routing Key. Obviously, you can also provide your own implementation.

By default, the RoutingStrategy implementations will throw an exception when no key can be resolved from a Command Message. This behavior can be altered by providing a UnresolvedRoutingKeyPolicy in the constructor of the MetaDataRoutingStrategy or AnnotationRoutingStrategy. There are three possible policies:

- ERROR: This is the default, and will cause an exception to be thrown when a Routing Key is not available
- RANDOM_KEY: Will return a random value when a Routing Key cannot be resolved from the Command Message. This effectively means that those commands will be routed to a random segment of the Command Bus.
- STATIC_KEY: Will return a static key (being "unresolved") for unresolved Routing Keys. This effectively means that all those commands will be routed to the same segment, as long as the configuration of segments does not change.

3.6.1. JGroupsConnector

The `JGroupsConnector` uses (as the name already gives away) JGroups as the underlying discovery and dispatching mechanism. Describing the feature set of JGroups is a bit too much for this reference guide, so please refer to the JGroups User Guide for more details.

The JGroupsConnector has four mandatory configuration elements:

- The first is a JChannel, which defines the JGroups protocol stack. Generally, a JChannel is constructed with a reference to a JGroups configuration file. JGroups comes with a number of default confgurations which can be used as a basis for your own configuration. Do keep in mind that IP Multicast generally doesn't work in Cloud Services, like Amazon. TCP Gossip is generally a good start in such type of environment.
- The Cluster Name defines the name of the Cluster that each segment should register to. Segments with the same Cluster Name will eventually detect eachother and dispatch Command among eachother.
- A "local segment" is the Command Bus implementation that dispatches Commands destined for the local JVM. These commands may have been dispatched by instances on other JVM's or from the local one.
- Finally, the Serializer is used to serialize command messages before they are sent over the wire.

Ultimately, the JGroupsConnector needs to actually connect, in order to dispatch Messages to other segments. To do so, call the `connect()` method. It takes a single parameter: the load factor. The load factor defines how much load, relative to the other segments this segment should receive. A segment with twice the load factor of another segment will be assigned (approximately) twice the amount of routing keys as the other segments. Note that when commands are unevenly distributed over the rouing keys, segments with lower load factors could still receive more command than a segment with a higher load factor.

```
JChannel channel = new JChannel("path/to/channel/config.xml");
CommandBus localSegment = new SimpleCommandBus();
Serializer serializer = new XStreamSerializer();

JGroupsConnector connector = new JGroupsConnector(channel, "myCommandBus", localSegment, serializer);
DistributedCommandBus commandBus = new DistributedCommandBus(connector);

// on one node:
connector.connect(50);
commandBus.subscribe(CommandType.class.getName(), handler);

// on another node with more CPU:
connector.connect(150);
commandBus.subscribe(CommandType.class.getName(), handler);
commandBus.subscribe(AnotherCommandType.class.getName(), handler2);

// from now on, just deal with commandBus as if it is local...
```

Note

Note that it is not required that all segments have Command Handlers for the same type of Comands. You may use different segments for different Command Types altogether. The Distributed Command Bus will always choose a node to dispatch a Command to that has support for that specific type of Command.

The `JGroupsConnector` and Spring Framework

If you use Spring, you may want to consider using the `JGroupsConnectorFactoryBean`. It automatically connects the Connector when the ApplicationContext is started, and does a proper disconnect when the `ApplicationContext` is shut down. Furthermore, it uses sensible defaults for a testing environment (but should not be considered production ready) and autowiring for the configuration.

4. Domain Modeling

In a CQRS-based application, a Domain Model (as defined by Eric Evans and Martin Fowler) can be a very powerful mechanism to harness the complexity involved in the validation and execution of state changes. Although a typical Domain Model has a great number of building blocks, two of them play a major role when applied to CQRS: the Event and the Aggregate.

The following sections will explain the role of these building blocks and how to implement them using the Axon Framework.

4.1. Events

Events are objects that describe something that has occurred in the application. A typical source of events is the Aggregate. When something important has occurred within the aggregate, it will raise an Event. In Axon Framework, Events can be any object. You are highly encouraged to make sure all events are serializable.

When Events are dispatched, Axon wraps them in a Message. The actual type of Message used depends on the origin of the Event. When an Event is raised by an Aggregate, it is wrapped in a `DomainEventMessage` (which extends `EventMesssage`). All other Events are wrapped in an `EventMessage.` The `EventMessage` contains a unique Identifier for the event, as well as a Timestamp and Meta Data. The `DomainEventMessage` additionally contains the identifier of the aggregate that raised the Event and the sequence number which allows the order of events to be reproduced.

Even though the DomainEventMessage contains a reference to the Aggregate Identifier, you should always include that identifier in the actual Event itself as well. The identifier in the DomainEventMessage is meant for the EventStore and may not always provide a reliable value for other purposes.

The original Event object is stored as the Payload of an EventMessage. Next to the payload, you can store information in the Meta Data of an Event Message. The intent of the Meta Data is to store additional information about an Event that is not primarily intended as business information. Auditing information is a typical example. It allows you to see under which circumstances an Event was raised, such as the User Account that triggered the processing, or the name of the machine that processed the Event.

Note

In general, you should not base business decisions on information in the meta-data of event messages. If that is the case, you might have information attached that should really be part of the Event itself instead. Meta-data is typically used for auditing and tracing.

Although not enforced, it is good practice to make domain events immutable, preferably by making all fields final and by initializing the event within the constructor.

Note

Although Domain Events technically indicate a state change, you should try to capture the intention of the state in the event, too. A good practice is to use an abstract implementation of a domain event to capture the fact that certain state has changed, and use a concrete sub-implementation of that abstract class that indicates the intention of the change. For example, you could have an abstract `AddressChangedEvent`, and two implementations `ContactMovedEvent` and `AddressCorrectedEvent` that capture the intent of the state change. Some listeners don't care about the intent (e.g. database updating event listeners). These will listen to the abstract type. Other listeners do care about the intent and these will listen to the concrete subtypes (e.g. to send an address change confirmation email to the customer).

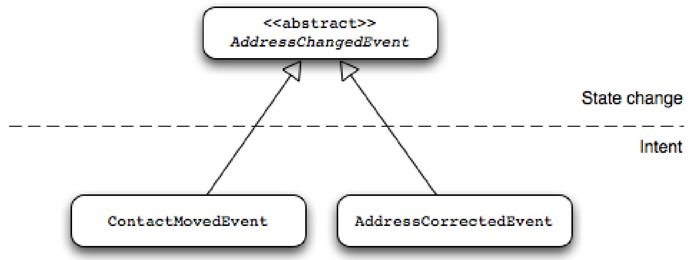

Figure 4.1. Adding intent to events

When dispatching an Event on the Event Bus, you will need to wrap it in an (Domain) Event Message. The `GenericEventMessage` is an implementation that allows you to wrap your Event in a Message. You can use the constructor, or the static `asEventMessage()` method. The latter checks whether the given parameter doesn't already implement the `Message` interface. If so, it is either returned directly (if it implements `EventMessage`,) or it returns a new `GenericEventMessage` using the given `Message`'s payload and Meta Data.

4.2. Aggregate

An Aggregate is an entity or group of entities that is always kept in a consistent state. The aggregate root is the object on top of the aggregate tree that is responsible for maintaining this consistent state.

Note

The term "Aggregate" refers to the aggregate as defined by Evans in Domain Driven Design:

"A cluster of associated objects that are treated as a unit for the purpose of data changes. External references are restricted to one member of the Aggregate, designated as the root. A set of consistency rules applies within the Aggregate's boundaries. "

A more extensive definition can be found on: http://domaindrivendesign.org/freelinking/Aggregate.

For example, a "Contact" aggregate could contain two entities: Contact and Address. To keep the entire aggregate in a consistent state, adding an address to a contact should be done via the Contact entity. In this case, the Contact entity is the appointed aggregate root.

In Axon, aggregates are identified by an Aggregate Identifier. This may be any object, but there are a few guidelines for good implementations of identifiers. Identifiers must:

- implement equals and hashCode to ensure good equality comparison with other instances,
- implement a toString() method that provides a consistent result (equal identifiers should provide an equal toString() result), and
- preferably be serializable.

The test fixtures (see Chapter 8, *Testing*) will verify these conditions and fail a test when an Aggregate uses an incompatible identifier. Identifiers of type `String`, `UUID` and the numeric types are always suitable.

Note

It is considered a good practice to use randomly generated identifiers, as opposed to sequenced ones. Using a sequence drastically reduces scalability of your application, since machines need to keep each other up-to-date of the last used sequence numbers. The chance of collisions with a UUID is very slim (a chance of 10^{-15}, if you generate 8.2×10^{11} UUIDs).

Furthermore, be careful when using functional identifiers for aggregates. They have a tendency to change, making it very hard to adapt your application accordingly.

4.2.1. Basic aggregate implementations

`AggregateRoot`

In Axon, all aggregate roots must implement the `AggregateRoot` interface. This interface describes the basic operations needed by the Repository to store and publish the generated domain events. However, Axon Framework provides a number of abstract implementations that help you writing your own aggregates.

Note

Note that only the aggregate root needs to implement the `AggregateRoot` interface or implement one of the abstract classes mentioned below. The other entities that are part of the aggregate do not have to implement any interfaces.

`AbstractAggregateRoot`

The `AbstractAggregateRoot` is a basic implementation that provides a `registerEvent(DomainEvent)` method that you can call in your business logic method to have an event added to the list of uncommitted events. The `AbstractAggregateRoot` will keep track of all uncommitted registered events and make sure they are forwarded to the event bus when the aggregate is saved to a repository.

4.2.2. Event sourced aggregates

Axon framework provides a few repository implementations that can use event sourcing as storage method for aggregates. These repositories require that aggregates implement the `EventSourcedAggregateRoot` interface. As with most interfaces in Axon, we also provide one or more abstract implementations to help you on your way.

`EventSourcedAggregateRoot`

The interface `EventSourcedAggregateRoot` defines an extra method, `initializeState()`, on top of the `AggregateRoot` interface. This method initializes an aggregate's state based on an event stream.

Implementations of this interface must always have a default no-arg constructor. Axon Framework uses this constructor to create an empty Aggregate instance before initialize it using past Events. Failure to provide this constructor will result in an Exception when loading the Aggregate from a Repository.

`AbstractEventSourcedAggregateRoot`

The `AbstractEventSourcedAggregateRoot` implements all methods on the `EventSourcedAggregateRoot` interface. It defines an abstract `handle()` method, which you need to implement with the actual logic to apply state changes based on domain events. When you extend the `AbstractEventSourcedAggregateRoot`, you can register new events using `apply()`. This method will register the event to be committed when the aggregate is saved, and will call the `handle()` method with the event as parameter. You need to implement this `handle()` method to apply the state changes represented by that event. Below is a sample implementation of an aggregate.

```java
public class MyAggregateRoot extends AbstractEventSourcedAggregateRoot {

    private String aggregateIdentifier;
    private String someProperty;

    public MyAggregateRoot(String id) {
        apply(new MyAggregateCreatedEvent(id));
    }

    // constructor required for reconstruction
    protected MyAggregateRoot() {
    }

    public void handle(DomainEvent event) {
        if (MyAggregateCreatedEvent.class.isAssignableFrom(event.getPayloadType())) {
            // make sure to always initialize the aggregate identifier
            this.aggregateIdentifier = ((MyAggregateCreatedEvent) this.getPayload()).getMyIdentifier();
            // do something with someProperty
        }
        // and more if-else-if logic here
    }

    public Object getAggregateIdentifier() {
        return aggregateIdentifier;
    }
}
```

isAssignableFrom versus instanceof

Note that the code sample used `isAssignableFrom` to verify the payload type of a command. This method is preferred over "`getPayload() instanceof`" for performance reasons. When reading events from the Event Store, they are actually not deserialized until the `getPayload()` method is invoked. By using `getPayloadType()` to detect interesting Events, you skip deserialization for events that you are not interested in.

`AbstractAnnotatedAggregateRoot`

As you see in the example above, the implementation of the `handle()` method can become quite verbose and hard to read. The `AbstractAnnotatedAggregateRoot` can help. The `AbstractAnnotatedAggregateRoot` is a specialization of the `AbstractAggregateRoot` that provides `@EventHandler` annotation support to your aggregate. Instead of a single `handle()` method, you can split the logic in separate methods, with names that you may define yourself. Just annotate the event handler methods with `@EventHandler`, and the `AbstractAnnotatedAggregateRoot` will invoke the right method for you.

> **Note**
>
> Note that `@EventHandler` annotated methods on an `AbstractAnnotatedAggregateRoot` are only called when events are applied directly to the aggregate locally. This should not be confused with annotating event handler methods on `EventListener` classes, in which case event handler methods handle events dispatched by the `EventBus`. See Section 6.2, “Event Listeners”.

```java
public class MyAggregateRoot extends AbstractAnnotatedAggregateRoot {

    @AggregateIdentifier
    private String aggregateIdentifier;
    private String someProperty;

    public MyAggregateRoot(String id) {
        apply(new MyAggregateCreatedEvent(id));
    }

    // constructor needed for reconstruction
    protected MyAggregateRoot() {
    }

    @EventHandler
    private void handleMyAggregateCreatedEvent(MyAggregateCreatedEvent event) {
        // make sure identifier is always initialized properly
        this.aggregateIdentifier = event.getMyAggregateIdentifier();
        // do something with someProperty
    }
}
```

`@EventHandler` annotated methods are resolved using specific rules. These rules are thorougly explained in Section 6.2.2, “Annotated Event Handler”.

> **Using private methods**
>
> Event handler methods may be private, as long as the security settings of the JVM allow the Axon Framework to change the accessibility of the method. This allows you to clearly separate the public API of your aggregate, which exposes the methods that generate events, from the internal logic, which processes the events.
>
> Most IDE's have an option to ignore "unused private method" warnings for methods with a specific annotation. Alternatively, you can add an `@SuppressWarnings("UnusedDeclaration")` annotation to the method.

The `AbstractAnnotatedAggregateRoot` also requires that the field containing the aggregate identifier is annotated with `@AggregateIdentifier`. If you use JPA and have JPA annotations on the aggregate, Axon can also use the `@Id` annotation provided by JPA.

4.2.3. Complex Aggregate structures

Complex business logic often requires more than what an aggregate with only an aggregate root can provide. In that case, it important that the complexity is spread over a number of entities within the aggregate. When using event sourcing, not only the aggregate root needs to use event to trigger state transitions, but so does each of the entities within that aggregate.

Axon provides support for event sourcing in complex aggregate structures. All entities other than the aggregate root need to implement `EventSourcedEntity` (but extending `AbstractEventSourcedEntity` or `AbstractAnnotatedEntity` makes implementing these entities easier). The `EventSourcedAggregateRoot` implementations provided by Axon Framework are aware of these entities and will call their event handlers when needed.

When an entity (including the aggregate root) applies an Event, it is registered with the Aggregate Root. The aggregate root applies the event locally first. Next, it will evaluate all fields annotated with `@EventSourcedMember` for any implementations of `EventSourcedEntity` and handle the event on them. Each entity does the same thing to its fields.

To register an Event, the Entity must know about the Aggregate Root. Axon will automatically register the Aggregate Root with an Entity before applying any Events to it. This means that Entities (in constrast to the Aggregate Root) should never apply an Event in their constructor. Non-Aggregate Root Entities should be created in an `@EventHandler` annotated method in their parent Entity (as creation of an entity can be considered a state change of the Aggregate). Axon will ensure that the Aggregate Root is properly registered in time. This also means that a single Entity cannot be part of more than one aggregate at any time.

Fields that (may) contain child entities must be annotated with `@EventSourcedMember`. This annotation may be used on a number of field types:

- directly referenced in a field;
- inside fields containing an `Iterable` (which includes all collections, such as `Set`, `List`, etc);
- inside both they keys and the values of fields containing a `java.util.Map`

If the value contained in an `@EventSourcedMember` annotated field does not implement `EventSourcedEntity`, it is simply ignored.

If you need to reference an Entity from any other location than the above mentioned, you can override the `getChildEntities()` method. This method should return a `Collection` of entities that should be notified of the Event. Note that each entity is invoked once for each time it is located in the returned `Collection`.

Note that in high-performance situations, the reflective approach to finding child entities can be costly. In that case, you can override the `getChildEntities()` method.

5. Repositories and Event Stores

The repository is the mechanism that provides access to aggregates. The repository acts as a gateway to the actual storage mechanism used to persist the data. In CQRS, the repositories only need to be able to find aggregates based on their unique identifier. Any other types of queries should be performed against the query database, not the Repository.

In the Axon Framework, all repositories must implement the `Repository` interface. This interface prescribes three methods: `load(identifier, version)`, `load(identifier)` and `add(aggregate)`. The `load` methods allows you to load aggregates from the repository. The optional `version` parameter is used to detect concurrent modifications (see Section 5.6, “Advanced conflict detection and resolution”). `add` is used to register newly created aggregates in the repository.

Depending on your underlying persistence storage and auditing needs, there are a number of base implementations that provide basic functionality needed by most repositories. Axon Framework makes a distinction between repositories that save the current state of the aggregate (see Section 5.1, “Standard repositories”), and those that store the events of an aggregate (see Section 5.2, “Event Sourcing repositories”).

Note that the Repository interface does not prescribe a `delete(identifier)` method. Deleting aggregates is done by invoking the (protected) `markDeleted()` method in an aggreate. This method is protected and not available from outside the aggregate. The motivation for this, is that the aggregate is responsible for maintaining its own state. Deleting an aggregate is a state migration like any other, with the only difference that it is irreversible in many cases. You should create your own meaningful method on your aggregate which sets the aggregate's state to "deleted". This also allows you to register any events that you would like to have published.

Repositories should use the `isDeleted()` method to find out if an aggregate has been marked for deletion. If such an aggregate is then loaded again, the repository should throw an `AggregateNotFoundException` (or when possible, an `AggregateDeletedException`). Axon's standard repository implementations will delete an aggregate from the repository, while event sourcing repositories will throw an Exception when an aggregate is marked deleted after initialization.

5.1. Standard repositories

Standard repositories store the actual state of an Aggregate. Upon each change, the new state will overwrite the old. This makes it possible for the query components of the application to use the same information the command component also uses. This could, depending on the type of application you are creating, be the simplest solution. If that is the case, Axon provides some building blocks that help you implement such a repository.

`AbstractRepository`

The most basic implementation of the repository is `AbstractRepository`. It takes care of the event publishing when an aggregate is saved. The actual persistence mechanism must still be implemented. This implementation doesn't provide any locking mechanism and expects the underlying data storage mechanism to provide it.

The `AbstractRepository` also ensures that activity is synchronized with the current Unit of Work. That means the aggregate is saved when the Unit of Work is committed.

`LockingRepository`

If the underlying data store does not provide any locking mechanism to prevent concurrent modifications of aggregates, consider using the abstract `LockingRepository` implementation. Besides providing event dispatching logic, it will also ensure that aggregates are not concurrently modified.

You can configure the `LockingRepository` with a locking strategy, such an optimistic or a pessimistic one. When the optimistic lock detects concurrent access, the second thread saving an aggregate will receive a `ConcurrencyException`. The pessimistic lock will prevent concurrent access to the aggregate alltogether. The pessimistic locking strategy is the default strategy. A custom locking strategy can be provided by implementing the `LockManager` interface.

Deadlocks are a common problem when threads use more than one lock to complete their operation. In the case of Sagas, it is not uncommon that a command is dispatched -causing a lock to be acquired-, while still holding a lock on the aggregate that cause the Saga to be invoked. The `PessimisticLockManager` will automatically detect an imminent deadlock and will throw a `DeadlockException` before the deadlock actually occurs. It is safe to retry the operation once all nested Units of Work have been rolled back (to ensure all locks are released). The `CommandGateway` will not invoke the `RetryScheduler` if a `DeadlockException` occured to prevent a retry before all held locks have been released.

Event ordering and optimistic locking strategy

Note that the optimistic lock doesn't lock any threads at all. While this reduces contention, it also means that the thread scheduler of your underlying architecture (OS, CPU, etc) is free to schedule threads as it sees fit. In high-concurrent environments (many threads accessing the same aggregate simultaneously), this could lead to events not being dispatched in exactly the same order as they are generated. If this guarantee is important, use pessimistic locking instead.

ConcurrencyException vs ConflictingModificationException

Note that there is a clear distinction between a `ConcurrencyException` and a `ConflictingModificationException`. The first is used to indicate a repository cannot save an aggregate, because the changes it contains were not applied to the latest available version. The latter indicates that the loaded aggregate contains changes that might not have been seen by the end-user. See Section 5.6, "Advanced conflict detection and resolution" for more information.

`GenericJpaRepository`

This is a repository implementation that can store JPA compatible Aggregates. It is configured with an `EntityManager` to manage the actual persistence, and a class specifiying the actual type of Aggregate stored in the Repository.

5.2. Event Sourcing repositories

Aggregate roots that implement the `EventSourcedAggregateRoot` interface can be stored in an event sourcing repository. Those repositories do not store the aggregate itself, but the series of events generated by the aggregate. Based on these events, the state of an aggregate can be restored at any time.

EventSourcingRepository

The `EventSourcingRepository` implementation provides the basic functionality needed by any event sourcing repository in the AxonFramework. It depends on an `EventStore` (see Section 5.3, "Event store implementations"), which abstracts the actual storage mechanism for the events and an `AggregateFactory`, which is reponsible for creating uninitialized aggregate instances.

The AggregateFactory specifies which aggregate needs to be created and how. Once an aggregate has been created, the `EventSourcingRepository` can initialize is using the Events it loaded from the Event Store. Axon Framework comes with a number of `AggregateFactory` implementations that you may use. If they do not suffice, it is very easy to create your own implementation.

GenericAggregateFactory

The `GenericAggregateFactory` is a special `AggregateFactory` implementation that can be used for any type of Event Sourced Aggregate Root. The `GenericAggregateFactory` creates an instance of the Aggregate type the repository manages. The Aggregate class must be non-abstract and declare a default no-arg constructor that does no initialization at all.

The GenericAggregateFactory is suitable for most scenarios where aggregates do not need special injection of non-serializable resources.

SpringPrototypeAggregateFactory

Depending on your architectural choices, it might be useful to inject dependencies into your aggregates using Spring. You could, for example, inject query repositories into your aggregate to ensure the existance (or inexistance) of certain values.

To inject dependencies into your aggregates, you need to configure a prototype bean of your aggregate root in the Spring context that also defines the `SpringPrototypeAggregateFactory`. Instead of creating regular instances of using a constructor, it uses the Spring Application Context to instantiate your aggregates. This will also inject any dependencies in your aggregate.

Implementing your own AggregateFactory

In some cases, the `GenericAggregateFactory` just doesn't deliver what you need. For example, you could have an abstract aggregate type with multiple implementations for different scenarios (e.g. `PublicUserAccount` and `BackOfficeAccount` both extending an `Account`). Instead of creating different repositories for each of the aggregates, you could use a single repository, and configure an AggregateFactory that is aware of the different implementations.

The AggregateFactory must specify the aggregate type identifier. This is a String that the Event Store needs to figure out which events belong to which type of aggregate. Typically, this name is deducted from the abstract super-aggregate. In the given example that could be: Account.

The bulk of the work the Aggregate Factory does is creating uninitialized Aggregate instances. It must do so using a given aggregate identifier and the first Event from the stream. Usually, this Event is a creation event which contains hints about the expected type of aggregate. You can use this information to choose an implementation and invoke its constructor. Make sure no Events are applied by that constructor; the aggregate must be uninitialized.

CachingEventSourcingRepository

Initializing aggregates based on the events can be a time-consuming effort, compared to the direct aggregate loading of the simple repository implementations. The `CachingEventSourcingRepository` provides a cache from which aggregates can be loaded if available. You can configure any jcache implementation with this repository. Note that this implementation can only use caching in combination with a pessimistic locking strategy.

> **Note**
> Using a cache with optimistic locking could cause undesired side-effects. Optimistic locking allows concurrent access to objects and will only fail when two threads have concurrently made any modifications to that object. When using a cache, both threads will receive the same instance of the object. They will both apply their changes to that same instance, potentially interfering with eachother.

HybridJpaRepository

The `HybridJpaRepository` is a combination of the `GenericJpaRepository` and an Event Sourcing repository. It can only deal with event sourced aggregates, and stores them in a relational model as well as in an event store. When the repository reads an aggregate back in, it uses the relational model exclusively.

This repository removes the need for Event Upcasters (see Section 5.4, “Event Upcasting”), making data migrations potentially easier. Since the aggregates are event sourced, you keep the ability to use the given-when-then test fixtures (see Chapter 8, *Testing*). On the other hand, since it doesn't use the event store for reading, it doesn't allow for automated conflict resolution.

5.3. Event store implementations

Event Sourcing repositories need an event store to store and load events from aggregates. Typically, event stores are capable of storing events from multiple types of aggregates, but it is not a requirement.

Axon provides a number of implementations of event stores, all capable of storing all domain events (those raised from an Aggregate). These event stores use a `Serializer` to serialize and deserialize the event. By default, Axon provides an implementation of the Event Serializer that serializes events to XML: the `XStreamSerializer`.

5.3.1. `FileSystemEventStore`

The `FileSystemEventStore` stores the events in a file on the file system. It provides good performance and easy configuration. The downside of this event store is that is does not provide transaction support and doesn't cluster very well. The only configuration needed is the location where the event store may store its files and the serializer to use to actually serialize and deserialize the events.

Note that the FileSystemEventStore is not aware of transactions and cannot automatically recover from crashes. Furthermore, it stores a single file for each aggregate, potentially creating too many files for the OS to handle. It is therefore not a suitable implementation for production environments.

Out of diskspace

When using the `FileSystemEventStore` in a test environment (or other environment where many aggregate may be created), you may end up with an "Out of disk space" error, even if there is plenty of *room* on the disk. The cause is that the filesystem runs out of i-nodes (or an equivalent when not on a Unix filesystem). This typically means that a filesystem holds too many files.

To prevent this problem, make sure the output directory of the `FileSystemEventStore` is cleaned after each test run.

5.3.2. `JpaEventStore`

The `JpaEventStore` stores events in a JPA-compatible data source. Unlike the file system version, the `JPAEventStore` supports transactions. The JPA Event Store stores events in so called entries. These entries contain the serialized form of an event, as well as some fields where meta-data is stored for fast lookup of these entries. To use the `JpaEventStore`, you must have the JPA (`javax.persistence`) annotations on your classpath.

By default, the event store needs you to configure your persistence context (defined in `META-INF/persistence.xml` file) to contain the classes `DomainEventEntry` and `SnapshotEventEntry` (both in the `org.axonframework.eventstore.jpa` package).

Below is an example configuration of a persistence context configuration:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
    <persistence-unit name="eventStore"❶ transaction-type="RESOURCE_LOCAL">
        <class>org...eventstore.jpa.DomainEventEntry</class> ❷
        <class>org...eventstore.jpa.SnapshotEventEntry</class>
    </persistence-unit>
</persistence>
```

❶ In this sample, there is is specific persistence unit for the event store. You may, however, choose to add the third line to any other persistence unit configuration.

❷ This line registers the `DomainEventEntry` (the class used by the `JpaEventStore`) with the persistence context.

Detecting duplicate key violations in the database

Axon uses Locking to prevent two threads from accessing the same Aggregate. However, if you have multiple JVMs on the same database, this won't help you. In that case, you'd have to rely on the database to detect conflicts. Concurrent access to the event store will result in a Key Constraint Violation, as the table only allows a single Event for an aggregate with any sequence number. Inserting a second event for an existing aggregate with an existing sequence number will result in an error.

The JPA EventStore can detect this error and translate it to a ConcurrencyException. However, each database system reports this violation differently. If you register your DataSource with the JpaEventStore, it will try to detect the type of database and figure out which error codes represent a Key Constraint Violation. Alternatively, you may provide a PersistenceExceptionTranslator instance, which can tell if a given exception represents a Key Constraint Violation.

If no DataSource or PersistenceExceptionTranslator is provided, exceptions from the Database driver are thrown as-is.

Working with multiple Persistence Contexts

By default, the JPA Event Store expects the application to have only a single, container managed, persistence context. In many cases, however, an application has more than one. In that case, you must provide an explicit `EntityManagerProvider` implementation that returns the `EntityManager` instance for the `EventStore` to use. This also allows for application managed persistence contexts to be used. It is the `EntityManagerProvider`'s responsiblity to provide a correct instance of the `EntityManager`.

There are a few implementations of the `EntityManagerProvider` available, each for different needs. The `SimpleEntityManagerProvider` simply returns the `EntityManager` instance which is given to it at construction time. This makes the implementation a simple option for Container Managed Contexts. Alternatively, there is the `ContainerManagedEntityManagerProvider`, which returns the default persistence context, and is used by default by the Jpa Event Store.

If you have a persistence unit called "myPersistenceUnit" which you wish to use in the `JpaEventStore`, this is what the EntityManagerProvider implementation could look like:

```
public class MyEntityManagerProvider implements EntityManagerProvider {

    private EntityManager entityManager;

    @Override
    public EntityManager getEntityManager() {
        return entityManager;
    }

    @PersistenceContext(unitName = "myPersistenceUnit")
    public void setEntityManager(EntityManager entityManager) {
        this.entityManager = entityManager;
    }
```

Customizing the Event storage

By default, the JPA Event Store stores entries in `DomainEventEntry` and `SnapshotEventEntry` entities. While this will suffice in many cases, you might encounter a situation where the meta-data provided by these entities is not enough. Or you might want to store events of different aggregate types in different tables.

If that is the case, you may provide your own implementation of `EventEntryStore` in the JPA Event Store's constructor. You will need to provide implementations of methods that load and store serialized events. Check the API Documentation of the `EventEntryStore` class for implementation requirements.

Memory consumption warning

Note that persistence providers, such as Hibernate, use a first-level cache on their EntityManager implementation. Typically, this means that all entities used or returned in queries are attached to the EntityManager. They are only cleared when the surrounding transaction is committed or an explicit "clear" in performed inside the transaction. This is especially the case when the Queries are executed in the context of a transaction.

To work around this issue, make sure to exclusively query for non-entity objects. You can use JPA's "SELECT new SomeClass(parameters) FROM ..." style queries to work around this issue. Alternatively, call `EntityManager.flush()` and `EntityManager.clear()` after fetching a batch of events. Failure to do so might result in `OutOfMemoryException`s when loading large streams of events.

Hibernate support

The `EventEntryStore` implementation used by Axon automatically detects whether Hibernate is used as JPA implementation. If that is the case, it will use some Hiberante-specific features to optimize performance while reducing memory consumption. To prevent Hibernate specific logic, configure the JPA Event Store to use the `DefaultEventEntryStore` instead.

5.3.3. MongoDB Event Store

MongoDB is a document based NoSQL store. It scalability characteristics make it suitable for use as an Event Store. Axon provides the MongoEventStore, which uses MongoDB as backing database. It is contained in the Axon Mongo module (Maven artifactId `axon-mongo`).

Events are stored in two separate collections: one for the actual event streams and one for the snapshots.

By default, the MongoEventStore stores each event in a separate document. It is, however, possible to change the `StorageStrategy` used. The alternative provided by Axon is the DocumentPerCommitStorageStrategy, which creates a single document for all Events that have been stored in a single commit (i.e. in the same DomainEventStream).

Storing an entire commit in a single document has the advantage that a commit is stored atomically. Furthermore, it requires only a single roundtrip for any number of events. A disadvantage is that it becomes harder to query events manually or through the `EventStoreManagement` methods. When refactoring the domain model, for example, it is harder to "transfer" events from one aggregate to another if they are included in a "commit document".

The MongoDB doesn't take a lot of configuration. All it needs is a reference to the collections to store the Events in, and you're set to go.

5.3.4. Implementing your own event store

If you have specific requirements for an event store, it is quite easy to implement one using different underlying data sources. Reading and appending events is done using a `DomainEventStream`, which is quite similar to iterator implementations.

Instead of eagerly deserializing Events, consider using the `SerializedDomainEventMessage`, which will postpone deserialization of Meta Data and Payload until it is actually used by a handler.

Tip

The `SimpleDomainEventStream` class will make the contents of a sequence (`List` or `array`) of `EventMessage` instances accessible as event stream.

5.3.5. Influencing the serialization process

Event Stores need a way to serialize the Event to prepare it for storage. By default, Axon provides the `XStreamSerializer`, which uses XStream to serialize Events into XML and vice versa. XStream is reasonably fast and is more flexible than Java Serialization. Furthermore, the result of XStream serialization is human readable. Quite useful for logging and debugging purposes.

The XStreamSerializer can be configured. You can define aliases it should use for certain packages, classes or even fields. Besides being a nice way to shorten potentially long names, aliases can also be used when class definitions of event change. For more information about aliases, visit the XStream website.

Spring XML Configuration and Serializer Customization

Configuring the serializer using Java code (or other JVM languages) is easy. However, configuring it in a Spring XML Application Context is not so trivial, due to its limitations to invoke methods. One of the options is to create a `FactoryBean` that creates an instance of an XStreamSerializer and configures it in code. Check the Spring Reference

You may also implement your own Serializer, simply by creating a class that implements `Serializer`, and configuring the Event Store to use that implementation instead of the default.

5.4. Event Upcasting

Due to the ever-changing nature of software applications it is likely that event definitions also change over time. Since the Event Store is considered a read and append-only data source, your application must be able to read all events, regardless of when they have been added. This is where upcasting comes in.

Originally a concept of object-oriented programming, where: "a subclass gets cast to it's superclass automatically when needed", the concept of upcasting can also be applied to event sourcing. To upcast an event means to transform it from its original structure to its new structure. Unlike OOP upcasting, event upcasting cannot be done in full automation because the structure of the new event is unknown to the old event. Manually written Upcasters have to be provided to specify how to upcast the old structure to the new structure.

Upcasters are classes that take one input event of revision `x` and output zero or more new events of revision `x + 1`. Moreover, upcasters are processed in a chain, meaning that the output of one upcaster is sent to the input of the next. This allows you to update events in an incremental manner, writing an Upcaster for each new event revision, making them small, isolated, and easy to understand.

Note

Perhaps the greatest benefit of upcasting is that it allows you to do non-destructive refactoring, i.e. the complete event history remains intact.

In this section we'll explain how to write an upcaster, describe the two implementations of the Upcaster Chain that come with Axon, and explain how the serialized representations of events affects how upcasters are written.

To allow an upcaster to see what version of serialized object they are receiving, the Event Store stores a revision number as well as the fully qualified name of the Event. This revision number is generated by a `RevisionResolver`, configured in the serializer. Axon provides several implementations of the `RevisionResolver`, such as the `AnnotationRevisionResolver`, which checks for an `@Revision` annotation on the Event payload, a

`SerialVersionUIDRevisionResolver` that uses the `serialVersionUID` as defined by Java Serialization API and a `FixedValueRevisionResolver`, which always returns a predefined value. The latter is useful when injecting the current application version. This will allow you to see which version of the application generated a specific event.

5.4.1. Writing an upcaster

To explain how to write an upcaster for Axon we'll walk through a small example, describing the details of writing an upcaster as we go along.

Let's assume that there is an Event Store containing many `AdministrativeDetailsUpdated` events. New requirements have let to the introduction of two new events: `AddressUpdatedEvent` and `InsurancePolicyUpdatedEvent`. Previously though, all information in these two events was contained in the old `AdministrativeDetailsUpdatedEvent`, which is now deprecated. To nicely handle this situation we'll write an upcaster to transform the `AdministrativeDetailsUpdatedEvent` into an `AddressUpdatedEvent` and an `InsurancePolicyUpdatedEvent`.

Here is the code for an upcaster:

```
import org.dom4j.Document;

public class AdministrativeDetailsUpdatedUpcaster implements Upcaster<Document> { ❶

    @Override
    public boolean canUpcast(SerializedType serializedType) { ❷
        return serializedType.getName().equals("org.example.AdministrativeDetailsUpdated") &&
               "0".equals(serializedType.getRevision());
    }

    @Override
    public Class<Document> expectedRepresentationType() { ❸
        return Document.class;
    }

    @Override
    public List<SerializedObject<Document>> upcast(SerializedObject<Document> intermediateRepresentation,
                                                   List<SerializedType> expectedTypes, Up❹castingContext context) {
        Document administrativeDetailsUpdatedEvent = intermediateRepresentation.getData();

        Document addressUpdatedEvent =
                new DOMDocument(new DOMElement("org.example.InsurancePolicyUpdatedEvent"));
        addressUpdatedEvent.getRootElement()
                .add(administrativeDetailsUpdatedEvent.getRootElement().element("address"));

        Document insurancePolicyUpdatedEvent =
                new DOMDocument(new DOMElement("org.example.InsurancePolicyUpdatedEvent"));
        insurancePolicyUpdatedEvent.getRootElement()
                .add(administrativeDetailsUpdatedEvent.getRootElement().element("policy"));

        List<SerializedObject<?>> upcastedEvents = new ArrayList<SerializedObject<?>>();
        upcastedEvents.add(new SimpleSerializedObject<Document>(
                addressUpdatedEvent, Document.class, expectedTypes.get(0)));
        upcastedEvents.add(new SimpleSerializedObject>Document>(
                insurancePolicyUpdatedEvent, Document.class, expectedTypes.get(1)));
        return upcastedEvents;
    }

    @Override
    public List<SerializedType> upcast(SerializedType serializedType) { ❺
        SerializedType addressUpdatedEventType =
                new SimpleSerializedType("org.example.AddressUpdatedEvent", "1");
        SerializedType insurancePolicyUpdatedEventType =
                new SimpleSerializedType("org.example.InsurancePolicyUpdatedEvent", "1");
        return Arrays.asList(addressUpdatedEventType, insurancePolicyUpdatedEventType);
    }
}
```

❶ First we have to create a class that implements the `Upcaster` interface. Then we have to decide on which content type to work. For this example dom4j documents are used as they'll fit nicely with an event store that uses XML to store events.

❷ In Axon, Events have a revision, if the definition of an event changes, you should update its revision as well. The `canUpcast` method can then be used to check if an event needs to be upcasted. In the example we return true on the `canUpcast` method only when the incoming event has the type `org.example.AdministrativeDetailsUpdatedEvent` and has revision number 0. Axon will take care of calling the Upcaster's `upcast` method if the `canUpcast` method on that Upcaster returns true.

❸ Due to Java's type erasure we have to implement the `expectedRepresentationType` method to provide Axon with runtime type information on the content type by returning the `Class` object of the content type. The content type is used at runtime to determine how the incoming event needs to be converted to provide the upcaster with the event with the correct type.

❹ By copying the address and policy element from the AdministrativeDetailsUpdatedEvent each into its own document, the event is upcasted. Deserializing the result would get us an instance of the InsurancePolicyUpdatedEvent and an instance of the AddressUpdatedEvent.

❺ Upcasting can be expensive, possibly involving type conversion, deserialization and logic. Axon is smart enough to prevent this from happening when it is not neccesary through the concept of SerializedTypes. SerializedTypes provide Axon with the information to delay event upcasting until the application requires it.

5.4.2. The Upcaster Chain

The Upcaster Chain is responsible for upcasting events by chaining the output of one upcaster to the next. It comes in the following two flavours:

- The `SimpleUpcasterChain` immediately upcasts all events given to it and returns them.
- The `LazyUpcasterChain` prepares the events to be upcasted but only upcasts the events that are actually used. Whether or not your application needs all events this can give you a significant performance benefit. In the worst case it's as slow as the SimpleUpcasterChain. The LazyUpcasterChain does not guarantee that all the events in an Event Stream are in fact upcasted. When your upcasters rely on information from previous events, this may be a problem.

The LazyUpcasterChain is a safe choice if your upcasters are stateless or do not depend on other upcasters. Always consider using the LazyUpcasterChain since it can provide a great performance benefit over the SimpleUpcasterChain. If you want guaranteed upcasting in a strict order, use the SimpleUpcasterChain.

5.4.3. Content type conversion

An upcaster works on a given content type (e.g. dom4j Document). To provide extra flexibility between upcasters, content types between chained upcasters may vary. Axon will try to convert between the content types automatically by using ContentTypeConverters. It will search for the shortest path from type `x` to type `y`, perform the conversion and pass the converted value into the requested upcaster. For performance reasons, conversion will only be performed if the `canUpcast` method on the receiving upcaster yields true.

The ContentTypeConverters may depend on the type of serializer used. Attempting to convert a byte[] to a dom4j Document will not make any sence unless a Serializer was used that writes an event as XML. To make sure the UpcasterChain has access to the serializer-specific ContentTypeConverters, you can pass a reference to the serializer to the constructor of the UpcasterChain.

Tip

To achieve the best performance, ensure that all upcasters in the same chain (where one's output is another's input) work on the same content type.

If the content type conversion that you need is not provided by Axon you can always write one yourself using the `ContentTypeConverter` interface.

5.5. Snapshotting

When aggregates live for a long time, and their state constantly changes, they will generate a large amount of events. Having to load all these events in to rebuild an aggregate's state may have a big performance impact. The snapshot event is a domain event with a special purpose: it summarises an arbitrary amount of events into a single one. By regularly creating and storing a snapshot event, the event store does not have to return long lists of events. Just the last snapshot events and all events that occurred after the snapshot was made.

For example, items in stock tend to change quite often. Each time an item is sold, an event reduces the stock by one. Every time a shipment of new items comes in, the stock is incremented by some larger number. If you sell a hundred items each day, you will produce at least 100 events per day. After a few days, your system will spend too much time reading in all these events just to find out wheter it should raise an "ItemOutOfStockEvent". A single snapshot event could replace a lot of these events, just by storing the current number of items in stock.

5.5.1. Creating a snapshot

Snapshot creation can be triggered by a number of factors, for example the number of events created since the last snapshot, the time to initialize an aggregate exceeds a certain threshold, time-based, etc. Currently, Axon provides a mechanism that allows you to trigger snapshots based on an event count threshold.

The `EventCountSnapshotterTrigger` provides the mechanism to trigger snapshot creation when the number of events needed to load an aggregate exceeds a certain threshold. If the number of events needed to load an aggregate exceeds a certain configurable threshold, the trigger tells a `Snapshotter` to create a snapshot for the aggregate.

The snapshot trigger is configured on an Event Sourcing Repository and has a number of properties that allow you to tweak triggering:

- `Snapshotter` sets the actual snapshotter instance, responsible for creating and storing the actual snapshot event;
- `Trigger` sets the threshold at which to trigger snapshot creation;
- `ClearCountersAfterAppend` indicates whether you want to clear counters when an aggregate is stored. The optimal setting of this parameter depends mainly on your caching strategy. If you do not use caching, there is no problem in removing event counts from memory. When an aggregate is loaded, the events are loaded in, and counted again. If you use a cache, however, you may lose track of counters. Defaults to `true` unless the `AggregateCache` or `AggregateCaches` is set, in which case it defaults to `false`.
- `AggregateCache` and `AggregateCaches` allows you to register the cache or caches that you use to store aggregates in. The snapshotter trigger will register itself as a listener on the cache. If any aggregates are evicted, the snapshotter trigger will remove the counters. This optimizes memory usage in the case your application has many aggregates. Do note that the keys of the cache are expected to be the Aggregate Identifier.

A Snapshotter is responsible for the actual creation of a snapshot. Typically, snapshotting is a process that should disturb the operational processes as little as possible. Therefore, it is recommended to run the snapshotter in a different thread. The `Snapshotter` interface declares a single method: `scheduleSnapshot()`, which takes the aggregate's type and identifier as parameters.

Axon provides the `AggregateSnapshotter`, which creates and stores `AggregateSnapshot` instances. This is a special type of snapshot, since it contains the actual aggregate instance within it. The repositories provided by Axon are aware of this type of snapshot, and will extract the aggregate from it, instead of instantiating a new one. All events loaded after the snapshot events are streamed to the extracted aggregate instance.

> **Note**
> Do make sure that the `Serializer` instance you use (which defaults to the `XStreamSerializer`) is capable of serializing your aggregate. The `XStreamSerializer` requires you to use either a Hotspot JVM, or your aggregate must either have an accessible default constructor or implement the `Serializable` interface.

The AbstractSnapshotter provides a basic set of properties that allow you to tweak the way snapshots are created:

- `EventStore` sets the event store that is used to load past events and store the snapshots. This event store must implement the `SnapshotEventStore` interface.
- `Executor` sets the executor, such as a `ThreadPoolExecutor` that will provide the thread to process actual snapshot creation. By default, snapshots are created in the thread that calls the `scheduleSnapshot()` method, which is generally not recommended for production.

The `AggregateSnapshotter` provides on more property:

- `AggregateFactories` is the property that allows you to set the factories that will create instances of your aggregates. Configuring multiple aggregate factories allows you to use a single Snapshotter to create snapshots for a variety of aggregate types. The `EventSourcingRepository` implementations provide access to the `AggregateFactory` they use. This can be used to configure the same aggregate factories int he Snapshotter as the ones used in the repositories.

> **Note**
> If you use an executor that executes snapshot creation in another thread, make sure you configure the correct transaction management for your underlying event store, if necessary. Spring users can use the `SpringAggregateSnapshotter`, which allows you to configure a `PlatformTransactionManager`. The `SpringAggregateSnapshotter` will autowire all aggregate factories (either directly, or via the Repository), if a list is not explicitly configured.

5.5.2. Storing Snapshot Events

All Axon-provided Event Store implementations are capable of storing snapshot events. They provide a special method that allows a `DomainEventMessage` to be stored as a snapshot event. You have to initialize the snapshot event completely, including the aggregate identifier and the sequence number. There is a special constructor on the `GenericDomainEventMessage` for this purpose. The sequence number must be equal to the sequence number of the last event that was included in the state that the snapshot represents. In most cases, you can use the `getVersion()` on the `AggregateRoot` (which each event sourced aggregate implements) to obtain the sequence number to use in the snapshot event.

When a snapshot is stored in the Event Store, it will automatically use that snapshot to summarize all prior events and return it in their place. All event store implementations allow for concurrent creation of snapshots. This means they allow snapshots to be stored while another process is adding Events for the same aggregate. This allows the snapshotting process to run as a separate process alltogether.

Note

Normally, you can archive all events once they are part of a snapshot event. Snapshotted events will never be read in again by the event store in regular operational scenario's. However, if you want to be able to reconstruct aggregate state prior to the moment the snapshot was created, you must keep the events up to that date.

Axon provides a special type of snapshot event: the `AggregateSnapshot`, which stores an entire aggregate as a snapshot. The motivation is simple: your aggregate should only contain the state relevant to take business decisions. This is exactly the information you want captured in a snapshot. All Event Sourcing Repositories provided by Axon recognize the `AggregateSnapshot`, and will extract the aggregate from it. Beware that using this snapshot event requires that the event serialization mechanism needs to be able to serialize the aggregate.

5.5.3. Initializing an Aggregate based on a Snapshot Event

A snapshot event is an event like any other. That means a snapshot event is handled just like any other domain event. When using annotations to demarcate event handers (`@EventHandler`), you can annotate a method that initializes full aggregate state based on a snapshot event. The code sample below shows how snapshot events are treated like any other domain event within the aggregate.

```
public class MyAggregate extends AbstractAnnotatedAggregateRoot {

    // ... code omitted for brevity

    @EventHandler
    protected void handleSomeStateChangeEvent(MyDomainEvent event) {
        // ...
    }

    @EventHandler
    protected void applySnapshot(MySnapshotEvent event) {
        // the snapshot event should contain all relevant state
        this.someState = event.someState;
        this.otherState = event.otherState;
    }
}
```

There is one type of snapshot event that is treated differently: the `AggregateSnapshot`. This type of snapshot event contains the actual aggregate. The aggregate factory recognizes this type of event and extracts the aggregate from the snapshot. Then, all other events are re-applied to the extracted snapshot. That means aggregates never need to be able to deal with `AggregateSnapshot` instances themselves.

5.5.4. Pruning Snapshot Events

Once a snapshot event is written, it prevents older events and snapshot events from being read. Domain Events are still used in case a snapshot event becomes obsolete due to changes in the structure of an aggregate. The older snapshot events are hardly ever needed. `SnapshotEventStore` implementation may choose to keep only a limited amount of snapshots (e.g. only one) for each aggregate.

The `JpaEventStore` allows you to configure the amount of snapshots to keep per aggregate. It defaults to 1, meaning that only the latest snapshot event is kept for each aggregate. Use `setMaxSnapshotsArchived(int)` to change this setting. Use a negative integer to prevent pruning altogether.

5.6. Advanced conflict detection and resolution

One of the major advantages of being explicit about the meaning of changes, is that you can detect conflicting changes with more precision. Typically, these conflicting changes occur when two users are acting on the same data (nearly) simultaneously. Imagine two users, both looking at a specific version of the data. They both decide to make a change to that data. They will both send a command like "on version X of this aggregate, do that", where X is the expected version of the aggregate. One of them will have the changes actually applied to the expected version. The other user won't.

Instead of simply rejecting all incoming commands when aggregates have been modified by another process, you could check whether the user's intent conflicts with any unseen changes. One way to do this, is to apply the command on the latest version of the aggregate, and check the generated events against the events that occurred since the version the user expected. For example, two users look at a Customer, which has version 4. One user notices a typo in the customer's address, and decides to fix it. Another user wants to register the fact that the customer moved to another address. If the fist user applied his command first, the second one will make the change to version 5, instead of the version 4 that he expected. This second command will generate a CustomerMovedEvent. This event is compared to all unseen events: AddressCorrectedEvent, in this case. A ConflictResolver will compare these events, and decide that these conflicts may be merged. If the other user had committed first, the ConflictResolver would have decided that a AddressCorrectedEvent on top of an unseen CustomerMovedEvent is considered a conflicting change.

Axon provides the necessary infrastructure to implement advanced conflict detection. By default, all repositories will throw a `ConflictingModificationException` when the version of a loaded aggregate is not equal to the expected version. Event Sourcing Repositories offer support for more advanced conflict detection, as decribed in the paragraph above.

To enable advanced conflict detection, configure a `ConflictResolver` on the `EventSourcingRepository`. This `ConflictResolver` is responsible for detecting conflicting modifications, based on the events representing these changes. Detecting these conflicts is a matter of comparing the two lists of DomainEvents provided in the `resolveConflicts` method declared on the `ConflictResolver`. If such a conflict is found, a `ConflictingModificationException` (or better, a more explicit and explanatory subclass of it) must be thrown. If the `ConflictResolver` returns normally, the events are persisted, effectively meaning that the concurrent changes have been merged.

6. Event Processing

The Events generated by the application need to be dispatched to the components that update the query databases, search engines or any other resources that need them: the Event Listeners. This is the responsibility of the Event Bus.

6.1. Event Bus

The `EventBus` is the mechanism that dispatches events to the subscribed event listeners. Axon Framework provides two implementation of the event bus: `SimpleEventBus` and `ClusteringEventBus`. Both implementations manage subscribed `EventListeners` and forward all incoming events to all subscribed listeners. This means that Event Listeners must be explicitly registered with the Event Bus in order for them to receive events. The registration process is thread safe. Listeners may register and unregister for events at any time.

6.1.1. Simple Event Bus

The `SimpleEventBus` is, as the name suggests, a very basic implementation of the `EventBus` interface. It just dispatches each incoming `Event` to each of the subscribed `EventListeners` sequentially. If an EventListener throws an `Exception`, dispatching stops and the exception is propagated to the component publising the `Event`.

The `SimpleEventBus` is suitable for most cases where dispatching is done synchronously and locally, (i.e. in a single JVM). Once you application requires `Events` to be published across multiple JVMs, you could consider using the `ClusteringEventBus` instead.

6.1.2. Clustering Event Bus

The `ClusteringEventsBus` allows application developers to bundle `EventListener`s into `Cluster`s based on their properties and non-functional requirements. The ClusteringEventBus is also more capable to deal with Events being dispatched among different machines.

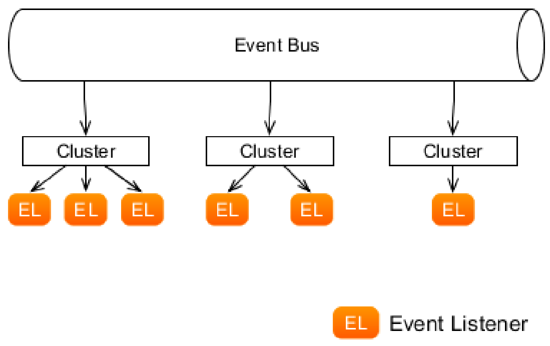

Figure 6.1. Structure of the Clustering Event Bus

The ClusteringEventsBus contains two mechanisms: the `ClusterSelector`, which selects a `Cluster` instance for each of the registered `EventListener`s, and the `EventBusTerminal`, which is responsible for dispatching Events to each of the relevant clusters.

Background: Axon Terminal

In the nervous system, an Axon (one of the components of a Neuron) transports electrical signals. These Neurons are interconnected in very complex arrangements. The Axon Terminal is responsible for transmitting these signals from one Neuron to another.

More information: www.wikipedia.org/wiki/Axon_terminal.

`ClusterSelector`

The primary responsibility of the `ClusterSelector` is to, as the name suggests, select a cluster for each Event Listener that subscribes to the Event Bus. By default, all Event Listeners are placed in a single Cluster instance, which dispatches events to its members sequentially and in the calling thread (similar to how the `SimpleEventBus` works). By providing a custom implementation, you can arrange the Event Listeners into different Cluster instances to suit the requirements of your architecture.

A number of `ClusterSelector` implementations are available. The `ClassNamePrefixClusterSelector`, for example, uses a mapping of package prefixes to decide which cluster is (most) suitable for an Event Listener. Similarly, the `ClassNamePatternClusterSelector` uses pattern matching to decide whether a given cluster is suitable. You can use the `CompositeClusterSelector` to combine several cluster selectors into a single one.

The `Cluster` interface describes the behavior of a Cluster. By adding information in the Meta Data of a Cluster, the selector can provide hints to the Terminal about the expected behavior.

Clusters and Cluster Selectors in Spring

Spring users can define clusters using the <axon:cluster> element. This element allows you to define the selection criteria for Event Handlers in that cluster. These criteria are transformed into cluster selectors and used to assign each Listener to one of the clusters. By default, this creates a cluster that handles events in the publishing thread. To use a cluster with other semantics, you can define a bean inside the <axon:cluster> element that specifies the Cluster implementation to use.

The Clusters are automatically detected and connected to the Event Bus in the application context.

`EventBusTerminal`

The `EventBusTerminal` forms the bridge between the Clusters inside the Event Bus. While some terminals will dispatch within the same JVM, others are aware of messaging technologies, such as AMQP to dispatch Event Messages to clusters on remote machines. The default implementation dispatches published events to each of the (local) clusters using the publishing thread. This means that with the default terminal, and the default `ClusterSelector`, the behavior of the `ClusteringEventBus` is exactly the same as that of the `SimpleEventBus`.

In a typical AMQP based configuration, the `EventBusTerminal` would send published events to an Exchange. For each cluster, a Queue would be connected to that exchange. The `EventBusTerminal` will create a consumer for each cluster, which reads from its related Queue and forwards each message to that cluster. Event Listeners in a distributed environment where at most one instance should receive an Events should be placed in a separate cluster, which competes with the other instances on a single Queue. See Section 6.4, "Distributing the Event Bus" for more information.

6.2. Event Listeners

Event listeners are the component that act on incoming events. Events may be of any type. In the Axon Framework, all event listeners must implement the `EventListener` interface.

6.2.1. Basic implementation

Event listeners need to be registered with an event bus (see Section 6.1, "Event Bus") to be notified of events. The EventListener interface prescribes a single method to be implemented. This method is invoked for each Event Message passed on the Event Bus that it is subscribed to:

```
public class MyEventListener implements EventListener {

    public void handle(EventMessage message) {
        if (SomeEvent.class.isAssignableFrom(message.getPayloadType) {
            // handle SomeEvent
        } else if (OtherEvent.class.isAssignableFrom(message.getPayloadType) {
            // handle SomeOtherEvent
        }
    }
}
```

6.2.2. Annotated Event Handler

Implementing the EventListener interface can produce a large if-statement and verbose plumbing code. Using annotations to demarcate Event Handler methods is a cleaner alternative.

`AnnotationEventListenerAdapter`

The `AnnotationEventListenerAdapter` can wrap any object into an event listener. The adapter will invoke the most appropriate event handler method available. These event handler methods must be annotated with the `@EventHandler` annotation.

The `AnnotationEventListenerAdapter`, as well as the `AbstractAnnotatedAggregateRoot`, use `ParameterResolver`s to resolve the value that should be passed in the parameters of methods annotated with `@EventHandler`. By default, Axon provides a number of parameter resolvers that allow you to use the following parameter types:

- The first parameter is always the payload of the Event message
- Parameters annotated with `@MetaData` will resolve to the Meta Data value with the key as indicated on the annotation. If `required` is `false` (default), `null` is passed when the meta data value is not present. If `required` is `true`, the resolver will not match and prevent the method from being invoked when the meta data value is not present.
- Parameters of type `MetaData` will have the entire `MetaData` of an `EventMessage` injected.
- Parameters annotated with `@Timestamp` and of type `org.joda.time.DateTime` will resolve to the timestamp of the `EventMessage`. This is the time at which the Event was generated.
- Parameters assignable to Message will have the entire `EventMessage` injected (if the message is assignable to that parameter). If the first parameter is of type message, it effectively matches an Event of any type, even if generic parameters would suggest otherwise. Due to type erasure, Axon cannot detect what parameter is expected. In such case, it is best to declare a parameter of the payload type, followed by a parameter of type Message.
- When using Spring and `<axon:annotation-config/>` is declared, any other parameters will resolve to autowired beans, if exactly one autowirable candidate is available in the application context. This allows you to inject resources directly into `@EventHandler` annotated methods.

You can configure additional `ParameterResolver`s by extending the `ParameterResolverFactory` class and creating a file named `/META-INF/service/org.axonframework.common.annotation.ParameterResolverFactory` containing the fully qualified name of the implementing class. Alternatively, you can register your implementation at runtime using `ParameterResolverFactory.registerFactory()`. Make sure to do so before any adapters are created, otherwise handlers may have been initialized without those parameter resolvers.

In all circumstances, exactly one event handler method is invoked per listener instance. Axon will search the most specific method to invoke, in the following order:

1. On the actual instance level of the class hierarchy (as returned by `this.getClass()`), all annotated methods are evaluated
2. If one or more methods are found of which all parameters can be resolved to a value, the method with the most specific type is chosen and invoked
3. If no methods are found on this level of the class hierarchy, the super type is evaluated the same way
4. When the top level of the hierarchy is reached, and no suitable event handler is found, the event is ignored.

```
// assume EventB extends EventA
// and    EventC extends EventB

public class TopListener {

    @EventHandler
    public void handle(EventA event) {
    }

    @EventHandler
    public void handle(EventC event) {
    }
}

public class SubListener extends TopListener {

    @EventHandler
    public void handle(EventB event) {
    }

}
```

In the example above, the `SubListener` will receive all instances of `EventB` as well as `EventC` (as it extends `EventB`). In other words, the `TopListener` will not receive any invocations for `EventC` at all. Since `EventA` is not assignable to `EventB` (it's its superclass), those will be processed by `TopListener`.

The constructor of the `AnnotationEventListenerAdapter` takes two parameters: the annotated bean, and the `EventBus`, to which the listener should subscribe. You can subscribe and unsubscribe the event listener using the `subscribe()` and `unsubscribe()` methods on the adapter. Alternatively, you can use `AnnotationEventListenerAdapter.subscribe(listener, eventBus)` to create and subscribe the listener in one invocation.

Tip
If you use Spring, you can automatically wrap all annotated event listeners with an adapter automatically by adding `<axon:annotation-config/>` to your application context. Axon will automatically find and wrap annotated event listeners in the Application Context with an `AnnotationEventListenerAdapter` and register them with the Event Bus.

6.3. Asynchronous Event Processing

By default, event listeners process events in the thread that dispatches them. This means that the thread that executes the command will have to wait untill all event handling has finished. For some types of event listeners this is not the optimal form of processing. Asynchronous event processing improves the scalability of the application, with the penalty of added complexity to deal with "eventual consistency". Axon Framework provides the `AsynchronousCluster` implementation, which dispatches Events to Event Listeners asynchronously from the thread that published them to the cluster.

Configuring the Asynchronous Cluster in Spring
In Spring, you can place a Spring `<bean>` element inside the `<axon:cluster>` element, to indicate which cluster implementation you wish to use. Simply specify the bean configuration of an `AsynchronousCluster` implementation to make a Cluster asynchronous.

The `AsynchronousCluster` needs an `Executor`, for example a `ThreadPoolExecutor` and a `SequencingPolicy`, a definition of which events may be processed in parallel, and which sequentially. Finally a `TransactionManager` can be provided to process events within a transaction, such as a database transaction.

The `Executor` is responsible for executing the event processing. The actual implementation most likely depends on the environment that the application runs in and the SLA of the event handler. An example is the `ThreadPoolExecutor`, which maintains a pool of threads for the event handlers to use to process events. The `AsynchronousCluster` will manage the processing of incoming events in the provided executor. If an instance of a `ScheduledThreadPoolExecutor` is provided, the `AsynchronousCluster` will automatically leverage its ability to schedule processing in the cases of delayed retries.

The `SequencingPolicy` defines whether events must be handled sequentially, in parallel or a combination of both. Policies return a sequence identifier of a given event. If the policy returns an equal itentifier for two events, this means that they must be handled sequentially be the event handler. A `null` sequence identifier means the event may be processed in parallel with any other event.

Axon provides a number of common policies you can use:

- The `FullConcurrencyPolicy` will tell Axon that this event handler may handle all events concurrently. This means that there is no relationship between the events that require them to be processed in a particular order.
- The `SequentialPolicy` tells Axon that all events must be processed sequentially. Handling of an event will start when the handling of a previous event is finished.
- `SequentialPerAggregatePolicy` will force domain events that were raised from the same aggregate to be handled sequentially. However, events from different aggregates may be handled concurrently. This is typically a suitable policy to use for event listeners that update details from aggregates in database tables.

Besides these provided policies, you can define your own. All policies must implement the `EventSequencingPolicy` interface. This interface defines a single method, `getSequenceIdentifierFor`, that returns the identifier sequence identifier for a given event. Events for which an equal sequence identifer is returned must be processed sequentially. Events that produce a different sequence identifier may be processed concurrently. For performance reasons, policy implementations should return `null` if the event may be processed in parallel to any other event. This is faster, because Axon does not have to check for any restrictions on event processing.

A `TransactionManager` can be assigned to a `AsynchronousCluster` to add transactional processing of events. To optimize processing, events can be processed in small batches inside a transaction. When using Spring, you can use the `SpringTransactionManager` to manage transactions with Spring's `PlatformTransactionManager` . For more customization of transactional behavior, you can alternatively configure a `UnitOfWorkFactory`. That factory will be used to generate the Unit of Work wrapping the Event Handling process. By default, a `DefaultUnitOfWorkFactory` is used, which uses the provided `TransactionManager`, if any, to manage the backing Transactions.

Error handling

The `AsynchronousCluster` uses an `ErrorHandler` to decide what needs to be done when an Event Listener or Unit of Work throws an Exception. The default behavior depends on the availability of a TransactionManager. If a `TransactionManager` is provided, the default `ErrorHandler` will request a rollback and retry the Event Handling after 2 seconds. If no `TransactionManager` is provided, the defautl `ErrorHandler` will simply log the Exception and proceed with the next Event Listener, guaranteeing that each Event Listener will receive each Event. In any situation, a rollback will *not* be requested when the exception is explicitly non-transient (i.e. is caused by an `AxonNonTransientException`).

You can change this behavior by configuring another *ErrorHandler*, or by creating your own. The ErrorHandler interface has a single method, which provides the Exception that occurred, the EventMessage being processed and a reference to the EventListener throwing the exception. The return value is of type RetryPolicy. The RetryPolicy tells the Event Processor what it needs to do with the failure. There are three static methods on RetryPolicy for the most common scenarios:

- `retryAfter(int timeout, TimeUnit unit)` tells the scheduler that the Unit of Work should be rolled back, and the Event Message should be rescheduled for handling after the given amount of time. This means that some Event Listeners may have received the Event more than once.
- `proceed()` tells the scheduler to ignore the Exception and proceed with the processing of the Event Message. This may be with the intent to skip the Event on the Event Listener, or because the `ErrorHandler` has managed to resolve the problem by retrying invoking the `EventHandler` itself.
- `skip()` tells the scheduler to rollback the Unit of Work and proceed with the next Event Message. If all Event Listeners properly support Transactions, will effectively mean that the Event is skipped altogether.

If the `RetryPolicy` you wish to use does not depend on the type of `Exception`, `EventMessage` or `EventListener`, you can use the `DefaultErrorHandler` and pass the desired `RetryPolicy` as its constructor parameter. It will return that `RetryPolicy` on each exception, unless it requests a retry of an Event that caused an explicitly non-transient exception.

6.4. Distributing the Event Bus

In a distributed environment, it may be necessary to transport Event Messages between JVM's. The `ClusteringEventBus` has a possiblity to define an `EventBusTerminal`. This is an interface to a mechansim that publishes Events to all relevant clusters. Some EventBusTerminal implementations allow distribution of Events over multiple JVM's.

Background of the name "Terminal"

While most developers association the word "terminal" to a thin client computer connected to a mainframe, the association to make here is slightly different. In Neurology, an Axon Terminal is an endpoint of an Axon that transmits electronic impulses from one Neuron to another.

For more detailed information, see http://en.wikipedia.org/wiki/Axon_terminal.

6.4.1. Spring AMQP Terminal

The Spring AMQP Terminal uses the Spring AMQP module to transmit events to an AMQP compatible message broker, such as Rabbit MQ. It also connects local clusters to queues on that message broker.

The `axon-amqp` namespace (`http://www.axonframework.org/schema/amqp`) allows you to configure an AMQP Terminal by adding the `<axon-amqp:terminal>` element to the Spring application context. On this element, you can define different properties for the terminal, as well as a configuration containing defaults to use to connect each cluster to an AMQP Queue.

The example below shows an example configuration for an AMQP Terminal. The `default-configuration` element specified the defaults for the Clusters if they don't provide their own values.

```
<axon-amqp:terminal id="terminal"
                    connection-factory="amqpConnection"
                    serializer="serializer"
                    exchange-name="AxonEventBusExchange">
    <axon-amqp:default-configuration transaction-manager="transactionManager"
                                     transaction-size="25" prefetch="200"
                                     error-handler="loggingErrorHandler"/>
</axon-amqp:terminal>

<bean id="amqpConnection" class="org.springframework.amqp.rabbit.connection.CachingConnectionFactory"/>
```

The configure the Spring AMQP Terminal "manually", you need to specify a number of beans in your application context:

- The `ListenerContainerLifecycleManager` is responsible for creating ListenerContainers. These are the Spring classes that listen for messages on the AMQP Queues and forward them to the processing components. The `ListenerContainerLifecycleManager` allows you to configure the number of messages to process in a single transaction, the number of messages it may read ahead, etc.

  Note that the `ListenerContainerLifecycleManager` must be defined as a top-level bean.

- An AMQP `ConnectionFactory`, which creates the connections to the AMQP Message Broker. Spring provides the `CachingConnectionFactory`, which is a sensible default.

- The Spring AMQPTerminal itself, which connects the aforementioned components to the event publishers and event listeners. There is a large number of configuration options that allow you to tweak the terminal's behavior:

  - transactional: indicates whether the messages should be dispatched to the AMQP Broker inside a transaction. This is especially useful when multiple events need to be sent either completely, or not at all.

  - durable: indicates whether messsages should be durable (i.e. survive a Broker shutdown) or not. Obviously, message durability involves a performance impact.

- connectionFactory: configures the ConnectionFactory to use. Useful when the application context contains more than one instance. Otherwise, the only available instance is autowired.
- serializer: the serializer to serialize the MetaData and Payload of EventMessages with. Defaults to an autowired serializer.
- exchangeName or exchange: defines the exchange (either defined by the name, or by a reference to an Exchange bean) to which published Event Messages should be sent. Defaults to "Axon.EventBus"
- queueNameResolver: defines the mechanism that chooses the Queue that each Cluster should be connected to. By default, the resolver will use the configuration provided in each Cluster's Meta Data under the name "AMQP.Config". Otherwise, it uses the Cluster's name as Queue Name.
- routingKeyResolver: defines the mechanism that generates an AMQP routing key for an outgoing Message. Defaults to a routing key resolver that returns the package name of the Message's payload. Routing keys can be used by echanges to define which queues should receive a (copy of a) Message
- listenerContainerLifecycleManager: when the application context contains more than one, defines which listenerContainerLifecycleManager instance to use.
- exclusive: indicates whether this Cluster accepts to share a Queue with other Cluster instances. Default to `true`. If a second cluster is attampting to connect exclusively to a queue, an exception is thrown. The Connector catches this exception and reattempts to connect each 2 seconds. This allows for automatic failover when a machine drops its connection.

When a cluster is selected for an Event Listener, it will be registered with the terminal. At that point, the Spring AMQP terminal will check if there is any cluster-specific configuration available. It does so by checking the `AMQP.Config` MetaData value. If that value is an instance of `AMQPConsumerConfiguration` (such as `SpringAMQPConsumerConfiguration`) any settings configured there will override the defaults set on the terminal itself. This allows you to specify different behavior (such as transaction size) for different clusters.

```
// XML Configuration for a Cluster with AMQPConsumerConfiguration
<axon:cluster id="myDefaultCluster" default="true">
    <axon:meta-data>
        <entry key="AMQP.Config">
            <axon-amqp:configuration transaction-size="20000"/>
        </entry>
    </axon:meta-data>
</axon:cluster>
```

6.5. Replaying Events on a Cluster

One of the advantages of Event Sourcing is that you keep track of the entire history of the application. This history allows you to extract valuable information and build new models out of them. For example, when a screen is added to the application, you can create a new query model and database tables, and have these filled using events you have collected in the past. Sometimes, replays of the Event Store are also needed to fix data that has been corrupted due to a bug.

Configuration of Replays

Axon provides the `ReplayingCluster`, a wrapper around another `Cluster` implementation that adds the replay capability. The ReplayingCluster is initialized with a number of resources. First of all, it needs another Cluster implementation. That other cluster is the actual cluster that takes care of dispatching Event Messages to subscribed listeners. It also needs a reference to the Event Store (implementing `EventStoreManagement`) that will supply the Events for the replay. A transaction manager is used to wrap the replay in a transaction. Since a single transaction may be too large to be efficient, you can configure a "commit threshold" to indicate the number of messages that should be processed before performing an intermediate commit. Finally, you need to supply an `IncomingMessageHandler`. The `IncomingMessageHandler` tells the ReplayingCluster what to do when an Event Message is published to the cluster while it is replaying.

Warning
Make sure not to replay onto Clusters that contain Event Listeners that do not support replaying. A typical example is an Event Listener that sends emails. Replaying on a cluster that contains such an Event Listener can have nasty side-effects.

Axon provides two `IncomingMessageHandler` implementations. The `BackloggingIncomingMessageHandler` simply backlogs any incoming events (in-memory) and postpones processing of these events until the replay is finished. If an event is backlogged, but was also part of the replay, it is automatically removed from the backlog to prevent duplicate processing. The other implementation is the `DiscardingIncomingMessageHandler`. As the name suggests, it simply discards any messages published during a replay. This implementation will ensure the fastest replay, but is not safe to use when you expect messages to be published to the cluster during the replay. You can also create your own implementation. The JavaDoc describes the requirements (incl. thread safety) for each method.

Note

Although it is technically possible to do a full replay at runtime, it should be considered a maintenance operation and be executed while the cluster is not in active use by the application.

ReplayCluster configuration in Spring

In Spring, a Cluster can be marked as replayable by adding the `<axon:replay-config>` element as a child if the `<axon:cluster>` element. When the replay-config element is present, Axon will automatically wrap the cluster in a `ReplayingCluster` using the provided configuration. This also means that `applicationContext.getBean(clusterName)` will return a bean of type `ReplayingCluster`.

Preparing for a replay

In many cases, the data source used by Event Listeners needs to be prepared for a replay. Database tables, for example, typically need to be cleared. Event Listeners can implement the `ReplayAware` interface. When they do, their `beforeReplay` and `afterReplay` will be invoked before and after the replay respectively. Both methods are invoked within the scope of a transaction.

Triggering a Replay

Axon does not automatically trigger a replay. The ReplayingCluster provides two methods that can start a replay: `startReplay()` and `startReplay(Executor)`. The first will execute the replay in the calling thread, meaning that the call will return when the replay is finished. The latter will execute the replay using the given executor and return a Future object that allows the caller to check if the replay is finished.

Note

Note that a replay may take a long time to finish, depending on the number of Events that need to be processed. Therefore, ensure that it is not possible to rebuild the model using other models already available, which is typically faster. Also make sure to properly test a replay before applying it in a production environment.

7. Managing complex business transactions

Not every command is able to completely execute in a single ACID transaction. A very common example that pops up quite often as an argument for transactions is the money transfer. It is often believed that an atomic and consistent transaction is absolutely required to transfer money from one account to another. Well, it's not. On the contrary, it is quite impossible to do. What if money is transferred from an account on Bank A, to another account on Bank B? Does Bank A acquire a lock in Bank B's database? If the transfer is in progress, is it strange that Bank A has deducted the amount, but Bank B hasn't deposited it yet? Not really, it's "underway". On the other hand, if something goes wrong while depositing the money on Bank B's account, Bank A's customer would want his money back. So we do expect some form of consistency, ultimately.

While ACID transactions are not necessary or even impossible in some cases, some form of transaction management is still required. Typically, these transactions are referred to as BASE transactions: **B**asic **A**vailability, **S**oft state, **E**ventual consistency. Contrary to ACID, BASE transactions cannot be easily rolled back. To roll back, compensating actions need to be taken to revert anything that has occurred as part of the transaction. In the money transfer example, a failure at Bank B to deposit the money, will refund the money in Bank A.

In CQRS, Sagas are responsible for managing these BASE transactions. They respond on Events produced by Commands and may produce new commands, invoke external applications, etc. In the context of Domain Driven Design, it is not uncommon for Sagas to be used as coordination mechanism between several bounded contexts.

7.1. Saga

A Saga is a special type of Event Listener: one that manages a business transaction. Some transactions could be running for days or even weeks, while others are completed within a few milliseconds. In Axon, each instance of a Saga is responsible for managing a single business transaction. That means a Saga maintains state necessary to manage that transaction, continuing it or taking compensating actions to roll back any actions already taken. Typically, and contrary to regular Event Listeners, a Saga has a starting point and an end, both triggered by Events. While the starting point of a Saga is usually very clear, while there could be many ways for a Saga to end.

In Axon, all Sagas must implement the `Saga` interface. As with Aggregates, there is a Saga implementation that allows you to annotate event handling methods: the `AbstractAnnotatedSaga`.

7.1.1. Life Cycle

As a single Saga instance is responsible for managing a single transaction. That means you need to be able to indicate the start and end of a Saga's Life Cycle.

The `AbstractAnnotatedSaga` allows you to annotate Event Handlers with an annotation (`@SagaEventHandler`). If a specific Event signifies the start of a transaction, add another annotation to that same method: `@StartSaga`. This annotation will create a new saga and invoke its event handler method when a matching Event is published.

By default, a new Saga is only started if no suitable existing Saga (of the same type) can be found. You can also force the creation of a new Saga instance by setting the `forceNew` property on the `@StartSaga` annotation to `true`.

Ending a Saga can be done in two ways. If a certain Event always indicates the end of a Saga's life cycle, annotate that Event's handler on the Saga with `@EndSaga`. The Saga's Life Cycle will be ended after the invocation of the handler. Alternatively, you can call `end()` from inside the Saga to end the life cycle. This allows you to conditionally end the Saga.

Note

If you don't use annotation support, you need to properly configure your Saga Manager (see Section 7.2.1, “Saga Manager” below). To end a Saga's life cycle, make sure the `isActive()` method of the Saga returns `false`.

7.1.2. Event Handling

Event Handling in a Saga is quite comparable to that of a regular Event Listener. The same rules for method and parameter resolution are valid here. There is one major difference, though. While there is a single instance of an Event Listener that deals will all incoming events, multiple instances of a Saga may exist, each interested in different Events. For example, a Saga that manages a transaction around an Order with Id "1" will not be interested in Events regarding Order "2", and vice versa.

Using association values

Instead of publishing all Events to all Saga instances (which would be a complete waste of resources), Axon will only publish Events containing properties that the Saga has been associated with. This is done using `AssociationValue`s. An `AssociationValue` consists of a key and a value. The key represents the type of identifier used, for example "orderId" or "order". The value represents the corresponding value, "1" or "2" in the previous example.

The `@SagaEventHandler` annotation has two attributes, of which `associationProperty` is the most important one. This is the name of the property on the incoming Event that should be used to find associated Sagas. The key of the association value is the name of the property. The value is the value returned by property's getter method.

For example, consider an incoming Event with a method "`String getOderId()`", which returns "123". If a method accepting this Event is annotated with `@SagaEventHandler(associationProperty="orderId")`, this Event is routed to all Sagas that have been associated with an `AssociationValue` with key "orderId" and value "123". This may either be exactly one, more than one, or even none at all.

Sometimes, the name of the property you want to association is not the name of the association you want to use. For example, you have a Saga that matches Sell orders against Buy orders, you could have a Transaction object that contains the "buyOrderId" and a "sellOrderId". If you want the saga to associate that value as "orderId", you can define a different keyName in the `@SagaEventHandler` annotation. It would then become `@SagaEventHandler(associationProperty="sellOrderId", keyName="orderId")`

Associating Sagas with Domain Concepts

When a Saga manages a transaction around one or more domain concepts, such as Order, Shipment, Invoice, etc, that Saga needs to be associated with instances of those concepts. An association requires two parameters: the key, which identifies the type of association (Order, Shipment, etc) and a value, which represents the identifier of that concept.

Associating a Saga with a concept is done in several ways. First of all, when a Saga is newly created when invoking a `@StartSaga` annotated Event Handler, it is automatically associated with the property identified in the `@SagaEventHandler` method. Any other association can be created using the `associateWith(String key, String/Number value)` method. Use the `removeAssociationWith(String key, String/Number value)` method to remove a specific association.

Imagine a Saga that has been created for a transaction around an Order. The Saga is automatically associated with the Order, as the method is annotated with `@StartSaga`. The Saga is responsible for creating an Invoice for that Order, and tell Shipping to create a Shipment for it. Once both the Shipment have arrived and the Invoice has been paid, the transaction is completed and the Saga is closed.

Here is the code for such a Saga:

```
public class OrderManagementSaga extends AbstractAnnotatedSaga {

    private boolean paid = false;
    private boolean delivered = false;
    private transient CommandGateway commandGateway;

    @StartSaga
    @SagaEventHandler(associationProperty = "orderId")
    public void handle(OrderCreatedEvent event) {
        // client generated identifiers ❶
        ShippingId shipmentId = createShipmentId();
        InvoiceId invoiceId = createInvoiceId();
        // associate the Saga with these values, before sending the commands ❷
        associateWith("shipmentId", shipmentId);
        associateWith("invoiceId", invoiceId);
        // send the commands
        commandGateway.send(new PrepareShippingCommand(...));
        commandGateway.send(new CreateInvoiceCommand(...));
    }

    @SagaEventHandler(associationProperty = "shipmentId")
    public void handle(ShippingArrivedEvent event) {
        delivered = true;
        if (paid) {
            end(); ❸
        }
    }

    @SagaEventHandler(associationProperty = "invoiceId")
    public void handle(InvoicePaidEvent event) {
        paid = true;
        if (delivered) {
            end(); ❹
        }
    }

    // ...

}
```

❶	By allowing clients to generate an identifier, a Saga can be easily associated with a concept, without the need to a request-response type command.
❷	We associate the event with these concepts before publishing the command. This way, we are guaranteed to also catch events generated as part of this command.
❸❹	This will end this saga once the invoice is paid and the shipment has arrived.

Of course, this Saga implementation is far from complete. What should happen if the invoice is not paid in time. What if the shipment cannot be delivered? The Saga should be able to cope with those scenarios as well.

7.1.3. Keeping track of Deadlines

It is easy to make a Saga take action when something happens. After all, there is an Event to notify the Saga. But what if you want your Saga to do something when *nothing* happens? That's what deadlines are used for. In invoices, that's typically several weeks, while the confirmation of a credit card payment should occur within a few seconds.

In Axon, you can use an `EventScheduler` to schedule an Event for publication. In the example of an Invoice, you'd expect that invoice to be paid within 30 days. A Saga would, after sending the `CreateInvoiceCommand`, schedule an `InvoicePaymentDeadlineExpiredEvent` to be published in 30 days. The EventScheduler returns a `ScheduleToken` after scheduling an Event. This token can be used to cancel the schedule, for example when a payment of an Invoice has been received.

Axon provides two EventScheduler implementations: a pure Java one and one using Quartz 2 as a backing scheduling mechanism.

`SimpleEventScheduler`

This pure-Java implementation of the `EventScheduler` uses a `ScheduledExecutorService` to schedule Event publication. Although the timing of this scheduler is very reliable, it is a pure in-memory implementation. Once the JVM is shut down, all schedules are lost. This makes this implementation unsuitable for long-term schedules.

The `SimpleEventScheduler` needs to be configured with an `EventBus` and a `SchedulingExecutorService` (see the static methods on the `java.util.concurrent.Executors` class for helper methods).

`QuartzEventScheduler`

The `QuartzEventScheduler` is a more reliable and enterprise-worthy implementation. Using Quartz as underlying scheduling mechanism, it provides more powerful features, such as persistence, clustering and misfire management. This means Event publication is guaranteed. It might be a little late, but it will be published.

It needs to be configured with a Quartz `Scheduler` and an `EventBus`. Optionally, you may set the name of the group that Quartz jobs are scheduled in, which defaults to "AxonFramework-Events".

Scheduled Events and Transactions

One or more components will be listening for scheduled Events. These components might rely on a Transaction being bound to the Thread that invokes them. Scheduled Events are published by Threads managed by the `EventScheduler`. To manage transactions on these threads, you can configure a `TransactionManager` or a`UnitOfWorkFactory` that creates Transaction Bound Unit of Work.

> **Note**
> Spring users can use the `QuartzEventSchedulerFactoryBean` or `SimpleEventSchedulerFactoryBean` for easier configuration. It allows you to set the PlatformTransactionManager directly.

7.1.4. Injecting Resources

Sagas generally do more than just maintaining state based on Events. They interact with external components. To do so, they need access to the Resources necessary to address to components. Usually, these resources aren't really part of the Saga's state and shouldn't be persisted as such. But once a Saga is reconstructed, these resources must be injected before an Event is routed to that instance.

For that purpose, there is the `ResourceInjector`. It is used by the `SagaRepository` (for existing Saga instances) and the `SagaFactory` (for newly created instances) to inject resources into a Saga. Axon provides a `SpringResourceInjector`, which injects annotated fields and methods with Resources from the Application Context, and a `SimpleResourceInjector`, which detects setters and injects resources which have been registered with it.

> **Mark fields holding injected resources transient**
> Since resources should not be persisted with the Saga, make sure to add the `transient` keyword to those fields. This will prevent the serialization mechanism to attempt to write the contents of these fields to the repository. The repository will automatically re-inject the required resources after a Saga has been deserialized.

SimpleResourceInjector

The `SimpleResourceInjector` uses, as the name suggests, a simple mechanism to detect required resources. It scans the methods of a Saga to find one that starts with "set" and contains a single parameter, of a type that matches with of its known resources.

The `SimpleResourceInjector` is initialized with a collection of objects that a Saga may depend on. If a Saga has a setter method (name starts with "set" and has a single parameter) in which a resource can be passed, it will invoke that method.

SpringResourceInjector

The `SpringResourceInjector` uses Spring's dependency injection mechanism to inject resources into an aggregate. This means you can use setter injection or direct field injection if you require. The method or field to be injected needs to be annotated in order for Spring to recognize it as a dependency, for example with `@Autowired`.

Note that when you use the Axon namespace in Spring to create a Saga Repository or Saga Manager, the `SpringResourceInjector` is configured by default. For more information about wiring Sagas with Spring, see Section 9.9, “Configuring Sagas”.

7.2. Saga Infrastructure

Events need to be redirected to the appropriate Saga instances. To do so, some infrastructure classes are required. The most important components are the `SagaManager` and the `SagaRepository`.

7.2.1. Saga Manager

The `SagaManager` is responsible for redirecting Events to the appropriate Saga instances and managing their life cycle. There are two `SagaManager` implementations in Axon Framework: the `AnnotatedSagaManager`, which provides the annotation support and the `SimpleSagaManager`, which is less powerful, but doesn't force you into using annotations.

Sagas operate in a highly concurrent environment. Multiple Events may reach a Saga at (nearly) the same time. This means that Sagas need to be thread safe. By default, Axon's `SagaManager` implementations will synchronize access to a Saga instance. This means that only one thread can access a Saga at a time, and all changes by one thread are guaranteed to be visible to any successive threads (a.k.a happens-before order in the Java Memory Model). Optionally, you may switch this locking off, if you are sure that your Saga is completely thread safe on its own. Just `setSynchronizeSagaAccess(false)`. When disabling synchronization, do take note of the fact that this will allow a Saga to be invoked while it is in the process of being stored by a repository. The result may be that a Saga is stored in an inconsistent state first, and overwritten by it's actual state later.

SimpleSagaManager

This is by far the least powerful of the two implementations, but it doesn't require the use of annotations. The `SimpleSagaManager` needs to be configured with a number of resources. Its constructor requires the type of Saga it manages, the `SagaRepository`, an `AssociationValueResolver`, a `SagaFactory` and the `EventBus`. The `AssociationValueResolver` is a component that returns a `Set` of `AssociationValue` for a given Event.

Then, you should also configure the types of Events the SagaManager should create new instances for. This is done through the `setEventsToAlwaysCreateNewSagasFor` and `setEventsToOptionallyCreateNewSagasFor` methods. They both accept a List of Event classes.

AnnotatedSagaManager

This SagaManager implementation uses annotations on the Sagas themselves to manage the routing and life cycle of that Saga. As a result, this manager allows all information about the life cycle of a Saga to be available inside the Saga class itself. It can also manage any number of saga types. That means only a single AnnotatedSagaManager is required, even if you have multiple types of Saga.

The `AnnotatedSagaManager` is constructed using a SagaRepository, a SagaFactory (optional) and a vararg array of Saga classes. If no `SagaFactory` is provided, a `GenericSagaFactory` is used. It assumes that all Saga classes have a public no-arg constructor.

If you use Spring, you can use the `axon` namespace to configure an AnnotatedSagaManager. The supported Saga types are provided as a comma separated list. This will also automatically configure a SpringResourceInjector, which injects any annotated fields with resources from the Spring Application Context.

```
<axon:saga-manager id="sagaManager" saga-factory="optionalSagaFactory"
                   saga-repository="sagaRepository" event-bus="eventBus">
    <axon:types>
        fully.qualified.ClassName,
        another.fq.ClassName
    </axon:types>
</axon:saga-manager>
```

Asynchronous Event Handling for Sagas

As with Event Listeners, it is also possible to asynchronously handle events for sagas. To handle events asynchronously, the SagaManager needs to be configured with an `Executor` implementation. The `Executor` supplies the threads needed to process the events asynchronously. Often, you'll want to use a thread pool. You may, if you want, share this thread pool with other asynchronous activities.

When an executor is provided, the SagaManager will automatically use it to find associated Saga instances and dispatch the events each of these instances. The SagaManager will guarantee that for each Saga instance, all events are processed in the order they arrive. For optimization purposes, this guarantee does not count in between Sagas.

Because Transactions are often Thread bound, you may need to configure a Transaction Manager with the SagaManager. This transaction manager is invoked before and after each invocation to the Saga Repository and before and after each batch of Events has been processed by the Saga itself.

In a Spring application context, a Saga Manager can be marked as asynchronous by adding the `executor` and optionally the `transaction-manager` attributes to the `saga-manager` element, as shown below. The `processor-count` attribute defines the number of threads that should process the sagas.

```
<axon:saga-manager id="sagaManager" saga-factory="optionalSagaFactory"
                   saga-repository="sagaRepository" event-bus="eventBus">
    <axon:async processor-count="10" executor="myThreadPool" transaction-manager="txManager"/>
    <axon:types>
        fully.qualified.ClassName,
        another.fq.ClassName
    </axon:types>
</axon:saga-manager>
```

The transaction-manager should point to a `PlatformTransactionManager`, Spring's interface for transaction managers. Generally you can use the same transaction manager as the other components in your application (e.g. `JpaTransactionManager`).

Exceptions while processing Events

Since the processing is asynchronous, Exceptions cannot be propagated to the components publishing the Event. Exceptions raised by Sagas while handling an Event are logged and discarded. When an exception occurs while persisting Saga state, the SagaManager will retry persisting the Saga at a later stage. If there are incoming Events, they are processed using the in-memory representation, re-attemtping the persistence later on.

When shutting down the application, the SagaManager will do a final attempt to persist the Sagas to their repository, and give up if it does not succeed, allowing the application to finish the shutdown process.

7.2.2. Saga Repository

The `SagaRepository` is responsible for storing and retrieving Sagas, for use by the `SagaManager`. It is capable of retrieving specific Saga instances by their identifier as well as by their Association Values.

There are some special requirements, however. Since concurrency in Sagas is a very delicate procedure, the repository must ensure that for each conceptual Saga instance (with equal identifier) only a single instance exists in the JVM.

Axon provides three `SagaRepository` implementations: the `InMemorySagaRepository`, the `JpaSagaRepository` and the `MongoSagaRepository`.

`InMemorySagaRepository`

As the name suggests, this repository keeps a collection of Sagas in memory. This is the simplest repository to configure and the fastest to use. However, it doesn't provide any persistence. If the JVM is shut down, any stored Saga is lost. This implementation is particularly suitable for testing and some very specialized use cases.

JpaSagaRepository

The `JpaSagaRepository` uses JPA to store the state and Association Values of Sagas. Saga's do no need any JPA annotations; Axon will serialize the sagas using a `Serializer` (similar to Event serialization, you can use either a `JavaSerializer` or an `XStreamSerializer`).

The JpaSagaRepository is configured with a JPA `EntityManager`, a `ResourceInjector` and a `Serializer`. Optionally, you can choose whether to explicitly flush the `EntityManager`after each operation. This will ensure that data is sent to the database, even before a transaction is committed. the default is to use explicit flushes.

MongoSagaRepository

Similar to the `JpaSagaRepository`, the `MongoSagaRepository` stores the Saga instances and their associations in a database. The `MongoSagaRepository` stores sagas in a single Collection in a MongoDB database. Per Saga instance, a single document is created.

The MongoSagaRepository also ensures that at any time, only a single Saga instance exists for any unique Saga in a single JVM. This ensures that no state changes are lost due to concurrency issues.

7.2.3. Caching

If a database backed Saga Repository is used, saving and loading Saga instances may be a relatively expensive operation. Especially in situations where the same Saga instance is invoked multiple times within a short timespan, a cache can be beneficial to the application's performance.

Axon provides the `CachingSagaRepository` implementation. It is a Saga Repository that wraps another repository, which does the actual storage. When loading Sagas or Association Values, the `CachingSagaRepository` will first consult its caches, before delegating to the wrapped repository. When storing information, all call are always delegated, to ensure that the backing storage always has a consistent view on the Saga's state.

To configure caching, simply wrap any Saga repository in a `CachingSagaRepository`. The constructor of the CachingSagaRepository takes three parameters: the repository to wrap and the caches to use for the Association Values and Saga instances, respectively. The latter two arguments may refer to the same cache, or to different ones. This depends on the eviction requirements of your specific application.

Spring namespace support

Spring users can use the `<axon:cache-config>` element to add caching behavior to a `<axon:jpa-saga-repository>` element. It is configured with two cache references (which may refer to the same cache): one which stores the cached Saga instances, and one that stores the associations.

```
<axon:jpa-saga-repository id="cachingSagaRepository">
    <axon:cache-config saga-cache="sagaCacheRef" associations-cache="associationCacheRef"/>
</axon:jpa-saga-repository>
```

8. Testing

One of the biggest benefits of CQRS, and especially that of event sourcing is that it is possible to express tests purely in terms of Events and Commands. Both being functional components, Events and Commands have clear meaning to the domain expert or business owner. This means that tests expressed in terms of Events and Commands don't only have a functional meaning, it also means that they hardly depend on any implementation choices.

The features described in this chapter require the `axon-test` module, which can be obtained by configuring a maven dependency (use `<artifactId>axon-test</artifactId>`) or from the full package download.

The fixtures described in this chapter work with any testing framework, such as JUnit and TestNG.

8.1. Command Component Testing

The command handling component is typically the component in any CQRS based architecture that contains the most complexity. Being more complex than the others, this also means that there are extra test related requirements for this component. Simply put: the more complex a component, the better it must be tested.

Although being more complex, the API of a command handling component is fairly easy. It has command coming in, and events going out. In some cases, there might be a query as part of command execution. Other than that, commands and events are the only part of the API. This means that it is possible to completely define a test scenario in terms of events and commands. Typically, in the shape of:

- given certain events in the past,
- when executing this command,
- expect these events to be published and/or stored.

Axon Framework provides a test fixture that allows you to do exactly that. This GivenWhenThenTestFixture allows you to configure a certain infrastructure, composed of the necessary command handler and repository, and express you scenario in terms of given-when-then events and commands.

The following example shows the usage of the given-when-then test fixture with JUnit 4:

```
public class MyCommandComponentTest {

    private FixtureConfiguration fixture;

    @Before
    public void setUp() {
        fixture = Fixtures.newGivenWhenThenFixture(MyAggregate.class); ❶
        MyCommandHandler myCommandHandler = new MyCommandHandler(
                              fixture.getRepository()); ❷
        fixture.registerAnnotatedCommandHandler(myCommandHandler); ❸
    }

    @Test
    public void testFirstFixture() {
        fixture.given(new MyEvent(1)) ❹
               .when(new TestCommand())
               .expectVoidReturnType()
               .expectEvents(new MyEvent(2));
    }
}
```

❶ This line creates a fixture instance that can deal with given-when-then style tests. It is created in configuration stage, which allows us to configure the components that we need to process the command, such as command handler and repository. An event bus and command bus are automatically created as part of the fixture.

❷ The `getRepository()` method returns an `EventSourcingRepository` instance capable of storing `MyAggregate` instances. This requires some conventions on the MyAggregate class, as described in Section 5.2, "Event Sourcing repositories". If there is need for a custom `AggregateFactory`, use the `registerRepository(...)` method to register another repository with the correct `AggregateFactory`.

❸ The `registerAnnotatedCommandHandler` method will register any bean as being an `@CommandHandler` with the command bus. All supported command types are automatically registered with the command bus.

❹ These four lines define the actual scenario and its expected result. The first line defines the events that happened in the past. These events define the state of the aggregate under test. In practical terms, these are the events that the event store returns when an aggregate is loaded. The second line defines the command that we wish to execute against our system. Finally, we have two more methods that define expected behavior. In the example, we use the recommended void return type. The last method defines that we expect a single event as result of the command execution.

The given-when-then test fixture defines three stages: configuration, execution and validation. Each of these stages is represented by a different interface: `FixtureConfiguration`, `TestExecutor` and `ResultValidator`, respectively. The static `newGivenWhenThenFixture()` method on the `Fixtures` class provides a reference to the first of these, which in turn may provide the validator, and so forth.

Note

To make optimal use of the migration between these stages, it is best to use the fluent interface provided by these methods, as shown in the example above.

Configuration

During the configuration phase (i.e. before the first "given" is provided), you provide the building blocks required to execute the test. Specialized versions of the event bus, command bus and event store are provided as part of the fixture. There are getters in place to obtain references to them. The repository and command handlers need to be provided. This can be done using the `registerRepository` and `registerCommandHandler` (or `registerAnnotatedCommandHandler`) methods. If your aggregate allows the use of a generic repository, you can use the `createGenericRepository` method to create a generic repository and register it with the fixture in a single call. The example above uses this feature.

If the command handler and repository are configured, you can define the "given" events. The test fixture will wrap these events as DomainEventMessage. If the "given" event implements Message, the payload and meta data of that message will be included in the DomainEventMessage, otherwise the given event is used as payload. The sequence numbers of the DomainEventMessage are sequential, starting at 0.

Alternatively, you may also provide commands as "given" scenario. In that case, the events generated by those commands will be used to event source the Aggregate when executing the actual command under test. Use the "`givenCommands(...)`" method to provide Command objects.

Execution

The execution phase allows you to provide a command to be executed against the command handling component. That's all. Note that successful execution of this command requires that a command handler that can handle this type of command has been configured with the test fixture.

Inspecting illegal state changes in Aggregates

During the execution of the test, Axon attempts to detect any illegal state changes in the Aggregate under test. It does so by comparing the state of the Aggregate after the command execution to the state of the Aggregate if it sourced from all "given" and stored events. If that state is not identical, this means that a state change has occurred outside of an Aggregate's Event Handler method. Static and transient fields are ignored in the comparison, as they typically contain references to resources.

You can switch detection in the configuration of the fixture with the `setReportIllegalStateChange` method.

Validation

The last phase is the validation phase, and allows you to check on the activities of the command handling component. This is done purely in terms of return values and events (both stored and dispatched).

The test fixture allows you to validate return values of your command handlers. You can explicitly define an expected void return value or any arbitrary value. You may also express the expectancy of an exception.

The other component is validation of stored and dispatched events. In most cases, the stored and dispatched are equal. In some cases however, you may dispatch events (e.g. `ApplicationEvent`) that are not stored in the event store. In the first case, you can use the `expectEvents` method to validate events. In the latter case, you may use the `expectPublishedEvents` and `expectStoredEvents` methods to validate published and stored events, respectively.

There are two ways of matching expected events.

The first is to pass in Event instances that need to be literally compared with the actual events. All properties of the expected Events are compared (using `equals()`) with their counterparts in the actual Events. If one of the properties is not equal, the test fails and an extensive error report is generated.

The other way of expressing expectancies is using Matchers (provided by the Hamcrest library). `Matcher` is an interface prescribing two methods: `matches(Object)` and `describeTo(Description)`. The first returns a boolean to indicate whether the matcher matches or not. The second allows you to express your expectation. For example, a "GreaterThanTwoMatcher" could append "any event with value greater than two" to the description. Descriptions allow expressive error messages to be created about why a test case fails.

Creating matchers for a list of events can be tedious and error-prone work. To simplify things, Axon provides a set of matchers that allow you to provide a set of event specific matchers and tell Axon how they should match against the list.

Below is an overview of the available Event List matchers and their purpose:

- **List with all of**: `Matchers.listWithAllOf(event matchers...)`

This matcher will succeed if all of the provided Event Matchers match against at least one event in the list of actual events. It does not matter whether multiple matchers match against the same event, nor if an event in the list does not match against any of the matchers.

- **List with any of**: `Matchers.listWithAnyOf(event matchers...)`

  This matcher will succeed if one of more of the provided Event Matchers matches against one or more of the events in the actual list of events. Some matchers may not even match at all, while another matches against multiple others.

- **Sequence of Events**: `Matchers.sequenceOf(event matchers...)`

  Use this matcher to verify that the actual Events are match in the same order as the provided Event Matchers. It will succeed if each Matcher matches against an Event that comes after the Event that the previous matcher matched against. This means that "gaps" with unmatched events may appear.

  If, after evaluating the events, more matchers are available, they are all matched against "`null`". It is up to the Event Matchers to decide whether they accept that or not.

- **Exact sequence of Events**: `Matchers.exactSequenceOf(event matchers...)`

  Variation of the "Sequence of Events" matcher where gaps of unmatched events are not allowed. This means each matcher must match against the Event directly following the Event the previous matcher matched against.

For convenience, a few commonly required Event Matchers are provided. They match against a single Event instance:

- **Equal Event**: `Matchers.equalTo(instance...)`

  Verifies that the given object is semantically equal to the given event. This matcher will compare all values in the fields of both actual and expected objects using a null-safe equals method. This means that events can be compared, even if they don't implement the equals method. The objects stored in fields of the given parameter are compared using equals, requiring them to implement one correctly.

- **No More Events**: `Matchers.andNoMore()` or `Matchers.nothing()`

  Only matches against a `null` value. This matcher can be added as last matcher to the Exact Sequence of Events matchers to ensure that no unmatched events remain.

Since the matchers are passed a list of Event Messages, you sometimes only want to verify the payload of the message. There are matchers to help you out:

- **Payload Matching**: `Matchers.messageWithPayload(payload matcher)`

  Verifies that the payload of a Message matches the given payload matcher.

- **Payloads Matching**: `Matchers.payloadsMatching(list matcher)`

  Verifies that the payloads of the Messages matches the given matcher. The given matcher must match against a list containing each of the Messages payload. The Payloads Matching matcher is typically used as the outer matcher to prevent repetition of payload matchers.

Below is a small code sample displaying the usage of these matchers. In this example, we expect two events to be stored and published. The first event must be a "ThirdEvent", and the second "aFourthEventWithSomeSpecialThings". There may be no third event, as that will fail against the "andNoMore" matcher.

```
fixture.given(new FirstEvent(), new SecondEvent())
       .when(new DoSomethingCommand("aggregateId"))
       .expectEventsMatching(exactSequenceOf(
           // we can match against the payload only:
           messageWithPayload(equalTo(new ThirdEvent())),
           // this will match against a Message
           aFourthEventWithSomeSpecialThings(),
           // this will ensure that there is no more events
           andNoMore()
       ));
```

```
// or if we prefer to match on payloads only:
       .expecteEventsMatching(payloadsMatching(
               exactSequenceOf(
                   // we only have payloads, so we can equalTo directly
                   equalTo(new ThirdEvent()),
                   // now, this matcher matches against the payload too
                   aFourthEventWithSomeSpecialThings(),
                   // this still requires that there is no more events
                   andNoMore()
               )
       ));
```

8.2. Testing Annotated Sagas

Similar to Command Handling components, Sagas have a clearly defined interface: they only respond to Events. On the other hand, Saga's have a notion of time and may interact with other components as part of their event handling process. Axon Framework's test support module contains fixtures that help you writing tests for sagas.

Each test fixture contains three phases, similar to those of the Command Handling component fixture described in the previous section.

- given certain events (from certain aggregates),
- when an event arrives or time elapses,
- expect certain behavior or state.

Both the "given" and the "when" phases accept events as part of their interaction. During the "given" phase, all side effects, such as generated commands are ignored, when possible. During the "when" phase, on the other hand, events and commands generated from the Saga are recorded and can be verified.

The following code sample shows an example of how the fixtures can be used to test a saga that sends a notification if an invoice isn't paid within 30 days:

```
AnnotatedSagaTestFixture fixture = new AnnotatedSagaTestFixture(InvoicingSaga.class); ❶
fixture.givenAggregate(invoiceId).published(new InvoiceCreatedEvent()) ❷
       .whenTimeElapses(Duration.standardDays(31)) ❸
       .expectDispatchedCommandsMatching(Matchers.listWithAllOf(aMarkAsOverdueCommand())); ❹

// or, to match against the payload of a Command Message only
       .expectDispatchedCommandsMatching(Matchers.payloadsMatching(
               Matchers.listWithAllOf(aMarkAsOverdueCommand())));
```

❶ Creates a fixture to test the InvoiceSaga class
❷ Notifies the saga that a specific aggregate (with id "invoiceId") has generated an event
❸ Tells the saga that time elapses, triggering events scheduled in that time frame
❹ Verifies that the saga has sent a command matching the return value of `aMarkAsOverdueCommand()` (a Hamcrest matcher)

Defining behavior of Command Callbacks

Sagas can dispatch commands using a callback to be notified of Command processing results. Since there is no actual Command Handling done in tests, the behavior is defined using a `CallbackBehavior` object. This object is registered using `setCallbackBehavior()` on the fixture and defines if and how the callback must be invoked when a command is dispatched.

Instead of using a `CommandBus` directly, you can also use Command Gateways. See below on how to specify their behavior.

Injecting Resources

Often, Sagas will interact with external resources. These resources aren't part of the Saga's state, but are injected after a Saga is loaded or created. The test fixtures allows you to register resources that need to be injected in the Saga. To register a resource, simply invoke the `fixture.registerResource(Object)` method with the resource as parameter. The fixture will detect appropriate setter methods on the Saga and invoke it with an available resource.

Injecting mock objects as resources
It can be very useful to inject mock objects (e.g. Mockito or Easymock) into your Saga. It allows you to verify that the saga interacts correctly with your external resources.

Using Command Gateways

Command Gateways provide Saga's with an easier way to dispatch Commands. Using a custom command gateway also makes it easier to create a mock or stub to define its behavior in tests. When providing a mock or stub, however, the actual command might not be dispatched, making it impossible to verify the sent commands in the test fixture.

Therefore, the fixture provides two methods that allow you to register Command Gateways and optionally a mock defining its behavior: `registerCommandGateway(Class)` and `registerCommandGateway(Class, Object)`. Both methods return an instance of the given class that represents the gateway to use. This instance is also registered as a resource, to make it eligible for resource injection.

When the `registerCommandGateway(Class)` is used to register a gateway, it dispatches Commands to the CommandBus managed by the fixture. The behavior of the gateway is mostly defined by the `CallbackBehavior` defined on the fixture. If no explicit `CallbackBehavior` is provided, callbacks are not invoked, making it impossible to provide any return value for the gateway.

When the `registerCommandGateway(Class, Object)` is used to register a gateway, the second parameter is used to define the behavior of the gateway.

Time as a parameter in your tests

The test fixture tries to eliminate elapsing system time where possible. This means that it will appear that no time elapses while the test executes, unless you explicitly state so using `whenTimeElapses()`. All events will have the timestamp of the moment the test fixture was created.

Having the time stopped during the test makes it easier to predict at what time events are scheduled for publication. If your test case verifies that an event is scheduled for publication in 30 seconds, it will remain 30 seconds, regardless of the time taken between actual scheduling and test execution.

Note
Time is stopped using Joda Time's `JodaTimeUtils` class. This means that the concept of stopped time is only visible when using Joda time's classes. The `System.currentTimeMillis()` will keep returning the actual date and time. Axon only uses Joda Time classes for Date and Time operations.

You can also use the `StubEventScheduler` independently of the test fixtures if you need to test scheduling of events. This `EventScheduler` implementation allows you to verify which events are scheduled for whch time and gives you options to manipulate the progress of time. You can either advance time with a specific `Duration`, move the clock to a specific `DateTime` or advance time to the next scheduled event. All these opertaions will return the events scheduled within the progressed interval.

9. Using Spring

The AxonFramework has many integration points with the Spring Framework. All major building blocks in Axon are Spring configurable. Furthermore, there are some Bean Post Processors that scan the application context for building blocks and automatically wires them.

In addition, the Axon Framework makes use of Spring's Extensible Schema-based configuration feature to make Axon application configuration even easier. Axon Framework has a Spring context configuration namespace of its own that allows you to create common configurations using Spring's XML configuration syntax, but in a more functionally expressive way than by wiring together explicit bean declarations.

9.1. Adding support for the Java Platform Common Annotations

Axon uses JSR 250 annotations (`@PostConstruct` and `@PreDestroy`) to annotate lifecycle methods of some of the building blocks. Spring doesn't always automatically evaluate these annotations. To force Spring to do so, add the `<context:annotation-config/>` tag to your application context, as shown in the example below:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context">

    <context:annotation-config/>

</beans>
```

9.2. Using the Axon namespace shortcut

As mentioned earlier, the Axon Framework provides a separate namespace full of elements that allow you to configure your Axon applications quickly when using Spring. In order to use this namespace you must first add the declaration for this namespace to your Spring XML configuration files.

Assume you already have an XML configuration file like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

        ...

</beans>
```

To modify this configuration file to use elements from the Axon namespace, just add the following declarations:

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:axon="http://www.axonframework.org/schema/core"             ❶
        xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.axonframework.org/schema/core http://www.axonframework.org/schema/axon-core.xsd">
```

❶ The declaration of the `axon` namespace reference that you will use through the configuration file.
❷ Maps the Axon namespace to the XSD where the namespace is defined.

9.3. Wiring event and command handlers

9.3.1. Event handlers

Using the annotated event listeners is very easy when you use Spring. All you need to do is configure the `AnnotationEventListenerBeanPostProcessor` in your application context. This post processor will discover beans with `@EventHandler` annotated methods and automatically connect them to the event bus.

```
<beans xmlns="http://www.springframework.org/schema/beans">

    <bean class="org...AnnotationEventListenerBeanPostProcessor"> ❶
        <property name="eventBus" ref="eventBus"/> ❷
    </bean>

    <bean class="org.axonframework.sample.app.query.AddressTableUpdater"/> ❸

</beans>
```

❶ This bean post processor will scan the application context for beans with an `@EventHandler` annotated method.
❷ The reference to the event bus is optional, if only a single `EventBus` implementation is configured in the application context. The bean postprocessor will automatically find and wire it. If there is more than one `EventBus` in the context, you must specify the one to use in the postprocessor.
❸ This event listener will be automatically recognized and subscribed to the event bus.

You can also wire event listeners "manually", by explicitly defining them within a `AnnotationEventListenerAdapter` bean, as shown in the code sample below.

```
<beans xmlns="http://www.springframework.org/schema/beans">

    <bean class="org.axonframework...annotation.AnnotationEventListenerAdapter"> ❶
        <constructor-arg>
            <bean class="org.axonframework.sample.app.query.AddressTableUpdater"/>
        </constructor-arg>
        <property name="eventBus" ref="eventBus"/> ❷
    </bean>

</beans>
```

❶ The adapter turns any bean with `@EventHandler` methods into an `EventListener`

❷ You need to explicitly reference the event bus to which you like to register the event listener

Warning

Be careful when wiring event listeners "manually" while there is also an `AnnotationEventListenerBeanPostProcessor` in the application context. This will cause the event listener to be wired twice.

9.3.2. Command handlers

Wiring command handlers is very much like wiring event handlers: there is an `AnnotationCommandHandlerBeanPostProcessor` which will automatically register classes containing command handler methods (i.e. methods annotated with the `@CommandHandler` annotation) with a command bus.

```
<beans xmlns="http://www.springframework.org/schema/beans">

    <bean class="org...AnnotationCommandHandlerBeanPostProcessor"> ❶
        <property name="commandBus" ref="commandBus"/> ❷
    </bean>

    <bean class="org.axonframework.sample.app.command.ContactCommandHandler"/> ❸

</beans>
```

❶ This bean post processor will scan the application context for beans with a `@CommandHandler` annotated method.

❷ The reference to the command bus is optional, if only a single `CommandBus` implementation is configured in the application context. The bean postprocessor will automatically find and wire it. If there is more than one `CommandBus` in the context, you must specify the one to use in the postprocessor.

❸ This command handler will be automatically recognized and subscribed to the command bus.

As with event listeners, you can also wire command handlers "manually" by explicitly defining them within a `AnnotationCommandHandlerAdapter` bean, as shown in the code sample below.

```
<beans xmlns="http://www.springframework.org/schema/beans">

    <bean class="org.axonframework...annotation.AnnotationCommandHandlerAdapter"> ❶
        <constructor-arg>
            <bean class="org.axonframework.sample.app.command.ContactCommandHandler"/>
        </constructor-arg>
        <property name="commandBus" ref="commandBus"/> ❷
    </bean>

</beans>
```

❶ The adapter turns any bean with `@EventHandler` methods into an `EventListener`

❷ You need to explicitly reference the event bus to which you like to register the event listener

Warning

Be careful when wiring command handlers "manually" while there is also an `AnnotationCommandHandlerBeanPostProcessor` in the application context. This will cause the command handler to be wired twice.

Wiring AggregateCommandHandlers

When the `@CommandHandler` annotations are placed on the Aggregate, Spring will not be able to automatically configure them. You will need to specify a bean for each of the annotated Aggregate Roots as follows:

```
<bean class="org.axonframework.commandhandling.annotation.AggregateAnnotationCommandHandler"
      init-method="subscribe">
    <constructor-arg value="fully.qualified.AggregateClass"/>
    <constructor-arg ref="ref-to-repo"/>
    <constructor-arg ref="ref-to-command-bus"/>
</bean>

<!-- or, when using Namespace support -->

<axon:aggregate-command-handler aggregate-type="fully.qualified.AggregateClass"
                                repository="ref-to-repo"
                                command-bus="ref-to-command-bus"/>
```

9.3.3. Annotation support using the axon namespace

The previous two sections explained how you wire bean post processors to activate annotation support for your command handlers and event listeners. Using support from the Axon namespace you can accomplish the same in one go, using the annotation-config element:

```
<axon:annotation-config />
```

The annotation-config element has the following attributes that allow you to configure annotation support further:

Table 9.1. Attributes for annotation-config

Attribute name	Usage	Expected value type	Description
commandBus	Conditional	Reference to a CommandBus Bean	Needed only if the application context contains more than one command bus.
eventBus	Conditional	Reference to an EventBus Bean	Needed only if the application context contains more than one event bus.
executor	Optional	Reference to a java.util.concurrent.Executor instance bean	An executor to be used with asynchronous event listeners

9.4. Wiring the event bus

In a typical Axon application there is only one event bus. Wiring it is just a matter of creating a bean of a subtype of `EventBus`. The `SimpleEventBus` is the provided implementation.

```
<beans xmlns="http://www.springframework.org/schema/beans">

    <bean id="eventBus" class="org.axonframework.eventhandling.SimpleEventBus"/>

</beans>

<!-- or using the namespace:

<axon:event-bus id="eventBus"/>
```

9.4.1. Configuration of Clusters

Using a Clustering Event Bus in Spring is very simple. In the Spring context, you just need to define the clusters you wish to use and tell them which Event Listeners you would like to be part of that Cluster. Axon will create the necessary infrastructure to assign listeners to the clusters.

```
<axon:cluster id="myFirstCluster">
    <axon:selectors>
        <axon:package prefix="com.mycompany.mypackage"/>
    </axon:selectors>
</axon:cluster>

<axon:cluster id="defaultCluster" default="true"/>
```

The example above will create two clusters. Event Listeners in the package `com.mycompany.mypackage` will be assigned to `myFirstCluster`, while all others are assigned to `defaultCluster`. Note that the latter does not have any selectors. Selectors are optional if the cluster is a default.

If there are conflicting selectors, and you would like to influence the order in which they are evaluated, you can add the `order` attribute to a cluster. Clusters with a lower value are evaluated before those with a higher value. Only if there are no matching selectors at all, Axon will assign a Listener to the Cluster with `default="true"`. If no suitable cluster for any listener is found, Axon throws an exception.

Organizing Cluster definitions in context files
When you have an application that consists of a number of modules (represented in separate config files), it is possible to define the `<axon:cluster>` in the Context where the listeners are also defined. This makes the application more modular and less dependent on a centralized configuration.

9.4.2. Replayable Clusters

To make a cluster replayable, simply add the `<axon:replay-config>` element to a `<axon:cluster>`, as in the example below:

```
<axon:cluster id="replayingCluster">
    <axon:replay-config event-store="eventStore" transaction-manager="mockTransactionManager"/>
    <axon:selectors>
        <axon:package prefix="com.mycompany.mypackage"/>
    </axon:selectors>
</axon:cluster>
```

The `<axon:replay-config>` element provides the necessary configuration to execute replays on a Cluster. The resulting Cluster bean will be of type `ReplayingCluster`.

9.4.3. Custom Cluster implementation

It is also possible to use the <cluster> element while using a custom Cluster implementation:

```
<axon:cluster id="custoCluster">
    <bean class="com.mycompany.MyPersonalClusterImplementation"/>
    <axon:selectors>
        <axon:package prefix="com.mycompany.mypackage"/>
    </axon:selectors>
</axon:cluster>
```

9.5. Wiring the command bus

The basics

The command bus doesn't take any configuration to use. However, it allows you to configure a number of interceptors that should take action based on each incoming command.

```
<beans xmlns="http://www.springframework.org/schema/beans">

    <bean id="commandBus" class="org.axonframework.commandhandling.SimpleCommandBus">
        <property name="handlerInterceptors">
            <list>
                <bean class="my-interceptors"/>
            </list>
        </property>
    </bean>

</beans>
```

Using the Axon namespace

Setting up a basic command bus using the Axon namspace is a piece of cake: you can use the `commandBus` element:

```
<axon:command-bus id="commandBus"/>
```

Configuring command interceptors for your command bus is also possible using the `<axon:command-bus>` element, like so:

```
<axon:command-bus id="commandBus">
    <axon:dispatchInterceptors>
        <bean class="..."/>
    </axon:dispatchInterceptors>
    <axon:handlerInterceptors>
        <bean class="..."/>
        <bean class="..."/>
    </axon:handlerInterceptors>
</axon:command-bus>
```

Of course you are not limited to bean references; you can also include local bean definitions if you want.

9.6. Wiring the Repository

Wiring a repository is very similar to any other bean you would use in a Spring application. Axon only provides abstract implementations for repositories, which means you need to extend one of them. See Chapter 5, *Repositories and Event Stores* for the available implementations.

Repository implementations that do support event sourcing just need the event bus to be configured, as well as any dependencies that your own implementation has.

```
<bean id="simpleRepository" class="my.package.SimpleRepository">
    <property name="eventBus" ref="eventBus"/>
</bean>
```

Repositories that support event sourcing will also need an event store, which takes care of the actual storage and retrieval of events. The example below shows a repository configuration of a repository that extends the `EventSourcingRepository`.

```
<bean id="contactRepository" class="org.axonframework.sample.app.command.ContactRepository">
    <property name="eventBus" ref="eventBus"/>
    <property name="eventStore" ref="eventStore"/>
</bean>
```

In many cases, you can use the `EventSourcingRepository`. Below is an example of XML application context configuration to wire such a repository.

```
<bean id="myRepository" class="org.axonframework.eventsourcing.EventSourcingRepository">
    <constructor-arg value="fully.qualified.class.Name"/>
    <property name="eventBus" ref="eventBus"/>
    <property name="eventStore" ref="eventStore"/>
</bean>
```

```
<!-- or, when using the axon namespace -->

<axon:event-sourcing-repository id="myRepository"
                                aggregate-type="fully.qualified.class.Name"
                                event-bus="eventBus" event-store="eventStore"/>
```

The repository will delegate the storage of events to the configured `eventStore`, while these events are dispatched using the provided `eventBus`.

9.7. Wiring the Event Store

All event sourcing repositorties need an Event Store. Wiring the `JpaEventStore` and the `FileSystemEventStore` is very similar, but the `JpaEventStore` needs a way to get a hold of an EntityManager. In general, applications use a Container Managed EntityManager:

```
<bean id="eventStore" class="org.axonframework.eventstore.jpa.JpaEventStore">
    <constructor-arg>
        <bean class="org.axonframework.common.jpa.ContainerManagedEntityManagerProvider"/>
    </constructor-arg>
</bean>

<!-- declare transaction manager, data source, EntityManagerFactoryBean, etc -->
```

Using the Axon namespace support, you can quickly configure event stores backed either by the file system or a JPA layer using the one of the following elements:

```
<axon:jpa-event-store id="jpaEventStore"/>

<axon:filesystem-event-store id="fileSystemEventStore" base-dir="/data"/>
```

The annotation support will automatically configure a Container Managed EntityManager on the Jpa Event Store, but you may also configure a custom implementation using the `entity-manager-provider` attribute. This is useful when an application uses more than one EntityManagerFactory.

9.8. Configuring Snapshotting

Configuring snapshotting using Spring is not complex, but does require a number of beans to be configured in your application context.

The `EventCountSnapshotterTrigger` needs to be configured as a proxy for your event store. That means all repositories should load and save aggregate from the `EventCountSnapshotterTrigger`, instead of the acutal event store.

```
<bean id="myRepository" class="org.axonframework...GenericEventSourcingRepository">
    <!-- properties omitted for brevity -->
    <property name="snapshotterTrigger">
        <bean class="org.axonframework.eventsourcing.EventCountSnapshotterTrigger">
            <property name="trigger" value="20" />
        </bean>
    </property>
</bean>

<!-- or, when using the namespace -->

<axon:event-sourcing-repository> <!-- attributes omitted for brevity -->
    <axon:snapshotter-trigger event-count-threshold="20" snapshotter-ref="snapshotter"/>
</axon:event-sourcing-repository>
```

The sample above configures an EventCountSnapshotter trigger that will trigger Snapshot creation when 20 or more events are required to reload the aggregate's current state.

The snapshotter is configured as follows:

```
<bean id="snapshotter" class="org.axonframework.eventsourcing.SpringAggregateSnapshotter">
    <property name="eventStore" ref="eventStore"/>
    <property name="executor" ref="taskExecutor"/>
```

```
</bean>

<!-- or, when using the namespace -->

<axon:snapshotter id="snapshotter" event-store="eventStore" executor="taskExecutor"/>

<!-- the task executor attribute is optional. When used you can define (for example) a thread pool to perform the snapshotting -->
<bean id="taskExecutor" class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
    <property name="corePoolSize" value="2"/>
    <property name="maxPoolSize" value="5"/>
    <property name="waitForTasksToCompleteOnShutdown" value="true"/>
</bean>
```

The `SpringAggregateSnapshotter` will automatically detect any `PlatformTransactionManager` in your application context, as well as `AggregateFactory` instances, which all repositories typically are. That means you only need very little configuration to use a `Snapshotter` within Spring. If you have multiple `PlatformTransactionManager` beans in your context, you should explicitly configure the one to use.

9.9. Configuring Sagas

To use Sagas, two infrastructure components are required: the SagaManager and the SagaRepository. Each have their own element in the Spring application context.

The SagaManager is defined as follows:

```
<axon:saga-manager id="sagaManager" saga-repository="sagaRepository"
                   saga-factory="sagaFactory"
                   resource-injector="resourceInjector">
    <axon:async executor="taskExecutor" transaction-manager="transactionManager" />
    <axon:types>
        fully.qualified.ClassName
        another.ClassName
    </axon:types>
</axon:saga-manager>
```

All properties are optional. The `saga-repository` will default to an in-memory repository, meaning that Sagas will be lost when the VM is shut down. The `saga-factory` can be provided if the Saga instances do not have a no-argument accessible constructor, or when special initialization is required. An `async` element with `executor` can be provided if Sagas should not be invoked by the event dispatching thread. When using asynchronous event handling it is required to provide the `transaction-manager` attribute. The default resource injector uses the Spring Context to autowire Saga instances with Spring Beans.

Use the `types` element to provide a comma and/or newline separated list of fully qualified class names of the annotated sagas.

When an in-memory Saga repository does not suffice, you can easily configure one that uses JPA as persistence mechanism as follows:

```
<axon:jpa-saga-repository id="sagaRepository" resource-injector="resourceInjector"
                          use-explicit-flush="true" saga-serializer="sagaSerializer"/>
```

The resource-injector, as with the saga manager, is optional and defaults to Spring-based autowiring. The saga-serializer defines how Saga instances need to be serialized when persisted. This defaults to an XStream based serialization mechanism. You may choose to explicitly flush any changes made in the repository immediately or postpone it until the transaction in which the changes were made are executed by setting the `use-explicit-flush` attribute to `true` or `false`, respectively. This property defaults to `true`.

10. Performance Tuning

This chapter contains a checklist and some guidelines to take into consideration when getting ready for production-level performance. By now, you have probably used the test fixtures to test your command handling logic and sagas. The production environment isn't as forgiving as a test environment, though. Aggregates tend to live longer, be used more frequently and concurrently. For the extra performance and stability, you're better off tweaking the configuration to suit your specific needs.

10.1. Database Indexes and Column Types

10.1.1. SQL Databases

If you have generated the tables automatically using your JPA implementation (e.g. Hibernate), you probably do not have all the right indexes set on your tables. Different usages of the Event Store require different indexes to be set for optimal performance. This list suggests the indexes that should be added for the different types of queries used by the default `EventEntryStore` implementation:

- Normal operational use (storing and loading events):

  Table 'DomainEventEntry', columns `type`, `aggregateIdentifier` and `sequenceNumber` (primary key or unique index)

- Snapshotting:

  Table 'SnapshotEventEntry', columns `type` and `aggregateIdentifier`.

- Replaying the Event Store contents

  Table 'DomainEventEntry', column `timestamp`, `sequenceNumber` and `aggregateIdentifier`

- Sagas

  Table 'AssociationValueEntry', columns `associationKey` and `sagaId`,

  Table 'SagaEntry', column `sagaId` (unique index)

The default column lengths generated by e.g. Hibernate may work, but won't be optimal. A UUID, for example, will always have the same length. Instead of a variable length column of 255 characters, you could use a fixed length column of 36 characters for the aggregate identifier.

The 'timestamp' column in the DomainEventEntry table only stores ISO 8601 timestamps. If all times are stored in the UTZ timezone, they need a column length of 24 characters. If you use another timezone, this may be up to 28. Using variable length columns is generally not necessary, since time stamps always have the same length.

The 'type' column in the DomainEventEntry stores the Type Identifiers of aggregates. Generally, these are the 'simple name' of the aggregate. Event the infamous 'AbstractDependencyInjectionSpringContextTests' in spring only counts 45 characters. Here, again, a shorter (but variable) length field should suffice.

10.1.2. MongoDB

By default, the MongoEventStore will only generate the index it requires for correct operation. That means the required unique index on "Aggregate Identifier", "Aggregate Type" and "Event Sequence Number" is created when the Event Store is created. However, when using the MongoEventStore for certain opertaions, it might be worthwile to add some extra indices.

Note that there is always a balance between query optimization and update speed. Load testing is ultimately the best way to discover which indices provide the best performance.

- Normal operational use

  An index is automatically created on "aggregateIdentifier", "type" and "sequenceNumber" in the domain events (default name: "domainevents") collection

- Snapshotting

  Put a (unique) index on "aggregateIdentifier", "type" and "sequenceNumber" in the snapshot events (default name: "snapshotevents") collection

- Replaying events:

  Put a non-unique index on "timestamp" and "sequenceNumber" in the domain events (default name: "domainevents") collection

- Sagas

Put a (unique) index on the "sagaIdentifier" in the saga (default name: "sagas") collection

Put an index on the "sagaType", "associations.key" and "associations.value" properties in the saga (default name: "sagas") collection

10.2. Caching

A well designed command handling module should pose no problems when implementing caching. Especially when using Event Sourcing, loading an aggregate from an Event Store is an expensive operation. With a properly configured cache in place, loading an aggregate can be converted into a pure in-memory process.

Here are a few guidelines that help you get the most out of your caching solution:

- Make sure the Unit Of Work never needs to perform a rollback for functional reasons.

  A rollback means that an aggregate has reached an invalid state. Axon will automatically invalidate the cache entries involved. The next requrest will force the aggregate to be reconstructed from its Events. If you use exceptions as a potential (functional) return value, you can configure a `RollbackConfiguration` on your Command Bus. By default, the Unit Of Work will be rolled back on runtime exceptions.

- All commands for a single aggregate must arrive on the machine that has the aggregate in its cache.

  This means that commands should be consistently routed to the same machine, for as long as that machine is "healthy". Routing commands consistently prevents the cache from going stale. A hit on a stale cache will cause a command to be executed and fail at the moment events are stored in the event store.

- Configure a sensible time to live / time to idle

  By default, caches have a tendency to have a relatively short time to live, a matter of minutes. For a command handling component with consistent routing, a longer time-to-idle and time-to-live is usually better. This prevents the need to re-initialize an aggregate based on its events, just because its cache entry expired. The time-to-live of your cache should match the expected lifetime of your aggregate.

10.3. Snapshotting

Snapshotting removes the need to reload and replay large numbers of events. A single snapshot represents the entire aggregate state at a certain moment in time. The process of snapshotting itself, however, also takes processing time. Therefor, there should be a balance in the time spent building snapshots and the time it saves by preventing a number of events being read back in.

There is no default behavior for all types of applications. Some will specify a number of events after which a snapshot will be created, while other applications require a time-based snapshotting interval. Whatever way you choose for your application, make sure snapshotting is in place if you have long-living aggregates.

See Section 5.5, “Snapshotting” for more about snapshotting.

10.4. Aggregate performance

The actual structure of your aggregates has a large impact of the performance of command handling. Since Axon manages the concurrency around your aggregate instances, you don't need to use special locks or concurrent collections inside the aggregates.

Override `getChildEntities`

By default, the getChildEntities method in AbstractEventSourcedAggregateRoot and AbstractEventSourcedEntity uses reflection to inspect all the fields of each entity to find related entities. Especially when an aggregate contains large collections, this inspection could take more time than desired.

To gain a performance benefit, you can override the `getChildEntities` method and return the collection of child entities yourself.

10.5. Event Serializer tuning

XStream is very configurable and extensible. If you just use a plain `XStreamSerializer`, there are some quick wins ready to pick up. XStream allows you to configure aliases for package names and event class names. Aliases are typically much shorter (especially if you have long package names), making the serialized form of an event smaller. And since we're talking XML, each character removed from XML is twice the profit (one for the start tag, and one for the end tag).

A more advanced topic in XStream is creating custom converters. The default reflection based converters are simple, but do not generate the most compact XML. Always look carefully at the generated XML and see if all the information there is really needed to reconstruct the original instance.

Avoid the use of upcasters when possible. XStream allows aliases to be used for fields, when they have changed name. Imagine revision 0 of an event, that used a field called "clientId". The business prefers the term "customer", so revision 1 was created with a field called "customerId". This can be configured completely in XStream, using field aliases. You need to configure two aliases, in the following order: alias "customerId" to "clientId" and then alias "customerId" to "customerId". This will tell XStream that if it encounters a field called "customerId", it will call the corresponding XML element "customerId" (the second alias overrides the first). But if XStream encounters an XML element called "clientId", it is a known alias and will be resolved to field name "customerId". Check out the XStream documentation for more information.

For ultimate performance, you're probably better off without reflection based mechanisms alltogether. In that case, it is probably wisest to create a custom serialization mechanism. The `DataInputStream` and `DataOutputStream` allow you to easilly write the contents of the Events to an output stream. The `ByteArrayOutputStream` and `ByteArrayInputStream` allow writing to and reading from byte arrays.

10.5.1. Preventing duplicate serialization

Especially in distributed systems, Event Messages need to be serialized in multiple occasions. In the case of a Command Handling component that uses Event Sourcing, each message is serialized twice: once for the Event Store, and once to publish it on the Event Bus. Axon's components are aware of this and have support for SerializationAware messages. If a SerializationAware message is detected, its methods are used to serialize an object, instead of simply passing the payload to a serializer. This allows for performance optimizations.

By configuring the `SerializationOptimizingInterceptor`, all generated Events are wrapped into `SerializationAware` messages, and thus benefit from this optimization. Note that the optimization only helps if the same serializer is used for different components. If you use the `DisruptorCommandBus`, serialization can be optimized by providing a `Serializer` in the `DisruptorConfiguration`. The `DisruptorCommandBus` will then use an extra thread (or more when configured) to pre-serialize the Event Message using that serializer.

When you serialize messages yourself, and want to benefit from the SerializationAware optimization, use the `MessageSerializer` class to serialize the payload and meta data of messages. All optimization logic is implemented in that class. See the JavaDoc of the MessageSerializer for more details.

10.6. Custom Identifier generation

The Axon Framework uses an `IdentifierFactory` to generate all the identifiers, whether they are for Events or Commands. The default `IdentifierFactory` uses randomly generated `java.util.UUID` based identifiers. Although they are very safe to use, the process to generate them doesn't excell in performance.

IdentifierFactory is an abstract factory that uses Java's ServiceLoader (since Java 6) mechanism to find the implementation to use. This means you can create your own implementation of the factory and put the name of the implementation in a file called "`/META-INF/services/org.axonframework.domain.IdentifierFactory`". Java's ServiceLoader mechanism will detect that file and attempt to create an instance of the class named inside.

There are a few requirements for the `IdentifierFactory`. The implementation must

- have its fully qualified class name as the contents of the `/META-INF/services/org.axonframework.domain.IdentifierFactory` file on the classpath,
- have an accessible zero-argument constructor,
- extend `IdentifierFactory`,
- be accessible by the context classloader of the application or by the classloader that loaded the `IdentifierFactory` class, and must
- be thread-safe.