Reference Guide

Axon Framework 1.2

Allard Buijze

Jettro Coenradie

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 1.4, “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.

Greg Young, initiator of the Command Query Responsiblity Segregation (CQRS) pattern addressed 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 [http://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.

Most of the concerns Axon addresses are located within a single JVM. However, for an application to be scalable, a single JVM is not enough. Therefore, Axon provides the `axon-intregration` module, which allows events to be sent to a Spring Integration channel. From there, you can use Spring Integration to dispatch events to application components on different machines. In the near future, Axon will provide more ways to dispatch commands and events between JVM's and physical machines.

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 [http://www.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 also checkout the sources from the subversion repository: `http://axonframework.googlecode.com/svn/trunk/`.

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>1.2</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. This section discusses these dependencies and describes in what scenarios you need them.

Spring Integration

The Axon Framework provides connectors that allow you to publish events on a Spring Integration channel. These connectors require Spring Integration on the classpath. You need the following maven dependencies to use these connectors. Axon was compiled against Spring Integration 2.

```
<dependency>
    <groupId>org.axonframework</groupId>
    <artifactId>axon-integration</artifactId>
    <version>1.2</version>
</dependency>
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-core</artifactId>
    <version><!-- Add version here --></version>
</dependency>
```

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 `Executors`, 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.

1.3. 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 [http://www.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 [mailto:axonframework@googlegroups.com].

1.4. 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 [http://www.apache.org/licenses/LICENSE-2.0] for the specific language governing permissions and limitations under the License.

2. Architecture 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).

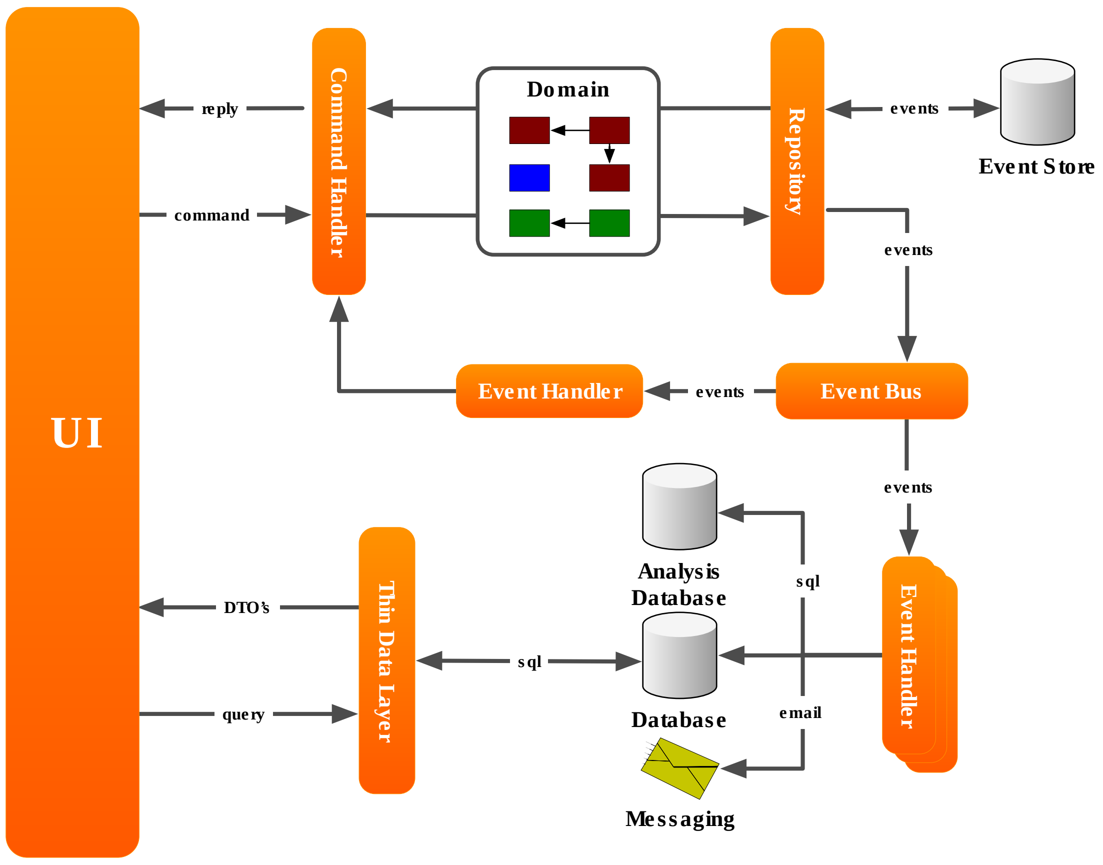

Figure 2.1. Architecture overview of a CQRS application

Command Handling

Commands are represented by simple and straightforward objects that contain all data necessary for a command handler to execute it. Typically, a command expresses some 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 inChapter 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 inChapter 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 inChapter 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 inChapter 6, *Event Processing*.

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.

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.

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

3.1. Creating a Command Handler

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. The Command Handler, however, must implement the `CommandHandler` interface. This interface declares only a single method: `Object handle(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”. It is not recommended to use return values, but they are allowed. Always consider using 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 use `Void.TYPE`, the official representation of the `void` keyword.

Note

Note that Command Handlers need to be explicitly subscribed to the Command Bus for the specific types of Command they can handle. See Section 3.3, “Configuring the Command Bus”.

Annotation support

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 an optional second parameter, which is 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.

You can use the `AnnotationCommandHandlerAdapter` to turn your annotated class into a `CommandHandler`. The adapter also takes a `CommandBus` instance. Use the `subscribe()` method on the adapter to subscribe all the annotated handlers to the command bus using the correct command type.

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, this will automatically subscribe any command handler in your application.

Note that you need to be careful when mixing manual wrapping and the use of annotation-config element. This might result in command handler being subscribed twice.

`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 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 this value is an instance of `AggregateIdentifier`, it will be directly used. If it is a `java.util.UUID`, it will be wrapped in a `UUIDAggregateIdentifier`. Any other object will have its `toString()` value wrapped in a `StringAggregateIdentifier`.

If you prefer not to use annotations, the behavior can be overridden by supplying a custom `CommandTargetResolver`. This class should return the `AggregateIdentifier` 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.

3.2. Dispatching commands

The CommandBus provides two methods to dispatch commands to their respective handler: `dispatch(command, callback)` and `dispatch(command)`. The first parameter is 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 (which is a highly discouraged approach), you can use the `FutureCallback`. It is a combination of a `Future` (as defined in the java.concurrent package) and Axon's `CommandCallback`.

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 received. Your application will just have to guarantee that the command is processed and with "positive outcome", sooner or later...

3.3. Configuring 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.

Axon provides a single implementation of the Command Bus: `SimpleCommandBus`. The `SimpleCommandBus` dispatches and commands and executes the handler in the calling thread. You can subscribe and unsubscribe command handlers using the `subscribe` and `unsubscribe` methods, 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 done 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.

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 send 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 the optional second 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.

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 Section 3.5.1, “Transaction management”.

UnitOfWork and Exceptions

Your command handlers may throw an Exception as a result of command processing. By default, these exceptions will cause the UnifOfWork 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`, which is the default, will cause a rollback on any exception (or error). The other is the `RollbackOnUncheckedExceptionConfiguration`, which will commit the Unit of Work on unchecked exceptions (those not extending `RuntimeException`) while still performing a rollback on Errors and Runtime Exceptions.

3.5. Command Handler 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 Command Handler Interceptors. These 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, the current `UnitOfWork` and an `InterceptorChain`. The `InterceptorChain` is used to continue the dispatching process.

3.5.1. Transaction management

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. If this is important to your application (although it should be avoided as much as possible), consider using a Transaction Interceptor on the command bus that attaches a transaction to the Unit of Work.

Axon provides the `SpringTransactionalInterceptor`, which uses Spring's `PlatformTransactionManager` to manage the actual transactions. A transaction is committed after a successful commit of the Unit of Work, or rolled back as the Unit of Work is rolled back.

3.5.2. 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. Transaction Management is better done after structural validation, as it often requires remote resources.

3.5.3. 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 arbitray 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.

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

The Axon Framework makes a distinction between three types of events, each with a clear use and type of origin. Regardless of their type, all events must implement the `Event` interface or one of the more specific sub-types, Domain Events, Application Events and System Events, each described in the sections below.

All events may carry data and meta-data. Typically, the data is added to each event as fields in the event implementation. Meta-data, on the other hand is stored separately. The Auditing interceptor uses this mechanism to attach meta-data to events for auditing purposes. All Axon's implementations of Events allow the subclasses to attach arbitrary information as meta-data.

Note

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

4.1.1. Domain Events

The most important type of event in any CQRS application is the domain event. It represents an event that occurs inside your domain logic, such as a state change or special notification of a certain state. The latter not being per definition a state change.

In the Axon Framework, all domain events should extend the abstract `DomainEvent` class. This abstract class keeps track of the aggregate they are generated by, and the sequence number of the event inside the aggregate. This information is important for the Event Sourcing mechanism, as well as for event handlers (see Section 6.2, “Event Listeners”) that need to know the origin of an event.

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).

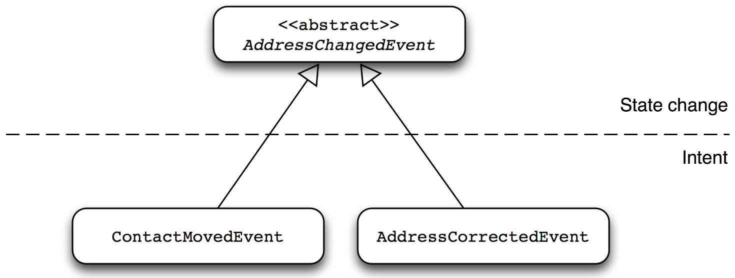

Figure 4.1. Adding intent to events

There is a special type of `Event`, which has a special meaning: the `AggregateDeletedEvent`. This is a marker interface that indicates a migration to a "deleted" state of the aggregate. Repositories must treat aggregates that have applied such an event as deleted. Hence, loading an aggregate that has an `AggregateDeletedEvent` results in an exception.

Snapshot events are instances of `DomainEvent` with a special intent. They are typically not dispatched via the event bus, but are used to summarize an arbitrary number of events from the past into a single entry. This can drastically improve performance when initializing an aggregate's state from a series of events. See Section 5.4, “Snapshotting” for more information about snapshot events and their use.

4.1.2. Application Events

Application events are events that cannot be categorized as domain events, but do have a significant importance for the application. When using application events, check if the event is actually a domain event that you overlooked. Examples of application events are the expiry of a user session, or the notification of an email being successfully sent. The usefulness of these events depend on the type of application you are creating.

In the Axon Framework, you can extend the abstract `ApplicationEvent` class for application events. This class will generate a unique identifier and a time stamp for the current event. Optionally, you can attach an object that acts as the source of the event. This source is loosely referenced, which means that if the garbage collector cleans up the source, or when the event is serialized and de-serialized, the original source class is not available anymore. Instead, you will have access to the type of source and the value of it's `toString()` method.

4.1.3. System Events

The third type of event identified by Axon Framework is the System Event. These events typically provide notifications of the status of the system. These events could, for example, indicate that a subsystem is non-responsive or has raised an exception.

All system events extend the abstract `SystemEvent` class. Upon construction of this event, you may pass an exception, defining the cause of the event, and a source object which is considered the source of the event. As with application events, the source is loosely referenced from the event.

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 will 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 `AggregateIdentifier`. There are two basic implementations of these identifiers: `UUIDAggregateIdentifier`, which used Java's `UUID` to generate random identifiers, and the `StringAggregateIdentifier`, which allows you to choose a `String` which should be used as identifier. You can choose any identifier type you like, and even create your own, as long as they have a valid String representation.

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.

`AbstractJpaAggregateRoot`

The `AbstractJpaAggregateRoot` is a JPA-compatible implementation of the `AggregateRoot` interface. It has the annotation necessary to persist the aggregate's state and reconstruct it from database tables. It uses the `@Version` annotation on one of it's field to perform optimistic locking on the database level. Similar to the `AbstractAggregateRoot`, the `AbstractJpaAggregateRoot` keeps track of uncommitted events, which have been registered using `registerEvent(DomainEvent)`.

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.

`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 someProperty;

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

    public MyAggregateRoot(UUID identifier) {
        super(identifier);
    }

    public void handle(DomainEvent event) {
        if (event instanceof MyAggregateCreatedEvent) {
            // do something with someProperty
        }
        // and more if-else-if logic here
    }
}
```

`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 {
    private String someProperty;

    public MyAggregateRoot() {
        apply(new MyAggregateCreatedEvent());
    }
    public MyAggregateRoot(UUID identifier) {
        super(identifier);
    }

    @EventHandler
    private void handleMyAggregateCreatedEvent(MyAggregateCreatedEvent event) {
        // do something with someProperty
    }
}
```

In all circumstances, exactly one event handler method is invoked. The `AbstractAnnotatedAggregateRoot` 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 the parameter is of the event type or a super type, 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 level of the `AbstractAnnotatedAggregateRoot` is reached, and no suitable event handler is found, the event is ignored.

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.

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 extend from `AbstractEventSourcedEntity`. 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 its fields for any implementations of `AbstractEventSourcedEntity` 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 (unlike usual in 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. Axon will ensure that the Aggregate Root is properly registered in time.

Axon will automatically detect most of the child entities in the fields of an Entity (albeit aggregate root or not). The following Entities are found:

- 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 you 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`.

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 (seeSection 5.5, “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 (seeSection 5.1, “Standard repositories”), and those that store the events of an aggregate (seeSection 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 append a special `AggregateDeletedEvent` to the event store.

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.

`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` to use an optimistic locking strategy, 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.

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 5.5, “Advanced conflict detection and resolution” for more information.

`GenericJpaRepository`

This is a repository implementation that can store JPA compatible Aggregates, such as `AbstractJpaAggregateRoot` subclasses. 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` (seeSection 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.

`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 would create 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.

`GenericAggregateFactory`

The `GenericAggregateFactory` is a special `AggregateFactory` implementation that can be used for any type of Event Sourced Aggregate Root. There is however, a convention that these `EventSourcedAggregateRoot` classes must adhere to: the type must declare an accessible constructor accepting an `AggregateIdentifier` as single parameter. This constructor may not perform any initialization on the aggregate, other than setting the identifier.

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.

`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, making data migrations potentially easier. Since the aggregates are event sourced, you keep the ability to use the given-when-then test fixtures (seeChapter 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 two implementations of event stores, both are capable of storing all domain events (those that extend the `DomainEvent` class). 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 `XStreamEventSerializer`.

`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 provided url must end on a slash. This is due to the way Spring's `Resource` implementations work.

`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 `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.

Customizing the Event storage

By default, the JPA Event Store stores entries in `DomainEventEntry` and `SnapshotEventEntry` entities. While still 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.

To work around this issue, make sure to exclusively query the serialized form (the `byte[]`) of events. Do not fetch the entire entity object. 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.

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.

Tip

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

Influencing the serialization process

Event Stores need a way to serialize the Domain Event to prepare it for storage. By default, Axon uses the `XStreamEventSerializer`, which uses XStream (see xstream.codehaus.org [http://xstream.codehaus.org/]) to serialize Domain 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 XStreamEventSerializer 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: xstream.codehaus.org [http://xstream.codehaus.org/].

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

Changing the definition of Events

It is not unlikely that a definition of an Event remains unchanged during the entire lifespan of an application. New insights, changes in requirements and many other factors can lead to modifications in Events. 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.

In Axon, Events have a revision, a numeric value that defaults to 0. If an Event definition changes, you should update the revision number too. The deserialization process can then be adapted when your application needs to cope with old revisions of events. A class file for an Event only needs to support the latest revision. When old revisions are deserialized, they can be "upcasted" to the newer revision. The revision number of an Event is provided to the constructor on the abstract event classes. The revision of an event is passed as a parameter in the constructor of `DomainEvent`.

The `EventUpcaster` is responsible for transforming old events into the last revision. Upcasters typically work on an intermediate representation of the Event. In the case of the `XStreamEventSerializater`, this intermediate representation is dom4j (see http://dom4j.sourceforge.net/ [http://dom4j.sourceforge.net/]), an easy to use java XML library that integrates nicely with XStream. The `EventUpcaster` can modify the XML structure of the event so that it matches the new definition.

5.4. 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.

5.4.1. 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 `XStreamEventSerializer`) is capable of serializing your aggregate. The `XStreamEventSerializer` requires you to use either a Sun 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. All Axon provided repositories implement the `AggregateFactory` interface, and are capable of processing `AggregateSnapshot`s. Configuring multiple aggregate factories allows you to use a single Snapshotter to create snapshots for a variety of aggregate types.

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 repositores, if they are not explicitly configured.

Spring users should use the `SpringAggregateSnapshotter` instead. See Section 9.8, “Configuring Snapshotting” for more details about configuring snapshotting in Spring.

5.4.2. Storing Snapshot Events

Both the `JpaEventStore` and the `FileSystemEventStore` are capable of storing snapshot events. They provide a special method that allows a `DomainEvent` 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 `DomainEvent` 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 `getLastCommittedEventSequenceNumber()` on the `VersionedAggregate` (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. Both 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.

5.4.3. Initializing an Aggregate based on a Snapshot Event

Every snapshot event is a `DomainEvent` instance. 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 repository recognizes this type of event and extract 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.

5.4.4. 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.

5.5. 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. Axon Framework provides an Event Bus and some base classes to help you implement Event Listeners.

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 manages 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` sequentiall. 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 only done locally, 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.

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.

At this moment, there is a single implementation of the `Cluster` interface: `SimpleCluster`. This implementation calls each EventListener sequentially in the calling thread. By adding information in the Meta Data of a cluster, the selector can provide hints to the Terminal about the expected behavior.

EventBusTerminal

The `EventBusTerminal` forms the bridge between the events being dispatched and the Clusters inside the Event Bus. The terminal is aware of any remoting technologies used, such as JMS, AMQP, etc. 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.

6.2. Event Listeners

Event listeners are the component that act on incoming events. These events may be of any type of the events mentioned in Section 4.1, “Events”. In the Axon Framework, all event listeners must implement the `EventListener` interface.

6.2.1. Basic configuration

Event listeners need to be registered with an event bus (see Section 6.1, “Event Bus”) to be notified of events. Axon provides a base implementation that take care of this, and other things, for you.

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 and are resolved according to the same rules that count for annotated aggregate roots (see the section called " AbstractAnnotatedAggregateRoot ").

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.

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 inside an `AnnotationEventListenerAdapter` and register them with an event bus.

6.2.2. 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". With the Axon Framework, you can easily convert any event handler into an asynchronous event handler by wrapping it in an `AsynchronousEventHandlerWrapper` or, when using annotations, adding the type-level `AsynchronousEventListener` annotation.

The `AsynchronousEventHandlerWrapper` needs some extra configuration to make an event handler asynchronous. The first thing that the wrapper needs is an `Executor`, for example a `ThreadPoolExecutor`. The second is the `SequencingPolicy`, a definition of which events may be processed in parallel, and which sequentially. The last one is optional: the `TransactionManager`, which enables you to run event processing within a transaction. The next pragraphs will provide more details about the configuration options.

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 `AsynchonousEventHandlerWrapper` will manage the processing of incoming events in the provided executor. If an instance of a `ScheduledThreadPoolExecutor` is provided, the `AsynchronousEventHandlerWrapper` will automatically leverage its ability to schedule processing in the cases of delayed retries. See Section 6.2.3, "Managing transactions in asynchronous event handling" for more information about transactions.

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 two events have the same sequence identifier, 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. For annotated event handlers, this is the default policy.
- `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 equals 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 `AsynchronousEventHandlerWrapper` to add transactional processing of events. To optimize processing, events can be processed in small batches inside a transaction. The transaction manager has the ability to influence the size of these batches and can decide to either commit, skip or retry event processing based on the result of a batch. See Section 6.2.3, “Managing transactions in asynchronous event handling” for more information.

Annotation support for concurrent processing

If you use the `AnnotationEventListenerAdapter`, or `<axon:annotation-config/>`, an `Executor` must be configured to allow asynchronous processing of events.

You can configure the event sequencing policy on the `@AsynchronousEventListener` annotation. You then set the `sequencePolicyClass` to the type of policy you like to use. Note that you can only choose policy classes that provide a public no-arg constructor.

```java
@AsynchronousEventListener(sequencingPolicyClass = MyCustomPolicy.class)
public class MyEventListener() {

    @EventHandler
    public void onSomeImportantEvent(MyEvent event) {
        // eventProcessing logic
    }
}

public class MyCustomPolicy implements EventSequencingPolicy {
    public Object getSequenceIdentifierFor(Event event) {
        if (event instanceof MyEvent) {
            // let's assume that we do processing based on the someProperty field.
            return ((MyEvent) event).someProperty();
        }
        return null;
    }
}
```

With annotation support, the event handler bean must also act as a transaction manager in order to support transactions. There is annotation support for transaction management, too (see Section 6.2.3, “Managing transactions in asynchronous event handling”).

6.2.3. Managing transactions in asynchronous event handling

In some cases, your event handlers have to store data in systems that use transactions. Starting and committing a transaction for each single event has a big performance impact. In Axon, events are processed in batches. The batch size depends of the number of events that need to be processed and the settings provided by the event handler. By default, the batch size is set to the number of events available in the processing queue at the time a batch starts.

Note

Typically, when using synchronous event handling, the transaction boundary is managed at the Command Bus level. Asynchronous event handlers, on the other hand, run in another thread and are often unable to act within the same transaction. The transaction managers used by event handlers should not be confused with the transaction interceptors, which are used with the Command Bus. See Section 3.5.1, “Transaction management” for more information about transactions in the command bus.

In most cases, event handling is done using a thread pool executor, or scheduler. The scheduler will schedule batches of event processing as soon as event become available. When a batch is completed, the scheduler will reschedule processing of the next batch, as long as more events are available. The smaller a batch, the more "fair" the distribution of event handler processing is, but also the more scheduling overhead you create.

When an event listener is wrapped with the `AsynchronousEventHandlerWrapper`, you can configure a `TransactionManager` to handle transactions for the event listener. The transaction manager can, based on the information in the `TransactionStatus` object, decide to start, commit or rollback a transaction to an external system.

The `beforeTransaction(TransactionStatus)` method is invoked just before Axon will start handling an event batch. You can use the TransactionStatus object to configure the batch before it is started. For example, you can change the maximum number of events that may run in the batch.

The `afterTransaction(TransactionStatus)` method is invoked after the batch has been processed, but before the scheduler has scheduled the next batch. Based on the value of `isSuccessful()`, you can decide to commit or rollback the underlying transaction.

Configuring transactional batches

There are a number of settings you can use on the `TransactionStatus` object.

You can configure a yielding policy, which gives the scheduler an indication of that to do when a batch has finished, but more events are available for processing. Use `DO_NOT_YIELD` if you want the scheduler to continue processing immediately as long as new events are available for processing. The `YIELD_AFTER_TRANSACTION` policy will tell the scheduler to reschedule the next batch for processing when a thread is available. The first will make sure events are processed earlier, while the latter provides a fairer execution of events, as yielding provides waiting thread a chance to start processing. The choice of yielding policy should be driven by the SLA of the event listener.

You can set the maximum number of events to handle within a transaction using `setMaxTransactionSize(int)`. The default of this value is the number of events ready for processing at the moment the transaction started.

Error handling

When an event handler throws an exception, for example because a data source is not available, the transaction is marked as failed. In that case, `isSuccessful()` on the `TransactionStatus` object will return `false` and `getException()` will return the exception that the scheduler caught. It is the responsibility of the event listener to rollback or commit any active underlying transactions, based on the information provided by these methods.

The event handler can provide a policy `setRetryPolicy(RetryPolicy)` to tell the scheduler what to do in such case. There are three policies, each for a specific scenario:

- `RETRY_TRANSACTION` tells the event handler scheduler that the entire transaction should be retried. It will reschedule all the events in the current transaction for processing. This policy is suitable when the event listener processes events to a transactional data source that rolls back an entire transaction.
- `RETRY_LAST_EVENT` is the policy that tells the scheduler to only retry the last event in the transaction. This is suitable if the underlying data source does not support transactions or if the transaction was committed without the last event.
- `SKIP_FAILED_EVENT` will tell the scheduler to ignore the exception and continue processing with the next event. The event listener can still try to commit the underlying transaction to persist any changed made while processing other events in this transaction. This is the default policy.

Note that the `SKIP_FAILED_EVENT` is the default policy. For event handlers that use an underlying mechanism to perform actions, this might not be a suitable policy. Exceptions resulting from errors in these underlying systems (such as databases or email clients) would cause events to be ignored when the underlying system is unavailable. In error situations, the event listener should inspect the exception (using the `getException()` method) and decide whether it makes sense to retry processing of this event. If that is the case, it should set the `RETRY_LAST_EVENT` or `RETRY_TRANSACTION` policy, depending on the transactional behavior of the underlying system.

When the chosen policy forces a retry of event processing, the processing is delayed by the number of milliseconds defined in the `retryInterval` property. The default interval is 5 seconds.

Manipulating transactions during event processing

You can change transaction semantics event during event processing. This can be done in one of two ways, depending on the type of event handler you use.

If you use the `@EventHandler` annotation to mark event handler methods, you may use a second parameter of type `TransactionStatus`. If such parameter is available on the annotated method, the current `TransactionStatus` object is passed as a parameter.

Alternatively, you can use the static `TransactionStatus.current()` accessor to gain access to the status of the current transaction. Note that this method returns `null` if there is no active transaction.

With the current transaction status, you can use the `requestImmediateYield()` and `requestImmediateCommit()` methods to end the transaction after processing of the event. The former will also tell the scheduler to reschedule the remainder of the events for another batch. The latter will use the yield policy to see what needs to be done. Since the default yielding policy is `YIELD_AFTER_TRANSACTION`, the behavior of both methods is identical when using these defaults.

Annotation support

As with many of the other supported features in Axon, there is also annotation support for transaction management. You have several options to configure transactions.

The first is to annotate methods on your EventListener with `@BeforeTransaction` and `@AfterTransaction`. These methods will be called before and after the execution of a transactional batch, respectively. The annotated methods may accept a single parameter of type `TransactionStatus`, which provides access to transaction details, such as current status and configuration.

Alternatively, you can use an external Transaction Manager, which you assign to a field. If you annotate that field with `@TransactionManager`, Axon will autodetect it and use it as transaction manager for that listener. The transaction manager may be either one that implements the TransactionManager interface, or any other type that uses annotations.

Provided TransactionManager implementations

Currently, Axon Framework provides one TransactionManager implementation, the `SpringTransactionManager`. This implemenation uses Spring's `PlatformTransactionManager` as underlying transaction mechanism. That means the `SpringTransactionManager` can manage any transactions in resources that Spring supports.

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 with `@SagaEventHandler`: 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, “SagaManager” 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. 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.

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, Object value)` method. Use the `removeAssociationWith(String key, Object 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 CommandBus commandBus;

    @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
        commandBus.dispatch(new PrepareShippingCommand(...));
        commandBus.dispatch(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.

AssociationValue considerations

As said before, an AssociationValue consists of a key, which is a String, and a value, which may be of any type. However, a few simple rules should be conformed to. The value of an AssociationValue should always be a Value Object (as defined in Domain Driven Design). In terms of implementation, that means they should be immutable and have a proper `equals` and `hashCode` method. Preferably, they should implement the Comparable interface, too. If Sagas need to be persisted, `AssociationValue`s need to be serializable. In that case, only use Serializable values. Generally, it shouldn't be a problem to make Value Objects Serializable.

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.

Scheduled Events must extend `ApplicationEvent` or `ScheduledEvent`. The latter is aware of its own publication time. Axon provides two EventScheduler implementations: a pure Java one and one using Quartz 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 threads, you can configure a `EventTriggerCallback` to listen for publication of scheduled Events and manage transactions around them.

> **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 use by the `SagaRepository` to inject resources into a Saga. Axon provides a `SpringResourceInjector`, which injects annotated fields and methods with Resources from the Application Context.

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.

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. `SagaManager`

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 Saga 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. The Transaction Manager has the opportunity to configure the batch size each time a transaction starts. The batch size describes the number of Events that a Saga may process before the transaction is committed. This mechanism can be compared to the mechanism for Event Listeners, desribed in Section 6.2.3, “Managing transactions in asynchronous event handling”.

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.

```
<axon:saga-manager id="sagaManager" saga-factory="optionalSagaFactory"
                   saga-repository="sagaRepository" event-bus="eventBus"
                   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`).

7.2.2. SagaRepository

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 two `SagaRepository` implementations: the `InMemorySagaRepository` and the `JpaSagaRepository`.

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 `SagaSerializer` (comparable to Event serialization, you can use either a `JavaSagaSerializer` or an `XStreamSagaSerializer`). Although Sagas are persisted and may be garbage collected when not used, the Association Values are kept in memory. The memory footprint of these values is generally quite small and should no be a problem. These values are reconstructed after a JVM restart, without loss of data.

In order to ensure that only a single instance exists for each conceptual Saga, the JpaSagaRepository uses a specialized cache. Unlike many other caches, the primary goal of this cache is to prevent multiple instances of a single Saga. These Sagas are Weakly Referenced. That means that once a Saga is no longer referenced, the Garbage Collector may clean them up. When the Saga is needed, a new instance is automatically created.

The JpaSagaRepository is configured with a JPA `EntityManager`, a `ResourceInjector` and a `SagaSerializer`. 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.

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 configuration a maven dependency (use `<artifactId>axon-text</artifactId>`) or from the full package download.

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(); ❶
        MyCommandHandler myCommandHandler = new MyCommandHandler(
                                  fixture.createGenericRepository(MyAggregate.class)); 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 `createGenericRepository` method creates, as expected, a `GenericEventSourcingRepository` instance capable of storing `MyAggregate` instances. This requires some conventions on the MyAggregate class, as described in Section 5.2, “Event Sourcing repositories”.

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 event 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, 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. These events need to be subclasses of `DomainEvent`, as they represent events coming from the event store. You do not need to set aggregate identifiers of sequence numbers. The fixture will inject those for you (using the aggregate identifier exposed by `getAggregateIdentifier` and a sequence number starting with 0.

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.

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(event instance...)`

  Verifies that the given event is semantically equal to the actual event. This matcher will compare all values in the fields of both actual and expected Events using a null-safe equals method. The aggregate identifier and sequence number are ignored, as they are often not set on the "expected" Event.

- **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.

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 "aThirdEvent", 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(fixture.getAggregateIdentifier()))
        .expectEvents(exactSequenceOf(
            aThirdEvent(),
            aFourthEventWithSomeSpecialThings(),
            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
```

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)

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.

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.

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>
```

Setting up an event bus can also be accomplished using support from the axon namespace:

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

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="eventBus" class="org.axonframework.commandhandling.CommandBus">
        <property name="interceptors">
            <list>
                <bean class="org.axonframework...SpringTransactionalInterceptor">
                    <property name="transactionManager" ref="transactionManager"/>
                </bean>
                <bean class="other-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:interceptors>
  <ref>interceptor-zero</ref>
  <ref>interceptor-one</ref>
  <ref>interceptor-two</ref>
 </axon:interceptors>
</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 `GenericEventSourcingRepository`. Below is an example of XML application context configuration to wire such a repository.

```
<bean id="myRepository" class="org.axonframework.eventsourcing.GenericEventSourcingRepository">
    <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 to run in a Spring managed transaction. Unless you use the `SpringTransactionalInterceptor` on your command bus, you need to declare the annotation-driven transaction-manager as shown in the sample below.

```
<bean id="eventStore" class="org.axonframework.eventstore.jpa.JpaEventStore"/>
```

```
<!-- enable the configuration of transactional behavior based on annotations -->
<tx:annotation-driven transaction-manager="txManager"/>

<!-- 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" baseDir="/data"/>
```

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 id="snapshotterTrigger" 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" executor="taskExecutor"
                   transaction-manager="transactionManager"
                   resource-injector="resourceInjector">
    <axon:types>
        fully.qualified.ClassName,
        another.ClassName
    </axon:types>
</axon:saga-manager>
```

All properties, except for the `id` 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 `executor` can be provided if Sagas should not be invoked by the event dispatching thread. When using asynchronous event handling, or event scheduling within the Saga's, 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-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

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 explains which fields are used for the different types of queries by the default `EventEntryStore` implementation:

- Normal operational use (storing and loading events):

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

- Snapshotting:

  Table 'SnapshotEventEntry', columns `type`, `aggregateIdentifier` and `sequenceNumber` (optionally a unique index)

- Replaying the Event Store contents

  Table 'DomainEventEntry', column `timestamp` (optionally also `sequenceNumber`)

- Sagas

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

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

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, and will invalidate the cache. 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 every exception.

- 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, an eternal time-to-idle and time-to-live is the better default. 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 5.4, “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. If an entity is a leaf node (i.e. has no child entities), you may either return an empty collection or `null`.

10.5. Event Serializer tuning

XStream is very configurable and extensible. If you just use a plain `XStreamEventSerializer`, 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. An since we're talking XML, 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 customer 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. The `DomainEvent` class provides a constructor that you can use to do a full reconstruction based on existing data: `DomainEvent(String eventIdentifier, DateTime creationTimeStamp, long eventRevision, long sequenceNumber, AggregateIdentifier aggregateIdentifier)`.

10.6. Custom Identifier generation

The Axon Framework uses an `IdentifierFactory` to generate all the identifiers, whether they are for Events or Aggregates. 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.