



Reference Guide

Axon Framework 0.5

Allard Buijze

Jetro Coenradie

Table of Contents

1. Introduction	1
1.1. Axon Framework Background	1
1.1.1. A brief history	1
1.1.2. What is Axon?	2
1.1.3. When to use Axon?	2
1.2. Getting started	3
1.2.1. Download Axon	3
1.2.2. Configure Maven	4
1.3. Contributing to Axon Framework	5
1.4. License information	5
2. Architecture Overview	6
3. Command Handling	9
3.1. Creating a Command Handler	9
3.2. Configuring the Command Bus	10
3.3. Command Handler Interceptors	10
3.3.1. Managing transactions	11
4. Domain Modeling	12
4.1. Events	12
4.1.1. Domain Events	12
4.1.2. Application Events	13
4.1.3. System Events	13
4.2. Aggregate	14
4.2.1. Basic aggregate implementations	14
4.2.2. Event sourcing aggregates	15
5. Repositories and Event Stores	18
5.1. Standard repositories	18
5.2. Event Sourcing repositories	19
5.3. Event store implementations	19
5.4. Using Snapshot Events	21
6. Event Processing	23
6.1. Event Bus	23
6.2. Event Listeners	23
6.2.1. Basic configuration	23
6.2.2. Asynchronous event processing	24
6.2.3. Managing transactions in asynchronous event handling	26
7. Using Spring	29
7.1. Wiring event handlers	29
7.2. Wiring the event bus	30
7.3. Wiring the command bus	30
7.4. Wiring the Repository	31

7.5. Wiring the event store	31
-----------------------------------	----

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. Next to this reference guide, the sources and javadoc, we also have a sample application that shows how you can leverage the provided features to quickly build a dynamic and extensible application.

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”. All help is welcome. If you're interested in helping out building the Axon Framework, Section 1.3, “Contributing to Axon Framework” will contain the information you require. 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 a bare necessity.

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 application 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 Responsibility 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 completely different components than those that query for the application's state. The most important reason for this separation is the fact that there are different technical and non-technical requirements for each of them. When commands are executed, the query components are (a)synchronously updated using events. This mechanism of updates through events, is what makes this architecture is extensible, scalable and ultimately more maintainable.



Note

A full explanation of CQRS is not within the scope of this document. If you would like to have more background information about CQRS, visit the Axon Framework website: www.axonframework.org. It contains links to background information.

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

1.1.2. What is Axon?

Axon Framework helps build scalable, extensible and maintainable applications by supporting developers apply the Command Query Responsibility Segregation (CQRS) architectural pattern. It does so by providing implementations, sometimes complete, sometimes abstract, of the most important building blocks, such as aggregates, repositories and event busses (the dispatching mechanism for events). Furthermore, Axon provides annotation support, which allows you to build aggregates and event listeners without tying your code to Axon specific logic. This allows you to focus on your business logic, instead of the plumbing, and helps you 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 inside the JVM. However, for an application to be scalable, a single JVM is not enough. Therefore, Axon provides the `axon-integration` 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 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 occurred 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 customer place orders and query for their order status. With Axon, these components can be deployed on separate machines and scaled completely differently. They are kept up-to-date using the events, which Axon will dispatch to all subscribed components, regardless 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 (if you use maven) configure your pom.xml file to include the Axon binaries in your build.

1.2.1. Download Axon

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

This page offers a number of downloads. Typically, you would want to use the latest stable release. However, if you're eager to get started using the latest and greatest features, you could consider using the snapshot

releases instead. The downloads page contains a number of assemblies for you to download. Some of them only provide the Axon library itself, while others also provide the libraries that Axon depends on. There is also a "full" zip file, which contains Axon, its dependencies, the sources and the documentation, all in a single download.

If you really want to stay on the bleeding edge of development, you can 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>0.5</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.

Event Sourcing

The event sourcing repository that is provided in Axon (`XStreamFileSystemEventStore`) uses `XStream` by default. You need to add the following maven dependency to your project to use this repository implementation if you would like to use these defaults. If you provide your own serialization strategy based on another library, then you do not have to include this dependency.

```
<dependency>
  <groupId>com.thoughtworks.xstream</groupId>
  <artifactId>xstream</artifactId>
  <version>1.3.1</version>
</dependency>
```

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 dependency to use these connectors.

```
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-core</artifactId>
  <version>1.0.3.RELEASE</version>
</dependency>
```

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 scalable and extensible application easier. This means we are constantly looking for help in developing our framework.

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

- You can report any bugs, feature requests or ideas about improvements on our issue page: axonframework.org/issues. All ideas are welcome. Please be as exact as possible when reporting bugs. This will help us reproduce and thus solve the problem faster.
- If you have created a component for your own application that you think might be useful to include in the framework, send us a patch or a zip containing the source code. We will evaluate it and try to fit it in the framework. Please make sure code is properly documented using javadoc. This helps us to understand what is going on.
- If you know of any other way you think you can help us, please do not hesitate to contact us.

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](#) 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 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).

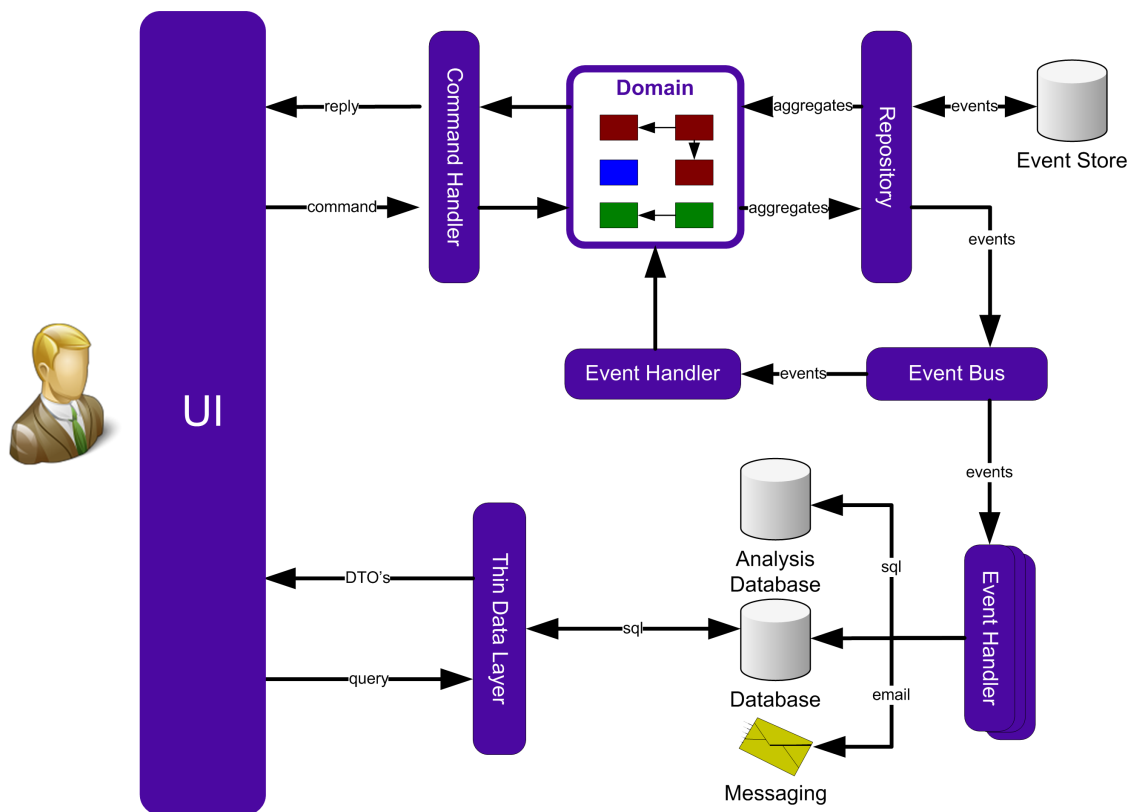


Figure 2.1. Architecture overview of a CQRS application

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 execute logic based on their contents. In some cases, however, you would also want to do things regardless of the actual type of command, such as logging or authorization.

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

Domain Modeling

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

Repositories and Event Stores

Once the command execution is complete, the command handler saves the aggregate's state by handing it over to the repository. The repository does two things when it saves an aggregate. It persists the aggregate's state for future commands that need to execute on this aggregate. Furthermore, events are handed over to the event bus, which is reposonsible for dispatching them to all interested listeners.

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

Event Processing

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

Event listeners receive events and handle them. Some handlers might issue commands to update other aggregates, based on information in the event. An example of this is the placement of an order in an online store. When an order comes in, you might want to update stock information for all ordered items. The listener would listen to events regarding placed orders and send stock update commands to the inventory. Other event listeners will update data sources used for querying or 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.

The building blocks related to event handling and dispatching are explained in Chapter 6, *Event Processing*.

Querying for data

Some event handlers will update data in data sources, such as tables in a database. 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.

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 receiving 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. 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)`, where `T` is the type of Command this Handler can process. 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 either `null` or `Void.TYPE`. The latter being 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 3.2, “Configuring the Command Bus”.

Annotation support

Comparable to the annotation support for Event Listeners, you can also use any POJO as command handler. The added advantage is that you can configure a single class to process several types of (related) commands. Just add the `@CommandHandler` annotated to your methods to turn them into a command handler. These methods may only accept a single parameter, which is the command to process. 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 needs the `CommandBus` instance. Use the `subscribe()` method on the adapter to subscribe the annotated handlers to the command bus using the correct command type.

If you use Spring, you may also define an `AnnotationCommandHandlerBeanPostProcessor`. This post processor detects any beans that have an `@CommandHandler` annotated method in them and wrap them in an `AnnotationCommandHandlerAdapter` automatically. They will also be automatically subscribed to the `CommandBus`.



Note

Note that you need to be careful when mixing manual wrapping and the use of the post processor. This might result in command handler being subscribed twice. This does not have to be a problem for most command handlers, since only a single command handler can be subscribed to a specific type of command at any one time. Their subscriptions will just overwrite each other.

3.2. Configuring the Command Bus

The Command Bus is the mechanism that dispatches commands to their respective Command Handler. Though similar to the Event Bus, there is a very clear distinction to be made between the command bus and the event bus. Where Events are published to all registered listeners, commands are 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`. 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.

3.3. Command Handler Interceptors

One of the advantages of using a command bus is the ability to undertake action based on all incoming commands, such as logging or authentication. The `SimpleCommandBus` provides the ability to register interceptors. These interceptors provide the ability to take action both before and after command processing.

Interceptors must implement the `CommandHandlerInterceptor` interface. This interface declares two methods, `beforeCommandHandling()` and `afterCommandHandling()`, that both take two parameters: a `CommandContext` and a `CommandHandler`. The first contains the actual command and provides the possibility to add meta-data to the command. This meta-data is not forwarded to the

command handler, but is intended for the command handler interceptor itself. You could, for example, store transactional information in the context if your transactional boundary is at the command handling. The second parameter, the `CommandHandler` is the command handler that will process or has processed the command. You could, for example, base authorization requirements on information in the command handler.

If you use annotation support, the `AnnotationCommandHandlerAdapter` is passed as the command handler. You may call `getTarget()` on it to obtain the actual annotated command handler. To obtain a reference to the method that handles the command, you can use the `findCommandHandlerMethodFor(Object command)` method. You could, for example, use the reference to this method to find security-related annotations and perform authorization on them.

3.3.1. Managing transactions

In some cases, it is desirable to set a transaction scope around the command handling process. For example when using synchronous event handling with event handlers that update tables in a database in combination with the `JpaEventStore`. By setting the transaction scope in the command dispatching process, all changes can be performed within a single transaction. This provides full consistency guarantees.

Axon provides the `SpringTransactionalInterceptor`, which uses Spring's `PlatformTransactionManager` to manage the actual transactions. A transaction is committed when command handling is successful, or rolled back if the command handler (or one of the downstream interceptors) threw an exception.

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.

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 will care about the intent (e.g. to send an address change confirmation email to the customer), while others don't (e.g. database updating event listeners). The latter will listen to events of the abstract type, while the former will listen to the concrete subtypes.

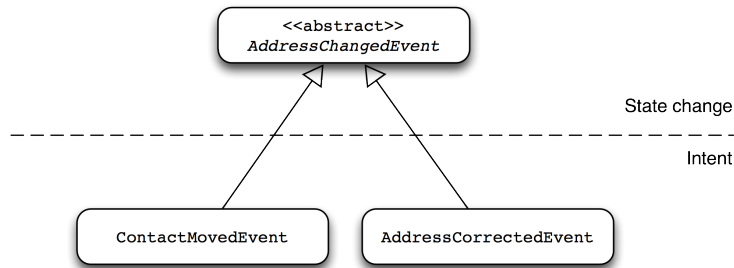


Figure 4.1. Adding intent to events

There is a special type of `DomainEvent`, which has a special meaning: `theAggregateDeletedEvent`. This event can be extended to indicate that the event indicates a migration to a "deleted" state of the aggregate. Repositories must consider aggregates that have applied such an event as deleted. Loading such an aggregate in again 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, "Using Snapshot Events" 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 over looked. Examples of application events are the expiry of a user session, or the notification of an email being successfully send. 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 attached, which means that if the garbage collector cleans up the source, or when the event is serialized and deserialized, 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. This object 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.

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.

VersionedAggregateRoot

The `VersionedAggregateRoot` interface provides the information needed by repositories to perform optimistic locking. The only method added to the `AggregateRoot` interface is `getLastCommittedEventSequenceNumber`, which returns the sequence number of the event that was last committed. See the section called “`LockingRepository`” for more information about the abstract `LockingRepository` implementation.

AbstractAggregateRoot

The `AbstractAggregateRoot` is a basic implementation that provides a `registerEvent(DomainEvent)` method that you can call in your business logic method to have an

event added to the list of uncommitted events. The `AbstractAggregateRoot` will keep track of all uncommitted registered events and make sure they are forwarded to the event bus when the aggregate is saved to a repository.

4.2.2. Event sourcing 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 implementation to help you on your way.

EventSourcedAggregateRoot

The `EventSourcedAggregateRoot` defines an extra method, `initializeState()`, on top of the `VersionedAggregateRoot` 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 the `apply()` method. 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.

```
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.

```
public class MyAggregateRoot extends AbstractEventSourcedAggregateRoot {
    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 class (the subclass) 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, an `UnhandledEventException` is thrown.

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.



Tip

An Aggregate will only contain fields of properties it uses for validation or business logic decisions. That means you will likely have some events that have no direct effect on any fields in the aggregate. In that case you can choose to create a `handleOtherEvents(DomainEvent event)` method with an empty body. This

handler will be called for any event for which there is no specific handler, preventing any exception being thrown. Do consider, however, that doing so may result in unexpected behavior if an event handler for a specific type of event is forgotten.

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 two methods: `load(identifier)` and `save(aggregate)`.

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

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.

Note that the `Repository` interface does not prescribe a `delete(identifier)` method. This is because not all types of repositories use that functionality. Of course, nothing withholds you from adding it to your repository implementation.

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 altogether.

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 abstract `EventSourcingRepository` implementation provides the basic functionality needed by any event sourcing repository in the Axon Framework. It depends on an `EventStore`, which abstracts the actual storage mechanism for the events. See Section 5.3, “Event store implementations”.

The `EventSourcingRepository` has two abstract methods: `getTypeIdentifier()` and `instantiateAggregate(identifier)`. The first is a value passed to the event store that provides information about the type of aggregate that the events relate to. A good starting point to use as return value is the simple name of a class (i.e. the fully qualified class name without the package name). The second method requires you to create an uninitialized instance of the aggregate using the given identifier. The repository will initialize this instance with the events obtained from the event store.

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.

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 an `EventSerializer` 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 only downside of this event store is that it 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 `XStream` version, the `JpaEventStore` supports transactions. The JPA event store can also load events based on their timestamps.

To use the `JpaEventStore`, you must have the `javax.persistence` annotations on your classpath. Furthermore, you should configure your persistence context (defined in `META-INF/persistence.xml` file) to contain the classes `org.axonframework.eventstore.jpa.DomainEventEntry` and `org.axonframework.eventstore.jpa.SnapshotEventEntry`.

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 a 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.

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) to serialize Domain Events into XML and vice versa. `XStream` is very fast and is more flexible than Java Serialization. For example, if you remove a field from a class, you can still deserialize instances from that class using the old XML. `XStream` will simply ignore that field. 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.

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

5.4. Using Snapshot Events

When aggregates live for a long time, and their state constantly change, 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 whether 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.

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 altogether.



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.

Triggering snapshot creation

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 does not provide a triggering mechanism (yet).

However, Axon does provide an interface that instances that produce snapshots should implement: `SnapshotProducer`. Typically, this interface is implemented by an aggregate root, since that is typically the only object that has full access to the aggregate's full state information.

Initializing an aggregate based on a Snapshot Event

A snapshot event is just a regular `DomainEvent`. That means a snapshot event is handled just like any other domain event. When using annotations to demarcate event handlers (`@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;
    }
}
```

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 abstract 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 an implementation of the event bus: `SimpleEventBus`. The `SimpleEventBus` manages subscribed `EventListener`s and forwards 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.

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, however, 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 configuring a bean of type `AnnotationEventListenerBeanPostProcessor`. This post processor 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 until 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 paragraphs 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 `AsynchronousEventHandlerWrapper` 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 by 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 identifier 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 the `AnnotationEventListenerBeanPostProcessor`), the annotated bean will be automatically wrapped in an `AsynchronousEventHandlerWrapper` if the bean is annotated with `@AsynchronousEventListener`. In that case, an `Executor` must have been configured on the `AnnotationEventListenerAdapter` or the `AnnotationEventListenerBeanPostProcessor`. If no `Executor` is provided, an exception is thrown.

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.

```
@AsynchronousEventListener(sequencePolicyClass = 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.

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 or if the Event Bus does not support transactions.

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 implementation uses Spring's `PlatformTransactionManager` as underlying transaction mechanism. That means the `SpringTransactionManager` can manage any transactions in resources that Spring supports.

7. 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.

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>
```

7.1. Wiring 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.

7.2. 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>

```

7.3. Wiring the command bus

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>

```

7.4. 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>
```

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

7.5. Wiring the event store

All event sourcing repositories 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 -->
```