Axon Framework

Reference Guide

Table of Contents

About this guide 1.1

Part I - Getting started

Introduction 2.1
Architecture overview 2.2
Messaging concepts 2.3
Configuration API 2.4

Part II - Domain Logic

Command Model 3.1
Event Handling 3.2
Query Handling 3.3
Sagas 3.4
Testing 3.5

Part III - Infrastructure components

Command Dispatching 4.1
Event Processing 4.2
Query Processing 4.3
Repository and Event Store 4.4
Spring Boot AutoConfiguration 4.5

Part IV - Advanced tuning

Advanced customizations 5.1
Performance Tuning 5.2

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 Axon Framework Background. If you're eager to get started building your own application using Axon, go quickly to Getting Started. If you're interested in helping out building the Axon Framework, Contributing will contain the information you require. All help is welcome. Finally, this chapter covers some legal concerns in License.

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 Axon Framework Background. If you're eager to get started building your own application using Axon, go quickly to Getting Started. If you're interested in helping out building the Axon Framework, Contributing will contain the information you require. All help is welcome. Finally, this chapter covers some legal concerns in License.

Axon Framework Background

A brief history

The demands on software projects increase rapidly as time progresses. Companies 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 web applications target the audience of potentially billions of people, making scalability an indisputable requirement.

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

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

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 without tying your code to Axon specific logic. This allows you to focus on your business logic, instead of the plumbing, and helps you to make your code easier to test in isolation.

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

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

When to use Axon?

Not every application will benefit from Axon. Simple CRUD (Create, Read, Update, Delete) applications which are not expected to scale will probably not benefit from CQRS or Axon. However, there is a wide variety of applications that do benefit from Axon.

Applications that will 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 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, 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.

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).

Download Axon

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

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

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

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>${axon.version}</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.

Infrastructure requirements

Axon Framework doesn't impose many requirements on the infrastructure. It has been built and tested against Java 8, making that more or less the only requirement.

Since Axon doesn't create any connections or threads by itself, it is safe to run on an Application Server. Axon abstracts all asynchronous behavior by using `Executor` s, meaning that you can easily pass a container managed Thread Pool, for example. If you don't use a full blown 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.

When you're stuck

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

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

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 applications 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 for improvements on our issue page: github.com/AxonFramework/AxonFramework/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 mailing list.

Commercial Support

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

For more information about AxonIQ and its services, visit axoniq.io or axoniq.io/services.

License information

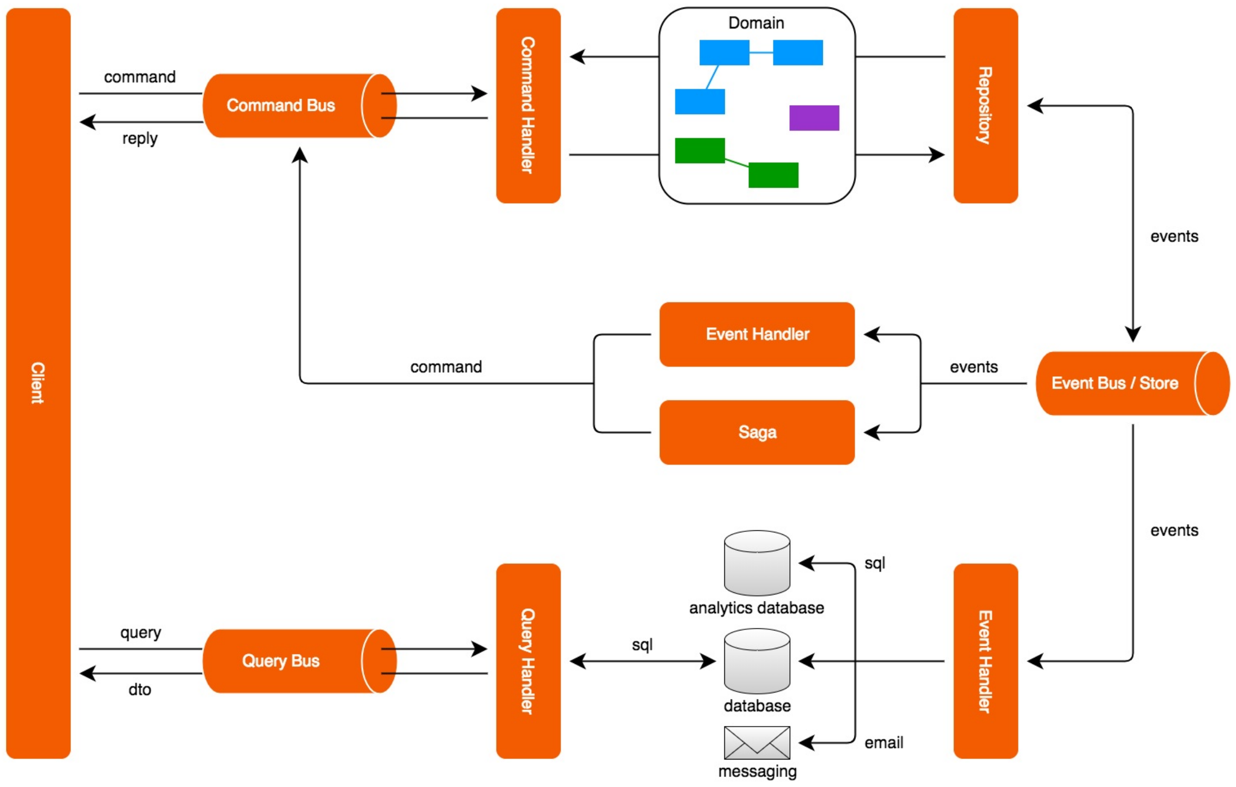
The Axon Framework and its documentation are licensed under the Apache License, Version 2.0. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0.

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.

Architectural Overview

CQRS on itself is a very simple pattern. It only prescribes 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 blocks 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).



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

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

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.

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 others 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.

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. By default, synchronous processing will process event listeners in the same transaction that also processed the command.

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.

Since Axon 3.1 the framework provides components to handle queries. The Query Bus receives queries and routes them to the Query Handlers. A query handler is registered at the query bus with both the type of query it handles as well as the type of response it providers.

Both the query and the result type are typically simple, read-only DTO objects. The contents of these DTOs 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).

It is possible to register multiple query handlers for the same type of query and type of response. When dispatching queries, the client can indicate whether he wants a result from one or from all available query handlers.

Axon Module Structure

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

As of Axon 2.1, all modules are OSGi compatible bundles. This means they contain the required headers in the manifest file and declare the packages they import and export. At the moment, only the Slf4J bundle (1.7.0 <= version < 2.0.0) is required. All other imports are marked as optional, although you're very likely to need others.

Main modules

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

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

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

The Distributed CommandBus modules contain implementations that can be used to distribute commands over multiple nodes. It comes with JGroups and Spring Cloud Connectors that are used to connect these nodes.

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

The Spring module allows Axon components to be configured in the Spring Application context. It also provides a number of building block implementations specific to Spring Framework, such as an adapter for publishing and retrieving Axon Events on a Spring Messaging Channel.

MongoDB is a document based NoSQL database. The Mongo module provides Event and Saga Store implementations that store event streams and sagas in a MongoDB database.

Several AxonFramework components provide monitoring information. The Metrics module provides basic implementations based on Codehale to collect the monitoring information.

Working with Axon APIs

CQRS is an architectural pattern, making it impossible to provide a single solution that fits all projects. Axon Framework does not try to provide that one solution, obviously. Instead, Axon provides implementations that follow best practices and the means to tweak each of those implementations to your specific requirements.

Almost all infrastructure building blocks will provide hook points (such as Interceptors, Resolvers, etc.) that allow you to add application-specific behavior to those building blocks. In many cases, Axon will provide implementations for those hook points that fit most use cases. If required, you can simply implement your own.

Non-infrastructural objects, such as Messages, are generally immutable. This ensures that these objects are safe to use in a multi-threaded environment, without side-effects.

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

Spring Support

Axon Framework provides extensive support for Spring, but does not require you to use Spring in order to use Axon. All components can be configured programmatically and do not require Spring on the classpath. However, if you do use Spring, much of the configuration is made easier with the use of Spring's annotation support.

Messaging concepts

One of the core concepts in Axon is messaging. All communication between components is done using message objects. This gives these components the location transparency needed to be able to scale and distribute these components when necessary.

Although all these messages implement the `Message` interface, there is a clear distinction between the different types of Messages and how they are treated.

All Messages contain a payload, meta data and unique identifier. The payload of the message is the functional description of what the message means. The combination of the class name of this object and the data it carries, describe the application's meaning of the message. The meta data allows you to describe the context in which a message is being sent. You can, for example, store tracing information, to allow the origin or cause of messages to be tracked. You can also store information to describe the security context under which a command is being executed.

> **Note**
>
> Note that all Messages are immutable. Storing data in a Message actually means creating a new Message based on the previous one, with extra information added to it. This guarantees that Messages are safe to use in a multi-threaded and distributed environment.

Commands

Commands describe an intent to change the application's state. They are implemented as (preferably read-only) POJOs that are wrapped using one of the `CommandMessage` implementations.

Commands always have exactly one destination. While the sender doesn't care which component handles the command or where that component resides, it may be interesting in knowing the outcome of it. That's why Command messages sent over the Command Bus allow for a result to be returned.

Events

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

When Events are dispatched, Axon wraps them in an `EventMessage`. The actual type of Message used depends on the origin of the Event. When an Event is raised by an Aggregate, it is wrapped in a `DomainEventMessage` (which extends `EventMessage`). All other Events are wrapped in an `EventMessage.` Aside from common `Message` attributes like a unique Identifier an `EventMessage` also contains a timestamp. The `DomainEventMessage` additionally contains the type and identifier of the aggregate that raised the Event. It also contains the sequence number of the event in the aggregate's event stream, which allows the order of events to be reproduced.

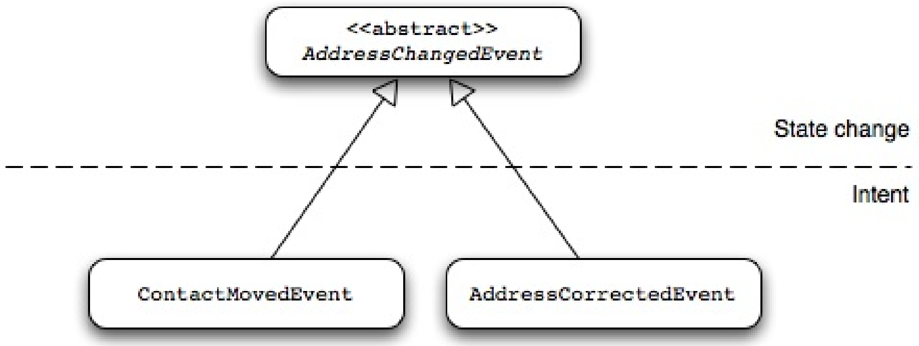
> **Note**
>
> Even though the `DomainEventMessage` contains a reference to the Aggregate Identifier, you should always include the identifier in the actual Event itself as well. The identifier in the DomainEventMessage is used by the EventStore to store events and may not always provide a reliable value for other purposes.

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

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

Although not enforced, it is good practice to make domain events immutable, preferably by making all fields final and by initializing the event within the constructor. Consider using a Builder pattern if Event construction is too cumbersome.

> **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).



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

Queries

Queries describe a request for information or state. A query can have multiple handlers. When dispatching queries, the client indicates whether he wants a result from one or from all available query handlers.

Unit of Work

The Unit of Work is an important concept in the Axon Framework, though in most cases you are unlikely to interact with it directly. The processing of a message is seen as a single unit. The purpose of the Unit of Work is to coordinate actions performed during the processing of a message (Command, Event or Query). Components can register actions to be performed during each of the stages of a Unit of Work, such as onPrepareCommit or onCleanup.

You are unlikely to need direct 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 Handler receives the Unit Of Work through a parameter in the handle method. If you use annotation support, you may add a parameter of type `UnitOfWork` to your annotated method. In other locations, you can retrieve the Unit of Work bound to the current thread by calling `CurrentUnitOfWork.get()`. Note that this method will throw an exception if there is no Unit of Work bound to the current thread. Use `CurrentUnitOfWork.isStarted()` to find out if one is available.

One reason to require access to the current Unit of Work is to attach resources that need to be reused several times during the course of message processing, or if created resources need to be cleaned up when the Unit of Work completes. In such case, the `unitOfWork.getOrComputeResource()` and the lifecycle callback methods, such as `onRollback()`, `afterCommit()` and `onCleanup()` allow you to register resources and declare actions to be taken during the processing of this Unit of Work.

> **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 others 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. Use `unitOfWork.onCommit(..)` to register actions that need to be taken when the Unit of Work is being committed.

Your handlers may throw an Exception as a result of processing a message. By default, unchecked exceptions will cause the UnitOfWork to roll back all changes. As a result, scheduled side effects are cancelled.

Axon provides a few Rollback strategies out-of-the-box:

- `RollbackConfigurationType.NEVER`, will always commit the Unit of Work,
- `RollbackConfigurationType.ANY_THROWABLE`, will always roll back when an exception occurs,
- `RollbackConfigurationType.UNCHECKED_EXCEPTIONS`, will roll back on Errors and Runtime

  Exception
- `RollbackConfigurationType.RUNTIME_EXCEPTION`, will roll back on Runtime Exceptions (but not on Errors)

When using Axon components to process messages, the lifecycle of the Unit of Work will be automatically managed for you. If you choose not to use these components, but implement processing yourself, you will need to programmatically start and commit (or roll back) a Unit of Work instead.

In most cases, the `DefaultUnitOfWork` will provide you with the functionality you need. It expects processing to happen within a single thread. To execute a task in the context of a Unit Of Work, simply call `UnitOfWork.execute(Runnable)` or `UnitOfWork.executeWithResult(Callable)` on a new `DefaultUnitOfWork`. The Unit Of Work will be started and committed when the task completes, or rolled back if the task fails. You can also choose to manually start, commit or rollback the Unit Of Work if you need more control.

Typical usage is as follows:

```
UnitOfWork uow = DefaultUnitOfWork.startAndGet(message);
// then, either use the autocommit approach:
uow.executeWithResult(() -> ... logic here);

// or manually commit or rollback:
try {
    // business logic comes here
    uow.commit();
} catch (Exception e) {
    uow.rollback(e);
    // maybe rethrow...
}
```

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

- Active phase: this is where the Unit of Work is started. The Unit of Work is generally registered with the current thread in this phase (through `CurrentUnitOfWork.set(UnitOfWork)`). Subsequently the message is typically handled by a message handler in this phase.

- Commit phase: after processing of the message is done but before the Unit of Work is committed, the `onPrepareCommit` listeners are invoked. If a Unit of Work is bound to a transaction, the `onCommit` listeners are invoked to commit any supporting transactions.

When the commit succeeds, the `afterCommit` listeners are invoked. If a commit or any step before fails, the `onRollback` listeners are invoked. The message handler result is contained in the `ExecutionResult` of the Unit Of Work, if available.

- Cleanup phase: This is the phase where any of the resources held by this Unit of Work (such as locks) are to be released. If multiple Units Of Work are nested, the cleanup phase is postponed until the outer unit of work is ready to clean up.

The message 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 message handlers. After the handler completed, Axon will try to commit the actions registered with the Unit Of Work.

It is possible to bind a transaction to a Unit of Work. Many components, such as the CommandBus and QueryBus implementations and all asynchronously processing Event Processors, allow you to configure a Transaction Manager. This Transaction Manager will then be used to create the transactions to bind to the Unit of Work that is used to manage the process of a Message.

When application components need resources at different stages of message processing, such as a Database Connection or an EntityManager, these resources can be attached to the Unit of Work. The `unitOfWork.getResources()` method allows you to access the resources attached to the current Unit of Work. Several helper methods are available on the Unit of Work directly, to make working with resources easier.

When nested Units of Work need to be able to access a resource, it is recommended to register it on the root Unit of Work, which can be accessed using `unitOfWork.root()`. If a Unit of Work is the root, it will simply return itself.

Configuration API

Axon keeps a strict separation when it comes to business logic and infrastructure configuration. In order to do so, Axon will provide a number of building blocks that take care of the infrastructural concerns, such as transaction management around a message handler. The actual payload of the messages and the contents of the handler are implemented in (as much as possible) Axon-independent Java classes.

To make the configuration of these infrastructure components easier and to define their relationship with each of the functional components, Axon provides a Configuration API.

Setting up a configuration

Getting a default configuration is very easy:

```
Configuration config = DefaultConfigurer.defaultConfiguration()
                                        .buildConfiguration();
```

This configuration provides the building blocks for dispatching Messages using implementations that handle messages on the threads that dispatch them.

Obviously, this configuration would not be very useful. You would have to register your Command Model objects and Event Handlers to this configuration to be useful.

To do so, use the `Configurer` instance returned by the `.defaultConfiguration()` method.

```
Configurer configurer = DefaultConfigurer.defaultConfiguration();
```

The configurer provides a multitude of methods that allow you to register these components. How to configure those is described in detail in each component's respective chapter.

The general form in which components are registered, is the following:

```
Configurer configurer = DefaultConfigurer.defaultConfiguration();
configurer.registerCommandHandler(c -> doCreateComponent());
```

Note the lambda expression in the `registerCommandBus` invocation. The `c` parameter of this expression is the configuration object that described the complete configuration. If your component requires any other components to function properly, it can use this configuration to retrieve them.

For example, to register a Command Handler that requires a serializer:

```
configurer.registerCommandHandler(c -> new MyCommandHandler(c.serializer());
```

Not all components have their explicit accessor method. To retrieve a component from the configuration, use:

```
configurer.registerCommandHandler(c -> new MyCommandHandler(c.getComponent(MyOtherComp
onent.class));
```

This component must be registered with the Configurer, using `configurer.registerComponent(componentType, builderFunction)`. The builder function will receive the `Configuration` object as input parameter.

Setting up a configuration using Spring

When using Spring, there is no need to explicitly use the `Configurer`. Instead, you can simply put the `@EnableAxon` on one of your Spring `@Configuration` classes.

Axon will use the Spring Application Context to locate specific implementations of building blocks and provide default for those that are not there. So, instead of registering the building blocks with the `Configurer`, in Spring you just have to make them available in the Application Context as `@Bean`s.

Command Model

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, one of them plays a dominant role when applied to Command processing in CQRS: the Aggregate.

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. The Command Model is used to process the incoming command, to validate it and define the outcome. Within this model, a Command Handler is responsible for handling commands of a certain type and taking action based on the information contained inside it.

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. This makes the Aggregate the prime building block for implementing a Command Model in any CQRS based application.

> **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.”

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

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

- implement `equals` and `hashCode` to ensure good equality comparison with other instances,

- implement a `toString()` method that provides a consistent result (equal identifiers should provide an equal toString() result), and
- preferably be `Serializable` .

The test fixtures (see Testing) will verify these conditions and fail a test when an Aggregate uses an incompatible identifier. Identifiers of type `String` , `UUID` and the numeric types are always suitable. Do **not** use primitive types as identifiers, as they don't allow for lazy initialization. Axon may, in some circumstances, falsely assume the default value of a primitive to be the value of the identifier.

> **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.

Aggregate implementations

An Aggregate is always accessed through a single Entity, called the Aggregate Root. Usually, the name of this Entity is the same as that of the Aggregate entirely. For example, an Order Aggregate may consist of an Order entity, which references several OrderLine entities. Order and Orderline together, form the Aggregate.

An Aggregate is a regular object, which contains state and methods to alter that state. Although not entirely correct according to CQRS principles, it is also possible to expose the state of the aggregate through accessor methods.

An Aggregate Root must declare a field that contains the Aggregate identifier. This identifier must be initialized at the latest when the first Event is published. This identifier field must be annotated by the `@AggregateIdentifier` annotation. If you use JPA and have JPA annotations on the aggregate, Axon can also use the `@Id` annotation provided by JPA.

Aggregates may use the `AggregateLifecycle.apply()` method to register events for publication. Unlike the `EventBus` , where messages need to be wrapped in an EventMessage, `apply()` allows you to pass in the payload object directly.

```
import static org.axonframework.commandhandling.model.AggregateLifecycle.apply;

@Entity // Mark this aggregate as a JPA Entity
public class MyAggregate {

    @Id // When annotating with JPA @Id, the @AggregateIdentifier annotation is not ne
cessary
    private String id;

    // fields containing state...

    @CommandHandler
    public MyAggregate(CreateMyAggregateCommand command) {
        // ... update state
        apply(new MyAggregateCreatedEvent(...));
    }

    // constructor needed by JPA
    protected MyAggregate() {
    }
}
```

Entities within an Aggregate can listen to the events the Aggregate publishes, by defining an `@EventHandler` annotated method. These methods will be invoked when an EventMessage is published (before any external handlers are published).

Event sourced aggregates

Besides storing the current state of an Aggregate, it is also possible to rebuild the state of an Aggregate based on the Events that it has published in the past. For this to work, all state changes must be represented by an Event.

For the major part, Event Sourced Aggregates are similar to 'regular' aggregates: they must declare an identifier and can use the `apply` method to publish Events. However, state changes in Event Sourced Aggregates (i.e. any change of a Field value) must be *exclusively* performed in an `@EventSourcingHandler` annotated method. This includes setting the Aggregate Identifier.

Note that the Aggregate Identifier must be set in the `@EventSourcingHandler` of the very first Event published by the Aggregate. This is usually the creation Event.

The Aggregate Root of an Event Sourced Aggregate must also contain a no-arg constructor. Axon Framework uses this constructor to create an empty Aggregate instance before initializing it using past Events. Failure to provide this constructor will result in an Exception when loading the Aggregate.

```
public class MyAggregateRoot {

    @AggregateIdentifier
    private String aggregateIdentifier;

    // fields containing state...

    @CommandHandler
    public MyAggregateRoot(CreateMyAggregate cmd) {
        apply(new MyAggregateCreatedEvent(cmd.getId()));
    }

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

    @EventSourcingHandler
    private void handleMyAggregateCreatedEvent(MyAggregateCreatedEvent event) {
        // make sure identifier is always initialized properly
        this.aggregateIdentifier = event.getMyAggregateIdentifier();

        // ... update state
    }
}
```

`@EventSourcingHandler` annotated methods are resolved using specific rules. These rules are the same for the `@EventHandler` annotated methods, and are thoroughly explained in Annotated Event Handler.

> **Note**
>
> Event handler methods may be private, as long as the security settings of the JVM allow the Axon Framework to change the accessibility of the method. This allows you to clearly separate the public API of your aggregate, which exposes the methods that generate events, from the internal logic, which processes the events.
>
> Most IDE's have an option to ignore "unused private method" warnings for methods with a specific annotation. Alternatively, you can add an `@SuppressWarnings("UnusedDeclaration")` annotation to the method to make sure you don't accidentally delete an Event handler method.

In some cases, especially when aggregate structures grow beyond just a couple of entities, it is cleaner to react on events being published in other entities of the same aggregate. However, since Event Handler methods are also invoked when reconstructing aggregate state, special precautions must be taken.

It is possible to `apply()` new events inside an Event Sourcing Handler method. This makes it possible for an Entity B to apply an event in reaction to Entity A doing something. Axon will ignore the apply() invocation when replaying historic events. Do note that, in this case, the Event of the inner `apply()` invocation is only published to the Entities after all Entities have received the first Event. If more events need to be published, based on the state of an entity after applying an inner event, use `apply(...).andThenApply(...)`

You can also use the static `AggregateLifecycle.isLive()` method to check whether the aggregate is 'live'. Basically, an aggregate is considered live if it has finished replaying historic events. While replaying these events, isLive() will return false. Using this `isLive()` method, you can perform activity that should only be done when handling newly generated events.

Complex Aggregate structures

Complex business logic often requires more than what an aggregate with only an aggregate root can provide. In that case, it is 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 events to trigger state transitions, but so does each of the entities within that aggregate.

> **Note** A common misinterpretation of the rule that Aggregates should not expose state is that none of the Entities should contain any property accessor methods. This is not the case. In fact, an Aggregate will probably benefit a lot if the entities *within* the aggregate expose state to the other entities in that same aggregate. However, is is recommended not to expose the state *outside* of the Aggregate.

Axon provides support for event sourcing in complex aggregate structures. Entities are, just like the Aggregate Root, simple objects. The field that declares the child entity must be annotated with `@AggregateMember`. This annotation tells Axon that the annotated field contains a class that should be inspected for Command and Event Handlers.

When an Entity (including the Aggregate Root) applies an Event, it is handled by the Aggregate Root first, and then bubbles down through all `@AggregateMember` annotated fields to its child entities.

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

- the Entity Type, directly referenced in a field;
- inside fields containing an `Iterable` (which includes all collections, such as `Set`, `List`, etc);

- inside the values of fields containing a `java.util.Map`

Handling commands in an Aggregate

It is recommended to define the Command Handlers directly in the Aggregate that contains the state to process this command, as it is not unlikely that a command handler needs the state of that Aggregate to do its job.

To define a Command Handler in an Aggregate, simply annotate the Command Handling method with `@CommandHandler`. The rules for an `@CommandHandler` annotated method are the same as for any handler method. However, Commands are not only routed by their payload. Command Messages carry a name, which defaults to the fully qualified class name of the Command object.

By default, `@CommandHandler` annotated methods allow the following parameter types:

- The first parameter is the payload of the Command Message. It may also be of type `Message` or `CommandMessage`, if the `@CommandHandler` annotation explicitly defined the name of the Command the handler can process. By default, a Command name is the fully qualified class name of the Command's payload.
- Parameters annotated with `@MetaDataValue` will resolve to the Meta Data value with the key as indicated on the annotation. If `required` is `false` (default), `null` is passed when the meta data value is not present. If `required` is `true`, the resolver will not match and prevent the method from being invoked when the meta data value is not present.
- Parameters of type `MetaData` will have the entire `MetaData` of a `CommandMessage` injected.
- Parameters of type `UnitOfWork` get the current Unit of Work injected. This allows command handlers to register actions to be performed at specific stages of the Unit of Work, or gain access to the resources registered with it.
- Parameters of type `Message`, or `CommandMessage` will get the complete message, with both the payload and the Meta Data. This is useful if a method needs several meta data fields, or other properties of the wrapping Message.

In order for Axon to know which instance of an Aggregate type should handle the Command Message, the property carrying the Aggregate Identifier in the Command object must be annotated with `@TargetAggregateIdentifier`. The annotation may be placed on either the field or an accessor method (e.g. a getter).

Commands that create an Aggregate instance do not need to identify the target aggregate identifier, although it is recommended to annotate the Aggregate identifier on them as well.

If you prefer to use another mechanism for routing Commands, the behavior can be overridden by supplying a custom `CommandTargetResolver`. This class should return the Aggregate Identifier and expected version (if any) based on a given command.

> **Note**
>
> When the `@CommandHandler` annotation is placed on an Aggregate's constructor, the respective command will create a new instance of that aggregate and add it to the repository. Those commands do not require to target a specific aggregate instance. Therefore, those commands do not require any `@TargetAggregateIdentifier` or `@TargetAggregateVersion` annotations, nor will a custom `CommandTargetResolver` be invoked for these commands.
>
> When a command creates an aggregate instance, the callback for that command will receive the aggregate identifier when the command executed successfully.

```
import static org.axonframework.commandhandling.model.AggregateLifecycle.apply;

public class MyAggregate {

    @AggregateIdentifier
    private String id;

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

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

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

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

public class DoSomethingCommand {

    @TargetAggregateIdentifier
    private String aggregateId;

    // code omitted for brevity

}
```

The Axon Configuration API can be used configure the Aggregate. For example:

```
Configurer configurer = ...
// to use defaults:
configurer.configureAggreate(MyAggregate.class);

// allowing customizations:
configurer.configureAggregate(
        AggregateConfigurer.defaultConfiguration(MyAggregate.class)
                           .configureCommandTargetResolver(c -> new CustomCommandTarge
tResolver())
);
```

`@CommandHandler` annotations are not limited to the aggregate root. Placing all command handlers in the root will sometimes lead to a large number of methods on the aggregate root, while many of them simply forward the invocation to one of the underlying entities. If that is the case, you may place the `@CommandHandler` annotation on one of the underlying entities' methods. For Axon to find these annotated methods, the field declaring the entity in the aggregate root must be marked with `@AggregateMember`. Note that only the declared type of the annotated field is inspected for Command Handlers. If a field value is null at the time an incoming command arrives for that entity, an exception is thrown.

```
public class MyAggregate {

    @AggregateIdentifier
    private String id;

    @AggregateMember
    private MyEntity entity;

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

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

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

    // code omitted for brevity. The event handler for MyAggregateCreatedEvent must se
t the id field
    // and somewhere in the lifecycle, a value for "entity" must be assigned to be abl
e to accept
    // DoSomethingInEntityCommand commands.
}

public class MyEntity {

    @CommandHandler
    public void handleSomeCommand(DoSomethingInEntityCommand command) {
        // do something
    }
}
```

> **Note**
>
> Note that each command must have exactly one handler in the aggregate. This means that you cannot annotate multiple entities (either root nor not) with @CommandHandler, that handle the same command type. In case you need to conditionally route a command to an entity, the parent of these entities should handle the command, and forward it based on the conditions that apply.
>
> The runtime type of the field does not have to be exactly the declared type. However, only the declared type of the `@AggregateMember` annotated field is inspected for `@CommandHandler` methods.

It is also possible to annotate Collections and Maps containing entities with `@AggregateMember`. In the latter case, the values of the map are expected to contain the entities, while the key contains a value that is used as their reference.

As a command needs to be routed to the correct instance, these instances must be properly identified. Their "id" field must be annotated with `@EntityId`. The property on the command that will be used to find the Entity that the message should be routed to, defaults to the name of the field that was annotated. For example, when annotating the field "myEntityId", the command must define a property with that same name. This means either a `getMyEntityId` or a `myEntityId()` method must be present. If the name of the field and the routing property differ, you may provide a value explicitly using `@EntityId(routingKey = "customRoutingProperty")`.

If no Entity can be found in the annotated Collection or Map, Axon throws an IllegalStateException; apparently, the aggregate is not capable of processing that command at that point in time.

> **Note**
>
> The field declaration for both the Collection or Map should contain proper generics to allow Axon to identify the type of Entity contained in the Collection or Map. If it is not possible to add the generics in the declaration (e.g. because you're using a custom implementation which already defines generic types), you must specify the type of entity used in the `entityType` property on the `@AggregateMember` annotation.

External Command Handlers

In certain cases, it is not possible, or desired to route a command directly to an Aggregate instance. In such case, it is possible to register a Command Handler object.

A Command Handler object is a simple (regular) object, which has `@CommandHandler` annotated methods. Unlike in the case of an Aggregate, there is only a single instance of a Command Handler object, which handles all commands of the types it declares in its methods.

```
public class MyAnnotatedHandler {

    @CommandHandler
    public void handleSomeCommand(SomeCommand command, @MetaDataValue("userId") String
 userId) {
        // whatever logic here
    }

    @CommandHandler(commandName = "myCustomCommand")
    public void handleCustomCommand(SomeCommand command) {
       // handling logic here
    }

}

// To register the annotated handlers to the command bus:
Configurer configurer = ...
configurer.registerCommandHandler(c -> new MyAnnotatedHandler());
```

Returning results from Command Handlers

In some cases, the component dispatching a Command needs information about the processing results of a Command. A Command handler method can return a value from its method. That value will be provided to the sender as the result of the command.

One exception is the `@CommandHandler` on an Aggregate's constructor. In this case, instead of returning the return value of the method (which is the Aggregate itself), the value of the `@AggregateIdentifier` annotated field is returned instead

> **Note**
>
> While it's possible to return results from Commands, it should be used sparsely. The intent of the command should never be in getting a value, as that would be an indication the message should be designed as a Query Message instead. A typical example for a Command result is the identifier of a newly created entity.

Event Handling

Event listeners are the components that act on incoming events. They typically execute logic based on decisions that have been made by the command model. Usually, this involves updating view models or forwarding updates to other components, such as 3rd party integrations. In some cases Event Handlers will throw Events themselves based on (patterns of) Events that they received, or even send Commands to trigger further changes.

Defining Event Handlers

In Axon, an object may declare a number of Event Handler methods, by annotating them with `@EventHandler`. The declared parameters of the method define which events it will receive.

Axon provides out-of-the-box support for the following parameter types:

- The first parameter is always the payload of the Event Message. In the case the Event Handlers doesn't need access to the payload of the message, you can specify the expected payload type on the `@EventHandler` annotation. When specified, the first parameter is resolved using the rules specified below. Do not configure the payload type on the annotation if you want the payload to be passed as a parameter.
- Parameters annotated with `@MetaDataValue` will resolve to the Meta Data value with the key as indicated on the annotation. If `required` is `false` (default), `null` is passed when the meta data value is not present. If `required` is `true`, the resolver will not match and prevent the method from being invoked when the meta data value is not present.
- Parameters of type `MetaData` will have the entire `MetaData` of an `EventMessage` injected.
- Parameters annotated with `@Timestamp` and of type `java.time.Instant` (or `java.time.temporal.Temporal`) will resolve to the timestamp of the `EventMessage`. This is the time at which the Event was generated.
- Parameters annotated with `@SequenceNumber` and of type `java.lang.Long` or `long` will resolve to the `sequenceNumber` of a `DomainEventMessage`. This provides the order in which the Event was generated (within the scope of the Aggregate it originated from).

- Parameters assignable to Message will have the entire `EventMessage` injected (if the message is assignable to that parameter). If the first parameter is of type message, it effectively matches an Event of any type, even if generic parameters would suggest otherwise. Due to type erasure, Axon cannot detect what parameter is expected. In such case, it is best to declare a parameter of the payload type, followed by a parameter of type Message.
- When using Spring and the Axon Configuration is activated (either by including the Axon Spring Boot Starter module, or by specifying `@EnableAxon` on your `@Configuration` file), any other parameters will resolve to autowired beans, if exactly one injectable candidate is available in the application context. This allows you to inject resources directly into `@EventHandler` annotated methods.

You can configure additional `ParameterResolver` s by implementing the `ParameterResolverFactory` interface and creating a file named `/META-INF/service/org.axonframework.common.annotation.ParameterResolverFactory` containing the fully qualified name of the implementing class. See Advanced Customizations for details.

In all circumstances, at most one event handler method is invoked per listener instance. Axon will search for the most specific method to invoke, using following rules:

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

```
// assume EventB extends EventA
// and    EventC extends EventB
// and    a single instance of SubListener is registered

public class TopListener {

    @EventHandler
    public void handle(EventA event) {
    }

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

public class SubListener extends TopListener {

    @EventHandler
    public void handle(EventB event) {
    }
}
```

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

Registering Event Handlers

Event Handling components are defined using an `EventHandlingConfiguration` class, which is registered as a module with the global Axon `Configurer`. Typically, an application will have a single `EventHandlingConfiguration` defined, but larger more modular applications may choose to define one per module.

To register objects with `@EventHandler` methods, use the `registerEventHandler` method on the `EventHandlingConfiguration`:

```
// define an EventHandlingConfiguration
EventHandlingConfiguration ehConfiguration = new EventHandlingConfiguration()
    .registerEventHandler(conf -> new MyEventHandlerClass());

// the module needs to be registered with the Axon Configuration
Configurer axonConfigurer = DefaultConfigurer.defaultConfiguration()
    .registerModule(ehConfiguration);
```

See Event Handling Configuration for details on registering event handlers using Spring AutoConfiguration.

Query Handling

Query handling components act on incoming query messages. They typically read data from the view models created by the Event listeners. Query handling components typically do not raise new Events or send Commands.

Defining Query Handlers

In Axon, an object may declare a number of Query Handler methods, by annotating them with `@QueryHandler` . The declared parameters of the method define which messages it will receive.

By default, `@QueryHandler` annotated methods allow the following parameter types:

- The first parameter is the payload of the Query Message. It may also be of type `Message` or `QueryMessage` , if the `@QueryHandler` annotation explicitly defined the name of the Query the handler can process. By default, a Query name is the fully qualified class name of the Query's payload.
- Parameters annotated with `@MetaDataValue` will resolve to the Meta Data value with the key as indicated on the annotation. If `required` is `false` (default), `null` is passed when the meta data value is not present. If `required` is `true` , the resolver will not match and prevent the method from being invoked when the meta data value is not present.
- Parameters of type `MetaData` will have the entire `MetaData` of a `QueryMessage` injected.
- Parameters of type `UnitOfWork` get the current Unit of Work injected. This allows query handlers to register actions to be performed at specific stages of the Unit of Work, or gain access to the resources registered with it.
- Parameters of type `Message` , or `QueryMessage` will get the complete message, with both the payload and the Meta Data. This is useful if a method needs several meta data fields, or other properties of the wrapping Message.

You can configure additional `ParameterResolver` s by implementing the `ParameterResolverFactory` interface and creating a file named `/META-INF/service/org.axonframework.common.annotation.ParameterResolverFactory` containing the fully qualified name of the implementing class. See Advanced Customizations for details.

In all circumstances, at most one query handler method is invoked per query handling instance. Axon will search for the most specific method to invoke, using following rules:

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

Note that similar to command handling, and unlike event handling, query handling does not take the class hierarchy of the Query message into account.

```
// assume QueryB extends QueryA
// and    QueryC extends QueryB
// and    a single instance of SubHandler is registered

public class TopHandler {

    @QueryHandler
    public MyResult handle(QueryA query) {
    }

    @QueryHandler
    public MyResult handle(QueryB query) {
    }

    @QueryHandler
    public MyResult handle(QueryC query) {
    }
}

public class SubHandler extends TopHandler {

    @QueryHandler
    public MyResult handleEx(QueryB query) {
    }
}
```

In the example above, the handler method of `SubHandler` will be invoked for queries for `QueryB` and result `MyResult`; the handler methods of `TopHandler` are invoked for queries for `QueryA` and `QueryC` and result `MyResult`.

Registering Query Handlers

It is possible to register multiple query handlers for the same query name and type of response. When dispatching queries, the client can indicate whether he wants a result from one or from all available query handlers.

Using Spring

When using Spring AutoConfiguration, all singleton Spring beans are scanned for methods that have the `@QueryHandler` annotation. For each method that is found, a new query handler is registered with the query bus.

Using Configuration API

It is also possible to use the Configuration API to register query handlers. To do so, use the `registerQueryHandler` method on the `Configurer` class:

```
// Sample query handler
public class MyQueryHandler {
    @QueryHandler
    public String echo(String echo) {
        return echo;
    }
}

...

// To register your query handler
Configurer axonConfigurer = DefaultConfigurer.defaultConfiguration()
    .registerQueryHandler(conf -> new MyQueryHandler);
```

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 can be used to manage these BASE transactions. They respond on Events and may dispatch 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.

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, there could be many ways for a Saga to end.

In Axon, Sagas are classes that define one or more `@SagaEventHandler` methods. Unlike regular Event Handlers, multiple instances of a Saga may exist at any time. Sagas are managed by a single Processor (Tracking or Subscribing), which is dedicated to dealing with Events for that specific Saga type.

Life Cycle

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.

In a Saga, Event Handlers are annotated with `@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 `SagaLifecycle.end()` from inside the Saga to end the life cycle. This allows you to conditionally end the Saga.

Event Handling

Event Handling in a Saga is quite comparable to that of a regular Event Listener. The same rules for method and parameter resolution are valid here. There is one major difference, though. While there is a single instance of an Event Listener that deals with 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.

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 order in which `@SagaEventHandler` annotated methods are evaluated is identical to that of `@EventHandler` methods (see Annotated Event Handler). A method matches if the parameters of the handler method match the incoming Event, and if the saga has an association with the property defined on the handler method.

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

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

Managing associations

When a Saga manages a transaction across multiple 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 `SagaLifecycle.associateWith(String key, String/Number value)` method. Use the `SagaLifecycle.removeAssociationWith(String key, String/Number value)` method to remove a specific association.

> Note
>
> The API to associate domain concepts within a Saga intentionally only allows a `String` or a `Number` as the identifying value, since a `String` representation of the identifier is required for the association value entry which is stored. Using simple identifier values in the API with a straightforward `String` representation is by design, as a `String` column entry in the database makes the comparison between database engines simpler. It is thus intentionally that there is no `associateWith(String, Object)` for example, as the result of an `Object#toString()` call might provide unwieldy identifiers.

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 {

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

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

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

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

    // ...
}
```

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. 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. This will end this saga once the invoice is paid and the shipment has arrived.

Keeping track of Deadlines

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

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

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

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).

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".

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

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

Injecting Resources

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

For that purpose, there is the `ResourceInjector`. It is used by the `SagaRepository` to inject resources into a Saga. Axon provides a `SpringResourceInjector`, which injects annotated fields and methods with Resources from the Application Context, and a `SimpleResourceInjector`, which injects resources that have been registered with it into `@Inject` annotated methods and fields.

> **Tip**
>
> 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.

The `SimpleResourceInjector` allows for a pre-specified collection of resources to be injected. It scans the (setter) methods and fields of a Saga to find ones that are annotated with `@Inject`.

When using the Configuration API, Axon will default to the `ConfigurationResourceInjector`. It will inject any resource available in the Configuration. Components like the `EventBus`, `EventStore`, `CommandBus` and `CommandGateway` are available by default, but you can also register your own components using `configurer.registerComponent()`.

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

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`.

Saga Manager

Like any component that handles Events, the processing is done by an Event Processor. However, since Sagas aren't singleton instances handling Events, but have individual life cycles, they need to be managed.

Axon supports life cycle management through the `AnnotatedSagaManager` , which is provided to an Event Processor to perform the actual invocation of handlers. It is initialized using the type of the Saga to manage, as well as a SagaRepository where Sagas of that type can be stored and retrieved. A single `AnnotatedSagaManager` can only manage a single Saga type.

When using the Configuration API, Axon will use sensible defaults for most components. However, it is highly recommended to define a `SagaStore` implementation to use. The `SagaStore` is the mechanism that 'physically' stores the Saga instances somewhere. The `AnnotatedSagaRepository` (the default) uses the `SagaStore` to store and retrieve Saga instances as they are required.

```
Configurer configurer = DefaultConfigurer.defaultConfiguration();
configurer.registerModule(
        SagaConfiguration.subscribingSagaManager(MySagaType.class)
                         // Axon defaults to an in-memory SagaStore, defining another
is recommended
                         .configureSagaStore(c -> new JpaSagaStore(...)));

// alternatively, it is possible to register a single SagaStore for all Saga types:
configurer.registerComponent(SagaStore.class, c -> new JpaSagaStore(...));
```

Saga Repository and Saga Store

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 handling 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 the `AnnotatedSagaRepository` implementation, which allows the lookup of Saga instances while guaranteeing that only a single instance of the Saga is accessed at the same time. It uses a `SagaStore` to perform the actual persistence of Saga instances.

The choice for the implementation to use depends mainly on the storage engine used by the application. Axon provides the `JdbcSagaStore` , `InMemorySagaStore` , `JpaSagaStore` and `MongoSagaStore` .

In some cases, applications benefit from caching Saga instances. In that case, there is a `CachingSagaStore` which wraps another implementation to add caching behavior. Note that the `CachingSagaStore` is a write-through cache, which means save operations are always immediately forwarded to the backing Store, to ensure data safety.

JpaSagaStore

The `JpaSagaStore` uses JPA to store the state and Association Values of Sagas. Sagas themselves do not need any JPA annotations; Axon will serialize the sagas using a `Serializer` (similar to Event serialization, you can use either a `JavaSerializer` or an `XStreamSerializer`).

The `JpaSagaStore` is configured with an `EntityManagerProvider` , which provides access to an `EntityManager` instance to use. This abstraction allows for the use of both application managed and container managed `EntityManager` s. Optionally, you can define the serializer to serialize the Saga instances with. Axon defaults to the `XStreamSerializer` .

JdbcSagaStore

The `JdbcSagaStore` uses plain JDBC to store stage instances and their association values. Similar to the `JpaSagaStore` , Saga instances don't need to be aware of how they are stored. They are serialized using a Serializer.

The `JdbcSagaStore` is initialized with either a `DataSource` or a `ConnectionProvider` . While not required, when initializing with a `ConnectionProvider` , it is recommended to wrap the implementation in a `UnitOfWorkAwareConnectionProviderWrapper` . It will check the current Unit of Work for an already open database connection, to ensure that all activity within a unit of work is done on a single connection.

Unlike JPA, the JdbcSagaRepository uses plain SQL statement to store and retrieve information. This may mean that some operations depend on the Database specific SQL dialect. It may also be the case that certain Database vendors provide non-standard features that you would like to use. To allow for this, you can provide your own `SagaSqlSchema` . The `SagaSqlSchema` is an interface that defines all the operations the repository needs to perform on the underlying database. It allows you to customize the SQL statement executed for each one of them. The default is the `GenericSagaSqlSchema` . Other implementations available are `PostgresSagaSqlSchema` , `Oracle11SagaSqlSchema` and `HsqlSagaSchema` .

MongoSagaStore

The `MongoSagaStore` stores the Saga instances and their associations in a MongoDB database. The `MongoSagaStore` stores all sagas in a single Collection in a MongoDB database. Per Saga instance, a single document is created.

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

The `MongoSagaStore` is initialized using a `MongoTemplate` and optionally a `Serializer`. The `MongoTemplate` provides a reference to the collection to store the Sagas in. Axon provides the `DefaultMongoTemplate`, which takes the `MongoClient` instance as well as the database name and name of the collection to store the sagas in. The database name and collection name may be omitted. In that case, they default to "axonframework" and "sagas", respectively.

Caching

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

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

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

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. Not only does this mean that tests expressed in terms of Events and Commands have a clear functional meaning, it also means that they hardly depend on any implementation choices.

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

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

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.

Although being more complex, the API of a command handling component is fairly easy. It has a 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. The `AggregateTestFixture` allows you to configure a certain infrastructure, composed of the necessary command handler and repository, and express your 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<MyAggregate> fixture;

    @Before
    public void setUp() {
        fixture = new AggregateTestFixture<>(MyAggregate.class);
    }

    @Test
    public void testFirstFixture() {
        fixture.given(new MyEvent(1))
               .when(new TestCommand())
               .expectSuccessfulHandlerExecution()
               .expectEvents(new MyEvent(2));
        /*
        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.

During the configuration phase (i.e. before the first "given" is provided), you provide the building blocks required to execute the test. Specialized versions of the event bus, command bus and event store are provided as part of the fixture. There are accessor methods in place to obtain references to them. Any Command Handlers not registered directly on the Aggregate need to be explicitly configured using the `registerAnnotatedCommandHandler` method. Besides the Annotated Command Handler, you can configure a wide variety of components and settings that define how the infrastructure around the test should be set up.

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

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

The execution phase allows you to provide a Command to be executed against the command handling component. The behavior of the invoked handler (either on the Aggregate or as an external handler) is monitored and compared to the expectations registered in the validation phase.

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

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.

The test fixture allows you to validate return values of your command handlers. You can explicitly define the expected return value, or simply require that the method successfully returned. You may also express any exceptions you expect the CommandHandler to throw.

The other component is validation of published events. 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 or more of the provided Event Matchers matches against one or more of the events in the actual list of events. Some matchers may not even match at all, while another matches against multiple others.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 often 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:

```
FixtureConfiguration<InvoicingSaga> fixture = new SagaTestFixture<>(InvoicingSaga.clas
s);
fixture.givenAggregate(invoiceId).published(new InvoiceCreatedEvent())
       .whenTimeElapses(Duration.ofDays(31))
       .expectDispatchedCommandsMatching(Matchers.listWithAllOf(aMarkAsOverdueCommand(
)));
       // or, to match against the payload of a Command Message only
       .expectDispatchedCommandsMatching(Matchers.payloadsMatching(Matchers.listWithAl
lOf(aMarkAsOverdueCommand())));
```

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

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

Often, Sagas will interact with resources. These resources aren't part of the Saga's state, but are injected after a Saga is loaded or created. The test fixtures allow 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 or fields (annotated with `@Inject`) on the Saga and invoke it with an available resource.

> **Tip**
>
> 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.

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

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

When the `registerCommandGateway(Class)` is used to register a gateway, it dispatches Commands to the CommandBus managed by the fixture. The behavior of the gateway is mostly defined by the `CallbackBehavior` defined on the fixture. If no explicit

`CallbackBehavior` is provided, callbacks are not invoked, making it impossible to provide any return value for the gateway.

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

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**
>
> The Fixture uses a `StubScheduler` for time based activity, such as scheduling events and advancing time. Fixtures will set the timestamp of any events sent to the Saga instance to the time of this scheduler. This means time is 'stopped' as soon as the fixture starts, and may be advanced deterministically using the `whenTimeAdvanceTo` and `whenTimeElapses` methods.

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 which 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 date and time or advance time to the next scheduled event. All these operations will return the events scheduled within the progressed interval.

Command Dispatching

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 Testing). The last major advantage is that it is very easy to switch between synchronous and asynchronous as well as local versus distributed command processing.

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

The next sections provide an overview of the tasks related to setting up a Command dispatching infrastructure with the Axon Framework.

The Command Gateway

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

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

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

Configuring the Command Gateway

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

The `RetryScheduler` is capable of scheduling retries when command execution has failed. The `IntervalRetryScheduler` is an implementation that will retry a given command at set intervals until it succeeds, or a maximum number of retries is done. When a command fails due to an exception that is explicitly non-transient, no retries are done at all. Note that the retry scheduler is only invoked when a command fails due to a `RuntimeException` . Checked exceptions are regarded "business exception" and will never trigger a retry. Typical usage of a `RetryScheduler` is when dispatching commands on a Distributed Command Bus. If a node fails, the Retry Scheduler will cause a command to be dispatched to the next node capable of processing the command (see Distributing the Command Bus).

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

The `CommandCallback` s are invoked for each command sent. This allows for some generic behavior for all Commands sent through this gateway, regardless of their type.

Creating a Custom Command Gateway

Axon allows a custom interface to be used as a Command Gateway. The behavior of each method declared in the interface is based on the parameter types, return type and declared exception. Using this gateway is not only convenient, it makes testing a lot easier by allowing you to mock your interface where needed.

This is how parameters affect the behavior of the CommandGateway:

- The first parameter is expected to be the actual command object to dispatch.
- Parameters annotated with `@MetaDataValue` will have their value assigned to the meta data field with the identifier passed as annotation parameter
- Parameters of type `MetaData` will be merged with the `MetaData` on the CommandMessage. Meta data defined by latter parameters will overwrite the meta data of earlier parameters, if their key is equal.

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

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

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

Exceptions have the following effect:

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

Finally, there is the possibility to use annotations:

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

```java
public interface MyGateway {

    // fire and forget
    void sendCommand(MyPayloadType command);

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

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

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

// To configure a gateway:
CommandGatewayFactory factory = new CommandGatewayFactory(commandBus);
// note that the commandBus can be obtained from the `Configuration` object returned o
n `configurer.initialize()`.
MyGateway myGateway = factory.createGateway(MyGateway.class);
```

The Command Bus

The Command Bus is the mechanism that dispatches commands to their respective Command Handlers. Each Command is always sent to exactly one command handler. If no command handler is available for the dispatched command, a

`NoHandlerForCommandException` exception 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.

Dispatching commands

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

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

If an application isn't directly interested in the outcome of a Command, the `dispatch(commandMessage)` method can be used.

SimpleCommandBus

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

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

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

AsynchronousCommandBus

As the name suggest, the `AsynchronousCommandBus` implementation executes Commands asynchronously from the thread that dispatches them. It uses an Executor to perform the actual handling logic on a different Thread.

By default, the `AsynchronousCommandBus` uses an unbounded cached thread pool. This means a thread is created when a Command is dispatched. Threads that have finished processing a Command are reused for new commands. Threads are stopped if they haven't processed a command for 60 seconds.

Alternatively, an `Executor` instance may be provided to configure a different threading strategy.

Note that the `AsynchronousCommandBus` should be shut down when stopping the application, to make sure any waiting threads are properly shut down. To shut down, call the `shutdown()` method. This will also shutdown any provided `Executor` instance, if it implements the `ExecutorService` interface.

DisruptorCommandBus

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

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

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

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

To construct a `DisruptorCommandBus` instance, you need an `EventStore`. This component is explained in Repositories and Event Stores.

Optionally, you can provide a `DisruptorConfiguration` instance, which allows you to tweak the configuration to optimize performance for your specific environment:

- Buffer size: The number of slots on the ring buffer to register incoming commands. Higher values may increase throughput, but also cause higher latency. Must always be a power of 2. Defaults to 4096.
- ProducerType: Indicates whether the entries are produced by a single thread, or multiple. Defaults to multiple.
- WaitStrategy: The strategy to use when the processor threads (the three threads taking care of the actual processing) need to wait for each other. The best WaitStrategy depends on the number of cores available in the machine, and the number of other processes running. If low latency is crucial, and the DisruptorCommandBus may claim cores for itself, you can use the `BusySpinWaitStrategy`. To make the Command Bus claim less of the CPU and allow other threads to do processing, use the `YieldingWaitStrategy`. Finally, you can use the `SleepingWaitStrategy` and `BlockingWaitStrategy` to allow other processes a fair share of CPU. The latter is suitable if the Command Bus is not expected to be processing full-time. Defaults to the `BlockingWaitStrategy`.

- Executor: Sets the Executor that provides the Threads for the `DisruptorCommandBus`. This executor must be able to provide at least 4 threads. 3 of the threads are claimed by the processing components of the `DisruptorCommandBus`. Extra threads are used to invoke callbacks and to schedule retries in case an Aggregate's state is detected to be corrupt. Defaults to a `CachedThreadPool` that provides threads from a thread group called "DisruptorCommandBus".

- TransactionManager: Defines the Transaction Manager that should ensure that the storage and publication of events are executed transactionally.

- InvokerInterceptors: Defines the `CommandHandlerInterceptor`s that are to be used in the invocation process. This is the process that calls the actual Command Handler method.

- PublisherInterceptors: Defines the `CommandHandlerInterceptor`s that are to be used in the publication process. This is the process that stores and publishes the generated events.

- RollbackConfiguration: Defines on which Exceptions a Unit of Work should be rolled back. Defaults to a configuration that rolls back on unchecked exceptions.

- RescheduleCommandsOnCorruptState: Indicates whether Commands that have been executed against an Aggregate that has been corrupted (e.g. because a Unit of Work was rolled back) should be rescheduled. If `false` the callback's `onFailure()` method will be invoked. If `true` (the default), the command will be rescheduled instead.

- CoolingDownPeriod: Sets the number of seconds to wait to make sure all commands are processed. During the cooling down period, no new commands are accepted, but existing commands are processed, and rescheduled when necessary. The cooling down period ensures that threads are available for rescheduling the commands and calling callbacks. Defaults to 1000 (1 second).

- Cache: Sets the cache that stores aggregate instances that have been reconstructed from the Event Store. The cache is used to store aggregate instances that are not in active use by the disruptor.

- InvokerThreadCount: The number of threads to assign to the invocation of command handlers. A good starting point is half the number of cores in the machine.

- PublisherThreadCount: The number of threads to use to publish events. A good starting point is half the number of cores, and could be increased if a lot of time is spent on IO.

- SerializerThreadCount: The number of threads to use to pre-serialize events. This defaults to 1, but is ignored if no serializer is configured.

- Serializer: The serializer to perform pre-serialization with. When a serializer is configured, the `DisruptorCommandBus` will wrap all generated events in a `SerializationAware` message. The serialized form of the payload and meta data is attached before they are published to the Event Store.

Command Interceptors

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

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

Message Dispatch Interceptors

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

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

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

Message Handler Interceptors

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

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

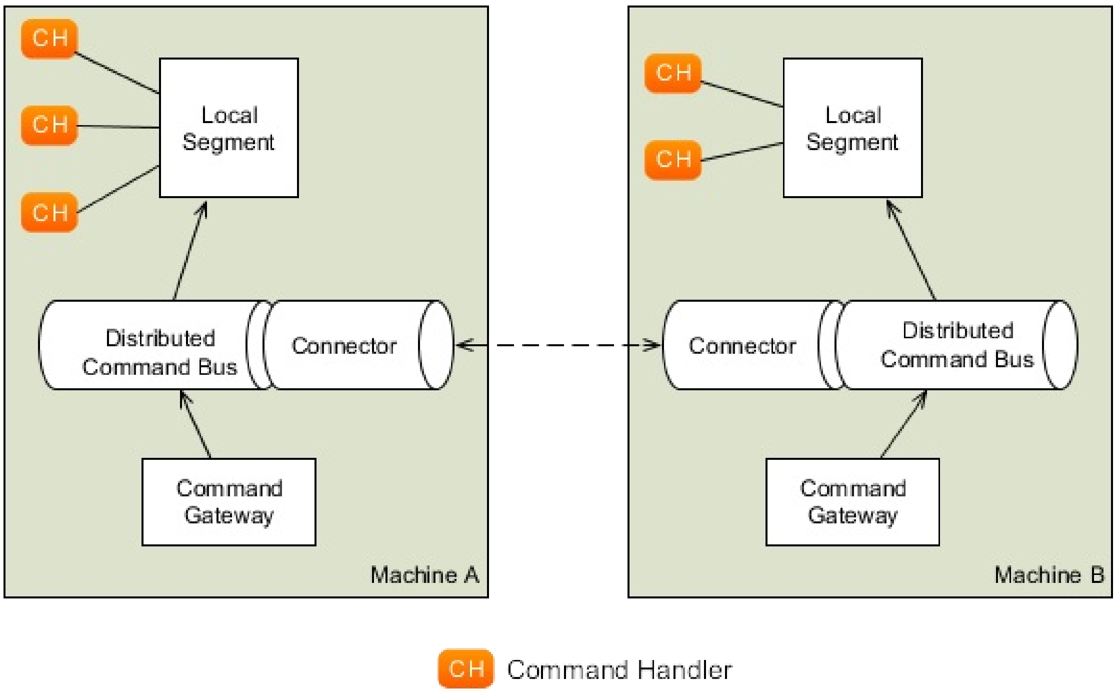
Unlike Dispatch Interceptors, Handler Interceptors are invoked in the context of the Command Handler. That means they can attach correlation data based on the Message being handled to the Unit of Work, for example. This correlation data will then be attached to messages being created in the context of that Unit of Work.

Handler Interceptors are also typically used to manage transactions around the handling of a command. To do so, register a `TransactionManagingInterceptor`, which in turn is configured with a `TransactionManager` to start and commit (or roll back) the actual transaction.

Distributing the Command Bus

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

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



> **Note**
>
> While the distributed command bus itself is part of the Axon Framework Core module, it requires components that you can find in one of the *axon-distributed-commandbus-...* modules. If you use Maven, make sure you have the appropriate dependencies set. The groupId and version are identical to those of the Core module.

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

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

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

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

JGroupsConnector

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

Since JGroups handles both discovery of nodes and the communication between them, the `JGroupsConnector` acts as both a `CommandBusConnector` and a `CommandRouter` .

> **Note**
>
> You can find the JGroups specific components for the `DistributedCommandBus` in the `axon-distributed-commandbus-jgroups` module.

The JGroupsConnector has four mandatory configuration elements:

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

> **Note**
>
> When using a Cache, it should be cleared out when the `ConsistentHash` changes to avoid potential data corruption (e.g. when commands don't specify a `@TargetAggregateVersion` and a new member quickly joins and leaves the JGroup, modifying the aggregate while it's still cached elsewhere.)

Ultimately, the JGroupsConnector needs to actually connect, in order to dispatch Messages to other segments. To do so, call the `connect()` method.

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

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

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

// on another node, with more CPU:
commandBus.subscribe(CommandType.class.getName(), handler);
commandBus.subscribe(AnotherCommandType.class.getName(), handler2);
commandBus.updateLoadFactor(150); // defaults to 100
connector.connect();

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

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

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

Spring Cloud Connector

The Spring Cloud Connector setup uses the service registration and discovery mechanism described by Spring Cloud for distributing the Command Bus. You are thus left free to choose which Spring Cloud implementation to use to distribute your commands. An example implementations is the Eureka Discovery/Eureka Server combination.

> **Note**
>
> The `SpringCloudCommandRouter` uses the Spring Cloud specific `ServiceInstance.Metadata` field to inform all the nodes in the system of its message routing information. It is thus of importance that the Spring Cloud implementation selected supports the usage of the `ServiceInstance.Metadata` field. If the desired Spring Cloud implementation does not support the modification of the `ServiceInstance.Metadata` (e.g. Consul), the `SpringCloudHttpBackupCommandRouter` is a viable solution. See the end of this chapter for configuration specifics on the `SpringCloudHttpBackupCommandRouter`.

Giving a description of every Spring Cloud implementation would push this reference guide to far. Hence we refer to their respective documentations for further information.

The Spring Cloud Connector setup is a combination of the `SpringCloudCommandRouter` and a `SpringHttpCommandBusConnector`, which respectively fill the place of the `CommandRouter` and the `CommandBusConnector` for the `DistributedCommandBus`.

> **Note**
>
> When using the `SpringCloudCommandRouter`, make sure that your Spring application is has heartbeat events enabled. The implementation leverages the heartbeat events published by a Spring Cloud application to check whether its knowledge of all the others nodes is up to date. Hence if heartbeat events are disabled the majority of the Axon applications within your cluster will not be aware of the entire set up, thus posing issues for correct command routing.

The `SpringCloudCommandRouter` has to be created by providing the following:

- A "discovery client" of type `DiscoveryClient`. This can be provided by annotating your Spring Boot application with `@EnableDiscoveryClient`, which will look for a Spring Cloud implementation on your classpath.
- A "routing strategy" of type `RoutingStrategy`. The `axon-core` module currently provides several implementations, but a function call can suffice as well. If you want to route the Commands based on the 'aggregate identifier' for example, you would use the `AnnotationRoutingStrategy` and annotate the field on the payload that identifies the aggregate with `@TargetAggregateIdentifier`.

Other optional parameters for the `SpringCloudCommandRouter` are:

- A "service instance filter" of type `Predicate<ServiceInstance>`. This predicate is used to filter out `ServiceInstances` which the `DiscoveryClient` might encounter which by forehand you know will not handle any command messages. This might be useful if you've got several services within the Spring Cloud Discovery Service set up which you do not want to take into account for command handling, ever.
- A "consistent hash change listener" of type `ConsistentHashChangeListener`. Adding a consistent hash change listener provides you the opportunity to perform a specific task if new members have been added to the known command handlers set.

The `SpringHttpCommandBusConnector` requires three parameters for creation:

- A "local command bus" of type `CommandBus`. This is the Command Bus implementation that dispatches Commands to the local JVM. These commands may have been dispatched by instances on other JVMs or from the local one.
- A `RestOperations` object to perform the posting of a Command Message to another instance.
- Lastly a "serializer" of type `Serializer`. The serializer is used to serialize the command messages before they are sent over the wire.

> **Note**
>
> The Spring Cloud Connector specific components for the `DistributedCommandBus` can be found in the `axon-distributed-commandbus-springcloud` module.

The `SpringCloudCommandRouter` and `SpringHttpCommandBusConnector` should then both be used for creating the `DistributedCommandsBus`. In Spring Java config, that would look as follows:

```
// Simple Spring Boot App providing the `DiscoveryClient` bean
@EnableDiscoveryClient
@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }

    // Example function providing a Spring Cloud Connector
    @Bean
    public CommandRouter springCloudCommandRouter(DiscoveryClient discoveryClient) {
        return new SpringCloudCommandRouter(discoveryClient, new AnnotationRoutingStra
tegy());
    }

    @Bean
    public CommandBusConnector springHttpCommandBusConnector(@Qualifier("localSegment")
 CommandBus localSegment,
                                                             RestOperations restOperat
ions,
                                                             Serializer serializer) {
        return new SpringHttpCommandBusConnector(localSegment, restOperations, seriali
zer);
    }

    @Primary // to make sure this CommandBus implementation is used for autowiring
    @Bean
    public DistributedCommandBus springCloudDistributedCommandBus(CommandRouter comman
dRouter,
                                                                  CommandBusConnector
commandBusConnector) {
        return new DistributedCommandBus(commandRouter, commandBusConnector);
    }

}
```

```
// if you don't use Spring Boot Autoconfiguration, you will need to explicitly define
the local segment:
@Bean
@Qualifier("localSegment")
public CommandBus localSegment() {
    return new SimpleCommandBus();
}
```

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

Spring Cloud Http Back Up Command Router

Internally, the `SpringCloudCommandRouter` uses the `Metadata` map contained in the Spring Cloud `ServiceInstance` to communicate the allowed message routing information throughout the distributed Axon environment. If the desired Spring Cloud implementation however does not allow the modification of the `ServiceInstance.Metadata` field (e.g. Consul), one can choose to instantiate a `SpringCloudHttpBackupCommandRouter` instead of the `SpringCloudCommandRouter`.

The `SpringCloudHttpBackupCommandRouter`, as the name suggests, has a back up mechanism if the `ServiceInstance.Metadata` field does not contained the expected message routing information. That back up mechanism is to provide an HTTP endpoint from which the message routing information can be retrieved and by simultaneously adding the functionality to query that endpoint of other known nodes in the cluster to retrieve their message routing information. As such the back up mechanism functions is a Spring Controller to receive requests at a specifiable endpoint and uses a `RestTemplate` to send request to other nodes at the specifiable endpoint.

To use the `SpringCloudHttpBackupCommandRouter` instead of the `SpringCloudCommandRouter`, add the following Spring Java configuration (which replaces the `SpringCloudCommandRouter` method in our earlier example):

```
@Configuration
public class MyApplicationConfiguration {
    @Bean
    public CommandRouter springCloudHttpBackupCommandRouter(DiscoveryClient discoveryC
lient,
                                                            RestTemplate restTemplate,

                                                            @Value("${axon.distributed
.spring-cloud.fallback-url}") String messageRoutingInformationEndpoint) {
        return new SpringCloudHttpBackupCommandRouter(discoveryClient, new AnnotationR
outingStrategy(), restTemplate, messageRoutingInformationEndpoint);
    }
}
```

Event Publishing & 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 Handlers. It is the responsibility of the Event Bus to dispatch Event Messages to all components interested. On the receiving end, Event Processors are responsible for handling those events, which includes invocation of the appropriate Event Handlers.

Publishing Events

In the vast majority of cases, the Aggregates will publish events by applying them. However, occasionally, it is necessary to publish an event (possibly from within another component), directly to the Event Bus. To publish an event, simply wrap the payload describing the event in an `EventMessage`. The `GenericEventMessage.asEventMessage(Object)` method allows you to wrap any object into an `EventMessage`. If the passed object is already an `EventMessage`, it is simply returned.

Event Bus

The `EventBus` is the mechanism that dispatches events to the subscribed event handlers. Axon provides two implementation of the Event Bus: `SimpleEventBus` and `EmbeddedEventStore`. While both implementations support subscribing and tracking processors (see Events Processors), the `EmbeddedEventStore` persists events, which allows you to replay them at a later stage. The `SimpleEventBus` has a volatile storage and 'forgets' events as soon as they have been published to subscribed components.

When using the Configuration API, the `SimpleEventBus` is used by default. To configure the `EmbeddedEventStore` instead, you need to supply an implementation of a StorageEngine, which does the actual storage of Events.

```
Configurer configurer = DefaultConfigurer.defaultConfiguration();
configurer.configureEmbeddedEventStore(c -> new InMemoryEventStorageEngine());
```

Event Processors

Event Handlers define the business logic to be performed when an Event is received. Event Processors are the components that take care of the technical aspects of that processing. They start a Unit of Work and possibly a transaction, but also ensure that correlation data can be correctly attached to all messages created during Event processing.

Event Processors come in roughly two forms: Subscribing and Tracking. The Subscribing Event Processors subscribe themselves to a source of Events and are invoked by the thread managed by the publishing mechanism. Tracking Event Processors, on the other hand, pull their messages from a source using a thread that it manages itself.

Assigning handlers to processors

All processors have a name, which identifies a processor instance across JVM instances. Two processors with the same name, can be considered as two instances of the same processor.

All Event Handlers are attached to a Processor whose name is the package name of the Event Handler's class.

For example, the following classes:

- `org.axonframework.example.eventhandling.MyHandler` ,
- `org.axonframework.example.eventhandling.MyOtherHandler` , and
- `org.axonframework.example.eventhandling.module.MyHandler`

will trigger the creation of two Processors:

- `org.axonframework.example.eventhandling` with 2 handlers, and
- `org.axonframework.example.eventhandling.module` with a single handler

The Configuration API allows you to configure other strategies for assigning classes to processors, or even assign specific instances to specific processors.

Ordering Event Handlers within a single Event Processor

To order Event Handlers within an Event Processor, the ordering in which Event Handlers are registered (as described in the Registering Event Handlers section) is guiding. Thus, the ordering in which Event Handlers will be called by an Event Processor for Event Handling is their insertion ordering in the configuration API.

If Spring is selected as the mechanism to wire everything, the ordering of the Event Handlers can be specified by adding the `@Order` annotation. This annotation should be placed on class level of your Event Handler class, adding a `integer` value to specify the ordering.

Do note that it is not possible to order Event Handlers which are not a part of the same Event Processor.

Configuring processors

Processors take care of the technical aspects of handling an event, regardless of the business logic triggered by each event. However, the way "regular" (singleton, stateless) event handlers are Configured is slightly different from Sagas, as different aspects are important for both types of handlers.

Event Handlers

By default, Axon will use Subscribing Event Processors. It is possible to change how Handlers are assigned and how processors are configured using the `EventHandlingConfiguration` class of the Configuration API.

The `EventHandlingConfiguration` class defines a number of methods that can be used to define how processors need to be configured.

- `registerEventProcessorFactory` allows you to define a default factory method that creates Event Processors for which no explicit factories have been defined.
- `registerEventProcessor(String name, EventProcessorBuilder builder)` defines the factory method to use to create a Processor with given `name`. Note that such Processor is only created if `name` is chosen as the processor for any of the available Event Handler beans.
- `registerTrackingProcessor(String name)` defines that a processor with given name should be configured as a Tracking Event Processor, using default settings. It is configured with a TransactionManager and a TokenStore, both taken from the main configuration by default.
- `registerTrackingProcessor(String name, Function<Configuration, TrackingEventProcessorConfiguration> processorConfiguration, Function<Configuration, SequencingPolicy<? super EventMessage<?>>> sequencingPolicy)` defines that a processor with given name should be configured as a Tracking Processor, and use the given `TrackingEventProcessorConfiguration` to read the configuration settings for multi-threading. The `SequencingPolicy` defines which expectations the processor has on sequential processing of events. See Parallel Processing for more details.
- `usingTrackingProcessors()` sets the default to Tracking Processors instead of Subscribing ones.

Sagas

Sagas are configured using the `SagaConfiguration` class. It provides static methods to initialize an instance either for Tracking Processing, or Subscribing.

To configure a Saga to run in subscribing mode, simply do:

```
SagaConfiguration<MySaga> sagaConfig = SagaConfiguration.subscribingSagaManager(MySaga
.class);
```

If you don't want to use the default EventBus / Store as source for this Saga to get its messages from, you can define another source of messages as well:

```
SagaConfiguration.subscribingSagaManager(MySaga.class, c -> /* define source here */);
```

Another variant of the `subscribingSagaManager()` method allows you to pass a (builder for an) `EventProcessingStrategy`. By default, Sagas are invoked in synchronously. This can be made asynchronous using this method. However, using Tracking Processors is the preferred way for asynchronous invocation.

To configure a Saga to use a Tracking Processor, simply do:

```
SagaConfiguration.trackingSagaManager(MySaga.class);
```

This will set the default properties, meaning a single Thread is used to process events. To change this:

```
SagaConfiguration.trackingSagaManager(MySaga.class)
                 // configure 4 threads
                 .configureTrackingProcessor(c -> TrackingProcessingConfiguration.forP
arallelProcessing(4))
```

The `TrackingProcessingConfiguration` has a few methods allowing you to specify how many segments will be created and which ThreadFactory should be used to create Processor threads. See Parallel Processing for more details.

Check out the API documentation (JavaDoc) of the `SagaConfiguration` class for full details on how to configure event handling for a Saga.

Token Store

Tracking Processors, unlike Subscribing ones, need a Token Store to store their progress in. Each message a Tracking Processor receives through its Event Stream is accompanied by a Token. This Token allows the processor to reopen the Stream at any later point, picking up where it left off with the last Event.

The Configuration API takes the Token Store, as well as most other components Processors need from the Global Configuration instance. If no TokenStore is explicitly defined, an `InMemoryTokenStore` is used, which is *not recommended in production*.

To configure a different Token Store, use `Configurer.registerComponent(TokenStore.class, conf -> ... create token store ...)`

Note that you can override the TokenStore to use with Tracking Processors in the respective `EventHandlingConfiguration` or `SagaConfiguration` that defines that Processor. Where possible, it is recommended to use a Token Store that stores tokens in the same database as where the Event Handlers update the view models. This way, changes to the view model can be stored atomically with the changed tokens, guaranteeing exactly once processing semantics.

Parallel Processing

As of Axon Framework 3.1, Tracking Processors can use multiple threads to process an Event Stream. They do so, by claiming a so-called segment, identifier by a number. Normally, a single thread will process a single Segment.

The number of Segments used can be defined. When a Processor starts for the first time, it can initialize a number of segments. This number defines the maximum number of threads that can process events simultaneously. Each node running of a TrackingProcessor will attempt to start its configured amount of Threads, to start processing these.

Event Handlers may have specific expectations on the ordering of events. If this is the case, the processor must ensure these events are sent to these Handlers in that specific order. Axon uses the `SequencingPolicy` for this. The `SequencingPolicy` is essentially a function, that returns a value for any given message. If the return value of the `SequencingPolicy` function is equal for two distinct event messages, it means that those messages must be processed sequentially. By default, Axon components will use the `SequentialPerAggregatePolicy` , which makes it so that Events published by the same Aggregate instance will be handled sequentially.

A Saga instance is never invoked concurrently by multiple threads. Therefore, a Sequencing Policy for a Saga is irrelevant. Axon will ensure each Saga instance receives the Events it needs to process in the order they have been published on the Event Bus.

> **Note**
>
> Note that Subscribing Processors don't manage their own threads. Therefore, it is not possible to configure how they should receive their events. Effectively, they will always work on a sequential-per-aggregate basis, as that is generally the level of concurrency in the Command Handling component.

Multi-node processing

For tracking processors, it doesn't matter whether the Threads handling the events are all running on the same node, or on different nodes hosting the same (logical) TrackingProcessor. When two instances of TrackingProcessor, having the same name, are active on different machines, they are considered two instances of the same logical processor. They will 'compete' for segments of the Event Stream. Each instance will 'claim' a segment, preventing events assigned to that segment from being processed on the other nodes.

The `TokenStore` instance will use the JVM's name (usually a combination of the host name and process ID) as the default `nodeId`. This can be overridden in `TokenStore` implementations that support multi-node processing.

Distributing Events

In some cases, it is necessary to publish events to an external system, such as a message broker.

Spring AMQP

Axon provides out-of-the-box support to transfer Events to and from an AMQP message broker, such as Rabbit MQ.

Forwarding events to an AMQP Exchange

The `SpringAMQPPublisher` forwards events to an AMQP Exchange. It is initialized with a `SubscribableMessageSource`, which is generally the `EventBus` or `EventStore`. Theoretically, this could be any source of Events that the publisher can Subscribe to.

To configure the SpringAMQPPublisher, simply define an instance as a Spring Bean. There is a number of setter methods that allow you to specify the behavior you expect, such as Transaction support, publisher acknowledgements (if supported by the broker), and the exchange name.

The default exchange name is 'Axon.EventBus'.

> **Note**
>
> Note that exchanges are not automatically created. You must still declare the Queues, Exchanges and Bindings you wish to use. Check the Spring documentation for more information.

Reading Events from an AMQP Queue

Spring has extensive support for reading messages from an AMQP Queue. However, this needs to be 'bridged' to Axon, so that these messages can be handled from Axon as if they are regular Event Messages.

The `SpringAMQPMessageSource` allows Event Processors to read messages from a Queue, instead of the Event Store or Event Bus. It acts as an adapter between Spring AMQP and the `SubscribableMessageSource` needed by these processors.

The easiest way to configure the SpringAMQPMessageSource, is by defining a bean which overrides the default `onMessage` method and annotates it with `@RabbitListener`, as follows:

```
@Bean
public SpringAMQPMessageSource myMessageSource(Serializer serializer) {
    return new SpringAMQPMessageSource(serializer) {
        @RabbitListener(queues = "myQueue")
        @Override
        public void onMessage(Message message, Channel channel) throws Exception {
            super.onMessage(message, channel);
        }
    };
}
```

Spring's `@RabbitListener` annotation tells Spring that this method needs to be invoked for each message on the given Queue ('myQueue' in the example). This method simply invokes the `super.onMessage()` method, which performs the actual publication of the Event to all the processors that have been subscribed to it.

To subscribe Processors to this MessageSource, pass the correct `SpringAMQPMessageSource` instance to the constructor of the Subscribing Processor:

```
// in an @Configuration file:
@Autowired
public void configure(EventHandlingConfiguration ehConfig, SpringAmqpMessageSource myM
essageSource) {
    ehConfig.registerSubscribingEventProcessor("myProcessor", c -> myMessageSource);
}
```

Note that Tracking Processors are not compatible with the SpringAMQPMessageSource.

Asynchronous Event Processing

The recommended approach to handle Events asynchronously is by using a Tracking Event Processor. This implementation can guarantee processing of all events, even in case of a system failure (assuming the Events have been persisted).

However, it is also possible to handle Events asynchronously in a `SubscribingProcessor`. To achieve this, the `SubscribingProcessor` must be configured with an `EventProcessingStrategy`. This strategy can be used to change how invocations of the Event Listeners should be managed.

The default strategy (`DirectEventProcessingStrategy`) invokes these handlers in the thread that delivers the Events. This allows processors to use existing transactions.

The other Axon-provided strategy is the `AsynchronousEventProcessingStrategy`. It uses an Executor to asynchronously invoke the Event Listeners.

Even though the `AsynchronousEventProcessingStrategy` executes asynchronously, it is still desirable that certain events are processed sequentially. The `SequencingPolicy` defines whether events must be handled sequentially, in parallel or a combination of both. Policies return a sequence identifier of a given event. If the policy returns an equal identifier for two events, 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.
- `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 `SequencingPolicy` interface. This interface defines a single method, `getSequenceIdentifierFor`, that returns the sequence identifier for a given event. Events for which an equal 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.

It is recommended to explicitly define an `ErrorHandler` when using the `AsynchronousEventProcessingStrategy`. The default `ErrorHandler` propagates exceptions, but in an asynchronous execution, there is nothing to propagate to, other than the Executor. This may result in Events not being processed. Instead, it is recommended to use an `ErrorHandler` that reports errors and allows processing to continue. The `ErrorHandler` is configured on the constructor of the `SubscribingEventProcessor`, where the `EventProcessingStrategy` is also provided.

Query Dispatching

Since version 3.1 Axon Framework also offers components for the Query handling. Although creating such a layer is fairly straight-forward, using Axon Framework for this part of the application has a number of benefits, such as the reuse of features such as interceptors and message monitoring.

The next sections provide an overview of the tasks related to setting up a Query dispatching infrastructure with the Axon Framework.

Query Gateway

The Query Gateway is a convenient interface towards the Query dispatching mechanism. While you are not required to use a Gateway to dispatch Queries, it is generally the easiest option to do so. Axon provides a `QueryGateway` interface and the `DefaultQueryGateway` implementation. The query gateway provides a number of methods that allow you to send a query and wait for a single or multiple results either synchronously, with a timeout or asynchronously. The query gateway needs to be configured with access to the Query Bus and a (possibly empty) list of `QueryDispatchInterceptor`s.

Query Bus

The Query Bus is the mechanism that dispatches queries to Query Handlers. Queries are registered using the combination of the query request name and query response type. It is possible to register multiple handlers for the same request-response combination, which can be used to implement for instance the scatter-gather pattern. When dispatching queries, the client must indicate whether it wants a response from a single handler or from all handlers.

If the client requests a response from a single handler, and no handler is found, a `NoHandlerForQueryException` is thrown. In case multiple handlers are registered, it is up to the implementation of the Query Bus to decide which handler is actually invoked.

If the client requests a response from all handlers, a stream of results is returned. This stream contains a result from each handler that successfully handled the query, in unspecified order. In case there are no handlers for the query, or all handlers threw an exception while handling the request, the stream is empty.

SimpleQueryBus

The `SimpleQueryBus` is the only Query Bus implementation in Axon 3.1. It does straightforward processing of queries in the thread that dispatches them. The `SimpleQueryBus` allows interceptors to be configured, see Query Interceptors for more information.

Query Interceptors

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

There are different types of interceptors: Dispatch Interceptors and Handler Interceptors. Dispatch Interceptors are invoked before a query is dispatched to a Query Handler. At that point, it may not even be sure that any handler exists for that query. Handler Interceptors are invoked just before a Query Handler is invoked.

Dispatch Interceptors

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

Structural validation

There is no point in processing a query if it does not contain all required information in the correct format. In fact, a query that lacks information should be blocked as early as possible. Therefore, an interceptor should check all incoming queries 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 queries 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 Query 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 few resources on an invalid queries as possible. Therefore, this interceptor is generally placed in the very front of the interceptor chain. In some cases, a Logging or Auditing interceptor might need to be placed in front, with the validating interceptor immediately following it.

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

Handler Interceptors

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

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

Unlike Dispatch Interceptors, Handler Interceptors are invoked in the context of the Query Handler. That means they can attach correlation data based on the Message being handled to the Unit of Work, for example. This correlation data will then be attached to messages being created in the context of that Unit of Work.

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.

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

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

Note that the Repository interface does not prescribe a `delete(identifier)` method. Deleting aggregates is done by invoking the `AggregateLifecycle.markDeleted()` method from within an aggregate. 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.

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.

Axon provides one out-of-the-box implementation for a standard Repository: the `GenericJpaRepository` . It expects the Aggregate to be a valid JPA Entity. It is configured with an `EntityManagerProvider` which provides the `EntityManager` to manage the actual persistence, and a class specifying the actual type of Aggregate stored in the Repository. You also pass in the `EventBus` to which Events are to be published when the Aggregate invokes the static `AggregateLifecycle.apply()` method.

You can also easily implement your own repository. In that case, it is best to extend from the abstract `LockingRepository`. As aggregate wrapper type, it is recommended to use the `AnnotatedAggregate`. See the sources of `GenericJpaRepository` for an example.

Event Sourcing repositories

Aggregate roots that are able to reconstruct their state based on events may also be configured to be loaded by 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.

The `EventSourcingRepository` implementation provides the basic functionality needed by any event sourcing repository in the AxonFramework. It depends on an `EventStore` (see Event store implementations), which abstracts the actual storage mechanism for the events.

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

GenericAggregateFactory

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

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

SpringPrototypeAggregateFactory

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

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.

Event store implementations

Event Sourcing repositories need an event store to store and load events from aggregates. An Event Store offers the functionality of an Event Bus, with the addition that it persists published events, and is able to retrieve events based on an Aggregate Identifier.

Axon provides an event store out of the box, the `EmbeddedEventStore`. It delegates actual storage and retrieval of events to an `EventStorageEngine`.

There are multiple `EventStorageEngine` implementations available:

`JpaEventStorageEngine`

The `JpaEventStorageEngine` stores events in a JPA-compatible data source. 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 `JpaEventStorageEngine`, you must have the JPA (`javax.persistence`) annotations on your classpath.

By default, the event store needs you to configure your persistence context (e.g. as defined in `META-INF/persistence.xml` file) to contain the classes `DomainEventEntry` and `SnapshotEventEntry` (both in the `org.axonframework.eventsourcing.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" transaction-type="RESOURCE_LOCAL"> (1)
        <class>org...eventstore.jpa.DomainEventEntry</class> (2)
        <class>org...eventstore.jpa.SnapshotEventEntry</class>
    </persistence-unit>
</persistence>
```

1. 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.
2. This line registers the `DomainEventEntry` (the class used by the `JpaEventStore`) with the persistence context.

> **Note**
>
> 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 `JpaEventStorageEngine` 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.

By default, the JPA Event Storage Engine requires an `EntityManagerProvider` implementation that returns the `EntityManager` instance for the `EventStorageEngine` to use. This also allows for application managed persistence contexts to be used. It is the `EntityManagerProvider` 's responsibility to provide a correct instance of the `EntityManager` .

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

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

```
public class MyEntityManagerProvider implements EntityManagerProvider {

    private EntityManager entityManager;

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

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

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

If that is the case, you can extend the `JpaEventStorageEngine` . It contains a number of protected methods that you can override to tweak its behavior.

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

JDBC Event Storage Engine

The JDBC event storage engine uses a JDBC Connection to store Events in a JDBC compatible data storage. Typically, these are relational databases. Theoretically, anything that has a JDBC driver could be used to back the JDBC Event Storage Engine.

Similar to its JPA counterpart, the JDBC Event Storage Engine stores Events in entries. By default, each Event is stored in a single Entry, which corresponds with a row in a table. One table is used for Events and another for the Snapshots.

The `JdbcEventStorageEngine` uses a `ConnectionProvider` to obtain connections. Typically, these connections can be obtained directly from a DataSource. However, Axon will bind these connections to a Unit of Work, so that a single connection is used in a Unit of Work. This ensures that a single transaction is used to store all events, even when multiple Units of Work are nested in the same thread.

> **Note**
>
> Spring users are recommended to use the `SpringDataSourceConnectionProvider` to attach a connection from a `DataSource` to an existing transaction.

MongoDB Event Storage Engine

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

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

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

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

The MongoDB doesn't take a lot of configuration. All it needs is a reference to the collections to store the Events in, and you're set to go. For production environments, you may want to double check the indexes on your collections.

Event Store Utilities

Axon provides a number of Event Storage Engines that may be useful in certain circumstances.

Combining multiple Event Stores into one

The `SequenceEventStorageEngine` is a wrapper around two other Event Storage Engines. When reading, it returns the events from both event storage engines. Appended events are only appended to the second event storage engine. This is useful in cases where two different implementations of Event Storage are used for performance reasons, for example. The first would be a larger, but slower event store, while the second is optimized for quick reading and writing.

Filtering Stored Events

The `FilteringEventStorageEngine` allows Events to be filtered based on a predicate. Only Events that match this predicate will be stored. Note that Event Processors that use the Event Store as a source of Events, may not receive these events, as they are not being stored.

In-Memory Event Storage

There is also an `EventStorageEngine` implementation that keeps the stored events in memory: the `InMemoryEventStorageEngine`. While it probably outperforms any other event store out there, it is not really meant for long-term production use. However, it is very useful in short-lived tools or tests that require an event store.

Influencing the serialization process

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

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

Alternatively, Axon also provides the `JacksonSerializer`, which uses Jackson to serialize Events into JSON. While it produces a more compact serialized form, it does require that classes stick to the conventions (or configuration) required by Jackson.

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

Serializing Events vs 'the rest'

Since Axon 3.1, it is possible to use a different serializer for the storage of events, than all other objects that Axon needs to serializer (such as Commands, Snapshot, Sagas, etc). While the `XStreamSerializer`'s capability to serialize virtually anything makes it a very decent default, its output isn't always a form that makes it nice to share with other applications. The `JacksonSerializer` creates much nicer output, but requires a certain structure in the objects to serialize. This structure is typically present in events, making it a very suitable event serializer.

Using the Configuration API, you can simply register an Event Serializer as follows:

```
Configurer configurer = ... // initialize
configurer.configureEventSerializer(conf -> /* create serializer here*/);
```

If no explicit `eventSerializer` is configured, Events are serialized using the main serializer that has been configured (which in turn defaults to the XStream serializer).

Event Upcasting

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

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

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

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

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

To allow an upcaster to see what version of serialized object they are receiving, the Event Store stores a revision number as well as the fully qualified name of the Event. This revision number is generated by a `RevisionResolver`, configured in the serializer. Axon provides several implementations of the `RevisionResolver`, such as the `AnnotationRevisionResolver`, which checks for an `@Revision` annotation on the Event payload, a `SerialVersionUIDRevisionResolver` that uses the `serialVersionUID` as defined by Java Serialization API and a `FixedValueRevisionResolver`, which always returns a predefined value. The latter is useful when injecting the current application version. This will allow you to see which version of the application generated a specific event.

Maven users can use the `MavenArtifactRevisionResolver` to automatically use the project version. It is initialized using the groupId and artifactId of the project to obtain the version for. Since this only works in JAR files created by Maven, the version cannot always be resolved by an IDE. If a version cannot be resolved, `null` is returned.

Axon's upcasters do not work with the `EventMessage` directly, but with an `IntermediateEventRepresentation`. The `IntermediateEventRepresentation` provides functionality to retrieve all necessary fields to construct an `EventMessage` (and thus a upcasted `EventMessage` too), together with the actual *upcast* functions. These upcast functions by default only allow the adjustment of the events payload, payload type and additions to the event its metadata. The actual representation of the events in the upcast function may vary based on the event serializer used or the desired form to work with, so the upcast function of the `IntermediateEventRepresentation` allows the selection of the expected representation type. The other fields, for example the message/aggregate identifier, aggregate type, timestamp etc. are not adjustable by the `IntermediateEventRepresentation`. Adjusting those fields is not the intended work for an Upcaster, hence those options are not provided by the provided `IntermediateEventRepresentation` implementations.

The basic `Upcaster` interface for events in the Axon Framework works on a `Stream` of `IntermediateEventRepresentations` and returns a `Stream` of `IntermediateEventRepresentations`. The upcasting process thus does not directly return the end result of the introduced upcast functions, but chains every upcasting function from one revision to another together by stacking `IntermediateEventRepresentations`. Once this process has taken place and the end result is pulled from them, that is when the actual upcasting function is performed on the serialized event.

Provided abstract Upcaster implementations

As described earlier, the `Upcaster` interface does not upcast a single event; it requires a `Stream<IntermediateEventRepresentation>` and returns one. However, an Upcaster is usually written to adjust a single event out of this stream. More elaborate upcasting set ups are also imaginable, for example from one events to multiple, or an upcaster which pulls state from an earlier event and pushes it in a later one. This section describes the currently provided abstract implementations of Event Upcasters which a user can extend to add its own desired upcast functionality.

- `SingleEventUpcaster` - This is a one to one implementation of an event Upcaster. Extending from this implementation requires one to implement a `canUpcast` and `doUpcast` function, which respectively check whether the event at hand is to be upcasted, and if so how it should be upcasted. This is most likely the implementation to extend from, as most event adjustments are based on self contained data and are one to one.
- `EventMultiUpcaster` - This is a one to many implementation of an event Upcaster. It is mostly identical to a `SingleEventUpcaster`, with the exception that the `doUpcast` function returns a `Stream` instead of a single `IntermediateEventRepresentation`. As

such this upcaster allows you to revert a single event to several events. This might be useful if you for example have figured out you want more fine grained events from a *fat* event.

- `ContextAwareSingleEventUpcaster` - This is a one to one implementation of an Upcaster, which can store context of events during the process. Next to the `canUpcast` and `doUpcast`, the context aware Upcaster requires one to implement a `buildContext` function, which is used to instantiate a context which is carried between events going through the upcaster. The `canUpcast` and `doUpcast` functions receive the context as a second parameter, next to the `IntermediateEventRepresentation`. The context can then be used within the upcasting process to pull fields from earlier events and populate other events. It thus allows you to move a field from one event to a completely different event.
- `ContextAwareEventMultiUpcaster` - This is a one to many implementation of an Upcaster, which can store context of events during the process. This abstract implementation is a combination of the `EventMultiUpcaster` and `ContextAwareSingleEventUpcaster`, and thus services the goal of keeping context of `IntermediateEventRepresentations` and upcasting one such representation to several. This implementation is useful if you not only want to copy a field from one event to another, but have the requirement to generate several new events in the process.

Writing an upcaster

The following Java snippets will serve as a basic example of a one to one Upcaster (the `SingleEventUpcaster`).

Old version of the event:

```
@Revision("1.0")
public class ComplaintEvent {
    private String id;
    private String companyName;

    // Constructor, getter, setter...
}
```

New version of the event:

```
@Revision("2.0")
public class ComplaintEvent {
    private String id;
    private String companyName;
    private String description; // New field

    // Constructor, getter, setter...
}
```

Upcaster:

```
// Upcaster from 1.0 revision to 2.0 revision
public class ComplaintEventUpcaster extends SingleEventUpcaster {
    private static SimpleSerializedType targetType = new SimpleSerializedType(Complain
Event.class.getTypeName(), "1.0");

    @Override
    protected boolean canUpcast(IntermediateEventRepresentation intermediateRepresenta
tion) {
        return intermediateRepresentation.getType().equals(targetType);
    }

    @Override
    protected IntermediateEventRepresentation doUpcast(IntermediateEventRepresentation
 intermediateRepresentation) {
        return intermediateRepresentation.upcastPayload(
                new SimpleSerializedType(targetType.getName(), "2.0"),
                org.dom4j.Document.class,
                document -> {
                    document.getRootElement()
                            .addElement("description")
                            .setText("no complaint description"); // Default value
                    return document;
                }
        );
    }
}
```

Spring boot configuration:

```
@Configuration
public class AxonConfiguration {

    @Bean
    public SingleEventUpcaster myUpcaster() {
        return new ComplaintEventUpcaster();
    }

    @Bean
    public JpaEventStorageEngine eventStorageEngine(Serializer serializer,
                                                    DataSource dataSource,
                                                    SingleEventUpcaster myUpcaster,
                                                    EntityManagerProvider entityManage
rProvider,
                                                    TransactionManager transactionMana
ger) throws SQLException {
        return new JpaEventStorageEngine(serializer,
                myUpcaster::upcast,
                dataSource,
                entityManagerProvider,
                transactionManager);
    }
}
```

Content type conversion

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

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

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

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

The `XStreamSerializer` supports Dom4J as well as XOM as XML document representations. The `JacksonSerializer` supports Jackson's `JsonNode`.

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 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.

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 definition of when snapshots should be created, is provided by the `SnapshotTriggerDefinition` interface.

The `EventCountSnapshotTriggerDefinition` 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;

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

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

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

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

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

The `AggregateSnapshotter` provides one more property:

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

> **Note**
>
> If you use an executor that executes snapshot creation in another thread, make sure you configure the correct transaction management for your underlying event store, if necessary.
>
> Spring users can use the `SpringAggregateSnapshotter`, which will automatically look up the right `AggregateFactory` from the Application Context when a snapshot needs to be created.

Storing Snapshot Events

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

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.

Initializing an Aggregate based on a Snapshot Event

A snapshot event is an event like any other. That means a snapshot event is handled just like any other domain event. When using annotations to demarcate event 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.

```java
public class MyAggregate extends AbstractAnnotatedAggregateRoot {

    // ... code omitted for brevity

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

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

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

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.

To detect conflict, pass a parameter of type `ConflictResolver` to the `@CommandHandler` method of your aggregate. This interface provides `detectConflicts` methods that allow you to define the types of events that are considered a conflict when executing that specific type of command.

> **Note**
>
> Note that the `ConflictResolver` will only contain any potentially conflicting events if the Aggregate was loaded with an expected version. Use `@TargetAggregateVersion` on a field of a command to indicate the expected version of the Aggregate.

If events matching the predicate are found, an exception is thrown (the optional second parameter of `detectConflicts` allows you to define the exception to throw). If none are found, processing continues as normal.

If no invocations to `detectConflicts` are made, and there are potentially conflicting events, the `@CommandHandler` will fail. This may be the case when an expected version is provided, but no `ConflictResolver` is available in the parameters of the `@CommandHandler` method.

Spring Boot AutoConfiguration

Axon's support for Spring Boot AutoConfiguration is by far the easiest option to get started configuring your Axon infrastructure components. By simply adding the `axon-spring-boot-starter` dependency, Axon will automatically configure the basic infrastructure components (Command Bus, Event Bus), as well as any component required to run and store Aggregates and Sagas.

Depending on other components available in the application context, Axon will define certain components, if they aren't already explicitly defined in the application context. This means that you only need to configure components that you want different from the default.

Event Bus and Event Store Configuration

If JPA is available, an Event Store with a JPA Event Storage Engine is used by default. This allow storage of Aggregates using Event Sourcing without any explicit configuration.

If JPA is not available, Axon defaults to a `SimpleEventBus`, which means that you need to specify a non-event sourcing repository for each Aggregate, or configure an `EventStorageEngine` in your Spring Configuration.

To configure a different Event Storage Engine, even if JPA is on the class path, simply define a bean of type `EventStorageEngine` (to use Event Sourcing) or `EventBus` (if Event Sourcing isn't required).

Command Bus Configuration

Axon will configure a `SimpleCommandBus` if no `CommandBus` implementation is explicitly defined in the Application Context. This `CommandBus` will use the `TransactionManager` to manage transactions.

If the only `CommandBus` bean defined is a `DistributedCommandBus` implementation, Axon will still configure a CommandBus implementation to serve as the local segment for the DistributedCommandBus. This bean will get a Qualifier "localSegment". It is recommended to define the `DistributedCommandBus` as a `@Primary`, so that it gets priority for dependency injection.

Query Bus Configuration

Axon will configure a `SimpleQueryBus` if no `QueryBus` implementation is explicitly defined in the Application Context. This `QueryBus` will use the `TransactionManager` to manage transactions.

Transaction Manager Configuration

If no `TransactionManager` implementation is explicitly defined in the Application Content, Axon will look for the Spring `PlatformTransactionManager` bean and wrap that in a `TransactionManager`. If the Spring bean is not available, the `NoOpTransactionManager` will be used.

Serializer Configuration

By default, Axon uses an XStream based serializer to serialize objects. This can be changed by defining a bean of type `Serializer` in the application context.

While the default Serializer provides an arguably ugly xml based format, it is capable of serializing and deserializing virtually anything, making it a very sensible default. However, for events, which needs to be stored for a long time and perhaps shared across application boundaries, it is desirable to customize the format.

You can define a separate Serializer to be used to serialize events, by assigning it the `eventSerializer` qualifier. Axon will consider a bean with this qualifier to be the event serializer. If no other bean is defined, Axon will use the default serializer for all other objects to serialize.

Example:

```
@Qualifier("eventSerializer")
@Bean
public Serializer eventSerializer() {
    return new JacksonSerializer();
}
```

When overriding both the default serializer and defining an event serializer, we must instruct Spring that the default serializer is, well, the default:

```java
@Primary // marking it primary means this is the one to use, if no specific Qualifier
is requested
@Bean
public Serializer serializer() {
    return new MyCustomSerializer();
}

@Qualifier("eventSerializer")
@Bean
public Serializer eventSerializer() {
    return new JacksonSerializer();
}
```

Aggregate Configuration

The `@Aggregate` annotation (in package `org.axonframework.spring.stereotype`) triggers AutoConfiguration to set up the necessary components to use the annotated type as an Aggregate. Note that only the Aggregate Root needs to be annotated.

Axon will automatically register all the `@CommandHandler` annotated methods with the Command Bus and set up a repository if none is present.

To set up a different repository than the default, define one in the application context. Optionally, you may define the name of the repository to use, using the `repository` attribute on `@Aggregate`. If no `repository` attribute is defined, Axon will attempt to use the repository with the name of the aggregate (first character lowercase), suffixed with `Repository`. So on a class of type `MyAggregate`, the default Repository name is `myAggregateRepository`. If no bean with that name is found, Axon will define an `EventSourcingRepository` (which fails if no `EventStore` is available).

Saga Configuration

The configuration of infrastructure components to operate Sagas is triggered by the `@Saga` annotation (in package `org.axonframework.spring.stereotype`). Axon will configure a `SagaManager` and `SagaRepository`. The SagaRepository will use a `SagaStore` available in the context (defaulting to `JPASagaStore` if JPA is found) for the actual storage of Sagas.

To use different `SagaStore`s for Sagas, provide the bean name of the `SagaStore` to use in the `sagaStore` attribute of each `@Saga` annotation.

Sagas will have resources injected from the application context. Note that this doesn't mean Spring-injecting is used to inject these resources. The `@Autowired` and `@javax.inject.Inject` annotation can be used to demarcate dependencies, but they are injected by Axon by looking for these annotations on Fields and Methods. Constructor injection is not (yet) supported.

To tune the configuration of Sagas, it is possible to define a custom SagaConfiguration bean. For an annotated Saga class, Axon will attempt to find a configuration for that Saga. It does so by checking for a bean of type `SagaConfiguration` with a specific name. For a Saga class called `MySaga`, the bean that Axon looks for is `mySagaConfiguration`. If no such bean is found, it creates a Configuration based on available components.

If a `SagaConfiguration` instance is present for an annotated Saga, that configuration is used to retrieve and register the components for this type of Saga. If the SagaConfiguration bean is not named as described above, it is possible that the Saga is registered twice, and receives events in duplicate. To prevent this, you can specify the bean name of the `SagaConfiguration` using the @Saga annotation:

```
@Saga(configurationBean = "mySagaConfigBean")
public class MySaga {
    // methods here
}

// in the Spring configuration:
@Bean
public SagaConfiguration<MySaga> mySagaConfigBean() {
    // create and return SagaConfiguration instance
}
```

Event Handling Configuration

By default, all singleton Spring beans components containing `@EventHandler` annotated methods will be subscribed to an Event Processor to receive Event Messages published to the Event Bus.

The `EventHandlingConfiguration` bean, available in the Application Context, has methods to tweak the configuration of the Event Handlers. See Configuration API for details on configuring Event Handlers and Event Processors.

To update the Event Handling Configuration, create an autowired method that set the configuration you desire:

```
@Autowired
public void configure(EventHandlingConfiguration config) {
    config.usingTrackingProcessors(); // default all processors to tracking mode.
}
```

Certain aspect of Event Processors can also be configured in `application.properties`.

```
axon.eventhandling.processors.name.mode=tracking
axon.eventhandling.processors.name.source=eventBus
```

If the name of a processor contains periods `.`, use the map notation:

```
axon.eventhandling.processors["name"].mode=tracking
axon.eventhandling.processors["name"].source=eventBus
```

Or using application.yml:

```
axon:
    eventhandling:
        processors:
            name:
                mode: tracking
                source: eventBus
```

The source attribute refers to the name of a bean implementing `SubscribableMessageSource` or `StreamableMessageSource` that should be used as the source of events for the mentioned processor. The source default to the Event Bus or Event Store defined in the application context.

Query Handling Configuration

All singleton Spring beans are scanned for methods that have the `@QueryHandler` annotation. For each method that is found, a new query handler is registered with the query bus.

Parallel processing

Tracking Processors can use multiple threads to process events in parallel. Not all threads need to run on the same node.

One can configure the number of threads (on this instance) as well as the initial number of segments that a processor should define, if non are yet available.

```
axon.eventhandling.processors.name.mode=tracking
# Sets the number of maximum number threads to start on this node
axon.eventhandling.processors.name.threadCount=2
# Sets the initial number of segments (i.e. defines the maximum number of overall thre
ads)
axon.eventhandling.processors.name.initialSegmentCount=4
```

Enabling AMQP

To enable AMQP support, ensure that the `axon-amqp` module is on the classpath and an AMQP `ConnectionFactory` is available in the application context (e.g. by including the `spring-boot-starter-amqp`).

To forward Events generated in the application to an AMQP Channel, a single line of `application.properties` configuration is sufficient:

```
axon.amqp.exchange=ExchangeName
```

This will automatically send all published events to the AMQP Exchange with the given name.

By default, no AMQP transactions are used when sending. This can be overridden using the `axon.amqp.transaction-mode` property, and setting it to `transactional` or `publisher-ack` .

To receive events from a queue and process them inside an Axon application, you need to configure a `SpringAMQPMessageSource` :

```
@Bean
public SpringAMQPMessageSource myQueueMessageSource(AMQPMessageConverter messageConver
ter) {
    return new SpringAMQPMessageSource(messageConverter) {

        @RabbitListener(queues = "myQueue")
        @Override
        public void onMessage(Message message, Channel channel) throws Exception {
            super.onMessage(message, channel);
        }
    };
}
```

and then configure a processor to use this bean as the source for its messages:

```
axon.eventhandling.processors.name.source=myQueueMessageSource
```

Distributing commands

Configuring a distributed command bus can (mostly) be done without any modifications in Configuration files.

First of all, the starters for one of the Axon Distributed Command Bus modules needs to be included (e.g. JGroups or SpringCloud).

Once that is present, a single property needs to be added to the application context, to enable the distributed command bus:

```
axon.distributed.enabled=true
```

There in one setting that is independent of the type of connector used:

```
axon.distributed.load-factor=100
```

Axon will automatically configure a DistributedCommandBus when a `CommandRouter` as well as a `CommandBusConnector` are present in the application context. In such case, specifying `axon.distributed.enabled` isn't even necessary. The latter merely enables autoconfiguration of these routers and connectors.

Using JGroups

This module uses JGroups to detect and communicate with other nodes. The AutoConfiguration will set up the JGroupsConnector using default settings, that may need to be adapted to suit your environment.

By default, the JGroupsConnector will attempt to locate a GossipRouter on the localhost, port 12001.

The settings for the JGroups connector are all prefixed with `axon.distributed.jgroups`.

```
# the address to bind this instance to. By default, attempts to find the Global IP add
ress
axon.distributed.jgroups.bind-addr=GLOBAL
# the port to bind the local instance to
axon.distributed.jgroups.bind-port=7800

# the name of the JGroups Cluster to connect to
axon.distributed.jgroups.cluster-name=Axon

# the JGroups Configuration file to configure JGroups with
axon.distributed.jgroups.configuration-file=default_tcp_gossip.xml

# The IP and port of the Gossip Servers (comma separated) to connect to
axon.distributed.jgroups.gossip.hosts=localhost[12001]
# when true, will start an embedded Gossip Server on bound to the port of the first me
ntioned gossip host.
axon.distributed.jgroups.gossip.auto-start=false
```

The JGroups Configuration file can be use for much more fine-grained control of the Connector's behavior. Check out JGroups's Reference Guide for more information.

Using Spring Cloud

Spring Cloud comes with nice abstractions on top of discovery. Axon can use these abstractions to report its availability and find other Command Bus nodes. For communication with these nodes, Axon uses Spring HTTP, by default.

The Spring Cloud AutoConfiguration doesn't have much to configure. It uses an existing Spring Cloud Discovery Client (so make sure `@EnableDiscoveryClient` is used and the necessary client is on the classpath).

However, some discovery clients aren't able to update instance metadata dynamically on the server. If Axon detects this, it will automatically fall back to querying that node using HTTP. This is done once on each discovery heartbeat (usually 30 seconds).

This behavior can be configured or disabled, using the following settings in `appplication.properties` :

```
# whether to fall back to http when no meta-data is available
axon.distributed.spring-cloud.fallback-to-http-get=true
# the URL on which to publish local data and retrieve from other nodes.
axon.distributed.spring-cloud.fallback-url=/message-routing-information
```

For more fine-grained control, provide a `SpringCloudHttpBackupCommandRouter` or `SpringCloudCommandRouter` bean in your application context.

Advanced Customizations

Parameter Resolvers

You can configure additional `ParameterResolver`s by extending the `ParameterResolverFactory` class and creating a file named `/META-INF/service/org.axonframework.common.annotation.ParameterResolverFactory` containing the fully qualified name of the implementing class.

> **Caution**
>
> At this moment, OSGi support is limited to the fact that the required headers are mentioned in the manifest file. The automatic detection of `ParameterResolverFactory` instances works in OSGi, but due to classloader limitations, it might be necessary to copy the contents of the `/META-INF/service/org.axonframework.common.annotation.ParameterResolverFactory` file to the OSGi bundle containing the classes to resolve parameters for (i.e. the event handler).

Meta Annotations

TODO

Customizing Message Handler behavior

TODO

Performance Tuning

TODO: Update to Axon 3

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.

Database Indexes and Column Types

SQL Databases

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

- Normal operational use (storing and loading events):

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

  Table 'DomainEventEntry', `eventIdentifier` (unique index)

- Snapshotting:

  Table 'SnapshotEventEntry', `aggregateIdentifier` column.

  Table 'SnapshotEventEntry', `eventIdentifier` (unique index)

- Sagas

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

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

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

> **Warning**
>
> It is highly recommended to store all timestamps in UTC format. In countries with Daylight Savings Time, storing timestamps in local time may result in sorting errors for events generated around and during the timezone switch. This does not occur when UTC is used. Some servers are configured to always use UTC. Alternatively, you should configure the Event Store to convert timestamps to UTC before storing them.

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

MongoDB

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

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

- Normal operational use:

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

- Snapshotting:

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

- Replaying events:

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

- Sagas:

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

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

Caching

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

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

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

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

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

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

- Configure a sensible time to live / time to idle

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

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. Therefore, 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 Using Snapshot Events for more about snapshotting.

Event Serializer tuning

XStream Serializer

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

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

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

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

Preventing duplicate serialization

Especially in distributed systems, Event Messages need to be serialized on multiple occasions. Axon's components are aware of this and have support for `SerializationAware` messages. If a `SerializationAware` message is detected, its methods are used to serialize an object, instead of simply passing the payload to a serializer. This allows for performance optimizations.

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

Different serializer for Events

When using Event Sourcing, Serialized events can stick around for a long time. Therefore, consider the format to which they are serializer carefully. Consider configuring a separate serializer for events, carefully optimizing for the way they are stored. The JSON format generated by Jackson is generally more suitable for the long term than XStream's XML format.

Custom Identifier generation

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