Table of Contents

Implementing Domain Logic

> To communicate effectively, the code must be based on the same language used to write the requirements—the same language that the developers speak with each other and with domain experts.
>
> — Eric Evans

In the Command Query Responsibility Segregation (CQRS) section, we describe how DDD and CQRS complement each other and how the command side (write requests) is the home of business logic. In this chapter, we will implement the command side API for the LC application using Spring Boot and Axon Framework, JSR-303 Bean Validations and persistence options by contrasting between state-stored vs event-sourced aggregates. The list of topics to be covered is as follows:

- Identifying aggregates
- Handling commands and emitting events

- Test-driving the application
- Persisting aggregates
- Performing validations

By the end of this chapter, you would have learnt how to implement the core of your system (the domain logic) in a robust, well encapsulated manner. You will also learn how to decouple your domain model from persistence concerns. Finally, you will be able to appreciate how to perform DDD's tactical design using services, repositories, aggregates, entities and value objects.

Technical Requirements

- JDK 1.8+ (We have used Java 16 to compile sample sources)
- Maven 3.x
- Spring Boot 2.4.x
- JUnit 5.7.x (Included with spring boot)
- Axon Framework 4.4.7 (DDD and CQRS Framework)
- Project Lombok (To reduce verbosity)
- Moneta 1.4.x (Money and currency reference implementation - JSR 354)

Command Query Responsibility Segregation (CQRS)

In this chapter, we will make use of the Command Query Responsibility Segregation (CQRS) architecture pattern to express the domain logic for our solution. The CQRS pattern strictly separates **write** (those that mutate state) and **read** (those that answer questions) operations.

Recap: What is CQRS

In traditional applications, a single domain, data/persistence model is used to handle all kinds of operations. With CQRS, we create distinct models to handle updates and enquiries. This is depicted in the following diagram:

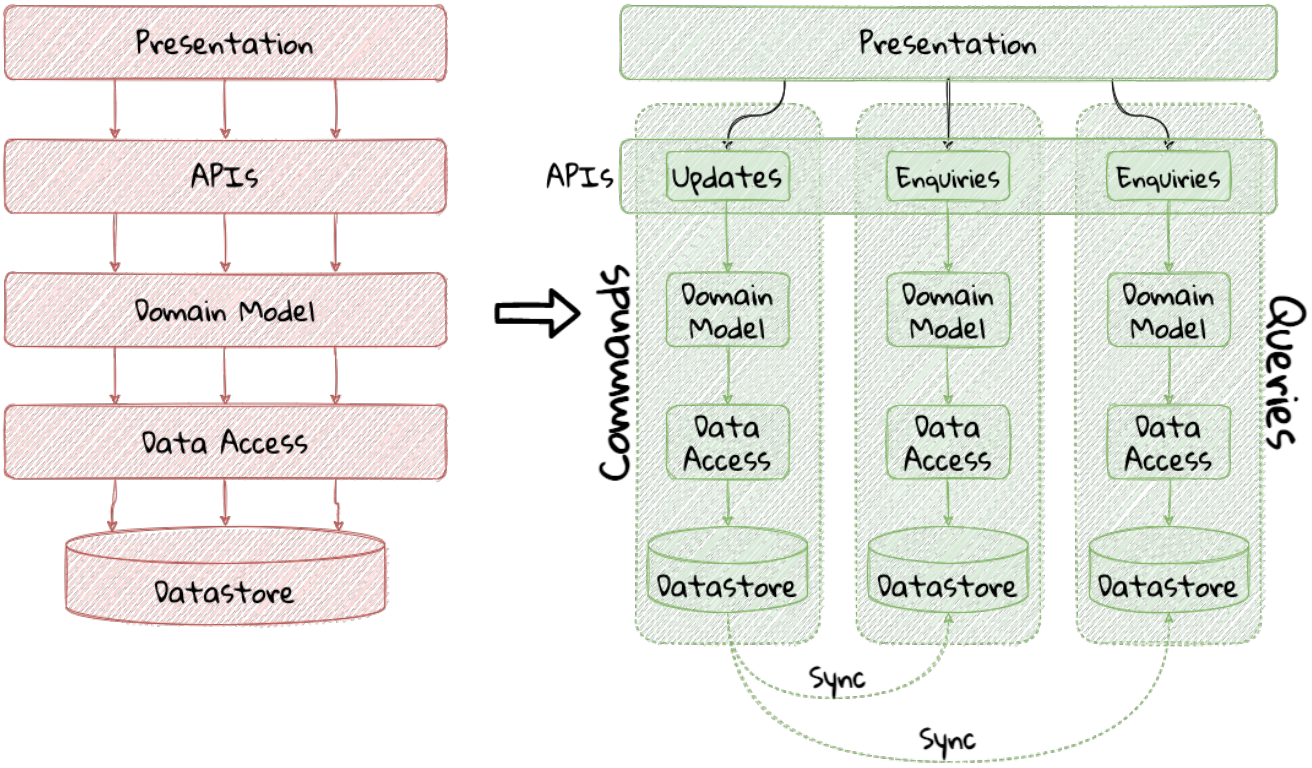


Figure 1. Traditional vs. CQRS Architecture

We depict multiple read models above because it is possible (but not necessary) to create more than one read model, depending on the kinds of query use cases that need to be supported.

For this to work predictably, the read model(s) need to be kept in sync with the write models (we will examine some of the techniques to do that in detail later.

Why CQRS?

The traditional, single-model approach works well for simple, CRUD-style applications, but starts to become unwieldy for more complex scenarios. We discuss some of these scenarios below:

- **Volume imbalance between read and writes**: In most systems, read operations often outnumber write operations by significant orders of magnitude. For example, consider the number of times a trader checks stock prices vs. the number of times they actually transact (buy or sell stock trades). It is also usually true that write operations are the ones that make businesses money. Having a single model for both reads and writes in a system with a majority of read operations can overwhelm a system to an extent where write performance can start getting affected.
- **Need for multiple read representations**: When working with relatively complex systems, it is not uncommon to require more than one representation of the same data. For example, when looking at personal health data, one may want to look at a daily, weekly, monthly view. While these views can be computed on the fly from the *raw* data, each transformation (aggregation, summarization, etc.) adds to the cognitive load on the system. Several times, it is not possible to predict ahead of time, the nature of these requirements. By extension, it is not feasible to design a single canonical model that can provide answers to all these requirements. Creating domain models specifically designed to meet a focused set of requirements can be much easier.

- **Different security requirements**: Managing authorization and access requirements to data/APIs when working a single model can start to become cumbersome. For example, higher levels of security may be desirable for debit operations in comparison to balance enquiries. Having distinct models can considerably ease the complexity in designing fine-grained authorization controls.
- **More uniform distribution of complexity**: Having a single model to serve all use cases means that they can now be focused towards solving a single concern and thereby reduce complexity. It is worth noting that the essence of domain-driven design is mainly to work effectively with complex software systems and CQRS fits well with this line of thinking.

Implementing CQRS does not require the use of any framework. However, in this book we will look at using Axon Framework because in our opinion it provides a set of conveniences to do so while staying out of the way. There are other frameworks that work comparably, like Lagom Framework[1] and Eventuate[2] that are worth exploring as well.

Bootstrapping the application

To get started, let's create a simple spring boot application using the following command:

```
curl -G https://start.spring.io/starter.zip \ ①
       -d dependencies=web,data-jpa,lombok,validation,h2,actuator \ ②
       -d name=lc-issuance-api \
       -d artifactId=lc-issuance-api \
       -d groupId=com.example.api \
       -d packaging=jar \
       -d description='LC Issuance API' \
       -d package-name=com.example.api \
       -o lc-issuance-api.zip ③
```

① The spring initializr to create the application archive in zip form

② The list of dependencies separated by a comma

③ The name of the archive containing the generated sources

Alternatively, you can use use the spring initializr directly at https://start.spring.io or the spring boot CLI to generate the application.

This should create a file named `lc-issuance-api.zip` in the current directory. Unzip this file to a location of your choice and add a dependency on the Axon framework in the `dependencies` section of the `pom.xml` file:

```
    <dependency>
        <groupId>org.axonframework</groupId>
        <artifactId>axon-spring-boot-starter</artifactId>
        <version>${axon-framework.version}</version> ①
    </dependency>
```

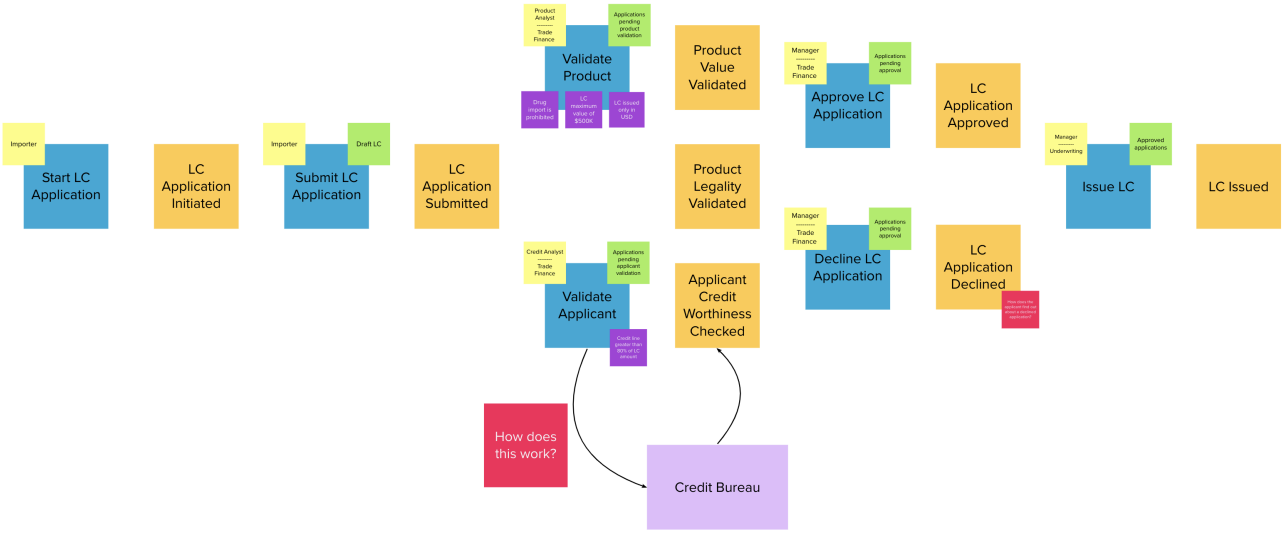
① You may need to change the version

Also, add the following dependency on the `axon-test` library to enable unit testing of aggregates:

```
    <dependency>
        <groupId>org.axonframework</groupId>
        <artifactId>axon-test</artifactId>
        <scope>test</scope>
        <version>${axon-framework.version}</version>
    </dependency>
```

With the above set up, you should be able to run the application and start implementing the LC issuance functionality.

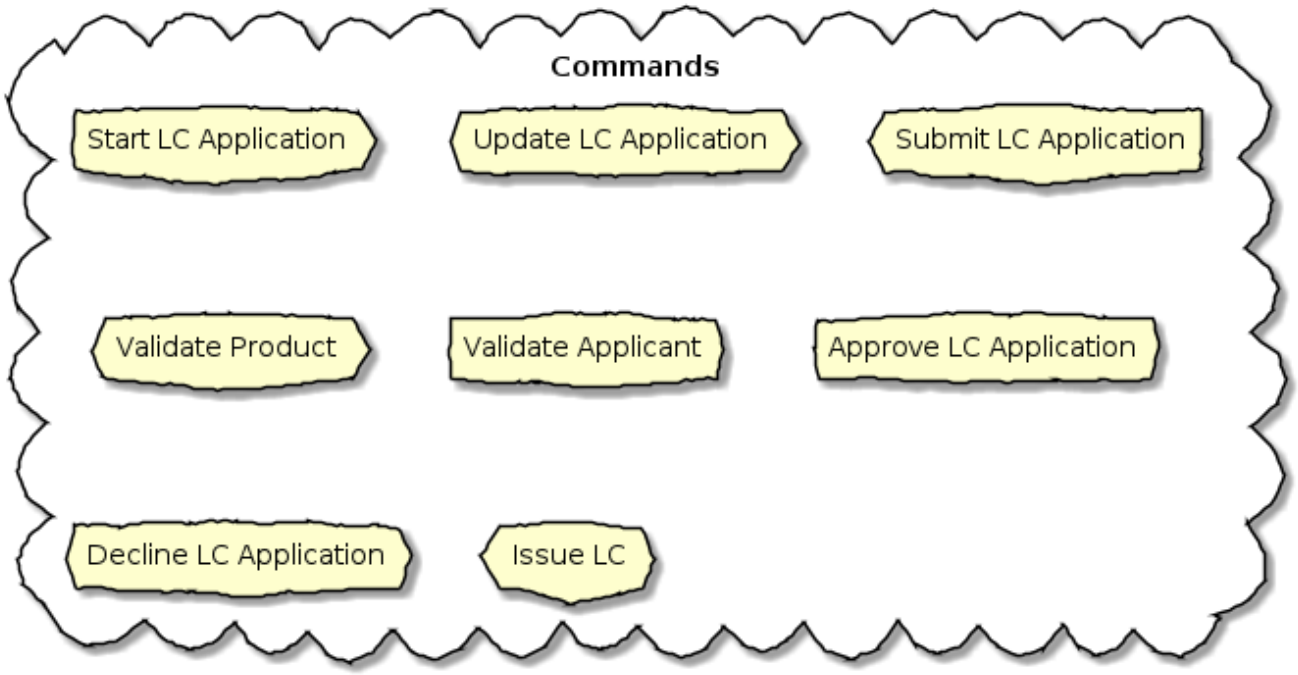
As a reminder, this is the output produced from our eventstorming session:



The blue stickies in this diagram represent commands. Let's look at how to implement these commands using the Axon framework.

Identifying Commands

From the eventstorming session, we have the following commands to start with:



Commands are always directed to an aggregate for processing (handling). This means that we need to resolve each of these commands to be handled by an aggregate. While the sender of the command does not care which component within the system handles it, we need to decide which aggregate will handle each command. It is also important to note that any given command can only be handled by a single aggregate within the system. Let's look at how to group these commands and assign them to aggregates. To be able to do that, we need to identify the aggregates in the system first.

Identifying Aggregates

Looking at the output of the eventstorming session, one potential grouping can be as follows:

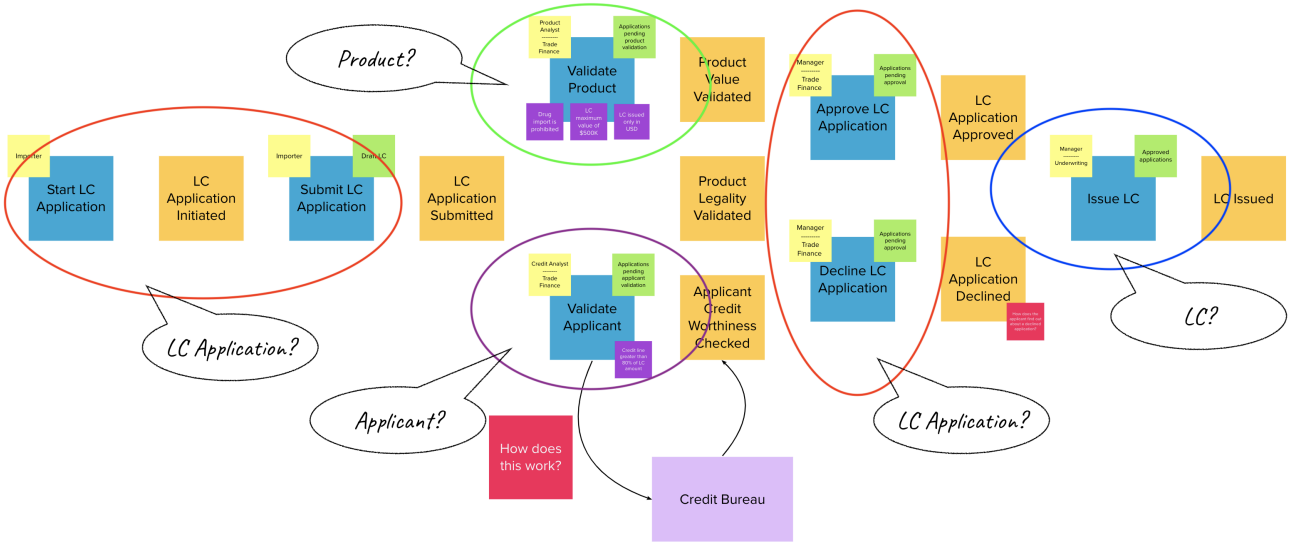


Figure 2. First cut attempt at aggregate design

At first glance, it appears that we have four potential entities to handle these commands:

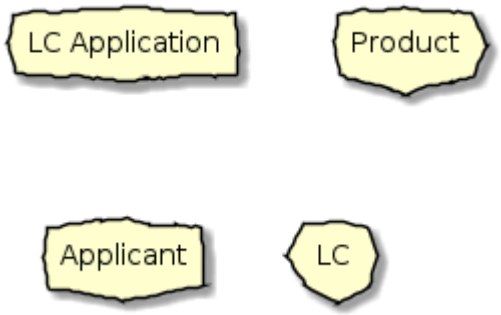


Figure 3. Potential aggregates at first glance

At first glance, each of these entities may be classified as aggregates in our solution. Here, the `LC Application` feels like a reasonably good choice for aggregate, given that we are building a solution to manage LC applications. However, do the others make sense to be classified as such? The `Product` and `Applicant` look like potential entities, but we need to ask ourselves if we will need to operate on these outside of the purview of the `LC Application`. If the answer is a **yes**, then `Product` and `Applicant` *may* be classified as aggregates. But both `Product` and `Applicant` do not seem to require being operated on without an enveloping `LC Application` within this bounded context. It feels that way because both product and applicant details are required to be provided as part of the LC application process. At least from what we know of the process thus far, this seems to be true. This means we are left with two potential aggregates — `LC` and `LC Application`.

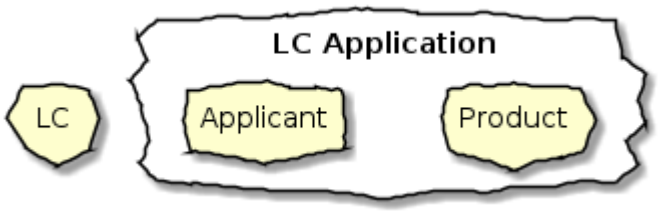


Figure 4. Slightly more refined aggregate structure

When we look at the output of our eventstorming session, the `LC Application` transitions to become an `LC` much later in the lifecycle. Let's work on the `LC Application` right now, and suspend further analysis on the need for the `LC` aggregate to a later time.

For a more detailed explanation of the differences between aggregates, aggregate roots, entities and value objects, refer to Chapter 2.

Let's start writing our first command to see how this manifests itself in code.

Test-driving the system

While we have a reasonably good conceptual understanding of the system, we are still in the process of refining this understanding. Test-driving the system allows us to exercise our understanding by acting as the first client of the solution that we are producing.

This is very well illustrated in the best-selling book — *Growing Object-Oriented Software, Guided by Tests* by authors Nat Price and Steve Freeman. This is worth looking at, to gain a deeper understanding of this practice.

So let's start with the first test. To the external world, an event-driven system typically works in a manner depicted below:

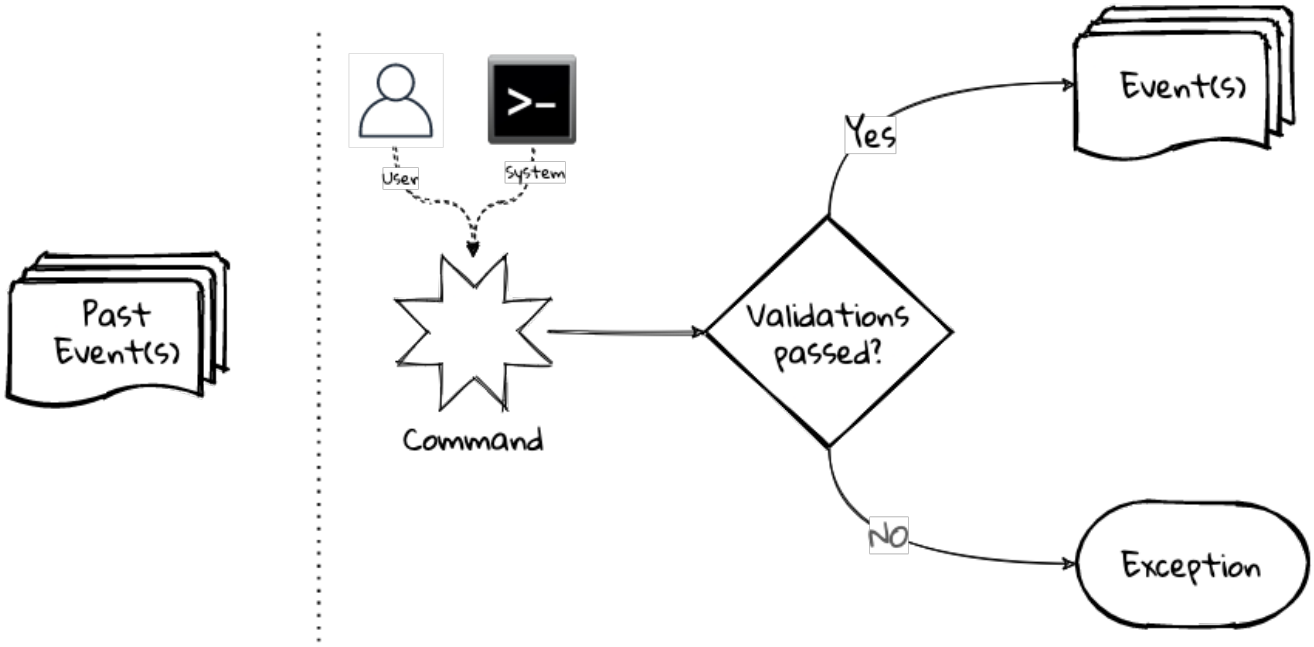


Figure 5. An event-driven system

An optional set of events may have occurred in the past. A command is received by the system (initiated manually by a user or automatically by a part of the system), which acts as a stimulus. The system will react in one of two ways when it handles a command:

- Emit one or more events
- Throw an exception

The Axon framework allows us to express tests in this form. This is outlined below:

```
public class LCApplicationAggregateTests {

    private FixtureConfiguration<LCApplication> fixture;                          ①

    @BeforeEach
    void setUp() {
        fixture = new AggregateTestFixture<>(LCApplication.class);                ②
    }

    @Test
    void shouldPublishLCApplicationCreated() {
        fixture.given()                                                           ③

                .when(new CreateLCApplicationCommand())                           ④

                .expectEventsMatching(exactSequenceOf(                            ⑤
                        messageWithPayload(any(LCApplicationCreatedEvent.class)), ⑥
                        andNoMore()                                               ⑦
                ));
    }
}
```

① `FixtureConfiguration` is an Axon framework utility to aid testing of aggregate behaviour using a BDD style given-when-then syntax.

② `AggregateTestFixture` is a concrete implementation of `FixtureConfiguration` where you need to register your aggregate class — `LCApplication` in our case as the candidate to handle commands directed to our solution.

③ Since this is the start of the business process, there are no events that have occurred thus far. This is signified by the fact that we do not pass any arguments to the `given` method. In other examples we will discuss later, there will likely be events that have already occurred prior to receiving this command.

④ This is where we instantiate a new instance of the command object. Command objects are usually similar to data transfer objects, carrying a set of information. This command will be routed to our aggregate for handling. We will take a look at how this works in detail shortly.

⑤ Here we are declaring that we expect events matching an exact sequence.

⑥ Here we are expecting an event of type `LCApplicationCreated` to be emitted as a result of successfully handling the command.

⑦ We are finally saying that we do not expect any more events — which means that we expect exactly one event to be emitted.

Implementing the command

The `CreateLCApplicationCommand` in the previous simplistic example does not carry any state. Realistically, the command will likely look something like what is depicted as follows:

```
import lombok.Data;

@Data
public class CreateLCApplicationCommand {  ①

    private LCApplicationId id;            ②
    private ClientId clientId;
    private Party applicant;               ③
    private Party beneficiary;
    private AdvisingBank advisingBank;     ③
    private LocalDate issueDate;
    private MonetaryAmount amount;
    private String merchandiseDescription;

}
```

① The command class. When naming commands, we typically use an imperative style i.e. they usually begin with a verb denoting the action required. Note that this is a data transfer object. In other words, it is simply a bag of data attributes. Also note how it is devoid of any logic (at least at the moment).

② The identifier for the LC Application. We are assuming client generated identifiers in this case. The topic of using server-generated versus client-generated identifiers is out of scope for the subject of this book. You may use either depending on what is advantageous in your context. Also note that we are using a strong type for the identifier `LCApplicationId` as opposed to a primitive such as a numeric or a string value. It is also common in some cases to use UUIDs as the identifier. However, we prefer using strong types to be able to differentiate between identifier types. Notice how we are using a type `ClientId` to represent the creator of the application.

③ The `Party` and `AdvisingBank` types are complex types to represent those concepts in our solution. Care should be taken to consistently use names that are relevant in the problem (business) domain as opposed to using names that only make sense in the solution (technology) domain. Note the attempt to make use of the *ubiquitous language* of the domain experts in both cases. This is a practice that we should always be conscious of when naming things in the system.

It is worth noting that the `merchandiseDescription` is left as a primitive `String` type. This may feel contradictory to the commentary we present above. We will address this in the upcoming section on validations.

Now let's look at what the event we will emit as a result of successfully processing the command will look like.

Implementing the event

In an event-driven system, mutating system state by successfully processing a command usually results in a domain event being emitted to signal the state mutation to the rest of the system. A simplified representation of a real-world `LCApplicationCreatedEvent` is shown here:

```
import lombok.Data;

@Data
public class LCApplicationCreatedEvent {    ①

    private LCApplicationId id;
    private ClientId clientId;
    private Party applicant;
    private Party beneficiary;
    private AdvisingBank advisingBank;
    private LocalDate issueDate;
    private MonetaryAmount amount;
    private String merchandiseDescription;

}
```

① The event type. When naming events, we typically use names in the past tense to denote things that have already occurred and are to be accepted unconditionally as empirical facts that cannot be changed.

You will likely notice that the structure of the event is currently identical to that of the command. While this is true in this case, it may not always be that way. The amount of information that we choose to disclose in an event is context-dependent. It is important to consult with domain experts when publishing information as part of events. One may choose to withhold certain information in the event payload. For example, consider a `ChangePasswordCommand` which contains the newly changed password. It might be prudent to not include the changed password in the resulting `PasswordChangedEvent`.

We have looked at the command and the resulting event in the previous test. Let's look at how this is implemented under the hood by looking at the aggregate implementation.

Designing the aggregate

The aggregate is the place where commands are handled and events are emitted. The good thing about the test that we have written is that it is expressed in a manner that hides the implementation details. But let's look at the implementation to be able to appreciate how we can get our tests to pass and meet the business requirement.

```
public class LCApplication {

    @AggregateIdentifier                                                    ①
    private LCApplicationId id;

    @SuppressWarnings("unused")
    private LCApplication() {
        // Required by the framework
    }

    @CommandHandler                                                         ②
    public LCApplication(CreateLCApplicationCommand command) {              ③
        // TODO: perform validations here
        AggregateLifecycle.apply(new LCApplicationCreatedEvent(command.getId()));   ④
    }

    @EventSourcingHandler                                                   ⑤
    private void on(LCApplicationCreatedEvent event) {
        this.id = event.getId();
    }
}
```

① The aggregate identifier for the `LCApplication` aggregate. All aggregates are required to declare an identifier and mark it so using the `@AggregateIdentifier` annotation provided by the framework.

② The method that is handling the command needs to be annotated with the `@CommandHandler` annotation. In this case, the command handler happens to be the constructor of the class given that this the first command that can be received by this aggregate. We will see examples of subsequent commands being handled by other methods later in the chapter.

③ The `@CommandHandler` annotation marks a method as being a command handler. The exact command that this method can handle needs to be passed as a parameter to the method. Do note that there can only be one command handler in the **entire** system for any given command.

④ Here, we are emitting the `LCApplicationCreatedEvent` using the `AggregateLifecycle` utility provided by the framework. In this very simple case, we are emitting an event unconditionally on receipt of the command. In a real-world scenario, it is conceivable that a set of validations will be performed before deciding to either emit one or more events or failing the command with an exception. We will look at more realistic examples later in the chapter.

⑤ The need for the `@EventSourcingHandler` and its role are likely very unclear at this time. We will explain the need for this in detail in an upcoming section of this chapter.

This was a whirlwind introduction to a simple event-driven system. We still need to understand the role of the `@EventSourcingHandler`. To understand that, we will need to appreciate how aggregate persistence works and the implications it has on our overall design.

Persisting aggregates

When working with any system of even moderate complexity, we are required to make interactions durable. That is, interactions need to outlast system restarts, crashes, etc. So the need for persistence is a given. While we should always endeavour to abstract persistence concerns from the rest of the system, our persistence technology choices can have a significant impact on the way we architect our overall solution. We have a couple of choices in terms of how we choose to persist aggregate state that are worth mentioning:

1. State stored
2. Event sourced

Let's examine each of these techniques in more detail below:

State stored aggregates

Saving current values of entities is by far the most popular way to persist state — thanks to the immense popularity of relational databases and object-relational mapping (ORM) tools like Hibernate. And there is good reason for this ubiquity. Until recently, a majority of enterprise systems used relational databases almost as a default to create business solutions, with ORMs arguably providing a very convenient mechanism to interact with relational databases and their object representations. For example, for our `LCApplication`, it is conceivable that we could use a relational database with a structure that would look something like below:

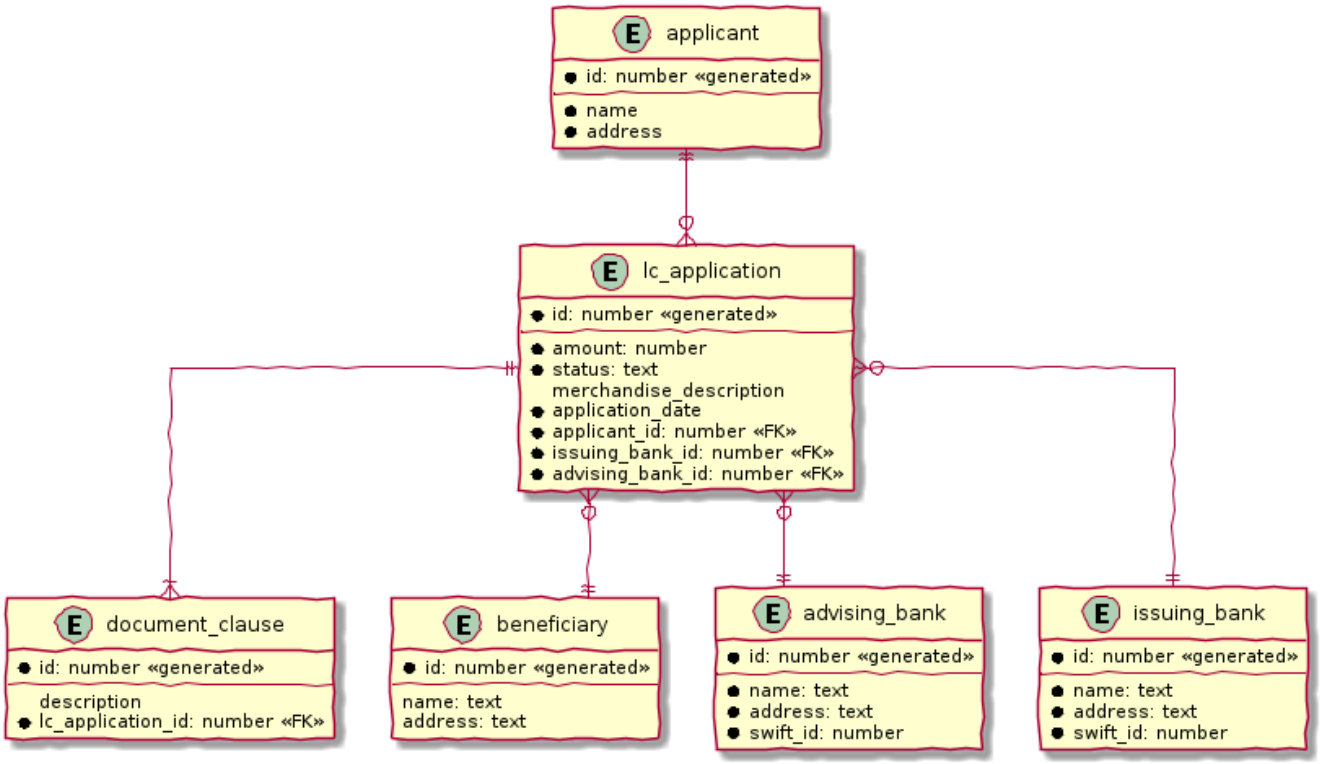


Figure 6. Typical entity relationship model

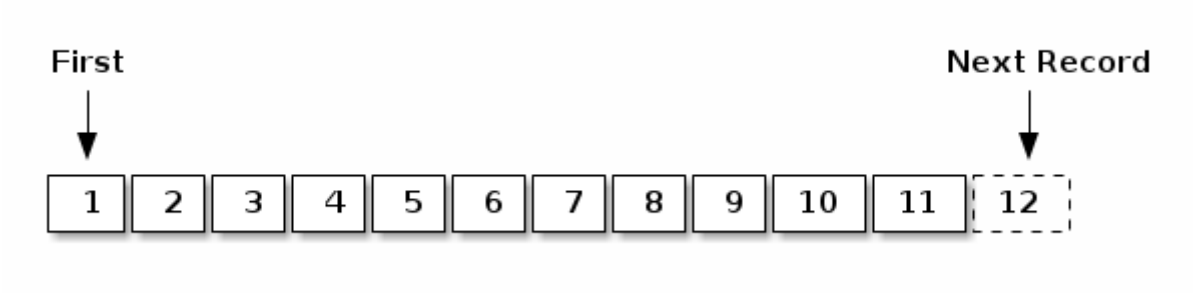
Irrespective of whether we choose to use a relational database or a more modern NoSql store — for instance, a document store, key-value store, column family store, etc., the style we use to persist information remains more or less the same — which is to store the current values of the attributes

of the said aggregate/entity. When the values of attributes change, we simply overwrite old values with newer ones i.e. we store the current state of aggregates and entities — hence the name *state stored*. This technique has served us very well over the years, but there is at least one more mechanism that we can use to persist information. We will look at this in more detail below.

Event sourced aggregates

Developers have also been relying on logs for a variety of diagnostic purposes for a very long time. Similarly, relational databases have been employing commit logs to store information durably almost since their inception. However, developers' use of logs as a first class persistence solution for structured information in mainstream systems remains extremely rare.

A log is an extremely simple, append-only sequence of immutable records ordered by time.



Writing to a log as compared to a more complex data structure like a table is a relatively simple and fast operation and can handle extremely high volumes of data while providing predictable performance. Indeed, a modern event streaming platform like Kafka makes use of this pattern to scale to support extremely high volumes. We do feel that this can be applied to act as a persistence store when processing commands in mainstream systems because this has benefits beyond the technical advantages listed above. Consider the example of an online order flow below:

User Action	Traditional Store	Event Store
Add milk to cart	Order 123: Milk in cart	E1: Cart#123 created E2: Milk added to cart
Add white bread to cart	Order 123: Milk, White bread in cart	E1: Cart#123 created E2: Milk added to cart E3: White bread added to cart
Remove White bread from cart	Order 123: Milk in cart	E1: Cart#123 created E2: Milk added to cart E3: White bread added to cart E4: White bread removed from cart
Add Wheat bread to cart	Order 123: Milk, Wheat bread in cart	E1: Cart#123 created E2: Milk added to cart E3: White bread added to cart E4: White bread removed from cart E5: Wheat bread added to cart

User Action	Traditional Store	Event Store
Confirm cart checkout	Order 123: Ordered Milk, Wheat bread	E1: Cart#123 created E2: Milk added to cart E3: White bread added to cart E4: White bread removed from cart E5: Wheat bread added to cart E6: Order 123 confirmed

As you can see, in the event store, we continue to have full visibility of all user actions performed. This allows us to reason about these behaviors more holistically. In the traditional store, we lost the information that the user replaced white with wheat bread. While this does not impact the order itself, we lose the opportunity to gather insights from this user behavior. We recognize that this information can be captured in other ways using specialized analytical solutions, however, the event log mechanism provides a natural way to do this without requiring any additional effort, thereby reducing the complexity of the system being built. It also acts as an audit log providing full history of all events that have occurred thus far. This fits well with the essence of domain-driven design where we are constantly exploring ways in which to reduce complexity.

However, there are implications to persisting data in the form of a simple event log. Before processing any command, we will need to hydrate past events in exact order of occurrence and reconstruct aggregate state to allow us to perform validations. For example, when confirming checkout, just having the ordered set of elapsed events will not suffice. We still need to compute the exact items that are in the cart before allowing the order to be placed. This *event replay* to restore aggregate state (at least those attributes that are required to validate said command) is necessary before processing that command. For example, we need to know which items are in the cart currently before processing the `RemoveItemFromCartCommand`. This is illustrated in the following table:

Elapsed Events	Aggregate State	Command	Event(s) Emitted
—	—	Add item: milk	E1: Cart#*123* created E2: Milk added
E1: Cart#123 created E2: Milk added	**Cart Items**: Milk	Add item: white bread	E2: White bread added
E1: Cart#123 created E2: Milk added E3: White bread added	**Cart Items**: Milk, White Bread	Remove item: white bread	E3: White bread removed
E1: Cart#123 created E2: Milk added E3: White bread added E4: White bread removed	**Cart Items**: Milk	Add item: wheat bread	E4: Wheat bread added
E1: Cart#123 created E2: Milk added E3: White bread added E4: White bread removed E5: Wheat bread added	**Cart Items**: Milk Wheat bread	Confirm checkout for Cart#123	E5: Order created!

The corresponding source code for the whole scenario is illustrated in the following code snippet:

```
public class Cart {

    private boolean isNew;
    private CartItems items;
    //..

    private Cart() {                                                    ①
        // Required by the framework
    }

    @CommandHandler
    public void addItem(AddItemToCartCommand command) {
        // Business validations here
        if (this.isNew) {
            apply(new CartCreatedEvent(command.getId()));               ②
        }
        apply(new ItemAddedEvent(id, command.getItem()));               ②
    }

    @CommandHandler
    public void removeItem(RemoveItemFromCartCommand command) {
        // Business validations here
        apply(new ItemRemovedEvent(id, commmand.getItem()));
    }

    @CommandHandler
    public void checkout(ConfirmCheckoutCommand command) {
        // Business validations here
        apply(new OrderCreatedEvent(this.items));
    }

    @EventSourcingHandler                                               ③
    private void on(CartCreatedEvent event) {
        this.id = event.getCartId();
        this.items = new CartItems();
        this.isNew = true;
    }

    @EventSourcingHandler                                               ③
    private void on(ItemAddedEvent event) {
        this.items.add(event.getItem());
        this.isNew = false;
    }

    @EventSourcingHandler                                               ③
    private void on(ItemRemovedEvent event) {
        this.items.remove(event.getItem());
    }
```

```
    @EventSourcingHandler                                    ③
    private void on(CheckoutConfirmedEvent event) {
        // ..
    }
}
```

① Before processing any command, the aggregate loading process commences by first invoking the no-args constructor. For this reason, we need the no-args constructor to be **empty** i.e. it should **not** have any code that restores state. State restoration **must** happen only in those methods that trigger an event replay. In the case of the Axon framework, this translates to methods embellished with the `@EventSourcingHandler` annotation.

② It is important to note that it is possible (but not necessary) to emit **more than one event** after processing a command. This is illustrated in the first instance of the `AddItemCommand` in the previous code where we emit `CartCreatedEvent` and `ItemAddedEvent`.

③ The loading process continues through the invocation of event sourcing handler methods in exactly the order of occurrence for that aggregate instance.

When working with event sourced aggregates, it is very important to be disciplined about the kind of code that one can write:

Type of Method	State Restoration	Business Logic	Event Emission
`@CommandHandler`	No	Yes	Yes
`@EventSourcingHandler`	Yes	No	No

If there are a large number of events, aggregate loading can become a time-consuming operation — directly proportional to the number of elapsed events for that aggregate. There are techniques (like snapshotting) we can employ to overcome this. We will cover this in more detail in a subsequent chapter.

Which persistence mechanism should we choose?

Now that we have a reasonably good understanding of the two types of aggregate persistence mechanisms, it begs the question of which one we should choose. We list a few benefits of using event sourcing below:

- We get to use the events as a **natural audit log** in high compliance scenarios.
- It provides the ability to perform **more insightful analytics** on the basis of the fine-grained events data.
- It arguably produces more flexible designs when we work with an system based on **immutable events** — because the complexity of the persistence model is capped. Also, there is no need to deal with complex ORM impedance mismatch problems.
- The domain model is much more **loosely coupled** with the persistence model — enabling it to evolve mostly independently from the persistence model.
- Enables going back in time to be able to create **adhoc views and reports** without having to deal with upfront complexity.

On the flip side, these are some challenges that you might have to consider when implementing an event sourced solution:

- Event sourcing requires a **paradigm shift**. Which means that development and business teams will have to spend time and effort understanding how it works.
- The persistence model does not store state directly. This means that **adhoc querying** directly on the persistence model can be a lot more **challenging**. This can be alleviated by materializing new views, however there is added complexity in doing that.
- Event sourcing usually tends to work very well when implemented in conjunction with **CQRS** which arguably may add more complexity to the application. It also requires applications to pay closer attention to strong vs **eventual consistency** concerns.

Our experiences indicate that event sourced systems bring a lot of benefits in modern event-driven systems. However, you will need to be cognizant of the considerations presented above in the context of your own ecosystems when making persistence choices.

Policy enforcements (validations)

When processing commands, we need to enforce policies or rules. Policies come in two broad categories:

- Structural rules — those that enforce that the syntax of the dispatched command is valid.
- Domain rules — those that enforce that business rules are adhered to.

It may also be prudent to perform these validations in different layers of the system.And it is also common for some or all of these policy enforcements to be repeated in more than one layer of the system.However, the important thing to note is that before a command is successfully handled, all these policy enforcements are uniformly applied.Let's look at some examples of these in the upcoming section.

Structural validations

Currently, to create an LC application, one is required to dispatch a `CreateLCApplicationCommand`. While the command dictates a structure, none of it is enforced at the moment. Let's correct that.

To be able to enable validations declaratively, we will make use of the JSR-303 bean validation libraries. We can add that easily using the `spring-boot-starter-validation` dependency to our `pom.xml` file as shown here:

```
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
```

Now we can add validations to the command object using the JSR-303 annotations as depicted below:

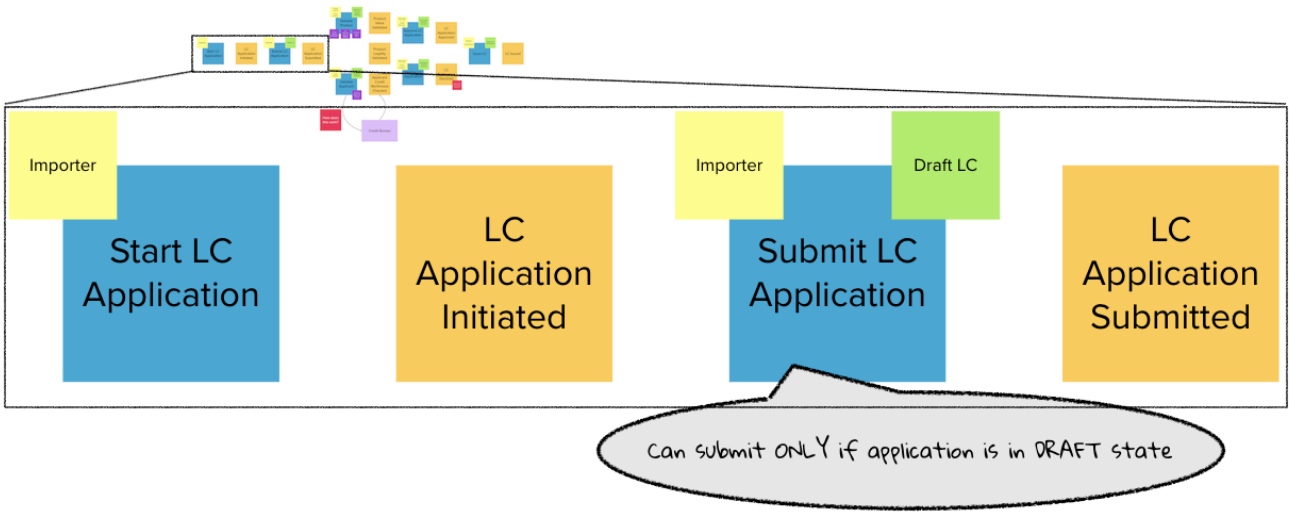
```
import lombok.Data;
import javax.validation.*;
import javax.validation.constraints.*;

@Data
public class CreateLCApplicationCommand {

    @NotNull
    private LCApplicationId id;

    @NotNull
    private ClientId clientId;

    @NotNull
    @Valid
    private Party applicant;

    @NotNull
    @Valid
    private Party beneficiary;

    @NotNull
    @Valid
    private AdvisingBank advisingBank;

    @Future
    private LocalDate issueDate;

    @Positive
    private MonetaryAmount amount;

    @NotBlank
    private String merchandiseDescription;
}
```

Most structural validations can be accomplished using the built-in validator annotations. It is also possible to create custom validators for individual fields or to validate the entire object (for example, to validate inter-dependent attributes). For more details on how to do this, please refer to the bean validation specification at https://beanvalidation.org/2.0/ and the reference implementation at http://hibernate.org/validator/.

Business rule enforcements

Structural validations can be accomplished using information that is already available in the command.However, there is another class of validations that requires information that is not present in the incoming command itself.This kind of information can be present in one of two places: within the aggregate that we are operating on or outside of the aggregate itself, but made available within the bounded context.

Let's look at an example of a validation that requires state present within the aggregate. Consider the example of submitting an LC.While we can make several edits to the LC when it is in draft state, no changes can be made after it is submitted.This means that we can only submit an LC once.This act of submitting the LC is achieved by issuing the `SubmitLCApplicationCommand` as shown in the artifact from the eventstorming session:



Let's begin with a test to express our intent:

```
class LCApplicationAggregateTests {
    //..
    @Test
    void shouldAllowSubmitOnlyInDraftState() {
        final LCApplicationId applicationId = LCApplicationId.randomId();

        fixture.given(new LCApplicationCreatedEvent(applicationId))            ①
                .when(new SubmitLCApplicationCommand(applicationId))           ②
                .expectEvents(new LCApplicationSubmittedEvent(applicationId)); ③
    }
}
```

① Given that the `LCApplicationCreatedEvent` has already occurred — in other words, the LC application is already created.

② When we try to submit the application by issuing the `SubmitLCApplicationCommand` for the same application.

③ We expect the `LCApplicationSubmittedEvent` to be emitted.

The corresponding implementation will look something like:

```
class LCApplication {
    // ..
    @CommandHandler
    public void submit(SubmitLCApplicationCommand command) {
        apply(new LCApplicationSubmittedEvent(id));
    }
}
```

The implementation above allows us to submit an LC application unconditionally — more than once. However, we want to restrict users to be able to submit only once. To be able to do that, we need to remember that the LC application has already been submitted. We can do that in the `@EventSourcingHandler` of the corresponding events as shown below:

```
class LCApplication {
    // ..
    @EventSourcingHandler
    private void on(LCApplicationSubmittedEvent event) {
        this.state = State.SUBMITTED; ①
    }
}
```

① When the `LCApplicationSubmittedEvent` is replayed, we set the state of the `LCApplication` to `SUBMITTED`.

While we have remembered that the application has changed to be in `SUBMITTED` state, we are still not preventing more than one submit attempt. We can fix that by writing a test as shown below:

```
class LCApplicationAggregateTests {
    @Test
    void shouldNotAllowSubmitOnAnAlreadySubmittedLC() {
        final LCApplicationId applicationId = LCApplicationId.randomId();

        fixture.given(
                new LCApplicationCreatedEvent(applicationId),           ①
                new LCApplicationSubmittedEvent(applicationId))         ①

                .when(new SubmitLCApplicationCommand(applicationId))    ②

                .expectException(AlreadySubmittedException.class)       ③
                .expectNoEvents();                                      ④
    }
}
```

① The `LCApplicationCreatedEvent` and `LCApplicationSubmittedEvent` have already happened — which means that the `LCApplication` has been submitted once.

② We now dispatch another `SubmitLCApplicationCommand` to the system.

③ We expect an `AlreadySubmittedException` to be thrown.

④ We also expect no events to be emitted.

The implementation of the command handler to make this work is shown below:

```
class LCApplication {
    // ..
    @CommandHandler
    public void submit(SubmitLCApplicationCommand command) {
        if (this.state != State.DRAFT) {                                    ①
            throw new AlreadySubmittedException("LC is already submitted!");
        }
        apply(new LCApplicationSubmittedEvent(id));
    }
}
```

① Note how we are using the state attribute from the `LCApplication` aggregate to perform the validation. If the application is not in `DRAFT` state, we fail with the `AlreadySubmittedException` domain exception.

Let's also look at an example where information needed to perform the validation is not part of either the command or the aggregate. Let's consider the scenario where country regulations prohibit transacting with a set of so called *sanctioned* countries. Changes to this list of countries may be affected by external factors. Hence it does not make sense to pass this list of sanctioned countries as part of the command payload. Neither does it make sense to maintain it as part of every single aggregate's state — given that it can change (albeit very infrequently). In such a case, we may want to consider making use of a command handler that is outside the confines of the aggregate class. Thus far, we have only seen examples of `@CommandHandler` methods within the aggregate. But the `@CommandHandler` annotation can appear on any other class external to the aggregate. However, in such a case, we need to load the aggregate ourselves. The Axon framework provides a `org.axonframework.modelling.command.Repository` interface to allow us to do that. It is important to note that this `Repository` is distinct from spring framework's interface that is part of the spring data libraries. An example of how this works is shown below:

```
import org.axonframework.modelling.command.Repository;

class MyCustomCommandHandler {

    private final Repository<LCApplication> repository; ①

    MyCustomCommandHandler(Repository<LCApplication> repository) {
        this.repository = repository;  ①
    }

    @CommandHandler
    public void handle(SomeCommand command) {
        Aggregate<LCApplication> application = repository.load(command.getAggregateId
());  ②
        // Command handling code
    }

    @CommandHandler
    public void handle(AnotherCommand command) {
        Aggregate<LCApplication> application = repository.load(command.getAggregateId
());
        // Command handling code
    }
}
```

① We are injecting the Axon `Repository` to allow us to load aggregates. This ws not required previously because the `@CommandHandler` annotation appeared on aggregate methods directly.

② We are using the `Repository` to load aggregates and work with them. The `Repository` interface supports other convenience methods to work with aggregates. Please refer to the Axon framework documentation for more usage examples.

Coming back to the sanctioned countries example, let's look at how we need to set up the test slightly differently:

```
public class CreateLCApplicationCommandHandlerTests {
    private FixtureConfiguration<LCApplication> fixture;

    @BeforeEach
    void setUp() {
        final Set<Country> sanctioned = Set.of(SOKOVIA);
        fixture = new AggregateTestFixture<>(LCApplication.class);                ①

        final Repository<LCApplication> repository = fixture.getRepository();    ②

        CreateLCApplicationCommandHandler handler =
                new CreateLCApplicationCommandHandler(repository, sanctioned);   ③
        fixture.registerAnnotatedCommandHandler(handler);                         ④
    }
}
```

① We are creating a new aggregate fixture as usual

② We are using the fixture to obtain an instance of the Axon `Repository`

③ We instantiate the custom command handler passing in the `Repository` instance. Also note how we inject the collection of sanctioned countries into the handler using simple dependency injection. In real life, this set of sanctioned countries will likely be obtained from external configuration.

④ We finally need to register the command handler with the fixture, so that it can route commands to this handler as well.

The tests for this look fairly straightforward:

```
class CreateLCApplicationCommandHandlerTests {
    // ..

    @BeforeEach
    void setUp() {
    final Set<Country> sanctioned = Set.of(SOKOVIA);                          ①
        fixture = new AggregateTestFixture<>(LCApplication.class);

        final Repository<LCApplication> repository = fixture.getRepository();

        CreateLCApplicationCommandHandler handler =
                new CreateLCApplicationCommandHandler(repository, sanctioned);  ②
        fixture.registerAnnotatedCommandHandler(handler);
    }

    @Test
    void shouldFailIfBeneficiaryCountryIsSanctioned() {
        fixture.given()
                .when(new CreateLCApplicationCommand(randomId(), SOKOVIA))      ③
                .expectNoEvents()
                .expectException(CannotTradeWithSanctionedCountryException.class);
    }

    @Test
    void shouldCreateIfCountryIsNotSanctioned() {
        final LCApplicationId applicationId = randomId();
        fixture.given()
                .when(new CreateLCApplicationCommand(applicationId, WAKANDA))   ④
                .expectEvents(new LCApplicationCreatedEvent(applicationId));
    }
}
```

① For the purposes of the test, we mark the country `SOKOVIA` as a *sanctioned* country. In a more realistic scenario, this will likely come from some form external configuration (e.g. a lookup table or form of external configuration). However, this is appropriate for our unit test.

② We then inject this set of *sanctioned countries* into the command handler.

③ When the `LCApplication` is created for the sanctioned country, we expect no events to be emitted and furthermore, the `CannotTradeWithSanctionedCountryException` exception to be thrown.

④ Finally, when the beneficiary belongs to a non-sanctioned country, we emit the `LCApplicationCreatedEvent` to be emitted.

The implementation of the command handler is shown below:

```
import org.springframework.stereotype.Service;

@Service ①
public class CreateLCApplicationCommandHandler {
    private final Repository<LCApplication> repository;
    private final Set<Country> sanctionedCountries;

    public CreateLCApplicationCommandHandler(Repository<LCApplication> repository,
                                             Set<Country> sanctionedCountries) {
        this.repository = repository;
        this.sanctionedCountries = sanctionedCountries;
    }

    @CommandHandler
    public void handle(CreateLCApplicationCommand command) {
        // Validations can be performed here as well ②
        repository.newInstance(() -> new LCApplication(command, sanctionedCountries));
③
    }
}
```

① We mark the class as a `@Service` to mark it as a component devoid of encapsulated state and enable auto-discovery when using annotation-based configuration or classpath scanning. As such, it can be used to perform any "plumbing" activities.

② Do note that the validation for the beneficiary's country being sanctioned could have been performed on line 18 as well. Some would argue that this would be ideal because we could avoid a potentially unnecessary invocation of the Axon `Repository` method if we did that. However, we prefer encapsulating business validations within the confines of the aggregate as much as possible — so that we don't suffer from the problem of creating an anemic domain model[3].

Finally, the aggregate implementation along with the validation is shown here:

```
class LCApplication {
// ...
    public LCApplication(CreateLCApplicationCommand command, Set<Country> sanctioned)
{
        if (sanctioned.contains(command.getBeneficiaryCountry())) { ①
            throw new CannotTradeWithSanctionedCountryException();
        }
        apply(new LCApplicationCreatedEvent(command.getId()));
    }
}
```

① The validation itself is fairly straightforward. We throw a `CannotTradeWithSanctionedCountryException` when the validation fails.

With the above examples, we looked at different ways to implement the policy enforcements encapsulated within the boundaries the aggregate.

Summary

In this chapter, we used the outputs of the eventstorming session and used it as a primary aid to create a domain model for our bounded context. We looked at how to implement this using the command query responsibility segregation (CQRS) architecture pattern. We looked at persistence options and the implications of using event sourced vs state stored aggregates. Finally, we rounded off by looking at a variety of ways in which to perform business validations. We looked at all this through a set of code examples using Spring boot and the Axon framework.

With this knowledge, we should be able to implement robust, well encapsulated, event-driven domain models. In the next chapter, we will look at implementing a user interface for these domain capabilities and examine a few options such as CRUD-based vs task-based UIs.

Questions

1. Can you examine the eventstorming session artifact from the last chapter, and identify the possible aggregates that would be required?
2. In your problem domain, can you determine the right approach for persisting aggregates? What are the reasons for choosing one approach over the other?
3. Based on your current understanding, would you apply CQRS architecture pattern in your solution? And how would you justify the choice to your team ?

Further reading

Title	Author	Location
CQRS	Martin Fowler	https://martinfowler.com/bliki/CQRS.html
Bootiful CQRS and Event Sourcing with Axon Framework	SpringDeveloper and Allard Buijze	https://www.youtube.com/watch?v=7e5euKxHhTE
The Log: What every software engineer should know about real-time data's unifying abstraction	Jay Kreps	https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying
Event Sourcing	Martin Fowler	https://martinfowler.com/eaaDev/EventSourcing.html
Using a DDD Approach for Validating Business Rules	Fabian Lopez	https://www.infoq.com/articles/ddd-business-rules/
Anemic Domain Model	Martin Fowler	https://www.martinfowler.com/bliki/AnemicDomainModel.html

Answers

1. Refer to section Identifying Aggregates
2. Refer to section Persisting aggregates, note down the pros and cons of state stored and event sourced approach, and discuss the reasons for your choice with your teammates.
3. Refer to section Why CQRS? to list down the advantages of the approach versus the traditional approach. Share the reasoning with your teammates.

[1] https://www.lagomframework.com/

[2] https://eventuate.io/

[3] https://www.martinfowler.com/bliki/AnemicDomainModel.html