Axon Training

Module 2 – Command Model

AxonIQ

Agenda

Week 1

1. DDD and CQRS Fundamentals
2. **Command Model**
3. Event Handling & Projections
4. Sagas and Deadlines

Week 2

1. Snapshotting and Event Processors
2. Preparing for Production
3. CQRS and Distributed Systems
4. Monitoring, Tracing, Advanced Tuning

Remember?

DDD & CQRS recap

Model

> A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain.

Two Models

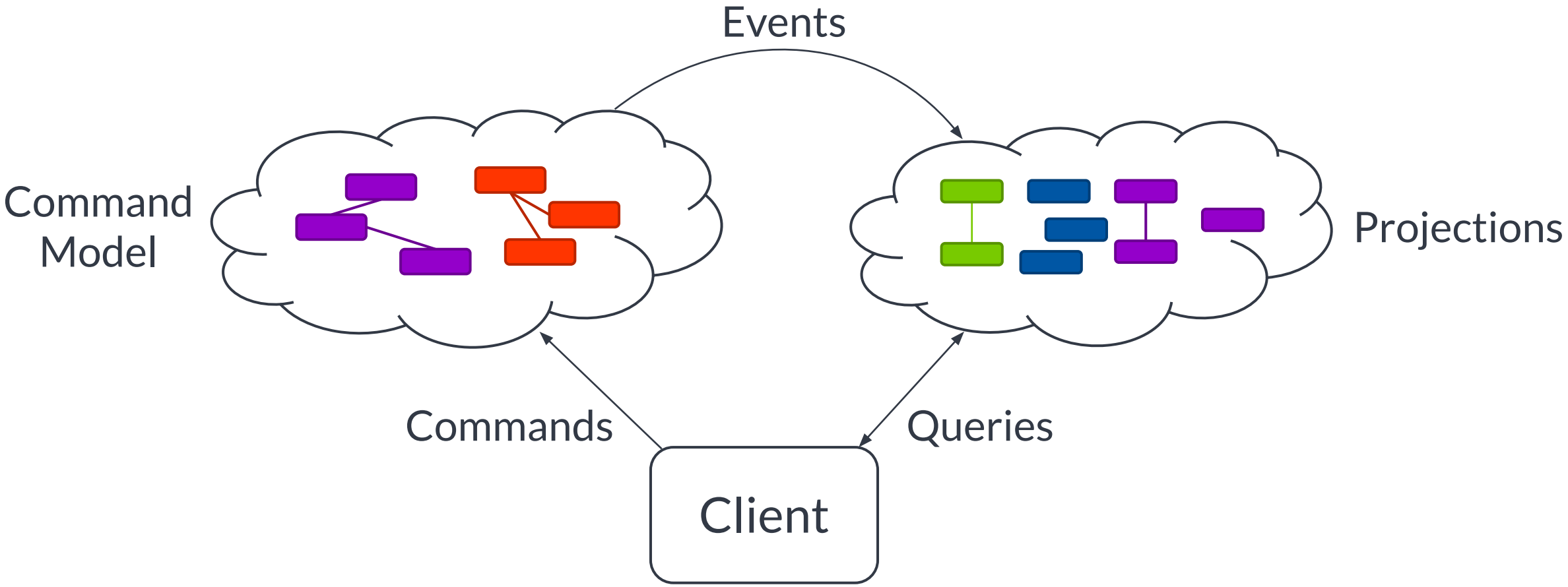
Command Model

- Focused on executing tasks.
- Primarily expressed in operations.
- Only contains data necessary for task execution and decision making.

Query Model / Projections

- Focused on delivering information.
- Data is stored the way it is used.
- Denormalized / “table-per-view”

Command Query Responsibility Segregation



Aggregate

> A group of associated objects which are considered as one unit with regard to data changes…

Entity

> Objects that are not fundamentally defined by their attributes, but rather by a thread of continuity and identity.

Event

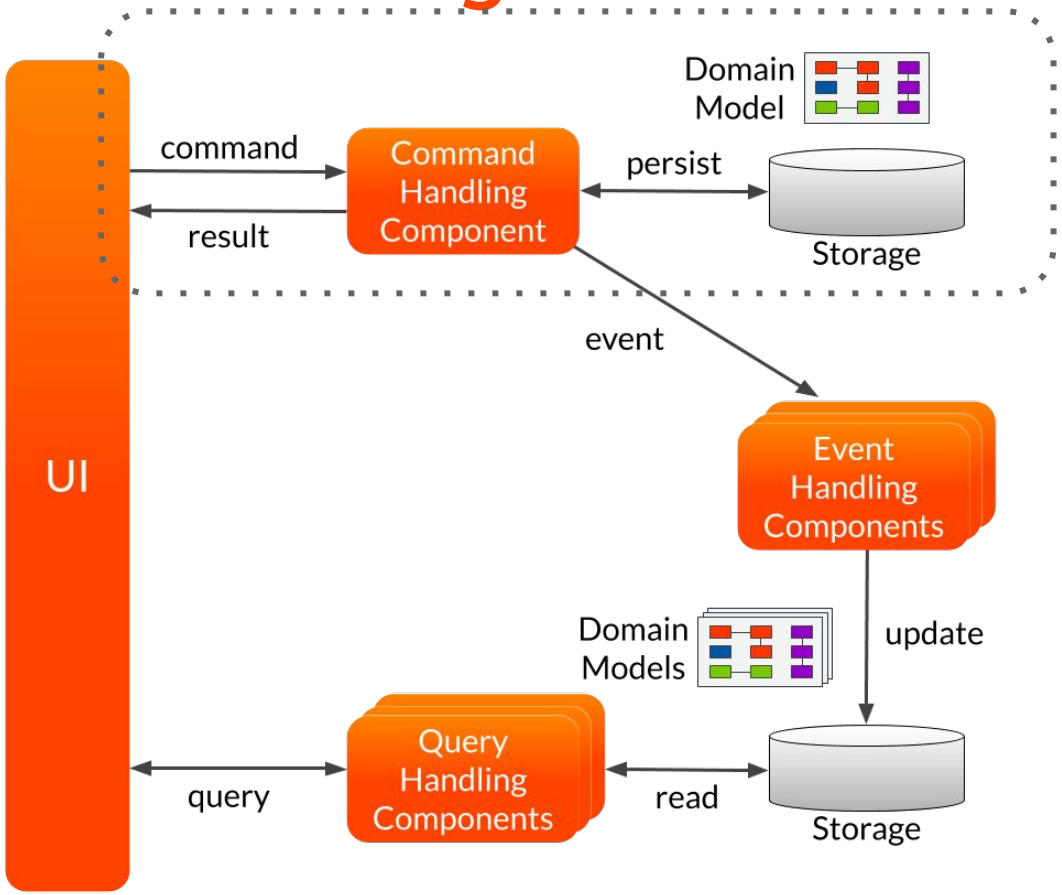
> A notification that something relevant has happened inside the domain.

Unraveling the consequences of intent

Command Handling

AxonIQ

Command Handling



Command Handler

- Accepts incoming commands
- Consults (and updates) the command model and publishes events
- Command model only contains data necessary for task execution and decision making

Command Handlers in Axon Framework

- Component that is subscribed to the Command Bus to process specific Commands
- `@CommandHandler`
  - On (singleton) component
  - Directly on Command Model

```
@CommandHandler
public void handle(MyCommand command) {
  ...
}
```

Command Handling Component (with Spring)

```
@Component
public class CommandHandlingComponent {

  @CommandHandler
  public void handle(SomeCommand cmd) {
    // do what you need to do
  }

}
```

Makes an instance available in the Application Context (Spring)

Command Model (with Spring)

```
import static org.axonframework.modelling.command.AggregateLifecycle.apply

public class MyCommandModel {

  private String id;
  // other state

  public void doSomething() {
    apply(new SomethingDoneEvent(id, ...));
  }
}
```

Publishes a SomethingDoneEvent via the event bus registered with the Repository that manages this instance's lifecycle.

Annotated Command Model (with Spring)

```
import static org.axonframework.modelling.command.AggregateLifecycle.apply

@Aggregate
public class MyCommandModel {

  @AggregateIdentifier
  private final String id;

  @CommandHandler
  public void handle(SomeCommand cmd) {
    apply(new SomethingDoneEvent(id, …));
  }
}
```

Tells Axon Spring auto configuration to set up necessary infrastructure

Indicates which of the fields is the identifier (may also be JPA's @Id)

Registers this method as a Command Handler for "SomeCommand"

Command Handler Parameters

Supported parameter types

- First parameter (if none of below) resolves to Message payload
- `Message` → Resolves to entire message
- `CommandMessage` → Resolves to `CommandMessage`
- `UnitOfWork` → Resolves to the current Unit of Work
- `MetaData` → Resolves to the Meta Data of the Message
- `@MetaDataValue ("name") ...` → Resolves to a Meta Data value of the Message
- Any Spring bean or component registered using Configuration API

Custom values using `ParameterResolverFactory`

Command Message routing

```
public class SomeCommand {

  @TargetAggregateIdentifier
  private final String id;
  // other state
  public SomeCommand(String id, …) {
    this.id = id;
  }
  // getters
}


// or in Kotlin:
class SomeCommand(@TargetAggregateIdentifier val id: String)
class SomethingDoneEvent(val id: String)
```

Marks the field that contains the value to use to load an Aggregate

Tip: Kotlin allows one-liner definitions of events. You can also group many of them in a single file.

Routing Commands to an Entity within an Aggregate

```
@Aggregate
public class MyCommandModel {
  @AggregateMember
  private MyChildEntity entity;
}

class MyChildEntity {
  @CommandHandler
  public void handle(ChildEntityCommand command) { ...}
}
```

Routing Commands to an Entities within an Aggregate

```
@Aggregate
public class MyCommandModel {
  @AggregateMember
  private List<MyChildEntity> entities;
}

class MyChildEntity {
  @EntityId(routingKey="someProperty")
  private String myChildEntityId;
  @CommandHandler
  public void handle(ChildEntityCommand command) { ...}
}
```

By default, the name of the Entity's field is looked up as property on the commands as routing key.

Dispatching Commands

- Directly on CommandBus:

```
CommandBus commandBus = ...;
commandBus.dispatch(asCommandMessage(new DoSomethingCommand()));
```

- Using Command Gateway

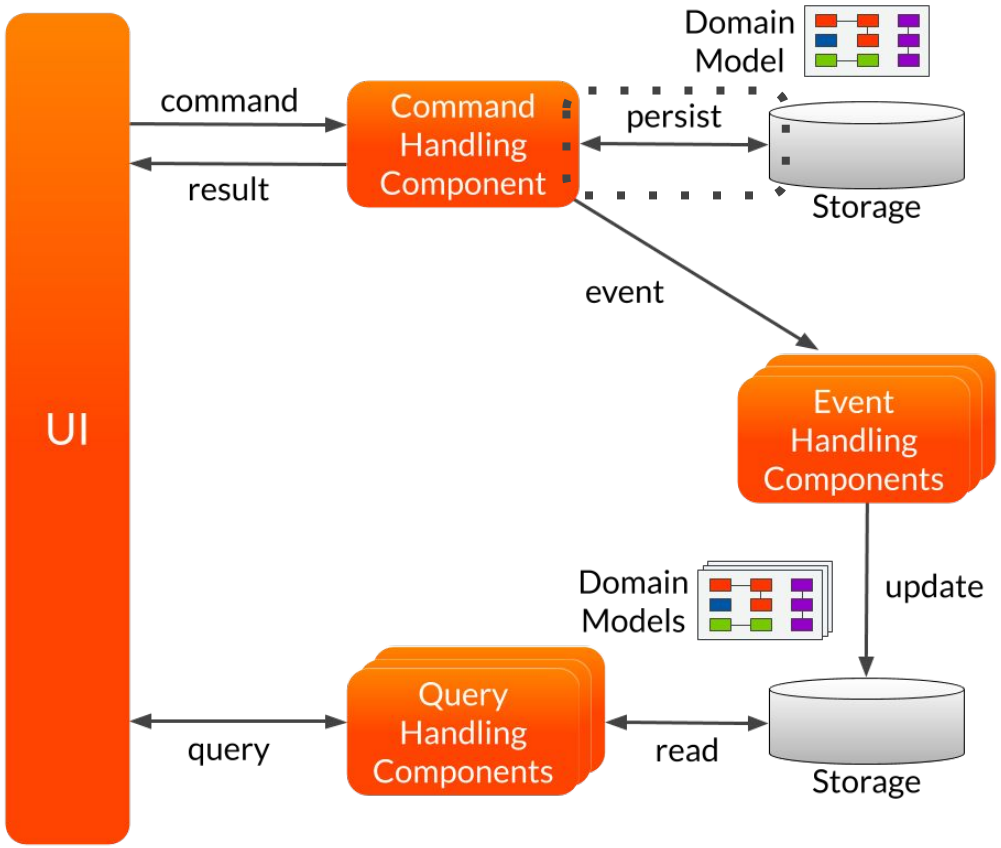
```
CommandGateway gateway = DefaultCommandGateway.builder().commandBus(commandBus).build();
// non-blocking
gateway.send(new DoSomethingCommand()); // returns CompletableFuture<>
gateway.send(new DoSomethingCommand(), callback);

// blocking
gateway.sendAndWait(new DoSomethingCommand());
gateway.sendAndWait(new DoSomethingCommand(), 1, TimeUnit.SECONDS);
```

Persisting state between commands

Storing Aggregate state

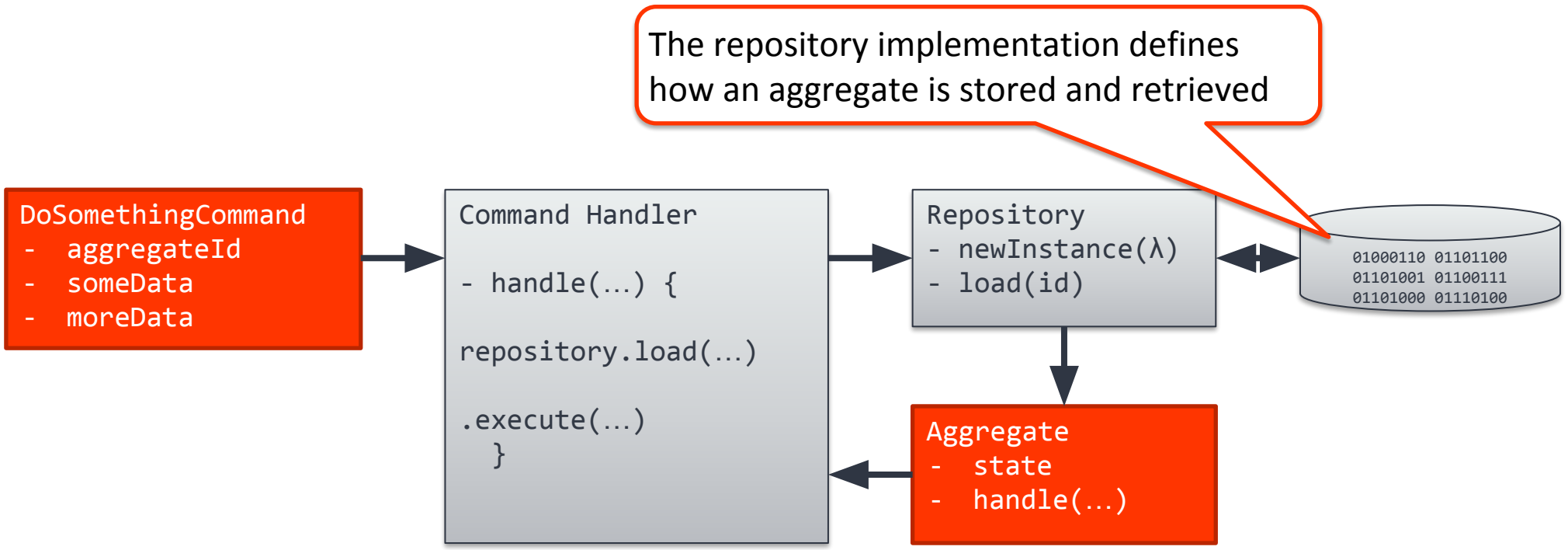
Aggregate persistence



Repository

> A mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects.

Repository in context



State storage

What happened

1. Flight scheduled: 6:00 - 10:00
2. Captain assigned: T. Cruise

What we store

Flight

flightNo: AC-323

planned departure time: 6:00

planned arrival time: 10:00

captain: T. Cruise

status: awaiting departure

State storage

What happened

1. Flight scheduled: 6:00 - 10:00
2. Captain assigned: T. Cruise

...

3. Departure slot missed: 6:00
4. Captain assigned: C. Lindbergh

What we store

Flight

flightNo: AC-323

planned departure time: 6:00

planned arrival time: 10:00

captain: C. Lindbergh

status: delayed at gate

State storage

What happened

1. Flight scheduled: 6:00 - 10:00
2. Captain assigned: T. Cruise

...

3. Departure slot missed: 6:00
4. Captain assigned: C. Lindbergh
5. Flight departed: 7:34

What we store

Flight

flightNo: AC-323

planned departure time: 6:00

planned arrival time: 10:00

captain: C. Lindbergh

actual departure time : 7:34

status: in-flight

State storage

What happened

1. Flight scheduled: 6:00 - 10:00
2. Captain assigned: T. Cruise

...

3. Departure slot missed: 6:00
4. Captain assigned: C. Lindbergh
5. Flight departed: 7:34
6. Arrival time estimated: 11:30
7. Arrival time estimated: 10:40
8. Flight arrived: 10:24

What we store

Flight

flightNo: AC-323
planned departure time: 6:00
planned arrival time: 10:00
captain: C. Lindbergh
actual departure time : 7:34
actual arrival time: 10:24
status: arrived

The valuable information is here…

Not here…

Events...

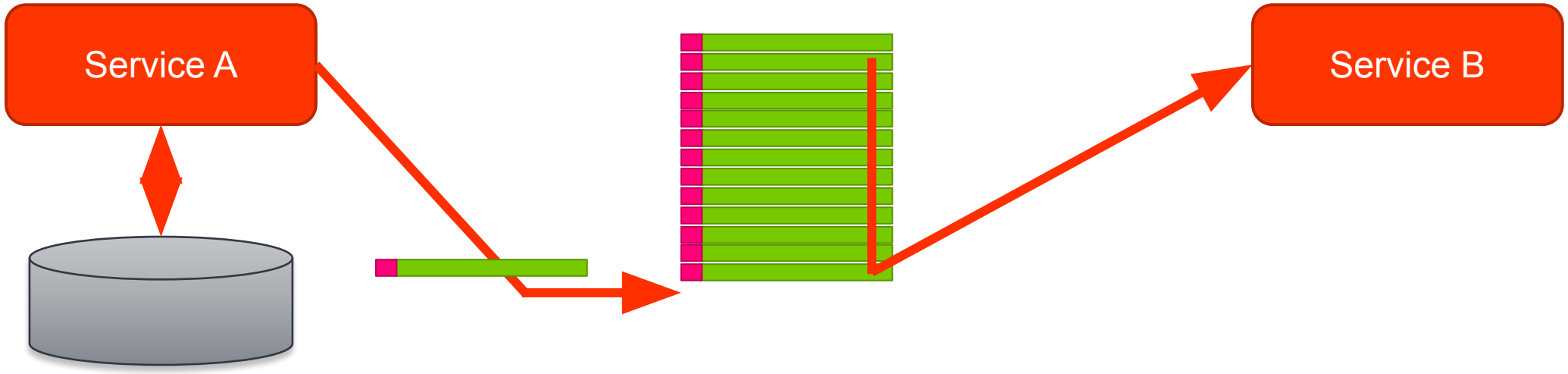
How do we guarantee that our events are a complete and truthful representation of an entity's history?

Event Sourcing

- Storage method for Command Model
  - Only persist changes
  - The applied Events represent all changes

- To load an aggregate:
  - Replay all past Events on an “empty” instance

Event Sourcing ≠ Streaming



Event Streaming

Event Sourcing ≠ Streaming

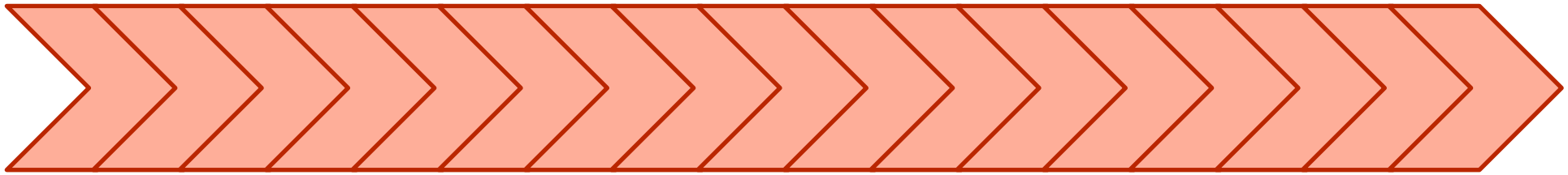
Event Sourcing

Event Store

An Event Store stores the published events to be retrieved both by consumers as well as the publishing component itself.

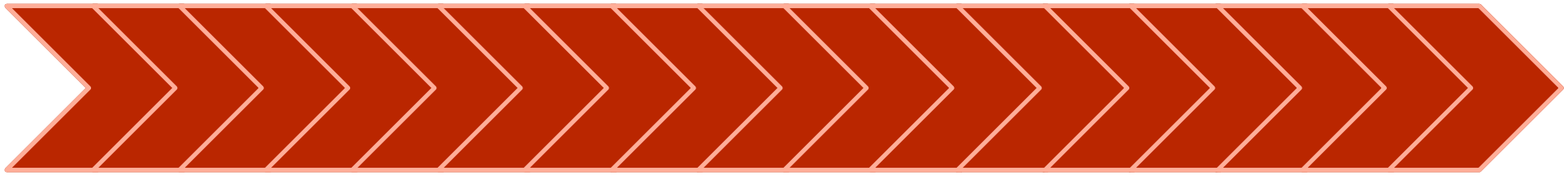
Event Store operations

- Append
- Validate ‘sequence’



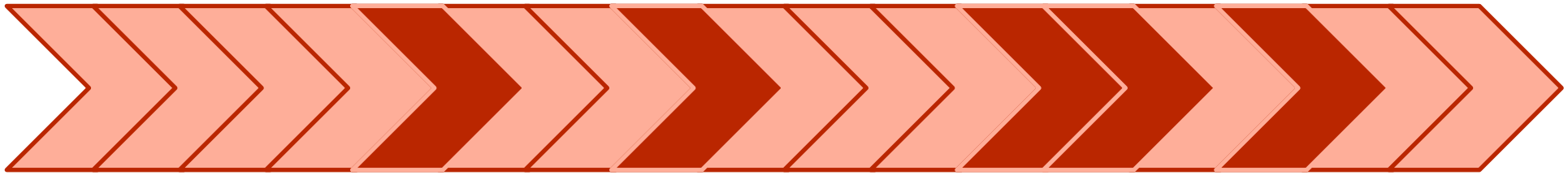
Event Store operations

- Full sequential read



Event Store operations

- Read aggregate's events



Event Sourcing infrastructure

- Event Store
  - AxonServer
  - JPA / JDBC
  - MongoDB

- Repository
  - `EventSourcingRepository`

Why use event sourcing?

Business reasons

- Auditing / compliance / transparency
- Data mining, analytics: value from data

Technical reasons

- *Guaranteed completeness of event stream*
- *Single source of truth*
- *Concurrency / conflict resolution*
- *Facilitates debugging*
- *Replay into new read models (+CQRS)*
- *Easily capture intent*
- *Deal with complexity in models*

Event Sourcing, or not...

- There is a price...
  - Your events must remain "readable" at all times. This means you must keep support for them, even old versions of Events.
  - Big refactoring of the Domain Model requires use-once custom tools
- Event Streams grow... indefinitely
  - How do you prevent the need for reading millions of events just to apply a single command?

We will answer this question later.

Event Sourcing – Entity layout

```java
@Aggregate
public class Flight {

    // state required to make decisions

    @CommandHandler
    public void handle(RegisterArrivalTime command) {
        if (currentArrivalTime.isBefore(command.getNewArrivalTime())) {
            apply(new FlightDelayed(flightId, command.getNewArrivalTime()));
        } else {
            apply(new ArrivalTimeChanged(flightId, command.getNewArrivalTime()));
        }
    }

    @EventSourcingHandler
    protected void on(ArrivalTimeChanged event) {
        this.currentArrivalTime = event.getNewArrivalTime();
    }
```

Decision making

State changes

AxonIQ

Applying Events

`AggregateLifecycle.apply(event)` will:

1. Dispatch the Event to *all* handlers *inside* the Aggregate
2. Send the Event to the Event Bus, which stages it for publication in the “prepare commit” phase of the Unit of Work

Note: Other aggregates will *not* receive the event

Declarative Testing

Given: a set of historic events

When: I send a command

Then: expect certain events

```
private FixtureConfiguration<Flight> fixture;

@BeforeEach
void setup() {
    fixture = new AggregateTestFixture<>(Flight.class);
}

@Test
void shouldFlagAsDelayed() {
    fixture.given(new FlightScheduled(FLIGHT_ID, 8_00, AMS, 10_00, LAX))
           .when(new RegisterArrivalTime(FLIGHT_ID, 10_01), ...)
           .expectEvents(new FlightDelayed(FLIGHT_ID, LAX, 10_01));
}
```

Whatever else you wanted to know...

Questions