

# Axon Training

Module 4 – Sagas, Process Managers & Deadlines

# Agenda

## Week 1

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

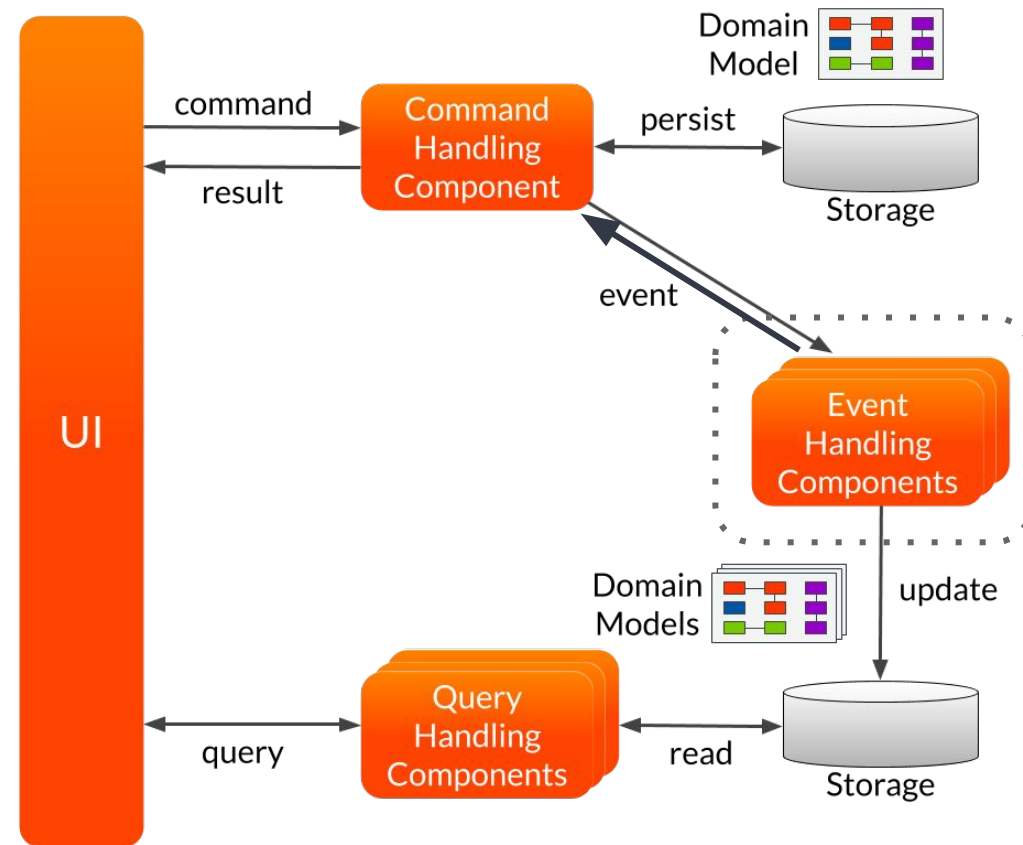
## Week 2

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

Managing business transactions

# Sagas / Process Managers

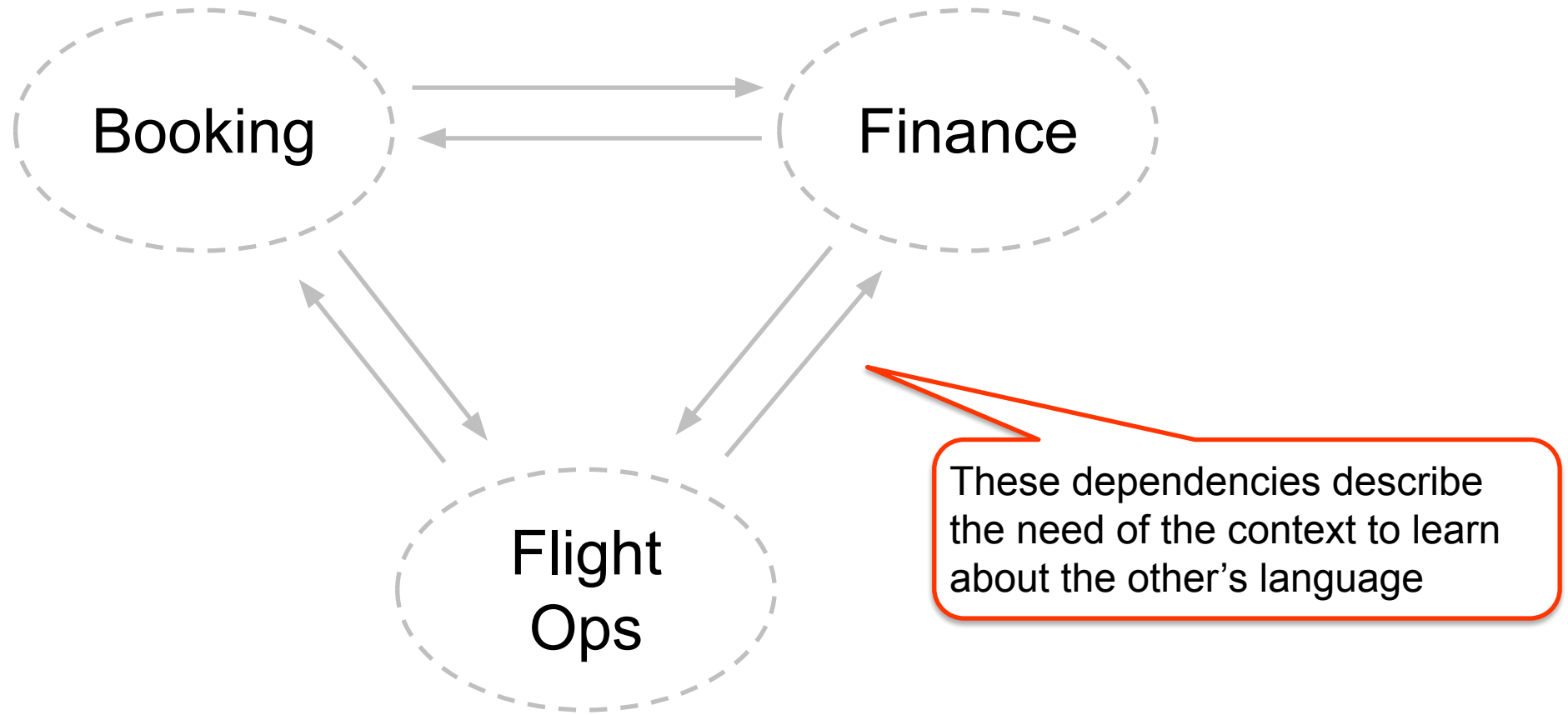
# Complex Transaction Management



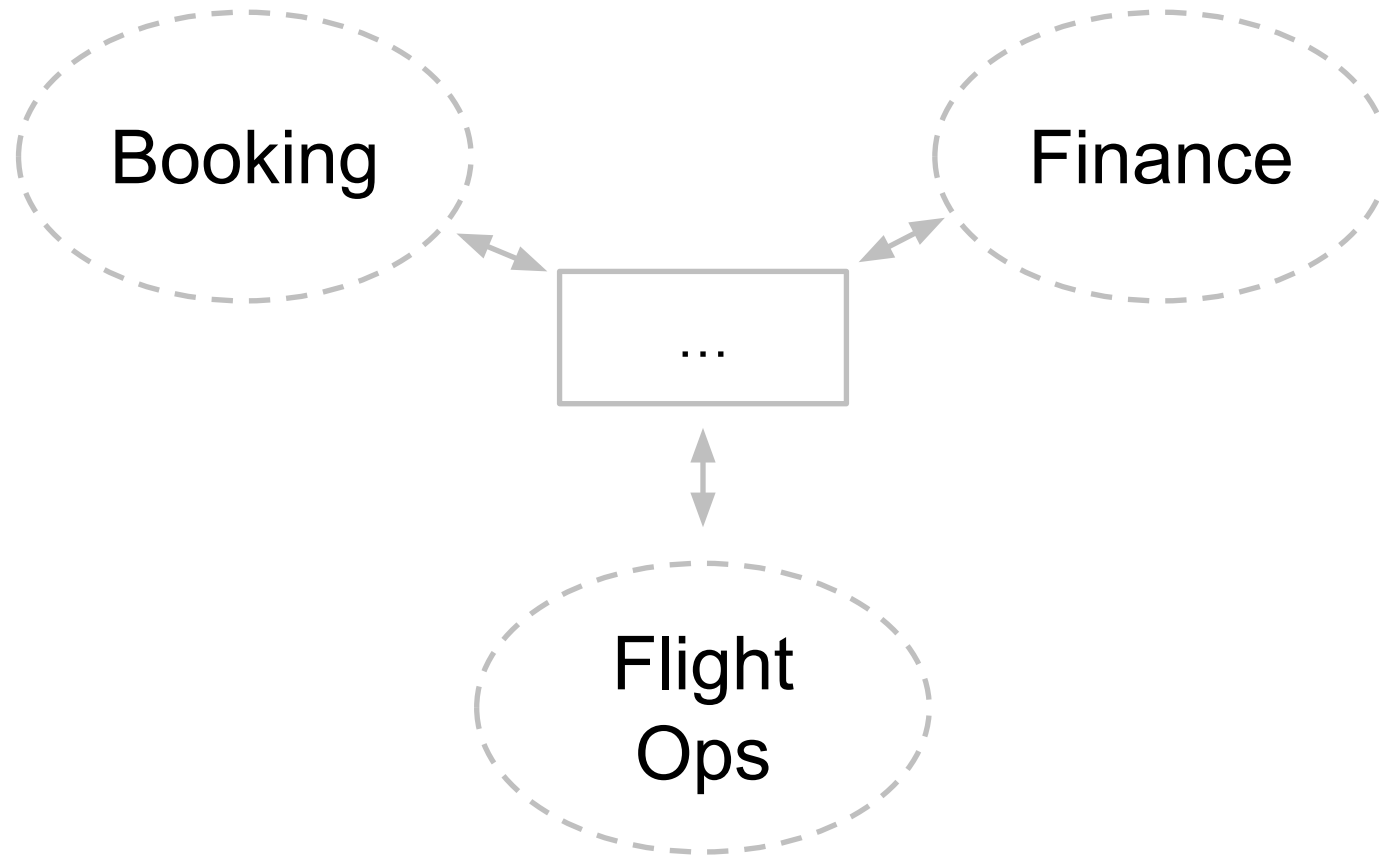
# Transactions

- Not all transactions are atomic
- Business Transactions often have concept of “time” as transaction parameter
- Money transfer
- Finance, Booking and FlightOps

# Cross-Context transaction



# Cross-Context transaction




# Saga

- Coordinates activities between
  - bounded contexts
  - aggregates
- React on Events
- Initiates actions (e.g. by sending commands)
- Maintain state during the transaction



# Sagas in Axon Framework

- Saga Manager
  - Manages instances of a Saga
  - Finds the correct instances for an Event
  - Lifecycle management
- Saga Repository
  - Persists Saga instances
  - Provides access to specific Saga instances
- Saga
  - Manages a single transaction
  - Takes action based on Events



The Saga is the component that implements the actual process

# Saga Implementation Example

```
@Saga
public class BookingExecutionSaga {

    @StartSaga
    @SagaEventHandler(associationProperty = "bookingCode")
    public void handle(BookingMade event) {
        // prepare payment instructions and notify finance
    }

    @SagaEventHandler(associationProperty = "paymentRef")
    public void handle(PaymentConfirmed event) {
        // confirm the booking
    }

    @SagaEventHandler(associationProperty = "flightNr")
    public void handle(FlightDelayed event) {
        // we check the validity of the booking
        // and reschedule where necessary
    }
}
```

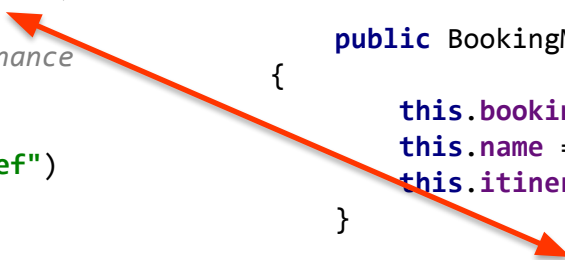
```
public class BookingMade {

    private final String bookingCode;
    private final String name;
    private final Itinerary itinerary;

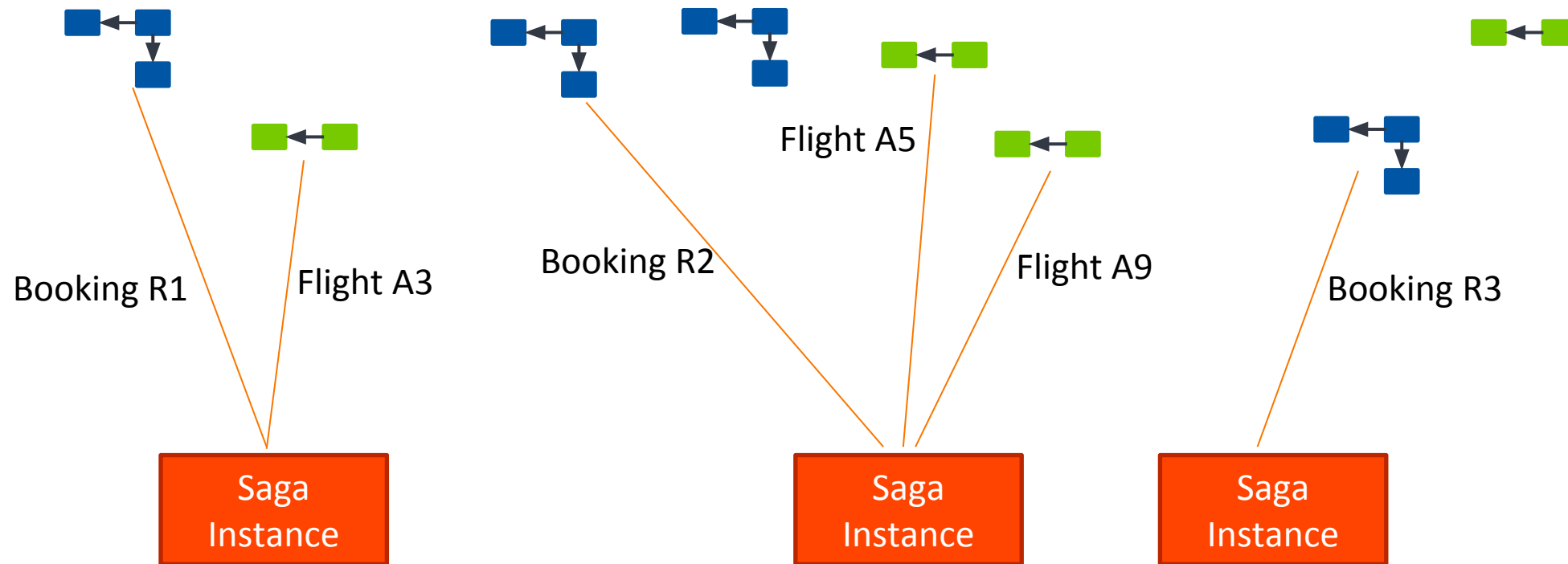
    public BookingMade(String bookingCode, ...)
    {
        this.bookingCode = bookingCode;
        this.name = name;
        this.itinerary = itinerary;
    }

    public String getBookingCode() {
        return bookingCode;
    }

    // more getters
}
```



# Associations



# Managing Lifecycle and Associations

- Lifecycle
  - `@StartSaga`
  - `@EndSaga` or `end()`;
- Associations
  - `associateWith(key, value);`
  - `removeAssociation(key, value);`

`@StartSaga` annotation will automatically create an association for the property mentioned in the `@SagaEventHandler` annotation.

Available as static methods on the `SagaLifecycle` class.

# Resource Injection

```
@Saga // For Spring auto configuration
public class BookingSaga {
    @Inject // or @Autowired
    private transient CommandGateway commandGateway;
    //...
}
```

Note that, although `@Autowired` can be used as annotation for injectable resources, Spring doesn't manage these dependencies.

- You can inject
  - Any components registered with the Configuration API
  - Spring beans (when using Spring Boot AutoConfiguration)
  - Any resource supported by the ResourceInjector passed to the SagaRepository.
- Into fields and annotated “setter” methods

# Handling failure

- A good Saga can deal with unexpected situations
  - Always react to failures on sent commands
  - Concurrency-aware

Note that this doesn't mean Sagas need to be implemented in a Thread-safe manner. They need to be aware that the world is "moving on" while a message is being received.

# Handling failure

- Beware of updating Saga state asynchronously

```
@SagaEventHandler(associationProperty= "bookingCode")
public void handle(BookingMade event) {
    // state changes safe here
    commandGateway.send(new PreparePayment(...))
        .exceptionally(t -> {
            // dealing with exceptions is recommended, but
            // don't change state here!
        });
}
```

Instead, publish Event, schedule activity, or send compensating commands.

# To Saga or Not to Saga

- Ideal terrain for a Saga
  - Relatively straightforward process
  - Requires state throughout
  - Coordinates with more than 1 components
- Alternatives
  - Just an Event Handler
  - BPMN tools for complex processes



Time's up...

# Deadlines

# Deadline Manager

- Used to schedule “deadlines”
- A deadline is an Event Message, targeted at a specific component (Scope)

```
DeadlineManager deadlineManager = ...;
Instant expectedDepartureTime = ...;
deadlineManager.schedule(
    expectedDepartureTime,
    "departure",
    new DepartureDeadline(...)
);

@DeadlineHandler(deadlineName = "departure")
public void on(DepartureDeadline deadline) {
    // React on missed Departure Deadline
}
```

# Deadline Support

- Sagas
  - Typically timeouts in process steps
  - Can trigger state changes
  - Can trigger side-effects / commands
- Aggregates
  - Typically timeout waiting for input (commands)
  - May apply events (when Event Sourcing) or change state (state-stored aggregates)
  - Deadlines are not “sourced”
- Custom defined Scope

# Testing with Deadlines

```
// Expect a deadline to trigger (Saga)
FixtureConfiguration fixture = new SagaTestFixture(FlightSchedule.class);
fixture.givenAggregate(flightId).published(new DepartureScheduled(...))
    .whenTimeAdvancesTo(departureTime)
    .expectDispatchedCommands(new SendReminderCommand(...));
```

---

```
// Expect a deadline to trigger (Aggregate)
FixtureConfiguration fixture = new AggregateTestFixture(Flight.class);
fixture.given(new DepartureScheduled(...))
    .whenTimeAdvancesTo(departureTime)
    .expect(new DepartureSlotMissedEvent(...));
```

---

# Configuring DeadlineManager

- SimpleDeadlineManager
  - Uses ScheduledExecutorService
  - Keeps deadlines in memory
- QuartzDeadlineManager
  - Uses Quartz for scheduling
  - Can keep deadlines in database
- Other implementations on roadmap

# Accessing the Deadline Manager

- Saga
  - Inject as Resource
  - Add as parameter to @SagaEventHandler method
- Aggregate
  - Add as parameter to @CommandHandler method

```
@CommandHandler
public void handle(BookFlight command, DeadlineManager deadlineManager) {
    deadlineManager.schedule(
        command.getFirstDeparture().minus(24, HOURS), "validatePayment"
    );

    AggregateLifecycle.apply(new FlightBooked(...));
}
```

Whatever else you wanted to know...

# Questions