Introduction to Axon Framework

Agenda

- Architecture overview
- Command Handling & Aggregate design
- Event Processing
- Query Models
- Event Sourcing
- Advanced Transaction Management (Sagas)
- Task Based UI

Architecture Overview

CQRS & DDD - Terminology

The components that make up the models

Domain

“

A sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software.

”

Model

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

Entity

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

Value Object

“Value objects have *no conceptual identity*, but are fundamentally *defined by their attributes*.
They describe some characteristic of a thing.

Value Objects are Immutable”

Repository

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

Aggregate

“

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

”

Aggregate

> “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”

Event

“

A notification that something relevant has happened inside the domain

”

Command

“

An expression of intent to trigger an action in the domain

”

Query

“

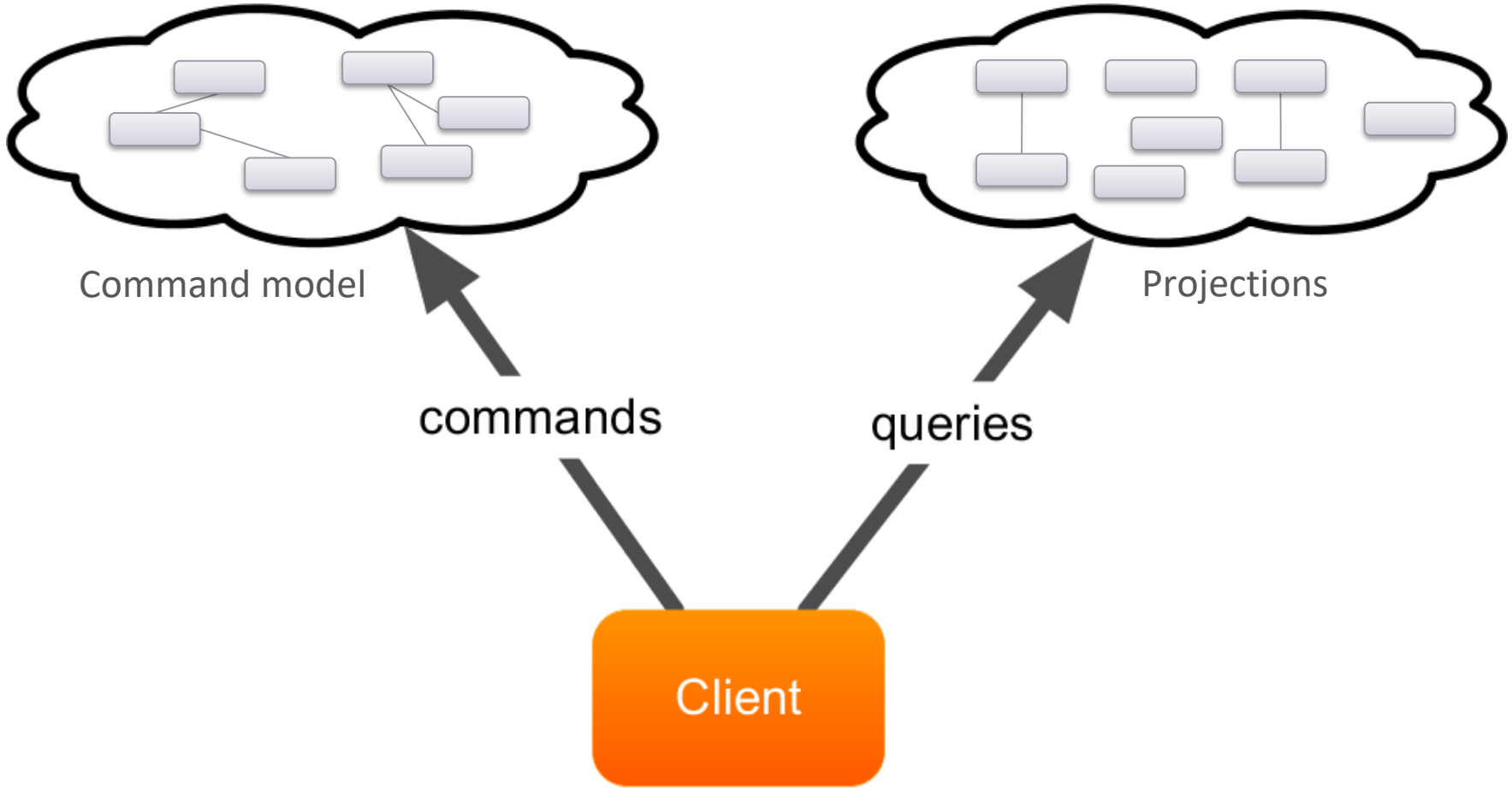
A request for information or state

”

CQRS Architecture

Anatomy of a CQRS based application

CQRS Architecture



Command Model

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

Query Model / Projections

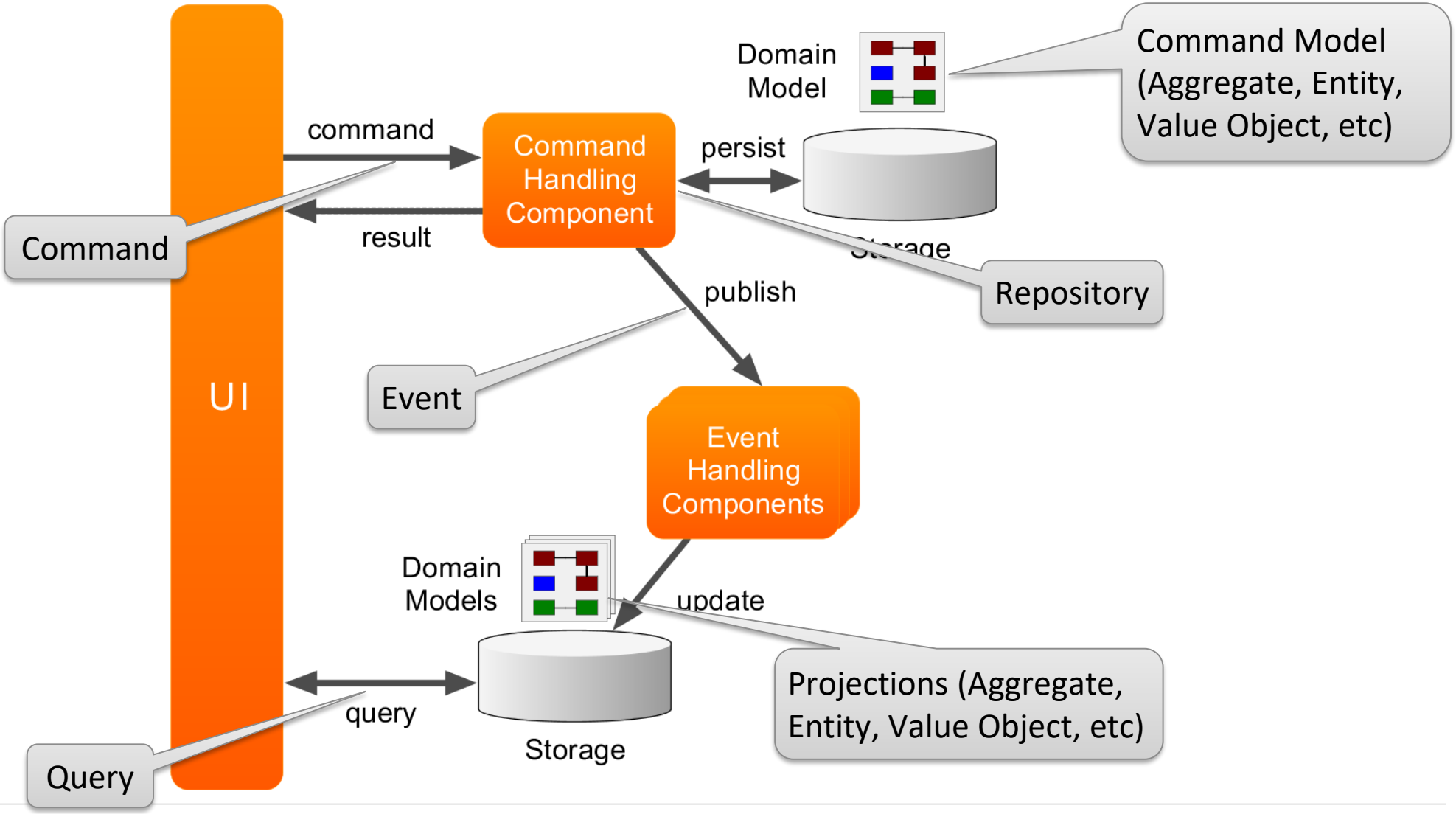
- Focused on delivering information
- Data is stored the way it is used
- Denormalized
- “table-per-view”

Synchronization of models

- Changes in the Command Model should (eventually) be visible in the Query Model

- Shared data source
- Stored procedures
- Event Driven Architecture

CQRS, DDD & EDA


UI
command
result
Command
Command Handling Component
persist
Domain Model
Command Model (Aggregate, Entity, Value Object, etc)
Repository
publish
Event
Event Handling Components
Domain Models
update
Projections (Aggregate, Entity, Value Object, etc)
query
Storage
Query

Location Transparency

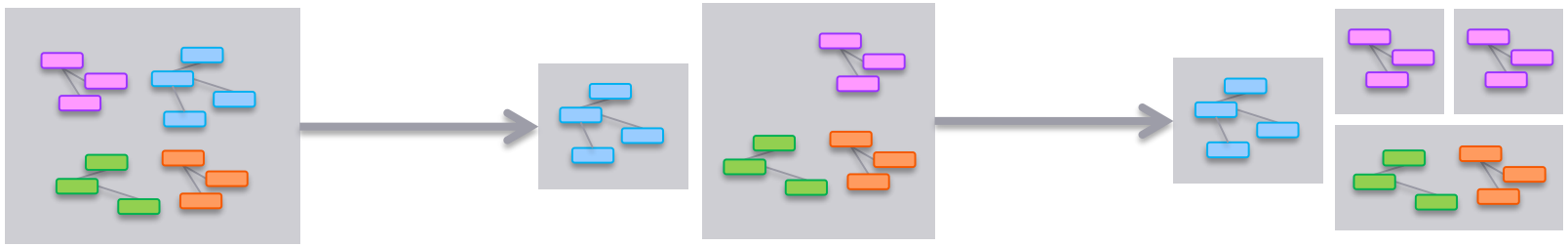
- A Component should not be aware, nor make any assumptions, of the physical location of Components it interacts with
- Beware of APIs & method signatures:
  - Not location transparent:

```
public Result doSomething(Request request) {…}
```

  - Location transparent alternatives:

```
public void doSomething(Request request, Callback<Response> callback) {…}
public CompletableFuture<Result> doSomething(Request request) {…}
```

Location Transparency – Microservice Architecture



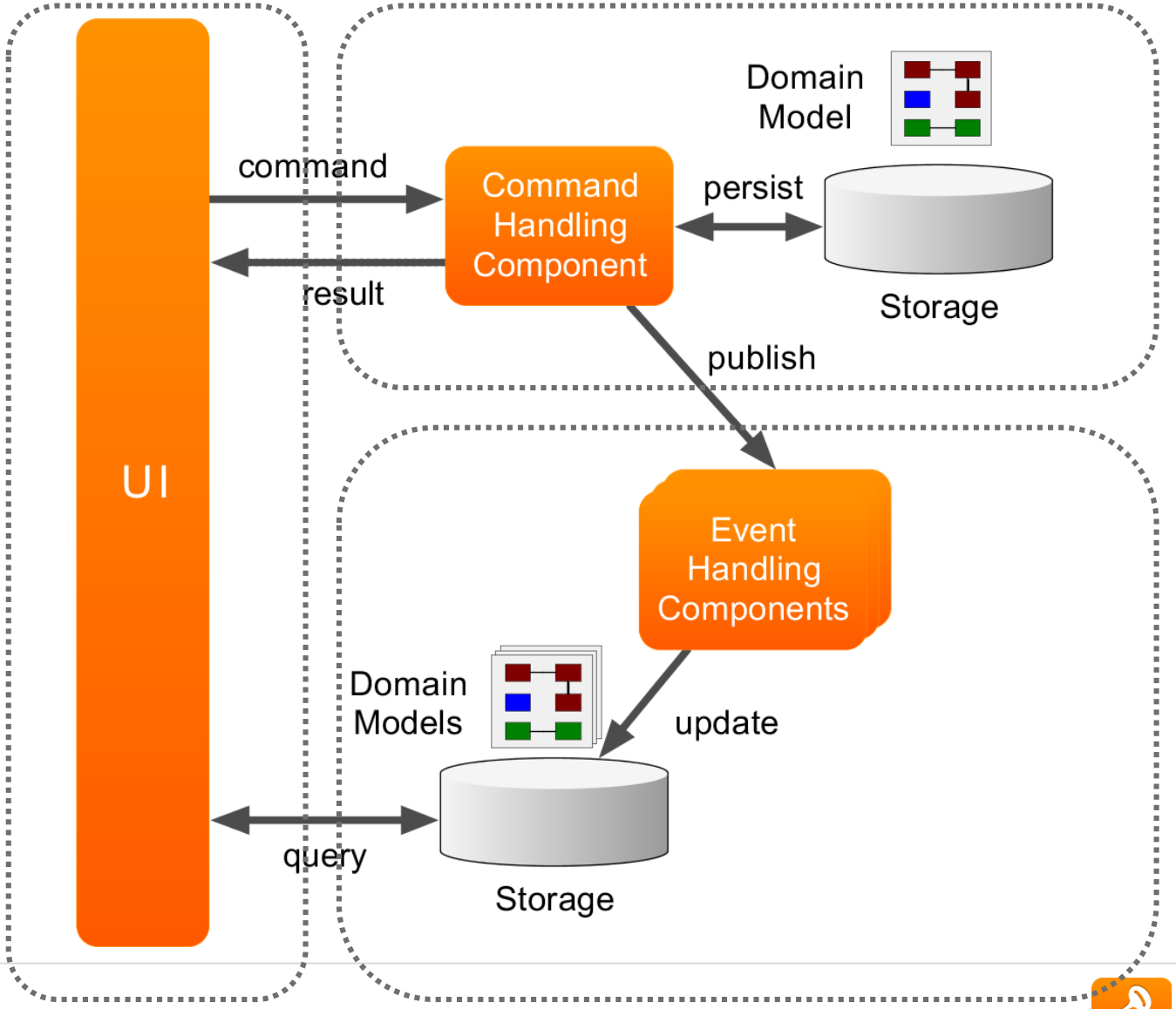
1 – Build a monolith

2 – Extract components

3 – Extract more

4 – Scale as needed

Location Transparency boundaries

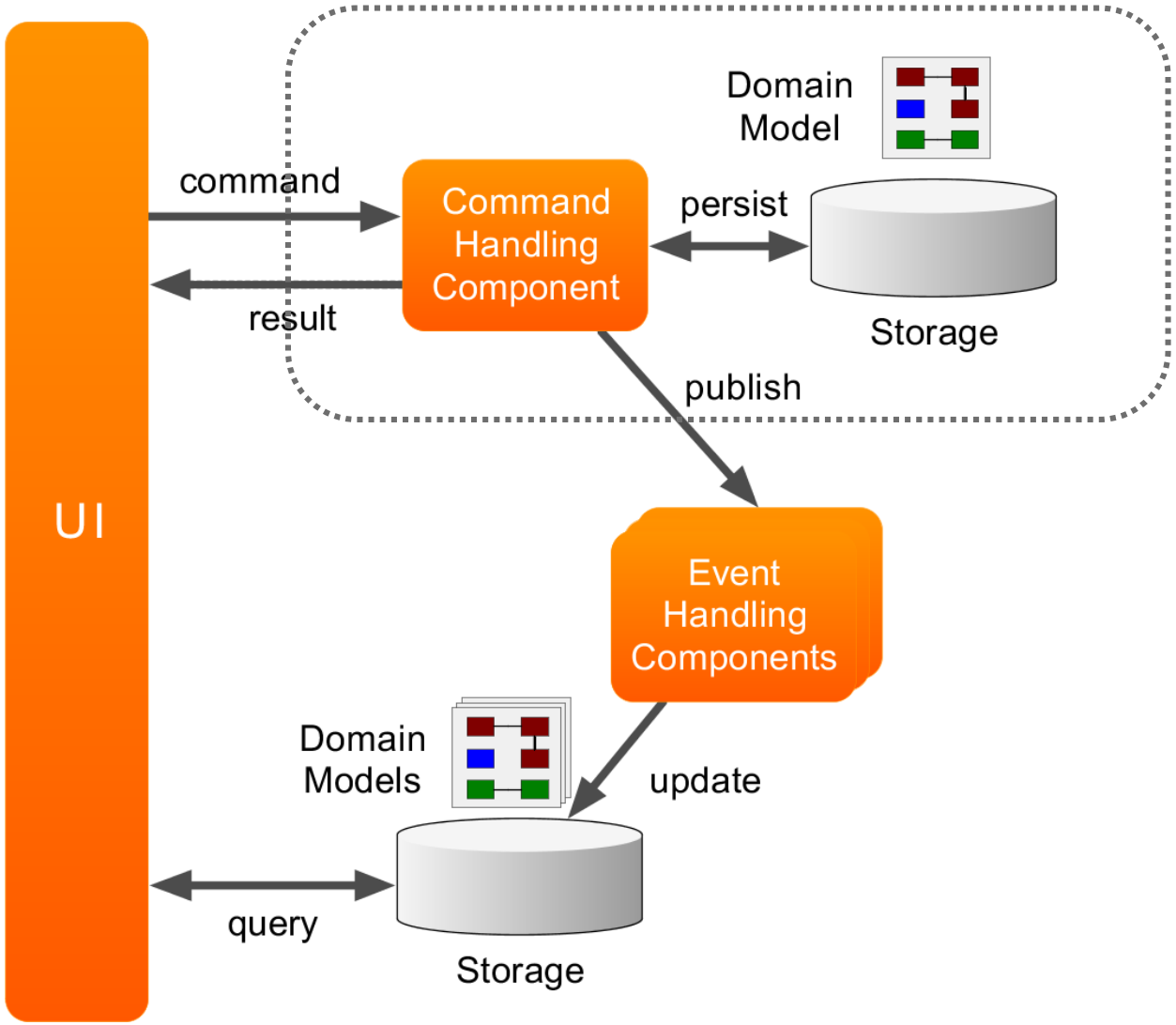


Lab 1

Set up the Application’s foundation

Command Handling & Aggregate Design

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 {

  @Autowired
  private Repository<MyCommandModel> repository;

  @CommandHandler
  public void handle(SomeCommand cmd) {
    repository.load(cmd.getId())
              .execute(ar -> ar.doSomething());
  }
}
```

Makes an instance available in the Application Context (Spring)

Returns an Aggregate<MyCommandModel>

Command Model (with Spring)

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

public class MyCommandModel {

  private String id;
  // other state

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

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

Annotated Command Model (with Spring)

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

@Aggregate
public class MyCommandModel {

  @AggregateIdentifier
  private final String id;

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

Tells Axon Spring Autoconfiguration to set up necessary infrastructure

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

Registers this method as a Command Handler for "SomeCommand"

Command Message routing

```
public class SomeCommand {

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

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.

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

Routing Command to Entities within Aggregate

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

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

Routing Command to Entities within 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 = new DefaultCommandGateway(commandBus);
// 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);
```

Advanced Command Handling features

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 MetaData 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

Unit of Work

- Coordinate lifecycle of message handling
  - start → prepare commit → commit → after commit → cleanup
    → rollback → cleanup
- Register for resources used during processing
  - e.g. Database connections
- Correlation data management
  - Correlation data automatically attached to generated messages

Intercepting Commands

- Command Dispatch Interceptors
  - Invoked in the thread that dispatches Command
  - Active Unit of Work is that of incoming message (if any)
  - Allows transformation of Command Message or force failure
- Command Handler Interceptors
  - Invoked in thread that handles Command
  - Active Unit of Work is that of intercepted message
  - Can force early return / failure

Intercepting Commands – Use Cases

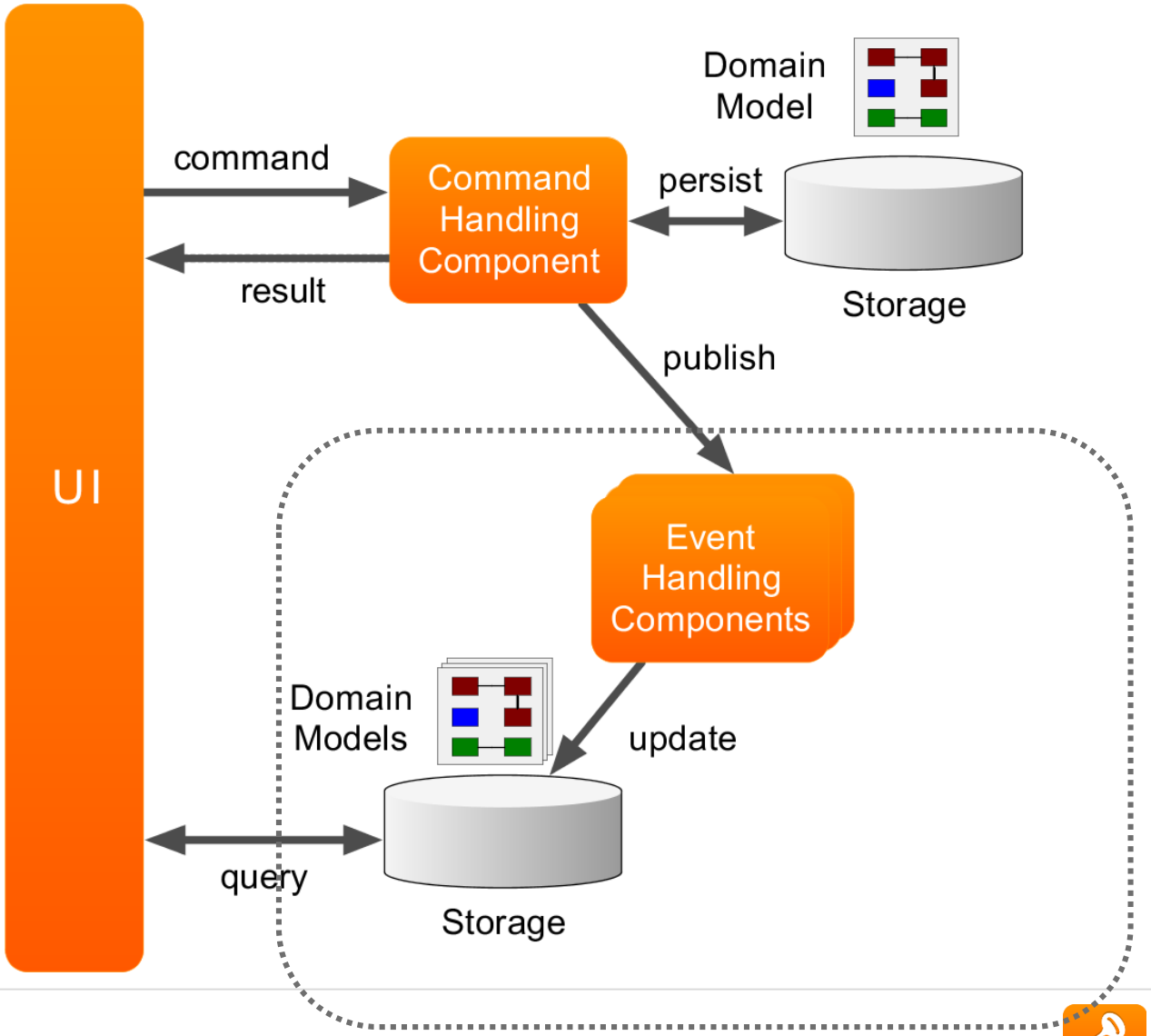
- Command Handler Interceptors
  - Attach (database) transaction
  - Validate security meta data
- Command Dispatch Interceptors
  - Structural validation
  - Attach security meta-data

Lab 2

Command Handling

Event Processing

Event Handling



Event Handler

- Handles published events
  - Projections
  - Trigger (external) activities
  - Manage complex transactions (Sagas)

Event Handling in Axon Framework

- Component that is subscribed to the Event Bus to handle specific Events
- `@EventHandler`
  - On (singleton) component

```
@EventHandler
public void handle(MyEvent event) {
        …
}
```

Event Handling Component (with Spring)

```
@Component
public class EventHandlingComponent {

  @EventHandler
  public void handle(SomeEvent event) {
    // do what you need to do
  }
}
```

Organizing Event Handlers

- Event Processor
  - Responsible for managing the technical aspect of processing an Event
  - Starts and Commits Unit of Work
  - Invokes handler methods

- Each handler is assigned to a single Processor
  - @ProcessingGroup on Event Handler class
  - Assignment rules in EventHandlingConfiguration (part of Configuration API)

Event Processors

- SubscribingEventProcessor (default)
  - Receives messages as they are published, in the thread that publishes the messages
  - Requires a subscribable message source
- TrackingEventProcessor
  - Uses its own thread(s) to read EventMessages from a Stream
  - Requires a streamable message source
  - Records progress using TrackingToken

Event Handler Parameters

- Supported parameter types
  - First parameter (if none of below) resolves to Message payload
  - Message → *Resolves to entire message*
  - EventMessage → *Resolves to EventMessage*
  - UnitOfWork → *Resolves to the current Unit of Work*
  - MetaData → *Resolves to the MetaData 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

Intercepting Events

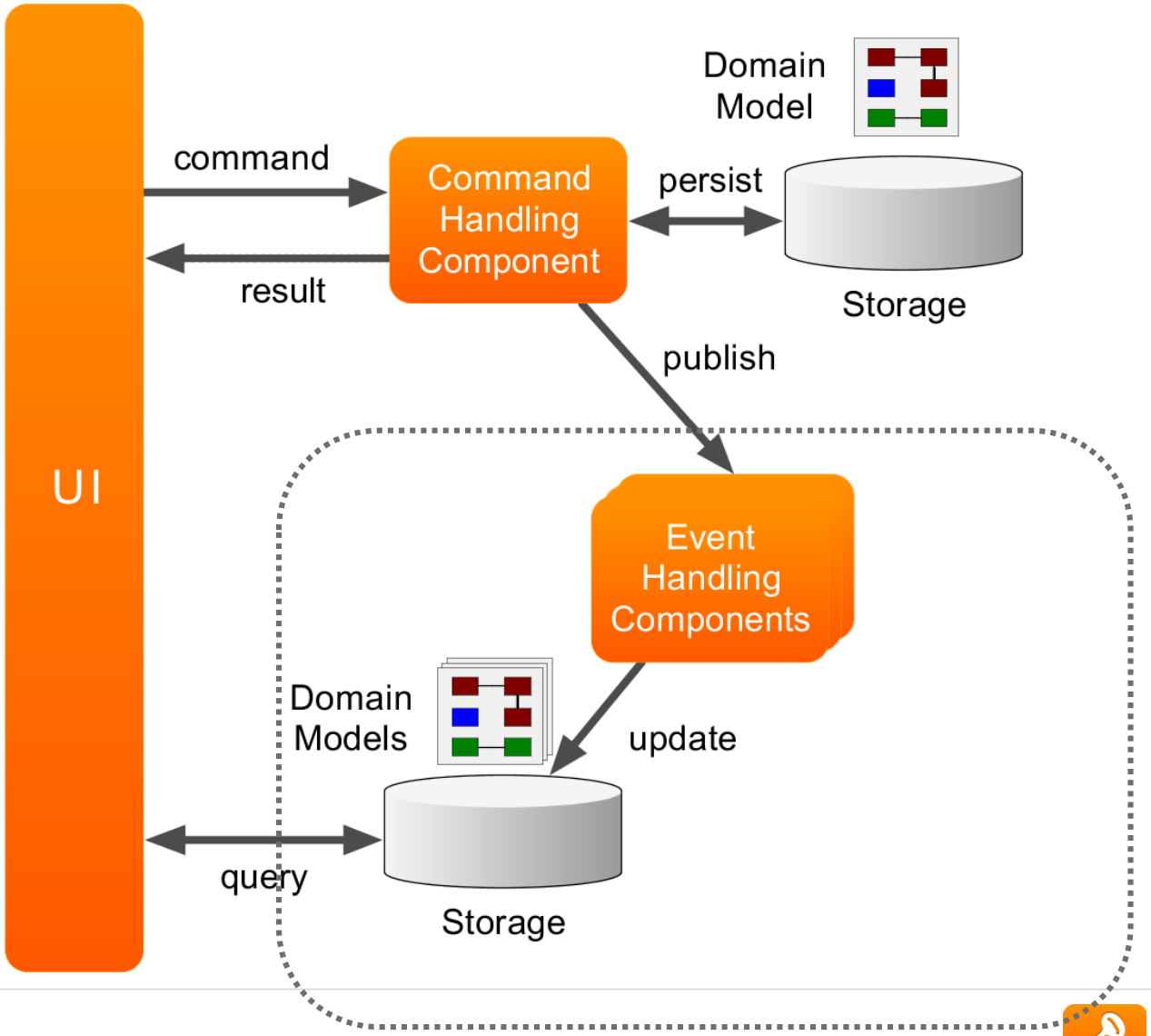
- Dispatch Interceptors
  - Defined on Event Bus
  - Invoked in the thread that dispatches Event
- Handler Interceptors
  - Defined on Event Processor
  - Invoked in thread that handles Events

Lab 3

Event Processing

Query Models

Event Handling



Query Model

- Model optimized to answer queries
  - Focused on data
  - Denormalized to suit information needs (e.g. table per view)
  - Updated by Event Handling component

- Consciously optimize for
  - Performance
  - Storage
  - Flexibility

Query Database Denormalization

- Optimize the query database (i.e. the query model) for your UI

OrderHeader Table

CustId	OrderId	CustomerName	Address	Total amount
12	56	John Doe	Amsterdam	€ 38,00
12	57	John Doe	Amsterdam	€ 85,00
13	58	Sjonnie	Den Haag	€ 12,00

Optimized for specific Use Case

- Optimized for full-data retrieval based on ID
  - Give all order details for Order 123

OrderDetails Table

OrderId	OrderData
56	{“customer”:”John Doe”, “orderItems” : [{“itemId”: “123, “ite…
57	{“customer”:”John Doe”, “orderItems” : [{“itemId”: “456, “ite…
58	{“customer”:”Sjonnie”, “orderItems” : [{“itemId”: “789, “itemN…

Storage technology selection

- Use the storage that fits the method of access
  - Generic Query → Relational DataBase
  - Relationships → Graph Database
  - Full-text search → Search Engine
  - Etc.

- Do not create a single model that can answer all queries.

  It will answer none efficiently.

- Do not fear (data) duplication

Query API

- Query Bus
- Query Gateway
- Query Handler
  - `@QueryHandler`

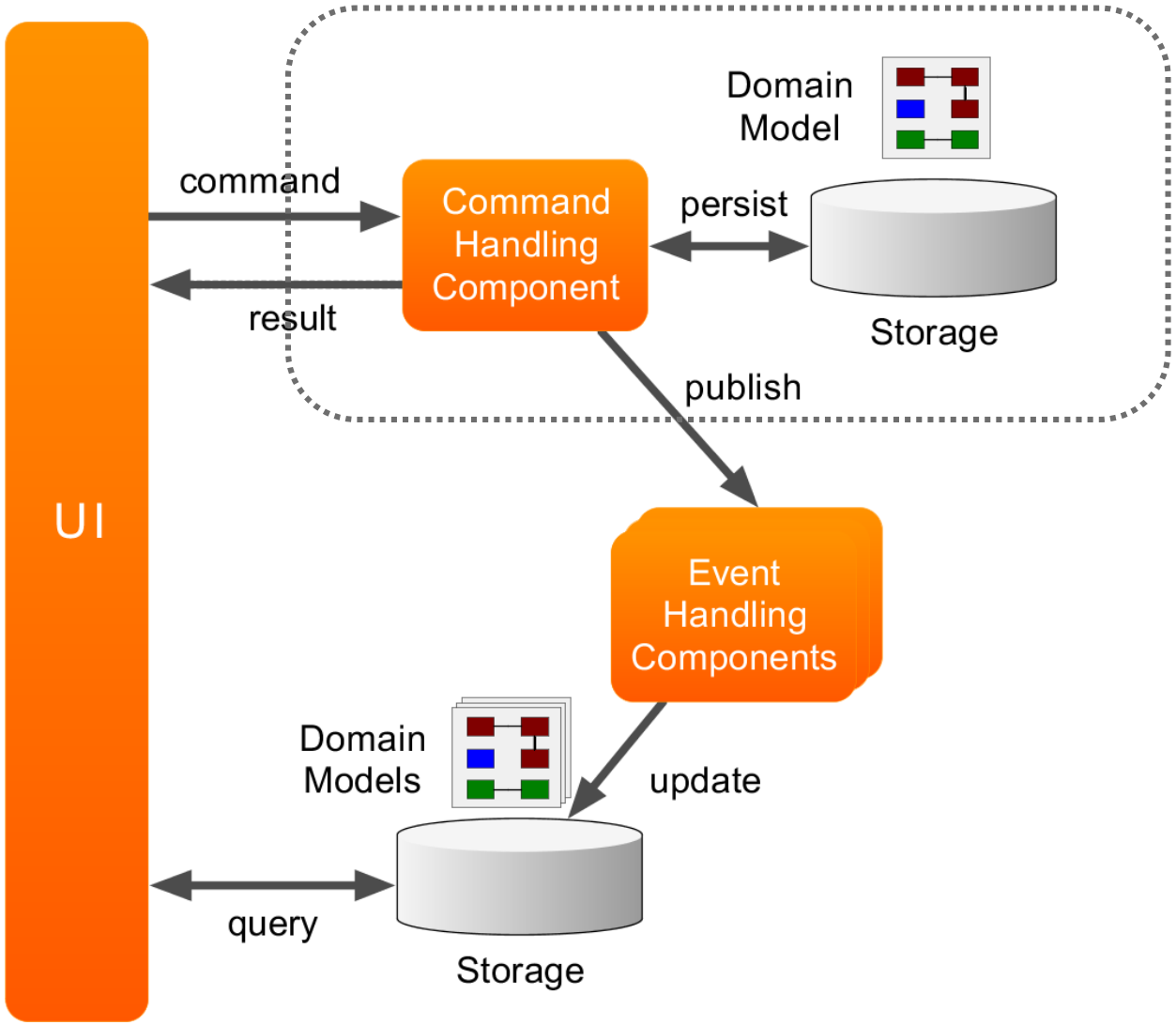
```
@Component
public class QueryHandlingComponent {

  @QueryHandler
  public SomeResponse handle(SomeQuery event) {
    // find that data and return it
  }
}
```

Lab 4

Query Models

Event Sourcing

Event Sourcing



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 Store

- Responsible for storage of events
- Primary focus: writing (append)
- Relational database is good start for most applications
- Axon has implementation that supports JPA, JDBC and MongoDB

Event Sourcing as Business Case

- Event Sourcing has less information loss
  - Event Store contains information that can be used in different ways in the future
- Event Store is a reliable audit log
  - Not only state, but also how it is reached.
- Event Sourcing increases performance
  - Only deltas need to be stored. Caches prevent reads.

Event Sourcing

State storage

Order

id: 123

items

- 1x Deluxe Chair - € 399

status: return shipment rcvd

Event Sourcing

OrderCreatedEvent

- id: 123

ItemAddedEvent

- 2x Deluxe Chair - € 399

ItemRemovedEvent

- 1x Deluxe Chair - € 399

OrderConfirmed

OrderShipped

OrderCancelledByUserEvent

ReturnShipmentReceived

Event Sourcing or not…

- Data staleness…
  - User makes a change. A big one.
  - System: Sorry, someone else has made a change too. Try again…

- With Event Sourcing, the system knows what the other user did, and can try to merge the changes

Event Sourcing or not…

- Audit trail…
  - Developer 1: The order state is “Paid”, but the payment never came through.
  - Developer 2: Let’s scan through the log files
  - Developer 1: Useless, the log level has to be “info” for us to see anything useful

- An Event Store stores everything *

** If you have a properly designed Domain model*

Event Sourcing or not…

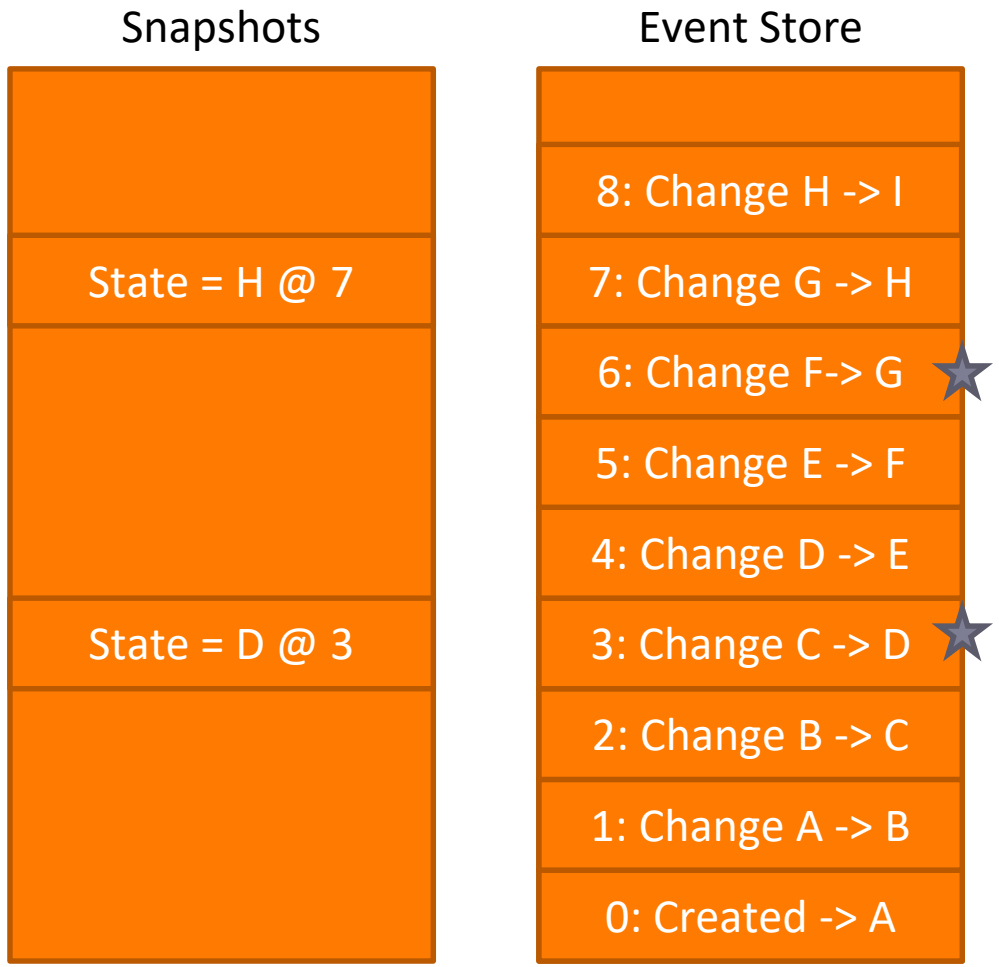
- Reporting…
  - Manager: I need to know on which day of the week most orders are accepted
  - Developer: we're not recording that right now. We'll build it now, deploy it in 2 months, and you'll have reliable reports 3 months after.

- ES: Build a component and replay old events on it.

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?

Snapshotting



Snapshotting

- Snapshots are a (temporary) replacement for a set of historical events

- Snapshotting may be an asynchronous process
  - Regular intervals
  - After x events
  - When loading takes >= x ms

Event Sourcing – Entity layout

```
@Aggregate
public class Order {

    // fields containing state

    @CommandHandler
    public void handle(ConfirmOrderCommand cmd) {
        assertNotEmpty(items);
        // some more assertions
        apply(new OrderConfirmedEvent(cmd.getOrderId()));
    }

    @EventSourcingHandler
    public void on(OrderConfirmedEvent event) {
        this.status = Status.CONFIRMED;
    }
```

Decision making

State changes

Applying Events

- `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

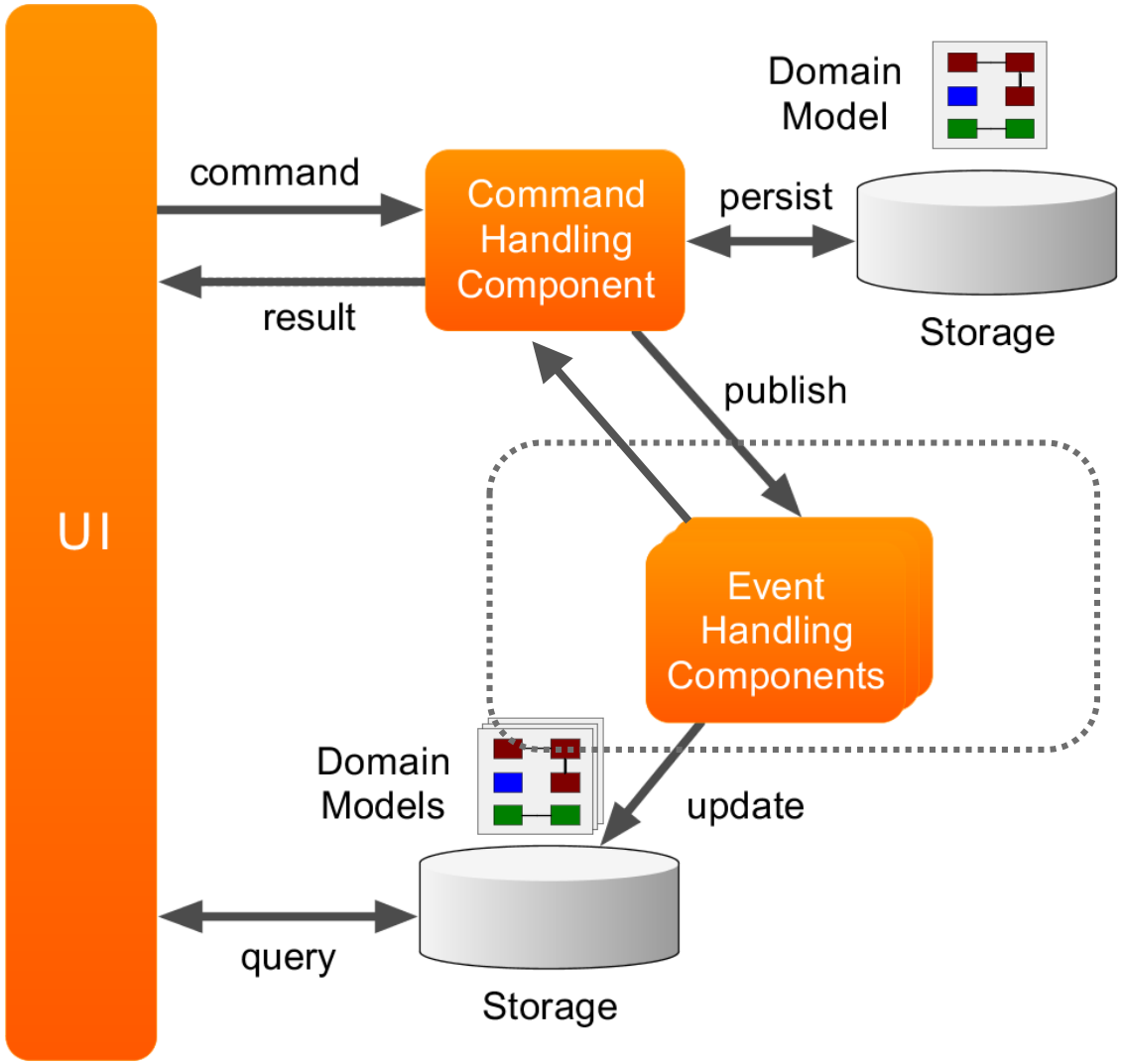
- Note: Other aggregates will *not* receive the event

Lab 5

Event Sourcing

Complex Transaction Management

Complex Transaction Management



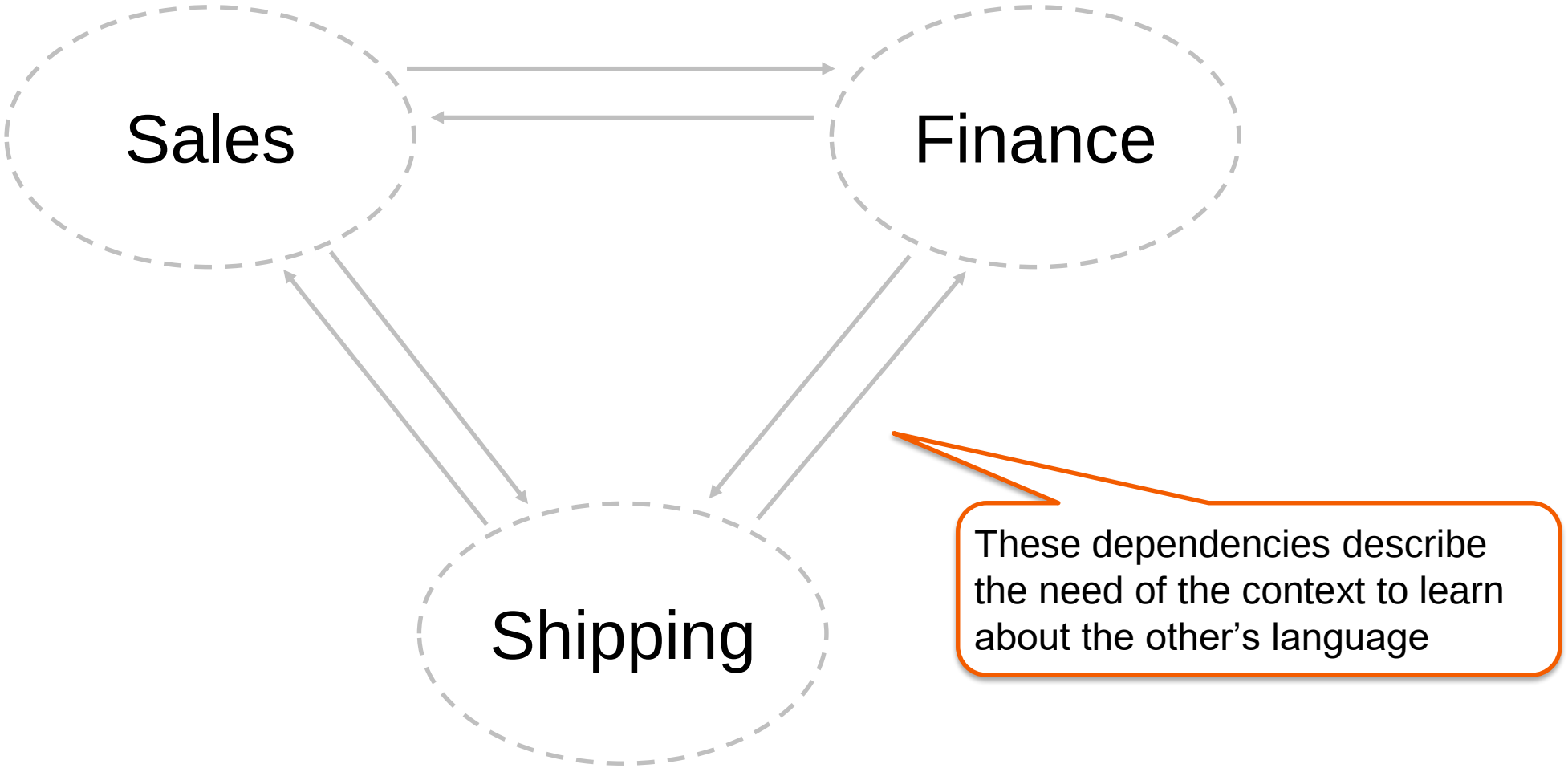
Transactions

- Not all transactions are atomic

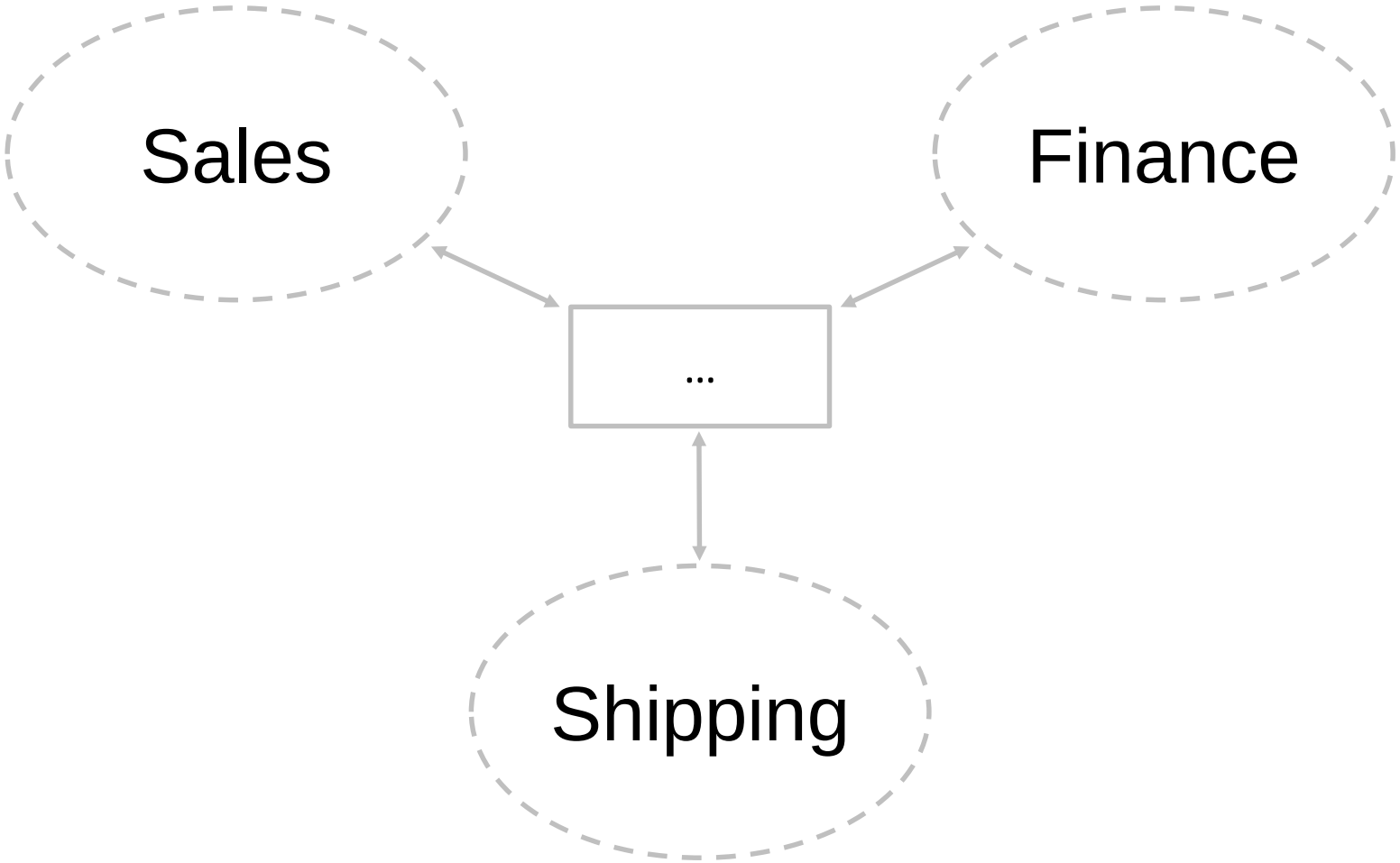
- Business Transactions often have concept of “time” as transaction parameter

- Money transfer
- Sales, shipping and finance

Cross-Context transaction



Cross-Context transaction

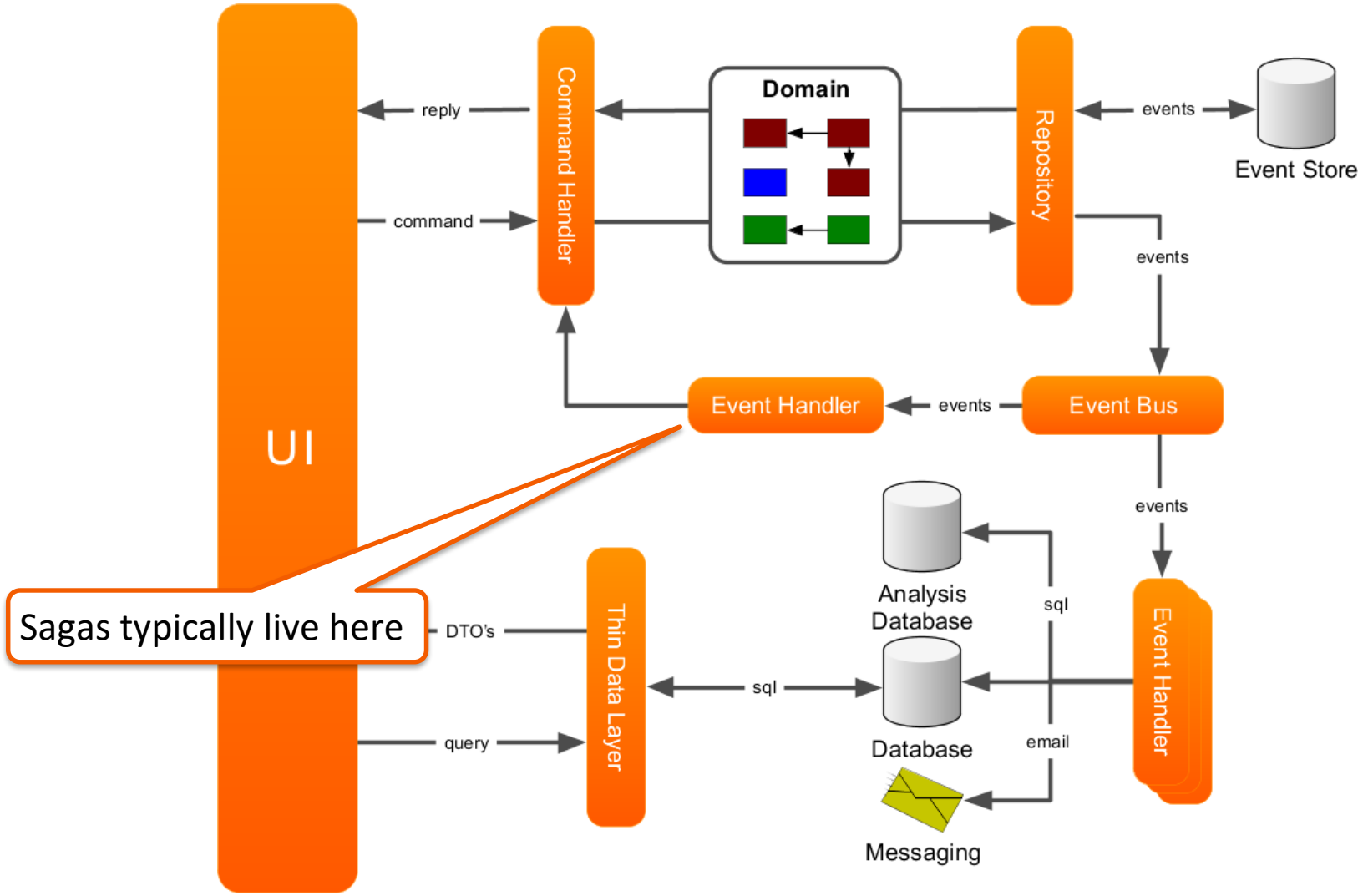


Saga

- Coordinates activities between
  - bounded contexts
  - aggregates

- React on Events
- Initiates actions (e.g. by sending commands)
- Maintain state during the transaction

CQRS Overview



Sagas in Axon Framework

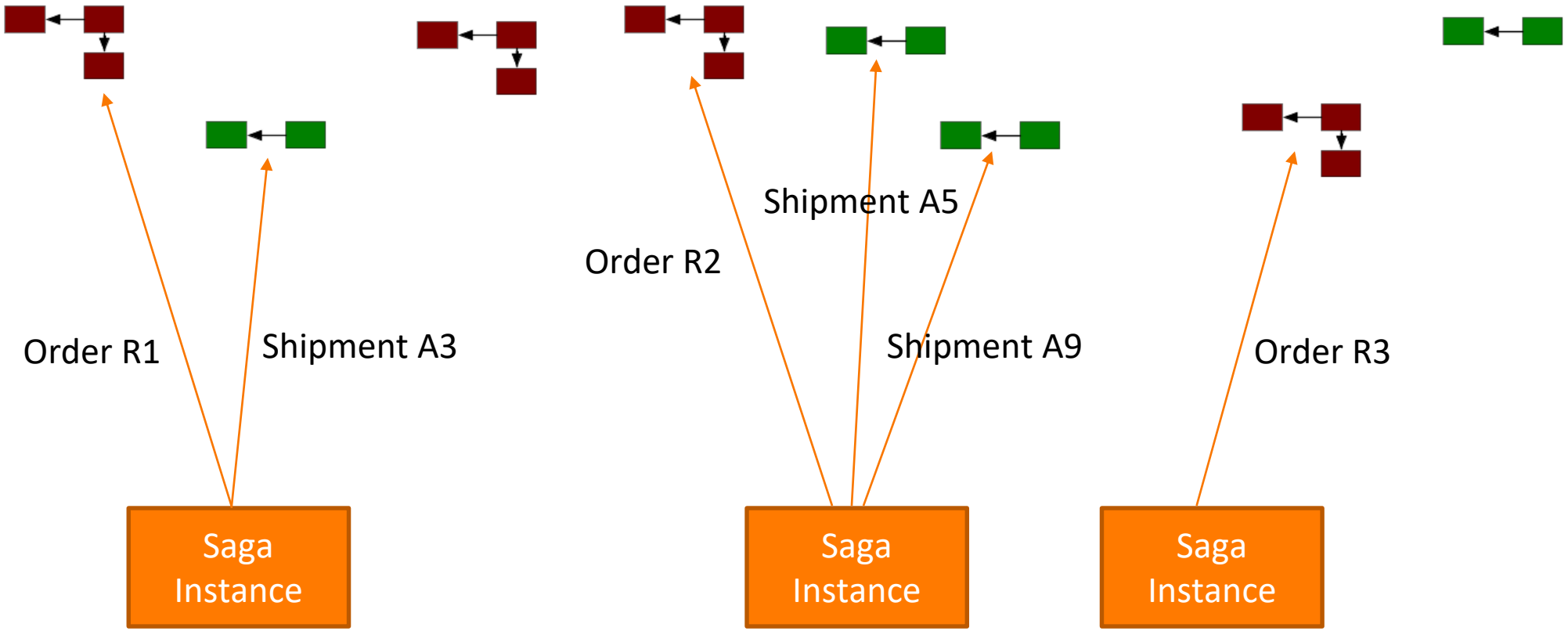
- Saga Manager
  - Manages instances of a Saga
  - Finds the correct instances for an Event
  - Lifecycle management
- Saga Repository
  - Persists Saga instances
  - Provides access to specific Saga instances
- Saga
  - Manages a single transaction
  - Takes action based on Events

The Saga is the component that implements the actual process

Saga Implementation Example

```java
@Saga
public class OrderHandlingSaga {

    /* State required to process events */

    @StartSaga
    @SagaEventHandler(associationProperty = "orderId")
    public void handle(OrderPlacedEvent event) {
        // notify invoice and shipping
    }

    @SagaEventHandler(associationProperty = "invoiceId")
    public void handle(InvoicePaidEvent event) {
        // continue process
    }

    @EndSaga
    @SagaEventHandler(associationProperty = "shipmentId")
    public void handle(ShipmentDeliveredEvent event) {
        // some last touches
    }
}
```

Associations



Saga Implementation Example

```java
@Saga
public class OrderHandlingSaga {

    /* State required to process events */

    @StartSaga
    @SagaEventHandler(associationProperty = "orderId")
    public void handle(OrderPlacedEvent event) {
        // notify invoice and shipping
    }

    @SagaEventHandler(associationProperty = "invoiceId")
    public void handle(InvoicePaidEvent event) {
        // continue process
    }

    @EndSaga
    @SagaEventHandler(associationProperty = "shipmentId")
    public void handle(ShipmentDeliveredEvent event) {
        // some last touches
    }
}
```

```java
public class OrderPlacedEvent {

    private final String orderId;
    private final String customerId;
    private final Map<String, Integer> itemSkuAndCount;

    public OrderPlacedEvent(String orderId,
                            String customerId,
                            Map<String, Integer> itemSkuAndCount) {
        this.orderId = orderId;
        this.customerId = customerId;
        this.itemSkuAndCount = itemSkuAndCount;
    }

    public String getOrderId() {
        return orderId;
    }

    // more getters
}
```

Managing Lifecycle and Associations

- Lifecycle
  - `@StartSaga`
  - `@EndSaga or end();`

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

- Associations
  - `associateWith(key, value);`
  - `removeAssociation(key, value);`

Available as static methods on the `SagaLifecycle` class.

Deadlines

- If an invoice isn't paid within 30 days, send a reminder.

```
FixtureConfiguration fixture = new SagaTestFixture(MySaga.class);
fixture .givenAggregate(invoiceId).published(new InvoiceSentEvent(...))
        .whenTimeElapses(Duration.standardDays(30))
        .expectDispatchedCommands(new SendReminderCommand(...));
```

```
eventScheduler.schedule(Duration.standardDays(30),
                        new InvoicePaymentDeadlineExpiredEvent(...));
```

```
@SagaEventHandler(associationProperty = "invoiceId")
public void handle(InvoicePaymentDeadlineExpiredEvent event) {
   commandGateway.send(new SendReminderCommand(...);
}
```

Resource Injection

```
@Saga // for Spring autoconfiguration
public class OrderDeliverySaga {
  @Inject // or @Autowired
  private 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(associationValue="orderId")
public void handle(OrderPlacedEvent event) {
    // state changes safe here
    commandGateway.send(new CreateInvoiceCommand(…))
                .exceptionally(t -> {
                        // dealing with exceptions is recommended, but
                        // don't change state here!
                });
}
```

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

Lab 6

Saga

For support, info and resources

- Web: axoniq.io / axonframework.org
- Twitter: @axon_iq
- LinkedIn: AxonIQ https://www.linkedin.com/company/18225081/
- Email: allard@axoniq.io