Advanced Features of Axon Framework

Agenda – Day 2

- Event Processors & Replays
- Refactoring and evolving your application
  - Evolving Commands and Events
  - Upcasting
- Building Microservices with Axon
- Monitoring, measuring throughput & latency, message tracing
- Advanced tuning

CQRS & DDD Fundamentals

Terminology recap

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…

> 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

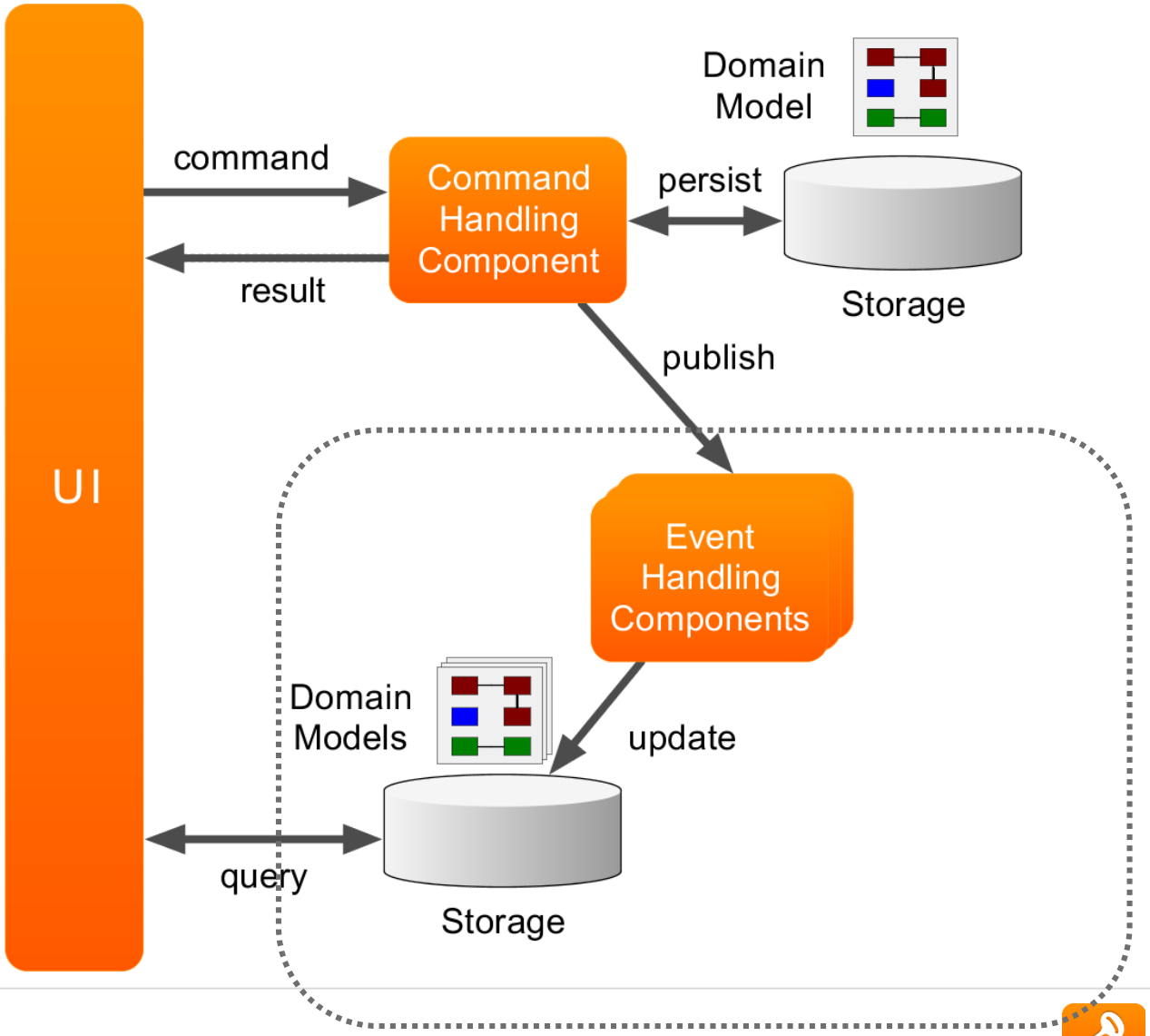
“

A request for information or state

”

Event Processors & Replays

Event Handling



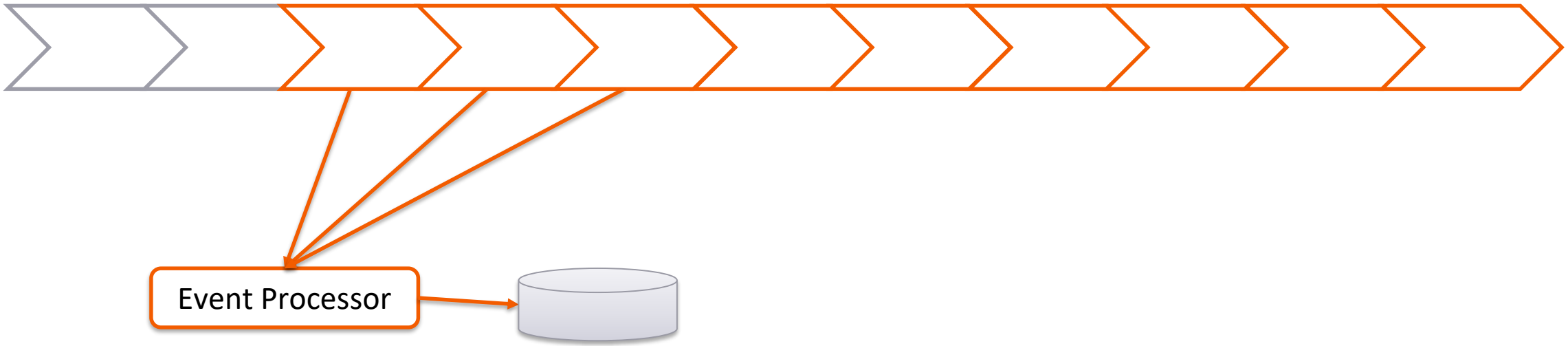
Organizing Event Handlers

- 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
  - Saves progress using TrackingToken

Tracking Token



Tracking Token – Segmentation

- Multi-threading and/or multi-node
- Each thread “claims” a segment
- SequencingPolicy defines segment
  - the same value for two messages means they ‘belong’ to same segment
  - Message in same segment are always handles sequentially
  - E.g. `SequentialPerAggregatePolicy`

Note: This feature is planned for Axon 3.1

Event Processor Configuration

```
public void configure(EventHandlingConfiguration config) {
    StreamableMessageSource<TrackedEventMessage<?>> source = …;
    config.registerTrackingProcessor("com.example.viewmodel", c -> source);
}
```

The name of the processor to explicitly register.
It is only created when handlers are actually assigned to it.

A function returning the source to use, given Configuration c.

Replays

- Concept of replay “disappeared” in Axon 3

- Tracking Processors can be “reset”
  - Clean up any state their handlers have
  - Remove all tokens for that processor

Lab 7

Event Processors & Replays

Refactoring & Application Evolution

Schemas and Message Versioning

- Your Command & Event format are a contract
  - Implicit vs explicit schema
- Axon supports “schema revisions”
  - Maven version
  - Sequential revision
  - Any arbitrary (String) value

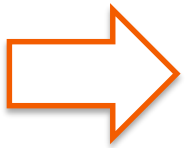
Naïve approach: class per version

```
public class ProductPurchasedEvent_v1 {

  private String myDomainObjectId;
  private String productId;
  private BigDecimal price;

}
```

```
public class ProductPurchasedEvent_v2 {

  private String myDomainObjectId;
  private List<Product> products;

}

public class Product {

  private String productId;
  private BigDecimal price;
}
```

Last representation only

```
public class ProductPurchasedEvent {

  private String myDomainObjectId;
  private String productId;
  private BigDecimal price;

}
```



```
public class ProductPurchasedEvent {

  private String myDomainObjectId;
  private List<Product> products;

}

public class Product {

  private String productId;
  private BigDecimal price;
}
```

Upcasters

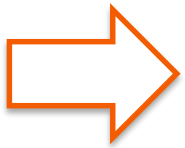
- Upcasters transform old event representations to the newer format
  - One format, one revision per upcaster
  - Chain upcasters into an “Upcaster Chain”

- Upcasters work on an intermediate representation
  - e.g. xml, json

Last representation only – Upcasters

ProductPurchasedEvent, revision ‘0’

```
{
  “orderId” : “1234”,
  “productId” : “abcd”,
  “price” : “10.23”;
}
```



ProductPurchasedEvent, revision ‘1’

```
{
  “orderId” : “1234”,
  “products” : [
    {
      “productId” : “abcd”,
      “price” : “10.23”
    }
  ]
}
```

Upcaster API

```
interface Upcaster<T> {

  Stream<T> upcast(Stream<T> intermediateRepresentations);

}


interface EventUpcaster extends
                        Upcaster<IntermediateEventRepresentation> {

}
```

Upcaster API

```
EventUpcaster upcaster = new SingleEventUpcaster() {
    @Override
    protected boolean canUpcast(IntermediateEventRepresentation ir) {
        return … // is this the type that should be upcast?
    }

    @Override
    protected IntermediateEventRepresentation
                    doUpcast(IntermediateEventRepresentation ir) {
        return … // create the upcast version of the representation
    }
};
```

Upcaster API – Example

```
EventUpcaster upcaster = new SingleEventUpcaster() {

    @Override
    protected boolean canUpcast(IntermediateEventRepresentation intermediateRepresentation) {
        return intermediateRepresentation.getType().getName().equals("com.example.MyEvent")
                && intermediateRepresentation.getType().getRevision().equals("1.
    }

    @Override
    protected IntermediateEventRepresentation doUpcast(IntermediateEventRepresentation intermediateRepresentation) {
        return intermediateRepresentation.upcastPayload(
                new SimpleSerializedType("com.example.MyEvent", "2.0"),
                JsonNode.class,
                n -> {
                    ((ObjectNode) n).put("newAttribute", "newValue");
                    return n;
                }
        );
    }
};
```

Allows to return an instance that lazily upcasts to the new version

The type of representation to work with

The actual modification of the intermediate representation

Deployment strategy: Big Bang

- Easiest approach
- No need for concurrent versions
- Deploy upcaster with new application

- Not a feasible solution for distributed systems
- Requires downtime

Deployment strategy: Blue-Green deployment

- Bring new version “up to speed” in parallel
- No need for concurrent versions
- Deploy upcaster with new application

- Not a feasible solution for large-scale distributed systems

Deployment strategy: Rolling upgrade

- Both versions run side-by-side for a ‘while’
- Old version needs to be able to understand new events
  - Forward and backward compatibility
- Don’t publish both ‘old’ and ‘new’ events separately

Message Schema – Compatibility Recommendations

- Never change semantic meaning of an event
  - That would mean it's a new event
- Never remove or change attributes
  - Deprecate attributes instead
- Only add (optional) attributes
- Use lenient deserialization
  - Use sensible defaults for missing attributes
  - Ignore unknown attributes

Lab 8

Upcasters

Distributed Systems

“Evolutionary” Microservices

- “Microservices are a journey, not a destination”
- Build microservices, monolith-first
  - Separate components as requirement comes up
  - Ensure correct abstraction of monolith’s components

Location Transparency

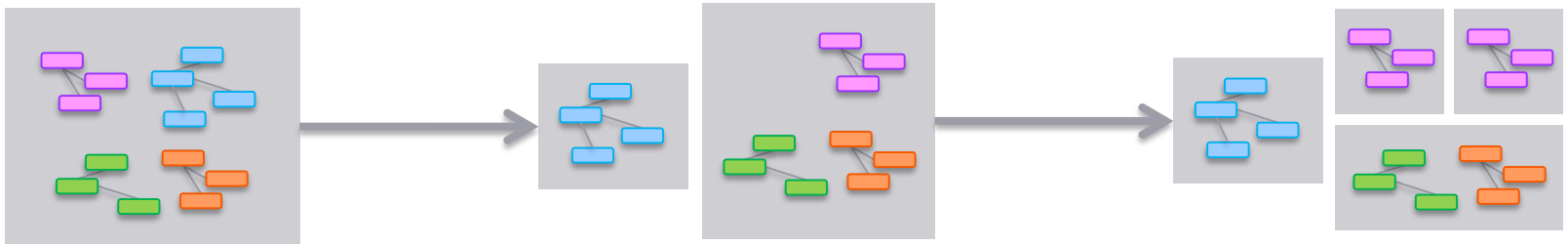
- A Component should not be aware, nor make any assumptions, of the physical location of Components it interacts with
- Beware of APIs & method signatures:
  - Not location transparent:

```
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



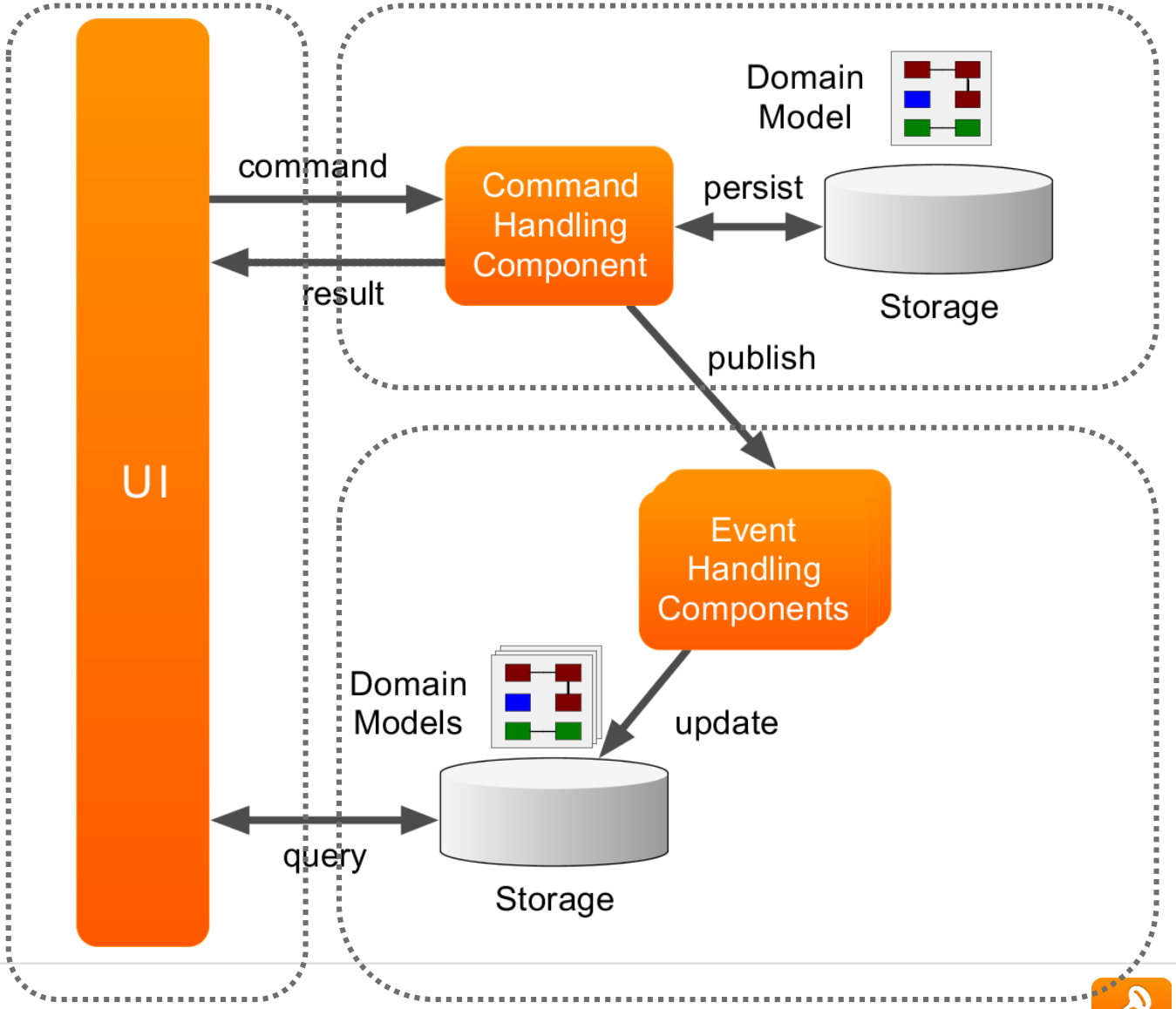
1 – Build a monolith

2 – Extract components

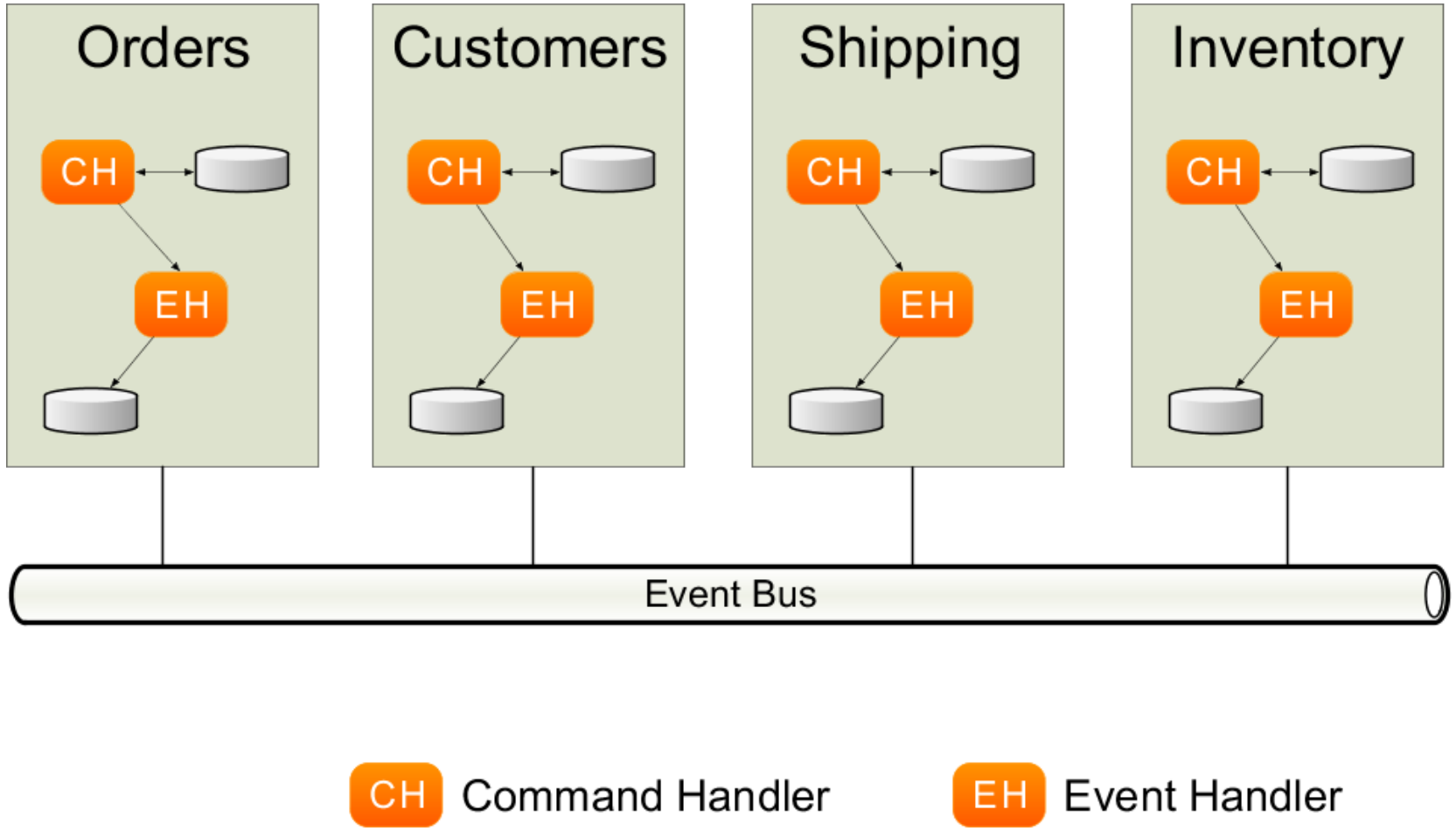
3 – Extract more

4 – Scale as needed

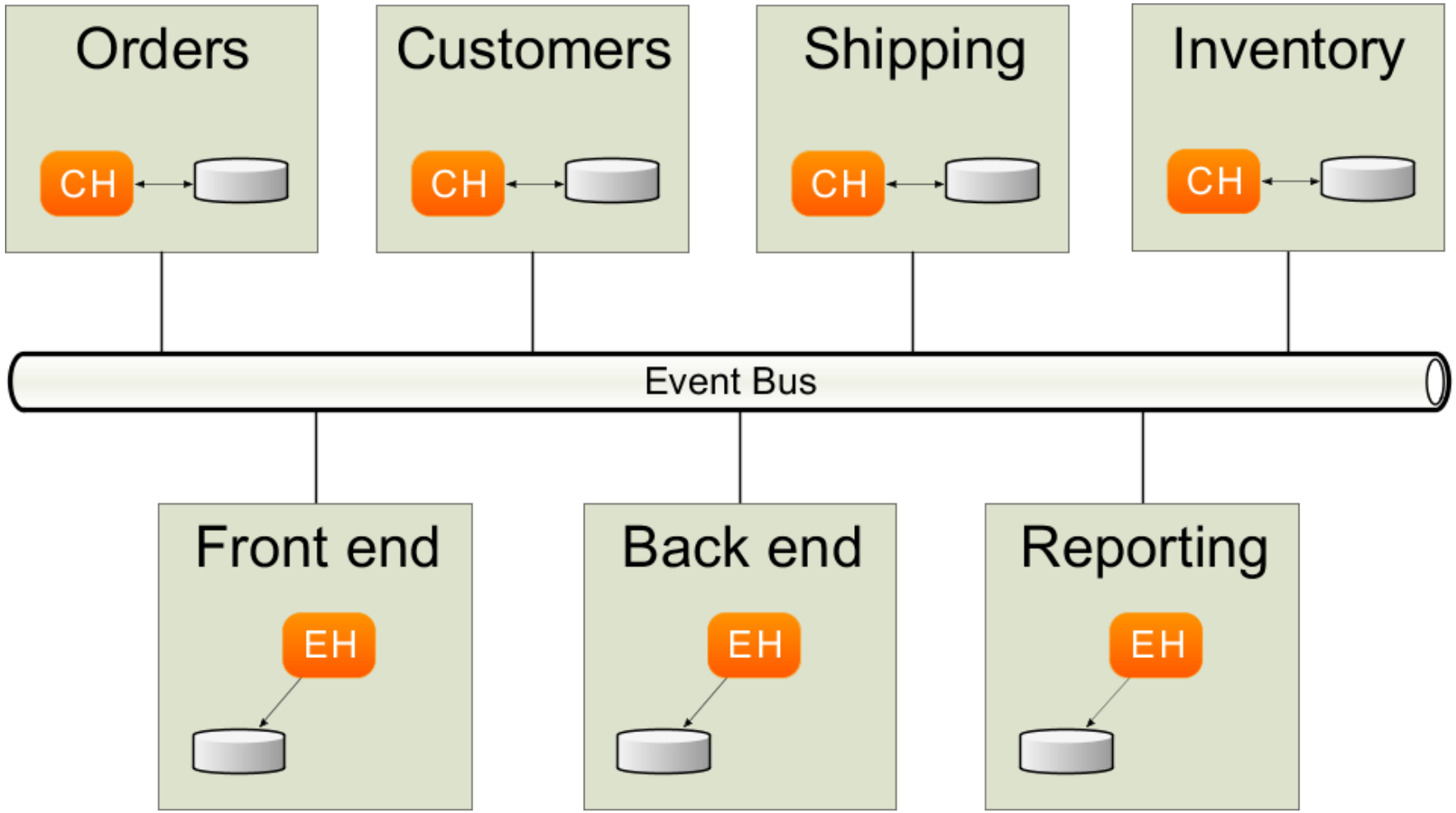
Location Transparency boundaries



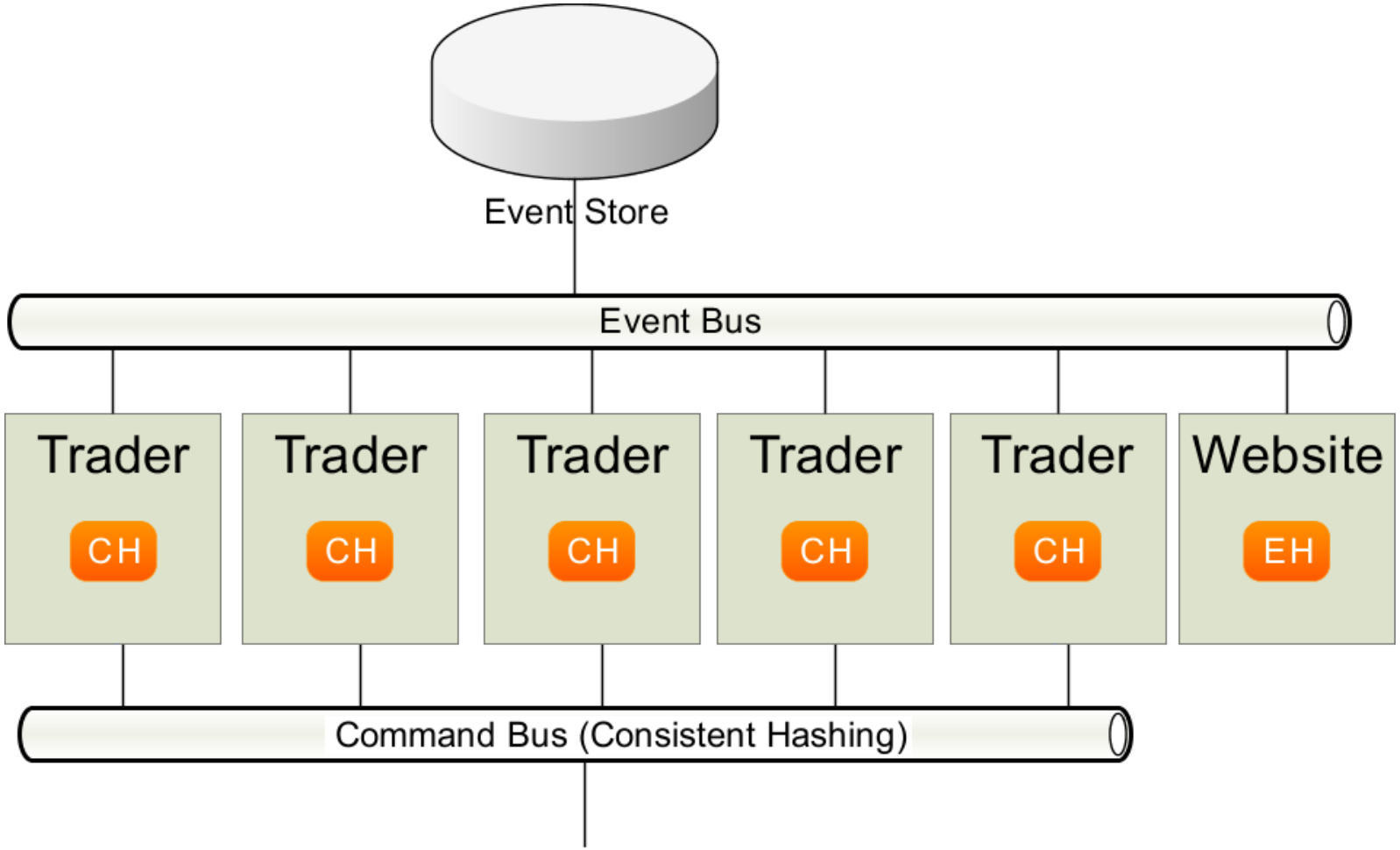
Scaling scenario – Bounded context



Scaling scenario – Separate audience



Scaling scenario – High Volume Command Processing



Sharing Events between nodes

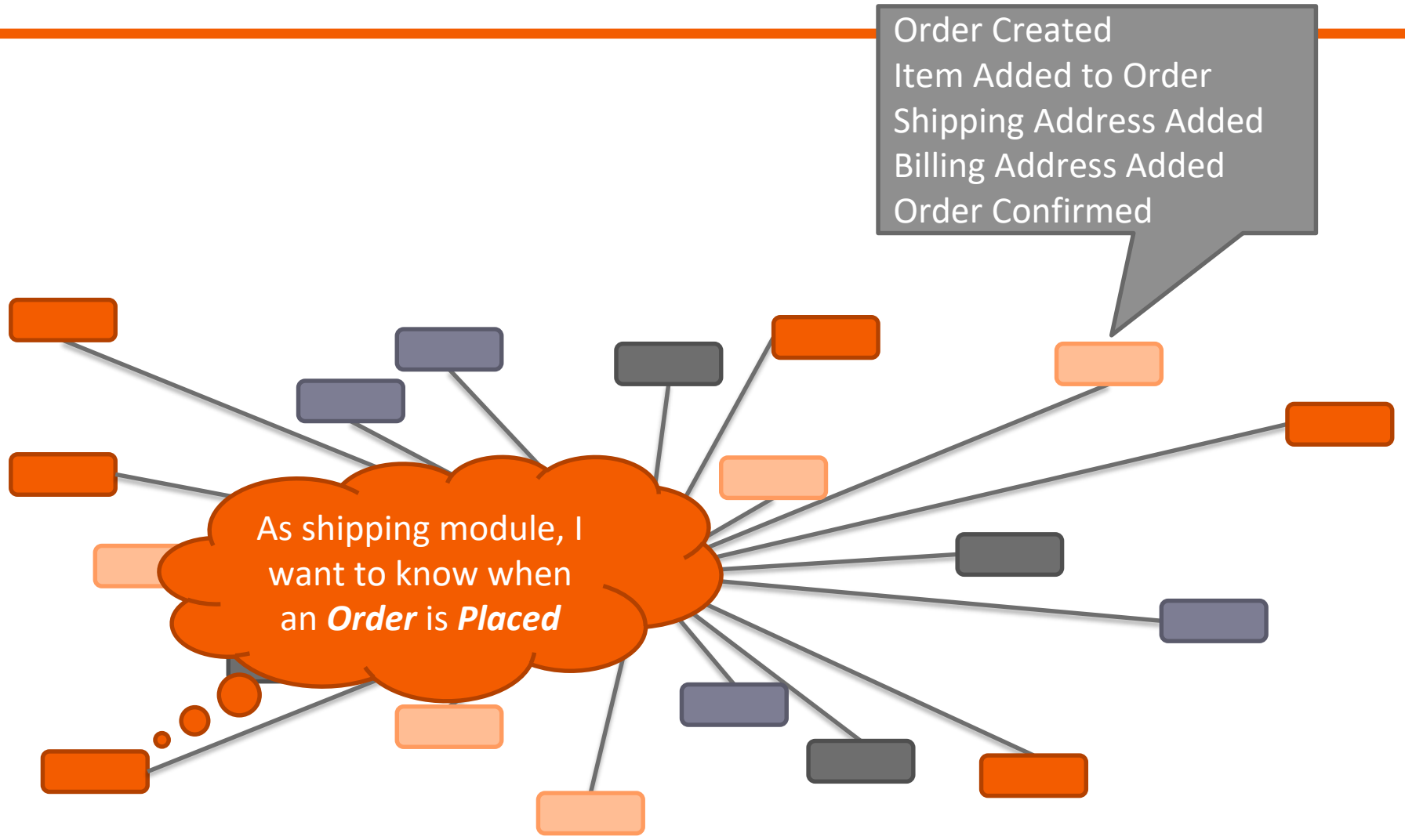
- Embedded Event Store with shared data source
  - Tracking processors will track all stored events
- “Proper” Event Store
  - For example: AxonDB
- Message Broker
  - Publish all events messages to broker
  - Register Message Broker as message source for Processors
  - Spring AMQP: SpringAMQPPublisher
- Beware the “contract”!

Distributing Command Messages

- Distributed Command Bus
  - Command Router
  - Command Bus Connector
- Provided implementations
  - Spring Cloud Discovery
  - JGroups

Large Scale Distributed Systems

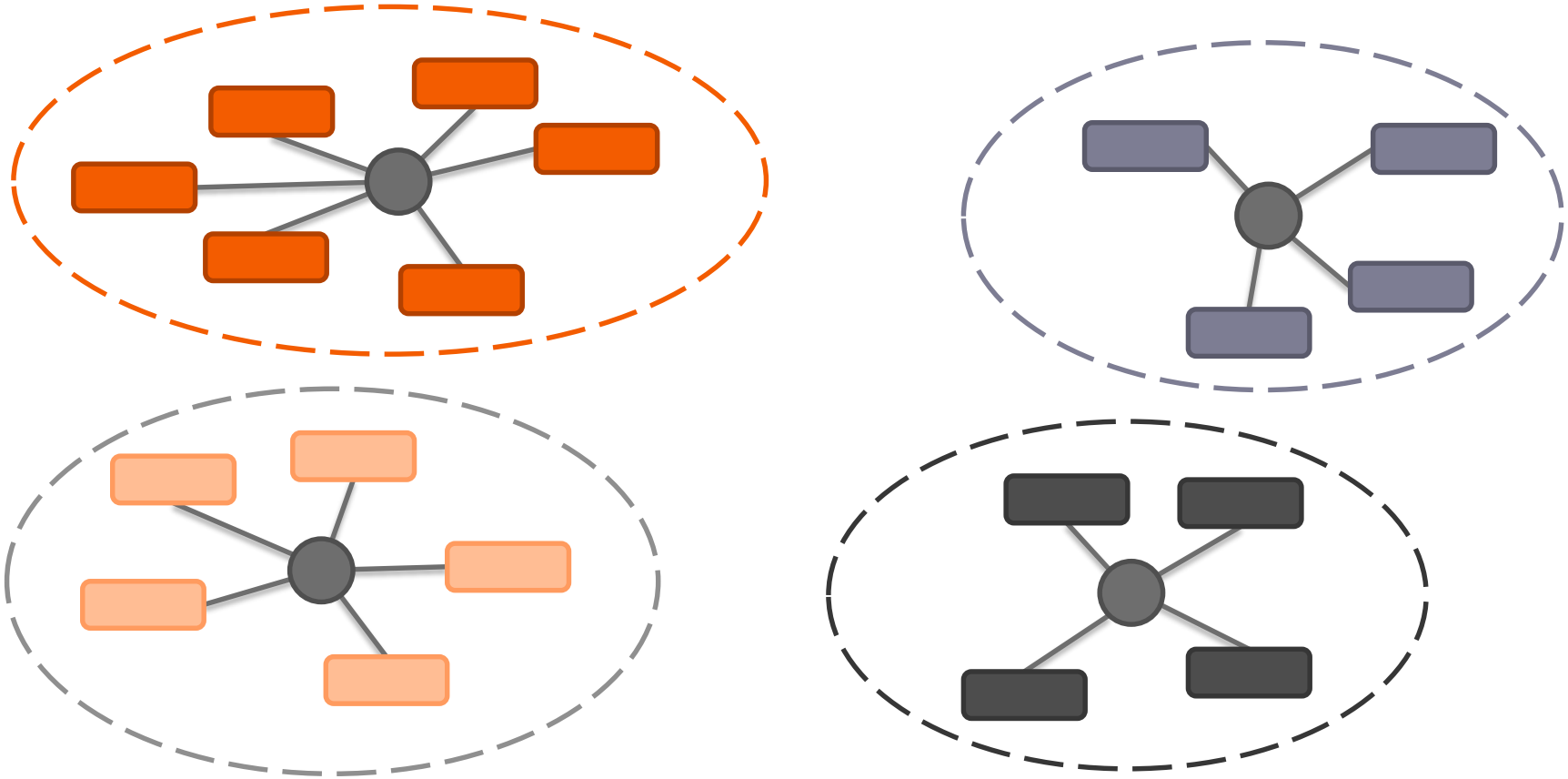
Unmanageable mess



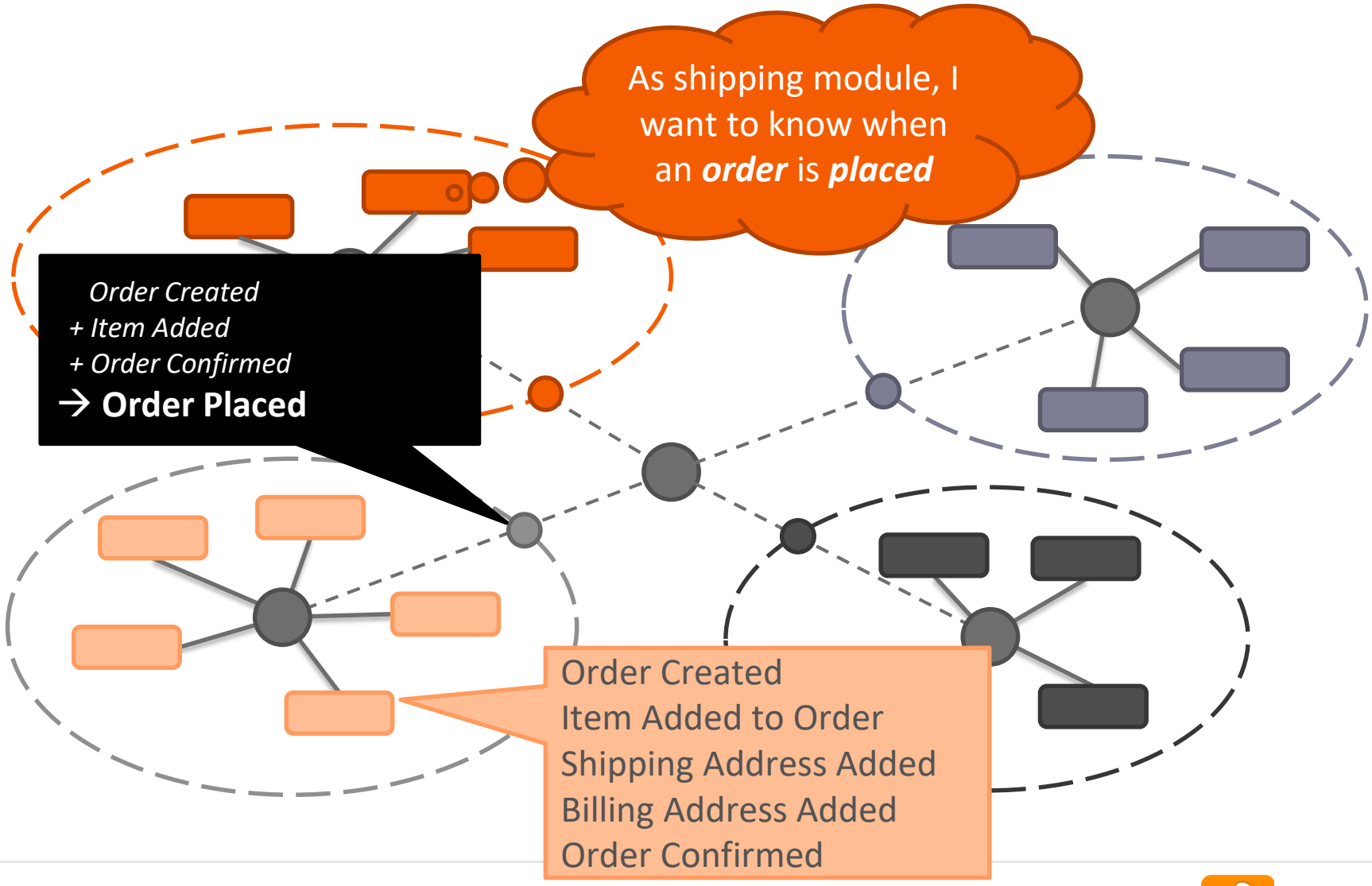
Bounded context

> Explicitly define the context within which a model applies. Explicitly set boundaries in terms of team organization, usage within specific parts of the application, and physical manifestations such as code bases and database schemas.
>
> Keep the model strictly consistent within these bounds, but don't be distracted or confused by issues outside.

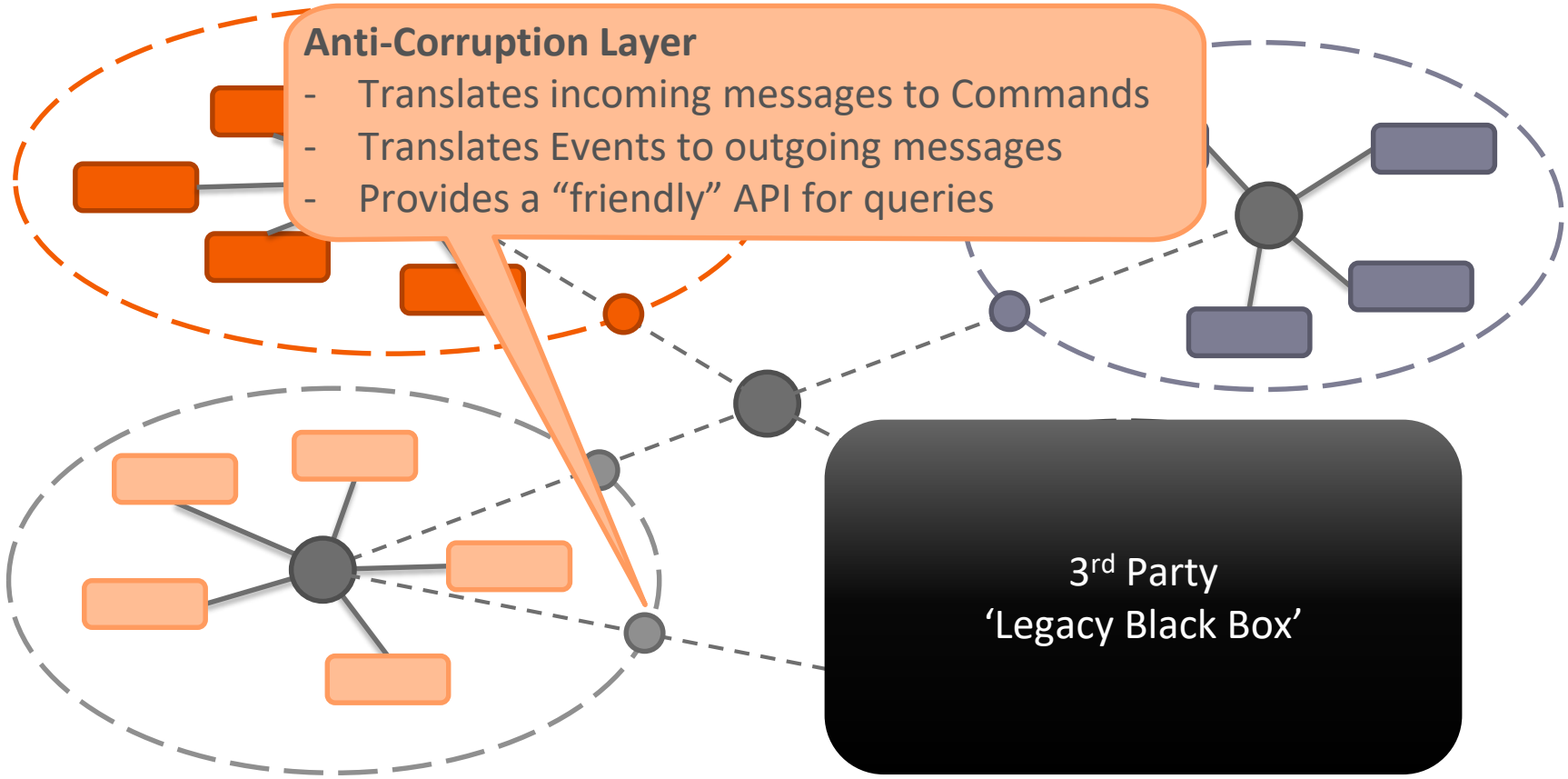
Within a context, share ‘everything’



Between contexts, share ‘consciously’



3rd Party Integration



Lab 9

Distributing the application

Monitoring

Messaging infrastructure

- Message flow provides valuable information about component health
- Cause-and-effect flow gives insight in what's happening

- Axon components allow for Message Monitor
  - Invoked on ingest and after processing of message
  - Measure throughput, response times, utilization, etc.

Correlation Data Providers

- Unit of Work attaches correlation data
  - Based on incoming message
  - Attached to all outgoing messages
- CorrelationDataInterceptor
- CorrelationDataProvider
  - MessageOriginProvider (correlationId, traceId)

Message Monitors

- Generic monitors
  - NoOpMessageMonitor
  - MultiMessageMonitor
- Dropwizard Metrics monitors
  - MessageTimerMonitor
  - CapacityMonitor
  - PayloadTypeMessageMonitorWrapper
  - MessageCountingMonitor
  - EventProcessorLatencyMonitor

Lab 10

Monitoring

Advanced Tuning

Unit of Work

Advanced Tuning…

Unit of Work

- Records Message and Execution Result
- 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

Unit of Work

- To access the current Unit of Work
  - Parameter on Message Handler method
  - CurrentUnitOfWork.get();
- Unit of Work is created by all components processing Messages
- Only 1 Unit of Work can be active at any time
  - Nesting is supported

Unit of Work – Message and Execution Result

- `T getMessage();`
- `ExecutionResult getExecutionResult()`
- `boolean isRolledBack()`

- `transformMessage(`

  `Function<T, ? extends Message<?>> transformOperator)`

Unit of Work – Hooking into the lifecycle

- `phase()`

- `onPrepareCommit(Consumer<UnitOfWork<T>> handler)`
- `onCommit(Consumer<UnitOfWork<T>> handler)`
- `afterCommit(Consumer<UnitOfWork<T>> handler)`
- `onRollback(Consumer<UnitOfWork<T>> handler)`
- `onCleanup(Consumer<UnitOfWork<T>> handler)`

Unit of Work – Registering resources

- `Map<String, Object> resources()`
- `R getResource(String name)`
- `R getOrDefaultResource(String key, R defaultValue)`
- `R getOrComputeResource(`

  `String key,`

  `Function<? super String, R> mappingFunction)`

Register resources that should be reused by nested unit of work should with the `root()` Unit of Work.

Unit of Work – Correlation Data Management

- `registerCorrelationDataProvider(CorrelationDataProvider correlationDataProvider)`
- `MetaData getCorrelationData()`

Used by constructors of all Message implementations to initialize MetaData
(except copy-constructors)

```
interface CorrelationDataProvider {

    Map<String, ?> correlationDataFor(Message<?> message);

}
```

Parameter Resolvers

Advanced features…

Parameter Resolver

- Resolves parameters of @MessageHandler methods
  - Based on incoming Message
  - Resolves single parameter value
- Explicitly configured on components
- Located using ServiceLoader
  - META-INF/services/org.axonframework.messaging.annotation.ParameterResolverFactory

Parameter Resolver

```
interface ParameterResolverFactory {
  ParameterResolver createInstance(Executable executable,
                                   Parameter[] parameters,
                                   int parameterIndex);
}

interface ParameterResolver<T> {
  T resolveParameterValue(Message<?> message);
  boolean matches(Message<?> message);
}
```

Message Interceptors

Advanced features…

Intercepting Messages

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

Intercepting Messages – Use Cases

- Handler Interceptors
  - Attach (database) transaction
  - Validate security meta data
- Dispatch Interceptors
  - Structural validation (of Commands)
  - Attach node-id for distributed tracing
  - Attach security meta-data

Message Interceptors

```
interface MessageHandlerInterceptor<T extends Message<?>> {

    Object handle(UnitOfWork<? extends T> unitOfWork,
                  InterceptorChain interceptorChain) throws Exception
}


interface MessageDispatchInterceptor<T extends Message<?>> {

    BiFunction<Integer, T, T> handle(List<T> messages);
}
```

Handler Enhancers

Advanced features…

Message Handler Enhancers

- All message handlers are (meta-)annotated with @MessageHandler
- Type specific logic is implemented as Handler Enhancers
  - Wrap handler method
  - Provide additional information about handler (e.g. routing keys)
  - Add additional behavior to handler (e.g. end Saga lifecycle)
- Configure using ServiceLoader
  - META-INF/services/org.axonframework.messaging.annotation.HandlerEnhancerDefinition

Handler Enhancer

```
interface HandlerEnhancerDefinition {
  <T> MessageHandlingMember<T> wrapHandler(
                                 MessageHandlingMember<T> original);
}
```

- For convenience, return a WrappedMessageHandlingMember instance
- Return “original” to reject “enhancement”

Handler Enhancer use cases

- More specific alternative to Handler Interceptor
  - Handler Enhancers have more information about handling type / instance
- React to (additional) annotations on Handler methods
  - e.g. Security annotations
- Detailed logging / tracing

Serialization

Advanced tuning

XStream serializer

- Default
- Can serialize *everything*
- Aliases
  - Package aliases
  - Class name aliased
  - Etc.
- Lenient serialization
  - XStream.ignoreUnknownElements()

Jackson Serializer

- Serialized to JSON
- Cleaner output
- Has requirements on objects to serialize
  - Annotations
  - Getters/Setters
- Suitable for events (and commands) mainly
- Lenient serialization
  - @JsonIgnoreProperties(ignoreUnknown = true)
  - objectMapper.disable(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES);

Tuning serialization

- Content type converters
  - Serializer-specific
  - Generic
- Event Serializer
  - Serialize used only for events in the Event Store
  - Spring Boot: `@Qualifier("eventSerializer")`
- Custom serializer
  - Wrapper to serialize specific events

Recap – Day 1

- Advanced Aggregate and Command Handling
- Sagas
- Event Processors & Replays
- Refactoring and evolving your application
  - Evolving Commands and Events
  - Deployment strategies
  - Upcasting

Recap – Day 2

- Building Microservices with Axon:
  - Distributed systems
  - Distributed command and event handling
- 3rd party integration
- Monitoring, measuring throughput & latency, message tracing
- Advanced tuning:
  - Unit of Work
  - Parameter Resolvers
  - Message Interceptors
  - Handler Enhancers
  - Serialization