AXON FRAMEWORK

EXPLORING CQRS AND EVENT SOURCING ARCHITECTURE

POINTS COVERED

1. CQRS (Command and Query Responsibility Segregation).
2. Event Sourcing.
3. Axon Framework
4. SAGA

What is CQRS?

Command and Query Responsibility Segregation, it is a software building pattern that works on separating the part of an application that changes the state of an application and the part that queries the state of the application.

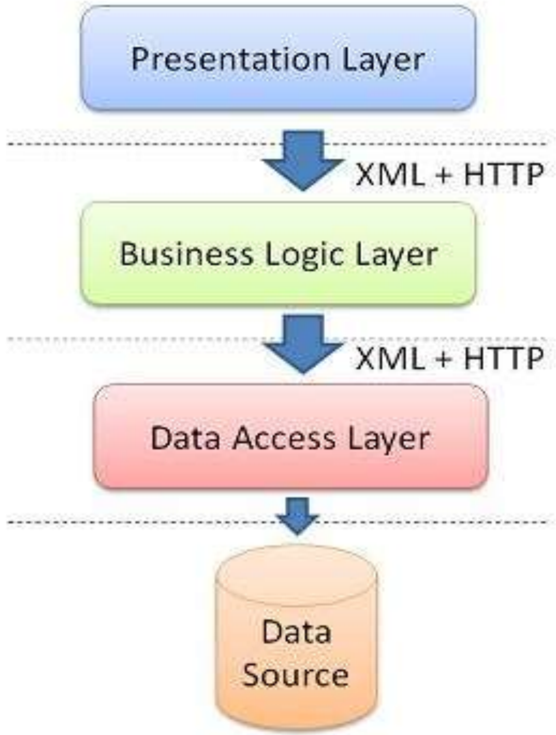
Simple concept, which is, have the “write” part of an application distinctly separate from the “read’ part of an application.

Remember…

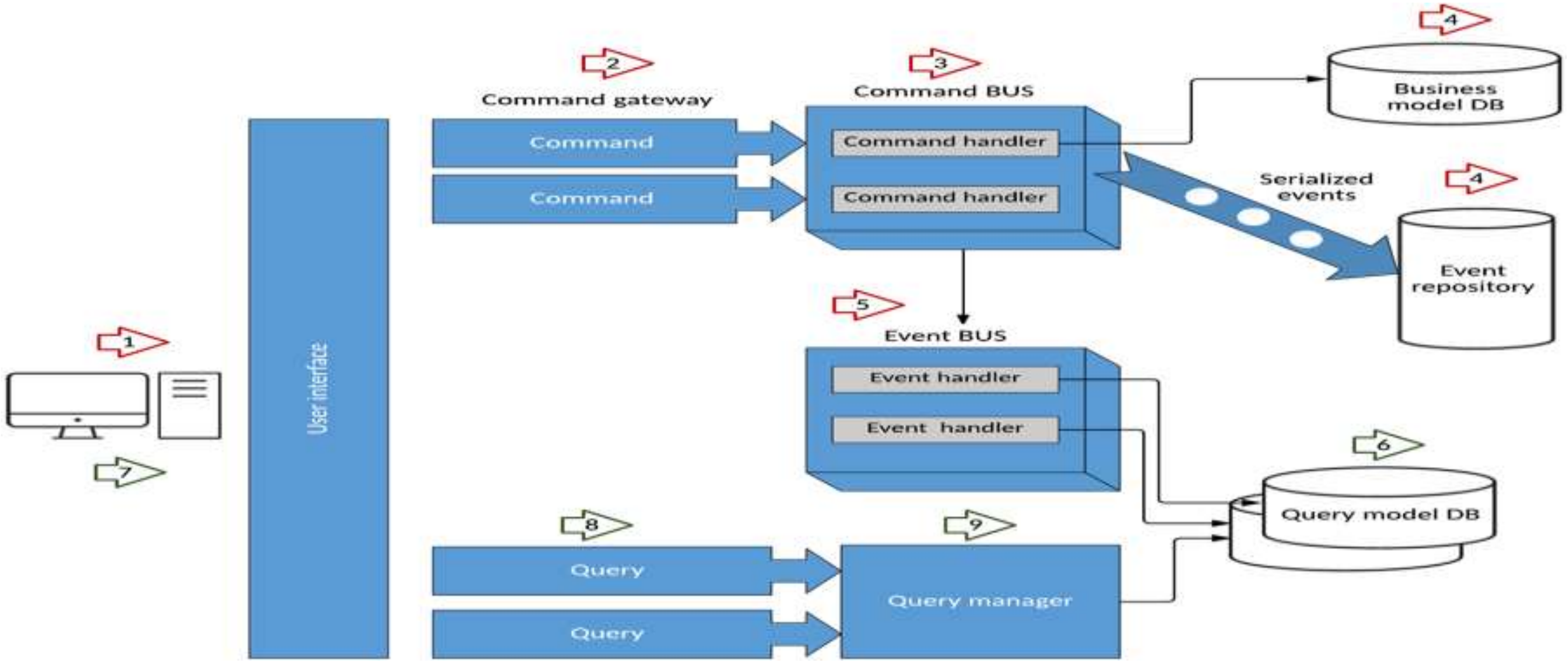
CQRS is not seen as an “Architecture” but a “Pattern”. This open ups the component based approach.

Layered arch. vs Component based approach

In layered architecture the components are arranged in layers, and can make direct calls to the layers below it. Whereas in component based approach components are semi-autonomous and collaborate with each other using messaging system.



CQRS components



What is Event Sourcing?

1. Changes made to state are tracked as events.
2. Events are stored in event store(any database).
3. Use stored events and summation of all these events always arrive at the current state.

Note : - Event Sourcing is not part of CQRS.

What is Axon Framework?

By name it says is a framework that helps developers implement Command and Query Responsibility Segregation pattern to build a scalable and extensible application. It does so by providing implementation to the building blocks of the framework like EventBus, CommandBus, EventStore, aggregate, repositories etc...

Overview of some DDD concepts

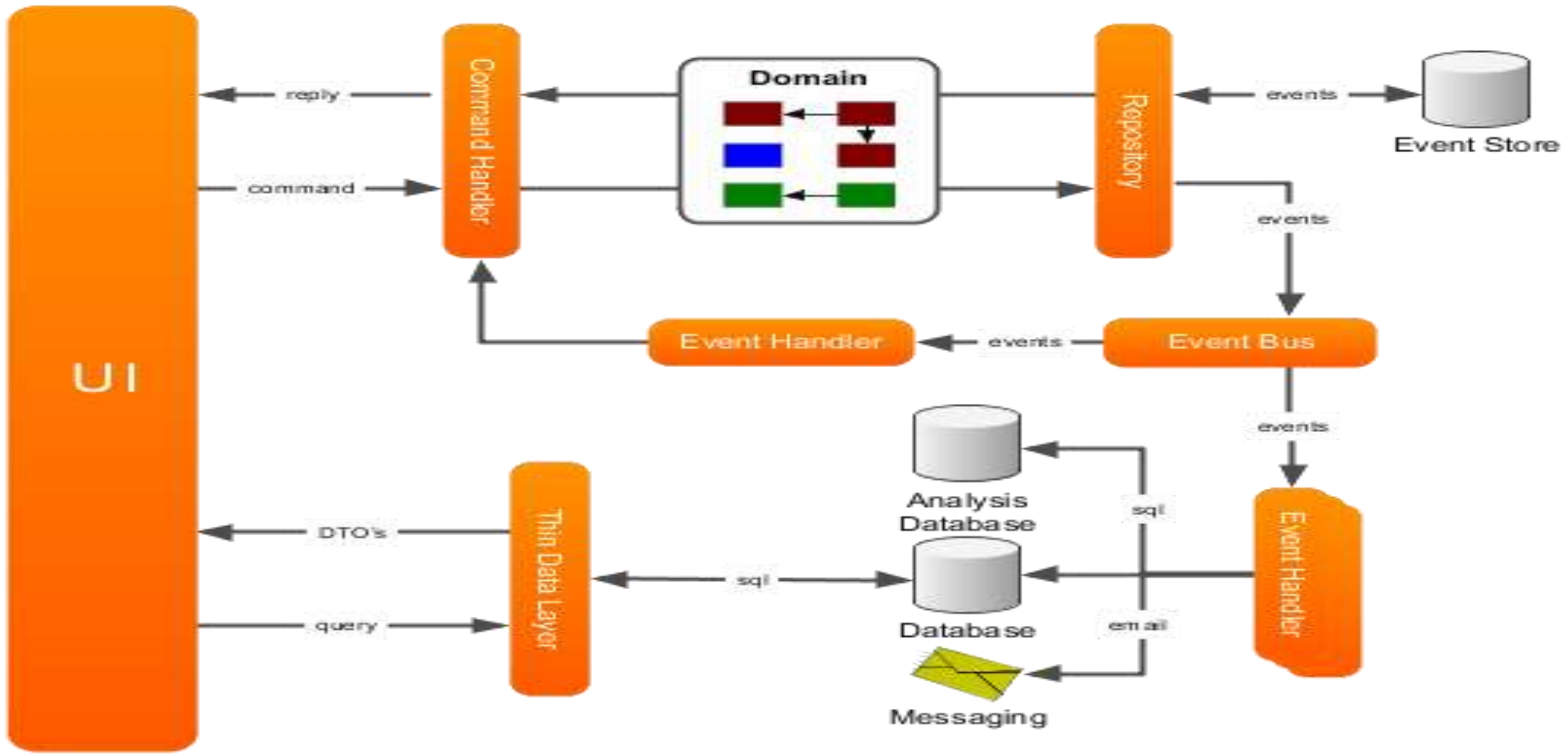
Domain Objects : Objects that model domain. Hold the state of the application.

Entity : Domain objects with an identity.

Aggregate : A logical grouping of domain objects to form an atomic and cohesive whole.

Aggregate Root : A grouping object that has been designed to be a container object.

Axon Framework Architecture



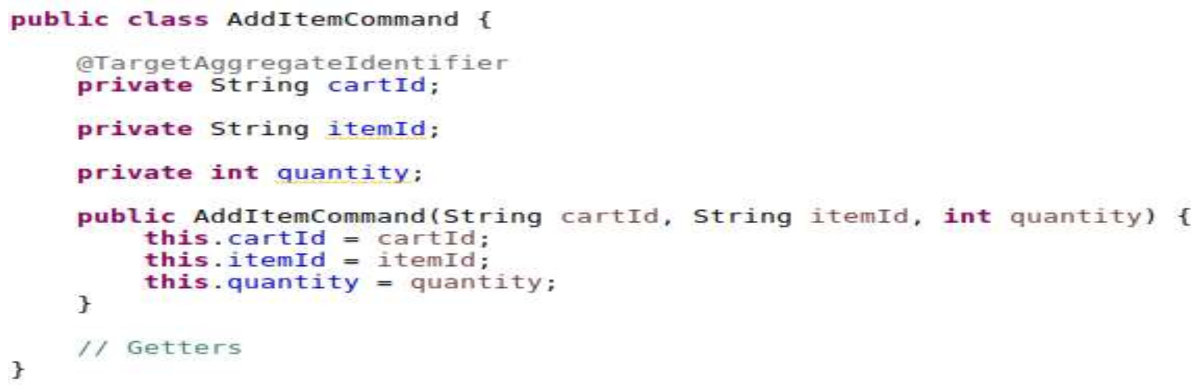
Axon components

Command

Represents that something should happen within a system. In axon it's a plain object that represents the intent with necessary information required.

1. Plain POJO object.

2. Doesn't need to implement any interface or extend any class.



```
public class AddItemCommand {

    @TargetAggregateIdentifier
    private String cartId;

    private String itemId;

    private int quantity;

    public AddItemCommand(String cartId, String itemId, int quantity) {
        this.cartId = cartId;
        this.itemId = itemId;
        this.quantity = quantity;
    }

    // Getters
}
```

continuing...

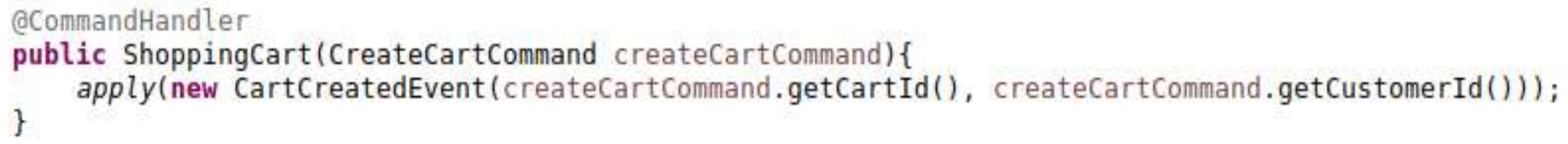
Command Handler

Component that performs a task based on the command that it receives. Usually command handlers are plain methods with @CommandHandler annotation. Performs operations based on the intent captured in command it receives.

Command Handler can be created via two ways

1. Implementing CommandHandler interface.

2. If using axon with spring, use @CommandHandler on a method to be turned into a command handler



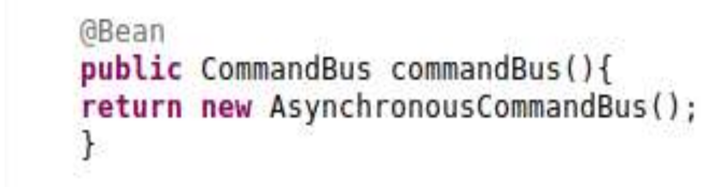
```
@CommandHandler
public ShoppingCart(CreateCartCommand createCartCommand){
    apply(new CartCreatedEvent(createCartCommand.getCartId(), createCartCommand.getCustomerId()));
}
```

continuing...

Command Bus

Component that routes the commands to their respective command handlers. Command handlers subscribe themselves to the command bus with CommandHandlerInvoker class.

When using axon with spring, axon auto configuration subscribes all command handlers with the bus automatically. Four different types of command bus are provided by the axon namely SimpleCommandBus, DistributedCommandBus, AsynchronousCommandBus and DisruptorcommandBus.



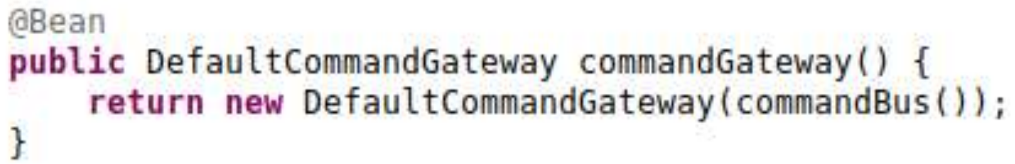
```
@Bean
public CommandBus commandBus(){
return new AsynchronousCommandBus();
}
```

continuing...

Command Gateway

It is possible to use command bus directly to send out commands, but it's usually recommended to use command gateway. Command gateway provides simpler APIs to send out commands than Command Bus.

Axon provides DefaultCommandGateway as an implementation for the CommandGateway.



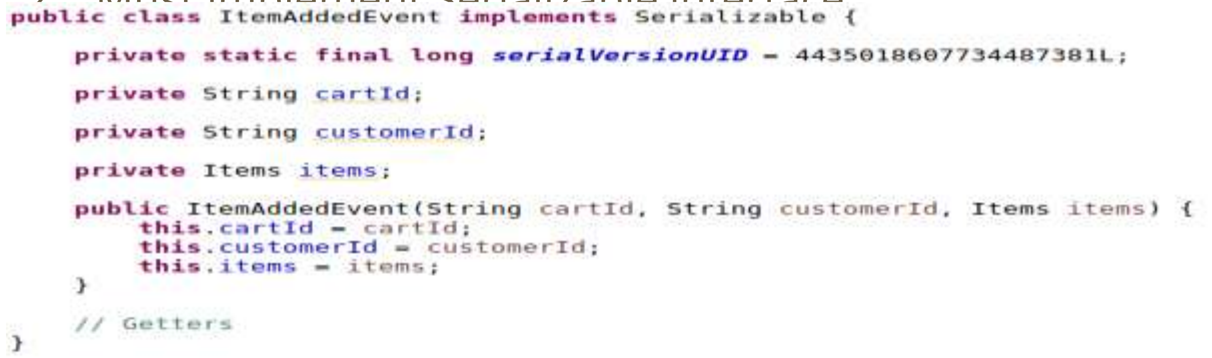
```
@Bean
public DefaultCommandGateway commandGateway() {
    return new DefaultCommandGateway(commandBus());
}
```

continuing...

Event

Represents that something has happened within the application. In axon it’s plain object with data representing the state change. Raised as a result of command process.

1. Plain POJO object.
2. Must implement Serializable interface



```
public class ItemAddedEvent implements Serializable {

    private static final long serialVersionUID = 4435018607734487381L;

    private String cartId;

    private String customerId;

    private Items items;

    public ItemAddedEvent(String cartId, String customerId, Items items) {
        this.cartId = cartId;
        this.customerId = customerId;
        this.items = items;
    }

    // Getters
}
```

continuing...

1. **Event Message**

   Any message wrapped in an EventMessage object. EventMessage object contains timestamp attribute.

1. **Domain Event**

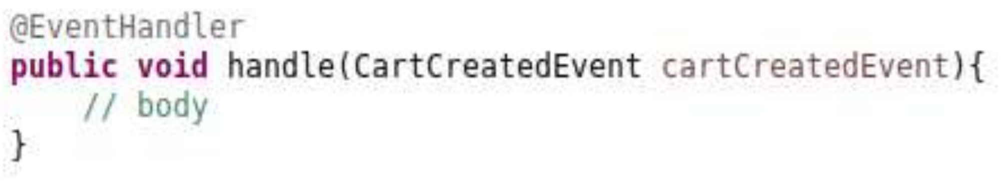
Event objects that are originated from aggregate. Wrapped in DomainEventMessage(which extends EventMessage) object. The DomainEventMessage additionally contains type and identifier of the aggregate that has raised an event.

continuing...

Event Handler

Performs an action on receiving the event of type defined as the first parameter of the method. The method is annotated with @EventHandler annotation that represent that method as event handler.

If you are using axon with spring, then axon will automatically register these handlers with the Event Bus.



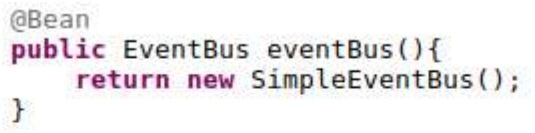
```
@EventHandler
public void handle(CartCreatedEvent cartCreatedEvent){
    // body
}
```

continuing...

Event Bus

Similar to the Command Bus Event Bus is a component that routes the events to the event handlers. EventHandlerInvoker class is responsible for invoking handler methods.

Axon provides SimpleEventBus as an implementation for Event Bus.



```
@Bean
public EventBus eventBus(){
    return new SimpleEventBus();
}
```

Aggregate

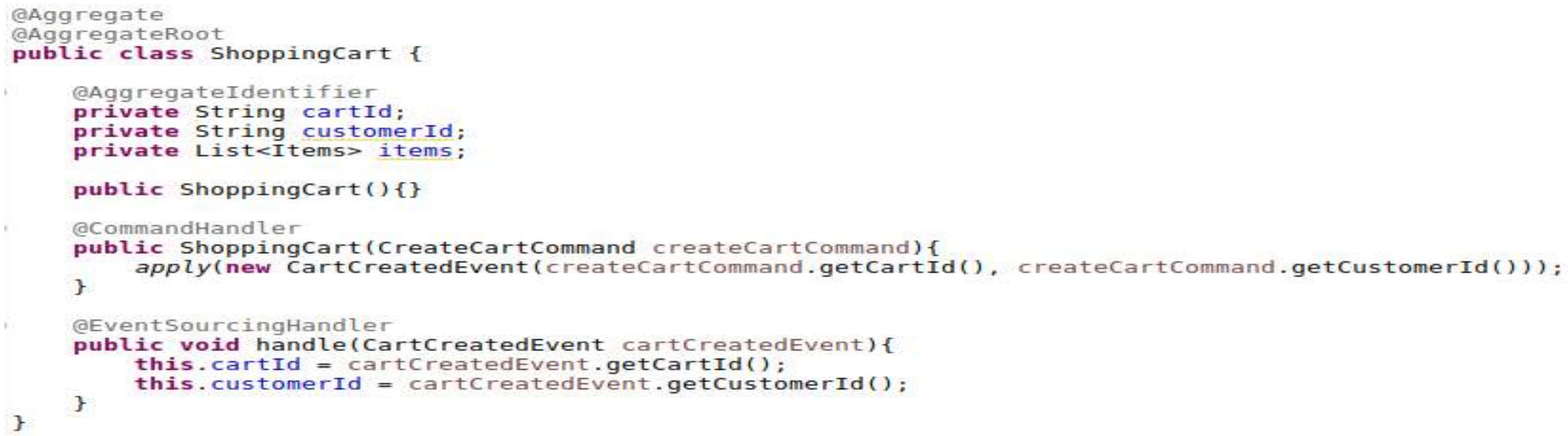
Domain objects annotated with @Aggregate annotation. Aggregate Root is annotated with @AggregateRoot annotation.

1. **Regular Aggregate Root**

   Aggregates state is stored in the persistent medium.

1. **Event Sourcing Aggregate Root**

   Events resulted from the aggregate are stored to the persistence medium.

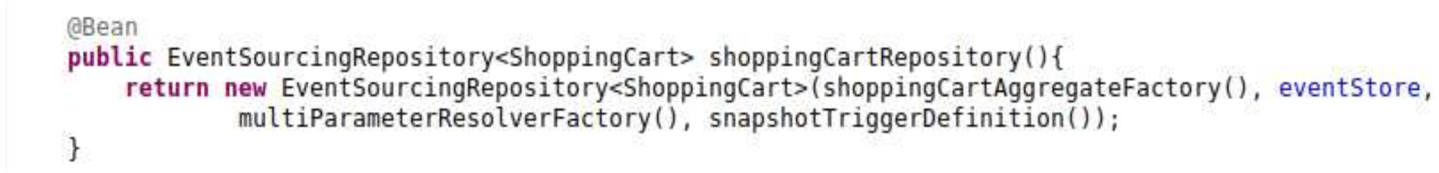


```
@Aggregate
@AggregateRoot
public class ShoppingCart {

    @AggregateIdentifier
    private String cartId;
    private String customerId;
    private List<Items> items;

    public ShoppingCart(){}

    @CommandHandler
    public ShoppingCart(CreateCartCommand createCartCommand){
        apply(new CartCreatedEvent(createCartCommand.getCartId(), createCartCommand.getCustomerId()));
    }

    @EventSourcingHandler
    public void handle(CartCreatedEvent cartCreatedEvent){
        this.cartId = cartCreatedEvent.getCartId();
        this.customerId = cartCreatedEvent.getCustomerId();
    }
}
```

Repository

Repository provides mechanism to store and retrieve aggregate to and from the persistence medium.

1. **Standard Repository**
   Stores regular aggregates. Axon provides JPA backed repository for this purpose.
1. **Event Sourcing Repository**
   Stores events raised from aggregates. Axon provides GenericEventSourcingRepository for this purpose.



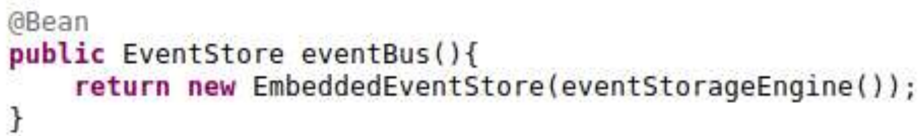
```
@Bean
public EventSourcingRepository<ShoppingCart> shoppingCartRepository(){
    return new EventSourcingRepository<ShoppingCart>(shoppingCartAggregateFactory(), eventStore,
            multiParameterResolverFactory(), snapshotTriggerDefinition());
}
```

Event Store (READ-ONLY and APPEND-ONLY)

It is a type of Event Bus that stores the events in the persistence medium. Repositories need event store to store and load events from aggregates.

Axon provides out-of-the-box, the EmbeddedEventStore. It delegates the actual storage and retrieval of events to the EventStorageEngine.

EventStore stores uses DomainEventEntry and SnapshotEventEntry domains to store events in the persistence medium. These domains have properties like aggregate type, identifier, payload, payload type, sequence number etc.

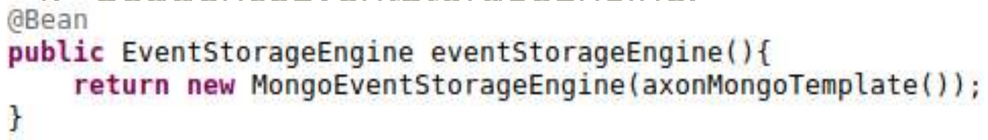


```
@Bean
public EventStore eventBus(){
    return new EmbeddedEventStore(eventStorageEngine());
}
```

EventStorageEngine

EventStorageEngine stores events in it’s compatible data source. Axon provides different StorageEngine implementations for different storage mediums.

1. JPAEventStorageEngine.
2. MongoEventStorageEngine.
3. JDBCEventStorageEngine.
4. SequenceEventStorageEngine.

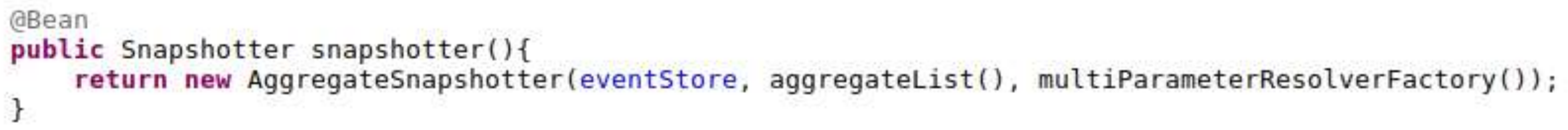


```
@Bean
public EventStorageEngine eventStorageEngine(){
    return new MongoEventStorageEngine(axonMongoTemplate());
}
```

Snapshotting

Re-creating current state of the Aggregate object from stored events is a time-consuming process when your application is in production, as your application is loaded with thousands of events. Here event snapshotting comes to the rescue.

A snapshot event is a domain event that summarizes an arbitrary amount of event into one. By regularly creating and storing of snapshot event, the event store does not have to load all the events to re-create the current state of an Aggregate.



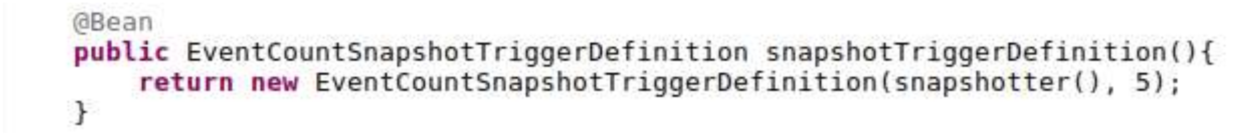
```
@Bean
public Snapshotter snapshotter(){
    return new AggregateSnapshotter(eventStore, aggregateList(), multiParameterResolverFactory());
}
```

Snapshot Trigger

Snapshotting can be triggered by a number of factors like a number of events stored in since the last snapshot, time-based etc..

The definition of when snapshot should be triggered is provided by SnapShotTriggerDefinition interface.

EventCountSnapShotTriggerDefinition triggers snapshotting when a number of events required to load an aggregate exceed a certain threshold.



```
@Bean
public EventCountSnapshotTriggerDefinition snapshotTriggerDefinition(){
    return new EventCountSnapshotTriggerDefinition(snapshotter(), 5);
}
```

SAGA

BASE Transaction?

Basic **A**vailability **S**oft State **E**ventual Consistency

Basically Available — The system must ensure the availability of data. There will be an answer for every request.

Soft State — The state of the system could change over time, so even during times without input there may be changes going on due to ‘eventual consistency,’ thus the state of the system is always ‘soft.’

Eventually consistent— The system will *eventually* become consistent once it stops receiving input. The data will propagate to everywhere it should sooner or later, but the system will continue to receive input and is not checking the consistency of every transaction before it moves onto the next one.

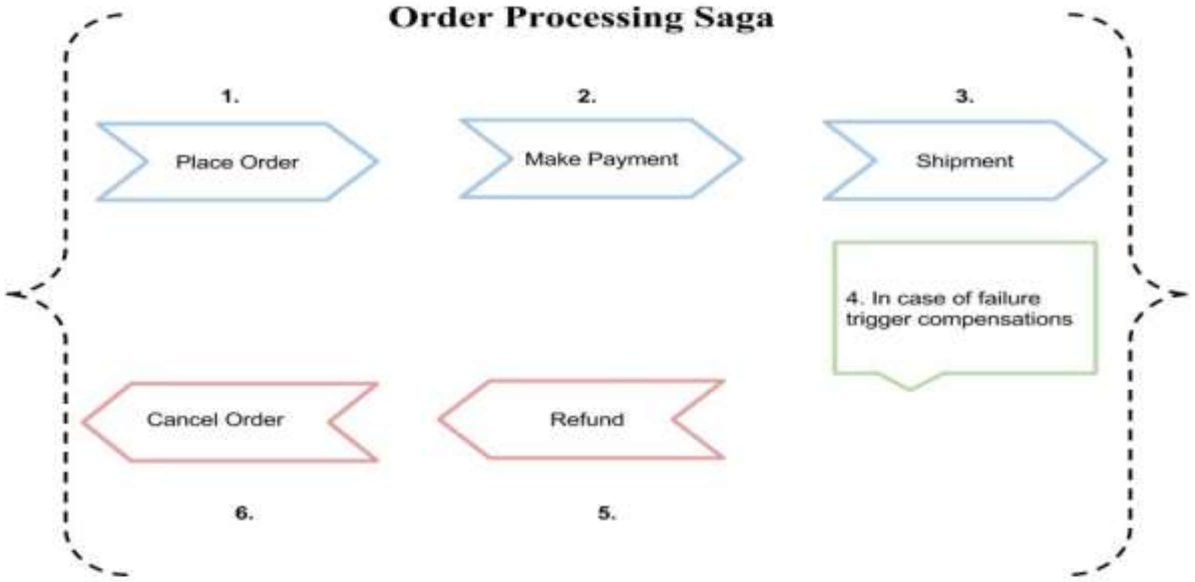
SAGA?

- ➢In CQRS, Sagas are responsible for managing these BASE transactions. They respond on Events produced by Commands and may produce new commands, invoke external applications, etc. In the context of Domain Driven Design, it is not uncommon for Sagas to be used as coordination mechanism between several bounded contexts.

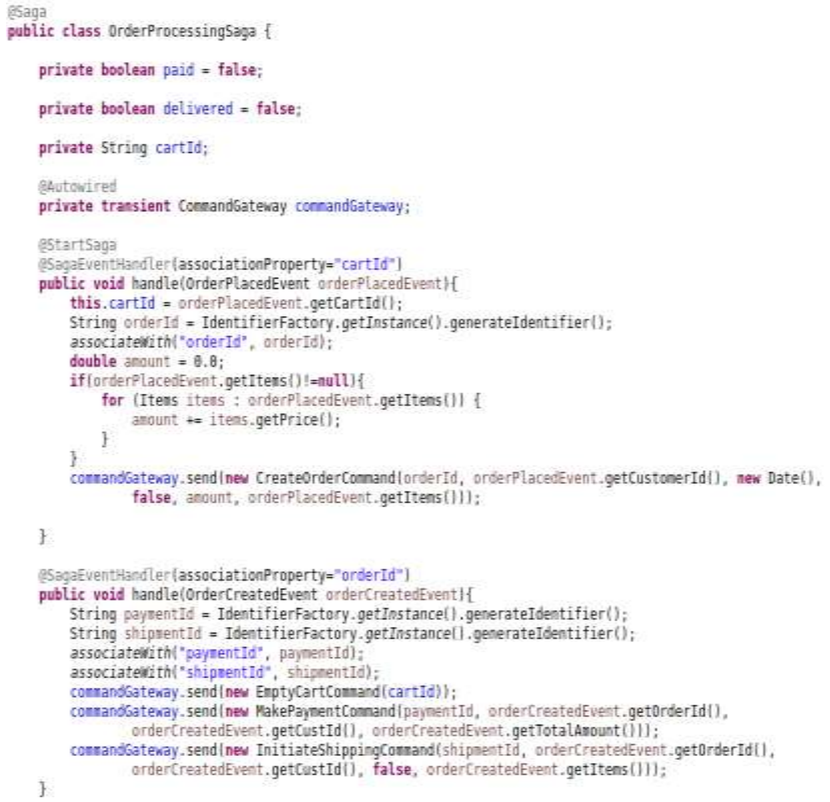
- ➢Saga has a starting point and an end, both triggered by Events.

Continuing...

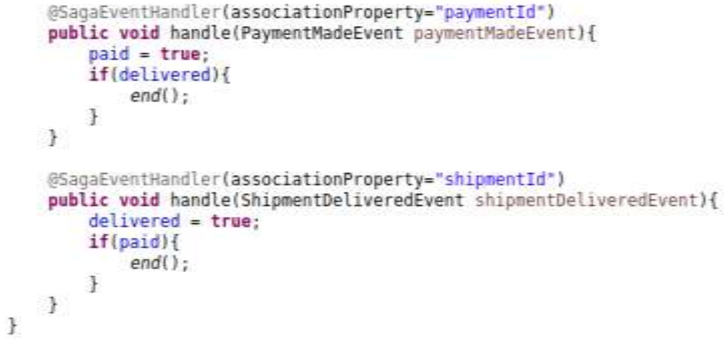
Contrary to ACID, BASE transactions cannot be easily rolled back. To roll back, compensating actions need to be taken to revert anything that has occurred as part of the transaction.



Example



```
@Saga
public class OrderProcessingSaga {

    private boolean paid = false;

    private boolean delivered = false;

    private String cartId;

    @Autowired
    private transient CommandGateway commandGateway;

    @StartSaga
    @SagaEventHandler(associationProperty="cartId")
    public void handle(OrderPlacedEvent orderPlacedEvent){
        this.cartId = orderPlacedEvent.getCartId();
        String orderId = IdentifierFactory.getInstance().generateIdentifier();
        associateWith("orderId", orderId);
        double amount = 0.0;
        if(orderPlacedEvent.getItems()!=null){
            for (Items items : orderPlacedEvent.getItems()) {
                amount += items.getPrice();
            }
        }
        commandGateway.send(new CreateOrderCommand(orderId, orderPlacedEvent.getCustomerId(), new Date(),
                false, amount, orderPlacedEvent.getItems()));

    }

    @SagaEventHandler(associationProperty="orderId")
    public void handle(OrderCreatedEvent orderCreatedEvent){
        String paymentId = IdentifierFactory.getInstance().generateIdentifier();
        String shipmentId = IdentifierFactory.getInstance().generateIdentifier();
        associateWith("paymentId", paymentId);
        associateWith("shipmentId", shipmentId);
        commandGateway.send(new EmptyCartCommand(cartId));
        commandGateway.send(new MakePaymentCommand(paymentId, orderCreatedEvent.getOrderId(),
                orderCreatedEvent.getCustId(), orderCreatedEvent.getTotalAmount()));
        commandGateway.send(new InitiateShippingCommand(shipmentId, orderCreatedEvent.getOrderId(),
                orderCreatedEvent.getCustId(), false, orderCreatedEvent.getItems()));
    }
```



```
    @SagaEventHandler(associationProperty="paymentId")
    public void handle(PaymentMadeEvent paymentMadeEvent){
        paid = true;
        if(delivered){
            end();
        }
    }

    @SagaEventHandler(associationProperty="shipmentId")
    public void handle(ShipmentDeliveredEvent shipmentDeliveredEvent){
        delivered = true;
        if(paid){
            end();
        }
    }
}
```

Examples

Find examples on CQRS and Event Sourcing implementations using Axon framework on github.

Banking App - https://github.com/meta-magic/cqrs-axon-example/tree/master/Bankingapp

Issue Traking App - https://github.com/meta-magic/cqrs-axon-example/tree/master/IssueTrackingApp

Order Process App - https://github.com/meta-magic/cqrs-axon-example/tree/master/ecommapp

END