



AXON FRAMEWORK

AGENDA

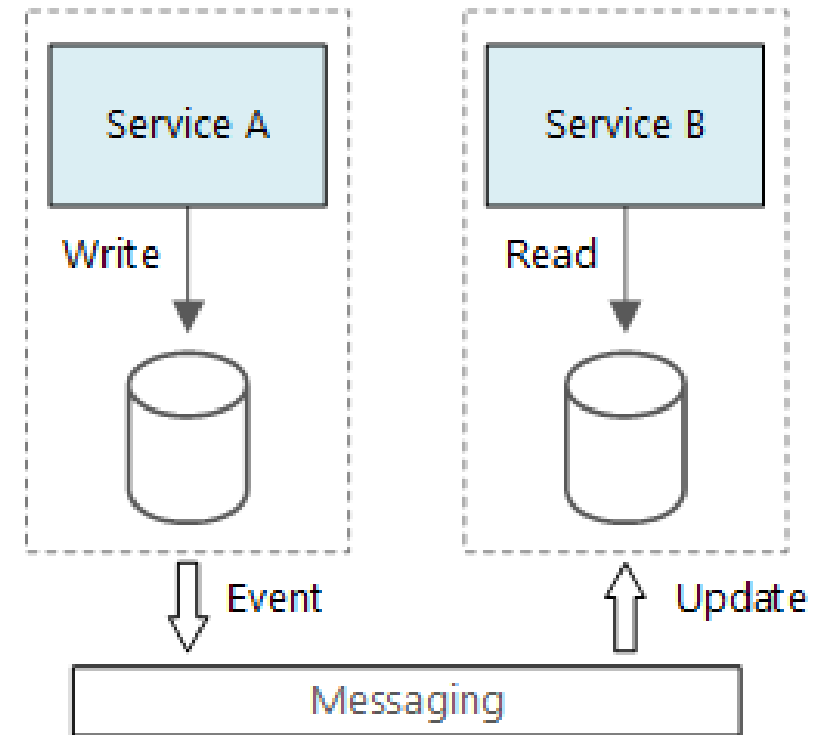


- ▶ Patrones:
 - ▶ CQRS
 - ▶ Agregado
 - ▶ Event Sourcing
 - ▶ Vista Materializada
 - ▶ Saga
- ▶ Axon Framework
 - ▶ The Reactive Manifesto
 - ▶ Arquitectura
 - ▶ Commands Bus/Gateway/Handler
 - ▶ Agregado
 - ▶ Querys Bus/Gateway/Handler
 - ▶ Event Bus/Handler
 - ▶ Saga
 - ▶ Repository/Event Store
 - ▶ Snapshotting/Event Upcasting

CQRS - COMMAND QUERY RESPONSIBILITY SEGREGATION



- ▶ Las operaciones de escritura y de lectura escalan independientemente.
- ▶ Cada uno cuenta con un esquema optimizado para la tarea a la que es encomendada.
- ▶ Toda la lógica de negocio se realiza en la escritura o modificación de los datos, mientras que la lectura debe ser lo mas simple posible (como se lee es como se presenta los datos).
- ▶ En la lectura al ser vistas materializadas estas no requiere que se realicen joins complejos.
- ▶ Como desventajas:
 - ▶ incrementa la complejidad al momento del diseño e implementación, aún mas si se agrega event sourcing.
 - ▶ Eventual consistencia.



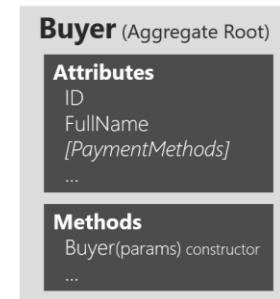
AGREGADO



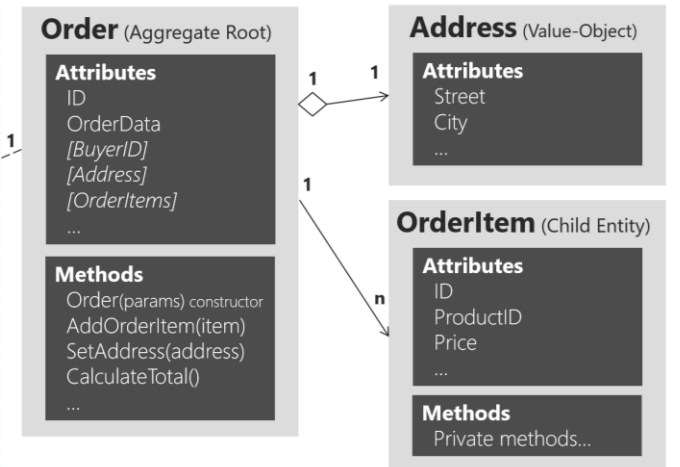
- ▶ Es una representación conceptual de un grupo de entidades de negocio que se encarga de ejecutar la lógica de negocio.
- ▶ Se debe empezar por un concepto de dominio y pensar en las entidades que se usan en las transacciones más comunes relacionadas con ese concepto.
- ▶ El agregado esta compuesto por una raíz, entidades de negocio y objetos de valor.
- ▶ La raíz proporciona el acceso a todas las entidades y objetos de valor a través de sus métodos.
- ▶ Las entidades son objetos de negocio que poseen identificación o que se pueden identificar para el dominio.
- ▶ Los objetos de valor son objetos que no poseen una identificación o que no necesitan contar con una identificación para el dominio.
- ▶ Como desventajas:
 - ▶ Almacenamiento y posterior recuperación.
 - ▶ Requiere de practica para poder separar la lógica que corresponde a cada uno de los agregados.

Aggregate pattern

Buyer Aggregate (One entity)



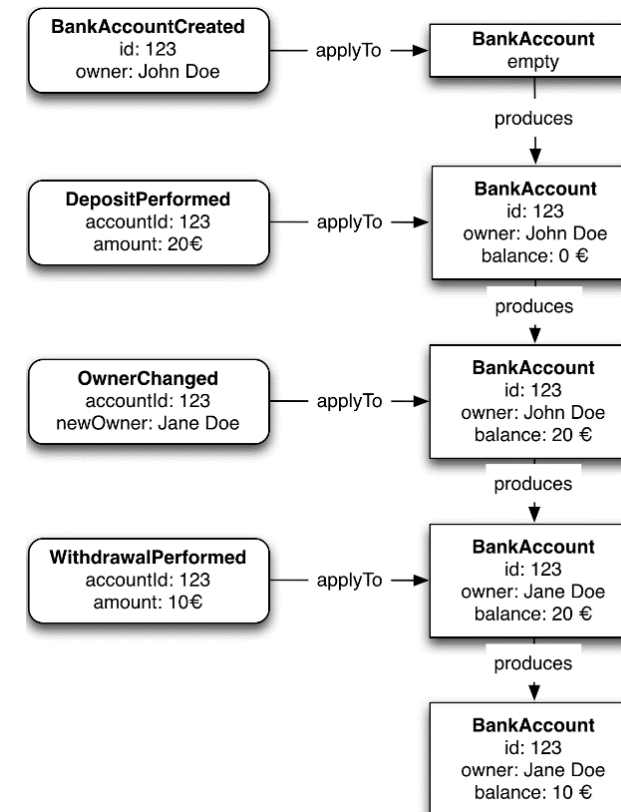
Order Aggregate (Multiple entities and Value-Object)



EVENT SOURCING



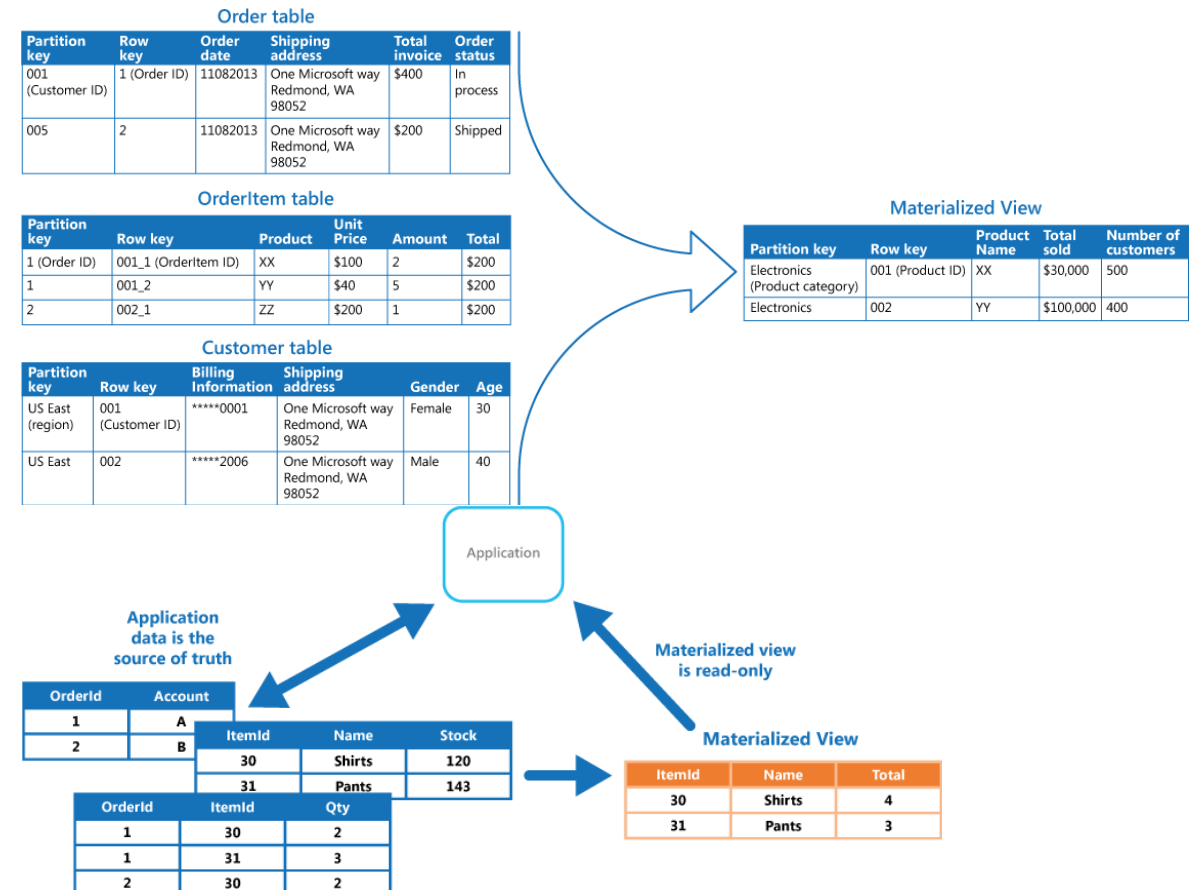
- ▶ Consiste en recuperar una serie de **eventos** previamente almacenados para ejecutarlos sobre nuestro modelo de dominio para obtener el estado actual.
- ▶ Se aplica sobre los agregados.
- ▶ Desventajas:
 - ▶ Procesar una enorme cantidad de eventos (ejemplo: 1000 eventos).
 - ▶ Si el contenido de los eventos cambia, ¿Cómo se procesaría los anteriores?.



VISTA MATERIALIZADA

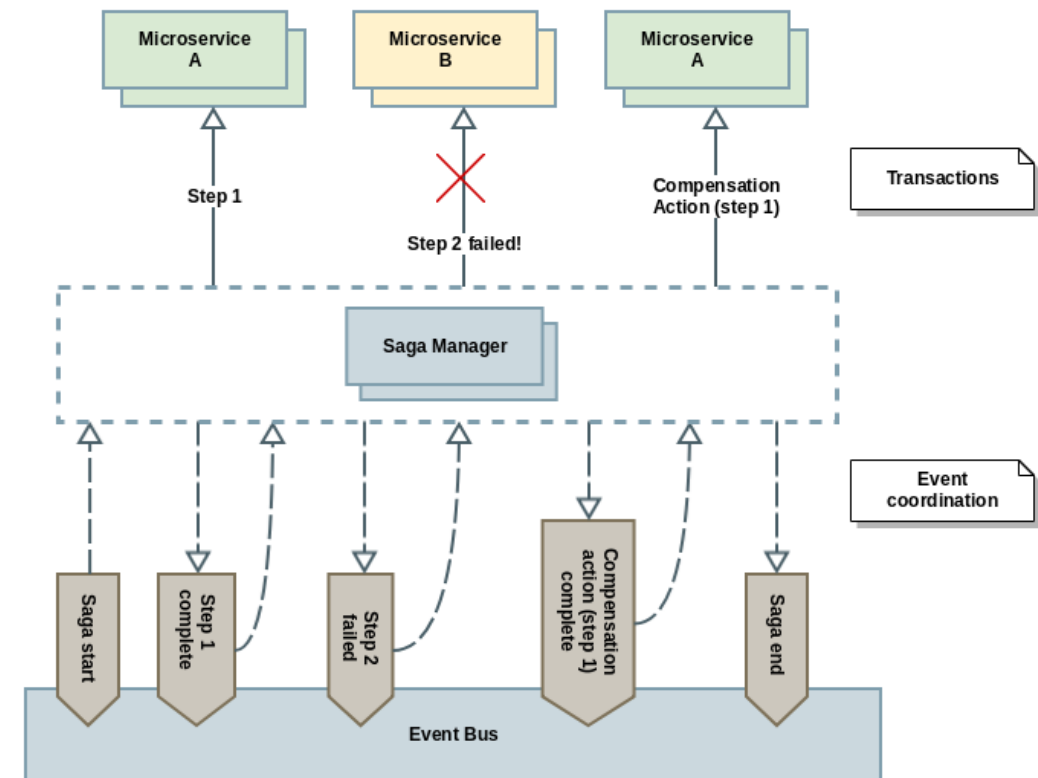


- ▶ Son tablas/colecciones/etc, donde se almacena la data que cumplen un objetivo de negocio.
- ▶ Se utiliza cuando el origen de los datos posee data no estructurada, semiestructura o cuando no es posible realizar consultas de manera eficiente.
- ▶ Normalmente una vista materializada es la agrupación de varias tablas/colecciones/etc, evitando los clásicos joins por tanto es optimizado.
- ▶ A través de un proceso de ETL se actualiza la vista materializada.
- ▶ Desventajas:
 - ▶ Como transportar la información para actualizar las vistas materializadas.



SAGA

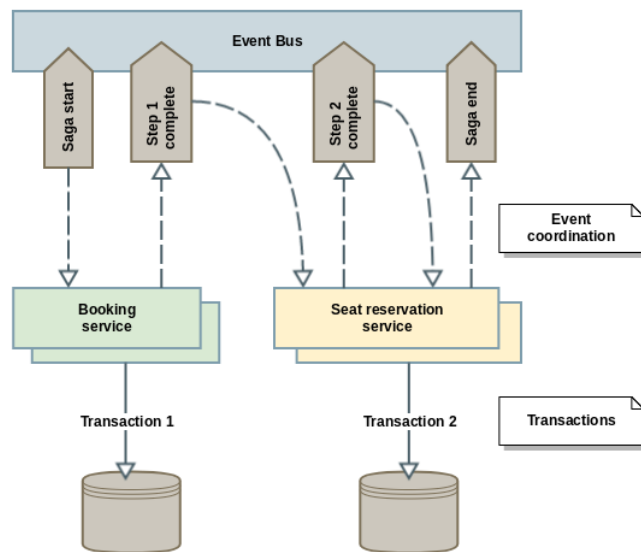
- ▶ La saga es una secuencia de transacciones locales que hay que coordinar. Además para cada una de estas transacciones se debe definir una **acción compensatoria** que deshaga el cambio que ha hecho la transacción.
- ▶ Existen dos tipos de saga:
 - ▶ Coreografía.
 - ▶ Orquestación.
- ▶ Como desventajas:
 - ▶ En teoría parece sencillo, pero ponerlo a funcionar no lo es (gestión de eventos, recuperar el estado de la saga, operaciones idempotentes, etc).
 - ▶ Concurrencia/aislamiento entre sagas.



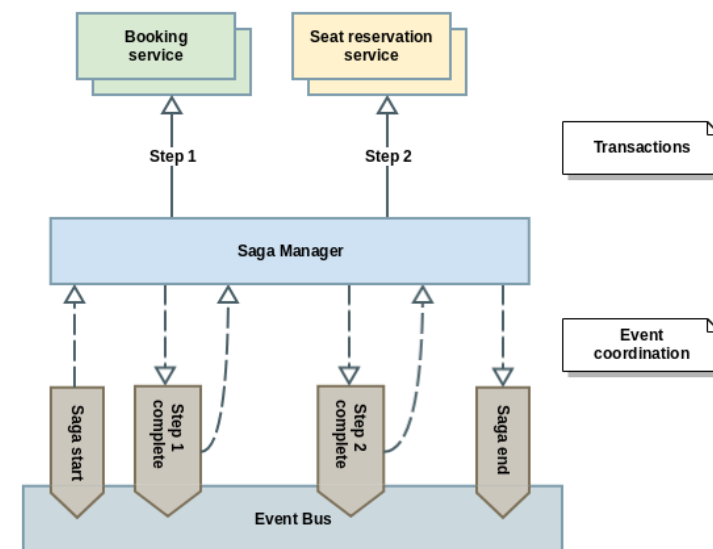
SAGA



Coreografía



Orquestación





AXON FRAMEWORK

- ▶ Es un framework enfocado para el desarrollo de aplicaciones CQRS/ES.
- ▶ Proporciona APIs para Event Sourcing, Agregados y Sagas.
- ▶ Orientado a mensajería y existen 3 tipos de mensajes:
 - ▶ Comandos (Commands): Representa la intención de realizar una acción.
 - ▶ Eventos (Events): Representa a las notificaciones cuando ha ocurrido algo importante.
 - ▶ Consultas (Querys): Representa una petición de información.
- ▶ Axon esta ligado con el manifiesto reactivo:
<https://dzone.com/articles/building-reactive-systems-with-axon-framework>



COMAND/EVENT/QUERY

COMMAND

CreateBookCmd
AddAuthorToBookCmd
TransferBookCmd
IncreaseStockCmd
SaveAuthorQueryCmd
SaveBookQueryCmd
UpdateAllSubscriptionQueryCmd

EVENT

CreateBookEvt
AddAuthorToBookEvt
TransferBookEvt
OkIncreaseStockEvt
OkSaveAuthorQueryEvt
OkSaveBookQueryEvt
UpdateAllSubscriptionQueryEvt

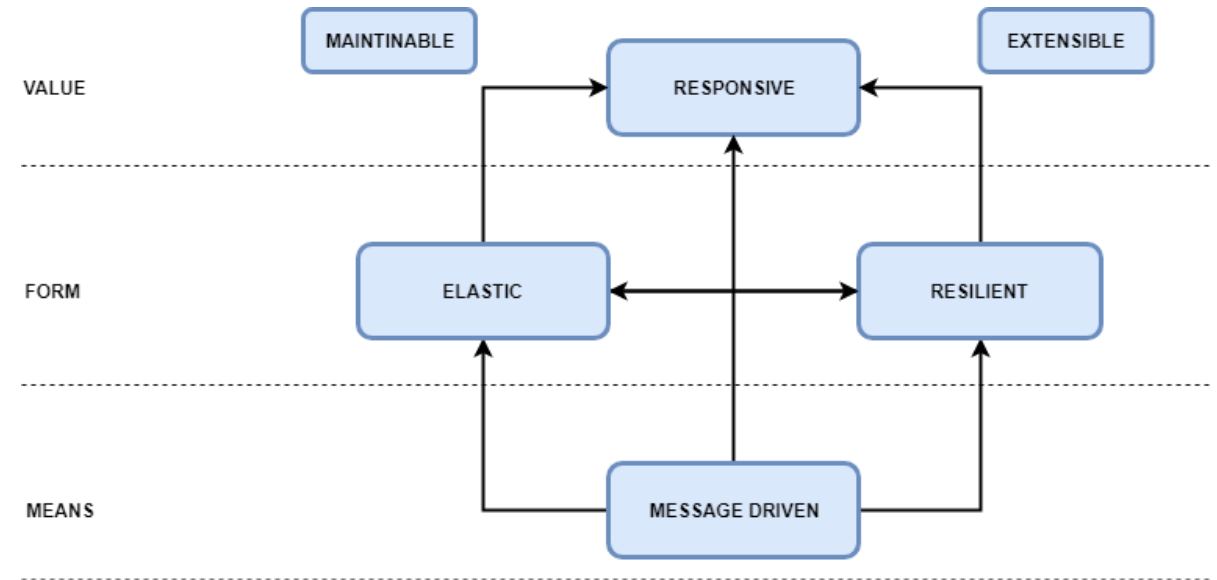
QUERY

FetchAllBooksQry
FetchBookByIdQry

THE REACTIVE MANIFESTO



- ▶ Orientado a mensajes (Message Driven): la comunicación es asíncrona aunque también puede tener el enfoque síncrono.
 - ▶ Bus de comandos (Command Bus).
 - ▶ Bus de eventos (Event Bus).
 - ▶ Bus de consultas (Query Bus).
- ▶ Sensibilidad (Responsive): Refiriéndose a la capacidad para completar una tarea en un tiempo determinado (Comandos síncronos o asíncronos).
- ▶ Resiliencia (Resilient): Gestionar los fallos y recuperarse de ellos (Proporcionar eventos que notifiquen el error a fin de controlarlo y que no existan bloqueos ej: Sagas).
- ▶ Elasticidad (Elastic): A través del escalamiento vertical u horizontal para gestionar la carga de trabajo (Los comandos podrían ser ejecutados en dos nodos distintos).



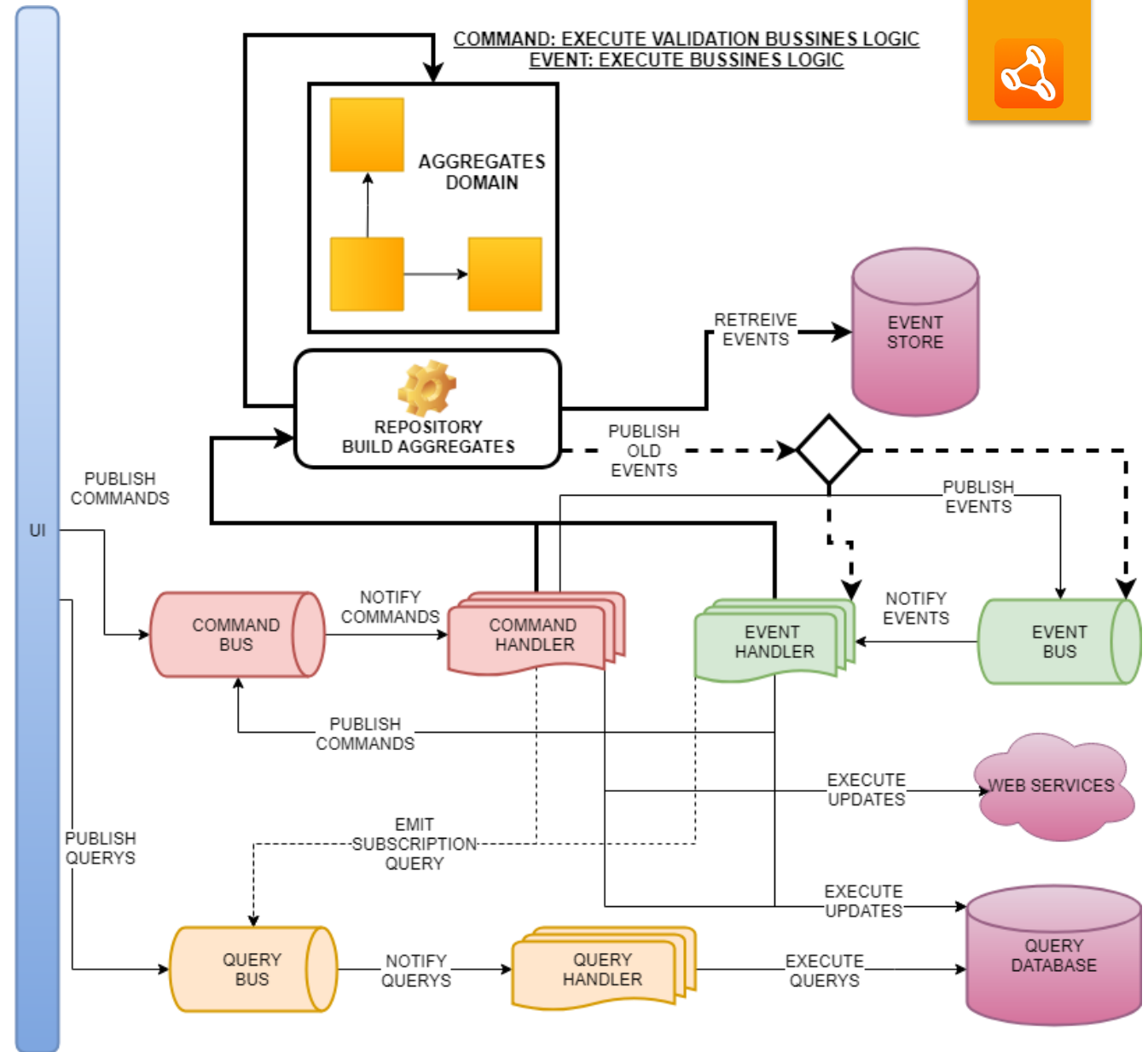
ARQUITECTURA

Repository: Encargado de la construcción del agregado.

Command Handler: Receptor de comandos.

Event Handler: Receptor de eventos.

Query Handler: Receptor de consultas.





COMMAND BUS / GATEWAY

- ▶ Interfaz encargada de transportar los mensajes de comandos.
- ▶ Existen diversas implementaciones de este bus:
 - ▶ SimpleCommandBus: El Comando se procesa en el mismo hilo que en el invoco al bus, por tanto se bloquea la operación hasta que el comando se procese.
 - ▶ AsynchronousCommandBus: El comando se procesa en un hilo diferente que en el que invoco al bus, por tanto no hay bloqueo y se debe tratar la respuesta a través de un callback.
 - ▶ DisruptorCommandBus y DistributedCommandBus: proporcionan configuraciones más avanzadas.

```
CreateBookCmd cmd = new CreateBookCmd();  
this.commandGateway  
    .send(cmd);  
this.commandBus  
    .dispatch(GenericCommandMessage.asCommandMessage(cmd));
```

COMMAND HANDLER



Agregado

```
@Aggregate(snapshotTriggerDefinition = "clasicSnapShotter")
@NoArgsConstructor
@Getter
public class Book {

    @AggregateIdentifier
    private String idBook;
    private String title;
    private Date publish;

    private List<Author> authors;

    @CommandHandler
    public Book(CreateBookCmd cmd) {
        AggregateLifecycle.apply(
            new CreateBookEvt(cmd.getIdBook(),
                             cmd.getTitle(),
                             cmd.getPublish(),
                             cmd.getAuthors()));
    }
}
```



Handler

```
@Component
public class BookCommandHandler {

    private Repository<Book> repository;
    private EventBus eventBus;
    private BookRepository bookRepository;

    public void setRepository(Repository<Book> repository) {}

    public void setEventBus(EventBus eventBus) {}

    public void setBookRepository(BookRepository bookRepository) {}

    @CommandHandler
    public void on(AddAuthorToBookCmd cmd) {
        try {
            Aggregate<Book> aggregate = repository.load(cmd.getIdBook());
            aggregate.execute(b -> b.addAuthor(cmd.getFullname()));
        } catch (AggregateNotFoundException ex) {
            ex.printStackTrace();
        }
    }
}
```



AGREGADO


Todo agregado debe contar con un identificador.

Los identificadores generalmente son del String o Number. También pueden ser objetos (consultar la documentación).

En el ejemplo el agregado presenta una definición de captura.

Es importante que el agregado cuente con un constructor vacío, para que el repositorio de agregados posteriormente pueda instanciarlo.

Toda la lógica de negocio debe estar implementada en los eventos producidos por comandos.



```
@Aggregate(snapshotTriggerDefinition = "classicSnapshotter")
@NoArgsConstructor
@Getter
public class Book {

    @AggregateIdentifier
    private String idBook;
    private String title;
    private Date publish;

    private List<Author> authors;

    @CommandHandler
    public Book(CreateBookCmd cmd) {
        AggregateLifecycle.apply(
            new CreateBookEvt(cmd.getIdBook(),
                cmd.getTitle(),
                cmd.getPublish(),
                cmd.getAuthors()));
    }

    public void on(AddAuthorToBookCmd cmd) {}

    public void on(CreateBookEvt evt) {}

    public void on(AddAuthorToBookEvt evt) {}

    public void addAuthor(String fullname) {}
}
```



QUERY BUS / GATEWAY

- ▶ Interfaz encargada de transportar los mensajes de consulta.
- ▶ Este bus proporciona las siguientes formas de trabajo:
 - ▶ Direct Query: Invoca solo a un receptor de consulta y devuelve su resultado.
 - ▶ Scatter Gatter Query: Invocara a todos los receptores de consulta y devolverá los resultados de cada uno de ellos.
 - ▶ Subscription Query: Permite obtener el estado inicial de la consulta (Mono) y se actualiza cuando se emiten cambios en la consulta (Flux). Esta integrado con el proyecto Reactor.

```
FetchAllBooksQry query = new FetchAllBooksQry();
CompletableFuture<List<BookDocument>> future1 =
    this.queryGateway.query(query,
        ResponseTypes.multipleInstancesOf(BookDocument.class));

CompletableFuture<QueryResponseMessage<List<BookDocument>>> future2 =
    this.queryBus.query(new GenericQueryMessage<>(query,
        ResponseTypes.multipleInstancesOf(BookDocument.class)));

Mono<List<BookDocument>> initialResult =
    this.queryGateway
        .subscriptionQuery(query,
            ResponseTypes.multipleInstancesOf(BookDocument.class),
            ResponseTypes.multipleInstancesOf(BookDocument.class))
        .initialResult();

Flux<List<BookDocument>> updates =
    this.queryGateway
        .subscriptionQuery(query,
            ResponseTypes.multipleInstancesOf(BookDocument.class),
            ResponseTypes.multipleInstancesOf(BookDocument.class))
        .updates();
```


QUERY HANDLER



```
@Component
public class BookQueryHandler {

    private BookDocumentRepository bookDocumentRepository;

    @Autowired
    public void setBookDocumentRepository(BookDocumentRepository bookDocumentRepository) {
        this.bookDocumentRepository = bookDocumentRepository;
    }

    @QueryHandler
    public List<BookDocument> fetchAll(FetchAllBooksQry query) {
        return bookDocumentRepository.findAll();
    }

    @QueryHandler
    public BookDocument fetchById(FetchBookByIdQry query) {
        return bookDocumentRepository.findById(query.getIdBook())
            .orElse(null)
            ;
    }
}
```

EVENT BUS



- ▶ Interfaz encargada de transportar los mensajes de eventos.
- ▶ Se la invoca luego de haber recepcionado un comando.

EVENT BUS



Agregado - AggregateLifecycle

```
@Aggregate(snapshotTriggerDefinition = "classicSnapshotter")
@NoArgsConstructor
@Getter
public class Book {

    @AggregateIdentifier
    private String idBook;
    private String title;
    private Date publish;

    private List<Author> authors;

    @CommandHandler
    public Book(CreateBookCmd cmd) {
        AggregateLifecycle.apply(
            new CreateBookEvt(cmd.getIdBook(),
                cmd.getTitle(),
                cmd.getPublish(),
                cmd.getAuthors()));
    }
}
```

Event Bus

```
@Component
public class BookCommandHandler {

    private Repository<Book> repository;
    private EventBus eventBus;
    private BookRepository bookRepository;

    public void setRepository(Repository<Book> repository) {}

    public void setEventBus(EventBus eventBus) {}

    public void setBookRepository(BookRepository bookRepository) {}

    public void on(AddAuthorToBookCmd cmd) {}

    public void on(TransferBookCmd cmd) {}

    @CommandHandler
    public void on(IncreaseStockCmd cmd) {
        try {
            bookRepository.increaseStock(cmd.getIdBook());
            eventBus.publish(asEventMessage(new OkIncreaseStockEvt(cmd.getIdBook())));
        } catch (Exception ex) {
            eventBus.publish(asEventMessage(new ErrorIncreaseStockEvt(cmd.getIdBook())));
        }
    }
}
```



EVENT HANDLER

Event Handler

```
@Component
public class BookEventHandler {

    private QueryBus queryBus;

    private BookDocumentRepository bookDocumentRepository;

    public void setQueryBus(QueryBus queryBus) {}

    public void setBookDocumentRepository(BookDocumentRepository bookDocumentRepository) {}

    @EventHandler
    public void on(UpdateAllSubscriptionQueryEvt evt) {
        BookDocument book = bookDocumentRepository
            .findById(evt.getIdBook())
            .orElse(null);

        // ACTUALIZAMOS TODAS LAS QUERYS SUSCRITAS A LA CONSULTA FetchAllBooks
        this.queryBus
            .queryUpdateEmitter()
            .emit(FetchAllBooksQry.class, f -> {return true;}, book);

        // ACTUALIZAMOS TODAS LAS QUERYS SUSCRITAS A LA CONSULTA FetchBookById
        this.queryBus
            .queryUpdateEmitter()
            .emit(FetchBookByIdQry.class, f -> {return f.getIdBook().equalsIgnoreCase(book.getIdBook());}, book);
    }
}
```

Event Sourcing Handler - Agregado

```
@Aggregate(snapshotTriggerDefinition = "classicSnapshotter")
@NoArgsConstructor
@Getter
public class Book {

    @AggregateIdentifier
    private String idBook;
    private String title;
    private Date publish;

    private List<Author> authors;

    public Book(CreateBookCmd cmd) {}

    public void on(AddAuthorToBookCmd cmd) {}

    @EventSourcingHandler
    public void on(CreateBookEvt evt) {
        this.idBook = evt.getIdBook();
        this.title = evt.getTitle();
        this.publish = evt.getPublish();
        this.authors = new ArrayList<>();
        if(evt.getAuthors() != null && !evt.getAuthors().isEmpty()) {
            evt.getAuthors()
                .stream()
                .forEach(c -> this.authors.add(new Author(UUID.randomUUID().toString(), c)));
        }
    }
}
```

SAGA – EVENT HANDLER

Toda Saga tiene un inicio y un final.

La Saga puede persistir en base de datos para su posterior recuperación en caso de fallos.

La Saga persiste a través de la interfaz SagaStore.

La Saga recibe eventos y emite comandos.

Hay dos maneras de terminar una saga:

Declarativa - Anotaciones

Programática – SagaLifecycle.end();

La saga necesita de propiedades getters y setters para reiniciar la saga, en caso esta haya persistido en base de datos.



```
@Saga
@Getter
@Setter
@NoArgsConstructor
public class BookTransferSaga {

    private transient CommandBus commandBus;

    public void setCommandBus(CommandBus commandBus) {}

    private BookTransferData data;

    @StartSaga
    @SagaEventHandler(associationProperty = "idBook")
    public void on(TransferBookEvt evt) {
        System.out.println("Start " + SagaLifecycle.describeCurrentScope().scopeDescription());
        this.data = new BookTransferData(evt.getIdBook(), evt.getTitle(), evt.getPublish(), evt.getIdAuthors(), evt.getIdBook());
        SagaLifecycle.associateWith("idBook", this.data.getIdBook());
        increaseStock();
    }

    // ----- PASO 1 -----

    @SagaEventHandler(associationProperty = "idBook")
    public void on(OkIncreaseStockEvt evt) {
        saveBookQuery();
    }

    @EndSaga
    @SagaEventHandler(associationProperty = "idBook")
    public void on(ErrorIncreaseStockEvt evt) {
        // NO SE COMPENSA NADA DEBIDO A QUE ES EL PRIMER COMMIT
        System.out.println("End " + SagaLifecycle.describeCurrentScope().scopeDescription());
    }

    // ----- PASO 2 -----

    @SagaEventHandler(associationProperty = "idBook")
    public void on(OkSaveBookQueryEvt evt) {
        saveAuthorQuery();
    }
}
```

REPOSITORY

Repositorio de agregados.

Encargada de reconstruir el agregado a su estado actual a través de eventos ordenados y previamente almacenados.

Entre sus implementaciones tenemos:

EventSourcingRepository

CachingEventSourcingRepository

GenericJpaRepository

Existen dos maneras de usarlo:

Automática: Por Axon.

Programática: Por el desarrollador.



```
@Component
public class BookCommandHandler {

    private Repository<Book> repository;
    private EventBus eventBus;
    private BookRepository bookRepository;

    public void setRepository(Repository<Book> repository) {}

    public void setEventBus(EventBus eventBus) {}

    public void setBookRepository(BookRepository bookRepository) {}

    public void on(AddAuthorToBookCmd cmd) {}

    @CommandHandler
    public void on(TransferBookCmd cmd) {
        try {
            Aggregate<Book> aggregate = repository.load(cmd.getIdBook());

            aggregate.execute(b -> {
                TransferBookEvt evt = new TransferBookEvt(
                    b.getIdBook(),
                    b.getTitle(),
                    b.getPublish(),
                    b.getAuthors().stream().map(Author::getIdAuthor).collect(Collectors.toList()),
                    b.getAuthors().stream().map(Author::getFullname).collect(Collectors.toList()));
                eventBus.publish(asEventMessage(evt));
            });
        } catch (AggregateNotFoundException ex) {
            ex.printStackTrace();
        }
    }
}
```

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class TransferBookCmd {

    @TargetAggregateIdentifier
    private String idBook;
}
```



EVENT STORE

Permite la recuperación de los eventos almacenados en una base de datos.

La gestión del almacenamiento y lectura se lo delega a EventStorageEngine.

El EventStorageEngine tiene implementaciones para trabajar con bases de datos SQL, MongoDB, Memoria, entre otros.

El event store permite reconstruir el agregado hasta su estado actual.

```
public class EventSourcingRepository<T> extends LockingRepository<T, EventSourcedAggregate<T>> {

    private final EventStore eventStore;
    private final SnapshotTriggerDefinition snapshotTriggerDefinition;
    private final AggregateFactory<T> aggregateFactory;
    private final RepositoryProvider repositoryProvider;

    * Instantiate a {@link EventSourcingRepository} based on the fields contained in the {@link Builder}.
    protected EventSourcingRepository(Builder<T> builder) {}

    * Instantiate a Builder to be able to create a {@link EventSourcingRepository} for aggregate type {@code T}.
    public static <T> Builder<T> builder(Class<T> aggregateType) {}

    * Perform the actual loading of an aggregate. The necessary locks have been obtained.
    @Override
    protected EventSourcedAggregate<T> doLoadWithLock(String aggregateIdentifier, Long expectedVersion) {
        DomainEventStream eventStream = readEvents(aggregateIdentifier);
        SnapshotTrigger trigger = snapshotTriggerDefinition.prepareTrigger(aggregateFactory.getAggregateType());
        if (!eventStream.hasNext()) {
            throw new AggregateNotFoundException(aggregateIdentifier, "The aggregate was not found in the event s
        }
        EventSourcedAggregate<T> aggregate = EventSourcedAggregate
            .initialize(aggregateFactory.createAggregateRoot(aggregateIdentifier, eventStream.peek()),
                aggregateModel(), eventStore, repositoryProvider, trigger);
        aggregate.initializeState(eventStream);
        if (aggregate.isDeleted()) {
            throw new AggregateDeletedException(aggregateIdentifier);
        }
        return aggregate;
    }

    * Reads the events for the given aggregateIdentifier from the eventStore. this method may be overridden to
    protected DomainEventStream readEvents(String aggregateIdentifier) {
        return eventStore.readEvents(aggregateIdentifier);
    }
}
```

SNAPSHOTTING



- ▶ Mejora para tratar agregados que para ser reconstruidos tienen una gran cantidad mensajes (ejm: 10000).
- ▶ El snapshotting realiza una captura del agregado y lo almacena en la base de datos, serializándolo generalmente como json o xml.
- ▶ Snapshotter interfaz encargada de realizar el snapshot o captura del agregado.
- ▶ SnapshotTriggerDefinition encargada de decir cuando se realiza el snapshot.

```
@Configuration
public class AxonSnapshotConfig {

    @Bean
    public Snapshotter snapshotter(ApplicationContext ctx) {
        SpringAggregateSnapshotter snapshotter = SpringAggregateSnapshotter
            .builder()
            .eventStore(ctx.getBean(EventStore.class))
            .build();
        snapshotter.setApplicationContext(ctx);
        return snapshotter;
    }

    @Bean("classicSnapshotter")
    public SnapshotTriggerDefinition snapshotTriggerDefinition(Snapshotter snapshotter) {
        return new EventCountSnapshotTriggerDefinition(snapshotter, 5);
    }

    @Aggregate(snapshotTriggerDefinition = "classicSnapshotter")
    @NoArgsConstructor
    @Getter
    public class Book {

        @AggregateIdentifier
        private String idBook;
        private String title;
        private Date publish;
        private List<Author> authors;

        @EventHandler
        public void on(SnapshotBookEvt snapshot) {
            this.idBook = snapshot.getIdBook();
            this.publish = snapshot.getPublish();
            this.title = snapshot.getTitle();
            this.authors = snapshot.getAuthors();
        }
    }
}
```


UPCASTING



- ▶ Permite transformar eventos almacenados de una versión a otra, sobre todo cuando se modifican los atributos del evento.

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Revision("1.0")
public class CreateBookEvt {

    private String idBook;
    private String title;
    private Date publish;
    private List<String> authors;
}
```

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Revision("2.0")
public class CreateBookEvt {

    private String idBook;
    private String title;
    private Date publish;
    private List<String> authors;
    private String isbn; // nuevo campo
}
```

```
@Component
public class CreateBookEvtUpcaster extends SingleEventUpcaster {

    private static SimpleSerializedType targetType =
        new SimpleSerializedType(CreateBookEvt.class.getTypeName(), "1.0");

    @Override
    protected boolean canUpcast(IntermediateEventRepresentation intermediateRepresentation) {
        return intermediateRepresentation.getType().equals(targetType);
    }

    @Override
    protected IntermediateEventRepresentation doUpcast(IntermediateEventRepresentation intermedi
        return intermediateRepresentation
            .upcastPayload(new SimpleSerializedType(targetType.getName(), "2.0"),
                JsonNode.class,
                json -> {
                    ((ObjectNode)json).put("isbn", "no-isbn-registration");
                    return json;
                });
    }
}
```

REFERENCIAS



- ▶ <https://docs.axoniq.io/reference-guide/>
- ▶ <https://dzone.com/articles/building-reactive-systems-with-axon-framework>
- ▶ <https://github.com/AxonFramework>



THANK YOU - GRACIAS