INSTITUTO INFNET

ESCOLA SUPERIOR DE TECNOLOGIA

LEANDRO MEDEIROS

ENGENHARIA DE SOFTWARE

DOMAIN-DRIVEN DESIGN (DDD) E ARQUITETURA DE SOFTWARES ESCALÁVEIS COM JAVA

Professor Armênio Torres Santiago Cardoso

Teste de Performance 3

Dezembro de 2025

Índice

01. Classe base de evento: Crie um trecho de código Java para uma classe base de um evento a ser persistido em um event store.

```java
package br.com.petfriends.sales_api.coreapi.base;

import lombok.Getter;

@Getter
public abstract class BaseEvent<T> {

    private final T id;

    public BaseEvent(T id) {
        this.id = id;
    }
}
```

Explicação: A classe abstrata `BaseEvent` funciona como a raiz de todos os fatos históricos do sistema. Sua finalidade é garantir a identidade, obrigando que todo evento carregue o ID do agregado ao qual pertence (Generics `<T>`), permitindo o rastreamento no Event Store.

02. Classe base de command: Crie um trecho de código Java para uma classe base de um command que vai provocar a persistência de um evento em um event store.

```java
package br.com.petfriends.sales_api.coreapi.base;

import org.axonframework.modelling.command.TargetAggregateIdentifier;
import lombok.Getter;

@Getter
public abstract class BaseCommand<T> {

    @TargetAggregateIdentifier
    private final T id;

    public BaseCommand(T id) {
        this.id = id;
    }
}
```

Explicação: A classe abstrata `BaseCommand` define a estrutura fundamental de qualquer **intenção de mudança** no sistema. A anotação `@TargetAggregateIdentifier` é o componente crucial: ela serve como "endereço de entrega", informando ao framework Axon qual instância específica do Agregado deve ser carregada para processar este comando.

03. Classe de evento: Crie um trecho de código Java para uma classe que estenda a classe base de evento da questão 2.

```java
package br.com.petfriends.sales_api.coreapi.events;

import br.com.petfriends.sales_api.coreapi.base.BaseEvent;
import lombok.Getter;
import java.math.BigDecimal;

@Getter
public class PedidoCriadoEvent extends BaseEvent<String> {

    private final String clienteId;
    private final BigDecimal valorTotal;
    private final String status;

    public PedidoCriadoEvent(String id, String clienteId, BigDecimal
valorTotal) {
        super(id);
        this.clienteId = clienteId;
        this.valorTotal = valorTotal;
        this.status = "CRIADO";
    }
}
```

Explicação: A classe `PedidoCriadoEvent` materializa um **fato histórico** específico do domínio Pet Friends: a criação de um pedido. Ela estende `BaseEvent` para herdar a identidade e carrega apenas os dados imutáveis (primitivos ou Value Objects) necessários para informar o restante do sistema (como `valorTotal` e `clienteId`) sobre o que acabou de acontecer.

04. Classe de command: Crie um trecho de código Java para uma classe que estenda a classe base de command da questão 3.

```java
package br.com.petfriends.sales_api.coreapi.commands;

import br.com.petfriends.sales_api.coreapi.base.BaseCommand;
import lombok.Getter;
import java.math.BigDecimal;

@Getter
public class CriarPedidoCommand extends BaseCommand<String> {

    private final String clienteId;
    private final BigDecimal valorTotal;

    public CriarPedidoCommand(String id, String clienteId, BigDecimal
valorTotal) {
        super(id);
        this.clienteId = clienteId;
        this.valorTotal = valorTotal;
    }
}
```

Explicação: A classe `CriarPedidoCommand` representa a intenção imperativa de iniciar um processo de negócio: criar um novo pedido para um cliente. Ela estende `BaseCommand` para herdar o identificador de destino e carrega o "payload" (carga útil) — neste caso, o `clienteId` e o `valorTotal` — dados essenciais para que o Agregado possa validar a regra de negócio e decidir se aceita ou rejeita o comando.

05. Classe command service: Crie uma classe de serviço que utilize o CommandGateway para executar o command criado na questão 3.

```
package br.com.petfriends.sales_api.command.services;

import br.com.petfriends.sales_api.coreapi.commands.CriarPedidoCommand;
import org.axonframework.commandhandling.gateway.CommandGateway;
import org.springframework.stereotype.Service;

import java.math.BigDecimal;
import java.util.UUID;
import java.util.concurrent.CompletableFuture;

@Service
public class PedidoCommandService {

    private final CommandGateway commandGateway;

    public PedidoCommandService(CommandGateway commandGateway) {
        this.commandGateway = commandGateway;
    }

    public CompletableFuture<String> criarPedido(String clienteId, BigDecimal
valorTotal) {
        String pedidoId = UUID.randomUUID().toString();

        CriarPedidoCommand command = new CriarPedidoCommand(pedidoId,
clienteId, valorTotal);

        return commandGateway.send(command);
    }
}
```

Explicação: O `PedidoCommandService` atua como um **orquestrador de entrada.** Ele não contém regras de negócio complexas; sua função é converter a solicitação externa (geralmente vinda da API REST) em um Comando explícito (`CriarPedidoCommand`). Ao utilizar o `CommandGateway`, ele despacha essa intenção para o Command Bus do Axon, desacoplando completamente quem pede (API) de quem executa (Agregado). O retorno `CompletableFuture` permite que a operação seja assíncrona.

06. Classe do agregado 1: Crie um trecho de código Java para uma classe que implemente um CommandHandler que use o command criado na questão 5.

```java
package br.com.petfriends.sales_api.command.aggregates;

import br.com.petfriends.sales_api.coreapi.commands.CriarPedidoCommand;
import br.com.petfriends.sales_api.coreapi.events.PedidoCriadoEvent;
import org.axonframework.commandhandling.CommandHandler;
import org.axonframework.eventsourcing.EventSourcingHandler;
import org.axonframework.modelling.command.AggregateIdentifier;
import org.axonframework.modelling.command.AggregateLifecycle;
import org.axonframework.spring.stereotype.Aggregate;

import java.math.BigDecimal;

@Aggregate
public class PedidoAggregate {

    @AggregateIdentifier
    private String id;

    private BigDecimal valorTotal;
    private String status;

    public PedidoAggregate() {
    }

    @CommandHandler
    public PedidoAggregate(CriarPedidoCommand command) {
        // Validação de Invariante: Não existe pedido grátis.
        if (command.getValorTotal().compareTo(BigDecimal.ZERO) ≤ 0) {
            throw new IllegalArgumentException("O valor do pedido deve ser
maior que zero.");
        }

        AggregateLifecycle.apply(new PedidoCriadoEvent(
                command.getId(),
                command.getClienteId(),
                command.getValorTotal()));
    }

    @EventSourcingHandler
    public void on(PedidoCriadoEvent event) {
        this.id = event.getId();
        this.valorTotal = event.getValorTotal();
        this.status = event.getStatus();
    }
}
```

Explicação: O Agregado `PedidoAggregate` é a **fronteira de consistência transacional** do sistema. O método anotado com `@CommandHandler` atua como o interceptador da intenção de criação. Sua finalidade principal é **proteger as invariantes de negócio** (como validar se o valor é positivo). Se a regra for satisfeita, ele não altera o estado diretamente; em vez disso, ele usa o `AggregateLifecycle.apply()` para registrar que um evento (`PedidoCriadoEvent`) ocorreu, iniciando o ciclo de persistência e notificação.

07. Classe do agregado 2: Crie um trecho de código Java para a mesma classe da questão 6 para implementar um EventSourcingHandler que responda à criação do evento da questão 4.

```java
// ... (Código anterior da Q6)

@EventSourcingHandler
public void on(PedidoCriadoEvent event) {
    this.id = event.getId();
    this.valorTotal = event.getValorTotal();
    this.status = event.getStatus();
}
```

Explicação: O método anotado com `@EventSourcingHandler` é o responsável pela **mutação de estado** do Agregado. No padrão Event Sourcing, o estado atual não é salvo no banco; ele é derivado do histórico. Quando o Agregado é carregado, o Axon "re-toca" todos os eventos passados através deste método para reconstruir o objeto (processo chamado de *Rehydration*). É aqui, e apenas aqui, que as variáveis de instância (`this.id`, `this.status`) são alteradas.

08. Classe query service 1: Crie um serviço de consulta que mostre todos os eventos da entidade criada na questão 7.

```java
package br.com.petfriends.sales_api.query.pedido;

import br.com.petfriends.sales_api.coreapi.events.PedidoCriadoEvent;
import br.com.petfriends.sales_api.coreapi.models.FindAllPedidosQuery;
import br.com.petfriends.sales_api.coreapi.models.FindPedidoByIdQuery;

import org.axonframework.eventhandling.EventHandler;
import org.axonframework.queryhandling.QueryHandler;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class PedidoProjector {

    private final PedidoRepository repository;

    public PedidoProjector(PedidoRepository repository) {
        this.repository = repository;
    }

    @EventHandler
    public void on(PedidoCriadoEvent event) {
        PedidoProjection projection = new PedidoProjection(
                event.getId(),
                event.getClienteId(),
                event.getValorTotal(),
                event.getStatus());
        repository.save(projection);
    }

    @QueryHandler
    public List<PedidoProjection> handle(FindAllPedidosQuery query) {
        return repository.findAll();
    }

    @QueryHandler
    public PedidoProjection handle(FindPedidoByIdQuery query) {
        return repository.findById(query.getId())
                .orElse(null);
    }
}
```

Explicação: A classe PedidoProjector atua como um **sincronizador de modelos.** No CQRS, a escrita e a leitura são separadas.

1. **Escrita (EventHandler)**: O método `on` escuta o evento `PedidoCriadoEvent` (emitido pelo Agregado) e materializa esses dados na tabela de consulta (`PedidoProjection`). Isso garante a consistência eventual.
2. **Leitura (QueryHandler)**: O método `handle` responde à pergunta "Dê-me todos os pedidos", acessando o repositório otimizado para leitura, sem sobrecarregar o Event Store.

09. Interface repository: Crie um JpaRepository para cuidar da persistência da entidade criada na questão 7.

```java
package br.com.petfriends.sales_api.query.pedido;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface PedidoRepository extends JpaRepository<PedidoProjection,
String> {
}
```

Explicação: A interface `PedidoRepository` é a ponte entre o código Java e o banco de dados relacional (H2). Ao estender `JpaRepository`, herdamos automaticamente dezenas de métodos prontos para salvar e buscar a entidade `PedidoProjection`, eliminando a necessidade de escrever SQL manual. É este repositório que o *Projector* usa para salvar os dados e que os *Controllers* usam (indiretamente) para buscar informações.

10. Classe query service 2: Crie no serviço de consulta da questão 10 um método para obter o registro da entidade da questão 7 pelo ID.

```
package br.com.petfriends.sales_api.query.pedido;

import br.com.petfriends.sales_api.coreapi.events.PedidoCriadoEvent;
import br.com.petfriends.sales_api.coreapi.models.FindAllPedidosQuery;
import br.com.petfriends.sales_api.coreapi.models.FindPedidoByIdQuery;

import org.axonframework.eventhandling.EventHandler;
import org.axonframework.queryhandling.QueryHandler;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class PedidoProjector {

    // ... (Código anterior)

    @QueryHandler
    public PedidoProjection handle(FindPedidoByIdQuery query) {
        return repository.findById(query.getId())
                .orElse(null);
    }
}
```

Explicação: Este método adiciona granularidade ao modelo de leitura. Enquanto o método anterior listava tudo, este `QueryHandler` responde especificamente à pergunta "Quero os detalhes do pedido X". Ele utiliza o `findById` do repositório JPA. Note que ele recebe um objeto `FindPedidoByIdQuery` (que carrega o ID) e retorna a projeção (`PedidoProjection`) correspondente, completando a separação de responsabilidades onde a leitura é direta e eficiente.

11. **Classe Rest Controller 1:** Crie uma rota Rest em um controller que execute o command criado na questão 3.

```
package br.com.petfriends.sales_api.controllers;

import br.com.petfriends.sales_api.command.services.PedidoCommandService;
import lombok.Data;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.math.BigDecimal;
import java.util.concurrent.CompletableFuture;

@RestController
@RequestMapping("/orders")
public class OrderCommandController {

    private final PedidoCommandService commandService;

    public OrderCommandController(PedidoCommandService commandService) {
        this.commandService = commandService;
    }

    @PostMapping
    public CompletableFuture<ResponseEntity<String>> criarPedido(@RequestBody
PedidoDTO dto) {
        return commandService.criarPedido(dto.getClienteId(),
dto.getValorTotal())
                .thenApply(id ->
ResponseEntity.status(HttpStatus.CREATED).body(id));
    }

    @Data
    static class PedidoDTO {
        private String clienteId;
        private BigDecimal valorTotal;
    }
}
```

Explicação: O `OrderCommandController` atua como o Adaptador de Entrada (na arquitetura Hexagonal/Ports & Adapters). Ele expõe uma API REST pública (rota `POST /orders`) para o mundo externo. Sua responsabilidade é puramente de infraestrutura: receber o JSON (`@RequestBody`), chamar o serviço de domínio (`PedidoCommandService`) e devolver a resposta HTTP adequada (`201 Created`). Note o uso de

`CompletableFuture`: como o barramento de comandos é assíncrono, o controller não bloqueia a thread enquanto espera o Agregado processar a regra.

12. **Classe Rest Controller:** Crie uma rota Rest em um controller que execute a query criada na questão 8 e outra que execute a query da questão 10.

```java
package br.com.petfriends.sales_api.controllers;

import br.com.petfriends.sales_api.coreapi.models.FindAllPedidosQuery;
import br.com.petfriends.sales_api.coreapi.models.FindPedidoByIdQuery;
import br.com.petfriends.sales_api.query.pedido.PedidoProjection;
import org.axonframework.messaging.responsetypes.ResponseTypes;
import org.axonframework.queryhandling.QueryGateway;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;
import java.util.concurrent.CompletableFuture;

@RestController
@RequestMapping("/orders")
public class OrderQueryController {

    private final QueryGateway queryGateway;

    public OrderQueryController(QueryGateway queryGateway) {
        this.queryGateway = queryGateway;
    }

    @GetMapping
    public CompletableFuture<List<PedidoProjection>> listarTodos() {
        return queryGateway.query(
                new FindAllPedidosQuery(),
                ResponseTypes.multipleInstancesOf(PedidoProjection.class));
    }

    @GetMapping("/{id}")
    public CompletableFuture<PedidoProjection> buscarPorId(@PathVariable String
id) {
        return queryGateway.query(
                new FindPedidoByIdQuery(id),
                ResponseTypes.instanceOf(PedidoProjection.class));
    }
}
```

Explicação: O `OrderQueryController` materializa o Lado de Leitura (Read Side) do padrão CQRS. Ao contrário de um controller MVC

tradicional que chamaria um Repository diretamente, este componente utiliza o `QueryGateway` para despachar mensagens de consulta (`FindAllPedidosQuery` e `FindPedidoByIdQuery`). Isso mantém o desacoplamento total: o Controller não sabe onde os dados estão armazenados (se no H2, ElasticSearch ou Mongo), ele apenas pergunta ao barramento. O uso de `ResponseTypes` é crucial para informar ao Axon qual o formato esperado da resposta (uma lista ou um objeto único).