Documentação Técnica: Microserviço report-request-service

Autor: alex caje feix
Versão: 1.0
Data: 24 de Junho de 2025

1. Visão Geral

O **report-request-service** é um microsserviço projetado para orquestrar a solicitação e o processamento de relatórios. Sua principal responsabilidade é atuar como um gateway, recebendo requisições via uma API REST, validando-as, persistindo o estado inicial da solicitação e publicando um evento em um tópico Kafka para processamento assíncrono por outros serviços.
O projeto foi desenvolvido com um forte embasamento nos princípios da **Clean Architecture** e **Domain-Driven Design (DDD)**, visando alta coesão, baixo acoplamento e manutenibilidade a longo prazo.

2. Arquitetura e Filosofia de Design

A arquitetura do serviço é dividida em camadas lógicas (Domínio, Aplicação, Infraestrutura), promovendo uma clara separação de responsabilidades.

2.1. Clean Architecture

A estrutura segue a regra de dependência da Clean Architecture: as camadas internas (Domínio) não conhecem as camadas externas (Infraestrutura). Isso é alcançado através de abstrações (interfaces de casos de uso) que são implementadas pelas camadas externas.

2.2. Domain-Driven Design (DDD)

A filosofia do DDD foi aplicada para modelar o núcleo do negócio de forma rica e expressiva.

- **Entidade Rica:** A principal entidade, ReportRequest, não é apenas um contêiner de dados. Ela possui lógica de negócio e é responsável por garantir seu próprio estado de consistência através da auto-validação.
- **Linguagem Ubíqua:** Os nomes de classes e métodos (ex: ReportRequest, CreateReportUseCase, PublishToKafkaUseCase) refletem a linguagem do domínio de negócio.

3. Detalhamento das Camadas

3.1. Camada de Domínio (Domain)

Esta é a camada mais interna e importante, contendo a lógica de negócio pura.

3.1.1. Entidades Ricas e Auto-Validação

A entidade ReportRequest é o coração do domínio. Seguindo os princípios de Eric Evans e Martin Fowler, ela se valida no momento de sua criação.
// Em ReportRequest.java

```
public ReportRequest(String reportType, ...) {
   // ... atribuição de campos ...
   this.requestedAt = LocalDateTime.now();
   validateSelf(); // Garante que nenhum objeto inválido pode ser criado.
}
```

3.1.2. Padrão de Notificação de Erros (Notification Pattern)

Para evitar o lançamento de exceções no primeiro erro encontrado e fornecer um feedback mais completo ao cliente, foi implementado o **Notification Pattern**.

- **Estrutura:** Utiliza uma classe Notification que acumula objetos Error em uma lista.
- **Funcionamento:** Um Validator (implementado por ReportRequestValidator) realiza todas as checagens na entidade, adicionando cada erro encontrado à Notification. Apenas no final do processo, se a lista de erros não estiver vazia, uma única exceção customizada é lançada, contendo todas as mensagens de erro concatenadas.
- **Benefício:** Permite que a API retorne todos os erros de validação de uma só vez, melhorando a experiência do usuário.

3.1.3. Exceções Customizadas e RFC 7807

O serviço utiliza exceções customizadas (ReportRequestInvalidException, MessagePublishingException) que não carregam a StackTrace. Esta é uma otimização de performance importante para exceções de fluxo de controle, onde a pilha de chamadas completa não é necessária.
Um @ControllerAdvice global intercepta essas exceções e as formata em uma resposta HTTP padronizada, seguindo o padrão **RFC 7807 (Problem Details for HTTP APIs)**, garantindo consistência no tratamento de erros da API.

3.2. Camada de Aplicação (Application)

Esta camada orquestra o fluxo de dados e coordena as entidades de domínio para realizar as tarefas de negócio.

3.2.1. Casos de Uso (Use Cases) Abstratos

Para reforçar a Clean Architecture, foram criadas classes abstratas que definem os "contratos" dos casos de uso, como:

- InputOutputUseCase<In, Out>
- InputNoOutputUseCase<In>
- NoInputOutputUseCase<Out>

Isso permite que as implementações concretas (ex: CreateReportUseCaseImpl) sejam facilmente substituídas e testadas, e que a camada de infraestrutura (Controller) dependa apenas das abstrações.

3.2.2. Fluxo de Execução Assíncrono

O processo de criação e publicação é totalmente assíncrono para não bloquear a thread da

requisição HTTP e garantir alta vazão.

1. O ReportRequestController recebe a requisição e chama o CreateReportUseCase.
2. O CreateReportUseCaseImpl.execute() é anotado com @Async, liberando a thread do controller imediatamente.
3. Dentro dele, um CompletableFuture.supplyAsync é usado para invocar o PublishToKafkaUseCase, que também é assíncrono.
4. Isso cria uma pipeline reativa e não-bloqueante, onde a resposta é retornada ao cliente enquanto a publicação no Kafka ocorre em background.
5. Em caso de falha na publicação, o exceptionally do CompletableFuture trata o erro, evitando que a exceção se perca.

3.2.3. Resiliência e Retentativas

- **Circuit Breaker (Resilience4j):** O PublishToKafkaUseCaseUseCaseImp é protegido por um Circuit Breaker. Se o Kafka estiver indisponível, o circuito abre e as chamadas são redirecionadas para um método de fallback (sendToKafkaFallback), que atualiza o status da solicitação para FAILED.
- **Scheduler para Reenvio:** Um @Scheduled (resendPendingRequests) roda periodicamente para buscar solicitações com status FAILED e tenta reenviá-las, garantindo a entrega eventual das mensagens.

3.3. Camada de Infraestrutura (Infra/Controller)

A camada mais externa, responsável pela interação com o "mundo exterior".

3.3.1. API Endpoints

O ReportRequestController expõe dois endpoints:

- POST /api/report-requests: Recebe um CreateReportRequestDto, inicia o processo de criação de forma assíncrona e retorna uma resposta imediata (200 OK ou 202 Accepted).
- GET /api/report-requests/{id}: Permite consultar o estado atual de uma solicitação de relatório.

3.3.2. Configuração de Ambiente

O serviço é configurado com três perfis do Spring (profiles):

- dev: Para desenvolvimento local.
- test: Para execução de testes automatizados, usando configurações específicas como banco de dados em memória e Kafka embutido.
- prod: Para o ambiente de produção.

4. Estratégia de Testes

- **Padrão Triple-A:** Todos os testes seguem o padrão Arrange-Act-Assert para clareza e organização.
- **Testes em Memória:** Para agilidade, os testes unitários e de integração de componentes utilizam um banco de dados H2 em memória e o **Embedded Kafka** (spring-kafka-test), o que acelera significativamente a suíte de testes.

- **Nomenclatura:**
    - Variáveis: camelCase.
    - Nomes de métodos de teste: snake_case (ex: given_valid_input_when_execute_then_return_success).

5. Conclusão

O report-request-service é um exemplo robusto de aplicação de design moderno de software. As escolhas arquiteturais promovem um sistema resiliente, testável e manutenível. O projeto serve como um excelente estudo de caso para a implementação de Clean Architecture e padrões de Domain-Driven Design em um contexto de microsserviços reativos.