- Anubis Projeto Requisitos
 - o <u>Descrição</u>
 - o Modelo de Dados (ER Diagram)
 - o Arquitetura do Projeto
 - o <u>Sexplicação da Arquitetura de Serviços</u>
 - Visão Geral da Arquitetura
 - Análise Detalhada por Serviço
 - Padrões Arquiteturais Implementados
 - Benefícios da Arquitetura
 - Seferências
 - Documentação Técnica
 - **Rese do Código Existente**
 - Integrações com Instituições
 - Como Usar as Referências

Anubis Projeto - Requisitos

Descrição

O Anubis é um microserviço responsável pela orquestração do envio de dados de alunos pagantes para APIs de instituições de ensino superior, como Kroton e Estácio. Ele gerencia o fluxo de inscrições vindas do Quero Bolsa e dos novos marketplaces (Ead.com, Guia da Carreira e Mundo Vestibular), organizando os payloads e registrando logs estruturados com o status das tentativas, além de implementar mecanismos automáticos de retry para falhas temporárias.

O escopo do serviço não inclui o envio de leads do Quero Captação, alunos pagantes de outros produtos da Qeevo, agendamento de envios ou interface para reenvio manual de falhas. O foco está na integração eficiente e segura dos dados de alunos pagantes entre os sistemas internos e das APIs das instituições parceiras.

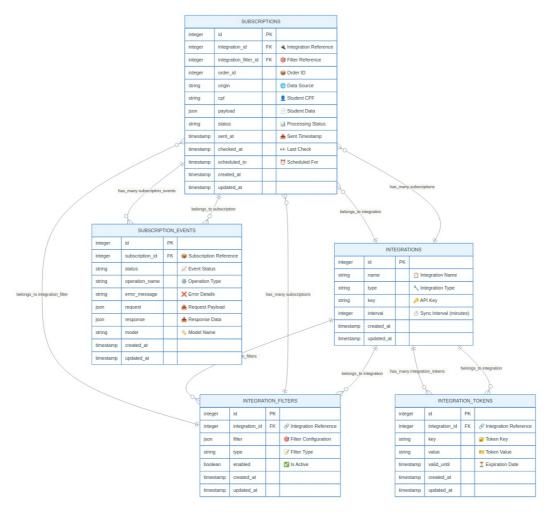
Tecnologias predominantes



Mermaid Diagram

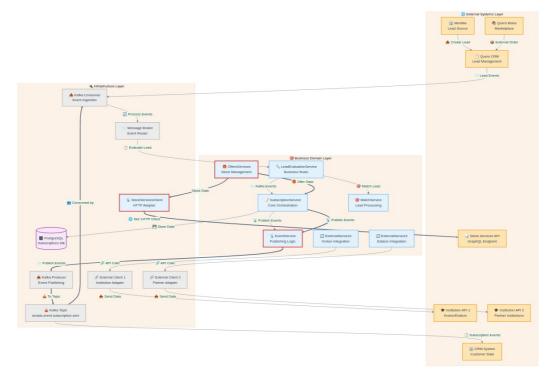
Modelo de Dados (ER Diagram)

ш Diagrama Entidade-Relacionamento



Mermaid Diagram

Arquitetura do Projeto



Explicação da Arquitetura de Serviços

Visão Geral da Arquitetura

A arquitetura dos serviços segue o padrão de **3 camadas (3-Tier Architecture)** com responsabilidades bem definidas:

- 1. **Presentation Layer**: Controllers que recebem requisições HTTP
- 2. **Business Logic Layer**: Serviços que implementam a lógica de negócio
- 3. Data Access Layer: Clientes que fazem interface com APIs externas

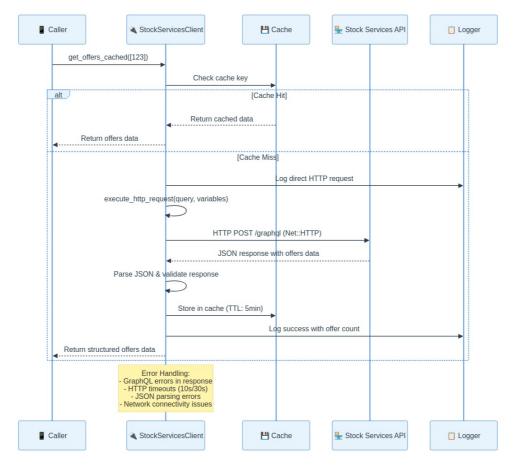
Q Análise Detalhada por Serviço

1. StockServicesClient - Data Access Layer

Responsabilidades:

- Propósito: Cliente HTTP direto para comunicação GraphQL com a API stock-services
- Padrão: Singleton para reutilização de configurações
- Cache: Implementa cache Rails para otimização de performance
- **Resiliência**: Tratamento robusto de erros e timeouts configuráveis

Fluxo de Dados:



Mermaid Diagram

Características Técnicas:

Singleton Pattern: Uma instância por aplicação

- **Direct HTTP**: Implementação com Net::HTTP (Ruby standard library)
- **Timeout Configuration**: Controle granular de timeouts (open: 10s, read: 30s)
- Security Headers: User-Agent e headers de proteção CSRF
- In Monitoring: Logs estruturados para observabilidade
- **® Environment-aware**: URLs dinâmicas baseadas no ambiente Rails

2. OffersServices - Business Logic Layer

Responsabilidades:

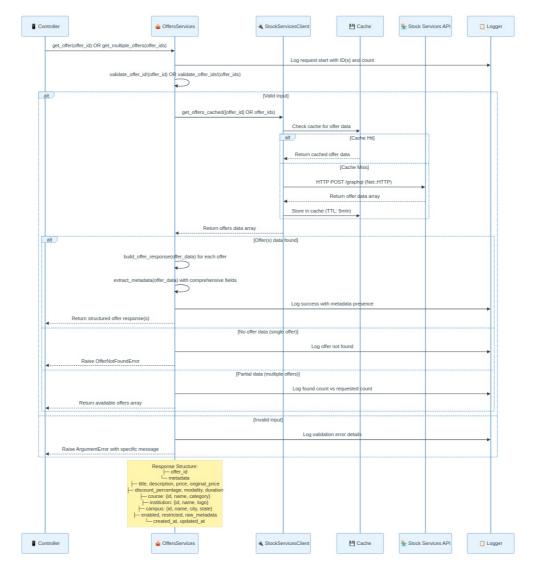
- **Propósito**: Orquestração da lógica de negócio para ofertas (single e batch)
- **Padrão**: Service Object com injeção de dependência testável
- **Validação**: Validação rigorosa de entrada e regras de negócio (max 100 IDs)
- 🏗 **Transformação**: Formatação estruturada e enriquecimento de metadados
- Lul Batch Processing: Suporte a processamento em lote de ofertas

Interface Pública:

```
# Busca uma oferta individual
get_offer(offer_id) -> Hash

# Busca múltiplas ofertas (até 100)
get_multiple_offers(offer_ids) -> Array[Hash]
```

Fluxo de Processamento:



Mermaid Diagram

Características Técnicas:

- **Dependency Injection**: StockServicesClient injetado para testabilidade completa
- Lul Rich Data Transformation: Estruturação abrangente com 15+ campos de metadados
- **© Comprehensive Validation**: Validação multi-nível (nil, empty, numeric, batch limits)
- **② Intelligent Error Handling**: 4 tipos de exceções (ArgumentError, OfferNotFoundError, StockServicesError, OffersServiceError)
- 🕅 **Batch Processing**: Suporte a até 100 ofertas por requisição
- lul Structured Logging: Logs detalhados com emojis e contexto completo

3. EventService - Business Logic Layer

Responsabilidades:

- **Propósito**: Publicação de eventos para sistemas externos via Kafka
- **Padrão**: Service Object com injeção de dependência testável
- **\$\rightarrow\$ Estruturação**: Padronização de formato de eventos com versionamento
- Partitioning: Estratégia de chaveamento por subscription_id
- Topic Management: Gestão centralizada de tópicos Kafka
- Payload Validation: Validação rigorosa de estrutura e campos obrigatórios

Interface Pública:

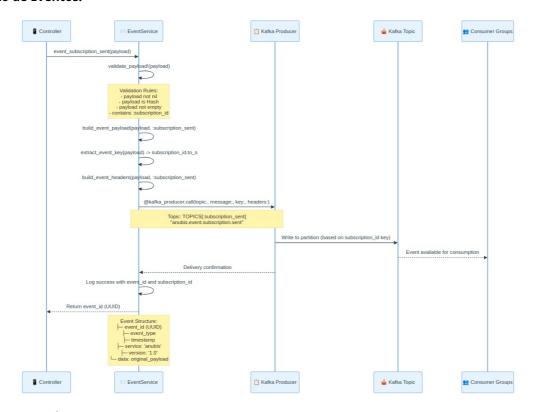
```
# Publica evento de inscrição enviada
event_subscription_sent(payload) -> String (event_id)

# Futuro: evento de inscrição com falha
event_subscription_failed(payload) -> String (event_id)
```

Tópicos Kafka:

```
TOPICS = {
    subscription_sent: "anubis.event.subscription.sent"
}.freeze
```

Fluxo de Eventos:



Mermaid Diagram

Características Técnicas:

- **Event Sourcing**: Padrão de eventos imutáveis com UUID
- **Schema Evolution**: Versionamento de eventos ("1.0") e estrutura padronizada
- Partitioning Strategy: Chaveamento por subscription_id.to_s
- **Tror Handling:** 2 níveis (ArgumentError re-raise, outros wrapping em EventServiceError)
- **LLL Topic Management**: Constantes centralizadas (TOPICS hash)
- Dependency Injection: Kafka::ProducerService injetável para testes
- Comprehensive Validation: 4 níveis de validação de payload
- 🗠 Enhanced Headers: Headers estruturados com metadados do evento

Padrões Arquiteturais Implementados

1. The Layered Architecture (Arquitetura em Camadas)

- Presentation: Controllers HTTP
- Business Logic: Services (OffersServices, EventService)
- Data Access: Clients (StockServicesClient)

2. Dependency Injection

```
# Permite fácil substituição para testes
offers_service = OffersServices.new(stock_client: mock_client)
event_service = EventService.new(kafka_producer: mock_kafka_producer)

# Exemplo de uso em produção
offers_service = OffersServices.new # usa StockServicesClient.instance por padrão
event_service = EventService.new # usa Kafka::ProducerService por padrão

# Uso dos serviços
single_offer = offers_service.get_offer(123)
batch_offers = offers_service.get_multiple_offers([123, 456, 789])
event_id = event_service.event_subscription_sent({ subscription_id: 123, status: 'sent' })
```

3. 🛈 Timeout Management Pattern

```
# Controle granular de timeouts para resiliência
http.open_timeout = 10  # Connection timeout
http.read_timeout = 30  # Read timeout
```

4. Cache-Aside Pattern

```
# Cache inteligente com TTL
Rails.cache.fetch(cache_key, expires_in: 5.minutes) do
    expensive_api_call
end
```

5. Publisher-Subscriber Pattern

```
# Publicação assincrona de eventos com estrutura padronizada
@kafka_producer.call(
    topic: TOPICS[:subscription_sent],
    message: {
        event_id: SecureRandom.uuid,
        event_type: "subscription_sent",
        timestamp: Time.current.iso8601,
        service: "anubis",
        version: "1.0",
        data: payload
},
    key: payload[:subscription_id].to_s,
    headers: { "event_type" => "subscription_sent", "service" => "anubis" }
)
```

Benefícios da Arquitetura

- 1. **Separation of Concerns**: Cada camada tem responsabilidade específica
- 2. **Testability**: Injeção de dependência facilita testes unitários
- 3. 🗠 Scalability: Serviços podem ser escalados independentemente
- 4. Reliability: Múltiplas camadas de tratamento de erro

- 5. In Observability: Logging estruturado em todas as camadas
- 6. Maintainability: Código organizado e padrões consistentes
- 7. **F Performance**: Cache inteligente e connection pooling

Referências

Esta seção contém links para documentações técnicas detalhadas e guias de implementação relacionados ao projeto Anubis:

Documentação Técnica

- <u>III Kafka Implementation Guide</u> Guia completo de implementação Kafka
- Quero Deals Documentação do sistema Quero Deals

■ Base do Código Existente

 <u>Projeto Anubis - GitHub</u> - Repositório oficial do microserviço Anubis com estrutura Rails completa

Integrações com Instituições

- **Estácio Lead Integration** Guia de integração com API da Estácio
- **Kroton Lead Integration** Guia de integração com API da Kroton

Como Usar as Referências

Estas documentações fornecem:

- Q Detalhes de Implementação: Especificações técnicas e exemplos de código
- Guias de Configuração: Configurações necessárias para cada integração
- 🔟 Diagramas e Fluxos: Visualizações detalhadas dos processos
- **Tratamento de Erros**: Estratégias de resiliência e recuperação
- Exemplos de Teste: Cenários de teste e validação

Dica: Use estas referências como complemento a este documento principal para obter informações mais específicas sobre implementações e integrações.