**Relatório Técnico de Desenvolvimento – API de Simulação de Crédito (API-Credito)**

**Data: 21 de agosto de 2025 Autor: Gemini, Assistente Sênior de IA Versão: 6.0 (Final Revisado)**

**1. Introdução e Objetivo**

**O presente relatório detalha o processo de desenvolvimento e a arquitetura da API-Credito, uma solução criada para atender ao desafio técnico de simulação de crédito. O objetivo central do projeto foi disponibilizar uma interface programática (API) robusta, resiliente e bem documentada, capaz de oferecer simulações de empréstimo para qualquer pessoa ou sistema no Brasil.**

**2. Arquitetura da Solução**

**A API foi estruturada seguindo os princípios da Arquitetura em Camadas (Layered Architecture) para garantir manutenibilidade, testabilidade e escalabilidade.**

* **Controller Layer: A porta de entrada da API, responsável por expor os endpoints REST, validar os dados de entrada (@Valid) e orquestrar as chamadas para os serviços.**
* **Service Layer: O cérebro da aplicação, onde reside toda a lógica de negócio. Esta camada é modularizada, contendo serviços específicos para:**
  + **Simulação (SimulacaoService): Orquestra o fluxo principal de criação de simulações.**
  + **Cálculos (CalculoAmortizacaoService): Isola a complexidade matemática dos cálculos de amortização.**
  + **Análise de Fraude (AnaliseFraudeService): Provê uma camada de segurança que analisa a legitimidade das solicitações.**
* **Repository Layer: A camada de acesso a dados. Utiliza o Spring Data JPA e queries customizadas com @Query no ProdutoRepository para otimizar a busca por produtos elegíveis.**
* **Model/Entity e DTO Layers: As camadas de Model representam as entidades do domínio (Produto, Simulacao), enquanto os DTOs definem os contratos seguros para a comunicação externa.**

**3. Qualidade e Estratégia de Testes**

**A qualidade e a resiliência da API foram garantidas através de uma estratégia de Testes Unitários, focada em validar os componentes da aplicação de forma rápida e isolada. Utilizando JUnit 5 e Mockito, foram criados testes para os cenários de sucesso e de exceção nas camadas de Serviço e Controller, validando a lógica de negócio, a interação com o serviço de fraude e o contrato da API.**

**4. Ciclo de Vida de uma Requisição de Simulação**

**O fluxo de uma chamada de simulação foi projetado com segurança e auditoria como prioridades:**

1. **Requisição e Validação: Um cliente envia uma requisição POST para /api/simulacoes. O SimulacaoController recebe e valida os dados de entrada.**
2. **Análise de Fraude (Primeira Barreira): O SimulacaoService imediatamente invoca o AnaliseFraudeService para avaliar a requisição. Se a análise retornar um status NEGADO, o processo é interrompido e uma exceção é lançada, bloqueando a simulação antes mesmo de consultar o banco de dados de produtos.**
3. **Lógica de Negócio e Persistência Inicial: Com a aprovação da análise de fraude, o serviço executa a query otimizada para encontrar o produto, realiza os cálculos de amortização e cria uma nova entidade Simulacao. Esta entidade é salva no banco de dados com o status da análise de fraude (APROVADO ou REQUER\_ANALISE\_MANUAL) e o status de envio para o Event Hub como AGUARDANDO\_ENVIO.**
4. **Comunicação Externa (Assíncrona): O serviço invoca o EventHubService de forma não-bloqueante (assíncrona), fornecendo "callbacks" para registrar o sucesso ou falha do envio.**
5. **Resposta Imediata ao Cliente: A API retorna a resposta ao cliente com o status 201 Created e o corpo da simulação, garantindo baixa latência.**
6. **Atualização de Status (em Background): Quando o Event Hub confirma o recebimento, o callback de sucesso é acionado em uma nova transação, atualizando o status da simulação no banco para ENVIADO, completando a trilha de auditoria.**

**5. Documentação da API (Swagger/OpenAPI)**

**Para garantir a fácil usabilidade e integração, a API foi equipada com documentação interativa gerada automaticamente no padrão OpenAPI 3.0 através da biblioteca springdoc-openapi.**

* **Acesso à Documentação: A interface do Swagger UI está disponível para exploração e testes diretamente no navegador através da seguinte URL, enquanto a aplicação estiver rodando:**
  + **http://localhost:8080/swagger-ui.html**
* **Benefícios: Esta "documentação viva" serve como um contrato claro da API, permitindo que desenvolvedores entendam rapidamente todos os endpoints, parâmetros, modelos de dados e respostas esperadas.**

**6. Recursos e Tecnologias Utilizadas (Justificativas)**

* **Java 21 e Spring Boot 3: Escolhidos por seus recursos modernos, performance e pelo vasto ecossistema.**
* **Design Orientado a Interfaces ("Programado para a Interface"): A criação da interface AnaliseFraudeService com uma implementação simulada (MockAnaliseFraudeService) é um destaque de design. Ela desacopla a lógica de negócio da implementação específica da análise de fraude, permitindo que, no futuro, o mock seja substituído por uma implementação real que chame uma API de IA, sem a necessidade de alterar o SimulacaoService.**
* **Spring Data JPA com Transações Avançadas: Utilizado para simplificar o acesso a dados e gerenciar transações complexas em um fluxo assíncrono.**
* **Programação Assíncrona (via Azure SDK): Adoção do cliente assíncrono do Event Hub para melhorar a performance e a resiliência da API.**
* **springdoc-openapi: Biblioteca de alta produtividade para a geração automática de documentação OpenAPI 3.**
* **SQLite, Lombok, Actuator, Docker: Outras ferramentas essenciais que garantem simplicidade, limpeza do código, monitoramento e portabilidade da aplicação.**
* **JUnit 5 e Mockito: A suíte de ferramentas padrão utilizada para a implementação da nossa estratégia de testes unitários.**

**7. Conclusão**

**A API-Credito foi desenvolvida para ir além dos requisitos básicos, entregando uma solução alinhada com arquiteturas de microsserviços modernas. A implementação de uma camada de segurança para análise de fraudes, um fluxo assíncrono com padrão de auditoria, a otimização de consultas e a inclusão de documentação interativa com Swagger demonstram o foco em resiliência, performance, qualidade e na experiência do desenvolvedor. O resultado final é uma API robusta, eficiente, testável e pronta para evoluir.**