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

**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 e resiliente, capaz de oferecer simulações de empréstimo para qualquer pessoa ou sistema no Brasil.**

**A solução visa permitir que, através de uma simples requisição, um usuário possa descobrir as condições de negociação de crédito oferecidas, com base em um conjunto de produtos pré-definidos, fortalecendo a presença do serviço em canais digitais.**

**2. Arquitetura da Solução**

**Para garantir manutenibilidade, testabilidade e escalabilidade, a API foi estruturada seguindo os princípios da Arquitetura em Camadas (Layered Architecture). Esta abordagem promove uma clara separação de responsabilidades, facilitando a evolução e, crucialmente, o teste de cada componente de forma isolada.**

* **Controller Layer (Camada de Controle): 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. Adere às melhores práticas REST, retornando códigos de status apropriados, como 201 Created para criação de recursos.**
* **Service Layer (Camada de Serviço): O cérebro da aplicação, onde reside toda a lógica de negócio, como a filtragem de produtos, cálculos de amortização e a coordenação da persistência de dados.**
* **Repository Layer (Camada de Repositório): 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, delegando a lógica de filtragem diretamente para o banco de dados.**
* **Model/Entity e DTO Layers: As camadas de Model representam as entidades do domínio (Produto, Simulacao), enquanto os DTOs (Data Transfer Object) 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 menores componentes da aplicação de forma rápida e isolada.**

* **Estratégia Adotada: A abordagem principal foi testar cada classe de serviço e controller de forma independente. Para alcançar o isolamento, utilizamos o framework Mockito para criar "dublês" (mocks) das dependências. Por exemplo, ao testar o SimulacaoService, o ProdutoRepository era um mock, garantindo que o teste validasse a lógica do serviço, e não a funcionalidade do banco de dados.**
* **Cobertura dos Testes:**
  + **Camada de Serviço (SimulacaoService): Foram criados testes para validar os principais fluxos de negócio:**
    - **Cenário de Sucesso: Verificou-se que, para uma entrada válida, o serviço encontra o produto correto, chama os cálculos, salva a simulação e invoca o serviço de eventos.**
    - **Cenário de Exceção: Validou-se que o serviço lança corretamente a exceção ProdutoNaoEncontradoException quando os parâmetros de entrada não correspondem a nenhum produto.**
  + **Camada de Controller (SimulacaoController): Utilizando o MockMvc do Spring Test, foram simuladas requisições HTTP para validar:**
    - **Contrato da API: Verificou-se se o controller retorna os códigos de status HTTP corretos (201 Created para sucesso, 400 Bad Request para entradas inválidas).**
    - **Corpo da Resposta: Garantiu-se que o JSON retornado ao cliente possui a estrutura e os dados esperados.**

**Essa bateria de testes automatizados assegura que futuras alterações no código não quebrem funcionalidades existentes, sustentando um ciclo de desenvolvimento ágil e seguro (DevSecOps).**

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

**Para ilustrar o funcionamento da API na prática, a seguir é descrito o fluxo completo de uma chamada de simulação bem-sucedida:**

1. **Requisição (Cliente → Controller): Um cliente envia uma requisição POST para /api/simulacoes com um corpo JSON.**
2. **Controle e Validação (Controller): O SimulacaoController recebe a requisição e o @Valid dispara a validação dos dados.**
3. **Delegação (Controller → Service): O controller invoca o método criarSimulacao no SimulacaoService.**
4. **Lógica de Negócio (Service Layer): O serviço executa a query otimizada findProdutoElegivel, realiza os cálculos de amortização (SAC e Price) , cria e persiste a nova entidade**

**Simulacao no banco de dados.**

1. **Comunicação Externa (Service → Event Hub): O serviço invoca o EventHubService, que envia uma mensagem com o JSON da simulação para o Azure Event Hub.**
2. **Resposta (Controller → Cliente): O SimulacaoController recebe o DTO de resposta, constrói uma resposta HTTP com status 201 Created, cabeçalho Location e envia o corpo JSON de volta ao cliente.**

**5. Funcionalidades Implementadas e Atendimento ao Desafio**

**A API implementa todos os endpoints solicitados de forma coesa e unificada:**

* **POST /api/simulacoes: Realiza a simulação completa.**
* **GET /api/simulacoes: Lista todas as simulações realizadas.**
* **GET /api/simulacoes/volume-diario: Retorna o volume simulado por produto em uma data específica.**
* **GET /api/simulacoes/telemetria: Retorna os dados de telemetria da API.**

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

* **Java 21 e Spring Boot 3: Escolhidos por seus recursos modernos de performance, segurança e pelo vasto ecossistema que acelera o desenvolvimento.**
* **Spring Data JPA com @Query: O uso de uma query customizada otimiza a busca por produtos, delegando a lógica de filtragem para o banco de dados, o que é significativamente mais performático.**
* **SQLite: Utilizado conforme solicitado, por sua simplicidade e facilidade de configuração.**
* **Lombok: Adotado para reduzir o código "boilerplate", tornando as classes mais limpas.**
* **Spring Boot Actuator: Utilizado para a telemetria, oferecendo uma solução robusta e padrão de mercado para monitoramento.**
* **JUnit 5 e Mockito: A suíte de ferramentas padrão utilizada para a implementação da nossa estratégia de testes unitários, detalhada na seção "Qualidade e Estratégia de Testes".**
* **Docker: Um Dockerfile está incluído para garantir que a aplicação possa ser empacotada e executada de forma consistente.**

**7. Conclusão**

**A API-Credito foi desenvolvida com sucesso, atendendo a todos os requisitos funcionais e não-funcionais do desafio. A arquitetura em camadas, combinada com decisões de otimização, uma estratégia de testes robusta e a adesão às melhores práticas REST, resultou em uma solução final que é não apenas funcional, mas também eficiente, resiliente, segura e de fácil manutenção.**