**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 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. Aqui reside toda a lógica de negócio, incluindo a filtragem de produtos, cálculos de amortização e, crucialmente, a gestão de estado de operações assíncronas, como a comunicação com o Event Hub, garantindo a auditoria das transações.**
* **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 (ex: repositórios, outros serviços).**
* **Cobertura dos Testes: Foram criados testes para os cenários de sucesso ("caminho feliz") e cenários de exceção ("caminho triste") nas camadas de Serviço e Controller, validando desde a lógica de negócio até o contrato da API (status HTTP e formato do JSON).**
* **Ferramentas: A suíte de testes foi construída com JUnit 5 e Mockito, ferramentas padrão e robustas do ecossistema Spring.**

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

**Para ilustrar a robustez da API, o fluxo completo de uma chamada de simulação bem-sucedida é descrito abaixo:**

1. **Requisição e Validação: Um cliente envia uma requisição POST para /api/simulacoes. O SimulacaoController recebe e valida os dados de entrada.**
2. **Lógica de Negócio e Persistência Inicial: O SimulacaoService é chamado. Ele executa a query otimizada para encontrar o produto, realiza os cálculos de amortização e cria uma nova entidade Simulacao, salvando-a no banco de dados com um status inicial: AGUARDANDO\_ENVIO.**
3. **Comunicação Externa (Assíncrona): O SimulacaoService invoca o EventHubService para enviar o evento. A chamada é não-bloqueante (assíncrona), ou seja, a API não espera pela resposta do Azure e já prossegue para o próximo passo. O serviço fornece "callbacks" (funções de retorno) para serem executados em caso de sucesso ou falha no envio.**
4. **Resposta Imediata ao Cliente: A API retorna imediatamente a resposta ao cliente com o status 201 Created e o corpo da simulação, garantindo baixa latência e uma ótima experiência para o usuário.**
5. **Atualização de Status (em Background): Em um momento posterior (geralmente milissegundos depois), o Event Hub confirma o recebimento do evento. O callback de sucesso fornecido é acionado. Este callback executa o método atualizarStatusEnvio em uma nova transação, atualizando o status da simulação no banco de dados para ENVIADO. Em caso de falha, o status seria atualizado para FALHA\_NO\_ENVIO, criando assim uma trilha de auditoria completa e confiável.**

**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 relatório de volume simulado por produto.**
* **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, performance e pelo vasto ecossistema que acelera o desenvolvimento.**
* **Spring Data JPA com Transações Avançadas: Além de simplificar o acesso a dados, utilizamos recursos avançados de transação (@Transactional(propagation = Propagation.REQUIRES\_NEW)) para garantir a consistência dos dados ao atualizar o status da simulação de forma assíncrona.**
* **Programação Assíncrona (via Azure SDK): A adoção do cliente assíncrono do Event Hub é uma decisão de arquitetura chave para melhorar a performance e a resiliência da API, evitando que a thread principal fique bloqueada por operações de rede.**
* **SQLite, Lombok, Actuator, Docker: Outras ferramentas essenciais que garantem simplicidade de desenvolvimento, 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, detalhada na seção "Qualidade e Estratégia de Testes".**

**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 um fluxo assíncrono com um padrão de auditoria para a comunicação externa demonstra o foco em resiliência, performance e confiabilidade. O resultado final é uma API robusta, eficiente, testável e pronta para evoluir.**