**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)**, uma abordagem consolidada no desenvolvimento de software. A escolha dessa arquitetura promove uma clara separação de responsabilidades, facilitando a evolução e o teste de cada componente de forma isolada.

A estrutura do projeto foi dividida da seguinte forma:

* **Controller Layer (Camada de Controle):** A porta de entrada da API, unificada no SimulacaoController. É responsável por expor os endpoints REST, receber as requisições HTTP, validar os dados de entrada (@Valid) e orquestrar as chamadas para os serviços. Esta camada 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. Módulos como SimulacaoService, RelatorioService e CalculoAmortizacaoService trabalham em conjunto para executar as regras de negócio de forma coesa.
* **Repository Layer (Camada de Repositório):** A camada de acesso a dados. Utilizando o Spring Data JPA, ela abstrai a comunicação com o banco de dados. Um destaque nesta camada é o uso de queries customizadas com @Query no ProdutoRepository, o que otimiza a busca por produtos elegíveis, delegando a lógica de filtragem diretamente para o banco de dados e melhorando a performance.
* **Model/Entity e DTO Layers:** As camadas de Model representam as entidades do domínio (Produto, Simulacao), enquanto os DTOs (Data Transfer Objects) definem os contratos seguros para a comunicação externa, prevenindo o vazamento de detalhes da implementação interna.

**3. 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 o endpoint /api/simulacoes, contendo um corpo JSON com valorDesejado e prazo.
2. **Controle e Validação (Controller):** O SimulacaoController recebe a requisição. A anotação @Valid no DTO de entrada dispara o mecanismo de validação do Spring, garantindo que os dados sejam válidos (ex: não nulos) antes de prosseguir.
3. **Delegação (Controller → Service):** O controller invoca o método criarSimulacao no SimulacaoService, passando os dados da requisição.
4. **Lógica de Negócio (Service Layer):**
   * O SimulacaoService chama o método produtoRepository.findProdutoElegivel(), executando a query otimizada diretamente no banco de dados para encontrar um produto compatível.
   * Caso nenhum produto seja encontrado, uma exceção customizada (ProdutoNaoEncontradoException) é lançada, resultando em uma resposta de erro clara para o cliente.
   * Com um produto válido em mãos, o serviço invoca o CalculoAmortizacaoService para calcular as parcelas dos sistemas SAC e PRICE.
   * Uma nova entidade Simulacao é criada com todos os dados calculados e o JSON completo do resultado. Ela é então persistida no banco de dados em uma única operação de save().
5. **Comunicação Externa (Service → Event Hub):** Após a persistência, o SimulacaoService chama o EventHubService, que envia uma mensagem contendo o JSON da simulação para o Azure Event Hub.
6. **Resposta (Controller → Cliente):** O SimulacaoService retorna o DTO de resposta completo para o SimulacaoController. O controller, por sua vez, constrói uma resposta HTTP com o status **201 Created**, inclui o Location da nova simulação no cabeçalho e envia o corpo JSON da resposta de volta ao cliente, finalizando o ciclo.

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

A API implementa todos os endpoints solicitados de forma coesa e unificada sob o caminho /api/simulacoes:

* **POST /api/simulacoes**: Realiza a simulação completa, persistindo os dados e enviando o evento.
* **GET /api/simulacoes**: Lista todas as simulações realizadas, com suporte a paginação.
* **GET /api/simulacoes/volume-diario**: Retorna o relatório de volume simulado por produto em uma data específica.
* **GET /api/simulacoes/telemetria**: Retorna os dados de telemetria da API, como métricas de performance dos endpoints.
* Um endpoint **GET /api/telemetria** também foi mantido para acesso direto aos dados de telemetria do dia corrente.

**5. 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 e Hibernate com @Query**: O uso de uma query customizada no ProdutoRepository foi uma decisão de otimização chave. Em vez de trazer todos os produtos para a memória da aplicação (findAll), a lógica de filtragem é executada diretamente no banco de dados, o que é significativamente mais performático, especialmente com um grande número de produtos.
* **SQLite**: Utilizado conforme solicitado, por sua simplicidade e facilidade de configuração em um ambiente de desenvolvimento.
* **Lombok**: Adotado para reduzir o código "boilerplate", tornando as classes mais limpas e legíveis.
* **Spring Boot Actuator**: Utilizado para a telemetria, oferecendo uma solução robusta e padrão de mercado para monitoramento com mínimo esforço de implementação.
* **JUnit 5 e Mockito**: Ferramentas padrão para a criação de testes unitários, essenciais para validar a lógica de cada componente de forma isolada e garantir a qualidade do software.
* **Docker**: Um Dockerfile está incluído para garantir que a aplicação possa ser empacotada e executada de forma consistente em qualquer ambiente.

**6. Conclusão**

A **API-Credito** foi desenvolvida com sucesso, atendendo a todos os requisitos do desafio. A arquitetura em camadas, combinada com decisões de otimização (como queries customizadas) 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, demonstrando uma abordagem profissional e completa para a resolução do problema proposto.