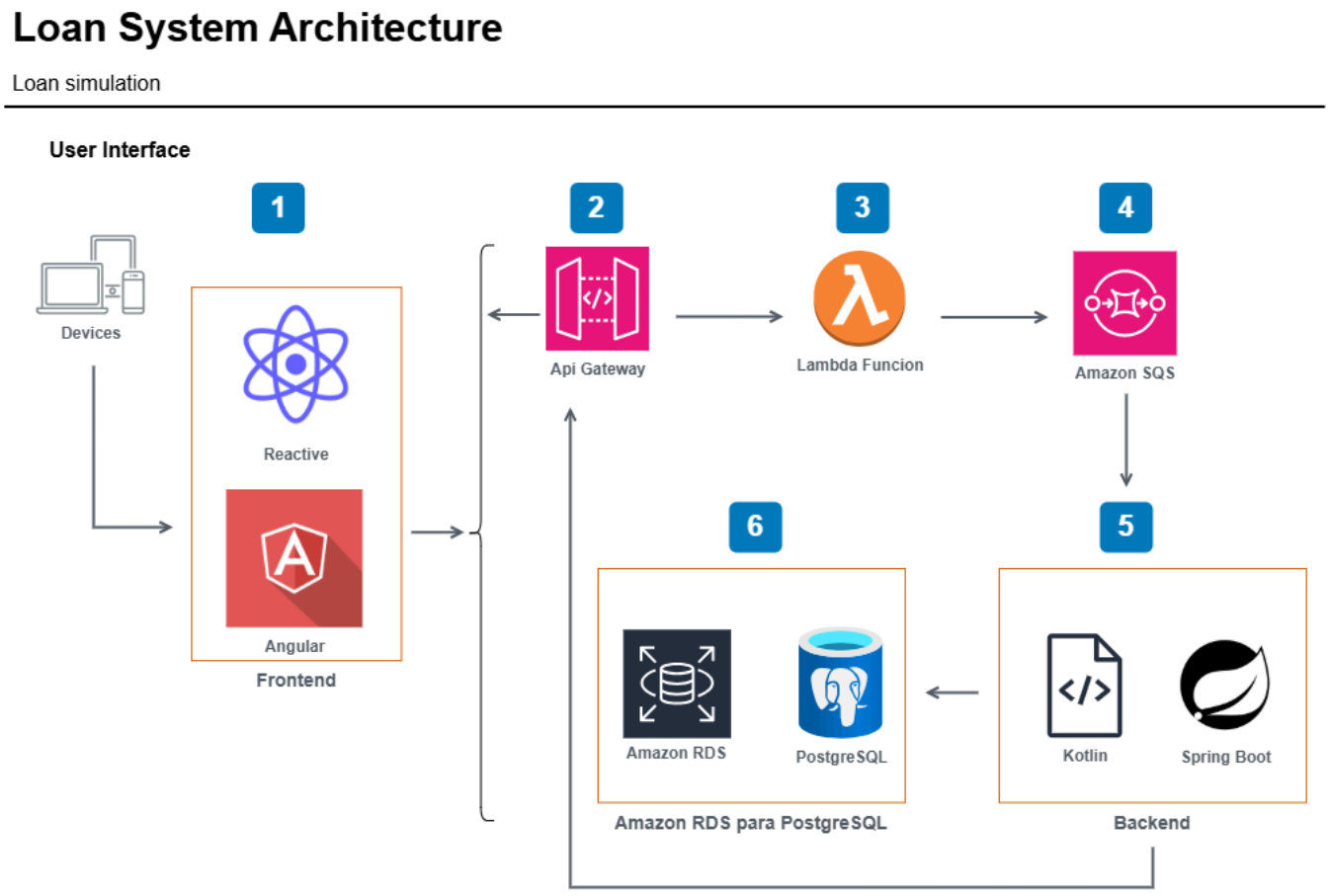
SIMULADOR DE EMPRÉSTIMOS

Proposta de arquitetura



Funcionalidade básica

- Permitir que os usuários insiram dados financeiros.
- Realizar simulações de empréstimos com base nos dados fornecidos, calculando diferentes cenários de taxa de juros e valor das parcelas.
- Gerar propostas personalizadas de empréstimos.

Desenho da Arquitetura

Frontend: Envia a requisição para o endpoint exposto pelo API Gateway. Para o frontend duas linguagens são sugeridas:

- **React tem as seguintes vantagens:** Facilidade de aprendizado, componentes reutilizáveis, melhor desempenho, compatibilidade com SEO (Search Engine Optmization), flexibilidade, curva de aprendizado suave.
- **Angular tem as seguintes vantagens:** Arquitetura organizada, suporte do google, comunidade ativa, combinação de dados bidirecional, diretivas, injeção de dependências, recarregamento rápido, biblioteca reativa RxJS.

API Gateway AWS: Recebe a requisição do frontend. Pode fazer algumas validações iniciais. Encaminha a requisição para uma função Lambda que coloca a mensagem na fila Amazon SQS. Os benefícios de seu uso são:

- **Abstração do Backend**: O API Gateway atua como uma camada intermediária que abstrai a lógica do backend, fornecendo uma interface RESTful para o frontend interagir.
- **Escalabilidade**: O API Gateway é totalmente gerenciado e pode escalar automaticamente para lidar com um grande número de requisições, aliviando a pressão sobre os servidores backend.
- **Throttle e Limites de Taxa**: Ele permite definir limites de taxa e throttling para proteger seu backend contra ataques DDoS e picos de carga inesperados.
- **Autenticação e Autorização**: Integração com AWS Cognito ou outros provedores de identidade para gerenciar autenticação e autorização sem necessidade de implementar isso diretamente no backend.
- **Transformação de Requisições**: O API Gateway pode transformar requisições e respostas, permitindo flexibilidade na comunicação entre o frontend e o backend.

AWS Lambda Function: Faz algumas validações dos dados recebidos, transforma para o formato adequado e envia a mensagem para a fila Amazon SQS. Razões para seu uso:

- **Custo-eficiência**: Paga-se apenas pelo tempo de execução da função Lambda e pela quantidade de recursos consumidos, o que é ideal para workloads que não são constantes. Se o sistema recebe requisições esporádicas ou com picos sazonais, Lambda pode ser mais econômico do que manter servidores ativos o tempo todo.
- **Escalabilidade Automática:** Escala automaticamente com o número de requisições, sem necessidade de configuração manual de escalonamento. Em cenários de alto volume de tráfego, a Lambda pode criar múltiplas instâncias automaticamente para atender a todas as requisições simultaneamente.
- **Menor Gerenciamento:** Com Lambda, você não precisa gerenciar servidores, VMs, ou contêineres. Toda a infraestrutura é gerenciada pela AWS, permitindo que você se concentre apenas na lógica de negócios.
- **Fácil Integração com Amazon SQS:** Oferece integração nativa para os serviços AWS, transformando a comunicação entre os componentes da arquitetura mais fluida e menos propensa a erros.
- **Recuperação de Falhas:** Lambda pode ser configurado para re-tentar automaticamente em caso de falhas. Se algo der errado ao tentar enviar uma mensagem para SQS, Lambda pode reprocessar a função até que a mensagem seja enviada com sucesso.

Amazon SQS: Armazena a mensagem até que o backend a processe. Razões para a escolha do SQS:

- **Confiabilidade:** O SQS garante a entrega de mensagens sem perdas, mesmo em grandes volumes.
- **Segurança:** O SQS permite enviar dados confidenciais de forma segura.
- **Escalabilidade:** O SQS permite escalar de forma econômica, sem a necessidade de planejar e provisionar capacidade antecipadamente.
- **Processamento assíncrono:** O SQS permite que os serviços processem mensagens de forma independente.
- **Retenção de mensagens:** O SQS permite armazenar mensagens por até 14 dias.
- **Personalização:** O SQS permite definir atraso padrão em uma fila.

- **Agrupamento:** O SQS permite que os serviços processem mensagens em grupos.
- **Desacoplamento de sistemas:** O SQS permite separar os aplicativos em componentes independentes, o que facilita o desenvolvimento, a implantação e a manutenção

Backend Kotlin e Spring Boot: Um serviço em Kotlin lê as mensagens da fila SQS. Processa cada mensagem (simulação de empréstimo) e armazena o resultado no banco de dados. Notifica o frontend sobre o processamento concluído. Motivos para sua escolha:

- **Kotlin:** Sintaxe clara e moderna, aumentando a produtividade. Compatível com bibliotecas Java, permitindo o uso de tecnologias maduras. Suporte nativo a corrotinas para operações assíncronas eficientes.
- **Spring Boot:** Ecossistema rico para desenvolvimento rápido e seguro. Integração com padrões de segurança, como OAuth 2.0. Escalável com suporte a arquiteturas baseadas em microsserviços.

Amazon RDS e PostgreSQL: Os dados são armazenados no banco de dados e disponibilizados para visualização ou ajustes posteriores. Amazon RDS como serviço de banco de dados relacional gerenciando o banco de dados PostgreSQL para manter a consistência e o suporte a consultas complexas. Razões para estas escolhas:

- **PostgreSQL:** Escalabilidade, segurança, flexibilidade, personalização, baixo custo, suporte a transações.
- **Amazon RDS:** Facilidade de gerenciamento, alta disponibilidade, escalabilidade, segurança, monitoramento e manutenção automática.

Padrões de Projeto e Boas Práticas

Para este cenário de simulação e proposta de empréstimos, com componentes como **API Gateway**, **AWS Lambda**, **Amazon SQS**, e um backend em **Kotlin**, a Arquitetura Orientada a Eventos (Event-Driven Architecture) se destaca por suas vantagens em termos de escalabilidade, resiliência e manutenibilidade.

Descrição: As mensagens enviadas para a fila SQS representam eventos que disparam o processamento no backend.

Vantagens:

- **Escalabilidade**: O sistema é naturalmente escalável, pois novas mensagens na fila SQS podem ser processadas por instâncias adicionais de workers conforme necessário.
- **Resiliência**: O sistema é resiliente a falhas, pois SQS atua como um buffer que armazena mensagens até que elas possam ser processadas.
- **Desacoplamento**: Os componentes do sistema são fortemente desacoplados, permitindo que o frontend, backend e serviços de processamento evoluam independentemente.

Autenticação: O uso do **OAuth 2.0** nessa arquitetura proporciona uma maneira segura e eficiente de gerenciar **autenticação e autorização** dos usuários que interagem com o sistema, especialmente considerando que o sistema envolve múltiplos componentes como frontend, API Gateway, Lambda functions, e microserviços backend.

Componentes Principais do OAuth 2.0 na Arquitetura

- **Authorization Server**: Um servidor dedicado (ou serviço como AWS Cognito ou Auth0) que gerencia a autenticação dos usuários e emite **tokens de acesso** e **tokens de refresh**.
- **Resource Owner (Usuário)**: O usuário que deseja acessar os recursos protegidos (como simulações de empréstimos).
- **Client (Frontend)**: A aplicação frontend que inicia a solicitação de autenticação para o usuário.
- **Resource Server**: O backend (incluindo API Gateway, Lambda, e serviços backend) que expõe APIs protegidas que requerem tokens de acesso válidos para serem acessadas.

Fluxo de Autenticação usando OAuth 2.0

Passo 1: Login e Obtenção do Token de Acesso

1. O usuário acessa o **frontend** (Client) e tenta realizar uma ação que requer autenticação, como uma simulação de empréstimo.
2. O frontend solicita a autenticação do usuário para o **Authorization Server**.
3. Após a autenticação bem-sucedida, o Authorization Server fornece um **token de acesso** (e possivelmente um **token de refresh**) que o frontend irá armazenar.

Passo 2: Uso do Token para Acessar APIs

4. O frontend agora inclui o **token de acesso** no cabeçalho de autorização das requisições para a **API Gateway**. Exemplo de cabeçalho HTTP:
   `Authorization: Bearer <access_token>`
5. O **API Gateway** valida o token de acesso usando o **Authorization Server** ou as chaves públicas do servidor para verificar a assinatura do token.

Passo 3: Acesso ao Backend

6. Se o token for válido, o API Gateway encaminha a requisição para o **Lambda**.
7. O Lambda inclui este token na mensagem enviada ao SQS.
8. O backend processa a mensagem e retorna o resultado ao frontend via API Gateway.

Escalabilidade e Resiliência

A arquitetura em microsserviços e o uso de API Gateway e filas Amazon SQS permitem o processamento assíncrono sem sobrecarregar o fluxo como um todo.

A resiliência pode ser observada com monitoramento via Prometheus e Grafana. Com paineis claros e objetivos, é possível verificar o status da aplicação em tempo real, bem como ter alertas em caso de eventuais falhas.

API Design

Principais Endpoints:

- *POST /api/v1/loan/simulate*: Recebe os dados do cliente e retorna os cenários.

  Exemplos de Request:

```
{
    "loanAmount": 10000.00,
    "birthDate": "1990-01-01",
    "term": 12
}
```

  Exemplos de Response:

```
{
    "proposalId": "47dfe707-b055-4406-8e5f-0bac05066204",
    "interestRate": 0.03,
    "monthlyPayment": 846.94,
    "totalAmount": 10163.24
}
```

- GET */api/v1/loan/47dfe707-b055-4406-8e5f-0bac05066204*: Retorna uma proposta salva anteriormente.

  Exemplos de Response:

```
{
    "proposalId": "47dfe707-b055-4406-8e5f-0bac05066204",
    "interestRate": 0.03,
    "monthlyPayment": 846.94,
    "totalAmount": 10163.24
}
```

- *GET / interest-rates:* Obtém taxas atualizadas de serviços externos.

Motor de Simulação

O motor de simulação deve usar bibliotecas de cálculos que apoiam o desenvolvimento de forma segura. São elas:

Apache Commons Math: Uma biblioteca matemática robusta que oferece suporte para várias operações matemáticas e estatísticas, incluindo cálculos financeiros. Pode ser usada para cálculos de juros compostos, taxas de retorno e outras funções financeiras básicas.

Java Money: A evolução da aplicação pode requerer conversão monetária, a indicação é do uso da biblioteca Java Money: uma API padrão para operações monetárias em Java, que também pode ser usada em Kotlin. Oferece suporte para cálculos monetários precisos e manipulação de moedas.

BigDecimal: Embora não seja uma biblioteca específica, o uso de BigDecimal é essencial para cálculos financeiros precisos em Kotlin/Java, especialmente para evitar erros de arredondamento em operações monetárias.

Considerações finais.

Esta proposta tem sugestões cabíveis de mudanças, melhorias e futuras adaptações, contudo, é válido lembrar que uma boa arquitetura evita retrabalhos futuros. Um código pode ser melhorado sem grandes esforços, mas uma arquitetura mal desenhada pode acarretar perdas financeiras de grandes proporções.

Espero que esta proposta esteja adequada aos padrões da empresa.

Por Regiane Martins de Brito

Janeiro 2025