Emerson V. Rafael - COORD ENGENHARIA TI - EXPERIÊNCIA E DESIGN - 792388

Atores envolvidos no processo:

- **Plataforma de Testes A/B**

  Responsável por orquestrar, randomizar, registrar, monitorar e analisar os experimentos.

- **Produto FrontEnd**

  Squad responsável por aplicar a lógica de teste em seu app (web/mobile) e enviar eventos de uso.

Objetivo do time de produto frontend

Esse time quer **testar uma nova experiência com clientes reais** — por exemplo:

- Um novo botão no app.
- Um texto diferente na home.
- Uma tela nova de fluxo de cadastro.

Para isso, ele precisa expor **duas ou mais variantes (ex: A e B)** para usuários reais **de forma controlada e rastreável**, sem fazer deploy separado para cada uma.

O que o time frontend precisa fazer (passo a passo)

1. Registrar o experimento (ou usar um já criado)

Pode ser feito via UI da plataforma de testes A/B (ou AP). O time fornece:

- Nome do experimento
- Qual feature será testada

- Grupos (ex: A e B)
- Porcentagem por grupo (ex: 50/50)
- Métrica principal (ex: clique no botão)

> Isso gera um ID único do experimento e parâmetros de exposição.

2. Integrar o SDK ou fazer chamada à API da plataforma

No **código do frontend**, antes de decidir qual variante exibir, o time:

- **Chama o SDK** (ou API da plataforma), passando:
  - `user_id` (ou outro ID persistente)
  - `experiment_id`

*No apêndice, coloquei uma estratégia possível para evitar acesso indevido à experimentos de outro time.

O SDK ou API vai retornar algo como:

```
{
  "variant": "B",
  "feature_flag": "new_button",
  "enabled": true}
```

3. Renderizar a experiência com base na resposta

Com base no retorno da plataforma, o time decide qual interface mostrar:

```
if (variant === "A") {
  renderBotaoClássico();
} else {
  renderBotaoComTooltip();
```

4. Emiti eventos de uso (tracking)

Sempre que o usuário interagir com a feature (ex: clicar no botão), o front **dispara um evento** com os seguintes dados mínimos:

```
{
  "user_id": "abc123",
  "experiment_id": "exp-001",
  "variant": "B",
  "event": "click_botao",
  "timestamp": "2025-06-26T09:00:00Z"
}
```

Esse evento vai para a pipeline da nossa plataforma de testes A/B (via Kinesis/EventBridge), alimentando as análises de dados (que podem ser consultadas via dashboard no frontend da plataforma de testes a/b.

5. Consultar resultado (opcional)

Se quiser, o time pode:

- Acompanhar o andamento no **dashboard da plataforma de testes A/B**.
- Ver métricas em tempo real (ex: conversão por grupo).

Como funciona o Backend da Plataforma de Testes A/B

Vamos quebrar em 4 partes:

1. Registro de experimento
2. Consulta da variante (get_variant)
3. Registro de evento (tracking)
4. Estrutura de dados e segurança

1. Registro de Experimento

O time de produto envia uma requisição para criar um experimento.

Exemplo de requisição (API ou UI):

```
POST /experiments

Headers:
```

```
  Authorization: Bearer <JWT>

Body:
{
  "name": "Botão com tooltip",
  "description": "Teste do novo botão com texto explicativo",
  "app_id": "app_home_v1",
  "squad_id": "squad_experiencia",
  "metric": "click_botao",
  "groups": {
    "A": 50,
    "B": 50
  },
  "start_date": "2025-06-27",
  "end_date": "2025-07-07"
}
```

Backend:

- Valida autenticidade e escopo do token (`app_id`, `squad_id`).
- Salva no banco com `experiment_id` gerado (`exp-001`).
- Marca status como `"active"` e registra logs de auditoria.

2. Consulta da Variante (`get_variant`)

Quando o frontend quer saber **qual experiência mostrar**, ele chama:

```
POST /get-variant

Headers:
  Authorization: Bearer <JWT>
  X-App-Id: app_home_v1

Body:
{
  "experiment_id": "exp-001",
  "user_id": "abc123"
}
```

Backend executa:

1. Validação do token e do `app_id` contra o experimento.
2. Geração determinística da variante com base no `user_id`:

```
hash_value = hash(str(user_id) + str(experiment_id)) % 100
if hash_value < 50:
    variant = "A"
else:
    variant = "B"
```

3. Retorna a resposta ao frontend:

```
{
  "variant": "A",
  "enabled": true,
  "experiment_id": "exp-001"
}
```

Necessidades

- Receber dos times as decisões de negócio realizadas com os testes e os resultados obtidos com essas decisões.
    - Objetivo: Identificar o impacto no negócio da plataforma de teste.
    - Objetivo: Analisar a nossa precisão em casos em que a hipótese alternativa é confirmada.

Tecnologias da Plataforma de Testes A/B (Atualizado)

Frontend: Angular + Tailwind CSS

- **Motivo da escolha:**
    - Angular oferece estrutura robusta e opinativa, ideal para portais internos com múltiplas telas e formulários complexos.

    - Excelente suporte para controle de estado, rotas, validações e serviços reutilizáveis (ex: serviço de autenticação via JWT).
    - Tailwind CSS complementa com personalização visual ágil e sem acoplamento a bibliotecas externas de UI.
- **Diferenciais:**
    - Recomendado no Itaú Design System (IDS)
    - Possui componentes para aceleração no desenvolvimento
        - Componente: Design2Code (converte Figma para Angular, utilizando um agente do Stackspot)

Backend: Python com FastAPI

- **Motivo da escolha:**
    - FastAPI combina performance com código limpo — ideal para criar APIs REST com validação automática de dados via Pydantic.
    - Integração fluida com bancos de dados (RDS Aurora ou DynamoDB) e autenticação via JWT.
    - Excelente para criar endpoints como `/experiments`, `/get-variant`, `/track-event` com documentação automática (Swagger UI).
- **Diferenciais:**
    - Tipagem forte (combinado com pydantic) e segurança em tempo de desenvolvimento.

Banco de Dados: RDS Aurora ou DynamoDB

- **Para metadados de experimentos:**
    - `experiment_id`, `squad_id`, `app_id`, datas, métricas, status, etc.
- **Justificativa:**
    - RDS Aurora permite queries SQL complexas e relatórios internos.
    - DynamoDB garante performance em escala com acesso direto por chave.
    - A escolha depende da criticidade analítica x escalabilidade de leitura.

Distribuição de Variante (lógica A/B)

- **Tecnologia:** Algoritmo de hash determinístico
- **Motivo:**
  - Evita viés, garante reprodutibilidade e reduz dependência de banco.

Registro de Eventos

- **Tecnologia:** AWS Kinesis
- **Motivo:**
  - Suporte nativo a eventos em tempo real (ex: clique em botão, envio de formulário).
  - Fácil integração com API Gateway e Lambda.
  - Integração com pipelines de ingestão e dashboards analíticos.
  - Escalável com baixo overhead de manutenção.

Dashboard Analítico

- **Tecnologia:** Módulo em Angular
- **Motivo:**
  - Ferramentas open source com filtros por experimento, variante, métrica, tempo, etc.
  - Integração direta com banco de dados e pipelines.
  - Com o módulo próprio em Angular, garantimos experiência customizada na mesma plataforma de testes a/b.

Autenticação

- **Tecnologia:** JWT com escopo por `app_id` e `squad_id`
- **Motivo:**
  - Controle de acesso granular.
  - Cada squad só acessa seus próprios experimentos.
  - Fácil integração com FastAPI (middleware) e Angular (interceptor).

Infraestrutura

- **Tecnologia:** Microfrontends - Microserviços - Stackspot EDP

- **API Gateway**: Exposição de endpoints REST.
- **Lambda**: Execução de lógica de negócio (opcional, se desacoplar partes do backend).
- **S3**: Armazenamento de assets e logs.
- **CloudWatch**: Observabilidade.
- **IAM**: Controle de acesso.

Segurança e Governança

- Logs de auditoria para qualquer ação no experimento.
- Validação de escopo em cada chamada (`app_id` obrigatório).
- Possibilidade de adicionar MFA no acesso à UI administrativa.

Apêndices

Soluções para evitar acesso indevido a experiment_id de outro time

1. Vincular o experiment_id a um "owner" (squad ou app)

Ao criar o experimento na plataforma de testes A/B, ele **deve ser associado a um identificador de time/aplicação**, como:

- `app_id = "app_carteira_digital"`
- `squad_id = "squad_experiencias_home"`

Esse campo vira parte do metadata do experimento.

2. Validar o app_id no momento da requisição

No momento em que o frontend chama a API/SDK para buscar a variante, ele **deve se identificar**, e a plataforma só responde se o experimento pertencer ao app ou squad correto.

Exemplo de requisição:

```
POST /get-variant
Headers:
  Authorization: Bearer <JWT>
  X-App-Id: app_carteira_digital

Body:
{
  "experiment_id": "exp-001",
  "user_id": "abc123"
}
```

Resposta (caso autorizado):

```
{
  "variant": "A",
  "enabled": true}
```

Resposta (caso não autorizado):

```
{
  "error": "unauthorized_experiment_access"
}
```

Por que usar user_id e experiment_id combinados na definição do grupo de teste?

1. Evitar colisões entre experimentos diferentes

Se usássemos **só** `user_id`, o usuário **cairia sempre na mesma variante em todos os testes**, o que:

- Quebra a aleatoriedade.
- Cria viés (ex: mesmo usuário sempre em "A" em todos os experimentos).

- Pode distorcer resultados em experimentos simultâneos.

Com `user_id + experiment_id`, a distribuição é única para cada experimento.

2. Garante consistência por experimento

> A cada execução da função, o usuário abc123 pode cair na variante A no exp-001, mas na variante B no exp-002.

Assim conseguimos:

Experimento	user_id = abc123	Variante
exp-001	A	A
exp-002	B	B
exp-003	A	A

Ou seja, **o mesmo usuário pode participar de múltiplos testes, com distribuições distintas e independentes**.

3. Evita dependência de banco

Ao gerar a randomização com essa lógica, não precisamos **persistir** em banco qual grupo o usuário recebeu — basta calcular de novo sempre que precisar.

4. Reprodutibilidade

Se alguém quiser auditar o resultado do experimento ou reprocessar os dados depois, pode **recalcular exatamente qual variante o usuário recebeu**, com base em `user_id` e `experiment_id`.

Exemplo Prático

```
hash("abc123exp-001") % 100  ➜ 43 → Variante A
hash("abc123exp-002") % 100  ➜ 72 → Variante B
```

Mesmo usuário (`abc123`), mas com distribuições diferentes conforme o experimento.