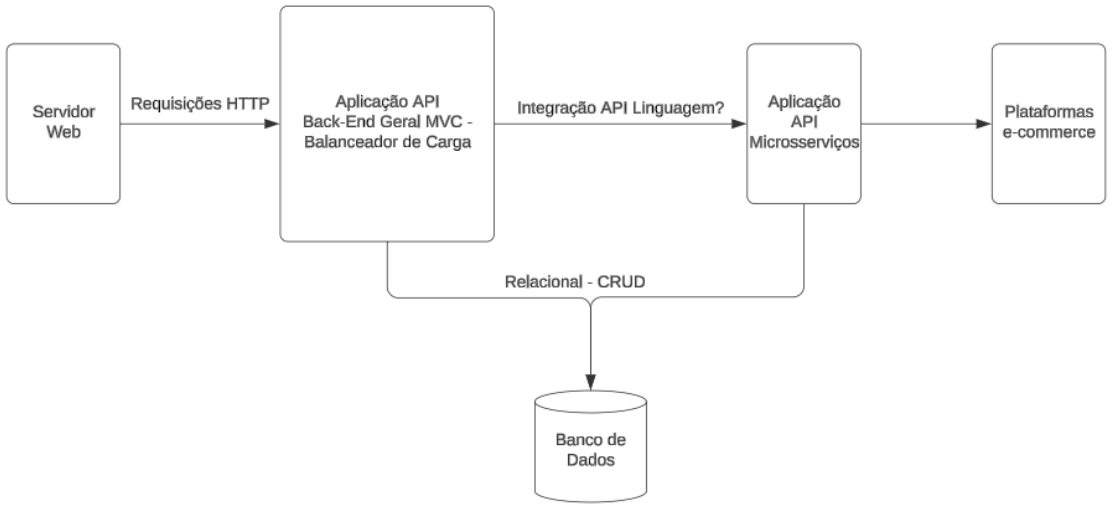
API INTEGRATION



Servidor Web:

UI padrão conectada a API MVC, HTML, CSS e JavaScript. Páginas responsivas e com bom visual, nada muito complicado o mais minimalista e intuitivo possível.

Aplicação API Back-End Geral MVC - Balanceador de Carga:

API que controla o que será mostrado ao usuário e recebe os dados da UI, junto da regra de negócios da empresa. Conecta ao Banco de Dados para armazenamento de login e produtos. Em um futuro também pode atuar como um Balanceador de Carga, porém a estrutura arquitetural pode sofrer mudanças.

Possíveis linguagem: Python, Java(mais familiarização), C#(mais recomendada na minha opinião), sugestões?

Sugestões de Linguagem

Aqui está uma análise das linguagens sugeridas e algumas opções adicionais:

C# (mais recomendado):

Prós:
Excelente integração com o Microsoft SQL Server.
Ferramentas robustas, como o framework
ASP.NET Core, que é moderno, eficiente e facilita a implementação do padrão MVC.
Ótima documentação e suporte.
Contras: Pode ter uma curva de aprendizado maior se você não estiver muito habituado.

Java (familiar):

Prós:
Frameworks como Spring Boot oferecem suporte completo para APIs e MVC.
Estável, robusto e com suporte a balanceamento de carga integrado.
Contras: Mais verboso e pode demandar configurações iniciais mais complexas.

Python:

Prós:
Frameworks como Django e Flask são simples e eficientes para APIs e MVC.
Ideal para prototipagem e desenvolvimento rápido.
Contras: Pode não ser a melhor escolha para projetos de longo prazo ou que exijam desempenho extremo, comparado a C# ou Java.

Node.js:

Prós:
Muito usado para APIs devido à velocidade e escalabilidade.
Suporte excelente para JSON e APIs RESTful.
Contras: O gerenciamento de threads pode ser mais desafiador para sistemas

altamente complexos.

Go (Golang):

Prós:
Simplicidade com alto desempenho e suporte nativo a aplicações concorrentes.
Ótimo para sistemas com balanceamento de carga e alta escalabilidade.
Contras: Menos familiaridade e menor número de bibliotecas pré-prontas.

Estando mais confortável com Java, ele é uma boa escolha para começar devido ao suporte a frameworks robustos. No entanto, se C# é sua recomendação preferida, o ASP.NET Core é um ótimo framework para criar essa API com padrão MVC.

Banco de Dados:

Armazena dados de clientes assim como seus produtos, também ficaria responsável por armazenar informações da conexão as Plataformas E-commerce como as chaves principais. As informações dos clientes e das plataformas seriam em DataBases diferentes.

Sugestões de Banco de Dados:

MySQL ou PostgreSQL (Relacional):

Prós:
Altamente confiável e com suporte para transações complexas.
Fácil de integrar com as linguagens sugeridas (Java, C#, Python).
Uso: Perfeito para manter integridade e relacionamentos claros entre clientes, produtos e plataformas.

MongoDB (NoSQL):

Prós:
Mais flexível para armazenar dados não estruturados, como logs ou integrações variadas de plataformas.
Bom para sistemas que precisam escalar horizontalmente.
Contras:
Pode exigir mais trabalho para manter consistência de dados em cenários complexos.

Azure SQL ou Amazon RDS (Bancos gerenciados na nuvem):

Prós:
Facilidade de manutenção e backups automáticos.
Boa escolha para escalabilidade e alta disponibilidade.
Contras:
Dependência de custos e serviços de nuvem.

Sugestão de Estrutura de Tabelas

Database 1: Clientes e Produtos

Tabela clientes:

id_cliente (chave primária)
nome
email
senha (criptografada)
criado_em (timestamp)

Tabela produtos:

id_produto (chave primária)
id_cliente (chave estrangeira)
nome_produto
descricao
preco
status_sincronizacao (ex.: "sincronizado", "pendente")
criado_em (timestamp)

Database 2: Configurações de E-commerce

Tabela plataformas:

id_plataforma (chave primária)
nome_plataforma (ex.: Mercado Livre, Amazon)
url_api
documentacao

Tabela credenciais:

id_credencial (chave primária)
id_cliente (chave estrangeira para clientes)
id_plataforma (chave estrangeira para plataformas)
api_key (criptografada)
api_secret (criptografada)
criado_em (timestamp)

Sugestão de Implementação

Use ORMs (Object-Relational Mappers):

Para C#: Entity Framework Core.
Para Java: Hibernate.
Para Python: SQLAlchemy ou Django ORM.

Segurança:

Criptografar senhas e chaves API usando AES ou bcrypt.
Implementar roles e permissões para evitar acesso indevido.

Usando Bancos de Dados Diferentes

1. Database 1 (MySQL): Clientes e Produtos

- MySQL seria ótimo para armazenar informações de clientes e produtos devido à sua robustez em **relacionamentos** e consultas estruturadas.

2. Database 2 (MongoDB): Configurações de Plataformas

- MongoDB seria adequado para armazenar configurações de plataformas de e-commerce porque oferece:
  - Flexibilidade no armazenamento de dados variáveis.
  - Suporte para JSON-like documents, o que é ideal para armazenar credenciais de APIs que podem variar conforme a plataforma.

Como Configurar o `id_cliente` Entre Bancos de Dados Diferentes

Cenário: Relacionamento entre MySQL e MongoDB

- O `id_cliente` gerado na Database 1 (MySQL) pode ser usado como **chave de referência** na Database 2 (MongoDB). Veja como:

1. `id_cliente` **como UUID**:
   - Gere o `id_cliente` como um **UUID** (Universally Unique Identifier) no MySQL ao criar um cliente.
   - Exemplo: `f4a3c17e-bc5d-4114-909a-1345e1f9a39c`.
   - Salve esse mesmo ID na coleção do MongoDB como um campo.
2. **Sincronização entre os Bancos**:
   - Quando um cliente é criado no MySQL:
     - Crie um documento correspondente no MongoDB para armazenar as configurações de plataformas.
     - Exemplo no MongoDB:

```
{
  "_id": "f4a3c17e-bc5d-4114-909a-1345e1f9a39c",
  "platform_credentials": [
    {
      "platform": "Mercado Livre",
      "api_key": "xyz123",
      "api_secret": "abc456"
    },
    {
      "platform": "Amazon",
      "api_key": "pqr789",
```

```
      "api_secret": "lmn012"
    }
  ]
}
```

3. **Consulta Unificada**:
   - Use uma camada intermediária na API (ex.: em C# ou Python) para unificar as consultas:
     - Consulte o MySQL para os dados do cliente.
     - Consulte o MongoDB para as configurações das plataformas.

Usando o Mesmo Banco de Dados

Se você optar por usar o mesmo banco de dados (por exemplo, MySQL para ambas as databases), o `id_cliente` seria configurado como uma **chave estrangeira** diretamente:

1. **Chave Estrangeira Relacional**:
   - A tabela `credenciais` no MySQL teria um campo `id_cliente` referenciando a tabela `clientes`:

```
ALTER TABLE credenciais
ADD CONSTRAINT fk_cliente
FOREIGN KEY (id_cliente) REFERENCES clientes(id_clien
te);
```

2. **Vantagens**:
   - Consultas SQL únicas para unir informações de clientes e plataformas:

```
SELECT c.nome, p.nome_plataforma, cr.api_key
FROM clientes c
JOIN credenciais cr ON c.id_cliente = cr.id_cliente
JOIN plataformas p ON cr.id_plataforma = p.id_platafo
rma;
```

Implementação Técnica

1. Integração Entre MySQL e MongoDB

- Ferramentas como **Apache Kafka** ou **Debezium** podem ser usadas para sincronizar eventos entre bancos.
- Exemplo em Python:
  - Criar cliente no MySQL:

```
mysql_cursor.execute("INSERT INTO clientes (id_client
e, nome, email) VALUES (%s, %s, %s)", (uuid, nome, em
ail))
```

  - Adicionar configuração no MongoDB:

```
mongo_db.platform_credentials.insert_one({
    "_id": uuid,
    "platform_credentials": []
})
```

2. Operações no MongoDB com o `id_cliente`

- Para vincular plataformas:

```
mongo_db.platform_credentials.update_one(
    {"_id": "f4a3c17e-bc5d-4114-909a-1345e1f9a39c"},
    {"$push": {
        "platform_credentials": {
            "platform": "Mercado Livre",
            "api_key": "xyz123",
            "api_secret": "abc456"
        }
    }}
)
```

Comparação Entre os Cenários

Aspecto	Bancos Separados (MySQL + MongoDB)	Banco Único (MySQL ou PostgreSQL)
Complexidade de Integração	Alta: Requer sincronia entre bancos	Baixa: Relacionamentos nativos.
Escalabilidade	Alta: Cada banco pode escalar separadamente	Média: Menos flexível para dados não estruturados.
Desempenho	Alta para consultas específicas	Boa para consultas gerais.
Custo de Manutenção	Alto: Mais ferramentas e camadas envolvidas	Baixo: Simplicidade na manutenção.

Aplicação API Microsserviços (CORAÇÃO DO PROJETO):

Esta API estará responsável por lidar com todas as requisições as Plataformas E-commerce, informando chaves de acesso principais e passando as informações dos usuários e dos produtos. Em uma arquitetura de microsserviços, cada microsserviço ficaria responsável de uma Plataforma e-commerce diferente, lidando da melhor maneira com suas requisições individuais. Também ficaria responsável de receber notificações sobre os produtos e seu status, passando as informações ao usuário vendedor daquele produto.

Possíveis linguagens: Acredito que dependeria muito da plataforma e-commerce ao qual o microsserviço está se conectando, pois cada uma pode ter uma especificação diferente para conexão, sendo assim diferentes linguagens podem ser adotadas. Porém um modo padrão deve ser adotado por todas para conexão ao Banco de Dados e a API MVC.

1. Estrutura dos Microsserviços

1.1. Cada Microsserviço é Independente

- Cada microsserviço é responsável por:

- Realizar a autenticação com a plataforma de e-commerce (ex.: API keys, OAuth).
- Gerenciar requisições específicas da plataforma, como:
  - Cadastro de produtos.
  - Atualização de preços e estoques.
  - Recebimento de notificações (ex.: venda realizada, produto desativado).
- Adaptar as particularidades da API da plataforma para um formato padrão (interno).

1.2. Comunicação com o Banco de Dados

- Cada microsserviço se comunica com o **Banco de Dados Centralizado** para:
  - Obter credenciais da plataforma (chaves API, tokens).
  - Sincronizar dados dos produtos com as tabelas do cliente.

1.3. Comunicação com a API Back-End Geral

- Todos os microsserviços devem se comunicar com a **API MVC** por um protocolo padrão, como **HTTP/REST** ou **gRPC**, garantindo:
  - Respostas padronizadas para a interface do usuário.
  - Notificações enviadas para o usuário (via WebSocket, push ou polling).

---

2. Linguagens de Programação

2.1. Critérios de Escolha

1. **Compatibilidade com a API da Plataforma de E-commerce**:
   - Linguagens com suporte nativo ou bibliotecas para os protocolos usados pela plataforma.
2. **Padronização no Projeto**:
   - Uma linguagem padrão para os microsserviços ajuda a manter o código consistente, mas não é obrigatória.
3. **Desempenho e Manutenção**:

- Linguagens fáceis de escalar e manter.

2.2. Recomendações por Cenário

Padrão Universal (Sugestão Mais Consistente)

- Use **Python**:
  - **Vantagens**:
    - Bibliotecas robustas para HTTP, JSON e APIs REST (como `requests`, `Flask`).
    - Facilidade para lidar com autenticação OAuth ou JWT.
    - Boa integração com MongoDB, MySQL e Redis.
    - Ideal para prototipagem rápida e manutenção.
  - **Exemplo**:
    - Microsserviço para Amazon:

```
from flask import Flask, request, jsonify
app = Flask(__name__)

@app.route('/add_product', methods=['POST'])
def add_product():
    # Processa o JSON enviado
    data = request.json
    # Conecta à API da Amazon e registra o produto
    return jsonify({"status": "Produto registrado
com sucesso!"})
```

Cenários de Alto Desempenho ou Linguagens Orientadas à Plataforma

- **Java**:
  - Suporte excelente para integrações empresariais e APIs complexas.
  - Recomendado para **Mercado Livre** e plataformas que exijam estruturas robustas.
- **Node.js**:

    - Ideal para APIs que demandam comunicação assíncrona e alta performance.
    - Bom para **Shopify** ou outras plataformas baseadas em eventos (Webhooks).
- **C#**:
    - Altamente recomendado para integrações com Microsoft Azure ou outras ferramentas do ecossistema Microsoft.

---

3. Comunicação Entre Microsserviços

3.1. Protocolos Padrão

1. **REST API**:
    - Simples, fácil de implementar e monitorar.
    - Respostas padronizadas em JSON.
2. **gRPC (Google RPC)**:
    - Mais eficiente para comunicação de microsserviços internos.
    - Suporte a múltiplas linguagens.
    - Melhor desempenho do que REST para altas cargas.
3. **Mensageria Assíncrona** (Ex.: RabbitMQ, Kafka):
    - Microsserviços que recebem notificações de status podem usar filas para processar eventos em tempo real.
    - Exemplo: Uma venda confirmada dispara um evento no Kafka, que é consumido pelo microsserviço de notificações.

---

4. Notificações para o Usuário Vendedor

4.1. Tipos de Notificações

- **Status do Produto**:
    - Produto aprovado, rejeitado ou em análise.
- **Venda Realizada**:
    - Confirmação de venda com dados do comprador.

- **Mudança de Estoque**:
  - Notificação sobre baixa ou alta no estoque.

4.2. Métodos de Envio

1. **Push Notifications**:
   - Usando Firebase Cloud Messaging (FCM) para enviar notificações diretamente para o aplicativo.
2. **WebSocket**:
   - Para aplicativos web ou mobile que precisem de atualizações em tempo real.
3. **Polling**:
   - Requisições periódicas da interface para verificar notificações (menos eficiente, mas mais simples de implementar).

---

5. Integração de Bancos de Dados

Se os bancos de dados forem diferentes:

- Cada microsserviço acessa o banco de dados correto conforme a informação que ele necessita.
- O `id_cliente` é passado entre microsserviços como referência principal para garantir a coesão.

Exemplo de Fluxo:

1. **Microsserviço "Mercado Livre"**:
   - Obtém o `id_cliente` e acessa o MongoDB para recuperar credenciais da plataforma.
2. **Microsserviço "Amazon"**:
   - Usa o mesmo `id_cliente` para obter informações de produtos no MySQL.

---

6. Exemplo de Microsserviço em Python

Microsserviço para o Mercado Livre

```
from flask import Flask, request, jsonify
import requests

app = Flask(__name__)

@app.route('/post_product', methods=['POST'])
def post_product():
    # Dados do cliente enviados pelo Front-end ou API Back-
end Geral
    data = request.json
    id_cliente = data['id_cliente']
    produto = data['produto']

    # Recupera credenciais do MongoDB
    credentials = get_credentials_from_mongodb(id_cliente,
"Mercado Livre")

    # Conecta à API do Mercado Livre
    api_response = requests.post(
        url="https://api.mercadolibre.com/items",
        headers={"Authorization": f"Bearer {credentials['ac
cess_token']}"},
        json=produto
    )

    return jsonify(api_response.json())

def get_credentials_from_mongodb(id_cliente, plataforma):
    # Simulação de credenciais recuperadas do MongoDB
    return {
        "access_token": "xyz123"
    }

if __name__ == '__main__':
    app.run(debug=True)
```

7. Exemplo de Integração Entre Dois Microsserviços

A seguir, vou criar dois microsserviços: um para o **Mercado Livre** e outro para a **Amazon**. Eles irão demonstrar como o `id_cliente` pode ser compartilhado entre os microsserviços e como as informações podem ser sincronizadas.

Microsserviço 1: Mercado Livre

Este microsserviço recebe a requisição para adicionar um produto e utiliza o `id_cliente` para recuperar credenciais do banco de dados e publicar o produto na API do Mercado Livre. Além disso, ele também envia uma notificação para o microsserviço da **Amazon**, informando que um novo produto foi criado.

```
from flask import Flask, request, jsonify
import requests

app = Flask(__name__)

# Rota para adicionar um produto ao Mercado Livre
@app.route('/mercadolivre/add_product', methods=['POST'])
def add_product_mercadolivre():
    data = request.json
    id_cliente = data['id_cliente']
    produto = data['produto']

    # Recupera credenciais do banco de dados MongoDB
    credentials = get_credentials_from_mongodb(id_cliente,
"Mercado Livre")

    # Faz a requisição à API do Mercado Livre
    mercado_livre_response = requests.post(
        url="https://api.mercadolibre.com/items",
        headers={"Authorization": f"Bearer {credentials['ac
cess_token']}"},
        json=produto
    )

    # Se o produto foi criado com sucesso, notifica o micro
sserviço da Amazon
```

```
    if mercado_livre_response.status_code == 201:
        notify_amazon_service(id_cliente, produto)

    return jsonify({"mercadolivre_response": mercado_livre_
response.json()})

# Função para recuperar credenciais do MongoDB (simulação)
def get_credentials_from_mongodb(id_cliente, plataforma):
    # Aqui você faria uma consulta real ao banco
    return {
        "access_token": "token_mercadolivre_123"
    }

# Função para notificar o microsserviço da Amazon
def notify_amazon_service(id_cliente, produto):
    amazon_service_url = "http://localhost:5001/amazon/sync
_product"
    payload = {
        "id_cliente": id_cliente,
        "produto": produto
    }
    response = requests.post(amazon_service_url, json=paylo
ad)
    print(f"Notificação enviada para a Amazon: {response.st
atus_code}")

if __name__ == '__main__':
    app.run(port=5000, debug=True)
```

Microsserviço 2: Amazon

Este microsserviço recebe notificações do microsserviço do Mercado Livre e utiliza o `id_cliente` para sincronizar o produto na API da Amazon.

```
from flask import Flask, request, jsonify
import requests

app = Flask(__name__)
```

```
# Rota para sincronizar produto na Amazon
@app.route('/amazon/sync_product', methods=['POST'])
def sync_product_amazon():
    data = request.json
    id_cliente = data['id_cliente']
    produto = data['produto']

    # Recupera credenciais do banco de dados MongoDB
    credentials = get_credentials_from_mongodb(id_cliente,
"Amazon")

    # Faz a requisição à API da Amazon para adicionar o pro
duto
    amazon_response = requests.post(
        url="https://api.amazon.com/products",
        headers={"Authorization": f"Bearer {credentials['ac
cess_token']}"},
        json=produto
    )

    return jsonify({"amazon_response": amazon_response.json
()})

# Função para recuperar credenciais do MongoDB (simulação)
def get_credentials_from_mongodb(id_cliente, plataforma):
    # Aqui você faria uma consulta real ao banco
    return {
        "access_token": "token_amazon_456"
    }

if __name__ == '__main__':
    app.run(port=5001, debug=True)
```

Fluxo de Comunicação Entre os Microsserviços

1. **Microsserviço do Mercado Livre**:
    - Recebe uma requisição para cadastrar um produto.

- Recupera o `id_cliente` e as credenciais correspondentes do MongoDB.
- Cadastra o produto na API do Mercado Livre.
- Envia uma notificação para o **Microsserviço da Amazon**, passando o `id_cliente` e os dados do produto.

2. **Microsserviço da Amazon**:
   - Recebe a notificação do Mercado Livre.
   - Utiliza o mesmo `id_cliente` para recuperar credenciais correspondentes.
   - Cadastra o produto na API da Amazon.

Exemplo de Requisição para o Microsserviço do Mercado Livre

Simule uma requisição ao microsserviço do Mercado Livre com o seguinte `cURL`:

```
curl -X POST http://localhost:5000/mercadolivre/add_product
\
-H "Content-Type: application/json" \
-d '{
    "id_cliente": "12345",
    "produto": {
        "titulo": "Smartphone XYZ",
        "preco": 1500.00,
        "estoque": 10
    }
}'
```

- O microsserviço do Mercado Livre irá:
  1. Adicionar o produto na API do Mercado Livre.
  2. Enviar uma notificação para o microsserviço da Amazon.

Como o `id_cliente` Funciona Entre Bancos de Dados Diferentes

1. O `id_cliente` é a chave comum entre os bancos de dados (MySQL e MongoDB).

2. Cada microsserviço consulta o banco correto usando o `id_cliente` como identificador para acessar:
    - Credenciais da plataforma no MongoDB.
    - Dados adicionais do cliente ou produtos no MySQL.