Faculdade

## Requisitos e Funcionalidades

O desafio consiste em desenvolver uma API RESTful utilizando Python com Flask, seguindo o padrão arquitetural MVC, Clean Code e padrões UML. A API deve expor um endpoint capaz de realizar operações CRUD (Create, Read, Update, Delete) sobre um domínio escolhido (Clientes, Produtos ou Pedidos), além de funcionalidades adicionais como contagem total de registros, busca por todos os registros, busca por ID e busca por nome.

### Funcionalidades da API:

- **CRUD:** Criação (Create), Leitura (Read), Atualização (Update) e Exclusão (Delete).
- **Contagem:** Endpoint para retornar o número total de registros.
- **Find All:** Endpoint para retornar todos os registros.
- **Find By ID:** Endpoint para retornar um registro específico com base no ID.
- **Find By Name:** Endpoint para retornar registros que correspondam a um nome específico.

### Arquitetura e Boas Práticas:

- **Padrão Arquitetural:** MVC (Model-View-Controller).
- **Linguagem/Framework:** Python com Flask.
- **Organização do Código:** Cada componente (Controller, Service, Model) deve ser bem documentado e isolado em suas respectivas responsabilidades.
- **Persistência de Dados (Opcional):** Incluir persistência de dados será um diferencial.

### Entregáveis:

- **Desenho Arquitetural:** Diagrama UML e/ou C4 Model (ou outro de preferência) utilizando ferramentas como draw.io.
- **Estrutura de Pastas:** Apresentação da estrutura de pastas do projeto com breve descrição do papel de cada componente (Controller, Model, Service).
- **Código (Opcional):** Upload em repositório GitHub/GitLab.
- **Persistência (Opcional):** Código funcionando com persistência de dados.

# Escolha do Domínio

Para este desafio, o domínio escolhido para a implementação da API RESTful é **Clientes**. Este domínio permitirá a demonstração das operações CRUD e das funcionalidades adicionais de contagem e busca de forma clara e concisa, alinhando-se com os exemplos fornecidos no enunciado do desafio.

# Documentação da Arquitetura

## Visão Geral da Arquitetura

A solução proposta para o desafio final da pós-graduação em Arquitetura de Software consiste em uma API RESTful desenvolvida em Python utilizando o framework Flask. A arquitetura segue o padrão Model-View-Controller (MVC), que promove a separação de responsabilidades e facilita a manutenção e escalabilidade do código. O domínio escolhido para a implementação é o de **Clientes**, permitindo a demonstração de operações CRUD (Create, Read, Update, Delete) e funcionalidades adicionais de consulta.

## Padrão Arquitetural MVC

O padrão MVC é amplamente utilizado no desenvolvimento de aplicações web para organizar o código em três componentes principais, cada um com uma responsabilidade distinta:

- **Model (Modelo):** Representa os dados e a lógica de negócios da aplicação. No contexto desta API, o Modelo incluirá a entidade `Cliente` e a lógica para interagir com a persistência dos dados (simulada ou real).
- **View (Visão):** Responsável pela apresentação dos dados ao usuário. Em uma API RESTful, a "View" é tipicamente a representação dos dados em formato JSON ou XML que é retornada ao cliente. Não há uma interface de usuário gráfica tradicional neste tipo de aplicação.
- **Controller (Controlador):** Atua como um intermediário entre o Modelo e a Visão, recebendo as requisições do cliente, processando-as, interagindo com o Modelo para obter ou manipular dados e, em seguida, retornando a resposta apropriada. No nosso caso, o `ClienteController` será responsável por gerenciar as rotas da API e coordenar as operações.

Esta separação de preocupações facilita o desenvolvimento, teste e manutenção da aplicação, garantindo que as alterações em uma camada não afetem diretamente as outras.

## Diagrama de Classes UML

O diagrama de classes UML a seguir ilustra a estrutura das classes e interfaces que compõem a camada de Model, Service e Controller da aplicação, bem como suas relações. Ele detalha os atributos e métodos de cada componente, fornecendo uma visão clara da organização do código e das responsabilidades de cada parte.

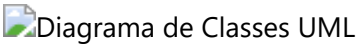
Diagrama de Classes UML

**Explicação dos Componentes:**

- `Cliente` **(Classe):** Representa a entidade de domínio. Contém atributos como `id`, `nome` e `email`, além de métodos para construção e acesso (Getters e Setters).

- `ClienteRepository` **(Interface):** Define o contrato para operações de persistência de dados relacionadas à entidade `Cliente`. Inclui métodos para buscar todos os clientes, buscar por ID, buscar por nome, salvar, deletar e contar clientes. Esta interface será implementada por uma classe concreta que interage com o banco de dados (ou uma simulação).
- `ClienteService` **(Classe):** Contém a lógica de negócios da aplicação. Atua como uma camada intermediária entre o `ClienteController` e o `ClienteRepository`, orquestrando as operações de dados e aplicando regras de negócio. Possui uma dependência do `ClienteRepository` para realizar as operações de persistência.
- `ClienteController` **(Classe):** Responsável por receber as requisições HTTP, processá-las e retornar as respostas. Mapeia os endpoints da API para os métodos correspondentes no `ClienteService`. Possui uma dependência do `ClienteService` para executar as operações de negócio.

## Diagrama de Componentes UML

O diagrama de componentes UML a seguir ilustra a organização e as dependências entre os principais componentes da arquitetura MVC da aplicação. Ele fornece uma visão de alto nível de como as diferentes partes do sistema interagem entre si.


Diagrama de Componentes UML

**Explicação dos Componentes:**

- **Controller:** Representa a camada de controle da aplicação, responsável por receber as requisições e coordenar a resposta. O `ClienteController` é o principal componente nesta camada.
- **Service:** Representa a camada de serviço, onde a lógica de negócios é implementada. O `ClienteService` é o componente chave aqui, orquestrando as operações e interagindo com a camada de repositório.
- **Repository:** Representa a camada de acesso a dados, responsável pela interação com o mecanismo de persistência. O `ClienteRepository` define a interface para essas operações.
- **Model:** Representa a camada de dados, contendo as entidades de domínio. A entidade `Cliente` é o componente principal nesta camada.

As setas indicam as dependências entre os componentes, mostrando o fluxo de controle e dados na aplicação.

## Diagrama de Sequência UML para Operações CRUD

O diagrama de sequência UML detalha a interação entre os diferentes componentes da aplicação para cada uma das operações CRUD (Create, Read, Update, Delete) para a entidade `Cliente`. Ele ilustra a ordem das chamadas de método e o fluxo de dados entre o usuário, o controlador, o serviço, o repositório e o banco de dados.

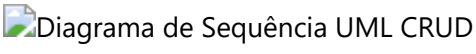
Diagrama de Sequência UML CRUD

**Fluxo de Operações:**

- **Criação (POST /clientes):** O usuário envia uma requisição POST com os dados do cliente para o `ClienteController`. O controlador invoca o método `salvar` no `ClienteService`, que por sua vez chama o método `save` no `ClienteRepository`. O repositório interage com o banco de dados para inserir o novo cliente e retorna o cliente salvo através das camadas de serviço e controlador de volta ao usuário.
- **Leitura por ID (GET /clientes/{id}):** O usuário envia uma requisição GET com o ID do cliente. O `ClienteController` chama `buscarPorId` no `ClienteService`, que delega a chamada para `findById` no `ClienteRepository`. O repositório consulta o banco de dados e retorna o cliente encontrado (ou `None` se não encontrado) através das camadas de serviço e controlador de volta ao usuário.
- **Atualização (PUT /clientes/{id}):** Similar à criação, o usuário envia uma requisição PUT com os dados atualizados do cliente. O fluxo segue do `ClienteController` para o `ClienteService` e `ClienteRepository`, que atualiza o registro no banco de dados e retorna o cliente atualizado.
- **Exclusão (DELETE /clientes/{id}):** O usuário envia uma requisição DELETE com o ID do cliente a ser removido. O `ClienteController` chama `deletar` no `ClienteService`, que invoca `deleteById` no `ClienteRepository`. O repositório remove o cliente do banco de dados e as camadas de serviço e controlador confirmam a operação ao usuário.

Este diagrama fornece uma compreensão detalhada do comportamento dinâmico da aplicação e das interações entre seus componentes.

## Estrutura de Pastas e Componentes

A estrutura de pastas do projeto foi organizada seguindo as melhores práticas de desenvolvimento e o padrão arquitetural MVC. A organização clara e lógica facilita a manutenção, escalabilidade e compreensão do código por parte da equipe de desenvolvimento.

```
cliente_api/
├── venv/                          # Ambiente virtual Python
├── src/                           # Código fonte da aplicação
│   ├── models/                    # Camada Model (Modelo)
│   │   ├── __init__.py
│   │   ├── cliente.py             # Entidade Cliente
│   │   └── cliente_repository.py  # Repositório de Clientes
│   ├── services/                  # Camada Service (Lógica de Negócios)
│   │   ├── __init__.py
│   │   └── cliente_service.py     # Serviço de Clientes
│   ├── routes/                    # Camada Controller (Controlador)
│   │   ├── __init__.py
│   │   └── cliente.py             # Controlador de Clientes
│   ├── static/                    # Arquivos estáticos
│   ├── database/                  # Banco de dados SQLite
│   │   └── app.db                 # Arquivo do banco de dados
│   └── main.py                    # Arquivo principal da aplicação
├── tests/                         # Testes automatizados
│   ├── __init__.py
│   ├── test_cliente.py            # Testes do modelo Cliente
│   ├── test_cliente_repository.py  # Testes do repositório
│   ├── test_cliente_service.py    # Testes do serviço
```

```
│   └── test_cliente_controller.py  # Testes do controlador
└── requirements.txt                # Dependências do projeto
```

**Explicação dos Componentes:**

**Camada Model (src/models/):** Esta camada contém as entidades de domínio e a lógica de acesso a dados. O arquivo `cliente.py` define a entidade `Cliente` com seus atributos e métodos, enquanto `cliente_repository.py` implementa o padrão Repository para abstrair as operações de persistência de dados. Esta separação permite que a lógica de negócios seja independente da tecnologia de persistência utilizada.

**Camada Service (src/services/):** A camada de serviços contém a lógica de negócios da aplicação. O `cliente_service.py` orquestra as operações entre o controlador e o repositório, aplicando regras de validação, transformações de dados e coordenando transações complexas. Esta camada garante que as regras de negócio sejam centralizadas e reutilizáveis.

**Camada Controller (src/routes/):** Os controladores são responsáveis por receber as requisições HTTP, processá-las e retornar as respostas apropriadas. O arquivo `cliente.py` nesta pasta define os endpoints da API RESTful e mapeia as operações HTTP para os métodos correspondentes no serviço. Esta camada atua como a interface entre o mundo externo e a lógica interna da aplicação.

**Testes (tests/):** A pasta de testes contém uma suíte abrangente de testes automatizados que cobrem todas as camadas da aplicação. Os testes unitários verificam o comportamento individual de cada componente, enquanto os testes de integração validam a interação entre diferentes partes do sistema. Esta estrutura de testes garante a qualidade e confiabilidade do código.

## Endpoints da API RESTful

A API de Clientes implementa um conjunto completo de endpoints que seguem os princípios REST e fornecem todas as funcionalidades CRUD solicitadas no desafio, além de operações adicionais de consulta e contagem. Todos os endpoints retornam dados em formato JSON e implementam tratamento adequado de erros.

### Endpoint de Verificação de Saúde

#### GET /api/health

Este endpoint fornece uma verificação rápida do status da API, útil para monitoramento e verificação de disponibilidade do serviço.

- **Método:** GET
- **URL:** `/api/health`
- **Resposta de Sucesso (200):**

```
{
  "status": "OK",
  "message": "API de Clientes está funcionando"
}
```

**Operações CRUD para Clientes**

**1. Criar Cliente (Create)**

**POST /api/clientes**

Cria um novo cliente no sistema com os dados fornecidos no corpo da requisição.

- **Método:** POST
- **URL:** `/api/clientes`
- **Content-Type:** `application/json`
- **Corpo da Requisição:**

```
{
  "nome": "João Silva",
  "email": "joao.silva@email.com"
}
```

- **Resposta de Sucesso (201):**

```
{
  "id": 1,
  "nome": "João Silva",
  "email": "joao.silva@email.com"
}
```

- **Resposta de Erro (400):**

```
{
  "erro": "Nome é obrigatório"
}
```

**Validações Implementadas:**

- Nome é obrigatório e deve ter entre 2 e 100 caracteres
- Email é obrigatório, deve ter formato válido e no máximo 120 caracteres
- Email deve ser único no sistema
- Content-Type deve ser `application/json`

**2. Listar Todos os Clientes (Read All)**

**GET /api/clientes**

Retorna uma lista com todos os clientes cadastrados no sistema.

- **Método:** GET
- **URL:** `/api/clientes`

- **Resposta de Sucesso (200):**

```
[
  {
    "id": 1,
    "nome": "João Silva",
    "email": "joao.silva@email.com"
  },
  {
    "id": 2,
    "nome": "Maria Santos",
    "email": "maria.santos@email.com"
  }
]
```

**3. Buscar Cliente por ID (Read by ID)**

**GET /api/clientes/{id}**

Retorna os dados de um cliente específico baseado no seu ID.

- **Método:** GET
- **URL:** `/api/clientes/{id}`
- **Parâmetros de URL:** `id` (integer) - ID do cliente
- **Resposta de Sucesso (200):**

```
{
  "id": 1,
  "nome": "João Silva",
  "email": "joao.silva@email.com"
}
```

- **Resposta de Erro (404):**

```
{
  "erro": "Cliente não encontrado"
}
```

**4. Atualizar Cliente (Update)**

**PUT /api/clientes/{id}**

Atualiza os dados de um cliente existente.

- **Método:** PUT
- **URL:** `/api/clientes/{id}`
- **Parâmetros de URL:** `id` (integer) - ID do cliente

- **Content-Type:** `application/json`
- **Corpo da Requisição:**

```
{
  "nome": "João Silva Atualizado",
  "email": "joao.novo@email.com"
}
```

- **Resposta de Sucesso (200):**

```
{
  "id": 1,
  "nome": "João Silva Atualizado",
  "email": "joao.novo@email.com"
}
```

**5. Deletar Cliente (Delete)**

**DELETE /api/clientes/{id}**

Remove um cliente do sistema baseado no seu ID.

- **Método:** DELETE
- **URL:** `/api/clientes/{id}`
- **Parâmetros de URL:** `id` (integer) - ID do cliente
- **Resposta de Sucesso (204):** Sem conteúdo
- **Resposta de Erro (404):**

```
{
  "erro": "Cliente não encontrado"
}
```

**Operações Adicionais**

**6. Buscar Clientes por Nome**

**GET /api/clientes/nome/{nome}**

Busca clientes cujo nome contenha a string fornecida (busca parcial, case-insensitive).

- **Método:** GET
- **URL:** `/api/clientes/nome/{nome}`
- **Parâmetros de URL:** `nome` (string) - Nome ou parte do nome do cliente
- **Resposta de Sucesso (200):**

```
[
  {
    "id": 1,
    "nome": "João Silva",
    "email": "joao.silva@email.com"
  }
]
```

**7. Contar Total de Clientes**

**GET /api/clientes/contar**

Retorna o número total de clientes cadastrados no sistema.

- **Método:** GET
- **URL:** `/api/clientes/contar`
- **Resposta de Sucesso (200):**

```
{
  "total": 5
}
```

**Tratamento de Erros**

A API implementa um sistema robusto de tratamento de erros que retorna códigos de status HTTP apropriados e mensagens de erro descritivas:

- **400 Bad Request:** Dados inválidos ou malformados
- **404 Not Found:** Recurso não encontrado
- **500 Internal Server Error:** Erro interno do servidor

Todos os erros retornam uma resposta JSON no formato:

```
{
  "erro": "D
(Content truncated due to size limit. Use line ranges to read in chunks)"
}
```

**Link do Projeto**

Repositório Github