Relatório do Meu Projeto: API Simples de Imagens com FastAPI

Oi, aqui é o relatório que eu escrevi sobre o código que desenvolvi para uma API simples de gerenciamento de imagens. Usei FastAPI para criar uma galeria onde posso fazer upload, listar, buscar, baixar e deletar fotos via uma interface web básica. Vou explicar as funcionalidades, as metodologias que apliquei e responder a perguntas comuns que eu mesmo poderia ter, como por que escolhi certas abordagens.

Introdução ao Meu Projeto

Objetivo: Eu queria criar uma API RESTful leve para gerenciar uma galeria de imagens, com um frontend HTML simples (sem frameworks externos como React). Priorizei a simplicidade, para que fosse fácil de usar e integrar backend com frontend.

Tecnologias que Usei: Python com FastAPI (para o backend), HTML/CSS/JS inline (para o frontend), e bibliotecas como PIL (para gerar um favicon automático).

Estrutura de Arquivos:

- main.py: Meu código principal da API.
- static/index.html: Arquivo HTML separado para o frontend (decidi separar para não deixar o Python bagunçado).
- images/: Pasta para armazenar as imagens enviadas (ela se cria sozinha).
- Como Executo: Rodo python main.py (que usa Uvicorn internamente) e acesso http://127.0.0.1:8000/ no navegador.

Funcionalidades do Meu Código

O código que criei implementa uma API completa para gerenciar imagens, com endpoints REST e uma interface web integrada. Aqui vão as principais funcionalidades que adicionei:

Servir Interface Web:

Endpoint GET /: Serve o arquivo static/index.html, que tem um formulário de upload, um campo de busca e uma galeria de imagens. Incluí CSS inline para deixar bonitinho e JS para as interações (como upload via fetch e busca em tempo real).

Favicon automático: Gero um ícone azul simples com PIL se o static/favicon.ico não existir, para não dar erro 404 no navegador.

Gerenciamento de Imagens:

Upload: POST /upload/: Recebe arquivos de imagem via FormData, salva na pasta images/ e retorna uma confirmação. Aceita qualquer tipo, mas o HTML restringe a imagens.

Listagem: GET /list/: Retorna um JSON com a lista de nomes dos arquivos em images/.

Busca: GET /search/{query}: Filtra as imagens por nome parcial (não diferencia maiúsculas/minúsculas), útil para encontrar fotos específicas.

Download: GET /download/{filename}: Permite baixar uma imagem como arquivo.

Exclusão: DELETE /delete/{filename}: Remove uma imagem do servidor, verificando se ela existe antes.

Recursos Técnicos:

CORS Habilitado: Configurei para permitir requisições do frontend no mesmo domínio, sem bloqueios.

Tratamento de Erros: Usei HTTPException para respostas padronizadas (como 404 se a imagem não existir).

Execução Direta: O código roda o servidor sozinho ao executar python main.py, sem precisar de comandos extras.

Separação de Arquivos: Coloquei o HTML em static/ para editar facilmente; o backend fica focado na API.

Metodologias que Apliquei

No desenvolvimento, segui boas práticas para manter tudo organizado e fácil de manter:

Arquitetura RESTful:

Os endpoints seguem convenções REST (GET para ler, POST para criar, DELETE para excluir). Cada rota tem uma docstring explicando o que faz.

FastAPI gera documentação automática (Swagger em /docs).

Separação de Responsabilidades:

Backend (Python): Cuida da lógica, armazenamento e APIs.

Frontend (HTML/JS): Gerencia a interface e interações. O JS usa fetch para falar com a API sem recarregar a página.

Arquivos estáticos montados com StaticFiles para servir HTML/CSS/JS de forma eficiente.

Tratamento de Arquivos e Segurança:

Sempre verifico se o arquivo existe antes de deletar ou baixar.

Usei caminhos absolutos com os.path.join para evitar problemas de segurança.

CORS configurado para localhost, mas posso restringir para produção.

Desenvolvimento Iterativo:

Código modular com seções comentadas (configurações, middleware, rotas).

Testei manualmente no navegador e com Postman.

Reload automático com Uvicorn para mudanças rápidas.

Dependências e Portabilidade:

Bibliotecas mínimas: FastAPI, Uvicorn, PIL (opcional).

Compatível com Python 3.7+, roda em qualquer sistema com suporte a arquivos locais.

Respostas a Perguntas que Eu Poderia Fazer

Aqui, respondo dúvidas que eu mesmo tive durante o desenvolvimento, justificando minhas escolhas:

Por que usar FastAPI em vez de Flask/Django?

- Escolhi FastAPI porque é assíncrono e moderno, com validação automática de dados via Pydantic, docs OpenAPI prontas e melhor performance para APIs com I/O (como upload). É mais simples que Django para algo pequeno e evita o boilerplate de Flask.

Por que embutir HTML no código inicialmente e depois separar?

- No começo, embuti para testar rápido e evitar problemas com arquivos externos (como o erro que tive com o HTML não sendo encontrado). Depois, separei porque é melhor para manutenção — edito o HTML sem mexer no Python, e o código fica mais limpo.

Por que usar CORS e como configurei?

- CORS permite que o frontend faça requisições para a API sem erros de segurança do navegador. Configurei como allow_origins=["*"] para desenvolvimento local; em produção, restringiria a domínios específicos (ex.: ["http://meusite.com"]).

Por que o bloco EXECUÇÃO DIRETA if __name__ == "__main__": import uvicorn uvicorn.run("main:app", host="127.0.0.1", port=8000, reload=True)?

- Usei isso para executar o script diretamente com python main.py sem precisar rodar uvicorn main:app no terminal. O if __name__ == "__main__" garante que só roda quando executo o arquivo como principal (não quando importo). Uvicorn é o servidor ASGI para FastAPI, com reload para desenvolvimento (reinicia sozinho em mudanças). Host 127.0.0.1 limita a localhost por segurança; porta 8000 é padrão para dev.

Por que gerar favicon automaticamente com PIL?

- Para não dar erro 404 se o arquivo não existir. PIL cria um ícone simples na memória, sem precisar de arquivos extras. Em produção, eu colocaria um favicon.ico real em static/.

Por que não validar tipos de imagem no upload?

- Para manter simples, aceito qualquer arquivo, mas o HTML já restringe com accept="image/*". Se precisar, adicionaria uma checagem como if not file.content_type.startswith("image/").

Como lidar com muitas imagens (performance)?

- Para escalar, adicionaria paginação em /list/ (parâmetros skip e limit). Usaria um banco como SQLite em vez de arquivos para metadados.

Por que usar FileResponse para servir HTML?

- FileResponse serve arquivos estáticos de forma eficiente, com headers certos (como media_type="text/html"). É melhor que strings embutidas para arquivos grandes ou que eu queira editar.

Concluindo, esse relatório cobre tudo que fiz; o código é fácil de expandir para produção com autenticação ou deploy na nuvem. Se eu precisar de mais detalhes ou mudanças, posso ajustar!