Alternativas, Inspiração e Comparações

O que inspirou **FastAPI**, como ele se compara a outras alternativas e o que FastAPI aprendeu delas.

Introdução

FastAPI não poderia existir se não fosse pelos trabalhos anteriores de outras pessoas.

Houveram tantas ferramentas criadas que ajudaram a inspirar sua criação.

Tenho evitado criar um novo framework por anos. Primeiramente tentei resolver todos os recursos cobertos pelo **FastAPI** utilizando muitos frameworks diferentes, plug-ins e ferramentas.

Mas em algum ponto, não houve outra opção senão criar algo que fornecesse todos esses recursos, pegando as melhores idéias de ferramentas anteriores, e combinando eles da melhor forma possível, utilizando recursos da linguagem que não estavam disponíveis antes (*Type Hints* no Python 3.6+).

Ferramentas anteriores

Django

É o framework mais popular e largamente confiável. É utilizado para construir sistemas como o *Instagram*.

É bem acoplado com banco de dados relacional (como MySQL ou PostgreSQL), então, tendo um banco de dados NoSQL (como Couchbase, MongoDB, Cassandra etc) como a principal ferramenta de armazenamento não é muito fácil.

Foi criado para gerar HTML no *backend*, não para criar APIs utilizando um *frontend* moderno (como React, Vue.js e Angular) ou por outros sistemas (como dispositivos IoT) comunicando com ele.

Django REST Framework

Django REST framework foi criado para ser uma caixa de ferramentas flexível para construção de APIs web utilizando Django por baixo, para melhorar suas capacidades de API.

Ele é utilizado por muitas companhias incluindo Mozilla, Red Hat e Eventbrite.

Ele foi um dos primeiros exemplos de **documentação automática de API**, e essa foi especificamente uma das primeiras idéias que inspirou "a busca por" **FastAPI**.

!!! note "Nota" Django REST Framework foi criado por Tom Christie. O mesmo criador de Starlette e Uvicorn, nos quais **FastAPI** é baseado.

!!! check "**FastAPI** inspirado para" Ter uma documentação automática da API em interface web.

Flask

Flask é um "microframework", não inclui integração com banco de dados nem muitas das coisas que vêm por padrão no Django.

Sua simplicidade e flexibilidade permitem fazer coisas como utilizar bancos de dados NoSQL como principal sistema de armazenamento de dados.

Por ser tão simples, é relativamente intuitivo de aprender, embora a documentação esteja de forma mais técnica em alguns pontos.

Ele é comumente utilizado por outras aplicações que não necessariamente precisam de banco de dados, gerenciamento de usuários, ou algum dos muitos recursos que já vem instalados no Django. Embora muitos desses recursos possam ser adicionados com plug-ins.

Esse desacoplamento de partes, e sendo um "microframework" que pode ser extendido para cobrir exatamente o que é necessário era um recurso chave que eu queria manter.

Dada a simplicidade do Flask, parecia uma ótima opção para construção de APIs. A próxima coisa a procurar era um "Django REST Framework" para Flask.

!!! check "**FastAPI** inspirado para" Ser um microframework. Fazer ele fácil para misturar e combinar com ferramentas e partes necessárias.

`Ser simples e com sistema de roteamento fácil de usar.`

Requests

FastAPI não é uma alternativa para **Requests**. O escopo deles é muito diferente.

Na verdade é comum utilizar Requests *dentro* de uma aplicação FastAPI.

Ainda assim, FastAPI pegou alguma inspiração do Requests.

Requests é uma biblioteca para interagir com APIs (como um cliente), enquanto **FastAPI** é uma biblioteca para *construir* APIs (como um servidor).

Eles estão, mais ou menos, em pontas opostas, um complementando o outro.

Requests tem um projeto muito simples e intuitivo, fácil de usar, com padrões sensíveis. Mas ao mesmo tempo, é muito poderoso e customizável.

É por isso que, como dito no site oficial:

> Requests é um dos pacotes Python mais baixados de todos os tempos

O jeito de usar é muito simples. Por exemplo, para fazer uma requisição `GET`, você deveria escrever:

```
response = requests.get("http://example.com/some/url")
```

A contra-parte da aplicação FastAPI, *rota de operação*, poderia parecer como:

```
Python hl_lines="1" @app.get("/some/url") def read_url():     return
{"message": "Hello World"}
```

Veja as similaridades em `requests.get(...)` e `@app.get(...)`.

!!! check "**FastAPI** inspirado para" * Ter uma API simples e intuitiva. * Utilizar nomes de métodos HTTP (operações) diretamente, de um jeito direto e intuitivo. * Ter padrões sensíveis, mas customizações poderosas.

Swagger / OpenAPI

O principal recurso que eu queria do Django REST Framework era a documentação automática da API.

Então eu descobri que existia um padrão para documentar APIs, utilizando JSON (ou YAML, uma extensão do JSON) chamado Swagger.

E tinha uma interface web para APIs Swagger já criada. Então, sendo capaz de gerar documentação Swagger para uma API poderia permitir utilizar essa interface web automaticamente.

Em algum ponto, Swagger foi dado para a Fundação Linux, e foi renomeado OpenAPI.

Isso acontece porquê quando alguém fala sobre a versão 2.0 é comum dizer "Swagger", e para a versão 3+, "OpenAPI".

!!! check "**FastAPI** inspirado para" Adotar e usar um padrão aberto para especificações API, ao invés de algum esquema customizado.

```
E integrar ferramentas de interface para usuários baseado nos padrões:

* <a href="https://github.com/swagger-api/swagger-ui" class="external-link" target="_blank">
* <a href="https://github.com/Rebilly/ReDoc" class="external-link" target="_blank">ReDoc</a>

Esses dois foram escolhidos por serem bem populares e estáveis, mas fazendo uma pesquisa rá
```

Flask REST frameworks

Existem vários Flask REST frameworks, mas depois de investir tempo e trabalho investigando eles, eu descobri que muitos estão descontinuados ou abandonados, com alguns tendo questões que fizeram eles inadequados.

Marshmallow

Um dos principais recursos necessários em sistemas API é "serialização" de dados, que é pegar dados do código (Python) e converter eles em alguma coisa que possa ser enviado através da rede. Por exemplo, converter um objeto contendo dados de um banco de dados em um objeto JSON. Converter objetos `datetime` em strings etc.

Outro grande recurso necessário nas APIs é validação de dados, certificando que os dados são válidos, dados certos parâmetros. Por exemplo, algum campo é `int`, e não alguma string aleatória. Isso é especialmente útil para dados que estão chegando.

Sem um sistema de validação de dados, você teria que realizar todas as verificações manualmente, no código.

Esses recursos são o que Marshmallow foi construído para fornecer. Ele é uma ótima biblioteca, e eu já utilizei muito antes.

Mas ele foi criado antes da existência do *type hints* do Python. Então, para definir todo o *schema* você precisa utilizar específicas ferramentas e classes fornecidas pelo Marshmallow.

!!! check "**FastAPI** inspirado para" Usar código para definir "schemas" que forneçam, automaticamente, tipos de dados e validação.

Webargs

Outro grande recurso necessário pelas APIs é a análise de dados vindos de requisições.

Webargs é uma ferramente feita para fornecer o que está no topo de vários frameworks, inclusive Flask.

Ele utiliza Marshmallow por baixo para validação de dados. E ele foi criado pelos mesmos desenvolvedores.

Ele é uma grande ferramenta e eu também a utilizei muito, antes de ter o **FastAPI**.

!!! info Webargs foi criado pelos mesmos desenvolvedores do Marshmallow.

!!! check "**FastAPI** inspirado para" Ter validação automática de dados vindos de requisições.

APISpec

Marshmallow e Webargs fornecem validação, análise e serialização como plug-ins.

Mas a documentação ainda está faltando. Então APISpec foi criado.

APISpec tem plug-ins para muitos frameworks (e tem um plug-in para Starlette também).

O jeito como ele funciona é que você escreve a definição do *schema* usando formato YAML dentro da *docstring* de cada função controlando uma rota.

E ele gera *schemas* OpenAPI.

É assim como funciona no Flask, Starlette, Responder etc.

Mas então, nós temos novamente o problema de ter uma micro-sintaxe, dentro de uma string Python (um grande YAML).

O editor não poderá ajudar muito com isso. E se nós modificarmos os parâmetros dos *schemas* do Marshmallow e esquecer de modificar também aquela *docstring* YAML, o *schema* gerado pode ficar obsoleto.

!!! info APISpec foi criado pelos mesmos desenvolvedores do Marshmallow.

!!! check "**FastAPI** inspirado para" Dar suporte a padrões abertos para APIs, OpenAPI.

Flask-apispec

É um plug-in Flask, que amarra junto Webargs, Marshmallow e APISpec.

Ele utiliza a informação do Webargs e Marshmallow para gerar automaticamente *schemas* OpenAPI, usando APISpec.

É uma grande ferramenta, mas muito subestimada. Ela deveria ser um pouco mais popular do que muitos outros plug-ins Flask. É de ser esperado que sua documentação seja bem concisa e abstrata.

Isso resolveu o problema de ter que escrever YAML (outra sintaxe) dentro das *docstrings* Python.

Essa combinação de Flask, Flask-apispec com Marshmallow e Webargs foi meu *backend stack* favorito até construir **FastAPI**.

Usando essa combinação levou a criação de vários geradores Flask *full-stack*. Há muitas *stacks* que eu (e vários times externos) estou utilizando até agora:

- https://github.com/tiangolo/full-stack
- https://github.com/tiangolo/full-stack-flask-couchbase
- https://github.com/tiangolo/full-stack-flask-couchdb

E esses mesmos geradores *full-stack* foram a base dos Geradores de Projetos **FastAPI**.

!!! info Flask-apispec foi criado pelos mesmos desenvolvedores do Marshmallow.

!!! check "**FastAPI** inspirado para" Gerar *schema* OpenAPI automaticamente, a partir do mesmo código que define serialização e validação.

NestJS (and Angular)

NestJS, que não é nem Python, é um framework NodeJS JavaScript (TypeScript) inspirado pelo Angular.

Ele alcança de uma forma similar ao que pode ser feito com o Flask-apispec.

Ele tem um sistema de injeção de dependência integrado, inspirado pelo Angular dois. É necessário fazer o pré-registro dos "injetáveis" (como todos os sistemas de injeção de dependência que conheço), então, adicionando verbosidade e repetição de código.

Como os parâmetros são descritos com tipos TypeScript (similar aos *type hints* do Python), o suporte ao editor é muito bom.

Mas como os dados TypeScript não são preservados após a compilação para o JavaScript, ele não pode depender dos tipos para definir a validação, serialização e documentação ao mesmo tempo. Devido a isso e a algumas decisões de projeto, para pegar a validação, serialização e geração automática do *schema*, é necessário adicionar decoradores em muitos lugares. Então, ele se torna muito verboso.

Ele também não controla modelos aninhados muito bem. Então, se o corpo JSON na requisição for um objeto JSON que contém campos internos que contém objetos JSON aninhados, ele não consegue ser validado e documentado apropriadamente.

!!! check "**FastAPI** inspirado para" Usar tipos Python para ter um ótimo suporte do editor.

```
Ter um sistema de injeção de dependência poderoso. Achar um jeito de minimizar repetição de
```

Sanic

Ele foi um dos primeiros frameworks Python extremamente rápido baseado em `asyncio`. Ele foi feito para ser muito similar ao Flask.

!!! note "Detalhes técnicos" Ele utiliza `uvloop` ao invés do '*loop*' `asyncio` padrão do Python. É isso que deixa ele tão rápido.

```
Ele claramente inspirou Uvicorn e Starlette, que são atualmente mais rápidos que o Sanic em
```

!!! check "**FastAPI** inspirado para" Achar um jeito de ter uma performance insana.

```
É por isso que o **FastAPI** é baseado em Starlette, para que ele seja o framework mais ráp
```

Falcon

Falcon é outro framework Python de alta performance, e é projetado para ser minimalista, e funciona como fundação de outros frameworks como Hug.

Ele usa o padrão anterior para frameworks web Python (WSGI) que é síncrono, então ele não pode controlar *WebSockets* e outros casos de uso. No entanto, ele também tem uma boa performance.

Ele é projetado para ter funções que recebem dois parâmetros, uma "requisição" e uma "resposta". Então você "lê" as partes da requisição, e "escreve" partes para a resposta. Devido ao seu design, não é possível declarar parâmetros de requisição e corpos com *type hints* Python padrão como parâmetros de funções.

Então, validação de dados, serialização e documentação tem que ser feitos no código, não automaticamente. Ou eles terão que ser implementados como um framework acima do Falcon, como o Hug. Essa mesma distinção acontece em outros frameworks que são inspirados pelo design do Falcon, tendo um objeto de requisição e um objeto de resposta como parâmetros.

!!! check "**FastAPI** inspirado para" Achar jeitos de conseguir melhor performance.

```
Juntamente com Hug (como Hug é baseado no Falcon) inspirou **FastAPI** para declarar um parâ

Embora no FastAPI seja opcional, é utilizado principalmente para configurar cabeçalhos, cook
```

Molten

Eu descobri Molten nos primeiros estágios da construção do **FastAPI**. E ele tem umas idéias bem similares:

- Baseado em *type hints* Python.
- Validação e documentação desses tipos.
- Sistema de injeção de dependência.

Ele não utiliza validação de dados, seriallização e documentação de bibliotecas de terceiros como o Pydantic, ele tem seu prórpio. Então, essas definições de tipo de dados não podem ser reutilizados tão facilmente.

Ele exige um pouco mais de verbosidade nas configurações. E como é baseado no WSGI (ao invés de ASGI), ele não é projetado para ter a vantagem da alta performance fornecida por ferramentas como Uvicorn, Starlette e Sanic.

O sistema de injeção de dependência exige pré-registro das dependências e as dependências são resolvidas baseadas nos tipos declarados. Então, não é possível declarar mais do que um "componente" que fornece um certo tipo.

Rotas são declaradas em um único lugar, usando funções declaradas em outros lugares (ao invés de usar decoradores que possam ser colocados diretamente acima da função que controla o *endpoint*). Isso é mais perto de como o Django faz isso do que como Flask (e Starlette) faz. Ele separa no código coisas que são relativamente amarradas.

!!! check "**FastAPI** inspirado para" Definir validações extras para tipos de dados usando valores "padrão" de atributos dos modelos. Isso melhora o suporte do

editor, e não estava disponível no Pydantic antes.

```
Isso na verdade inspirou a atualização de partes do Pydantic, para dar suporte ao mesmo esti
```

Hug

Hug foi um dos primeiros frameworks a implementar a declaração de tipos de parâmetros usando Python *type hints*. Isso foi uma ótima idéia que inspirou outras ferramentas a fazer o mesmo.

Ele usou tipos customizados em suas declarações ao invés dos tipos padrão Python, mas mesmo assim foi um grande passo.

Ele também foi um dos primeiros frameworks a gerar um *schema* customizado declarando a API inteira em JSON.

Ele não era baseado em um padrão como OpenAPI e JSON Schema. Então não poderia ter interação direta com outras ferramentas, como Swagger UI. Mas novamente, era uma idéia muito inovadora.

Hug tinha um incomum, interessante recurso: usando o mesmo framework, é possível criar tanto APIs como CLIs.

Como é baseado nos padrões anteriores de frameworks web síncronos (WSGI), ele não pode controlar *Websockets* e outras coisas, embora ele ainda tenha uma alta performance também.

!!! info Hug foi criado por Timothy Crosley, o mesmo criador do `isort`, uma grande ferramenta para ordenação automática de *imports* em arquivos Python.

!!! check "Idéias inspiradas para o **FastAPI**" Hug inspirou partes do APIStar, e foi uma das ferramentas que eu achei mais promissora, ao lado do APIStar.

```
Hug ajudou a inspirar o **FastAPI** a usar _type hints_ do Python para declarar parâmetros,

Hug inspirou **FastAPI** a declarar um parâmetro de `resposta` em funções para definir cabeç
```

APIStar (<= 0.5)

Antes de decidir construir **FastAPI** eu encontrei o servidor **APIStar**. Tinha quase tudo que eu estava procurando e tinha um grande projeto.

Ele foi uma das primeiras implementações de um framework usando Python *type hints* para declarar parâmetros e requisições que eu nunca vi (antes no NestJS e Molten). Eu encontrei ele mais ou menos na mesma época que o Hug. Mas o APIStar utilizava o padrão OpenAPI.

Ele tinha validação de dados automática, serialização de dados e geração de *schema* OpenAPI baseado nos mesmos *type hints* em vários locais.

Definições de *schema* de corpo não utilizavam os mesmos Python *type hints* como Pydantic, ele era um pouco mais similar ao Marshmallow, então, o suporte ao editor não seria tão bom, ainda assim, APIStar era a melhor opção disponível.

Ele obteve as melhores performances em testes na época (somente batido por Starlette).

A princípio, ele não tinha uma interface web com documentação automática da API, mas eu sabia que poderia adicionar o Swagger UI a ele.

Ele tinha um sistema de injeção de dependência. Ele exigia pré-registro dos componentes, como outras ferramentas já discutidas acima. Mas ainda era um grande recurso.

Eu nunca fui capaz de usar ele num projeto inteiro, por não ter integração de segurança, então, eu não pude substituir todos os recursos que eu tinha com os geradores *full-stack* baseados no Flask-apispec. Eu tive em minha gaveta de projetos a idéia de criar um *pull request* adicionando essa funcionalidade.

Mas então, o foco do projeto mudou.

Ele não era mais um framework web API, como o criador precisava focar no Starlette.

Agora APIStar é um conjunto de ferramentas para validar especificações OpenAPI, não um framework web.

!!! info APIStar foi criado por Tom Christie. O mesmo cara que criou:

```
* Django REST Framework
* Starlette (no qual **FastAPI** é baseado)
* Uvicorn (usado por Starlette e **FastAPI**)
```

!!! check "**FastAPI** inspirado para" Existir.

```
A idéia de declarar múltiplas coisas (validação de dados, serialização e documentação) com o

E após procurar por um logo tempo por um framework similar e testar muitas alternativas dife

Então APIStar parou de existir como um servidor e Starlette foi criado, e foi uma nova melho

Eu considero **FastAPI** um "sucessor espiritual" para o APIStar, evoluindo e melhorando os
```

Usados por FastAPI

Pydantic

Pydantic é uma biblioteca para definir validação de dados, serialização e documentação (usando JSON Schema) baseado nos Python *type hints*.

Isso faz dele extremamente intuitivo.

Ele é comparável ao Marshmallow. Embora ele seja mais rápido que Marshmallow em testes de performance. E ele é baseado nos mesmos Python *type hints*, o suporte ao editor é ótimo.

!!! check "**FastAPI** usa isso para" Controlar toda a validação de dados, serialização de dados e modelo de documentação automática (baseado no JSON Schema).

```
**FastAPI** então pega dados do JSON Schema e coloca eles no OpenAPI, à parte de todas as ou
```

Starlette

Starlette é um framework/caixa de ferramentas ASGI peso leve, o que é ideal para construir serviços assíncronos de alta performance.

Ele é muito simples e intuitivo. É projetado para ser extensível facilmente, e ter componentes modulares.

Ele tem:

- Performance seriamente impressionante.
- Suporte a WebSocket.
- Suporte a GraphQL.
- Tarefas de processamento interno por trás dos panos.
- Eventos de inicialização e encerramento.
- Cliente de testes construído com requests.
- Respostas CORS, GZip, Arquivos Estáticos, Streaming.
- Suporte para Sessão e Cookie.
- 100% coberto por testes.
- Código base 100% anotado com tipagem.
- Dependências complexas Zero.

Starlette é atualmente o mais rápido framework Python testado. Somente ultrapassado pelo Uvicorn, que não é um framework, mas um servidor.

Starlette fornece toda a funcionalidade básica de um microframework web.

Mas ele não fornece validação de dados automática, serialização e documentação.

Essa é uma das principais coisas que **FastAPI** adiciona no topo, tudo baseado em Python *type hints* (usando Pydantic). Isso, mais o sistema de injeção de dependência, utilidades de segurança, geração de *schema* OpenAPI, etc.

!!! note "Detalhes Técnicos" ASGI é um novo "padrão" sendo desenvolvido pelos membros do time central do Django. Ele ainda não está como "Padrão Python" (PEP), embora eles estejam em processo de fazer isso.

```
No entanto, ele já está sendo utilizado como "padrão" por diversas ferramentas. Isso melhora
```

!!! check "**FastAPI** usa isso para" Controlar todas as partes web centrais. Adiciona recursos no topo.

```
A classe `FastAPI` em si herda `Starlette`.

Então, qualquer coisa que você faz com Starlette, você pode fazer diretamente com **FastAPI*
```

Uvicorn

Uvicorn é um servidor ASGI peso leve, construído com uvloop e httptools.

Ele não é um framework web, mas sim um servidor. Por exemplo, ele não fornece ferramentas para roteamento por rotas. Isso é algo que um framework como Starlette (ou **FastAPI**) poderia fornecer por cima.

Ele é o servidor recomendado para Starlette e **FastAPI**.

!!! check "**FastAPI** recomenda isso para" O principal servidor web para rodar aplicações **FastAPI**.

```
Você pode combinar ele com o Gunicorn, para ter um servidor multi-processos assíncrono.

Verifique mais detalhes na seção [Deployment](deployment/index.md){.internal-link target=_bl
```

Performance e velocidade

Para entender, comparar e ver a diferença entre Uvicorn, Starlette e FastAPI, verifique a seção sobre Benchmarks.