代替ツールから受けたインスピレーションと比較

何が**FastAPI**にインスピレーションを与えたのか、他の代替ツールと比較してどうか、そしてそこから何を学んだのかについて。

はじめに

FastAPIは、代替ツールのこれまでの働きがなければ存在しなかったでしょう。

以前に作られた多くのツールが、作成における刺激として役立ってきました。

私は数年前から新しいフレームワークの作成を避けてきました。まず、**FastAPI**でカバーされているすべての機能を、さまざまなフレームワーク、プラグイン、ツールを使って解決しようとしました。

しかし、その時点では、これらの機能をすべて提供し、以前のツールから優れたアイデアを取り入れ、可能な限り最高の方法でそれらを組み合わせ、それまで利用できなかった言語機能 (Python 3.6以降の型ヒント) を利用したものを作る以外に選択肢はありませんでした。

以前のツール

Django

Pythonのフレームワークの中で最もポピュラーで、広く信頼されています。Instagramのようなシステムの構築に使われています。

リレーショナルデータベース (MySQLやPostgreSQLなど) と比較的強固に結合されているので、NoSQLデータベース (Couchbase、MongoDB、Cassandraなど) をメインに利用することは簡単ではありません。

バックエンドでHTMLを生成するために作られたものであり、現代的なフロントエンド (ReactやVue.js、Angularなど) や、他のシステム (IoTデバイスなど) と通信するAPIを構築するために作られたものではありません。

Django REST Framework

Django REST Frameworkは、Djangoを下敷きにしてWeb APIを構築する柔軟なツールキットとして、APIの機能を向上させるために作られました。

Mozilla、Red Hat、Eventbrite など多くの企業で利用されています。

これは**自動的なAPIドキュメント生成**の最初の例であり、これは**FastAPI**に向けた「調査」を触発した最初のアイデアの一つでした。

!!! note "備考" Django REST Framework は Tom Christie によって作成されました。StarletteとUvicornの生みの親であり、**FastAPI**のベースとなっています。

!!! check "**FastAPI**へ与えたインスピレーション" 自動でAPIドキュメントを生成するWebユーザーインターフェースを持っている点。

Flask

Flask は「マイクロフレームワーク」であり、データベースとの統合のようなDjangoがデフォルトで持つ多くの機能は含まれていません。

このシンプルさと柔軟性により、メインのデータストレージシステムとしてNoSQLデータベースを使用するといったようなことが可能になります。

非常にシンプルなので、ドキュメントはいくつかの点でやや技術的でありますが、比較的直感的に学ぶことができます。

また、データベースやユーザ管理、あるいはDjangoにあらかじめ組み込まれている多くの機能を必ずしも必要としないアプリケーションにもよく使われています。これらの機能の多くはプラグインで追加することができます。

このようにパーツを分離し、必要なものを正確にカバーするために拡張できる「マイクロフレームワーク」であることは、私が残しておきたかった重要な機能でした。

Flaskのシンプルさを考えると、APIを構築するのに適しているように思えました。次に見つけるべきは、Flask 用の「Django REST Framework」でした。

!!! check "**FastAPI**へ与えたインスピレーション" マイクロフレームワークであること。ツールやパーツを目的に合うように簡単に組み合わせられる点。

シンプルで簡単なルーティングの仕組みを持っている点。

Requests

FastAPIは実際には**Requests**の代替ではありません。それらのスコープは大きく異なります。

実際にはFastAPIアプリケーションの*内部*でRequestsを使用するのが一般的です。

しかし、FastAPIはRequestsからかなりのインスピレーションを得ています。

Requestsは (クライアントとして) APIと*通信*するためのライブラリであり、**FastAPI**は (サーバーとして) APIを*構築*するためのライブラリです。

これらは多かれ少なかれ両端にあり、お互いを補完し合っています。

Requestsは非常にシンプルかつ直感的なデザインで使いやすく、適切なデフォルト値を設定しています。しかし同時に、非常に強力でカスタマイズも可能です。

公式サイトで以下のように言われているのは、それが理由です。

> *Requestsは今までで最もダウンロードされたPythonパッケージである*

使い方はとても簡単です。例えば、`GET`リクエストを実行するには、このように書けば良いです:

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

対応するFastAPIのパスオペレーションはこのようになります:

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

`requests.get(...)`と`@app.get(...)`には類似点が見受けられます。

!!! check "**FastAPI**へ与えたインスピレーション" * シンプルで直感的なAPIを持っている点。 * HTTPメソッド名を直接利用し、単純で直感的である。 * 適切なデフォルト値を持ちつつ、強力なカスタマイズ性を持っている。

Swagger / OpenAPI

私がDjango REST Frameworkに求めていた主な機能は、APIの自動的なドキュメント生成でした。

そして、Swaggerと呼ばれる、 JSON (またはYAML、JSONの拡張版) を使ってAPIをドキュメント化するための標準があることを知りました。

そして、Swagger API用のWebユーザーインターフェースが既に作成されていました。つまり、API用のSwaggerドキュメントを生成することができれば、このWebユーザーインターフェースを自動的に使用することができるようになります。

ある時点で、SwaggerはLinux Foundationに寄贈され、OpenAPIに改名されました。

そのため、バージョン2.0では「Swagger」、バージョン3以上では「OpenAPI」と表記するのが一般的です。

!!! check "**FastAPI**へ与えたインスピレーション" 独自のスキーマの代わりに、API仕様のオープンな標準を採用しました。

そして、標準に基づくユーザーインターフェースツールを統合しています。

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

この二つは人気で安定したものとして選択されましたが、少し検索してみると、 (**FastAPI**と同時に使用できる) OpenAPIのための多くの代替となるツールを見つけることができます。

Flask REST フレームワーク

いくつかのFlask RESTフレームワークがありますが、それらを調査してみたところ、多くのものが不適切な問題が残ったまま、中断されたり放置されていることがわかりました。

Marshmallow

APIシステムで必要とされる主な機能の一つに、コード (Python) からデータを取り出して、ネットワークを介して送れるものに変換するデータの「シリアライゼーション」があります。例えば、データベースのデータを含むオブジェクトをJSONオブジェクトに変換したり、 `datetime` オブジェクトを文字列に変換するなどです。

APIが必要とするもう一つの大きな機能はデータのバリデーションであり、特定のパラメータが与えられた場合にデータが有効であることを確認することです。例えば、あるフィールドがランダムな文字列ではなく `int` であることなどです。これは特に受信するデータに対して便利です。

データバリデーションの仕組みがなければ、すべてのチェックを手作業でコードに実装しなければなりません。

これらの機能は、Marshmallowが提供するものです。Marshmallowは素晴らしいライブラリで、私も以前に何度も使ったことがあります。

しかし、それはPythonの型ヒントが存在する前に作られたものです。そのため、すべてのスキーマを定義するためには、Marshmallowが提供する特定のユーティリティやクラスを使用する必要があります。

!!! check "**FastAPI**へ与えたインスピレーション" コードで「スキーマ」を定義し、データの型やバリデーションを自動で提供する点。

Webargs

APIに求められる他の大きな機能として、受信したリクエストデータのパースがあります。

WebargsはFlaskをはじめとするいくつかのフレームワークの上にそれを提供するために作られたツールです。

データのバリデーションを行うために内部ではMarshmallowを使用しており、同じ開発者によって作られました。

素晴らしいツールで、私も**FastAPI**を持つ前はよく使っていました。

!!! info "情報" Webargsは、Marshmallowと同じ開発者により作られました。

!!! check "**FastAPI**へ与えたインスピレーション" 受信したデータに対する自動的なバリデーションを持っている点。

APISpec

MarshmallowとWebargsはバリデーション、パース、シリアライゼーションをプラグインとして提供しています。

しかし、ドキュメントはまだ不足しています。そこでAPISpecが作られました。

これは多くのフレームワーク用のプラグインです (Starlette用のプラグインもあります) 。

仕組みとしては、各ルーティング関数のdocstringにYAML形式でスキーマの定義を記述します。

そして、OpenAPIスキーマを生成してくれます。

Flask, Starlette, Responderなどにおいてはそのように動作します。

しかし、Pythonの文字列 (大きなYAML) の中に、小さな構文があるという問題があります。

エディタでは、この問題を解決することはできません。また、パラメータやMarshmallowスキーマを変更したときに、YAMLのdocstringを変更するのを忘れてしまうと、生成されたスキーマが古くなってしまいます。

!!! info "情報" APISpecは、Marshmallowと同じ開発者により作成されました。

!!! check "**FastAPI**へ与えたインスピレーション" OpenAPIという、APIについてのオープンな標準をサポートしている点。

Flask-apispec

Webargs、Marshmallow、APISpecを連携させたFlaskプラグインです。

WebargsとMarshmallowの情報から、APISpecを使用してOpenAPIスキーマを自動的に生成します。

これは素晴らしいツールで、非常に過小評価されています。多くの Flask プラグインよりもずっと人気があるべきです。ドキュメントがあまりにも簡潔で抽象的であるからかもしれません。

これにより、PythonのdocstringにYAML (別の構文) を書く必要がなくなりました。

Flask、Flask-apispec、Marshmallow、Webargsの組み合わせは、**FastAPI**を構築するまで私のお気に入りのバックエンドスタックでした。

これを使うことで、いくつかのFlaskフルスタックジェネレータを作成することになりました。これらは私 (といくつかの外部のチーム) が今まで使ってきたメインのスタックです。

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

そして、これらのフルスタックジェネレーターは、**FastAPI** Project Generators{.internal-link target=_blank}の元となっていました。

!!! info "情報" Flask-apispecはMarshmallowと同じ開発者により作成されました。

!!! check "**FastAPI**へ与えたインスピレーション" シリアライゼーションとバリデーションを定義したコードから、OpenAPIスキーマを自動的に生成する点。

NestJS (とAngular)

NestJSはAngularにインスパイアされたJavaScript (TypeScript) NodeJSフレームワークで、Pythonですらありません。

Flask-apispecでできることと多少似たようなことを実現しています。

Angular 2にインスピレーションを受けた、統合された依存性注入の仕組みを持っています。(私が知っている他の依存性注入の仕組みと同様に) 「injectable」を事前に登録しておく必要があるため、冗長性とコードの繰り返しが発生します。

パラメータはTypeScriptの型で記述されるので (Pythonの型ヒントに似ています) 、エディタのサポートはとても良いです。

しかし、TypeScriptのデータはJavaScriptへのコンパイル後には残されないため、バリデーション、シリアライゼーション、ドキュメント化を同時に定義するのに型に頼ることはできません。そのため、バリデーション、シリアライゼーション、スキーマの自動生成を行うためには、多くの場所でデコレータを追加する必要があり、非常に冗長になります。

入れ子になったモデルをうまく扱えません。そのため、リクエストのJSONボディが内部フィールドを持つJSONオブジェクトで、それが順番にネストされたJSONオブジェクトになっている場合、適切にドキュメント化やバリデーションをすることができません。

!!! check "**FastAPI**へ与えたインスピレーション" 素晴らしいエディターの補助を得るために、Pythonの型ヒントを利用している点。

強力な依存性注入の仕組みを持ち、コードの繰り返しを最小化する方法を見つけた点。

Sanic

`asyncio` に基づいた、Pythonのフレームワークの中でも非常に高速なものの一つです。Flaskと非常に似た作りになっています。

!!! note "技術詳細" Pythonの `asyncio` ループの代わりに、 `uvloop` が利用されています。それにより、非常に高速です。

`Uvicorn`と`Starlette`に明らかなインスピレーションを与えており、それらは現在オープンなベンチマークにおいてSanicより高速です。

!!! check "**FastAPI**へ与えたインスピレーション" 物凄い性能を出す方法を見つけた点。

`**FastAPI**`が、(サードパーティのベンチマークによりテストされた) 最も高速なフレームワークである`Starlette`に基づいている理由です。

Falcon

Falconはもう一つの高性能Pythonフレームワークで、ミニマムに設計されており、Hugのような他のフレームワークの基盤として動作します。

Pythonのウェブフレームワーク標準規格 (WSGI) を使用していますが、それは同期的であるためWebSocketなどの利用には対応していません。とはいえ、それでも非常に高い性能を持っています。

これは、「リクエスト」と「レスポンス」の2つのパラメータを受け取る関数を持つように設計されています。そして、リクエストからデータを「読み込み」、レスポンスにデータを「書き込み」ます。この設計のため、Python標準の型ヒントでリクエストのパラメータやボディを関数の引数として宣言することはできません。

そのため、データのバリデーション、シリアライゼーション、ドキュメント化は、自動的にできずコードの中で行わなければなりません。あるいは、HugのようにFalconの上にフレームワークとして実装されなければなりません。このような分断は、パラメータとして1つのリクエストオブジェクトと1つのレスポンスオブジェクトを持つというFalconのデザインにインスピレーションを受けた他のフレームワークでも起こります。

!!! check "**FastAPI**へ与えたインスピレーション" 素晴らしい性能を得るための方法を見つけた点。

`Hug (HugはFalconをベースにしています) と一緒に、**FastAPI**が`response`引数を関数に持つことにインスピレーションを与えました。`

`**FastAPI**では任意ですが、ヘッダーやCookieやステータスコードを設定するために利用されています。`

Molten

FastAPIを構築する最初の段階でMoltenを発見しました。そして、それは非常に似たようなアイデアを持っています。

- Pythonの型ヒントに基づいている
- これらの型から行われるバリデーションとドキュメント化
- 依存性注入の仕組み

Pydanticのようなデータのバリデーション、シリアライゼーション、ドキュメント化をするサードパーティライブラリを使用せず、独自のものを持っています。そのため、これらのデータ型の定義は簡単には再利用できません。

もう少し冗長な設定が必要になります。また、 (ASGIではなく) WSGIに基づいているので、UvicornやStarletteやSanicのようなツールが提供する高性能を得られるようには設計されていません。

依存性注入の仕組みは依存性の事前登録が必要で、宣言された型に基づいて依存性が解決されます。そのため、特定の型を提供する「コンポーネント」を複数宣言することはできません。

ルーティングは一つの場所で宣言され、他の場所で宣言された関数を使用します (エンドポイントを扱う関数のすぐ上に配置できるデコレータを使用するのではなく) 。これはFlask (やStarlette) よりも、Djangoに近いです。これは、比較的緊密に結合されているものをコードの中で分離しています。

!!! check "**FastAPI**へ与えたインスピレーション" モデルの属性の「デフォルト」値を使用したデータ型の追加バリデーションを定義します。これはエディタの補助を改善するもので、以前はPydanticでは利用できませんでした。

`同様の方法でのバリデーションの宣言をサポートするよう、Pydanticを部分的にアップデートするインスピーレションを与えました。(現在はこれらの機能は全てPydanticで可能となっています。)`

Hug

Hugは、Pythonの型ヒントを利用してAPIパラメータの型宣言を実装した最初のフレームワークの1つです。これは素晴らしいアイデアで、他のツールが同じことをするきっかけとなりました。

Hugは標準のPython型の代わりにカスタム型を宣言に使用していましたが、それでも大きな進歩でした。

また、JSONでAPI全体を宣言するカスタムスキーマを生成した最初のフレームワークの1つでもあります。

OpenAPIやJSON Schemaのような標準に基づいたものではありませんでした。そのため、Swagger UIのような他のツールと統合するのは簡単ではありませんでした。しかし、繰り返しになりますが、これは非常に革新的なアイデ

アでした。

同じフレームワークを使ってAPIとCLIを作成できる、面白く珍しい機能を持っています。

以前のPythonの同期型Webフレームワーク標準 (WSGI) をベースにしているため、Websocketなどは扱えませんが、それでも高性能です。

!!! info "情報" HugはTimothy Crosleyにより作成されました。彼は `isort` など、Pythonのファイル内のインポートの並び替えを自動的におこうなう素晴らしいツールの開発者です。

!!! check "**FastAPI**へ与えたインスピレーション" HugはAPIStarに部分的なインスピレーションを与えており、私が発見した中ではAPIStarと同様に最も期待の持てるツールの一つでした。

```
Hugは、**FastAPI**がPythonの型ヒントを用いてパラメータを宣言し自動的にAPIを定義するスキーマを生成することを触発しました。

Hugは、**FastAPI**がヘッダーやクッキーを設定するために関数に `response`引数を宣言することにインスピレーションを与えました。
```

APIStar (<= 0.5)

FastAPIを構築することを決める直前に、**APIStar**サーバーを見つけました。それは私が探していたものがほぼすべて含まれており、素晴らしいデザインでした。

これは、私がこれまでに見た中で (NestJSやMoltenの前に) Pythonの型ヒントを使ってパラメータやリクエストを宣言するフレームワークの最初の実装の一つでした。Hugと多かれ少なかれ同時期に見つけました。ただ、APIStarはOpenAPI標準を使っていました。

いくつかの場所で同じ型ヒントを元に、データの自動バリデーション、データのシリアライゼーション、OpenAPIスキーマの生成を行っていました。

ボディのスキーマ定義にはPydanticのようなPythonの型ヒントは使われておらず、もう少しMarshmallowに似ていたので、エディタの補助はあまり良くないかもしれませんが、それでもAPIStarは利用可能な最良の選択肢でした。

当時のベンチマークでは最高のパフォーマンスを発揮していました (Starletteのみが上回っていました) 。

最初は自動的なAPIのドキュメント化のWeb UIを持っていませんでしたが、Swagger UIを追加できることはわかっていました。

依存性注入の仕組みを持っていました。上記で説明した他のツールのようにコンポーネントの事前登録が必要でした。しかし、それでも素晴らしい機能でした。

セキュリティの統合がなかったので、Flask-apispecを元にしたフルスタックジェネレータにあるすべての機能を置き換えることはできませんでした。私はその機能を追加するプルリクエストを作成するというプロジェクトのバックログを持っていました。

しかし、その後、プロジェクトの焦点が変わりました。

制作者はStarletteに集中する必要があったため、APIウェブフレームワークではなくなりました。

今ではAPIStarはOpenAPI仕様を検証するためのツールセットであり、ウェブフレームワークではありません。

!!! info "情報" APIStarはTom Christieにより開発されました。以下の開発者でもあります:

```
* Django REST Framework
* Starlette (**FastAPI**のベースになっています)
```

* Uvicorn (Starletteや**FastAPI**で利用されています)

!!! check "**FastAPI**へ与えたインスピレーション" 存在そのもの。

複数の機能 (データのバリデーション、シリアライゼーション、ドキュメント化) を同じPython型で宣言し、同時に優れたエディタの補助を提供するというアイデアは、私にとって素晴らしいアイデアでした。

そして、長い間同じようなフレームワークを探し、多くの異なる代替ツールをテストした結果、APIStarが最良の選択肢となりました。

その後、APIStarはサーバーとして存在しなくなり、Starletteが作られ、そのようなシステムのための新しくより良い基盤となりました。これが**FastAPI**を構築するための最終的なインスピレーションでした。

私は、これまでのツールから学んだことをもとに、機能や型システムなどの部分を改善・拡充しながら、**FastAPI**をAPIStarの「精神的な後継者」と考えています。

FastAPIが利用しているもの

Pydantic

Pydanticは、Pythonの型ヒントを元にデータのバリデーション、シリアライゼーション、 (JSON Schemaを使用した) ドキュメントを定義するライブラリです。

そのため、非常に直感的です。

Marshmallowに匹敵しますが、ベンチマークではMarshmallowよりも高速です。また、Pythonの型ヒントを元にしているので、エディタの補助が素晴らしいです。

!!! check "**FastAPI**での使用用途" データのバリデーション、データのシリアライゼーション、自動的なモデルの (JSON Schemaに基づいた) ドキュメント化の全てを扱えます。

FastAPIはJSON SchemaのデータをOpenAPIに利用します。

Starlette

Starletteは、軽量なASGIフレームワーク/ツールキットで、高性能な非同期サービスの構築に最適です。

非常にシンプルで直感的です。簡単に拡張できるように設計されており、モジュール化されたコンポーネントを持っています。

以下のような特徴があります。

- 非常に感動的な性能。
- WebSocketのサポート。
- GraphQLのサポート。
- インプロセスのバックグラウンドタスク。
- 起動およびシャットダウンイベント。
- requestsに基づいて構築されたテストクライアント。
- CORS、GZip、静的ファイル、ストリーミング応答。
- セッションとクッキーのサポート。
- 100%のテストカバレッジ。
- 100%の型注釈付きコードベース。
- ハードな依存関係はない。

Starletteは、現在テストされているPythonフレームワークの中で最も速いフレームワークです。フレームワークではなくサーバーであるUvicornだけが上回っています。

Starletteは基本的なWebマイクロフレームワークの機能をすべて提供します。

しかし、自動的なデータバリデーション、シリアライゼーション、ドキュメント化は提供していません。

これは **FastAPI** が追加する主な機能の一つで、すべての機能は Pythonの型ヒントに基づいています (Pydanticを使用しています) 。これに加えて、依存性注入の仕組み、セキュリティユーティリティ、OpenAPIスキーマ生成などがあります。

!!! note "技術詳細" ASGIはDjangoのコアチームメンバーにより開発された新しい「標準」です。まだ「Pythonの標準 (PEP) 」ではありませんが、現在そうなるように進めています。

しかしながら、いくつかのツールにおいてすでに「標準」として利用されています。このことは互換性を大きく改善するもので、Uvicornから他のASGIサーバー (DaphneやHypercorn) に乗り換えることができたり、あなたが`python-socketio`のようなASGI互換のツールを追加することもできます。

!!! check "**FastAPI**での使用用途" webに関するコアな部分を全て扱います。その上に機能を追加します。

`FastAPI`クラスそのものは、`Starlette`クラスを直接継承しています。

基本的には`Starlette`の強化版であるため、`Starlette`で可能なことは`**FastAPI**`で直接可能です。

Uvicorn

Uvicornは非常に高速なASGIサーバーで、uvloopとhttptoolsにより構成されています。

ウェブフレームワークではなくサーバーです。例えば、パスルーティングのツールは提供していません。それらは、Starlette (や**FastAPI**) のようなフレームワークがその上で提供するものです。

Starletteや**FastAPI**のサーバーとして推奨されています。

!!! check "**FastAPI**が推奨する理由" **FastAPI**アプリケーションを実行するメインのウェブサーバーである点。

`Gunicorn`と組み合わせることで、非同期でマルチプロセスなサーバーを持つことがきます。

詳細は`[デプロイ](deployment/index.md){.internal-link target=_blank}`の項目で確認してください。

ベンチマーク と スピード

Uvicorn、Starlette、FastAPIの違いを理解、比較、確認するには、ベンチマーク{.internal-link target=_blank}を確認してください。