Alternatives, inspiration et comparaisons

Ce qui a inspiré **FastAPI**, comment il se compare à d'autres solutions et ce qu'il en a appris.

Intro

FastAPI n'existerait pas sans les précédentes contributions d'autres projets.

De nombreux outils ont été créés auparavant et ont contribué à inspirer sa création.

J'ai évité la création d'un nouveau framework pendant plusieurs années. J'ai d'abord essayé de combler toutes les fonctionnalités couvertes par **FastAPI** en utilisant de nombreux frameworks, plug-ins et outils différents.

Mais à un moment donné il n'y avait pas d'autre option que de créer quelque chose qui offrait toutes ces fonctionnalités, en reprenant et en combinant de la meilleure façon possible les meilleures idées des outils précédents, en utilisant des fonctionnalités du langage qui n'étaient même pas disponibles auparavant (type hints depuis Python 3.6+).

Outils précédents

Django

C'est le framework Python le plus populaire et il bénéficie d'une grande confiance. Il est utilisé pour construire des systèmes tel qu'Instagram.

Il est relativement fortement couplé aux bases de données relationnelles (comme MySQL ou PostgreSQL), de sorte qu'il n'est pas très facile d'utiliser une base de données NoSQL (comme Couchbase, MongoDB, Cassandra, etc.) comme principal moyen de stockage.

Il a été créé pour générer le HTML en backend, pas pour créer des API consommées par un frontend moderne (comme React, Vue.js et Angular) ou par d'autres systèmes (comme les appareils IoT) communiquant avec lui.

Django REST Framework

Django REST framework a été conçu comme une boîte à outils flexible permettant de construire des API Web à partir de Django, afin d'améliorer ses capacités en matière d'API.

Il est utilisé par de nombreuses entreprises, dont Mozilla, Red Hat et Eventbrite.

Il s'agissait de l'un des premiers exemples de **documentation automatique pour API**, et c'est précisément l'une des premières idées qui a inspiré "la recherche de" **FastAPI**.

!!! note Django REST framework a été créé par Tom Christie. Le créateur de Starlette et Uvicorn, sur lesquels **FastAPI** est basé.

!!! check "A inspiré **FastAPI** à" Avoir une interface de documentation automatique de l'API.

Flask

Flask est un "micro-framework", il ne comprend pas d'intégrations de bases de données ni beaucoup de choses qui sont fournies par défaut dans Django.

Cette simplicité et cette flexibilité permettent d'utiliser des bases de données NoSQL comme principal système de stockage de données.

Comme il est très simple, son apprentissage est relativement intuitif, bien que la documentation soit quelque peu technique par moments.

Il est aussi couramment utilisé pour d'autres applications qui n'ont pas nécessairement besoin d'une base de données, de gestion des utilisateurs ou de l'une des nombreuses fonctionnalités préinstallées dans Django. Bien que beaucoup de ces fonctionnalités puissent être ajoutées avec des plug-ins.

Ce découplage des parties, et le fait d'être un "micro-framework" qui puisse être étendu pour couvrir exactement ce qui est nécessaire, était une caractéristique clé que je voulais conserver.

Compte tenu de la simplicité de Flask, il semblait bien adapté à la création d'API. La prochaine chose à trouver était un "Django REST Framework" pour Flask.

!!! check "A inspiré **FastAPI** à" Être un micro-framework. Il est donc facile de combiner les outils et les pièces nécessaires.

Proposer un système de routage simple et facile à utiliser.

Requests

FastAPI n'est pas réellement une alternative à **Requests**. Leur cadre est très différent.

Il serait en fait plus courant d'utiliser Requests *à l'intérieur* d'une application FastAPI.

Mais quand même, FastAPI s'est inspiré de Requests.

Requests est une bibliothèque pour *interagir* avec les API (en tant que client), tandis que **FastAPI** est une bibliothèque pour *créer* des API (en tant que serveur).

Ils sont, plus ou moins, aux extrémités opposées, se complétant l'un l'autre.

Requests a un design très simple et intuitif, il est très facile à utiliser, avec des valeurs par défaut raisonnables, tout en étant très puissant et personnalisable.

C'est pourquoi, comme le dit le site officiel :

> Requests est l'un des packages Python les plus téléchargés de tous les temps

La façon dont vous l'utilisez est très simple. Par exemple, pour faire une requête `GET`, vous devez écrire :

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

En contrepartie l'API *des opérations de chemin* de FastAPI pourrait ressembler à ceci :

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

Notez les similitudes entre `requests.get(...)` et `@app.get(...)`.

!!! check "A inspiré **FastAPI** à" _ Avoir une API simple et intuitive. _ Utiliser les noms de méthodes HTTP (opérations) directement, de manière simple et intuitive. * Avoir des valeurs par défaut raisonnables, mais des personnalisations puissantes.

Swagger / OpenAPI

La principale fonctionnalité que j'ai emprunté à Django REST Framework était la documentation automatique des API.

Puis j'ai découvert qu'il existait une norme pour documenter les API, en utilisant JSON (ou YAML, une extension de JSON) appelée Swagger.

Il existait déjà une interface utilisateur Web pour les API Swagger. Donc, être capable de générer une documentation Swagger pour une API permettrait d'utiliser cette interface utilisateur web automatiquement.

À un moment donné, Swagger a été cédé à la Fondation Linux, puis a été rebaptisé OpenAPI.

C'est pourquoi, lorsqu'on parle de la version 2.0, il est courant de dire "Swagger", et pour la version 3+ "OpenAPI".

!!! check "A inspiré **FastAPI** à" Adopter et utiliser une norme ouverte pour les spécifications des API, au lieu d'un schéma personnalisé.

```
Intégrer des outils d'interface utilisateur basés sur des normes :

* <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>

Ces deux-là ont été choisis parce qu'ils sont populaires et stables, mais en faisant une rec
```

Frameworks REST pour Flask

Il y a plusieurs frameworks REST pour Flask, mais après avoir investi du temps et du travail pour les étudier, j'ai découvert que le développement de beaucoup d'entre eux sont suspendus ou abandonnés, avec plusieurs problèmes permanents qui les rendent inadaptés.

Marshmallow

L'une des principales fonctionnalités nécessaires aux systèmes API est la "sérialisation" des données, qui consiste à prendre les données du code (Python) et à les convertir en quelque chose qui peut être envoyé sur le réseau. Par exemple, convertir un objet contenant des données provenant d'une base de données en un objet JSON. Convertir des objets `datetime` en strings, etc.

La validation des données est une autre fonctionnalité importante dont ont besoin les API. Elle permet de s'assurer que les données sont valides, compte tenu de certains paramètres. Par exemple, qu'un champ est un `int`, et non un string. Ceci est particulièrement utile pour les données entrantes.

Sans un système de validation des données, vous devriez effectuer toutes les vérifications à la main, dans le code.

Ces fonctionnalités sont ce pourquoi Marshmallow a été construit. C'est une excellente bibliothèque, et je l'ai déjà beaucoup utilisée.

Mais elle a été créée avant que les type hints n'existent en Python. Ainsi, pour définir chaque schéma, vous devez utiliser des utilitaires et des classes spécifiques fournies par Marshmallow.

!!! check "A inspiré **FastAPI** à" Utilisez du code pour définir des "schémas" qui fournissent automatiquement les types de données et la validation.

Webargs

Une autre grande fonctionnalité requise par les API est le parsing des données provenant des requêtes entrantes.

Webargs est un outil qui a été créé pour fournir cela par-dessus plusieurs frameworks, dont Flask.

Il utilise Marshmallow pour effectuer la validation des données. Et il a été créé par les mêmes développeurs.

C'est un outil formidable et je l'ai beaucoup utilisé aussi, avant d'avoir **FastAPI**.

!!! info Webargs a été créé par les développeurs de Marshmallow.

!!! check "A inspiré **FastAPI** à" Disposer d'une validation automatique des données des requêtes entrantes.

APISpec

Marshmallow et Webargs fournissent la validation, l'analyse et la sérialisation en tant que plug-ins.

Mais la documentation fait toujours défaut. C'est alors qu'APISpec a été créé.

Il s'agit d'un plug-in pour de nombreux frameworks (et il existe également un plug-in pour Starlette).

Le principe est le suivant : vous écrivez la définition du schéma au format YAML dans la docstring de chaque fonction gérant une route.

Et il génère des schémas OpenAPI.

C'est ainsi que cela fonctionne dans Flask, Starlette, Responder, etc.

Mais alors, nous avons à nouveau le problème d'avoir une micro-syntaxe, dans une docstring Python (un gros morceau de YAML).

L'éditeur ne peut guère aider en la matière. Et si nous modifions les paramètres ou les schémas Marshmallow et que nous oublions de modifier également cette docstring YAML, le schéma généré deviendrait obsolète.

!!! info APISpec a été créé par les développeurs de Marshmallow.

!!! check "A inspiré **FastAPI** à" Supporter la norme ouverte pour les API, OpenAPI.

Flask-apispec

C'est un plug-in pour Flask, qui relie Webargs, Marshmallow et APISpec.

Il utilise les informations de Webargs et Marshmallow pour générer automatiquement des schémas OpenAPI, en utilisant APISpec.

C'est un excellent outil, très sous-estimé. Il devrait être beaucoup plus populaire que de nombreux plug-ins Flask. C'est peut-être dû au fait que sa documentation est trop concise et abstraite.

Cela a permis de ne pas avoir à écrire YAML (une autre syntaxe) à l'intérieur des docstrings Python.

Cette combinaison de Flask, Flask-apispec avec Marshmallow et Webargs était ma stack backend préférée jusqu'à la création de **FastAPI**.

Son utilisation a conduit à la création de plusieurs générateurs Flask full-stack. Ce sont les principales stacks que j'ai (ainsi que plusieurs équipes externes) utilisées jusqu'à présent :

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

Ces mêmes générateurs full-stack ont servi de base aux Générateurs de projets pour **FastAPI**.

!!! info Flask-apispec a été créé par les développeurs de Marshmallow.

!!! check "A inspiré **FastAPI** à" Générer le schéma OpenAPI automatiquement, à partir du même code qui définit la sérialisation et la validation.

NestJS (et Angular)

Ce n'est même pas du Python, NestJS est un framework JavaScript (TypeScript) NodeJS inspiré d'Angular.

Il réalise quelque chose de similaire à ce qui peut être fait avec Flask-apispec.

Il possède un système d'injection de dépendances intégré, inspiré d'Angular 2. Il nécessite de pré-enregistrer les "injectables" (comme tous les autres systèmes d'injection de dépendances que je connais), donc, cela ajoute à la verbosité et à la répétition du code.

Comme les paramètres sont décrits avec des types TypeScript (similaires aux type hints de Python), la prise en charge par l'éditeur est assez bonne.

Mais comme les données TypeScript ne sont pas préservées après la compilation en JavaScript, il ne peut pas compter sur les types pour définir la validation, la sérialisation et la documentation en même temps. En raison de cela et de certaines décisions de conception, pour obtenir la validation, la sérialisation et la génération automatique de schémas, il est nécessaire d'ajouter des décorateurs à de nombreux endroits. Cela devient donc assez verbeux.

Il ne peut pas très bien gérer les modèles imbriqués. Ainsi, si le corps JSON de la requête est un objet JSON comportant des champs internes qui sont à leur tour des objets JSON imbriqués, il ne peut pas être correctement documenté et validé.

!!! check "A inspiré **FastAPI** à" Utiliser les types Python pour bénéficier d'un excellent support de l'éditeur.

`Disposer d'un puissant système d'injection de dépendances. Trouver un moyen de minimiser la`

Sanic

C'était l'un des premiers frameworks Python extrêmement rapides basés sur `asyncio`. Il a été conçu pour être très similaire à Flask.

!!! note "Détails techniques" Il utilisait `uvloop` au lieu du système par défaut de Python `asyncio`. C'est ce qui l'a rendu si rapide.

`Il a clairement inspiré Uvicorn et Starlette, qui sont actuellement plus rapides que Sanic d`

!!! check "A inspiré **FastAPI** à" Trouvez un moyen d'avoir une performance folle.

```
C'est pourquoi **FastAPI** est basé sur Starlette, car il s'agit du framework le plus rapide
```

Falcon

Falcon est un autre framework Python haute performance, il est conçu pour être minimal, et est utilisé comme fondation pour d'autres frameworks comme Hug.

Il utilise le standard précédent pour les frameworks web Python (WSGI) qui est synchrone, donc il ne peut pas gérer les WebSockets et d'autres cas d'utilisation. Néanmoins, il offre de très bonnes performances.

Il est conçu pour avoir des fonctions qui reçoivent deux paramètres, une "requête" et une "réponse". Ensuite, vous "lisez" des parties de la requête et "écrivez" des parties dans la réponse. En raison de cette conception, il n'est pas possible de déclarer des paramètres de requête et des corps avec des indications de type Python standard comme paramètres de fonction.

Ainsi, la validation, la sérialisation et la documentation des données doivent être effectuées dans le code, et non pas automatiquement. Ou bien elles doivent être implémentées comme un framework au-dessus de Falcon, comme Hug. Cette même distinction se retrouve dans d'autres frameworks qui s'inspirent de la conception de Falcon, qui consiste à avoir un objet de requête et un objet de réponse comme paramètres.

!!! check "A inspiré **FastAPI** à" Trouver des moyens d'obtenir de bonnes performances.

```
Avec Hug (puisque Hug est basé sur Falcon), **FastAPI** a inspiré la déclaration d'un paramè

Bien que dans FastAPI, il est facultatif, et est utilisé principalement pour définir les en-
```

Molten

J'ai découvert Molten lors des premières étapes de développement de **FastAPI**. Et il a des idées assez similaires :

- Basé sur les type hints Python.
- Validation et documentation via ces types.
- Système d'injection de dépendances.

Il n'utilise pas une librairie tiers de validation, sérialisation et de documentation tel que Pydantic, il utilise son propre système. Ainsi, ces définitions de types de données ne sont pas réutilisables aussi facilement.

Il nécessite une configuration un peu plus verbeuse. Et comme il est basé sur WSGI (au lieu dASGI), il n'est pas conçu pour profiter des hautes performances fournies par des outils comme Uvicorn, Starlette et Sanic.

Le système d'injection de dépendances exige le pré-enregistrement des dépendances et les dépendances sont résolues sur la base des types déclarés. Ainsi, il n'est pas possible de déclarer plus d'un "composant" qui fournit un certain type.

Les routes sont déclarées à un seul endroit, en utilisant des fonctions déclarées à d'autres endroits (au lieu d'utiliser des décorateurs qui peuvent être placés juste au-dessus de la fonction qui gère l'endpoint). Cette méthode est plus proche de celle de Django que de celle de Flask (et Starlette). Il sépare dans le code des choses qui sont relativement fortement couplées.

!!! check "A inspiré **FastAPI** à" Définir des validations supplémentaires pour les types de données utilisant la valeur "par défaut" des attributs du modèle. Ceci améliore le support de l'éditeur, et n'était pas disponible dans Pydantic auparavant.

```
Cela a en fait inspiré la mise à jour de certaines parties de Pydantic, afin de supporter le
```

Hug

Hug a été l'un des premiers frameworks à implémenter la déclaration des types de paramètres d'API en utilisant les type hints Python. C'était une excellente idée qui a inspiré d'autres outils à faire de même.

Il utilisait des types personnalisés dans ses déclarations au lieu des types Python standard, mais c'était tout de même un énorme pas en avant.

Il a également été l'un des premiers frameworks à générer un schéma personnalisé déclarant l'ensemble de l'API en JSON.

Il n'était pas basé sur une norme comme OpenAPI et JSON Schema. Il ne serait donc pas simple de l'intégrer à d'autres outils, comme Swagger UI. Mais encore une fois, c'était une idée très innovante.

Il présente une caractéristique intéressante et peu commune : à l'aide du même framework, il est possible de créer des API et des CLI.

Comme il est basé sur l'ancienne norme pour les frameworks web Python synchrones (WSGI), il ne peut pas gérer les Websockets et autres, bien qu'il soit également très performant.

!!! info Hug a été créé par Timothy Crosley, le créateur de `isort`, un excellent outil pour trier automatiquement les imports dans les fichiers Python.

!!! check "A inspiré **FastAPI** à" Hug a inspiré certaines parties d'APIStar, et était l'un des outils que je trouvais les plus prometteurs, à côté d'APIStar.

```
Hug a contribué à inspirer **FastAPI** pour utiliser les type hints Python
pour déclarer les paramètres, et pour générer automatiquement un schéma définissant l'API.

Hug a inspiré **FastAPI** pour déclarer un paramètre `response` dans les fonctions pour défi
```

APIStar (<= 0.5)

Juste avant de décider de développer **FastAPI**, j'ai trouvé le serveur **APIStar**.
Il contenait presque tout ce que je recherchais et avait un beau design.

C'était l'une des premières implémentations d'un framework utilisant les type hints Python pour déclarer les paramètres et les requêtes que j'ai vues (avant NestJS et Molten). Je l'ai trouvé plus ou moins en même temps que Hug. Mais APIStar utilisait le standard OpenAPI.

Il disposait de la validation automatique, sérialisation des données et d'une génération de schéma OpenAPI basée sur les mêmes type hints à plusieurs endroits.

La définition du schéma de corps de requête n'utilisait pas les mêmes type hints Python que Pydantic, il était un peu plus proche de Marshmallow, donc le support de l'éditeur n'était pas aussi bon, mais APIStar était quand même la meilleure option disponible.

Il avait les meilleures performances d'après les benchmarks de l'époque (seulement surpassé par Starlette).

Au départ, il ne disposait pas d'une interface web de documentation automatique de l'API, mais je savais que je pouvais lui ajouter une interface Swagger.

Il avait un système d'injection de dépendances. Il nécessitait un pré-enregistrement des composants, comme d'autres outils discutés ci-dessus. Mais c'était quand même une excellente fonctionnalité.

Je n'ai jamais pu l'utiliser dans un projet complet, car il n'avait pas d'intégration de sécurité, et je ne pouvais donc pas remplacer toutes les fonctionnalités que j'avais avec les générateurs complets basés sur Flask-apispec. J'avais dans mon backlog de projets de créer une pull request pour ajouter cette fonctionnalité.

Mais ensuite, le projet a changé d'orientation.

Il ne s'agissait plus d'un framework web API, le créateur devant se concentrer sur Starlette.

Maintenant, APIStar est un ensemble d'outils pour valider les spécifications OpenAPI, et non un framework web.

!!! info APIStar a été créé par Tom Christie. Le même gars qui a créé :

```
* Django REST Framework
* Starlette (sur lequel **FastAPI** est basé)
* Uvicorn (utilisé par Starlette et **FastAPI**)
```

!!! check “A inspiré **FastAPI** à” Exister.

```
L'idée de déclarer plusieurs choses (validation des données, sérialisation et documentation)

Et après avoir longtemps cherché un framework similaire et testé de nombreuses alternatives,

Puis APIStar a cessé d'exister en tant que serveur et Starlette a été créé, et a constitué u

Je considère **FastAPI** comme un "successeur spirituel" d'APIStar, tout en améliorant et en
```

Utilisés par FastAPI

Pydantic

Pydantic est une bibliothèque permettant de définir la validation, la sérialisation et la documentation des données (à l'aide de JSON Schema) en se basant sur les Python type hints.

Cela le rend extrêmement intuitif.

Il est comparable à Marshmallow. Bien qu'il soit plus rapide que Marshmallow dans les benchmarks. Et comme il est basé sur les mêmes type hints Python, le support de l'éditeur est grand.

!!! check "**FastAPI** l'utilise pour" Gérer toute la validation des données, leur sérialisation et la documentation automatique du modèle (basée sur le schéma JSON).

```
**FastAPI** prend ensuite ces données JSON Schema et les place dans OpenAPI, en plus de tout
```

Starlette

Starlette est un framework/toolkit léger ASGI, qui est idéal pour construire des services asyncio performants.

Il est très simple et intuitif. Il est conçu pour être facilement extensible et avoir des composants modulaires.

Il offre :

- Des performances vraiment impressionnantes.
- Le support des WebSockets.
- Le support de GraphQL.
- Les tâches d'arrière-plan.
- Les événements de démarrage et d'arrêt.
- Un client de test basé sur request.
- CORS, GZip, fichiers statiques, streaming des réponses.
- Le support des sessions et des cookies.
- Une couverture de test à 100 %.
- 100 % de la base de code avec des annotations de type.
- Zéro forte dépendance à d'autres packages.

Starlette est actuellement le framework Python le plus rapide testé. Seulement dépassé par Uvicorn, qui n'est pas un framework, mais un serveur.

Starlette fournit toutes les fonctionnalités de base d'un micro-framework web.

Mais il ne fournit pas de validation automatique des données, de sérialisation ou de documentation.

C'est l'une des principales choses que **FastAPI** ajoute par-dessus, le tout basé sur les type hints Python (en utilisant Pydantic). Cela, plus le système d'injection

de dépendances, les utilitaires de sécurité, la génération de schémas OpenAPI, etc.

!!! note “Détails techniques” ASGI est une nouvelle “norme” développée par les membres de l’équipe principale de Django. Il ne s’agit pas encore d’une “norme Python” (un PEP), bien qu’ils soient en train de le faire.

```
Néanmoins, il est déjà utilisé comme "standard" par plusieurs outils. Cela améliore grandeme
```

!!! check “**FastAPI** l’utilise pour” Gérer toutes les parties web de base. Ajouter des fonctionnalités par-dessus.

```
La classe `FastAPI` elle-même hérite directement de la classe `Starlette`.

Ainsi, tout ce que vous pouvez faire avec Starlette, vous pouvez le faire directement avec *
```

Uvicorn

Uvicorn est un serveur ASGI rapide comme l’éclair, basé sur uvloop et httptools.

Il ne s’agit pas d’un framework web, mais d’un serveur. Par exemple, il ne fournit pas d’outils pour le routing. C’est quelque chose qu’un framework comme Starlette (ou **FastAPI**) fournirait par-dessus.

C’est le serveur recommandé pour Starlette et **FastAPI**.

!!! check “**FastAPI** le recommande comme” Le serveur web principal pour exécuter les applications **FastAPI**.

```
Vous pouvez le combiner avec Gunicorn, pour avoir un serveur multi-processus asynchrone.

Pour plus de détails, consultez la section [Déploiement](deployment/index.md){.internal-link
```

Benchmarks et vitesse

Pour comprendre, comparer et voir la différence entre Uvicorn, Starlette et FastAPI, consultez la section sur les Benchmarks.