Alternatives, Inspiration and Comparisons

What inspired **FastAPI**, how it compares to other alternatives and what it learned from them.

Intro

FastAPI wouldn't exist if not for the previous work of others.

There have been many tools created before that have helped inspire its creation.

I have been avoiding the creation of a new framework for several years. First I tried to solve all the features covered by **FastAPI** using many different frameworks, plug-ins, and tools.

But at some point, there was no other option than creating something that provided all these features, taking the best ideas from previous tools, and combining them in the best way possible, using language features that weren't even available before (Python 3.6+ type hints).

Previous tools

Django

It's the most popular Python framework and is widely trusted. It is used to build systems like Instagram.

It's relatively tightly coupled with relational databases (like MySQL or PostgreSQL), so, having a NoSQL database (like Couchbase, MongoDB, Cassandra, etc) as the main store engine is not very easy.

It was created to generate the HTML in the backend, not to create APIs used by a modern frontend (like React, Vue.js and Angular) or by other systems (like IoT devices) communicating with it.

Django REST Framework

Django REST framework was created to be a flexible toolkit for building Web APIs using Django underneath, to improve its API capabilities.

It is used by many companies including Mozilla, Red Hat and Eventbrite.

It was one of the first examples of **automatic API documentation**, and this was specifically one of the first ideas that inspired "the search for" **FastAPI**.

!!! note Django REST Framework was created by Tom Christie. The same creator of Starlette and Uvicorn, on which **FastAPI** is based.

!!! check "Inspired **FastAPI** to" Have an automatic API documentation web user interface.

Flask

Flask is a "microframework", it doesn't include database integrations nor many of the things that come by default in Django.

This simplicity and flexibility allow doing things like using NoSQL databases as the main data storage system.

As it is very simple, it's relatively intuitive to learn, although the documentation gets somewhat technical at some points.

It is also commonly used for other applications that don't necessarily need a database, user management, or any of the many features that come pre-built in Django. Although many of these features can be added with plug-ins.

This decoupling of parts, and being a "microframework" that could be extended to cover exactly what is needed was a key feature that I wanted to keep.

Given the simplicity of Flask, it seemed like a good match for building APIs. The next thing to find was a "Django REST Framework" for Flask.

!!! check "Inspired **FastAPI** to" Be a micro-framework. Making it easy to mix and match the tools and parts needed.

`Have a simple and easy to use routing system.`

Requests

FastAPI is not actually an alternative to **Requests**. Their scope is very different.

It would actually be common to use Requests *inside* of a FastAPI application.

But still, FastAPI got quite some inspiration from Requests.

Requests is a library to *interact* with APIs (as a client), while **FastAPI** is a library to *build* APIs (as a server).

They are, more or less, at opposite ends, complementing each other.

Requests has a very simple and intuitive design, it's very easy to use, with sensible defaults. But at the same time, it's very powerful and customizable.

That's why, as said in the official website:

> Requests is one of the most downloaded Python packages of all time

The way you use it is very simple. For example, to do a `GET` request, you would write:

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

The FastAPI counterpart API *path operation* could look like:

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

See the similarities in `requests.get(...)` and `@app.get(...)`.

!!! check “Inspired **FastAPI** to” * Have a simple and intuitive API. * Use HTTP method names (operations) directly, in a straightforward and intuitive way. * Have sensible defaults, but powerful customizations.

Swagger / OpenAPI

The main feature I wanted from Django REST Framework was the automatic API documentation.

Then I found that there was a standard to document APIs, using JSON (or YAML, an extension of JSON) called Swagger.

And there was a web user interface for Swagger APIs already created. So, being able to generate Swagger documentation for an API would allow using this web user interface automatically.

At some point, Swagger was given to the Linux Foundation, to be renamed OpenAPI.

That’s why when talking about version 2.0 it’s common to say “Swagger”, and for version 3+ “OpenAPI”.

!!! check “Inspired **FastAPI** to” Adopt and use an open standard for API specifications, instead of a custom schema.

```
And integrate standards-based user interface tools:

* <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>

These two were chosen for being fairly popular and stable, but doing a quick search, you cou
```

Flask REST frameworks

There are several Flask REST frameworks, but after investing the time and work into investigating them, I found that many are discontinued or abandoned, with several standing issues that made them unfit.

Marshmallow

One of the main features needed by API systems is data “serialization” which is taking data from the code (Python) and converting it into something that can be sent through the network. For example, converting an object containing data from a database into a JSON object. Converting `datetime` objects into strings, etc.

Another big feature needed by APIs is data validation, making sure that the data is valid, given certain parameters. For example, that some field is an `int`, and not some random string. This is especially useful for incoming data.

Without a data validation system, you would have to do all the checks by hand, in code.

These features are what Marshmallow was built to provide. It is a great library, and I have used it a lot before.

But it was created before there existed Python type hints. So, to define every schema you need to use specific utils and classes provided by Marshmallow.

!!! check "Inspired **FastAPI** to" Use code to define "schemas" that provide data types and validation, automatically.

Webargs

Another big feature required by APIs is parsing data from incoming requests.

Webargs is a tool that was made to provide that on top of several frameworks, including Flask.

It uses Marshmallow underneath to do the data validation. And it was created by the same developers.

It's a great tool and I have used it a lot too, before having **FastAPI**.

!!! info Webargs was created by the same Marshmallow developers.

!!! check "Inspired **FastAPI** to" Have automatic validation of incoming request data.

APISpec

Marshmallow and Webargs provide validation, parsing and serialization as plug-ins.

But documentation is still missing. Then APISpec was created.

It is a plug-in for many frameworks (and there's a plug-in for Starlette too).

The way it works is that you write the definition of the schema using YAML format inside the docstring of each function handling a route.

And it generates OpenAPI schemas.

That's how it works in Flask, Starlette, Responder, etc.

But then, we have again the problem of having a micro-syntax, inside of a Python string (a big YAML).

The editor can't help much with that. And if we modify parameters or Marshmallow schemas and forget to also modify that YAML docstring, the generated schema would be obsolete.

!!! info APISpec was created by the same Marshmallow developers.

!!! check "Inspired **FastAPI** to" Support the open standard for APIs, OpenAPI.

Flask-apispec

It's a Flask plug-in, that ties together Webargs, Marshmallow and APISpec.

It uses the information from Webargs and Marshmallow to automatically generate OpenAPI schemas, using APISpec.

It's a great tool, very under-rated. It should be way more popular than many Flask plug-ins out there. It might be due to its documentation being too concise and abstract.

This solved having to write YAML (another syntax) inside of Python docstrings.

This combination of Flask, Flask-apispec with Marshmallow and Webargs was my favorite backend stack until building **FastAPI**.

Using it led to the creation of several Flask full-stack generators. These are the main stack I (and several external teams) have been using up to now:

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

And these same full-stack generators were the base of the **FastAPI** Project Generators.

!!! info Flask-apispec was created by the same Marshmallow developers.

!!! check "Inspired **FastAPI** to" Generate the OpenAPI schema automatically, from the same code that defines serialization and validation.

NestJS (and Angular)

This isn't even Python, NestJS is a JavaScript (TypeScript) NodeJS framework inspired by Angular.

It achieves something somewhat similar to what can be done with Flask-apispec.

It has an integrated dependency injection system, inspired by Angular two. It requires pre-registering the "injectables" (like all the other dependency injection systems I know), so, it adds to the verbosity and code repetition.

As the parameters are described with TypeScript types (similar to Python type hints), editor support is quite good.

But as TypeScript data is not preserved after compilation to JavaScript, it cannot rely on the types to define validation, serialization and documentation at the same time. Due to this and some design decisions, to get validation, serialization and automatic schema generation, it's needed to add decorators in many places. So, it becomes quite verbose.

It can't handle nested models very well. So, if the JSON body in the request is a JSON object that has inner fields that in turn are nested JSON objects, it cannot be properly documented and validated.

!!! check "Inspired **FastAPI** to" Use Python types to have great editor support.

```
Have a powerful dependency injection system. Find a way to minimize code repetition.
```

Sanic

It was one of the first extremely fast Python frameworks based on `asyncio`. It was made to be very similar to Flask.

!!! note "Technical Details" It used `uvloop` instead of the default Python `asyncio` loop. That's what made it so fast.

```
It clearly inspired Uvicorn and Starlette, that are currently faster than Sanic in open benc
```

!!! check "Inspired **FastAPI** to" Find a way to have a crazy performance.

```
That's why **FastAPI** is based on Starlette, as it is the fastest framework available (test
```

Falcon

Falcon is another high performance Python framework, it is designed to be minimal, and work as the foundation of other frameworks like Hug.

It is designed to have functions that receive two parameters, one "request" and one "response". Then you "read" parts from the request, and "write" parts to the response. Because of this design, it is not possible to declare request parameters and bodies with standard Python type hints as function parameters.

So, data validation, serialization, and documentation, have to be done in code, not automatically. Or they have to be implemented as a framework on top of Falcon, like Hug. This same distinction happens in other frameworks that are inspired by Falcon's design, of having one request object and one response object as parameters.

!!! check "Inspired **FastAPI** to" Find ways to get great performance.

```
Along with Hug (as Hug is based on Falcon) inspired **FastAPI** to declare a `response` para

Although in FastAPI it's optional, and is used mainly to set headers, cookies, and alternati
```

Molten

I discovered Molten in the first stages of building **FastAPI**. And it has quite similar ideas:

- Based on Python type hints.
- Validation and documentation from these types.
- Dependency Injection system.

It doesn't use a data validation, serialization and documentation third-party library like Pydantic, it has its own. So, these data type definitions would not be reusable as easily.

It requires a little bit more verbose configurations. And as it is based on WSGI (instead of ASGI), it is not designed to take advantage of the high-performance provided by tools like Uvicorn, Starlette and Sanic.

The dependency injection system requires pre-registration of the dependencies and the dependencies are solved based on the declared types. So, it's not possible to declare more than one "component" that provides a certain type.

Routes are declared in a single place, using functions declared in other places (instead of using decorators that can be placed right on top of the function that handles the endpoint). This is closer to how Django does it than to how Flask (and Starlette) does it. It separates in the code things that are relatively tightly coupled.

!!! check "Inspired **FastAPI** to" Define extra validations for data types using the "default" value of model attributes. This improves editor support, and it was not available in Pydantic before.

```
This actually inspired updating parts of Pydantic, to support the same validation declaratio
```

Hug

Hug was one of the first frameworks to implement the declaration of API parameter types using Python type hints. This was a great idea that inspired other tools to do the same.

It used custom types in its declarations instead of standard Python types, but it was still a huge step forward.

It also was one of the first frameworks to generate a custom schema declaring the whole API in JSON.

It was not based on a standard like OpenAPI and JSON Schema. So it wouldn't be straightforward to integrate it with other tools, like Swagger UI. But again, it was a very innovative idea.

It has an interesting, uncommon feature: using the same framework, it's possible to create APIs and also CLIs.

As it is based on the previous standard for synchronous Python web frameworks (WSGI), it can't handle Websockets and other things, although it still has high performance too.

!!! info Hug was created by Timothy Crosley, the same creator of `isort`, a great tool to automatically sort imports in Python files.

!!! check "Ideas inspired in **FastAPI**" Hug inspired parts of APIStar, and was one of the tools I found most promising, alongside APIStar.

```
Hug helped inspiring **FastAPI** to use Python type hints to declare parameters, and to gene

Hug inspired **FastAPI** to declare a `response` parameter in functions to set headers and c
```

APIStar (<= 0.5)

Right before deciding to build **FastAPI** I found **APIStar** server. It had almost everything I was looking for and had a great design.

It was one of the first implementations of a framework using Python type hints to declare parameters and requests that I ever saw (before NestJS and Molten). I found it more or less at the same time as Hug. But APIStar used the OpenAPI standard.

It had automatic data validation, data serialization and OpenAPI schema generation based on the same type hints in several places.

Body schema definitions didn't use the same Python type hints like Pydantic, it was a bit more similar to Marshmallow, so, editor support wouldn't be as good, but still, APIStar was the best available option.

It had the best performance benchmarks at the time (only surpassed by Starlette).

At first, it didn't have an automatic API documentation web UI, but I knew I could add Swagger UI to it.

It had a dependency injection system. It required pre-registration of components, as other tools discussed above. But still, it was a great feature.

I was never able to use it in a full project, as it didn't have security integration, so, I couldn't replace all the features I was having with the full-stack generators based on Flask-apispec. I had in my backlog of projects to create a pull request adding that functionality.

But then, the project's focus shifted.

It was no longer an API web framework, as the creator needed to focus on Starlette.

Now APIStar is a set of tools to validate OpenAPI specifications, not a web framework.

!!! info APIStar was created by Tom Christie. The same guy that created:

```
* Django REST Framework
* Starlette (in which **FastAPI** is based)
* Uvicorn (used by Starlette and **FastAPI**)
```

!!! check "Inspired **FastAPI** to" Exist.

```
The idea of declaring multiple things (data validation, serialization and documentation) wit

And after searching for a long time for a similar framework and testing many different alter

Then APIStar stopped to exist as a server and Starlette was created, and was a new better fo

I consider **FastAPI** a "spiritual successor" to APIStar, while improving and increasing th
```

Used by FastAPI

Pydantic

Pydantic is a library to define data validation, serialization and documentation (using JSON Schema) based on Python type hints.

That makes it extremely intuitive.

It is comparable to Marshmallow. Although it's faster than Marshmallow in benchmarks. And as it is based on the same Python type hints, the editor support is great.

!!! check "**FastAPI** uses it to" Handle all the data validation, data serialization and automatic model documentation (based on JSON Schema).

```
**FastAPI** then takes that JSON Schema data and puts it in OpenAPI, apart from all the othe
```

Starlette

Starlette is a lightweight ASGI framework/toolkit, which is ideal for building high-performance asyncio services.

It is very simple and intuitive. It's designed to be easily extensible, and have modular components.

It has:

- Seriously impressive performance.
- WebSocket support.
- In-process background tasks.
- Startup and shutdown events.
- Test client built on requests.
- CORS, GZip, Static Files, Streaming responses.
- Session and Cookie support.
- 100% test coverage.
- 100% type annotated codebase.

- Few hard dependencies.

Starlette is currently the fastest Python framework tested. Only surpassed by Uvicorn, which is not a framework, but a server.

Starlette provides all the basic web microframework functionality.

But it doesn't provide automatic data validation, serialization or documentation.

That's one of the main things that **FastAPI** adds on top, all based on Python type hints (using Pydantic). That, plus the dependency injection system, security utilities, OpenAPI schema generation, etc.

!!! note "Technical Details" ASGI is a new "standard" being developed by Django core team members. It is still not a "Python standard" (a PEP), although they are in the process of doing that.

```
Nevertheless, it is already being used as a "standard" by several tools. This greatly improv
```

!!! check "**FastAPI** uses it to" Handle all the core web parts. Adding features on top.

```
The class `FastAPI` itself inherits directly from the class `Starlette`.

So, anything that you can do with Starlette, you can do it directly with **FastAPI**, as it
```

Uvicorn

Uvicorn is a lightning-fast ASGI server, built on uvloop and httptools.

It is not a web framework, but a server. For example, it doesn't provide tools for routing by paths. That's something that a framework like Starlette (or **FastAPI**) would provide on top.

It is the recommended server for Starlette and **FastAPI**.

!!! check "**FastAPI** recommends it as" The main web server to run **FastAPI** applications.

```
You can combine it with Gunicorn, to have an asynchronous multi-process server.

Check more details in the [Deployment](deployment/index.md){.internal-link target=_blank} se
```

Benchmarks and speed

To understand, compare, and see the difference between Uvicorn, Starlette and FastAPI, check the section about Benchmarks.