LearnDjango

Flask vs. Django im Jahr 2025: Ein umfassender Vergleich von Python-Webframeworks

Aktualisiert11. September 2025

▶ Inhaltsverzeichnis

Flask und Django sind die beiden führenden Python-Webframeworks. Beide ermöglichen es Entwicklern, schnell Websites zu erstellen, verfolgen dabei aber grundverschiedene Ansätze. In diesem Artikel betrachten wir die Funktionen und die Funktionsweise der einzelnen Frameworks und erläutern, warum Entwickler sich für das eine oder das andere entscheiden. Um diese Unterschiede zu verdeutlichen, entwickeln wir drei verschiedene Projekte von Grund auf – eine Hello-World-App, eine persönliche Website und eine Aufgabenliste –, damit Sie sich selbst ein Bild von der Funktionsweise machen und die beste Entscheidung für Ihre Bedürfnisse treffen können.

Was ist ein Web-Framework?

Datenbankbasierte Websites haben bemerkenswert ähnliche Anforderungen: URL-Routing, Logik, Datenbankanbindung, Darstellung von HTML-Vorlagen, Benutzerauthentifizierung usw. In den Anfängen des World Wide Web mussten Entwickler all diese Komponenten selbst implementieren, bevor sie überhaupt mit der eigentlichen Website-Entwicklung beginnen konnten.

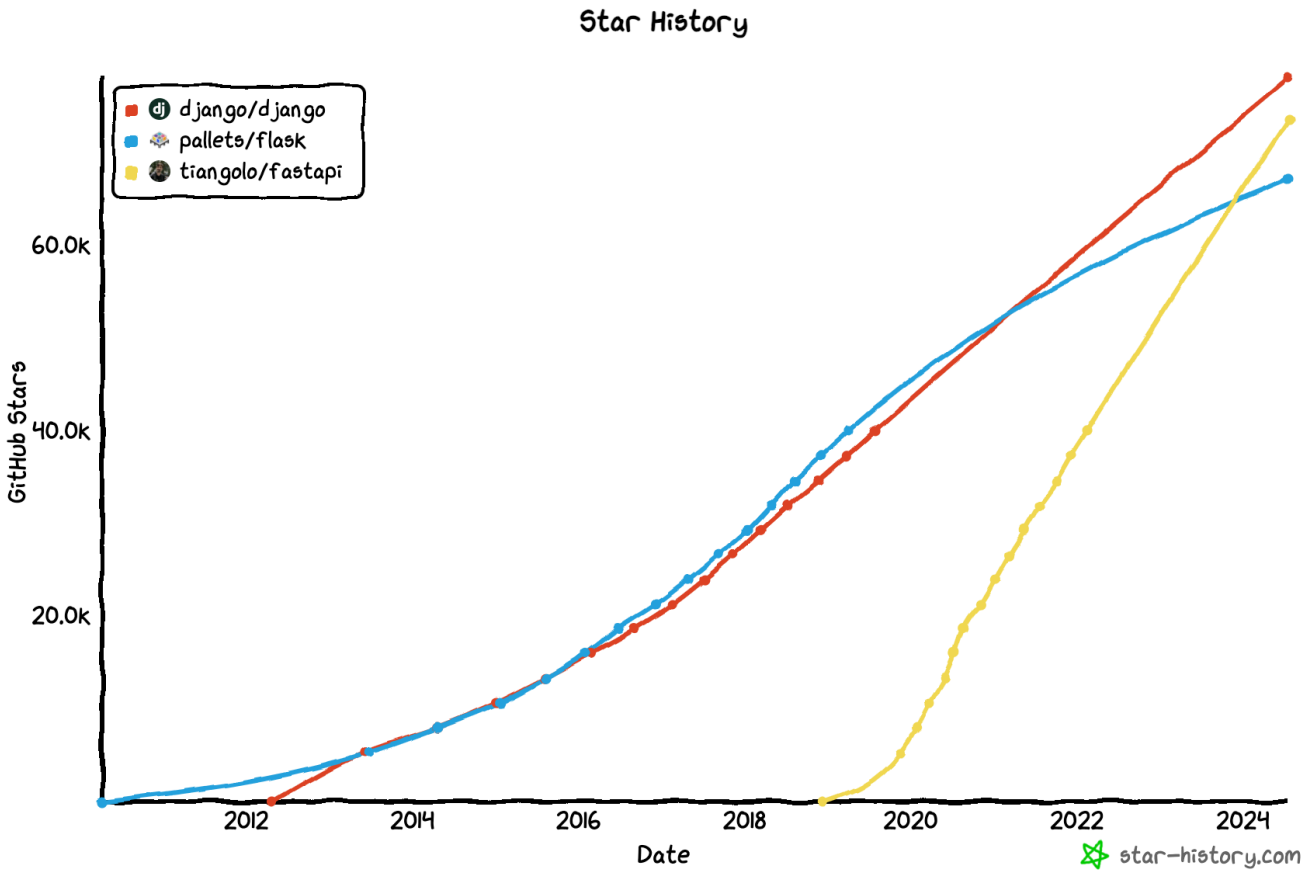
Bald entstanden Open-Source-Webframeworks, die es Entwicklergruppen ermöglichten, gemeinsam an dieser Herausforderung zu arbeiten, Best Practices auszutauschen, Code zu überprüfen und generell das Rad nicht jedes Mal neu erfinden zu müssen, wenn jemand eine neue

Website erstellen wollte. Es gibt Webframeworks für jede gängige Programmiersprache. Bekannte Beispiele sind Ruby on Rails (in Ruby geschrieben), Laravel (in PHP geschrieben), Spring (in Java geschrieben) und die beiden hier vorgestellten Python-Frameworks Flask und Django.

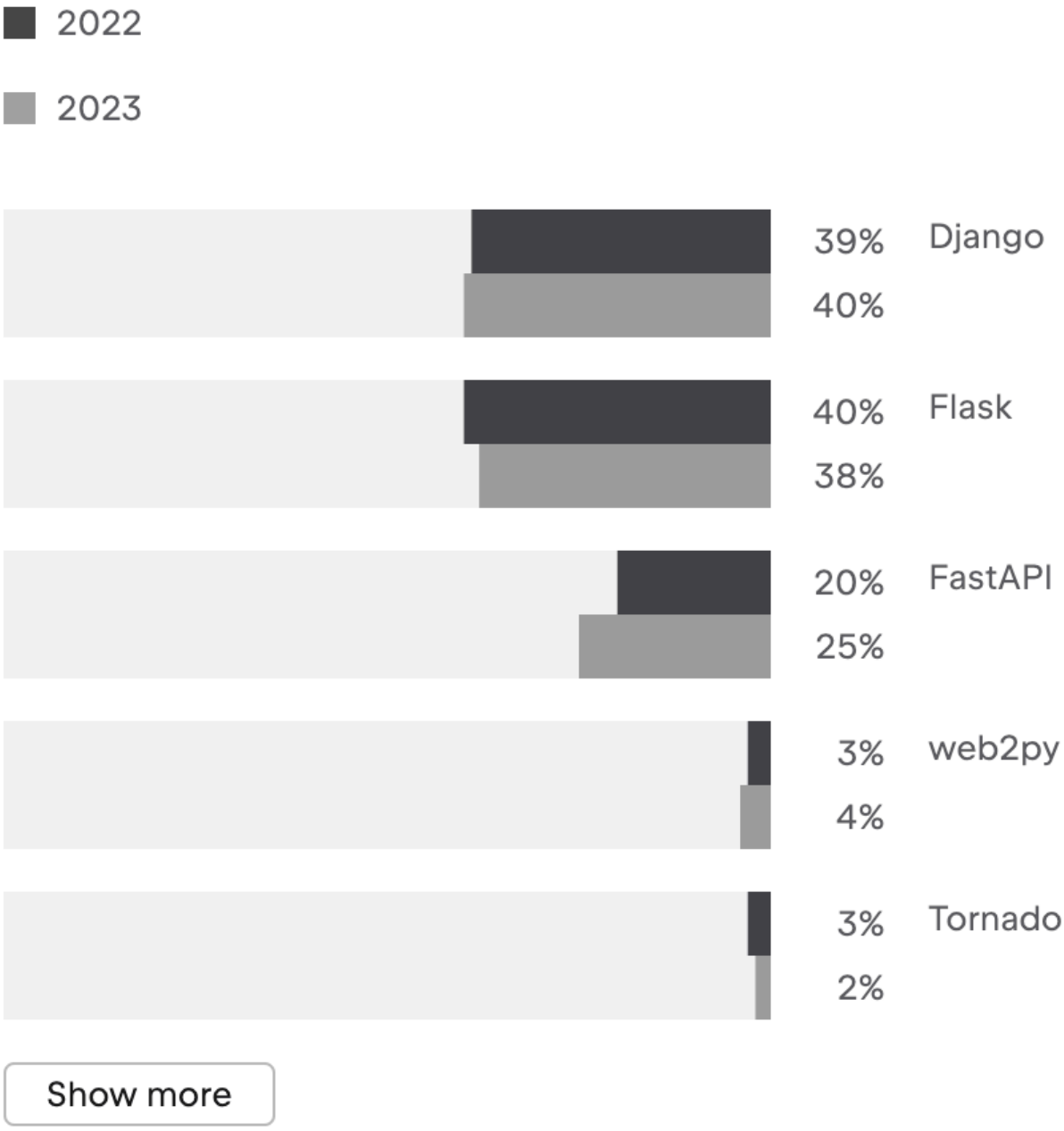
Wenn Sie noch nicht viel Programmiererfahrung haben, mag der Unterschied zwischen Python-Framework und -Bibliothek zunächst verwirrend sein. Beide bezeichnen Software, unterscheiden sich aber in ihrer Komplexität: Eine Bibliothek konzentriert sich auf ein spezifisches Problem, während ein Framework eine größere Herausforderung angeht und dafür oft viele kleinere Bibliotheken einbindet. Wie wir sehen werden, verwenden sowohl Flask als auch Django zahlreiche Python-Bibliotheken.

Welche ist beliebter?

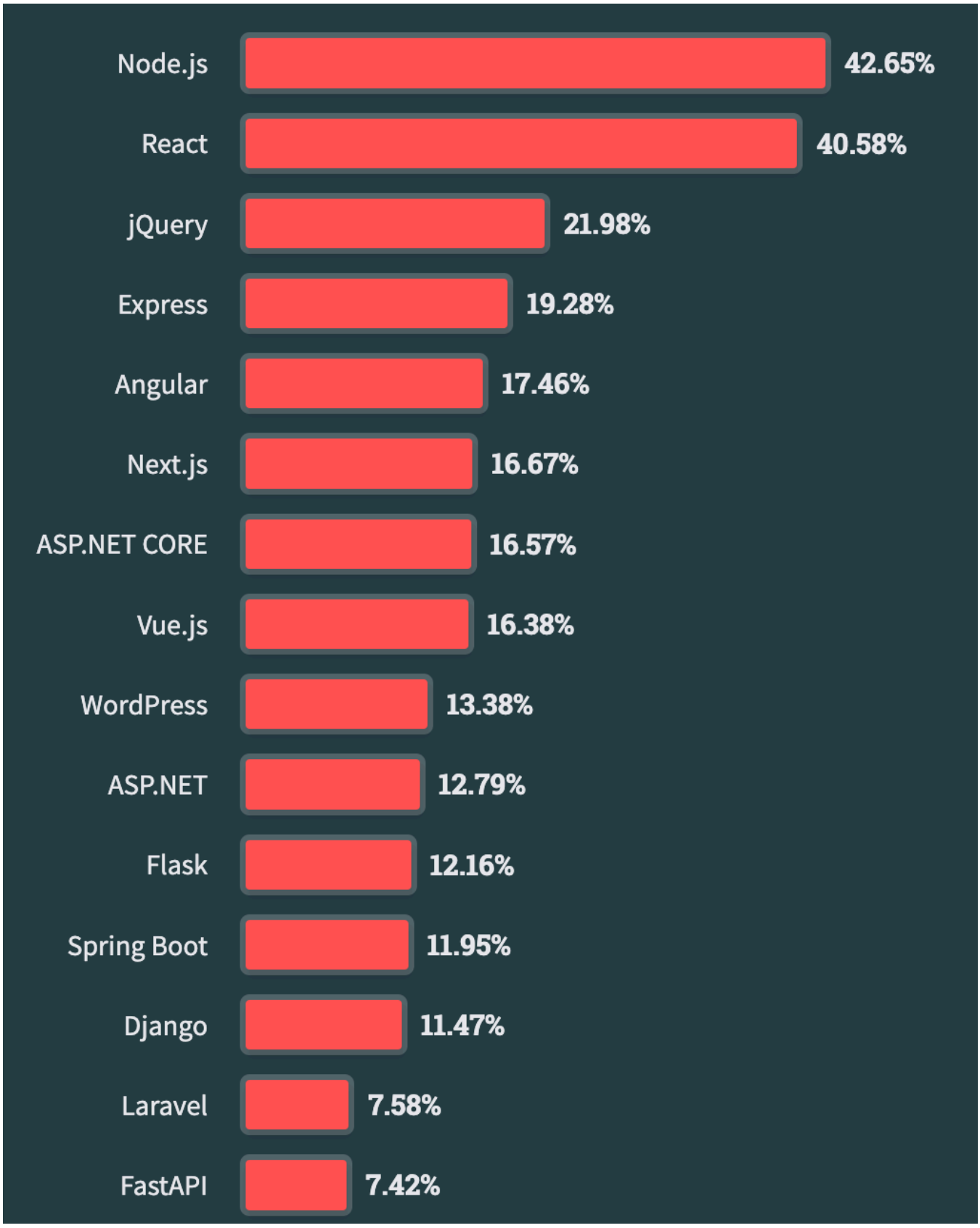
Wenn wir uns die GitHub-Sterne ansehen , liegen Flask und Django relativ gleichauf, aber wir können das explosive Wachstum von FastAPI erkennen , das sich mittlerweile klar unter den Top 3 der Python-Webframeworks etabliert hat.



Laut der Python Developers Survey 2023 liegt Django im Jahr 2023 knapp vor Flask, wobei FastAPI ebenfalls an Bedeutung gewinnt.



Laut einer Umfrage von Stack Overflow aus dem Jahr 2023 unter Entwicklern aller Programmiersprachen liegt Flask leicht vorn, wird aber fast unmittelbar von Django gefolgt, während FastAPI etwas zurückliegt.



Diese Vergleiche sind zwar interessant, um Trends zu erkennen, berücksichtigen aber viele Aspekte nicht. Nur weil ein Web-Framework populär ist, heißt das beispielsweise nicht, dass es auch von etablierten Unternehmen und professionellen Entwicklern eingesetzt wird, oder ist es nur etwas, mit dem Anfänger gerne experimentieren?

Unabhängig vom Vergleichsmaßstab ist klar, dass Flask und Django derzeit die beiden führenden Python-Webframeworks sind.

Jobs

Wenn Sie einen Job als Python-Webentwickler suchen, ist Django die bessere Wahl. Auf großen Jobportalen wie Indeed.com gibt es fast doppelt so viele Stellenangebote für Django-Entwickler wie für Flask- Entwickler.

Diese Diskrepanz dürfte jedoch darauf zurückzuführen sein, dass Django eine deutlich spezialisiertere Lösung als Flask darstellt. Ein Startup oder Unternehmen kann nahezu alle seine Dienste ausschließlich mit Django betreiben, während Flask aufgrund seines geringen Ressourcenbedarfs häufig in Kombination mit anderen Technologien eingesetzt wird.

Der vielversprechendste Ansatz ist, zunächst Python wirklich zu beherrschen und dann Webentwicklungskenntnisse mit Django oder Flask (idealerweise beides!) hinzuzufügen.

Gemeinschaft

Django verfügt über die größere und besser organisierte Community der beiden Frameworks. Über 1.800 Entwickler tragen zum Django-Quellcode bei, im Vergleich zu etwa 550 bei Flask. Auf Stack Overflow finden sich rund 212.500 Fragen zu Django, verglichen mit etwa 31.500 Fragen zu Flask .

Django veranstaltet außerdem jährliche Konferenzen in den USA , Europa und Australien . Flask kann nicht auf ein vergleichbares Konferenzniveau zurückblicken, obwohl beide Frameworks auf PyCon- Veranstaltungen rege diskutiert werden.

Was ist eine Flasche?

Flask ist ein bewusst minimalistisches und flexibles Mikro-Framework, dessen Nutzen dadurch aber keineswegs eingeschränkt wird. Wie wir sehen werden, bringt diese Designentscheidung sowohl Stärken als auch Schwächen mit sich.

Flask entstand 2010 als Aprilscherz von Armin Ronacher und wurde vom Sinatra Ruby-Framework inspiriert. Flask sollte so einfach sein, dass es in eine einzige Python-Datei passte, und trotz seines

humorvollen Ursprungs gewann Flask aufgrund seiner Einfachheit und Flexibilität schnell an Popularität.

Flask selbst hat eine recht kleine Codebasis und stützt sich stark auf zwei wichtige Abhängigkeiten: Werkzeug und Jinja , die beide ursprünglich von Armin Ronacher entwickelt wurden.

Werkzeug ist ein WSGI-Toolkit (Web Server Gateway Interface), das die Kernfunktionalität für Flask bereitstellt. Es verarbeitet HTTP-Anfragen und -Antworten, bietet ein URL-Routing-System, einen integrierten Entwicklungsserver, einen interaktiven Debugger, einen Testclient und Middleware. Jinja ist eine Template-Engine zur Generierung dynamischer HTML-Dokumente mit eigener Syntax für grundlegende Logik, Variablen, if/else-Schleifen, Template-Vererbung und mehr.

Obwohl Flask keine spezifische Struktur vorschreibt, wird es häufig im Model-View-Controller (MVC)-Muster verwendet, das auch bei anderen Web-Frameworks wie Ruby on Rails üblich ist.

- **Modell** : Interagiert mit der Datenbank und verarbeitet die Datenlogik.
- **Ansicht** : Rendert (in der Regel) HTML-Vorlagen mit Daten für den Benutzer.
- **Controller** : Verarbeitet Benutzereingaben, interagiert mit dem Modell und wählt die darzustellende Ansicht aus.

Flask's micro-framework architecture means it performs a few tasks extremely well and relies on third-party libraries (and the developer) to implement the rest. This approach is well-suited to smaller web applications that don't require all the bells and whistles built into Django. On the other extreme, experienced programmers who demand complete control over their application often prefer Flask, though this means making more design decisions than they would with a full framework like Django.

What is Django?

Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. It was created in at the Lawrence Journal-World newspaper and publicly released in 2005. "High-level" means that Django is designed to minimize the actual coding required during the web application process by providing built-in "batteries" for most use cases, including an ORM (Object-Relational Mapper), URL routing, template engine, form handling, authentication system, admin interface, and robust security features. In Flask, the developer must choose and implement these various features but with Django they are included out-of-the-box.

Django is managed by the non-profit Django Software Foundation and has a large and dedicated community behind it working on new releases, extensive documentation, active online communities, and regular community-run conferences.

Django follows a variant of the MVC architecture called the Model-View-Template (MVT) pattern that emphasizes separation of concerns:

- **Model**: Handles data and business logic, including methods to interact with the data
- **View**: Handles business logic and interacts with the Model and Template. It also processes user requests.
- **Templates**: Render the user interface, usually as HTML using the Django templating language

A fourth component, **URLs**, is also included and used to handle URL routing, matching a user request to a specific View that then generates a response.

Flask: Hello, World

Python should already be installed on your computer so all we need to do is create a virtual environment and install Flask.

```
# Windows
$ python -m venv .venv
$ .venv\Scripts\Activate.ps1
(.venv) $ python -m pip install flask

# macOS/Linux
$ python3 -m venv .venv
$ source .venv/bin/activate
(.venv) $ python -m pip install flask
```

With your text editor, create a new file called **`hello.py`**. Flask famously requires only five lines for a Hello World web page.

```
# app.py
from flask import Flask

app = Flask(__name__)
```

```
@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"
```

This code imports the `Flask` class at the top and creates an instance called `app` on the next line. The `route()` decorator tells Flask what URL should trigger the function; here it is set to the homepage at `/`. Then the function, `hello_world`, returns an HTML string between paragraph `<p>` tags with our message.

To run the code, use the `flask run` command.

```
(.venv)
$ flask run
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deploymen
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

If you navigate to `127.0.0.1:5000` in your web browser the message is visible. Flask uses port `5000` by default.

This is about as simple as can be and speaks to both Flask's roots as an attempt at a single-file way to create a web application and also an example of its inherent flexibility.

Django: Hello, World

The Django docs don't provide a similar quickstart guide but we can accomplish a similar feat with only a few more lines of code. In fact, doing so has become a bit of a game among seasoned Django developers and there is an entire repo, django-microframework, dedicated to these efforts. We will choose Option2, which is not the most concise, but is more easily understood than some of the other approaches.

Navigate to a new directory, perhaps called **`django`** on your Desktop, and create a virtual environment containing Django.

```
# Windows
> cd onedrive\desktop\code
> mkdir django
> cd django
> python -m venv .venv
> .venv\Scripts\Activate.ps1
(.venv) > python -m pip install django

# macOS
% cd ~/desktop/code
% mkdir django
% cd django
% python3 -m venv .venv
% source .venv/bin/activate
(.venv) % python3 -m pip install django
```

In your text editor create a **`hello_django.py`** file with the following code:

```
# hello_django.py
from django.conf import settings
from django.core.handlers.wsgi import WSGIHandler
from django.core.management import execute_from_command_line
from django.http import HttpResponse
from django.urls import path

settings.configure(
    ROOT_URLCONF=__name__,
    DEBUG=True,
)


def hello_world(request):
    return HttpResponse("Hello, Django!")


urlpatterns = [path("", hello_world)]

application = WSGIHandler()
```

```
if __name__ == "__main__":
    execute_from_command_line()
```

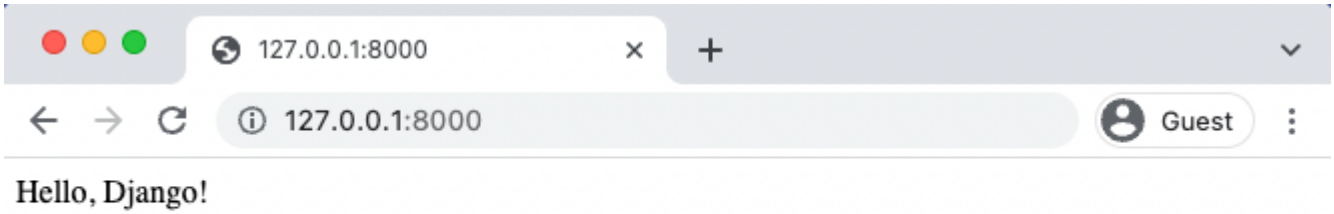
Django is designed for larger web application and typically relies on a global `settings.py` file for many configurations, however we can import what we need in a single file. The key points of reference are the `hello_world` function that returns the string, "Hello, Django!" and the `urlpatterns` defining our URL routes, namely at `""`, meaning the empty string, so the homepage.

Start up Django's built-in server using the `runserver` command

```
(.venv) > python hello_django.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
July 17, 2024 - 13:48:54
Django version 5.0, using settings None
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Navigate to Django's standard port of 8000, http://127.0.0.1:8000/, to see the "Hello, Django!" message.



Django required twelve lines of code rather than Flask's five, but both these examples are intended as quickstart guides; they are not how you structure a real-world Flask or Django app.

Flask Personal Website

Now let's build a Personal Website with a home page and an about page. This will give a chance to introduce templates and repeat some of the patterns we saw around how routes are defined in Flask.

Update the **`app.py`** file as follows:

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def home():
    return render_template('home.html')

@app.route('/about')
def about():
    return render_template('about.html')

if __name__ == '__main__':
    app.run(debug=True)
```

Both the **`home`** and **`about`** View functions now return a template. We're also using the **`route()`** decorator again to define the URL path for each. We've also added **`debug=True`** at the bottom so that the development server runs now in debug mode.

The next step is creating our two templates. Flask will look for template files in a **`templates`** directory so create that now.

```
(.venv) $ mkdir templates
```

Within it add the two files with the following code:

1. Create templates in **`templates/`**:

```
<!-- templates/home.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Personal Website</title>
```

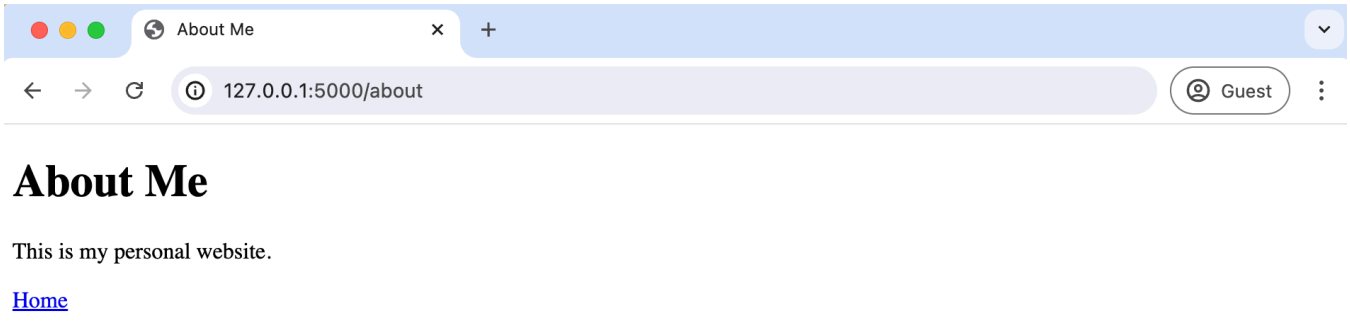
```
</head>
<body>
    <h1>Welcome to My Website</h1>
    <a href="{{ url_for('about') }}">About Me</a>
</body>
</html>
```

```
<!-- templates/about.html -->
<!DOCTYPE html>
<html>
<head>
    <title>About Me</title>
</head>
<body>
    <h1>About Me</h1>
    <p>This is my personal website.</p>
    <a href="{{ url_for('home') }}">Home</a>
</body>
</html>
```

Each file use the method url_for to define links based on the view function name.

Run the server again with `flask run` and navigate to the homepage:

Then click the "About Me" link.



This is a rudimentary example but you can start to see how templates and views interact in Flask. We still have only one main Python file powering the whole thing, but once we have many more pages and start to introduce logic, the single-file approach stops making sense and it's time to start organizing the Flask app in different ways. There are some common patterns used in the Flask community, however, it is ultimately up to the developer.

Django: Personal Website

Django is designed for full-bodied web applications so building a Personal Website is a chance to see this in action. We'll start by creating a project, which is the central hub for our website, using the `**startproject**` command.

```
(.venv) $ django-admin startproject django_project .
```

We've named the project `**django_project**` here. Adding the period, `**.**`, means the new folder and files are installed in the current directory. If you don't have the period Django creates a new directory and *then* adds the project folder and files there.

This is what your directory should look like now. The `**hello_django.py**` file remains and can either be left there or removed entirely: we will no longer use it. There is an entirely new `**django_project**` folder containing several files and a `**manage.py**` file used for running Django commands.

```
├── django_project
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
```

```
├── hello_django.py
└── manage.py
```

We want to create an app now using these **`startapp`** command which will be called **`pages`**. A single Django project typically has multiple apps for different functionality. If we added user registration that code should be in its own app, same for payments, and so on. This is a way to help developers reason better about their code.

```
(.venv) $ python manage.py startapp pages.
```

This command creates a **`pages`** directory with the following files:

```
└── pages
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── migrations
    │   └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
```

Our first step is updating the **`django_project/settings.py`** file to tell Django about our new app. This is a global settings file for the entire project.

```
# django_project/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "pages",  # new
]
```

Second, update the **`django_project/urls.py`** file. When a URL request comes in it will hit this file first and then be either processed or redirected to a specific app. In this case, we want to send

requests to the `pages` app. To do this we'll import `include` and set a new path at `""`, meaning the homepage. Django defaults to including the URL configuration for the built-in admin, a powerful visual way to interact with your database.

```
# django_project/urls.py
from django.contrib import admin
from django.urls import path, include  # new

urlpatterns = [
    path("admin/", admin.site.urls),
    path("", include("pages.urls")),  # new
]
```

Within the `pages` app we need a view and a URLs file. Let's start with the view at `pages/views.py`.

```
# pages/views.py
from django.shortcuts import render


def home(request):
    return render(request, "home.html")


def about(request):
    return render(request, "about.html")
```

In Django, views receive web requests and return web responses. The `request` parameter is an object containing metadata about the request from the user. We'll define two function-based views here, `home` and `about`, that use the shortcut function `render` to combine a template with an `HttpResponse` object sent back to the user. The two templates are `home.html` and `about.html`.

For the templates, we can create a `templates` directory within `pages`, then another directory with the app name, and finally our template files. This approach removes any concerns about confusing the Django template loader in larger projects.

```
(.venv) $ mkdir pages/templates
```

```
(.venv) $ mkdir pages/templates/pages
```

Then in your text editor add two new files: **`pages/templates/pages/home.html`** and **`pages/templates/pages/about.html`**.

```
<!-- pages/templates/pages/home.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Personal Website</title>
</head>
<body>
    <h1>Welcome to My Website</h1>
    <a href="{% url 'about' %}">About Me</a>
</body>
</html>
```

```
<!-- pages/templates/pages/about.html -->
<!DOCTYPE html>
<html>
<head>
    <title>About Me</title>
</head>
<body>
    <h1>About Me</h1>
    <p>This is my personal website.</p>
    <a href="{% url 'home' %}">Home</a>
</body>
</html>
```

The final step is configuring the URLs for these two pages. To do this, create a **`urls.py`** file within the **`pages`** app with the following code.

```
# pages/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path("", views.home, name="home"),
```

```
    path("about/", views.about, name="about"),
]
```

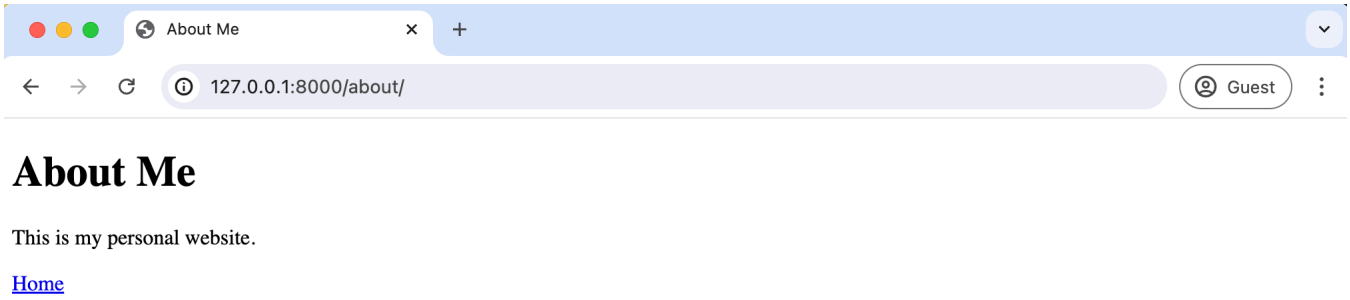
At the top we import our views and then set a URL path for each. The syntax is defining the URL path, the view name, and optionally adding a URL `name` that allows us to link to each path in our templates.

Start up the Django local server with the `runserver` command.

```
(.venv) $ python manage.py runserver
```

You can see the homepage:

Click the "About Me" to be redirected to the about page:



As you can see Django required more scaffolding than Flask, however this approach provides a consistent structure that is quite scalable.

Detailed Comparison

The true comparison of these web frameworks depends on your project's needs. Are you building a traditional web application that connects to a database, requires CRUD (Create-Read-Update-

Delete) functionality, and user authentication? If yes, Django has built-in solutions for all of these needs. By comparison, Flask requires installing multiple third-party libraries: `Flask-SQLAlchemy` to connect to the database, `Flask-Migrate` to manage database migrations, `Flask-WTF` and `WTForms` for forms, `Flask-Login` for user authentication, `FLask-Mail` for email support, `Flask-Security` for security features, `Flask-Admin` for an admin interface to manage application data, `Flask-Caching` for caching support, `Flask-BCrypt` for password hashing and so on.

The power of Django is that you don't have to worry about any of these things. They are included, tested, and supported by the community. For Flask, the third-party libraries are not as tightly integrated and require more manual installation by the developer. This affords greater flexibility but also requires more programmer expertise.

Conclusion

Ultimately, you can't go wrong choosing Flask or Django for your web application needs. They both are mature, scalable, and well-documented. This difference is in approach and the best way to determine what you prefer is to try each out by building more complex projects.

Wenn Sie mehr über Django erfahren möchten, empfehle ich Ihnen „Django für Anfänger“. Dort lernen Sie, wie Sie sechs zunehmend komplexere Webanwendungen erstellen, inklusive Tests und Deployment. Für Flask gibt es das Flask Mega-Tutorial als kostenlose Online-Version. Außerdem sind zwei Kurse auf TestDriven.io empfehlenswert: „TDD mit Python, Flask und Docker“ und „Authentifizierung mit Flask, React und Docker“ . Falls Sie Videos bevorzugen, finden Sie auf Udemy zahlreiche Flask-Kurse. Der beste Videokurs, den ich kenne, ist „ Eine SaaS-App mit Flask und Docker erstellen“ .

you@domain.com

Abonnieren