

# What is Flask Python



Flask is a web framework, it's a Python module that lets you develop web applications easily. It's has a small and easy-to-extend core: it's a microframework that doesn't include an ORM (Object Relational Manager) or such features.

It does have many cool features like url routing, template engine. It is a WSGI web app framework.

# What is Flask?

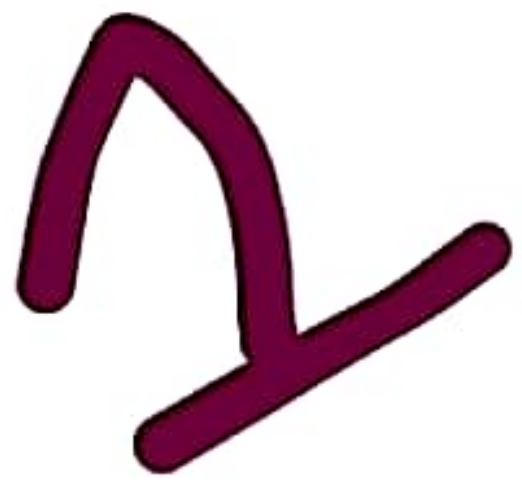
---



Flask is a web application framework written in Python. It was developed by Armin Ronacher, who led a team of international Python enthusiasts called Pooeco. Flask is based on the Werkzeug WSGI toolkit and the Jinja2 template engine. Both are Pooeco projects.

# What is a Web Framework?

---



A Web Application Framework or a simply a Web Framework represents a collection of libraries and modules that enable web application developers to write applications without worrying about low-level details such as protocol, thread management, and so on.

## What is Flask?

---

Flask is a web application framework written in Python. It was developed by Armin Ronacher, who led a team of international Python enthusiasts called Pocco. Flask is based on the Werkzeug WSGI toolkit and the Jinja2 template engine. Both are Pocco projects.



# jinja2



jinja2 is a popular template engine for Python. A web template system combines a template with a specific data source to render a dynamic web page.

This allows you to pass Python variables into HTML templates like this:

```
<html>
  <head>
    <title>{{ title }}</title>
  </head>
  <body>
    <h1>Hello {{ username }}</h1>
  </body>
</html>
```

# Microframework



Flask is often referred to as a microframework. It is designed to keep the core of the application simple and scalable.

Instead of an abstraction layer for database support, Flask supports extensions to add such capabilities to the application.

Unlike the Django framework, Flask is very Pythonic. It's easy to get started with Flask, because it doesn't have a huge learning curve.

On top of that it's very explicit, which increases readability. To create the "Hello World" app, you only need a few lines of code.

This is a boilerplate code example.

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

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

If you want to develop on your local computer, you can do so easily. Save this program as `server.py` and run it with `python server.py`.

If you want to develop on your local computer, you can do so easily. Save this program as `server.py` and run it with `python server.py`.

```
$ python server.py
* Serving Flask app "hello"
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

It then starts a web server which is available only on your computer. In a web browser open localhost on port 5000 (the url) and you'll see "Hello World" show up.

To host and develop online, you can use [PythonAnywhere](#)



# Application Setup

The first step in creating a Flask application is creating the application object. Each Flask application is an instance of the **Flask** class, which collects all configuration, extensions, and views.



```
from flask import Flask

app = Flask(__name__)
app.config.from_mapping(
    SECRET_KEY="dev",
)
app.config.from_prefixed_env()

@app.route("/")
def index():
    return "Hello, World!"
```



# Application Structure and Lifecycle

Flask makes it pretty easy to write a web application. But there are quite a few different parts to an application and to each request it handles.

Knowing what happens during application setup, serving, and handling requests will help you know what's possible in Flask and how to structure your application.

This is known as the “application setup phase”, it’s the code you write that’s outside any view functions or other handlers. It can be split up between different modules and sub-packages, but all code that you want to be part of your application must be imported in order for it to be registered.



All application setup must be completed before you start serving your application and handling requests. This is because WSGI servers divide work between multiple workers, or can be distributed across multiple machines. If the configuration changed in one worker, there’s no way for Flask to ensure consistency between other workers.

- Adding routes, view functions, and other request handlers with `@app.route`, `@app.errorhandler`, `@app.before_request`, etc.
- Registering blueprints.
- Loading configuration with `app.config`.
- Setting up the Jinja template environment with `app.jinja_env`.
- Setting a session interface, instead of the default `itsdangerous` cookie.
- Setting a JSON provider with `app.json`, instead of the default provider.
- Creating and initializing Flask extensions.





# Introduction



Flask is one of the most popular web application frameworks written in [Python](#). It is a microframework designed for an easy and quick start. Extending with tools and libraries adds more functionality to Flask for more complex projects.

**This article explains how to install Flask in a virtual testing environment and create a simple Flask application.**



## Prerequisites

- Installed Python 2.7 or Python 3.5 and newer
- CLI with administrator privileges

# Step 1: Install

## Virtual

3

## Environment

Install Flask in a virtual environment to avoid problems with conflicting libraries. [Check Python version](#) before starting:

- Python 3 comes with a virtual environment module called *venv* preinstalled. If you have Python 3 installed, skip to Step 2.
- Python 2 users must install the *virtualenv* module. If you have Python 2, follow the instructions outlined in Step 1.

## Linux

# Install virtualenv on

The package managers on Linux provides *virtualenv*.

• For Debian/Ubuntu:

# Install virtualenv on Linux

The package managers on Linux provides *virtualenv*.

- **For Debian/Ubuntu:**

1. Start by opening the Linux terminal.
2. Use **apt** to install *virtualenv* on Debian, Ubuntu and other related distributions:



```
sudo apt install python  
-virtualenv
```



- **For CentOS/Fedora/Red Hat:**

1. Open the Linux terminal.
2. Use **yum** to install *virtualenv* on CentOS, Red Hat, Fedora and related distributions:



```
sudo yum install python  
virtualenv
```





# Install virtualenv on MacOS



1. Open the terminal.
2. Install *virtualenv* on Mac using **pip**:

```
sudo python2 -m pip install virtualenv
```



# Install virtualenv on Windows



1. Open the command line with administrator privileges.
2. Use **pip** to install *virtualenv* on Windows:

```
py -2 -m pip install virtualenv
```



# Basic Routing in Flask

Routing in Flask determines how incoming requests are handled based on the URL a user has requested. Flask uses the `route()` decorator method of the Flask application instance to define routes and then bind them to appropriate view functions. To demonstrate basic routing in Flask, we start by importing the `Flask` class from the `flask` module:

```
from flask import Flask
```

Once we have the `Flask` class, we can create the application instance and store it in a variable called `app`. The `Flask` class takes in a `__name__` argument, which is a special Python variable denoting the name of the current module containing the Flask application instance:

```
app = Flask(__name__)
```

Using the application instance, we have access to its various methods and decorators, which we can use to define routes, handle requests, and perform other tasks in our web application. However, for this example, we'll be interested in the `route()` decorator, a special method which, when applied to a function in Flask, turns it into a view function that will handle web requests. It takes in a mandatory URL pattern and optional HTTP methods as its arguments. The `route()` decorator enables us to associate a URL pattern with the decorated function, essentially saying that if a user visits the URL defined in the decorator, the function will be triggered to handle this request:



```
@app.route('/')  
def index():  
    return "This is a basic flask"
```

In the code snippet above, we have the `route(/)` decorator applied to the `index()` function, meaning that the function will handle requests to the root URL `'/'`. So when a user accesses the URL, Flask will trigger the `index()` function that will return the string "This is a basic Flask application", and it will be displayed in the browser. To ensure the application runs when this module is invoked with Python in the command line, add the `if __name__` check:

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

# Flask Rendering Templates

Flask is a backend web framework based on the Python programming language. It basically allows the creation of web applications in a Pythonic syntax and concepts. With Flask, we can use Python libraries and tools in our web applications. Using Flask we can set up a web server to load up some basic HTML templates along with Jinja2 templating syntax. In this article, we will see how we can render the HTML templates in [Flask](#).



# Rendering a Template in a Flask Application



Setting up Flask is quite easy. We can use a virtual environment to create an isolated environment for our project and then install the Python packages in that environment. After that, we set up the environment variables for running Flask on the local machine. This tutorial assumes that you have a Python environment configured, if not please follow through for [setting up Python](#) and [pip](#) on your system. Once you are done, you are ready to develop Flask applications.

## Setting up the Virtual Environment

To set up a virtual environment, we can make use of the Python Package Manager “pip” to install the “virtualenv” package.

```
pip install virtualenv
```



starting point of our application. We can achieve this by creating a file called **“server.py”** You can call this anything you like, but keep it consistent with other Flask projects you create. Inside the server.py paste the following code:

Python

```
from flask import Flask
app = Flask(__name__)

if __name__ == "__main__":
    app.run()
```

This is the code for actually running and creating the Flask app. This is so-called the entry point of a Flask web server. As you can see we are importing the Flask module and instantiating with the current file name in **“Flask(\_\_name\_\_)”**. Hence after the check, we are running a function called **run()**.

can be optionally replaced later. We start with a single block called the body.

## templates\index.html

### HTML



```
<!DOCTYPE html>
<html>
<head>
    <title>FlaskTest</title>
</head>
<body>
    <h2>Welcome To GFG</h2>
    <h4>Flask: Rendering Templa
    <!-- this section can be repla
    {% block body %}

    <p>This is a Flask application.
    {% endblock %}

</body>
</html>
```

# Flask - pass variables to templates



## Flask Tip - Jinja Templates

You can pass variables as arguments to

`render_template()` to use those variables in a template file.



```
# app.py
@app.route('/about')
def about():
    return render_template('abo
```

```
# about.html
<h2>Course developed by {{ orga
```



# Flask - Message Flashing



## Flask Tip - Message Flashing

Flash messages are used to provide useful information to the user based on their actions with the app. The `flash()` method is used to create a flash message to be displayed in the next request.

```
from flask import request, redirect,  
url_for, render_template, flash
```

```
@stocks_blueprint.route('/add_stock',  
methods=['GET', 'POST'])
```

```
def add_stock():
```

```
    if request.method == 'POST':
```

```
        # ... save the data ...
```

```
        flash(f"Added new stock  
({stock_data.stock_symbol})!") # <-- !!
```

```
        return
```

```
    redirect(url_for('stocks.list_stocks'))
```

```
    return render_template('stocks/  
add_stock.html')
```



## Retrieving HTML Form data using Flask

Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications. It began as a simple wrapper around Werkzeug and Jinja and has become one of the most popular Python web application frameworks.

Read this article to know more about Flask Create form as HTML

We will create a simple HTML Form, very simple Login form

```
<form action="{{ url_for('gfg')}}"  
method="post">
```

```
<label for="firstname">First Name:</  
label>
```

```
<input type="text" id="firstname"  
name="fname"  
placeholder="firstname">
```

```
<label for="lastname">Last Name:</  
label>
```





```
<label for="lastname">Last Name:</label>
```

```
<input type="text" id="lastname"  
name="lname"  
placeholder="lastname">
```



```
<button type="submit">Login</button>
```

Its an simple HTML form using the post method the only thing is unique is action URL. URL\_for is an Flask way of creating dynamic URLs where the first arguments refers to the function of that specific route in flask. In our



Templates are an essential ingredient in full-stack web development. With **Jinja**, you can build rich templates that power the front end of your Python web applications.

Jinja is a text templating language. It allows you to process a block of text, insert values from a context dictionary, control how the text flows using conditionals and loops, modify inserted data with filters, and compose different templates together using inheritance and inclusion.

**In this video course, you'll learn how to:**

- Install the Jinja **template engine**
- Create your first Jinja **template**
- Render a Jinja template in **Flask**
- Use **for loops** and **conditional statements** with Jinja
- **Nest** Jinja templates
- Modify variables in Jinja with **filters**





# Creating Templates with Jinja

Once Jinja is installed, we can start creating templates. A template is a text file that can contain placeholders for variables. When we render a template, we replace these placeholders with actual values.

Here's a simple example of a Jinja template:

```
from jinja2 import Template

t = Template('Hello, {{ name }}!')
print(t.render(name='John Doe'))

# Output:
# 'Hello, John Doe!'
```

In this example, we first import the `Template` class from the `jinja2` module. We then create a new template with a placeholder for `name`. Finally, we render the template with `name` set to `'John Doe'`, and print the result.



## Back at it again!

Welcome to part 2 of 2 of this series on **Introduction to Python Flask!**

In this walkthrough, we are going to use the same file set up from part 1 to create our flask calculator application. Where we last left off, we learned the boiler plate code for:

- Importing flask
- Creating the flask instance
- Creating routes in the application.
- Creating methods that are invoked when a specific route is called.



If you don't know too much about Python Flask or the file structure that is used, you can refer to Part 1 of this series [here](#).

# Calculator Application (Walkthrough)



## Step 1: Creating our file structure of the project

Similar to part 1 of this series, we need to set up our files to run a Flask application. For this example I created a new folder **calculator\_flask** and inside it are two folders which are **static** and **templates**.

*(Remember static and templates are folders we create in our root directory in order to let our flask app know where to look)*

You should end up with the following file tree

```
calculator_flask
|- -static(folder)
|- -templates (folder)
```

## Step 2: Create your main.py file and index.html file

Now let's cd into the directory we just made |\$ cd calculator\_flask|

Create our main run file in this directory |\$ touch calculator.py|



Now cd into templates |\$ cd templates|

Create your index.html file in this directory |\$ touch index.html|

*(You can also create these files without terminal commands if you prefer)*

After doing all this you should end up with the following file tree



# calculator\_flask

— — static

— — templates

| — — index.html

— — calculator.py

File Tree

You can also add css or image files under the static folder, but for this tutorial I am solely going to focus on:

**index.html**

and

**calculator.py**

## Step 3: Create a GET route for the index page and POST route called /operation\_result/

How:

If you remember from part 1 you need to create a Flask instance and add routes to that Flask instance. Right after you declare/create a route, you need to define a method that will be associated with that route.



Code:

```
calculator_flask > calculator.py > ...
1  from flask import Flask, render_template, request
2
3  Flask_App = Flask(__name__) # Creating our Flask Instance
4
5  @Flask_App.route('/', methods=['GET'])
6  def index():
7      """ Displays the index page accessible at '/' """
8
9      return render_template('index.html')
10
11 @Flask_App.route('/operation_result/', methods=['POST'])
12 def operation_result():
13     """Route where we send calculator form input"""
14     return (render_template, 'index.html')
15
16 if __name__ == '__main__':
17     Flask_App.debug = True
18     Flask_App.run()
```

# Flask Project Structure

The reason I called it entry level is because I am not going to add custom commands or something like that to complicate the main objective.

Our main objective is to define a flask project structure that is highly scalable.

I have myself worked on various back-end frameworks including Laravel, Django, CodeIgniter, etc.

All of these frameworks have a different architecture that they follow. Now let's first understand why people think about the scalability of a web application.





# Flask Blueprint


**Blueprint** provides a very clean way to build a modular domain-driven flask project structure. If you have used Django then you will find it very similar.

## Reasons to use blueprints

There are various reasons to use blueprints in Flask.



- Blueprints can help to split the entire codebase into smaller blocks
- It can help us to design a domain-driven flask project structure
- Increases code re-usability
- Blueprints can have their own set of static files, templates, and custom error pages as well.



```
// Flask folder structure with Blueprint for domain driven design
// codersdiaries.com
app.py
config.py
requirements.txt
static
    css
    js
    img
templates
    layouts
        app.html
        header.html
        footer.html
apps
    __init__.py
    app1
        __init__.py
        views.py
        models.py
        templates
            app1
                index.html

    app2
        __init__.py
        views.py
        models.py
        templates
            app2
                index.html

database
    __init__.py
    database.py
```

