Vihar Kurama Follow
Philomat
May 24 · 13 min read

Flask Web Programming from Scratch

Complete guide for Flask 1.0 from scratch with SQLAlchemy and Postgres.

Choosing a web framework is one of the important and the most frustrating tasks for building dynamic websites. There are more than thousands of web frameworks in several programming languages. We use these web frameworks based on different use cases. For example, Whatsapp uses Erlang Programming Language which is good at concurrency so for applications like Whatsapp we need to choose a framework which is concurrent enough to handle multiple connections. So before starting web programming or building websites, it is essential to have a robust framework which can handle all types of features. In this article, we will learn about one of the most used and famous web frameworks "Flask."

> *A web framework (**WF**) or web application framework (WAF) is a software framework that is designed to support the development of web applications including web services, web resources, and web APIs.*

About & Why Flask?

It is a micro level web framework which is written in Python by Armin Ronacher and several open source contributors. Flask is based on Werkzeug(Utilities and Requests), Jinja 2(Templating).

Here's the link to official *Flask 1.0 Documentation*.

Flask community recently released its stable version "Flask 1.0" which is production ready which has several new features and updates included.

Pros—Flask provides extreme simplicity, flexibility and fine-grained control. It implements bare-bones and leaves the bells and whistles to add-ons or the developers. Routing URLs is simple. Required database connections can be synced explicitly. It's easier to learn and implement and hence could be a good starting point for all the people out there who're craving to start their web development journey.

Cons—Non-Asynchronous. Lack of database and ORM(Object Relational Mapping—It is a programming technique in which a metadata descriptor is used to connect object code to a relational database).

Finally, Flask is the best choice for all the web programmers, to start from the scratch and for building minimalistic web applications. One more advantage of using Flask is you can integrate machine learning algorithms(Python Based) within the functions with ease which makes your applications more intelligent and cognitive.

What can you do with Flask?

From blog applications to cloning facebook/twitter, almost everything is possible in Flask. There are many libraries like flask-sockets, flask-google-maps etc. where you can embed several features in your application. Flask supports MySQL, Postgresql, MongoDB and other few databases and based on the use case we need to choose the most suitable database.

Here are few websites models you can build with Flask:

Blog Applications, Chat Applications, Data Visualisation, Dashboards, REST Applications, Admin Pages, Email-Services.

Getting Started with Flask 1.0

Prerequisites: Before getting started with flask one should be good with Python, if you are learning Python, refer this article on Python or documentation here. Install Python>=3, configured with PIP (Python Package Index) on your computer.

Installing Flask

To install flask all you need to do is just run this single command on your command prompt if you are using windows, terminal if you are using MacOS or Linux.

```
$ pip install flask
```

The following packages will also be installed with this command: Werkzeug, Jinja, MarkupSafe, ItsDangerous, Click which are necessary for your application to be running. We need not worry about this now, and Flask takes care of this at the back.

Writing out first traditional program "Hello World!"

To start our first flask "Hello World" application, here are few steps to follow. Create a new app.py file and follow accordingly.

1. Firstly we need to import Flask Class from flask module which we installed earlier, this Flask class will contain all the methods and attributes that we use in writing our application, so this will be our first line in our program.

```
from flask import Flask
```

2. Secondly, we need to declare a variable containing flask object. This variable will be used for running and configuring our app.

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

> *"__name__" is a special variable in Python. If the source file is executed as the main program, the interpreter sets the __name__ variable to have a value "__main__". If this file is being imported from another module, __name__ will be set to the module's name.*

3. Writing Main Function.

In our main function, we need to run our flask variable, to do that we need to use the app variable which we declared before and use the run method on it. This run method starts a local server in your machine. Usually, the default address is “localhost:5000.” To change the port address, we can pass in a parameter to the run method assigning it to any desired port number. We need to restart our server whenever we change the code in our application, to overcome that we can use debug parameter given to run method and set it to “True”. Now, whenever if there are any changes made to the source code the server automatically restarts. The debugger is also used to track the errors if there’re any. Below is the code snippet of main function.

```
if __name__ =="__main__":
    app.run(debug=True, port=8080)
```

4. Routing

We then use the **route() decorator** on the app variable to tell Flask what URL should trigger our function. The desired URL pattern should be written in the string to the route decorator. The function which should be triggered by the route will be declared under it. Below is the code snippet of how to declare a route.

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

> *“/” is the home route, as soon as you run your server it triggers first.*

5. Running the Flask application.

Once all the above steps are complete, now our source code looks like this.

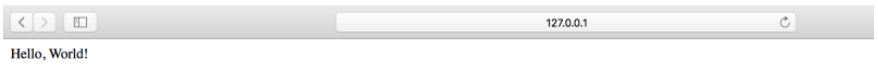
```
from flask import Flask
app = Flask(__name__)

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

```

Running flask application is similar to how you execute your python programs.Once the server starts running you can see the below information.

```
$ python app.py
```

```
* Serving Flask app "app" (lazy loading)
* Environment: production
* Debug mode: on
* Running on http://127.0.0.1:8080/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 770-937-705
```

Now, open your web browser and hit the URL: *http://127.0.0.1:8080/* you can see "Hello World!" printed there.



Hello, World!

Flask Templating (Jinja 2)

So, how to insert plain HTML code into a Flask Application?

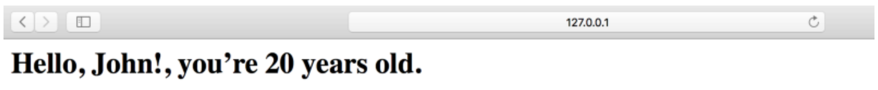
Here we're creating a dictionary comprising of username and age and later in the HTML code those values are being extracted.

Code snippet explaining templating.

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


@app.route('/greet')
def greet():
    user = {'username': 'John', 'age': "20"}
    return '''
<html>
    <head>
        <title>Templating</title>
    </head>
    <body>
        <h1>Hello, ''' + user['username'] + '''!, you’re '
```

This gives the output: Hello, John!, you’re 20 years old.

http://127.0.0.1:8080/greet


127.0.0.1

Hello, John!, you’re 20 years old.

This is totally cumbersome if at all HTML code must be changed on a regular basis. Hence, it isn’t really feasible and scalable. This would be better if logic part is separated from the presentation. Because of that, Flask configures the Jinja2 template engine for us automatically. Instead of hard coding HTML into Flask, we can instead insert the HTML file using the render_template() function.

The hierarchy that must be followed for placing the HTML file assuming the .py file is being placed in a module named ‘Apps’

```
Apps folder
/app.py
templates
   |-/index.html
```

So, create a folder named templates and add the HTML file into it.

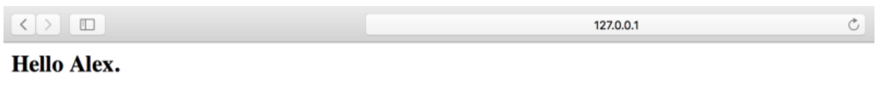
A view/function must return render_template() and the name of html page that should be rendered. Below is code snippet, how you render templates from view.

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

@app.route('/hello')
def hello():
    return render_template('index.html', name="Alex")

```

In HTML page we render the template passed and the variable name is also returned on HTML page.

```
<html>
<body>
  {% if name %}
    <h2>Hello {{ name }}.</h2>
  {% else %}
    <h2>Hello.</h2>
  {% endif %}
```

http://127.0.0.1:8080/hello/alex



Hello Alex.

The above code snippet clearly explains how to insert HTML file comprising of conditional statements, say, 'if' in this case. {{ }} indicates the placeholder for variables.

{% %} is the container for inserting control statements.

Flask Forms and Requests.

Forms are building blocks for every web application, using these forms we take inputs of several categories like username forms and e-mail forms. These help us receive information from users/clients and store them into the database.

Now let’s create a simple bio-data form which takes in few inputs like Name, Age, Email and Hobbies in Flask and renders it on a HTML Page. The HTML code for forms is given below- “bio_form.html”.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h1>Bio Data Form</h1>
    <form action="showbio">
        <label>Username</label>
        <input type="name" name="username"><br>
        <label>Email</label>
        <input type="email" name="email"><br>
```

Now, we use requests library to retrieve the posted data from the form input fields. First, we need to know how the request-response work on the Internet. Whenever a form is submitted it comes under POST method, to load all the pages on to internet it is GET request. We need to create a view to request the information from the forms. As soon as the form is submitted the view compares to POST method and requests the data from the input tags. The index of the form should be equal to the name attribute of input tag.

```
--HTML--
<input type="text" name="username" placeholder="Your Name"/>

--Python--
if request.method == "POST":
    username = request.form['username']
```

Now for the above HTML form let’s declare a view bio_data_form() to store input into Python variables.—*this view renders the above “bio_form.html”*

```
@app.route('/form', methods=['POST', 'GET'])
def bio_data_form():
    if request.method == "POST":
        username = request.form['username']
        age = request.form['age']
        email = request.form['email']
        hobbies = request.form['hobbies']
        return redirect(url_for('showbio',
                                username=username,
                                age=age,
                                email=email,
                                hobbies=hobbies))
    return render_template("bio_form.html")
```

Once all the variables from the form are requested these variables are sent to '**showbio**' view in order to render information in a HTML page. We use "url_for" to redirect to views. So as soon as the request method is POST, the form is submitted the data is sent in the form of arguments, these arguments are fetched from the url and rendered on "**show_bio.html**"

This how the arguments are submitted : http://127.0.0.1:8080/showbiousername=Stark&email=stark07%40gmail.com&hobbies=Play%2C+Code

```
@app.route('/showbio', methods=['GET'])
def showbio():
    username = request.args.get('username')
    age = request.args.get('age')
    email = request.args.get('email')
    hobbies = request.args.get('hobbies')
    return render_template("show_bio.html",
                           username=username,
                           age=age,
                           email=email,
                           hobbies=hobbies)
```

Once these two views are added, this is how your source code should look like,

```
from flask import Flask, render_template, request, url_for
app = Flask(__name__)


@app.route('/')
def hello_there():
    return 'Hello there!'


@app.route('/form', methods=['POST', 'GET'])
def bio_data_form():
    if request.method == "POST":
        username = request.form['username']
        age = request.form['age']
        email = request.form['email']
        hobbies = request.form['hobbies']
        return redirect(url_for('showbio',
                                username=username,
                                age=age,
                                email=email,
                                hobbies=hobbies))
    return render_template("bio_form.html")


@app.route('/showbio', methods=['GET'])
def showbio():
```

We use Jinja 2 for rendering these variables on HTML page, make sure that this function returns the fetched variables. If the function doesn't return the variables they won't be rendered on HTML page.

```
<!DOCTYPE html>
<html>
<head>
    <title>Bio-Data Details</title>
</head>
<body>
    <h1>Bio-Data Details</h1>
    <hr>
    <h1>Username: {{ username }}</h1>
```

Run your application, and visit the : http://127.0.0.1:8080/form

Integrating Flask with Postgres(Database) using Flask-SQLAlchemy

Flask cannot connect to databases directly so we need to have a medium to connect them. The Medium which we use is called as ORM (Object Relational Mapper).

Here we shall explore about Flask-SQLAlchemy, an extension that provides a Flask-friendly wrapper to the popular SQLAlchemy package, which is an ORM. This allow applications to manage a database using high-level entities such as classes, objects and methods instead of tables and SQL. The job of the ORM is to translate the high-level operations into database commands. SQLAlchemy supports a long list of database engines, including the popular MySQL, PostgreSQL and SQLite.

To configure your application with SQLAlchemy and PostgreSQL here are the following steps:

1. **Install Flask-SQLAlchemy and PostgresQL on your machine.**

Install Flask-SQlAlchemy using pip. Run the following command on terminal or command prompt to install Flask-SQLAlchemy.

```
$ pip install flask-sqlalchemy
```

To install PostgreSQL on MAC use Homebrew

```
brew install postgres
```

Or download Postgres application—https://postgresapp.com/

In windows you need to download and install Postgres, here is the link to download Postgres—
https://www.postgresql.org/download/windows/

2. Create a Database

Once you have Postgres installed and running, create a database called “appdb” to use as our local development database:

In your terminal or command prompt use the following command to create Postgres database.

```
$ createdb appdb
```

This command creates a Postgres database in your local machine. If any error occurs, try reinstalling Postgres.

3. Update App Settings.

Configure your source code with all SQlAlchemy and Postgres by setting app.config variables.

Now, we need to link our application with this database using SQLAlchemy as the medium. In our source code add SQLALCHEMY_DATABASE_URI field to the app.config and declare a new variable called db created using an SQLAlchemy object by passing it to the application named app in this case. That object then contains all the functions and helpers from both sqlalchemy and sqlalchemy.orm. Furthermore, it provides a class called Model that is a declarative base which can be used to declare models.

Here is the code for configuring SQlAlchemy and Postgres.

```
app.config['DEBUG'] = True
app.config['SQLALCHEMY_DATABASE_URI']='postgresql://localhos
t/appdb'
SQLALCHEMY_TRACK_MODIFICATIONS = True
db = SQLAlchemy(app)
```

Once you've updated app.py this is how it should look like,

```python
from flask import Flask, request, render_template
from flask_sqlalchemy import SQLAlchemy

# Settings
app = Flask(__name__)
app.config['DEBUG'] = True
app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://loca
db = SQLAlchemy(app)

@app.route('/')
```

These three steps, will let you connect your app with Postgres database.

Now, let's create our first M**odel**. The base class for all your models is called db.Model. It's stored on the SQLAlchemy instance we've to create. Use Column to define a column. The name of the column is the name you assign it .

Below is the code snippet to declare models in flask. This is similar to how we create classes in Python. In this we are declaring a Post Model which has three attributes id(primary_key), title for the post, and post description(post_text).

```python
class Post(db.Model):
    id = db.Column(db.Integer(), primary_key=True)
    title = db.Column(db.String(80), unique=True)
    post_text = db.Column(db.String(255))

    def __init__(self, title, post_text):
        self.title = title
        self.post_text = post_text
```

Add your Model to your source code.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://loca

db = SQLAlchemy(app)


class Post(db.Model):
    id = db.Column(db.Integer(), primary_key=True)
    title = db.Column(db.String(80), unique=True)
    post_text = db.Column(db.String(255))

    def __init__(self, title, post_text):
        self.title = title
        self.post_text = post_text


```

Flask-Migrations

Next migrations must be done to the database in order to keep up with the existing modifications and the changing needs of the application. The second extension that must be used is Flask-Migrate. This extension is a Flask wrapper for Alembic, a database migration framework for SQLAlchemy.

Install flask migrations and flask script(—has the manager class to activate manager commands) package using PIP.

```
$ pip install flask-migrate
$ pip install flask_script
```

To setup migrations in our app, we need to define Migrate class within the app as our base instance. Add the below lines in your source file to configure migrations.

First import flask migrations and Manager class to your source code.

```
from flask_script import Manager
from flask_migrate import Migrate, MigrateCommand
```

Next, configure app settings with migrations.

```
migrate = Migrate(app, db)
manager = Manager(app)
manager.add_command('db', MigrateCommand)
```

To elaborate upon this, we set our config to get our scene—based on the environment variable—created a migrate instance, with app and db as the arguments, and set up a manager command to initialise a Manager instance for our app. Lastly, we added the db command to the manager so that we can run the migrations from the command line.

Now, since our manager app has the whole app and db instances we need to run the replaced app variable which is manager variable in the main function.

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

The configured source code(app.py) now will look like this:

```python
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_script import Manager
from flask_migrate import Migrate, MigrateCommand

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://loca

db = SQLAlchemy(app)
migrate = Migrate(app, db)
manager = Manager(app)
manager.add_command('db', MigrateCommand)


class Post(db.Model):
    id = db.Column(db.Integer(), primary_key=True)
    title = db.Column(db.String(80), unique=True)
    post_text = db.Column(db.String(255))

    def __init__(self, title, post_text):
```

In order to run the migrations to initialize Alembic, use the following command.

```
$ python app.py db init
Creating directory /Users/Vihar/Desktop/flask-
databases/migrations ... done
...
...
...
Generating /Users/Vihar/Desktop/flask-
databases/migrations/alembic.ini ... done
```

Once you run the command, the above information is provided by the manager saying, migrations are successfully created. After you run the database initialisation, you will see a new folder called “migrations” in the project.

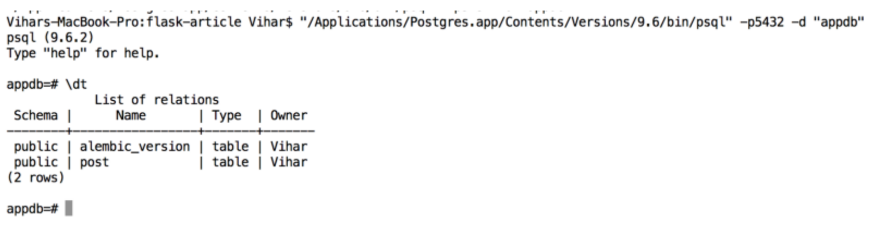
Let’s do our first migration by running the migrate command. Use the following command to update and create tables in your database.

```
$ python app.py db migrate
INFO  [alembic.runtime.migration] Context impl
PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional
DDL.
INFO  [alembic.autogenerate.compare] Detected added table
'post'

Generating /Users/Vihar/Desktop/flask-
databases/migrations/versions/ed3b3a028447_.py ... done
```

With this command your tables will now be created if there aren't and modified in there are any.

Now we'll apply the upgrades to the database using the `db upgrade` command:

```
$ python app.py db upgrade
```



```
Vihars-MacBook-Pro:flask-article Vihar$ "/Applications/Postgres.app/Contents/Versions/9.6/bin/psql" -p5432 -d "appdb"
psql (9.6.2)
Type "help" for help.

appdb=# \dt
           List of relations
 Schema |      Name       | Type  | Owner
--------+-----------------+-------+-------
 public | alembic_version | table | Vihar
 public | post            | table | Vihar
(2 rows)

appdb=#
```

Here, you can see that two row are added in appdb from Postgres shell.

Flask-Postgres Minimal Blog.

We can add information to database from forms using Flask—WTF Extension which is a wrapper around WTForms Package. When working with WTForms we have to define forms as classes first. So, to install Flask-WTF using PIP, the command is

```
$ pip install flask-wtf
```

Your form class should contain all the fields, for the above Post Model, this is how we declare Form Class

```
class Post(db.Model):
    id = db.Column(db.Integer(), primary_key=True)
    title = db.Column(db.String(80), unique=True)
    post_text = db.Column(db.String(255))

    def __init__(self, title, post_text):
        self.title = title
        self.post_text = post_text

# Declaring Flask WTF-Form

class PostForm(FlaskForm):
    title = StringField('Title', validators=
[DataRequired()])
    post_text = StringField('Post_Text',
                             validators=[DataRequired()]
                           )
```

Now we need to declare a view which sends all the form values to the HTML Template.

Here in the below code snippet, we are declaring a new view "add_post" under "/addpost" route.

In this view we are rendering "post_form.html" which has the form input fields. Now in the view, first we need to create a new instance of PostForm so whenever the page is refreshed or reloaded a new instance is created.

Here we are using Flask requests to fetch the information from forms. Whenever something is submitted from a web page, it comes under POST request method. In this view, initially it would be GET which renders post_form.html and later when we submit PostForm, this would call POST method thereby collecting the data from the form.

Once we POST the data to the page a session should be created for the information, in order to add the data to the database. We use db.session.add() which adds the data to the session and db.session.commit() to push the data to the database. Hence, add_post view is defined as

```
@app.route('/addpost', methods=['GET', 'POST'])
def add_post():
    postform = PostForm()
        if request.method == 'POST':
            pf = Post(
                postform.title.data,
                postform.post_text.data,
            )
            db.session.add(pf)
            db.session.commit()
        return redirect(url_for('view_posts'))
    return render_template('post_form.html', postform =
postform)
```

To render flask-wtf forms onto HTML we need to call with the postform variable and thereby pass these as the parameters to model Post which will return on to “post_form.html”

```
<!DOCTYPE html>
<html>
<head>
    <title>Add Post</title>
</head>
<body>
    <h1>Add Post</h1>
    <form action="/posts" method="post">
        <label>Post Title</label>
        {{ postform.title }}
        <label>Post Text</label>
```

After the form is committed to the database we redirect our page to view_posts where we query all the posts and render on a template. Below is the code snippet for ‘view_posts’

```
@app.route('/posts', methods=['GET', 'POST'])
def view_posts():
    posts = Post.query.all()
    return render_template('view_posts.html', posts=posts)
```

All the post are queried in view_posts and returned view_post.html and iterated in HTML page using jinja loops. Here is code snippet for view_post.html which explains looping variables in Jinja 2.

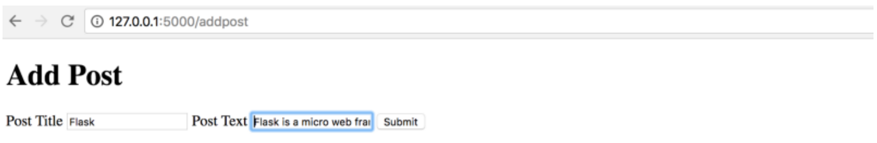
As all the posts are returned from view_posts and rendered on view_posts we iterate over a loop and render on HTML.

```
<!DOCTYPE html>
<html>
<head>
    <title>Posts</title>
</head>
<body>
    <h1>Posts</h1>
    {% for i in posts %}
    <h1>Post Title : {{ i.title }}</h1>
    <p>Description : {{ i.post_text }}</p>
```

The complete source code is given below:

```python
from flask import Flask, request, render_template, url_for
from flask_sqlalchemy import SQLAlchemy
from flask_script import Manager
from flask_migrate import Migrate, MigrateCommand
from flask_wtf import FlaskForm
from wtforms import StringField
from wtforms.validators import DataRequired


app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://loca
app.config.update(dict(
    SECRET_KEY="powerful secretkey",
    WTF_CSRF_SECRET_KEY="a csrf secret key"
))

db = SQLAlchemy(app)
migrate = Migrate(app, db)
manager = Manager(app)
manager.add_command('db', MigrateCommand)


class Post(db.Model):
    __tablename__ = 'post'

    id = db.Column(db.Integer(), primary_key=True)
    title = db.Column(db.String(80), unique=True)
    post_text = db.Column(db.String(255))

    def __init__(self, title, post_text):
        self.title = title
        self.post_text = post_text


# Declaring Flask WTF-Form
class PostForm(FlaskForm):
    title = StringField('Title', validators=[DataRequired(
    post_text = StringField('Post_Text', validators=[DataR


@app.route('/')
def index():
    return "Hello World"

```

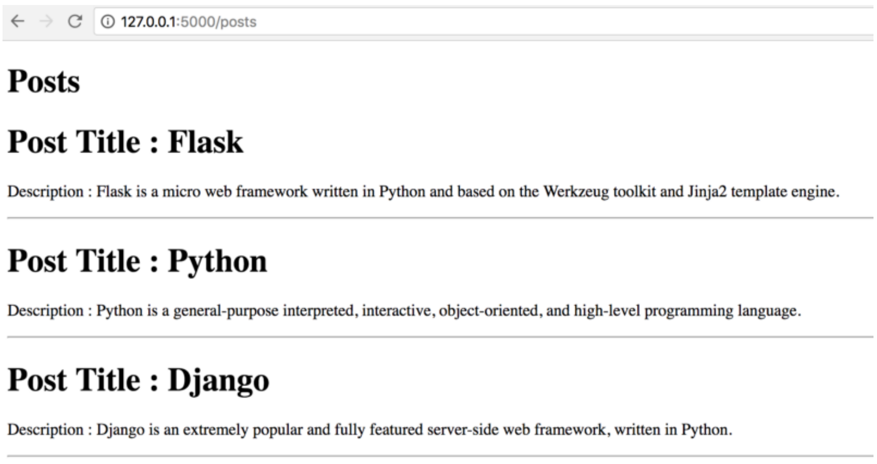
http://localhost:5000/addpost



Add Post

Post Title Flask Post Text Flask is a micro web fra Submit

http://localhost:5000/posts



Posts

Post Title : Flask

Description : Flask is a micro web framework written in Python and based on the Werkzeug toolkit and Jinja2 template engine.

Post Title : Python

Description : Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language.

Post Title : Django

Description : Django is an extremely popular and fully featured server-side web framework, written in Python.

This explains the minimal working of blog application, All the frontend part can be added in the HTML files directly.

Congratulations on learning Flask 1.0, with this you will be able to create seamless and intelligent web applications.

The source-code for the article can be found here.

. . .

Thanks for reading. If you found this story helpful, please click the below 👏 to spread the love.

This article is authored by Vihar Kurama and Samhita Alla.

Stay tuned for more articles on Flask.

Important Links:

Python Programming in 15 min Part 1

About Python

towardsdatascience.com