

# Flask - Python

Flask is back-end web framework, born in 2010, that can browser requests, and user session between requests; route HTTP request to the controllers, evaluate form data, respond to HTML and JS request, and so on.

## First web app with flask

In this session, we will write our first app explained line by line; we will also cover how to set up our environment, what tools to use for development, and how to work with HTML in our app.

### Installation and tools

**Step 1)** In the command window, let's install the virtual environment for python to work with flask. In the command terminal, go to your project folder, or root folder, and type:

```
pip install virtualenv
```

```
mac: sudo pip install virtualenv
```

[pip](#) is the default Python package management tool and helps us install the [virtualenv](#) package.



```
Microsoft Windows [Version 10.0.17134.1365]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Student>cd C:\Users\Student\Desktop\JavaScript

C:\Users\Student\Desktop\JavaScript>pip install virtualenv
Collecting virtualenv
  Downloading https://files.pythonhosted.org/packages/ef/a1/4e170f25211b3851e6be6675061e0c8eae7585d80177a40e9b02d1105d8/virtualenv-20.13.0-py2.py3-none-any.whl (6.5MB)
    100% |#####| 6.6MB 232KB/s
Collecting importlib-metadata>=0.12; python_version < "3.8" (from virtualenv)
  Downloading https://files.pythonhosted.org/packages/a0/a1/b153a0a4caf7a7e3f15c2cd56c7702e2cf3d89b1b359d1f1c5e59d68f4ce/importlib_metadata-4.8.3-py3-none-any.whl
Collecting filelock<4,>=3.2 (from virtualenv)
  Downloading https://files.pythonhosted.org/packages/84/ce/8916d10ef537f3b046843255f9799504aa41862bfa87844b9bdc5361cd/filelock-3.4.1-py3-none-any.whl
Collecting platformdirs<3,>=2 (from virtualenv)
  Downloading https://files.pythonhosted.org/packages/b1/78/dcf8d84d3aab46a9c77260fb47ea5d244806e4dae83aa6fe5d83adb182c/platformdirs-2.4.0-py3-none-any.whl
Collecting distlib<1,>=0.3.1 (from virtualenv)
  Downloading https://files.pythonhosted.org/packages/ac/a3/8ee4f54d5f12e16eeeda6b7df3dfdbda24e6cc572c86ff959a4ce118391b/distlib-0.3.4-py2.py3-none-any.whl (461kB)
    100% |#####| 471kB 2.6MB/s
Collecting importlib-resources>=1.0; python_version < "3.7" (from virtualenv)
  Downloading https://files.pythonhosted.org/packages/24/1b/33e489669a94da3ef4562938cd306e8fa915e13939d7b8277cb5569cb405/importlib_resources-5.4.0-py3-none-any.whl
Collecting six<2,>=1.9.0 (from virtualenv)
  Downloading https://files.pythonhosted.org/packages/d9/5a/e7c31adbe875f2abbb91bd84cf2dc52d792b5a01506781dbcf25c91daf11/six-1.16.0-py2.py3-none-any.whl
Collecting zipp>=0.5 (from importlib-metadata>=0.12; python_version < "3.8"->virtualenv)
  Downloading https://files.pythonhosted.org/packages/bd/df/d4a4974a3e3957fd1cfa3082366d7fff6e428ddb55f074bf64876f8e8ad/zipp-3.6.0-py3-none-any.whl
Collecting typing-extensions>=3.6.4; python_version < "3.8" (from importlib-metadata>=0.12; python_version < "3.8"->virtualenv)
  Downloading https://files.pythonhosted.org/packages/05/e4/baf0031e39cf545f0c9edd5b1a2ea12609b7fcb2d58e118b11753d68cf0/typing_extensions-4.0.1-py3-none-any.whl
Installing collected packages: zipp, typing-extensions, importlib-metadata, filelock, platformdirs, distlib, importlib-resources, six, virtualenv
Successfully installed distlib-0.3.4 filelock-3.4.1 importlib-metadata-4.8.3 importlib-resources-5.4.0 platformdirs-2.4.0 six-1.16.0 typing-extensions-4.0.1 virtualenv-20.13.0 zipp-3.6.0

C:\Users\Student\Desktop\JavaScript>
```

A virtual environment is the way Python isolates full package environments from one another. This means you can easily manage dependencies. Imagine you want to define the minimum necessary packages for a project; a virtual environment would be perfect to let you test and export the list of needed packages.

**Step 2)** Once the first step is completed, type:

```
python -m venv virtual
```

```
mac: python -m virtualenv virtual
```

This step specify which folder you want to install virtual environment to be created in. After this step, you should see virtual folder created under your app root folder.

```
C:\Users\Student\Desktop\JavaScript>python -m venv virtual  
C:\Users\Student\Desktop\JavaScript>_
```

In your root or project folder should look as:

 PythonFormsPractice	1/27/2022 3:26 PM	File folder
 virtual	1/27/2022 3:41 PM	File folder

**Step 3)** (recommended) **activate** will trigger your virtual environment under Scripts folder  
(root folder) > **virtual\Scripts\activate**

```
C:\Users\Student\Desktop\JavaScript>virtual\Scripts\activate  
(virtual) C:\Users\Student\Desktop\JavaScript>
```

**Step 4)** After step 3, point to virtual folder, Scripts, then install flask:

```
(virtual)(root folder) > pip install flask
```

```
Command Prompt
(virtual) C:\Users\Student\Desktop\JavaScript>pip install flask
Collecting flask
  Downloading https://files.pythonhosted.org/packages/8f/b6/b4fdbcb6d01ee20f9cfe81dcf9d3cd6c2f874b996f186f1c0b898c4a59c04/Flask-2.0.2-py3-none-any.whl (95kB)
    100% |████████████████████████████████████████| 102kB 602kB/s
Collecting Werkzeug>=2.0 (from flask)
  Downloading https://files.pythonhosted.org/packages/1e/73/51137805d1b8d97367a8a77cae4a792af14bb7ce58fbd071af294c740cf0/Werkzeug-2.0.2-py3-none-any.whl (288kB)
    100% |████████████████████████████████████████| 296kB 2.6MB/s
Collecting itsdangerous>=2.0 (from flask)
  Downloading https://files.pythonhosted.org/packages/9c/96/26f935afba9cd6140216da5add223a0c465b99d0f112b68a4ca426441019/itsdangerous-2.0.1-py3-none-any.whl
Collecting Jinja2>=3.0 (from flask)
  Downloading https://files.pythonhosted.org/packages/20/9a/e5d9ec41927401e41aea8af6d16e78b5e612bca4699d417f646a9610a076/Jinja2-3.0.3-py3-none-any.whl (133kB)
    100% |████████████████████████████████████████| 143kB 5.1MB/s
Collecting click>=7.1.2 (from flask)
  Downloading https://files.pythonhosted.org/packages/48/58/c8aa6a8e62cc75f39fee1092c45d6b6ba684122697d7ce7d53f64f98a129/click-8.0.3-py3-none-any.whl (97kB)
    100% |████████████████████████████████████████| 102kB 5.1MB/s
Collecting dataclasses; python_version < "3.7" (from Werkzeug>=2.0->flask)
  Downloading https://files.pythonhosted.org/packages/fe/ca/75fac5856ab5cfa51bbbcefa250182e50441074fdc3f803f6e76451fab43/dataclasses-0.8-py3-none-any.whl

(virtual) C:\Users\Student\Desktop\JavaScript>
```

**Step 5)** Open Atom or any text editor, create **main.py** under your App Root Folder. You can also create a subfolder in your root folder to save the **main.py** file. The **main.py** file should have the following lines:

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

@app.route("/")

def hello():
    return "Hello World"

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

Line **from flask import Flask** is importing Flask class from the flask package.  
Line **app = Flask(\_\_name\_\_)** means that the name of our application is **app**. The argument is required and is used to tell the application where to look for resources such as static content and template.

In order to create our "Hello World", we need to tell our Flask instance how to respond when a user tries to access our Web application (using a browser or whatever). For that purpose, Flask has routes. Routes are the way Flask reads a request header and decides which view should respond to that request. It does so by analyzing the path part of the requested URL and finding which route is registered with that path. Therefore we have the line **@app.route("/")**

We use the route decorator to register the **hello** function to the path `"/"`. Every time an app receives a request in which the path is `"/"`, **hello** will respond to that request.

We have the function that will respond the request. Notice that it receives no parameters and responds –with a familiar string. It receives no parameters because the request data, like a submitted form, is accessed through a thread-safe variable called **request**. With regard to the response, Flask can respond to requests in numerous formats. In our example, we respond with a plain string, but we could also respond with a JSON or HTML string.

Line:

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

check whether **main.py** is being called as a script or as a module. If it is as a script, it will run the built-in development server that comes bundled with Flask.

### *Running python file in command window*

Let us run the **main.py** file in the command window:

```
(virtual) C:\Users\Student\Desktop\JavaScript\PythonFormsPractice>main.py  
* Serving Flask app 'main' (lazy loading)  
* Environment: production  
  WARNING: This is a development server. Do not use it in a production deployment.  
  Use a production WSGI server instead.  
* Debug mode: off  
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Just open <http://127.0.0.1:5000/> in your browser to see your app working.

Running **main.py** as a script is usually a very simple and handy setup. Usually, you have Flask-Script to handle calling the development server for you and other setups.

That's pretty much all there is to know about our *"Hello World"* application. One thing our world application lacks is a fun factor. So let's add that; let's make your application fun! Maybe some HTML, CSS, and JavaScript could do the trick here. Let's try that!

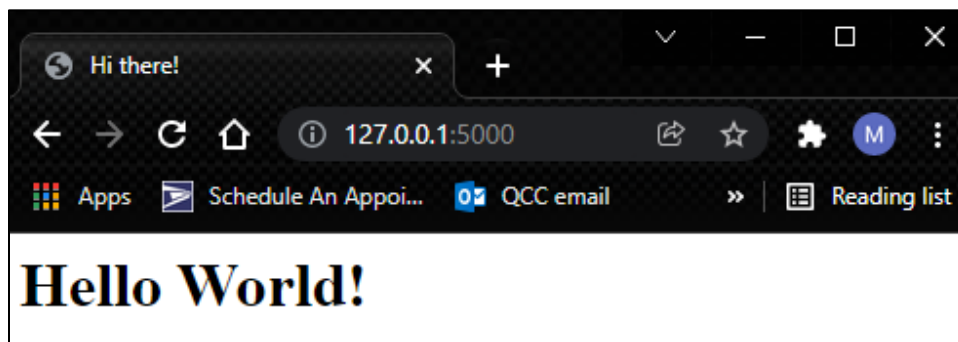
## Serving HTML pages

First, to make our **hello** function respond with HTML, all we have to do is change it like this:

```
def hello():  
    return "<html><head><title>Hi there!</title></head><body><h1>Hello  
World!</h1></body></html>", 200
```

In the preceding example, **hello** is returning a HTML formatted string and a number. The string will be parsed as HTML by default while **200** is an optional HTTP code indicating a successful response. **200** is returned by default.

If we run the main.py again:



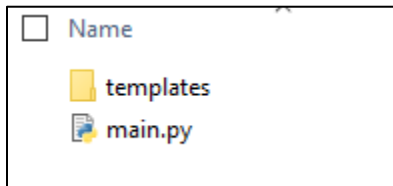
If you refresh your browser with *F5*, you'll notice that nothing has changed. That is why the Flask development server is not reloading when the source changes. That only happens when you run your application in debug mode. So let's do that:

**app.debug=True**

```
from flask import Flask  
app = Flask(__name__)  
  
@app.route("/")  
def hello():  
    return "<html><head><title>Hi there!</title></head><body><h2>Hello World!</h2></body></html>"  
  
if __name__ == "__main__":  
    app.debug=True  
    app.run()
```

Now go to the terminal where your application is running, type **Ctrl + C** then restart the server.

Instead of creating our own HTML code in the **main.py** file, we can also connect the **main.py** to a HTML file. For this, we create a folder called templates and a file called **forms.html** inside it.



After it, we need to import **render\_template** and change the return to **return render\_template("forms.html")**

**render\_template** is capable of loading templates from the **templates** folder (a default for Flask) and you can render it just by returning the output.

```
File Edit View Selection Find Packages Help
main.py
1  from flask import Flask, render_template
2  app = Flask(__name__)
3
4  @app.route("/")
5  def hello():
6      return render_template("forms.html")
7
8  if __name__ == "__main__":
9      app.debug=True
10     app.run()
```

## Working with forms

Have you ever imagined what happens when you fill in a form on a website and click on that fancy **Send** button at the end of it? Well, all the data you wrote—comment, name, checkbox, or whatever—is encoded and sent through a protocol to the server, which then routes that information to the Web application. The Web application will validate the data origin, read the form, validate the data syntactically then semantically, and then decide what to do with it. Do you see that long chain of events where every link might be the cause of a problem? That's forms for you.

Let us have a form with the following information in a HTML file named **index.html**:

### Collecting Weight and Height for BMI

Instruction: Please fill the entires to get poulations statistics on Weight, Height and BMI

In the preceding example, we define a view called `login_view` that accepts `get` or `post` requests; when the request is `post` (we ignore the form if it was sent by a `get` request), we fetch the values for `username` and `passwd`; then we run a very simple validation and change the value of `msg` accordingly.

```
<!--if you want to get something from server you put get-->
<!--if you want to send data to the server you put POST-->
<form class="" action="{{url_for('thankyou')}}" method="POST" >
<!-- name is for python to pull off user's input: give me the value of input what the
name is email_name-->
  <fieldset>
    <legend>Collecting Weight and Height for BMI</legend>
    <p>Instruction: Please fill the entires to get poulations statistics on
    Weight, Height and BMI</p>
    <input type="email" name="email_name" required value="" title="Your Info will
    be safe with us" placeholder="Enter your Email" style="width:300px">
    <input type="number" min="50" max="300" name="height_name" required value=""
    title="Your Info will be safe with us" placeholder="Enter your Height in
    Meter" style="width:200px">
    <input type="number" min="20" max="300" name="weight_name" required value=""
    title="Your Info will be safe with us" placeholder="Enter your Weight in Kg"
    style="width:200px">
    <!--define the role of the button-->
    <button type="submit" name="button">Submit</button>
  </fieldset>
</form>
```

Also create a **thankyou.html** as the returning HTML file:

## Thank you!

You will receive an email with the statistics result soon

```

<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>data gathering web app</title>
  <body>

    <div class="container">
      <br><br><br><br>
      <h1>Thank you!</h1>
      <h2>You will receive an email with the statistics result soon</h2>

    </div>

  </body>
</html>

```

Now, we can create our python code. We need to add the **request** to the library of python in order to use the request method to retrieve the information in the form.

```

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

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

@app.route('/', methods=['POST'])
def thankyou():
    # the methods that handle requests are called views, in flask
    # form is a dictionary like attribute that holds the form data
    if request.method == 'POST':
        email = request.form["email_name"]
        return render_template('thankyou.html')
if __name__ == '__main__':
    app.debug=True
    app.run()

```

### **Testing your code**

to test the code, you can run **main.py** file from the terminal:

```
(virtual) C:\Users\Student\Desktop\MicroCredential>main.py
```

Once you fill up the form and click on the Submit button, you should be able to see the information in the terminal window



## Collecting Weight and Height for BMI

Instruction: Please fill the entires to get poulations statistics on Weight, Height and BMI

```
127.0.0.1 - - [05/Feb/2022 07:51:45] "GET / HTTP/1.1" 200 -  
127.0.0.1 - - [05/Feb/2022 07:51:45] " " 404 -  
ImmutableMultiDict([('email_name', 'hwu@qcc.cuny.edu'), ('height_name', '90'), ('weight_name', '80'), ('button', '')])  
127.0.0.1 - - [05/Feb/2022 07:53:38] "POST / HTTP/1.1" 200 -  
127.0.0.1 - - [05/Feb/2022 07:53:38] " " 404 -
```

## Insert Data into Database from HTML

Now is time to route our data from the HTML form to our database, for this, we import a Python SQL toolkit SQLAlchemy to our **main.py** file and our root project. SQLAlchemy is written in Python and gives full power and flexibility of SQL to an application developer. It is an open source and cross-platform software released under MIT license. It is an amazing library for working with relational databases. It was made by the Pocomo Team, the same folks that brought you Flask, and is considered "The Facto" Python SQL library. It works with SQLite, Postgres, MySQL, Oracle, and all SQL databases, which comes with compatible drivers.

### Flask-SQLAlchemy

Flask-SQLAlchemy is a thin extension that wraps SQLAlchemy around Flask. It allows you to configure the SQLAlchemy engine through your configuration file and binds a session to each request, giving you a transparent way to handle transactions. Let's see how to do all that. First, let's make sure we have all the necessary packages installed.

**Step 1)** First, let us install python psycopg2 to our root folder with the virtual environment loaded, run:

**(virtual) root folder > pip install psycopg2**

```
(virtual) C:\Users\Student\Desktop\MicroCredential>pip install psycopg2  
Requirement already satisfied: psycopg2 in c:\users\student\desktop\microcre  
dential\virtual\lib\site-packages  
  
(virtual) C:\Users\Student\Desktop\MicroCredential>
```

**Step 2)** After it, we can install Flask-SQLAlchemy:

**(virtual) root folder > pip install Flask-SQLAlchemy**

```
(virtual) C:\Users\Student\Desktop\MicroCredential>pip install Flask-SQLAlchemy
```

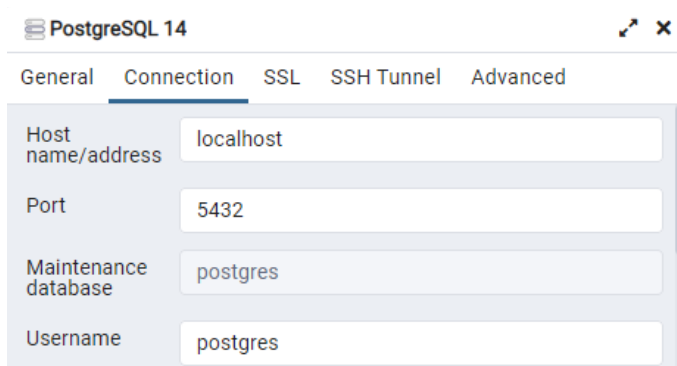
**Step 3a)** Add the sqlalchemy package to **main.py** file:

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

**Step 3b)** Add the database link to your file:

```
app.config [ 'SQLALCHEMY_DATABASE_URI' ]='postgresql://DB_USER:PASSWORD@HOST/DATABASE '
```

using the information in our postgresql:



The screenshot shows the 'PostgreSQL 14' connection settings window. The 'Connection' tab is selected. The fields are as follows:

Field	Value
Host name/address	localhost
Port	5432
Maintenance database	postgres
Username	postgres

we can fill up the **app.config** line as:

```
app.config[ 'SQLALCHEMY_DATABASE_URI' ]='postgresql://postgres:123@localhost/demoDB '
```

```
db = SQLAlchemy(app) → initiate the extension
```

**Step 3c)** Define our db model, let us to create a table named **data** into our database. In python, we can use the same name as our table, but need to type the name as **Data** because it is class.

```
# Define db model
class Data(db.Model):
    __tablename__ = "data"

    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(120), unique = True)
    height = db.Column(db.Integer)
    weight = db.Column(db.Integer)

    def __init__(self, email, height, weight):
        self.email = email
        self.height = height
        self.weight = weight

# Define db model
class Data(db.Model):
    __tablename__ = "data"
    id=db.Column(db.Integer, primary_key=True)
    email_ = db.Column(db.String(120), unique = True)
    height_ = db.Column(db.Integer)
    weight = db.Column(db.Integer)
```

You may have also noticed that `Data` extends `db.Model` which is your ORM<sup>1</sup> model base class. All your models should extend it in order to be known by `db`, which encapsulates our engine and holds our request aware session.

**Step 3d)** Now, let create a function that will initialize the transfer of data from the HTML form to the PostgreSQL database:

```
app.config['SQLALCHEMY_DATABASE_URI']='postgresql://postgres:123@localhost/demoDB'
db = SQLAlchemy(app)

# Define db model
class Data(db.Model):
    __tablename__ = "data"
    id=db.Column(db.Integer, primary_key=True)
    email_ = db.Column(db.Integer)
    height_ = db.Column(db.Integer)
    weight_ = db.Column(db.Integer)

    def __init__(self, email_, heighth_, weighth_):
        self.email_ = email_
        self.heighth_ = height_
        self.weight_ = weight_
```

---

<sup>1</sup> Object–relational mapping (ORM, O/RM, and O/R mapping tool) in computer science is a programming technique for converting data between incompatible type systems using object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language.

```
def __init__(self,email_, heigth_, weigth_):
    self.email_ = email_
    self.heigth_ = height_
    self.weight_ = weigth_
```

**Step 3e)** Add the method to fold form data

```
def thankyou():
    # the methods that handle requests are called views, in flask
    # form is a dictionary like attribute that holds the form data
    if request.method == 'POST':
        email =request.form["email_name"]
        height = request.form["height_name"]
        weigth = request.form["weight_name"]
```

```
if request.method == 'POST':
    email =request.form["email_name"]
    height = request.form["height_name"]
    weigth = request.form["weight_name"]
```

**Step 3f)** Print form request data

```
def thankyou():
    # the methods that handle requests are called views, in flask
    # form is a dictionary like attribute that holds the form data
    if request.method == 'POST':
        email =request.form["email_name"]
        height = request.form["height_name"]
        weigth = request.form["weight_name"]
        print(request.form)
        return render_template('thankyou.html')
```

**Step 3f)** The last step in our Python code, is to program add the data into our Data object and commit our changes to our program:

```
data=Data(email,height,weight)
db.session.add(data)
db.session.commit()
```

**Step 4)** Go back to Terminal, then type **python** to enter a line to import data from the app to the database

```
(virtual) C:\Users\Student\Desktop\MicroCredential>python
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

**Step 4)** Add the line in the command window to import from **main** python file into our database:

```
>>> from main import db
```

**Step 5)** Now, we can create all the elements in our python file, with the values collected from our form, in our database → **db.create\_all()**

**Step 6)** Finally, now is time to test your database and see a table data, with column id, email, height, weight, created.