Installation

Flask is a small framework by most standards—small enough to be called a “microframework,” and small enough that once you become familiar with it, you will likely be able to read and understand all of its source code. But being small does not mean that it does less than other frameworks.

Flask was designed as an extensible framework from the ground up; it provides a solid core with the basic services, while extensions provide the rest. Because you can pick and choose the extension packages that you want, you end up with a lean stack that has no bloat and does exactly what you need.

Flask has three main dependencies:

The routing, the debugging, the Web Server Gateway Interface (WSGI) subsystems come from Werkzeug;
The template support is provided by Jinja2;
The command-line integration comes from Click.

These dependencies are all authored by Armin Ronacher, the author of Flask.

Flask has no native support for accessing databases, validating web forms, authenticating users, or other high-level tasks.

These and many other key services most web applications need are available through extensions that integrate with the core packages. As a developer, you have the power to pick the extensions that work best for your project, or even write your own if you feel inclined to. This is in contrast with a larger framework, where most choices have been made for you and are hard or sometimes impossible to change.

INSTALLING THE REQUIRED SOFTWARE

If you will learn the flask framework, you must install the necessary software tools. The only requirement is a computer with Python installed.

Creating the Application Directory

Create a folder name it flask-ml

Virtual Environments

Now that you have the application directory created, it is time to install Flask. The most convenient way to do that is to use a virtual environment. A virtual environment is a copy of the Python interpreter into which you can install packages privately, without affecting the global Python interpreter installed in your system. Virtual environments are very useful because they prevent package clutter and version conflicts in the system’s Python interpreter. Creating a virtual environment for each project ensures that applications have access only to the packages that they use, while the global interpreter remains neat and clean and serves only as a source from which more virtual environments can be created. As an added benefit, unlike the system-wide Python interpreter, virtual environments can be created and managed without administrator rights.

Creating a Virtual Environment with Python 2

Python 2 does not have a venv package. In this version of the Python interpreter, virtual environments are created with the third-party utility virtualenv. Make sure your current directory is set to flask-ml, and then use one of the following two commands, depending on your operating system. On Microsoft Windows, make sure you open a command prompt window using the "Run as Administrator" option, and then run this command:

```
$ pip install virtualenv
```

The virtualenv command takes the name of the virtual environment as its argument. Make sure your current directory is set to flask-ml, and then run the following command to create a virtual environment called venv:

```
$ virtualenv venv
New python executable in venv/bin/python2.7
Also creating executable in venv/bin/python Installing setuptools, pip, wheel...done.
```

A subdirectory with the venv name will be created in the current directory, and all files associated with the virtual environment will be inside it.

Creating a Virtual Environment with Python 3

The creation of virtual environments is an area where Python 3 and Python 2 interpreters differ. With Python 3, virtual environments are supported natively by the venv package that is part of the Python standard library.If you are using the stock Python 3 interpreter on an Ubuntu Linux system, the standard venv package is not installed by default. To add it to your system, install the python3-venv package as follows: $ sudo apt-get install python3-venv

The command that creates a virtual environment has the following structure:

```
$ python3 -m venv virtual-environment-name
```

The -m venv option runs the venv package from the standard library as a standalone script, passing the desired name as an argument. You are now going to create a virtual environment inside the flask-ml directory. A commonly used convention for virtual environments is to call them venv, but you can use a different name if you prefer. Make sure your current directory is set to flask-ml, and then run this command:

```
$ python3 -m venv venv
```

After the command completes, you will have a subdirectory with the name venv inside flask-ml, with a brand-new virtual environment that contains a Python interpreter for exclusive use by this project.

On Windows 10 is a little bit tricky

Creating a virtual environment on Windows 10 with Python3 venv module. Once you had completed the installation of Python 3 on Windows 10, you will be ready to create the virtual environment for your application. In order to do so, open up a command prompt window and type the following command:

```
python -m venv %systemdrive%%homepath%\flask-ml
```

After the command completes, you will find the flask-ml directory inside your home directory. Inside the flask-ml, you will find the Python artefacts for your isolated virtual environment.

Activating your Python 3 virtual environment on Windows 10. Before you can run your Python 3 application inside of your Python 3 virtual environment, you will need to activate it. In order to activate your virtual environment, you will need to run the activate.bat script located inside your virtual environment directory.

For example, to activate the virtual environment inside my-venv, you can run the following command in your command prompt window:

```
%systemdrive%%homepath%\flask-ml \Scripts\activate.bat
```

After the activate.bat script had ran, you will see the prompt appended with (flask-ml):

To deactivate your Python 3 virtual environment on Windows 10. When you want to get out of your Python 3 virtual environment on Windows 10, you can simply run the following command:

```
deactivate
```

After the virtual environment is deactivated, your command prompt will switch to the global Python 3 environment. In addition, those Python 3 dependencies that you had installed in your virtual environment will not be available.

Installing Python Packages with pip

Python packages are installed with the pip package manager, which is included in all virtual environments. Like the python command, typing pip in a command prompt session will invoke the version of this tool that belongs to the activated virtual environment. To install Flask into the virtual environment, make sure the venv virtual environment is activated, and then run the following command:

```
(venv) $ pip install flask
```

When you execute this command, pip will not only install Flask, but also all of its dependencies. You can check what packages are installed in the virtual environment at any time using the pip freeze command:

```
(venv) $ pip freeze
click==6.7
Flask==0.12.2
itsdangerous==0.24
Jinja2==2.9.6
MarkupSafe==1.0
Werkzeug==0.12.2
```

The output of pip freeze includes detailed version numbers for each installed package. The version numbers that you get are likely going to be different from the ones shown here. You can also verify that Flask was correctly installed by starting the Python interpreter and trying to import it:

```
(venv) $ python
>>> import flask
 >>>
```

If no errors appear, you can congratulate yourself: you are ready for the next step, where you will write your first web application.

Basic Application Structure

Initialization

All Flask applications must create an application instance. The web server passes all requests it receives from clients to this object for handling, using a protocol called Web Server Gateway Interface (WSGI, pronounced “wiz-ghee”). The application instance is an object of class Flask, usually created as follows:

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

The only required argument to the Flask class constructor is the name of the main module or package of the application. For most applications, Python’s __name__ variable is the correct value for this argument. The __name__ argument that is passed to the Flask application constructor is a source of confusion among new Flask developers. Flask uses this argument to determine the location of the application, which in turn allows it to locate other files that are part of the application, such as images and templates

Routes and View Functions

Clients such as web browsers send requests to the web server, which in turn sends them to the Flask application instance. The Flask application instance needs to know what code it needs to run for each URL requested, so it keeps a mapping of URLs to Python functions. The association between a URL and the function that handles it is called a route. The most convenient way to define a route in a Flask application is through the app.route decorator exposed by the application instance. The following example shows how a route is declared using this decorator:

```
@app.route('/')
def index():
    return '<h1>Hello World!</h1>'
```

Note: Decorators are a standard feature of the Python language. A common use of decorators is to register functions as handler functions to be invoked when certain events occur.

The previous example registers function index() as the handler for the application's root URL. While the app.route decorator is the preferred method to register view functions, Flask also offers a more traditional way to set up the application routes with the app.add_url_rule() method, which in its most basic form takes three arguments: the URL, the endpoint name, and the view function. The following example uses app.add_url_rule() to register an index() function that is equivalent to the one shown previously:

```
def index():
    return '<h1>Hello World!</h1>'

app.add_url_rule('/', 'index', index)
```

Functions like index() that handle application URLs are called view functions. If the application is deployed on a server associated with the www.example.com domain name, then navigating to http://www.example.com/ in your browser would trigger index() to run on the server. The return value of this view function is the response the client receives. If the client is a web browser, this response is the document that is displayed to the user in the browser window. A response returned by a view function can be a simple string with HTML content, but it can also take more complex forms.

If you pay attention to how some URLs for services that you use every day are formed, you will notice that many have variable sections. For example, the URL for your Facebook profile page has the format https://www.facebook.com/<your-name>, which includes your username, making it different for each user. Flask supports these types of URLs using a special syntax in the app.route decorator. The following example defines a route that has a dynamic component:

```
@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, {}!</h1>'.format(name)
```

The portion of the route URL enclosed in angle brackets is the dynamic part. Any URLs that match the static portions will be mapped to this route, and when the view function is invoked, the dynamic component will be passed as an argument. In the preceding example, the name argument is used to generate a response that includes a personalized greeting.

Your First Complete Flask Web Application

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def index():
        return '<h1>Hello World!</h1>'
```

This hello.py application script shown above defines an application instance and a single route and view function, as described earlier.
Flask applications include a development web server that can be started with the flask run command. This command looks for the name of the Python script that contains the application instance in the FLASK_APP environment variable. To start the hello.py application, first make sure the virtual environment you created earlier is activated and has Flask installed in it.

```
(venv) $ set FLASK_APP=hello.py
(venv) $ flask run
* Serving Flask app "hello"
*Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Once the server starts up, it goes into a loop that accepts requests and services them. This loop continues until you stop the application by pressing Ctrl+C.
With the server running, open your web browser and type http://localhost:5000/ in the address bar.
If you type anything else after the base URL, the application will not know how to handle it and will return an error code 404 to the browser—the familiar error that you get when you navigate to a web page that does not exist.

NOTE: The web server provided by Flask is intended to be used only for development and testing.
The Flask development web server can also be started programmatically by invoking the app.run() method. Older versions of Flask that did not have the flask command required the server to be started by running the application's main script, which had to include the following snippet at the end:

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