One Day Microservices with Flask
Lab Manual

Introduction

This lab manual accompanies the *Innovation in Software* course "*Develop a Microservice with Python Flask in One Day.*"

Operating Systems

All the labs in this manual have been tested in both a Windows 10 and Linux environment. Since most of the work is done at the command line and since macOS uses a Linux shell, these labs also should work without issue in a macOS environment.

Specific operating systems differences between Windows and Linux/Mac – like setting environment variables – are mentioned when appropriate.

Links to standard documentation for the tools used in the lab are provided in the labs.

Lab One: Setting up the Environment

This lab walks through the basic steps needed to set up a Python environment used to develop the Flask microservice.

Notes on Python Versions: At the time this lab manual was prepared, version 3.10 of Python had been recently released. Historically, it has taken several months after an initial minor version release to review and fix any possible broken dependencies with various Python libraries and packages. This manual uses the Python 3.9 release and can also be used for 3.10 with some minor modifications to the this first lab. Specifically, some of the packages installed in this lab may already be part of the 3.10 Python installation.

1.1 Installing Python

Python may already be installed on your system. If it is already version 3.7 or higher, then you are good to go. If you are using 3.10, there is a small risk that some of the dependencies between Python and various libraries may be broken if you are taking the course before 2022.

To check if you have Python installed, open a shell or command window or PowerShell and check “python --version”

```
C:\Flask>python --version
Python 3.9.8
```

If you have Python installed and you get the “application not found” or some other similar message when you try the above command, ensure that the Python installation is on your path variable.

If you have Python 2 and Python 3 both installed, you can either uninstall Python 2 or make the appropriate aliasing of the python3 command so that it is used instead of Python 2 when you type the command “python”.

If you need to run both Python 2 and Python 3, the following resources will be of assistance.

https://www.freecodecamp.org/news/manage-multiple-python-versions-and-virtual-environments-venv-pyenv-pyvenv-a29fb00c296f/

https://levelup.gitconnected.com/how-to-install-and-manage-multiple-python-versions-on-windows-10-c90098d7ba5a

https://medium.datadriveninvestor.com/how-to-install-and-manage-multiple-python-versions-on-linux-916990dabe4b

https://chamikakasun.medium.com/how-to-manage-multiple-python-versions-in-macos-2021-guide-f86ef81095a6

Linux and Mac

Python is most easily installed using the appropriate package manager, such as yum/dnf, apt, homebrew etc.

Additional resources:

https://docs.python-guide.org/starting/install3/linux/

https://docs.python.org/3/using/mac.html

Windows 10

The Windows installer for Python 3.9 can be downloaded from here:

https://www.python.org/ftp/python/3.9.8/python-3.9.8-amd64.exe

Again, the latest version is not recommended until the first update is released, probably in the first quarter of 2022.

When you are installing Python for the first time, there will be a checkbox in the installation dialog box to add Python to the path. Check this box.

1.2 Installing an IDE (optional)

There is no required IDE for the class, you are free to use whatever Python IDE you currently user or prefer to use.

If you don't have an IDE available, the one that will be used in class is the free IDE called *Virtual Studio Code"* which can be downloaded for all operating systems from here:

https://code.visualstudio.com/

Install the IDE using the defaults as presented by the installers

1.3 Configuring Visual Studio Code

To make working with Python easier, the VSC community has provided plugins to support various programming languages to support syntax highlighting, language support features and pretty formatting.

It is recommended that if you are using VSC, that you install the Python extension as shown on the next page. The extension is the PyLance Intellisense – the one with over 40 million installs.

1.4 Installing Pip

Pip should already be installed because of installing Python 3. To confirm it is present on your system, execute the command shown below.

```
C:\Flask>pip --version
pip 21.2.4 from C:\Users\micro\AppData\Local\Programs\Python\Py-
thon39\lib\site-packages\pip (python 3.9)
```

If pip is not installed on your system, or if you have pip installed for Python 2, then you will need to install pip from here

https://pip.pypa.io/en/stable/

1.5 Virtual Environments

By default, every project you create in Python downloads the site packages (third party libraries) that are not part of the standard Python library, such as the Flask package.

The problem is that different projects may require different versions of these site packages which means that the installations done by one project may overwrite the installations of an existing project which would cause the existing project to break.

A virtual environment is an isolated workspace for a Python project. Each virtual environment has its own set of installed site packages so that two projects may have install different versions of site packages without conflict.

There are several different utilities for constructing python virtual environments. Two of them will be introduced in this lab and you are free to use the one you prefer, or even a different utility that you might already be comfortable with.

1.6 venv

This utility is part of the Python 3 core package.

Creating a virtual environment

To create a virtual environment in a directory (here we call it lab1), execute the following command: “python -m venv lab1”

```
C:\Flask>python -m venv lab1

C:\Flask>cd lab1

C:\Flask\lab1>dir
 Volume in drive C has no label.
 Volume Serial Number is 2451-F407

 Directory of C:\Flask\lab1

2021-11-13  12:48 PM    <DIR>          .
2021-11-13  12:48 PM    <DIR>          ..
2021-11-13  12:48 PM    <DIR>          Include
2021-11-13  12:48 PM    <DIR>          Lib
2021-11-13  12:48 PM               117 pyvenv.cfg
2021-11-13  12:48 PM    <DIR>          Scripts
               1 File(s)            117 bytes
               5 Dir(s)  277,052,010,496 bytes free
```

The command creates a directory tree and populates it to support the virtual environment.

It is recommended that you take a few minutes to look at the contents of these directories and files to get a peek under the hood at what venv has set up.

Before you can use the virtual environment, it must be *activated*. All this means is that the path variable is temporarily changed to use the virtual environment instead of the system paths.

To activate the environment in Unix like environments. Execute the command below in a bash shell:

```
$ source lab1/bin/activate
```

To activate in Windows, run the batch command as shown:

```
C:\Flask\lab1>Scripts\activate.bat

(lab1) C:\Flask\lab1>
```

The prompt has changed to show that the virtual environment lab1 has been activated. All the commands that we execute now will be re-routed to that virtual environment as we can see from the modified PATH variable.

```
(lab1) C:\Flask\lab1>echo %PATH%

C:\Flask\lab1\Scripts;C:\Program Files (x86)… and so on
```

To deactivate the virtual environment just run the “deactivate” command from inside the virtual environment. This works for both Unix like systems and Windows. Note that the prompt has changed to show we are no longer using the virtual environment

```
(lab1) C:\Flask\lab1>deactivate

C:\Flask\lab1>
```

For more details, refer to the documentation at:

https://docs.python.org/3/library/venv.html

And the tutorial at:

https://docs.python.org/3/tutorial/venv.html

1.7 Pipenv

An alternative approach to Python virtual environments is pipenv which is a more robust sort of virtual environment for more complex and demanding environments. A comparison of pipenv and venv is beyond the scope of this material. Just use whichever one you prefer for this course.

The Pipenv documentation is located at:

https://pypi.org/project/pipenv/

First, install pipenv

```
C:\Flask>pip install pipenv

Collecting pipenv
. . .
Successfully installed backports.entry-points-selectable-1.1.1 cer-
tifi-2021.10.8 distlib-0.3.3 filelock-3.3.2 pipenv-2021.11.9 plat-
formdirs-2.4.0 six-1.16.0 virtualenv-20.10.0 virtualenv-clone-0.5.7
```

To create a virtual environment using Python 3, create a directory for your project, change to it and execute the command “pipenv --three”. The “three” parameter means that the project will use python 3.

A full list of the options for creating a virtual environment are at the documentation link cited previously

```
C:\Flask>mkdir lab1b

C:\Flask>cd lab1b

C:\Flask\lab1b>pipenv --three
Creating a virtualenv for this project...
Pipfile: C:\Flask\lab1b\Pipfile
Using C:/Users/micro/AppData/Local/Programs/Python/Python39/py-
thon.exe (3.9.8) to create virtualenv...

. . .

Successfully created virtual environment!
Virtualenv location: C:\Users\micro\.virtualenvs\lab1b-Qgg1eujC
Creating a Pipfile for this project...

C:\Flask\lab1b>
```

The virtual environment is “activated” by creating a new virtual shell in the environment using the shell command:

```
C:\Flask\lab1b>pipenv shell
Launching subshell in virtual environment...
Microsoft Windows [Version 10.0.19043.1348]
(c) Microsoft Corporation. All rights reserved.

(lab1b-Qgg1eujC) C:\Flask\lab1b>
```

To exit from the shell, just use “exit” or ctl-D

```
(lab1b-Qgg1eujC) C:\Flask\lab1b> exit

C:\Flask\lab1b>
```

Lab Two: Install and Run Flask

This lab works through the installation of Flask in a virtual environment and how to run a basic “Hello World” Flask app.

2.1 Create a Virtual Environment

You can use whichever virtual environment you choose, but the example in the manual will use pipenv. Create the virtual environment and install Flask using pip as shown below. The commands are the same in Windows and Unix like environments.

```
C:\Flask>mkdir lab2

C:\Flask>cd lab2

C:\Flask\lab2>pipenv --three
Creating a virtualenv for this project...
. . .
C:\Flask\lab2>pipenv shell
Launching subshell in virtual environment...

(lab2-iROZ88Zw) C:\Flask\lab2> pip install Flask
Collecting Flask
. . .
Installing collected packages: MarkupSafe, colorama, Werkzeug,
Jinja2, itsdangerous, click, Flask
Successfully installed Flask-2.0.2 Jinja2-3.0.3 MarkupSafe-2.0.1
Werkzeug-2.0.2 click-8.0.3 colorama-0.4.4 itsdangerous-2.0.1
```

Confirm the details of the installed packages (optional)

```
(lab2-iROZ88Zw) C:\Flask\lab2>pip show Flask
Name: Flask
Version: 2.0.2
Summary: A simple framework for building complex web applications.
Home-page: https://palletsprojects.com/p/flask
Author: Armin Ronacher
Author-email: armin.ronacher@active-4.com
License: BSD-3-Clause
Location: c:\users\micro\.virtualenvs\lab2-iroz88zw\lib\site-pack-
ages
Requires: click, itsdangerous, Jinja2, Werkzeug
Required-by:

(lab2-iROZ88Zw) C:\Flask\lab2>pip show Werkzeug
Name: Werkzeug
Version: 2.0.2
Summary: The comprehensive WSGI web application library.
Home-page: https://palletsprojects.com/p/werkzeug/
Author: Armin Ronacher
Author-email: armin.ronacher@active-4.com
License: BSD-3-Clause
Location: c:\users\micro\.virtualenvs\lab2-iroz88zw\lib\site-pack-
ages
Requires:
Required-by: Flask

(lab2-iROZ88Zw) C:\Flask\lab2>pip show Jinja2
Name: Jinja2
Version: 3.0.3
Summary: A very fast and expressive template engine.
Home-page: https://palletsprojects.com/p/jinja/
Author: Armin Ronacher
Author-email: armin.ronacher@active-4.com
License: BSD-3-Clause
Location: c:\users\micro\.virtualenvs\lab2-iroz88zw\lib\site-pack-
ages
Requires: MarkupSafe
Required-by: Flask
```

2.2 Create the Hello World App

Create a file called Hello.py with the following contents in your IDE.

```
from flask import Flask

app = Flask(__name__ )

@app.route("/")
def hello():
    return "hello world from lab2"
```

For Flask to know what application to run and how to run it, we need to set several environment variables

1 FLASK_APP – which will be set to the file name that contains the code to run
2 FLASK_ENV – which tells Flask what sort of environment it is running in (eg. Production, development, etc.)

If you try to run flask and you get an error like this:

```
(lab2-iROZ88Zw) C:\Flask\lab2>flask run

Error: Could not locate a Flask application. You did not provide the
"FLASK_APP" environment variable, and a "wsgi.py" or "app.py" module
was not found in the current directory.
```

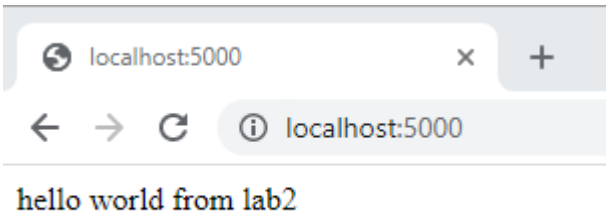
This means that Flask did not know what to run. If there is no FLASK_APP defined, then Flask looks for a file called “app.py” to run or a file called “wsgi.py” to run. If none of these are found, the error is displayed.

2.3 Running the App

Setting the FLASK_APP to “Hello.py” and running flask produces the following output

```
C:\Flask\lab2>set FLASK_APP=Hello.py

## In a bash shell use “export FLASK_APP=Hello.py

(lab2-iROZ88Zw) C:\Flask\lab2>flask run
 * Serving Flask app 'Hello.py' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a produc-
tion deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Pointing a browser to the URL listed shows the WEB service is up and running



However, we still get the warning that we are using a development server in a production environment. This is because Flask assumes that, unless told otherwise in the FLASK_ENV variable that we are in a production environment.

Note that when we ran Flask, it reported at start-up that it was running in a production environment

If we shut down the app and then tell Flask we are in a development environment and restart, the warning goes away.

```
(lab2-iROZ88Zw) C:\Flask\lab2>set FLASK_ENV=development

(lab2-iROZ88Zw) C:\Flask\lab2>flask run

 * Serving Flask app 'Hello.py' (lazy loading)
 * Environment: development
 * Debug mode: on
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 115-574-610
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

A couple of things to try:

1 You can change the port of your application by using the –port=xxxx option
2 You can change the IP address of the app by using –host=x.x.x.x

For example, using 0.0.0.0 publicly available, which means the app is usually using the IP address of the machine Flask is running on (specifically your machine's current IP on your network) if you try this.

```
(lab2-iROZ88Zw) C:\Flask\lab2>flask run --host=0.0.0.0 --port=8080
 * Serving Flask app 'Hello.py' (lazy loading)
 * Environment: development
 * Debug mode: on
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 115-574-610
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a produc-
tion deployment.
 * Running on http://192.168.1.133:8080/ (Press CTRL+C to quit)
```

2.4 Debug Mode

By enabling debug mode, the application will automatically reload whenever the code changes. However, because the debug feature is also a security vulnerability, it can only be used in development mode, not in production mode.

For this section of the lab, ensure that you are in development mode and start the application:

```
(lab2-iROZ88Zw) C:\Flask\lab2>set FLASK_APP=Hello.py

(lab2-iROZ88Zw) C:\Flask\lab2>set FLASK_ENV=development

(lab2-iROZ88Zw) C:\Flask\lab2>flask run
 * Serving Flask app 'Hello.py' (lazy loading)
 * Environment: development
 * Debug mode: on
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 115-574-610
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Confirm that the app is running as it should.

Using your IDE and make the change below, or an equivalent to the code

```
from flask import Flask

app = Flask(__name__ )

@app.route("/")
def hello():
    return "hello world from lab2 in debug mode"
```

As soon as the file is saved, you should see something like this:

```
(lab2-iROZ88Zw) C:\Flask\lab2>flask run
 * Serving Flask app 'Hello.py' (lazy loading)
 * Environment: development
 * Debug mode: on
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 115-574-610
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [13/Nov/2021 17:50:58] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [13/Nov/2021 17:50:58] "GET /favicon.ico HTTP/1.1" 404
-
 * Detected change in 'C:\\Flask\\lab2\\Hello.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 115-574-610
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [13/Nov/2021 17:51:29] "GET / HTTP/1.1" 200 -
```

Check the browser to check the changes. You may need to reload the page.

Lab Three: Basic Routing

In this lab, the endpoints for different URIs are created as a starting point for a web app.

3.1 Set Up the Environment

Just like the last lab, create a working directory, set up your virtual environment, install flask and start the shell.

Then set the environment variables as you did in the last lab.

```
C:\Flask>mkdir lab3

C:\Flask>cd lab3

C:\Flask\lab3>pipenv --three

C:\Flask\lab3>pip install Flask

C:\Flask\lab3>pipenv shell

(lab3-IcaXH67J) C:\Flask\lab3>set FLASK_APP=MyApp.py

(lab3-IcaXH67J) C:\Flask\lab3>set FLASK_ENV=development
```

One of the changes we will be making as well is to return snippets of HTML instead of just strings to be displayed in the browser.

3.2 Basic Routing App

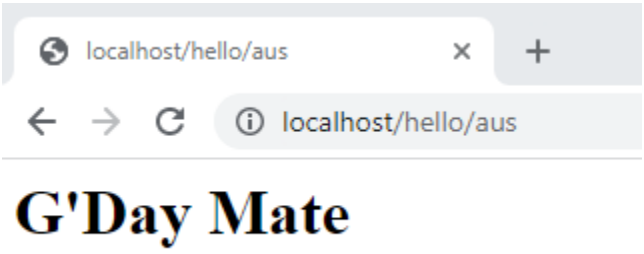
In your IDE, create the MyApp.py file with the following code

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def index():
    return "<h1>Welcome to my site</h1>"

@app.route("/hello")
def hello():
    return "<h1>Hello World</h1>"

@app.route("/hello/aus")
def aussie():
    return "<h1>G'Day Mate</h1>"

@app.route("/goodbye")
def goodbye():
    return "<h1>Seeya</h1>"
```

Run the application on port 80 so don’t have to keep typing in the port number.

```
(lab3-IcaXH67J) C:\Flask\lab3>flask run --port=80
 * Serving Flask app 'MyApp.py' (lazy loading)
 * Environment: development
 * Debug mode: on
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 132-840-729
 * Running on http://127.0.0.1:80/ (Press CTRL+C to quit)
```

Once it is running confirm that all of the URLs resolve to the correct endpoints



3.3 Processing GET and PUT Requests

To start this section, reset the FLASK_APP variable as follows.

```
(lab3-IcaXH67J) C:\Flask\lab3>set FLASK_APP=MyApp2.py
```

In your IDE, add the code for the MyApp2,py. Note the added from statement at the top of the file.

```
from flask import Flask
from flask import request

app = Flask(__name__)


@app.route("/")
def index():
    return "<h1>Welcome to my site</h1>"

@app.route("/user",methods =['GET','POST'])
def hello():
    if request.method == 'POST':
        return "Processed POST request"
    else:
        return "Processed GET request"
```

Run this in the terminal as before. Use port 80 make things a bit easier to see.

```
(lab3-IcaXH67J) C:\Flask\lab3>flask run --port=80
 * Serving Flask app 'MyApp2.py' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a produc-
tion deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:80/ (Press CTRL+C to quit)
```

Open another bash shell or Windows command window and use curl to execute a GET request and a POST request. Do not use the name ‘localhost’ for the POST method or it will not work, use the 127.0.0.1 IP address instead.

The “-d” option in the curl command means that we are sending data (which we will use later) via a POST operation, otherwise curl defaults to a GET operation

```
C:\Flask>curl http://127.0.0.1/user
Processed GET request

C:\Flask>curl -d user=walter http://127.0.0.1/user
Processed POST request
```

Trying the same two curl commands on the “/” endpoint works for the GET operation but fails for the POST operation

```
C:\Flask>curl http://127.0.0.1
<h1>Welcome to my site</h1>

C:\Flask>curl -d user=walter http://127.0.0.1

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>405 Method Not Allowed</title>
<h1>Method Not Allowed</h1>
<p>The method is not allowed for the requested URL.</p>
```

3.4 Processing POST Data

In this final modification of this lab we are going to use the data passed to the app by a POST method to update the global variable username.

In a real application, this post data would be used to update some data store or other persistent storage or be passed to another microservice for other processing, such as checking to see if the new username is a registered user of our system

To start this section, reset the FLASK_APP variable as follows.

```
(lab3-IcaXH67J) C:\Flask\lab3>set FLASK_APP=MyApp23.py
```

In your IDE, add the code for the MyApp3.py.

```
from flask import Flask
from flask import request

app = Flask(__name__)


@app.route("/")
def index():
    return "<h1>Welcome to my site</h1>"

username ="none"

@app.route("/user",methods =['GET','POST'])
def hello():
    global username
    if request.method == 'POST':
       username = request.form['name']
       return "User name updated to " + username
    else:
        return "User name = " + username
```

To test the app, start the app with the “flask run --port=80” command and then open a second window and use the curl commands

```
C:\Flask>curl http://127.0.0.1/user
User name = none

C:\Flask>curl -d name=Walter http://127.0.0.1/user
User name updated to Walter

C:\Flask>curl http://127.0.0.1/user
User name = Walter
```

Lab Four: URI Variables

The setup for this lab is the same as for the others. Set up your virtual environment, activate it or start a shell in it and set the FLASK_APP to “MyApp.py.”

Now that you have worked through a couple of labs, the instructions for this and the following labs will be less verbose since it is assumed you have got the basic setup process for lab sorted out.

In this lab, we will set up a hello web service that responds to either a username or a user id. If we were writing a real application, there would be some significant processing of the input data to do several things:

- Sanitize the data. A raw string might contain some malicious code like an SQL injection or other attack that exploits the fact that most web-based input is treated as character data and not executable script code.
- Determine if the data is the correct type. Just matching the URI router may not be enough – in our example, even though we are expecting integer input, the value -2 is not interpreted as an integer, even though in our business logic, that might be a valid value
- Determine if the data is valid according to some schema, for example US social security numbers have a specific format.

4.1 Enter the Code

In your IDE enter the following code

```
from flask import Flask
from flask import request

app = Flask(__name__)


@app.route("/")
def index():
    return "Welcome to the hello web service"


@app.route("/hello")
def hello():
   return "Hello anonymous user"

@app.route('/hello/<username>')
def helloname(username):
    if username == 'Jack' :
        return "HIT THE ROAD JACK!!!"
    else :
     return 'Hello {}'.format(username)

@app.route('/hello/<int:userid>')
def hellouserid(userid):
    return 'Hello user unit number {:d}'.format(userid)
```

Run as a Flask application like you did in the previous labs. In a separate command window or bash shell, exercise all the URIs

```
C:\Users\micro>curl localhost/hello
Hello anonymous user

C:\Users\micro>curl localhost/hello/Bob
Hello Bob

C:\Users\micro>curl localhost/hello/Jack
HIT THE ROAD JACK!!!

C:\Users\micro>curl localhost/hello/43
Hello, user unit number 43

C:\Users\micro>curl localhost/hello/43.2
Hello 43.2

C:\ Users\micro>curl localhost/hello/-2
Hello -2
```

Lab Five: Templates

For this lab, create your virtual environment as usual, but once you are in the shell or the environment has been activated, create a subdirectory called “templates.”

Put the following HTML files in the templates directory (query.html and response.html)

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8">
    <title>Entry Page</title>
</head>

<body>
    <form action="process" method="post">
        <label for="username">Please enter your name: </label>
        <input type="text" id="username" name="username"><br>
        <input type="submit" value="SEND">
    </form>
</body>

</html>
```

```
<!DOCTYPE html>
<html lang="en">

<head>
<meta charset="utf-8">
    <title>Response Page</title>
</head>

<body>
    <h1>Hello, {{ username }}</h1>
</body>

</html>
```

The code to execute the application is called Hello.py and is located in the main directory of the lab (NOT in the templates directory).

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

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

@app.route('/process', methods=['POST'])
def process():
    _username = request.form.get('username')
    if _username:
        return render_template('response.html', username=_username)
    else:
     return 'Please go back and enter your name...', 400
```

The fact that the application is directly involved in rendering the HTML means that this is not microservice. This is a monolithic application, albeit a really tiny one, since it has both processing logic and UI code intertwined. Generally, we also find calls to databases using to manage data like user account information.

Run the application and test it a few times to see how it responds. Notice that because it is tightly coupled to a web form, must run it through a browser.

Lab Six: Running a Flask Microservice

In the previous labs, we have been using flask to create web services. In the monolithic approach, if we needed access to logging or database or other facilities, the code to access these becomes part of the application. This is what being “monolithic” means – we can’t easily swap out the database code, even if the application is modularized correctly. But for legacy systems, there is often not a lot of modularization as the monolith has evolved over the years.

In the microservices architecture, the different services run independently but co-ordinate.

In this lab, you will create part of a financing approval microservice that might be used in a purchase application for a car. In this toy app, the name of a user is provided and, we assume, a number of checks are performed with results in a decision to offer financing, decline financing or require more information.

To do this, the finance microservice needs to get the credit status of the purchaser. To do this, it sends a request to a different microservice to produce a credit report (in this example, consisting of a one of three statuses: good, bad and unknown).

6.1 Creating the Credit Reporting Service

In real life, this would probably be executing queries into some data repository or sending requests to one of the credit reporting agencies, but we will just dummy it up.

The app takes a URI “/status/<name/>” and returns the status as a string.

Create a directory and prepare it the same way that you have previously. In the examples here, this directory is called “Service.”

The code to mock up the service is:

```
@app.route("/")
def index():
    return "Credit approval service."

@app.route("/status")
def status() :
    return "Please supply a name for credit status"

@app.route('/status/<username>')
def statusName(username):
    if username == 'Jack' :
        return "bad"
    if username == 'Walter':
        return "good"
    return "unknown"
```

Run the service on port 5000 and test all the different URIs to ensure it is working correctly. Once it is running correctly, just leave it running.

6.2 The Finance Service

In a new command window, create a new directory in lab6 called Finance. Prepare this directory for use in the same way you have in the previous labs.

For the finance application, user the following code. Notice the import at the top to allow the service to make GET requests to a URL. Also note that we should never hardcode a URL in our microservice code, but we are here just to make the lab go faster.

```
from flask import Flask
from urllib.request import urlopen


app = Flask(__name__)

# The URL of the web service would not be hardcoded
# It would be supplied as part of the environment configuration


@app.route("/")
def index():
        #with urlopen("http://localhost:5000") as r:
        #        return r.read()
        return "Finance Department"

@app.route("/report")
def report() :
        return "No name supplied for credit report"

@app.route("/report/<client>")
def reportClient(client):
        with urlopen("http://localhost:5000/status/" + client) as r:
                status = bytes.decode(r.read())
                if status == "good":
                        return "Offer Financing"
                if status == "bad" :
                        return "Decline Financing"
                return "More information needed"
```

Start the service running on port 80.

Test the service with the three test names hardcoded into the code

```
C:\Users\micro>curl localhost
Finance Department

C:\Users\micro> curl localhost/report
No name supplied for credit report

C:\Users\micro>curl localhost/report/Jack
Decline Financing

C:\Users\micro>curl localhost/report/Walter
Offer Financing

C:\Users\micro>curl localhost/report/Leslie
More information needed
```