Building Web Apps with Python and Flask

Learn to Develop and Deploy Responsive RESTful Web Applications Using Flask Framework

MALHAR LATHKAR

bpb

Building
Web Apps with
Python and Flask

Learn to Develop and Deploy Responsive
RESTful Web Applications Using Flask Framework

Malhar Lathkar

www.bpbonline.com

FIRST EDITION 2021

ISBN: 978-93-89898-835

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

MICRO MEDIA

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

Ph: 22078296/22078297

DECCAN AGENCIES

4-3-329, Bank Street,

Hyderabad-500195

Ph: 24756967/24756400

BPB BOOK CENTRE

376 Old Lajpat Rai Market,

Delhi-110006

Ph: 23861747

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai www.bpbonline.com

Dedicated to

All my teachers,
their profound influence has made me a lifelong learner

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Now is better than never.

Although never is often better than right now.

-- from The Zen of Python,
by Tim Peters

About the Author

Malhar Lathkar is an independent software professional, corporate trainer, freelance technical writer, and Subject Matter Expert with an experience of more than three decades. He has trained hundreds of students/professionals in Python, Data Science, Java and Android, PHP and web development, etc.

He also has the experience of delivering talks and conducting workshops on various IT topics. He writes regularly in a local newspaper on sports and technology-related current topics.

He is the author of the following books:

Computer Fundamentals, Software Concepts & Programming in BASIC: 1995

Ten in One: Handbook of Multilingual Programming: 2016

Python Data Persistence: With SQL and NoSQL Databases: 2019

Currently, he is working as the Director of the Institute of Programming Language Studies (Rama Computers), Nanded, Maharashtra.

https://www.linkedin.com/in/malharlathkar

About the Reviewer

Arun Kumar Gupta is a professional programmer. He is currently working as a **Sr. Technical Lead** in a leading multinational company. He has been developing software since 2010. He has worked with Python, using Flask/Django to build **APIs/web applications/Microservices** in e-commerce, health care, retail, and machine learning. He completed his **Masters in Computer Applications (MCA)** in 2010. When not at work, he enjoys spending time with his kids, and reading about new technologies.

Acknowledgement

First and foremost, I would like to express my gratitude from the bottom of my heart to BPB Publishers for giving me this opportunity to write this book. I am also grateful to BPB's editorial team, especially Mr. Arun Kumar Gupta – the technical reviewer, for their invaluable suggestions during the course of finalization of the manuscript.

My previous work – **Python Data Persistence with SQL and NoSQL Databases** – was also published by BPB in 2019, and it had an encouraging response. I am sure this book will also be equally well received by BPB's knowledgeable readers.

Most of this book has been written in the work-from-home scenario arising because of the Covid pandemic-induced lockdown, which means I had to encroach upon my wife's workspace! I thank her sincerely for being so accommodating! I would like to remain indebted to all my friends and family members who have always been appreciative and supportive.

Teaching is the best way of consolidating your knowledge of the subject. It helps in simplifying even the difficult concepts. I take this opportunity to thank all my students as well for being a part of my academic journey.

Finally, a big thank you to everyone involved in the making of this book!

Preface

In recent years, Python has established itself as one of the most popular programming languages. Although, the surge in its popularity is on account of its application in the field of Data Science and analytics, the Python-based web framework has become the preferred choice of a majority of web application developers worldwide.

Python's WSGI compliant framework is hugely popular among the Python developer community despite being fairly young (the first version of Flask was released only in 2010). Despite being classified as a micro framework, its extensible nature makes it very easy to interact with any database, template engine, form library, etc. In fact, the tagline of Flask as it appears on its official logo – **development, one drop at a** summarizes its philosophy very succinctly.

Admittedly, there are quite a few books on Flask. However, most of them have a tutorial-like approach. This book, on the other hand, aims to ease the reader with the process of web application development, starting from the basics of HTTP protocol, right up to how to deploy a Flask application on publicly visible servers.

How the book is organized?

Although, the reader is expected to have a reasonable proficiency in Python, a concise cheat sheet to help refresh Python skills is provided in the form of an appendix. Any web application technology needs the knowledge of front-end tools such as JavaScript and CSS. Their detailed discussion is beyond the scope of this book. However, a very brief introduction to these languages is given in *Chapter 6: Static*

There are twelve chapters in this book. The first three of them explain the foundational concepts of using Python for web application development. The next four chapters deal with the core features of Flask such as routing, templates, and HTTP objects. The remaining five chapters describe use of extensions, blueprints, and REST API along with deployment options.

Chapter 1: Python for discusses the basics of HTTP and explains how a Python code can be executed as a CGI script.

Chapter 2: explains the characteristic features of WSGI and demonstrates the functionality of the wsgiref module.

Chapter 3: Flask discusses the three Python packages on which the Flask API depends, namely, Werkzeug toolkit, Jinja2 template engine, and Click – command-line interface kit. It also discusses how to write a basic Flask application and run it in the debug mode.

Chapter 4: URL explains how Flask implements routing and dynamic URL rules.

Chapter 5: Rendering deals with how Flask renders templates dynamically using the Jinja2 syntax. Template inheritance and macros are explained with examples in this chapter.

Chapter 6: Static discusses how to handle static assets of a web application. It explains how Flask can leverage the power of JavaScript and CSS. An important aspect of any web app is the client-server interaction under the HTTP protocol.

Chapter 7: HTTP explains how Flask handles cookies, sessions, and other HTTP objects.

Chapter 8: Using explains how Flask can work with different relational and NoSQL databases with the help of Flask extensions such as and

Chapter 9: More Flask introduces important Flask extensions like Flask_WTF and Flask-Bootstrap along with some others.

Chapter 10: Blueprints and introduces other advanced features like application factory and context, along with blueprints, which are extremely helpful when it comes to building modular applications. Flask is extremely handy for building REST API services.

Chapter 11: Web API with describes how to build an API with a core routing mechanism as well as by using an extension with examples.

Chapter 12: Deploying Flask covers different deployment options available for a Flask application. It also covers deployment on shared hosting services as well as dedicated standalone servers.

This book is replete with ample code snippets. All example codes can be downloaded from the github repository of BPB publications. These examples have been thoroughly tested on the Windows system with the Python 3.7.2 version. However, barring some OS-specific syntax, they should work satisfactorily on any other OS such as Linux.

Downloading the code bundle and coloured images:

Please follow the link to download the ***Code Bundle*** and the ***Coloured Images*** of the book:

https://rebrand.ly/a7a06

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

BPB is searching for authors like you

If you're interested in becoming an author for BPB, please visit www.bpbonline.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

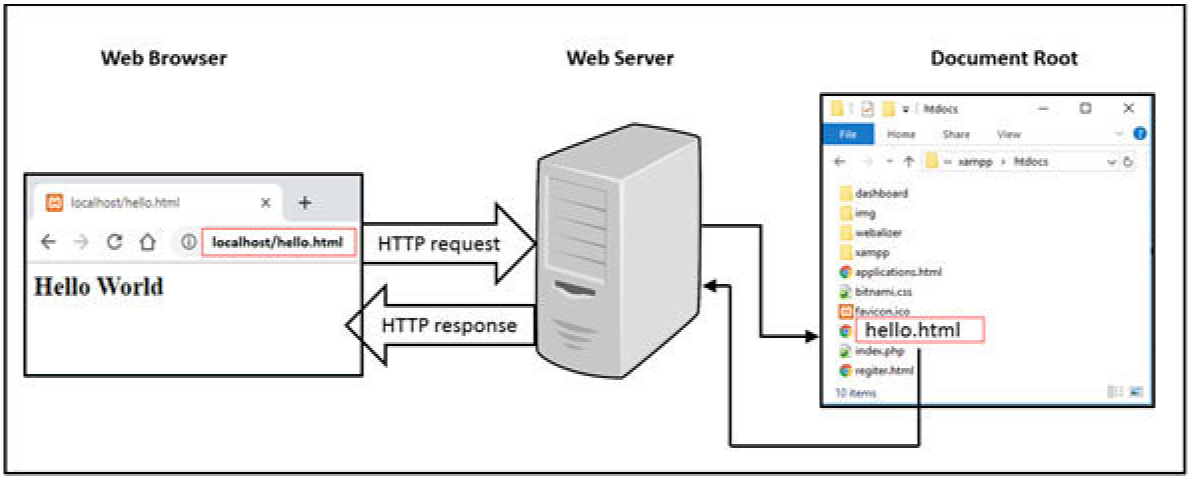
The code bundle for the book is also hosted on GitHub at In case there's an update to the code, it will be updated on the existing GitHub repository.

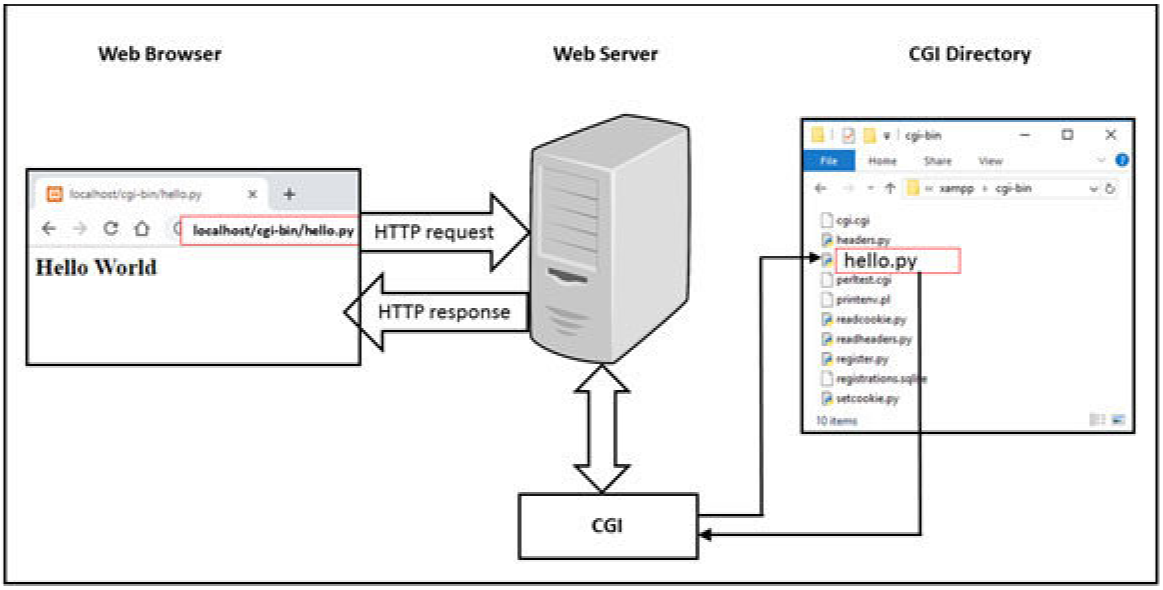
We also have other code bundles from our rich catalog of books and videos available at Check them out!

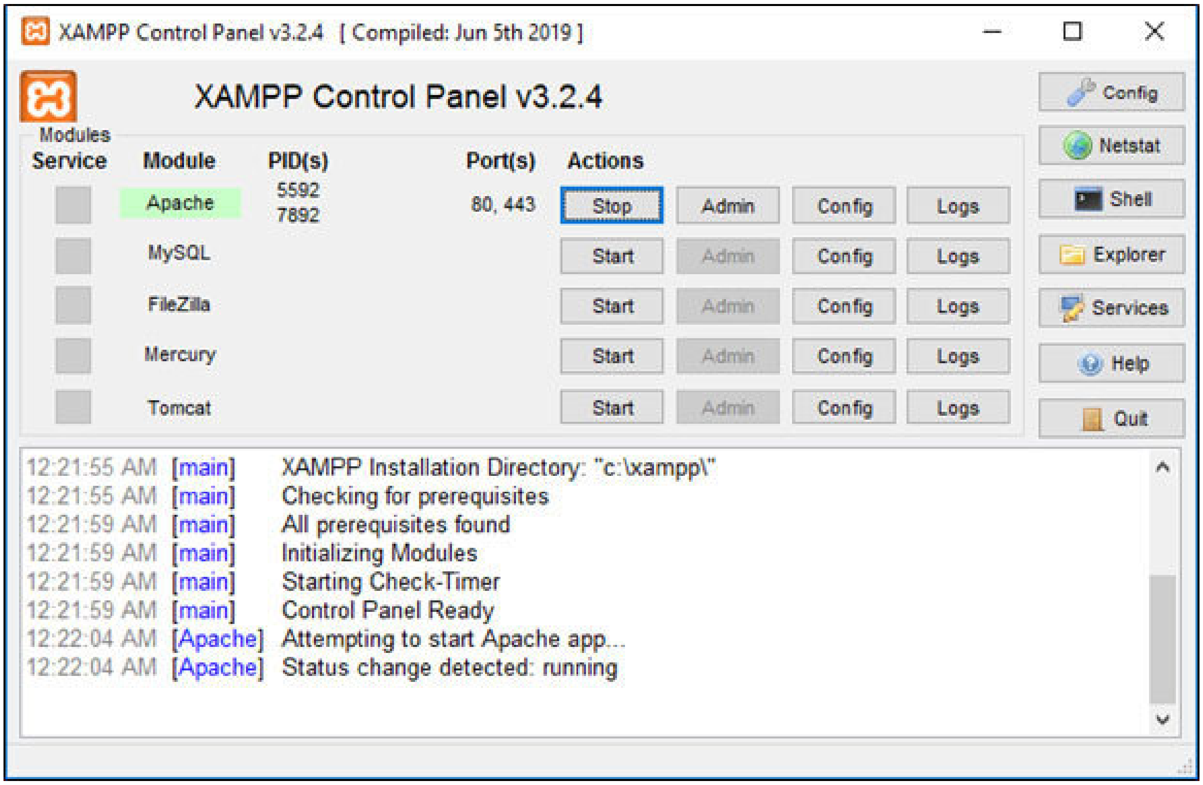
PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us

Table of Contents


Web Browser
Web Server
Document Root
localhost/hello.html
localhost/hello.html
Hello World
HTTP request
HTTP response
htdocs
File
Home
Share
View
« xampp › htdocs
dashboard
img
webalizer
xampp
applications.html
bitnami.css
favicon.ico
hello.html
index.php
regiter.html
10 items


Web Browser
Web Server
CGI Directory
localhost/cgi-bin/hello.py
localhost/cgi-bin/hello.py
Hello World
HTTP request
HTTP response
CGI
cgi-bin
File
Home
Share
View
« xampp › cgi-bin
cgi.cgi
headers.py
hello.py
perltest.cgi
printenv.pl
readcookie.py
readheaders.py
register.py
registrations.sqlite
setcookie.py
10 items


XAMPP Control Panel v3.2.4 [Compiled: Jun 5th 2019]
XAMPP Control Panel v3.2.4
Modules
Config
Netstat
Shell
Explorer
Services
Help
Quit

Service	Module	PID(s)	Port(s)	Actions			
	Apache	5592 7892	80, 443	Stop	Admin	Config	Logs
	MySQL			Start	Admin	Config	Logs
	FileZilla			Start	Admin	Config	Logs
	Mercury			Start	Admin	Config	Logs
	Tomcat			Start	Admin	Config	Logs

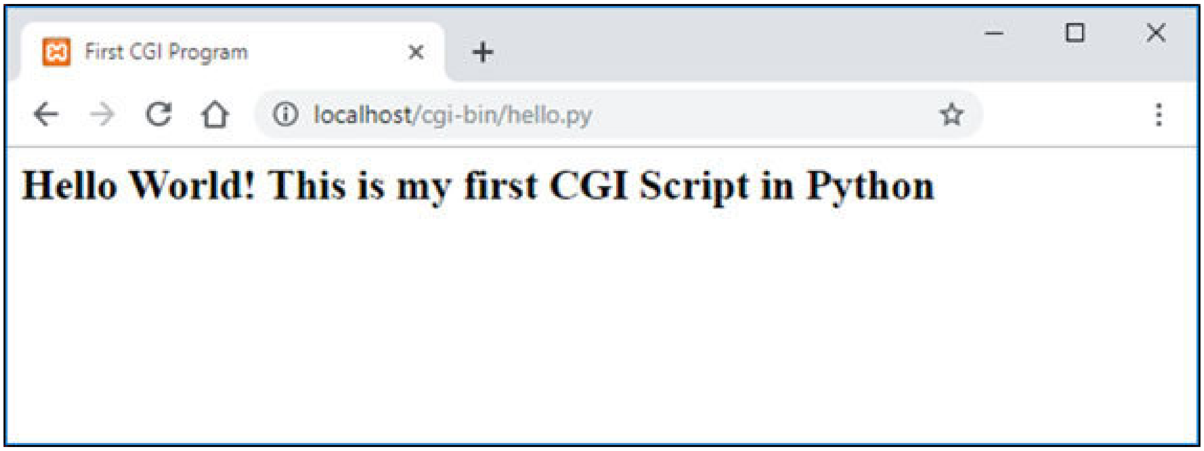
```
12:21:55 AM  [main]    XAMPP Installation Directory: "c:\xampp\"
12:21:55 AM  [main]    Checking for prerequisites
12:21:59 AM  [main]    All prerequisites found
12:21:59 AM  [main]    Initializing Modules
12:21:59 AM  [main]    Starting Check-Timer
12:21:59 AM  [main]    Control Panel Ready
12:22:04 AM  [Apache]  Attempting to start Apache app...
12:22:04 AM  [Apache]  Status change detected: running
```

```
('  ')
print  ('')
print  ('
```

Hello World! This is my first CGI Script in Python

```
')
print ('')
print ('')
```

Save this code as Hello.py in the C:\XAMPP\cgi-bin folder. Launch the Apache server, start a web browser application, and enter **http://localhost/cgi-bin/hello.py** as the URL in the address bar:



Hello World! This is my first CGI Script in Python

Figure 1.4: *CGI output*

The Basics of HTTP Protocol

At this juncture, it will be appropriate to understand how **HTTP** (stands for **Hypertext Transfer** works. As mentioned in the beginning, HTTP is the backbone of WWW. It is an application-level protocol for exchange of data in a client-server network working on the request – response model. Web browser software (Chrome, Firefox, etc.) is a client that sends HTTP request message for a certain web page hosted on the web server.

The HTTP web server accepts a request message and returns the corresponding HTML file as a HTTP response to the client, who is capable of interpreting the received HTML and renders the same in the browser. When a client browser sends a request, it also inserts additional information called HTTP headers. For example:

example: example: example:
example: example:
example: example: example: example: example: example: example: example:

example: example:
example:
example: example: example: example: example: example: example:
example:
example:

Table 1.1: *HTTP request headers*

Similarly, the web server also inserts some header information while it formulates an HTTP response to be sent to the client. Some of the response header fields are as listed:

listed: listed:
listed: listed:
listed: listed: listed:
listed: listed:
listed: listed:
listed: listed:

listed: listed:
listed: listed: listed: listed: listed: listed: listed: listed: listed: listed: listed: listed: listed: listed: listed: listed: listed:

Table 1.2: *HTTP response headers*

We will look at some of these headers in use in examples later in the chapter.

Request Methods

When a client sends a request, it also indicates by which method the server should perform the desired action on the requested resource - a web page or executable program. The HTTP protocol defines following request methods:

GET: This is one of the most common methods. It indicates that the request is made only for retrieving a specific resource. It is considered safe and can be repeated by the browser. While sending the request, message data may be appended to the URL in the form of query string. For example:

http://localhost/cgi-bin/test.py?name=xyz&age=20

The query string is a collection of key-value pairs concatenated by the & symbol. It is appended to the URL by the ? symbol. The GET method has some limitations. Its size can't exceed 2K bytes limit, and is exposed in the browser's address bar, so it may be a security hazard if it contains critical data such as password.

POST: This is another method that is used quite often. Unlike the GET method (which requests a certain resource on the server to be retrieved), data sent to the server is quite often meant to be stored (usually in backend databases). The POST method is not considered safe for repeated execution, so the server doesn't cache it.

Most common use of the POST method is to send HTML form data to a particular URL by specifying its method attribute to

```
action="http:/localhost/cgi-bin/test.py" method="POST">
HTML form element..
HTML form element..
type="submit" value="submit" />
```

One of the advantages of using the POST method is that there is no restriction on its data size. Also, the request query string is not exposed in the browser's URL.

HEAD: A request made using HEAD is identical to that of a GET request, but it is without the response body. This is particularly used for retrieving meta-information written in response headers, before actually retrieving the entire content; for example, downloading a file.

PUT: A PUT request is similar to POST in the sense it also indicates that the data sent is meant to be stored. However, it completely replaces the current resource at the target URL with message data. If nothing exists at the target URL, a new resource comprising the message data is created.

PATCH: The PATCH method was added to HTTP in 2010. It is used for making partial changes to an existing resource. While the PUT method replaces the original version of the resource (or creates a new resource if it doesn't exist already), the PATCH method updates a part of an already existing resource.

DELETE: As the name suggests, the DELETE request deletes the resource at the target URL.

This method is useful for checking the functionality of server. When a request is forwarded using the OPTIONS method, the server returns the HTTP methods that are supported for the given URL.

As mentioned earlier, most applications use the GET and POST methods. Following section explains how a Python CGI script handles these methods.

The cgi Module

Python has in-built support in the form cgi module as a part of standard distribution. There are two classes in this module: FieldStorage and They provide utilities using which the data sent by an HTTP client can be parsed and processed using Python script.

Python library also contains cgitb module. It acts as a special exception handler for Python's CGI script. This module is responsible for producing a detailed trace back report in case of an uncaught exception in the script. The report proves useful for debugging. A usual practice is to activate cgitb support by calling the enable() function:

```
import cgitb
cgitb.enable()
```

In the production environment though, the cgitb support may be removed. An object of the FieldStorage class is a dictionary like object. It parses the request query string into a collection of key-value pairs. Many features of the regular Python dict object can also be used with the FieldStorage object. For example, it is possible to use membership operator (in operator). Just as a dictionary, it has the keys() method to return a list of HTML form elements.

To demonstrate use of the FieldStorage class, let's design a simple HTML form using following script. The form has two text boxes for name and mobile number, two mutually exclusive radio buttons for selecting the gender, and a dropdown list of options for choosing from the available courses. There is a submit button too

When submit button is pressed, data in the form elements is submitted to a form handler script assigned as value of the form's action attribute. In this case, form handler is a Python script named Register.py to be stored in the cgi-bin folder. To specify the HTTP request method, the form has method attribute. It has been set to

Save following script as Register.html and store it in the document root of your XAMPP server, which is

#Register.html

align="center">Registration Form

action="/cgi-bin/Register.py" method="POST">

align="center">

Name : type="text" name="name"/>

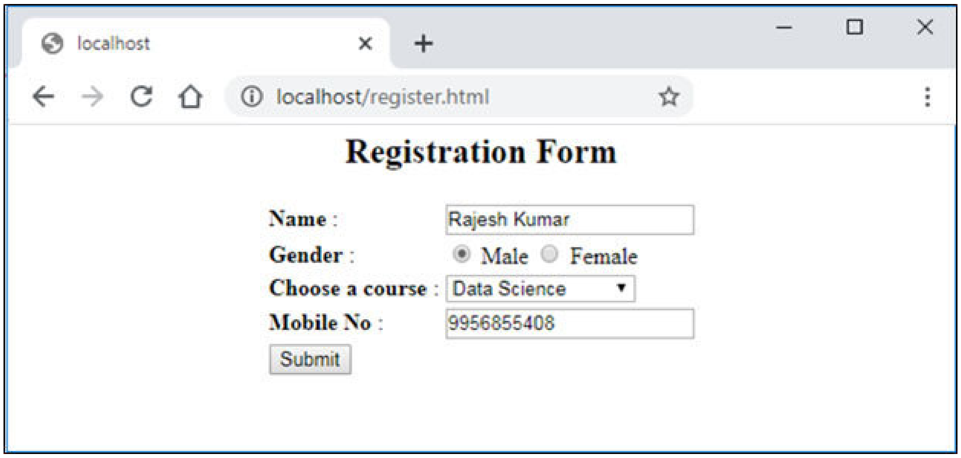
Gender : type = "radio" name = "gender" value = "Male" checked="checked" /> Male
type = "radio" name = "gender" value = "Female" /> Female

Choose a course : value = "Python with Flask" selected>Python With Flask

Mobile No : type="text" name="mobile"/>

type="submit"
value="Submit"/>

When opened in the browser, the Registration form should appear as follows:



Registration Form

Name : Rajesh Kumar
Gender : ● Male ○ Female
Choose a course : Data Science
Mobile No : 9956855408
Submit

Figure 1.5: *HTML form*

We now have to see how HTML form data can be accessed in a Python CGI script. This is where the FieldStorage object comes into the picture:

```
import cgi
form=cgi.FieldStorage()
```

This object can be considered as a dictionary object with the name attribute of HTML form element as key and the data entered as value. To obtain a string value of a certain key, use form["key"].value property. The FieldStorage class also has a convenient the same purpose.

Following Python program (actually a CGI script) is just intended to fetch HTML form data and render it back as HTTP response to the client. Save following script in the cgi-bin directory of our server

```
#Register.py
#!c:\python37\python.exe
print("Content-Type: text/html")
print()
# Import modules for CGI handling
import cgi, cgitb

cgitb.enable()

# Create instance of FieldStorage
form = cgi.FieldStorage()
# Get data from fields
nm=form.getvalue('name')
gnd=form.getvalue('gender')
course=form.getvalue('course')
mob=form.getvalue('mobile')

print ('')
print  ('')
print  ('
```

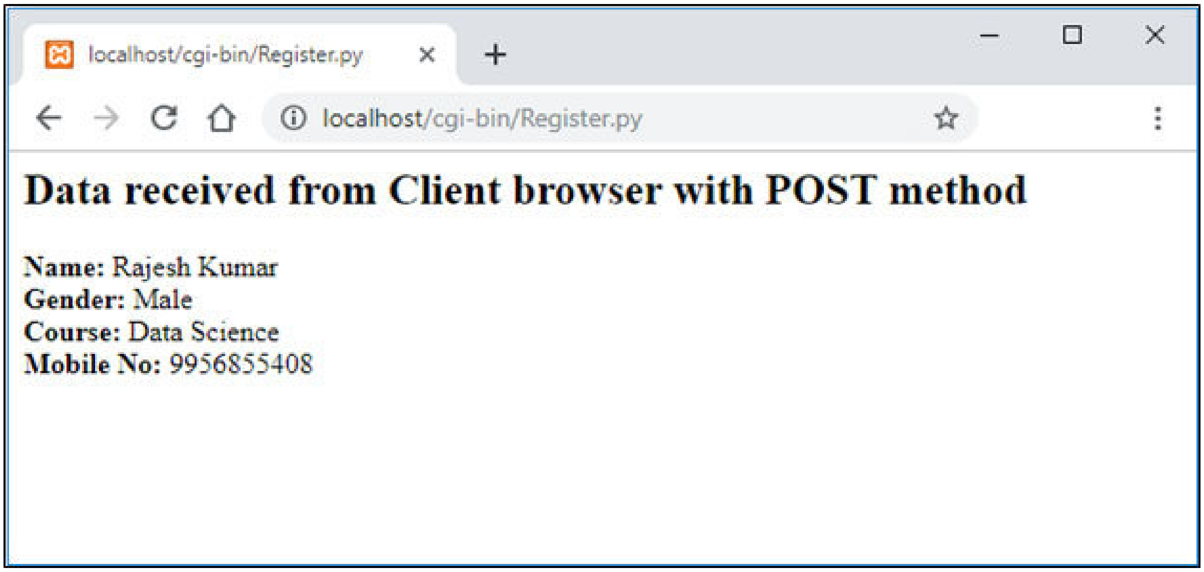
Data received from Client browser with POST method

```
')
print ('Name: {}
'.format(nm))
print ('Gender: {}
'.format(gnd))
print ('Course: {}
'.format(course))
print ('Mobile No: {}
'.format(mob))
print ('')
print ('')
```

With above script ready, open the browser and enter following URL in the address bar:

http://localhost/Register.html

Enter data in the form and press the **Submit** button. The Python CGI script will be executed. It fetches form data and renders it back as HTML output, as shown here:



```
Data received from Client browser with POST method

Name: Rajesh Kumar
Gender: Male
Course: Data Science
Mobile No: 9956855408
```

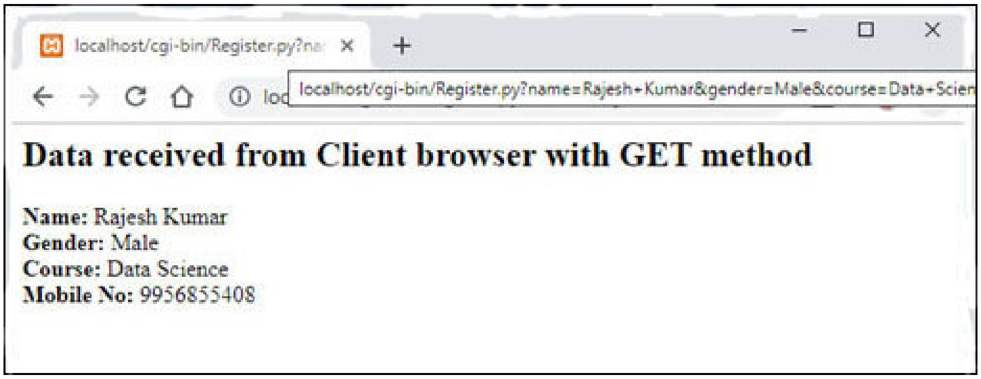
Figure 1.6: *Output of cgi script for the POST method*

It may be noted that browser's address bar shows the name of Python script but the form data is not displayed, which is not the case if GET method is used in the form's action attribute.

The FieldStorage object stores data received by the GET method too. However, the query string is exposed in the browser's address bar. To verify the same, just change the HTML form's method attribute and open Register.html again. Although the same Python script fetches form data, the browser bares it in its address bar:

http://localhost/cgi-bin/Register.py?name=Rajesh+Kumar&gender=Male&course=Data+Sci ence&mobile=9956855408

The browser shows this output:



localhost/cgi-bin/Register.py?name=Rajesh+Kumar&gender=Male&course=Data+Scien

Data received from Client browser with GET method

Name: Rajesh Kumar
Gender: Male
Course: Data Science
Mobile No: 9956855408

Figure 1.7: *Output of cgi script for the GET method*

Cookies

As mentioned earlier, cookie is one of the HTTP headers. The web server inserts one or more cookies in its response to the client browser's request. Such a cookie, which has a very small amount of data, is stored in the client machine. Whenever a client connects to the server, this cookie data is also attached along with the HTTP request.

The cookie mechanism is pretty useful for the server to record information about a client's browsing activity (for example, which pages of the website were visited by a particular client in the past). We often experience getting prompts based on what was entered previously in HTML form fields such as name and address, etc. This is possible because of cookies. Cookies are a reliable method of retrieving stateful information in otherwise stateless communication by HTTP protocol.

The Set-Cookie HTTP header is a string with one or more pairs of cookie name and its value. The pairs are separated by semicolon The easiest way to insert a cookie in HTTP response is to print a header string before sending other data. Consider this example:

```
#!c:\python37\python.exe
print ('Content-Type: text/html')
print ()
print ('Set-Cookie: msg="Hello world";')
print ()

htm= """
```

Some web page

```
"""
print (htm)
```

A more elegant way is to use SimpleCookie class defined in the http.cookies

```
>>> from http.cookies import SimpleCookie
>>> c=SimpleCookie()
```

The SimpleCookie object is similar to Python's built-in dict object. You can add values to it using normal dict syntax:

```
>>> c['userID']='admin'
>>> c['pwd']=1234
```

Just print the string representation of this object to generate the header:

```
>>> print (c)
Set-Cookie: pwd=1234
```

```
Set-Cookie: userID=admin
```

Following Python code is a CGI script that will store cookies in the client's browser:

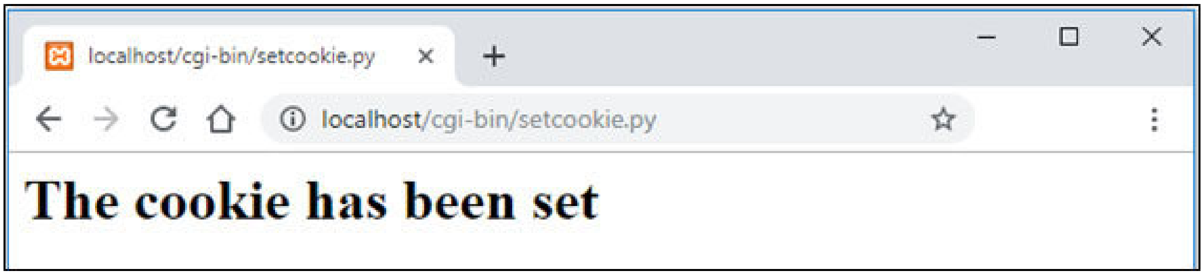
```
#setcookie.py
#!c:\python37\python.exe
from http import cookies
# create the cookie
c=cookies.SimpleCookie()
c['userID']='admin'
c['pwd']='1234'
# print the header, starting with the cookie
print (c)
print ("Content-type: text/html\n")

print ("")

htm="""
```

The cookie has been set

""""

print (htm)

When a client requests this script, cookies are set:


localhost/cgi-bin/setcookie.py
localhost/cgi-bin/setcookie.py

The cookie has been set

Figure 1.8: *Output of cgi script to set cookie*

On subsequent visits, the cookie objects are a part of the HTTP_COOKIE header. All the headers are stored as environment variables that are available in os.environ objects. In this example, the header will be:

Set-Cookie : UserID=Admin; Set-Cookie : pwd=1234;

To read the cookies programmatically, load the above cookie string as a SimpleCookie object:

```
>>> cookie_string="Set-Cookie : UserID=Admin; Set-Cookie :
pwd=1234;"
>>> c.load(cookie_string)
```

Each cookie can be obtained by splitting it at the = symbol to obtain a (key,value) tuple:

```
(key, value) = str.split(cookie, '=')
```

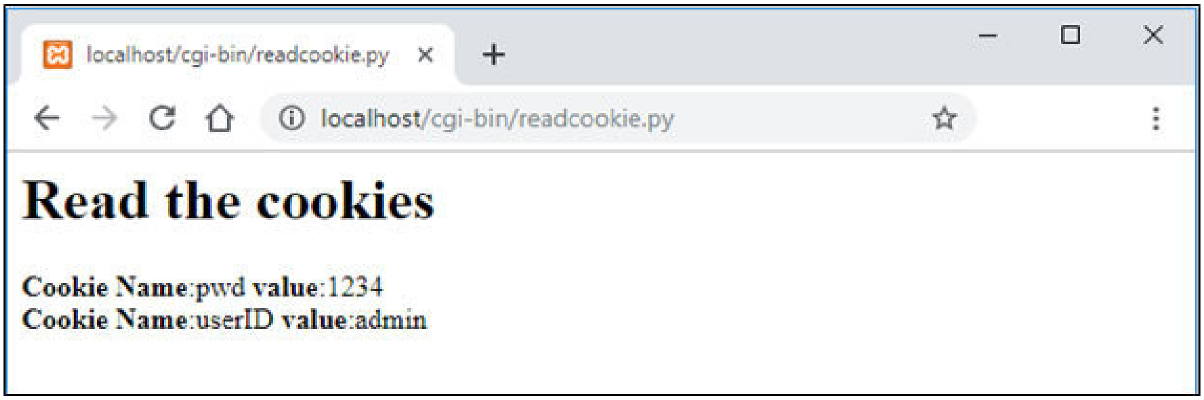
Given below is the complete Python script to read back cookies:

```
#readcookies.py
#!c:\python37\python.exe
from http import cookies
import os

print ("Content-type: text/html\n\n")
print()
print ("
```

Read the cookies

```
")

if 'HTTP_COOKIE' in os.environ:
cookie_string=os.environ.get('HTTP_COOKIE')
c=cookies.SimpleCookie()
c.load(cookie_string)
for cookie in map(str.strip, str.split(cookie_string, ';')):
(key, value) = str.split(cookie, '=')
print ("Name:{} value:{}".format(key,value))
print ("
")

print ("")
```

When the client browser visits this CGI script, cookie names and their values are displayed as follows:



localhost/cgi-bin/readcookie.py
localhost/cgi-bin/readcookie.py

Read the cookies

Cookie Name:pwd **value**:1234
Cookie Name:userID **value**:admin

Figure 1.9: *Output of cgi script to read cookie*

Alternatives to CGI

A CGI-based web application tends to slow down the server. CGI doesn't provide mechanism for data to be cached between page loads. Hence, the server starts a new process for each HTTP request, which consumes large memory and processor time, thereby affecting its response time.

This overhead can be minimized by using FastCGI, which is an improved variation of CGI interface. FastCGI works on the principle of preforking a process so that one such process can handle many requests during its lifetime and therefore, is able to reduce overhead and make the server more efficient.

Various other alternative approaches have been used to improve the web server's performance. Java Servlet container installed on a web server can serve dynamic content much more efficiently. Apache Tomcat server is popular for running Java-based web applications.

As far as Python-based web applications are concerned, a mod_python extension module significantly improves performance of Apache web server significantly. However, after Web Sever Gateway Interface (WSGI, which defines conventions to be used by web server for serving Python-based web applications) specifications were developed, the Apache server is equipped with mod_wsgi extension module.

Conclusion

In this chapter, we learned about the specifications of CGI to be implemented by web server. A Python executable program also serves as a CGI script. Python's standard library contains the cgi module, and we learned its functionality to build a basic CGI web application.

Next chapter introduces the features of WSGI and web frameworks based on WSGI. Flask, a WSGI-compliant web framework will also be introduced.

CHAPTER 2

WSGI

Introduction

Web Server Gateway Interface (WSGI) is a set of conventions recommended to be implemented by web server software to forward a client's request to a web application written in Python or Python-based web framework. It is an implementation-agnostic interface between web server and a web application. WSGI standard specifications have been developed with a view to provide common ground for portable web applications written in Python.

This chapter aims to acquaint the reader with WSGI specifications. It also explains how a WSGI application is developed with the wsgiref module, a part of Python standard library.

Structure

Following topics will be discussed in this chapter:

PEP 333

The environ dictionary object

WSGI Environment variables

The wsgiref package

Validator

HTML form in WSGI application

URL routing

Web frameworks

Flask

Objectives

After studying this chapter, you should be able to:

Understand the basics of WSGI

Write a simple WSGI application

PEP 333

As mentioned at the end of the previous chapter, a major drawback of CGI is the fact that the server starts a new process for each call to CGI script. It consumes a lot of memory and results in poor response time. Talking of Python-based web applications, use of the mod_python module significantly improves the web server performance.

A wide variety of Python web application frameworks can be problematic though, because the choice of framework limits the choice of web server software. So, it was felt that there should be a simple and universal interface between server and Python application – on the lines of Java's Servlet API that makes it possible for any Servlet-enabled server to run a Java application written in any Java framework.

Python Enhancement Proposal was raised to formulate standards for such an interface in 2003 (and later updated in 2010 with PEP 3333). It has been named as **Web Server Gateway Interface** WSGI is not web server software. It is neither any application framework nor a Python package. It is just a set of specifications recommended for both web server as well as Python web application frameworks. WSGI aims to facilitate easy interconnection of existing servers and applications or frameworks, not to create a new web framework.

This PEP specifies the roles of three participants in a typical web application: that of a web server, the framework or application itself, and a middleware object. These specifications can be summarized as follows. Upon receiving a request from a client, a WSGI-compliant web server passes it to the application, along with two positional arguments:

A Python dict object that is similar to CGI environment variables and certain WSGI specific variables. Certain server-specific extension variables may also be included.

A callback function to be used by the application to return its response along with headers and status code. HTTP headers expected by the client must be wrapped as a list of tupled pairs. The format of this function is as follows:

start_response(status, headers, exc_info=None)

WSGI application object is invoked by the server by passing the above-mentioned arguments. This object can be any callable object in Python, such as a function, method, a class, or its instance with the __call__() method available to it. This application object must return an iterator consisting of a single byte string:

```
def application (environ, start_response):
...
...
return [("Hello World!".encode("utf-8")]
```

Following block diagram illustrates the functioning of WSGI:

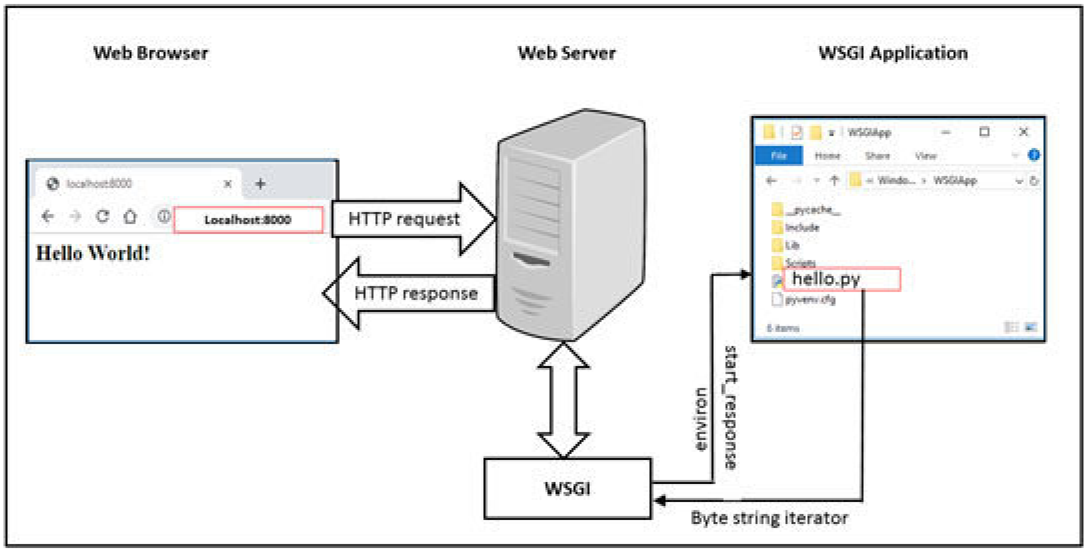


Figure 2.1: *WSGI server*

The status argument is an HTTP status string such as 200 OK or 404 Not The headers argument is a list of tuples. The exc_info argument is optional. If given, it must be a Python sys.exc_info() tuple and must be supplied only if start_response is being called by an error handler.

In addition to server/gateway and application specifications, WSGI also recommends specifications for middleware components that interact with both (application and server). These components can perform actions such as:

Routing a request to different application objects

Allowing multiple applications to run simultaneously

Load balancing and remote processing

The environ Dictionary object

The environ dictionary object is supplied by the server while invoking a WSGI application. It must contain the following CGI environment variables as keys:

Must be either GET or This cannot ever be an empty string.

This variable is set to initial portion of the request URL's path that corresponds to the application object. However, it may be an empty string if the application corresponds to the root of the server.

This corresponds to remainder of the request URL's path. It may also be an empty string if the request URL targets the application root and does not have a trailing slash.

Value of this variable is the portion of the request URL that follows the if any. It may be empty or absent.

The contents of any Content-Type fields in the HTTP request. It may be empty or absent; for example: content_type:'text/plain'.

Evaluates to contents of any Content-Length fields in the HTTP request. It may be empty or absent.

SERVER_NAME, When combined with SCRIPT_NAME and these two strings can be used to complete the URL. These values can never be empty strings and so, they are always required.

The version of the protocol the client used to send the request. Its value will typically be something like HTTP/1.0 or

HTTP_ These variables correspond to the client-supplied HTTP request headers (i.e., variables whose names begin with

WSGI environment variables

The PEP 3333 specifications require that the environ dictionary object (passed by the server while invoking WSGI application) must include the following WSGI-specific variables:

The value of this variable is an input stream from which the byte data of HTTP request body bytes is read. The input stream object must be a File like object (as returned by the built-in open() function) similar to sys.stdin and must support read() and readline() methods.

This variable points to an output stream to which program error output is directed. The output stream object should also be a file like object and must be a text mode stream that allows write() method.

This variable should be set to true if the application object may be simultaneously invoked by another thread in the same process; otherwise it should be false.

Value of this environment variable should evaluate to True if an equivalent application object may be simultaneously invoked by another process.

This variable should evaluate to True if the server or gateway expects the application to be invoked only this one time during the life of its containing process.

Value of this variable is always a two-member tuple corresponding to the WSGI version and sub version. For example: tuple (1, 0), representing the WSGI version 1.0.

It should be a string representing the scheme portion of the URL at which the application is being invoked. Normally, its value is http or

The wsgiref package

Python's standard library contains the wsgiref package. It is a reference implementation of WSGI specifications recommended by PEP 3333. It contains a set of utilities that can handle environment variables and response headers. It also provides a development server and a demo app.

The wsgiref.simple_server module implements an HTTP server that serves WSGI applications. Its make_server() function returns a server instance that serves an application at the given host and port. The module also contains a demo_app() function. The function receives the environ and start_response arguments (as required by WSGI specifications) and returns Hello World! message along with environment variables from the environ parameter:

```
from wsgiref.simple_server import make_server, demo_app
server = make_server('', 8000, demo_app)
```

The serve_forever() method of server instance makes the server to start accepting connections. Client browser can now visit http://localhost:8000

Following code is saved as

```
from wsgiref.simple_server import make_server, demo_app
server = make_server('', 8000, demo_app)
print ("Serving HTTP on port 8000...")

# Respond to requests until process is killed with ctrl-c
server.serve_forever()
```

Run this script from command line and open the above-mentioned URL in a browser:

```
C:\python37>python demoapp.py
Serving HTTP on port 8000...
```

As mentioned earlier, a WSGI application may be any Python callable object, such as a function, a class, or object with the __call__() method. Let's first define a Python function to be used as a WSGI application.

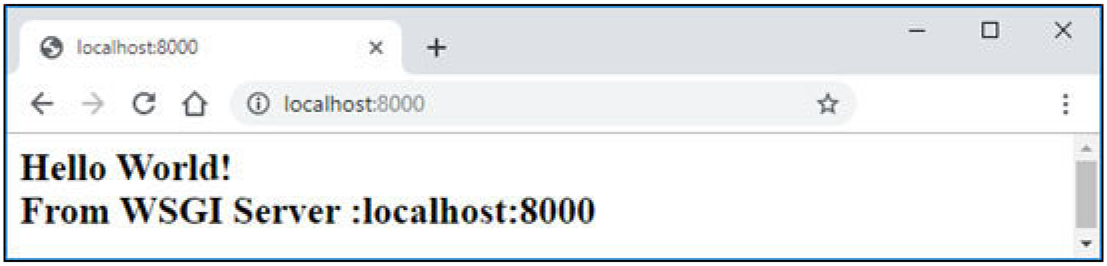
Again, as mentioned earlier, the web server provides environ dictionary object and the start_response() function while invoking this application. The start_response() function itself requires a status code string and response headers. The function should return an iterable with response body. Given below is a WSGI application function that reads HTTP_HOST environment variable and returns it as a response:

```
from wsgiref.simple_server import make_server

def application(environ, start_response):
host=environ.get('HTTP_HOST')
start_response("200 OK", [("Content-type", "text/html")])
ret = [("
```

Hello World!
From WSGI Server :%s

```
" % (host)).encode("utf-8")]
return ret


server = make_server('localhost', 8000, application)
server.serve_forever()
```

The client browser shows following result:



Hello World!
From WSGI Server :localhost:8000

Figure 2.2: *WSGI application*

As mentioned earlier, a WSGI application can be any callable object. In the following example, we use an object of a custom class as a WSGI application. Here, the __call__() method (that is executed after an object is initialized) receives the environ and start_response arguments:

```
from wsgiref.simple_server import make_server


class Application:
def __call__(self, environ, start_response):
host=environ.get('HTTP_HOST')
start_response('200 OK', [('Content-Type', 'text/html')])
ret = [("
```

Hello World!
From WSGI Server :%s

```
"\
% (host)).encode("utf-8")]
return ret


app = Application()


server = make_server('localhost', 8000, app)
server.serve_forever()
```

Validator

Another useful module in wsgiref package is the wsgiref.validate module. The compliance of a WSGI application with WSGI standards can be verified by the validator() function defined in this module. This function actually wraps a regular WSGI application and returns an object that forwards all requests to the original application, and checks whether the application and the server comply with the WSGI specifications. Any nonconformance results in an

The WSGI application is wrapped as follows:

```
from wsgiref.validate import validator
wrap_app = validator(app)
```

This object is further used to construct server object and serve the original application at a specified port. The complete code is given below. In the following example, we deliberately introduce an error by returning a string instead of an iterable (as required by WSGI standards). When a client visits the application, Python console reports

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server
def app(environ, start_response):
status = '200 OK'
headers = [('Content-type', 'text/plain')]
start_response(status, headers)

# application should return a list, but here it is returning
string
return b"Hello World"

wrap_app = validator(app)

with make_server('', 8000, wrap_app) as httpd:
print("Listening on port 8000....")

httpd.serve_forever()
```

When the browser visits **http://localhost:8000** URL, it displays an error message, as follows:

A server error occurred. Please contact the administrator.

On the other hand, Python console shows following error log:

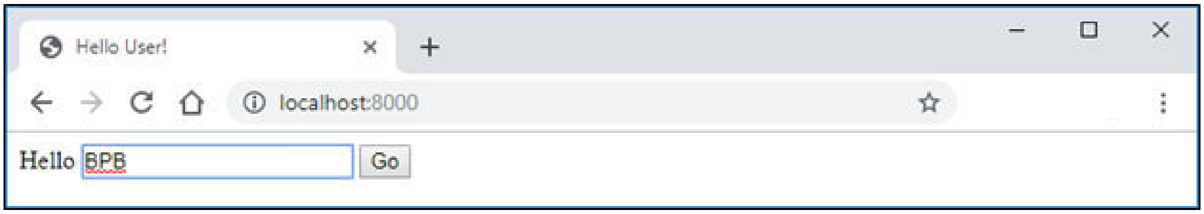
HTML Form in a WSGI application

We now know that a WSGI application returns a list object with a byte string, so we can easily render a HTML form on the browser's page by returning its raw HTML script. Following Python script runs an instance of WSGI server serving an application at port 8000 of When browser visits this URL, a simple HTML form is displayed, as shown here:

```
from wsgiref.simple_server import make_server
form = '''
method="GET" action="">
Hello
type="text" name="name">
type="submit" name="submit"
value="Go">
'''


def app(environ, start_response):
html = form
start_response('200 OK', [('Content-Type', 'text/html')])
return [html.encode('utf-8')]


httpd = make_server('', 8000, app)
```

```
print('Serving on port 8000...')
httpd.serve_forever()
```

The browser displays following form at



Hello BPB

Figure 2.3: HTML form

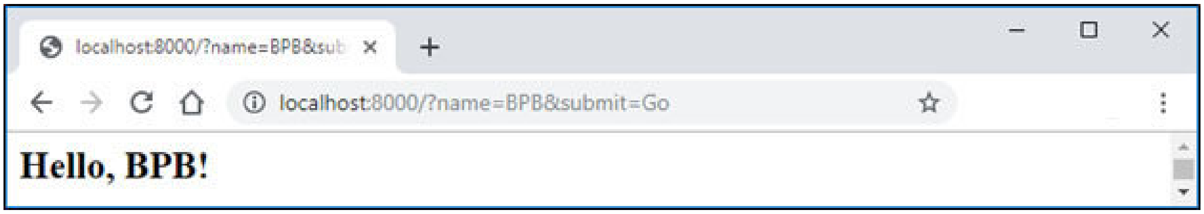
That said, we would like to handle the form data in our WSGI application. As the Form data is sent as HTTP request using the GET method, it is appended to the URL in the form of query string. We shall first parse this query string to a Python dictionary object using parse_qs() function from the urllib.parse module:

```
from urllib.parse import parse_qs
d = parse_qs(environ['QUERY_STRING'])
```

If the user has submitted the form (pressing **GO** button), we obtain value of the element and use it as application's return expression. Application's code is as follows:

```
#wsgiget.py
def app(environ, start_response):
html = form

if environ['REQUEST_METHOD'] == 'GET':
d = parse_qs(environ['QUERY_STRING'])
if d.get('submit',[''])[0]=='Go':

user = d.get('name', [''])[0]
html = ('
Hello, ' + str(user) + '!

')

start_response('200 OK', [('Content-Type', 'text/html')])
return [html.encode('utf-8')]
```

The application retrieves the form data and renders it in the browser, as illustrated here:


localhost:8000/?name=BPB&sub
localhost:8000/?name=BPB&submit=Go

Hello, BPB!

Figure 2.4: *The GET method in WSGI application*

Parsing POST data in a WSGI application is slightly tricky, but let's focus on two issues in the preceding example before discussing it.

Firstly, the web form is rendered on the client browser page in the form of a Python string, which looks a bit odd. Instead, we would like to use a prebuilt HTML file (such as register.html that was used in the previous chapter). The register.html file for this example is almost similar to the earlier one, except that the action attribute is set to

#register.html

align="center">Registration Form

action="" method="POST">

align="center">

Name : type="text" name="name"/>

type = "radio" name = "gender" value = "Male" checked="checked" /> Male

Gender : type = "radio" name = "gender" value = "Female" /> Female

Choose a course : value = "Python with Flask" selected>Python With Flask

Mobile No : type="text" name="mobile"/>

type="submit"
name="submit"
value="Submit"/>

One way of rendering HTML file through the application is to read its contents and convert it to byte string for rendering:

```
form=open('register.html')
html = (form.read()).encode('utf-8')
```

Output of the application is also first composed by inserting parsed form data in a string and converted to byte string before returning. Instead of it, we would like to use a HTML file having place holders to be filled in runtime by parsed query string. Let's call such a HTML file with place holders as a template.

Data collected by the preceding web form will be displayed in the following template web page:

#regdata.html

Registration confirmed!

Name: {name}
Gender: {gender}
Course: {course}
Mobile No: {mobile}

The identifiers put in curly brackets (such as are the place holders. They can be substituted by variables populated by parsing HTML form as shown here:

```
result=open('regdata.html')
html=result.read()
html=(html.format(name=name, gender=gender, course=course,
mobile = mobile)).encode('utf-8')
```

Finally, how do we parse a form having the POST request method? Here, we use FieldStorage class from the cgi module after removing QUERY_STRING attribute from the environ argument passed by the server:

```
post_env = environ.copy()
post_env['QUERY_STRING'] = ''
```

In the previous chapter, we had used the FieldStorage constructor with default arguments. Now we shall have to use its overridden prototype as follows:

```
form = cgi.FieldStorage(
fp=environ['wsgi.input'],
environ=post_env,
keep_blank_values=True)
```

The first argument is a file pointer pointing to the input stream from which HTTP request stream is read. Second argument is environ after stripping the QUERY_STRING attribute. Setting keep_lank_values to True indicates blank fields in the form to be treated as blank strings.

We can now parse the field values of each form element in corresponding string variables with the help of the getvalue() function as before:

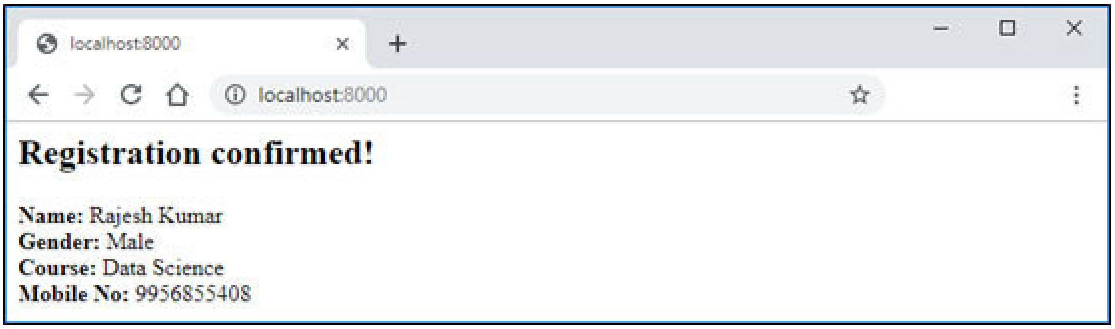
```
name=form.getvalue('name')
gender=form.getvalue('gender')
course=form.getvalue('course')
mobile=form.getvalue('mobile')
```

These variables are substituted in the regdata.html template as described earlier. Following Python code puts the above discussion in one place:

```
#wsgipost.py
from wsgiref.simple_server import make_server
from urllib.parse import parse_qs

import cgi

def app(environ, start_response):
form=open('register.html')
html = (form.read()).encode('utf-8')

if environ['REQUEST_METHOD'] == 'POST':
post_env = environ.copy()
post_env['QUERY_STRING'] = ''
form = cgi.FieldStorage(
fp=environ['wsgi.input'],
environ=post_env,
keep_blank_values=True
)

name=form.getvalue('name')
gender=form.getvalue('gender')
```

```
course=form.getvalue('course')
mobile=form.getvalue('mobile')
result=open('regdata.html')
html=result.read()
html=(html.format(name=name, gender=gender, course=course,
mobile=mobile)).encode('utf-8')
start_response('200 OK', [('Content-Type', 'text/html')])
return [html]

httpd = make_server('', 8000, app)
print('Serving on port 8000...')
httpd.serve_forever()
```

The registration form will be displayed when the browser visits http://localhost:8000 and the submission results in following output:



Registration confirmed!

Name: Rajesh Kumar
Gender: Male
Course: Data Science
Mobile No: 9956855408

***Figure 2.5:** POST method in a WSGI application*

URL Routing

So far, our WSGI application is performing activity at a single URL (that is, A web application written in PHP or JSP redirects the application's flow to different URLs that are essentially web pages interspersed with code. How can we achieve such redirection in a WSGI application?

One option is to dynamically construct URLs and use multiple conditional code blocks for each. Instead of this rather tedious approach, the second option is more elegant. To begin with, we need to use PATH_INFO attribute from the environ object provided by the server. It returns remainder of the request's URL, excluding SCRIPT_NAME and For instance, if the requested URL is **the PATH_INFO** returns 'page'.

We shall plan the web application such that it consists of a WSGI compliant function corresponding to each URL in it. For example, we may have and URLs mapped to the register() and login() functions that render registration.html and respectively. The following dictionary is a mapping of URL and function:

```
routes=[('/', index),('/register',register), ('/login', login),
('/loginresult', loginresult), ('/regok', regok),]
```

The regok() and loginresult() functions render the result of registration and login form, respectively. These HTML forms are displayed when application is redirected to the corresponding URLs from the index page. The index.html itself is first displayed when WSGI application function is invoked:

```
import fnmatch
def myapp (environ, start_response):

for path, app in routes:
if fnmatch.fnmatch(environ['PATH_INFO'], path):
return app(environ, start_response)
server = make_server('localhost', 8000, myapp)
server.serve_forever()
```

As we know, the application goes in an infinite loop. Note the use of fnmatch() function from the built-in fnmatch module. It checks for the PATH_INFO attribute of request URL in routes dictionary and returns the mapped function to be invoked subsequently. The functions are defined as follows:

```
#menu
```

```
def index(environ, start_response):
start_response("200 OK", [("Content-type", "text/html")])
f=open('index.html')
ret = (f.read()).encode('utf-8')
return [ret]

#registration form
def register(environment, start_response):
start_response("200 OK", [("Content-type", "text/html")])
f=open('registration.html')
ret = (f.read()).encode('utf-8')
return [ret]

#login form
def login(environ, start_response):
f=open('login.html')
ret = (f.read()).encode('utf-8')
start_response("200 OK", [("Content-type", "text/html")])
return [ret]

#registration confirmation


def regok(environ, start_response):
ret='''
```

You have been successfully registered

```
href="http://localhost:8000/">Main Menu
'''
start_response("200 OK", [("Content-type", "text/html")])
return [ret.encode('utf-8')]


#login confirmation
def loginresult(environ, start_response):
qry=parse_qs(environ['QUERY_STRING'])
name=qry.get('name', [''])[0]
ret=""
if name!=None:
pwd=qry.get('pwd',[''])[0]
if pwd=="1234":
ret="
```

Welcome {}

".format(name)

start_response("200 OK", [("Content-type", "text/html")])

else:

ret="

Incorrect password

```
"
start_response("404 Not Found", [("Content-type", "text/html")])
else:
ret=''
return [ret.encode('utf-8')]
```

To begin with, after the server starts, the **http://localhost:8000** URL in the address bar takes it to /index route and displays following page:

#index.html

Main Menu

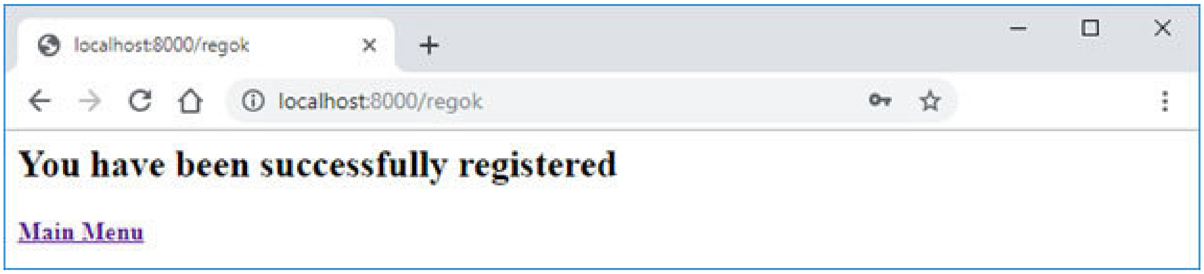
href="http://localhost:8000/register">

click here to register

href="http://localhost:8000/login">

click here to login

The registration page is similar to the one used earlier. The only difference is that the action attribute of form is set to **http://localhost:8000/regok** URL, which invokes the regok() function:



You have been successfully registered

Main Menu

Figure 2.6: *URL routing example*

The login page has two form elements: a text input for name field and a password field:

#login.html

align="center">Login Form

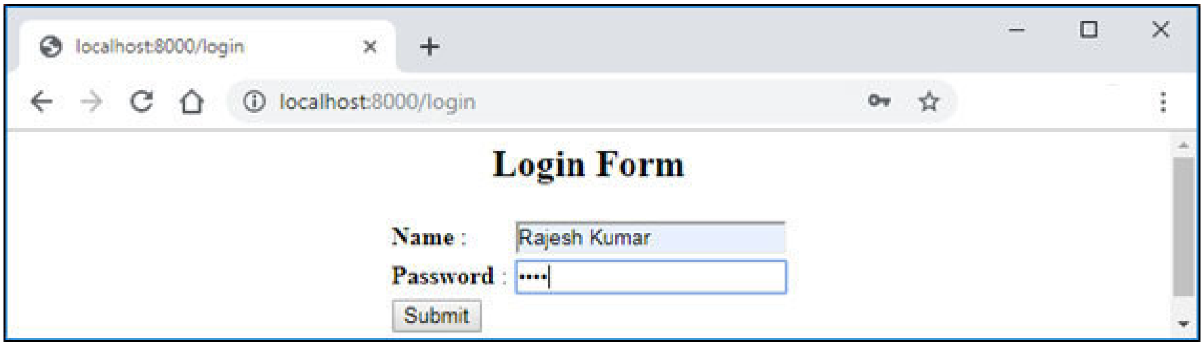
action="http://localhost:8000/loginresult" method="GET">
align="center">

Name : type="text" name="name"/>

Password : type="password" name="pwd"/>

type="submit" name="submit" value="Submit"/>

The **http://localhost:8000/login** URL calls the login() function and results in following display:

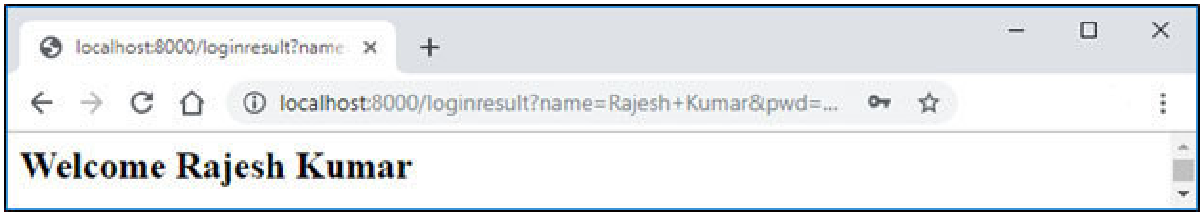


Login Form

Name : Rajesh Kumar
Password : ••••
Submit

__Figure 2.7:__ URL Routing Login form

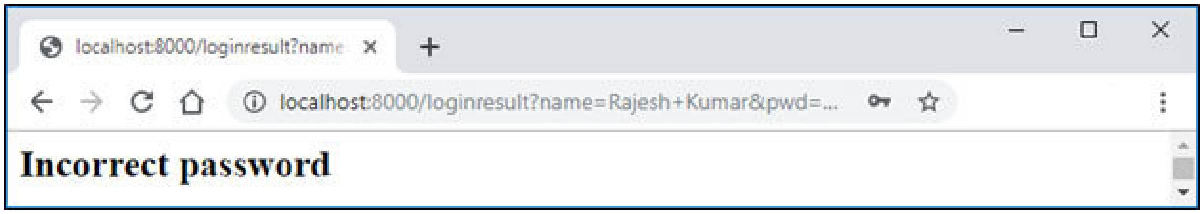
When expected password is entered in this case), following welcome message is rendered by the loginresult() function:


localhost:8000/loginresult?name
localhost:8000/loginresult?name=Rajesh+Kumar&pwd=...

Welcome Rajesh Kumar

Figure 2.8: *URL routing Login result*

However, in case of an incorrect password, appropriate message with an error status code is generated:


localhost:8000/loginresult?name
localhost:8000/loginresult?name=Rajesh+Kumar&pwd=...

Incorrect password

Figure 2.9: *URL routing Login unsuccessful*

Complete Python code demonstrating the URL routing is available in the code repository at The required HTML files and can also be found there.

Web Frameworks

Different aspects of a typical web application (such as URL routing, templating, handling request, and forming response, etc.) as described until now are a little verbose in nature. However, they happen to be the basic building blocks of Python-based web frameworks.

A web framework (also called **web application** is a set of utilities for easier handling of the above-mentioned aspects in a more robust manner. The use of a framework facilitates rapid development of scalable and reliable web applications. Web application frameworks are available for several programming languages, including Java, PHP, and Python.

Some frameworks are known as full-stack (also called batteries-included framework). They provide all-in-one software stack of utilities required for all components of an application, i.e., server-side processing, front-end interface development, and database handling. Others such as Flask are called as microframework. In contrast to full-stack framework, they are minimalistic in nature.

Apart from the core activities such as routing, templating, and request-response handling, a micro framework allows the developer to choose any library for other aspects of the web application. Micro frameworks such as Flask are especially useful for the development of smaller applications, APIs, and web services.

Many web frameworks use the **Model-View-Controller** (popularly known as architecture. The three components of a web application, namely data model (Model), application's algorithm and processing logic (Controller), and user interface (View) are segregated. This approach ensures modularity and allows code reuse:

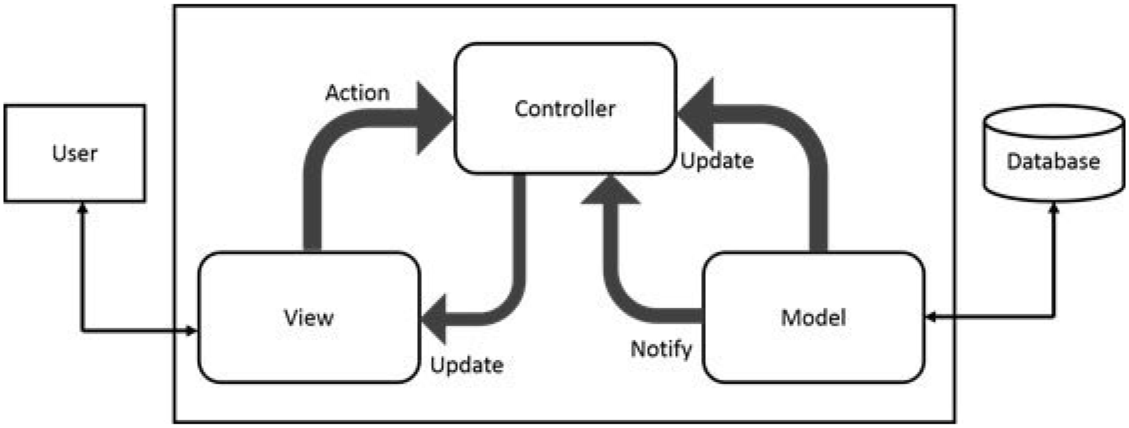


Figure 2.10: *MVC architecture*

Rest of this book explains the functionality of Flask, a very popular web application framework for Python.

Flask

Flask is one of the most popular Python-based web frameworks, especially for relatively small web applications. As mentioned earlier, it is classified as a micro framework since it offers the bare minimum functionality required in a framework.

Flask is entirely WSGI compliant. It was developed as an open-source project by Armin Ronacher, an Austrian developer, in 2004, when he was heading Pocoo, an international group of Python developers. Latest version of Flask is 1.1.1, which is hosted at Source code of the Flask package is available for download at

Flask is essentially a wrapper around two Python packages: Werkzeug – a set of WSGI utilities and Jinja – a web templating engine. Both packages are also developed by Ronacher and Pocco. We shall learn more about them in the next chapter.

Although Flask is a minimalistic framework, other features can be easily added in the application with the help of several

Flask extensions. For example, Flask doesn't have any in-built provision for database connectivity, but the Flask-SQLAlchemy extension enables using SQLAlchemy, an object relation mapper with different RDBMS platforms. Some useful Flask extensions are discussed later in this book.

Flask comes with an in-built development server, but it is not suitable for production. However, there are many options available for hosting a Flask application on shared servers such as Heroku or PythonAnywhere as well as on standalone web servers like Apache. These deployment options are explored in one of the subsequent chapters of this book.

Conclusion

In this chapter, we learned about WSGI specifications as prescribed by Python Enhancement Proposal (PEP 333). Python's standard library includes the wsgiref module, which is a reference implementation of WSGI standards. We developed a simple WSGI application with multiple routes.

This chapter also introduced the features of Flask web application framework library. In the next chapter, we will understand how to install Flask and run a simple Hello Flask application.

CHAPTER 3

Flask Fundamentals

Introduction

Flask is a lightweight web application framework written in Python. It is compatible with WSGI specifications. Flask API has two main dependencies: Werkzeug and Jinja—both Python packages developed by the Pocco group. Flask also uses click library for implementing command line interface.

In this chapter, we shall take a brief overview of these packages. Later, we shall cover the important features of Flask, followed by its installation procedure. Towards the end of the chapter, we shall write a simple Flask application displaying a **Hello Flask** message.

Structure

Following topics will be discussed in this chapter:

Werkzeug

Jinja2 Template Engine

Click

Installing Flask

Hello Flask Application

Debug mode

Objectives

After studying this chapter, you should be able to:

Write a simple Hello Flask application and run it from command prompt

Activate debug mode and make your application externally visible

Werkzeug

The Python ecosystem is replete with web application framework libraries. Flask stands out with its modularity and extensible nature. As mentioned in the previous chapter, Flask is a wrapper around two Python packages: Werkzeug and jinja2. To begin with, let's look at the functionality of the Werkzeug package.

Werkzeug is a German word whose meaning is closest to *tool* or Ronacher used this as the name of a Python library of a comprehensive toolkit required for WSGI-compliant web applications. In the previous chapter, we learned that a WSGI server sends environ object and the start_response() function to the callable object. To recap, a simple WSGI application using the wsgiref module is as follows:

#wsgiexample.py

```
from wsgiref.simple_server import make_server

class Application:
def __call__(self, environ, start_response):
```

```
start_response('200 OK', [('Content-Type', 'text/html')])
ret = ["
```

Hello World!

```
".encode("utf-8")]
return ret

app = Application()

server = make_server('localhost', 8000, app)
server.serve_forever()
```

The Werkzueg library eliminates the need to explicitly fetch response and request data from the environ parameter, which is received from the web server. The library itself extracts this data in the form of a Python object. The following code demonstrates how to render Hello World message using the werkzeug library:

#wrkexample.py

```
from werkzeug.wrappers import Response
from werkzeug.serving import run_simple

def application(environ, start_response):
response = Response('
```

Hello World!

```
', mimetype='text/html')
return response(environ, start_response)

run_simple('127.0.0.1', 5000, application)
```

The Request and Response objects are defined in the werkzeug.wrappers module. A development server named as run_simple is defined in the werkzeug.serving module, and there are Rule and Map functions in the werkzeug.routing module. These functions provide an easy mechanism of dynamic URL routing and redirection.

At this juncture, we need not explore these functions in depth because Flask packages them in easy-to-handle form. We shall, of course, understand these terms in more detail in the subsequent chapters. The important features of the werkzeug library can be summarized as follows:

As demonstrated earlier, Werkzeug has a built-in development server for WSGI applications.

The Request object provides access to HTTP headers, cookies, and form data.

The Response object efficiently handles the output stream towards client.

Werkzeug defines a URL routing system for dynamic URL routing.

Werkzeug is completely compatible with WSGI standards and perfectly capable of handling Unicode data.

It has integrated support for unit testing

Jinja2 Template Engine

Another important pillar of Flask framework is Jinja template library for Python. It is a fast and designer-friendly language for dynamically generating web documents. Jinja library (referred to as jinja2 hereafter, being the latest version) is also developed by the Pocco group.

A web template system comprises of a template engine that receives a content stream, which may be in the form of a database, an XML document, or any other data stream from a local or network source. Other input that goes into the engine is a web template, which is essentially a web page with embedded template language code. Variables in the templating language part of the otherwise static web page are populated by the data stream. The result is dynamically generated web documents. Typical use of template is generating a web page by inserting the result of a search operation by a web application.

Following diagram illustrates how a web template system works:

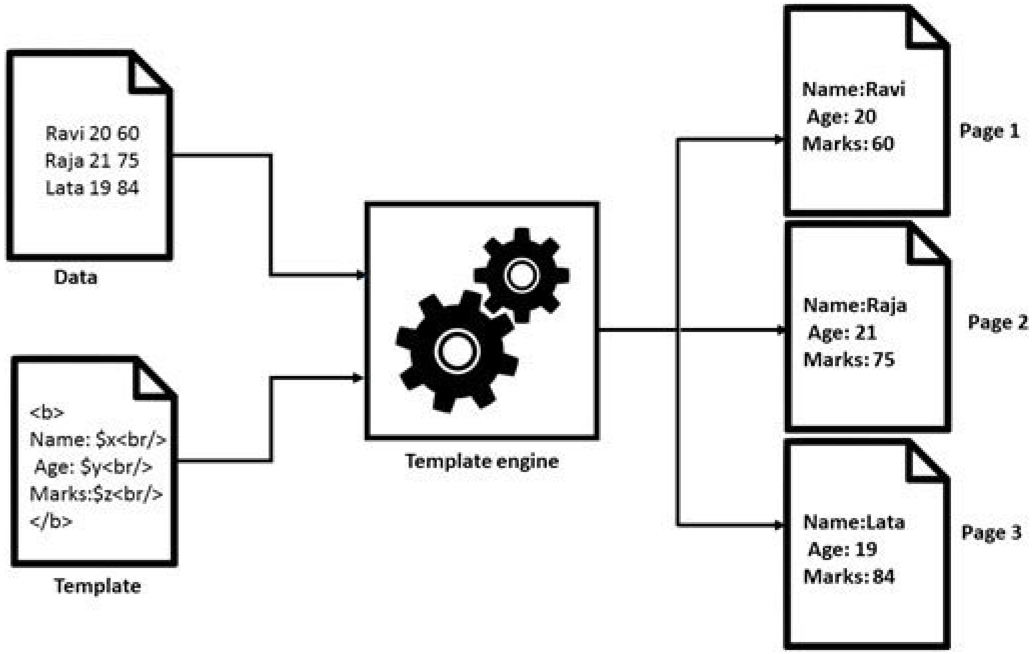


Figure 3.1: *Web template system*

These are some of the important features of Jinja library:

Jinja2 provides a sandboxed environment to evaluate any untrusted code so that potentially unsafe data and methods are prohibited.

To prevent cross-site scripting attacks (called **XSS** a mechanism for automatic HTML escaping is defined.

Jinja2 supports template inheritance. This is a very powerful feature. You can define a base template with elements that are common design features, which child templates can override.

The template syntax is easy to configure and debug.

The template code is embedded inside a HTML script. It is put inside curly brackets. Various code elements are identified by the engine with the following delimiters:

{% %} - statements

{{}} - expressions to print to the template output

{# #} - comments that are not included in the template output

- line statements

Template object is the fundamental object of the library. Most basic usage of Jinja2 API is calling the render() method of Template object:

```
>>> from jinja2 import Template
>>> name="Admin"
```

```
>>>tempobj=Template("Hello {{name}}!")
>>>tempobj.render(name=name)
```

'Hello Admin!'

Jinja2 also supports the use of traditional programming constructs such as conditionals and loop. The loop is used to iterate over a collection (such as list, tuple of dictionary) and is constituted with the help of the {% for x in list %} and the {% endfor %} statements:

```
>>> from jinja2 import Template
>>> t = Template("squares: {% for n in range(1,10) %}{{n*n}}
{% endfor %}")
>>>t.render()
'squares: 1 4 9 16 25 36 49 64 81 '
```

The conditional expression is implemented by the {% if expr {% else and {% endif %} expressions of Jinja2 syntax:

```
>>> t = Template("numbers>5 : {% for n in range(1,10) %}\
{% if n>5 %}\
{{n}} \
{% endif%}\
{% endfor %}")
>>>t.render()
'numbers>5 : 6 7 8 9 '
```

Let's put our knowledge about Jinja2 template library so far, to use with a Werkzeug-enabled WSGI application. Value for the name variable is read as command line argument when the following program is executed. The template object is constructed by inserting the user provided data, and the string rendered by this template object is returned as a response by the WSGI application.

Save the following code as

#TestTemplate.py

```
import sys
from werkzeug.wrappers import Response
from jinja2 import Template
name = sys.argv[1]
tm = Template("Hello {{name}}")
msg = tm.render(name=name)

defapplication(environ, start_response):
response = Response(msg, mimetype='text/html')
```

```
return response(environ, start_response)
```

```
from werkzeug.serving import run_simple
run_simple('127.0.0.1', 5000, application)
```

While running this script from command prompt, provide the name parameter:

```
python TestTemplate.py Admin
```

Visit and the browser displays Hello Admin message.

We are not going to dig further into the functionality of the Jinja2 library because Flask framework encapsulates it more efficiently, and we shall get to know more about it in the subsequent chapters.

Click

In addition to Werkzeug and jinja2, the later versions of Flask library (after version 0.10) has one more dependency in the form of the click package. Click stands for **Command Line Interface Creation Kit.** Flask implements its command-line interface with its help.

The basic idea is to define a function by decorating it with a command() decorator. Such a function becomes a command line tool. A simple example, as given below, has a hello() function decorated with the click.command() decorator:

```
#helloclick.py
import click

@click.command()
def hello():
click.echo('HelloClick!')
if __name__ == '__main__':
hello()
```

Run above script from Command Prompt, as follows:

A --help option is automatically initialized to show a help message such as:

Note the use of the echo() function, which behaves like a built-in print() function. However, it has an improved support to Unicode and binary data.

There is also a group() decorator. A function registered to this decorator can be attached to other commands. Here's an example of the Welcome() group function attached to the greetings() and goodbye() command functions:

#helloclickgroup.py

```
import click

@click.group()
def Welcome():
click.echo("Welcome to Click!")

@click.command()
def greetings():
click.echo('Hello user')

@click.command()
def goodbye():
click.echo('Goodbye user')

hello.add_command(greetings)
hello.add_command(goodbye)
if __name__ == '__main__':
Welcome()
```

Command line usage of the above Python script is as follows:

Another an argument to the command function. Let's add name argument to the greetings() and goodbye() functions as follows:

#helloclickarg.py

```
import click


@click.group()
def Welcome():
click.echo("Welcome to Click!")
```

```
@click.command()
@click.argument('name')
def greetings(name):
click.echo('Hello {}'.format(name))

@click.command()
def goodbye(name):
click.echo('Goodbye {}'.format(name))

Welcome.add_command(greetings)
Welcome.add_command(goodbye)
if __name__ == '__main__':
Welcome()
```

Value of the name argument is to be supplied from command line. If not provided, Click reports exception, as follows:

```
C:\python37>python helloclickarg.py greetings
Welcome to Click!
Usage: helloclickarg.py greetings [OPTIONS] NAME

Try "helloclickarg.py greetings --help" for help.

Error: Missing argument "NAME".
```

This is the appropriate use of command line argument:

We shall come across such CLI being used with Flask application in the following section. After getting acquainted with the functionality of Flask dependencies, we are now ready to get started with Flask itself.

Installing Flask

Like most of Python packages, Flask is very easy to install using the pip utility that is included in Python's standard distribution. However, instead of installing it in Python's home directory for system-wide use, it is recommended that we create a virtual environment and install the same in it.

To understand the importance of virtual environment, assume that project1 needs version 1.1 of a certain library (or module) X, and project2 depends on a newer (2.1) version of the same library. There are chances that the installation of a newer version might break the functionality of project1 if both projects are using system-wide installation of Python.

To avoid this, these projects can be put in different isolated environments. A virtual environment is a directory (or folder) having a local copy of Python interpreter (it may have different versions of Python too) and its own set of packages installed so that one environment doesn't infringe upon the other.

Standard library's venv module is used to create a new virtual environment. Following command creates a virtual Python environment in the c:\flaskenv directory:

```
C:\python37>python -m venv c:\flaskenv
```

The scripts folder under c:\flaskenv holds Python executable and other utilities, including It also has which activates the environment. Any package installed after activation will be installed for use within the environment only:

To get a list of available packages, use the freeze clause of pip utility:

Since Flask depends on Werkzeug, jinja2, and click, these packages are automatically installed. Additionally, we see itsdangerous and MarkupSafe packages. MarkupSafe is installed along with Its purpose is to escape any untrusted input while rendering the template. The itsdangerous package is used to protect session cookies of a Flask application. To check whether Flask is correctly installed, try importing it in a Python session and check its version:

```
>>> import flask
>>>flask.__version__
'1.1.1'
```

If you are having Anaconda distribution of Python, you need to use the conda utility for installation:

```
conda install -c anaconda flask
```

Hello Flask Application

Without any further ado, let's build a minimal Flask application. Save the following script as hello.py in the virtual environment and make sure that it is activated:

```
#hello.py
from flask import Flask
app = Flask(__name__)


@app.route('/')
def hello():
return 'Hello, Flask!'
```

At this juncture, we shall not dwell much on what this script does. For now, it is enough to know here that app is an object of Flask class that actually is a WSGI application, and the function is mapped to / URL. In the next chapter, we shall learn in detail what this script does.

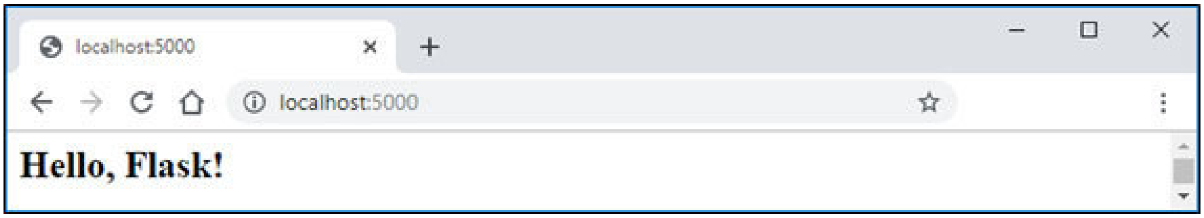
Do not run the above script (it will not result in anything even if you do!). There are two ways in which a Flask application is run: using Flask command or using Python executable's –m switch.

In either case, you need to assign our script to FLASK_APP environment variable:

A command line interface to Flask library is installed in the virtual environment by the name When invoked in the terminal, the following help page is displayed:

Clearly, this interface is built with the Click package we explored earlier in this chapter. As this help message shows, set the FLASK_APP variable (which we have done already) and then execute its run command. Our application will be served at the default port number (which is of the development server with localhost as its name:

Open your browser and point it to **http://localhost:5000/** as the URL. It shows the following output:



Hello, Flask!

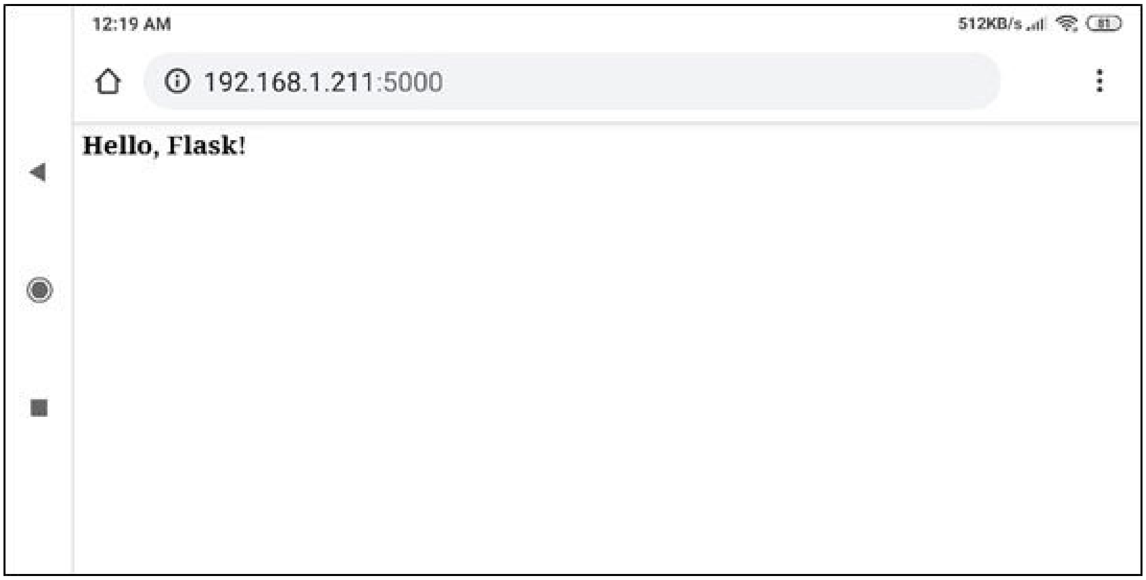
Figure 3.2: *Hello Flask*

Another way of running Flask application using Python -m switch as follows:

The result will be same, though. To stop the server, press

For now, this server is accessible from the same computer on which Flask is installed and not from any other system in the network. To make the localhost publicly available, specify 0.0.0.0 as --host parameter:

You need to find out the IP address of your localhost and use the same to access it from other devices on the same network. My system shows the IP address as With my mobile phone, I can access localhost running on my computer as follows:



Hello, Flask!

Figure 3.3: *Externally visible server*

Ensure that your system (on which localhost is running) and mobile are connected to the same network.

Debug Mode

You may have noted that the **Debug mode: off** in server's log in the console. When an application is still in the development stage, you are invariably required to make frequent changes. However, to see the effect of any changes, you need to interrupt the server and launch it again, which is annoying, to say the least.

This is where setting the debug mode to **on** helps. The debug mode is set to on by assigning the FLASK_ENV variable to

As you can see, the debugger operation of the server is active. Any changes to the application script will be automatically detected by the server. To test, make some change in the hello.py script. The console log immediately displays the following:

```
* Detected change in 'C:\\flaskenv\\hello.py', reloading
```

You should refresh the browser page to confirm the change. If any error is encountered while executing the script, the browser as well as the command shell report the error.

Let's deliberately induce an error in our script by removing the trailing quotation mark of the string in the return statement. This error message is reported:

```
File "C:\flaskenv\hello.py", line 6
return '
```

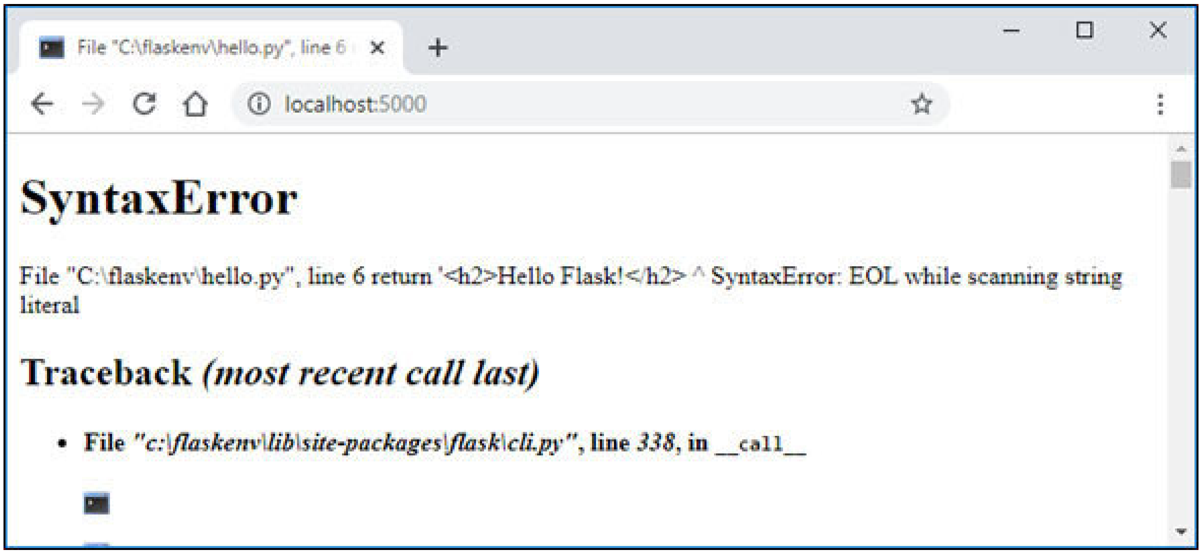
Hello Flask!

^

SyntaxError: EOL while scanning string literal

Setting FLASK_ENV to development activates debugger and reloader and enables the debug mode of Flask application. This debug mode may be separately set by another environment

Just as on the shell window, debugger output is also rendered on the browser. Here's a browser window showing partial debugger output:



SyntaxError

File "C:\flaskenv\hello.py", line 6 return '<h2>Hello Flask!</h2> ^ SyntaxError: EOL while scanning string literal

Traceback *(most recent call last)*

- **File *"c:\flaskenv\lib\site-packages\flask\cli.py"*, line *338*, in** `__call__`

Figure 3.4: Flask debugger

This was a gentle introduction to Flask framework. One can straightway make out from the minimal Hello Flask application that it depends on the Werkzeug toolkit, and its CLI is based on the Click package. We also learned how to use the debug mode and its advantage.

Conclusion

In this chapter, we had a brief overview of Flask's dependency packages, namely Werkzeug, Jinja2, and Click. We then learned how to install Flask and run Hello Flask application from command line interface.

The next chapter covers the Flask application class in further detail. It also explores the mapping of Python functions to URL rules. We shall also learn how to add data to a URL and pass it to a function that is bound to it.

CHAPTER 4

URL Routing

Introduction

Modern web applications employ routes instead of file-based URLs to control the flow of application. Route-based URLs are easy for the user to remember and revisit.

Flask framework also uses routing technique. In this chapter, we shall explore ways to form routing rules and learn how to dynamically redirect the client from one URL to another. This chapter also explains the construction of URL by adding variable parts.

Structure

Flask script

Application object

Route decorator

The add_url_rule() method

The url_for() function

The redirect() function

Variable parts in URL

Objectives

After completing this chapter, you should be able to write a Flask application with multiple routes with variable parts.

Flask Script

In the previous chapter, we learned how to run Hello Flask application, but we know very little about how the code works. So, let's discuss it in more detail:

```
#hello.py
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_flask():
return 'Hello Flask!'

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

Observe the preceding code carefully. Is it the same that we used in the previous chapter? Well, not really. The if block in the script wasn't there earlier. Why has it been added now?

First of all, running this script as a normal Python script also starts Flask development server, as does the Flask command that we used in the previous chapter. In fact, this had been the way to serve Flask application right from its initial version. The integration of Flask with the Click package to provide **Command Line Interface** in the form of Flask command has been introduced in its 0.11 version.

Although both ways (using the run() method and Flask run command) are available, using CLI is recommended and using run() is discouraged because of its unpredictable behavior of automatically reloading the code when in debug mode.

Behavior of Flask server can be configured by the following parameters of the run() method:

The hostname to listen on. The default value is 127.0.0.1 (also known as localhost). To make the server visible for external requests, set this to

Defines the port of the web server, which is 5000 by default.

This parameter enables or disables debug mode. To enable debug mode, set it to

Application object

Take a look at first two lines of the preceding script

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

First line imports the Flask class, and the second one creates its instance. This object is really the WSGI application object. All the view functions, routes, and configuration, etc. are available to this object.

Note the use of __name__ as argument to the Flask() constructor. You may be aware that __name__ is a module attribute that returns the context in which the Python interpreter is being executed. When Python is running in a shell, __name__ returns

```
>>> __name__
'__main__'
```

Accessing the value of __name__ from within an executable script also returns the same value, but the __name__ attribute of imported module returns its name:

```
>>> import hello
>>> hello.__name__
'hello'
```

Passing the __name__ parameter lets the server find the resources on the filesystem. Various flask extensions also use this parameter for debugging purpose.

If the entire Flask application resides in a single script (with .py extension), this parameter is given. However, if the application resides in a Python package, the package name has to be given as the parameter to Flask constructor.

For example, hello.py is present in a package named (i.e. the file is then the correct creation of Flask instance is as follows:

```
app=Flask('flaskapp')
```

We have already used the run() method defined in the Flask class (although it is not recommended to be used and Flask

CLI should be used to start the server instead). Other important methods of the Flask class will be explained in the following sections of this chapter as well as in subsequent chapters.

Route Decorator

In the next three lines of above code we register the view function hello_flask() to the URL / with the help of the route() method of the Flask class:

```
@app.route('/')
def hello_flask():
return 'Hello Flask!'
```

This effectively maps the hello_flask() function to the given URL, which means the mapped function will be triggered whenever / is requested. Note the presence of the @ symbol before It tells that route() is a decorator function. Here, we need to understand two things: what is a route, and what is a decorator?

In a classical web application, a URL is mapped to a certain script on the web server. Additionally, data is sent to it as a request in the form of a query string. A typical call to a Python CGI script could be as shown here:

http://example.com/cgi-bin/test.py?name=xyz&age=20

So, this is a file-based URL with the name of the script, followed by the query string, separated by the ? character. Modern web applications are built with frameworks (such as Flask that we are using). They use routing technique instead of file-based URLs. The route-based URL is easy for the user to remember and can be accessed directly without having to reach it by following hyperlinks from the home page.

The above URL can be as follows if a routing technique is used:

http://example.com/test/xyz/20

In this case, /test is the endpoint, whereas xyz and 20 being the data to be passed to the mapped function. The URL is mapped to a certain function that parses and processes the received data. We shall see how this is done later in this chapter.

What is a decorator? A detailed discussion on Python decorators is beyond the scope of this book, but here's a brief idea about it:

To begin with, a function in Python is treated as first class object. Just like any built-in data type (such as string, number, list, tuple, and a function can also be used as an argument to another function. Similarly, a function can have another function as its return value as well. What's more, the definition of a function can be wrapped inside another function.

A decorator is a function that receives another function as argument, calls it inside its wrapped function, and returns its modified version. The prototype of route decorator in the Flask class has following parameters:

@app.route(rule, endpoint, methods)

The URL rule as string.

For the above URL rule. By default, it is the function defined just below it.

A list of HTTP methods this rule responds to etc.).

Coming back to the hello_flask() function, it appears to be a normal Python function that returns a string However, the route decorator maps it to rule parameter and transforms it so that the string is returned as HTTP response to the client browser.

There are a number of decorator functions in the Flask library. The Blueprint class for example also uses route decorator. We shall come across it in a later chapter.

The add_url_rule() method

The purpose of this method in Flask class is exactly the same as that of the route() decorator—to register a view function for a given URL rule. However, registration happens after the definition of a view function. Remember that in case of a decorator function, the function to be decorated must be defined in the immediate next line.

In the following code, we define two view functions and register each of them with separate URL rules with the add_url_rule() method:

```
#urlrule.py
from flask import Flask
print (__name__)
app = Flask(__name__)

#@app.route('/hello')
def hello():
return 'Hello World!'
```

```
def welcome():
return 'Welcome to Flask Framework'

app.add_url_rule('/hello', 'hello', hello)
app.add_url_rule('/welcome','welcome',welcome)
```

One advantage of using add_url_rule() is that you can define all rules together. This is especially useful if the application contains many view functions defined at different places in the code, even in different classes.

Run the above script as Flask application:

```
(flaskenv) C:\flaskenv>set FLASK_APP=urlrule.py
(flaskenv) C:\flaskenv>flask run
```

As you would expect, produces Hello World! message in the browser and http://localhost:5000/welcome produces Welcome to Flask Framework! message.

The add_url_rule() method is defined with the following parameters:

```
add_url_rule(rule, endpoint, view_func, methods)
```

A string representing URL rule.

The endpoint for the above-mentioned URL rule. It is usually the name of the view function.

The function be called to serve a request to the provided endpoint.

The list of http methods this rule should be limited to etc.).

A view function can be registered with more than one URL rules, either through route decorator, the add_url_rule() method, or both.

Following snippet binds a function with two rules:

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

The same binding effect is achieved by the following code:

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

Following code is also equivalent:

```
def hello():
return 'Hello World!'

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

Under all these three variations, the URLs **http://localhost:5000** as well as **http://localhost:5000/hello** produce Hello World! message in the browser.

Interestingly, one view function can also have multiple URL rules, as demonstrated in the following example:

```
@app.route('/greeting')
@app.route('/welcome')
def welcome():
return '
```

Welcome to Flask Framework

The url_for() Function

This function is defined in flask module and not as a method in the Flask class. It returns a string corresponding to URL rule to which a specified view function is registered:

```
url_for(endpoint, **args)
```

In the route decorator method or add_url_rule() method, endpoint is a string usually bound to a view function of the same name. Hence for the hello() view function defined as above, url_for('hello') will return

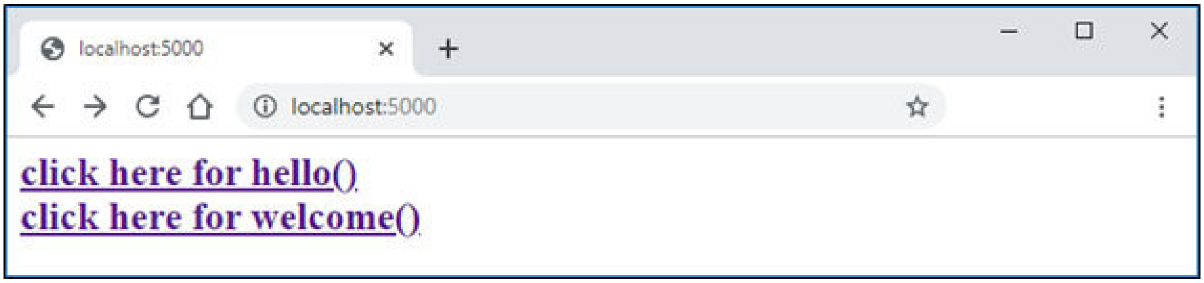
Let us consider a more elaborate example to understand how url_for works. In addition to the hello() and welcome() view functions, the following script defines one more view function which is registered to / URL:

#urlfor.py

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

```
@app.route('/')
def index():
url1=url_for('hello')
url2=url_for('welcome')
return "href={}>click here for hello()".format(url1)+\
"
href={}>click here for welcome()".format(url2)

def hello():
return "Hello World!"

def welcome():
return 'Welcome to Flask Framework'

app.add_url_rule('/hello', 'hello', hello)
app.add_url_rule('/welcome','welcome',welcome)
```

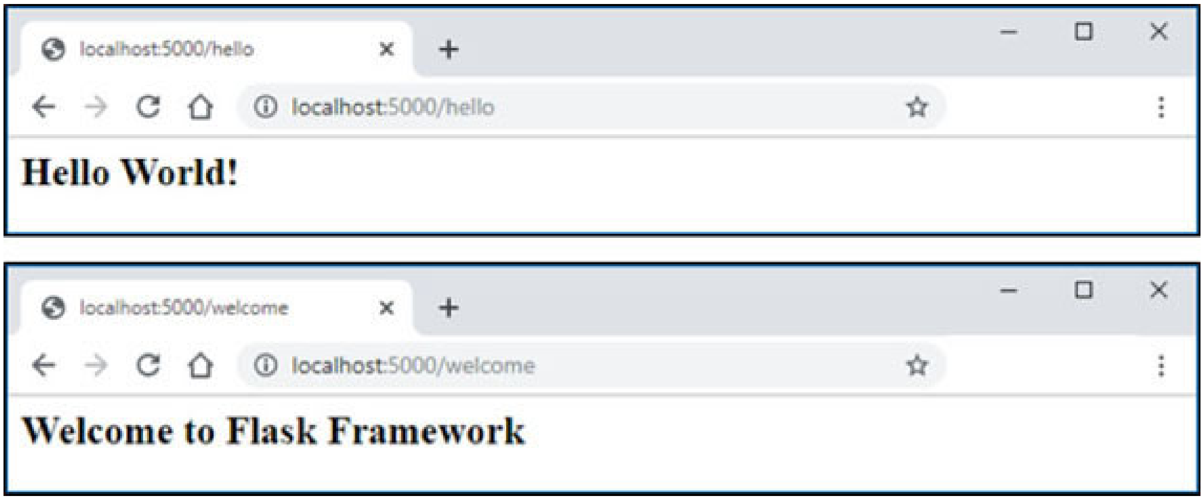
The index() function fetches URL strings of the hello() and welcome() functions and returns a string that constructs a hyperlink around them. Visit **http://localhost:5000** in the browser after starting the Flask server:



click here for hello()
click here for welcome()

__Figure 4.1:__ A url_for() example

Following figure shows the browser output when these links are followed:



Hello World!

Welcome to Flask Framework

__Figure 4.2:__ An add_url_rule() example

The url_for() function is also capable of receiving any number of keyword arguments. Each argument forms the variable part of URL rule and is appended to it as query argument. Later in this chapter, we shall learn how to build a URL rule dynamically by adding variable parts.

The redirect() Function

This is another function in the flask module. As the name suggests, it redirects the client browser to another endpoint from inside a view function. The redirect() function returns a response object. When called, the client is redirected to specified location. This function has the following signature:

redirect(location, status_code, response)

The response should redirect to a specified location.

The HTTP status code. Defaults to

Returned response object belongs to the Response class. The default is

The redirect() function considers the status code to be 302 if the parameter is not used. HTTP status code 302 indicates a found status. Other acceptable status codes are:

are: are:
are:
are: are:
are: are:
are: are:
are: are:
are: are:
are: are:

Table 4.1: *HTTP redirect codes*

In the following definition of / URL rule, the associated view function redirects the client to another rule

```
@app.route('/')
def index():
return redirect (url_for('hello'))
```

Another related function in Flask API is the abort() function, which causes the called view function to terminate with an error code. Some the frequently used HTTP error codes are:

are: are:
are:
are:
are: are:
are: are: are:
are: are:

Table 4.2: *HTTP error codes*

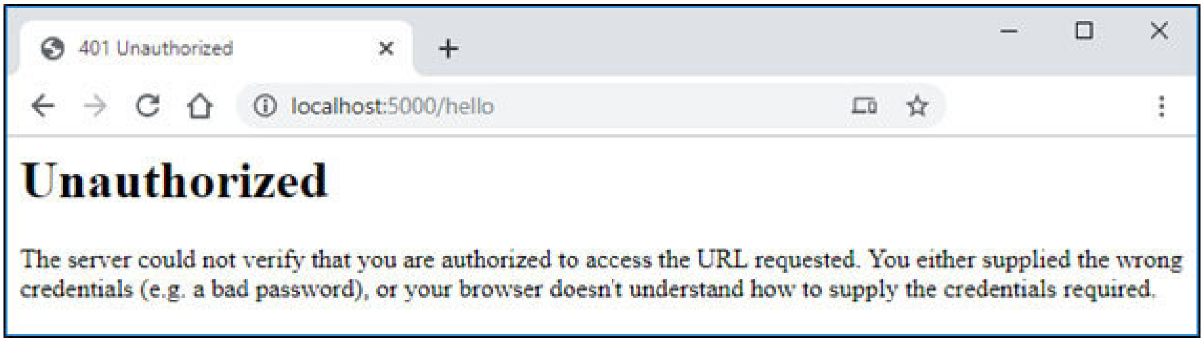
Let's insert call to abort() with error code 401 inside the hello() function and visit the localhost:

```
@app.route('/')
def index():
return redirect (url_for('hello'))
def hello():
abort(401)
return ''
```

Hello World!

"

Following error page is displayed:



Unauthorized

The server could not verify that you are authorized to access the URL requested. You either supplied the wrong credentials (e.g. a bad password), or your browser doesn't understand how to supply the credentials required.

Figure 4.3: *HTTP error code*

The redirection of URLs shows peculiar behavior. Consider the two rules defined here:

```
@app.route('/hello/')
def hello():
return "
```

Hello World!

```
''
@app.route('/welcome')
def welcome():
return '
```

Welcome to Flask Framework

'

The first rule has a trailing slash to the URL. The Hello World message will be displayed when the browser's URL is with or without trailing slash. However, the second rule doesn't have a trailing slash. In this case, the corresponding message is rendered only when Flask encounters a URL without slash. With a trailing slash, a 404 Not Found error page is flashed.

Variable Parts in URL

All the view functions mentioned so far in this chapter are without any parameter or argument, so one wonders if a function registered with a certain URL rule can have any parameter, and if yes, how it is passed? The answer is yes, a view function can be defined with one or more parameters, and their values are fetched from the URL itself.

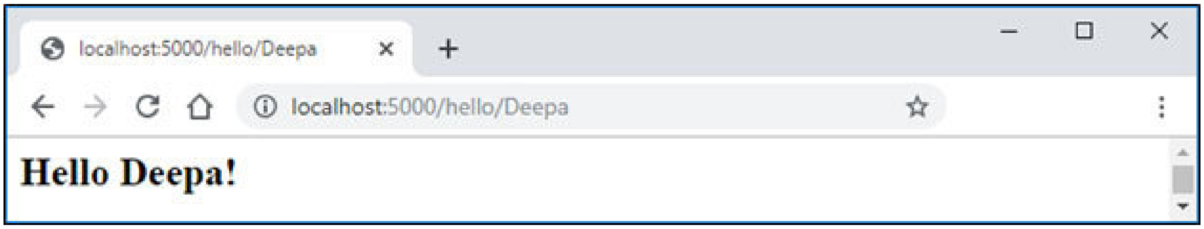
Suppose the URL in the browser is and the registered view function hello() should render the message Hello In such a case, the endpoint of URL rule is and the remainder of the URL is picked up in a place holder variable put inside angular brackets. So, the URL rule will be This variable name is then used as formal argument in the definition of the hello() function. The runtime value of the name parameter is passed to the function for further processing:

```
@app.route('/hello/')
def hello(name):
return ''
```

Hello {}!

".format(name)

In this case, the URL supplied string is inserted in the Hello message to be rendered as the response:



Hello Deepa!

Figure 4.4: *String variable part*

You can include variable parts in the add_url_rule() method as well:

```
def hello(name):
return "
```

Hello {}!

''.format(name)

app.add_url_rule('/hello/', 'hello', hello)

A URL rule can have more than one variable part, each separated by / and identified by a unique variable name. The variable defined within angular brackets is of string type by default. To treat the variable part of URL as of any other type, we have to use the appropriate converter in the format:

format: format: format: format: format: format:
format: format: format: format: format:
format: format: format: format: format: format:
format: format: format: format: format: format: format: format: format: format: format:
format: format: format:

Table 4.3: *Variable converters*

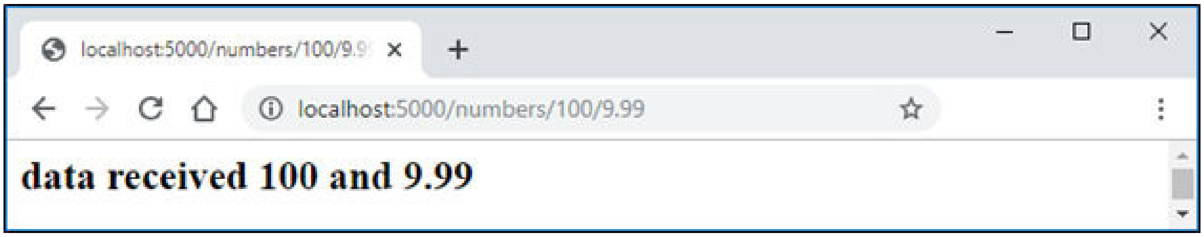
Following rule has an integer and a float variable part:

```
@app.route('/numbers//')
def  numbers(a,b):
return  '
```

data received {} and {}

```
'.format (a,b)
```

The browser shows the following output when values 100 and 9.99 are passed:


localhost:5000/numbers/100/9.9
localhost:5000/numbers/100/9.99

data received 100 and 9.99

Figure 4.5: Numeric variable parts

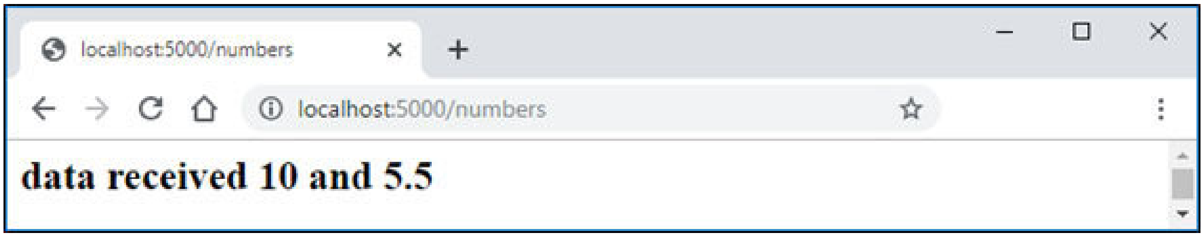
You can also set default values of variables by using the defaults parameter. It is a dictionary object with variable as key. If the URL doesn't provide values explicitly, defaults are added as variable part of the URL:

```
@app.route('/numbers', defaults={'a':10, 'b':5.5})
@app.route('/numbers//')
def numbers(a,b):
return '
```

data received {} and {}

'.format (a,b)

Try giving only the endpoint as the URL in the browser. Defaults are displayed as follows:



data received 10 and 5.5

Figure 4.6: *Variable parts with defaults*

The **UUID Unique** as variable part is used when a certain API key is to be passed, the view function needs to generate a token, or some authentication action is to be performed. URL path as variable part will be needed when a resource from the server's file system is to be accessed. We will need this in a subsequent chapter when we discuss static files in Flask application.

Conclusion

This chapter described how we can define routes with URL rules and register view functions. We also learnt various redirection methods and how to form URL dynamically adding variable parts. The next chapter will introduce another important constituent of Flask Template.

CHAPTER 5

Rendering Templates

Introduction

Rendering raw HTML code constructed by Python functions is tedious. Precisely for this purpose, Flask encapsulates the functionality of the Jinja2 template library.

In this chapter, we shall learn various powerful features of the Jinja2 package to render dynamic content generated by view functions registered with routes of a Flask application.

Structure

HTML tags

Flask application structure

The render_template() function

Conditional statements

Loops in template

Macros

Filters

Template inheritance

Objectives

After completing this chapter, you should be able to enhance your Flask application with beautifully designed HTML templates.

HTML Tags

As mentioned earlier, Jinja2 templating library is one of the important dependencies of Flask. In fact, we had worked briefly with jinja2 package in *Chapter 2:* In this chapter, we shall find out how Flask encapsulates Jinja2 to make it easily possible to render dynamic web content. Just to recap, a web template engine merges a static web page having place holder variables with a data source to generate multiple web pages. Now, this is something similar to the mail-merge feature that's usually present in word processing utilities. The principle of templating is well illustrated in *figure 3.1 3: Flask*

Before we proceed, a brief note on the structure of web page here is in order as a detailed discussion of HTML is beyond the scope of this book. A web page is composed using HTML scripting language, which comprises of a large number of HTML tags. We can think of tag as a keyword in any general purpose programming language such as Java and Python. Tags are placed in angular brackets – for example Each tag has a predefined behavior that is implemented by browser software.

Most of the tags have an opening and a closing tag. For example, and Hence, in any script, the opening tag must be matched by a corresponding closing tag. Also, closing tags must appear in such a way that the last opening tag must be closed first.

Any HTML script should begin with a doctype directive, as shown below:

```
HTML>
```

It may be noted that doctype is not a HTML tag, but it is recommended to be used to let the browser know of the type of document. The HTML script proper must start with the tag and end with the tag. Remainder of the script is divided in two sections: HEAD and

```
HTML>
. . . .
. . . .
```

Whatever that is rendered on the browser's page comes from the BODY section. The HEAD section doesn't contain any renderable text. Instead, JavaScript and stylesheet scripts are included in this section if required. We shall be using JavaScript and CSS files in the next chapter.

Apart from the above-mentioned tags, the following tags are used frequently:

-

to

- To format the heading of decrementing font size

- To display a numbered list

- To display a bulleted list

- - For items in a list

- To display data in a tabular format

- To render a row in the table

```
{% if student[1]>=50 %}
{% else %}
{% endif %}
{%- endmacro %}
```

The result macro defined above has one argument, which is a tuple whose two items are rendered in two columns of a table. The third column is filled conditionally Let's call this macro for each item in the students list received from the application's view function. Each item happens to be a tuple:

html>

style='text-align:center'>Mark List

- To place the value of each cell of a table
- To define a section in the document

- To create a form to collect user input

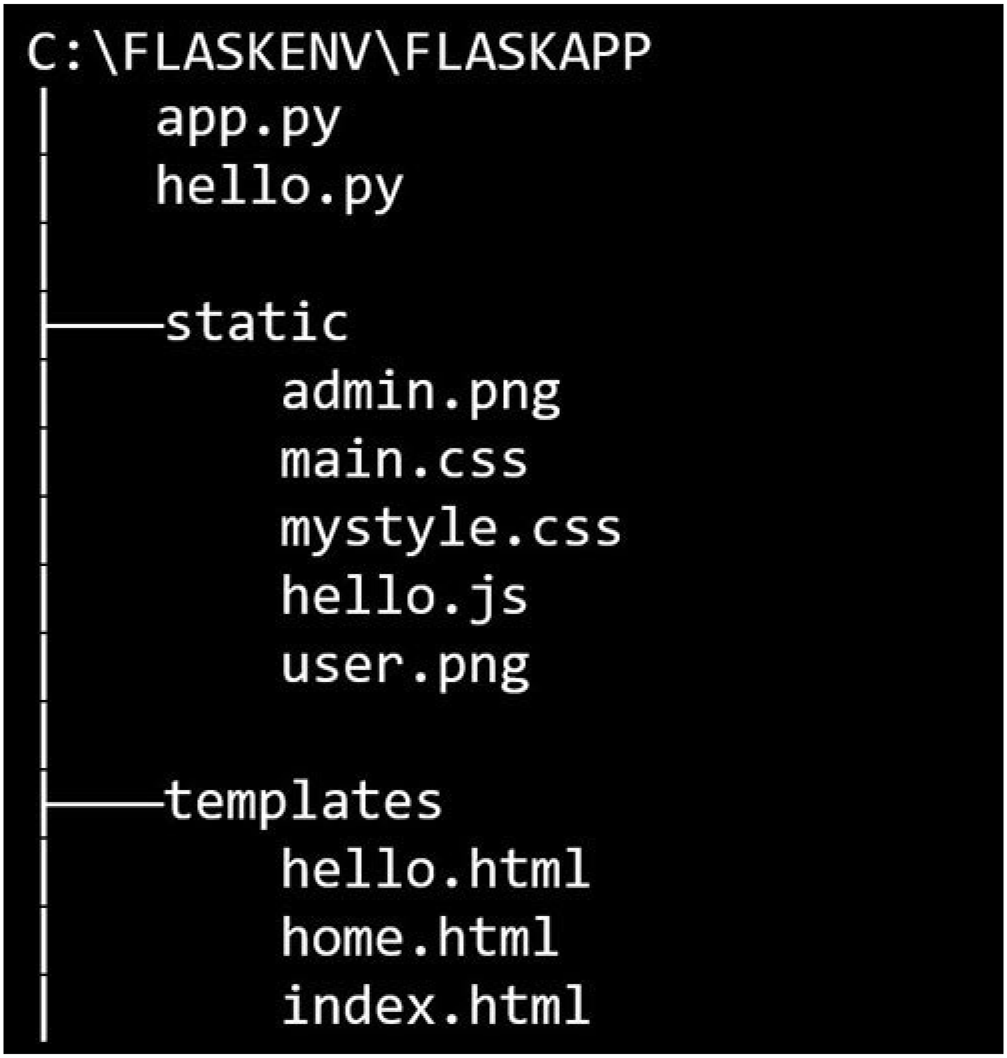
To render a single line text box

To provide a clickable button that submits form data to a URL

It is assumed that the reader is familiar with the basics of HTML (and JavaScript/CSS for the next chapter).

Flask application structure

Flask recommends a typical method of organizing various resources in an application. Assuming that the entire application is to be stored in a folder its directory structure should be as follows:



```
C:\FLASKENV\FLASKAPP
│    app.py
│    hello.py
│
├───static
│        admin.png
│        main.css
│        mystyle.css
│        hello.js
│        user.png
│
├───templates
│        hello.html
│        home.html
│        index.html
```

Figure 5.1: *Flask application folder structure*

As you would expect, the application code may be spread over one or more Python scripts (having the .py extension). These files are stored under root of application folder in this case). The templates, which are essentially web pages, should be placed in the templates subfolder.

Web pages may be using certain static assets, such as images and stylesheets. They are placed in the static folder. The next chapter deals with static files in detail.

However, Flask is flexible enough to designate any folder to contain templates while configuring Flask application object:

```
app=Flask(__name__)
app.template_folder='path/to/newfolder'
```

The render_template() Function

This function is primarily responsible for rendering HTML content of a specified web page as HTTP response to the client. Let's construct a basic web page with the following script and store it in the templates folder:

#hello.html

```
html>
```

Hello Flask!

Use this file as argument to the render_template() function to be called in the view function, as follows:

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

@app.route('/')
def hello():
return render_template('hello.html')
```

Note that the render_template() function should be imported in the beginning. Run the above script from command prompt and point the browser's URL to You should see Hello Flask! displayed in the browser.

Nothing great about it, you would say. But wait, the real power of this function is to pass variable data to the template web page.

The render_template() function has the following signature:

```
render_template(template_page, **context)
```

First argument is, of course, the name of the HTML page. The second argument, **context contains one or more keyword arguments, the values of which are passed to the template for consumption.

How does a template web page make use of these context variables? A web template is actually a web page interspersed with one or more blocks of Jinja code. As we saw earlier, Jinja code blocks within HTML script are identified by the following delimiters:

{% %} - Statements

{{}} - Expressions to print to the template output

{# #} - Comments that are not included in the template output

- Line statements

Let's first modify the / route registered with the hello() function to contain a variable part:

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

Variable part in the URL is passed to the underlying hello() view function, which, in turn, is used as a second argument in the return_template() function. We now need to modify hello.html to insert the string provided by the render_template() function in the

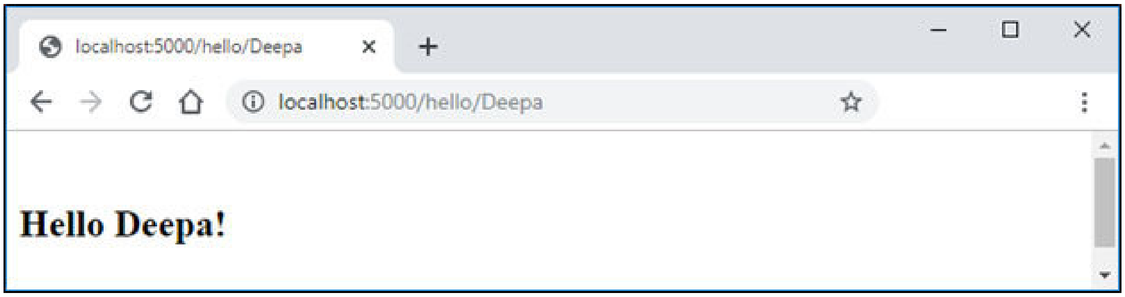
••

tag to dynamically generate the Hello message:

html>

Hello {{name}}!

Restart Flask's development server (as a matter of fact, you may not need to restart if you have already set the debug property of Flask as true!). Enter **http://localhost:5000/hello/Deepa** as the browser's URL:



Hello Deepa!

Figure 5.2: *Jinja2 template*

Go ahead and change name part of the URL to confirm that the appropriate message is rendered.

Conditional Statements

As mentioned earlier, Jinja2 allows embedding conditional statements inside HTML script. The general syntax of conditional block is as follows:

```
{% if TrueExpr %}
HTML block
{% else %}
HTML block
{% endif %}
```

To demonstrate the use of conditionals in a template, let's first modify our /hello route to accept two variable parts and in the URL and pass them to the template to be rendered with the help of its registered view function:

```
@app.route('/hello//')
def hello(name, age):
return render_template('hello.html', name=name, age=age)
```

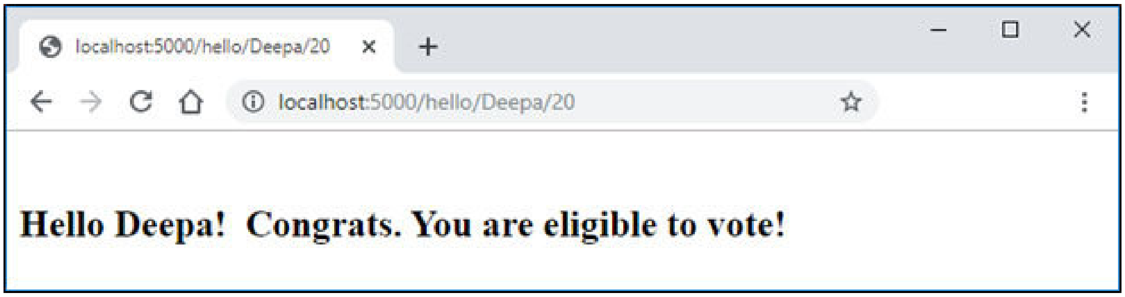
Obviously, the URL in the browser's address bar will be something like In turn, two variables are used as value of keyword arguments to be passed to the template

Here, the idea is to generate different messages for age parameter >=18 or not. We shall have to modify the template accordingly:

html>

```
Hello {{name}}!

{% if age>18 %}
Congrats. You are eligible to vote!


{% else %}
Unfortunately,You are not eligible to vote.
{% endif %}
```

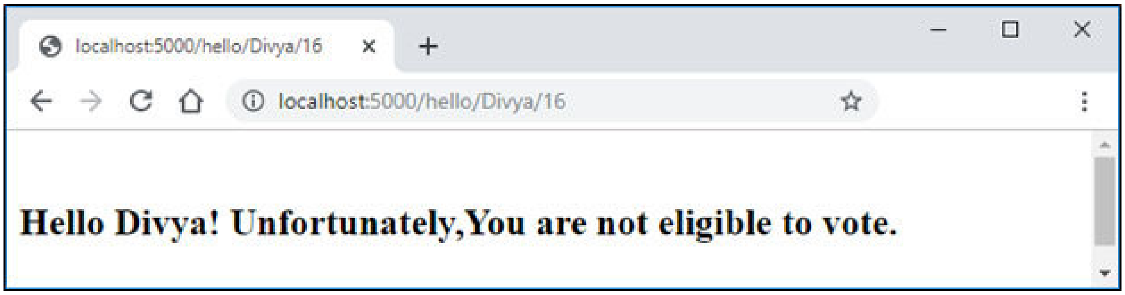
Refresh the browser (restart the server if needed). The browser output for two different use cases is as follows:



```
Hello Deepa!  Congrats. You are eligible to vote!
```

Figure 5.3: *Jinja2 template with variable data*

If age happens to be less than 18, the browser displays the following output:



Hello Divya! Unfortunately,You are not eligible to vote.

Figure 5.4: *Jinja2 template with variable data*

Note that Jinja templating language also supports the elif statement (just as in Python) if you need to insert cascading conditional statements.

Loops in Template

As discussed in *Chapter 3: Flask* Jinja2 supports inserting a looping construct within HTML script. Looping block is placed between the {% for %} and {% endfor %} statements. for statement is much like Python's for keyword, in the sense it iterates over a collection of objects. A general usage of Jinja2 loop follows this structure:

```
{% for item in collection %}
HTML block
{% endfor %}
```

First, we shall define a view function that passes a certain collection object (such as list, tuple, dict, etc.) to a web template. In the following Flask script, a list of strings representing the names of programming languages is passed to the langs.html template:

```
@app.route('/')
def index():
langs=['C', 'C++','Java', 'Python', 'PHP']
```

```
return render_template('langs.html', langs=langs)
```

The template itself traverses the langs object with the for..endfor construct and renders its items in the form of unordered (bulleted) list on the client's browser:

#langs.html

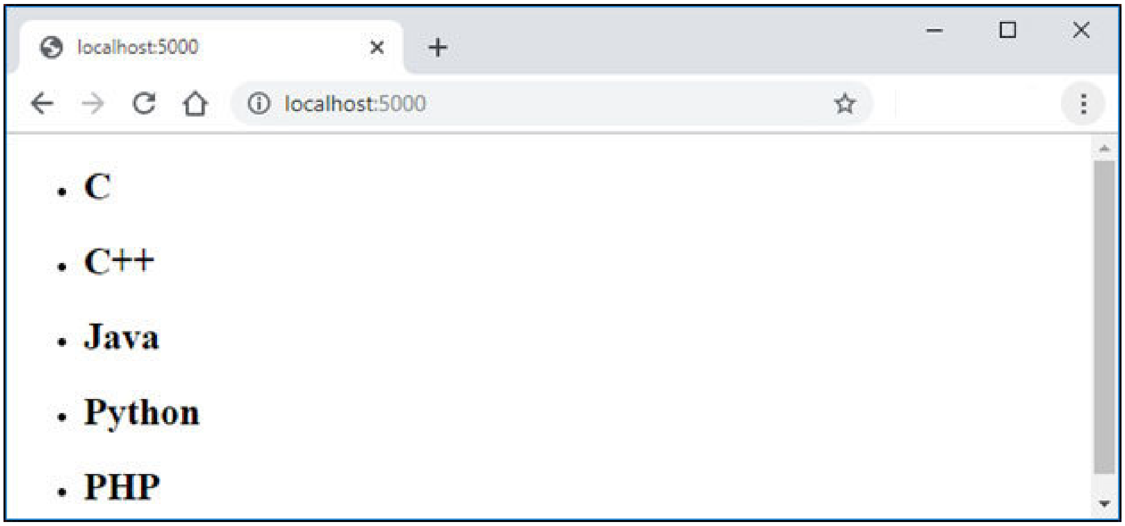
```
html>

{% for lang in langs %}
```

-

{{lang}}

{% endfor %}

Here's what your browser should display after running the server and visiting



- C
- C++
- Java
- Python
- PHP

Figure 5.5: *Loop in Jinja2 template*

Such loops are especially useful when tabular data fetched from an underlying database has to be rendered on a web page.

Macros

The term *macro* is often encountered in programming jargon. Most popular usage of macro is in VBA, the embedded programming language for MS Office applications. A macro is essentially a user defined function that can be called as and when required from rest of the code. It is a piece of reusable code that implements the principle of **DRY Repeat** while developing a programming solution.

In a macro is defined as per the following syntax:

```
{%macro macroname -%}
HTML block
{%- endmacro %}
```

Note that the HTML block may contain one or more conditional or looping constructs as defined in Jinja's templating language. A macro may also be defined with arguments if required.

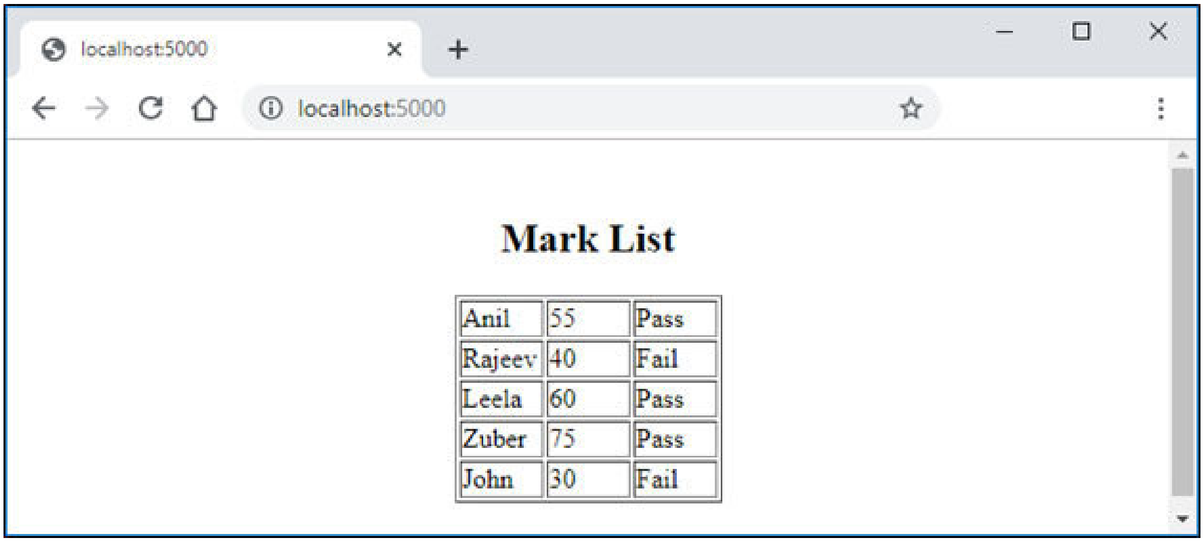
Typical use case of macro in Flask template is when you need to apply common formatting and/or processing logic repeatedly on variable data. Illustrated below is a simple example. To begin with, the view function of Flask application sends a list of tuples to a template Each tuple has two items, say name and marks:

```
@app.route('/')
def result():
students=[('Anil',55), ('Rajeev', 40),
('Leela', 60), ('Zuber', 75), ('John', 30)]
return render_template(langs.html',
students=students)
```

The template is required to render each tuple as a row in HTML table. Moreover, a third column should display result (pass/fail) depending on marks>=50 or not. For this purpose, a macro is defined as follows:

```
  {% macro result(student) -%}

{{student[0]}}                                  {{student[1]}} Pass Fail
align="center" border=1>
width="50">
width="50">
width="50">
{% for student in students %}
{{result(student)}}
{% endfor %}
```

Here's the browser output as we run the above Flask application:



Mark List

Anil	55	Pass
Rajeev	40	Fail
Leela	60	Pass
Zuber	75	Pass
John	30	Fail

Figure 5.6: *Using macro in Jinja2 template*

Filters

The Jinja2 library has a number of built-in filters. In addition, it is possible to define custom filters too. Jinja2 filter is, in a way, similar to Python function (built-in or user defined) with a peculiar difference in syntax. Unlike a function, the parameter is not put in parentheses but appears before it and is separated by | symbol generally known as pipe. These filters are applied to an expression or a statement placed inside delimiter symbols:

{{variable|filter}}

First, let's learn the usage of some built-in filters and then try and build a custom filter. Here are some of the filters to be used with collection types such as string, list, and tuple:

Arranges the items in reverse order

Returns the first item

Returns the last item

Rearranges items of the collection in ascending order

The view function defined here sends a string and a list to the template:

```
@app.route('/')
def index():
list=[5,8,4,6,7]
string='Hello World'
return render_template('hello.html', list=list, string=string)
```

The following hello.html template implements the above-mentioned filters:

```
html>
```

```
original string: {{string}}
string reversed: {{string|reverse}}
original list: {{list}}
first item: {{list|first}} last item: {{list|last}}
sorted list: {{list|sort}}
```

The output of above template should be as follows:

```
original string: Hello World
string reversed: dlroW olleH
original list: [5, 8, 4, 6, 7]
first item: 5 last item: 7
sorted list: [4, 5, 6, 7, 8]
```

Some of the filters to be used exclusively with string object are as follows:

Converts the first character to uppercase if not already

Returns a list of comma-separated characters in the given string

Returns the given string with all characters in lowercase

Returns the given string with all characters in uppercase

The first character of each substring (separated by white space) to uppercase, leaving the others as lowercase

The template code that uses the above-mentioned filters is:

html>

```
original string: {{string}}
list of characters: {{string|list}}
convert to lowercase: {{string|lower}}
convert to uppercase: {{string|upper}}
convert to titlecase: {{string|title}}
```

As a result, the browser's expected display is:

```
original string: Hello World
list of characters: ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
convert to lowercase: hello world
convert to uppercase: HELLO WORLD
convert to titlecase: Hello World
```

The next set of filters works with numbers:

Returns absolute value disregarding the sign.

Converts the given number to float representation.

Converts a number to integer. Conversion is, by default, in the decimal number system. To apply other number systems (such as binary, octal, or hexadecimal), use base 2, 8, or 16. For example, int(0,2) converts a binary number represented as a string, to integer.

The view function for this exercise is:

```
@app.route('/')
def index():
a=-10
b=20
c=3.55
d='1010'
return render_template('hello.html', a=a,b=b,c=c,d=d)
```

We design the template as given here:

```
html>
```

```
a: {{a}} absolute value: {{a|abs}}
b: {{b}} converted to float: {{b|float}}
c: {{c}} converted to int: {{c|int}}
binary: {{d}} converted to integer: {{d|int(0,2)}}
```

The output rendered to the browser is:

```
a: -10 absolute value: 10
b: 20 converted to float: 20.0
c: 3.55 converted to int: 3
binary: 1010 converted to integer: 10
```

In addition to the number of built-in filters (Flask library has about 50 built-in filters), you can define a custom filter. The Flask application class provides a template_filter('filter_name') decorator. Define your own logic in the registered view function and apply filter_name in the template. Here's a simple example.

Given below is an isodd() view function that returns true/false depending upon whether a number argument is odd. This function is decorated by the template_filter() decorator, as follows:

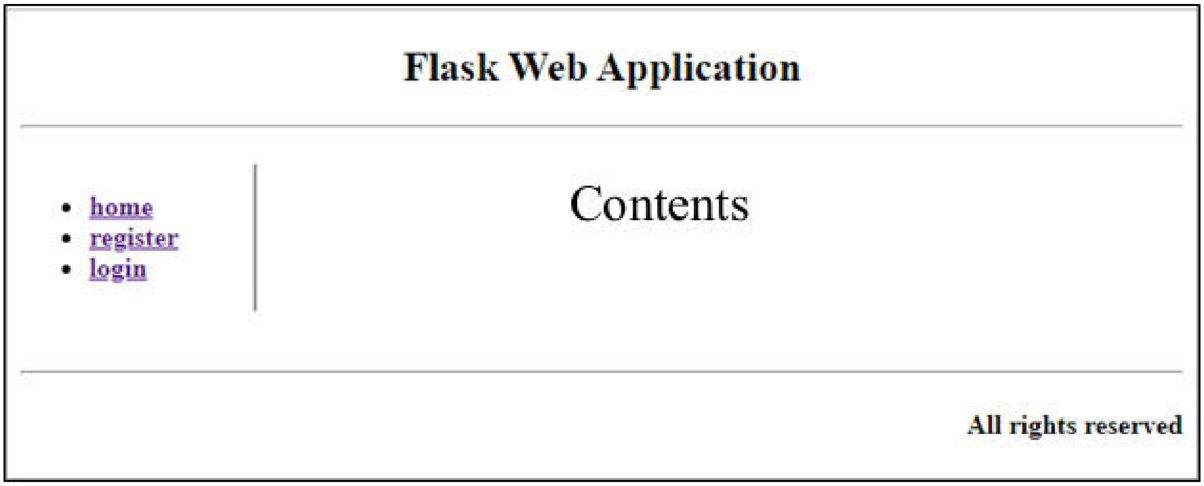
```
@app.template_filter("isodd")
def isodd(x):
if x%2==1:
result=True
else:
result=False
return result
```

We can now apply this isodd filter in our template. For instance, {{x|isodd}} renders True if x happens to be 35 and False for 50 as the value of

Template Inheritance

A typical web application will naturally have multiple views, and each view will render a different template (or same template with different data). However, from the user's perspective, it is desired that the output rendered by each view must follow a uniform pattern or look. For example, each response from the server should have header, navigation bar, and footer of similar formatting properties even if the output of each view may be different.

Let's say our application has three URL routes registered with three views. We want to design the template in such a way that each view should have a page header (website header you may call), a footer, and a side bar with links to the variable content displayed to its right. The schematics are as illustrated here:



Flask Web Application

- home
- register
- login

Contents

All rights reserved

Figure 5.7: *Jinja2 base template structure*

This is exactly in such situations, the fascinating feature of Jinja2 library called template inheritance comes in extremely handy. The concept of inheritance in Jinja2 is very similar to inheritance in object-oriented programming. Let's see how.

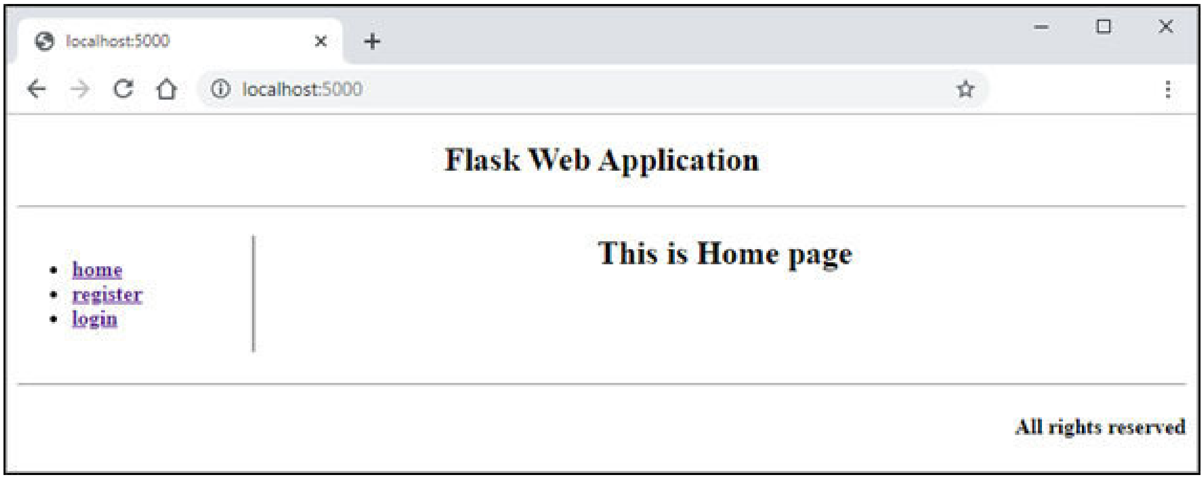
Just as a base class (in Python) defines attributes and methods and makes them available to the inherited class, we need to design a base template that provides an overall skeleton for other templates. We know that an inherited class can override methods defined in the base class. Here, along with the common structure, the base template also marks dummy blocks. Child template inherits the common structure and overrides the blocks to provide the respective content. Such blocks are marked with the block – endblock construct, as follows:

```
{% block block_name %}
{% endblock %}
```

There can, of course, be more than one such block in different places. Each one should be provided a unique identifier. In this case, we expect a single block that will be populated by three different views that we are planning to include in our application. This block, named is placed in the right div section of the middle row. The other three sections are fairly static. The HTML code for our base template is as follows:

#parent.html

```
html>
```

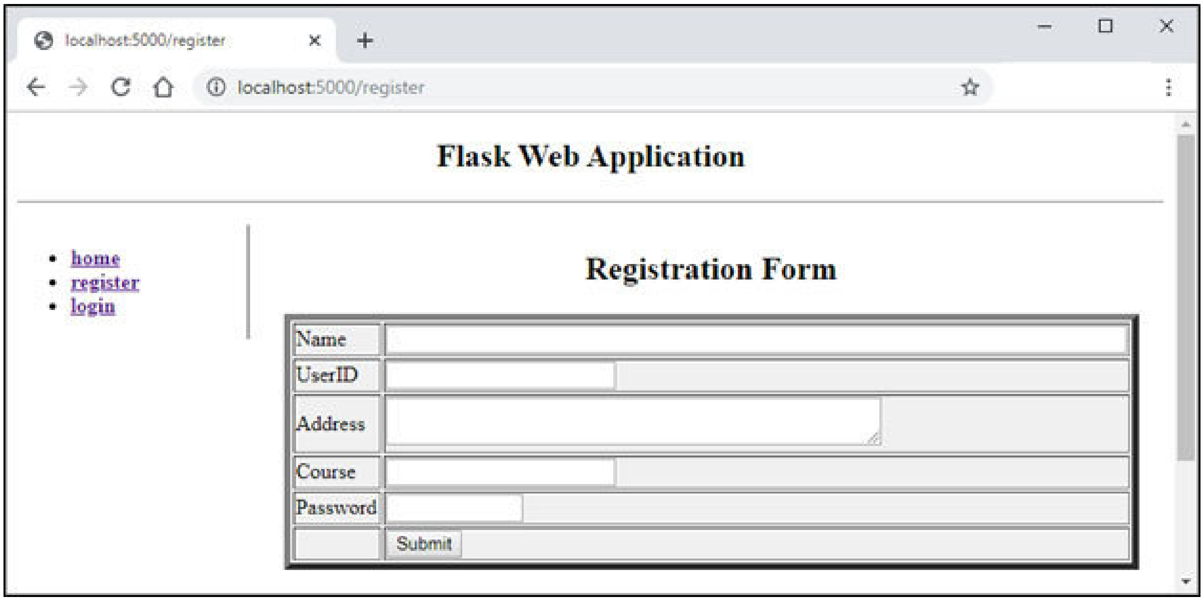


Flask Web Application

- home
- register
- login

This is Home page

All rights reserved

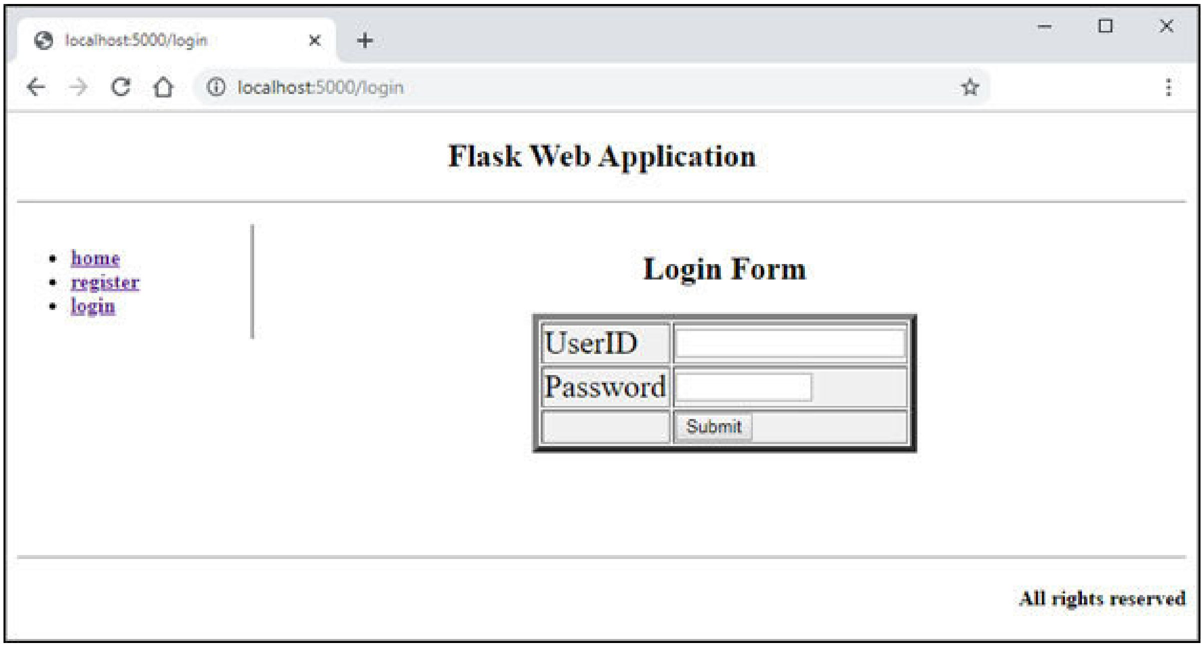


Flask Web Application

- home
- register
- login

Registration Form

Name	
UserID	
Address	
Course	
Password	
	Submit



Flask Web Application

- home
- register
- login

Login Form

UserID	
Password	
	Submit

All rights reserved

Hello World!

type="text" id="name">

type="submit" value="submit">

Browser software parses HTML script and constructs DOM tree before rendering the page. The DOM tree of the above HTML script will be as follows:

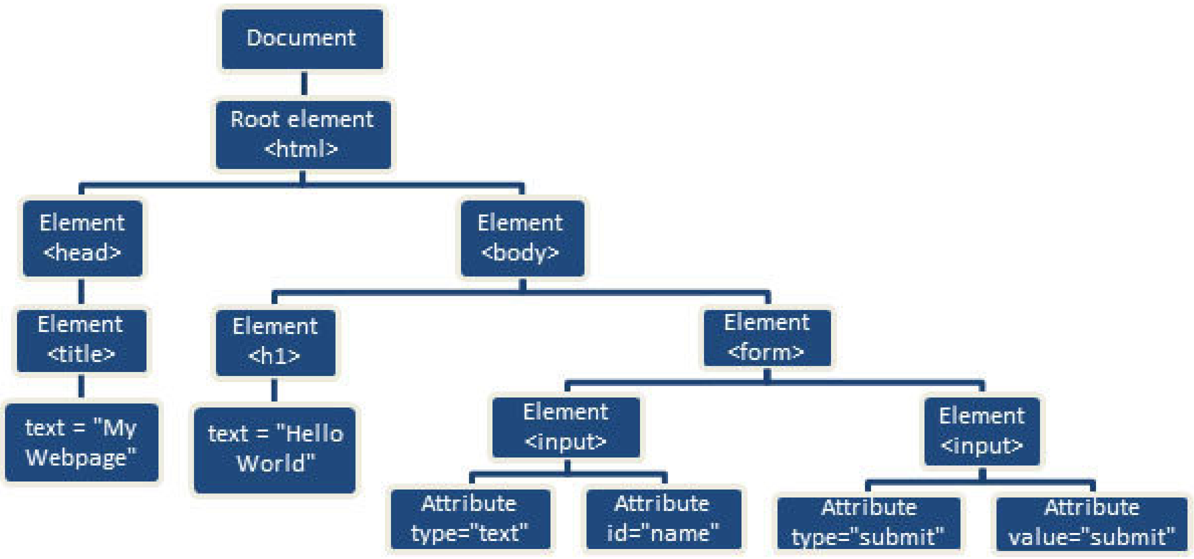


Figure 6.1: *DOM tree of HTML page*

JavaScript and DOM

JavaScript is a full-fledged programming language with all ingredients like variables, loops, arrays, functions, etc. Although its use is prevalent in mobile and desktop applications, and it is being increasing used for server-side processing also (through its implementations like Node.JS), it is vastly popular primarily because it enables the construction of interactive web pages. Its ability to tap into the DOM tree of a web page is an important (and hence essential) part of web application development.

JavaScript gets hold of the DOM tree with the help of the Document object. Child elements are obtained by employing the getElementById() or getElementsByTagName() methods. The innerHTML property of elements such as and

- returns the text contained inside element tags. Value attribute of input elements (such as text field) is also accessible from within JavaScript.

The HTML script given below has two paragraph tags with IDs flask and The tag with demo ID initially doesn't have any text. However, as JavaScript code executes, it is populated with text inside the

tag with flask ID:

#example.html

```
html>

id="flask">Hello Flask

Inner HTML set by JS code

id="demo">
```

Save above code as example.html and open it in a browser. It shows the following result:

```
Hello Flask
Inner HTML set by JS code
Hello Flask
```

DOM elements can recognize various events occurring as a result of a user's action (such as click or double-click) or because of change in their state (like change in value of an input field). JavaScript can register event handlers with these events. The most common event handlers are:

Associated JavaScript code is executed when a certain element is clicked.

When the page is loaded, this handler executes the corresponding code.

This handler is invoked when an input field has changed its value.

Following code demonstrates the onclick handler. Here, the HTML renders Hello World text in a

tag. However, the onclick handler is registered with it so that when it is clicked, its innerHTML property changes to Hello

```
html>


onclick="this.innerHTML='Hello Flask'">Hello World!
```

It is also possible to assign a user defined function to the event handler. In the following code, myfunction() is a user defined function. It has one parameter – the ID of element. We assign it to the onclick handler that is registered with the

tag, as above. The function is defined to change the innerHTML property:

```
html>

onclick="myfunction(this)">Hello World!
```

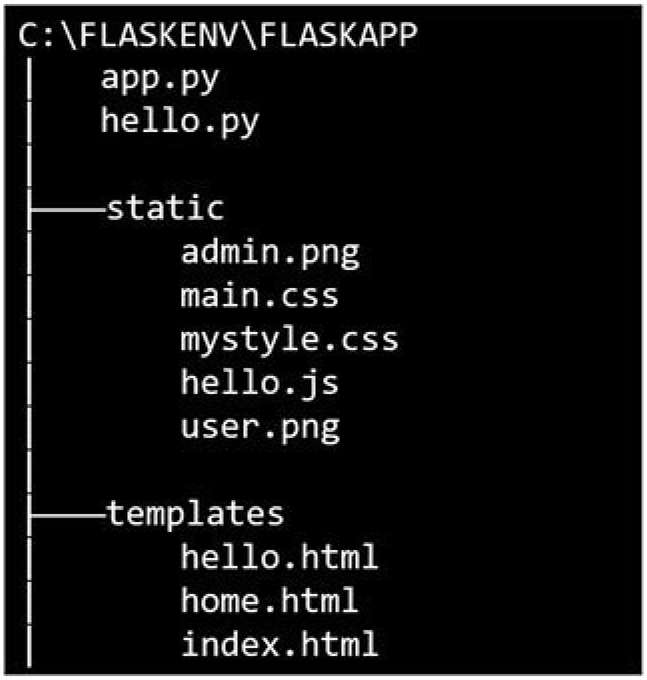
In the preceding examples, the JavaScript code is in line with HTML, i.e., it appears as a piece of code inside the web page. This can be cumbersome at times, especially if there are many JavaScript segments. Instead, we can write all JavaScript code in another file with the .js extension and include it in the HTML code. This is particularly of help when a JavaScript function is to be used in many web pages, in the sense that you don't have to rewrite the same code again and again.

Let's use this feature for the above example. We first save the myfunction() code in a separate file – say

#myscript.js

```
function myfunction(id) {
id.innerHTML = "Hello Flask";
}
```

In order to include this JavaScript file in HTML, we need to use the tag inside the



```
C:\FLASKENV\FLASKAPP
│    app.py
│    hello.py
│
├───static
│        admin.png
│        main.css
│        mystyle.css
│        hello.js
│        user.png
│
├───templates
│        hello.html
│        home.html
│        index.html
```

onload="myFunction()">

id="time" style="text-align:right; width="100%">

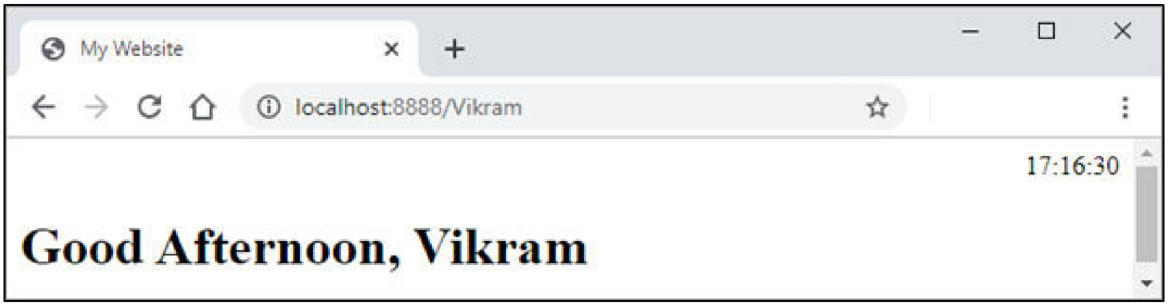
id="ttl">{{title}}

Note that there are two
tags with time and ttl IDs. JavaScript function defined in the following file puts current time in
with time ID and prepends the greeting message according to the time of day to user's name:

#hello.js

```
function myFunction() {
var today = new Date();
var h = today.getHours();
var m = today.getMinutes();

   var s = today.getSeconds();
   var msg="";
   if (h<12)
   {
   msg="Good Morning, ";
   }
   if (h>=12 && h<18)
   {
   msg="Good Afternoon, ";
   }
   if (h>=18)
   {
   msg="Good Evening, ";
   }
   var x=document.getElementById('txt').innerHTML;
document.getElementById('txt').innerHTML = msg+x;
document.getElementById('time').innerHTML = h + ":" + m +
":" + s;
}
```

Ensure that the HTML file is saved in the templates folder and JavaScript file is in the static folder. Start the Flask sever and enter **http://localhost:5000/Vikram** as the browser's URL:



17:16:30

Good Afternoon, Vikram

Figure 6.3: *JavaScript in Flask*

Try and run the application at different times of the day to obtain varying greeting messages (or better still, change your computer's time to confirm!).

CSS and DOM

In order to compose good-looking web pages, the contents of various HTML elements should be stylized by choosing suitable color for the text and background, font's type and size, etc. Early versions of HTML tag definitions used to have inline assignment of these attributes:

color="blue">Hello World!

The problem with this approach is that such tags controlling the presentation of content have to be explicitly described. Moreover, there is an unnecessary repetition of definition of the same attributes. For example, a document may be using the tag at many places. So, the styling tags for the

tag will have to be repeated as many times.

With the introduction of CSS, these presentational tags have been deprecated in later revisions of HTML. CSS stands for Cascaded Style Sheet. It is also a language that describes how a certain HTML element is formatted.

The properties controlling the appearance of HTML elements can be defined in different ways.

Inline definition

HTML introduced style attribute to define the styling properties of an element. The properties available for different HTML element may be different, but some of the frequently used properties are color, background-color, font-size, text-align, border, etc.

The general syntax of style attribute is as follows:

```
style="property1:value1; property:value2;">some text
```

For example, the

tag can be configured with style attribute like this:

style="color:blue;">Hello World!

Style definitions in the

. .

Here, styles for the

and

tags are defined. As a result, all paragraphs will show the text with justified alignment, and all top order headings will be displayed in blue.

External CSS file

Another method is to include a .css file in the HTML script. The file contains definitions of different styles. It is included by the tag and its rel as well as href attribute:

rel="stylesheet" href="styles.css">

This tag should normally appear in the

- href="#home">Home
- href="#news">News
- href="#contact">Contact
- href="#about">About

Open the above file with a browser. The horizontal menu is rendered as follows:

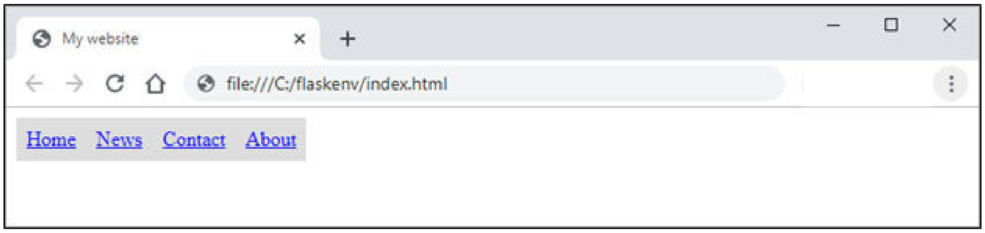


Figure 6.4: *CSS in HTML*

As another useful example, we shall configure the appearance of various tags used in the construction of HTML table, i.e., the tag itself, along with the

On the browser, it will translate to:

The entire jinja2 template code inside admission.html is as follows:

#admission.html

action="http://localhost:5000/admission" method=post>

and **tags. Text in the element is intended to be bold faced and with a grey background**

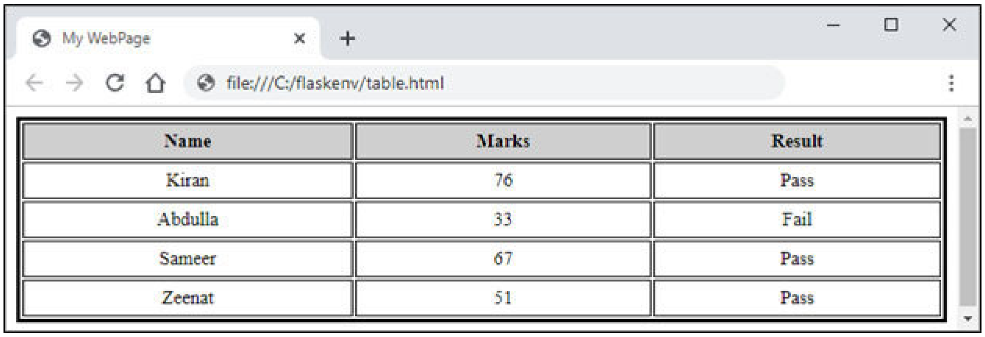
```
#table.css
table {
border: 3px solid #000000;
width: 100%;
text-align: left;
}
td, th {
border: 1px solid #000000;
padding: 5px 4px;
}
tbody td {
font-size: 15px;
text-align:center;
}

thead th {
font-size: 15px;
font-weight: bold;
color: #000000;background: #CFCFCF;
text-align: center;
}
```

Our table is a mark list of four students. The following HTML code displays the table as per the styles defined in the preceding file:

#table.html

Name	Marks	Result
Kiran	76	Pass
Abdulla	33	Fail
Sameer	67	Pass
Zeenat	51	Pass

A beautifully designed table will be displayed as we open table.html in a browser:



Name	Marks	Result
Kiran	76	Pass
Abdulla	33	Fail
Sameer	67	Pass
Zeenat	51	Pass

Figure 6.5: HTML Table with CSS

We shall be using the code from both style sheet files and in the next section. Both files are combined in styles.css for the next example.

CSS and Flask

As in case of JavaScript, the external CSS file is linked with template web page with static endpoint and the url_for() function:

```
rel="stylesheet" href="{{url_for('static', filename='styles.css')}}"
```

Instead of hardcoded HTML table data in the preceding example, data received from a Flask application is to be dynamically rendered in the table. The following Flask script has a view function that sends a list of dictionary items to a jinja2 web template:

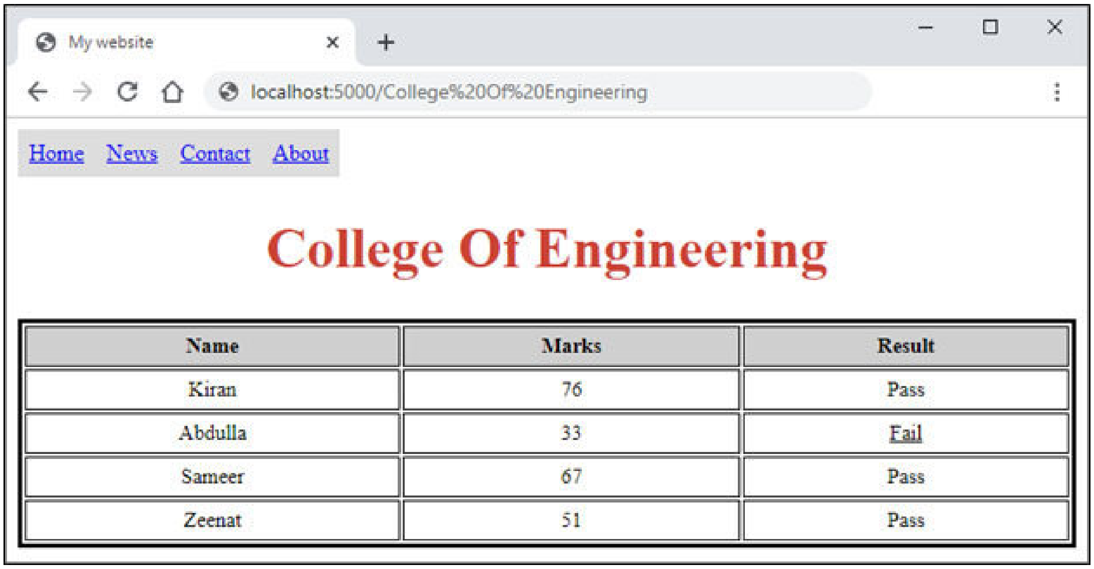
```
#app.py
@app.route('/')
def index(name):
name = "{}".format(name)

marks=[{'Name':'Kiran','Marks':76},{'Name':'Abdulla', 'Marks':33},
{'Name':'Sameer', 'Marks':67},{'Name':'Zeenat', 'Marks':51}]
return render_template('index.html', college=name, marks=marks)
```

Each item in the list is a dict object with name and marks keys; note that the result key is not present. The template code given below uses jinja2 syntax of for loop to iterate over the marks collection and puts name and marks in the first two columns of table. Third column is conditionally filled as Pass or Fail with the help of the {% if %} statement:

#index.html

```
{% for row in marks %}
{% endfor %}
Name            Marks                    Result
                                 {% if row['Marks']>=50 %}
                                            Pass
{{row['Name']}} {{row['Marks']}}          {% else %}
                              style="text-decoration:underline;">Fail
                                        {% endif %}
```

While filling the Result column, inline style attribute is used to display Fail with an underline. With the application code, template, and CSS file in place, start the Flask server and enter http://localhost:5000/College of Engineering as the browser's URL:



College Of Engineering

Name	Marks	Result
Kiran	76	Pass
Abdulla	33	Fail
Sameer	67	Pass
Zeenat	51	Pass

Figure 6.6: CSS in Flask Template

Images in Flask App

HTML tag library contains tag for creating a place holder for an image. A string that represents the name (along with path) of an image file is assigned as the value of src attribute of the tag:

src="image.png">

This tag is commonly placed between or anchor tag to establish a link with another document specified by the href attribute:

href="mypage.html">src="image.png">

An image is also used as a button. HTML defines an input element with the image attribute:

type="image" src="image.png" />

Using an image in a jinja2 template is more or less similar, except that the src attribute uses url_for() function to obtain path (as in the case of CSS or JavaScipt file) because image files are usually stored in the static folder of Flask application:

```
src="{{url_for('static',filename='image.png')}}">
```

Following Flask app makes use of static resources (a JavaScript code and a couple of images). Image files are used as image buttons, and their path is constructed by calling the url_for() function.

Here's the Flask application code:

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

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

The View function in the preceding code renders a template whose script is as follows:

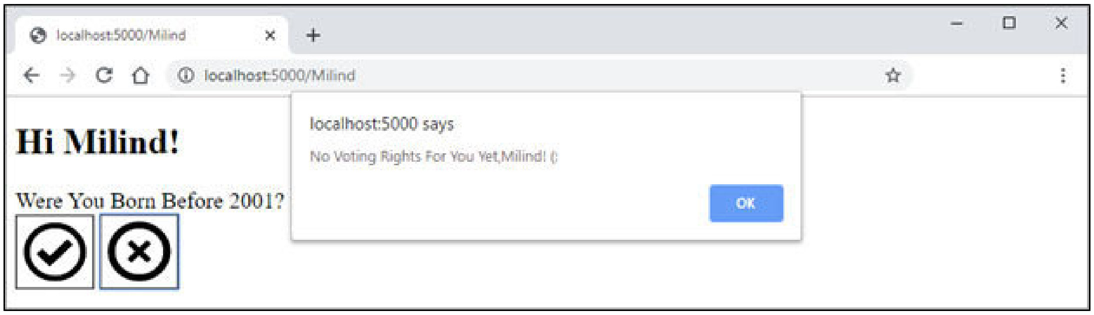
#btn.html

html>

Hi {{name}}!

```
style="font-size:20px;">Were You Born Before 2001?
type="hidden" id="nmId" name="name"
value={{name}}>
type="image" name="button1"
onClick="myfunction(this.id)" id="button1" src="
{{url_for('static',filename='checkmark.png')}}" />
type="image" name="button2"
onClick="myfunction(this.id)" id="button2" src="
{{url_for('static',filename='cancel.png')}}" />
```

This code renders two images (checkmark and cancel) as buttons. Their onClick event is transmitted to This function has been defined in which is included in the

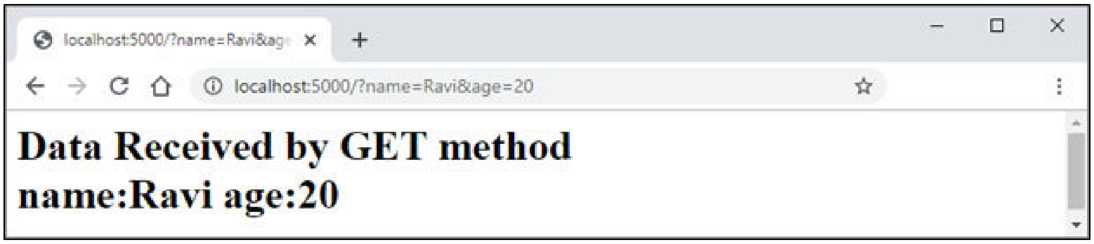


Hi Milind!

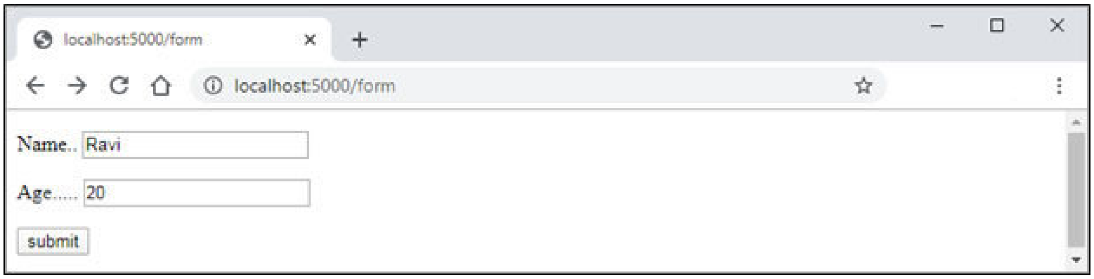
Were You Born Before 2001?

localhost:5000 says

No Voting Rights For You Yet,Milind! (:

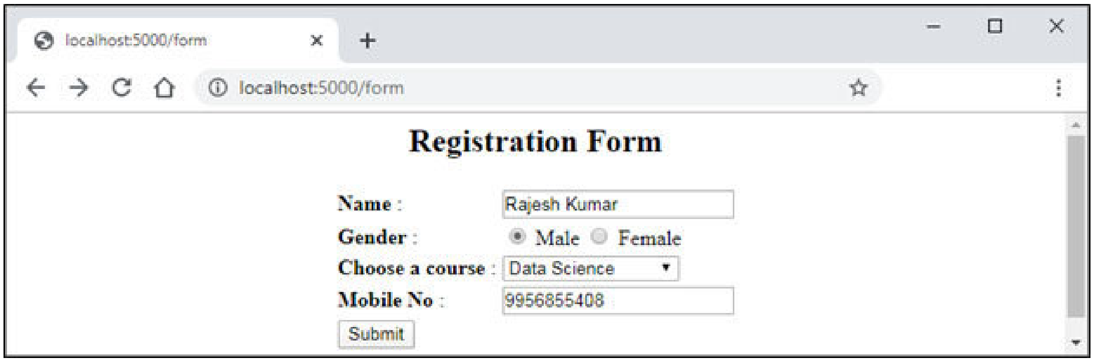

localhost:5000/?name=Ravi&age
localhost:5000/?name=Ravi&age=20

Data Received by GET method
name:Ravi age:20



Name.. Ravi

Age..... 20



Registration Form

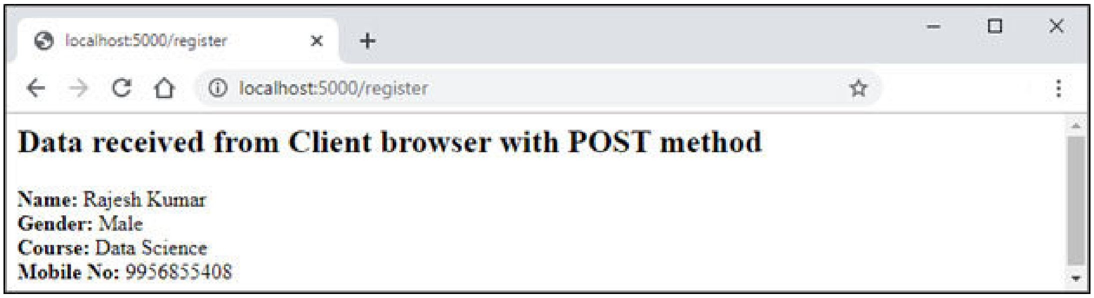
Name : Rajesh Kumar

Gender : ● Male ○ Female

Choose a course : Data Science ▾

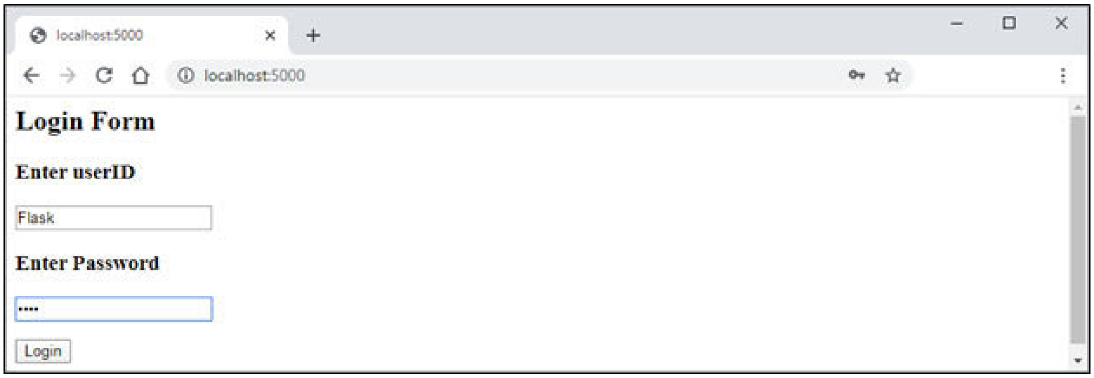
Mobile No : 9956855408

Submit



Data received from Client browser with POST method

Name: Rajesh Kumar
Gender: Male
Course: Data Science
Mobile No: 9956855408



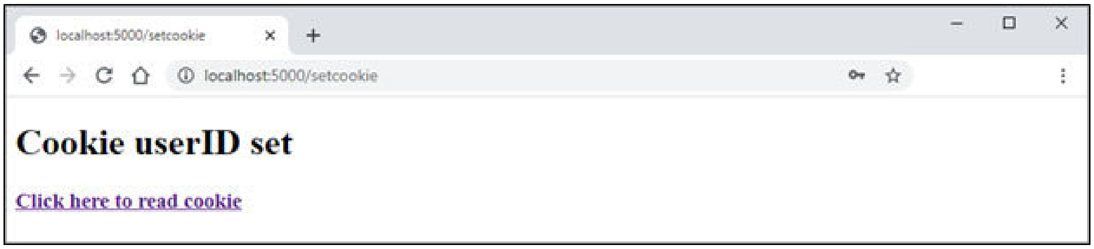
Login Form

Enter userID

Flask

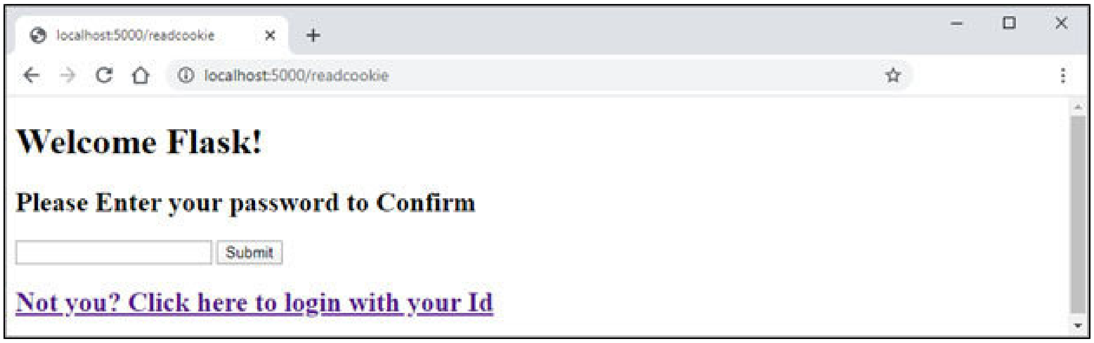
Enter Password

••••



Cookie userID set

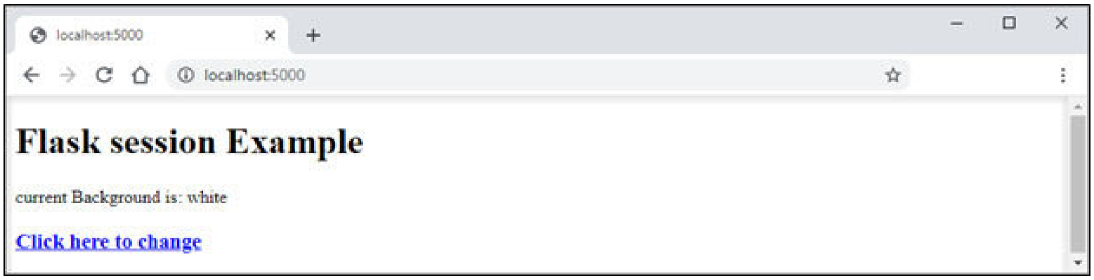
Click here to read cookie



Welcome Flask!

Please Enter your password to Confirm

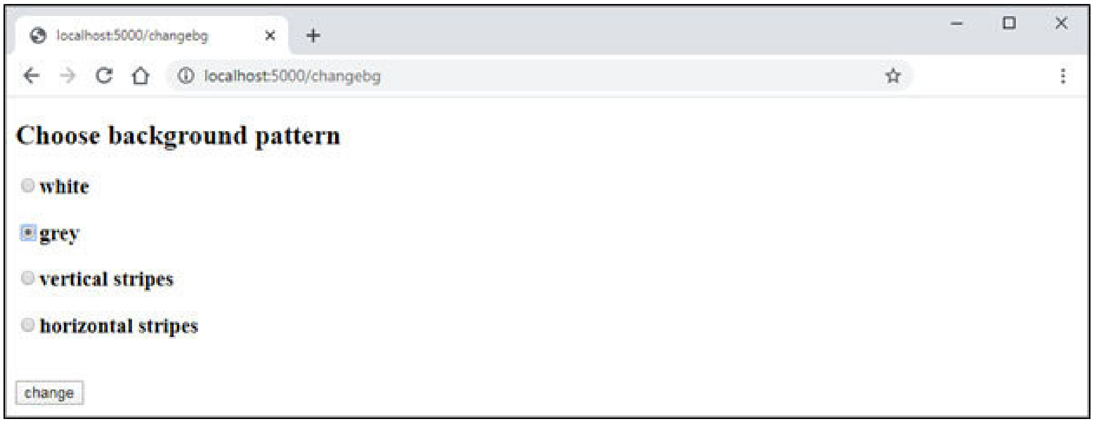
Not you? Click here to login with your Id



Flask session Example

current Background is: white

Click here to change



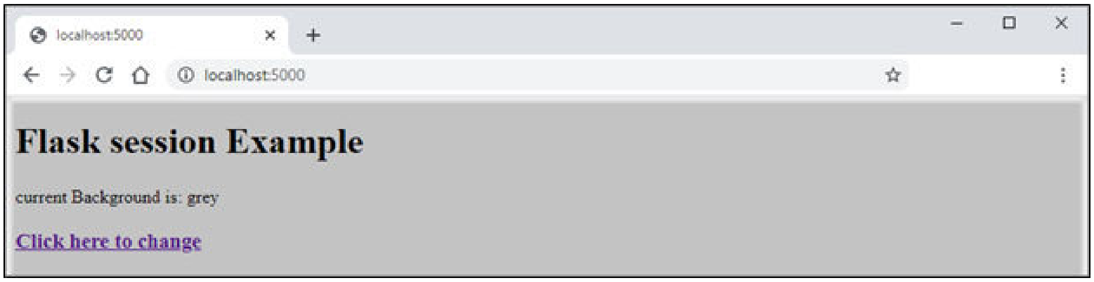
Choose background pattern

○ white

● grey

○ vertical stripes

○ horizontal stripes



Flask session Example

current Background is: grey

Click here to change

Flask Message Flashing Example

{% with messages = get_flashed_messages(with_categories=true) %}
{% if messages %}
{% for category, message in messages %}

{{category}} : {{message}}

{% endfor %}
{% endif %}

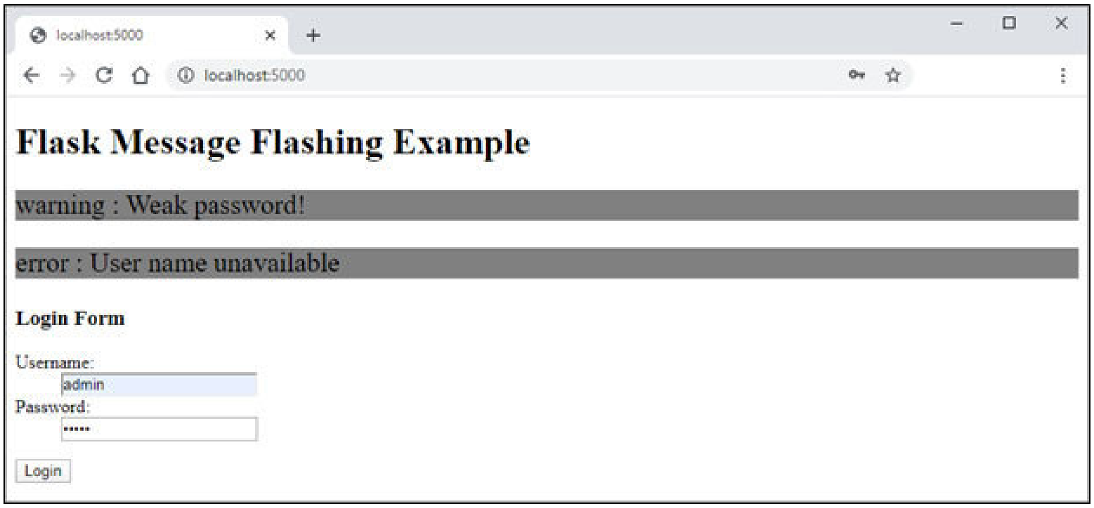
{% endwith %}

Login Form

action="" method="POST">

Username:
type="text" name="username">
Password:
type="password" name="password">
type="submit" value="Login">

When the home page is first visited, it displays a simple form with two input fields. If it contains data resulting in error flash messages, they appear on top and the form is displayed again:



Flask Message Flashing Example

warning : Weak password!

error : User name unavailable

Login Form

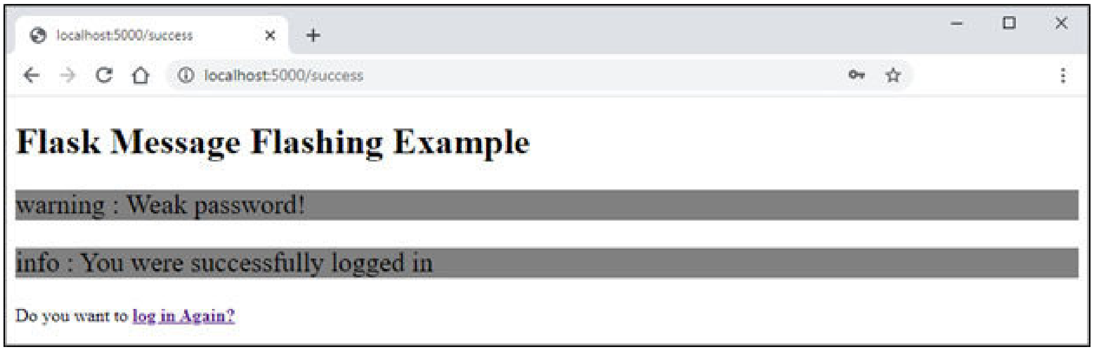
Username: admin

Password: •••••

Login

Figure 7.11: Message flashing

If the form contains no error, our application is redirected to the success route that renders flash message of a successful login:



Flask Message Flashing Example

warning : Weak password!

info : You were successfully logged in

Do you want to log in Again?

Figure 7.12: Message flashing categories

Note that a warning message is also appearing as the password may be less than 8 characters long. Lastly, the code for the /success route is straightforward, as follows:

```
@app.route('/success')
def success():
return render_template('success.html')
```

The WTForms library combines well with flashing technique to provide effective form validation. We shall discuss this in one of the upcoming chapters.

Uploading a File

Last topic in this chapter deals with a frequently required feature in a web application – enabling the user to upload a file on the server. Once again, Flask API has a simple solution for this requirement.

Before we find out how Flask server handles this task, we shall focus on creating a client form that lets the user choose a file to be uploaded from the local file system. In an HTML form, if type attribute of the input element is set to file, it renders a button with choose file caption. However, the form itself should have its enctype attribute set to The following web page renders a form with file button in addition to the regular Submit button:

```
html>
action="\upload", method="POST", enctype="multipart/form-data">
type="file" name="file">
type="submit" values="upload">
```

As usual, save the above file in the templates folder. The following Flask route renders this template:

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

```
@app.route('/')
def form():

return render_template('form.html')
```

As the application starts, a user can navigate local file system to choose a file to be uploaded by clicking on the Choose File button and then clicking on Upload button:

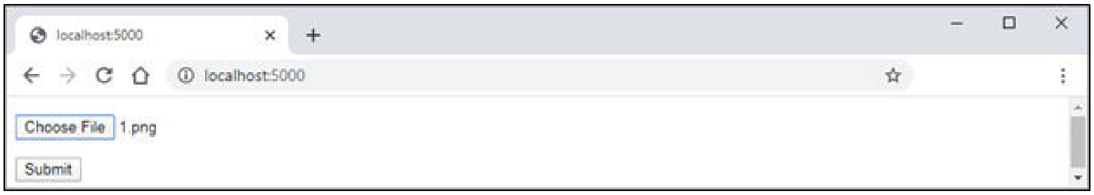


Figure 7.13: HTML form with File button

Now, we have to develop view function for \upload endpoint – the action attribute of the above form. Note that an HTML form used to upload a file must have its method attribute set to Data in the chosen file is available in files attribute of Flask's request object. It is essentially a Python dictionary containing details of the file to be uploaded, such as filename, content-type, and length. It also has a save() method that saves it in the server's file system:

```
@app.route('/upload', methods = ['GET', 'POST'])
def upload():
if request.method == 'POST':
f = request.files['file']
f.save('/uploads'+f.filename)
```

```
return 'file uploaded successfully'
```

Before running the application, ensure that the uploads folder is available inside application's folder. After finishing the save operation, the browser renders a success message.

Conclusion

This chapter equips the learner with techniques of adding an important feature in a Flask application – that of submitting user data using an HTML form. Message flashing is also an effective interaction tool that was discussed in this chapter. With this, it is now possible to develop a fairly useful application.

So far, we don't know how to provide backend database support to Flask application. In the next chapter, we will discuss how Flask interacts with SQL-based relational databases and NOSQL databases.

CHAPTER 8

Using Databases

Introduction

Every web application needs a database support for the storage and retrieval of data. Traditionally, relational databases are used as a backend of an application. Modern web applications need to handle large data with dynamic schema. This requirement is met by NoSQL databases.

In this chapter, we shall discuss the use of both SQL-based and NoSQL databases in a Flask application. We also intend to learn to employ ORM extensions for both types of databases.

Structure

DB-API

CREATE TABLE

INSERT DATA

READ DATA

SQLAlchemy

Flask extensions

Flask-SQLAlchemy

Relationship

NoSQL databases

Flask-PyMongo

Flask-MongoEngine

Objectives

After studying this chapter, you should be able to:

Use SQL-based relational database such as SQLite in Flask application.

Use Flask-SQLAlchemy extension.

Connect Flask application with MongoDB database.

Perform CRUD operations with Flask-PyMongo and Flask-MongoEngine extensions.

Almost every computer application – whether it is web-based or a standalone desktop application – should be able to store the data in some persistent medium, such as disk, so that it can be retrieved when required. Python's File API is a basic tool for storage and retrieval of data in raw form. Certain modules in Python's standard library do offer additional functionality to serialize Python objects in disk files. However, such files are largely unstructured, which can lead to data redundancy, may compromise integrity of data, thus they are not suitable for real-time transaction processing.

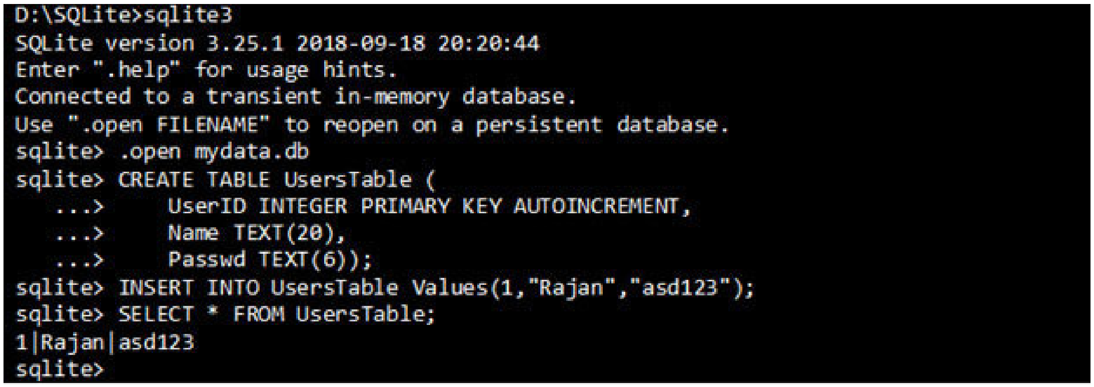
A database is a more organized method of data storage. Good database design avoids the above-mentioned drawbacks. Although database itself is independent software, it can be interfaced with applications written in various languages, including Python. Relational databases such as Oracle, MySQL, SQL server are predominantly used in large number of applications for data storage. However, in today's world of real-time web applications and big data, a new category of databases called NoSQL databases are increasingly being used.

In this chapter, we shall first learn how to use SQLite, which is a lightweight and serverless relational database, with Flask. Later, we shall discuss the connectivity of Flask with a NoSQL database named MongoDB.

DB-API

The fundamental concept of a relational database is to store data in different entity tables and establish a relationship among them on the basis of common attributes. Various Relational Database Management Systems (RDBMS), such as Oracle, MySQL, SQLite, etc. use a standardized Structured Query Language (SQL) for creating and manipulating a table. A SQL statement can be issued from either a command line interface or GUI environment of the respective RDBMS.

Following diagram shows how to create a database and table in SQLite console:



```
D:\SQLite>sqlite3
SQLite version 3.25.1 2018-09-18 20:20:44
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open mydata.db
sqlite> CREATE TABLE UsersTable (
   ...>     UserID INTEGER PRIMARY KEY AUTOINCREMENT,
   ...>     Name TEXT(20),
   ...>     Passwd TEXT(6));
sqlite> INSERT INTO UsersTable Values(1,"Rajan","asd123");
sqlite> SELECT * FROM UsersTable;
1|Rajan|asd123
sqlite>
```

Figure 8.1: SQLite console

In SQLiteStudio, database and table structure can be easily created using GUI, as follows:

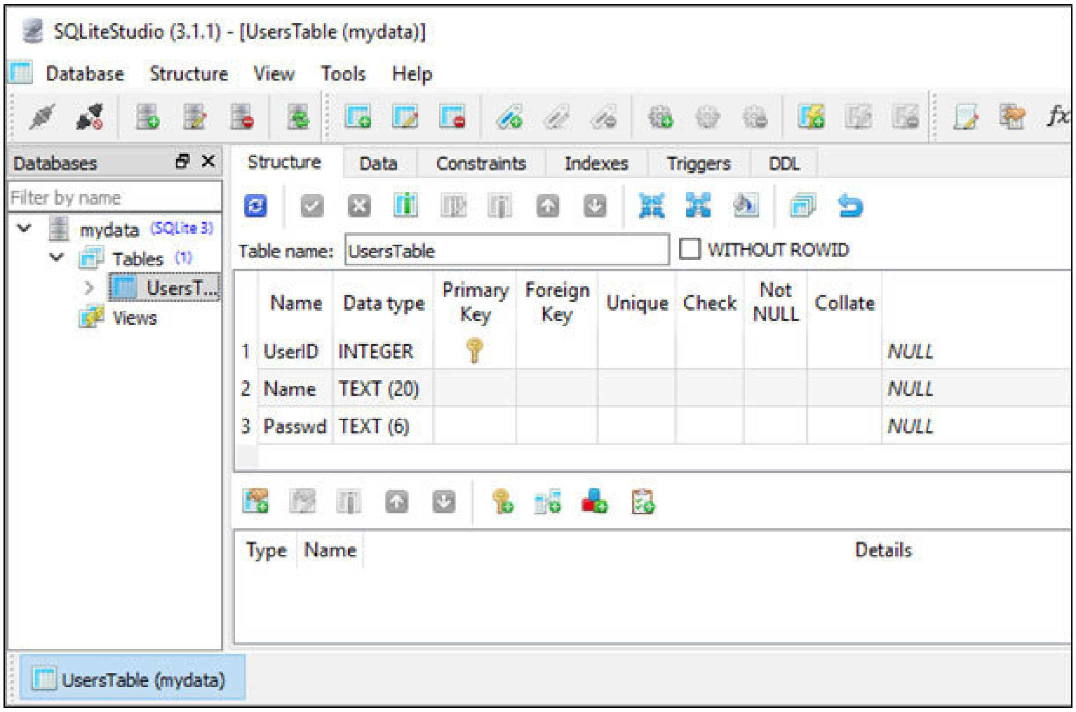


Table name: UsersTable □ WITHOUT ROWID

	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate	
1	UserID	INTEGER	🔑						NULL
2	Name	TEXT (20)							NULL
3	Passwd	TEXT (6)							NULL

Type	Name	Details

Figure 8.2: SQLiteStudio GUI

As mentioned above, all relational databases are built on top of SQL. There are certain differences in how SQL standard is implemented by each product. Obviously, their programming interface for interaction with programming language (Python in our case) is bound to reflect the difference in various implementations.

In order provide a uniform skeleton to be adapted by Python interface module for any relational database, a set of standards called DB-API has been recommended. Python's standard library already consists of the sqlite3 module, which is a reference implementation of DB-API standard for SQLite database. Python modules for connectivity with other SQL-based databases will have to be installed explicitly. For instance, to use MySQL database, we need to install the PyMySql module, as follows:

```
pip3 install pymysql
```

We shall use the sqlite3 module and learn how to perform CRUD operations on a SQLite database in a Flask application.

As per DB-API specifications, the first step is to obtain the Connection object that represents the database. The sqlite3 module (as indeed any DB-API compliant module to be used with the corresponding type of database) uses the connect() function to do this:

```
import sqlite3
conn=sqlite3.connect("mydata.db")
```

Note that a new database will be created if the one named as the argument to the connect() function doesn't exist.

The next step is to obtain a cursor from the connection object. A cursor is really a handle to the database in use so that transactions such as inserting and deleting data can be performed:

```
cur=conn.cursor()
```

Now, we can execute different SQL queries with the help of this cursor. We need to pass a query string to the execute() method. If the query string represents a SELECT statement, the execute() method returns a resultset containing the number of rows selected:

```
res=cur.execute("query string")
```

In the previous chapter, we have designed an HTML form and fetched the submitted data in a Flask route. Now, we intend to store this data in the Students table in the mydata.db database. We shall also add two more form elements for username (which must be unique) and

CREATE TABLE

As our Flask application starts, we establish a database connection and create the table as per the steps described above:

```
#app.py
app = Flask(__name__)
con=sqlite3.connect("mydata.db")
cur=con.cursor()
createqry='''
CREATE TABLE IF NOT EXISTS Students (
Name STRING (20) NOT NULL,
Course STRING (20),
Gender STRING (20),
Mobile INTEGER (10),
Username STRING (6) PRIMARY KEY NOT NULL,
Password TEXT (8) NOT NULL
);
'''

cur.execute(createqry)
```

INSERT DATA

Flask application route \addrec receives HTML form's data Its mapped view function invokes the INSERT query to add a new row in the Students table:

```
@app.route('/addrec',methods=['POST', 'GET'])
def addrec():
if request.method=='POST':
con=sqlite3.connect("mydata.db")
cur=con.cursor()
msg=''

try:
nm=request.form['name']
gndr=request.form['gender']
course=request.form['course']
mob=request.form['mobile']
usr=request.form['user']
pw=request.form['pwd']
ins="INSERT INTO Students VALUES (?,?,?,?,?,?)"
cur.execute(ins,(nm, gndr, course, mob, usr, pw))
con.commit()
```

```
msg= "Record successfully added"
except Exception as e:
con.rollback()
msg= "error in insert operation"
finally:
return render_template("result.html",msg=msg)
```

The data received from the HTML post form is parsed to Python objects and they in turn, are passed to the parameterized INSERT query. Note the use of exception handling to ensure that the transaction is committed only when the query is successfully executed. In the event of any SQL-related exception, the action is rolled back. The view returns a result.html template to show success or failure of the INSERT operation.

Data thus added in the Students table can, of course, be viewed by opening it in SQLiteStudio or in SQLite console. However, the password field will be revealed, which obviously is not desired. Hence, it is subjected to some sort of encryption before sending it for insertion.

Following code makes use of the base64 module to define the encpwd() and decpwd() functions:

```
#encryption.py
import base64
def encpwd(pwd):
e=base64.b64encode(pwd.encode('utf-8'))
secpwd=e.decode('utf-8')
return secpwd
def decpwd(pwd):
p1=base64.b64decode(pwd.encode('utf-8'))
origpwd=p1.decode('utf-8')
return origpwd
```

Let's apply the encpwd() function on the password form field in our Flask application:

```
from encryption import encpwd
```

Change the statement with call to as follows:

```
cur.execute(ins,(nm,gndr,course,mob,usr,encpwd(pw))
```

This will ensure that the table displays the password field in an encrypted form.

READ DATA

To retrieve the rows from a table, we need to issue a string to the execute() method such that it is a valid SELECT query:

```
cur.execute("SELECT * FROM Students;")
```

The fetchall() method of cursor objects returns a resultset that is a MultiDict object holding the rows affected by the SELECT statement. It is further passed to a HTML template that employs Jinja2 loop to render the rows in an HTML table. Here's the /list route whose mapped view function performs this process:

```
@app.route('/list')
def list():
con=sqlite3.connect("mydata.db")
con.row_factory = sqlite3.Row
cur = con.cursor()
cur.execute("select * from Students;")
```

```
students=cur.fetchall()
return
render_template("studentlist.html",students=students)
```

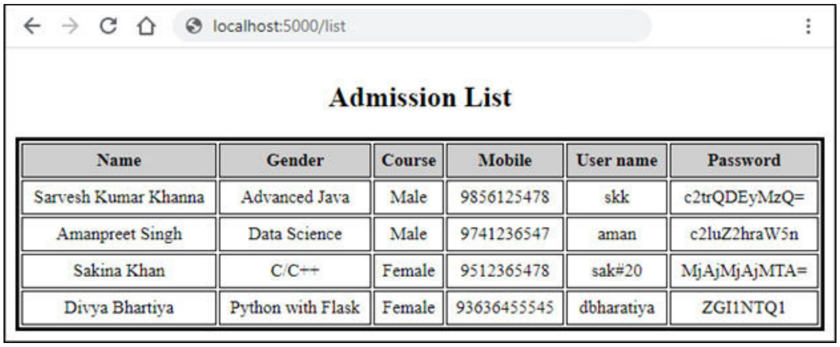
The studentlist.html template is put in the templates folder. It applies the stylesheet from styles.css available in the static folder:

```
#studentlist.html
{% for row in students %}
{% endfor %}
align="top"
    >
```

Admission List

Name	Gender	Course	Mobile	User name	Passwo
{{row['Name']}}	{{row['Gender']}}	{{row['Course']}}	{{row['Mobile']}}	{{row['Username']}}	{{row['Passw

Start the Flask server and visit All records available in the Students table will be displayed a formatted HTML table in the browser:



localhost:5000/list

Admission List

Name	Gender	Course	Mobile	User name	Password
Sarvesh Kumar Khanna	Advanced Java	Male	9856125478	skk	c2trQDEyMzQ=
Amanpreet Singh	Data Science	Male	9741236547	aman	c2luZ2hraW5n
Sakina Khan	C/C++	Female	9512365478	sak#20	MjAjMjAjMTA=
Divya Bhartiya	Python with Flask	Female	93636455545	dbharatiya	ZGI1NTQ1

Figure 8.3: SELECT query output in template

Performing the UPDATE and DELETE operations is straightforward. The functions and routes for activities can be found in the Flask application for this chapter in the code bundle.

Object Relation Model

The DB-API standard makes relational database handling very easy and virtually independent of the product. However, it is not particularly suitable to store Python's user defined objects in a table. Apart from number and string, other data types in Python have no equivalent data types in SQL. Moreover, when it comes to store objects of a user defined class, the object attributes need to be manually unpacked to equivalent SQL types that are scalar in nature.

Following Python code snippet has a Student class and s1 as its object:

```
class student:
def __init__(self, name, age, marks):
self.name=name
self.age=age
self.marks=marks
```

```
s1=Student("Sam",21,85)
```

We would like to store objects of the Student class (such as in a SQLite table. The structure of the Students table will be as follows:

```
CREATE TABLE Students (
Name STRING (20) NOT NULL,
Age INTEGER (3),
Marks INTEGER (3)
);
```

However, we need to deconstruct the object attributes in SQL compatible scalar variables so that they can be used as parameters for executing the INSERT query:

```
cur.execute("INSERT INTO Students VALUES (?,?,?)",
(s1.Name, s1.Age, s1.Marks))
```

On the other hand, row in a result set of the SELECT query is a dictionary, which needs to be converted into an object of the Student class:

```
cur.execute("select * from Students WHERE name=?",
("Sam",))
row=cur.fetchone()
s1=Student(row['Name'], row['Age'], row['Marks'])
```

This explicit conversion between Python object and SQL compatible types is tedious to say the least. Object Relation Mapping API provides an easier alternative.

SQLAlchemy

An ORM library provides an interface between the two incompatible environments: SQL-based database on one side and a programming environment of an object-oriented language such as Python. In an ORM system, a class maps to a table in the underlying database. Manipulation of attributes of an object through getters and setters is converted in the respective SQL query by the ORM so that you can focus on programming the logics of the system instead of constructing raw SQL queries.

SQLAlchemy is a popular ORM library for Python. A Python class (as the Student class defined earlier) is mapped to a SQL table (as a Student table with its structure corresponding to the instance attributes of the Student class). Each object represents a row in the table. Methods of this class encapsulate raw SQL queries such as and SELECT such that any change in the state of its object are transparently synchronized with the corresponding row.

SQLAlchemy uses dialect system to communicate with a wide variety of relational databases for which DB_API compliant modules are available. SQlAlchemy helps establish connection with a database based on URI that represents the type of database and its module. The following table shows different dialects, their respective DB-API driver modules, and the URI:

URI:
URI: URI:
URI:
URI:
URI:
URI:

Table 8.1: SQLAlchemy dialects

Any Python application can use SQLAlchemy, so it is perfectly possible to employ it to add database support in a Flask application. Flask-SQLAlchemy makes it even easier.

Flask-SQLAlchemy is an extension for Flask that simplifies SQLAlchemy's ORM. Before we discuss its functionality, let's first understand what a Flask extension is.

Flask extensions

Simply put, an extension is a Python package that customizes a certain functionality for Flask. As mentioned earlier (in *Chapter 2:* Flask is classified as a micro framework. Core API of Flask only provides routing, template engine, and support for HTTP objects (request, response, cookies etc.). So, it can be very tedious if a Flask application requires certain additional functionality (such as writing unit tests, database migration, handling background jobs, server side form validation, etc.). Thankfully, an extension library written for a specific purpose makes it easy to incorporate the same in a Flask application.

Hundreds of Flask extensions (virtually for all web application development needs) have been developed and are in public domain. In this chapter, we shall learn how to employ two extensions: Flask-SQLAlchemy and Flask-MongoEngine (which is an ORM for MongoDB database).

Generally, a Flask extension is named following the pattern although this is not mandatory. Similarly, the installation, configuration, and usage of each extension may vary, but in general, the configuration settings of extension is added to the configuration of Flask's application's instance:

```
from flask import Flask
from flask_example import Example

app=Flask(__name__)
app.config.update(
Example_Key1=value1,
Example_key2=value2,
..
)


obj=Example(app)
```

Methods of extension's main object, along with various view functions, are then used as a part of the Flask application workflow.

Flask-SQLAlchemy

We now turn our attention to this extension of Flask that simplifies and customizes SQLAlchemy ORM. Installation is fairly straightforward: just use the default package installer – as follows:

```
pip3 install flask-sqlalchemy
```

Latest stable version of Flask-SQLAlchemy is ver.2.4.1, although ver.3 is under development:

```
>>> import flask_sqlalchemy
>>> flask_sqlalchemy.__version__
'2.4.1'
```

The package contains two important classes: SQLAlchemy class and Model class. The first one facilitates the integration of SQLAlchemy with Flask application, while the Model class acts as a base class for declarative base model, as defined in SQLAlchemy.

As with most Flask extensions, an instance of the SQLAlchemy class is obtained by passing Flask Application object to its constructor. However, before passing, parameters specific to Flask-SQLAlchemy should be loaded in the configuration of Flask object:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI']='sqlite:///mydata.db'
db = SQLAlchemy(app)
```

The Above snippet defines the URI to be used for database connection. As mentioned earlier in this chapter, this URI includes the dialect to be used. In this case, we shall work with a SQLite database. Other optional configuration parameters are:

are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are:
are: are: are: are: are: are: are: are: are: are: are: are: are: are: are:
are: are: are: are: are: are: are: are: are: are: are: are:
are: are: are: are: are:

Table 8.2: Flask-SQLAlchemy configuration

Core SQLAlchemy uses declarative base class to define mapping between an entity class and a table in database. Its Flask extension uses the Model class, from which all user entity classes are inherited. By default, the name of the mapped table is the name of the class itself, unless specified by a class attribute. Other class attributes correspond to columns or fields in the mapped table.

Each field is created as an instance of the Column class. The field type and length arguments are passed to its constructor, and certain optional parameters such as primary_key and autoincrement may also be passed.

Let's design a Model class for the students table used earlier in this chapter while discussing DB-API:

```
#models.py
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI']='sqlite:///mydata.db'
db = SQLAlchemy(app)
```

```
class students(db.Model):
name=db.Column(db.String(20))
course=db.Column(db.String(20))
gender=db.Column(db.String(10))
mobile=db.Column(db.Integer)
username=db.Column(db.String(6), primary_key=True)
password=db.Column(db.String(8), nullable=False)

def __init__(self, name, course, gender, mobile, username, password):
self.name=name
self.course=course
self.gender=gender
self.mobile=mobile
self.username=username

self.password=password
```

All tables mapped with model classes will be created whenever the create_all() method of the SQLAlchemy object is called:

```
db.create_all()
```

Remember that if a database already has a table, it will not be created on above call. Usually, tables are created when the application is run. Once the models are created, it is possible to perform the insert, delete, and update operations. To insert or delete a record, we need to create a database session and call the add() and delete() methods, respectively.

To insert a record in a table, we should first set up an object of its model class. In the above example, we have a students model. Its object needs parameters, as defined in the __init__() method:

```
from models import students
s1=students("Amanpreet", "C/C++", "Male", 9741236457,
"aman", "aman123")
```

The object is then passed to the add() method of session to persistently reflect in the students table:

```
db.session.add(s1)
db.session.commit()
```

Similarly, db.session.delete() removes the row corresponding to the specified object from the table:

```
db.session.delete(s1)
db.session.commit()
```

Raw SELECT query, as in SQL, is implemented by the query attribute of the model class. The query object so obtained has the all() method to retrieve all rows in the mapped table:

```
list=students.query.all()
```

It returns a list of dictionary objects, each corresponding to a row in the table. Each item in dictionary is a pair of instance attribute and its value. The query object may be subjected to filter to restrict the number of rows to be fetched. For example, the following statement returns the first row that contains

```
obj=students.query.filter_by(course='C/C++').first()
```

These ORM operations can be performed through view functions registered with Flask routes. Complete example of database operations described with DB-API has been modified using Flask-SQLAlchemy instead. Its code is available in the code bundle, and the link for downloading the same is mentioned in this book's preface.

Relationship

One of the most important features of RDBMS is establishing a relation between tables on the basis of a common attribute. Table A is related with B by defining one of its attributes that appears as a primary key in the structure of table In table structure, this attribute carries a constraint called foreign

In a model class too, one of the class attributes is defined as a We already have a students class with username as primary key. Let's create another class which, incidentally, will be mapped to the books table in the underlying database. One of its attributes, is defined as foreign key with a reference to the username attribute of the students class. The idea is to show who has borrowed a certain book from the college library. The Books table shall only store username as the borrower, but we should be able to fetch the corresponding details from the students table:

```
#models.py
class books(db.Model):
bookID=db.Column(db.Integer, primary_key=True)
title=db.Column(db.String(100))
author=db.Column(db.String(50))
borrower=db.Column(db.String(6), db.ForeignKey('students.username'))

def __init__(self, id, title, author, borrower):
```

```
self.id=id
self.title=title
self.author=author
self.borrower=borrower
```

As a result, when the db.create_all() method is executed, the books table will be created effectively, representing following CREATE TABLE statement:

```
CREATE TABLE books (
bookID INTEGER NOT NULL,
title VARCHAR (100),
author VARCHAR (50),
borrower VARCHAR (6),
PRIMARY KEY (
bookID
),
FOREIGN KEY (
borrower
)
REFERENCES students (username)
);
```

We can clearly see the relationship between the books.borrower and students.username fields. The same is visually represented in SQLiteStudio GUI as shown in the following figure:

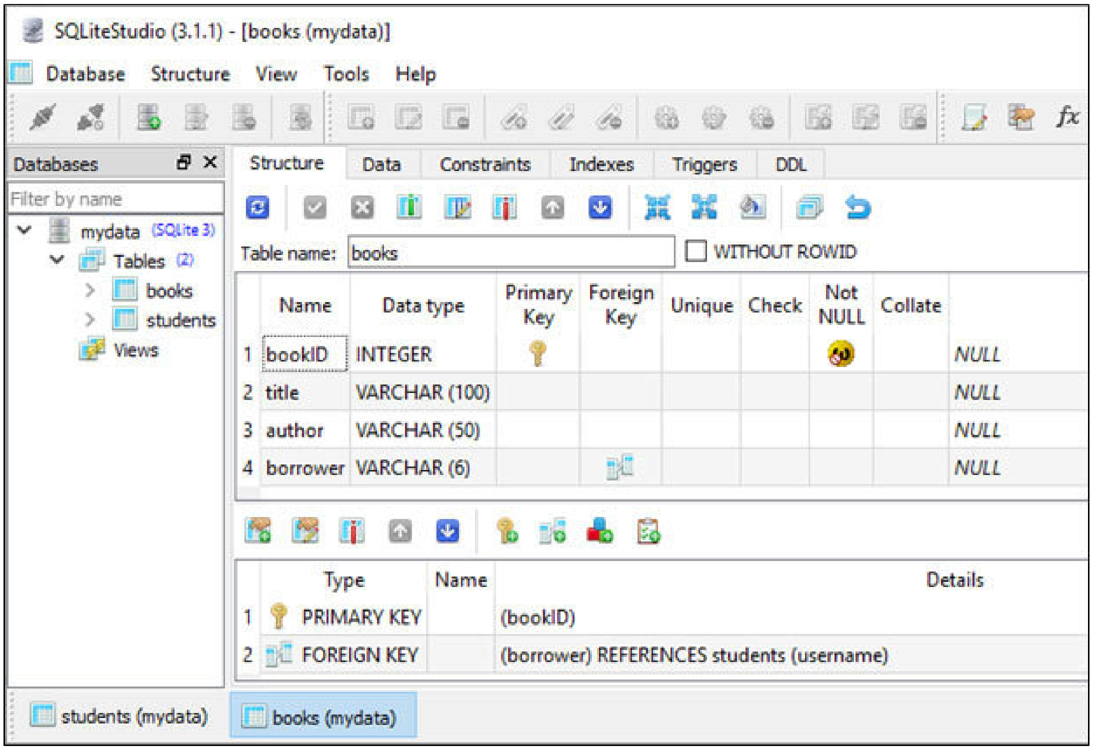

SQLiteStudio (3.1.1) - [books (mydata)]
Database Structure View Tools Help
Databases
Filter by name
mydata (SQLite 3)
Tables (2)
books
students
Views
Structure Data Constraints Indexes Triggers DDL
students (mydata)
books (mydata)

Table name: books ☐ WITHOUT ROWID

	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate	
1	bookID	INTEGER	✓				✓		NULL
2	title	VARCHAR (100)							NULL
3	author	VARCHAR (50)							NULL
4	borrower	VARCHAR (6)		✓					NULL

	Type	Name	Details
1	PRIMARY KEY		(bookID)
2	FOREIGN KEY		(borrower) REFERENCES students (username)

Figure 8.4: Table relationship

We now try to set up a query on both tables to show the name and mobile number of the student who has borrowed a book of certain title and author. For this purpose, obtain a query object on both classes from the database session:

```
q=db.session.query(books,students)
```

Next, apply filter on this query so that only those objects of both classes with matching primary and foreign key attributes will be fetched:

```
rows=q.filter(books.borrower==students.username).all()
```

The result of query is a list of tuples. Each tuple consists of matching objects from both classes. It is, in turn, passed to a Jinja template

```
#app.py
from flask import Flask, request, flash, url_for, redirect,
render_template
from models import app, db, students, books

@app.route('/books')
def booklist():
q=db.session.query(books,students)
rows=q.filter(books.borrower==students.username).all()
return render_template('showbooks.html', rows=rows)
```

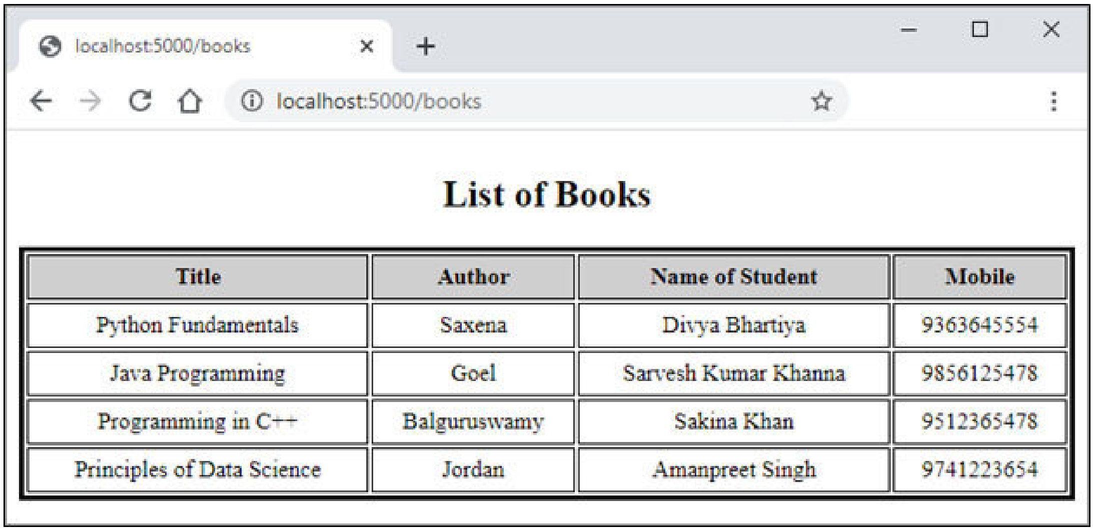
Inside the HTML code, Jinja loop renders details from books as well as students in a tabular form:

```
#showbooks.html

{% for book, student in rows %}
{% endfor %}
align="top"
    >
```

List of Books

Title	Author	Name of Student	Mobile
{{book.title}}	{{book.author}}	{{student.name}}	{{student.mobile}}

Start Flask server and use http://localhost:/books URL:



List of Books

Title	Author	Name of Student	Mobile
Python Fundamentals	Saxena	Divya Bhartiya	9363645554
Java Programming	Goel	Sarvesh Kumar Khanna	9856125478
Programming in C++	Balguruswamy	Sakina Khan	9512365478
Principles of Data Science	Jordan	Amanpreet Singh	9741223654

Figure 8.5: Model relationship with Flask-SQLAlchemy

Your browser should display the above output.

NoSQL Databases

Relational databases have been in use for more than 50 years now. However, but they are not found to be suitable in some use cases, especially when it comes to flexible data models, scalability, and agile development.

NoSQL database architecture is distributed and horizontally scalable. Unlike a relational database, the schema of a NoSQL database is flexible and can be modified anytime. Many real-time web applications, where a huge amount of data needs to be stored and processed, use NoSQL databases today. Some of the popular NoSQL databases are Cassandra and HBase along with MongoDB.

Depending upon the data model used, NoSQL databases are classified into four categories:

categories: categories:
categories: categories: categories: categories: categories: categories: categories: categories: categories: categories: categories: categories:

categories: categories: categories: categories: categories: categories: categories: categories: categories:
categories: categories:

categories: categories:

Table 8.3: Types of NoSQL databases

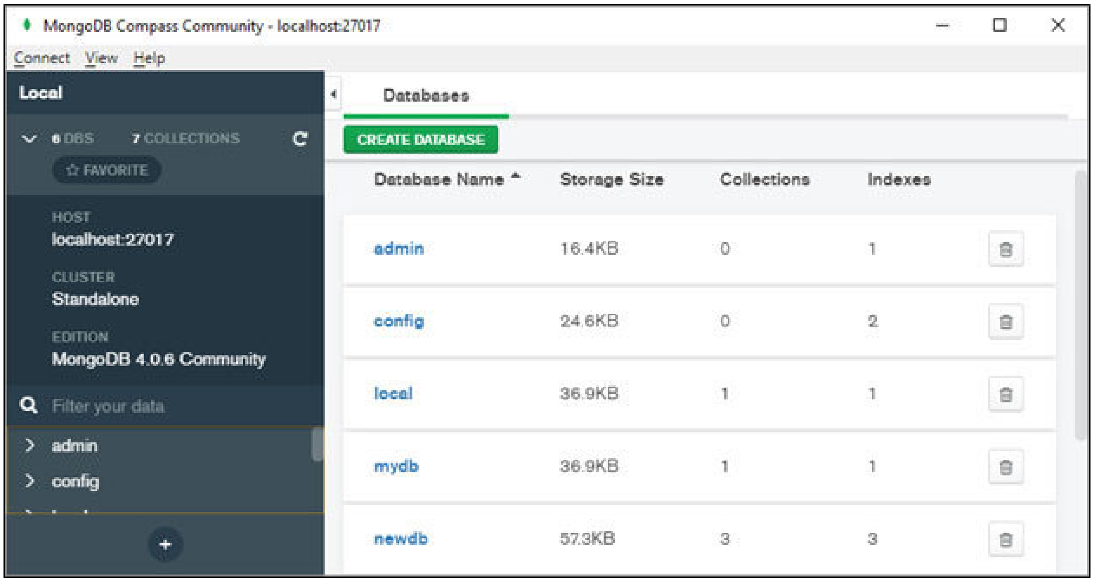
We shall restrict our discussion to using MongoDB in a Flask application. This software is developed by an American company MongoDB Inc. First of all, we need to download an appropriate version of MondoDB server community edition from https://www.mongodb.com/download-center/community and install it as per the instructions. Assuming that you are using Windows operating system and MongoDB is installed in the d:\mongodb folder, the server starts with the following command from the terminal:

```
D:\mongodb\bin>mongod
```

Local MongoDB server instance starts listening at port As in SQLite, one can also create MongoDB database in a console. To invoke MongoDB console, use the Mongo command:

```
D:\mongodb\bin>mongo
```

There is also an easy-to-use GUI tool for MongoDB, called Compass:



HOST
localhost:27017

CLUSTER
Standalone

EDITION
MongoDB 4.0.6 Community

Database Name ▴	Storage Size	Collections	Indexes
admin	16.4KB	0	1
config	24.6KB	0	2
local	36.9KB	1	1
mydb	36.9KB	1	1
newdb	57.3KB	3	3

Figure 8.6: MongoDB Compass

All databases currently available on the localhost are displayed as above.

Flask-PyMongo

Obviously, we need some programming interface that will enable a Python application to connect with MongoDB database. The PyMongo package is a native Python driver provided by MongoDB Inc. itself. Its Flask extension, further customizes it exclusively for use with Flask web application:

```
pip install flask-pymongo
```

While we do not intend to go much deeper into the architecture of MongoDB database, we need to understand a few terms, especially Collection and A MongoDB database contains one or more Collections, and a Collection may have one or more Documents. When compared with a relational database, Collection in MongoDB is equivalent of a table in relational database, and a document is similar to a row. However, there is a major difference – Collection doesn't have a predefined schema or structure as in an RDBMS table.

Document is a dictionary-like object with key-value pairs analogous to attribute and value present in a row. As there isn't a fixed schema, each Document in the same Collection may have variable number of key-value pairs. Just as in a table, there's one attribute defined as primary key, each Document is also uniquely identified by _id key, which is an auto-generated 24-character hex string.

With this much of terminology, let's understand how we can use Flask-PyMongo in a Flask application. As with most Flask extensions, application settings are updated, and an object of the main class in the extension is obtained. In this case, we should add the URL of our MongoDB database to Flask application:

```
from flask import Flask

from flask_pymongo import PyMongo

app = Flask(__name__)
app.config["MONGO_URI"] =
"mongodb://localhost:27017/mydata"
mongo = PyMongo(app)
```

Here, mongo is essentially our connection object for performing operations with the database and possesses db refers to the database itself. A new Collection can be added to it as follows:

```
obj=mongo.db.mycollection
```

To add a Document to this Collection object, the insert_one() method takes a dictionary of variable number of key-value pairs:

```
obj.insert_one({'key1': val1, 'key2':val2})
```

To add multiple documents at a time, use the insert_all() method providing a list of dictionary items. In the following Flask application, / route renders a registration form Form data is sent as HTTP POST request to the /addrec route. Its mapped view function parses form data as a dictionary with the help of which is inserted in the collection as a document:

```
#flask-pymngo-app.py
from flask import Flask, render_template, request
from flask_pymongo import PyMongo
```

```
app = Flask(__name__)
app.config["MONGO_URI"] =
"mongodb://localhost:27017/mydata"
mongo = PyMongo(app)

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

@app.route('/addrec',methods=['POST', 'GET'])
def addrec():
if request.method=='POST':
print (request.form.to_dict())
students_collection=mongo.db.students
students_collection.insert(request.form.to_dict())
return 'added successfully'
```

The MongoDB Compass utility can help us verify that a document is added in the database.

Flask-MongoEngine

Just as Python interacts with a relational database with an abstraction on top of a DB-API module in the form of SQLAlchemy, a similar ODM (Object Document Model) library is available for MongoDB. It is named as Flask-MongoEngine customizes it further for use with Flask applications by adding many helper utilities. As a final topic in this chapter, we shall take a brief look at the Flask_MongoEngine extension.

The installation of Flask-MongoEngine is easy as always:

```
pip3 install flask-mongoengine
```

As with the Flask extensions used in this chapter, Flask application object's configuration is updated by incorporating the MongoDB database URL:

```
from flask import Flask
from flask_mongoengine import MongoEngine
```

```
app = Flask(__name__)
app.config['MONGODB_SETTINGS'] = {
'db': 'mydata',
'host': 'localhost',
'port':27017
}
db = MongoEngine(app)
```

MongoEngine is an object relational mapper, so we need to create model class. In case of MongoDB database, the Collection (equivalent to table in RBMS) takes the name of the model class. Our Students model is as shown here:

```
import mongoengine as me
class students(me.Document):

name=me.StringField(required=True)
course=me.StringField()
gender=me.StringField()
mobile=me.IntField()
username=me.StringField(required=True,
primary_key=True)
password=me.StringField()
```

As we can see, the model class is inherited from the Document class. The instance attributes of this model correspond to the fields in the document. There are a large number of document field types defined in core MongoEngine; for example, and so on.

To add a document in the database, we need to initialize an object of model class and call its method. We shall populate the object with previously-used HTML form

```
#flask-mongoengine-app.py
@app.route('/')
def index():
return render_template('register.html')

@app.route('/addrec', methods=['GET', 'POST'])

def addrec():
if request.method=='POST':
student=students(name=request.form['name'])
student.course=request.form['course']
student.gender=request.form['gender']
student.mobile=request.form['mobile']
student.username=request.form['user']
student.password=request.form['pwd']
student.save()
```

```
return 'success'
```

The model class (in our case students class) has access to objects attribute, which returns the document set that can be traversed by a Jinja template, as we did while using plain SQL:

```
@app.route('/list')
def studentlist():
docs=students.objects
return render_template("studentlist.html", docs=docs)
```

Conditional retrieval of documents from collection is done by passing a keyword argument to objects attribute. For instance, all documents having C/C++ in course field will be returned by the following statement:

```
docs=students.objects(course="C/C++")
```

Deleting a document is very easy: just call its delete() method. Finally, reissuing save() by changing one or more attributes of a document will perform the update operation.

Conclusion

In this chapter, we learned how to provide database support to a Flask application. To begin with, we studied how Flask interacts with a relational database (with specific reference to SQLite) and followed it up with harnessing powerful ORM features of Flask-SQLAlchemy.

In the latter half of this chapter, we got acquainted with a popular NoSQL database – MongoDB. Python interacts with this database with the PyMongo package. We learned how to use the Flask-PyMongo extension and its ORM version called Flask-MongoEngine.

In this chapter, we also came across the concept of Flask extension. In the next chapter, we shall explore a few more extremely useful extensions such as flask-wtf and

CHAPTER 9

More Flask Extensions

Introduction

In the previous chapter, we came across a couple of Flask extension libraries and An extension is helpful for incorporating certain additional functionality to core Flask API, which is a micro framework.

In this chapter, we shall get acquainted with some of the very popular Flask extensions, as listed below.

Structure

Flask-WTF

Fields

Validators

Form class

Rendering Form object

Flask-Login

Flask-Bootstrap

Other Useful Extensions

Flask-Admin

Flask-Mail

Flask-Uploads

Objectives

After studying this chapter, you will be able to:

Design web forms using Flask-WTF

Manage user's session using Flask-Login

Build web application with responsive design using Flask-Bootstrap

Get a brief overview of other useful extensions

Flask-WTF

HTML form is the most common tool for accepting user input to a web application. It has been used more than once in the previous chapters of this book as well. However, HTML lacks the capability to validate user input. One can always rely on JavaScript event handlers for this purpose. Client-side validation of data has its disadvantages, though, as it can expose the server to malicious attacks. Moreover, the server-side script that intends to process user input has to recreate the form elements into Python variables. Hence, it is important to employ server-side validation mechanism.

Python ecosystem has WTForms - a very popular form rendering and validation library. Flask-WTF extension customizes it for use with Flask API. The ability of this extension to prevent CSRF Site Request attacks is one of its main features. Along with this, Flask-WTF supports reCAPTCHA and can handle file uploads.

When using a Flask extension, we first update Flask object with extension-specific settings and then obtain extension's main object. (See initialization of SQLAlchemy object in *Chapter 8: Using* In the case of this is not needed as the package contains ready-to-use the FlaskForm class.

FlaskForm is a declarative base class for user-defined form whose object shall represent the web form that will be rendered in a template. To start with, import FlaskForm and use it as base class to define the RegistrationForm class. Class variables of this class are objects of the Field classes defined in the WTForms library:

```
from flask_wtf import FlaskForm

from wtforms import TextField
class MyForm (FlaskForm):
name = StringField("Name")
```

Apparently, this looks similar to a Model class in SQLAlchemy's ORM. Just as variables in a Model class belong to Column types, there are different types of field classes in Each field is equivalent to the corresponding HTML form element. For example, when as defined above, is rendered, StringField translates to HTML form's text input element.

Fields

The first parameter to constructors of the Field classes is the label that identifies the input element. A field may have a validator parameter that determines the policy of validation of the data entered in the field. Certain fields need specific parameters. Other frequently used WTForm fields are as follows:

Most basic type of field to accept one-line text input. When rendered, it represents the type="text"> HTML form element.

This is also a text field customized to accept integer input.

This field is useful for entering a number with fractional part.

This field represents the checkbox element of HTML form and records its checked status (True or False).

```
resp = BooleanField('I agree to TOS',
[validators.DataRequired()])
```

A group of radio buttons is presented for the user to choose any one. This field is equivalent to the type="radio"> element of HTML form.

This one is similar to RadioField in the sense that it also provides user with multiple options, except that the options populate a drop down element. It renders the type="SELECT"> element with options filled by items in a list parameter.

The constructors of SelectField and RadioField use a keyword parameter named as It is a list of tuples with which their collection of items is populated. Each element in the list is a two-element tuple of value-label pairs:

```
Gender = RadioField('Gender', choices=[('M','Male'),
('F','Female')])
```

branch= SelectField('Branch', choices=[('IT', 'Information Technology'), ('CS', 'Computer Science'), ('Mech', 'Mechanical Engg')])

This field is used to provide a multi-line text input.

This is a convenient form of StringField that conceals the characters entered in it.

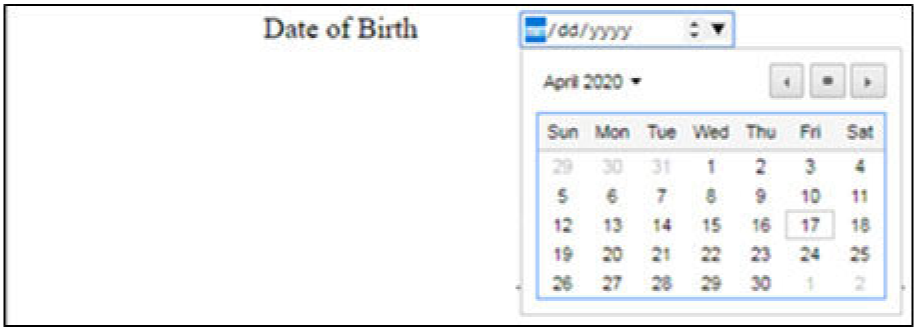
This is also a StringField modified to hold text as date. The format parameter allows users to enter date in the specified format:

dob=DateField("Enter date of birth", format="%d-%m-%Y")

This is another customized text field emulating type="hidden"> as in HTML. Flask-WTF automatically generates this field for CSRF token to prevent malicious attacks.

This field represents the type="submit"> tag and presents a submit button so that form data is submitted to the mentioned URL.

Additionally, the wtforms.fields.html5 module provides similar fields that follow the HTML5 standard. For instance, this module has EmailField that translates to the type="email"> tag and URLField that represents the type="url"> tag. DateField in this module provides a convenient DatePicker widget:



Sun	Mon	Tue	Wed	Thu	Fri	Sat
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	1	2

Figure 9.1: DateField with DatePicker widget

It is also possible to configure the date format for the desired sequence of date, month, and year numbers.

Validators

The WTForms library also consists of several validators, defined in the wtforms.validators module. This is an important aspect of the WTForms library. A validator, when attached to a certain form field, verifies if the input satisfies the given condition and raises ValidationError if the condition fails. Any number of validators can be attached to a field. Some of the frequently used validators are listed here:

This validator ensures that a certain field is not left empty by the user before submitting the form. This can be useful in maintaining NOT NULL constraint on a field in the corresponding database table:

```
student=StringField("Name of Student",
[validators.InputRequired("Please enter your name.")])
```

This validator checks whether the data attribute of a field returns True value. A message parameter contains a string that will be displayed if validation fails. A typical use case of this validator is when the user has to tick a checkbox indicating that certain terms and conditions have been accepted:

```
resp = BooleanField('I agree to TOS',
[validators.DataRequired()])
```

This validator is applied to any numeric form field (for example, IntegerField or You may want to force the user to input a number within a stipulated range, such as marks between 0 and 100. In this situation, the NumberRange validator is used:

```
marks = DecimalField("marks", validators=
[NumberRange(min=0, max=100, message='Marks out
of accepted range')])
```

On the other hand, this validator is applied on a string field whose input should not exceed a certain number of characters. For example, a password should be minimum 4 characters and maximum 10 characters long. The Length validator is appropriate for such cases.

A normal StringField acts as a placeholder for entering However, in order to verify if the entered text conforms to the format of a valid email address, we need to apply the Email validator on it.

Again, this validator is applied to a StringField to check whether it contains a valid URL:

```
email = StringField("Enter Email ID",
[validators.Email("Please enter your email ID.")])
```

We often come across a web form where it is required to choose a password and then retype the same in the following field. An error message pops up if both fields do not match. This is exactly where the EqualTo validator is applied. It compares the values of two fields and generates an error message if the comparison fails:

```
password = PasswordField('New Password', [validators.
EqualTo('confirm', message ='Passwords must
match')])
confirm = PasswordField(Re-type the password')
```

We shall use some of these validators in a subsequent example.

Form class

Let's put this functionality of the WTForms library in practice. The following AdmissionForm class represents a typical web form and makes use of different form fields and validations. We shall use an object of this class to render a form in a web page template:

```
#myform.py
class AdmissionForm(FlaskForm):
student = StringField("Name",
[validators.InputRequired("This field can not be
empty")])
Gender = RadioField('Gender',
[validators.DataRequired('Choose Gender')], choices=
[('M','Male'),('F','Female')])
dob = DateField("Date of Birth", format="%d-%m-
%Y")
percent = DecimalField("passing percentage",
[validators.NumberRange(min=0, max=100,
message="Invalid input")])
```

```
branch= SelectField('Branch', choices=[('IT',
'Information Technology'), ('CS', 'Computer Engg.'),
('E&T', 'Electronics & Telecom')])
userID = StringField('User Id',
[validators.Length(min=4, max=8,
message='ID length between 4-8 chars')])

pwd = PasswordField('Enter Password',
[validators. EqualTo('confirm',
message ='Passwords must match'),
validators.InputRequired("This field cannot be
empty")])
confirm = PasswordField('Re-type the password')


submit = SubmitField('Submit')
```

Next, we shall learn how a form of the above design is rendered in a template web page.

Rendering the WTForm object

How do we display an admission form that represents the above class? By using the jinja2 template, of course. First, we shall obtain an object of the AdmissionForm class, each attribute of which represents the fields defined in it:

```
from myform import AdmissionForm
form=AdmissionForm()
```

This object has to be passed to a jinja2 template that will render form's field elements dynamically:

```
@app.route('/')
def index():
```

```
form = AdmissionForm()
return
render_template("admission.html",
form=form)
```

Each field object has two important attributes: label and data. For example, form.student.label is while form.student.data itself will contain the data entered by the user. Hence, the following jinja2 template code will render HTML's form element:

{{form.student.label}}	{{form.student}}

for="student">Name	id="student"
	name="student" required
	type="text" value="">

```
{% for subfield in form.Gender %}
{% endfor %}
```

{{form.student.label}}	{{form.student}}
{{form.Gender.label}}	{{subfield}}{{subfield.label}}
{{form.dob.label}}	{{form.dob}}

```
{{form.percent.label}} {{form.percent}}
{{form.branch.label}}  {{form.branch}}
{{form.userID.label}}  {{form.userID}}
{{form.pwd.label}}     {{form.pwd}}
{{form.confirm.label}} {{form.confirm}}
{{form.submit}}
```

This form is submitted to /admission route which invokes admission() function. The function itself parses form data and renders it on successful validation:

#app.py

```
from flask import Flask, render_template, request, flash
from myform import AdmissionForm
app = Flask(__name__)
@app.route('/admission', methods=['GET', 'POST'])
def admission():
form = AdmissionForm()
if request.method == 'POST':
if form.validate() == False:
return render_template('admission.html', form=form)
else:
return '''
```

Form Data Received

Name :{}
DOB : {}
Branch : {}
'''.format(form.student.data, form.dob.data, form.branch.data)

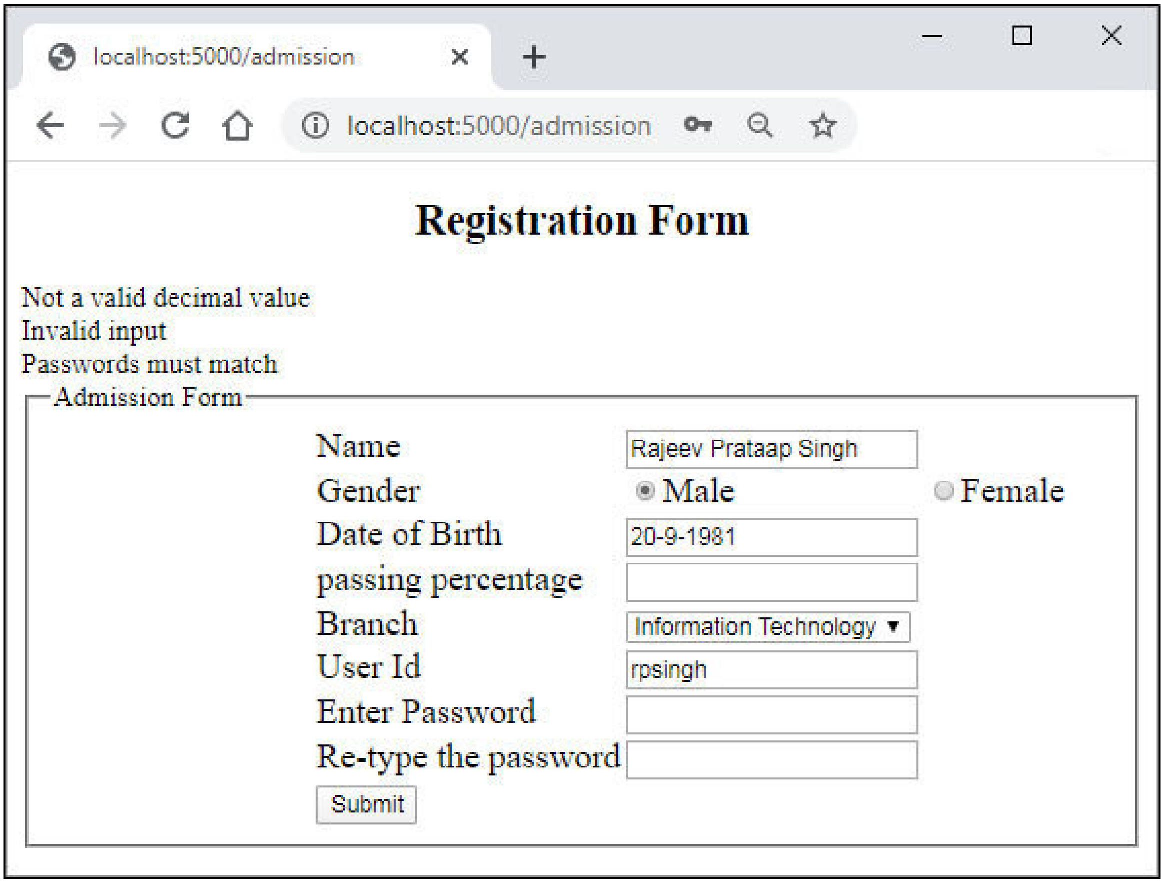
Here, we are calling the validate() method of the FlaskForm class that returns False if any of the validation rules fail. Corresponding error messages will be flashed on the browser if we include the following code in the template:

```
{% for message in form.student.errors %}
{{message}}
{% endfor %}
{% for message in form.userID.errors %}
{{message}}
{% endfor %}
{% for message in form.dob.errors %}
{{message}}
{% endfor %}
{% for message in form.percent.errors %}
{{message}}
{% endfor %}
{% for message in form.pwd.errors %}
{{message}}
```

```
{% endfor %}
```

After putting the Flask application code and the template in the appropriate place and then starting the server, **http://localhost:5000/** URL in browser opens the form. Incomplete or inappropriately filled form flashes an error message.

An example of an incorrectly filled form is as shown here:



Registration Form

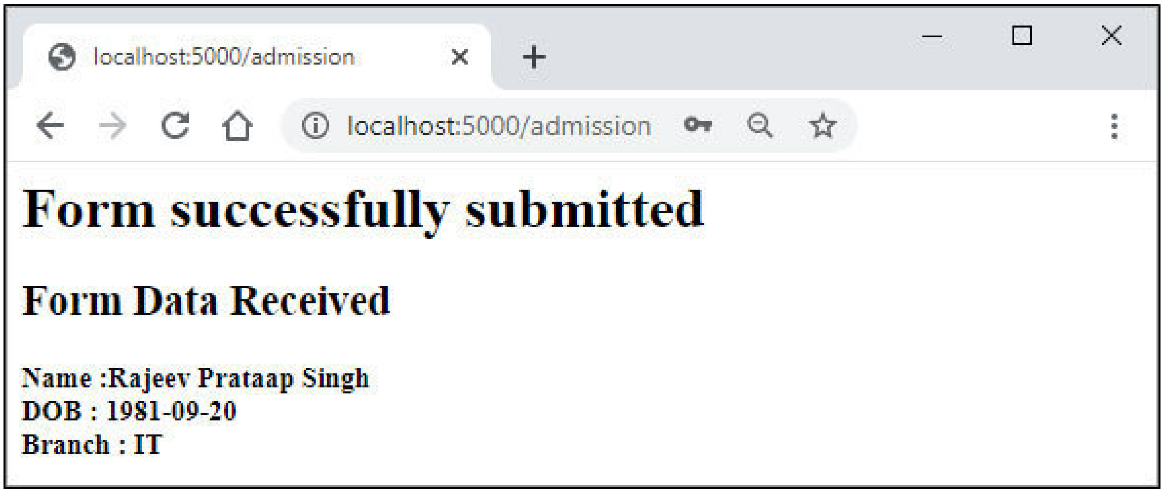
Not a valid decimal value
Invalid input
Passwords must match

Admission Form

Name: Rajeev Prataap Singh
Gender: ● Male ○ Female
Date of Birth: 20-9-1981
passing percentage:
Branch: Information Technology ▾
User Id: rpsingh
Enter Password:
Re-type the password:

Figure 9.2: WTForm with validation errors

If the form is successfully validated, the browser shows the result as follows:



Form successfully submitted

Form Data Received

Name :Rajeev Prataap Singh
DOB : 1981-09-20
Branch : IT

Figure 9.3: WTForm successful validation

It is now possible to perform any desired process on the data so fetched, such as adding it in a database table, etc.

Flask-Login

In many web-based applications, some content has unrestricted access – such as documentation, change log, use cases, etc. Anybody can consume this information. However, some other areas of the application require the user to be authenticated before giving access. For example, Amazon – an e-commerce marketplace – allows users to browse its product catalogue without any restriction but requires them to log in as a registered user for placing an order.

While this can be achieved by maintaining user's ID in a session variable and checking if it is set before allowing access, it becomes a tedious exercise. Flask-Login extension just performs this task without much hassle. It ensures that the condition of login is required is fulfilled before a certain page is opened. What is more, a default response page with 401 status code corresponding to unauthorized access is displayed if the condition fails.

So, let's explore the functionality of Flask-Login extension. First of all, install this extension with pip utility, as always:

pip install flask-login

The all-important task of handling session management of user's ID is performed by an object of the LoginManager class, whose constructor needs a Flask application object for initialization:

```
from flask import Flask
from flask_login import LoginManager
app=Flask(__name__)
login=LoginManager(app)
```

The next step is to prepare a User class. This class must have id as one of its attributes, whose value will be unique for each object. Of course, there may be other optional attributes of the user. This User class must provide the following methods and attributes:

If credentials of a user are authenticated, this property is set to Based on the value of this property, access to restricted resources is either granted or refused.

In addition to being authenticated, one must be an active user. If so, this property is set to

This property should be set to False for actual users and True for others (such as guests).

This mandatory method of the User class returns the id attribute of the object that should be a string and is used to load the user with the help of the user_loader callback.

Although one can provide customized implementations of the above-mentioned properties/methods, the User class can use their default implementation from In Python, the idea of mixin is somewhat similar to interface (as in Java), although their default implementations can be added in sub class without overriding. (That said, a detailed discussion of mixins is beyond the scope of this book.)

Flask-login library doesn't recommend using any particular type of database. However, using the Model class in SQLAlchemy makes it easy to build the user class. Hence, we shall define the User class based on the Model and UserMixin classes, as follows:

```
from flask_sqlalchemy import SQLAlchemy
from flask_login import UserMixin
app = Flask(__name__)
```

```
app.config['SQLALCHEMY_DATABASE_URI']='sqlite:///flaskdatabase.db'
app.config['SECRET_KEY']="any random string"
db = SQLAlchemy(app)
class User(db.Model, UserMixin):
id = db.Column(db.String, primary_key=True)
pwd = db.Column(db.String)
name = db.Column(db.String)
role = db.Column(db.String)
dept = db.Column(db.String)
salary = db.Column(db.Integer)

def __init__(self, id, pwd, name, role, dept, salary):
self.id=id
self.pwd=pwd
self.name=name
self.role=role
self.dept=dept
self.salary=salary
```

We can now create the User table (with the db.create_all() method) and add data, as we did in the previous chapter. Upon authenticating user's credentials, the User object is supplied to the login_user() function to do exactly the same – log the user in! Assuming that a form posts User ID and password data to /login route, the following mapped function authenticates and logs in the corresponding User object:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
if request.method=='POST':
id=request.form['uid']
pwd=request.form['pswd']
user=User.query.filter_by(id=id).first()
if user is not None and user.pwd==pwd:
login_user(user)
return ''
```

you are logged in.

href='/'>click to go back"

```
else:
return ''
```

Try again.

href='/'>click to go back"

Next, we should provide a function that maps the user_loader decorator defined in the LoginManager class. This function gets called every time the Flask server receives a request. The server retrieves the user ID stored in a session cookie and loads it as the current user:

```
@login.user_loader
def load_user(userid):
return User.query.get(userid)
```

The logged in user object is available in the current_user variable. Lastly, Flask-Login prevents unauthorized access to a view function with the @login_required decorator. Let's provide a /profile route mapped to the myprofile() view function that renders the profile information of currently logged user. This function is decorated by both the @app.route and @login_required decorators, ensuring a user's profile can be displayed only when he/she is logged in:

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

```
@login_required
def myprofile():
user=current_user
return render_template("myprofile.html", user=user)
```

The myprofile.html template uses the jinja2 syntax to render the details of the current user:

#myprofile.html

Welcome {{user.id}}

Name : {{user.name}}

Role : {{user.role}}

Dept : {{user.dept}}

href="/">Click here to go back

There's also a logout_user() function, which releases the session cookie. Obviously, it also needs to be called by a view function that is decorated by the @login_required decorator:

```
@app.route('/logout')
@login_required
def logout():
logout_user()
return "
```

you are logged out.

href='/'>click to go back"

The last piece in the application code is, of course, the initial template that displays a form for users to enter the ID and password. It opens as the **http://localhost:5000/** URL is entered after starting the server:

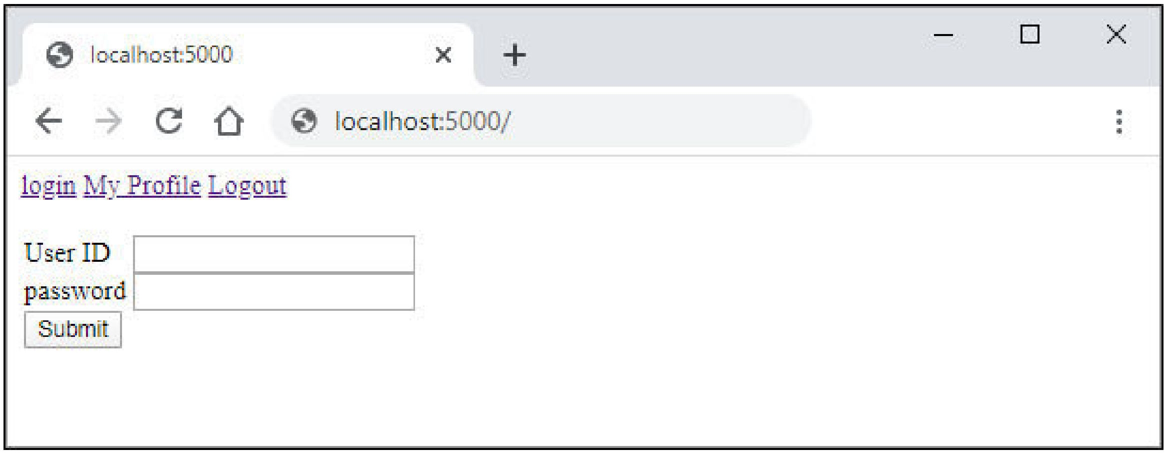


Figure 9.4: *Flask-Login*

Following is the HTML script of index page:

#index.html
href="/">login
href="/profile">My Profile

href="/logout">Logout

action="http://localhost:5000/login" method="post">

User ID

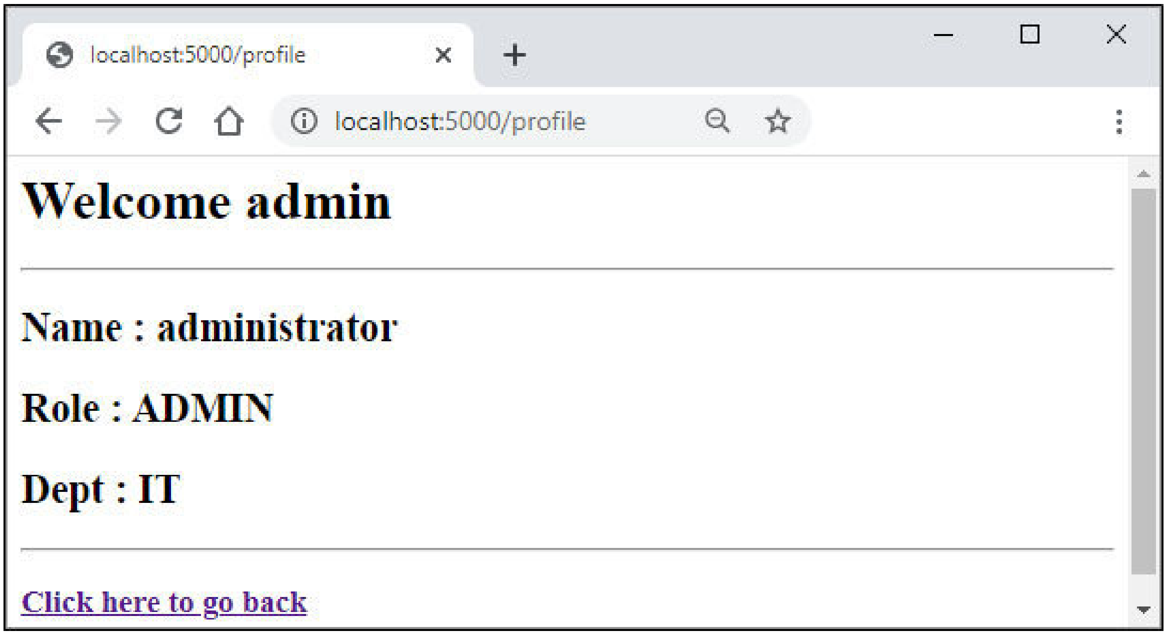
type="text" name="uid">

password

type="password" name="pswd">

type="submit" name="submit">

If the ID and password are authenticated, click on the profile link to view details of the current user:



Welcome admin

Name : administrator

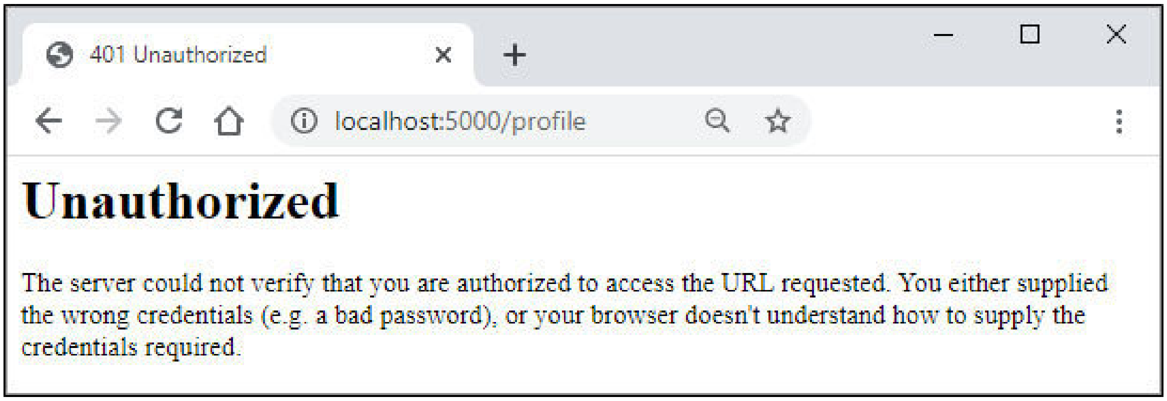
Role : ADMIN

Dept : IT

[Click here to go back]

Figure 9.5: *View with login_required decorator*

However, clicking on this link before logging in or logging out will display an error page with the 401 status code for unauthorized access:



Unauthorized

The server could not verify that you are authorized to access the URL requested. You either supplied the wrong credentials (e.g. a bad password), or your browser doesn't understand how to supply the credentials required.

Figure 9.6: *Unauthorized access error page*

As explained earlier in this book, we can design a custom web template for the 401 error.

Flask-Bootstrap

Earlier in this book *6: Static* we discussed how to use CSS and JavaScript in a Flask application. However, we haven't addressed the need to ensure that the content looks consistent on all browsers, operating systems, and devices of varying screen dimensions. Web page design has to adopt a *mobile first* approach in today's world with the ever-increasing use of smartphones and other handheld devices of different sizes. Customizing the page design for all these diverse factors (such a design is popularly called **responsive** can be extremely tricky.

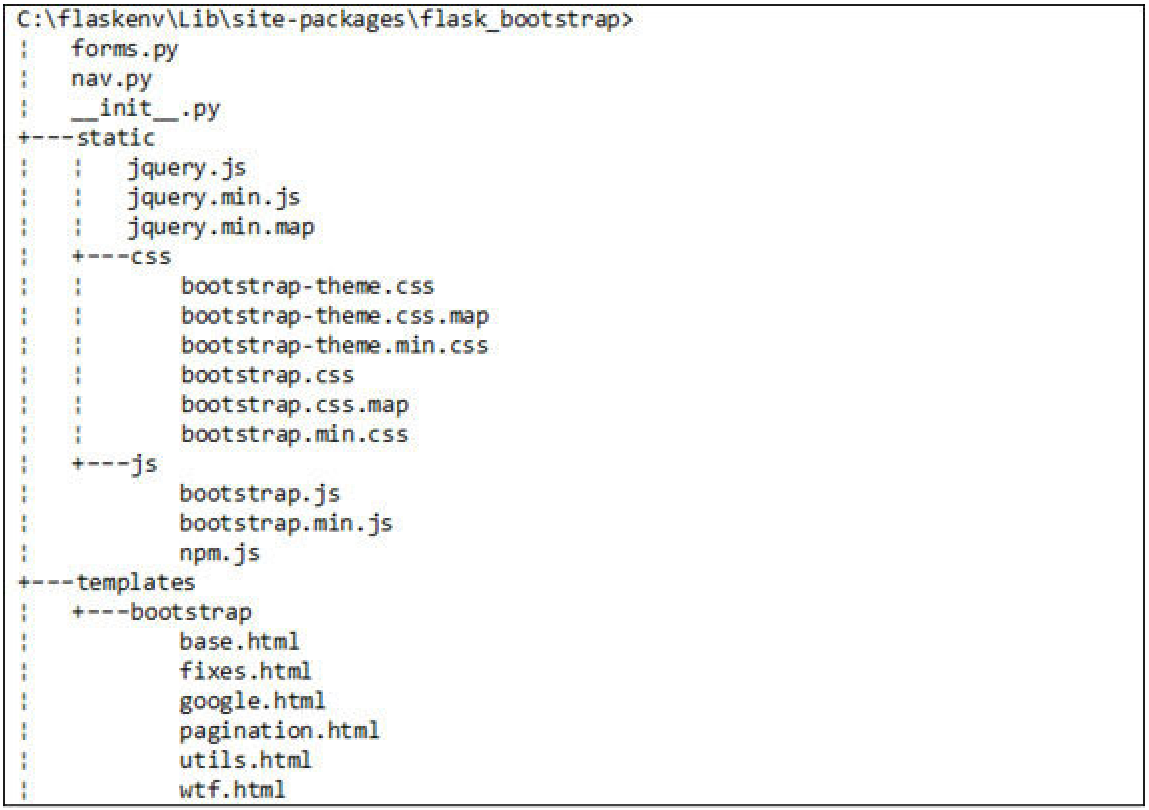
a mobile-first CSS front-end framework, developed by Twitter (a popular microblogging platform), makes this job relatively easy. It is a library of CSS and JavaScript templates for various interface components, such as navigation bar, buttons, etc. You can install these libraries locally or use it from Bootstrap **Content Delivery Network**

The Flask-Bootstrap extension beautifully integrates with Flask (in particular, with Jinja2 engine) and makes responsive design extremely simple. This extension also supports the WTForms and SQLAlchemy libraries.

First of all, start by installing this extension using the PIP installer:

```
pip install flask-bootstrap
```

This will install Bootstrap's CSS and JS files in the static\css and static\js folders in the bootstrap folder in site-packages of the current Python installation. It will also create a few templates in the corresponding templates folder:



```
C:\flaskenv\Lib\site-packages\flask_bootstrap>
¦   forms.py
¦   nav.py
¦   __init__.py
+---static
¦   ¦   jquery.js
¦   ¦   jquery.min.js
¦   ¦   jquery.min.map
¦   +---css
¦   ¦       bootstrap-theme.css
¦   ¦       bootstrap-theme.css.map
¦   ¦       bootstrap-theme.min.css
¦   ¦       bootstrap.css
¦   ¦       bootstrap.css.map
¦   ¦       bootstrap.min.css
¦   +---js
¦           bootstrap.js
¦           bootstrap.min.js
¦           npm.js
+---templates
¦   +---bootstrap
¦           base.html
¦           fixes.html
¦           google.html
¦           pagination.html
¦           utils.html
¦           wtf.html
```

Figure 9.7: *Flask-Bootstrap templates*

All files under the templates folder are really base templates, which a Flask application has to suitably inherit (Refer to Chapter 5: Rendering The most important among them is as we shall see shortly.

Next, instantiate the Bootstrap class, the main class in the flask_bootstrap extension:

```
from flask_bootstrap import Bootstrap
app = Flask(__name__)
```

```
bs=Bootstrap(app)
```

Now, we have to design a template. In order to make it responsive, it should inherit the above-mentioned base.html by including the following line at the top:

```
{% extends 'bootstrap/base.html' %}
```

Various template blocks together form the skeleton of this base template. As you know, the child template has to override one or more of these template blocks as needed. The most commonly used is content block. As a first example, of our application renders a customary Hello message in a responsive manner. We'll put this message in this block, as shown here:

#index.html

```
html>
{% extends 'bootstrap/base.html' %}
{% block title %}Flask-Bootstrap Example{% endblock %}
{% block content %}
```

align="center">Hello. Welcome to Flask-Bootstrap

{% endblock %}

Assuming that this template is served by / route of our application, you can open the browser on different devices to verify consistent appearance of the page.

In the preceding example, two template blocks (defined in have been overridden. This template has a few more convenience blocks defined for adding different CSS and JavaScript elements. They are listed in the following table:

table:
table: table: table: table:
table: table: table: table: table: table: table: table: table: table:
table: table: table: table: table: table:
table: table: table: table: table: table: table: table: table: table: table: table: table: table: table: table:

table: table: table: table: table: table: table: table: table: table: table: table: table:

table: table: table: table: table: table: table: table: table: table:
table: table: table: table: table: table:

Table 9.1: Flask-Bootstrap templates

For example, if a template needs myscript.js, it is included in {% block script %} and {% endblock

```
{% block scripts %}
{{super()}}
{% endblock %}
```

Note the use of the super() function. This is a jinja2 function that allows a block to amend instead of replacing it. As mentioned earlier, the Flask-Bootstrap extension has built-in support for the WTForms library. extension must be installed though). Along with another template is also downloaded. Some useful macros are defined in this template. They help in quick rendering of One such macro is As its name suggests, this will render the WTform object using predefined layouts. Three form types are available: basic, inline, and horizontal. Default value of the enctype parameter is If the form has it should be set to multi-part.

In the following example, we use the quick_form macro. First, we should have a A simple login form is as follows:

```
from flask_wtf import FlaskForm
from wtforms import *

from wtforms import validators, ValidationError
class LoginForm(FlaskForm):
username=StringField('username', [validators.InputRequired("This
field can not be empty"), validators.Email("Please enter your
email address.")])
password=PasswordField('Enter Password',
[validators.InputRequired("This field cannot be empty")])
submit = SubmitField("Submit")
```

This form will be rendered by /login route of Flask application:

```
from flask import Flask, render_template, request, flash
from flask_bootstrap import Bootstrap
from myform import LoginForm
app = Flask(__name__)
app.secret_key='1!2@3#4$5%'
bs=Bootstrap(app)
@app.route('/login')
def login():
form=LoginForm()
return render_template("loginform.html", form=form)
```

The loginform.html template uses base.html as well as The form is rendered with

#loginform.html

```
{% extends 'bootstrap/base.html' %}
{% import 'bootstrap/wtf.html' as wtf %}
action="http://localhost:5000/success" method="post">
{% block content %}
style="margin:3%;>
{{wtf.quick_form(form)}}

  {% endblock %}
```

Of course, you have option to render fields individually, as we did in the flask-wtf extension example:

```
{% block content %}
{{wtf.form_field(form.username)}}
{{wtf.form_field(form.password)}}
{{wtf.form_field(form.submit)}}
{% endblock %}
```

Here's how the automatically resizable login form appears:

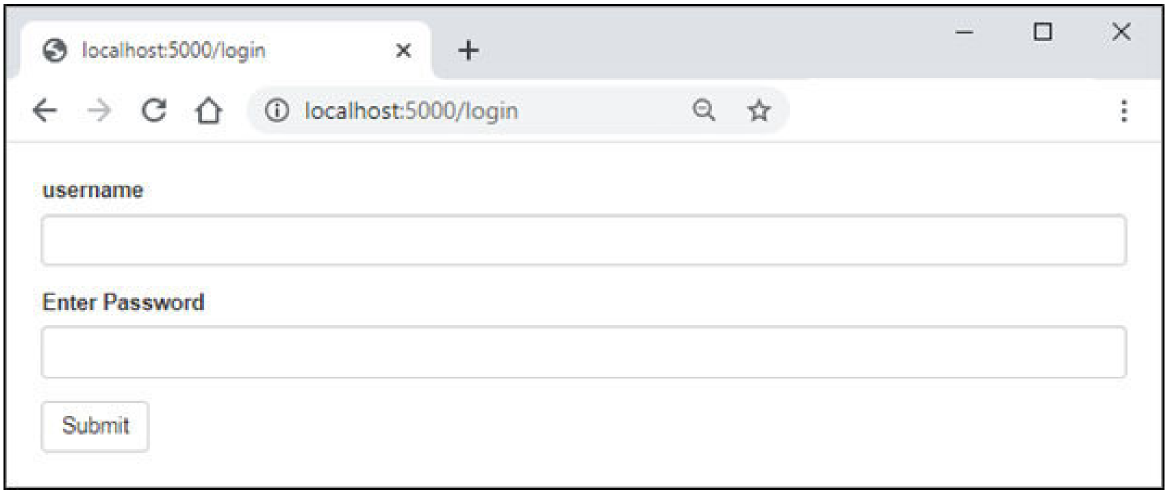


Figure 9.8: *Responsive login form*

To check the responsive design, try and open the above-mentioned URL on devices of different sizes and orientation.

Other useful extensions

Flask ecosystem has a large number of extension libraries. In this chapter (and the previous one), we learned how to use some of the most frequently used Flask extensions. We end this chapter with a brief introduction to a few more equally popular extensions.

Flask-Admin

As the name suggests, the Flask-Admin extension adds admin interface to the Flask application. An admin interface is an easy-to-use, menu-driven tool for performing CRUD operations on top of a data model.

Imagine that you want to provide a view in your Flask application that displays database records in a table with links to other views that perform Add/Edit/Delete operations. Such functionality will require elaborate coding of database operations, HTML templates, etc. Flask-admin saves all the labor work and creates beautiful views for the SQLAlchemy or MongoEngine models with very little code. All you have to do is declare a Model class and add its view to an instance of the Flask_Admin class:

#adminapp.py

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_admin import Admin
from flask_admin.contrib.sqla import ModelView
from models import book
app.config['SQLALCHEMY_DATABASE_URI'] =
'sqlite:///mydata.db'
app.config['SECRET_KEY'] = 'something'
db = SQLAlchemy(app)
admin = Admin()
admin.add_view(ModelView(book, db.session))
```

You can also build an admin interface for SQL-based databases by importing the appropriate class from the extension. This discussion is out of the scope of this book.

Flask-Mail

Often, it is required to be able to send email through a web application. You can easily set up SMTP for your Flask application with the help of the Flask-Mail extension. Like in case of all Flask extensions, you need to do the following:

Configure Flask application object by adding settings such as SMTP server and port, username and password, etc.

Obtain object of Mail class from Flask object

Set up the message with body, sender, and recipients' list

Call the send() method to send a message to all recipients

Although it is possible to use any dedicated/public SMTP server, the easiest way to use Flask-Mail extension is to use Gmail's SMTP server. The following brief example shows how to make your application send email using Gmail address:

```
from flask import Flask
from flask_mail import Mail, Message

app =Flask(__name__)
mail=Mail(app)

app.config['MAIL_SERVER']='smtp.gmail.com'
app.config['MAIL_PORT'] = 465
app.config['MAIL_USERNAME'] = 'test@gmail.com'
app.config['MAIL_PASSWORD'] = '*****'
app.config['MAIL_USE_TLS'] = False
app.config['MAIL_USE_SSL'] = True
mail = Mail(app)

@app.route("/")

def index():
msg = Message('Hello World', sender = 'test@gmail.com',
recipients = ['guest@gmail.com'])
msg.body = "I am using Flask Mail to send this
message!"
mail.send(msg)
return "Sent Successfully"
```

We can also integrate other mail services in Flask application if we can configure the Flask-Mail extension with the respective domain name and port, etc.

Flask-Uploads

Another very common requirement of any web application is letting the user upload resources such as images, documents, text files, etc. A straightforward but tedious way to do it is to fetch file part of the request object, read it using Python's built-in File API, and write its contents securely to a certain file on the server. The File-Uploads extension provides an easy-to-use abstraction layer over this process.

First of all, the upload form should either have a file button generated with or, if you are using a and the form's enctype should be set to Then, we need to create an UploadSet object, which is nothing but a single collection of files. The object is created by providing the name, list of extensions (default is and destination folder (the default is application root):

```
img=UploadSet("images", IMAGES)
```

Inside the /upload route, the desired file part from the request object is on the server by just calling the save()

method of the UploadSet object. That's all!

```
@app.route('/upload', methods=['GET', 'POST'])
def upload():
if request.method == 'POST' and 'img' in request.files:
filename = img.save(request.files['img'])
return("file saved.")
```

Flask-Migrate is another Flask extension worth mentioning here. Alembic is a Python tool that handles the migration of databases using the SQLAlchemy library. The Flask-migrate extension customizes Alembic for use with Flask applications.

The Flask-RESTful extension helps quickly build REST APIs. We shall learn more about REST API in a subsequent upcoming chapter.

The Flask-JSON extension provides better handling of JSON requests in a Flask application.

React JS has become extremely popular in recent days. It is a JavaScript library for building user interfaces. The Flask-ReactJS extension extends React JS support to Flask applications.

Python's package repository (PyPI) contains quite a few ready-to-download-and-use extensions practically for every requirement, but it is also possible to build a custom extension. That said, we shall not cover how to build an extension in this book.

Conclusion

In this chapter, we learned how to extend Flask application with very useful extensions. We discussed the Flask-WTF extension, which is a server-side form library, in detail, and we also looked at Flask-Login, which handles user management. Then, we learned how to add bootstrap support to Flask templates with the help of the Flask-Bootstrap extension.

We also covered a brief overview of some more extensions, including Flask-Admin, Flask-Mail, and Flask-Uploads, which make useful additions to the functionality of Flask application.

In the next chapter, we shall find out how to prepare a better structure of an application with the help of blueprints. We shall also discuss a few more advanced features of Flask library.

CHAPTER 10

Blueprints and Contexts

Introduction

Although described as a micro framework, one can very well build a Flask app involving more than one Application object, with one invoking another. In this chapter, we shall find out how we can separate the environment of one application from the other with the help of context.

We shall also learn to use application factory method to initialize Flask application using the application context. This chapter also describes the behavior of request context and its event handling.

We shall then discuss Blueprints, an advanced feature of Flask API that helps in developing a modular Flask application. We shall try to understand the use of Blueprints with the help of a small yet effective example.

Structure

Application package

Application context

Request context

Request callbacks

Application factory

Signals

Blueprints

Objectives

After studying the topics in this chapter, you will be able to use advanced features such as contexts, application factory, and blueprints to make Flask application more modular and extensible.

Application package

In all the examples discussed so far, our Flask application consists of a single Python script (most of the times named as that consists of one or more routes and associated views. Of course, there may be templates and static folders, and models and forms if required. While this approach may be suitable for simpler applications, it can prove to be detrimental as the complexity increases. If the business logic of the entire application is put in a single file, it eventually becomes difficult to maintain. If you are thinking of distributing your Flask application as a package, then a different approach should be taken.

Assuming that all resources of your application are put in the FlaskApp folder, it must have a __init__.py file for it to be recognized as a package. Flask application object is declared in this file. The folder has a views.py wherein all application routes are defined. In addition, there is a templates subfolder and static subfolder as usual. The __init__.py file should import this module. Finally, the application object is invoked from outside the package, either using command line interface or as a Flask script.

The structure of the flaskapp package is as follows:

Package initialization file is as follows:

```
#flaskapp/__init__.py

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

Application's routes are defined in a separate file

#flaskapp/views.py

```
from Flaskapp import app
from flask import render_template

@app.route("/")
def index():
return "Hello Flask"
```

Finally, invoke the Flask application from outside the flaskapp package with

#app.py

```
from Flaskapp import app
if __name__=='__main__':
app.run(debug=True)
```

Now, you can run the application using Command line interface of Flask:

Alternatively, you can execute it as Python script:

Now, this may work well for a simple use case wherein there's a single instance of Flask Application object. However, in more complex scenarios where there may be multiple Flask objects defined in separate modules. Importing these objects across different modules is likely to cause circular imports. This is especially prevalent when we develop reusable Blueprints (we'll discuss Blueprints later in this chapter).

Flask overcomes this issue with the help of application Instead of using the app instance directly, we can refer to it by a proxy called Let's understand how to deal with application context.

Application context

Flask actually provides two contexts – application context and request As the Flask application starts handling a request, it pushes both an application context and a request context. They are popped one after the other when the request ends. The application context generally has the same lifetime as a request context.

Generally, Flask automatically pushes an application context when handling a request, although it can be pushed manually. Any which way, as the application context is pushed (or activated), two special variables current_app and g are become available to the current thread of activity. The current_app acts as proxy to the Flask instance that is currently handling the request, whereas with the help of it is possible to store certain data temporarily and can be used inside any view function registered with the current app.

As mentioned earlier, the application context (as well as request context) is created and destroyed automatically.

However, if you try to call upon current_app outside its context, you will get a runtime error. Here's a simple

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

What happens if we try to access current_app before the application context is pushed?

```
>>> from app import app
>>> from flask import current_app
>>> print (current_app.name)
Traceback (most recent call last):
.


.
.
raise RuntimeError(_app_ctx_err_msg)
RuntimeError: Working outside of application context.
```

Python console also gives following explanation for the above exception:

"This typically means that you attempted to use functionality that neededto interface with the current application object in some way. To solve this, set up an application context with app.app_context()."

While Python interpreter is running inside the shell, there is no incoming request and therefore the application context isn't pushed yet. Hence, current_app variable is not available. Let's try to push the context manually and see whether current_app variable is available:

```
>>> from app import app
>>> from flask import current_app
>>>app_context=app.app_context()
>>>app_context.push()
>>> print (current_app.name)
app
>>>
```

The application context is manually pushed inside Python's context manager's with construct. To rewrite above console interaction, we have:

```
>>> from app import app
>>> from flask import current_app
>>> with app.app_context():
... print (current_app.name)


...
```

App

Resources such as views etc. are bound with corresponding Flask object inside the with block. As is the property of with statement, these resources not the current_app variable will be accessible outside. (If accessed, the runtime error mentioned above will be raised.) This effectively results in separation of logic related to one Flask app from another.

The application context also exposes another object named as flask.g which can be useful if you need to store some data during handling of a request. This g is a namespace object. Typically, it is used to store reference to some database object initialized within the application context:

```
>>> with app.app_context():
... g.db = init_db()
```

Request context

Have a look at following route in a typical Flask application:

```
from flask import Flask, request
app = Flask(__name__)
@app.route('/')
def index():
return "
```

Your IP address is {}

''.format(request.remote_addr)

The URL when visited, simply displays client's IP address. So nothing special about above code, you would say. There's one peculiarity in it that needs to be paid attention to though.

The way request object is used, one might think that it is a global object because index() function doesn't have it as an argument. Obviously it isn't and it can't be, because request by each client will be different and will be processed in different thread. Had it been a global object, our application wouldn't be able to distinguish between simultaneous requests.

Flask tackles this situation with the help of request context. Just as the application context, request context is also automatically pushed as Flask starts handling a request. This context will be unique to each thread, which means one request can't be passed to another thread.

The request context exposes two more proxies, request and session, which are available to the original thread in charge of handling current request. If request object is accessed outside its context, similar runtime error (as in case of application context) will be raised:

```
>>> from flask import Flask, request
>>> print (request.method)

Traceback (most recent call last):
.
.
.
raise RuntimeError(_request_ctx_err_msg)
RuntimeError: Working outside of request context.
```

Request context too can be pushed manually. For that, we need to use test_request_context() method of Flask class:

```
>>> from hello import app
>>> from flask import current_app, request
>>> with app.test_request_context('/'):
... print (request.path)
... print (current_app.name)
```

Both the request and application contexts work as stacks. The _request_ctx_stack and _app_ctx_stack respectively handle request context and application context instances.

When these contexts are pushed onto the respective stacks, the proxies that depend on them are available:

This variable is exposed when application context is pushed. It refers to the instance for the active application.

g: The global object is available as application context is pushed. This object is used for temporary storage during the handling of a request. Its contents are reset with each request.

When request context is pushed, this object is available. It refers to the request object and holds the contents of a client's HTTP request.

Pushing request context results in exposing this proxy. The user session is like a dictionary object stores values that are held between requests.

Request callbacks

We know that the @app.route() decorator binds a URL to specified view function so that every time a user visits the URL, the corresponding function is executed. Many a times, you may want to run a code before/after each and every request. For example, opening and closing database connection, checking user permissions, etc. One way could be to replicate the code in each view function, which, of course, is not the ideal approach. Thankfully, Flask provides the following useful decorators for the purpose:

Any function registered with this decorator will be executed before the first request after launching the application is handled.

A function that you want to be run before handling each request must be registered with this decorator.

As the name suggests, this decorator registers a function to execute after a request is handled. However, this function doesn't get called if the request handler encounters any exception. The function must accept a response object and return the same or new response.

Any function registered with this decorator will always be executed, whether or not the request handler raises any exception. Other than this, it works very much similar to the after_request decorator.

Here's a simple example. This Flask application code has two routes. The result of their view functions is quite straightforward. What we are interested in is the request callback functions:

#context-callbacks.py

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

@app.before_first_request
def first_request():
print ("This function is run before first request")

@app.before_request
def each_request():
g.user='guest'
```

```
print ("This is run before {} visits '{}'".format(g.user,
request.path))

@app.after_request
def after_request(response):
print ("This executes after exiting '{}'".format(request.path))
return response

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

Your IP address is {}

```
".format(request.remote_addr)
@app.route('/hello')
def hello():
string="
```

style='text-align:center'>Hello *{}*. Welcome to Flask

".format(g.user)
return string

While the application runs, the Python console shows the following log:

Incidentally, the above script also uses the global object g to store the value of a variable called user. Although it is hard coded, it can also be populated from a database, etc.

Application factory

Let's turn our attention back to the folder structure of the Flaskapp application. Here, the __init__.py script merely initializes Flask object. However, you may need to customize its configuration or objects of some extensions, such as flask-wtf or etc. This activity is conveniently performed by a factory method called

What is a factory method? The application factory method pattern allows you to create multiple instances of this app later. This provides a very elegant and scalable approach, especially when it comes to writing tests for the app.

The application factory function declares the Flask object, optionally updates its configuration settings, initializes one or more extension objects, and returns the Flask object itself.

In our application folder, there is a forms.py that contains WTForms classes. There is also which defines classes to map database table structure. Obviously, you have use flask-sqlalchemy and flask-wtf extensions in your application. The create_app() function is defined in __init__.py and is used to perform the following actions:

Create a Flask app object.

Initialize extension objects.

Update configuration with extension specific attributes.

Import application logic defined in routes.

The modified __init__.py in the package folder is now as follows:

```
from flask import Flask

from flask_sqlalchemy import SQLAlchemy

db=SQLAlchemy()
defcreate_app():
app=Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI']='sqlite:///mydata.db'
db.init_app(app)
with app.app_context():
from . import views
```

```
return app
```

The create_app() function is called from which is outside the package folder:

```
from Flaskapp import create_app
app=create_app()
if __name__=='__main__':
app.run(debug=True)
```

Flask will automatically detect the factory or in the run.py file. As before, you can run the application with Flask command line interface or as a script.

Signals

Another useful feature of Flask is signal handling, which is somewhat similar to context callback hooks discussed in previous section (albeit with certain differences). In general terms, signal is a notification issued to all those interested (who have subscribed) that a certain event has occurred. For example, a program running in console immediately receives keyboard interrupt signal or causing termination of running program.

In Flask, it is possible to decouple applications by sending notification when certain event occurs somewhere. A Flask application can set up a signal object and send notification to subscribing function within the application or another application as well.

Flask's signal support depends on Blinker library. Obviously, it must be explicitly installed using pip utility:

```
pip install blinker
```

This library consists of a Namespace class. Set up a Namespace object and create a new signal as follows:

```
from blinker import Namespace
nmspc = Namespace()
test = nmspc.signal('test')
```

There are two aspects to this. One is emitting a signal, other is subscribing to it. There is send() method of this signal object for former action:

```
signal.send(senderObj, **kwargs)
```

Using connect() method, other object can subscribe:

```
signal.connect(receiverObj, senderObj)
```

Here's a small example to demonstrate this behavior. The Flask route @app.route('/') emits signal with a string message argument containing client's IP address. On the other hand, test_signal() function subscribes to the signal object:

#flasksignal.py

```
from flask import Flask, current_app, request
from blinker import Namespace
import logging
logging.basicConfig(level=logging.DEBUG)

app = Flask(__name__)
app.secret_key = 'qwe123'

nmspc = Namespace()

deftest_signal(app, message, **extra):
logging.debug(message)

test = nmspc.signal('test')
test.connect(test_signal, app)

@app.route('/', methods=['POST', 'GET'])
def home():
test.send(current_app._get_current_object(), message='Send
client IP address : '+request.remote_addr)
return 'Hello World'
```

As you can expect, Python logger records remote IP address, whenever a client visits endpoint. However, the subscribing function or object may be programmed to perform more useful process such as updating a database file, etc.

As mentioned in the beginning of this topic, this behavior appears similar to @before_request decorator. However, signals are intended only to notify the subscribing object, and not modify the response. The request callback hooks on the other hand can modify or even abort the response. Signal handlers are not executed in any undefined order, again unlike request handlers – they are executed in a specific order.

Blueprints

We now come to a very interesting feature of Flask API. We know that Flask is classified as a 'micro' framework. Hence, entire Flask application can be accommodated in a single Python script that may have routes, models, forms all put in a single file (although we may put models and forms separately, it is not entirely necessary) with templates and static assets provided separately. For a small application, it may be a good idea. But for more complex use cases, where there are more than one distinct activities are involved, putting everything in a single script, it is certainly not a good idea.

Let us take a case in point. Suppose we have to build a web application for an educational organization, running two institutes – one offering Engineering courses and other Management courses. Apparently, their functionalities are distinct and do not overlap. Hence, we need to separate their logic and resources in two different package folders, with each one containing an independent application, and then combine the two to build application for parent organization. This is a typical scenario where Blueprints are useful.

A blueprint is a collection of related views, templates, static assets and other resources pertaining to one application. Let us say the views in engineering application are represented by etc. All resources such as templates, static files along with routes, etc. are assembled in one package folder such that it can become part of any institutional application. In that sense, a blueprint is a reusable application with all ingredients of a typical Flask app, but it can't be run on its own. Instead, it is used as a part of a bigger application.

Let us first rewrite our baseline Hello World app as a blueprint app. Inside of instantiating Flask app as always, we'll have to create Blueprint object and register a view function on its route decorator. Let the Python script be as follows:

#bp.py

```
from flask import Blueprint
bp = Blueprint('bp', __name__)
@bp.route('/')
def index():
return "
```

Hello World

"

Blueprint constructor needs two mandatory arguments, name of the blueprint and import_name which is usually __name__ that helps locating root path of blueprint. A blueprint object also possesses same set of decorators and functions such as etc. However, they must be registered on blueprint object rather than Flask object as we have done so far. (Note use of @bp.route instead of

As mentioned a blueprint object can't run on its own. Instead it has to be registered with a regular Flask object. In following we import above bp object and register it as follows:

#app.py

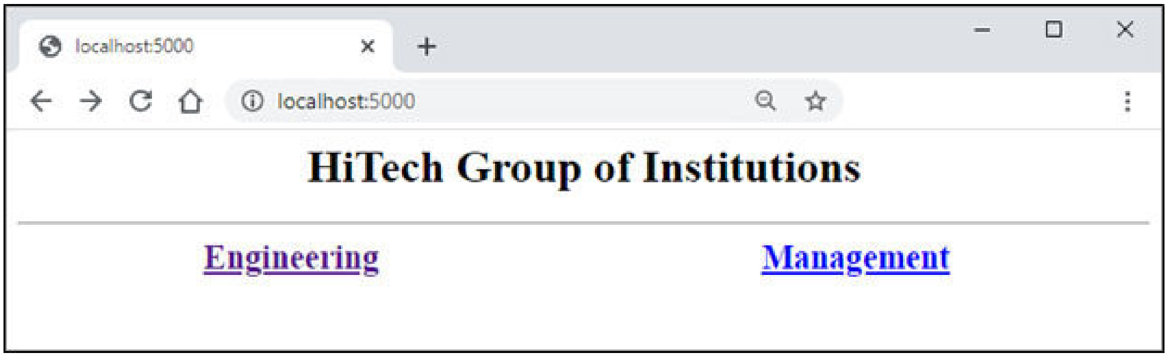
```
from flask import Flask
from bp import bp
app = Flask(__name__)
app.register_blueprint(bp)
```

We use register_blueprint() method of Flask class. One required argument is the blueprint object, although it may have other optional argument as we'll see later. Above Flask app is run and http://localhost:5000/ is visited in browser as usual. Flask automatically associates URL with index() view function registered with blueprint object when dispatching the request.

This fairly innocuous example is given merely to show the syntax of declaring and registering blueprint. Real power of blueprints comes to fore when there are multiple independent components in a larger application. Let us expand on the use case of an educational trust running two different educational institutes to implement concept of blueprints.

There can be two distinct approaches of organizing the resources of application with multiple resources. One may be to put of all blueprints in one folder, static assets in second and all routes in third. Second and more preferred approach is to make a separate package folder for each blueprint, having respective templates and static files in subfolders, and put them as subfolders of main application folder.

Here we have an organization called Hi Tech Group having two establishments, Engineering and Management Institutes. We shall try to build a Flask application whose home page should appear as follows:



HiTech Group of Institutions

Engineering

Management

Figure 10.1: *Home page of Blueprint based app*

Routes, templates and static files of two components are organized as shown in following structure:

The main application folder HiTech has two subfolders - engg and mngmnt – each for respective blueprint resources. For instance, in engg folder, engg blueprint and routes registered with it are defined in routes.py as follows:

#engg/routes.py

```
from flask import Blueprint, render_template
engg=Blueprint('engg',
__name__,template_folder='templates',static_folder='static')
@engg.route('/')
def index():
```

```
return  render_template('engindex.html')
@engg.route('/courses')
def  courses():
return  '
```

list of courses in Enginnering

```
'
@engg.route('/faculty')
def faculty():
return '
```

list of Engineering faculty members

```
'
@engg.route('/form')
def form():
return render_template('form.html')
```

Here, certain additional arguments are provided to Blueprint() constructor. The template_folder specifies name of folder in which this blueprint's templates are stored. This is relative the root path of blueprint. Likewise, you may specify path of static folder. There are a couple of routes attached to engg object (again, not to the main Flask object). Obviously, the templates and other resources are put in respective places.

Almost similar arrangement is there in mngmnt folder's routes.py with mngmnt as name of blueprint object there. Both the folders have __init__.py file that makes the folders recognized as Python package.

Now comes the important part of registering our blueprints. This is done in __init__.py file of main the application folder. It uses Flask's application factory function create_app() to set

Flask app and register above blueprint objects within its application context. Code of __init__.py is as follows:

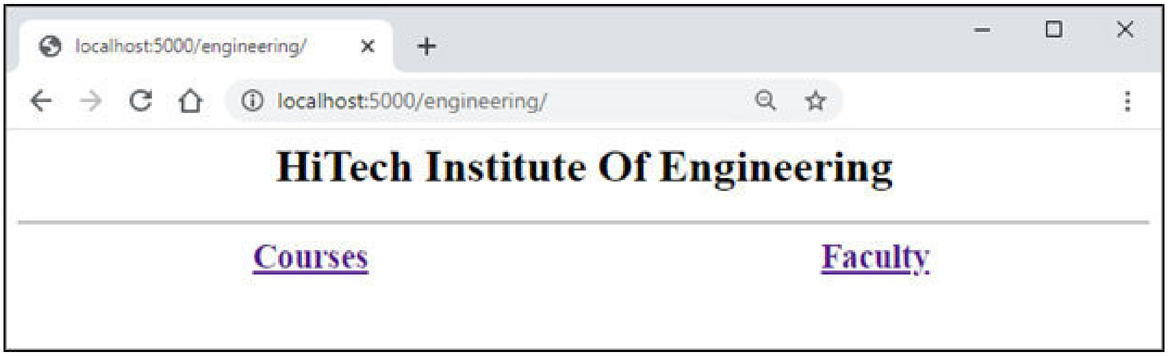
```
from flask import Flask, render_template

def create_app():
app=Flask(__name__)
with app.app_context():
from HiTech.engg.routes import engg
from HiTech.mngmnt.routes import mngmnt
app.register_blueprint(engg, url_prefix='/engineering')
app.register_blueprint(mngmnt, url_prefix='/management')
return app
```

First the two blueprint objects are imported from respective places and then registered. Providing url_prefix argument to register_blueprint() method allows corresponding blueprint routes distinctive URLs. For example, Flask server will dispatch /engineering/courses URL to /courses route in file and /management/courses URL takes the application to /courses route in Also, note that the main application may have its own templates and static files assembled inside in templates and static folder respectively.

How do we run our HiTech Flask app? As before, go to the root of this folder, create app.py in which call the application factory create_app() to set up Flask app and run it either with Flask CLI or as script:

```
from HiTech import create_app
from flask import render_template
app=create_app()

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

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

Go ahead and run the application and visit the root URL **http://localhost:5000/** to display HiTech Group's home page as shown before. Click on **Engineering** link will render following display in browser:



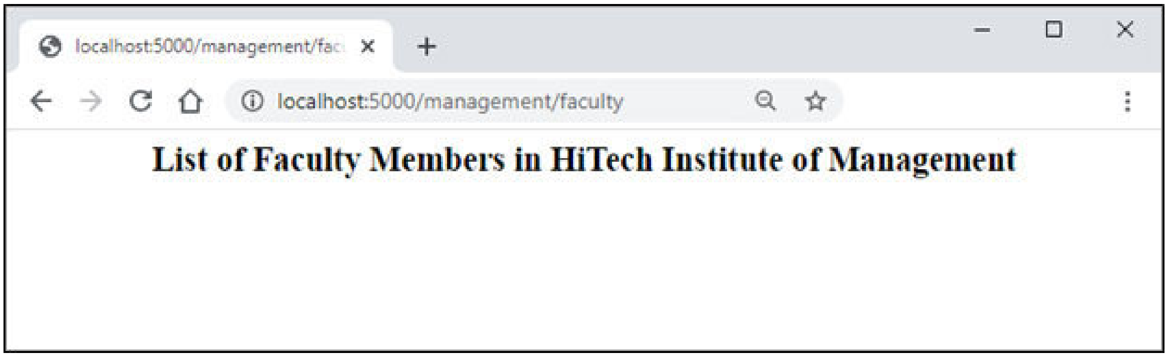
HiTech Institute Of Engineering

Courses

Faculty

Figure 10.2: *Blueprint Homepage*

This is obviously the index (or home) page of engg blueprint. Similar such homepage for mngmnt blueprint will be displayed in response to **http://localhost:5000/management** link. Its **Faculty** page will be as follows:


localhost:5000/management/fac
localhost:5000/management/faculty

List of Faculty Members in HiTech Institute of Management

Figure 10.3: *Blueprint route*

Thing to keep in mind is that each blueprint is a complete application on its own. Although this example shows a basic structure, we can have database operations, forms support and use other Flask extensions as we would do with a main Flask app. In fact, we can event test each blueprint's functionality in isolation by temporarily registering it with Flask object inside blueprint folder. That makes the application development process modular, scalable, and extensible.

Conclusion

In this chapter, we have explored many advanced features of Flask API. Starting with Application and request context objects, concept of using context callbacks before and after a Flask route has been discussed. We learned how to configure Flask application object with Application factory function.

Another extremely important part of Flask API, called Blueprints is covered in detail with a simple but effective example has been covered in this chapter, explaining how blueprints help in building modular applications.

Next chapter will introduce another powerful feature – building REST API using Flask. In short, REST (stands for **Representational State** is an architectural style of developing an interface to resources of a web application (such as database) so that other applications could interact with it. REST API uses HTTP requests and to perform CRUD operations in response to requests from other application. Let us discuss this concept in detail in next chapter.

CHAPTER 11

Web API with Flask

Introduction

So far in this book, we have discussed different aspects of building a comprehensive Flask-based web application – including managing database resources with corresponding Flask extensions. Now, the question is, can such resources be provided controlled access to other software applications? This is where API comes into the picture.

In this chapter, we shall learn to build a REST API, first with Flask's core routing mechanism, and then using RESTFul extension.

Structure

In this chapter, we shall discuss the following topics:

What is API?

REST

Route based REST API

What is cURL?

Class based Views

Flask-RESTful

Postman App

Objective

After we study this chapter, we will be able to:

Build a basic REST API app using Flask

Use Pluggable Views

Build API with Flask-RESTful extension

Test API with cURL and Postman app

What is API?

API is an acronym for Application Programming Interface. In general, an interface is like a window for give and take of objects between two otherwise isolated environments. The exchange between them takes place as per a set of mutually agreed rules and procedures. A computer's CPU, for instance, interacts with external devices like keyboard, mouse, printer, etc. through suitable interfaces. API, on the other hand, defines the rules of interaction between two software applications.

Knowingly or unknowingly, we routinely use lot of APIs (even if one is a normal user of various apps and websites and not a developer as such). We normally come across websites that let the user log in using Facebook, Google, or LinkedIn account. This website is able to tap into and authenticate users' credentials using Facebook API. Google Map API is often used by a third-party website to display its location on its home page. Similarly, AADHAAR API can be integrated with any application for identity verification. Various payment apps perform transactions through API exposed by bank's server.

So, an API is a set of request and response formats as well as endpoints exposed by an application such that any other software is able to add, retrieve, and update information in a prescribed manner. A web API is a case of web application letting its resources be accessed by others through the Internet (it may be other web application or a mobile app for Android/IOS platform) under well-defined set of protocols.

Different approaches available for building a web API are as follows:

RPC (Remote Procedure Call) approach is the simplest form of API. It involves invoking one of the functions (procedures) available on another computer on a network. Either XML or JSON representation is used to encode data transaction for these calls.

SOAP (Simple Object Access Protocol) has been in use for the development of web APIs since early 90s. A well-defined format of data and methods with the help of **WSDL Services Definition**

Unlike the above two, REST, which stands for **REpresentational State** is a set of principles (and not a protocol) on which the API architecture is based. REST has become a de-facto standard in modern-day web application development. The following figure shows the constituents of REST architecture:

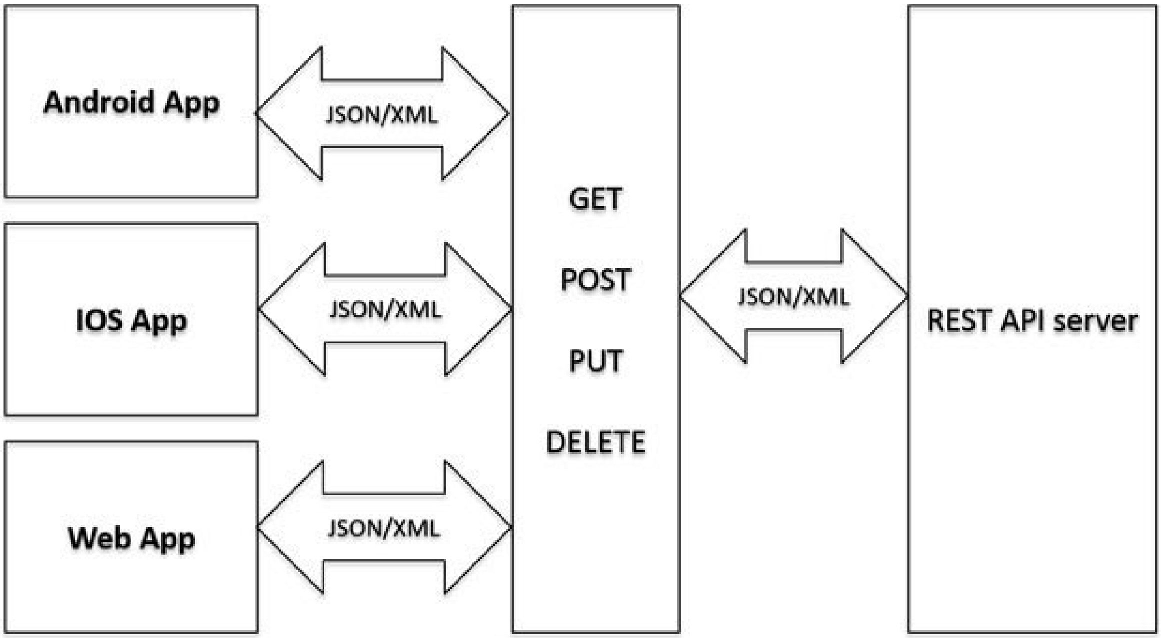


Figure 11.1: *REST architecture*

The principles of REST architecture are explained in the next section.

REST

The architecture of REST-based web services and APIs was defined by Roy Fielding in the year 2000. A RESTful API must comply with the following constraints as stipulated by him:

Uniform Obviously, this is a fundamental constraint for the design of a REST-based system. It greatly simplifies the architecture, allowing each part to evolve separately. A request from the client identifies resources by URIs. The server sends data representation in the form of HTML, XML, or JSON format. Each type of action to be performed on a resource is represented by HTTP verbs – and

Client-server Uniform interface achieves separation of concerns between the client and server. Both are kept independent of each other, thereby both can scale without affecting the other. Client-server architecture is also the backbone of HTTP protocol.

As the architecture involves state transfer, this constraint becomes a crucial factor. Clients request complete information of the required resource in the form of query string and other metadata as headers. No part of client's request is retained by the server after it has been serviced.

This feature of REST gives clients the ability to cache or store the server's responses. This potentially reduces the frequency of client-server interaction. Server response should be explicit in specifying cacheability.

Layered The functioning of API is irrespective of whether a client is communicating with the server directly or through certain proxies/load balancers. On the other hand, such intermediaries may improve scalability. Also there may be a separate security layer to enforce security policies.

Code on This constraint has been made optional in REST architecture. It enables the server to extend certain clients' functionality temporarily by transferring some executable code such as Java applets or client-side JavaScript functions.

As we can see, the first constraint stipulates most of the prerequisites of a REST API, namely, a resource available with the server identified by a URI that also underlines the type of action to be performed in terms of HTTP verbs. So, let us build an API that enables CRUD services on a book database (Structure of the Books table as used earlier in *Chapter 8: Using* to other applications. Each book record in this table is a resource, identified by a unique BookID which happens to be the primary key.

Since the client's request URL should convey all requisite information for the operation to be performed by the server, we should first decide the structure of URLs for retrieval, addition, modification, and deletion operations to be performed. These operations correspond to HTTP methods GET (retrieve all/specific record), POST (add new record), PUT (modify or update a certain record), and DELETE (to delete a specified record from database table on the server). We can specify URL string and associated HTTP method as the parameters of Flask routes. Parameters required for SQL query will be passed as variable arguments of URL.

Assuming that our service is running on the following table specifies the URL, HTTP method and the database operation expected to be performed:

performed: performed: performed: performed: performed: performed:
performed: performed: performed: performed: performed: performed:
performed: performed: performed: performed: performed:
performed: performed: performed: performed: performed:

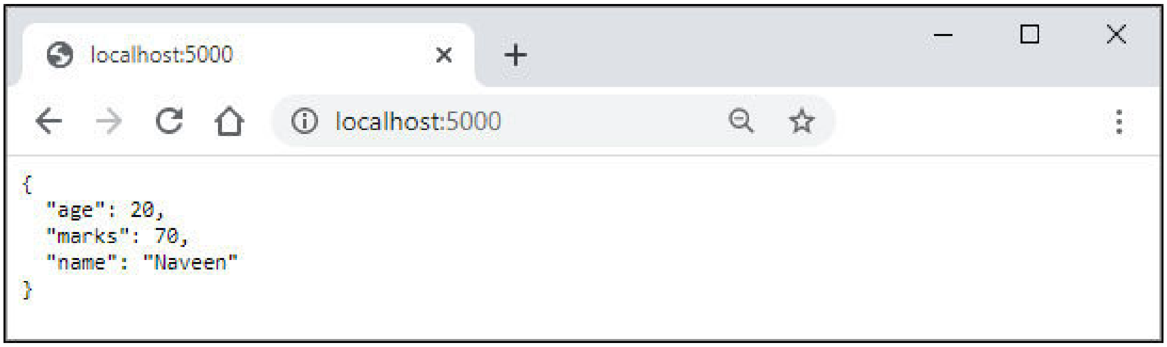
performed: performed: performed: performed: performed: performed:

Table 11.1: *URLs for API*

Before we develop the routes for the above-mentioned URLs and view functions registered with them, a Flask function hitherto not used, needs to be explained. As mentioned earlier in this topic, the server response of a RESTful application must be serialized in JSON (or XML) format, whereas the return type of the view function registered with the @route decorator is generally a plain text, although it is routinely converted in raw HTML to be rendered on the client's browser. This jsonify function converts the response of the view function in JSON output. Following snippet demonstrates the use of jsonify function:

```
from flask import Flask, jsonify
@app.route('/')
defindex():
dict={"name":"Naveen", "age":20, "marks":70}
return jsonify(dict)
```

Browser shows JSON representation as follows:



```
{
  "age": 20,
  "marks": 70,
  "name": "Naveen"
}
```

Figure 11.2: *The jsonify function*

Normally, a Flask route receives incoming data in the form of dictionary-like structure from the client, usually a HTML form. However, since REST architecture requires that the client should send data in JSON format, we also need some mechanism by which this JSON data is parsed in Python objects. Here, the request.json property of request object comes in handy for retrieving and processing incoming requests inside Flask's view function. How do we send a JSON input to Flask application? There are a couple of alternatives for this purpose, namely, cURL utility and Postman app. Their usage is illustrated with examples later in this chapter. First, let us develop a Flask API service application that is capable of processing URLs as decided in above table.

We also need to introduce a little tweak into the books model class. As you can see, an additional method serialize has been defined in the books class. This method returns a dictionary object constructed from the properties of books object so that it can be jsonified:

```
#models.py
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
```

```
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI']='sqlite:///flaskdatabase.db'
app.config['SECRET_KEY']="random string"
db = SQLAlchemy(app)

class books(db.Model):
bookID=db.Column(db.String(6), primary_key=True)
title=db.Column(db.String(20))
Author=db.Column(db.String(20))
price=db.Column(db.Integer)

def __init__(self, id, title, author, price):
self.bookID=id


self.title=title
self.Author=author
self.price=price

def serialize(self):
return {"bookID":self.bookID,
"title":self.title,
"author":self.Author,
"price":self.price
}
```

Using SQLAlchemy API, the books table is populated with data as the following table shows:

shows:
shows:
shows:
shows:
shows:

Table 11.2: *Data in books table*

It is easy to write Flask views for processing URLs with GET request. All they have to do is collect resultset of SELECT query and return its jsonified response.

Route-based REST API

View function registered with route processes GET request to return the JSON form of all books:

```
@app.route('/books/', methods=['GET'])
defallbooks():
res=books.query.all()
lb=[]
for i in res:
s=i.serialize()
lb.append(s)
return jsonify({"books":lb})
```

On the other hand, GET request to fetch one record corresponding to the bookID parameter provided in URL is handled by the following route:

```
#restapp.py
@app.route('/books/', methods=['GET'])
defgetbook(id):
res=books.query.filter_by(bookID=id).first()
book=res.serialize()
```

```
return jsonify({"book":book})
```

Testing these two GET calls is also fairly straightforward. We just need to start Flask server and enter the corresponding URL in browser, just as we have been doing all along. http://localhost:5000/books/ will display a list of books in the JSON format, and will show book details with bookID as A1.

However, for other HTTP method requests, especially the POST and PUT methods, we need to pass data in the JSON format, which is not possible via browser request. As mentioned earlier, it can be done either by Curl or Postman app. First, we shall have look at cURL utility.

What is cURL?

cURL is a free software project consisting of a client-side URL transfer library named libcurl and a command line tool for transferring data using various protocols, including HTTP, FTP, POP3, SMTP, etc. This library has been ported to different programming languages such as C/C++, Java, and Python.

In this chapter, we are concentrating more on the command line tool of cURL (incidentally, it stands for client URL), which internally uses This tool is designed to work without any user interaction. Since it gives direct and easy access to the protocols, it is ideally suited for interacting with REST API.

cURL is available for use on almost all operating system platforms, including Linux and Windows. It is a very comprehensive utility and can be used to handle a number of protocols. Hence, there are many command line options that are specific to a certain protocol. Since we need to interact with an HTTP server in our case, we shall discuss a few command line options that are related to HTTP protocol.

The –i and --include options result in displaying response headers on the command prompt Windows/Linux terminal. The following output shows an example of response headers displayed on the Windows Command Prompt terminal:

cURL uses GET as the default HTTP request method. To specify any other method, we have to specify it as a parameter of –X (or option. Whenever an additional header needs to be inserted in the request, it can be done with the –H (or option. For instance, to specify the content-type of data to be passed along with a POST request, will be inserted, as we shall see in an upcoming example.

As mentioned earlier, we will send data in the request. It is included as a part of –d (or option. Again, its usage will be required while calling a Flask route with POST as well as PUT methods.

Following table shows different command line options for the cURL tool:

tool: tool: tool: tool:
tool: tool: tool:
tool: tool: tool: tool: tool:
tool: tool: tool: tool: tool:
tool: tool: tool: tool: tool: tool:

Table 11.3: *cURL command line options*

Now, we can go ahead and test the GET URLs, routes that have been defined as above. The first one lists out all records in the books table in JSON format:

The following cuRL command displays book record with A1 as

Next, we have to add the view function in our Flask app to process the POST method. Remember that we shall be using the cURL client tool to add JSON data to the request. Hence, we shall use the request.json property, which retrieves a dict object from JSON. Individual field values are parsed and used as parameters for the constructor of books model class and a new record in the books table gets appended:

```
@app.route('/books',methods=['POST'])
def addbook():
```

```
book = books(request.json['bookID'],request.json['title'],
request.json['Author'],
request.json['price'])
db.session.add(book)
db.session.commit()

res=books.query.all()
lb=[]
for i in res:

s=i.serialize()
lb.append(s)
return jsonify({"books":lb})
```

The function returns jsonified list of all records in the end. Let's test this route from cURL terminal with the following command line:

Some command line options of cURL are used here. First, the -H option specifies JSON content type, then specify -X POST to override the default GET method, and –d option to insert data. Also, note the use of back slashes \ as escape character from double quote string, which is a specific requirement on Windows OS and not necessary on Linux. The Command Prompt terminal echoes a list of all records in books the table, including the newly added.

Handling of PUT request is more or less similar in the sense that the cURL command also needs the -X and –d options. We need to insert data for updation in the request. Let's say we want to update the price of book with bookID equal to A2 from Rs. 325 to 375. The cURL command therefore will be as follows:

Flask route and underlying view function obtains bookID from URL as variable, retrieves the corresponding record as an object, reads new price from and reassigns the object's price property:

```
@app.route('/books/',methods=['PUT'])
def updatebook(id):
id=request.json['bookID']
price=request.json['price']
res=books.query.filter_by(bookID=id).first()
```

```
res.price=price
db.session.commit()
book=res.serialize()
return jsonify({'book':book})
```

Windows Command Prompt terminal should echo the updated record, as follows:

The cURL command for delete operation is even straightforward. We needn't use the –H or –d options. Of course, -X DELETE specified the HTTP DELETE method. On Flask side, the view function fetches the record of a given removes it from the session, and commits the change. Following code is to be added in REST application:

```
@app.route('/books/',methods=['DELETE'])
def deletebook(id):
res=books.query.filter_by(bookID=id).first()
db.session.delete(res)
db.session.commit()
res=books.query.all()
lb=[]
for i in res:
s=i.serialize()
lb.append(s)
return jsonify({"books":lb})
```

The cURL command and list of books now in the table will be displayed on terminal, as follows:

Although this approach of building a Flask-based REST API is simple and good enough, there are a couple of more efficient ways to achieve this. One of them is using class-based (also called pluggable) views.

Class-based views

The idea behind pluggable views is to provide flexibility to be able to adopt to other models and templates. Flask has a View class (defined in that provides an alternative way to build view functions. Its subclass should implement the dispatch_request() method. The as_view() classmethod coverts class into a view function to be used by routing system.

The MethodView class in the flask.views module is a parent for any subclass that implements methods to handle the respective HTTP methods. For example, the get() method in this subclass handles GET requests.

Let's build a solution for our REST API based on the MethodView class. Let's call the subclass as Recall that there are two API calls with GET request, with and without bookID as variable part of URL. Hence, inside the get() method of this class, we have to put logic of fetching either all records or a specified record. Here's a skeleton of the BookAPI class:

```
#restpluggable.py
from flask.views import MethodView
from flask import Flask, request, jsonify
from models import app,db, books

class BookAPI(MethodView):

def get(self, id):
if id is None:
#fetch all records
pass
else:
#fetch single record
pass



def post(self):
#add a record
pass

def delete(self, id):
#delete a record
pass

def put(self, id):
#update a record
```

```
Pass
```

Dummy blocks of methods can be filled later (anyway we have the code ready in route-based example in the previous topic). We have to convert the BookAPI class to a view function using the as_view() method and use it as one of the parameters of the add_url_rule() function calls to register our desired URLs:

```
book_view = BookAPI.as_view('book_api')
app.add_url_rule('/books/', defaults={'id': None},
view_func=book_view, methods=['GET',])
app.add_url_rule('/books/', view_func=book_view, methods=
['POST',])
app.add_url_rule('/books/', view_func=book_view,methods=
['GET', 'PUT', 'DELETE'])
```

All we have to do is put code from the earlier example that handles each type of request method in the appropriate method. The complete code is available in the code repository of this book. As earlier, we can use cURL commands to test various calls.

In addition to the provisions in core Flask, there are several Flask extensions using which a REST API can be built. We shall explore functionality of Flask-RESTful extension in the next section.

Flask-RESTful

Flask-RESTful is a lightweight Flask extension. It provides abstraction over pluggable views to quickly build a REST API. It can easily integrate with other Flask libraries. The Twilio API team is credited to be the author of Flask-RESTful extension.

As always, we need to install the package with the pip utility:

```
pip install flask-restful
```

Object of API class in the flask_restful module is the entry point to be obtained with Flask application object as parameter:

```
#restful.py
from flask import Flask
from flask_restful import Api
# creating the flask app
app = Flask (__name__)
# creating an API object
```

```
api = Api(app)
```

Another important component of flask-restful extension package is Resource class. It is an abstract class with methods each corresponding to HTTP methods. It is built on top of class-based (or pluggable) views of Flask. We have to extend Resource class and override the and delete() methods to handle the respective HTTP requests from client.

As we can make out from the earlier two examples, functions/methods handling the PUT and DELETE methods receive bookID parameters. On the other hand, the GET method has two variants: with parameter to fetch a single record and without parameter to retrieve a list of all records. The POST method is handled by a function that doesn't receive the bookID parameter. Taking this into consideration, we shall declare two Resource classes, as follows:

```
class bookone(Resource):
def get(self, id):
#code to retieve a record with given id
def delete(self, id):
#code to delete record of given id
def put(self, id):
```

```
#code to update given record
```

```
class booklist(Resource):
def get(self):
#code to retrive list of all records
def post(self):
#code to add a new record and return list
```

The next step is to add these resources in our API object with its add_resource() method. This method needs two parameters: name of Resource class and URL:

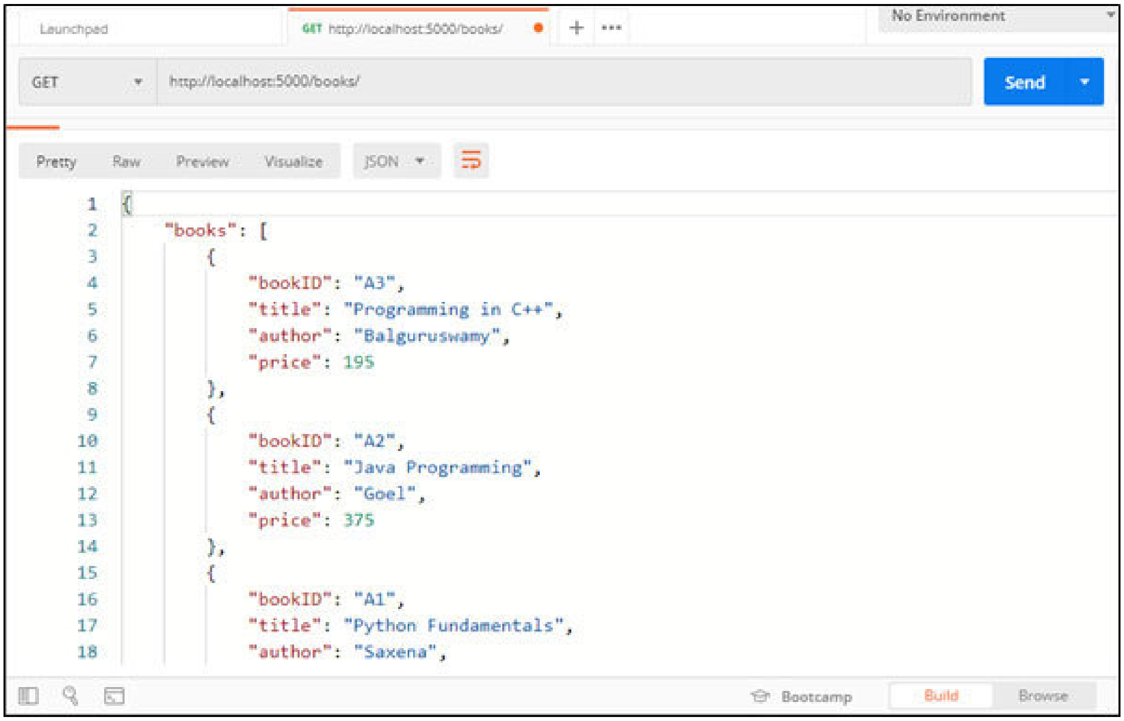
```
api.add_resource(booklist, '/books/')
api.add_resource(bookone, '/books/')
```

Simply copy the code snippets of different database operations from restpluggable.py in the corresponding methods of Resource classes. That's all! We can now start the Flask server and send cURL commands for performing CRUD operations on our database. However, at this point, we take a look Postman, a useful GUI tool for testing API.

Postman app

Postman is a free and interactive collaboration platform for API testing and development. It is available as a desktop application on Windows as well as Linux. It is also available as an extension for Chrome web browser.

You can launch Postman after downloading it and installing it from Choose GET request from the dropdown to the left of the address bar, enter **http://localhost:5000/books/** as the URL, and press the **Send** button. The response from the server will be displayed in the bottom pane of the Postman window:

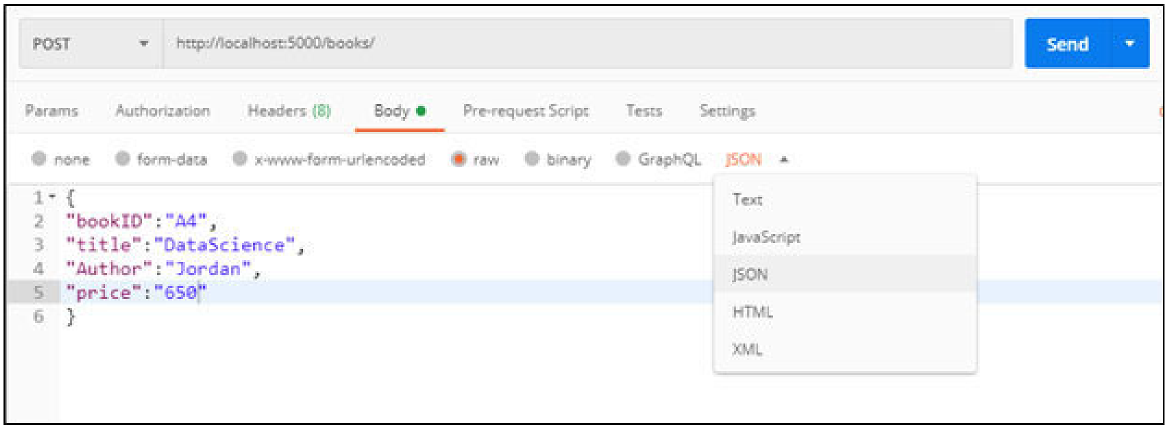


GET http://localhost:5000/books/

```
{
    "books": [
        {
            "bookID": "A3",
            "title": "Programming in C++",
            "author": "Balguruswamy",
            "price": 195
        },
        {
            "bookID": "A2",
            "title": "Java Programming",
            "author": "Goel",
            "price": 375
        },
        {
            "bookID": "A1",
            "title": "Python Fundamentals",
            "author": "Saxena",
```

Figure 11.3: *GET request in Postman*

To test the response for the **POST** request, click on the **Body** tab in the top pane of Postman window, choose JSON data format, and enter the details of new book recorded to be inserted, as shown below. Press **Send** and obtain the server response in the bottom pane:

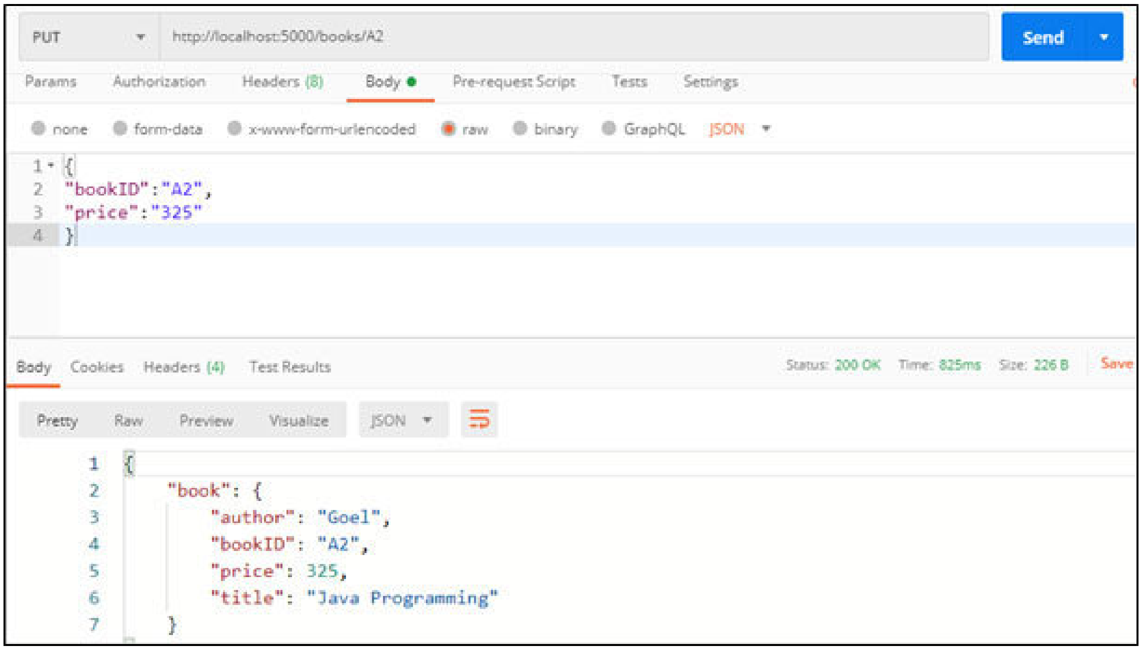


http://localhost:5000/books/

```
{
"bookID":"A4",
"title":"DataScience",
"Author":"Jordan",
"price":"650"
}
```

Figure 11.4: *POST data in Postman*

Testing the PUT request is very similar, as it also needs JSON data to be included in the request. Following screenshot shows body pane and response pane of the PUT request:



http://localhost:5000/books/A2

```
{
"bookID":"A2",
"price":"325"
}
```

Status: 200 OK Time: 825ms Size: 226 B

```
{
    "book": {
        "author": "Goel",
        "bookID": "A2",
        "price": 325,
        "title": "Java Programming"
    }
```

Figure 11.5: *PUT request in Postman*

Testing the DELETE method is fairly straightforward. We have to select DELETE from the method dropdown, enter **http://localhost:5000/A4** in the address bar, and press the **Send** button.

There are a few more alternatives for API testing. One of them is wget command line utility, which is similar to cURL. Another is the requests library for Python, using which API testing can be done programmatically.

Conclusion

Building a REST API with Flask is very easy. Normally, REST API provides an interface for other applications to perform standard database operations through a predefined set of URLs. In this chapter, we learned how to create a REST app using Flask's basic routing mechanism. Flask's pluggable views give a better control over formulating methods for handling HTTP requests. We also used Flask-RESTful extension, which is built on top of pluggable views and makes the task very easy.

We also learned to use two tools for testing APIs: cURL command line client tool and Postman app. Postman is a GUI tool that is convenient to use because one doesn't have to remember clumsy and tedious command line syntax.

The final chapter in this book is about deploying a Flask web application. All along, we have used the built-in development server. Now, it's time to deploy on a publicly available host. A Flask application can be hosted on a dedicated server or on shared hosting services. These options will be explored in the next chapter.

CHAPTER 12

Deploying Flask App

Introduction

It is now time to move our web application from built-in development server of Flask package to production environment so that it is publicly accessible. The various options available for deployment can be broadly classified into two categories:

On cloud platforms such as GCloud, Heroku, etc. offering shared hosting either for free or at subscription

Hosting on dedicated servers running on Apache, IIS, or other server software

In this chapter, we shall explore some of the options from both types.

Structure

Google App Engine

Heroku

PythonAnyWhere

mod_wsgi

uWSGI and NGinx

Flask app as CGI

Other options

Objectives

After going through this chapter, reader should be able to deploy their Flask app on any of the above-mentioned shared hosting options and dedicated server platforms.

Google App Engine

Google's cloud computing services, known as Google Cloud Platform, is a server-less computing environment that can be used for various applications like data analytics, machine learning, etc. In 2008, Google started **Google App Engine** as a **Platform as a Service** feature for developing and hosting web applications in languages like PHP, Python, and many others. While this service is free for applications consuming fewer resources, Google charges a subscription based on additional bandwidth, storage, etc.

Hosting a Flask app on App Engine is very easy. Obviously, we need to have a Flask application in local machine, preferably in a virtual environment (as described in *Chapter 3: Flask* For demonstration purposes, a basic Hello Flask application is assumed to reside in c:\flaskenv directory, which also contains its own Python virtual environment. The first step is to generate a requirement.txt file, which is a list of all Python packages in the current environment. We use the pip3 freeze command for this purpose, as follows:

```
(flaskenv) C:\flaskenv>pip3 freeze > requirements.txt
```

This requirements.txt file should appear as:

```
Click==7.0
Flask==1.1.1
itsdangerous==1.1.0
Jinja2==2.10.3
MarkupSafe==1.1.1
Werkzeug==0.16.0
```

It is also assumed that the app.py file containing the Hello Flask application is also in this folder. We need one more file in this directory - app.yaml - for the application to be deployed on Google App Engine.

YAML is a fancy acronym for YAML Ain't Markup Language. While a detailed explanation of syntax and other features of this language is beyond the scope of this book, it is sufficient to note here that YAML is a human readable data serialization language (somewhat similar to JSON) used to store application settings to specify runtime, version details, URL paths, and other such things.

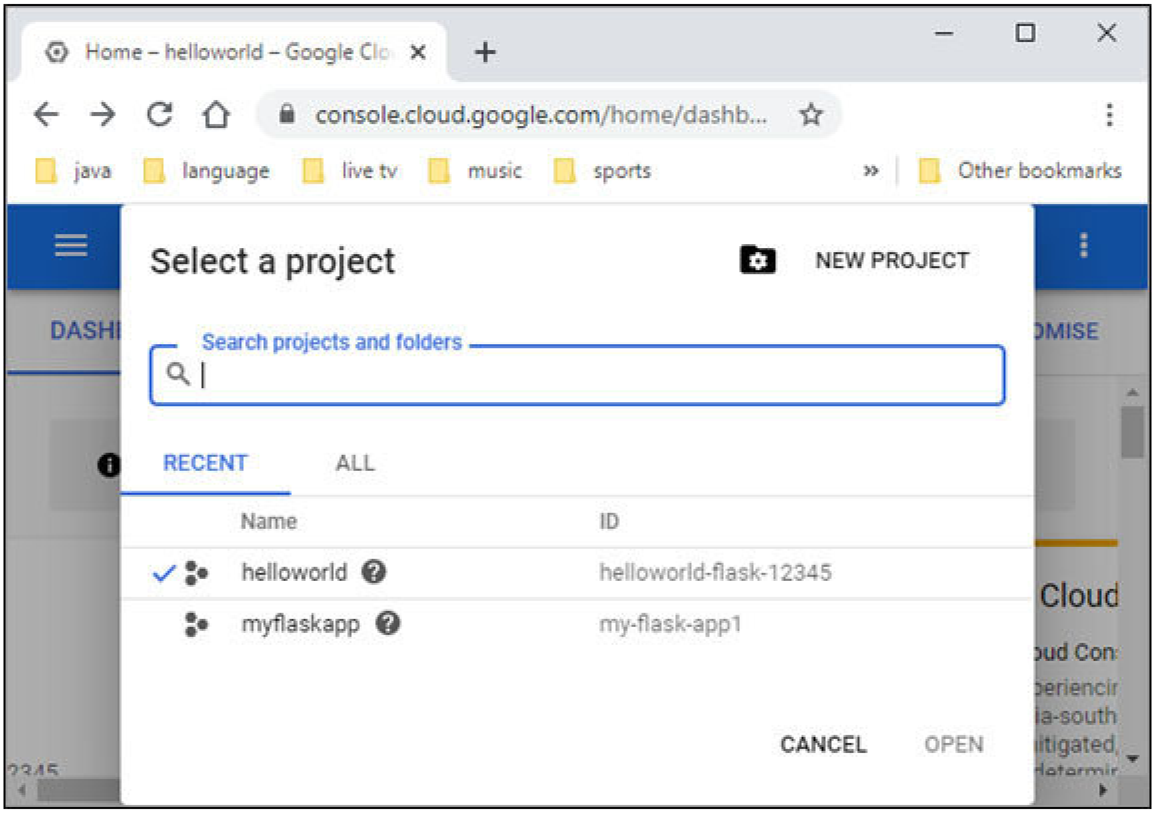
Since ours is a fairly basic app, a single line specifying runtime should be sufficient in the app.yaml file:

```
runtime : python37
```

Next step is to put contents of the application folder (in this case, c:\flaskenv folder with and app.py files) in a GitHub repository. Of course, you need to have an account on You may use GitHub desktop app or GitHub console, or you can manually upload files in a repository. Let's say the files are uploaded to the https://github.com/lathkar/hello_flask repository.

It is time now to deploy our app onto Google App Engine with the following steps:

First, you need to sign up on **https://cloud.google.com** (you may need to furnish your bank account details!) and log in to your account's console to start a new project:



Select a project

NEW PROJECT

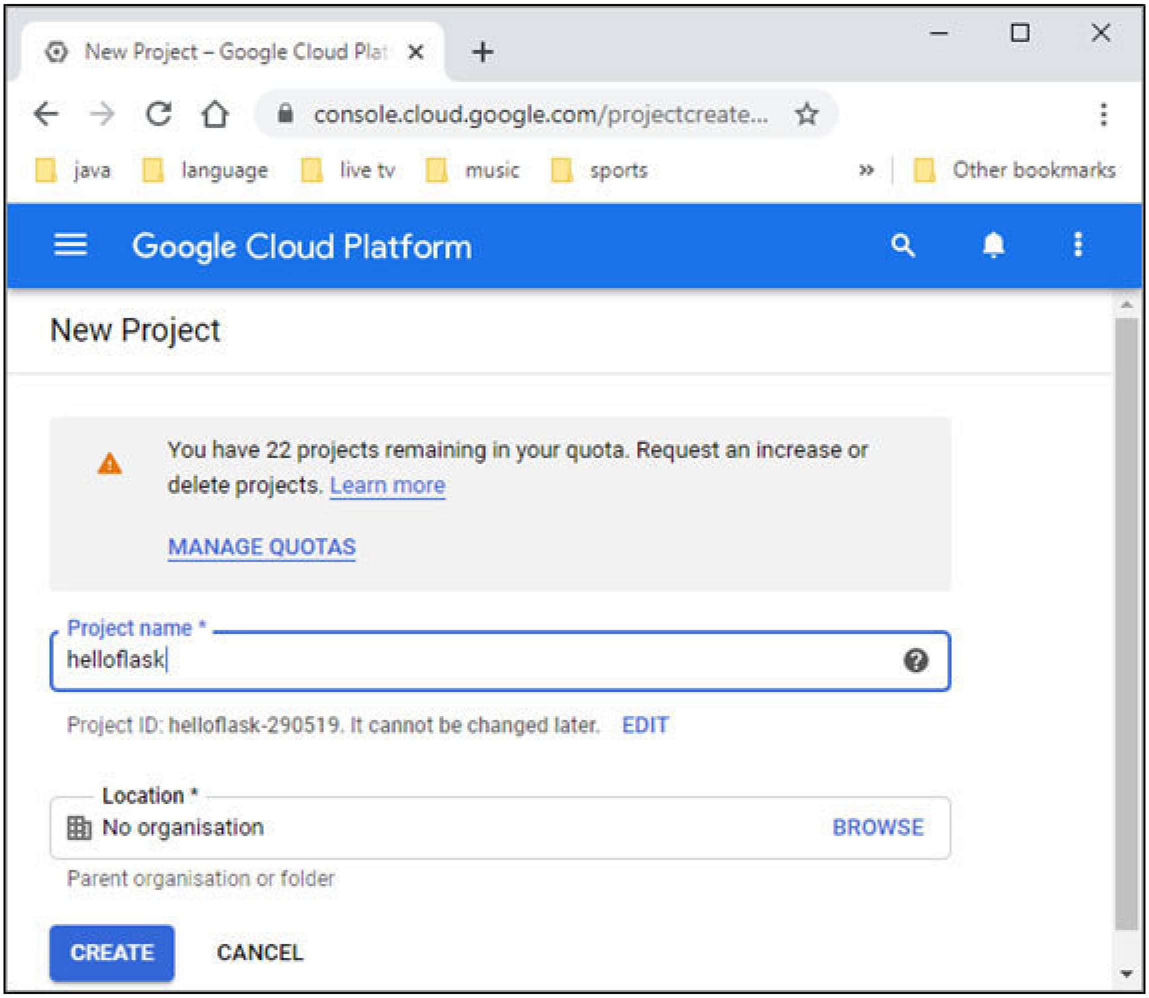
Search projects and folders

RECENT ALL

Name	ID
✓ helloworld	helloworld-flask-12345
myflaskapp	my-flask-app1

Figure 12.1: Google App Engine project

Enter the name for your Google App project and a suitable project ID:



New Project

You have 22 projects remaining in your quota. Request an increase or delete projects. Learn more

Project name *
helloflask

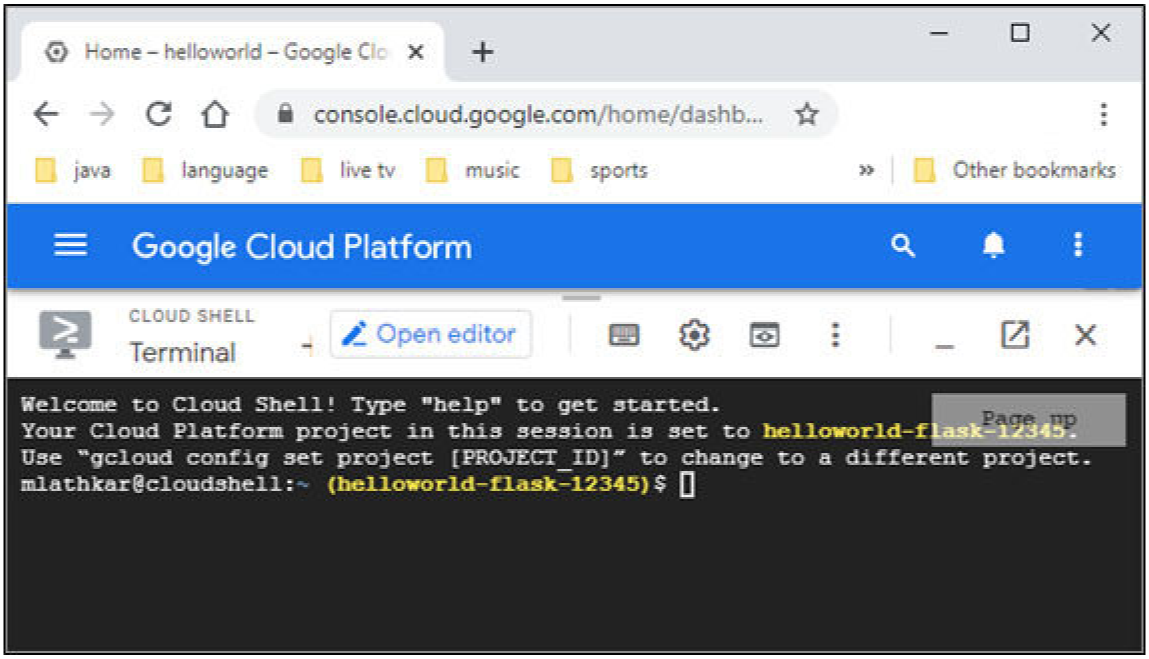
Project ID: helloflask-290519. It cannot be changed later.

Location *
No organisation

Parent organisation or folder

Figure 12.2: *Choose Project ID*

After the project is successfully created, select it for current session and open cloud shell by clicking on the >- button on the menu bar. The cloud shell terminal appears at the bottom of window, although it can also be opened in a new browser window:



```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to helloworld-flask-12345.
Use “gcloud config set project [PROJECT_ID]” to change to a different project.
mlathkar@cloudshell:~ (helloworld-flask-12345)$ 
```

Figure 12.3: *Google Cloud Shell*

We now upload our GitHub repository on Google cloud with the git as follows:

```
$ git clone https://github.com/lathkar/hello_flask.git
```

Files from the GitHub repository will be uploaded to the current project folder.

Next step is to deploy the application with following command and provide project ID when prompted:

```
$ gcloud app deploy
```

After the deployment is complete, the URL of your app can be obtained by issuing following command:

```
$ gcloud app browse
```

Open your browser and use the URL so obtained (which will be something like to launch your Flask web app!

Heroku

Heroku is another popular cloud **Platform as a Service** on which applications can be built in various languages, including Python, Ruby, Node.JS and others. Like GCloud, Heroku has free and subscription-based hosting services. We shall see how a basic Flask application is deployed on Heroku.

We already have a Hello Flask application. In order to deploy on Heroku, we also need to install gunicorn package in the c:\flaskenv virtual environment directory. Short for Green Unicorn, gunicorn is a HTTP server implementing Python's WSGI standard:

```
pip3 install gunicorn
```

We have to rebuild the requirements.txt file so that it now contains Flask and its dependencies, along with the gunicorn package:

```
click==7.1.2
Flask==1.1.2
```

```
gunicorn==20.0.4
itsdangerous==1.1.0
Jinja2==2.11.2
MarkupSafe==1.1.1
Werkzeug==1.0.1
```

Heroku also needs a It is essentially a specification of processes to be run before the application is deployed. For our application, the following line in Procfile is sufficient:

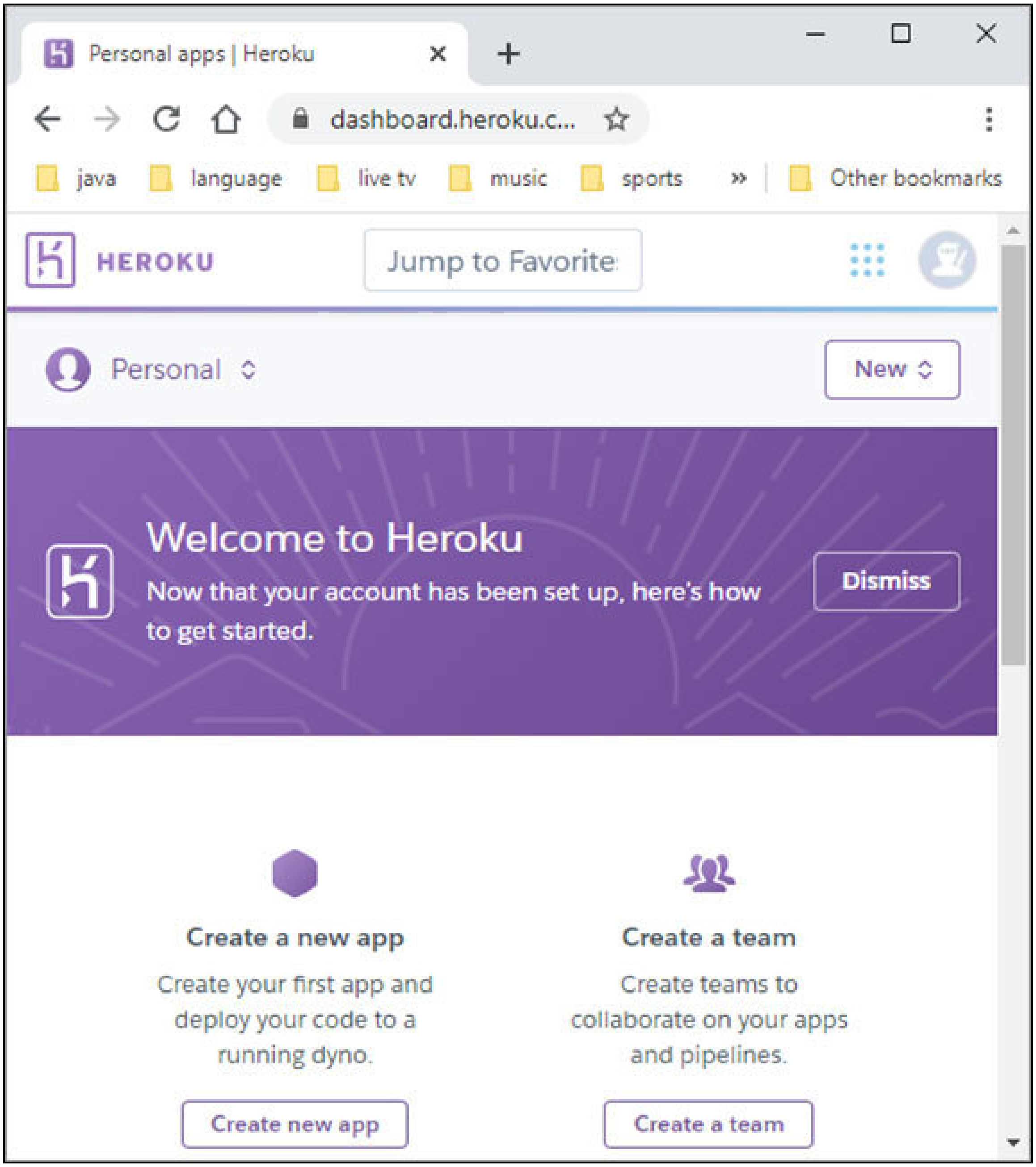
web: gunicornapp:app

The web command is an instruction to Heroku to start a gunicorn web server with a Flask app object in Finally, create runtime.txt specifying the Python version to be used:

```
python-3.7.4
```

All these files and are now put on a GitHub repository, as we did for Google App Engine deployment. Let the name of the repository be

Next, you'll need to sign up and then log in to Heroku account's dashboard and create a new app:



Welcome to Heroku

Now that your account has been set up, here's how to get started.

Create a new app

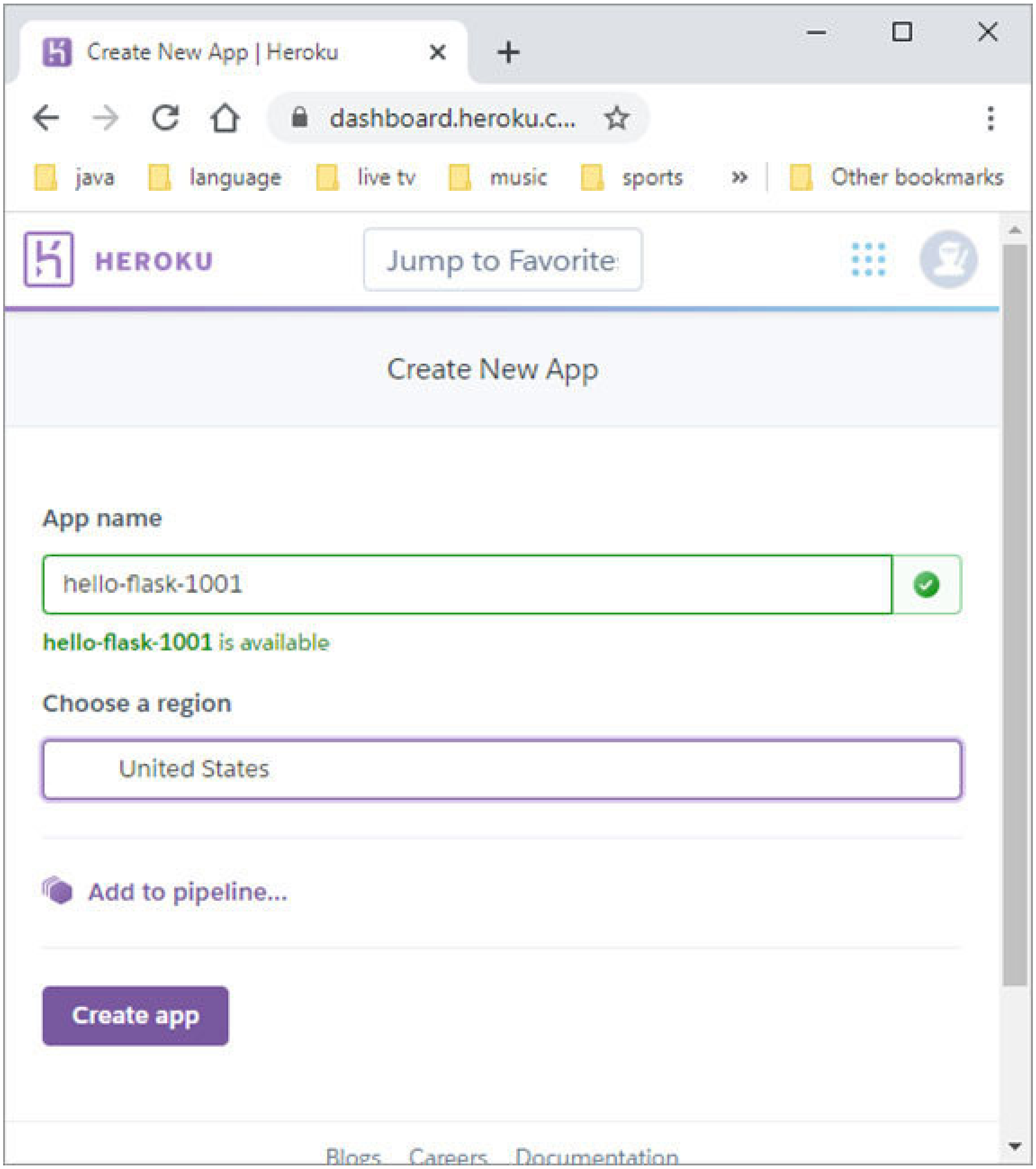
Create your first app and deploy your code to a running dyno.

Create a team

Create teams to collaborate on your apps and pipelines.

Figure 12.4: *Heroku dashboard*

Choose a suitable name for your app and create the application workspace:



Create New App

App name

hello-flask-1001

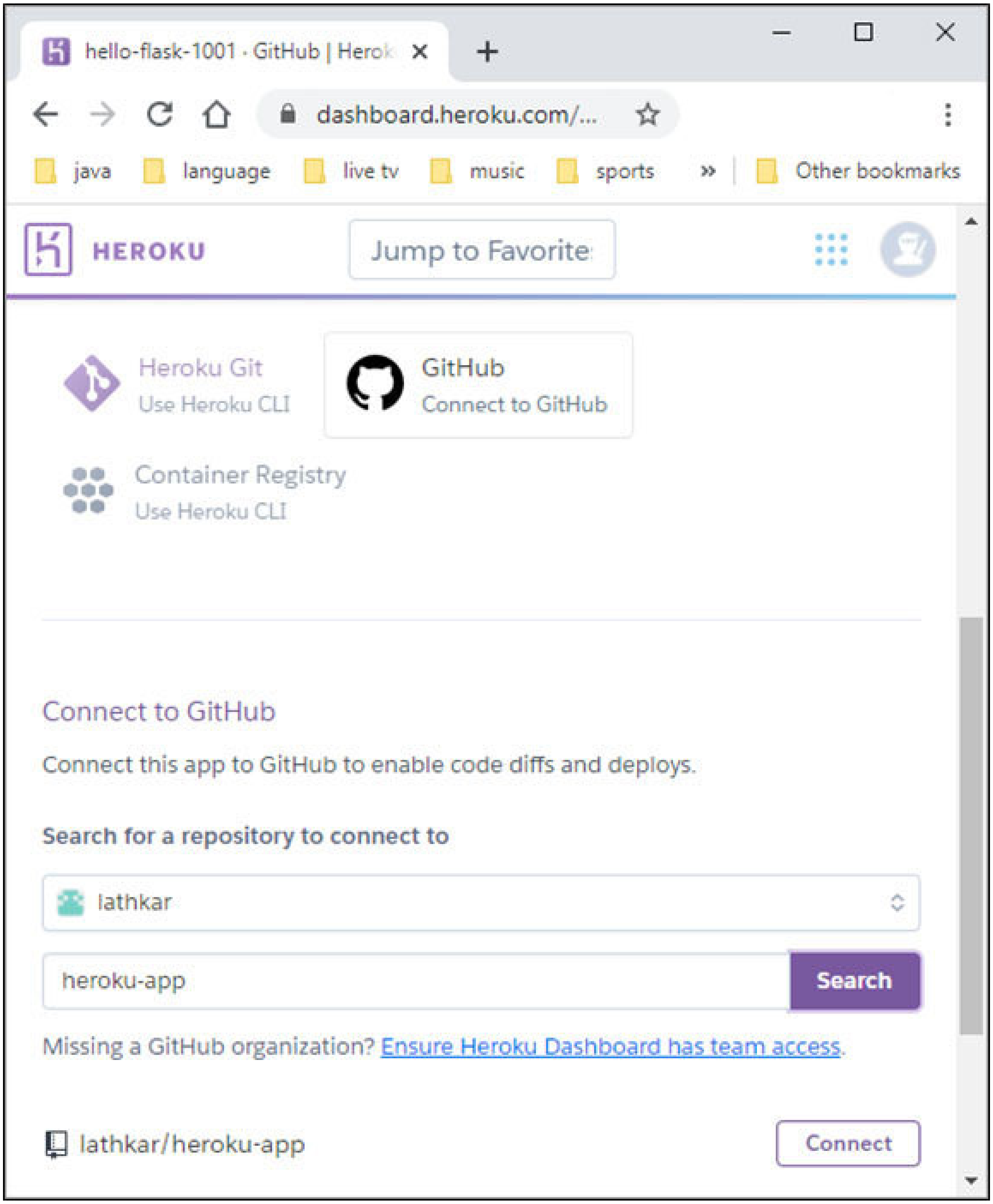
hello-flask-1001 is available

Choose a region

United States

Figure 12.5: *Create Heroku app*

Now, it's time to upload your project files on Heroku. There are two ways to do this: use Heroku's Git command line interface, for which you will require knowledge of git commands, or avail a user-friendly GUI to connect your GitHub repository to the Heroku server:



Connect to GitHub

Connect this app to GitHub to enable code diffs and deploys.

Search for a repository to connect to

lathkar

heroku-app

Missing a GitHub organization? Ensure Heroku Dashboard has team access.

lathkar/heroku-app

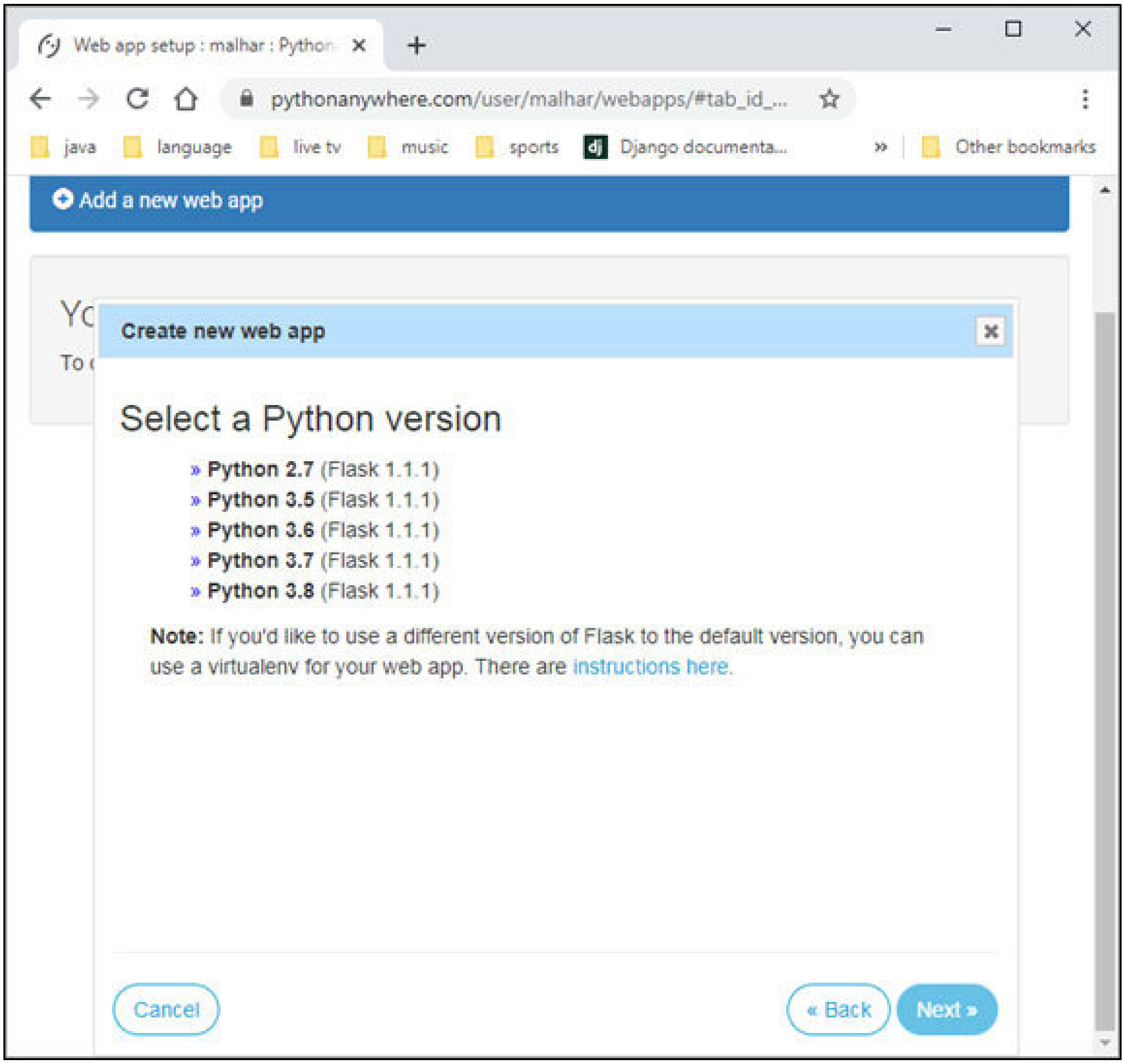
Figure 12.6: *Connect GitHub repository*

You are almost done. After the connection is established, just click on the **Deploy** button. If everything goes well, your app will be successfully deployed and can be browsed live at

PythonAnywhere

Another example of cloud-based hosting service for Flask application is available at where deployment is as simple as (if not simpler than) Google App Engine or Heroku platforms. Unlike the two, PythonAnywhere supports hosting Python web applications only (it doesn't support other languages). Incidentally, PythonAnywhere also offers online Python REPL shell to try out Python code. It also provides bash console for performing OS level tasks such as installing new modules. In addition to Flask, PythonAnywhere supports hosting web applications based on Django, Web2Py, and CherryPy frameworks.

Like GCloud and Heroku, PythonAnywhere also has free and subscription based plans. To begin deploying our Hello Flask application, head toward the dashboard of your free account on PythonAnywhere, click on the **Web** tab on its menu bar, and create a new web app. Select **Flask** as the Python framework of your choice and a suitable Python version:



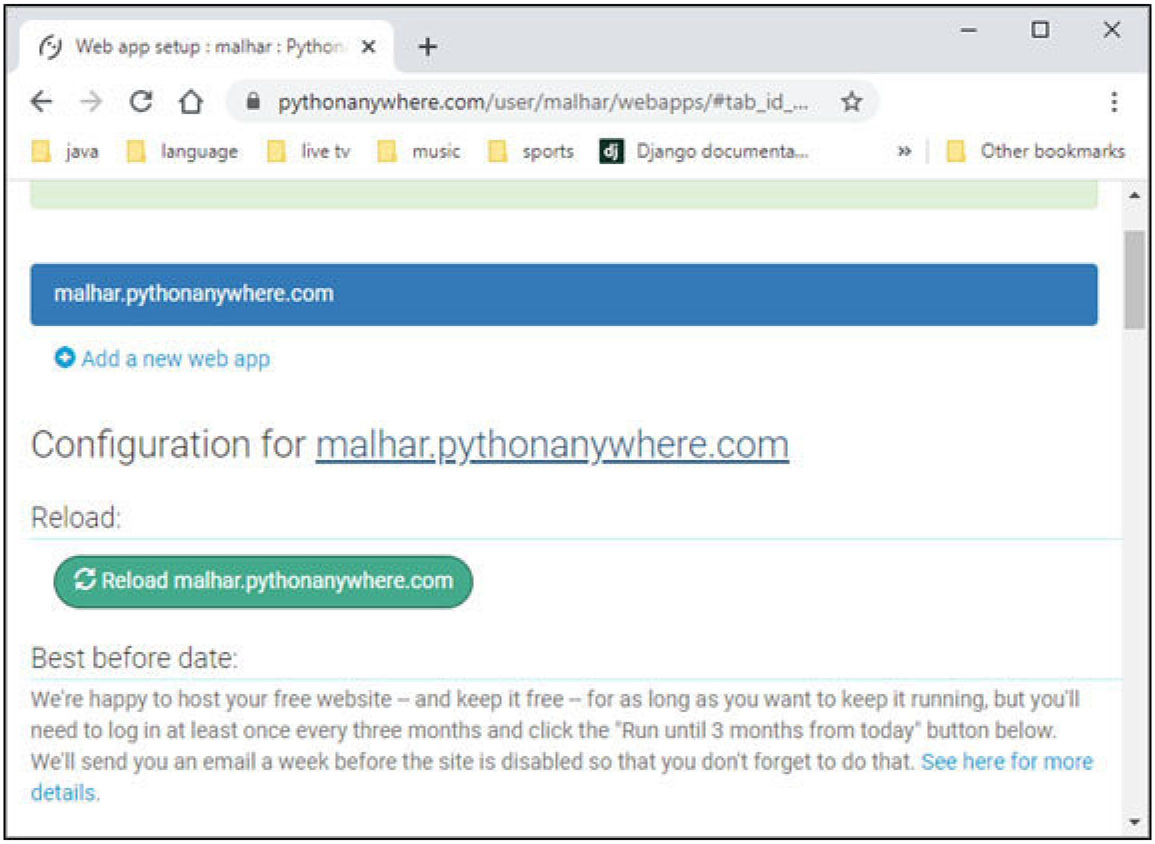
Select a Python version

- » **Python 2.7** (Flask 1.1.1)
- » **Python 3.5** (Flask 1.1.1)
- » **Python 3.6** (Flask 1.1.1)
- » **Python 3.7** (Flask 1.1.1)
- » **Python 3.8** (Flask 1.1.1)

Note: If you'd like to use a different version of Flask to the default version, you can use a virtualenv for your web app. There are instructions here.

Figure 12.7: *Create new web app on PythonAnywhere*

Provide /home/malhar/flaskapp/app.py as the path and click on There you go. Your app is deployed on the URL, as shown below:


Web app setup : malhar : Python
pythonanywhere.com/user/malhar/webapps/#tab_id_...
java
language
live tv
music
sports
Django documenta...
Other bookmarks
malhar.pythonanywhere.com
Add a new web app
Reload malhar.pythonanywhere.com

Configuration for malhar.pythonanywhere.com

Reload:

Best before date:

We're happy to host your free website -- and keep it free -- for as long as you want to keep it running, but you'll need to log in at least once every three months and click the "Run until 3 months from today" button below. We'll send you an email a week before the site is disabled so that you don't forget to do that. See here for more details.

Figure 12.8: *PythonAnywhere app URL*

Code for the Hello Flask application will be auto-generated, which you may edit if you wish. You can also add source files if required. All standard modules are preloaded. Using bash terminal, additional modules may be installed as needed.

mod_wsgi

We shall now take a look at some options for standalone deployment of the Flask app. Apache server, distributed by Apache Software Foundation, an open-source and most popular HTTP server software, powers most web servers across the Web. The mod_wsgi is an apache module that provides a WSGI interface for deploying Python-based web applications on Apache.

We have already seen how a Python script can be used as CGI on XAMPP, a Windows-based bundle of Apache, MySQL, PHP, and Perl. Now, we shall find out how to install the mod_wsgi module on XAMPP so that a Flask app can be run on it.

To start with, the mod_wsgi package needs to be installed in a Python setup with the pip utility:

```
C:\python37>pip3 install mod_wsgi
```

Note that you may need to install Microsoft's VC++ redistributable and VC++ build tools prior to above command. Once mod_wsgi is successfully installed, run the following command:

These are the mod_wsgi module settings to be incorporated in Apache's configuration file – So, open this file and add the above text in it.

Next, we need to create a virtual host configuration for our application. Apache stores virtual host information in the httpd-vhosts.conf file, which is found in the C:\XAMPP\Apache\conf\extra\ folder. Go ahead and open it to add following lines in it:

```
*>
ServerName localhost:5000
WSGIScriptAlias / C:\flaskapp\flaskapp.wsgi
C:\flaskapp>
Order deny,allow
Allow from all
Require all granted
```

Here, it is assumed that Flask application is present in the c:\flaskapp folder. The tag lets the Apache server access WSGIScript inside this folder. This script is executed when the browser sends a request for the **http://localhost:5000** address.

This virtual host configuration needs to be incorporated in Apache's httpd.conf file by adding following lines in it:

```
# Virtual hosts
Include conf/extra/httpd-vhosts.conf
```

The flaskapp.wsgi is the main application that will be run by Apache. The following Python script asks mod_wsgi to use Flask application object in the flaskapp.py module to be used as WSGI application:

```
import sys
sys.path.insert(0, 'c:/flaskapp')
from flaskapp import app as application
```

Lastly, we need to make sure that our Hello Flask application is present in the c:\flaskapp folder in the flaskapp.py file, as follows:

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

@app.route("/")
defhello():
return "Hello Flask with mod_wsgi!"

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

If everything goes well, we should be able to run this Flask application on Apache server! For apache on Linux, the procedure is almost the same as above, barring the difference in file path syntax. Also, Microsoft's VC++ build tools are not required as the C++ compiler is installed along with C++ libraries.

uWSGI

Another option for deployment of the Flask app is the use of uWSGI on Nginx (pronounced as **engine** which is a HTTP server. It provides accelerated support for uWSGI and FastCGI and is available for use on almost all OS platforms.

uWSGI is a software application that implements the WSGI standard. Nginx offers direct support to uWSGI's native uwsgi protocol, although Apache can also support the same through the mod_proxy_uwsgi module.

To get started, we need to first install it using the pip installer:

```
pip3 install uwsgi
```

Use the following syntax to run a Flask application with uWSGI on local server:

```
uwsgi --http 127.0.0.1:8080 --wsgi-file run.py --callable app
```

The application is now being served on localhost's 8080 port. Instead of passing command-line parameters, we can create an .ini file and use it in command line. Along with wsgi-file and callable parameters, processes and threads parameters are added to make the application asynchronous:

```
#http.ini

[uwsgi]
http = :8080
wsgi-file = run.py
callable = app
processes = 4
threads = 8
```

Use this file to start the application, as follows:

```
uwsgi http.ini
```

To put uWSGI behind Nginx web server, use the following nginx.config structure:

```
server {
```

```
listen 80;

location / {
include uwsgi_params;
uwsgi_passunix:/path/to/app.sock;
}
}
```

The http.ini file should have the following typical structure:

```
[uwsgi]
socket = app.sock
chmod-socket = 664

uid = www-data
gid = www-data

wsgi-file = run.py
callable = app

master = true
processes = 4
threads = 8
```

Flask app as CGI

This is the easiest way to deploy the Flask app. However, as discussed earlier (in *Chapter 1: Python for* CGI performance is invariably poor, so it is recommended to employ above-mentioned methods before adapting this approach as a last resort.

Simply put the Flask application code along with the following myapp.CGI file in the cgi-bin directory of Apache installation:

```
#!c:\python37\python.exe
from wsgiref.handlers import CGIHandler
from myapp import app
print("Content-Type: text/html")
print()
CGIHandler().run(app)
```

The application will be served at

Other options

In addition to the different deployment alternatives discussed in this chapter so far, few other options are also available. As far as shared hosting is concerned, deployment on AWS cloud with Elastic Beanstock is a popular technique. You can also deploy on Flask on Microsoft's Azure platform.

For dedicated deployment, using FastCGI – a more efficient version of CGI protocol – can also be performed. FastCGI is widely supported on a variety of web servers like Apache, Nginx, IIS, etc. and for almost all OS platforms including Linux, macOS, and Windows. Interested readers can explore these alternatives as well.

Conclusion

Deployment of a software application in production environment is crucial and, at times, a little tedious. In this chapter, we explored the different available options for deploying a Flask web application. If you prefer deployment on cloud platforms, this chapter illustrated, with examples, the deployment process on Google App Engine, Heroku, and PythonAnywhere.

While cloud deployment offers the advantage of scalability, hosting on dedicated servers provides better performance and security. Flask application can be deployed on variety of web servers, including Apache, Nginx, and IIS. In this chapter, we covered a detailed process of deployment on Apache with the mod_wsgi module and using uWSGI along with Nginx.

This incidentally is the final chapter of this book. Starting with the basics of HTTP protocol, we have come a long way up to a production-level deployment of a Flask-based web application. Almost all nitty-gritties of the Flask framework have been explained with the help of examples, code snippets, and figures. Hopefully, readers will appreciate this effort and will be able to build robust web apps.

Appendix: Python QuickStart

Python is a high-level, object-oriented, and interpreted language developed by *Guido van Rossum* . Official distribution of Python (current version 3.9) can be downloaded from https://python.org

Get Started

After starting the Python interpreter, Python shell appears with the >>> symbol as Python prompt.

```
>>> print ("Hello World")
Hello World
```

Data Types

```
>>> #this is an integer
>>> 100
>>> print (type(100))
'int'>
>>> #this is a float
>>> 5.65
>>> print (type(5.65))
'float'>
>>> #this is bool object (True or False)
>>> True
>>> print (type(True))
'bool'>
>>> #string is an indexed sequence of characters put
inside single, double or triple quote symbols
>>> #string using single quotes
>>> 'Hello. How are you?'
>>> #string using double quotes
>>> "Hello. How are you?"
>>> #string using triple quotes
>>> '''Hello. How are you?'''
>>> print (type('Hello. How are you?'))
'str'>
```

```
>>> #List is an inexed collection of items, not necessarily
of same type, put inside square brackets
>>> ['pen', 15, 25.50, True]
>>> print (type(['pen', 15, 25.50, True]))
'list'>


>>> #Tuple is a indexed of items, not necessarily of same
type, put inside parentheses
>>> ('Python', 3.72, 'Windows',10, False)
>>> print (type(('Python', 3.72, 'Windows',10, False)))
'tuple'>
>>> #Dictionary is an unordered collection of key-value
pairs
>>> {1:'one', 2:'two', 3:'three'}
>>> print (type({1:'one', 2:'two', 3:'three'}))
'dict'>
>>> #Variable is a label assigned to Python object. It
carries same id() - unique identifier - of the object
>>> age=20
>>> id(20), id(age)
(140712680190816, 140712680190816)
>>> #Python is dynamically typed. Type of variable
changes according to object srored in it
>>> var="Hello World"
>>> print (type(var))
'str'>
>>> var=100
>>> print (type(var))
```

'int'>

Indexing and Slicing

Any item at a certain position in a sequence, i.e. string, list, or tuple, can be retrieved. An item at index can be updated in the case of a list, but not in string or tuple as they are immutable:

```
>>> str1="Hello. How are you?"
>>> list1=['pen', 15, 25.50, True]
>>> tup1=('Python', 3.72, 'Windows',10, False)
>>> print (str1[7], list1[2], tup1[3])
>>> list1[0]='pencil'
>>> print (list1)
>>> tup1[1]=3.9
H 25.5 10
['pencil', 15, 25.5, True]
-------------------------------------------------------------------
TypeError Traceback (most recent call last)
5 list1[0]='pencil'
6 print (list1)
----> 7 tup1[1]=3.9
TypeError: 'tuple' object does not support item assignment
```

Slice is a part of a sequence type object. In Python, : is a slice operator with two operands. x:y slices items starting from index x to index

```
>>> str1="Hello. How are you?"
>>> list1=['pen', 15, 25.50, True]
>>> tup1=('Python', 3.72, 'Windows',10, False)
>>> str2=str1[0:5]
>>> list2=list1[1:3]
>>> tup2=tup1[2:3]
>>> print (str2, list2, tup2)
```

```
Hello [15, 25.5] ('Windows',)
```

Dictionary collection is not indexed. If a key exists, its value can be updated. If it's a new key, key-value pair is added:

```
>>> dict1={'Mumbai':'Maharashtra', 'Hyderabad':'Andhra
Pradesh', 'Patna':'Bihar'}
>>> dict1['Hyderabad']='Telangana'
>>> dict1['Jaipur']='Rajasthan'
>>> print (dict1)
{'Mumbai': 'Maharashtra', 'Hyderabad': 'Telangana', 'Patna':
'Bihar', 'Jaipur': 'Rajasthan'}
```

Conditionals

Python keywords - else and elif - are used to define blocks of statements to be executed conditionally. Block is a sequence of Python statements with equal indent length after the symbol. Block of statements after the if < expr >: statement is executed if its boolean expression clause evaluates to true. If it's false, another block after the else : statement is executed:

```
>>> salary=int(input("enter salary.."))
>>> tax=0
>>> if salary>=50000:
#tax @10%
tax=salary*10/100
else:
#tax@5%
tax=salary*5/100
>>> net_sal=salary-tax
>>> print ("Salary=",salary, "tax=", tax, "net payable=",
net_sal)
enter salary..30000
Salary= 30000 tax= 1500.0 net payable= 28500.0
```

The elif statement is used to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to If all previous conditions in elif fail, the last else block is executed:

```
>>> salary=int(input("enter salary.."))
>>> tax=0
>>> if salary>=50000:
#tax @10%
tax=salary*10/100
elif salary>25000:

#tax @5%
tax=salary*5/100
elif salary>10000:
#tax @2%
tax=salary*2/100
else:
#no tax
print ("No tax applicable")
>>> net_sal=salary-tax
>>> print ("Salary=",salary, "tax=", tax, "net payable=",
net_sal)
enter salary..9000
No tax applicable
Salary= 9000 tax= 0 net payable= 9000
```

while loop

A block of statements after while < expr >: is repetitively executed till < expr > is true. The repetition stops as soon as < expr > becomes false:

```
>>> count=0
>>> while count<5:
#count repetitions
count=count+1
print ("This is count number",count)
print ("end of repetitions")
This is count number 1
This is count number 2
This is count number 3
This is count number 4
This is count number 5
end of repetitions
```

for loop

Python's for loop allows traversal and processing of each element in a collection of objects, such as string, list, tuple, or dictionary:

```
>>> for element in collection:
statement1
statement2
...
>>> #for loop with string
>>> for char in "Python":
print (char)
P
y
t
h
o
n
>>> #for loop with list
>>> numbers=[1,2,3,4,5]
>>> for num in numbers:
print (num, "* 2 = ", num*2)
1 * 2 = 2
```

```
2 * 2 = 4
3 * 2 = 6
4 * 2 = 8
5 * 2 = 10
>>> #for loop with dictionary
>>> marklist={1:56, 2:87, 3:44, 4:74, 5:51}
>>> print ("rollno : marks")

>>> for rollno,marks in marklist.items():
print (rollno,":", marks)
rollno : marks
1 : 56
2 : 87
3 : 44
4 : 74
5 : 51
```

Functions

Python's standard library consists of many built-in functions id some of them used above). It also has more than 200 built-in modules, each with a number of functions that can be imported into the current namespace. Following statement shows how to import pow() function defined in the math module:

```
>>> from math import pow
>>> print ("5 raised to power 2:", pow(5,2))
5 raised to power 2: 25.0
```

To define a customized function, Python's def keyword is used:

```
>>> def SayHello():
print ("Hello World")
>>> # call function
>>> SayHello()
Hello World
```

We can define a function to receive arguments. It should be called by passing matching number of arguments:

```
>>> def SayHello(name):
print ("Hello ", name)
>>> #call function with arguments
>>> nm="Ravindra"
>>> SayHello(nm)
Hello Ravindra
```

Class and object

In Python, class is a user defined data type. The class keyword is used to define its structure. It can contain attributes and methods. Instance attributes are initialized by the constructor method Each instance method must have a mandatory argument self as reference to the calling object. In the following class, show() is an instance method:

```
>>> class myclass:
def __init__(self):
self.name="Ravindra"
self.age=25
def show(self):
print ("name:",self.name," age:",self.age)
>>> #object of myclass
>>> obj=myclass()
>>> obj.show()
name: Ravindra age: 25
```

Class constructor can be parameterized as well:

```
>>> class myclass:
def __init__(self, name=None, age=None):
self.name=name
self.age=age
def show(self):
print ("name:",self.name," age:",self.age)
>>> #object of myclass
>>> obj=myclass("Jyoti", 22)
>>> obj.show()
name: Jyoti age: 22
```

Inheritance

Attributes and methods defined in a parent class can be made available to a child class. In Python, this relationship is established by putting name of parent class inside parenthesis in front of the child class name definition.

Here's the definition of a rectangle class with the width and height attributes and the area() method:

```
>>> class rectangle:
def __init__(self, width, height):
self.width=width
self.height=height
def area(self):
area=self.width*self.height
return area
```

This class is used as parent by square class (all sides are equal). The area() method is inherited:

```
>>> class square(rectangle):
def __init__(self, side):
```

```
super().__init__(side, side)
>>> s1 = square(5)
>>> print ("area of square with side=5 : ", s1.area())
area of square with side=5 : 25
```

Overriding

Child class may modify the inherited method if required. Rewritten square class with the overridden area() method is as follows:

```
>>> class square(rectangle):
def __init__(self, side):
super().__init__(side, side)
def area(self):
area=self.width**2
return area
>>> s1 = square(5)
>>> print ("area of square with side=5 : ", s1.area())
area of square with side=5 : 25
```

Script mode

A Python script is a sequence of Python statements stored in a text file with the extension and is run from command line of the operating system. Any Python aware text editor (such as IDLE, VS Code, Sublime Text, and so on) may be used to create a Python script.

Save the following text as

```
name=input ("enter your name: ")
print ("Hello ", name)
```

Run test.py from command line, as follows:

```
C:\python37>python test.py
enter your name: Anil
Hello Anil
```

This QuickStart is meant to serve as a recap of the Python fundamentals. Many books and online resources are available for learning Python in detail. BPB Publications has published several books on Python, including one from this author – titled *Data Persistence: With SQL and NOSQL*

Index