

Designing Microservices Using Django

*Structuring, Deploying and Managing
the Microservices Architecture with Django*

by
Shayank Jain



FIRST EDITION 2020

Copyright © BPB Publications, India

ISBN: 978-93-89328-790

All Rights Reserved. No part of this publication may be reproduced or distributed in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's & publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but 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.

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

Dedicated to

Kratika Jain

My Wife, she is a gift of god.

About the Author

Shayank Jain is a software developer and data analyst. He is strongly passionate about coding and architectural design. He has more than 6 years of professional experience in developing scalable software solutions for various organizations. He has been programming since the age of 16 and has developed softwares for mobile, web, hardware gaming and standalone applications. After getting his hands dirty with programming, he found many new ways to debug and deploy the code successfully, with minimal time constraints. After reading and implementation, he found out that many critical concepts can be implemented easily in programming with correct and focused thinking. His research interests include information security, cryptography, analysis, design, and implementation of algorithms. He has extensively worked with python and implemented new ideas on various projects in his free time. He is also active in the computer science and education community. Through this book, he wants to share these methodologies and tricks with the beginners.

Apart from work, Shayank spends his spare time helping, coaching, and mentoring young people in taking up careers in technology.

Acknowledgements

There are a few people I want to thank for the continued and ongoing support that they have given me during the course of writing this book. First and foremost, I would like to thank my wife and my family for putting up with me while I was spending my weekends and evenings on writing - I could not have completed this book without their support.

I would like to especially thank my friend Krishna Vishwakarma, who guided and motivated me throughout the writing of this book. This book wouldn't have materialized if I did not have his support.

Finally, I would like to thank BPB Publications, for giving me the opportunity to write my first book for them.

Preface

Microservice architectures solve one of the most important questions of software engineering evolution. Companies are wondering how to fight the obsolescence of their products. During the first few months, a start-up must convince the investors to believe in them. Time is the key to everything. Investors are interested in maximizing the ROI of a project. The speed of development that the startup needs, and is required by investors to maximize their ROI, tends to completely ignore the evolution of the software. Software developed with microservices divides the software into many services - small and cohesive, to be used and "assembled" together. It is very common today, and very convenient, to develop mobile and desktop applications through web technologies, using server services. Internet is full of "Get started" tutorials that help the developer to make the first steps with Python, the first steps to creating an app with Django, steps to secure API with JWT tokens, first steps with the Asynchronous tasks, with RabbitMQ, with databases, with AWS, with the Cloud. Django is a full-stack development framework written in Python. It already includes everything that is necessary for the development of a web application, from the user views to the information storage - model, persistence, relationships, controllers, forms, validations, rest API, and a very useful back office. Building production-ready microservices will also be covered in this book. We will learn how to create restful APIs. We will learn about Redis and Celery, as well as how to use the cache framework.

You will also learn how to secure these services and deploy these microservices using Django. Finally, we will learn how to scale our services as well.

The primary goal of this book is to provide the information and skills that are necessary to deploy the microservice architecture with Django. This book contains real-life examples that will show you how to install, configure, and manage the Django application, as well as how to integrate it into other third-party solutions for deployment. Over the 15 chapters in this book, you will learn the following:

Chapter 1: It covers the introduction to Python and explains everything that a programmer needs to get started with Python. It also covers all the basic concepts of python with syntax, sample code, and examples.

Chapter 2: It discusses the major pillars of OCPs, and how it is used in python - like object, class, inheritance, abstraction, encapsulation, and polymorphism.

Chapter 3: It contains the introduction of Django, its architecture, its working flow, the basic functionality of Django.

Chapter 4: It covers how we can create an API with Django and what are the steps for deployment.

Chapter 5: It covers database modeling with Django ORM. We use Postgresql, MySQL, SQLite as the database, and Django Shell for connecting the database.

Chapter 6: This will cover all the Django web deployment processes and the basic idea of Uwsgi, Gunicorn, Supervisor, Nginx, and Apache. We use Django logger for production debugging.

Chapter 7: It covers the Django API deployment on the production server, with different web servers and services like Apache, Nginx, Gunicorn, supervisor, and UWSGI.

Chapter 8: It will give you a basic introduction to microservice and the comparison of the old and new methodology of architecture. It will also cover the benefits and losses of microservices.

Chapter 9: It will give you a basic idea of microservices designing patterns, parameters of microservice development, service designing, and independent deploy-ability.

Chapter 10: An application that is monolithic, or a microservice, needs to authorize the users to use it, so you need a service that allows you to authenticate a user, recognize it and verify its permissions. Django already has its own way of determining the permissions and users. We will see how this can be useful in microservice architecture. We are deploying microservices with Django.

Chapter 11: We are deploying microservices with Django. This chapter will cover how we can connect multiple databases into a single project, with flexible working. It will cover how the data flows between multiple services at the same time.

Chapter 12: It covers the introduction of Json Web Token and how we can make our API more secure with the JWT token. It has examples and sample code.

Chapter 13: It covers the introduction of the asynchronous task and the best fit technology for Django like RabbitMQ, Redis, and Celery.

Chapter 14: AWS offers a complete and very interesting stack for microservice deployment. AWS Lambda allows the execution of the Django code, (in almost all the main languages) on request. AWS offers a gateway API, an Authentication API, database, storage, NoSQL, and asynchronous tasks, with business models related to their use and not related to the infrastructure. So, it covers all of them.

Chapter 15: It covers the common questions and answers regarding microservices.

Downloading the code bundle and colored images:

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

<https://rebrand.ly/06cc6>

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, 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

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Table of Contents

1. Basics of Python.....	1
Structure.....	1
Objective.....	2
Introduction of Python programming language	2
Definition of variables, expressions, and statements	4
Statements.....	6
Operations on strings.....	6
Functions.....	7
Python type conversion functions.....	7
User-defined functions.....	8
Execution flow of program.....	9
Scope of the variable	9
“if” conditional statement with nested conditions.....	11
Loops	14
The while loop.....	14
The range function.....	15
The for loop.....	16
Advanced data types of Python	17
List.....	17
<i>List built-in functions.....</i>	19
Tuple	23
Dictionaries.....	24
<i>Dictionaries built-in functions</i>	26
<i>Perform looping on the dictionary.....</i>	30
Set.....	32
<i>Built-in method of Set.....</i>	33
Additional topic – PEP8 style guide for Python code.....	35
Naming conventions	35
Python indentation	36
Maximum line length.....	37
Block and inline comments	37
Document strings.....	38

Imports	39
Connection closing	39
List comprehension	39
Iterators	40
Generators.....	41
Conclusion	44
Questions	44
2. Major Pillars of OOPS with Python	45
Structure.....	45
Objective.....	46
Introduction to object-oriented programming (OOP) language	46
Basic idea of object.....	46
What is class?	47
Decorators in Python.....	50
Introduction of class method and static method in Python	55
Inheritance	57
Single inheritance.....	57
Multiple inheritance	58
Multilevel inheritance	60
Use of super() function.....	62
Encapsulation.....	63
Polymorphism.....	64
Method overloading.....	64
Method overriding.....	64
Abstraction	65
Abstract class.....	65
Conclusion	66
Questions	67
3. Getting Started with Django	69
Structure.....	69
Objective.....	70
Basics of Django.....	70
Django architecture	70
Django execution and basic commands	73

Django program execution flow.....	75
How Django handles requests.....	75
HTTP request.....	77
Create the view file and configure user defined URL.....	78
URL understanding.....	80
Basics of Loggers and available logger types	80
Loggers.....	80
Handler	80
Filters	80
Formatters.....	81
Types of Loggers.....	81
Logger Implementation	81
Define template in Django.....	83
Conclusion	95
Questions	95
4. API Development with Django	97
Structure.....	97
Objective.....	97
The basic idea of API.....	98
Create an app in Django	98
Connect Django with database	101
Function-based and class-based views.....	101
Function-based views.....	101
Class-based views.....	103
Postman tool description.....	104
Create function-based views.....	108
Create class-based views	112
Conclusion	115
Questions	115
5. Database Modeling with Django	117
Structure.....	117
Objective.....	118
The Django ORM	118

Django shell	119
Django database modeling.....	119
Connect Django with various databases	121
Django connectivity with PostgreSQL.....	121
Django connectivity with SQLite.....	123
Django connectivity with MySQL.....	123
Create app with model	125
Access data through shell.....	127
Insert query.....	130
Select query.....	131
Update query.....	134
Delete query.....	134
Conclusion	135
Questions	135
.	
6. First Django API Deployment on Web.....	137
Structure.....	137
Objective.....	137
The introduction of web technologies	138
uWSGI	138
NGINX.....	139
Apache.....	140
Gunicorn	141
Supervisor	142
<i>Use case of Supervisor</i>	142
API deployment on web servers with Apache.....	142
Django loggers configuration and use case	150
Conclusion	154
Questions	154
7. Django Project Deployment on Various Web Servers.....	155
Structure.....	155
Objective.....	155
New Django project creation and its overview	156

PostgreSQL database connectivity	157
Model file creation through Django ORM	158
Get and post request creation with ORM queries.....	159
Loggers implementation in the project.....	162
Deploy Django with NGINX, PostgreSQL, and uWSGI.....	164
Deploy Django with NGINX, PostgreSQL, Gunicorn, and Supervisor.....	168
Conclusion	173
Questions	
8. What are Microservices	175
Structure.....	175
Objective.....	175
The introduction of the microservices	176
Monolithic vs. Microservices	178
Monolithic.....	178
Microservices.....	180
Some important characteristics of microservice	184
The microservices way and its benefits	187
In the microservices speed of changes.....	187
In the microservices safety of changes.....	187
The way of microservices	188
Scalability in microservices	188
Microservices pitfalls	188
Conclusion	189
Questions	190
9. Designing Microservice Systems	191
Structure.....	191
Objective.....	191
Approach to microservices.....	192
Service.....	192
Solution	193
Process	193
Organization.....	193
Culture.....	193
<i>When working on the all designing element together</i>	193

Follow the standardized process or set a standard for process	194
The designing process of microservices.....	194
Setting up the optimization goals.....	194
Principles of development.....	195
Sketch the system design.....	195
The designer of the microservices system.....	196
Important points for establishing a foundation.....	196
Goals and principles.....	196
Goals for the microservices way	196
<i>Reduce the system cost</i>	197
<i>Increase release speed</i>	197
<i>Improve the system resilience</i>	198
<i>Enable the visibility</i>	198
Operating the principles	198
Shared capabilities	199
<i>Hardware services</i>	199
<i>Management of the code, it's testing and deployment</i>	199
<i>Data stores</i>	199
<i>Coordination between the services</i>	200
<i>Security and identity</i>	200
Way to host	200
Service designing parameters and boundaries	201
Domain-driven design	202
Context should be bounded	203
Smaller is better.....	203
API design for microservices	203
<i>Message-oriented</i>	204
<i>Hypermedia-driven</i>	204
Data and microservices.....	204
Asynchronous message-passing and microservices.....	205
Independent deployability	206
Microservice architecture is a product of its time	207
Security.....	207
Monitoring and alerting.....	207
Conclusion	208
Questions	208

10. Service Authentication.....	209
Structure.....	209
Objective.....	209
Authentication	210
Authorization	210
Authentication and authorization with Django.....	211
Table configuration with Django admin	220
Authenticate the services.....	222
HTTP or HTTPs based	223
Session and cookies based	223
Token based	224
Conclusion	224
Questions	224
11. Microservices Deployment with Django.....	225
Structure.....	225
Objective.....	225
Flow figure for microservice architecture	226
Data flow and description of used API	227
Service name and its purpose	227
User service.....	228
<i>Folder architecture and code</i>	228
<i>Database connection code</i>	230
<i>API code</i>	232
API request and response description.....	239
<i>userregistartion API</i>	239
<i>userlogin API</i>	240
<i>userinfo API</i>	241
Product service.....	241
<i>Folder architecture and code</i>	242
<i>Database connection code</i>	243
<i>API code</i>	244
Shipment service.....	247
<i>Folder architecture and code</i>	247
<i>Database connection code</i>	248
<i>API code</i>	250

<i>API request and response description</i>	253
Payment service	254
<i>Folder architecture and code</i>	255
<i>Database connection code</i>	256
<i>API code</i>	258
<i>API request and response description</i>	263
Microservices deployment with Django	265
Microservice architecture flow figure.....	265
Conclusion	266
Questions	266
12. JWT Auth Service	267
Structure.....	267
Objective.....	267
Introduction of JSON Web Tokens (JWT)	268
Applicable scenarios for JWT implementation	268
Structure of JWT token	268
Header	269
Payload	269
Signature	270
JWT working flow	271
JWT deployment with Django	272
APIs request and response	275
Conclusion	278
Questions	278
13. Asynchronous Tasks	279
Structure.....	279
Objective.....	280
Introduction of asynchronous task	280
Why we need asynchronous task?	280
Overview of Celery, RabbitMQ, and Redis.....	281
Celery.....	281
<i>Flow case</i>	281
<i>Use case</i>	282
RabbitMQ.....	282

<i>Flow case</i>	283
Redis	284
<i>Use case of Redis</i>	285
Installation and basic configuration of Celery with Django.....	285
Deploy Django with Celery and RabbitMQ	285
Deploy Django with Celery and Redis	287
Tasks distribution and scheduling	289
Task scheduling.....	289
<i>Use case</i>	289
<i>Schedule the tasks in Ubuntu OS</i>	290
Conclusion	291
Questions	291
14. AWS Serverless	293
Structure.....	293
Objective.....	293
Introduction of AWS Serverless.....	294
AWS API Gateway	294
AWS Lambda.....	295
RDS	295
AWS DynamoDB	295
Sample code with API Gateway and Lambda.....	296
Serverless microservice architecture on AWS	303
Django microservice architecture on AWS	305
AWS EC2 service.....	305
Django microservices architecture on AWS.....	305
Conclusion	307
Questions	307
15. How to Adopt Microservices in Practice.....	309
Structure.....	309
Objective.....	310
Guidelines for creating the solution architecture	310
How to introduce microservice in the organization?	310
What are the best practices and processes for microservices?	311

What should be the count of new features or bug fixes in the single release?	311
How can we track that our project is completely transformed into microservice?	311
Is it necessary to write all microservices code in the same programming language?	312
What technologies are available for microservices?	312
Conclusion	312

CHAPTER 1

Basics of Python

Python is a programming language that lets you work quickly and integrate systems more effectively. It has many applications, such as data analysis, web development, and app development. It has dynamic and strong data type, which can be managed and defined easily.

For microservices, we require basic knowledge of Django. This book is about microservices with Django, and to use Django you should know about Python. Thus, we'll cover the basics of Python in this chapter.

Structure

- Introduction of Python programming language
- Definition of variables, expressions and statements
- Statements
- Operations on strings
- Functions
- Python type conversion functions
- User defined functions
- Execution flow of program
- Scope of the variable
- "if" conditional statement with nested conditions
- The for and while loops
- Advanced data types of Python

- List
- List built-in functions
- Tuple
- Dictionaries
- Dictionaries built-in functions
- Python data type set
- List comprehension
- Iterators and generators
- Additional – PEP8 style guide for Python code

Objective

This chapter covers all the basic concepts of Python and these are enough to understand Django. At the end of this chapter, you can easily understand variable definition, functions, statements, advance data type, code blocks, flow of the program and coding standard. All the topics are covered with real life examples and executable code.

Introduction of Python programming language

You must have heard about many programming languages, Python is one of them.

Python is one of the dynamic programming languages compared to other languages such as Java, Perl, PHP, and Ruby. It is often termed as a scripting language. It provides support for automatic memory management, multiple programming paradigms, and implements the basic concepts of **object-oriented programming (OOP)**. Python executes its code line by line, thus it is known as an interpreter language.

The interpreter functionality lets it read a program and execute one line at a time.

The following image shows the steps of Python code compilation:

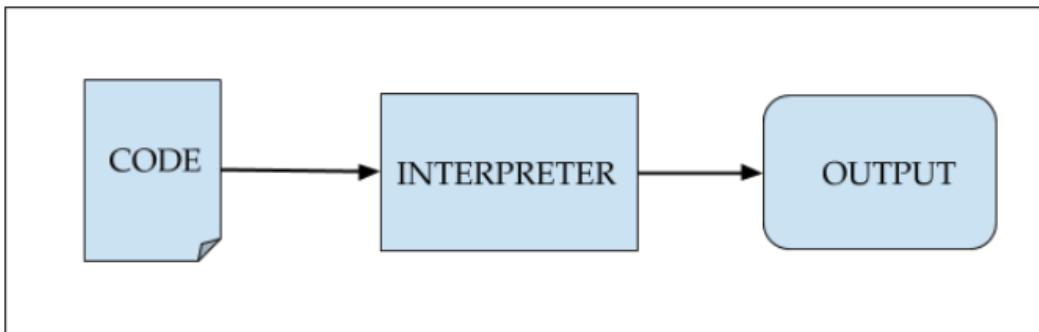


Figure 1.1: Python code compilation steps

Python code can be executed in two ways, one we can open the Python console (command-line mode) and write our code line by line, which executes the code line by line. The second way is to put all your code into a single file then execute the script. The following example shows how command-line mode works, you type the programs then interpreter prints the result:

```
$ python
Python 3.5.2 (default, Nov 12 2018, 13:43:14)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Welcome to Python World!!!!")
Welcome to Python World!!!!
```

In the above command, Python command starts the Python console application. The next two lines contain the start messages, which is the default start message from the Python interpreter. The next line starts with `>>>`, it is the prompt that indicates it is ready. We write `print ("Welcome to Python World!!!!")`, and the interpreter replies `Welcome to Python World!!!!`.

Alternatively, we can write a program inside a file and execute the file. That file is called a script. For example, we create a file `first_python_script.py` with the following code:

```
print( "Welcome to Python World!!!!" )
```

As per convention, the names of files which contain Python should end with `.py`. To execute the program, we have to tell the interpreter the name of the script:

```
$ python first_python_script.py
Welcome to Python World!!!!
```

Definition of variables, expressions, and statements

As we know, Python is a dynamic programming language. Compared to other languages like C, C++ or Java, In Python, we are not bound to define data type at the time of defining any variable. Please refer to the examples mentioned below:

- Let's take an example of string definition:

```
>>> a = "Hello Shayank"  
>>> type(a)  
<class 'str'>  
>>> print(a)  
Hello Shayank
```

- Let's take an example of integer definition:

```
>>> a = 5  
>>> type(a)  
<class 'int'>  
>>> print(a)  
5
```

- Let's take an example of float definition:

```
>>> a = 5.0  
>>> type(a)  
<class 'float'>  
>>> print(a)  
5.0
```

- Let's take an example of Boolean definition:

```
>>> a = True  
>>> type(a)  
<class 'bool'>  
>>> print(a)  
True
```

In the above mentioned examples, `type()` method is used. It is a built-in method in Python, which is used to know the data type of the variable. The syntax is `type(variable)` as mentioned in the above example.

There are few ways in Python which are not allowed to define the variable; if we use these ways, it will throw a syntax error:

```
>>> 09var = "program"  File "<stdin>", line 1
    09var = "program"
    ^
SyntaxError: invalid syntax
```

```
>>> variable$ = 1000000
    File "<stdin>", line 1
        variable$ = 1000000
        ^
SyntaxError: invalid syntax
```

```
>>> class = "test hello"
    File "<stdin>", line 1
        class = "test hello"
        ^
SyntaxError: invalid syntax
```

`09var` is not allowed because the variable name does not start with a letter. `variable$` is gives an error because it contains the special symbol, that dollar sign. But you may be thinking why not `class`?

It is through error because the `class` belongs to the Python keywords list. Keywords are special words that define the rules of language and the structure. Thus, we cannot use them as variable names.

Python has the following reserved keywords:

<code>False</code>	<code>class</code>	<code>finally</code>	<code>is</code>	<code>return</code>	<code>None</code>	<code>continue</code>	<code>for</code>
<code>lambda</code>	<code>try</code>	<code>True</code>	<code>def</code>	<code>from</code>	<code>nonlocal</code>	<code>while</code>	<code>And</code>
<code>del</code>	<code>global</code>	<code>not</code>	<code>with</code>	<code>as</code>	<code>elif</code>	<code>if</code>	<code>or</code>
<code>yield</code>	<code>assert</code>	<code>else</code>	<code>import</code>	<code>pass</code>	<code>break</code>	<code>in</code>	<code>raise</code>

Statements

Instructions written in Python are called statements; these statements are executed by the Python interpreter. In the Python language, statements are divided into two types: print and assignment. When we write a statement on the console screen, it runs and displays the output. The script can contain one or many statements in the file. If it contains more than one statement, then the interpreter executes and prints the output one at a time just as the statements execute.

Example of the script is as follows:

```
print("Hi")  
variable1 = 35  
print(variable1)
```

We will get the following output for the above code snippet:

```
Hi  
3
```

Operations on strings

If we consider some other languages like C, C++, or Java, they don't allow performing mathematical operations on strings data type variable and the strings as numbers. The following are not allowed (assuming that message has type string):

```
shayank-1  
"shayank"/123  
shayank*"jain"  
"415"+2
```

Python has allowed the + operator which also works with strings. In Python, if we use the + operator for strings, it represents concatenation, which means, it joins values. For example:

```
variable1 = "I am"  
variable2 = " Python Developer"  
print (variable1 + variable2)
```

As expected, the output of this program is I am Python Developer. Note that there is a space before the word Python; it is counted as part of the string by the interpreter. The other operator is * that performs repetition of the string.

For example, one value should be a string and others have to be an integer. 2*3 is equivalent to 2+2+2, so as expected, "Shayank"*3 is "Shayank"+"Shayank"+"Shayank".

Functions

In the previous examples, we used the `type()` function:

```
>>> type("45")
<type 'str'>
```

In a function, it takes an argument and produces a result in return. The result is called the return value. We can also assign return value to a variable:

```
>>> value1 = type("24")
>>> print(value1)
<type 'str'>
```

Now, let's see another built-in function, that is, `id()`. In this function, we have to pass the parameters to get the output, so as a parameter; we can pass any value or a variable. In the result of this function, it returns an integer value, which works as a unique value. When we define any value has an ID, it is always a unique number that is related to where it is stored in the memory of the computer.

```
>>> id(56)
145387862
>>> value1 = 56
>>> id(value1)
145387862
```

Python type conversion functions

Type conversion is the functionality, which is available in many languages. It is the concept which converts the variable or value from one data type to another data type. In Python, few built-in functions are available for type conversion, which we have discussed in below mentioned examples:

```
>>> int("90")
90
```

Let's see a string example:

```
>>> int("Hello")
ValueError: invalid literal for int(): Hello
```

It can also convert `float` values to integers, but it removes the fractional part:

```
>>> int(9.99999)
9
>>> int(-4.9)
```

-4

If we use `float` function then it converts integers and strings to floating-point numbers:

```
>>> float(24)  
24.0  
>>> float("7.9059")  
7.9059
```

Let's use `str` function:

```
>>> str(78)  
'78'  
>>> str(4.4765)  
'4.4765'
```

User-defined functions

In this concept, we can define a function in our way. In simple words, we can create scenario wise function. These blocks of code are called user-defined functions. To understand this concept, let's look at an example:

```
def Function_Name(LIST OF PARAMETERS):  
    STATEMENTS
```

Here is the example of user-defined function without parsing the parameters:

```
def welcome():  
    print("Welcome to the Python world")
```

Here, `welcome()` is the name of a user-defined function. The empty parentheses indicate that it will execute without parameters parsing. It contains a single statement, which produces the outputs is `Welcome to the Python world`. In the below mentioned example we can see the definition and calling of the function:

```
def sum_fun(value1, value2):  
    return value1+value2  
sum_fun(4,6) # Way of calling function
```

We will get the following output for the above code snippet:

10

Execution flow of program

Python follows the top to bottom execution technique, so every time we create the function, it should be defined on top, before the calling of that function. Interpreter starts the execution of the code from the first line, it executes one statement at a time and this way, the whole program is executed.

If we define any statement inside any function, it will not execute until we call the function explicitly. To understand better, refer to the following example:

```
# Program to multiply two values:  
number1 = 5 #variable definition  
number2 = 6 #variable definition  
  
# definition of user defined function that is multi_value.  
def multi_value(v1, v2):  
    return v1 * v2  
  
# Assigning multi_value() function return value inside the var variable.  
var = multi_value(number1,number2)  
print(var)
```

We will get the following output for the above code snippet:

30

In the above code, the program will execute from the first statement variable initialization. Then it moves to the function and will read the function definition. After that, it goes to `var` and then calls the `multi_value()` function. When the call goes to function, it executes the `return` statement and finally, it calls the `print` statement.

Scope of the variable

In Python, variables are of two types: local variable and global variable. A local variable is that if we declared any variable inside the function then the variable can be accessed only inside the function. If we try to access it out of the function, it is not supported. Please refer to the following example:

```
def give_int_value():  
    value1 = 10  
    return value1
```

```
v1 = give_int_value() # Calling the function  
print(v1)
```

We will get the following output for the above code snippet:

10

In the given example, if we call `value1` from outside it will give an error, as shown below:

```
def give_int_value():  
    value1 = 10  
    return value1
```

```
v1 = give_int_value()  
print(v1)  
print(value1)
```

We will get the following output for the above code snippet:

10

NameError: name 'value1' is not defined

In the above code snippet, the sequence will print `v1` value in the first place, that is `10`, and then it produces the error, which means the `value1` is a local variable for the `give_int_value()` function.

Now, we are going to see a global variable with the same example:

```
value1 = 10 #global variable
```

```
def give_int_value():  
    return value1
```

```
v1 = give_int_value()  
print(v1)  
print(value1)
```

We will get the following output for the above code snippet:

10

10

In the above example the first statement of this program is `value1` declaration. The global variable definition should be above in the code. If we define a variable in all the blocks, it is called a global variable and it can be accessed in any blocks of our code. At the time of variable definition, it is stored in the memory in the first place in comparison to others. So by default, its scope is increased.

"if" conditional statement with nested conditions

The `if` statement is known as the conditional statement and it is used in programming for validation purpose. On the basis of validation results, it changes the program behavior. Let's take a look at the below example:

```
if val > 0:  
    print " val is positive value."
```

When we use the `if` statement, in the backend, it always executes a Boolean expression; for, example if the condition is true then it will execute the statement otherwise it will return nothing. This is an example of a single `if` statement: to complete the sequence, we write the `if` statement with `else`. It is prevention which saves our program from the exception. In programming, it is the standard, which we should follow, that is, `if` statement with `else`. Please refer to the following example:

```
if val > 18:  
    print (" You are Adult.")  
else:  
    print ("You are under age.")
```

So, if the first condition is false then the program goes to the second condition.

We can also write back to back `if` and `else` condition, as per the user's problem statement. This helps to manage the program more effectively. The `elif` statement is an abbreviation of `else...if`.

Please refer to the following example:

```
v1 = 6  
v2 = 8  
# Nested if statement  
if v1 < v2:  
    print (v1, "is less than", v2)  
elif v1 > v2:
```

```
    print (v1, "is greater than", v2)
else:
    print (v1, "and", v2, "are equal")
```

We will get the following output for the above code snippet:

6 is less than 8

In the above mentioned code, each statement is checked in order. If the first is false, it will check the next, and so on. If one of them is true, the corresponding code will execute, and the statement ends. If more than one condition is true, only the first true statement will execute.

Let's try nested conditions with functions. We are going to check what the legal age for election in India is, in this example:

```
age = 18
# function definition with nested condition.
def age_check(v1):
    if v1 > 18:
        print("Eligible for voting.")
    elif v1 == 18:
        print("Congratulation now you are eligible for voting.")
    else:
        print("Not eligible for voting.")
# calling function by parsing age variable.
age_check(age)
```

We will get the following output for the above code snippet:

Congratulation now you are eligible for voting.

In the above program, we defined a global variable `age`. We defined function `age_check()` with nested conditions and called the function with `age` variable parsed. It checks the first statement; if it is FALSE, it moves to the next. In the second it is TRUE, so the statement gets executed and the next check is stopped.

We have one method that is `input()` for taking input from the user, that means we will get to work on dynamic variables. Please refer to the following example:

```
# function definition with nested condition.
def age_check(v1):
    if v1 > 18:
```

```
        print("Eligible for voting.")

    elif v1 == 18:
        print("Eligible for voting.")

    else:
        print("Not eligible for voting.")

# function for taking input from user
def age_input():
    # defining age variable, which is taking input from user.
    age = input("Enter your age: ")

    # Checking variable that is it not null.
    if age!='':
        # when if statement is TRUE then it will call function.
        age_check(int(age)) # example for type casting.

    else:
        # It is the example of recursive function or calling function from
        # inside the function.
        age_input()

age_input()
```

age_input()

We will get the following output for the above code snippet:

Enter your age:

User press Enter key

Enter your age:

24

Eligible for voting.

In the above mentioned code, we used the `input()` built-in function for taking input from the user. We used type casting, recursive call of function into the program. Typecast is required in the program because when we take input from user it considers that the default data type of value is string, so for checking we have to typecast string value from `str` to `int` at the time of passing the variable. By using recursive call we resolve one issue which can arise at the time of input, i.e. if a user

presses enter then it will send blank as input. In the above program, it will call the `age_input()` function first and then check if the user has given a valid input. After that, it calls the `age_check()` function and gives output as per input check.

Loops

A loop is a programming function, which iterates a statement based on specified conditions. In this chapter we will learn about different types of loop.

The while loop

It is the loop which is used to execute a block of code repetitively till the user output does not come. It executes the code continuously and stops when its condition returns `True`. For better understanding, we refer to the following example:

```
# This program is going to execute till var value is less than 10 in while statement condition.
```

```
var = 1
while var < 10:
    print(var)
    if (var == 4):
        break # This keyword will break the execution.
    var += 1
```

We will get the following output for the above code snippet:

```
1
2
3
4
```

In the above mentioned example, we initialize the variable which is `var` and define the `while` statement. When the `while` loop is executed, it checks the `var` value till `var` value is less than `10`. Inside the `while` loop, we used the `if` statement for checking `var` value until it became `4`. When `var` value is `4`, it breaks the flow. Here, `break` is a predefined keyword in Python.

Now, see the other example using `while` loop `with` function:

```
# Program for showing countdown decrease.

def give_count(value):
    value = int(value)
    while value > 0:
```

```
if value == 1:  
    print("count is finished!!!!")  
    break  
else:  
    value = value-1  
    print(value)  
va = input("Enter value for count: ")  
give_count(va)
```

We will get the following output for the above code snippet:

```
Enter value for count: 4
```

```
3
```

```
2
```

```
1
```

```
count is finished!!!!
```

The range function

It is a Python built-in function, which is commonly used with **for** loop. There are several ways to use range, which have **range()** with for and without for. In the function, we give an integer value as a parameter; basically, we provide that to know how many times the function should execute. Let's see the **range()** syntax and arguments:

```
range (start, stop, step)
```

Here, the **range()** function takes three arguments, out of which 2 arguments are optional, that are **start** and **step**.

The range function always starts its count from 0 to n-1. If we define 5, that means it will execute 5 times but the count starts from 0.

For better understanding let's have a look at the example. I want to print the value from 0 to 5 by using the **range** function:

```
for x in range(6):  
    print(x)
```

We will get the following output for the above code snippet:

```
0
```

```
1
```

```
2
```

3

4

5

In another example, I want to print the value between 1 to 10 with a difference of 2 by using the `range` function:

```
for x in range(1,11,2):
    print(x)
```

We will get the following output for the above code snippet:

1

3

5

7

9

The for loop

Now, we will see how `range` function is used within `for` loop in user-defined function with a real-life example. When we go to an ATM machine, it asks us to enter the password and gives 3 attempts:

```
def pass_check(variable1):
    if variable1 == "5698": # password checking code
        print("Welcome")
        return 2
    else:
        print("Wrong Password!!!!")

def user_input():
    for x in range(4): # range function for gave the chance.
        if x == 3:
            print("Your wrong password limit is exceed.")
            break
        else:
            v1 = input("Enter your ATM PIN: ")
            v2 = pass_check(v1)
```

```
if v2:  
    break  
  
user_input()
```

We will get the following output for the above code snippet:

```
Enter your ATM PIN: 456.  
Wrong Password!!!!  
Enter your ATM PIN: 7894  
Wrong Password!!!!  
Enter your ATM PIN: 5698
```

```
Welcome
```

The above is an example of a password checking system, which gives three attempts to enter the correct password. It takes input from the user then checks it and responds as per input.

Advanced data types of Python

Python is one of the strongest languages for dynamic data type allocation. We previously saw the common datatype, which is available in many languages, but Python has some advanced level datatype, which can be stored in more than one value into a single data type. Also, it is not bound to store similar data type into a single variable. Advance datatype is as follows:

- List
- Tuple
- Dictionary
- Set

List

The list is an example of Python advanced data type, which can be defined into many variations. It has many ways to declare a list, but the simplest or you can say a common way, which is available in many places is to enclose the elements in square brackets ([]). Please refer to the following example:

```
[1, 2, 3, 4]  
['a', 'b', 'c']  
[1, 'a', 3.4, 'b']
```

If we want to define a sequential list with integer value then:

```
>>>range(5)
```

This example works with Python 2 and we will get the following output:

[0,1,2,3,4,5]

We can also define an empty list in Python and it is denoted with [].

a = []

Now, let's see how to access list and its elements individually, refer to the following example:

```
list1 = ["a", "b", "c", "d", "e"]
print(list1[1])
print(list1[4])
```

We will get the following output for the above code snippet:

b

E

In the above mentioned code, the example of list, **list1 = ["a", "b", "c", "d", "e"]**. If a user wants to access the element b from **list1**, it should be called by **list1[1]** way, because list works on the index value. Similarly, if we want to access another element like e then **list1[4]**.

- If the user tries to call list by **list1[5]** then it will raise "IndexError: list index out of range" error.
- List always start with index value [0].
- The list has many features. One special feature is that list can also be called by its negative index value. It produces the reverse list as a result, for example, **list1[-1]**.

Please refer to the following example:

```
list1 = ["a", "b", "c", "d", "e"]
print(list1[5])
```

We will get the following output for the above code snippet:

IndexError: list index out of range

Let's see another example where we have list and if we want to print reverse element of the list, then:

```
list1 = ["a", "b", "c", "d", "e"]
for x in range(1,(len(list1)+1)):
    print(list1[-x])
```

We will get the following output for the above code snippet:

```
e  
d  
c  
b  
a
```

In the above example, we have a list and for sequential printing, we took range function. Negative index starts from -1, so we took `range(1, (len(list1)))`. We also included +1 in `range` because when we use range it will count from 1 to `(n-1)`. In print statement, `[-x]` means it will search the list for negative values such as `list1[-1], list1[-2], list1[-3]`, and so on.

List built-in functions

The list has some built-in functions which are as follows:

- `count()`: The `count` function is used for getting the count of element or object inside the list. Let's see an example:

```
list1 = [1,2,3,4,5,2,2,3,4]  
print(list1.count(2))
```

We will get the following output for the above code snippet:

```
3
```

- `append()`: Sometimes, we miss some elements at the time of list initialization or sometimes, it is a product requirement that result should also be stored in the same list. Many other scenarios are available when we have to add a new element inside the list, so we use `append`. Let's see an example:

```
list1 = [1,2,3,4,5,2,2,3,4]  
list1.append(34)  
print(list1)
```

We will get the following output for the above code snippet:

```
[1, 2, 3, 4, 5, 2, 2, 3, 4, 34]
```

- `extend()`: It works similar to the `append` function, This means it will concat both the lists. Let's see an example:

```
list1 = [1, 2, 3, 4, 5, 2, 2, 3, 4, 34]  
list2 = [33,44,55,66]  
list1.extend(list2)
```

```
print(list1)
```

We will get the following output for the above code snippet:

```
[1, 2, 3, 4, 5, 2, 2, 3, 4, 34, 33, 44, 55, 66]
```

- **len()**: By using the `len()` function, we can get the length of the list. Let's see an example:

```
list1 = [1, 2, 3, 4, 5, 2, 2, 3, 4, 34, 33, 44, 55, 66]
```

```
print(len(list1))
```

We will get the following output for the above code snippet:

```
14
```

- **reverse()**: As per requirement, if we want to print list value, which should start from the last element then `reverse()` function is perfect to reverse the list elements. Let's see an example:

```
list1 = [1, 2, 3, 4, 5, 2, 2, 3, 4, 34, 33, 44, 55, 66]
```

```
list1.reverse()
```

```
print(list1)
```

We will get the following output for the above code snippet:

```
[66, 55, 44, 33, 34, 4, 3, 2, 2, 5, 4, 3, 2, 1]
```

- **min()**: This function works with the list, it supported a similar data type and also integer or float. With the help of `min()` function we can find the smallest value element of the list. Let's see an example:

```
list1 = [66, 55, 44, 33, 34, 4, 3, 2, 2, 5, 4, 3, 2, 1]
```

```
print(min(list1))
```

We will get the following output for the above code snippet:

```
1
```

- **max()**: It also works on similar data type elements and it gives the highest value of the element from the list. Let's see an example:

```
list1 = [66, 55, 44, 33, 34, 4, 3, 2, 2, 5, 4, 3, 2, 1]
```

```
print(max(list1))
```

We will get the following output for the above code snippet:

```
66
```

- **index()**: In a few scenarios, the user wants to know the element location from the list, or in technical terms, wants to know the index value of the element. In this case, the `index` function is used for the same. Let's see an example:

```
list1 = [66, 55, 44, 33, 34, 4, 3, 2, 2, 5, 4, 3, 2, 1]
print(list1.index(5))
```

We will get the following output for the above code snippet:

```
9
```

- **insert()**: If you want to add any new element into a list on a particular location, then the **insert()** function takes the position or index value for storing the new element. Let's see an example:

```
# syntax for insert function list1.insert(index, object)
list1 = [66, 55, 44, 33, 34, 4, 3, 2, 2, 5, 4, 3, 2, 1]
list1.insert(4,99)
print(list1)
```

We will get the following output for the above code snippet:

```
[66, 55, 44, 33, 99, 34, 4, 3, 2, 2, 5, 4, 3, 2, 1]
```

- **pop()**: It is used to remove an element from the list and we can give the position as well. Let's see an example:

```
list1 = [66, 55, 44, 33, 99, 34, 4, 3, 2, 2, 5, 4, 3, 2, 1]
list1.pop(2)
print(list1)
```

We will get the following output for the above code snippet:

```
[66, 55, 33, 99, 34, 4, 3, 2, 2, 5, 4, 3, 2, 1]
```

- **remove()**: If you want to delete any specific element from the list, then by using the **remove()** function we can remove an element from the list. Let's see an example:

```
list1 = [66, 55, 33, 99, 34, 4, 3, 2, 2, 5, 4, 3, 2, 1]
list1.remove(2) # It removes the first occurrence only.
print(list1)
```

We will get the following output for the above code snippet:

```
[66, 55, 33, 99, 34, 4, 3, 2, 5, 4, 3, 2, 1]
```

- **list()**: It is used for typecasting. Let's see an example:

```
tuple1 = (11,22,77,88)
print(type(tuple1)) # It will print tuple.
print(type(list(tuple1))) # It will print list.
```

We will get the following output for the above code snippet:

```
<class 'tuple'>
<class 'list'>
```

- `sort()`: If you want all elements of the list in sequence, we use this function for sorting the list. Let's see an example:

```
list1 = [66, 55, 33, 99, 34, 4, 3, 2, 5, 4, 3, 2, 1]
list1.sort()
print(list1)
```

We will get the following output for the above code snippet:

```
[1, 2, 2, 3, 3, 4, 4, 5, 33, 34, 55, 66, 99]
```

We can also call list in a different way. Let's take an example, I have a list of email IDs in my list and I have a user who wants to register, so we have to check if the email is already registered with us or it is new.

Please refer to the following example:

```
emp_list = ['test01@gmail.com','test02@gmail.com','test03@gmail.com']
emp = input('Enter your email id: ')
if emp in emp_list:
    print(" Your Email id is already registered.")
else:
    print(" Your email seems new for registration")
```

We will get the following output for the above code snippet:

```
Enter your email id: try_test@gmail.com
Your email seems new for registration
```

I want to introduce one more built-in function that is, slicing. It is very simple. The syntax is `slice[start : stop : step]`. It has following parameters:

- `start`: It specifies which position to start slicing from. It is optional.
- `stop`: It specifies the position to end the slicing.
- `step`: It specifies the step of slicing. Its default value is 1. It is optional.

Please refer to the following example:

```
value1 = ("a", "b", "c", "d", "e", "f", "g", "h")
v1 = slice(2,6,2)
print(value1[v1])
```

We will get the following output for the above code snippet:

```
('c', 'e')
```

Or

```
value1 = ("a", "b", "c", "d", "e", "f", "g", "h")
print(value1[slice(2,6,2)])
```

We will get the following output for the above code snippet:

```
('c', 'e')
```

Tuple

To understand tuple, first we should know about two words: mutable and immutable. Let's see them:

- **Mutable:** Those objects which can be changed after their creation are called mutable.
- **Immutable:** Those objects which cannot be changed after their creation are called immutable.

For example, we have seen some data type, which is available in Python, that are made up of their data type like string is made up of characters and list can be made up of any type of elements.

The differences between both of them are that we can modify the list elements but cannot change the character from the string. In that scenario, string is immutable and the list is mutable data type. So tuple is also immutable data type, it is sort of looks like list, which means defined values that are comma-separated but it starts and ends with () round brackets. The following is the syntax for it:

```
a = (1,2,3,4)
```

We can also tuple like this, a = 1,2,3,4. It is not bound with round brackets.

We can do some similar operations with tuple like a list, here are some examples:

```
tuple1 = ('a','b','c','d')
```

The following code snippet is for accessing tuple value:

```
print(tuple1[1])
```

We will get the following output:

```
b
```

Let's see an example of slicing tuple value:

```
print(tuple1[slice(1,3)])
```

We will get the following output for the above code snippet:

```
('b', 'c')
```

Now, let's concat the tuple:

```
tup = tuple1 + ('A',)  
print(tup)
```

We will get the following output for the above code snippet:

```
('a', 'b', 'c', 'd', 'A')
```

If we try to modify tuple then,

```
>>> tup = ('a', 'b', 'c', 'd', 'A')  
>>> tup[1] = 'B'
```

We will get the following output for the above code snippet:

```
TypeError: 'tuple' object does not support item assignment
```

So we cannot update tuple or change the value. If we want to update tuple then we can define another tuple and store the new value. Please refer to the following example:

```
tup = ('a', 'b', 'c', 'd', 'A')  
tup1 = (1,2,3,4)  
tup3 = tup + tup1  
print(tup3)
```

We will get the following output for the above code snippet:

```
('a', 'b', 'c', 'd', 'A', 1, 2, 3, 4)
```

Let's see an example of deleting the tuple:

```
tup3 = ('a', 'b', 'c', 'd', 'A', 1, 2, 3, 4)  
del tup3 # It will remove tup3.
```

Dictionaries

It is part of Python advance data type. It stores value in key and pair format, which means there is always one index which points to data. In dictionaries, keys are immutable but the value is mutable type. The definition of the dictionary is that it starts and ends with curly braces but it stores values in the key-value format which is separated by a colon (:). The following is the syntax for dictionary:

```
dict = { key : value }
```

We can also define the empty dictionary. Please refer to the following example:

```
dict1 = {} # Empty dictionary
dict2 = {"a": 23, "b": "hello", 45: 22.5} # sample dictionary
```

After assigning the empty dictionary we can add elements to it:

```
dict1['key1'] = 'value1'
dict1['key2'] = 'value2'
print(dict1)
```

We will get the following output:

```
{'key1': 'value1', 'key2': 'value2'}
```

For accessing dictionary we can call it with `dict1['key2']`, and it will return its value. Now we will see what is the difference between copy or referencing dictionary in the following examples:

```
dict1 = {"Acc01": "Rahul", "Acc02": "Raj", "Acc03": "Ram", "Acc04": "Rohan"}
# It is referencing the dict1 object to dict2.
dict2 = dict1
# so in result dict2 containing same object but if we changed into dict2
element that update into dict1.

print(dict2)
dict2['Acc03'] = "Sonal"
print(dict2)
print(dict1) # Changes are reflected directly.
```

We will get the following output:

```
{'Acc01': 'Rahul', 'Acc02': 'Raj', 'Acc03': 'Ram', 'Acc04': 'Rohan'}
{'Acc01': 'Rahul', 'Acc02': 'Raj', 'Acc03': 'Sonal', 'Acc04': 'Rohan'}
{'Acc01': 'Rahul', 'Acc02': 'Raj', 'Acc03': 'Sonal', 'Acc04': 'Rohan'}
```

That is reference of dictionary not copy, there is `copy()` function for copying all value from one to another. Please refer to the following example:

```
dict1 = {"Acc01": "Rahul", "Acc02": "Raj", "Acc03": "Ram", "Acc04": "Rohan"}
# It is coping the dict1 object to dict2.
dict2 = dict1.copy()
# In the result dict2 copied all object of dict1 so if we changed into
dict2 element that cannot reflect into dict1.
```

```
print(dict2)
dict2['Acc03'] = "Sonal"
print(dict2)
print(dict1)
```

We will get the following output for the above code snippet:

```
{'Acc01': 'Rahul', 'Acc02': 'Raj', 'Acc03': 'Ram', 'Acc04': 'Rohan'}
{'Acc01': 'Rahul', 'Acc02': 'Raj', 'Acc03': 'Sonal', 'Acc04': 'Rohan'}
{'Acc01': 'Rahul', 'Acc02': 'Raj', 'Acc03': 'Ram', 'Acc04': 'Rohan'}
```

Dictionaries built-in functions

The following are the built-in dictionary functions along with the description of function name:

- **items()**: It is used for fetching both key and value. Let's see an example:

```
>>> dict1 = {"Acc01": "Rahul", "Acc02": "Raj", "Acc03": "Ram", "Acc04": "Rohan"}
>>> dict1.items()
```

We will get the following output:

```
dict_items([('Acc01', 'Rahul'), ('Acc02', 'Raj'), ('Acc03', 'Ram'),
('Acc04', 'Rohan'))]
```

- **keys()**: It is used for fetching keys from the dictionary. Let's see an example:

```
>>> dict1 = {"Acc01": "Rahul", "Acc02": "Raj", "Acc03": "Ram", "Acc04": "Rohan"}
>>> dict1.keys()
```

We will get the following output:

```
dict_keys(['Acc01', 'Acc02', 'Acc03', 'Acc04'])
```

- **values()**: It is used for fetching values from the dictionary. Let's see an example:

```
>>> dict1 = {"Acc01": "Rahul", "Acc02": "Raj", "Acc03": "Ram", "Acc04": "Rohan"}
>>> dict1.values()
```

We will get the following output:

```
dict_values(['Rahul', 'Raj', 'Ram', 'Rohan'])
```

- **get()**: It is used for fetching value by providing key from the dictionary.

Let's see an example:

```
>>> dict1 = {"Acc01": "Rahul", "Acc02": "Raj", "Acc03": "Ram", "Acc04": "Rohan"}  
>>> dict1.get("Acc04")
```

We will get the following output:

```
'Rohan'
```

- **fromkey()**: It creates a new dictionary from the given sequence and value of elements from the user. Let's see an example:

```
d1 = [1,2,3,4]  
dict2 = dict.fromkeys(d1, 'value1')  
print(dict2)
```

We will get the following output:

```
{1: 'value1', 2: 'value1', 3: 'value1', 4: 'value1'}
```

- **clear()**: It removes all items from the dictionary. Let's see an example:

```
>>> dict1 = {"Acc01": "Rahul", "Acc02": "Raj", "Acc03": "Ram", "Acc04": "Rohan"}  
>>> dict1.clear()
```

We will get the following output:

```
>>> dict1  
{}
```

- **pop()**: It removes key and return values. Here's the syntax: `dict.pop(key, value)`. The `pop()` method returns value sequence. If a given key already exists in the dictionary, then it will remove an element from the dictionary. If the key does not exist then it returns a specified value, which is defined in the second argument. If the key is not found and the default argument is not specified then `KeyError` exception is raised.

Let's see the different cases in form of examples:

Case-1:

```
# If a given key already exists in the dictionary, then it will  
remove an element from the dictionary.  
  
class_rollnumber = { 'Ro001': 'Ram', 'Ro002': 'shayam', 'Ro003':  
'babubhaiya' }  
  
element = class_rollnumber.pop('Ro001')  
print('The popped element is:', element)
```

```
print('The dictionary is:', class_rollnumber)
```

We will get the following output for the above code snippet:

The popped element is: Ram

```
The dictionary is: {'Ro002': 'shayam', 'Ro003': 'babubhaiya'}
```

Case-2:

```
# If the key does not exist then it returns specified value, which  
defined in the second argument.
```

```
class_rollnumber = { 'Ro001': 'Ram', 'Ro002': 'shayam', 'Ro003':  
'babubhaiya' }  
  
element = class_rollnumber.pop('Ro004', 'RAJ')  
  
print('The popped element is:', element)  
  
print('The dictionary is:', class_rollnumber)
```

We will get the following output for the above code snippet:

The popped element is: RAJ

```
The dictionary is: {'Ro001': 'Ram', 'Ro002': 'shayam', 'Ro003':  
'babubhaiya'}
```

Case-3:

```
# If the key is not found and default argument is not specified then  
KeyError exception is raised.
```

```
class_rollnumber = { 'Ro001': 'Ram', 'Ro002': 'shayam', 'Ro003':  
'babubhaiya' }  
  
element = class_rollnumber.pop('Ro004')  
  
print('The popped element is:', element)  
  
print('The dictionary is:', class_rollnumber)
```

We will get the following output for the above code snippet:

KeyError: 'Ro004'

- **update():** It updates key and can adds a new key-value pair into the dictionary. Let's see an example:

```
dict1 = {"A001": "Sam", "A002": "Harry"}  
  
dict2 = {"A002": "Ronald"}  
  
# updating the value of key A002  
  
dict1.update(dict2)  
  
print(dict1)
```

```
dict3 = {"A003": "Jack"}  
# adding new element  
dict1.update(dict3)  
print(dict1)
```

We will get the following output for the above code snippet:

```
{'A001': 'Sam', 'A002': 'Ronald'}  
{'A001': 'Sam', 'A002': 'Ronald', 'A003': 'Jack'}
```

- **popitem():** It removes the last element from the dictionary with key and value. Let's see an example:

```
person = {'name': 'Phill', 'age': 22, 'salary': 3500.0, 'Account':  
'ICICI000120'}  
  
result = person.popitem()  
  
print(person)  
  
print(result)
```

We will get the following output for the above code snippet:

```
{'name': 'Phill', 'age': 22, 'salary': 3500.0}  
('Account', 'ICICI000120')
```

- **setdefault():** It returns the value of a key (if the key already exists). If not, then it inserts key with a given value. The syntax is: `dict.setdefault(key,value)`. It returns the following results:

- o If the key already exists into a dictionary then it returns the value of the key.
- o If the key is not available in the dictionary and value is not provided then it returns default None.
- o If the key is not available in the dictionary and value is provided then it returns provided value.

Let's see different cases in the form of examples:

```
address_info = {'city': 'Mumbai', 'area': 'Andheri'}
```

Case-1:

If the key already exists into a dictionary then it returns the value of the key.

```
area = address_info.setdefault('area')  
  
print('Output of case-1, address_info: ', address_info)  
print('area detail:', area)
```

We will get the following output for the above code snippet:

```
address_info: {'city': 'Mumbai', 'area': 'Andheri'}  
area detail: Andheri
```

Case-2:

if the key is not available in the dictionary and value is not provided then it returns default None.

```
pincode = address_info.setdefault('pincode')  
print('Output of case-2, address_info: ',address_info)  
print('pincode: ',pincode)
```

We will get the following output for the above code snippet:

```
address_info: {'city': 'Mumbai', 'area': 'Andheri', 'pincode': None}  
pincode: None
```

Case-3:

if the key is not available in the dictionary and value is provided then it returns provided value.

```
state = address_info.setdefault('state', 'maharashtra')  
print('Output of case-3, address_info: ',address_info)  
print('pincode: ',state)
```

We will get the following output for the above code snippet:

```
address_info: {'city': 'Mumbai', 'area': 'Andheri', 'pincode': None, 'state': 'maharashtra'}  
pincode: maharashtra
```

Perform looping on the dictionary

We go with the example to use looping on a dictionary. Most of you must be aware of tea or coffee vending machines, so we are going to use this example. If you are not aware then no issues, let's see some basic information about vending machines. In front of the machine, there are buttons and every button has some description like press 1 for coffee, 2 for masala tea, 3 for lemon tea, 4 for a latte, 5 for black tea, 6 for a cappuccino, etc. We are going to a dictionary similar to that, refer to the following example:

```
vending_machine = {1: "coffee", 2: "masala tea", 3: "lemon tea", 4: "latte", 5: "black tea", 6: "cappuccino"}  
# function for taking input and validation.
```

```
def user_input():
    for key1, value1 in vending_machine.items():
        print (key1 , value1)
    # Taking input from user and stored into variable.
    var_input = input("Please select from given\n Your choice: ")
    # if statement for checking value is given.
    if var_input:
        # Accessing value from dictionary.
        var1 = vending_machine.get(int(var_input))
        if var1:
            print("Machine is releasing",var1)
        else:
            print("Please select from given choice:")
            user_input()
    else:
        print("Please enter valid value:")
        user_input()

# calling function
user_input()
```

We will get the following output for the above code snippet:

```
1 coffee
2 masala tea
3 lemon tea
4 latte
5 black tea
6 cappuccino
Please select from given
Your choice: 5
Machine is releasing black tea
```

Set

Set is a collection of unordered elements and every element is unique (no duplicate is allowed). It is a mutable data type, we can add and remove element from the set. In comparison to list, set is more optimized to find the contained element. We can create set in two ways, first is to define an element with comma separated and bind inside {} curly braces and the second way is to define inside `set()`.

We have various ways to define a set, let's see them one-by-one. The first way to define is as follows:

```
set1 = {1,2,3,4,3,2}  
print(set1)
```

We will get the following output:

```
{1, 2, 3, 4}
```

The second way to do the same is:

```
set2 = set([1,2,3,2])  
print(set2)
```

We will get the following output:

```
{1, 2, 3}
```

Creating an empty set is tricky because if we define `my_set = {}` it is treated as empty dictionary. So for empty set we have to define `my_set = set()`. Refer to the below example:

```
my_set1 = {} # empty dictionary  
print(type(my_set1))  
my_set2 = set() # empty set  
print(type(my_set2))
```

We will get the following output for the above code snippet:

```
<class 'dict'>  
<class 'set'>
```

We can define set as immutable. These types of set objects are called frozen set.

```
normal_set_example = set(["This","is","Normal", "set"])  
# Adding an element into Normal set  
normal_set.add("example")  
print("Normal Set")  
print(normal_set)
```

```
# A frozen set example
frozen_set_example = frozenset(["This","is","Frozen", "set"])
print("Frozen Set")
print(frozen_set)
# Adding an element into Frozen set
frozen_set.add("example")
```

We will get the following output for the above code snippet:

Normal Set

```
{'set', 'example', 'is', 'This', 'Normal'}
```

Frozen Set

```
frozenset({'set', 'Frozen', 'is', 'This'})
```

Traceback (most recent call last):

```
  File "set_test.py", line 33, in <module>
    frozen_set_example.add("example")
```

```
AttributeError: 'frozenset' object has no attribute 'add'
```

Built-in method of Set

The following is the list of built-in methods of Set:

- **add():** It adds a new element into the set, if not already available. Let's see an example:

```
fruit_set1 = {"apple", "grapes", "mango"}
fruit_set2 = {"pineapple", "banana"}
fruit_set1.add("watermelon")
```

We will get the following output:

```
{'mango', 'apple', 'grapes', 'watermelon'}
```

- **union():** It combines both sets and returns a single result. Let's see an example:

```
fruit_set1.union(fruit_set2)
```

We will get the following output:

```
{'mango', 'apple', 'banana', 'pineapple', 'watermelon', 'grapes'}
```

- **difference():** It checks and returns a set containing all the elements of the invoking set but not of the second set. Let's see an example:

```
fruit_set1 = {"apple", "grapes", "mango"}  
fruit_set3 = {"pomegranate", "apple"}  
fruit_set1.difference(fruit_set3)
```

We will get the following output:

```
{'mango', 'grapes'}
```

- **copy()**: It copies all element from one set to other. Let's see an example:

```
fruit_set1 = {"apple", "grapes", "mango"}  
fruit_new_set = fruit_set1.copy()
```

We will get the following output:

```
{'mango', 'apple', 'grapes'}
```

- **clear()**: It deletes all set elements. Let's see an example:

```
fruit_new_set = {"apple", "grapes", "mango"}  
fruit_new_set.clear()
```

We will get the following output:

```
set()
```

- **remove()**: It deletes only a given element. Let's see an example:

```
fruit_set2 = {"pineapple", "banana"}  
fruit_set2.remove("pineapple")
```

We will get the following output:

```
{'banana'}
```

- **intersection()**: It checks and returns common elements from both. Let's see an example:

```
fruit_set1 = {"apple", "grapes", "mango"}  
fruit_set3 = {"pomegranate", "apple"}  
fruit_set1.intersection(fruit_set3)
```

We will get the following output:

```
{'apple'}
```

- **isdisjoint()**: It will checks into both sets and if it is not found any common element from sets then it return **True** else **False**. Let's see an example:

```
western_area_set = {"andheri", "bandra", "malad"}  
central_area_set = {"kalyan", "Thane", "kurla"}  
western_area_set.isdisjoint(central_area_set)
```

We will get the following output:

True

- `issubset()`: It will check that `set1` is part of `set2`. If yes then it returns `True` else `False`. Let's see an example:

```
mumbai_area_set = {"andheri", "bandra", "malad", "kalyan", "Thane", "kurla", "Ghatkopar"}
```

```
central_area_set = {"kalyan", "Thane", "kurla"}
```

```
central_area_set.issubset(mumbai_area_set)
```

We will get the following output:

True

- `pop()`: It removes the arbitrary element from the set. Let's see an example:

```
mumbai_area_set = {"andheri", "bandra", "malad", "kalyan", "Thane", "kurla", "Ghatkopar"}
```

```
mumbai_area_set.pop()
```

We will get the following output:

```
{'malad', 'Thane', 'Ghatkopar', 'kalyan', 'bandra', 'kurla'}
```

Additional topic – PEP8 style guide for Python code

PEP8 is a document that provides guidelines for beginners to write code as per the Python coding convention. It increases code readability to for developers and others as well.

Why code readability is important?

As *Guido van Rossum* said, *Code is read much more often than it is written*. We may spend more time to write a piece of code. Once it is completed, we may never write it again but we read that code many times. A small part of the code is part of project or module so when we go back to code, it is always required to recognize why we have written this code or what is the use of that code. Therefore, code readability is always required. There are a few points which we should always follow for coding conventions.

Naming conventions

Whenever we define a variable, using sensible names will save time while reading code. This goes for function, class, module, constant, method, and package name as well. Now, let's see how to choose the right name.

Let's see the different cases which are as follows:

- **Case-1:** For variable, if we have to take variable to store client-ID then,
`x1 = "emp001" # it is not recommended.`
`client_id = "emp001" # recommended.`
- **Case-2:** For class and method, if we have to process on e-commerce website then the class name should be,

```
# Not Recommended.  
  
class A(object):  
    def abc():  
        # Function for selling order.  
  
  
    def xyz():  
        # Function for buying order.  
  
  
# Recommended.  
  
class Order_management(object):  
  
    def selling_order():  
        # Function for selling order.  
  
  
    def buying_order():  
        # Function for buying order.
```

Python indentation

It is a common mistake that is done by a beginner. Instead of using a tab, we should use the space key and it requires the four-space for scope. Please refer to the following example:

```
# Not Recommended  
  
def dispatch_status():  
if dispach_token:  
print("Order is dispatched successfully.")  
else:  
print("Order dispatch is under process. ")
```

```
# Recommended.

def dispatch_status():
    if dispach_token:
        print("Order is dispatched successfully.")
    else:
        print("Order dispatch is under process.")
```

Maximum line length

Generally, 79 characters are good enough for a single line of code. It becomes easy to read lines and Python supports multiple statements reading.

```
# Recommended.

def employee_details(emp_name, emp_department,
                     emp_salary, emp_experience):
    return employee
```

If it is impossible to use implied continuation, then we can use backslashes to break lines:

```
# Recommended
from department import technology, management, support, marketing
```

```
# Recommended
salary_slip = (HRA + medical allowance + other allowance - EPF)
```

Block and inline comments

In coding, comments are very important for readability, if the developer spends time and chooses the right words for comments, it saves the reader's time. Please refer to the following example:

```
# Recommended

studen_name = "Raj"
# Function use for student search from database.

def student_result_seacrh(stud_name):
    for data in student_database: # Using For loop to searching data.
        if stud_name == data: # Statement for comparison
            print data
```

Document strings

In Python, we have seen single line comment and inline comment to explain functionality. It is possible to write multiple lines of comments in Python. It is defined as ("") or ('') and the same closing at the end of a comment. With the help of that, we can describe the functionality of the code, block, function, or program.

Please refer to the following example:

```
# Recommended

''' This program is created for addition of two values.
In that we already given two value and defined function sum.
In result it will return sum of both value. '''

# Sum function definition
def sum():
    value1 = 5
    value2 = 10
    return value1 + value2 # it return result.

sum() # calling sum function.
```

It is a fact that empty sequences are false when we use if statements.

When we use `if` statement there is always a check whether a list is empty, a variable is null or contains the value, value type or anything else. For example, if we have to check variable type is `int`, variable is not null and length of the `int` is 10 digits then go inside.

```
# Not Recommended

mobile_num = 9999999801
if not len(mobile_num) == 10 and type(mobile_num) == type(1):
    print("It is valid value", mobile_num)

# Recommended

mobile_num = 9999999801
# checking for value that is it null or not.
if mobile_num is not None:
    # checking the length of the mobile and type of the value.
```

```
if len(mobile_num) == 10 and type(mobile_num) == type(1):
    print("It is valid value", mobile_num)
```

Imports

Usually, we should remember that when we import multiple packages from a different source, it should be in a different line. Please refer to the following example:

```
# Not Recommended
```

```
import sys, os
```

```
# Recommended
```

```
import sys
```

```
import os
```

Connection closing

We have to always remember that when we open any file from the source and fetch or read data from that, after completing the process, we should close that file. It provides better performance for the program. One more place when it is required to close the connection after completing the work, is database connectivity. If we let all connections to a database to remain open, it will increase the load.

List comprehension

It is a powerful concept of Python. In list comprehension, we can write multiple statements into a single line. It is faster than for loop and improves the performance of our code. Please refer to the following example:

```
# Normal example of list comprehension.

list1 = [] # empty list

# loop for adding element into list.

for value in range(1,11):
    list1.append(value)

print(list1)
```

We will get the following output for the above code snippet:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Let's see a list comprehension example:

```
list2 = [value for value in range(1,11)]s

print(list2)
```

We will get the following output for the above code snippet:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Let's see one more example of list comprehension with `if` statement:

```
# Example for printing list which has only even value between 1 to 20.  
even_lits = [] # empty list  
for even_val in range(1,21):  
    if (even_val%2 == 0):  
        even_lits.append(even_val)  
print(even_lits)
```

We will get the following output for the above code snippet:

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Let's see an example of list comprehension for the same scenario:

```
even_lits = [even_val for even_val in range(1,21) if(even_val%2 == 0)]  
print(even_lits)
```

We will get the following output for the above code snippet:

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Iterators

A set of statements which execute repeatedly using either a recursive function call or a loop is called iterators. In Python iterator element or object returns one element at a time. In technical terms, Python iterator object always implements with two methods, `iter()` and `next()`, sequentially called the iterator protocol.

Let see an example of iterators:

```
>>> iter_var = iter(range(1,6))  
>>> print(next(iter_var))  
1  
>>> print(next(iter_var))  
2  
>>> print(next(iter_var))  
3  
>>> print(next(iter_var))  
4  
>>> print(next(iter_var))
```

```
5
>>> print(next(iter_var))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

In the given example `iter()` is for defining element sequence and `next()` is for calling the next element. If we call `next()` again when a new element is not available then it gives `StopIteration` error and stops. It is in sequence but it does not print another element until we call the `next` method. That is the difference between iterators and a `for` loop.

The basic difference between for loop and iteration is that "for loop" execute without interruption until it's all element end, but iteration start the execute and it will hold the execution till next() method calls.

Generators

Generators are like iterators, but are easier than iterators because they don't use `iter()` and `next()` function for calling the sequence. It works with the `yield` statement. When we call the `yield`, it changes the value and goes back to the loop again.

Tip: late binding is a concept behind the generator and the iterator. That means they are idle until you ask it for a value. After producing a single value, they turn it idle again. It is a helpful approach to work with a huge amount of data. If our data is huge and we have a limited amount of memory then there's no need to load all the data in the memory at the same time, we can use a generator or an iterator which will pass us single pieces of data at a time.

Generator creation in Python?

It is very simple to create a generator in Python. Create any function and instead of using `return` we should use `yield` and at the time of execution, it is treated as generators.

The difference between both of them is that the `return` statement terminates the function execution. The `yield` statement calls and stays idle until it gets the call again and in the next call, it continues from there on.

Let's take an example to understand generators. Nowadays everyone listens to music by using a phone app or web application. We are taking the same example:

```
# Function for music player
def music_player():
```

```
# defining that i have playlist which have only 5 songs.  
playlist = range(1,6)  
# Creating for loop to give choice to user.  
for song in playlist:  
    print("Playing the song no: ",song)      # print message to know  
which number song is playing.  
    yield song  
    # This will provide the functionality to user to change the new  
song.  
  
# defining the object of music_player().  
>>> next_song = music_player()  
# next method for change the song.  
>>> next(next_song)          # calling the first time.  
  
# Output:  
Playing the song no:  1  
1  
  
# calling the second time.  
>>> next(next_song)  
  
# Output:  
Playing the song no:  2  
2  
  
# calling the third time.  
>>> next(next_song)  
  
# Output:  
Playing the song no:  3  
3
```

```
# calling the fourth time.  
>>> next(next_song)  
  
# Output:  
Playing the song no: 4  
4  
  
# calling the fifth time.  
>>> next(next_song)  
  
# Output:  
Playing the song no: 5  
5  
  
# calling the sixth time.  
>>> next(next_song)  
  
# Output:  
# It threw exception because playlist having only 5 songs, so it stopped  
on sixth attempt.  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
StopIteration
```

In the above code, with the `music_player()` function, we capture a small module of music player, as per function, it's functionality is user can play the music from the playlist. Users have the option to change the song from the playlist.

In the above code, every required statement has comments, which can help the reader to understand the code better. Basic flow of the code is that, user opens the `music_player()` (in code it called the `music_player()` function), then he/she opens the playlist and plays the first song (at the time of playing the first song, control goes to for loop then it goes to `yield` statement, `yield` statement is executed then control goes back to user). When user presses the next button (in the code it calls `next(next_song)` method), it plays the next song from the playlist.

Conclusion

Python is powerful and easy to learn language. This chapter covered all commonly used topics. You learned the fundamentals of Python, its data type and data type functionality with examples. It also covered the standards of Python code writing and how important it is to follow these standards. Some strong concepts of Python were covered which make Python more usable for projects.

In the next chapter, you will gain the knowledge of object oriented programming language concept with Python. Nowadays, everyone knows about OOPS concept and Python strongly supports and works flexibly with oops concepts.

Questions

- 1 What is Python?
- 2 How to define local and global variable?
- 3 What is the difference between local and global variable?
- 4 How to define function?
- 5 What is the use of `type()` method?
- 6 What are the available data structures in Python?
- 7 What is tuple?
- 8 What is dictionary and how to access them?
- 9 Can we update dictionary keys? If yes then how?
- 10 What is list and give the name of list built-in function?
- 11 What is the difference between list and tuple?
- 12 What is mutable and immutable?
- 13 What is set?
- 14 Give the set built-in method name?
- 15 What is list comprehension?
- 16 How to define list comprehension?
- 17 What is iterator?
- 18 What is generator?
- 19 What is the difference between iterators and generators?
- 20 How to define and call the generators?
- 21 Give a real life example of generators.

CHAPTER 2

Major Pillars of OOPS with Python

Python is an **object-oriented programming (OOP)** language. It also supports many features of OOPS. This chapter covers all OOP concepts, which are available in Python. We included this chapter in the book, because OOP is a very important part of every object-oriented language. In this chapter, we have tried to explain the OOP concept with the simplest terms and real-world examples.

Structure

- Introduction to object-oriented programming (OOP) language
- Basic idea of object
- What is class?
- Decorators in Python
- Introduction of class method and static method in Python
- Inheritance
 - Single inheritance
 - Multiple inheritance
 - Multilevel inheritance
- Encapsulation
- Polymorphism
- Method overloading

- Method overriding
- Abstraction

Objective

At the end of this chapter, you should easily understand the object, classes, class variables, static method, class methods, major pillars of OOPs, and decorators.

Introduction to object-oriented programming (OOP) language

Earlier, while writing code programmers used to face the following difficulties:

- Code duplication
- less security of the code
- code following the procedural programming
- Difficulty to manage long codes
- less readability of code due to lack of relevance to the real world

To resolve these issues, OOP concepts were driven. At first, OOP was introduced with the C++ language. After the success of OOP, most of the programming languages include this functionality.

Mentioned below are the main features of the OOPs concept or we can also call them the major pillars of Python:

- Object
- Class
- Inheritance
- Encapsulation
- Polymorphism
- Abstraction

Basic idea of object

This is the very common definition of object, which is available on the internet: The object is a real-world entity, which has attributes and behavior. For example, a car is an object which has attributes such as color, brand, tiers, seats, speed, and more.

What is class?

It is the collection of objects, functions, and statements. We can also say that it is a user-defined structure, which has collective data related to the class. For example, we have class `Engineering_college` and it has an attribute like a stream (IT, CS, EC, Mechanical, Civil), fee collection department, labs, faculty rooms, and more.

The class can be defined anywhere in a program, but they are usually defined in the beginning just after the import statements. As a standard practice class name's first letter should be capital. In the below code, we create a class named `Aircraft`. To access the class and its attribute, we need to create an object of the class that is also known by the name instance. For example:

```
class Aircraft:
    print("Welcome to class")
```

Let's see an example of creating `Cars` class instance:

```
aircraft_object = Aircraft()
```

It is very important that every time when we define the instance of the class, it should refer to the same class, which means a class name is case sensitive in front of the class instance. For example, we created the class `Aircraft()` and defined the object as " `aircraft_object = Aircraft()` ".

We can also add new data to an instance by dot notation, which is shown below:

```
>>> aircraft_object.c1 = 35
>>> aircraft_object.c2 = 50
```

In the same way we can access class variable and also assign the value.

Method term use in a Python class: The functions which are defined inside the class are called method. I am introducing this word right now because we will use this term in other topics.

Here's an example to understand class and instance:

```
# Creating the class and calling by instance.

class Test:
    var1 = 3
    print("Calling from inside the class: ",var1)

# Create first class instance.
t1 = Test()
print("Calling class variable threw first instance the class: ",t1.var1)
```

```
print("class variable id: ",id(t1.var1))
# Assigning value to the instance.
t1.var1 = 5
print("Calling after assigning the value: ",t1.var1)
print("outside variable id: ",id(t1.var1))
# Create a second instance of the class.
t2 = Test()
# print the value of class variable with t2 instance.
print("Calling class variable threw second instance of the class: ",t2.var1)
```

We will get the following output for the above code snippet:

```
Calling from inside the class:  3
Calling class variable threw first instance the class:  3
class variable id:  10910464
Calling after assigning the value:  5
outside variable id:  10910528
Calling class variable threw second instance of the class:  3
```

In the above example, we created a class `Test` and inside the class, we defined the class variable value. We also created an instance of the class `Test()` and printed the value of `t1.var1`. With the help of instance, we can access the class variable. If you look carefully `t1.var1` is printing 3, 3, 5 and 3. Do you think it is magic or it is a program compilation issue, but in reality, it is neither magic nor compilation issue. It is happening because of variable scope.

In Python, we can define two variables with different values and the same name in different scope. We did the same thing here `var1` variable is declared with value 3, Inside the class as a global variable then we print the `var1`. So as an output it prints the value 3 with a message for the first time. We created the instance then called the `var1` variable value, so it refers to the class variable and prints 3. Then, we created the new object `t1.var1` with value 5, and when we call the print `t1.var1`, it shows 5 because `t1.var1` scope is changed.

After that, we create a new instance for `Test` class and call the `t2.var1` that refers to the class variable. Its output is 3 again.

- **Python allows you to create many instances of a single class.**
- **Any variable that is defined inside the class is called a class variable, such variables can be called outside of the class with the help of class objects.**

In the Python there are so many predefined keywords available, `self` is one of them. This keyword is commonly used in class. It is an object of the class that is used for accessing the attributes and method of the class. Python creates the standard for a method that should be defined with `self` keyword and `self` should be the first parameter.

It is a predefined and reserved method for the class. It is a constructor method in Python as per object-oriented programming. The `init` method is called automatically and suddenly after the declaration of the class instance. In simple terms, when a programmer creates the instance of a class, that internally calls the `init` method. It is a coding convention that `init()` method is defined with parameter and that first parameter is `self`.

The following code block is used to explain the `__init__` method and `self` keyword:

```
# create the class Cal for calculation.  
# It is like a calculator.  
  
class Cal(object):  
    # definition of __init__ method.  
    def __init__(self, val1, val2):  
        # variable initialization.  
        self.val1 = val1  
        self.val2 = val2  
  
    # This function is for addition.  
    def sum(self):  
        print(self.val1 + self.val2)  
  
    # This function is for Subtraction.  
    def sub(self):  
        print(self.val1 - self.val2)  
  
    # This function is for Multiplication.  
    def prod(self):  
        print(self.val1 * self.val2)
```

```
# Create the instance of cal class.  
c1 = Cal(20,10)  
# calling the functions in sequence.  
c1.sum()  
c1.sub()  
c1.prod()
```

We will get the following output for the above code snippet:

```
30  
10  
200
```

In the above mentioned example, we took the calculator example here, which has addition, subtraction, and multiplication functionality. `__init__()` constructor and it initialized the variable `val1` and `val2`. After initialization we used it into `sum()`, `sub()` and `prod()` function.

- If any variable is defined inside the class these are called class variable.
- If any method is defined inside the class these are called class method.
- To access class variable and method it requires an instance that is `self`.

Decorators in Python

Decorators are a very useful concept. We can change the function behavior without changing the function definition explicitly. That means we have a function which is used in many places, but in any specific condition, the user may want its behavior to change. For that requirement, we cannot change the existing function definition because it will reflect into every place, wherever we call that function. So we can solve this problem with the decorator. In that situation we will create the new function and link that function with the previous function. When a user required condition happens, it will change the previous function behavior.

Let's go *step-by-step*. Following steps are used to define the decorators:

1. **Normal function creation:** The following code block is the example of function:

```
# function for subtracting two values.  
def sub(v1,v2):  
    print(v1-v2)  
  
sub(35,20) # calling the function.
```

2. **Creating the decorator:** It is similar to a normal function. The difference between decorator and function is that it defines again another function inside the function. Here's the example of the decorator function:

```
# First function.

def sub_decorat(test): # Highlighted point - 1

    # Second function.

    def t1(var1,var2): # Highlighted point - 2
```

Highlighted point -1 means always define the function with one argument. That argument is going to take a function, which will call suddenly after the decorator call. For example:

```
@ test_decorator

def sum(v1 ,v2)
```

So, in that example sum(v1, v2) is going to pass implicitly as an argument into test_decorator decorator function.

Highlighted point -2 means after sum(v1 , v2) is processed into argument, we have to create a new function with the same signature of sum to process with function. So that is the reason why we create a new function inside the decorator function.

3. Write your condition of business logic inside the second function:

```
# definition of decorator.

def sub_decorat(test):

    # definition of t1 function, which should be a replica of
    sub(v1,v2) function.

    def t1(var1,var2):

        # checking that var1 < var2, if true then swap the value
        and return the new values to the function else return the same.

        if var1 < var2:

            var1, var2 = var2, var1

            return test(var1,var2) # Highlighted point - 1

        else:

            return test(var1,var2)

    # it should return the same name as function.

    return t1 # Highlighted point - 2
```

Highlighted point -1 means that in the current scenario we have swapped value to the function, so in return that decorator object name with the signature of the same function is received.

In the first function return (i.e. t1()), the second function name should be a pass but without brackets.

4. Linking and calling the decorator:

```
@sub_decorat # Highlighted point - 1  
def sub(v1,v2):
```

Highlighted point -1 means that we can call decorator in that way. If we call any decorator in a program in the above function, then the interpreter understands that we want to link this function with the decorator.

The following example helps to understand the decorator with all steps:

```
# Program for subtracting two values.  
"""
```

We have one function for subtraction. If the user wants that first value of the function,

should be higher than the second value, but it should be managed by the developer.

that user can give an order of value but it always subtracts from higher value.

```
"""
```

```
# definition of decorator.
```

```
def sub_decorat(test):
```

```
    # definition of t1 function, which should be a replica of sub(v1,v2)  
    function.
```

```
    def t1(var1,var2):  
        # checking that var1 < var2, if true then swap the value else  
        return the same.
```

```
        if var1 < var2:  
            var1, var2 = var2, var1  
            return test(var1,var2)  
        else:  
            return test(var1,var2)
```

```
# it should return the same name as function.
```

```
    return t1

@sub_decorat
def sub(v1,v2):
    print(v1-v2)

# calling the sub function.
sub(15,20)
```

We will get the following output for the above code snippet:

```
5
```

This example shows that if the user gives an order of value at the time of calling function, it will be handled into the program.

Let's use decorator with real world example:

```
""" This program is based on the shopping mall,
```

For example there is one shop which has 15 items and only item5 has a discount of 20 %.

If any customer takes item5 then he will get a discount of 20 % on that item MRP.

So we have one function `order_total()` for calculating the total amount, which is going to be used globally.

One item from the list has a discount price, so which is going to be calculated by the decorator?

```
"""
```

```
# This is the list of items in the format of a dictionary.
order_list = {'item1': 200, "item2": 1000, 'item3': 500, "item4": 300,
'item5': 2000}

# This is the decorator definition.

# In the discount_order(dist) function, dist is the order_total(odr)
function.

def discount_order(dist):
    # disc_cal(val) is the same replica of order_total(odr).
    # It is always the same as the passed function, which is parsed into
```

decorator.

```
def disc_cal(val):
    # For accessing the dictionary key and value.
    for key,value in val.items():
        # for accessing the value of "item5" and assign into val1.
        val1 = val.get('item5')
        # check for val1.
        if val1 is not None:
            # for calculating the 20% of the value.
            val1 = val1 - (val1 * 0.20)
            # It is reassigning the val['item5'] value.
            val['item5'] = val1
            # It is returning the new updated dictionary to the
            order_total(odr) function.
        return dist(val)
    else:
        return dist(val)
return disc_cal

# calling of decorator @discount_order
@discount_order
# normal function for total calculation.
def order_total(odr):
    amount = 0
    # For accessing the dictionary key and value.
    for key,value in odr.items():
        amount = amount + value
    # for printing the final amount.
    print("Final amount: ", amount)

# calling the order_total function.
order_total(order_list)
```

We will get the following output for the above code snippet:

Final amount: 3600.0

The above mentioned example is based on a shopping center bill generation machine. In the above code, all required comments are mentioned. The shopping center, in our example, has 15 items and only `item5` has a discount. The `order_total()` function is used for generating customer bills. The `discount_order()` function is decorator, which is used for discount calculation.

Introduction of class method and static method in Python

In Python, to define the class method and static method, we have to use Python built-in decorators, which are `@classmethod` and `@staticmethod`. Both are different from each other and have different uses.

The class method is defined inside the class. It is different from other methods because it takes the `cls` object instead of self-object. That means it is always bound with classes indirectly through object `cls`. Class method first argument should be `cls` object. In the previous method example, we defined the `@classmethod` decorator. We can change the class state because it takes the class as a parameter.

The static method is also defined inside the class, but the difference is that we can define methods without argument but not object. It is part of the class but cannot access or modify through the class object. In the above method example we defined `@staticmethod` decorator.

Python does not support method overloading like other OOP languages, so to achieve method overloading we can use the class method.

The following example is used to show how class method and static method work:

```
# This is an example of class method and static method.

class Employee:

    # It is a constructor method.

    def __init__(self, name, department):
        self.name = name
        self.department = department

    # It is an example of a class method to create an employee object by
    # technology.

    @classmethod
```

```
def tech_detail(cls, name, technology):
    return cls(name, technology)

# It is an example of a static method to check whether an employee is
available into db or not.

@staticmethod
def check_into_db.empid):
    emp_db = ['emp0010', 'emp0011', 'emp0012', 'emp0013', 'emp0014']
    if empid in emp_db:
        print("This employee is part of our organization.")
    else:
        print("This employee is new to our organization.")
    return "Execution completed."

emp_obj1 = Employee('Shayank', 'Technology')
emp_obj2 = Employee.tech_detail('Shayank', 'Python')

print(emp_obj1.department)
print(emp_obj2.department)

# print the result
print(Employee.check_into_db('emp01'))
```

We will get the following output for the above code snippet:

Technology

Python

This employee is new to our organization.

Execution completed.

In the above mentioned example, we create a class method and static method. In the output when we call `emp_obj1.department`, it prints technology. The second time, calling `emp_obj2.department` prints Python as an output, which is different from the first. To understand this better let's have a look at the code again. In the `tech_detail()` method, we return the `cls`, which overrides the class behavior. In the call of static method, we check if the employee is available into a list or not.

Inheritance

Inheritance means code reusability. In technical terms, it is writing code into one class and then instead of writing again reusing that code in another class. This concept is developed after the C programming language. In programming, often a coder has to use some code blocks in many places. To reduce code redundancy we use inheritance. There are some different types of inheritance, which are commonly used in Python:

- Single inheritance
- Multiple inheritance
- Multilevel inheritance

To understand inheritance better, there are few terminologies that are commonly used inside the class and are important to understand. Let's see them:

- **Base class:** In programming, when we define the attribute in the first class and then use it again into the second class, the first class is called the base class.
- **Drive class:** Form the above example, the second class where we use the defined attributes are called drive class.

Single inheritance

In single inheritance, we inherit the property of the base class and use them into drive class. In that scenario, we fetch the property from a single base class. Let's see the example of single inheritance:

```
# Python code to demonstrate how parent constructors
# base class
class mouse(object):
    # __init__ is the constructor
    def __init__(self, mouse_quantity):
        self.mouse_quantity = mouse_quantity

    def product_availability(self):
        print("Mouse Quantity from mouse class: ",self.mouse_quantity)

# drive class
class product_stock(mouse): # we are referencing the base class in
brackets.
```

```
# Init method for product stock class.

def __init__(self, mouse_quantity, keyboard_quantity, display_
quantity):
    self.keyboard_quantity = keyboard_quantity
    self.display_quantity = display_quantity
    # calling the __init__ of the parent class
    mouse.__init__(self, mouse_quantity)

# function for displaying the available stock quantity.
def product_stock_details(self):
    print("Mouse Quantity: ",self.mouse_quantity)
    print("Keyboard Quantity: ",self.keyboard_quantity)
    print("Display Quantity: ",self.display_quantity)

# creation of an instance
prod_object = product_stock(200,300,400)

prod_object.product_availability()
prod_object.product_stock_details()
```

We will get the following output for the above code snippet:

```
Mouse Quantity from mouse class:  200
Mouse Quantity:  200
Keyboard Quantity:  300
Display Quantity:  400
```

In the above example, we inherited the mouse class attribute `mouse_quantity`. In product stock drive class we reuse them again. `mouse.__init__(self, mouse_quantity)` by this way we called the base class `mouse_quantity`.

Multiple inheritance

Python supports multiple inheritance, if we compare to Java, it is not possible directly. To resolve this problem they introduce the interface. In multiple inheritance drive class fetches the property from two different classes.

The following is an example of multiple inheritance:

```
"""
```

This is a common scenario for now days. If anyone wants to open a bank account,

so for eligibility it requires aadhar card and pan card. For that i used Multiple inheritance.

"""

```
# Multiple Inheritance example.
```

```
# Base class1
```

```
class Pan_card(object):
```

```
    def __init__(self):
```

```
        self.pan_number = "Pan0000001"
```

```
        print("From Pan_card class")
```

```
# Base class2
```

```
class Adhar_card(object):
```

```
    def __init__(self):
```

```
        self.adhar_number = "5000000000000001"
```

```
        print("From Adhar_card class")
```

```
# Drive class
```

```
class Bank_account_opening(Pan_card, Adhar_card):
```

```
    def __init__(self):
```

```
        # Calling constructors of Base class1
```

```
        Pan_card.__init__(self)
```

```
        # Calling constructors of Base class2
```

```
        Adhar_card.__init__(self)
```

```
        print("From Bank_account_opening class")
```

```
    def cutomer_details(self):
```

```
        print("Pan_number: ",self.pan_number)
```

```
        print("Adhar_number: ",self.adhar_number)
```

```
bank_obj = Bank_account_opening()
```

```
bank_obj.customer_details()
```

We will get the following output for the above code snippet:

```
From Pan_card class
From Adhar_card class
From Bank_account_opening class
Pan_number: Pan0000001
Adhar_number: 5000000000000001
```

In the above example, `Pan_card` and `Adhar_card` class attributes are reused in the new class that is `Bank_account_opening`. This is a simple example of multiple inheritance. With this line, we inherit both classes: `Bank_account_opening(Pan_card, Adhar_card)` classes inside the `bank_account_opening`.

In the output, you can see that when we called the `Bank_account_opening` instance, it called the base `class1` then base `class2` and in the end it prints drive to class.

Multilevel inheritance

In multilevel inheritance, drive class reuses the attribute of the base class and we also fetch the property of drive class into another drive class. In simple words, on level one we define the attribute and on the second level we reuse that property. After that, on the third level, we use the second level and first level properties into third class. That kind of structure is called multilevel inheritance. We can perform inheritance on many levels, it is not bound with only three levels.

Following is the example of multilevel inheritance:

```
"""
This program is based on the leave management of the company.
When an employee applies for leave, it goes to the manager and after that
hr processes with holiday.

This is an example of multilevel Inheritance.

"""
# definition of junior class.
class Junior_employee(object):
    # Constructor
    def __init__(self, leave_status):
        self.leave_status = leave_status

    def leave_application_status(self):
```

```
        return ("Junior employee leave application status: ",self.leave_
status)

# definition of manager class.

class Manager_employee(Junior_employee):
    # Constructor

    def __init__(self, leave_status, approval_status):
        # Calling constructors of Junior_employee class.
        Junior_employee.__init__(self, leave_status)
        self.approval_status = approval_status

    def leave_approval_status(self):
        return ("Manager approval status on junior employee leave:
",self.approval_status)

# definition of hr class.

class Hr_employee(Manager_employee):

    # Constructor

    def __init__(self, leave_status, approval_status, leave_processing_
status):
        # Calling constructors of Manager_employee class.
        Manager_employee.__init__(self, leave_status, approval_status)
        self.leave_processing_status = leave_processing_status

    def leave_deduction_status(self):
        return ("Hr processing status on junior employee leave: ",self.
leave_processing_status)

# object created for Hr_employee class.

hr_emp_obj = Hr_employee("Applied", "Approved", "Processed")
print(hr_emp_obj.leave_application_status(), hr_emp_obj.leave_approval_
status(), hr_emp_obj.leave_deduction_status())
```

We will get the following output for the above code snippet:

```
('Junior employee leave application status: ', 'Applied')
('Manager approval status on junior employee leave: ', 'Approved')
('Hr processing status on junior employee leave: ', 'Processed')
```

In the above example, we reuse the property of `Junior_employee` class into `Manager_employee` class. Then we inherit the property of `Junior_employee` and `Manager_employee` class into `Hr_employee`. In a sequential way `Junior_employee` property is used inside the `Manager_employee` and then both class properties are used in the `Manager_employee`.

Use of `super()` function

This is a built-in function, it calls the base class implicitly and the benefit of super function is that we don't need to define the base class name at the time of calling. The following example is to show `super()` function use case:

```
# This is the class of plan detail.

class Plan_detail():

    def __init__(self, data_speed, usage_limit):
        self.data_speed = data_speed
        self.usage_limit = usage_limit


# This is the class for broadband numbers.

class Broad_band_ABC(Plan_detail):

    def __init__(self, data_speed, usage_limit, model_name):
        # using super functions in place of class names.
        super().__init__(data_speed, usage_limit)
        self.model_name = model_name


internet_obj = Broad_band_ABC('50Kb/s', '100GB', 'Broadband01234')
print('This plan data speed limit is', internet_obj.data_speed)
print('This Plan has usage limit of', internet_obj.usage_limit)
print('Broad band model name is', internet_obj.model_name)
```

We will get the following output for the above code snippet:

```
This plan data speed limit is 50Kb/s
This Plan has usage limit of 100GB
```

Broadband model name is Broadband01234

In the above example, we did a single inheritance and we use the super function for calling the base class.

Encapsulation

Encapsulation hides the data from the outer world. In Python, we can hide the variable and method by using the underscore symbol, which are double "_" and "__" double.

The following is an example of encapsulation:

```
# Example of encapsulation.

class User_authentication:
    # We defined the constructor.

    # calling __userAuth(), because we can call only inside the class.

    def __init__(self):
        self.__userAuth()

    def action_page(self):
        print('User authentication process is completed.')

    # definition of __userAuth() method.

    # It's access scope is only inside the class.

    def __userAuth(self):
        print('User validation process is going on....')

user_obj = User_authentication()
user_obj.action_page()
```

We will get the following output for the above code snippet:

```
User validation process is going on....  
User authentication process is completed.
```

If call the __userAuth() method then,

```
>>> user_obj.__userAuth()
```

We will get the following output:

```
Traceback (most recent call last):
```

```
File "encap.py", line 18, in <module>
    user_obj.__userAuth()
AttributeError: 'User_authentication' object has no attribute '__userAuth'
```

In the above example, if we define any function with "__" then it cannot be accessed from outside the class. In the next call which is `user_obj.__userAuth()`, we tried to call the method but it gives an error as an output. So, in that example, we created one method `__userAuth()`, which is encapsulated and which is not accessible from outside.

Polymorphism

In polymorphism one function can behave in different ways as per requirement. For example, we have a glass, which can be used as a container, as a vase or it can be used as a weapon to protect ourselves. In OOP there are two types of polymorphism.

Method overloading

It is the concept where we define the same method more than once with different data types, so at the time of execution, the compiler finds the method on a base of the parameter.

Python does not support method overloading, because Python uses dynamic allocation of data type to the variable.

Method overriding

It is the concept where we write the class more than once. In that case, one method overwrites the other that is depending upon the calling sequence.

Here's the example of method overriding:

```
# Class for printing string messages on console.
class Str_msg:

    def show_message(self):
        print("show_message() method from class Str_msg")

# Class for printing Integer value on console.
class Int_msg(Str_msg):
```

```
def show_message(self):  
    print("show_message() method from class Int_msg")  
  
obj_int_msg = Int_msg()  
obj_str_msg = Str_msg()  
obj_int_msg.show_message()  
obj_str_msg.show_message()
```

We will get the following output for the above code snippet:

```
show_message() method from class Int_msg  
show_message() method from class Str_msg
```

In the above example, we created two classes with a different name and defined the two different class instances. We created the same name function in both classes.

In this example, it shows that the same function works differently in different places.

Abstraction

In this concept, data is hidden from the user and only the required part is shown. For example, the Facebook login page is a very common real-life example which provides the username, password field, and submit button to the user for logging into Facebook, but how it works behind the login page is not visible to a user. This is a good example of abstraction.

If we look at a technical example, there is one more live example of abstract classes, which we saw many times. In the `print()` function, how to call `print` built-in function and what are the parameters of `print`. Only this functionality is important to know the program. How it is executed in the background is not necessary.

In Python, for abstraction, we use abstract class which can be designed as per user requirement.

Abstract class

It is a different kind of class, where we create the blank class into base class. Then in the drive class we give the functionality to those classes. Let's see an example for better understanding.

If we want to use an abstract class concept, then it is not allowed to be used directly in the Python programming language. It has a special module to achieve an abstract class that is ABC. An example is classes (ABC).

To make class abstract we use ABC inside the round bracket and for method, we use decorated which is as `@abstractmethod`.

Let's see the example of abstract class:

```
# importing the ABC module and abstractmethod.  
from abc import ABC, abstractmethod  
  
# example of abstract class  
class Test_abstract_base_class(ABC):  
    # It is an example of an abstract method.  
    @abstractmethod  
    def show(self):  
        print("It is declaration of abstract class.")  
  
class Abstract_drive_class(Test_abstract_base_class):  
    def show(self):  
        super().show() # referring to the base class show.  
        print("It is implementation of abstract class.")  
  
obj_abstract = Abstract_drive_class()  
obj_abstract.show()
```

We will get the following output for the above code snippet:

It is a declaration of abstract class.

It is an implementation of abstract class.

Conclusion

Generally, people think OOP concepts are difficult to understand but my personal view is that if any reader gives some extra attention to understand, then they can understand easily and to recognize these concepts for a long time, the reader should relate these topics with real life. After reading this chapter, the reader can easily relate the OOP concept with real life.

If you read the previous chapter followed by this chapter that means, you have a good idea about code and its flow. Also, we have covered many concepts of Python.

In the next chapter, we will start with Django. It is a web framework of Python, which requires basic knowledge of Python that is already covered.

Questions

1. What is the class?
2. What is an object?
3. What is the class method?
4. What is self and why is it used?
5. How to define a class and call the class?
6. What is `__init__()` method?
7. What is the use of `__init__()` method?
8. What is a static class?
9. What is a class variable?
10. What are decorators?
11. Give the real-life example of decorators?
12. Give the name of built-in decorators?
13. What is inheritance?
14. Type of Python supported inheritance?
15. Is multiple inheritances supported by Python and if yes then how?
16. What is encapsulation?
17. What is abstraction?
18. What is the difference between encapsulation and abstraction?
19. How data hiding is achieved in Python?
20. Types of polymorphism?
21. Which polymorphism is not supported Python and why?
22. What is the use of super keywords?
23. What is an abstraction class?
24. What is the abstraction method?

CHAPTER 3

Getting Started with Django

Django is a Python web framework that is used to develop web applications, restful APIs, and website development. Django framework's base language is Python. So if you are not aware of Python or you want to brush up on Python then please refer to the previous chapters, which are used to explain Python. This chapter is included in this book because microservices require the knowledge of Django, so it is an important chapter for microservices.

We see that nowadays many websites are developed with Django. So if you want to create any website or any APIs then you can use Django. This chapter will cover the basic idea about Django, the architecture of Django, project execution with Django, request handling, and code debugging in Django project.

Structure

- Basics of Django
- Django architecture
- Django execution and basic commands
- Django program execution flow
- How Django handles requests
- Create the view file and configure user defined URL
- URL understanding
- Basic of Loggers and available logger types
- Logger implementation

- Define template in Django
- The powerful concept of Django

Objective

This chapter covers Django from level 0 for readers who are new to Django, please refer to this chapter and practice with examples. The topics which are covered in this chapter are Django default architecture, request and response lifecycle, basic commands of Django, control flow in the application, Django handler, create a view, URL file, and understanding of the templates. At the end of this chapter, you should be able to easily understand the web applications and how to make rest APIs with Django and how the request serves the data on the server and provides the response.

Basics of Django

Django is a Python language based web framework. If any user wants to make web application and web API with managed services then Django is the perfect choice. It is faster than PHP API response and more structured, which makes it easy to handle large projects. Django supports Python which means we can use all the features of Python like objects, class, inheritance, polymorphism, list, tuple, dictionary, and many more.

Its base language is python, so all coding syntax and compilation are similar to Python. Django uses a python interpreter internally to execute the files. We can run the Django files individually on the Python interpreter.

Django architecture

We heard about the MVC architecture design pattern before. The full form of MVC is Model, View, and Controller. It is an architectural pattern that divides the application into three logical components. Every component has different functionality, so every single component is built to handle specific development aspects of an application. That means, all the architecture of the project will be distributed into three parts. Django has few similarities with MVC as it follows the MVT (Model View Template) architecture design pattern. Django project design is divided into three parts which makes it easy to handle bigger project structures.

MVT meaning is as follows:

- **Model:** It is used for data handling like fetching data from database and transferring that data to view.
- **View:** The view part is to work for storing the business logic of the project that communicates with model and template.

= controller

- **Template:** It is the front end part, that means HTML pages or GUI related works come into the template section. It communicates with model and views independently.

Please refer to the following screenshot to see the Django project default architecture:

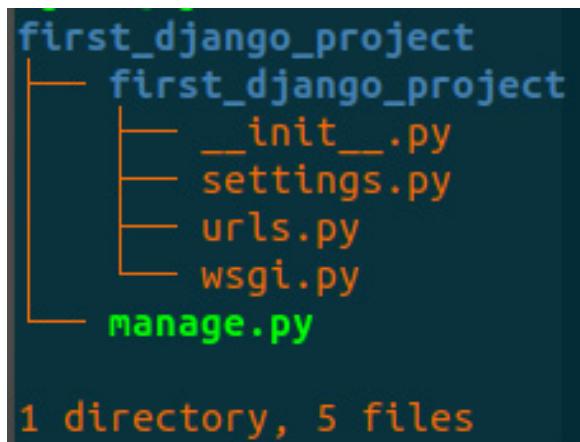


Figure 3.1: Django project architecture

In the above screenshot, we have defined our project name, which is `first_django_project`. All the file and subfolder is created automatically when we create a Django project. In the `project`, there is a subfolder whose name is the same as a project name. The `subfolder` has four files, which are important for project execution.

When we create the Django project, it creates some default files, which are as follows and also shown in *Figure 3.1*. Every file has different meaning and special functionality, let's see a few of them:

- `__init__.py`: In the previous chapter, we saw that `__init__` method is the constructor of the class. It executes when a class instance is created. In Django, `__init__.py` is executed in the first place. We can also use this file in that way, suppose I have some package which is used in many places, then we can define them into that file.
- `settings.py`: It is the main file, which is important for Django project. It is important because all Django project configurations are written here. The following screenshot shows some sections of the `setting.py` file:

```

ALLOWED_HOSTS = []

# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]

```

Figure 3.2:(a): Settings.py file description

- o **Section 1:** In the allowed host we can write an IP address, which we won't make our application listen. If we don't define anything then it is considered as localhost.
- o **Section 2:** In the given section we have to add user-defined app name inside the [] brackets. Django added the few by default application names, which is mentioned to run the project.

The following is the other section of the `settings.py` file:

```

ROOT_URLCONF = 'first_django_project.urls'          → Section 3
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
WSGI_APPLICATION = 'first_django_project.wsgi.application'

# Database
# https://docs.djangoproject.com/en/1.11/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}

```

Figure 3.2 (b): settings.py file description

- o **Section 3:** In the `ROOT_URLCONF` variable, it defines the default root path for the project folder. Our current project name is `first_django_project`, so it took that name and set into the variable. It is done by Django automatically. We don't need to set this manually and if we modify it then it can create an impact on execution.
- o **Section 4:** This section is important for database connectivity. This contains the driver info and database connectivity parameters, which is shown in the above screenshot. Django provides the `sqlite3` as default database with drivers.
- o **Summary:** The above mentioned sections are commonly used that is why these are highlighted and explained.
- `urls.py`: In that file, we defined the URLs, which are used by the user to process request and response. It is the file which connects to the views. It contains all URLs which are provided by Django and are user-defined as well.
- `uwsgi.py`: This file is used when we deploy our project on the application server in Django, it creates the default setting for deployment. With the help of this file, web server can easily communicate with the Django application.
- `manage.py`: This is the most important file for the Django project because our Django project starts with the help of `manage.py` file. If you can see in *Figure 3.1*, `manage.py` file and subfolder both are on the same level which means subfolder files and `manage.py` file both are equally important and are created by Django by default. In other words, it is a Django setup file, which executes the project.

Django execution and basic commands

To execute the Django project, we have to follow two basic steps, which are as follows:

1. The first step is to create a Django project. Command to start the Django project is `django-admin startproject first_django_project`. Here, `django-admin startproject` is the command for creating the project and `first_django_project` is the name of the project. After executing the above command, it creates the default Django architecture which is shown in *Figure 3.1*.
2. Now, the project is created and you want to execute a project. So the command is `python manage.py runserver`.

With the help of `runserver` command, we can run the project and it will deploy this project on default IP. The default IP is `127.0.0.1` and the default port is `8000`. This command has variations. Let's see the command shown

below. By passing the port **8001** with command, we can override the port explicitly. So now our code will run on **8001** port:

```
python manage.py runserver 8001
```

One more variation of **runserver** command is shown below:

```
python manage.py runserver 0.0.0.0:8001
```

If we pass the IP and port with **runserver** command then it will execute the given IP and port. In that way we can execute on any IP which user wants to execute, but we also want to insert the entry of that IP in the **settings.py** file.

Entry should be in the allowed host section. If we execute the command **python manage.py runserver** then it generates the following output in the console, which means your server is started and your project is executed perfectly.

The following screenshot shows the output for the same command:

```
Performing system checks...
System check identified no issues (0 silenced).
You have 13 unapplied migration(s). Your project may not work properly.
Run 'python manage.py migrate' to apply them.

May 23, 2019 - 16:51:08
Django version 1.11.11, using settings 'first_django_project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Figure 3.3 (a): Console output of runserver command

If we go to the browser and type the **http://127.0.0.1:8000/** then it will give the following output:

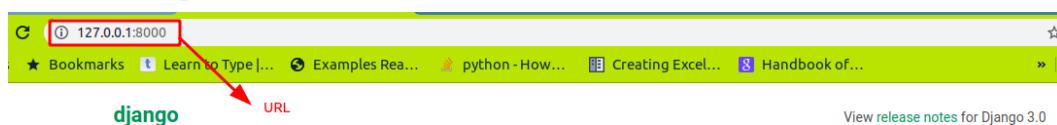


Figure 3.3 (b): Web browser output of runserver command

Django program execution flow

In Django, for executing the project, we run the command `python manage.py runserver` then,

1. Inside the `manage.py` file, `settings.py` file is mentioned that means it calls the setting file.
2. Inside the setting file there is a code section which executes line by line and sections are as follows:
 - 1) In the first statement, it imports the `os` module, which is required for execution.
 - 2) Then it sets the project build path.
 - 3) In the allowed host section if we define any IP then it starts the server on a particular IP.
 - 4) In the next section, it loads all apps, which are defined inside the installed app section.
 - 5) In the next section, it sets the defined URLs into the root section, which are defined inside the `urls.py` file.
 - 6) Now in the template section it searches for folder, if it is not there then default is set as blank.
 - 7) In the database section it executes the code and connects the project with database.
3. After executing the `setting.py` file, it starts the server and makes available our project on localhost with `8000` port.

All the above mentioned steps are followed by Django.

How Django handles requests

In Django, we saw the file `settings.py`. It uploads all sections of the file. Let's have a look at the steps and how it works in the back end in simple terms:

1. It loads the `ROOT_URLCONF` variable from the settings file.
2. This sets this globally and then marks the path for finding the user-defined URLs.
3. In the `urls.py` file URLs are defined, which are associated with the function.
4. The screenshot of the `urls.py` file is shown below:

```
"""first_django_project URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/3.0/topics/http/urls/
Examples:
Function views
1. Add an import: from my_app import views
2. Add a URL to urlpatterns: path('', views.home, name='home')
Class-based views
1. Add an import: from other_app.views import Home
2. Add a URL to urlpatterns: path('', Home.as_view(), name='home')
Including another URLconf
1. Import the include() function: from django.urls import include, path
2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

Urls are defined inside the urlpatterns block.

Figure 3.4: Screen shot of urls.py file

5. In the marked section all URLs are defined and after the comma (,) we write the function name, which is going to be executed at the time of URL call.

The following steps describe the control flow of request:

1. When URL is called from the user's end, the request travels through the Django server to the url file and checks if the user requested URL is defined inside the file.
2. When the user sends the request, it takes the `HttpRequest` object in the first place.
3. Then for sending the response, it uses the `HttpResponse` object.
4. If the URL is matched then it calls the associated function, which is defined inside the view file.
5. If the URL does not match with any URL, it raises the exception error. In Django, we have the predefined error handling pages, which are called automatically on the basis of error type.

Here are the default pre-defined error pages:

- **Handler 400:** If the user sends a request and any condition fails, then its error response status code is 400. In the background it calls the `django.views.defaults.bad_request()` and it produces the default exception `HttpResponseBadRequest`.

- **Handler 403:** If the user sends a request and it shows an error then the cause of error code is that the user does not have permission to access. In the background by default it calls the `django.views.defaults.permission_denied()` and it produces the default exception `HttpResponseForbidden`.
- **Handler 404:** If the user sends a request and it shows an error then the cause of error code is that URL does not match with any defined URLs. In the background by default it calls the `django.views.defaults.page_not_found()` and it produces the default exception `HttpResponseNotFound`.
- **Handler 500:** If the user sends a request and it shows an error then the cause of error code is that the server is not responding. Generally, this code is called when server-side has some issue or error occurred. In the background by default it calls the `django.views.defaults.server_error()` and it produces the default exception `HttpResponseServerError`.

HTTP request

HTTP request is a packet of data which is used for sharing information from one machine to another. Usually data format in the HTTP request is binary. In simple words it is used for communication between client and server.

In the server and client side communication, we require GET, PUT, or POST method. To transfer data from one end to another end, we use predefined methods which are as follows:

- **GET:** This method is used for getting data by sending related data to the server. We can cache the get request into the client side. It has a limitation of data length for sending the data. In **GET**, request data is visible into URL, so it is a little unsafe for transferring data.
- **POST:** It is used for sending data to the server like information of the customer, any type of file upload, HTML forms, and many more. In this method, data sending limit is more than **GET** method. In the **POST** method data is not sent in the URL, so it is not visible to others. We can say that the **POST** method is much safer than a **GET** request for a small amount of data transfer.
- **CONNECT:** This request establishes a connection pipeline to the server, which is always authenticating the request first then create.
- **PUT:** This request has some similarities with the **POST** method. It is also used for sending data. The basic difference between **PUT** and **POST** is that if we send the request again and again through **PUT** method, it always produces the same result but if we use **POST**, it can create replication or can give a different reaction every time. The **PUT** method is used to create or replace the target resource with the uploaded data.

- **DELETE:** If a client wants to remove any data from the current database then it is perfect for use.

Create the view file and configure user defined URL

As per the MVT model, business logic is defined in the views file. So `views.py` file is called by URLs, they communicate with models and templates as well. Now we are going to create a user-defined URL, which requires the `views.py` file. Currently, our project name is the `first_django_project`, so it is located in the subfolder that is `first_django_project`.

We create `views.py` file. Let's see the following screenshot:

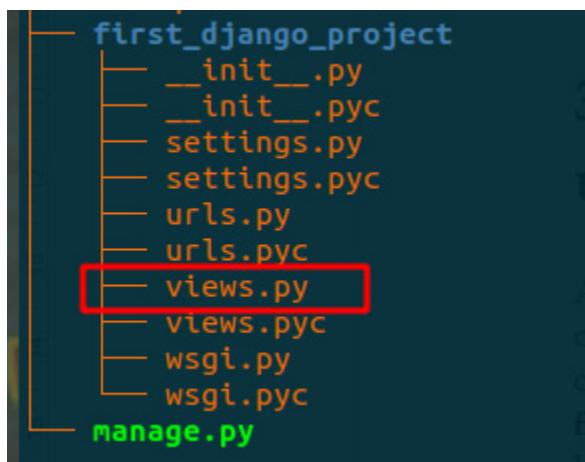


Figure 3.5 (a): Screenshot of `urls.py` file

In the above screenshot, we have highlighted block, which is `views.py` file; this file is created manually. When `views.py` file is executed it generates the `views.pyc` file automatically. The `.pyc` extension files are executable files.

Let's see the code which we write inside the `views.py` file, which is shown in the below screenshot:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, world. This is my first URL.")
```

Figure 3.5 (b): Review of `views.py` file

In the above screenshot, we imported the Django module, whose name is `HttpResponse`. Create the function `index`, which has a parameter `request`. `Index` function returns the `HttpResponse` object.

Now, we have to configure the `urls.py` file, configuration is mentioned below:

```
"""first_django_project URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/1.11/topics/http/urls/
Examples:
Function views
    1. Add an import: from my_app import views
    2. Add a URL to urlpatterns: url(r'^$', views.home, name='home')
Class-based views
    1. Add an import: from other_app.views import Home
    2. Add a URL to urlpatterns: url(r'^$', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.conf.urls import url, include
    2. Add a URL to urlpatterns: url(r'^blog/', include('blog.urls'))
"""

from django.conf.urls import url
from django.contrib import admin
from . import views          → For connecting the views.py file

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url('polls/', views.index), → For calling the index function
]
```

Figure 3.5 (c): `Urls.py` configuration screen shot

In the above screenshot, we have to use `index` function on the calling of `polls/` url. So we import that file first then call that function with the file name. That gives reference to the interpreter to find the function.

Now the configuration is done, we are going to start the server by command `python manage.py runserver` and call the URL from the browser:

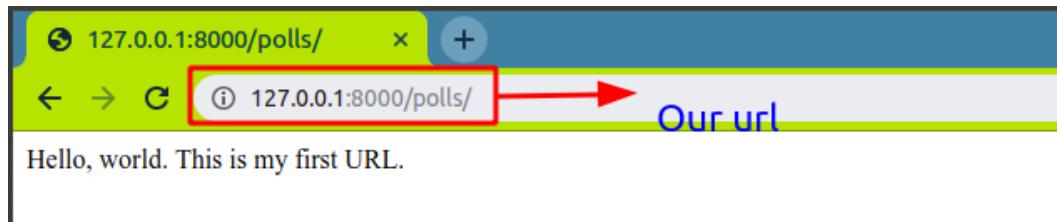


Figure 3.5 (d): Browser output on URL call

We call the URL and it produces the same result which we write inside the `index` function.

URL understanding

In Django, we can define the **dynamic URL**, for the security point of view, the user can create their own pattern dynamic URLs. Let's see the meaning of the following line:

```
url(r'^admin/', admin.site.urls)
```

Let's see the details of the above line:

- `url`: It is the method , which takes two arguments for execution.
- `r`: It means we are implementing regular expression.
- `^`: This symbol meaning is that URL will start from character a.
- `/`: This symbol means url will end with / sign.

So that way we can define the URLs.

Basics of Loggers and available logger types

In a project, a developer tries to handle all the exception but sometimes few unpredicted errors can happen or on the basis of user input, system can behave differently. So for tracking the program flow and capturing that error Python has a built-in module, which is used by Django. Python logger is divided into four parts, which are mentioned in the following sections.

Loggers

When we implement loggers into the program then it should be the first point of the program, which means they are defined in the top of the program. Loggers have different types, which handle data differently. We will discuss the types of logger in the upcoming topic.

Handler

We can say that the handler is the engine of the logger. It describes the behavior of every message, which is depended on the type of logger. We can pass the message or error as a message. If you want to print logs on the console screen or writing the logs on the individual file then both options are available.

Filters

We can say that filter is the provider which provides the additional control on log records, which is passed from logger to handler.

In the default, if any log message satisfies the defined condition, then the log level requirements will be handled. For example, we can enable a filter which can only allow ERROR messages to be printed.

Formatters

With formatters, we describe the format of the log record, which should be print as text. We can also define our own format to print logs.

Types of Loggers

Here are the available types of logger in the Django:

- **DEBUG**: It is used for printing or writing small information about the system or program behavior. It is used to debug the program.
- **INFO**: It is used for giving information about the system.
- **WARNING**: It is mostly used by a system that provides the information of a minor problem which occurred in the systems. You have seen this logger message from the system.
- **ERROR**: This logger gives information on issues, which have occurred in the program or system.
- **CRITICAL**: It is used to print a message stating that system or program has critical issue.

Example of the logger type calling, it contains the entry method:

- `logger.debug()`
- `logger.info()`
- `logger.warning()`
- `logger.error()`
- `logger.critical()`

Logger Implementation

It is very easy to implement loggers in Django. In the first phase, we have to configure `settings.py` file, create the log file, and mention the log type in the code.

Sample code of logger configuration:

```
LOGGING = {  
    'version': 1,  
    'disable_existing_loggers': False,  
    'handlers': {
```

```
'file': {  
    'level': 'DEBUG',  
    'class': 'logging.FileHandler',  
    'filename': '/home/debug.log',  
},  
},  
'loggers': {  
    'django': {  
        'handlers': ['file'],  
        'level': 'DEBUG',  
        'propagate': True,  
    },  
},  
}  
}
```

In the given example we have seen the code, which is used in the program. In the filename variable, we defined the path of our log file. Rest all is the predefined way to declare loggers into `settings.py` file.

Let's see an example of how we call the logger in our code:

```
from django.http import HttpResponse  
  
import logging  
logger = logging.getLogger(__name__)  
  
def index(request):  
    return HttpResponse("Hello, world. This is my first URL.")  
  
    logger.debug("This is the example of debug logger.")  
    logger.info("This is the example of info logger.")  
    logger.warn("This is the example of warning logger.")  
    logger.error("This is the example of error logger.")
```

Figure 3.6: Way of calling different types of loggers

In the above screenshot, we have shown how to call the loggers. At the top, we import the `logging` module. We create the object of `logging.getLogger(__name__)`. Then we refer that object and call the loggers type.

We will get the following output:

```
DEBUG This is the example of debug logger.  
INFO This is the example of info logger.  
WARNING This is the example of warning logger.  
ERROR This is the example of error logger.
```

This output is generated on the basis of the format definition.

The following code snippet is for formatter:

```
formatters = {  
    'f': {'format':  
        '%(levelname)-8s %(message)s'  
    },  
},
```

If we change the format then output will change. Let's use the other format:

```
formatters = {  
    'f': {'format':  
        '%(asctime)s %(name)-12s %(levelname)-8s %(message)s'  
    },  
},
```

We will get the above mentioned format output:

```
2019-05-29 15:01:12,957 root DEBUG This is the example of debug  
logger.  
2019-05-29 15:01:12,962 root INFO This is the example of info  
logger.  
2019-05-29 15:01:12,964 root WARNING This is the example of warning  
logger.  
2019-05-29 15:01:12,965 root ERROR This is the example of error  
logger.
```

Define ~~template~~ in Django

In Django, ~~template~~ means view part. It is that part which is used for presentation. We can configure HTML pages into the project with the user-defined URL. Let's see how to configure HTML pages with Django:

1. Create the **templates** folder in the same level of the project folder. This is done by manually. Then create the HTML file, which you want to configure. For example, I created the normal file which is shown below:

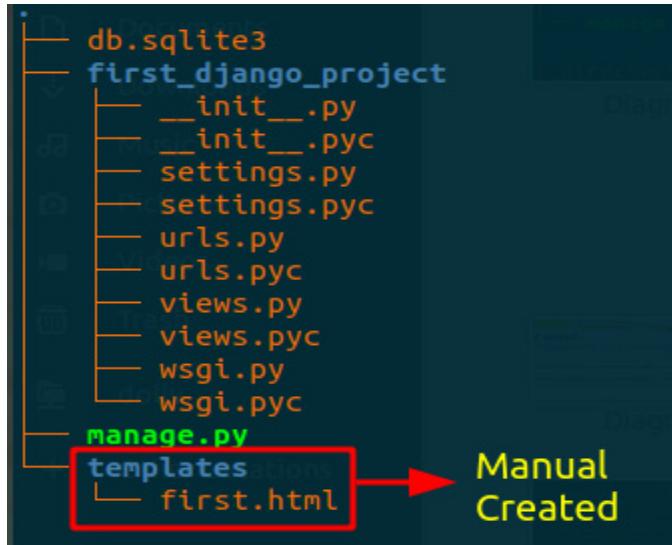


Figure 3.7 (a): Folder and file created with name templates and first.HTML.

The `first.html` file will look similar to the following screenshot:

```
<html>
<head>First Page</head>
<body>
<h1>This is My First Template Page.</h1>
</body>
</html>
```

Figure 3.7 (b): `first.html` file coding view

2. Set a few variables into `settings.py` file and define the static file path. Both are mentioned in the below screenshot:

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/1.11/howto/static-files/
STATIC_URL = '/static/'
```

Static path

Figure 3.7 (c): Define the `STATIC_URL` variable into `settings.py` file.

The templates section of `setting.py` file, where we have to write our code path, is shown in the following screenshot:

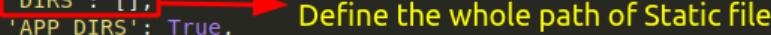
```
TEMPLATES = [
{
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'DIRS': [], 
    'APP_DIRS': True,
    'OPTIONS': {
        'context_processors': [
            'django.template.context_processors.debug',
            'django.template.context_processors.request',
            'django.contrib.auth.context_processors.auth',
            'django.contrib.messages.context_processors.messages',
        ],
    },
},
]
```

Figure 3.7 (d): Templates section of setting.py file

The `DIRS` value is filled in the `TEMPLATES` section of `setting.py` file, as shown in the following screenshot:

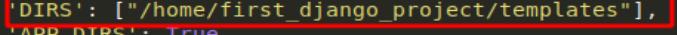
```
TEMPLATES = [
{
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'DIRS': ["/home/first_django_project/templates"], 
    'APP_DIRS': True,
    'OPTIONS': {
        'context_processors': [
            'django.template.context_processors.debug',
            'django.template.context_processors.request',
            'django.contrib.auth.context_processors.auth',
            'django.contrib.messages.context_processors.messages',
        ],
    },
},
]
```

Figure 3.7 (e): Dir value into the templates section.

3. Now create the new function inside the `views.py` file for calling the `index.html` file. It is mentioned in the below screenshot:

```

from django.http import HttpResponse
from django.shortcuts import render → Import package

import logging
logger = logging.getLogger(__name__)

def index(request):
    return HttpResponse("Hello, world. This is my first URL.")

def call_temp(request):
    return render(request, "first.html") → Calling Page

```

Figure 3.7 (f): view.py file screenshot.

In the above screenshot, we have highlighted two blocks. The first one shows that we called the new package `render`, which is used for giving a response to the request. In the second block, we defined the `call_temp` function. Inside that we called the `first.html` file, which is stored into the `templates` folder.

4. We have to configure the `url` for calling that function, thus, it is mentioned into `urls.py` file which is mentioned in the below screenshot:

```

from django.conf.urls import url
from django.contrib import admin
from . import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url('polls/', views.index),
    url('show_temp/', views.call_temp), → Config. Url
]

```

Figure 3.7 (g): urls.py file screenshot.

5. Now all configurations are done to run the command:

`python manage.py runserver`

Then call the URL `http://127.0.0.1:8000/show_temp/` on the browser. The output of the `url` is shown in the below screenshot:



Figure 3.7 (h): Browser output after hitting the url.

6. In the `first.html` file, we have mentioned the text `This is My First Template Page.` It is generated output of the `first.html` page and more description is given in *Figure 3.7(h)*.

If we follow the above mentioned steps then we can easily configure our HTML page into Django module.

The powerful concept of Django

If we talk about one of the most powerful concepts of Django then it is an admin module. Django provides the automatic admin module interface. It provides the model access functionality; we can connect and show the data in the front end.

For using admin module it requires mentioned below packages:

- `django.contrib.admin`
- `django.contrib.auth`
- `django.contrib.contenttypes`
- `django.contrib.sessions`
- `django.contrib.messages`

These packages are available into the `setting.py` file, if you refer the `INSTALLED_APPS` section. These are already generated by Django. In the `TEMPLATES` section:

- `django.contrib.auth.context_processors.auth`
- `django.contrib.messages.context_processors.messages`

In the `MIDDLEWARE` section:

- `django.contrib.auth.middleware.AuthenticationMiddleware`
- `django.contrib.messages.middleware.MessageMiddleware`

To access the Django admin module that is already mentioned into the url file review the following screenshot, and refer to the highlighted part:

```
from django.conf.urls import url
from django.contrib import admin
from . import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url('polls/', views.index),
    url('show_temp/', views.call_temp),
]
```

Figure 3.8 (a): Admin module url.

Now call that url in the browser and the output is mentioned in the below screenshot:

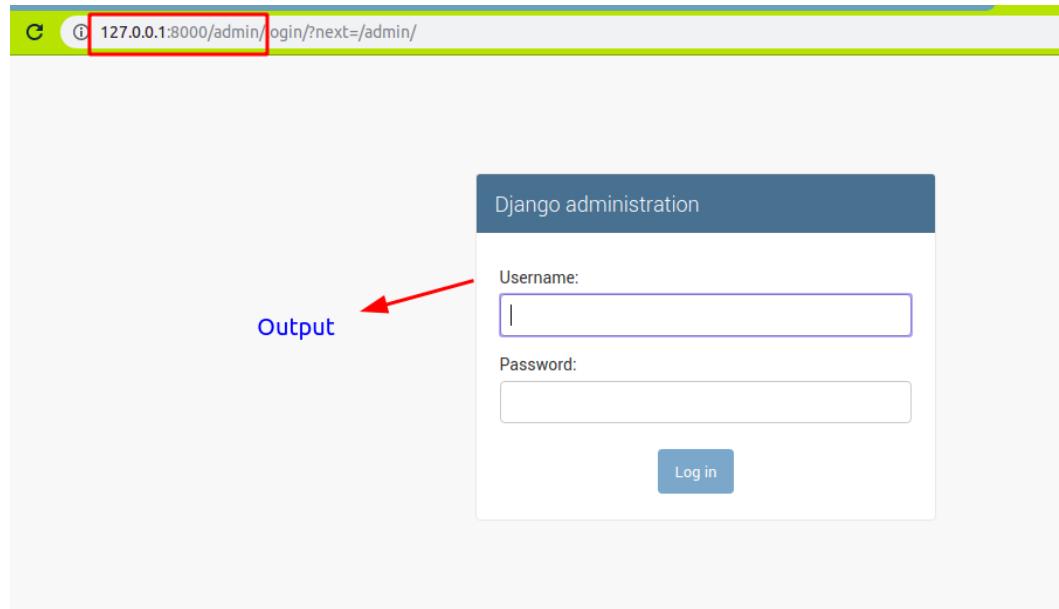


Figure 3.8 (b): Admin module url output page.

We hit the URL `http://127.0.0.1:8000/admin/` from the browser after `admin/...` part is generated automatically. Output of that URL is mentioned above.

In the above screenshot, it asks for username and password for login on admin module. For authentication, we need to create a username and password. So we can create a new user from the console by command, `python manage.py createsuperuser`.

Then it will ask for the following details:

Username (leave blank to use 'sjain'): testuser

Email address: test@gmail.com

Password:

Password (again):

This password is too common.

Password:

Password (again):

Superuser created successfully.

If a user is created then it gives the message Superuser created successfully. Now we are going to hit admin URL again and enter the username and password, which we created. Then we submit it and it will show the following screen:

The screenshot shows the Django admin interface. At the top, there's a blue header bar with the text "Django administration" on the left and "WELCOME, TESTUSER | VIEW SITE / CHANGE PASSWORD / LOG OUT" on the right. Below the header, the main content area has a light gray background. On the left, there's a sidebar with a dark blue header labeled "AUTHENTICATION AND AUTHORIZATION". Underneath, there are two items: "Groups" and "Users", each with a green "+ Add" button and a yellow "Change" link. To the right of the sidebar, there are two sections: "Recent actions" (which is currently empty) and "My actions" (which also says "None available").

Figure 3.8 (c): Screenshot of the page, which comes after login.

The above screen appears after login to admin module. If the user is valid then it calls the above-mentioned page. Admin module provides so many functionalities for the developer automatically. Only we need to configure the Django admin module and create the new **Superuser** for login on the admin module.

After logging into the Django module, it provides a few more functionalities from the front end. We don't need to develop from the code. It works from the front end. In the above screenshot, we have shown two headings: **Groups** and **Users**. We can create a new user by clicking on **Users +Add** functionality and give access for the user from the front end. Following is the user screen output:

The screenshot shows the Django admin 'Users' list page. At the top, there's a blue header bar with 'Home', 'Authentication and Authorization', and 'Users'. Below it, a search bar and a 'Search' button are on the left. On the right, there's an 'ADD USER +' button. The main area has a table with columns: USERNAME, EMAIL ADDRESS, FIRST NAME, LAST NAME, and STAFF STATUS. A single row is shown for 'testuser' with email 'test@gmail.com'. The 'STAFF STATUS' column shows a green circle with a checkmark. To the right of the table is a 'FILTER' sidebar with sections for 'By staff status' (All, Yes, No), 'By superuser status' (All, Yes, No), and 'By active' (All, Yes, No).

Figure 3.8 (d): Screen after click on the users.

It is the screen which comes after clicking on the users tab that is shown in the following screenshot:

This screenshot is identical to Figure 3.8 (d), showing the 'Users' list page in the Django admin. The 'ADD USER +' button is highlighted with a red box in the top right corner. The rest of the interface, including the table data and the filter sidebar, is the same.

Figure 3.8 (e): Same screen but highlighted into the corner.

It is the same screen which is shown in the above screenshot, but in the right corner, there is **ADD USER** button for adding a new user. So by clicking on that button, we can create a new user. Following is the add new user screen:

Django administration

Welcome, TESTUSER | View site / Change password / Log out

Home > Authentication and Authorization > Users > Add user

Add user

First, enter a username and password. Then, you'll be able to edit more user options.

Username:

Required: 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password:

Your password can't be too similar to your other personal information.
Your password must contain at least 8 characters.
Your password can't be a commonly used password.
Your password can't be entirely numeric.

Password confirmation:

Enter the same password as before, for verification.

Actions:

- Save and add another
- Save and continue editing
- SAVE**

Figure 3.8 (f): After click on add user button

The above screen is shown when a user clicks on **ADD USER** button. We need to write the username and password then click on the **SAVE** button. We can also check if it is created or not. Following is the user info update screen:

Django administration

Welcome

Home > Authentication and Authorization > Users > djngotest

Change user

Personal info

Username:

Required: 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password: algorithm: pbkdf2_sha256 iterations: 36000 salt: Dsb19z***** hash: WYXpF+*****

Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form.

Actions:

- First name:
- Last name:
- Email address:

Figure 3.8 (g): After click on save button.

In the above screenshot, we can see that in the username field `djangotest` is mentioned because we created the new user. At the bottom, three more fields are there, which are **First name**, **Last name**, and **Email address**.

In the next *Figure 3.8 (h)*, we filled all fields with **First name**, **Last name**, and **Email address**. After that three types of permission options are given, which are as follows:

- **Active:** It is selected by default, which means that the user is active and created.
- **Staff status:** It is selected by us because it gives access to login.
- **Superuser status:** If we select this, it means the user has all rights.

The screenshot shows the Django admin 'User' creation form. The top section, 'Personal info', contains fields for First name ('krishna'), Last name ('vishwkarma'), and Email address ('krishnav@gmail.com'). The bottom section, 'Permissions', contains three checkboxes: 'Active' (selected), 'Staff status' (selected), and 'Superuser status' (unselected). Descriptions below each checkbox explain their function.

Personal info	
First name:	krishna
Last name:	vishwkarma
Email address:	krishnav@gmail.com

Permissions	
<input checked="" type="checkbox"/> Active	Designates whether this user should be treated as active. Unselect this instead of deleting accounts.
<input checked="" type="checkbox"/> Staff status	Designates whether the user can log into this admin site.
<input type="checkbox"/> Superuser status	Designates that this user has all permissions without explicitly assigning them.

Figure 3.8 (h): User permission page.

In *Figure 3.8 (h)* the description is already given, the form is showing three types of permissions and we selected the only two, which make user active and provide him login access.

After selecting the permission options and clicking save, let's try to login on the admin module. The following screenshot is the welcome screen for new user login, which shows that the user has limited permission.

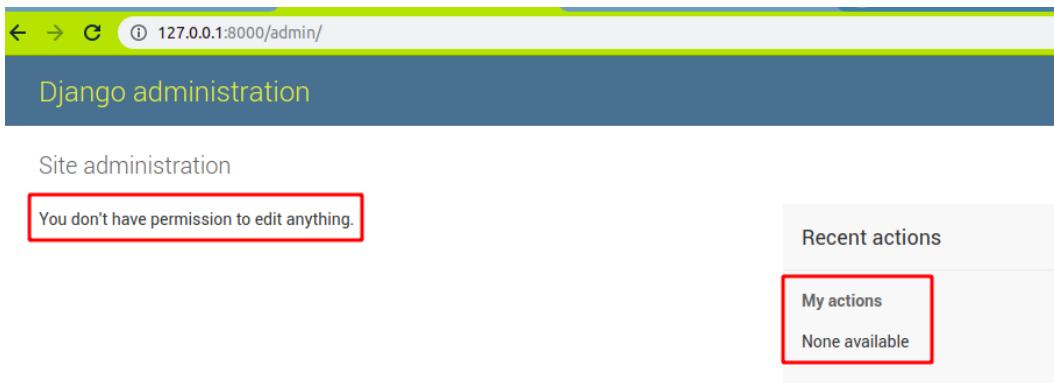


Figure 3.8 (i): Screen of new user login.

It is showing that the user has only login permission as the remaining modules are missing.

If we want to create a new group then from the group section, **Add** button is available to create a new group, which is shown in the below screenshot:

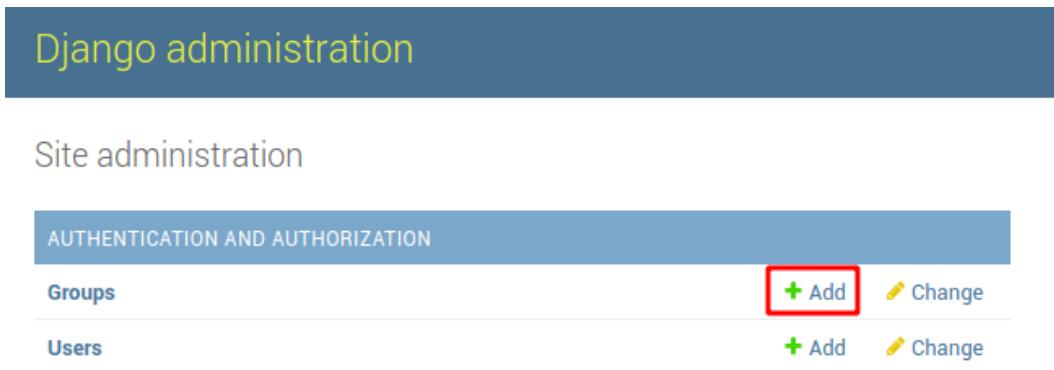


Figure 3.8 (j): Screen for new group creation

Now click on the **Add** button, then the following screen will appear:

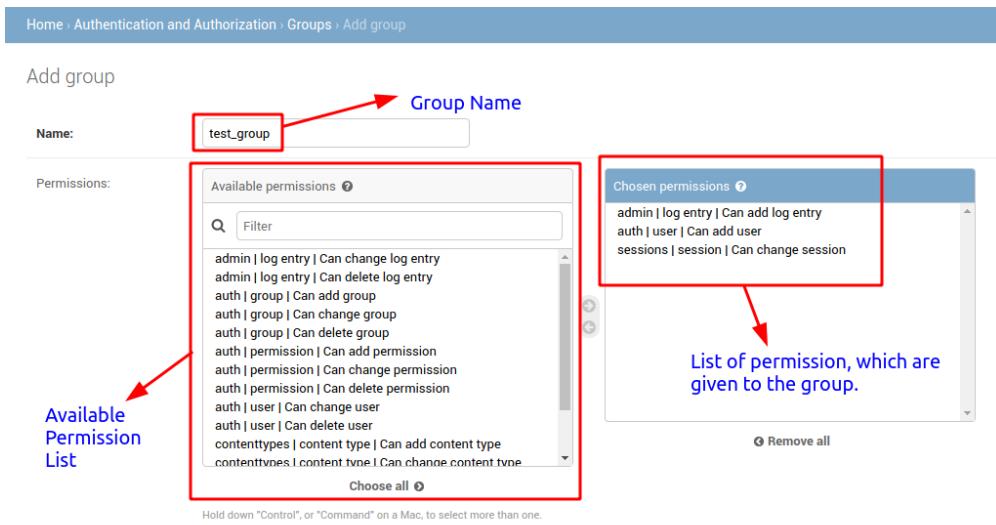


Figure 3.8 (k): Screen appeared after click on add group

In the above screenshot, we have mentioned the three highlighted points. First, we can give any name for the group. In the second section, a list of available permissions are highlighted. In the third part, we can give any permission to the group. After adding the permission we can also remove any permission.

In the next section, let's see how we can put a user inside the group. Now click on the users and it will open the new screen, which will show the list of the users, then click on the user. User permission screen will open. Here's the screen view:

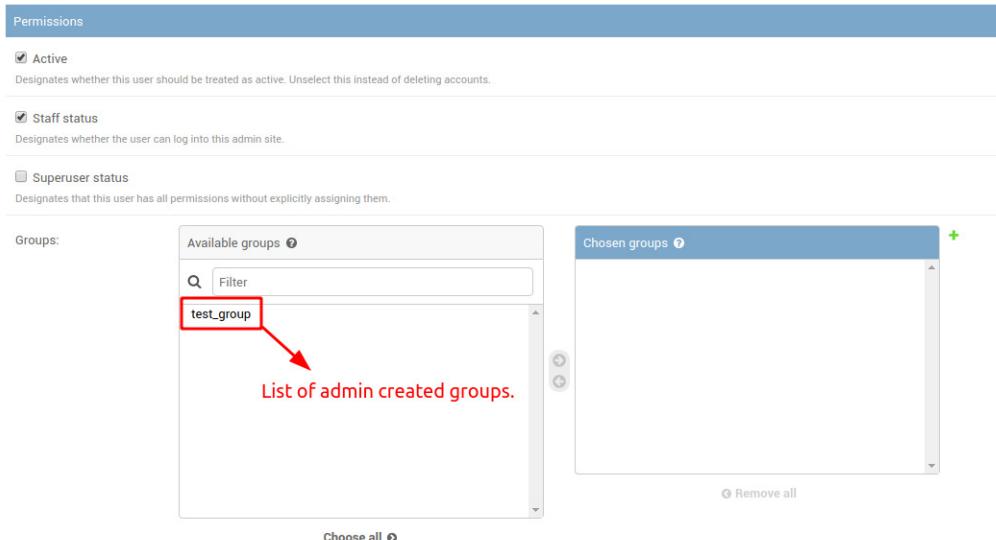


Figure 3.8 (l): Screen of user created group

In the above screenshot, we have highlighted a block, which shows the user defined groups. We can assign multiple groups to a single user and inside the single group, we can define multiple access permissions.

On the basis of permission, we can easily manage security. So this is Django-admin from the GUI, we can do many things from backend on admin module. We will use this in the upcoming chapters.

Conclusion

Django is a Python web framework, which develops web applications, websites, and APIs. This chapter covered the basics of Django, APIs, Django default architecture, request and response lifecycle, basic commands of Django, control flow in the application, Django handler, create a view, URL file, understanding of the templates, and knowledge of API development and debugging on the server side.

In the next chapter, we will create the restful API and connect with the database.

Questions

1. What is the Django?
2. What is web API?
3. What is the folder architecture of Django?
4. What is the use of `manage.py` file?
5. What is the use of `url.py` file?
6. What is the use of `settings.py` file?
7. What is the use of `uwsgi.py` file?
8. In Django, where is the database configuration defined?
9. What is the request and response cycle work?
10. List of available default error handlers in Django?
11. What is the difference between PUT and POST method?
12. How to configure loggers in Django?
13. What is admin module in Django?
14. How to define user-defined URL?

CHAPTER 4

API Development with Django

In this chapter, we will learn how to make APIs in the Django framework. We included this chapter in the book, because for implementing the microservices with Django, readers require the knowledge of APIs and how to work with APIs. This chapter covers how to make a new app with Django and knowledge of how the user sends a request to the server and how the server gives a response. We are using the JSON format of data, which will travel over the network. We also connect Django with basic database SQLite.

Structure

- The basic idea of API
- Create app in Django
- Connect Django with database
- Function-based and class-based views
- Postman tool description
- Create function-based views
- Create class-based views

Objective

This chapter covers the basic idea of APIs like what is API and why it is required and how it works. For better understanding, we have used screenshots, code, and theory of the concepts. It also covers how to create an app in Django project and its benefits.

In this chapter, we will learn about database connectivity and definition of views. The postman tool description in depth is also available. At the end of this chapter, you should easily understand the web APIs, class-based and function-based view, postman tool and Django app.

The basic idea of API

An **application programming interface (API)** is a set of operations, which execute and produce results. Basically, it is used for data transfer over the network from one end to another end. For example, we have a mobile application, which requires user details like username, password, email, and first name for registration on the app. So when the user clicks on submit, it calls the API for transferring data to the server. That data goes to the server and gets stored into the database.

Let's have a look at another example. If a user wants to login on the same application then they will click on login or sign up button. From the field, it will take the information like username and password, and then send to the server for validation. If they match with the available data, it redirects the user to the dashboard.

This is one use case of API; others are also available such as security handling, data security, hardware or software control, and many more. Nowadays with the help of APIs, we can deploy much functionality into a single application and it also provides the functionality to use the already developed applications instead of creating our own.

Create an app in Django

Let's create an app in the Django project for maintaining the code. The benefit of the app is that it becomes easy to handle the code module separately. Even if we change something in a particular module, it shouldn't affect the other modules. In simple words, the app we are going to create is a separate folder, which is used for dividing the whole code into small modules. To start with the app, we make the new Django project, whose name is `django_project01`.

Let's see the architecture that is shown below:

```
|── django_project01
|   ├── __init__.py
|   ├── settings.py
|   ├── urls.py
|   └── wsgi.py
└── manage.py
```

That is the default architecture of Django. These are auto-generated file. Now let's create the new app in the Django and its command is as follows:

```
python manage.py startapp first_app
```

Here, `startapp` is used for creating the app and `first_app` is the name of my Django app.

After executing the command, a new folder is created inside the project that is `django_project01` and our project's new structure is as follows. In the `app` folder that is `first_app` has some auto-generated files which are as follows:

- `admin.py`: It is used for importing the model and shows them on the admin page.
- `apps.py`: It is used to configure the app into the project. Default coding of `apps.py` file is mentioned below:

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals
from django.apps import AppConfig
```

```
class First AppConfig(AppConfig):
    name = 'first_app'
```

- `__init__.py`: It executes in the first place.
- `migrations`: It contains all migrated files. When we migrate our project, Django creates the migration file which is stored inside the `migrations` folder.
- `models.py`: It is used to create a model for our project. When we create the Django database model it is created by `models.py` file. A more detailed description of the Django model will be described in *Chapter 5: Database modeling with django*.
- `tests.py`: It is used for writing test cases of the Django project.
- `views.py`: It is used for writing business logic. For more details, please refer to *Chapter 3: Getting Started with Django*.

The following is our project's architecture:

```
└── django_project01
    ├── __init__.py
    ├── __init__.pyc
    ├── settings.py
    └── settings.pyc
```

```

|   └── urls.py
|   └── wsgi.py
└── first_app
    ├── admin.py
    ├── apps.py
    └── __init__.py
        └── migrations
            └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
└── manage.py

```

Now the application is created and for executing the app, we are required to configure the app. These are the steps for creating and configuring the app:

1. Create the app by command `python manage.py startapp first_app`.
2. Put the entry of a new app in the `settings.py` file which is shown in *Figure-4.1(a)*:

```

ALLOWED_HOSTS = []

# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'first_app',
]

```

Figure 4.1 (a): New app entry into setting file

The above screenshot is a snippet of `settings.py` file and it captures the `INSTALLED_APPS` section. At the end, we defined our app name `first_app`.

3. Then execute the command `python manage.py runserver`.
4. Hit the localhost URL `127.0.0.1:8000` on the browser.

Connect Django with database

Django provides SQLite as the default database. When we install Django it automatically installs SQLite. In the previous chapter, we have seen the available section `DATABASES` of the `settings.py` file. Please refer to the below screenshot which shows the way to connect with the SQLite database:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Figure 4.2 (a): Sqlite database connection string details

This is default setting of the `settings.py` file. In the initial phase, by default Django always connect with SQLite DB. We can change the settings manually, every database has different connection strings or parameters.

Function-based and class-based views

Both are available and have authenticated way to create the views file. By using any one way we can create the API and use it. The difference is that earlier, developers were using function based views and currently, they use class based views more commonly.

Function-based views

In the function based views, we define the different functions for every API. So if we customize the code, it creates complexity. To understand this better, let's see the example code:

```

# Example of function based views.

## get method function example

def get_method_example(request):

    # This is the way to use get method
    userid = request.GET.get('userid', '')
    resp = {}
    if userid:
        resp['status'] = 'Success'
        resp['status_code'] = '200'
        resp['data'] = userid
    return HttpResponse(json.dumps(resp), content_type = 'application/json')

## post method function example

def post_method_example(request):

    # This is the way to use post method
    user_name = request.POST.get("user_name")
    resp = {}
    if user_name:
        resp['status'] = 'Success'
        resp['status_code'] = '200'
        resp['data'] = user_name
    return HttpResponse(json.dumps(resp), content_type = 'application/json')

```

Figure 4.3 (a): Function based view screenshot

In the above screenshot, we have mentioned the two highlighted blocks. First is the GET method function example. Second is used for POST method function example.

Both functions return `resp` JSON, which will print on the browser.

In the following screenshot (*Figure 4.3 (b)*), have mentioned function based view, URLs settings. Whenever we create the POST or GET method, for calling these methods we need to associate them with URLs.

The way of assigning the method to URL is mentioned below. The following is the screenshot of `url.py` file:

```

urlpatterns = [
    url(r'^get_example/$', views.get_method_example),
    url(r'^post_example/$', views.post_method_example),
]

```

Figure 4.3 (b): Function based view, urls.py file screenshot

In the above screenshot, for using the two different methods, we required the two methods. The `get_example/` is used for GET method and `post_example/` is used for the POST method.

Class-based views

In the class-based view, we create a single class, and inside this class, we define two different functions, similar to function-based view. We require two functions, which differentiate the request methods. In the function based views, we create the two functions and both have different URLs for calling but in the class-based view, it requires only one URL for calling both methods. When we pass the POST parameters it will call the POST method and without parameter, it will call the GET method. This logic will be handled in the HTTP request.

Following is the example of class-based views:

```
from django.views import View

class Class_based_view_example(View):

    # This is get request code block
    def get(self, request):
        # some code as per logic

    # This is post request code block
    def post(self, request):
        # Code block for POST request
```

Figure 4.3 (c): Class-based view screenshot

In the above screenshot, we have mentioned the two functions, first is GET and second is POST.

Following is the screenshot of class-based views, `urls.py` file:

```
urlpatterns = [
    url(r'^example_class_based/$', views.Class_based_view_example.as_view(), name='clbased'),
]
```

Figure 4.3 (d): Class-based view, urls.py screenshot

In the above screenshot, we have mentioned the single URL for both POST and GET method. Which method it will call depends upon the calling URL patterns.

Postman tool description

This is a tool, which is used for testing the API functionality. When we create the API, it needs to be tested for its workability. To check its request and response we need a platform which produces the result. We cannot use our API directly in the mobile app or project. Before delivering the APIs, we test them with tools and Postman is one of them. We can also measure the API performance.

This tool is available in two formats: first is browser based Google Chrome plug-in and full standalone application.

Standalone applications have some extra features, which are not available in the browser based app. For example, there is the feature to put off SSL settings in standalone app, but not available in the browser based app. I am using the standalone application for demo. Let's start the Postman application.

Here is the first screen of the Postman application:

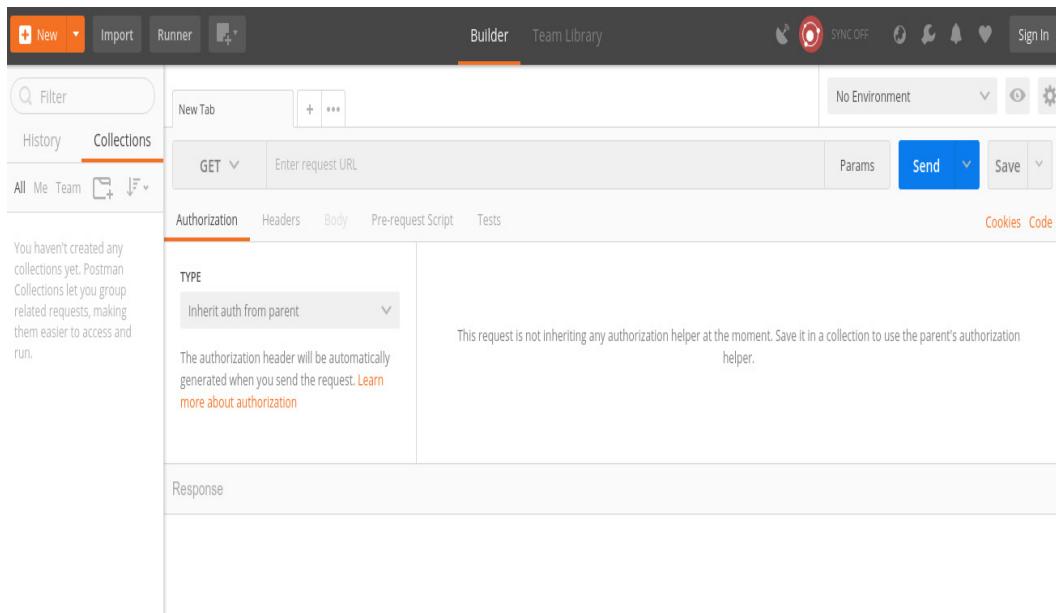


Figure 4.4 (a): Postman application first screen

Following is the list of methods available in the Postman application:

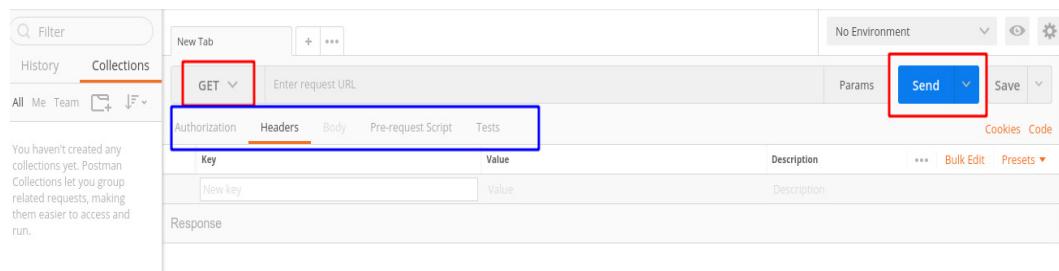


Figure 4.4 (b): Postman available options overview

In the above screenshot, we have three highlighted blocks. The first block shows **GET**, it means it gets type request. The second block shows the **Send** button, which means on click of a button, the request is sent.

The blue highlighted block shows the option to send the request like header settings, **Authorization** option, **BODY** parameters if the request type does not get, pre-request-scripts, and tests settings.

Beside the GET options, there is a text box, which is used to write URL.

Following is the **POST** method screenshot:

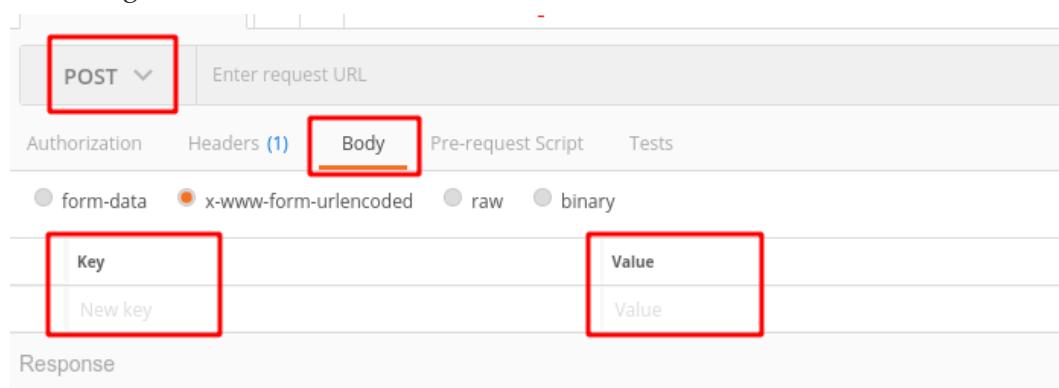


Figure 4.4 (c): Post method available options overview

In the above screenshot, we have highlighted four blocks. The **POST** block means the post method is for the request. In the second block, which is **BODY**, appears when we select the request type **POST**.

So it provides the new options, which have following formats for sending data:

- **Form-data**
- **X-www-form-urlencoded**
- **Raw**

- Binary

When we select the **X-www-form-urlencoded** format to send data, it takes data into key and value pairs. These key and value blocks are mentioned.

Following are the list of methods that we can select for sending request:

Get, Post, Put, Patch,

Delete, Copy, Head,

Options, Link, Unlink,

Purge, Lock, Unlock,

View, Propfind

Following is the **GET** request demo:

The screenshot shows the Postman interface with the following details:

- URL: `http://127.0.0.1:8000/polls/`
- Method: **GET** (highlighted with a red box)
- Send button: **Send** (highlighted with a red box)
- Authorization tab is selected.
- Headers tab shows **Content-Type: application/x-www-form-urlencoded**.
- Body tab is empty.
- Tests tab is empty.
- No Environment is selected.

Figure 4.4 (d): GET method request demo

In the above screenshot, we have highlighted three blocks. The first block defines the request type **GET**. In the second block, our hitting URL is defined. The third one shows the **Send** button.

Following is the view of output screen, after hitting the **Send** button:

The screenshot shows the Postman interface after hitting the **Send** button, displaying the following results:

- URL: `http://127.0.0.1:8000/polls/`
- Method: **GET**
- Headers tab is selected, showing **Content-Type: application/x-www-form-urlencoded**.
- Body tab is empty.
- Test Results tab shows the response: "Hello, world. This is my first URL."
- Status: **200 OK**
- Time: **36 ms**

Figure 4.4 (e): Output of GET method request

In the above screenshot, we have highlighted three red and one blue color block. In any request, we have to select the header and in our scenario, the content type is `application/x-www-form-urlencoded`. The third block shows the types of output, which are as **Pretty**, **Raw**, and **Preview**. The blue block has our output. This output is generated after hitting the send button.

Following is the example of **POST** method request with output:

The screenshot shows a POST request in Postman. The URL is `http://127.0.0.1:8000/post_example_url/`. The body contains a key-value pair: `user_name` with value `Shayank Jain`. The response JSON is:

```

1  {
2   "status": "Success",
3   "status_code": "200",
4   "Passed User_Name Data": "Shayank Jain"
5 }

```

Figure 4.4 (f): POST method request example

The above screenshot, shows the red blocks, which have the first block that shows the request type and in the side, URL is defined which we are going to call for request. In the next block key and value options are available. We used the `user_name` as key and `Shayank Jain` is value. In the last block, we have mentioned the output of our request, which is showing that request status is `Success` and `status_code` is `200`.

In the last parameter, it shows passes `user_name` data. All these output parameters are defined by the programmer.

Create function-based views

In the above topics, we have seen the function based view. Now we will see how to use function-based view in our Django project. We will create a new project, its name is `my_project`. After that, we create a new app, whose name is `func_based`.

Following is the tree view of our project:

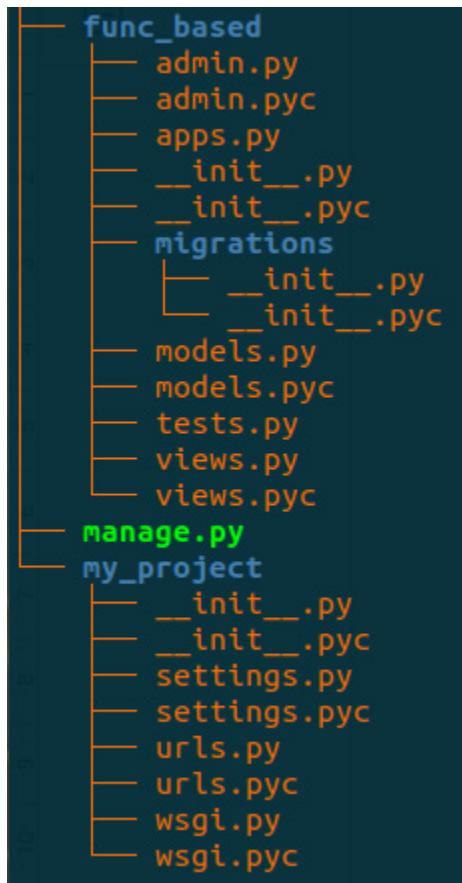


Figure 4.5 (a): `my_project` tree structure

In the `view.py` file, we are going to add our function-based view code. Following is the `views.py` code:

```

# starting point of our code.

from __future__ import unicode_literals
from django.shortcuts import render
import json
from django.http import HttpResponse

```

```
from django.views.decorators.csrf import csrf_exempt

## get method function example
def get_method_example(request):
# This is the way to use get method
    userid = request.GET.get('userid','')
    resp = {}
    if userid:
        resp['status'] = 'Success'
        resp['status_code'] = '200'
        resp['data'] = userid
    return HttpResponse(json.dumps(resp), content_type = 'application/json')

@csrf_exempt
## post method function example
def post_method_example(request):
# This is the way to use post method
    user_name = request.POST.get("user_name")
    resp = {}
    if user_name:
        resp['status'] = 'Success'
        resp['status_code'] = '200'
        resp['data'] = user_name
    return HttpResponse(json.dumps(resp), content_type = 'application/json')
```

The above mentioned code snippet is from `views.py` file, which mentions two different functions. One is for GET method and second for POST method. We import some packages that are required to run the code of our file.

Now, our view file is ready and we have to configure our `urls.py` also because for calling the function we require the URLs.

Following is the `url.py` file code for function based view:

```
from django.conf.urls import url
```

```

from django.contrib import admin
from func_based import views as app

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^test_function_get/$', app.get_method_example),
    url(r'^test_function_post/$', app.post_method_example),
]

```

In the above code, we imported the views file and two new URLs are defined for calling the GET and POST method. One URL `test_function_get` for GET method and `test_function_post` for POST method.

Let's have a look at how to call the URL from the Postman application and what output it will generate.

Following is the **GET** method request screenshot:

The screenshot shows the Postman interface with the following details:

- Method:** GET
- URL:** `http://127.0.0.1:8000/test_function_get/?userid=Emp0034`
- Authorization:** Inherit auth from parent
- Type:** Inheriting auth from parent
- Body:** JSON response containing:


```

1 {
2   "status": "Success",
3   "status_code": "200",
4   "data": "Emp0034"
5 }
```

Figure 4.5 (b): GET method function based view request and response

In the above screenshot, we have highlighted the block which shows the URL and its GET request. In the URL, we passed the `userid` with get request. The output is mentioned at the bottom. The output is user defined and programmer can change that function.

Following is the **POST** method request screenshot:

The screenshot shows a POST request in Postman. The URL is `http://127.0.0.1:8000/test_function_post/`. The request body is set to `form-data` and contains a key-value pair: `user_name` with value `Shayank Jain`. The response body is displayed in JSON format:

```

1  [
2   "status": "Success",
3   "status_code": "200",
4   "data": "Shayank Jain"
5 ]

```

Figure 4.5 (c): POST method function based view request and response

In the above screenshot, we have highlighted the blocks which show the used URL, request parameter and its value. In the URL we are passing the `user_name` with value `Shayank Jain`, it is POST request. The output is mentioned at the bottom.

We can modify the response as per business requirement.

In the function based view, we defined two functions for serving the request. In the result, we have to define the two different URLs in the urls.py file.

Create class-based views

We already know about the class-based views that are already covered in the previous topics. Now we will create the class-based view and see how it works in our Django project. We already have created a new project, its name is `my_project`. In the same project, we will create a new app, which is `class_based`. So now our project has two apps, one for function-based view and second for class-based view. Let's see the structure of our project:

```
class_based    db.sqlite3    func_based    manage.py    my_project
```

The bold highlighted word is used for showing the folder. So now our project `my_project` has two folders on the same level. For the class-based view, we are required to define class and function and it is mentioned below.

Following is the class based view example and it is `views.py` file code:

```
# Starting point of view file.

from __future__ import unicode_literals
from django.shortcuts import render
from django.http import HttpResponse
import json
from django.views import View

#####
# Example of function based views.
#####

class Class_based_view_example(View):

    # This is get request code block
    def get(self, request):
        userid = request.GET.get('userid', '')
        resp = {}
        if userid:
            resp['status'] = 'Success'
            resp['status_code'] = '200'
            resp['data'] = userid
        return HttpResponse(json.dumps(resp), content_type =
'application/json')
```

```
# This is post request code block
def post(self, request):
    # This is the way to use post method
    user_name = request.POST.get("user_name", '')
    resp = {}
    if user_name:
        resp['status'] = 'Success'
        resp['status_code'] = '200'
        resp['data'] = user_name
    return HttpResponse(json.dumps(resp), content_type =
'application/json')
```

The above mentioned code is an example of class-based view and it is used inside a `views.py` file. We create the class that is `Class_based_view_example`, inside the class, and we define two functions `post` and `get` to the server for the request. We also import some packages that are required to run the code of our file.

Now, our `view` file is ready and to configure `urls.py`, the code is mentioned below.

Following is the `urls.py` file code for *class-based* view:

```
from django.conf.urls import url
from django.contrib import admin
from func_based import views as app
from django.views.decorators.csrf import csrf_exempt
from class_based.views import Class_based_view_example as cl_view

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^test_function_get/', app.get_method_example),
    url(r'^test_function_post/', app.post_method_example),
    url(r'^test_class_view/', csrf_exempt(cl_view.as_view())),
]
```

In the above code, four URLs are mentioned, which have `test_function_get` and `test_function_post` as part of function based views. For class-based view, we defined the `test_class_view` URL only. It will serve both GET and POST method

by single URL call.

Following is the **GET** method request screenshot:

The screenshot shows the POSTMAN interface with a GET request. The URL is highlighted with a red box: `http://127.0.0.1:8000/test_class_view/?userid=Emp0034`. The response body is displayed in JSON format:

```

1  {
2      "status": "Success",
3      "status_code": "200",
4      "data": "Emp0034"
5  }

```

Figure 4.5 (d): GET method class based view request and response

In the above screenshot, we have highlighted the block which shows the URL and its GET method request. In the URL, we passed the `userid` with GET request. In the URL, we called the `test_class_view` with parameters. Its output is shown at the bottom of the same screenshot.

Following is the **POST** method request screenshot:

The screenshot shows the POSTMAN interface with a POST request. The URL is highlighted with a red box: `http://127.0.0.1:8000/test_class_view/`. The request body is set to `form-data` and contains a key-value pair: `user_name` with value `Shayank Jain`. The response body is displayed in JSON format:

```

1  {
2      "status": "Success",
3      "status_code": "200",
4      "data": "Shayank Jain"
5  }

```

Figure 4.5 (e): POST method class based view request and response

In the above screenshot, we have highlighted some blocks. We see that we are using the `test_class_view` URL for sending the request. In the URL we are passing the `user_name` with value `Shayank Jain`, it is POST request. The response output of the request is mentioned at the bottom of the same screenshot.

In the class based view, we only defined the one url and on the basis of parsing parameter, it automatically identifies the request type which is a GET or POST method.

So both are preferable to develop the APIs and it is your choice to select *function-based or class-based* views.

Conclusion

In today's world, APIs are showing the important role. If we create any mobile application then APIs are required to get and set data into parameters. This chapter covers the basic idea of APIs and request and response traveling over the server. We also discussed example code with implementation.

In the next chapter, we will discuss Django ORM and how to request a server throughout the database.

Questions

1. What is APIs?
2. How to create the API?
3. What are the common methods of API?
4. How to connect Django with database?
5. What is function-based view?
6. What is class-based view?
7. What is the difference between function-based and class-based views?
8. How to configure class-based view, `urls.py` file?
9. How to configure function-based view, `urls.py` file?
10. What is Postman tool?
11. Why we use Postman tool?
12. List of available methods in Postman?
13. How to test API through the Postman?

CHAPTER 5

Database Modeling with Django

In this chapter, we will get the information about what are Django Object relational mappers and learn how to manage them. It covers to following database connection strings and how to insert, select, or update query perform on the database. We will use the Django shell for all the operations. This chapter also covered the model.py file code and how it works with the database. We are going to use multiple databases like SQLite, MySQL, and PostgreSQL.

The study of this chapter is required because API connects with the database and performs an operation on the database (DB). User cannot write a query on DB she/he might not be aware of what is the underlying database. So for uploaded data, we introduced the APIs, which communicate with the DB in the background. To know all operation, we required the knowledge of Django's **object-relational mapper (ORM)**. This chapter talks about the ORM of Django.

Structure

- The Django ORM
- Django shell
- Django database modeling
- Connect Django with various databases
 - Django connectivity with PostgreSQL
 - Django connectivity with SQLite
 - Django connectivity with MySQL

- Create app with model
- Access data through shell

Objective

This chapter covers the ORM definition and how ORM is useful for Django, Information of Django shell and its functionality, Django model creation, various database connectivity like PostgreSQL, SQLite, and MySQL model file configuration with the database. Throughout this chapter, it helps the reader to understand how we can communicate with a database and how Django operates the database.

The Django ORM

ORM is used for connecting the database and to connect application. It stored the relational data on the object and that object creates the layer upon the database.

In the Django we have default ORM. It provides the abstraction on the database, instead of writing the SQL queries, we can write the Django query. It means for fetching data we write a select query on the database, but in Django, we can write the programming query for fetching the same data. Its example is mentioned below.

I have one table `Employee` and I want to get the employee detail whose `employee_id` is `Emp001`.

Following is the query format:

```
Select * from Employee where employee_id = ' Emp001 ';
```

Now, for the same result in Django, the query will look like mentioned below:

```
emp_det = Employee.objects.filter(employee_id = ' Emp001 ')
```

In the above code, we are storing the result value inside the `emp_det` variable.

If we use Django ORMs, instead of SQL query on web application then it provides a much faster response. Django has support for multiple databases. Which are as like SQLite, MySQL, Oracle, and PostgreSQL. If we create the model file then with the minimum changes we can change our database from one to another. For example in production in the initial phase database was MySQL but due to some reason want to replace with PostgreSQL, then it will update into a `settings.py` file and a few lines of code into the model file.

So that is the beauty of Django's ORM.

Django shell

It is the most powerful feature of Django. In the common command shell, we can access our machine through Command Prompt. In the Django shell, we can access our whole project by the shell and we can also execute the command. If developers want to use the project file or any function from the file then by a shell he can access them. We can start the Django shell by the command:

`python manage.py shell`

We will get the following output:

```
Python 3.6.9 (default, Nov  7 2019, 10:44:02)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> Documents
```

Figure 5.1 (a): Django shell start screen

Django database modeling

In the Django models, it is the object, which mapped to the database. At the time of creating a Django model, it runs SQL and creates a new table in the database. In such conditions, it is not required to write a single line of SQL query. Django create the model name is such way `appname_tablename` for example my app name is `firstapp` and table name is `BankCustomer`. So it will create `firstapp_bankcustomer`. The model also links related information in the database. It is shown below in the following screenshot:



Model BankCustomer:		Table firstapp_bankcustomer
first_name		Id 1001
last_name		first_name Shayank
account_number		last_name Jain
address		account_number 450000001
aadharcard		address Mumbai
pancard		aadharcard 5674600001
phone		pancard ABCD01EF3
		phone 9000000000

Figure 5.2 (a): Django model creates a corresponding table in the database

Let's see the example of `model` file:

```
from __future__ import unicode_literals

from django.db import models
from datetime import datetime, timedelta

class BankCustomer(models.Model):
    first_name = models.TextField()
    last_name = models.TextField(default = None)
    account_number = models.TextField(default = None)
    address = models.TextField(default = None)
    adharcard = models.TextField(default = None)
    pancard = models.TextField(default = None)
    phone = models.TextField(default = None)
```

Figure 5.2 (b): Django model file screenshot

Let's see the other example to understand the models working better way. There is the code of `model.py` file and it contains the table name and its column name in the Django ORM format.

Following shown below code is the `model` file code:

```
from django.db import models

class Student(models.Model):

    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

In the above code, `Student` is the name of class and `first_name` and `last_name` is the variable, which stored the value. In the database, `Student` is the table name and `first_name` and `last_name` is the columns of the same table. The above `Student` model will create a database table as mentioned below:

```
CREATE TABLE myapp_student
(
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
```

```
);
```

This is a SQL query code, which is very commonly used to create the table.

Connect Django with various databases

Django provides the default ORM for some databases, which are as follows:

- PostgreSQL
- MySQL
- SQLite
- Oracle

These are the databases which have the official support of Django and many third-party packages are available, which can help us to connect with other databases.

The commonly used database with Django is PostgreSQL. We can also use MySQL but most people use PostgreSQL. In the default, SQLite is used in Django, but we cannot go production with SQLite database.

Oracle is also used with Django, but we should use this only when a big enterprise is required for the database. In simple words selection of the database is depending upon the developer choice, but in the industry, we have some standard to select database. These standards are as follows:

- The database should be selected on the basis of data size and no of transactions.
- Costing of the database maintenance.
- Available resource of database support.

Now we are going to connect Django with various databases.

Django connectivity with PostgreSQL

PostgreSQL is an open source relational database. Most of the company and developer prefer the Postgres with Django. Its response is faster and it has a tabular view. Django has default ORMs for PostgreSQL database. Please refer the following screenshot for database connectivity.

Following is the `settings.py` file screenshot, which has mentioned the connection string of the PostgreSQL database:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'Test_DB',
        'USER': 'db_user_name',
        'PASSWORD': 'db_password',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Figure 5.3.1 (a): PostgreSQL database connectivity in settings file

In the above screenshot, some parameters are defined and every parameter has special meaning. Here, **ENGINE** is the name of a used package and **NAME** is the name of our database, which we want to connect. The **USER** is a username, for database connection, we required a user, which have permission. The **PASSWORD** is the username password. Basically, the user and password parameter is used for authentication. **HOST** parameter is used to provide the IP of the database machine. Currently, my database is located on my local machine, so I am using localhost. IP value can be changed if DB is located on the server side. **PORT** parameter is used for giving the port to access the database.

Following is the code of the **model** file, It is created for SQLite database:

```
from __future__ import unicode_literals
# Create your models here.
from django.db import models

class Education_Center(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    subject_name = models.CharField(max_length=50)
```

In the above code, our table name is **myapp_eduction_center**. This model file is defined inside the **myapp**, so it creates the table name with the app name. It contains the field name, which is **first_name**, **last_name**, and **subject_name**.

Django connectivity with SQLite

In Django by default SQLite connectivity is available, but for reference please refer to the following screenshot:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Figure 5.3.2 (a): SQLite database connection

In the above screenshot, there are two parameters are available one is an ENGINE and second NAME. The ENGINE is a package, which is used for making a connection with the SQLite database. In the SQLite DB NAME, a parameter is used for setting the file location.

Following is the code of the `model` file, It is created for SQLite database:

```
from django.db import models

class Education_Center(models.Model):

    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    subject_name = models.CharField(max_length=50)
```

In the above code, our table name is `myapp_eduction_center`. This model file is defined inside the `myapp`, so it creates the table name with the app name. It contains the field name, which is `first_name`, `last_name`, and `subject_name`.

Django connectivity with MySQL

In Django, it provides the default ORM for MySQL database which makes faster query execution on the DB. It is an open source database and today's world it uses commonly.

Now let's have a look at the connection parameters of the MySQL database. Following is the `setting.py` file screenshot, which contains the MySQL database connectivity detail:

```

DATASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'Test_DB',
        'USER': 'db_user_name',
        'PASSWORD': 'db_password',
        'HOST': 'localhost', # Or an IP Address that your DB is hosted on
        'PORT': '3306',
    }
}

```

Figure 5.3.3 (a): MySQL database connectivity

In the above screenshot, we have mentioned variables, each parameter has special meaning. Here the `ENGINE` parameter is the name of the used package and `NAME` parameter is the name of our database, which we want to connect. The `USER` parameter is a username, for database connection, we required a user, which have permission. The `PASSWORD` parameter is the username password. Basically, the user and password parameter is used for authentication. The `HOST` parameter is used to provide the IP of a database machine. Currently, my database is located on my local machine, so I am using localhost. IP value can be changed if DB is located on the server side. `PORT` parameter is used for giving the port to access the database.

One more way is available to connect the MySQL database with Django ORMs. In a second way, we create the separate file, which contains the connection parameter details and we give the reference of that connection string file inside the `settings.py` file. In the *Figure-5.3.3 (b)* have shown the connection string file coding and *Figure-5.3.3 (c)* have shown the `settings.py` file screenshot.

Following is the screenshot of connection string file, which name is `db_configuration_file.cnf`:

```

db_configuration_file.cnf
=====
[client]
database = Test_DB
host = localhost
user = db_user_name
password = db_password
default-character-set = utf8

```

Figure 5.3.3 (b): db_configuration_file.cnf file screenshot

These parameters are the same as before described. Following is the screenshot of `settings.py` file:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'OPTIONS': {
            'read_default_file': 'db_configuration_file.cnf',
        },
    }
}
```

Figure 5.3.3 (c): Setting.py file screenshot

In the `settings.py` file new parameter is introduced, which is `OPTIONS`. In the new parameter, we defined our file location with a name to connect the database.

In the all database SQLite, MySQL, Postgres and Oracle have the same way to define table and field. In simple words, the way of defining table and field declaration will be the same for all the above defined databases. Only the connection string and parameter will be change. So that is the reason for production we can switch the database easily with minimum efforts.

Create app with model

For defining the `model` file, we need to create a new app. So I am taking the new project, whose name is `db_connection_test`. Its current architecture is mentioned below:

```
└── db_connection_test
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── manage.py
```

For creating the app, we are going to use command `python manage.py startapp db_model`. Our app name is `db_model` and let's see the architecture of the current project:

```
└── db_connection_test
    ├── __init__.py
```

```
|   └── __init__.pyc
|   ├── settings.py
|   ├── settings.pyc
|   ├── urls.py
|   └── wsgi.py
|
└── db_model
    ├── admin.py
    ├── apps.py
    ├── __init__.py
    ├── migrations
    │   └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
|
└── manage.py
```

The above mentioned is the new structure of our project. In the `db_model` folder have a `models.py` file. So inside the model file, we will write our table creation code. It is an auto-generated file, which generates at the time of app creation. Let's have a look how it looks like.

Following is the `model.py` file screenshot:

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.db import models

# Create your models here.
```

Figure 5.4 (a): Auto-generated model file screenshot

In the above screenshot, we have two default lines are written, these are packages. Which are used for model creation.

Access data through shell

As we read before in previous topics, the shell is a very strong concept in Django. It is useful when we want to debug our code. In the debugging we can check the database connectivity, data fetching and **CRUD** (Create, Read, Update, Delete) operations. We can also execute our function individually without executing the whole project. If developers want to change any defined functionality then they can add new code, without updating the project code. Shell makes coders life easy for debugging. Let's see the example that how the shell is working. I am going to use a shell for performing select, insert, update, and delete on Postgres database.

So we have one project its name is `db_connection_test` and it has a `db_model` app, which is shown in the previous topic structure. Now I am going to create the `model` file, which will help me to create the table and its columns.

Following is the `model.py` file screen shot, after writing the table and column creation code:

```
from __future__ import unicode_literals

from django.db import models

# Create your models here.
from django.db import models

class Restaurant(models.Model):
    dish_name = models.CharField(max_length=100)
    dish_category = models.CharField(max_length=50)
    price = models.CharField(max_length=20)
```

Figure 5.5 (a): Code of model file

In the above screenshot, we have one class that is `Restaurant`, whose name is `db_model_restaurant` for the database table. It has three variables, which is going to treat as a column name. The `dish_name` variable is going to store the name of our available dish, the `dish_category` variable will store the value, which defines the dish category like starter, main course, snacks, and many more. The `price` variable is used for storing the dish price.

To connect the database with Django project, we need to write connection string, so to connect Postgres I used the mentioned below configuration.

Following is the Postgres database connection string details, which I used:

```

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'test_db',
        'USER': 'postgres',
        'PASSWORD': 'postgres',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}

```

Figure 5.5 (b): Database connectivity code of settings file

In the above screenshot, the `settings.py` file code, which is mainly used for database connectivity.

To communicate with the database, we required the user, which have access for operation on the database. So we used the Postgres user. Postgres user is the default admin user of the PostgreSQL database, which have all activity rights.

I configured my PostgreSQL database on the local machine, so I am using the default user, but for production release, we should follow the standard and make the different user for communication. For creating the table, I made the new database, which name is `test_db`, that is shown above in the above screenshot with the against of NAME parameter.

Host and port are the same which we discussed in the previous topic. Our app is created and the model file is also defined. Now some small configuration is remaining, which are as follows:

Put the entry of our app `db_model` inside the `settings.py` file. Following is a screenshot of `settings.py` file `INSTALLED_APPS` section:

```

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'diango.contrib.staticfiles',
    'db_model',
]

```

Figure 5.5 (c): Installed_Apps section screen shot

In the above screenshot, we have shown our app entry, which is highlighted. The entry of our app is important; it helps the interpreter to find the `model` file. Other entries are auto-generated, which are written by Django.

Now, we created the `model` file, but it should also reflect in the database, so we have to migrate the file. For migration there are two steps, in the first step we should run the command `python manage.py makemigrations your_app_name` and the second command is `python manage.py migrate`.

Following is the output of command `python manage.py makemigrations db_model`:

```
Migrations for 'db_model':
  db_model/migrations/0001_initial.py
    - Create model Restaurant
```

Figure 5.5 (d): Output of the `makemigrations` command

It creates the migration file, inside the `db_model` app.

The second command will be the same as `python mange.py migrate` and its output is as follows:

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, db_model, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying db_model.0001_initial... OK
  Applying sessions.0001_initial... OK
```

Figure 5.5 (e): Output of the `makemigrations` command

It executes the migrations from every defined app, in the second line our app name is mentioned and remainings are the part of the `admin` module, which executes by default.

In the first place we have to execute the `makemigrations` command then `migrate`. It is sequential for model generation. The `makemigrations` command scope is only on app level but `migrate` command check all the app migration files then execute all file with the order. So it is important to execute the command sequentially.

Now we are going to start the shell by the command, `python manage.py shell`.

Following is the output which will generate the output on the console screen:

```
Python 3.6.8 (default, Jan 14 2019, 11:02:34)
[GCC 8.0.1 20180414 (experimental) [trunk revision 259383]] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)

>>>
```

Our configuration part is done successfully. Now let's perform the insert, select, update, and delete operations through, Django shell.

Insert query

Currently our table is blank and for inserting data we will write the Django ORM code on the shell. Following is the code of data insert into our table:

```
from db_model.models import Restaurant
data = Restaurant(dish_name='Chola Bhatura', dish_category='Snacks',
price=120)
data.save()
```

In the above query, we have imported the model by writing the `db_model.models import Restaurant`. Then we will create the object of our query that is `data`, in our code. We are assigning a value to the columns, which are `dish_name`, `dish_category`, and `price`. In the end, we save the `insert` command output or in the database language, I commit the query for permanent save.

Following is the screenshot of our query:

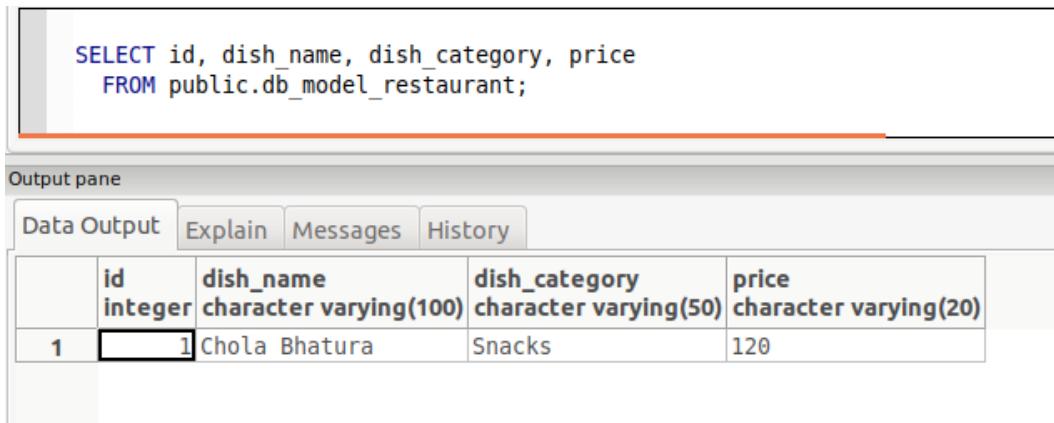
```
>>>
>>> from db_model.models import Restaurant
>>> data = Restaurant(dish_name='Chola Bhatura', dish_category='Snacks', price=120)
>>> data.save()
>>>
>>>
```

Figure 5.5 (f): Query on the shell

Now, let's see the output of our query, I used the PgAdmin-III to see out PostgreSQL table.

PgAdmin-III: It is a GUI tool, with the help of PgAdmin, we can see our table. We can run select, update, insert, and delete on the database.

I fired the select query on the PostgreSQL database through PgAdmin-III, which produce the mentioned below output:



The screenshot shows the PgAdmin-III interface with a query window containing the following SQL code:

```
SELECT id, dish_name, dish_category, price
FROM public.db_model_restaurant;
```

Below the query window is an "Output pane" tab bar with four tabs: Data Output (selected), Explain, Messages, and History.

The "Data Output" tab displays a table with the following data:

	id integer	dish_name character varying(100)	dish_category character varying(50)	price character varying(20)
1	1	Chola Bhatura	Snacks	120

Figure 5.5 (g): Query on the PgAdmin-III interface with output

I execute a select query on the table `db_model_restaurant` and it gives the same result, which I inserted through the shell. That means our ORM command executed successfully.

Select query

Now we are going to write select query through ORM. I think for executing select we required more data to show. I inserted the more records in the table so we have four entries. Following is the code of select query Django ORMs:

```
from db_model.models import Restaurant
all_entries = Restaurant.objects.all()
for tbl_value in all_entries.values():
    print("Value of our table: ", tbl_value)
```

In the above query, we have imported the model from the `model.py` file, which name is `Restaurant`. Then use the `all()` method, which behaves like * of database select query. Its output comes in the object format and I stored them into the `all_entries` variable. The converted object into a simple readable format, I used the `value()` method. To print the output value, I am using a `for` loop, which will produce a single value at a time on the console.

Following the output of our select query:

```
>>> from db_model.models import Restaurant
>>> all_entries = Restaurant.objects.all()
>>> for tbl_value in all_entries.values():
...     print("Value of our table: ",tbl_value)
...
('Value of our table: ', {'price': u'120', 'dish_category': u'Snacks', 'id': 1, 'dish_name': u'Chola Bhatura'})
('Value of our table: ', {'price': u'80', 'dish_category': u'Snacks', 'id': 2, 'dish_name': u'Masala Dosa'})
('Value of our table: ', {'price': u'150', 'dish_category': u'Rice', 'id': 3, 'dish_name': u'Fried Rice'})
('Value of our table: ', {'price': u'200', 'dish_category': u'Main Course Dishes', 'id': 4, 'dish_name': u'Sahi Paneer'})
>>>
```

Figure 5.5 (h): Django ORM select query output

It's our query output, which is printed line by line. I used a simple single query in the above, let's create the select query with a `where` condition in Django ORMs. In the Django ORM, replacement of `where` clause is predefined functions.

Following is the code of select with `where` clause:

```
from db_model.models import Restaurant
result = Restaurant.objects.filter(dish_category="Snacks")
for data in result.values():
    print(data)
```

All code is the same only change in the mentioned below the line:

```
Restaurant.objects.filter(dish_category="Snacks")
```

In the above line, we use the `filter` method and take the column name, which has a condition to fetch the data. There is one more way to call the same query and it will produce the same output as previously written query:

```
from db_model.models import Restaurant
result = Restaurant.objects.all().filter(dish_category="Snacks")
for data in result.values():
    print(data)
```

In the above query, only one small change is there, we called the `object.all()` method then `filter()`. It will execute in the same way, first, it will fetch all records then apply a `filter` on that. In the last, it prints the output on the screen.

Following is the output of the select query:

```

>>> from db_model.models import Restaurant
>>> result = Restaurant.objects.filter(dish_category="Snacks")
>>> for data in result.values():
...     print(data)
...
{'price': u'120', 'dish_category': u'Snacks', 'id': 1, 'dish_name': u'Chola Bhatura'}
{'price': u'80', 'dish_category': u'Snacks', 'id': 2, 'dish_name': u'Masala Dosa'}
>>>
>>> from db_model.models import Restaurant
>>> result = Restaurant.objects.all().filter(dish_category="Snacks")
>>> for data in result.values():
...     print(data)
...
{'price': u'120', 'dish_category': u'Snacks', 'id': 1, 'dish_name': u'Chola Bhatura'}
{'price': u'80', 'dish_category': u'Snacks', 'id': 2, 'dish_name': u'Masala Dosa'}
>>>
>>>

```

Figure 5.5 (i): Django ORM select with where query output

In the above screenshot, we had two highlighted blocks, which are two different ways of writing the query. Both queries produce the same output, which is mentioned below the query in the same *figure*.

There is one more way to call the select query, now I am going to use the `get()` method for fetching the data. This time I will call the data on the basis of record primary key value and its short form is `pk`.

Following is the code of `get()` method, which is used to get the data from the database:

```

from db_model.models import Restaurant
one_entry = Restaurant.objects.get(pk=3)
print(one_entry.id, one_entry.dish_name, one_entry.price)

```

We used the `get()` method with the parameter `pk` and its value is 3. Its result is stored into the `one_entry` variable and a value type is an object. So we print the value by calling the object with the column name. In `get()` method not found the `pk` value in the database then it will raise a `DoesNotExist` exception.

Following is the output of the `get()` method code:

```

>>>
>>> from db_model.models import Restaurant
>>> one_entry = Restaurant.objects.get(pk=3)
>>> print(one_entry.id, one_entry.dish_name, one_entry.price)
(3, u'Fried Rice', u'150')
>>>

```

Figure 5.5 (j): Django ORM get() method query output

Update query

In Django, we can update field value like SQL database. We will use the `filter` method for fetching the specific record and `update()` method to update the record.

Following is the Django ORM update query:

```
from db_model.models import Restaurant
data = Restaurant.objects.filter(pk=3).update(price=180)
```

In the above code, we used the `pk` for fetching that record, whose ID is 3. By using the `update` method, we changed the `price` column value.

Following the output of update query, from the PgAdmin-III:

	id integer	dish_name character varying(100)	dish_category character varying(50)	price character varying(20)
1	2	Masala Dosa	Snacks	80
2	4	Sahi Paneer	Main Course Dishes	200
3	1	Chola Bhatura	Snacks	150
4	3	Fried Rice	Rice	180

Figure 5.5 (k): Django ORM Update query output from PgAdmin-III.

In the above screenshot, we have highlighted block, which shows that `price` value is updated.

Delete query

In Django, we can delete records as well to delete the record. We will use the `filter` method for fetching the specific record and `delete()` method to remove the record from the database.

Following the Django ORM delete query:

```
from db_model.models import Restaurant
Restaurant.objects.filter(pk=4).delete()
```

Following the output of delete query, from the PgAdmin-III:

```

SELECT id, dish_name, dish_category, price
FROM public.db_model_restaurant;

```

	id integer	dish_name character varying(100)	dish_category character varying(50)	price character varying(20)
1	2	Masala Dosa	Snacks	80
2	1	Chola Bhatura	Snacks	150
3	3	Fried Rice	Rice	180

Figure 5.5 (l): Django ORM delete query output from PgAdmin-III.

So, we have seen the insert, select, update, and delete Django ORM query. These are the basic way to do a query; we can do in a more advanced way. In more detail, the reader can refer to the official Django documentation online.

Conclusion

This chapter covered the Django object-relational mapper. Connectivity with various databases and performs the operation on the table like create, read, update, and delete by Django models. We learnt how to create the `model` file creation and its configuration. All of them are covered with example code or related screenshot, which makes easier to understand the user.

In the next chapter, the reader will gain knowledge of Django APIs deployment on a web server and steps of application deployment.

Questions

1. What is ORM?
2. How to shell?
3. How Django shell different from the normal shell?
4. How to connect Django with PostgreSQL database?
5. How to connect Django with MySQL database?
6. How to create the model file?
7. What is `makemigrations` and why we use that command?
8. What is `migrate` and why we use that command?

9. What is the difference between `makemigrations` and `migrate`?
10. How we can use Django shell?
11. How to connect database table with Django shell?
12. How to update the db table with Django shell?

CHAPTER 6

First Django API Deployment on Web

Django is the web framework of Python, which is used for web development. We are using Django for API development. So it is incomplete without API deployment on web servers. The purpose of including this chapter is to know about web technologies, which are used with Django and are available nowadays. We are covering the basic information of uWSGI, Apache, NGINX, Gunicorn, and Supervisor.

Structure

- The introduction of web technologies
 - uWSGI
 - NGINX
 - Apache
 - Gunicorn
 - Supervisor
- API deployment on web servers with Apache
- Django loggers configuration and use case

Objective

In this chapter, we will deploy our Django application on the web server by using the Apache server. It covers the basic idea of following technology uWSGI, NGINX, Gunicorn, Supervisor, and Apache. The information on the technology mentioned above is required because they help us in the application deployment. For code

debugging, we have implemented the logger and also defined the steps to debug the code, when issues arise in our project.

The introduction of web technologies

To deploy a Django application on the web, we require some technology, which can communicate with the outer world and server. UWSGI and Gunicorn are these techs, which manage the server and communicate with a web server like Apache or NGINX.

Let's see these technologies in brief.

uWSGI

It stands for **Web Server Gateway Interface (WSGI)**. It is used for serving Python web applications, which work directly and smoothly with web servers like NGINX. uWSGI is an application server, which supports various programming languages like C, C++, Objective-C, and Python. We can write a plugin in the language mentioned above for application hosting. It is good in performance because it utilizes the low-resource for usage. Reliability is the strength of the project.

There are some components, which are included in the uWSGI:

- Its core functionality is to implement configuration with the various languages.
- It has an in-built processes management system, which can handle the loads automatically.
- It has the functionality of sockets creation, process monitoring, log creation, and shared memory areas.
- It has available request plugins, which helps to implement application server interfaces for various languages and platforms: WSGI, PSGI, Rack, Lua, WSAPI, CGI, PHP, Go, and many more.
- We can implement load balancers, proxies, and routers with the uWSGI server.
- In the uWSGI *Emperor* mode is available, which helps to implement massive instances management and monitoring.

Let's see how to install uWSGI. We can install uWSGI in various ways. It requires Python and to execute uWSGI with Python. It has some dependent packages. To install Python, use the following command:

```
sudo apt-get install python3-dev
```

Then use the mentioned below command for PIP:

```
sudo -H pip3 install uwsgi
```

Now, to install only uWSGI:

```
sudo apt-get install uwsgi
```

To check the uWSGI installation, we can run mentioned below command:

```
uwsgi --version
```

It will give you the version as a response.

Through network installer:

```
curl http://uwsgi.it/install | bash -s default /tmp/uwsgi
```

uWSGI is installed, now deploy the application on the uWSGI. In the first step, we will create a Python application.

Following is the code of Python,

```
def application(env, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return [b "Hello World"]
```

Save the file with the name `test.py`. To deploy this file on the uWSGI, we will execute the command:

```
uwsgi --http :9090 --wsgi-file foobar.py
```

In the above way, we can deploy a single file on the uWSGI. Next step is to deploy the whole Django project with the uWSGI, which we will cover in the upcoming topics.

NGINX

It is open-source software, which is used for serving request and response on the web. It has in-built functionality like reverse proxying, caching, load balancing, and many more. It has the capability of handling high load on servers and gives the maximum utilization and server stability.

It has a modular, event-driven, asynchronous, single-threaded architecture that scales extremely well on generic server hardware and across multi-processor systems that is why it is faster than other servers.

The following are the advantages of the NGINX web server:

- It is easy to install the NGINX web server and configure it with the application.
- It is faster and performs well with the static files.
- If we compared Apache with NGINX, four times more concurrent connections can be handled by the NGINX.

- It is compatible with many applications, which are commonly used nowadays like WordPress, Ruby, Python, Joomla, Drupal, vbulletin, PhpBB, xenforo, and many more.
- It has the feature of load balancing support, which makes it on demand. We can deploy NGINX for load balancing on the server.
- It is one of the web servers, which makes websites faster. Many of the companies like Yahoo, YouTube, Pinterest, Instagram, WordPress.com, and Tumblr use NGINX to manage traffic because they want to achieve the maximum website speed.

The common command for installing the NGINX on Ubuntu is as follows:

```
sudo apt update
```

```
sudo apt install nginx
```

Configuration of the NGINX we will cover in the upcoming topics.

Apache

It is a web server like NGINX. Apache use count is higher worldwide. It is developed and maintained by the Apache Software Foundation. It is also an open-source web server. Users can customize the server as per the requirement by using available extensions and modules.

The following are the advantages of Apache web server:

- Many modules are available, which helps the user to modify and adjust the code and also to fix errors.
- It is open-source, so no license is required.
- It is easy to install.
- If we deploy any changes on the project, it reflects directly without restarting the server.
- It is available for many operating systems like Windows, Linux, and more.
- It gives regular updates for maintenance.
- The documentation of Apache is very easy and useful.

The following table describes the difference between Apache and NGINX web servers:

Apache	NGINX
It follows the approach of multi-threaded, which helps to handle the client requests concurrently.	It follows the approach of event-driven to handle client requests.

Apache	NGINX
This server is an open-source HTTP server.	It provides the high-performance to the server, and it is asynchronous and reverse proxy server.
It can handle the content dynamically, which means without restarting the server, it reflects the changes on the server.	It requires the server restart.
If the server has heavy traffic, then it creates problems to handle multiple requests.	It can handle the heavy traffic by processing the multiple client requests concurrently.
It is flexible because we can add or remove the modules dynamically.	It does not work dynamically.
It processes one connection with a single thread.	It processes the multiple connections with a single thread.

Table 6.1: Difference between Apache and NGINX

Gunicorn

It is a WSGI server, which is developed for interacting with multiple web servers. It is not important for Gunicorn that which programming language we used to develop web applications, till it uses the WSGI interface. It is a stable, commonly-used for web app deployment.

It is an interface between the web server and web application, which can handle all requests and communication. If we develop our web application in Django, then it provides the following solutions:

- It can communicate with multiple web servers
- It managed the heavy load and can handle multiple requests.
- It runs and makes them alive in multiple processes.

The following are the advantages of Gunicorn:

- It is used as a load balancer in app servers.
- We can use it as a communicator who can communicate between the services.
- It handles the large numbers of connections with the utilization of small CPU and memory.

Supervisor

A Supervisor is a tool for monitoring, if we configure our code in that way, which can monitor the processes. It is very helpful in the production environment because sometimes it happens to our servers that are required to reboot. So, in such conditions, it restarts the process, if due to any reasoning process dies or gets stopped. It is used with Django.

Use case of Supervisor

In production, we used the Gunicorn as the web server. There was a scene when my Gunicorn process was running on a server, and my site was running properly. After a few days, it was reviewing the `502 Bad Gateway` error, without changing on the server-side. After debugging I found Gunicorn is not in running state that means the Gunicorn process died on its own. So to prevent this problem, I used Supervisor, the Supervisor is used to getting control of the Gunicorn process. It has the functionality to check whether Gunicorn demon is alive or dead if it found the Gunicorn process dies then it starts again.

There is another scenario when I want to use celery (a Python worker) on the production server. I cannot use to run in the foreground because if I get the log out from the server then it will stop the celery. So we are going to use Supervisor, which makes it easy to use celery.

API deployment on web servers with Apache

We will start with a new project, and our new project name is `firstapi`. Its structure is mentioned below:

```
firstapi
├── firstapi
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
```

To create the API we required the database in the background for fetching the data. So we are going to use the default database, which is SQLite:

```
# Database
# https://docs.djangoproject.com/en/1.11/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Figure 6.1(a): Used SQLite connection

Database settings are done and for API development, we require the new app. So we created a new app with the name `app01`. Now our application structure will look like as:

```
firstapi/
    └── app01
        ├── admin.py
        ├── apps.py
        ├── __init__.py
        ├── migrations
        │   └── __init__.py
        ├── models.py
        ├── tests.py
        └── views.py
    └── db.sqlite3
    └── firstapi
        ├── __init__.py
        ├── __init__.pyc
        ├── settings.py
        ├── settings.pyc
        ├── urls.py
        ├── urls.pyc
        ├── wsgi.py
        └── wsgi.pyc
└── manage.py
```

Inside the `app01/views.py` file, I will write the get API code, which is as follows:

```
from django.http import HttpResponse
from django.shortcuts import render
import json
from django.views.decorators.csrf import csrf_exempt

import logging
logger = logging.getLogger(__name__)

def get_req(request):
    return HttpResponse("This is my get request example.")
```

Figure 6.1(b): Views file screenshot

In the `views` file, we create the get API and configure that into `urls` file, which is shown in the below screenshot. This is `firstapi/firstapiUrls.py` file code is as follows:

```
"""firstapi URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/1.11/topics/http/urls/
Examples:
Function views
    1. Add an import: from my_app import views
    2. Add a URL to urlpatterns: url(r'^$', views.home, name='home')
Class-based views
    1. Add an import: from other_app.views import Home
    2. Add a URL to urlpatterns: url(r'^$', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.conf.urls import url, include
    2. Add a URL to urlpatterns: url(r'^blog/', include('blog.urls'))
"""
from django.conf.urls import url
from django.contrib import admin
from app01.views import get_req

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url('get_api/', get_req),
]
```

Figure 6.1(c): Urls file screenshot

To make a post request we have to create a table in the database then we will fetch the data through post request. So I am going to create a `model` file then insert data into the table.

Following is the `model` file screenshot:

```

from __future__ import unicode_literals

from django.db import models

# Create your models here.
from django.db import models

class student_data(models.Model):

    student_id = models.CharField(max_length=15)
    first_name = models.CharField(max_length=100)
    standard = models.CharField(max_length=20)

```

Figure 6.1(d): Model file screenshot

Our table **student_data** is created, so I am going to insert the data into the table and I am using the Django shell for inserting the data, I shared the information of Django shell in-depth in the previous chapter.

Following is the code for inserting the data:

```

>>> from app01.models import student_data
>>> st = student_data(student_id = "st01",first_name = "shayank",
standard = "fifth")
>>> st.save()
>>> st = student_data(student_id = "st02",first_name = "mayank", standard
= "sixth")
>>> st.save()
>>> st = student_data(student_id = "st03",first_name = "dolly", standard
= "first")
>>> st.save()

```

In the first query, we import the model and table name. In the second line, we inserted the data. To commit the data, we have used the `save()` function.

Now, to check that data is available or not, the code is mentioned below for checking the data:

```

>>> from app01.models import student_data
>>> student_data_obj = student_data.objects.all()
>>> for tbl_value in student_data_obj.values():

```

```
print("Value of our table: ",tbl_value)
```

We will get the following output:

```
('Value of our table: ', {'student_id': u'st01', 'first_name': u'shayank', u'id': 1, 'standard': u'fifth'})  
(('Value of our table: ', {'student_id': u'st02', 'first_name': u'mayank', u'id': 2, 'standard': u'sixth'})  
(('Value of our table: ', {'student_id': u'st03', 'first_name': u'dolly', u'id': 3, 'standard': u'first'})
```

In the first line, we import the model and table. For select query we used the `student_data.objects.all()` command. To fetch the data we use `print` statements. For fetching the data, we will create the `post` method.

Following is the `Post` method code:

```
| from django.http import HttpResponse
| from django.shortcuts import render
| import json
| from django.views.decorators.csrf import csrf_exempt
| from app01.models import student_data

def get_req(request):
|     return HttpResponse("This is my get request example.")

def data_get(st_id):
|     st_data = student_data.objects.filter(student_id = str(st_id))
|     for tbl_value in st_data.values():
|         return tbl_value

@csrf_exempt
def post_req(request):
    student_id = request.POST.get("Student_Id")
    resp = {}
    if student_id == '':
        resp['status'] = 'Failed'
        resp['status_code'] = '400'
        resp['result'] = 'None'
    else:
        resp['status'] = 'Success'
        resp['status_code'] = '200'
        resp['data'] = {}
        stud_data = data_get(student_id)
        if stud_data:
            resp['data'] = stud_data
    return HttpResponse(json.dumps(resp), content_type = 'application/json')
```

Figure 6.1(e): Post method definition screenshot

In the above screenshot, we have marked sections, which has we import the model file. In the second marked section, there is a select query, which fetches the data. In the third section, the `post` method is defined. `Post_req` is a function name, which we called the `data_get` function that is fetching the data from the database and in

response, sending fetched data.

Following is the `Urls` file code, which has mentioned the second url that is `post`:

```
from django.conf.urls import url
from django.contrib import admin
from app01.views import get_req,post_req

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url('get_api/', get_req),
    url('post_api/', post_req),
]
```

Figure 6.1(f): Urls file screenshot

We defined the URL for the Post method. To test the Post method, I use the Postman application for sending the request. Currently, we execute our Django project on the default server. The command is as follows:

```
python manage.py runserver
```

Following is the Postman testing screenshot:

Key	Value
<input checked="" type="checkbox"/> Student_Id	st02

```

1  {
2      "status": "Success",
3      "status_code": "200",
4      "data": [
5          {
6              "student_id": "st02",
7              "first_name": "mayank",
8              "id": 2,
9              "standard": "sixth"
10         }
11     ]
12 }
```

Figure 6.1(g): Postman API testing screenshot

In the above screenshot, URL is marked, which is used for testing that is `http://127.0.0.1:8000/post_api/`. In the post request, we used the `Student_Id` as key and `st02` its value. In the third section, its output is mentioned.

Now, we are going to deploy our project on a web server, which requires the web server to be installed. I am using the Apache web server. Following are the steps to install Apache on the Ubuntu machine.

If Django using Python 2, then commands are:

```
sudo apt-get update  
sudo apt-get install python-pip apache2 libapache2-mod-wsgi
```

If Django using Python 3, then commands are:

```
sudo apt-get update  
sudo apt-get install python3-pip apache2 libapache2-mod-wsgi-py3
```

Apache server is installed, and we have to follow some steps to configure the file.

We have to set `STATIC_ROOT` in the setting file to define the directory to store the static files. So in the first step, we will set the `STATIC_ROOT` at the end of our `settings.py` file. Let's see it in a step-by-step manner:

1. Add the mentioned below the line:

```
STATIC_ROOT = os.path.join(BASE_DIR, "static/")
```

2. Execute the following command:

```
python manage.py collectstatic
```

3. Configure the Apache server with Django project, Apache communicates with Django by using the `mod_wsgi` module. To configure the WSGI, we run the following command:

```
sudo nano /etc/apache2/sites-available/000-default.conf
```

It will open the file, and we have to write the following code.

```
<VirtualHost *:80>
```

```
    . . .
```

```
        Alias /static /home/user/myproject/static  
        <Directory /home/user/myproject/static>  
            Require all granted  
        </Directory>
```

```
<Directory /home/user/myproject/myproject>
```

```
<Files wsgi.py>
    Require all granted
</Files>
</Directory>

WSGIProcess myproject python-path=/home/user/myproject
WSGIProcessGroup myproject
WSGIScriptAlias / /home/user/myproject/myproject/wsgi.py

</VirtualHost>
```

In the code mentioned above snippet, `myproject` will be replaced with our project name that is `firstapi`.

In the directory path, we will replace that with our current project location, which is `/home/ubuntu/firstapi`. After doing all the mentioned changes, we need to run some commands, which provide access permission to our project.

4. The following command will execute for giving permission to access our database file:

```
chmod 664 ~/firstapi/db.sqlite3
```

5. The below command will change the user after this command user who can access the data is `www-data`:

```
sudo chown :www-data ~/firstapi/db.sqlite3
```

6. The following command will change the user to access our project; the new user will be `www-data`:

```
sudo chown :www-data ~/firstapi
```

7. The following command will restart the Apache server:

```
sudo service apache2 restart
```

8. Now the server is configured and is in running state, so we are going to check our server through Postman application. Our Django application is running on Apache, and the new URL for the API is `http://127.0.0.1/post_api/`. Following is the Postman test screenshot, where we are sending a request to the server:

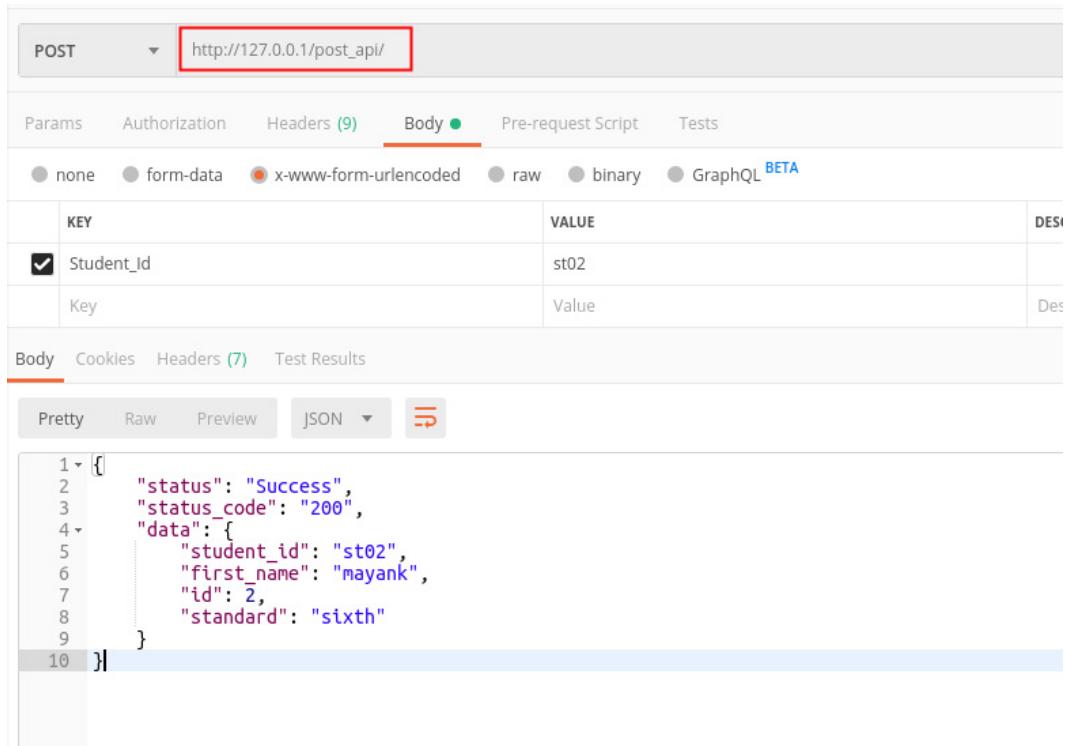


Figure 6.1(h): API response from Apache

Django loggers configuration and use case

I covered the In-depth knowledge of Django loggers in the previous chapters. Whenever we deploy the project, there might be a possibility that the project may occur the run time error during the user input or any running errors. So to track them, we required the loggers. It will not handle the error, but it will write the error in a separate file, so it makes it easy for the programmer to track the issue.

There was a scenario where I found loggers are important in project deployment. My site was deployed on production and it was running very well. One day the user sends the request to the server through a post method for fetching the data of particular id. It goes to a server but due to some reason, it failed to send a response against the request. I used the loggers in my project. When I checked into the loggers file, I found when the user sent the request that time our database was down, due to that reason our query was failing to give a response.

So it was easy for me to find the error because of loggers to track the particular request.

Now, I am going to configure loggers in the project for tracking the user requests. Following is the loggers code, which we are going to write inside the `settings.py` file at the end of the file:

```
LOGGING = {  
    'version': 1,  
    'disable_existing_loggers': False,  
    'formatters': {  
        'verbose': {  
            'format' : "[%(asctime)s.%(msecs)03d] %(levelname)s [%(name)s  
s:%(lineno)s] %(message)s",  
            'datefmt' : "%d/%b/%Y %H:%M:%S"  
        },  
        'simple': {  
            'format': '%(levelname)s %(message)s'  
        },  
    },  
    'handlers': {  
        'file': {  
            'level': 'INFO',  
            'class': 'logging.handlers.TimedRotatingFileHandler',  
            'filename': 'log/firstapi_application.log',  
            'when' : 'D',  
            'interval' : 1,  
            'backupCount': 40,  
            'formatter': 'verbose'  
        },  
    },  
    'loggers': {  
        'django': {  
            'handlers':['file'],  
            'propagate': True,  
            'level':'INFO',  
        }  
    }  
}
```

```
    },
    'app01': {
        'handlers': ['file'],
        'level': 'INFO',
    },
}
}
```

In the above code snippet, we have used the 'filename': 'log/firstapi_application.log', which will generate the log folder inside our project then `firstapi_application.log` file. The code mentioned above will write the log inside our log file. That is the first part now; we also have to include these settings in our `views` file code.

Following is the code which we are going to include in our `views.py` file code:

```
from django.http import HttpResponse
from django.shortcuts import render
import json
from django.views.decorators.csrf import csrf_exempt
from app01.models import student_data

# This part of code is used for importing the loggers and it will help to
# write the code.

import logging
logger = logging.getLogger(__name__)

def get_req(request):
    return HttpResponse("This is my get request example.")

def data_get(st_id):
    # This way we can write the loggers in the file.
    logger.info('Inside the data_get function.....')
    st_data = student_data.objects.filter(student_id = str(st_id))
    for tbl_value in st_data.values():
        return tbl_value
```

```

@csrf_exempt
def post_req(request):
    student_id = request.POST.get("Student_Id")
    logger.info('Inside the post_req function.....')
    resp = {}
    if student_id == '':
        resp['status'] = 'Failed'
        resp['status_code'] = '400'
        resp['result'] = 'None'
    else:
        resp['status'] = 'Success'
        resp['status_code'] = '200'
        resp['data'] = {}
        stud_data = data_get(student_id)
        if stud_data:
            resp['data'] = stud_data
    return HttpResponse(json.dumps(resp), content_type = 'application/json')

```

The above code snippet, is our whole code of `views.py` file and we also include the new lines, which imports the loggers setting and way of writing the code.

Loggers are implemented, so let's test our code. I put all the loggers in the `post` method so it will show us the flow of our program in the project.

Following is the output of our `log` file, which is `frstapi_application.log`:

```
[09/Jul/2019 18:09:54.318] INFO [app01.views:22] Inside the post_req function.....
[09/Jul/2019 18:09:54.318] INFO [app01.views:14] Inside the data_get function.....
[09/Jul/2019 18:09:54.320] INFO [django.server:124] "POST /post_api/ HTTP/1.1" 200 129
```

Figure 6.2(a): log file screenshot

Now, the entire code is developed, and all steps are mentioned to deploy Django API on the web servers.

Conclusion

This chapter covered the introduction of various technologies like uWSGI, Apache, NGINX, Gunicorn, and Supervisor. We also deployed our Django project on the Apache web server.

In this chapter, we have seen how Django logger is helpful for us.

In the next chapter, the reader will gain knowledge of Django deployment on various servers like NGINX, uWSGI, and Gunicorn.

Questions

1. What is uWSGI?
2. What is NGINX?
3. What is Gunicorn?
4. What is Apache?
5. Difference between NGINX and Apache?
6. Difference between uWSGI and NGINX?
7. Why are loggers important?
8. How to configure loggers?

CHAPTER 7

Django Project Deployment on Various Web Servers

In this chapter, we will learn how to deploy Django APIs on various web servers. It covers the deployment of Django project with NGINX, Gunicorn, Supervisor, and uWSGI servers.

Structure

- New Django project creation and its overview
 - PostgreSQL database connectivity
 - Model file creation through Django ORM
 - Get and post request creation with ORM queries
 - Loggers implementation in the project
- Deploy Django with NGINX, PostgreSQL, and uWSGI
- Deploy Django with NGINX, PostgreSQL, Gunicorn, and Supervisor.

Objective

This chapter will be covered the PostgreSQL database connectivity and operations on a database with Django ORMs. Django project deployment with NGINX, uWSGI, Gunicorn, and Supervisor servers. We are including this chapter in the book because nowadays, Django projects are deployed with the same technologies, and we will

use the same deployment process with microservices. At the end of this chapter, the user can easily understand the Django project deployment process.

New Django project creation and its overview

For the deployment, I am going to use the new project, which has PostgreSQL as a database, and we are using the ORM for data manipulation. To perform an operation on the project, we are using the post, get request, and loggers for tracking the flow of our project. Our new project name is `testproj`, and its structure is mentioned below:

```
testproj
└── app
    ├── admin.py
    ├── admin.pyc
    ├── apps.py
    ├── __init__.py
    ├── __init__.pyc
    └── migrations
        ├── 0001_initial.py
        ├── 0001_initial.pyc
        ├── __init__.py
        └── __init__.pyc
    ├── models.py
    ├── models.pyc
    ├── tests.py
    ├── views.py
    └── views.pyc
├── manage.py
└── query.sql
└── testproj
    ├── __init__.py
    ├── __init__.pyc
    └── settings.py
```

```
|── settings.pyc
|── urls.py
|── urls.pyc
|── wsgi.py
└── wsgi.pyc
```

PostgreSQL database connectivity

To connect with the database, we are going to create a new user and database for further process. So I am going to log in on the server by command:

```
sudo -u postgres psql
```

Then it will produce the following output:

```
psql (9.5.17)
Type "help" for help.
postgres=#
```

After login, we are going to create the new database and query will look like as follows:

```
postgres=# CREATE DATABASE restaurant;
CREATE DATABASE
postgres=#

```

Now, the database is created. So we are going to create a new user, which is `testuser`. For creating the user, we are going to execute the following query:

```
postgres=# create user testuser with password 'test123';
CREATE ROLE
postgres=# grant all on database restaurant to testuser;
GRANT
postgres=# \q
```

The `grant` command will give access to the particular user and we used them all, which means it will give all the admin access to the user. Our database name is a `restaurant`, and the username is `testuser`. The `\q` parameter will quit us from the database. Our database and user is created. Now we are going to configure our application with a PostgreSQL database.

Following is a screenshot of database connectivity:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'restaurant',
        'USER': 'testuser',
        'PASSWORD': 'test123',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

Figure 7.1.1 (a): PostgreSQL connection string

Model file creation through Django ORM

For data manipulation, we are going to use Django ORM. So following is the models file code:

```
from __future__ import unicode_literals
from django.db import models

class food_order(models.Model):
    category = models.CharField(max_length=50)
    item = models.CharField(max_length=100)
    price = models.CharField(max_length=20)

    def __str__(self):
        return '%s %s %s' % (self.category, self.item, self.price)
```

In the above code snippet, we have created the three columns, that is, `category`, `item`, and `price`. In the `__str__` function, it will return the `category`, `item`, and `price`.

For migration, we are going to execute the following commands:

```
python manage.py makemigrations app
```

```
python manage.py migrate
```

After that, all changes are going to reflect in the database. For testing, I inserted some data into a table. I used the Django shell for data upload.

Following is the code for uploading the data.

```
python manage.py shell
```

```

>>> from app.models import food_order
>>> st = food_order(category = "Rice",item = "veg biryani", price =
"150")
>>> st.save()

>>> st = food_order(category = "Rice",item = "kashmiri pulao", price =
"130")
>>> st.save()

>>> st = food_order(category = "Chiness",item = "Hakka noodles", price =
"120")
>>> st.save()

>>> st = food_order(category = "Sabji Khajana",item = "Shahi Paneer",
price = "200")
>>> st.save()

```

To check that data is uploaded or not, we will execute the select query on pgAdmin -III. Following is a select query output screenshot from the database:

	id [PK] serial	category character varying(50)	item character varying(100)	price character varying(20)
1	1	Rice	veg biryani	150
2	2	Rice	kashmiri pulaav	130
3	3	Chiness	Hakka nodels	120
4	4	Sabji Khajana	Shahi Paneer	200

Figure 7.1.2 (a): Data view from PostgreSQL

In the above screenshot, we have executed the `SELECT id, category, item, price FROM public.app_food_order;` query.

Get and post request creation with ORM queries

I created the `get` and `post` method for data manipulation with the database.

Following is the `views.py` file code:

```
# -*- coding: utf-8 -*-
```

```
from __future__ import unicode_literals
from django.http import HttpResponseRedirect
from django.shortcuts import render
import json
from django.views.decorators.csrf import csrf_exempt
from app.models import food_order

import logging
logger = logging.getLogger(__name__)

## This is used for fetching data from database.
def fetch_data_db(item_name):
    logger.info('Inside the fetch_data_db function.....')
    # This is our data select query.
    food_data = food_order.objects.filter(item = item_name)
    for tbl_value in food_data.values():
        return tbl_value

## This is used for inserting data into database.
def insert_data_db(item_name,item_category,item_price):
    logger.info('Inside the insert_data_db function.....')
    # This is our data insert query.
    rest = food_order(category = item_category, item = item_name, price
= str(item_price))
    rest.save()
    return 1

## It is used for inserting the data through user request.
@csrf_exempt
def post_req(request):
    item_name = request.POST.get('Item_name','')
    item_category = request.POST.get('Item_category','')
```

```
item_price = request.POST.get('Item_price','')
resp = {}
# This loop will check data is send in key or not.
if item_name == '' or item_category == '' or item_price == '':
    resp['status'] = 'Failed'
    resp['status_code'] = '400'
    resp['result'] = 'None'
else:
    resp['status'] = 'Success'
    resp['status_code'] = '200'
    resp['data'] = {}
    rest_data = insert_data_db(item_name,item_category,item_price)
    # If rest_data not black than if part will run.
    if rest_data:
        resp['data'] = "Data is added successfully."
    # If rest_data is black than else part will run.
    else:
        resp['status'] = 'Failed'
        resp['status_code'] = '400'
        resp['result'] = 'None'

    return HttpResponse(json.dumps(resp), content_type = 'application/json')

## It is used for getting the data from the database through user
request.

def get_req(request):
    item_name = request.GET.get('Item_name','')
    resp = {}
    # It will check the item_name value.
    if item_name == '':
        resp['status'] = 'Failed'
        resp['status_code'] = '400'
```

```
        resp['result'] = 'None'

else:
    resp['status'] = 'Success'
    resp['status_code'] = '200'
    resp['data'] = {}
    item_details = fetch_data_db(item_name)
    # If item_details not black than if part will run.
    if item_details:
        resp['data'] = item_details
    # If rest_data is black than else part will run.
    else:
        resp['data'] = "Data is not Available"
return HttpResponse(json.dumps(resp), content_type = 'application/json')
```

In our above code, we have created the four functions, which are as

- `fetch_data_db`: It is used for fetching the data from the database.
- `insert_data_db`: It is used for inserting the data inside the database.
- `post_req`: It is our post request, which is used for sending the data from the user end.
- `get_req`: It is our get request, which is used for fetching the data from the database.

Functions are internally connected to other function. For example, the user will call the get API for fetching the data, and in the background, it will call the `get_req`, and for getting the data from the database, the call goes to other function, which is `fetch_data_db`.

There is another API, which is post type, and when the user calls the post API, it will call the `post_req`, then it goes to interconnected function, which is `insert_data_db`.

In the post API, we are inserting the data from the user end. User can write the input which will go to insert in our code.

Loggers implementation in the project

Following is the logger configuration code, which is allocated in the `settings.py` file:

```
LOGGING = {
```

```
'version': 1,
'disable_existing_loggers': False,
'formatters': {
    'verbose': {
        'format' : "[%(asctime)s.%(msecs)03d] %(levelname)s [%(name)s:%(lineno)s] %(message)s",
        'datefmt' : "%d/%b/%Y %H:%M:%S"
    },
    'simple': {
        'format': '%(levelname)s %(message)s'
    },
},
'handlers': {
    'file': {
        'level': 'INFO',
        'class': 'logging.handlers.TimedRotatingFileHandler',
        'filename': 'log/firstapi_application.log',
        'when' : 'D',
        'interval' : 1,
        'backupCount': 40,
        'formatter': 'verbose'
    },
},
'loggers': {
    'django': {
        'handlers':['file'],
        'propagate': True,
        'level':'INFO',
    },
},
'app': {
```

```
    'handlers': ['file'],
    'level': 'INFO',
},
}

}
```

The above code snippet is written inside the `setting.py` file. This code description is already covered in the previous chapter.

Deploy Django with NGINX, PostgreSQL, and uWSGI

Currently, our project is created, and now we are going to deploy our project on a web server.

As per the topic, the project should use the PostgreSQL, and we already configure PostgreSQL in our project. To deploy a Django project with NGINX and uWSGI, we should follow the following steps:

1. After setting up our project, the first step is to install the uWSGI, so we are going to execute the following command:

```
sudo -H pip install uwsgi      # This command will be used for
python2
```

Or

```
sudo -H pip3 install uwsgi    # This command will be used for
python3
```

2. After installation, we will check that our uWSGI application is properly installed or not. So we are going to execute the following command:

```
uwsgi --http :8080 --home /home/sjain/testproj -w testproj.wsgi
```

By the above command, we told uWSGI to use the `wsgi.py` file, which is defined in our project `testproj` directory. To test our application, we execute our project on HTTP protocol with port `8080`.

If I want to run the project on the particular IP or any DNS, then we can also define that with the command by `127.0.0.1:8080` that way.

3. We run our project from the command line for testing, but in the actual deployment, it is not helpful because the production deployment process is different from the testing. In the actual deployment, we are going to run uWSGI in *Emperor mode*, which helps a master process to handle multiple applications separately automatically by giving the different configuration files.

For creating the directory, which contains our configuration files. To make the directory, we are going to execute the following command:

```
sudo mkdir -p /etc/uwsgi/sites
```

4. In the directory mentioned above, we have to place our configuration files which will take .ini files extension.

The following command will create the testproj.ini file and open it:

```
sudo nano /etc/uwsgi/sites/testproj.ini
```

The file always starts with [uwsgi] header. Rest all information will go inside the header. In the configuration file, we are using the variables, which makes our file reusable. There are variables, which are project and base. The project variable is used to store our project name, and the base parameter is used stored the path to your user's home directory.

Following is the code for above mentioned description:

```
/etc/uwsgi/sites/testproj.ini
```

```
[uwsgi]
project = testproj
base = /home/sjain
```

I am executing the project with the master process and taking the count of 5 workers. Following is the code for master configuration:

```
/etc/uwsgi/sites/testproj.ini
```

```
[uwsgi]
project = testproj
base = /home/sjain
```

```
chdir = %(base)%(project)
module = %(project).wsgi:application
master = true
processes = 5
```

We are going to use the UNIX socket because it is more secure than network port and provide better performance. It is an uWSGI protocol which is a fast protocol because it is using the binary format for communicating with other servers.

I set the vacuum option True, so it will clean the socket file automatically when the service is stopped and following is the code of changes mentioned

above:

```
/etc/uwsgi/sites/testproj.ini
[uwsgi]
project = testproj
uid = sjain
base = /home/%(uid)

chdir = %(base)/%(project)
module = %(project).wsgi:application
master = true
processes = 5

socket = %(base)/%(project)/%(project).sock
chmod-socket = 664
vacuum = true
```

Now our project uWSGI configuration is done.

5. We have to execute our project with the system files, so now we have to set uWSGI with `systemd` Unit File. Now for making the automated process with uWSGI, we have to create the `systemd` file to manage the uWSGI emperor process and also for automatic start. So that file should be placed in the following location `/etc/systemd/system` and the file name will be `uwsgi.service`:

```
sudo nano /etc/systemd/system/uwsgi.service
```

It will start with the `[Unit]` section similar to `testproj.ini` file, which is used for collecting the information. It will store the description of our service. In the next section `[Service]` is there, which will store all commands, which we are going to execute. One more important section that we need to add is `[Install]` section, which helps us to specify the service should start automatically:

```
/etc/systemd/system/uwsgi.service
```

```
[Unit]
```

```
Description=uWSGI Emperor service
```

```
[Service]
```

```
ExecStartPre=/bin/bash -c 'mkdir -p /run/uwsgi; chown sjain:www-data /run/uwsgi'
ExecStart=/usr/local/bin/uwsgi --emperor /etc/uwsgi/sites
Restart=always
KillSignal=SIGQUIT
Type=notify
NotifyAccess=all

[Install]
WantedBy=multi-user.target
```

Now it is completed but we are unable to start the service at this point because it will start with the `www-data` user, which is right now is not available. It will start with NGINX service.

6. Now our uWSGI configured is ready. So we are going to install and configure NGINX, which will provide us with a reverse proxy. For installation command will be like:

```
sudo apt-get install nginx
```

NGINX is installed, so now we are going ahead and create a server block configuration file:

```
sudo nano /etc/nginx/sites-available/testproj
```

```
/etc/nginx/sites-available/testproj
server {
    listen 80;
    server_name www.testproj.com;

    location = /favicon.ico { access_log off; log_not_found off; }
    location /static/ {
        root /home/sjain/testproj;
    }

    location / {
        include uwsgi_params;
```

```
uwsgi_pass      unix:/run/uwsgi/testproj.sock;
}

}
```

7. Next, link our new configuration files to NGINX's sites-enabled directory to enable them:

```
sudo ln -s /etc/nginx/sites-available/firstsite /etc/nginx/sites-enabled
```

8. Check the configuration syntax by typing:

```
sudo nginx -t
```

After the successful testing of command mentioned above, we can restart your NGINX service, which will load the new configuration:

```
sudo systemctl restart nginx
```

9. Now we will start our uWSGI server through mentioned below command:

```
sudo systemctl start uwsgi
```

10. Run the following command for delete the UFW rule for port 8080:

```
sudo ufw delete allow 8080
```

```
sudo ufw allow 'Nginx Full'
```

11. To enable the NGINX and uWSGI both services with automatic start. We will execute the following commands:

```
sudo systemctl enable nginx
```

```
sudo systemctl enable uwsgi
```

Deploy Django with NGINX, PostgreSQL, Gunicorn, and Supervisor

To deploy the application on Gunicorn and Supervisor, we need to setup our environment. So steps are mentioned below for deployment:

1. Django project should be created, and all required package should installed on the machine to execute a Django project. Which we already did in the previous topic.
2. We are deploying our application with the PostgreSQL database, so it required installation and database setup, which we also covered in the previous topic.
3. It required some dependency packages, which are as follows:

```
sudo apt-get install postgresql-contrib
```

```
sudo apt-get install libpq-dev python-dev
pip install psycopg2
pip install gunicorn
sudo apt install gunicorn
```

4. To check the Gunicorn installation. We are going to execute the following command:

```
gunicorn testproj.wsgi:application --bind 0.0.0.0:8001
```

This command will be executed from the folder `testproj`.

5. Create the file `gunicorn_start.bash`:

```
vim gunicorn_start.bash
```

We are going to write the code inside our bash file:

```
#!/bin/bash

NAME="testproj"                                     # Name of the
application

DJANGODIR=/home/sjain/testproj                      # Django project
directory

SOCKFILE=/home/sjain/testproj/gunicorn.sock          # we will
communicate using this unix socket

USER=sjain                                         # the user to
run as

GROUP=sjain                                         # the group to
run as

NUM_WORKERS=3                                       # how many
worker processes should Gunicorn spawn

DJANGO_SETTINGS_MODULE=testproj.settings           # which settings
file should Django use

DJANGO_WSGI_MODULE=testproj.wsgi                   # WSGI module name

echo "Starting $NAME as `whoami`"

# Activate the virtual environment
cd $DJANGODIR
source /home/sjain/testproj
export DJANGO_SETTINGS_MODULE=$DJANGO_SETTINGS_MODULE
export PYTHONPATH=$DJANGODIR:$PYTHONPATH
```

```
# Create the run directory if it doesn't exist

RUNDIR=$(dirname $SOCKFILE)
test -d $RUNDIR || mkdir -p $RUNDIR

# Start your Django Unicorn
# Programs meant to be run under supervisor should not daemonize
# themselves (do not use --daemon)

exec gunicorn ${DJANGO_WSGI_MODULE}:application \
--name $NAME \
--workers $NUM_WORKERS \
--user=$USER --group=$GROUP \
--bind=unix:$SOCKFILE \
--log-level=debug \
--log-file=-
```

6. Install the Supervisor and configure them:

```
sudo apt-get install supervisor
```

To configure the Supervisor, inside the `conf.d` folder we have to create a file with the same name as the project:

```
sudo vim /etc/supervisor/conf.d/testproj.conf
```

Inside the `testproj.conf` file, we will write the following code:

```
[program:testproj]
command = /home/sjain/testproj/gunicorn_start.bash           ;
Command to start app

user = sjain           ;
User to run as

stdout_logfile = /home/sjain/logs/gunicorn_supervisor.log    ;
Where to write log messages

redirect_stderr = true           ;
Save stderr in the same log

environment=LANG=en_US.UTF-8,LC_ALL=en_US.UTF-8           ;
Set UTF-8 as default encoding
```

7. To store the log file we are going to execute the following commands,
`mkdir -p /home/sjain/logs/
touch /home/sjain/logs/gunicorn_supervisor.log`
8. The following command will enable the Supervisor with the boot file and restart the Supervisor:

```
sudo systemctl restart supervisor  
sudo systemctl enable supervisor
```

To check the project status, we are going to run the following command:

```
sudo supervisorctl status testproj
```

9. Now Gunicorn and Supervisor are installed and configured with the Django project. So we are going to deploy this project with NGINX, to install it use the following command:

```
sudo apt-get install nginx
```

After installation, we are going to create the file and configure:

```
sudo vim /etc/nginx/sites-available/testproj.conf
```

Now write the following code inside the `testproj.conf` file:

```
upstream sample_project_server {  
    # fail_timeout=0 means we always retry an upstream even if it failed  
    # to return a good HTTP response (in case the Unicorn master nukes a  
    # single worker for timing out).  
    server unix:/home/ubuntu/django_env/run/gunicorn.sock fail_  
    timeout=0;  
}  
  
server {  
    listen 80;  
    server_name <your domain name>;  
    client_max_body_size 4G;  
    access_log /home/ubuntu/logs/nginx-access.log;  
    error_log /home/ubuntu/logs/nginx-error.log;  
  
    location /static/ {
```

```
        alias    /home/ubuntu/static/;

    }

location /media/ {
    alias    /home/ubuntu/media/;
}

location / {
    # an HTTP header important enough to have its own Wikipedia
    # entry:
    #   http://en.wikipedia.org/wiki/X-Forwarded-For
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_
for;

    # enable this if and only if you use HTTPS, this helps
    # Rack
    # set the proper protocol for doing redirects:
    # proxy_set_header X-Forwarded-Proto https;

    # pass the Host: header from the client right along so
    # redirects
    # can be set properly within the Rack application
    proxy_set_header Host $http_host;

    # we don't want nginx trying to do something clever with
    # redirects, we set the Host: header above already.
    proxy_redirect off;

    # set "proxy_buffering off" *only* for Rainbows! when doing
    # Comet/long-poll stuff. It's also safe to set if you're
    # using only serving fast clients with Unicorn + nginx.
    # Otherwise you _want_ nginx to buffer responses to slow
```

```
# clients, really.  
# proxy_buffering off;  
  
# Try to serve static files from nginx, no point in making  
an  
# *application* server like Unicorn/Rainbows! serve static  
files.  
  
if (!-f $request_filename) {  
    proxy_pass http://sample_project_server;  
    break;  
}  
}  
  
# Error pages  
error_page 500 502 503 504 /500.html;  
location = /500.html {  
    root /home/ubuntu/static/;  
}  
}
```

10. Run the following command to make a copy of the config file:

```
sudo ln -s /etc/nginx/sites-available/testproj.conf /etc/nginx/  
sites-enabled/testproj.conf
```

11. It will restart the NGINX service:

```
sudo service nginx restart
```

Conclusion

Nowadays Django API is deployed by many companies and people with a combination of Django + PostgreSQL + uWSGI + NGINX and Django + PostgreSQL + Gunicorn + Supervisor + NGINX. So that is why I covered this chapter in the book. We have seen that above-mentioned technology and with a different combination. In today's world people preferring the mobile application and for maintaining the communication between the mobile application and server it required APIs, to get and set data into servers we used to get and post a request in the above topics with working code. We covered the deployment combination with the code and example.

In the next chapter, the reader will gain knowledge of microservice.

Questions

1. What is uWSGI?
2. What is NGINX?
3. What is Gunicorn?
4. What is Supervisor?

CHAPTER 8

What are Microservices

Microservices are new trends of development. Nowadays, companies prefer microservice architecture for project development. It is a very compact solution for a project. It makes module manageability easy and project deployment faster. It also helps to develop the project faster. These mentioned reasons and many more make microservice in demand.

Structure

- The introduction of the microservices
- Monolithic vs. microservice approach
- Some important characteristics of microservice
- The microservices way and its benefits
- Microservices pitfalls

Objective

This chapter is important for this book because before the deployment of microservices with Django, it requires the basic information of microservice architecture. In this chapter, the reader will get the information of microservice and idea of its architecture. The comparison of monolithic and microservice architecture is also available in the book. It also covers the advantages and disadvantages of microservice architecture. These topics are important to know before deployment. After going through all topics, the reader can easily understand the microservices.

The introduction of the microservices

If we talk about microservices, then there is no particular definition available for microservice. Every person has a different perception for this term. I searched on the internet and found these two definitions, which relate this term:

- Microservices are small, autonomous services that work together.—Sam Newman, Thoughtworks
- Microservices are loosely coupled service-oriented architecture with bounded contexts.—Adrian Cockcroft, Battery Ventures

As per my understanding microservice is the component, which is deployed independently and communicates with another component internally. It is architecture, which is highly automated and evolves software systems.

If we have gone through with all terms then we will find out, we are already aware of these or probably implemented before. Let's see if you are aware of **service-oriented architecture (SOA)** then you also know the project modularity and communication through message. If you are aware of DevOps practices, then you also know about the automated deployment. Both are more near to the microservice approach.

There are three principals, which we should always remember while developing the microservices:

- **Microservices should deploy for big systems or projects:** For all scale of project should not require the microservices, in simple words for all small projects, it is not required to deploy the microservices. Ideally, microservices are for the big projects, which have so many modules and while handling those projects creating the problems. But it is difficult to identify the size of the project that it is small, normal or big. So we cannot measure that on the basis of user base and payload on the server. If our system has a high load of the user request and it requires scalability, then we can implement the microservice approach.
- **This approach is goal-oriented:** It is not important that when we face the problem, then we should follow the microservices approach. Nowadays many professionals are referring to this approach for developing the project because their goals are not just to give the solution of the problems, also taking such architecture in practice for more visibility and maintaining the easiness on the modules.
- **Replaceability of the modules:** In previously developed projects, which are not made with a microservice approach has less possibility to change any component from the project. Before changing the component developer should plan about the changes, find the dependence of particular code, list down the rollback steps if due to any reason code will fail and figure the

impact of code. After all, changes need to perform unit test operation, then integration testing with the whole product.

In the few scenarios projects are very complex, or their dependency is very critical, so in such condition on the coming issues or increases load size, we have to maintain the components instead of changing, that is the biggest loss of another approach.

In the microservices, we divide the whole project into a small component on the basis of modules. So that provides the facility of replicability on any component which makes it easy to a maintenance project.

Microservice applications have some important characteristics that are as follows:

- These are divided into small sizes.
- It required me to pass the messaging for communication.
- They are bound by contexts.
- They are independent for development.
- Also, deploy independently.
- Not bound with the centralized controller.
- Builds are deployed by automated processes.

These are the points which make microservice scope bigger. Some people think that microservices are the set of principles, so idealistic that they simply can't be realized in the real world. These kinds of claims are countered by growing companies, which successfully implement the microservice in their projects. These are Netflix, SoundCloud, and Spotify. They already share their experience publically about microservices.

If you are thinking about that microservice should be implemented with these technology sectors like banks, hospitals, or hotel chains, then none of these companies implement these. It is mostly used by those companies, which provide streamed content. In simple words microservice is not bound with any domain, it is an individual choice that how anyone adopts the microservice. In the microservice, there are two microservices characteristics, which they are focusing on are decentralization and autonomy.

In simple words, decentralization means that the whole project internal works, which includes execution of every module, task management and full journey of the project, will no longer be managed and controlled by a single system.

In simple words, autonomy means it is to believe in your development teams for the software they produce. The benefit of both approaches is that changes in the software become easier and faster. It also provides the facility to make decisions faster.

Monolithic vs. Microservices

If we talk about any approach like monolithic, SOA or microservices, almost every enterprise's application has this type of layered architecture:

- **Presentation:** It is a user interface for any websites, its web pages are UI and for application forms are UI.
- **Business logic:** It contains internal logic, which is used for execution.
- **Database:** Most of the projects used a database for collecting or retrieving the data, which can be SQL or NoSQL.
- **Application integration:** In the application, modules are integrated into the other modules. The communication between two we usually used the web service calls, which are as SOAP, REST or via messaging.

Let's talk about monolithic and microservices. Try to know about them deeply.

Monolithic

In the previously developed project, people referring to the monolithic approach for project development because they feel like all models in a single place would be easy to manage and control our project. In simple words, monolithic means collection of all in a single place. It describes a single-tiered software application, which means different modules of our project are combined into a single program. For example, the components are as follows:

- **User validation:** It validates that the user is authorized or not.
- **Presentation:** In that component, we handle the requests and responding cycle.
- **Business logic:** It contains the all internal execution logic, which we knew as business logic.
- **Database:** It is very important for every project if we are using the database in our project.
- **Integration of the services:** In the project, there might be many services that are available for different modules. So for integrating these services, we can use the messaging or REST API.

To understand the monolithic approach, we will take the example of an e-commerce application, which has such functionality like customer authorization, order placement, products inventory, payment validation and order shipment.

These are the basic functionalities for any e-commerce project. Nowadays many more functionalities are available for e-commerce projects like cart option, session tracker, product shipment tracking status, order confirmation through SMS or email, feedback on the product, customer reviews, customer rating on the product and many

more modules regarding payment option. In that example, I am considering only a few functionalities like product catalogue, order service, and payment service.

In the product catalogue, it will contain the product image, quantity, price, product serial number, and product description. In the order status, it considers the order number, shipment status and order confirmation. In the last payment service, we are considering the payment status and payment confirmation. For a better understanding diagram is shown below:

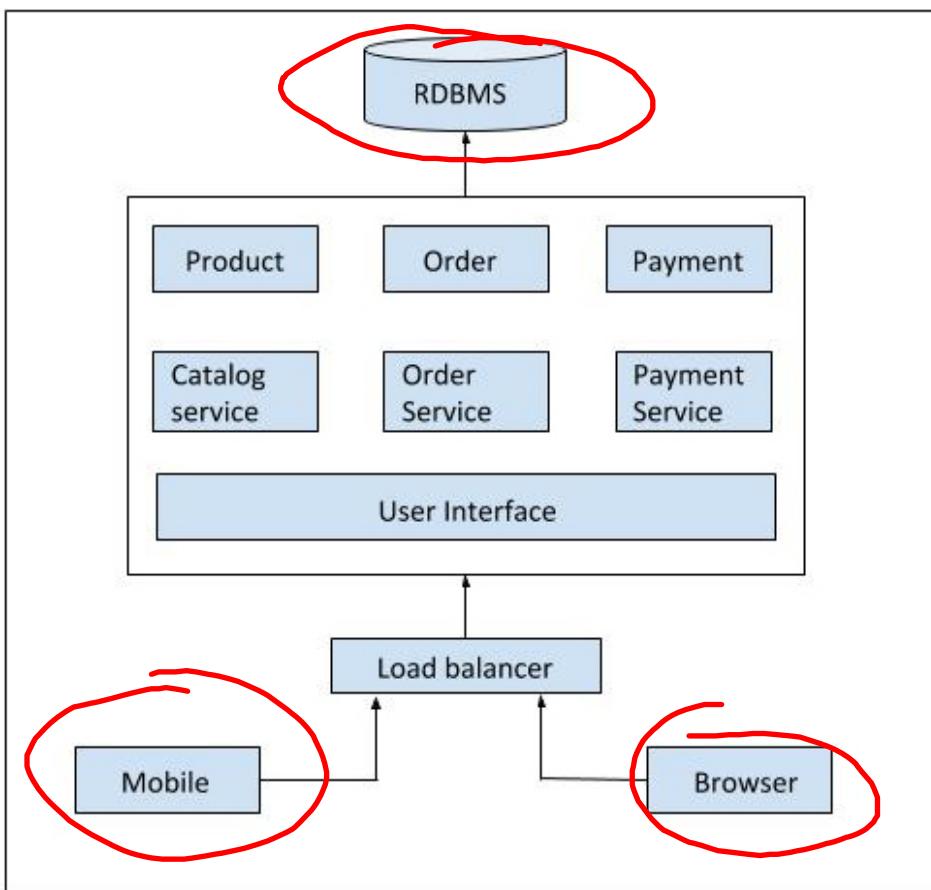


Figure 8.1.1 (a): Monolithic architecture through e-commerce project example

The above diagram has shown two ways for sending the request one is a browser based and second is mobile based. The request goes to load balancer then it flows to the server. The user interface is there, which is a website for browser based requests and forms for mobile. It is consuming the three services, which are catalogue service, order service, and payment service. In the background, all services are using a single database. That all services are running on a single server. All services are tightly coupled with each other.

We have seen the monolithic architecture, now let's see the advantage of that approach:

- In the start of the project, it is very simple to deploy the project.
- It also makes testing easy.
- It is also easy to handle a single server.
- If we want to scale our server horizontally, then we can clone our server and can run multiple copies simultaneously.

This approach also has some disadvantages, which are mentioned below:

- It is difficult to maintain the server for a large and complex project. It makes it difficult for changes and reduces the speed of development.
- If we have an application, which has huge tasks, then the application can slow down the start-up time.
- On every small change, we have to deploy our whole application on every update.
- It is also challenging to scale when other modules are conflicting resource requirements.
- If any bug occurs in any module or in the small process, it brings our whole server down.
- In that approach, if any small changes occurred like frameworks updates, it creates the difficulty to adopt these changes. If we implement, then it affects the entire process, which is costly, more effort and time-consuming process.

Microservices

In the microservice approach, the application is divided into small modules or services, instead of large systems. Every component of the project supports specific business logic, and for communication between the other components, they used simple services like REST or SOAP.

They also use individual databases for every single module. For microservice architecture, it is required that every component is having its database because it ensures loose coupling. There is also a facility where we can use a different type of database for every individual service. We are taking the same e-commerce example to understand microservices architecture, which has different components/modules. Each component or module is separate and loosely coupled. They communicate with each other, which depends on the scenario. For example, we are considering the following services:

- **Authorization module:** It is used for validating the user.
- **Order module:** It manages the user order.

- **Catalogue module:** It contains a list of products and updates the inventory.
- **Payment module:** It stores and works with the status report regarding payment.
- **Shipping module:** It stores and works with the shipments of the products.

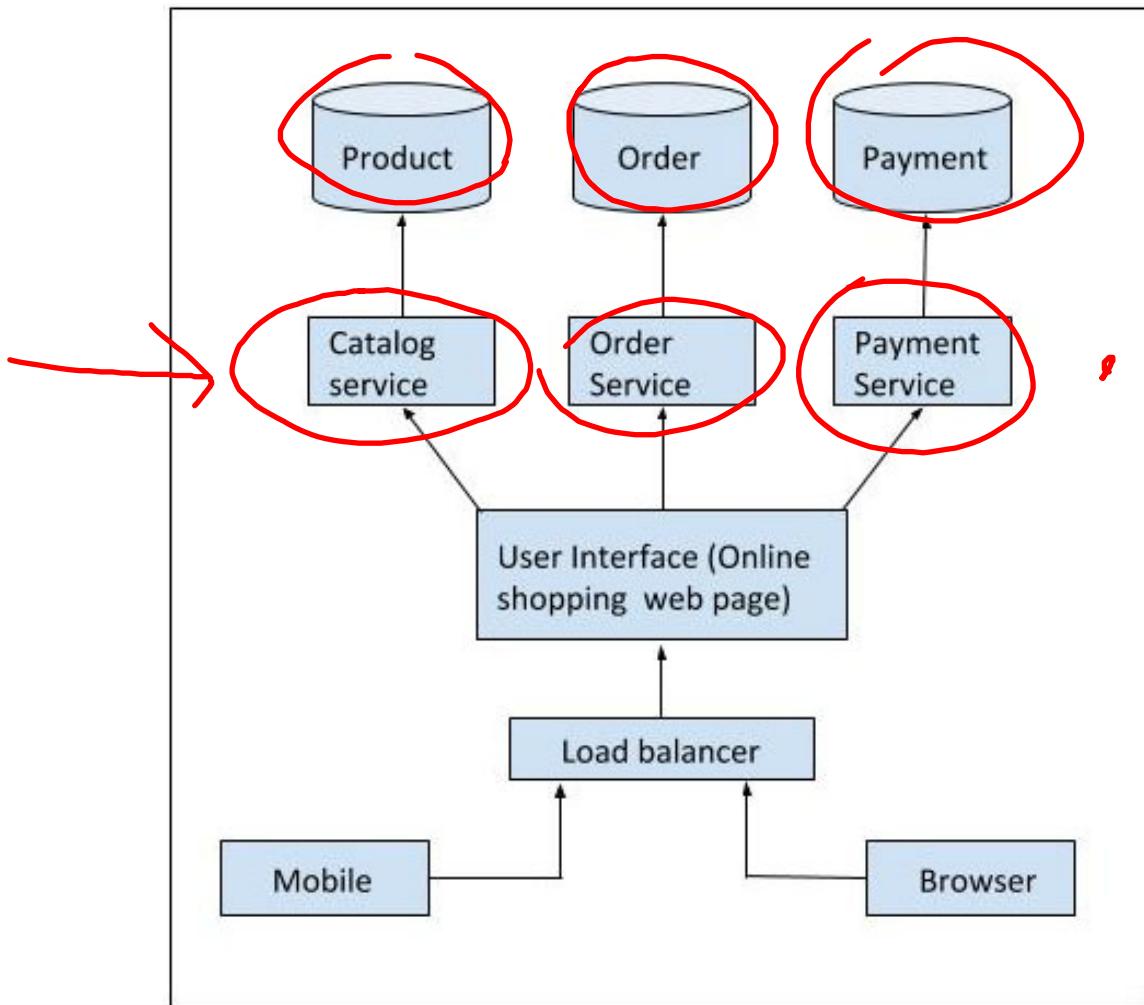


Figure 8.1.2 (a): Microservice architecture through e-commerce project example

The diagram has shown two ways for sending the request one is browser based and second is mobile based. The request goes to load balancer then it flows to the server. The user interface is there, which is a website for browser based requests and forms for mobile. It is consuming the three services, which are catalogue service, order service, and payment service. Every service is loosely coupled.

In the background, all services have their database. None of them is dependent on others. All services are running on their standalone servers. For communication between each other message passing or Rest API system is available.

This approach also has some advantages, which are as follows:

- It provides the facility of continuous deployment and delivery of large and complex applications.
- In the microservices, modules are divided into small services, so it provides better and faster ability to test the modules.
- We can also deploy services independently, instead of the whole project.
- It supports the module wise service development, so we can divide the modules team-wise, where each team will be responsible for module developments, deployment and to scale their service without depending on others.
- Every module is small, so that makes it easy for developers to understand.
- If we use the IDE, it makes them faster and more productive to the developer.
- The components are small, so it makes the application start faster and speeds up the deployments.
- The biggest benefit of the microservices is that if there is an issue in the particular module, then it only affects that service. Remaining service will work continuously to handle requests. If we compare it with a monolithic architecture, then one error can bring down the entire system.
- In the microservices approach, if there is any technology upgrade is required for a particular service, then it is easy to upgrade. We can also rewrite the whole service again with a new stack.
- In the microservice, we are not bound with any single programming language for writing the component code. For example, we can write one service in asp.net and other in Nodejs or Python as well into the single project.

This approach also has some disadvantages, which are as follows:

- That can create complexity for the developers to create a distributed system.
- If a developer is bound with any development tools or IDEs, then all tools are not supported for building the microservice architecture, they are bound to build monolithic applications.
- Its integrated testing is more difficult.

- For status update between the services, developers must implement the inter-service communication mechanism.
- Implementing use cases that span multiple services requires careful coordination between the teams.
- It can also create the deployment complexity in production server while deploying multiple services at the same time.
- It consumes more memory because microservice architecture has small modules, and every module has its own instance and databases. So based on operations, they took more memories. In the comparison of monolithic, there is only one instance for all operations.

So let's see the comparison between monolithic and microservice architecture through the diagram which is shown below:

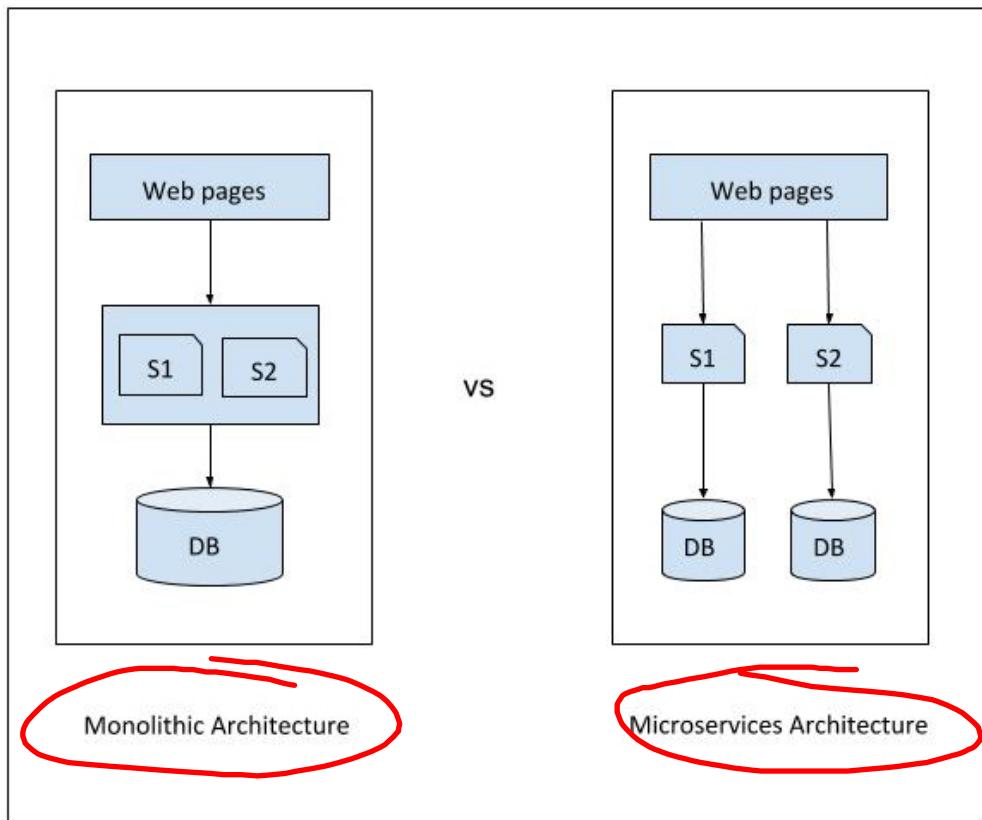


Figure 8.1.2 (b): Microservice vs. microservices architecture

The above diagram shows the two parts one is monolithic and second is microservice architecture. In the diagram **S1** and **S2** is service, **DB** is the database, and **web pages** means **UI**.

In the monolithic architecture, **S1** and **S2** are tightly coupled with the single server, whereas in the microservice architecture both are a standalone server. In the monolithic **S1** and **S2** using the single database, wherein the microservice **S1** and **S2** using their individual databases. In the monolithic web pages are sending requests to the server then it distributes and transfers to the service based on the request type.

In the microservice architecture, it distributes the request to the server independently. We have seen both architecture and its pros and cons. So now we can decide which is perfect for our project. As per my opinion, both approaches are perfect for deployment; it depends on our scenario or product requirements. The monolithic approach is the best for lightweight applications and heavyweight applications. I am going to recommend the microservices approach.

Some important characteristics of microservice

Microservices also have some boundaries and characteristics, which are going to be discussed in coming topics. In previously Spark team faced the problem, which provides SaaS business with basic services like authentication, account management, and billing.

The core problem was how to design their user account microservices to support everything which should work with the user accounts, API keys, authentication, business accounts, billing, and more.

As the solution, they created two microservices, one API for the **Users** and second for the **Accounts**. The use of Users API would handle user accounts, API keys and authentication. The second API that is Accounts would handle all types of billing-related logic. It is a good logical separation, but in the long run, they figure out a problem.

There were back to back calls going on the API and also lots of calls in the background. For example, there is one scenario billing-related operations are going on in the background, and the user is trying to update his details. In such conditions, it will create an impact. Both services were tightly coupled.

As per analysis in most cases, one service should not circularly call another service. So now the question is, how to avoid these microservice design pitfalls and what patterns should we look? Now we are going to discuss the challenges of microservices and their boundaries.

If we follow the microservice approach for developing the new system, then we get the core benefit of the architecture, which allows us to develop and modify individual components independently. In that, the boundaries are, problems can

come when we take steps to minimize the callbacks between each API.

We should know the boundaries of our system then approach the microservices. There are some rules, which will give clarity on the selection of microservices. There are some misconceptions in people's minds about the microservices, which I am going to discuss in the following. It is assumed that there are some sort of rules for a microservices architecture, which are as follows:

- In the microservice, there should be fewer lines of code: In reality, there are no limitations on the lines of code. It is not like that if you write the extra line of code, then it will become monolithic.
- In the microservice convert the code into functions: The function works based on its calling, and it returns the value as a result. So it comes to mind, that is it a good approach for a microservice? This depends on the function of what the function is and how it serves the entire system.

There are some characteristics of services which are designed well. If we have gone through with the microservices, then it will be clear in your mind that microservice is the well-designed service. In simple words, high cohesion and loose coupling makes it better.

Followings are some characteristics of microservices:

- **Characteristic 1:** If we are talking about a well-designed service that doesn't share database or tables with another service. In the above mentioned scenario of spark team, if we have multiple services and both are sharing the same table, then it is the cause of coupling between the services. It is really important how service is using the data. As per the main foundation principle while we developing the new service there should not be database boundaries between the services. Each service should deploy with an independent data source.
- **Characteristic 2:** In the well-designed architecture has a minimum amount of database tables. If we talk about the ideal size of the architecture, then it should be small but not much smaller, and it is also applicable to the number of database tables for every service.
- **Characteristic 3:** The services that we designed, which are stateful or stateless. We should clear in our mind before designing that it requires access to a database or is it going to be a stateless service which is used for processing the data like emails or logs.
- **Characteristic 4:** When we are developing the service, we should have the information about the data which we are going to use with service. There are two possibilities; one is when requested data is available as per request, and second is if requested data is not available, then what will be responsive data. If service is dependent on the data then we should keep in our mind

that if data is not found then how service will react.

In that scenario, one type of implementation is commonly used nowadays, make replication of the database. Replication can depend on our business logic. When issues arise, they are separated in different ways as per requirements. There are so many ways to do that we can create a partition for particular data storage, or we can create a different cluster for storing data.

- **Characteristic 5:** In the microservice, we should use a single source of information for transferring or accessing the data. Descriptively, if we took the example for a food delivery website when I ordered any food from the website, then it generates the unique order id for future processing. If we have one system, which generates that id and stored into the database, then the only ID should transfer from one server to another instead of whole information.

It should also be a considerable point that is our service too small or is it defined properly. Once we applied the characteristics mentioned above, we should take a step back and analyze that our created service is too small or is it defined properly. During the service testing and implementation, we should keep the few indicators in our mind and it is to check the dependency between the services. If two services are calling back to back continuously to each other, then it is a strong indication of coupling. It is better to combine them into a single service.

Let's talk about the Spark team problem, which they had two API services, accounts and users. Both are interrelated to each other. So as a solution, they came up with that idea, to merge the services and gave the new name Accuser API. This solved their problem completely, and it turned out into a good strategy. They eliminate the internal API calls between them, which turn the code simple.

It is important to manage the logs file. We can aggregate the log files somewhere not on the same server because log file consumes a lot of space. Logs also help us to monitor the activity on the server. The log file aggregation needs to be monitored, and if this process fails, then it should provide the notification. It can be possible by setting up an alert for it. Also, we should follow the standard for implementing the process, which will execute on failure.

It is important that a large team or organization should consider that point when different teams work on different services, then it comes into service boundaries. There are two points which keep in mind release scheduled independently and there uptime should be different. Amazon is an example of a large organization, which has multiple teams and large projects. As per the published article, Jeff Bezos made it mandatory to all employees that every team in the company has to communicate through APIs.

Let's summarize all the above mentioned characteristics of microservice:

- They don't share database tables with another service.
- It has minimum database tables.
- It is our choice to make service stateful or stateless.
- Data availability is needed for data relay services.
- We should use the single Id for transferring and fetching the data.

Above mentioned are the boundaries for new services.

The microservices way and its benefits

In the start, when you begin learning about microservice. It is easy to find the solution, which is similar to microservices. Docker is used for continuous delivery via Docker. We can build a system, which will work like microservices. For implementing the microservices, we should be focused on a goal to create that system, which makes change easier instead of focusing on a set of patterns, process, or tools.

The real value of the microservices can be realized when we focus on two special features that are *Speed* and *Safety*. In the end, any decision, which we make about the software development that impacts these two ideals. We have to maintain the balance between them at scale.

In the microservices speed of changes

If we are looking for immediate changes, then we keep the approach of adaptability. On one hand we can make software like, which can handle all problems itself. For such a system required advanced technology and would be a complex system or the solution, which is more realistic in the current state of technology. We can reduce the time of product delivery from development to the production environment.

In years ago, the software was released in a way that puts the efforts for creating and quality assessment then delivered to users. Nowadays, automation took the place of deployment and quality assurance, so it reduces the men's efforts and also reduces the cost of the software. There are so many companies, which have the resources and infrastructure to implement thousands of application releases within a single day. But they are not doing that because the quality of the product is more important than quantity.

In the microservices safety of changes

Microservice architecture gets lots of attention because of the term *Speed of change*, but it is equally important that changes would maintain the safety of the product. No one wants to breakdown the software on the production environment because of speed. Any small changes can break the system change. So we should implement

the changes in a fully tested and optimized way.

The way of microservices

If the probability of system breaking is zero, then speed is realistic. The development environment should be optimized, which can provide the speed, which helps to make multiple changes in a short time. On the production environments should be optimized for safety, restricting the rate of change to those releases that carry the minimum risk of damage.

Scalability in microservices

In today's development, the software architect should *think big* before building an application. The microservices style is rooted for solving the problems which arise when software gets too big. For building software at scale means that it can continue to work if the load grows more than initial expectations. Systems work easily under pressure.

These are the characteristics that are associated with microservice architecture that are replaceability, decentralization, context-bound, message-based communication, modularity, and many more. It is not simple and easy in the microservice architecture while taking the decisions of speed and safety-related to change. The microservices are fairly complex and will require you to understand a wide breadth of concepts.

Microservices pitfalls

For understanding the pitfalls of the microservices, I am discussing the few points, which are as follows:

- **Point-1:** One of the challenges, which developers and architects both face while creating the application with microservices architecture. How big microservice should be? How small microservices should be? These are the questions which decide the perfect implementation of microservice. If we made the wrong decision on the services, then it can impact the service performance, its robustness and development as well.

One of the primary reasons for this pitfall is because developers always confuse service with a class. Single service should be seen as a service component. A service component should be designed for performing a specific task in the system. It should be clear in the developer mind that service component has concise roles and statement is a well-defined set of operations. It is up to us how our service component should be implemented and how many classes are needed to implement the service.

There are three basic tests, which we can use to decide the right level of our services:

- o The scope of our service and its functionality.
 - o Accessibility of the database for the transactions.
 - o The last is the level of service sequence call.
- **Point-2:** Going with the new tech or going with the trends is good, but we must understand our business requirements before deployment. Microservices in trend, everyone wants to go with the approach. Before jumping on the microservices, we should also analyze your organizational structure, business requirements, project scope, and our technology environment. It is a very powerful architecture style. It is not easy to adapt to every application or environment. We can easily move from this pitfall by understanding our requirements.
If it fits with our requirements, then we can go with microservices. There are some questions which we should ask our self before finalizing the architecture. Following are the questions which can help us to make the perfect decision:
 - o What are my business and technical goals?
 - o What am I trying to accomplish with microservices?
 - o What are my current and foreseeable pain points?
 - o What are the primary driving architecture characteristics for this application (for example, performance, scalability, maintainability, and more)?
 - **Point-3:** In the microservices, architecture services are deployed separately, which means the services communicate with each other through a remote call. The pitfall may occur when we are not aware of the call time, which goes on the remote access.

Above mentioned are the major pitfalls, which we can remove by the good understanding of the microservices.

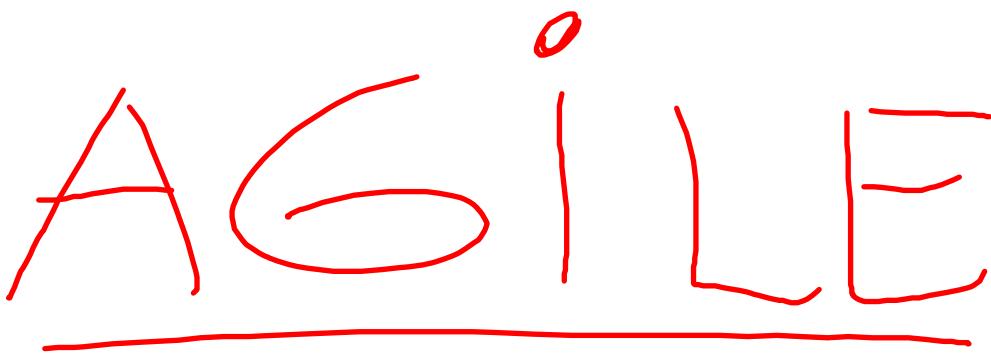
Conclusion

This chapter covered the introduction of microservice and the difference between the monolithic and microservices approach. Both are a useful approach to project development. Before selecting any approach, we should consider the following facts like how much load will come to the server, costing of the server, project scale (like it is a huge project or small project), programming language and database selection. These are important factors for every development approach.

The next chapter is covering the designing technique of the microservices architecture and approach, which we should follow ideally.

Questions

1. What is microservices architecture?
2. What is monolithic architecture?
3. Difference between monolithic and microservice approach?
4. Why is the microservices approach in demand?
5. What are the benefits of applying the microservices approach in a project?
6. What are the pitfalls of microservices?
7. Advantage and disadvantage of microservices?



CHAPTER 9

Designing Microservice Systems

Previously we have learned what are microservices and their pros and cons. We have also seen the benefits of microservices architecture and why an organization should be thought of building microservices architecture. The success of microservice architecture depends on the designing and the way of implementation. So it is very important to focus on every component of the microservice.

Structure

- Approach to microservices
- The designing process of microservices
- Important points for establishing a foundation
- Way to host
- Service designing parameters and boundaries
- Data and microservices
- Independent deployability

Objective

This chapter is included in the book because designing the microservice component is a very important factor. In this chapter, we will talk about the system models of microservices and a standard of the design process. Mentioned below are the important factors for our microservices system, which are as designing of organizational, their culture, architecture, and process which we follow for development? Continuous

observation and system updates approach are best for any system development.

It also covers key topics like important points for establishing microservice architecture, asynchronous messaging, and service development.

Approach to microservices

In the initial level, many developers are focused on the services, which they want to build. But when we develop the application with the microservice architecture we need to think about the services individually and their development on the basis of requirements. We also consider that fact how all services are going to work when we merge them together.

If we made the right decision for every service, then we can modify the system behavior and produce the behaviors, which we want. Handling all systems at the same time is difficult.

If we follow the model-based approach, then it can help us to conceptualize our system, and it also makes it easier to talk about the system.

The microservice design model is divided into five parts: **Service**, **Solution**, **Process**, **Organization**, and **Culture**, as shown in the following image:

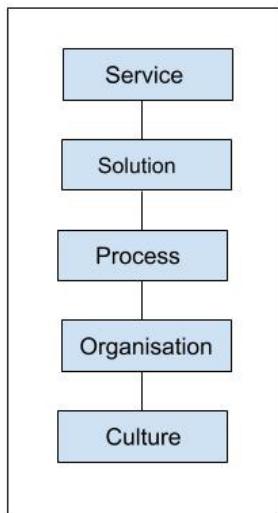


Figure 9.1: System design model for microservices

Let's talk about these parts one by one in detail:

Service

A well-designed microservices and APIs are important for the system. In a microservice system, the services form the atomic building blocks from which the entire organism is built. If we can decide the design, scope, and scaling of our service,

then we can reduce the complexity and make it simple. We will discuss the services with details in the coming topic.

Solution

The term solution meaning is to provide such solution architecture, which would be different for every service and also shows the futuristic view. When we design the microservice, then we should always be concerned about the coordination of inputs and outputs of multiple services. The futuristic system provides a more desirable system. For example, if we worked on the safety and routing features at first level implementation, then it can reduce the complexity of every single service.

Process

Microservice is not the only component, which makes microservice successful. We require the message passing mechanism for internal service communication. If individual services are worked successfully, then it is the result of the process and the tools, which we used to do their job on the systems. When we design the microservice system, we should follow the particular process for its development, code deployment, maintenance, and product management. Selecting the perfect tool and process is important because it produces the well-designed microservice system.

Organization

It is an important factor for microservices success. Because every organization has a different kind of team structure and working layers. It is based on how we work on the product and how we communicate.

Culture

It is an important factor too. It is intangible but also an important factor for microservices architecture. Culture is a set of values, which are shared by all workers in the organization. In simple words, if our organizational culture is good, they are welcoming the new ideas or new development strategies like microservices. With this approach, the organization will get mastering on their products and by the products they can influence the market.

When working on the all designing element together

When we put all of the design elements together for microservices, then we should remember that all elements are interconnected and on the change of a single element can be meaningful, and it can create an unpredictable impact on others. On the frequent changes, it took overtime, and these are unpredictable on other elements.

In simple words, the microservices system is complex and produces desirable outcomes from that system. It is not an easy task. But some of the organizations had their success story with microservices. So we can consider them as an example.

Follow the standardized process or set a standard for process

As we have seen before, using the right tools and following the right process, played an important role in the systems. If we set up the right benchmark in our organization for followed processes and tools, then we can easily predict the outcome from hard work. For example, if we make the standard deployment process in our organization and bound everyone to follow that process. As a result, it will decrease the deployment cost.

The designing process of microservices

Before designing the service, everyone should define the process in the first place because selecting the right designing process is the secret for great design. The right process towards the product helps to get the best product. In a more simple way to design the best microservices, we should follow the process, which helps us to make such a system on the small change whole system cannot get impacted. It should be in our practice that we can customize the process with the best fit for our services.

Setting up the optimization goals

It is perfectly fine when your microservice is giving you the same result which you wanted to achieve, but the set of goals are not perfectly applied on all organizations, so we should identify our goals and make this activity as the first task. It is an important task that helps us to optimize our goal. That doesn't mean if we optimize our goal, then we get the quality product. Many times it happens that we wanted many outcomes at the initial level. But as we go through the design process then we found that it is difficult to pull your system into many directions at the same time. If we follow the small set of optimization goals then it becomes easier to design. In such conditions, we get clarity on the goals, and it's a better chance to succeed.

Note: Sometimes, it happens that we changed our optimized goals on different requirements. It is fine to do that because our main motive is to follow the design process and implement the small changes, which helps to develop the new system. If changes are very different from the previous then we should create the new system instead of deploying the bigger changes on the same system.

Principles of development

The process of system optimization also follows some set of principles. They should be generalized and strictly followed by all systems, which can help to make better systems. It should be implemented in an organization like ideals or policies.

Sketch the system design

Basically, the meaning of sketch the system is that list down the core parts of your system and structure of the organization like:

- What is the perfect size of a team?
- How to give directions?
- Who will be in the team?

In the architecture:

- How I organized our services?
- What infrastructure must be in place?

In the service designing:

- What will the output of our services?
- How big our service should be?

In the processes and tools section:

- How are we going to live with the service?
- Which are useful tools as per our requirement?

We should evaluate these decisions against the goals and principles.

The purpose of doing the sketch stage is to get the process of designing. In this stage, we get developed the goals into the form of new ideas, get the impact of proposed designs, and it makes it easy to experiment in a safe way.

Implement the new, observe the system behavior and adjust the system:

Sometimes people work on the assumption that the system will work we expected but every time it does not work as we expect. It is the quality of good designers, who makes the system small initially, observes the system behavior as per changes then they continuously update the systems till they get the desired outcome. It seems easy, but it is difficult, the impact of a change on the small part in the system may change the result on the whole system, sometimes it is not visible. It is very useful if we get the information on our system and identifying the useful APIs. It could help us to predict the future. With the perfect information of our system, we can build our system boundaries, and it helps us to make perfect decisions about the size of our services and teams.

The designer of the microservices system

To make the effective microservice system designer should be able to change to a wide array of system concerns. We already discussed that culture, organization, process, effective architecture and services are the important points. Which designer should prefer in the first place. The boundaries of the system are the reflection of our company boundaries. That means if we made a small change, which can affect directly. This type of interrelated systems is common and visible all around us in the real world.

In simple words, the designer decides elements of the system, so we can say he or she is responsible for the bounded system. It is a designer task to make this possible to do minimal changes and achieve the desired output from the system and also asking for the systematic solution. In the company for every specialized task has different teams, like solution architect, always focused on the coordination between the services, developers of the service-focused for the efficient service design and team manager focuses on the people. There should be a team who can take responsibility for the whole microservices system to succeed.

Important points for establishing a foundation

We have seen the general model, which can help us to establish complex systems. It can be easier if we come up with proper principles, guidelines, and goals. The common challenge which companies face during the creation of microservice architecture is to find the set of principles to work. There is one way to find to copy other company working models and implement into your company, but it is not easy to copy blindly because every company has its own set of goals, culture, and process. If your company has the same, then you can use it directly, or you can copy all of them.

Following are the points for establishing the foundation:

Goals and principles

It is important to create goals and principles, which can guide our designing choices and guide into efforts that are required into implementation. In this coming topic, we will discuss some general goals and some example principles.

Goals for the microservices way

The better idea is to make higher goals, which can guide us to make the right decisions and flow of the project. As we know, our ultimate goal is to complete the project at the right speed and provide safety at any scale. The overarching goal helps us to aim

for the destination and provide the time, repetition, and consistency will help us to build the system, which hits the right notes for your organization.

In that approach, one problem arises is that it can take time to get the project at the perfect speed and safety when we create the project from scratch. But nowadays, technology world efforts can be reduced and boost both speed and safety. Instead of creating the software from scratch, we can use the already existing product in the projects. We should focus on four specific goals.

Following four goals should be considered:

- Reduce the system cost
- Increase the release speed
- Improve the system resilience
- Enable the visibility

Let's discuss this more in depth.

Reduce the system cost

Service cost varies based on the way to implement service, its designing and deployment. For example, if we create a new project and it took four months for designing, seven months for coding and testing, two weeks for deployment on production. Then it is approximately one year for the whole project. As we can see, assigning the dedicated resource (**Developer/Testers/Business analyst/Project architect**) for one year can be more costly. To stay out of such conditions, we should be very careful before starting the project. In the ideal situation as per microservice architecture if we create the service component then it takes only a few weeks. Nowadays, everyone wants to create components because it's easy to manage and handle the bigger problems.

The operation team reduces the cost by using virtual machines. Microservice architecture comes up with the solution where cost is reduced for coding and connecting services.

Increase release speed

Project deployment on the production server is our goal that increases the release speed. If we see the more specifically, then our motive is to reduce the time between getting the idea to project deployment on production. Instead of doing work fastest, we can go at a normal speed and take the shortcuts. Our main goal is to reduce the system cost, provide speed and lower the risk to the system. In the new project, we can speed up our deployment process by implementing the automation. If we make our deployment process automated, then our service gets faster on production. Many companies have a well-defined deployment process that increases the release on production, and because of the process, they can release multiple times in a day.

Improve the system resilience

Our solution speed or cost matters but more important is to build such systems which can handle the unexpected failures easily. In simple words, such systems which don't crash, even when errors occur. If we talk about the goal, then creating a single component, which is free of bugs or errors. It is more important than other goals. Practically it is impossible to create a component, which is bug-free.

There is one way if we do the automation testing whenever we deploy new changes. Then we improve the resilience of our service. If we include this into the build process, which increases the chances of finding errors in the code, it covers all code, but not the specific errors that could occur at runtime.

It is the part of the deployment process in many companies, before releasing the code on the production environment they test the code end-to-end, after the success of all the test cases it goes to live. Some organization deploys the project on production with a small size of the user then monitors the changes and its impact. If all goes well, then they increase the user base and release again for more users. In that way of deployment if any problem is encountered for some users, then it is easy to roll out with the previous release, and it impacts the small user base.

Enable the visibility

It means we have the runtime visibility on the code. In simple words, provide the facility where stakeholders keep watch and understand a system that is going on. Nowadays, we have many tools available, which enables the visibility on the coding process and also provides the reports of coding backlog, several builds were created and the number of reported bugs in the system. We can achieve runtime visibility by implementing logging and monitoring. We can track day to day activity through logs and monitor the operation level activity like storage, memory, throughput, and many more. There are so many monitoring tools available, which can take action or alert us when things go badly.

All the goals mentioned above are important. We might be able to reduce the system costs, but it can affect the runtime resilience. That can also happen that we can speed up our deployment but forget to maintain the track of which services are running on the production environment. So it is our task to maintain the balance.

Operating the principles

It is important to set goals before going with the microservice approach and it also requires the principles. Goals are generally used for achieving something but principles are the guidelines, which are more concrete. We cannot consider the principle as a rule setter; they help us to make the right decision in any situation. We can also call them the best practices. Every company has its own principle for work.

Shared capabilities

In the shared capabilities, we create a centered service in the organization. Which can be used by any services? For example, databases are common for many applications and policy enforcement. In large organizations, it is used commonly because it reduces the complexity and increases the cost-effectiveness. The term shared service meaning is not to share the same instance or shared data. It provides the cost savings facility, and its security goals are also different for others. Every component of safety is more important than centralized shared capabilities.

The following section is a quick rundown of what shared services platforms usually provide.

Hardware services

In most organizations dealing with the operating system deployment and for protocol-level changes, they have specific teams. Companies have to pay for the hardware and common software like windows or monitoring tools, and more.

There is one more approach available, where we can use the virtual machine instead of hardware. For example, EC2 service is available on AWS. We can use the VM machine for production deployment. Nowadays, one more approach is there, which is in trends, container service that solves this problem. Docker is the most popular in the containers.

Management of the code, it's testing and deployment

It becomes easy to manage the code if it is deployed on the server. When we talk about the deployment, then it also covers the testing and code management. Mostly code testing and code review stick with developers only. For example, if we have the server on Amazon web service cloud and we use their automated deployment service, then it will deploy the code on the production, and if it gets error, then it will face the problem.

So it better to treat code as shared service, where every team can see or review the code before deployment.

Data stores

Nowadays, many data storage platforms are available for storing the data like SQL-based, JSON document storage and graph-style database. It is not possible to use all kinds of technology by every organization. Some organizations face the problem of supporting their onsite databases which makes sense to use the same storage platforms in the organization and make it available to the team in the same infrastructure.

Coordination between the services

The technology which we use to make the coordination between the services is another example of shared service, which is shared across all teams. We have so many options available for coordination. Many of the flagship microservice companies (for example, Netflix and Amazon) wrote their system platforms for coordination.

Security and identity

Security of the platform is also a part of shared service. It covers the proxies and gateways. Some organizations created their security framework, whereas Netflix's Security Monkey is one of the examples. Many other products are available in the market.

As per the Merriam-Webster's *to do something in a new way, to have new ideas about how something can be done*. In simple words, enhance the functionality more valuable than creating something. Innovating something new is easier than creating something new.

Netflix calls this the principle of *context, not control*. Team leaders are taught to provide context for the team's work and guidance on meeting goals but do not control what the team does. Netflix's Steve Urban explains it like this:

I have neither the place, the time, nor the desire, to micromanage or make technical decisions for [my team].

—Steve Urban, Netflix engineer

Way to host

Nowadays **Vancouver-based Hootsuite** is popular, it saves our time and helps us to increase the business. This company was formed by members of Invoke Media, they built a platform to manage their social network interactions, and after some time, they realized that it is the need of other companies as well. In the start, they did the monolithic architecture, which is a PHP-based platform. At the increasing demands in the market, they increased the team size, which has 100+ developers, and they are evolving their application into a collection of product-oriented microservices.

They took a design-based approach to their microservice migration from the outset. As per them defining the right logic boundaries can be a harder problem than introducing new technology. They used the *distributed domain driven design*, which means to break services out of their monolith. The definition of the API and contracts associated with them provide a means of describing service scope and function. Also, the consumers of the API are involved in the creation of both. Hootsuite team divides their microservices into three categories: data services which encapsulate key business and ensure scalability, functional services that are the combination of

data services with business logic to execute core business logic, and facade services which decouple consumer contracts from core functional logic.

Hootsuite's design approach is evolving continuously to provide mature microservice architecture.

Hootsuite created a goal-oriented toolset for microservices. For deployment, they use Docker and Mesos. For services, they use Consul and NGINX. These are four open-source components, which are used together in their solution called *Skyline*. It enables the secure, dynamic, performant routing in their growing fabric of microservices. As they found Scala, Akka, and Play framework can be used by them for building individual services, and they used HTTP and Kafka both for interservice communication. Tooling also extends the design process. Developers should know what services and components are available to use in service development. The Hootsuite team created a tool, which generates visualizations of our system, that link to code repositories and operational documentation dynamically. As per needs, they created and discovered more tools.

The evolution of microservice architecture continues as they have dozens of microservices deployed in production, and many more are on the way. They adopt the microservices approach in that results it cuts across their architecture, organization, culture, processes, and tools, they have been able to improve their delivery speed, flexibility, autonomy, and developer morale.

Service designing parameters and boundaries

As per the previous topic, one thing that comes to mind is that the key element of the microservice architecture is to design the microservice components. It is a challenging task to design such a system that contains a large number of small service components, so we will talk about some tools that will help us.

There are a few questions, which arise during working on microservices:

How to properly size microservices? Or how to properly deal with data persistence to avoid sharing data across services?

Both questions are closely related. Sharing the data externally by two components can make the problems because it creates tight coupling and also stops the independent deployability.

There is one more problem which comes up frequently during implementing the asynchronous messaging and transaction-based modeling because it is supported by microservices. If we handle them properly, then it reduces the complexity of the overall systems and also helps to maintain the system speed and safety.

Let's talk about the boundaries of the microservices.

What is the optimal size of a microservice? So how micro should a microservice be?

There is no simple answer available for this question. The first things that come into our mind are fewer lines of code for microservice, or the team size, which is working on one are compelling. There is a problem with this kind of measures. They ignore the business context of what we are implementing, they forget to address the organizational context of who is implementing the service and how that service is being used in our system.

We should focus on the quality of each microservices, instead of finding some quantity to measure. Some microservice adoptive organizations used the **domain-driven design (DDD)** approach for getting a well-established set of processes and practices, which facilitate the business-context-friendly modularization and effective for large complex systems.

Domain-driven design

In the initial phase, people introduce the microservices in their companies in that manner, which they begin to divide their existing components into smaller parts to increase their ability to improve the quality of the service faster without sacrificing reliability. There are so many ways to divide a large system into smaller subsystems. In one way, you may decompose a system based on implemented technology.

Based on instance, all computation base heavy services should be written in C or Rust or Go (select as per your comfort) and that way we can separate subsystem, when we did I/O-heavy features that could benefit us and provide the non-blocking I/O of technology like Node.js, and we can make the subsystem of their own.

In other ways, we can divide a large system on the bases of team geography like one subsystem will be written in the US and others will be developed and maintained by other teams in Asia, Africa, Europe, Australia, or South America.

Another reason for dividing a system on the bases of geography is that every region has different legal, commercial and cultural. That requires them to operate in a particular way, and their local team may have a better understanding of their market. For example software development team of India: Can accurately capture the all necessary accounting details of accounting software, which will be used in New York?

Note: To understand the DDD approach, there is only a Model, which is important to remember that any software system, the model is used to connect with reality, which is not the actual reality. For example, when we log in to the online banking system and searching at our checking account, In that case, we were not looking at the actual account. It shows us an understanding based representation of our model, which gives us information about our checking accounts, such as available balance and our last transactions. That means if we go to other screens, it shows us different information, not the whole current account details because it is following our different model.

In the large systems, they don't have a single model. The model of a large system is a mixture of many small models, which are combined. Every small model is the representation of relevant business logic, which makes sense in their context and every context subject matters differently.

Context should be bounded

In a DDD approach, we need to be very careful when we combine every small context into a larger model of the software system. When we combine the distinct models, then the software becomes unreliable, buggy and difficult to understand. To solve such a problem *microservice* comes into the picture. To take the preceding quotation is an important observation for modeling; if we rely on a single model, then it becomes difficult to understand. In a microservices way, it breaks the large components into smaller ones, which reduces the confusion and makes it more clearly for each element of the system. The microservice architecture style is highly compatible with the DDD approach modeling.

If we use the DDD approach and bound contexts, then it is an excellent choice for designing components. We have the option that we can use DDD and create large components, but large components are not the microservice architecture.

Smaller is better

If we move from waterfall to Agile, then it will reduce the *batch size* of a development cycle. In the waterfall model, if the cycle was taking many months then we can reduce the similar tasks: define, architect, design, develop, and deploy, in much shorter cycles (weeks versus months). Lists down the other principles as well, but they only reinforce and complement the core principle of *shorter cycles* (that is, reduced batch size).

Note: There is no measurement available for "how small" a microservice should be. People use the word small which quality is like "reliable" and "coherent".

API design for microservices

When we are considering microservice component boundaries, then the code of the component is our concern. In the Microservice, components become valuable when they communicate with other components in the system. They use the API for communication. To deploy the microservice independently, a higher level of separation and modularity of our code than we should make sure that our APIs and the component interfaces both are also loosely coupled. That makes it easy to balance speed and safety.

There are two practices, which we can use for APIs communication in microservices:

- Message-oriented
- Hypermedia-driven

Message-oriented

It is like to write component code again in the structured format without changing the code functionality over time; the same efforts should be applied to the shared interfaces, which is between the components. One of the most effective ways is to adopt a message-oriented implementation on the microservice APIs.

Many companies that implement the microservice component design, they mentioned the notion of messaging, and it is a key design practice. For example, Netflix relies on message formats like Avro, Protobuf, and Thrift over TCP/IP for communicating internally and JSON over HTTP for communicating to external consumers. If we adopt the message-oriented approach, then it can expose the general entry points in our component (like port and IP address) and also receive the task-related messages at the same time. It provides the facility to change the message content.

Hypermedia-driven

Some companies use hypermedia-driven implementations. Which has the passed message between the components; it is more than a message (its data). Messages also have other descriptions, which are actions (like links and forms). So now not only is the data loosely coupled, actions as well.

Data and microservices

As a developer, our priority is data. When we start to design a new system then firstly we should design the data models. In the initial phase of the application, software engineers have the very first task to identify the entities and designing database tables for data storage. It is an efficient way of designing the systems, and all programmers follow the same. But it is not a good way to design a data-centric model. It is a kind of distributed system, which are not the independently deployable microservices. The reason for not deploying the centralized system is that a distributed system creates a dependency on the other component, which makes an inefficient system. Before starting the development, we should not go directly to data-centric habits; we need to rethink our system designs.

Let's see the example suppose we are designing the microservice architecture for a shipment company. So it works is to accept the packages and route them to the

nearest warehouse and deliver them to the destination. So nowadays the company is very tech-savvy related to Shipping. It provides the facility to track customer packages, and they build mobile applications for every different platform like Android, Windows, and IOS. All these data and functionality they need to get from the microservices. For every different operation and they have microservice, which collaborate and provide the data. If we go in a descriptive way shipment companies have accounting and sales functionality, then for this, they manage both subsystems through microservices. Because if they need to access the daily currency exchange rates to perform their operations. In the data-centric design, it will manage by creating a set of tables in a database, which contains the exchange rates. In such conditions, various queries from the different subsystems will create the load on our system. This solution has some issues if every microservices depend on the shared table, it leads to tight coupling and stopping them from the independent deployability. So as a solution to that problem, our solution architecture should be like for *currency exchange rates* has a different database, which communicates independently to the sales and accounting microservices. For establishing the communication between services, APIs are the way. It hides the implementation details. We can easily and securely exchange data and continuously provide the support to the currency rates service without noticing any consumers. By using the APIs, we can alter the interface of the currency rates microservice and easily provides data to the sales microservice, without affecting the accounting microservice.

Asynchronous message-passing and microservices

If we refer to this method for communication, then it makes the component loosely coupled. While microservice directly communicates through a message-queue channel, then they shared the data space. If we can do the encapsulate message-passing in the background of microservice, then it makes the message-passing capability loosely coupled. Refer to *Figure 9.2*, and it is showing us that two services communicate to the event hub, which provides the status of the task:

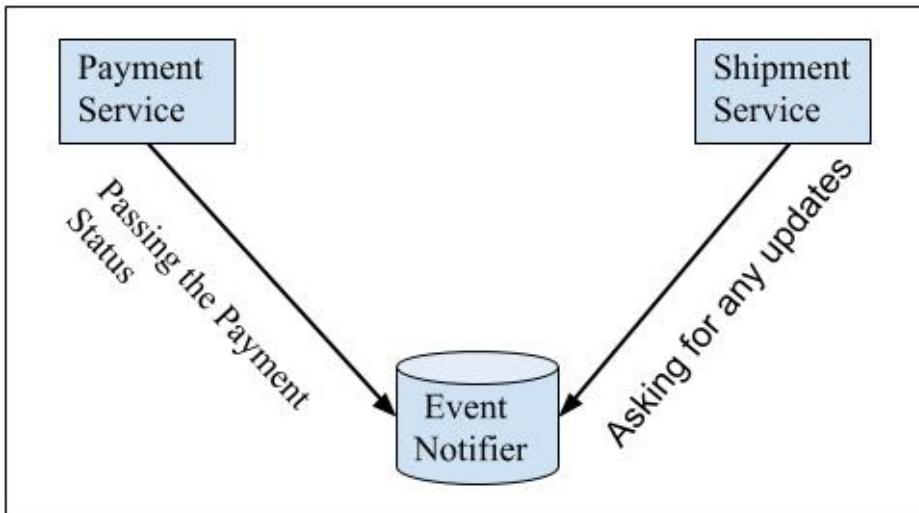


Figure 9.2: System designed for asynchronous message-passing implemented

Independent deployability

Microservice architecture core principles are independent deployability, which means each microservice must be deployed independently.

It provides the facility to perform selective or on-demand scaling. If any individual part of the system faced the high load then we can redeploy that microservice with more resources, it does not require to scale up hardware capacity. In many organizations, the operation of scaling can save large amounts of money, and it also provides some flexibility. In the operation of scaling hardware resources can be extremely costly. It is required to buy expensive hardware in anticipation of the usage.

The question is, could they deploy part of an application to a separate data center, a cloud-based one, in this case? The way most, typically monolithic, enterprise systems are architected; deploying selective parts of the application is either very hard or practically impossible. The operation of cost savings and flexibility is another significant benefit of the independent deployability in the organization. Two teams would be responsible for the development of separate microservices. The first team, which is responsible for the Customer Management, this microservice needs to make a change and can be released again, but Shipment Management microservice cannot be released independently, it requires to coordinate Customer Management's release with the team that owns Shipment Management. That coordination can be costly and complicated, since the latter team may have completely different priorities from the team responsible for Customer Management. To ensure independent deployability, we need to develop, package, and release every microservice using an autonomous, isolated unit of environment. But what does an *autonomous, isolated unit of environment*

mean in this context? What are some examples of such units of the environment?

Microservice architecture is a product of its time

We often get asked—what is the fundamental difference between microservice architecture and service-oriented architecture, especially given that so many underlying principles seem similar? We believe that the two architectural styles are creations of their respective eras, roughly a decade apart. In those ten years we, as an industry, have gotten significantly more skilled in effective ways for automating the infrastructure-related operations. In the architecture approaches, a microservice architecture approach is an achievement in continuous delivery and cost-effectiveness. Which attracts a large audience and few are bigger brands that adopted this approach early and made the handy in their organization (those are Amazon and Netflix).

For the production environment, many companies use the containers, which is very feasible for financially and operationally. Containers are extremely lightweight *virtual machine* which is very different from the VMs. It is based on the Linux kernel extension that provides the facility of running many different Linux environments on a single machine, which treat each other as independent service and that fulfilled the microservice need.

Security

In the microservice architecture, all components are independent of each other. So security is important to each component. The data which we shared between the services should be secure. For security, we refer to the APIs because it provides security and data sharing easy.

APIs provided by microservices, and it may be called by the front, backend and internal calls also. The calling of APIs depends on the microservice itself and the business needs.

Monitoring and alerting

As we know that microservices have so many different components, which have so many operations, so it is not an easy task to manage and keep track of all. That is why we required a special team to monitor our microservice component. That can watch the load and also alert us on the high utilizations on the components. We also have other options like using the tools, which can monitor our components and provide the alerts.

Conclusion

This chapter covered the approach for microservice architecture and its designing process. Those are important for microservice architecture success. We also discussed the best approach to system development which is continuous observation and updates the system accordingly. It also covers key topics like important points for establishing the microservice architecture, asynchronous messaging, and service development.

In the next chapter, we will see the service authentication for both monolithic and microservice architecture.

Questions

1. What is the approach for service design?
2. What is asynchronous message-passing?
3. What is the key factor for service design?

CHAPTER 10

Service Authentication

In the microservice architecture, multiple services are there which communicate with each other. To maintain security, we use the authentication between the services. This chapter will cover the authentication and authorization with Django. Any application which follows the monolithic or microservice architecture both needs to authorize users. In the Django framework already has his way of determining permissions and users.

Structure

- Authentication
- Authorization
- Authentication and authorization with Django
- Table configuration with Django admin
- Authenticate the services

Objective

This chapter is important for microservice architecture because security is a very important parameter for any application, which we cannot ignore. In this chapter, we will see the Django authentication and authorization. Also, get the information on how we can authenticate the service and what are the different ways are available. We will see the Django inbuilt mechanism, which maintains the authentications through the token.

Authentication

It is a common definition for authentication, the process of validation where we validate that request is authentic or not. The request contains the information of username or password or both or any field. It matches the request data with stored data.

Following are some common examples of authentication:

- Database login
- Server login
- Website login
- Operating system login
- Network login

The parameter of the authentication can vary from application to application. For example, nowadays, most people are using the smartphone, and for data security, they applied the lock on it. If we applied the mobile lock, then there are many ways to lock and unlock our phones like password-based, pattern matching, fingerprint, or face unlock. The type of lock depends upon the phone and our choice.

We can say that there are two levels of authentication are available.

- **Single level authentication:** We check the identity and after successful validation, send them into the system. For example, Facebook, LinkedIn login, and many more.
- **Multiple level authentication:** We check the identity and after successful validation passes it to the second level validation, if the user also passes the second level, then we send them to the system. In the multiple authentications, we can create multiple levels (more than one like two or three or many more) of validation. For example, Gmail two-level authentication, online banking login, and many more.

Authorization

In the general term, authorization is the functionality for specifying the access rights to resources. In simple words, it is the process to determine whether the authenticated user has access to particular resources. That verifies our rights on information, databases, files, and more. Authorization is the process which comes after the authentication process, which checks our privileges to perform.

Let's take the example of a window machine account login. In the organization, if we log in with the administrator user, then we get access every file and folder of our systems, but if we log in with the non-admin user, then there are some limitations on the access. This whole functionality is called the **authorization**.

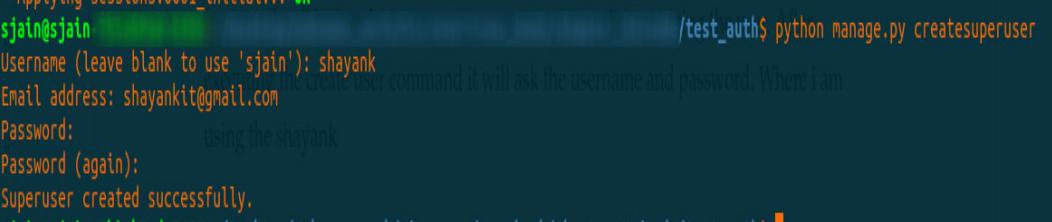
Authentication and authorization with Django

In the Django for handling the authentication, they made the system. That system can handle user accounts, groups, permissions, and cookie-based user sessions. Now we are going to use the user authentication in Django. So for that, I created a new project which is `test_auth`. Now, I am creating a new app inside the project, and its name is `user_manage`. So after both operations, our code structure will look like:

```
test_auth
    ├── db.sqlite3
    ├── manage.py
    └── test_auth
        ├── __init__.py
        ├── __init__.pyc
        ├── settings.py
        ├── settings.pyc
        ├── urls.py
        ├── urls.pyc
        ├── wsgi.py
        └── wsgi.pyc
    └── user_manage
        ├── admin.py
        ├── apps.py
        ├── __init__.py
            ├── migrations
            │   └── __init__.py
            ├── models.py
            ├── tests.py
            └── views.py
```

After creating the app in Django, we have to run the migration command that is `python manage.py migrate` then we will create the `superuser` for testing the authentication. So the command will be `python manage.py createsuperuser` for creating the user. After executing the create user command it will ask the username, email, password, and confirm password where I am using the `shayank` as my

username, for the email I am using the `shayankit@gmail.com` and for password `pass@123`. After inserting all the information, it successfully creates the user. Refer to the following screenshot for reference:



```
sjain@sjain: ~$ /test_auth$ python manage.py createsuperuser
Username (leave blank to use 'sjain'): shayank
Email address: shayankit@gmail.com
Password: using the shayank
Password (again):
Superuser created successfully.
```

Figure 10.3(a): Django superuser creation process

Now the user is created so we will start our server and check the login is working fine or not. So command to run the server is `python manage.py runserver`. After starting the server, I will hit the URL: `http://localhost:8000/admin`, and it will open the following page.

Refer the *Figure 10.3(b)* and *Figure 10.3(c)* for reference:

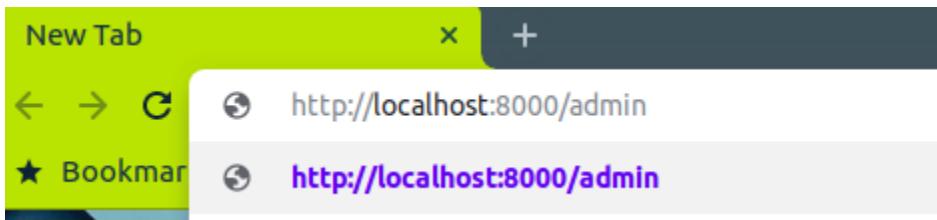


Figure 10.3(b): Link for opening the admin screen

In this screenshot, the link is mentioned, and we will hit this link to access the admin panel. Next is the screenshot of the admin panel:

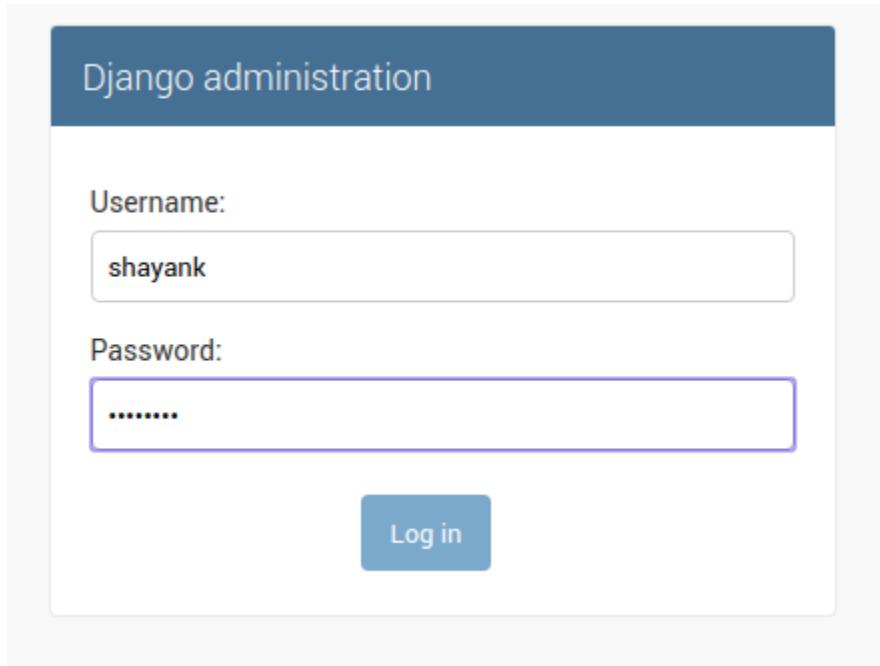


Figure 10.3(c): Admin screen first look

Preceding image is a screenshot of the admin screen. It appears when we hit the URL: <http://localhost:8000/admin>. After clicking on the **Login** button, it will show the welcome page for us, which is mentioned in *Figure 10.3(d)*:

A screenshot of the Django administration welcome page. The title bar says "Django administration". On the right, there is a "WELCOME, SHAYANK" message with links to "VIEW SITE / CHANGE PASSWORD / LOG OUT". Below the title bar is a "Site administration" link. Under "AUTHENTICATION AND AUTHORIZATION", there are sections for "Groups" and "Users", each with "Add" and "Change" buttons. To the right, there are two boxes: "Recent actions" (empty) and "My actions" (empty).

Figure 10.3(d): Welcome page after login

It is the welcome page, which comes after successful login credentials are entered. This is our authentication process, I wrote the valid username and password in the login page, and after validation, it shows the welcome to us.

When I click on the **Users**, it opens a new page, which has my registration information. It is shown in *Figure 10.3(e)*:

The screenshot shows the Django administration interface for the 'Users' list. At the top, there's a search bar with a placeholder 'Search' and a 'Go' button. Below it, a table lists a single user: 'shayank' with the email 'shayankit@gmail.com'. To the right of the table is a 'FILTER' sidebar with three sections: 'By staff status' (All, Yes, No), 'By superuser status' (All, Yes, No), and 'By active' (All, Yes, No). A 'Select user to change' link is at the bottom left.

Figure 10.3(e): User information page

Now, I will click on my name that is **shayank**, and then it will open the more description of the user, which is shown in *Figure 10.3(f)* and *Figure 10.3(g)*:

The screenshot shows the 'Change user' page for the user 'shayank'. It has fields for 'Username' (shayank) and 'Password' (algorithm: pbkdf2_sha256 iterations: 36000 salt: pJDi9***** hash: l4gj7L*****). A note below the password field states: 'Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form.'

Figure 10.3(f): Username and password details

In the above screenshot, we have two fields that are mentioned first is showing **Username** and second is **Password**. In the password field, you can see that I set the password **pass@123**, but it is showing into the encrypted format. It is auto encryption functionality which is provided by Django admin. It provides strong security for a password.

The screenshot shows a user profile edit form with two main sections: **Personal info** and **Permissions**.

Personal info section:

- First name: [Empty input field]
- Last name: [Empty input field]
- Email address: shayankit@gmail.com

Permissions section:

- Active**: Designates whether this user should be treated as active. Unselect this instead of deleting accounts.
- Staff status**: Designates whether the user can log into this admin site.
- Superuser status**: Designates that this user has all permissions without explicitly assigning them.

Figure 10.3(g): Personal info and permissions details

In the above screenshot, we have two sections, first is **Personal info** and second is **Permissions**. In the **Personal info**, it is showing the first name, last name, and email address which we can update. There is an email address column, and it is already filled because, at the time of registration, I gave `shayankit@gmail.com` in the registration process. In the **Permissions** section, three checkboxes are shown, which are **Active**, **Staff status** and **Superuser status**.

- **Active**: It means if we checked this box, then the user can log in and also active or if we unchecked then it will be deleted.
- **Staff status**: It means if we checked this box, then the user can log to this admin site.
- **Superuser status**: It means if checked this box, then the user will all the access.

Now I will create a new user from the admin site. The name will be `test_user01`, and password will be `test@123`.

The screenshot shows the Django Admin interface for the 'Users' model. At the top, there's a breadcrumb navigation: Home > Authentication and Authorization > Users. Below it, a search bar and a 'Search' button are present. A red box highlights the 'ADD USER +' button in the top right corner of the main content area. On the left, there's a table with columns: USERNAME, EMAIL ADDRESS, FIRST NAME, LAST NAME, and STAFF STATUS. One row is visible for 'shayank' with the email 'shayankit@gmail.com' and a green checkmark in the STAFF STATUS column. To the right, there are two filter sections: 'By staff status' (with 'All', 'Yes', and 'No' options) and 'By superuser status' (with 'All' and 'Yes' options). At the bottom left, it says '1 user'.

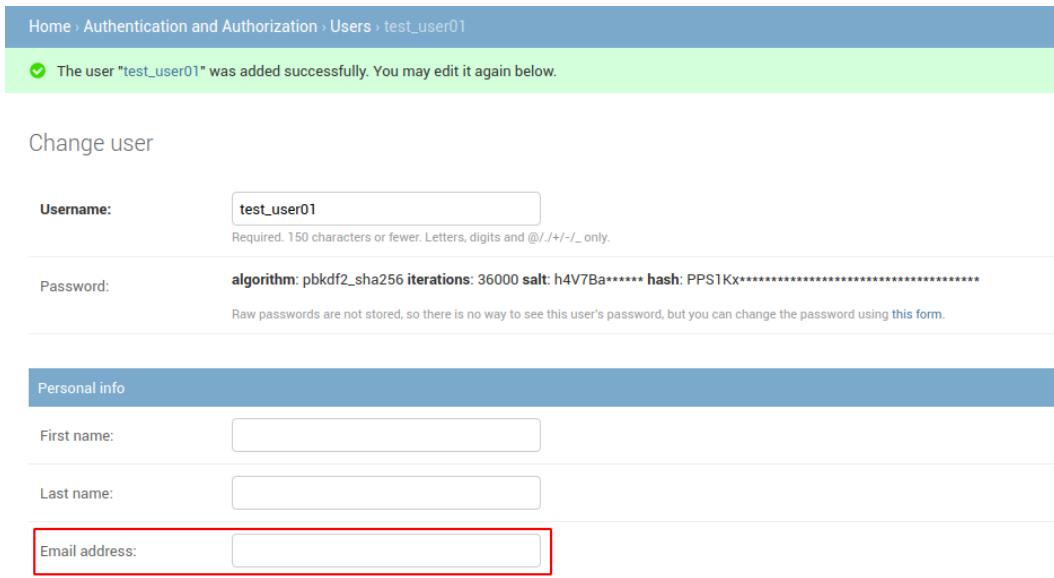
Figure 10.3(h): Create new user option

In the above screenshot, we have marked section is mentioned, through add users button, we can create a new user. So by using this option, I will create a new user. When we click on the **Add users** button, it opens a new page, which is mentioned below:

The screenshot shows the 'Add user' form in the Django Admin. The top header says 'Django administration' and 'WELCOME SHAYANK VIEW SITE / CHANGE PASSWORD / LOG OUT'. Below it, the URL is 'Home > Authentication and Authorization > Users > Add user'. The form has three main input fields: 'Username' (containing 'test_user01'), 'Password' (containing '*****'), and 'Password confirmation' (containing '*****'). Below each password field are four validation error messages. At the bottom, there are three buttons: 'Save and add another', 'Save and continue editing', and a larger blue 'SAVE' button.

Figure 10.3(i): User creation page

In the above screenshot, we have three fields mentioned, which we are used for creating the new user. After inserting the values and clicking on the save and continue editing button, it opens a new window, which is mentioned in the following screenshot:



The screenshot shows a user detail page for 'test_user01'. At the top, a green success message states: 'The user "test_user01" was added successfully. You may edit it again below.' Below this, the 'Change user' section displays the following fields:

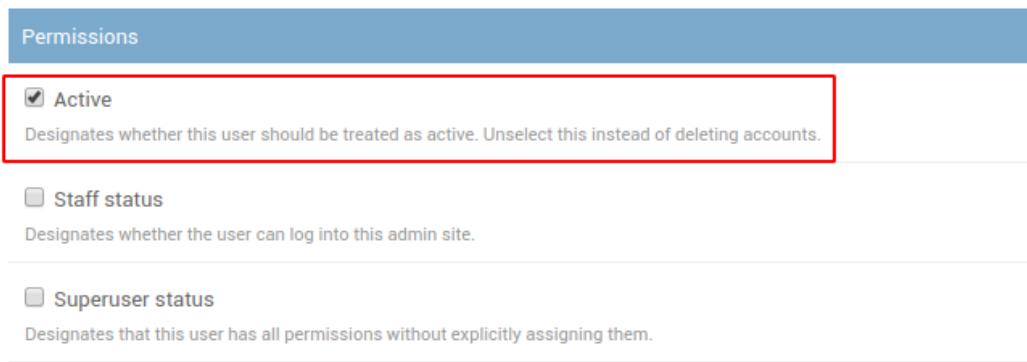
- Username:** test_user01 (highlighted with a red box)
- Password:** algorithm: pbkdf2_sha256 iterations: 36000 salt: h4V7Ba***** hash: PPSTKx***** (Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form.)

Under the 'Personal info' tab, there are three input fields:

- First name: [empty]
- Last name: [empty]
- Email address: [empty] (highlighted with a red box)

Figure 10.3(j): User detail page

In the above screenshot, it is showing us that user is created and its email block is blank. So now we can edit that block.



The screenshot shows the 'Permissions' section of the user detail page. It contains the following options:

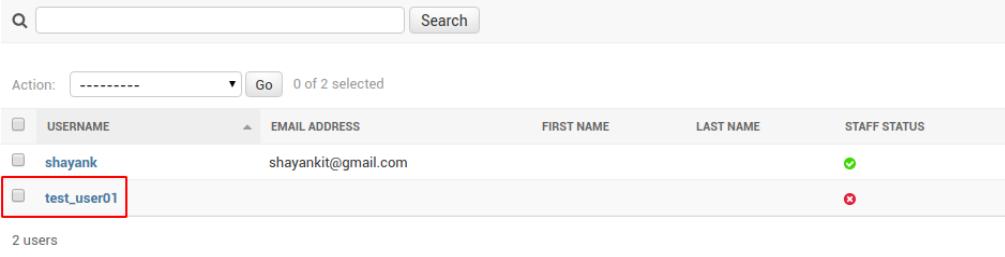
- Active** (checkbox checked, highlighted with a red box): Designates whether this user should be treated as active. Unselect this instead of deleting accounts.
- Staff status** (checkbox unselected): Designates whether the user can log into this admin site.
- Superuser status** (checkbox unselected): Designates that this user has all permissions without explicitly assigning them.

Figure 10.3(k): User permission details

In the above screenshot, it is showing the default permission of the user, which assigned automatically to the user.

Now, if we see the list of the user then it will show us the two users, which is mentioned in *Figure 10.3(l)*:

Select user to change



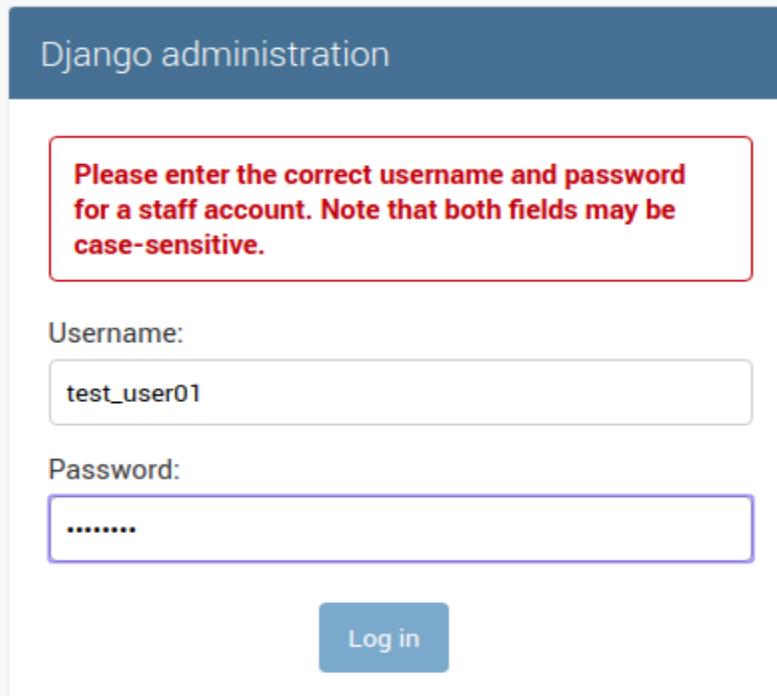
<input type="checkbox"/>	USERNAME	EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
<input type="checkbox"/>	shayank	shayankit@gmail.com			
<input type="checkbox" value="test_user01"/>	test_user01				

2 users

Figure 10.3(l): List of available users

In the above screenshot, it is showing the list of the available user. We recently created the `test_user01` user.

Now, I am trying to login with a new user that is `test_user01`, and it allows us to log in on the admin site. It shows the error message for us, which is mentioned below:



Django administration

Please enter the correct username and password for a staff account. Note that both fields may be case-sensitive.

Username:

Password:

Log in

Figure 10.3(m): test_user01 login fail

In the above screenshot, I am trying to login with `test_user01` user, and it is showing the error message. This message is coming because we only give active permission for `testuser01`, which means the user is active, but it cannot allow to

log in on the admin site.

The permission related task called the authorization, and above scenario user has limited permissions.

To resolve this issue, we will log in on admin site through superuser and update the `test_user01` permission which is shown below in the following screenshot:

Permissions

Active
Designates whether this user should be treated as active. Unselect this instead of deleting accounts.

Staff status
Designates whether the user can log into this admin site.

Superuser status
Designates that this user has all permissions without explicitly assigning them.

Figure 10.3(n): Adding new permission to the user

In the above screenshot, I checked the staff status option. So now user can access the admin site. Let's try to login again with the `test_user01` user.

Now I have tried to log in with a new user that is `test_user01` and password `test@123`. So it sends me in the admin site, but it also shows the new message. On the welcome page, that message is shown is mentioned below *Figure 10.3(o)*:

Django administration

WELCOME, TEST_USER01 | VIEW SITE / CHANGE PASSWORD / LOG OUT

Site administration

You don't have permission to edit anything.

Recent actions

My actions

None available

Figure 10.3(o): Welcome page of test_user01 user

In the above screenshot, it shows us that `test_user01` is logged in successfully. In the other marked block, it shows that `test_user01` cannot edit anything because of limited permissions. There is one more screenshot, which is showing the one more functionality of the admin site. I logged in with the superuser and in the welcome page, it shows me my recent activity, which is shown in *Figure 10.3(p)*:

Site administration

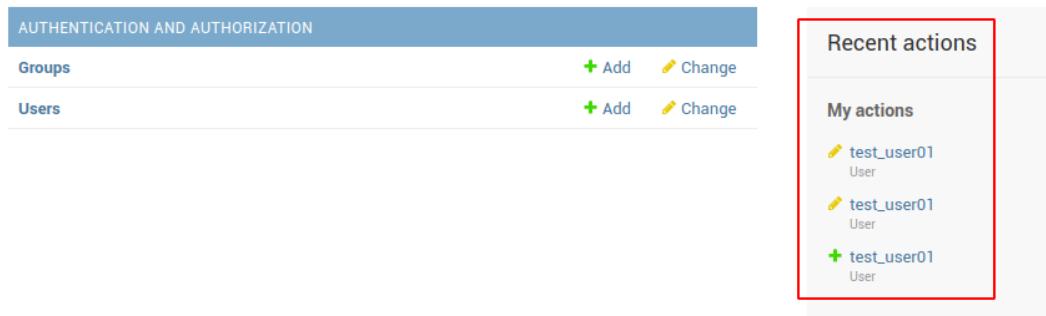


Figure 10.3(p): Recent activity tab on the welcome page

In the above screenshot, we can see the marked section. There are lines shown, with the plus marked it means we added the new user. In the second and third mark, it shows that we edited the user two times. One we open the user details but not save the email and second when we provide the staff status permission.

Table configuration with Django admin

To configure the table, we need to create a new app, which we already created with the name of `user_manage`. So now the `model` file is already available, and we are going to create the new model inside the table and put the entry of our app inside the `settings.py` file:

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.db import models

# Create your models here.

class Test_table(models.Model):
    fname = models.CharField(max_length=20)
    lname = models.CharField(max_length=20)
```

Figure 10.4(a): Models.py file code

This is `models.py` code, which we created the `Test_table` model. In that model, we also created the two columns, which are `fname` and `lname`.

In the mentioned below screenshot, we are adding our app name `user_manage` which is marked in red:

```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'user_manage',
]
```

Figure 10.4(b): Setting.py file code

After adding the entry of our app, there is one more file, where we have to enter our table details:

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.contrib import admin

# Register your models here.
from . models import Test_table

admin.site.register(Test_table)
```

Figure 10.4(c): Admin.py file code

This is `admin.py` code, which is located inside the `user_manage` folder. Red marked section is showing that we included the `Test_table` model here. The `admin.site.register(Test_table)` meaning is that we want to show `Test_table` in the front end.

Now, we did three changes, so for reflecting these changes, we will execute the

following commands line by line:

```
python manage.py makemigrations user_manage
```

```
python manage.py migrate
```

```
python manage.py runserver
```

So to check the changes, hit the URL: <http://localhost:8000/admin> and login with superuser shayank. It shows us the following output:

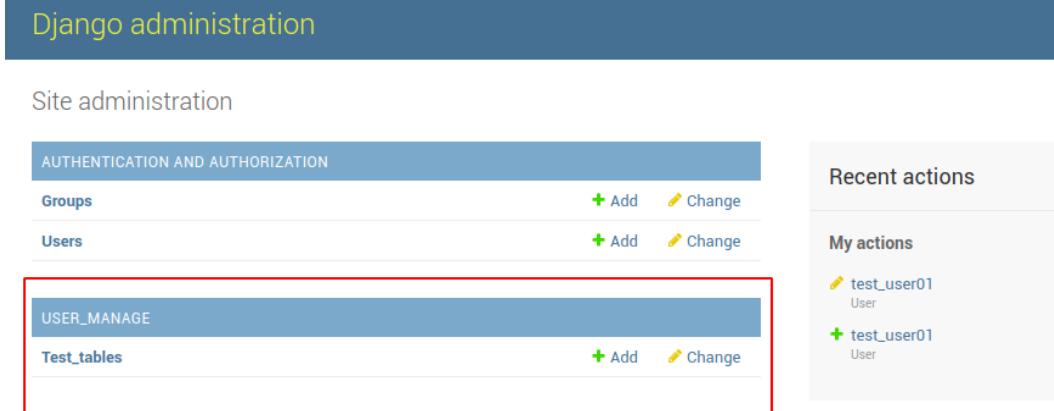


Figure 10.4(d): Admin site view with the table

This is our admin site screenshot, which is showing our app name that is `user_manage` and table name `Test_tables`. So above mentioned are steps for including the table in the admin site.

Authenticate the services

As we know that the process of validating the request is called the authentication. There are so many ways to authenticate the services. We can use any of the following authentication approaches:

- HTTP or HTTPS based
- Session and cookies based
- Token-based

To understand all of the above, we need to know the following terms before:

- **Client-side:** It means is our local machine browser or the place where we send the request to the internet.
- **Server-side:** It the place where our application is hosted or that place where we get the response.

For example, if I wrote the www.facebook.com on my local browser and login page is opened on my machine then in such scenario my local machine browser is client-side, and the page which is sent from the internet is called the server.

Now, the following are the authentication approach:

HTTP or HTTPs based

In the HTTP request, we send the request to the server, and it contains the information like username and password. After getting the username and password server to validate the identity and send back the response to the client. That way it works, it seems easy right, but it is not a secure way to communicate with the server because data is travelling into the network in plain text format. There is the provision that we can transfer the data into base64 encryption format, but it is also less secure.

So for better security, we can use the HTTPs based request. In the HTTPs request SSL certificate is required for communication on the client and server-side. For better understanding, we are going to take the example, suppose I wrote the www.gmail.com on my browser then request goes to the Gmail server and as a first response, it sent the SSL certificate on the client-side to store in the browser. After saving the certificate, browser again sends a request to the server through SSL certificate. In the certificate, Gmail public key is mentioned, so with the help of public-key, it encrypted the data and generated the symmetric key then sends the data from client to server. On the server-side, it decrypted the data with Gmail private key and process the request. That is the way HTTPs protocol works, and that is why it is more secure than HTTP.

Session and cookies based

For understanding the session and cookies based, it required the basic knowledge of session and cookies:

- **Session:** In the IT computer world, it is a reference to a certain time frame for communication between the two devices or client and server-side. It is handled by the server.
- **Cookies:** In the IT computer world, it is a text file, which is stored in the client-side web browser. It is the way to maintain application state.

Now, let's discuss that how it works when we send a request to the server then it creates the session on the server-side and as a response, it sent the user activity details to the client browser. Understand this with the example if we opened the www.amazon.com on our browser then it creates the new session-id and sends that session-id to the client-side. Browser stored that session-id in the cookies. Against that session-id, if I search the mobile or wallet on the amazon website, then it sends that searching detail to the cookies, and it updates our cookies. So in the authentication when we send the username and password to the server, then it creates the session-

id, and you have seen that at the time of login browser showed us the pop-up box that you want to save this username and password for future reference with the option of **Never** and **Save**. So if we save the password then it stored our credential in the cookies for future reference. When we type the www.amazon.com again, then the server sends a request to the client browser for searching the stored cookies, if they found then it open the page directly otherwise opened the login page again.

Token based

In the token-based authentication, the server validates a user based on the token. Let's have a look at how it worked when I opened the www.linkedin.com on my machine, then it will ask for login credentials. After inserting the username and password, it generates the token and sends this token to the client-side. In the next request, the client includes this token, and in the server-side, the server validates the token if the token is valid then the client can communicate continuously otherwise it breaks the communication and sends back the user to the login page. We have two options for token-based authentication one is JWT and second use the static token for authentication. Token-based authentication is used for API gateways.

Conclusion

In this chapter, we have seen what authentication and authorization is. We also discussed how Django provides the default functionality of user authentication and authorization. It also covers the Django admin module features. We have seen the different type of service authentication approach, which we can use to secure our service.

In the next chapter, we will see how we can implement and deploy microservices with Django, which contains all the information about microservice architecture and steps of implementations.

Questions

1. What is authentication?
2. What is authorization?
3. What is the difference between authentication and authorization?
4. Types of authentication?
5. What is a session?
6. What are cookies?
7. What is the difference between the HTTP and HTTPS protocol?
8. What is token-based authentication?
9. How to create a user in the Django admin?

CHAPTER 11

Microservices Deployment with Django

We have gone through with the Python, Django, and microservice in the previous chapter. Now we are going to use all of them together. In this chapter we will create the microservices and deploy them. In today's world, many organizations are jumping to the microservice architecture. Every company had a set of rules and business logic. In our demo project, I will try to follow the global standard for development and deployment.

Structure

- Flow figure for microservice architecture
- Data flow and description of used API
- Microservices deployment with Django
- Our microservice architecture flow figure

Objective

This chapter is included in the book because it helps the reader to understand the microservice workability. In the chapter, I am trying to create mobile e-commerce application. Like amazon or Flipkart but in the limited functionality like register

the user, login with the username and password, list of available products, buy the product and shipment of the product. We will divide these functionalities into services. I am deploying the services on different servers, and every service has a different database. Every service will communicate with each other through APIs. To see the microservice, I am using the APIs only. Now let's make the project.

Flow figure for microservice architecture

In our project, we will create the four services, which will be divided into four parts as follows:

- User service
 - userregistration API
 - userlogin API
 - userinfo API
- Product service
 - getproduct API
- Payment service
 - initiate_payment API
 - payment_status API
- Shipment service
 - shipment_updates API
 - shipment_status API

So, in our project user service, product service, payment service, and shipment service are small components of our microservice architecture. Those are playing a major role in our project. Let's see the overview of our project through *Figure 11.1(a)*:

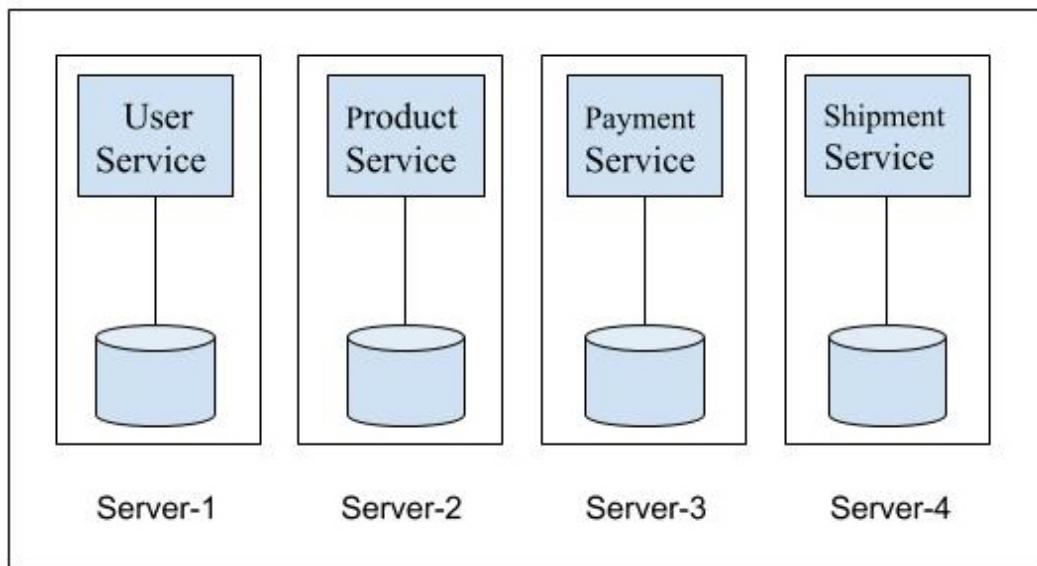


Figure 11.1 (a): Service Architecture for Ecommerce

In the above diagram, there are four individual servers created. Every service has its own application and database. This is a basic figure of individual service.

Data flow and description of used API

To understand the data flow of our architecture, we need to know our API functionality first.

For better understanding, we will go through with the following headline:

- Service name and its purpose
- Service-wise there database table and API description

Service name and its purpose

As per our application, we have four services and those descriptions are shown below:

- **User service:** It is used for handling user registration and login functionality.
- **Product service:** It is used for showing the available products and operations on the products.
- **Payment service:** It is used for handling payments related to all operations.
- **Shipment service:** It is used for maintaining the product shipment process as per the users.

Service wise database tables and API descriptions are mentioned in the following section.

User service

I am using the three APIs to handle the user service, which is as follows:

- **userregistration API:** In the registration API, we are taking the information of the user and storing it into a database. We are using PostgreSQL for storing the data. The table name is `user_registration`.

Following are the details, which we are asking from user to register:

- First name
- Last name
- Email ID
- Mobile number
- Password
- Confirm password
- Address

- **userlogin API:** In the user login API, we are validating the user from the database. We are using the same table, which is used for user registration. In the username field, we are considering the email id as the username, and the password field name is the same.

Following are the request parameters for user login:

- User name
 - Password
- **userinfo API:** It is used for fetching the user basic info. It is required while we process the shipment. In the user info API, we will provide the user name as request parameters and API give first name, last name, email ID, mobile number, and address of the user.

Folder architecture and code

Following is our project architecture:

```
user_service
├── manage.py
└── user_info
    ├── admin.py
    └── admin.pyc
```

```
|   ├── apps.py
|   ├── __init__.py
|   ├── __init__.pyc
|   ├── migrations
|   |   ├── __init__.py
|   |   └── __init__.pyc
|   ├── models.py
|   ├── models.pyc
|   ├── tests.py
|   ├── views.py
|   └── views.pyc
|       user_login
|           ├── admin.py
|           ├── admin.pyc
|           ├── apps.py
|           ├── __init__.py
|           ├── __init__.pyc
|           ├── migrations
|           |   ├── __init__.py
|           |   └── __init__.pyc
|           ├── models.py
|           ├── models.pyc
|           ├── tests.py
|           ├── views.py
|           └── views.pyc
|               user_model
|                   ├── admin.py
|                   ├── admin.pyc
|                   ├── apps.py
|                   ├── __init__.py
|                   ├── __init__.pyc
|                   └── migrations
```

```
|   |   └── 0001_initial.py
|   |   └── 0001_initial.pyc
|   |   └── __init__.py
|   |   └── __init__.pyc
|   ├── models.py
|   ├── models.pyc
|   ├── tests.py
|   ├── views.py
|   └── views.pyc
└── user_service
    ├── __init__.py
    ├── __init__.pyc
    ├── settings.py
    ├── settings.pyc
    ├── urls.py
    ├── urls.pyc
    ├── wsgi.py
    └── wsgi.pyc
```

Our project name is `user_service`, and it contains the three apps that are `user_model`, `user_info`, and `user_login`. In the `user_model` app model file is created, and it contains the registration API. In the `user_login` app, it contains the login API, and the `user_info` app is used for fetching the user basic info.

Following sections consist of code of API, model file, and database configuration.

Database connection code

We create the database inside the PostgreSQL then create the user and grant access to the `userservice` database.

The command for creating the database:

```
CREATE DATABASE userservice;
```

The command for creating user:

```
create user test_user with password 'test123';
```

The command for grant permission:

```
GRANT ALL PRIVILEGES ON DATABASE userservice TO test_user;
```

The following code snippet is the part of `settings.py` file:

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.postgresql',  
        'NAME': 'userservice',  
        'USER': 'test_user',  
        'PASSWORD': 'test123',  
        'HOST': '127.0.0.1',  
        'PORT': '5432',  
    }  
}
```

After a database setting, we will also include our app in the `settings` file, which are `user_model` and `user_login`:

```
# Application definition  
  
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'user_model',  
    'user_login',  
    'user_info',  
]
```

The command for database migrations:

```
python manage.py makemigrations user_model  
python manage.py makemigrations user_login  
python manage.py makemigrations user_info  
python manage.py migrate
```

Now, we will create the model file. So the following is the code of `models.py` file. Which is located in the `user_model/models.py`:

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals
from django.db import models

# This is our model for user registration.
class user_registration(models.Model):
    ### The following are the fields of our table.
    fname = models.CharField(max_length=50)
    lname = models.CharField(max_length=50)
    email = models.CharField(max_length=50)
    mobile = models.CharField(max_length=12)
    password = models.CharField(max_length=50)
    address = models.CharField(max_length=200)

    ### It will help to print the values.
    def __str__(self):
        return '%s %s %s %s %s' % (self.fname, self.lname, self.
email, self.mobile, self.password, self.address)
```

In the above code, we created the model and its name is `user_registration`. In the model, we create fields which are `fname`, `lname`, `email`, `mobile`, `password`, and `address`.

API code

Following is our API code:

`userregistartion` API code is mentioned below, it is located in the `user_service/user_model/views.py` path:

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals
from django.http import HttpResponse
from django.shortcuts import render
import json
from django.views.decorators.csrf import csrf_exempt
```

```
from user_model.models import user_registration

### This function is inserting the data into our table.

def data_insert(fname, lname, email, mobile, password, address):
    user_data = user_registration(fname = fname, lname = lname, email =
email, mobile = mobile, password = password, address = address)
    user_data.save()
    return 1

### This function will get the data from the front end.

@csrf_exempt
def registration_req(request):

    ### The Following are the input fields.

    fname = request.POST.get("First Name")
    lname = request.POST.get("Last Name")
    email = request.POST.get("Email Id")
    mobile = request.POST.get("Mobile Number")
    password = request.POST.get("Password")
    cnf_password = request.POST.get("Confirm Password")
    address = request.POST.get("Address")
    resp = {}

    ##### In this if statement, checking that all fields are available.

    if fname and lname and email and mobile and password and cnf_
password and address:

        ### This will check that the mobile number is only 10 digits.

        if len(str(mobile)) == 10:

            ### It will check that Password and Confirm Password both are
the same.

            if password == cnf_password:

                ### After all validation, it will call the data_insert
```

function.

```
        respdata = data_insert(fname, lname, email, mobile,
password, address)

        ### If it returns value then will show success.
        if respdata:
            resp['status'] = 'Success'
            resp['status_code'] = '200'
            resp['message'] = 'User is registered Successfully.'

        ### If it is not returning any value then the show will
fail.
        else:
            resp['status'] = 'Failed'
            resp['status_code'] = '400'
            resp['message'] = 'Unable to register user, Please
try again.'

        ### If the Password and Confirm Password is not matched then
it will be through error.
        else:
            resp['status'] = 'Failed'
            resp['status_code'] = '400'
            resp['message'] = 'Password and Confirm Password should
be same.'

        ### If the mobile number is not in 10 digits then it will be
through error.
        else:
            resp['status'] = 'Failed'
            resp['status_code'] = '400'
            resp['message'] = 'Mobile Number should be 10 digit.

        ### If any mandatory field is missing then it will be through a
```

```
failed message.

else:
    resp['status'] = 'Failed'
    resp['status_code'] = '400'
    resp['message'] = 'All fields are mandatory.'

return HttpResponse(json.dumps(resp), content_type = 'application/json')
```

In the above code, I created the two functions `registration_req` and `data_insert` in the `view.py` file. The `registration_req` function is used for fetching user inputs and validating the data. The `data_insert` function is used for inserting data into a database. I wrote comments on the above of every required place. It helps to understand the code blocks.

`userlogin` API code is mentioned below, it is located in the `user_service/user_login/views.py` path:

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals
from django.http import HttpResponse
from django.shortcuts import render
import json
from django.views.decorators.csrf import csrf_exempt
from user_model.models import user_registration

### This function is created for validating the user.

def user_validation(uname, password):
    user_data = user_registration.objects.filter(email = uname, password = password)
    if user_data:
        return "Valid User"
    else:
        return "Invalid User"

### This function is created for getting the user name and password.

@csrf_exempt
```

```
def user_login(request):
    uname = request.POST.get("User Name")
    password = request.POST.get("Password")
    resp = {}

    if uname and password:
        ## Calling the user_validation function for username and
        password validation.

        respdata = user_validation(uname, password)

        ### If a user is valid then it gives success as a response.

        if respdata == "Valid User":
            resp['status'] = 'Success'
            resp['status_code'] = '200'
            resp['message'] = 'Welcome to Ecommerce website.....'

        ### If a user is invalid then it give failed as a response.

        elif respdata == "Invalid User":
            resp['status'] = 'Failed'
            resp['status_code'] = '400'
            resp['message'] = 'Invalid credentials.'

        ### It will call when user name or password field value is missing.

    else:
        resp['status'] = 'Failed'
        resp['status_code'] = '400'
        resp['message'] = 'All fields are mandatory.'

    return HttpResponse(json.dumps(resp), content_type = 'application/
json')
```

In the above code, I created the two functions `user_login` and `user_validation` in the `views.py` file. The `user_login` function is used for getting input from the users. The `user_validation` function is used for validating the user from the database.

We wrote comments on the above of every required place. It helps to understand the code blocks.

userinfo API: code is mentioned below, it is located in the `user_service/user_info/views.py` path:

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals
from django.http import HttpResponse
from django.shortcuts import render
import json
from django.views.decorators.csrf import csrf_exempt
from user_model.models import user_registration as userreg

### This function is for fetching the user data.
def user_data(uname):
    user = userreg.objects.filter(email = uname)
    for data in user.values():
        return data

### This function is created for getting the username and password.
@csrf_exempt
def user_info(request):
    # uname = request.POST.get("User Name")
    if request.method == 'POST':
        if 'application/json' in request.META['CONTENT_TYPE']:
            val1 = json.loads(request.body)
            uname = val1.get('User Name')
            resp = {}
            if uname:
                ## Calling the getting the user info.
                respdata = user_data(uname)
                dict1 = {}
                if respdata:
```

```
        dict1['First Name'] = respdata.get('fname','')
        dict1['Last Name'] =  respdata.get('lname','')
        dict1['Mobile Number'] = respdata.get('mobile','')
        dict1['Email Id'] =  respdata.get('email','')
        dict1['Address'] =  respdata.get('address','')

    if dict1:
        resp['status'] = 'Success'
        resp['status_code'] = '200'
        resp['data'] = dict1

    ### If a user is not found then it give failed as a
response.

    else:
        resp['status'] = 'Failed'
        resp['status_code'] = '400'
        resp['message'] = 'User Not Found.'

    ### The field value is missing.

    else:
        resp['status'] = 'Failed'
        resp['status_code'] = '400'
        resp['message'] = 'Fields is mandatory.'

    else:
        resp['status'] = 'Failed'
        resp['status_code'] = '400'
        resp['message'] = 'Request type is not matched.'

else:
    resp['status'] = 'Failed'
    resp['status_code'] = '400'
    resp['message'] = 'Request type is not matched.'
```

```
return HttpResponse(json.dumps(resp), content_type = 'application/json')
```

In the above code, I created the two functions `user_data` and `user_info` in the `views.py` file. In the `user_data` function has mentioned the method type POST and request content type should be JSON. Handling is mentioned with every if loop. One more thing, code used repeatedly, so solve this problem we can create a function in the above then use it multiple times. In the user data function, it calls our model then gives a response.

API request and response description

Please refer to the below screenshot to understand the API request and response cycle.

userregistartion API

Following screenshot shows Postman view of userregistration API with request and response:

The screenshot shows the Postman interface with the following details:

- Request URL:** http://localhost:8000/userregistration/ (highlighted with a red box)
- Body (x-www-form-urlencoded):**

Key	Value	Description
First Name	shayank	
Last Name	Jain	
Email Id	shayankit@gmail.com	
Mobile Number	1234567890	
Password	test@1234	
Confirm Password	test@1234	
Address	Mumbai	

This table is labeled **Request Parameters** with a red arrow pointing to it.
- API Response:**

```

1: {
2:   "status": "Success",
3:   "status_code": "200",
4:   "message": "User is registered Successfully."
5: }
```

This JSON object is labeled **API Response** with a red arrow pointing to it.

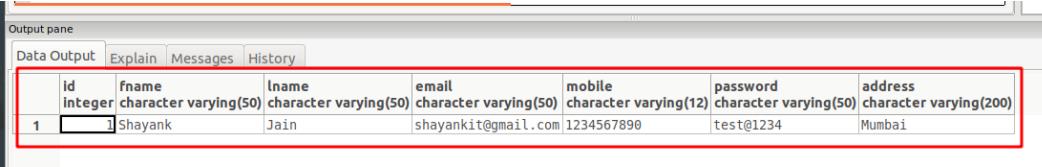
Figure 11.2.2 (a): Postman view of user registration API with request and response

In the above screenshot, we called the user registration API and after storing the data into the database, it is showing the success message. If due to any reason the application is not able to store data then it will show **Unable to register user,**

Please try again message. In the screenshot has mentioned three marked blocks. I also handle the following validations:

- All fields are mandatory, so if a user missed any then it will show **All fields are mandatory** messages.
- The mobile number should be 10 digits; if the user missed then it will show **Mobile Number should be 10 digits**.
- Password and confirm password fields should be the same if the user missed then it will show **Password and Confirm Password should be same**.

Following is the database view after inserting the data:

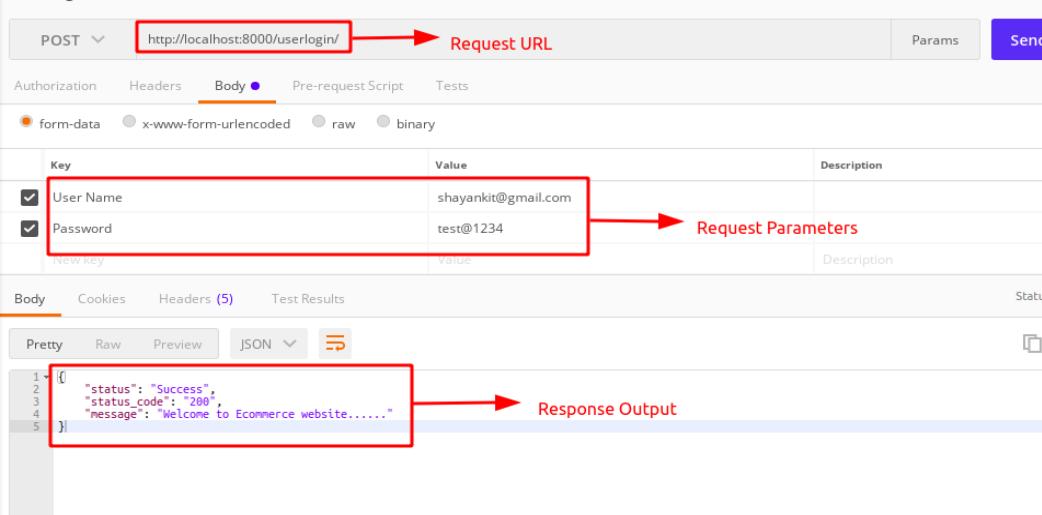


	id integer	fname character varying(50)	lname character varying(50)	email character varying(50)	mobile character varying(12)	password character varying(50)	address character varying(200)
1		Shayank	Jain	shayankit@gmail.com	1234567890	test@1234	Mumbai

Figure 11.2.2 (b): PostgreSQL database record view

userlogin API

Following screenshot shows Postman view of userlogin API with request and response:



POST [Request URL](http://localhost:8000/userlogin/) Params Send

Authorization Headers Body ● Pre-request Script Tests

form-data x-www-form-urlencoded raw binary

Key	Value	Description
User Name	shayankit@gmail.com	
Password	test@1234	

Body Cookies Headers (5) Test Results Status:

Pretty Raw Preview JSON ↴

```

1 < []
2   [
3     {
4       "status": "Success",
5       "status_code": 200,
6       "message": "Welcome to Ecommerce website....."
7     }
8   ]
9 
```

Request Parameters Response Output

Figure 11.2.2 (c): Postman view of user login API with request and response

In the above screenshot, we called the user login API and it will validate the username and password. After validation, it is showing the success message. If a user is not valid then it will show the **Invalid User** message. In the screenshot has mentioned three marked blocks.

userinfo API

Following screenshot shows Postman view of user info API with request and response:

The screenshot shows the Postman interface with the following details:

- Request URL:** http://127.0.0.1:8000/userinfo/ (highlighted with a red box)
- Request Parameter:** {"User Name": "shayankit@gmail.com"} (highlighted with a red box)
- Response Output:** A JSON object containing user information (highlighted with a red box). The JSON is as follows:

```

1 {
2     "status": "Success",
3     "status_code": "200",
4     "data": {
5         "Last Name": "Jain",
6         "First Name": "Shayank",
7         "Address": "Mumbai",
8         "Mobile Number": "1234567890",
9         "Email Id": "shayankit@gmail.com"
10    }
11
  
```

Figure 11.2.2 (d): Postman view of user info API with request and response

The above screenshot has mentioned the three blocks, and each block is marked with comments. This is a raw request, and its content type should be JSON.

Product service

In this service, I used one API to show product details and its name is `getproduct`. It is used for showing all available products from the database. In our code, we created

the GET request.

Folder architecture and code

Following is our project architecture:

```
product_service/
├── manage.py
├── product_model
│   ├── admin.py
│   ├── admin.pyc
│   ├── apps.py
│   ├── __init__.py
│   ├── __init__.pyc
│   ├── migrations
│   │   ├── 0001_initial.py
│   │   ├── 0001_initial.pyc
│   │   ├── __init__.py
│   │   └── __init__.pyc
│   ├── models.py
│   ├── models.pyc
│   ├── tests.py
│   ├── views.py
│   └── views.pyc
└── product_service
    ├── __init__.py
    ├── __init__.pyc
    ├── settings.py
    ├── settings.pyc
    ├── urls.py
    ├── urls.pyc
    ├── wsgi.py
    └── wsgi.pyc
└── query_for_data_insert.txt
```

Database connection code

We create the database inside the PostgreSQL then create the user and grant access to the `productservice` database.

The command for creating the database:

```
CREATE DATABASE productservice;
```

Command for create user:

```
create user product_user with password 'test123';
```

The command for grant permission:

```
GRANT ALL PRIVILEGES ON DATABASE productservice TO product_user;
```

The following code snippet is the part of `settings.py` file:

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.postgresql',  
        'NAME': 'productservice',  
        'USER': 'product_user',  
        'PASSWORD': 'test123',  
        'HOST': '127.0.0.1',  
        'PORT': '5432',  
    }  
}
```

After a database setting, we will also include our app in the `settings` file (installed app section) and its name is `product_model`:

```
# Application definition  
  
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'product_model',  
]
```

The command for database migrations:

```
python manage.py makemigrations product_model  
python manage.py migrate
```

Now, we will create the model file. So the following is the code of `models.py` file. Which is located in the `product_model/models.py`:

```
# -*- coding: utf-8 -*-  
from __future__ import unicode_literals  
from django.db import models  
  
# This is our model for user registration.  
class product_details(models.Model):  
    ### The following are the fields of our table.  
    product_id = models.CharField(max_length=10)  
    product_category = models.CharField(max_length=50)  
    product_name = models.CharField(max_length=100)  
    availability = models.CharField(max_length=15)  
    price = models.CharField(max_length=10)  
  
    ### It will help to print the values.  
    def __str__(self):  
        return '%s %s %s %s' % (self.product_id, self.product_category,  
                               self.product_name, self.availability, self.price)
```

In the above code, we have created the model and its name is `product_details`. In the model, we create fields which are `product_id`, `product_category`, `product_name`, `availability`, and `price`.

API code

Following is our `getproduct` API code:

```
# -*- coding: utf-8 -*-  
from __future__ import unicode_literals  
from django.http import HttpResponseRedirect
```

```
from django.shortcuts import render
import json
from django.views.decorators.csrf import csrf_exempt
from product_model.models import product_details

@csrf_exempt
def get_product_data(request):
    data = []
    resp = {}
    # This will fetch the data from the database.
    prodata = product_details.objects.all()
    for tbl_value in prodata.values():
        data.append(tbl_value)
    # If data is available then it returns the data.
    if data:
        resp['status'] = 'Success'
        resp['status_code'] = '200'
        resp['data'] = data
    else:
        resp['status'] = 'Failed'
        resp['status_code'] = '400'
        resp['message'] = 'Data is not available.'

    return HttpResponse(json.dumps(resp), content_type = 'application/json')
```

In the above code, I created a function and its name is `get_product_data` in the `views.py` file. This function is used for getting the data from the model. This is our GET request. Its purpose is to fetch all data.

Following is the data, which is stored inside the database. For demo purposes, I have inserted the six records, which have different types of products are available:

	id integer	product_id character varying(10)	product_category character varying(50)	product_name character varying(100)	availability character varying(15)	price character varying(10)
1	1	MB001	mobile accessories	power bank	available	500
2	2	MBPH001	mobile phone	One Plus 7T	available	40000
3	3	CLT001	men clothing	US Polo T-shirt	available	800
4	4	TOY001	Children toys	Teddy bear	available	1500
5	5	FUN001	Furniture	Sofa set	available	25000
6	6	MBPH002	mobile phone	MI-7Pro	not available	18000

Figure 11.2.2 (e): Data output from the product_details table

Following screenshot displays API request and response description:

The screenshot shows a POSTMAN interface with the following details:

- Method:** GET
- Request URL:** http://localhost:3001/getproduct/ (highlighted with a red box and arrow)
- Body:** (Empty)
- Headers:** (5 items listed)
- Test Results:** (Empty)
- Response Output:** (highlighted with a red box and arrow)

The response body is a JSON array containing six product objects, each with the following fields:

```

1: [
2:   {
3:     "status": "Success",
4:     "status_code": "200",
5:     "data": [
6:       {
7:         "product_id": "MB001",
8:         "product_category": "mobile accessories",
9:         "price": "500",
10:        "availability": "available",
11:        "id": 1,
12:        "product_name": "power bank"
13:      },
14:      {
15:        "product_id": "MBPH001",
16:        "product_category": "mobile phone",
17:        "price": "40000",
18:        "availability": "available",
19:        "id": 2,
20:        "product_name": "One Plus 7T"
21:      },
22:      {
23:        "product_id": "CLT001",
24:        "product_category": "men clothing",
25:        "price": "800",
26:        "availability": "available",
27:        "id": 3,
28:        "product_name": "US Polo T-shirt"
29:      },
30:      {
31:        "product_id": "TOY001",
32:        "product_category": "Children toys",
33:        "price": "1500",
34:        "availability": "available",
35:        "id": 4,
36:        "product_name": "Teddy bear"
37:      }
38:    ]
39:  }
40: ]

```

Figure 11.2.2 (f): Screenshot of getproduct API with request and response

In the above screenshot, there are two sections shown, one is to request URL and the second is response output. It is a GET type request and when it hits that will give all data, which is available in the database.

Shipment service

In the shipment service, we are using the two APIs to handle the shipment service, which is as follows:

- **shipment_updates API:** This API is used for tracking the product shipment. It is a combination of userinfo and payment_status API. This API is called internally. We will discuss this API descriptively in the payment_service. It's requested parameters are mentioned below:
 - First name
 - Last name
 - Email ID
 - Mobile number
 - Address
 - Product ID
 - Quantity
 - Payment status
 - Transaction ID
- **shipment_status API:** It is a POST request, which takes content type JSON as input and its request parameter is the username. This API is used for getting information about the shipment stage, and it also provides all product details which are bought by the particular user.

Folder architecture and code

Following is our project architecture:

```
shipment_service/
|__ db.sqlite3
|__ manage.py
|__ shipment_service
|   |__ __init__.py
|   |__ __init__.pyc
|   |__ settings.py
|   |__ settings.pyc
```

```
|   ├── urls.py
|   ├── urls.pyc
|   ├── wsgi.py
|   └── wsgi.pyc
└── ship_status
    ├── admin.py
    ├── admin.pyc
    ├── apps.py
    ├── __init__.py
    ├── __init__.pyc
    ├── migrations
    |   ├── 0001_initial.py
    |   ├── 0001_initial.pyc
    |   ├── __init__.py
    |   └── __init__.pyc
    ├── models.py
    ├── models.pyc
    ├── tests.py
    ├── views.py
    └── views.pyc
```

Our project name is `shipment_service`, and it contains the single app that is `ship_status`. We are using the model file, which is part of `ship_status`, and it contains the `shipment_updates` and `shipment_status` API.

Following section consists of the code of API, model file and database configuration.

Database connection code

I wrote the code inside the `models.py`, which is located in the `settings.py` file. For this service, I used the SQLite database.

The following code snippet is the part of `settings.py` file:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
```

```
'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),  
}  
}  
}
```

It is installed in the app section, which is also part of `settings.py` file:

```
# Application definition  
  
INSTALLED_APPS = [  
  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'ship_status',  
]
```

The command for database migrations:

```
python manage.py makemigrations ship_status  
python manage.py migrate
```

Now, we will create the model file. So the following is the code of `models.py` file. Which is located in the `shipment_service/ship_status/models.py`:

```
# -*- coding: utf-8 -*-  
  
from __future__ import unicode_literals  
from django.db import models  
  
# Create your models here.  
  
class shipment(models.Model):  
  
    ### The following are the fields of our table.  
    fname = models.CharField(max_length=50)  
    lname = models.CharField(max_length=50)  
    email = models.CharField(max_length=50)  
    mobile = models.CharField(max_length=12)  
    address = models.CharField(max_length=200)
```

```
product_id = models.CharField(max_length=10)
quantity = models.CharField(max_length=5)
payment_status = models.CharField(max_length=15)
transaction_id = models.CharField(max_length=5)
shipment_status = models.CharField(max_length=20)

### It will help to print the values.
def __str__(self):
    return '%s %s %s %s %s %s %s %s' % (self.fname, self.
lname, self.email, self.mobile, self.product_id, self.address, self.
quantity , self.payment_status, self.transaction_id, self.shipment_
status)
```

In the above code, we created the model that is shipment and its fields are mentioned above.

API code

Following is our both APIs code, which is mentioned below and it is located in the `shipment_service/ship_status/views.py` path:

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals
from django.http import HttpResponse
from django.shortcuts import render
import json
from django.views.decorators.csrf import csrf_exempt
from ship_status.models import shipment as ship_obj

### This function is inserting the data into our table.
def ship_data_insert(fname, lname, email, mobile, address, product_id,
quantity, payment_status, transaction_id, shipment_status):
    shipment_data = ship_obj(fname = fname, lname = lname, email = email,
mobile = mobile,
    address = address, product_id = product_id, quantity = quantity,
payment_status = payment_status, transaction_id = transaction_id,
shipment_status = shipment_status)
```

```
shipment_data.save()
return 1

### This function will get the data from the front end.
@csrf_exempt
def shipment_reg_update(request):
    if request.method == 'POST':
        if 'application/json' in request.META['CONTENT_TYPE']:
            val1 = json.loads(request.body)
            ### This is for reading the inputs from JSON.
            fname = val1.get("First Name")
            lname = val1.get("Last Name")
            email = val1.get("Email Id")
            mobile = val1.get("Mobile Number")
            address = val1.get("Address")
            product_id = val1.get("Product Id")
            quantity = val1.get("Quantity")
            payment_status = val1.get("Payment Status")
            transaction_id = val1.get("Transaction Id")
            shipment_status = "ready to dispatch"

            resp = {}
            ### After all validation, it will call the data_insert
            function.

            respdata = ship_data_insert(fname, lname, email, mobile,
address, product_id, quantity, payment_status, transaction_id, shipment_
status)

            ### If it returns value then will show success.

            if respdata:
                resp['status'] = 'Success'
                resp['status_code'] = '200'
                resp['message'] = 'Product is ready to dispatch.'
```

```
    ### If value is not found then it will give failed in
response.

    else:
        resp['status'] = 'Failed'
        resp['status_code'] = '400'
        resp['message'] = 'Failed to update shipment details.'

    return HttpResponse(json.dumps(resp), content_type = 'application/
json')

### This function is used for finding the transaction.

def shipment_data(uname):
    data = ship_obj.objects.filter(email = uname)
    for val in data.values():
        return val

### This function is used for getting the shipment status

@csrf_exempt
def shipment_status(request):
    if request.method == 'POST':
        if 'application/json' in request.META['CONTENT_TYPE']:
            variable1 = json.loads(request.body)
            ### This is for reading the inputs from JSON.
            uname = variable1.get("User Name")
            resp = {}
            ### It will call the shipment_data function.
            respdata = shipment_data(uname)
            ### If it returns value then will show success.
            if respdata:
                resp['status'] = 'Success'
                resp['status_code'] = '200'
                resp['message'] = respdata
            ### If it is not returning any value then it will show
failed response.
```

```

    else:
        resp['status'] = 'Failed'
        resp['status_code'] = '400'
        resp['message'] = 'User data is not available.'

    return HttpResponse(json.dumps(resp), content_type = 'application/json')

```

In the above code, there are four functions, namely, `ship_data_insert`, `shipment_reg_update`, `shipment_data` and `shipment_status` created in the `views.py` file. Function `ship_data_insert` and `shipment_reg_update` belongs to `shipment_updates` API. It inserts the data into the database. The `shipment_reg_update` function is used to get inputs from the user and `ship_data_insert` used for inserting the data. Function `shipment_status` and `shipment_data` is used for `shipment_status` API. It fetches the data from the database.

API request and response description

Please refer to the below screenshot to understand the API request and response cycle. In the shipment service, only `shipment_status` API is calling from outside, so in the screenshot, `shipment_status` API request and response is shown.

`shipment_status` API:

- **Request:** In *Figure 11.2.2(g)* shown the two blocks, one is our URL, and the second is request parameters.



Figure 11.2.2 (g): Screenshot of shipment_status API with request

- **Response:** It is the response to our request. It has many fields, which are shown in the screenshot:

```
1  {
2      "status": "Success",
3      "status_code": "200",
4      "message": {
5          "product_id": "CLT001",
6          "mobile": "1234567890",
7          "payment_status": "Success",
8          "shipment_status": "ready to dispach",
9          "email": "shayankit@gmail.com",
10         "lname": "jain",
11         "fname": "shayank",
12         "address": "Mumbai",
13         "id": 1,
14         "transaction_id": "1",
15         "quantity": "1"
16     }
17 }
```

Figure 11.2.2 (h): Screenshot of shipment_status API with response

Payment service

In the payment service, we will use the two APIs, which are as follows:

- **initiate_payment API:** This API is used for initiating the product payment. Followings are the request parameter of initiate payment API:
 - User name
 - Product ID
 - Product price
 - Product quantity
 - Payment mode
 - Mobile number

Internally it calls the `shipment_updates` and `userinfo` APIs. To update the shipment database.

- **payment_status API:** This API is used for fetching the user payment status, and its request parameter is the username.

Folder architecture and code

Following is our project architecture:

```
payment_service/
├── db.sqlite3
├── manage.py
└── payment
    ├── admin.py
    ├── admin.pyc
    ├── apps.py
    ├── __init__.py
    ├── __init__.pyc
    ├── migrations
    │   ├── 0001_initial.py
    │   ├── 0001_initial.pyc
    │   ├── __init__.py
    │   └── __init__.pyc
    ├── models.py
    ├── models.pyc
    ├── tests.py
    ├── views.py
    └── views.pyc
└── payment_service
    ├── __init__.py
    ├── __init__.pyc
    ├── settings.py
    ├── settings.pyc
    ├── urls.py
    ├── urls.pyc
    ├── wsgi.py
    └── wsgi.pyc
```

```
└ shipment_update
    ├── admin.py
    ├── admin.pyc
    ├── apps.py
    ├── __init__.py
    ├── __init__.pyc
    ├── migrations
    │   ├── __init__.py
    │   └── __init__.pyc
    ├── models.py
    ├── models.pyc
    ├── tests.py
    ├── views.py
    └── views.pyc
```

Our project name is `payment_service`, and it contains the two apps that are `payment` and `shipment_update`. In the `payment` app, I used the model file that contains the `payment_status` model class. Which is used for storing the product payment details? In the `shipment_update` app, I wrote the logic of calling the `shipment_updates` and `userinfo` API internally.

Following section consists of the code of API, model file and database configuration.

Database connection code

I am using the SQLite database for storage and shown below is the part of `settings.py` file:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

After the database setting, we will also include our app in the `settings` file, which are `payment` and `shipment_update`:

```
# Application definition
```

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'payment',
    'shipment_update',
]
```

Command for database migrations:

```
python manage.py makemigrations payment
python manage.py migrate
```

Now, we will create the model file. So the following is the code of `models.py` file. Which is located in the `payment_service/payment/models.py` path:

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.db import models

# This is our model for user registration.
class payment_status(models.Model):

    ### The following are the fields of our table.
    username = models.CharField(max_length=10)
    product_id = models.CharField(max_length=10)
    price = models.CharField(max_length=10)
    quantity = models.CharField(max_length=5)
    mode_of_payment = models.CharField(max_length=20)
    mobile = models.CharField(max_length=12)
    status = models.CharField(max_length=15)
```

```
### It will help to print the values.  
def __str__(self):  
    return '%s %s %s %s %s %s ' % (self.username, self.product_id, self.price, self.quantity, self.mode_of_payment, self.mobile, self.status)
```

In the above code, we have created the model and its name is `payment_status`. This model is used for storing the payment status.

API code

Following is `initiate_payment` API and `payment_status` API code are mentioned below. It is located in the `payment_service/payment/views.py` path:

```
# -*- coding: utf-8 -*-  
from __future__ import unicode_literals  
from django.http import JsonResponse  
from django.shortcuts import render  
import json  
from django.views.decorators.csrf import csrf_exempt  
from payment.models import payment_status as paystat  
from shipment_update.views import shipment_details_update as ship_update  
  
### This function is for fetching the user data.  
def get_transaction_details(uname):  
    user = paystat.objects.filter(username = uname)  
    for data in user.values():  
        return data  
  
### This function is used for storing the data.  
def store_data(uname, prodid, price, quantity, mode_of_payment, mobile):  
    user_data = paystat(username = uname, product_id = prodid, price = price, quantity = quantity, mode_of_payment = mode_of_payment, mobile = mobile, status = "Success")  
    user_data.save()  
    return 1
```

```
### This function is created for getting the payment.

@csrf_exempt
def get_payment(request):
    uname = request.POST.get("User Name")
    prodid = request.POST.get("Product id")
    price = request.POST.get("Product price")
    quantity = request.POST.get("Product quantity")
    mode_of_payment = request.POST.get("Payment mode")
    mobile = request.POST.get("Mobile Number")

    resp = {}
    if uname and prodid and price and quantity and mode_of_payment and mobile:
        ### It will call the store data function.
        respdata = store_data(uname, prodid, price, quantity, mode_of_payment, mobile)
        respdata2 = ship_update(uname)
        ### If it returns value then will show success.
        if respdata:
            resp['status'] = 'Success'
            resp['status_code'] = '200'
            resp['message'] = 'Transaction is completed.'

        ### If it is returning null value then it will show failed.
    else:
        resp['status'] = 'Failed'
        resp['status_code'] = '400'
        resp['message'] = 'Transaction is failed, Please try again.'

    ### If any mandatory field is missing then it will be through a failed message.

else:
    resp['status'] = 'Failed'
```

```
        resp['status_code'] = '400'
        resp['message'] = 'All fields are mandatory.'

    return HttpResponse(json.dumps(resp), content_type = 'application/
json')

### This function is created for getting the username and password.

@csrf_exempt
def user_transaction_info(request):
    # uname = request.POST.get("User Name")
    if request.method == 'POST':
        if 'application/json' in request.META['CONTENT_TYPE']:
            val1 = json.loads(request.body)
            uname = val1.get('User Name')
            resp = {}
            if uname:
                ## Calling the getting the user info.
                respdata = get_transaction_details(uname)
                if respdata:
                    resp['status'] = 'Success'
                    resp['status_code'] = '200'
                    resp['data'] = respdata

            ### If a user is not found then it give failed as
response.

            else:
                resp['status'] = 'Failed'
                resp['status_code'] = '400'
                resp['message'] = 'User Not Found.'

        ### The field value is missing.

        else:
            resp['status'] = 'Failed'
```

```
        resp['status_code'] = '400'
        resp['message'] = 'Fields is mandatory.'

    else:
        resp['status'] = 'Failed'
        resp['status_code'] = '400'
        resp['message'] = 'Request type is not matched.'


    else:
        resp['status'] = 'Failed'
        resp['status_code'] = '400'
        resp['message'] = 'Request type is not matched.'


    return HttpResponse(json.dumps(resp), content_type = 'application/json')
```

In the above code, I am using the four functions `get_transaction_details`, `store_data`, `get_payment` and `user_transaction_info`, which are created in the `views.py` file. Function `user_transaction_info` and `get_transaction_details` is used for getting the user data. Which fetch data from the `payment_status` table. Function `store_data` and `get_payment` is used for updating the payment and product detail. In the `get_payment` function we called the `shipment_details_update` function, which is located in the `payment_service/shipment_update/views.py` file. The `shipment_details_update` function is calling the `userinfo` and `shipment_updates` APIs.

Following is the code of `payment_service/shipment_update/views.py` file:

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.shortcuts import render
from payment.models import payment_status as paystat
import requests
import json

### This function is for fetching the user data.
def shipment_details_update(uname):
```

```
ship_dict = {}

### It is used for getting data from payment info.
user = paystat.objects.filter(username = uname)
for data in user.values():
    data
    ship_dict['Product Id'] = data['product_id']
    ship_dict['Quantity'] = data['quantity']
    ship_dict['Payment Status'] = data['status']
    ship_dict['Transaction Id'] = data['id']
    ship_dict['Mobile Number'] = data['mobile']

### It is used for getting the user info.
url = 'http://127.0.0.1:8000/userinfo/'
d1 = {}
d1["User Name"] = data['username']
data = json.dumps(d1)
headers = {'Content-Type': 'application/json'}
response = requests.post(url, data=data, headers=headers)
val1 = json.loads(response.content.decode('utf-8'))
ship_dict['First Name'] = val1['data']['First Name']
ship_dict['Last Name'] = val1['data']['Last Name']
ship_dict['Address'] = val1['data']['Address']
ship_dict['Email Id'] = val1['data']['Email Id']

### Data is ready for calling the shipment_updates API.
url = 'http://127.0.0.1:5000/shipment_updates/'
data = json.dumps(ship_dict)
headers = {'Content-Type': 'application/json'}
response = requests.post(url, data=data, headers=headers)
api_resp = json.loads(response.content.decode('utf-8'))

return api_resp
```

In the above code, we have only one function that is created: `shipment_details_update`, which is created in the `views.py` file. In the function, we call the `userinfo` API, which provides the user first name, last name, address, email ID and remaining parameters we get from the `initiate_payment` API. After getting all the parameters, it calls the `shipment_updates` API. Which is going to store the data into shipment database.

API request and response description

Please refer to the below screenshot to understand the API request and response cycle:

- **initiate_payment API:** It is the screenshot of the initiation payment with its request and response parameter:

The screenshot shows a POST request to `http://127.0.0.1:4001/initiate_payment/`. The Body tab is selected, showing form-data parameters:

Key	Value
User Name	shayankit@gmail.com
Product id	CLT001
Product price	800
Product quantity	1
Payment mode	Net banking
Mobile Number	1234567890
New key	Value

The response tab shows a successful JSON response:

```

1  {
2   "status": "Success",
3   "status_code": "200",
4   "message": "Transaction is completed."
5 }
    
```

Figure 11.2.2 (i): Screenshot of `initiate_payment` API with request and response

In the above screenshot we have mentioned the Request URL, request parameters and response output.

- **payment_status API:** This API is taking the content type JSON in a request. The screenshot is divided into two parts that are request and response.

- o **Request:** In the mentioned below, screenshot two blocks are marked. One is the request URL, and the other is the request parameter:

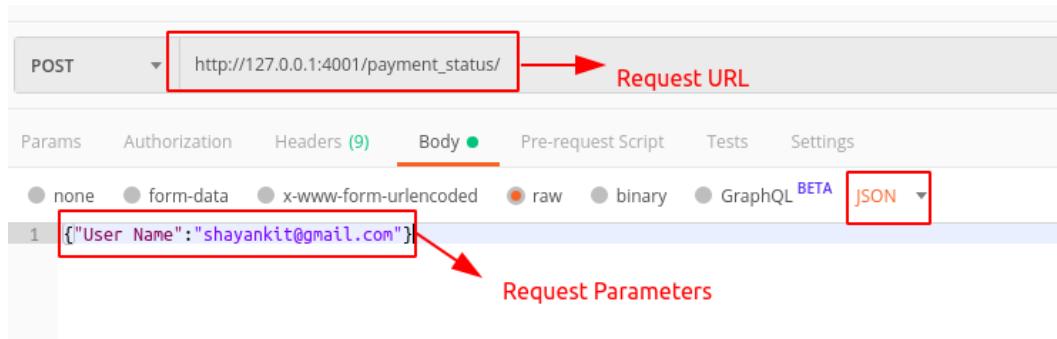


Figure 11.2.2 (j): Screenshot of payment_status API with request

- o **Response:** It is the response to our request:

```

1  [
2      "status": "Success",
3      "status_code": "200",
4      "data": {
5          "username": "shayankit@gmail.com",
6          "status": "Success",
7          "product_id": "CLT001",
8          "mobile": "1234567890",
9          "price": "800",
10         "mode_of_payment": "Net banking",
11         "id": 1,
12         "quantity": "1"
13     }
14 }
```

Figure 11.2.2 (k): Screenshot of payment_status API with response

Microservices deployment with Django

We have seen our services which we created for an e-commerce project. So for deployment, we will take four different servers. Where we will deploy each service individually. The Django project deployment process is simple. We already discussed the previous chapters.

In our architecture, every service is running on different servers but the localhost.

Following is the port details of our service.

- User service is running on the 8000 port.
- Product service is running on the 3001 port.
- Payment service is running on the 4001 port.
- Shipment service is running on the 5000 port.

For the web server, I used the Nginx, Gunicorn, and Supervisor for deployment. You can select other applications for deployment.

Microservice architecture flow figure

Following figure shows microservice architecture flow:

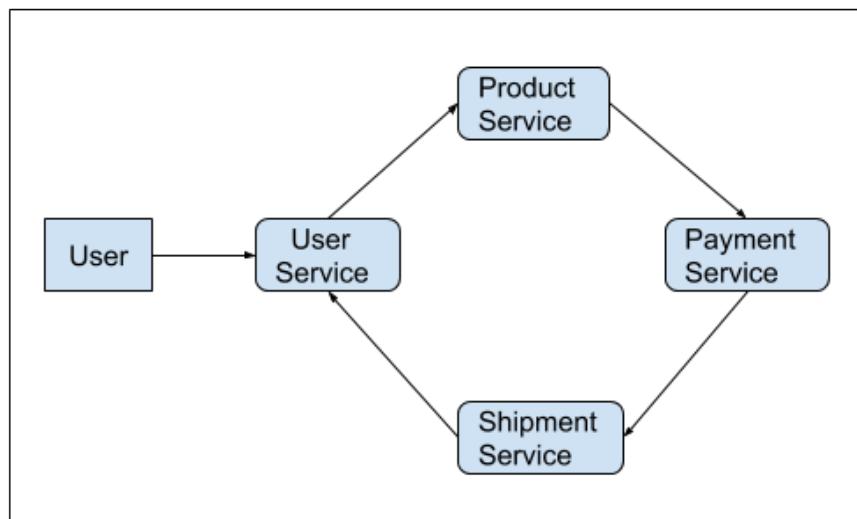


Figure 11.4 (a): Our Ecommerce project microservice architecture

The above diagram is the architecture, which has mentioned four services that are user service, product service, payment service, and shipment service. Our application execution starts in the user section. The user sends a request to the user service, it will call the registration API or login API. If a user is registered with us then it will call login API. After login, the user will call the product API which provides a

list of all items. If the user selects any product from the list then it goes to initiate payment API. `initiate_payment` API will update the payment details and also call the `shipment_updates` API internally that updates the shipment database. After payment, the user will check the payment status through the `payment_status` API. `shipment_updates` API calls internally, so it initiates the shipment process. To check the shipment process users will call the `shipment_status` API.

Conclusion

Our e-commerce demo project is deployed successfully. It follows the microservice approach. It has the basic functionality of an e-commerce website. Project is divided into four services, and all service codes are mentioned above with the description. The demo project's purpose is to explain to the reader how microservices are created and deployed. It will help to understand the project flow.

In the next chapter, you will get the information of **JSON Web Token (JWT)**, which is used for authentication and next level authorization.

Questions

1. How to create microservice architecture?
2. How to call API internally through Django?

CHAPTER 12

JWT Auth Service

Nowadays, security is an important concern for everyone. To secure the communication between any two bodies, JWT is introduced. There are so many other options that are also available, but JWT is lightweight, and it provides the end to end secure communication. In the previous chapter, we have gone through with the microservice implementation. In the microservices to maintain secure communication between the components, we required the JWT authentication. In this chapter, we will understand the JWT authentication in detail.

Structure

- Introduction of JSON Web Tokens (JWT)
- Applicable scenarios JWT implementation
- Structure of JWT token
- JWT working flow
- JWT deployment with Django

Objective

The purpose of including this chapter in the book is because in the microservices every component is independent and for creating the secure channel between other services we need JWT Token. Most people use the JWT in microservice architecture. In the chapter, the reader will gain knowledge of JWT, the perfect scenario for JWT implementation, the structure of JWT token, the flow of JWT token and how to implement JWT with Django. There is also a basic knowledge of the Django rest framework.

Introduction of JSON Web Tokens (JWT)

JWT stands for **JSON Web Tokens**. It is popular and commonly used these days. It is used for single-page applications and microservices. Like his name, it is token-based authentication, which increases the security on API. It maintains the secure communication between the client and server (any bodies). It is also an open, industry-standard *RFC 7519* method, which claims security between two parties. Open industry means it is available to use openly.

It sends the token data in encrypted format for communication. It sends the signed tokens for verifications that are integrated with data.

Applicable scenarios for JWT implementation

There are some scenarios where JWT are useful:

- In the authorization, we can use the JWT. At once the user logged into the server than with the help of token, it can access all resources through token like services, resources, and routes. Nowadays, in many websites, only single time authentication is required then with the help of tokens they don't need to write passwords again.
- When we need to exchange information between the client and the server, it helps us to maintain a secure communication channel. It uses the signed token for communication, which can be a combination of public and private keys. It also has additional parts, which are used for calculating the signature header and payloads.

Structure of JWT token

JWT has three parts, and all are separated by dots (.), which are as follows:

- Header
- Payload
- Signature

It looks like as mentioned below, which are differentiated with a dot(.):

aaaaaaaa.bbbbbbb.ccccccc

The above given is the example of JWT. In the example **aaaaaaaa** part is the header, **bbbbbbb** is the payload part, and **ccccccc** is the signature part of JWT.

Header

It has data into JSON format, and header also has two parts:

- **Type of the token:** It describes the type of our token, which is JWT, and it is fixed.
- **Signing algorithm:** It gives the information of our algorithm, which we use to encrypt the information. HMAC-SHA256 is the algorithm, which is used to secure the data and maintain user authenticity at the same time. It is very complex, and one of the security mechanisms is available nowadays, that is the reason for using the HMAC-SHA256 in JWT commonly.

Let us take the following example:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Above mentioned JSON is the header part of the JWT. After generation the JSON, it uses the Base64Url encoding to encode our data and that the first part of the JWT token. Base64Url is an encoding technique, which converts the data into binary format, which makes our data secure and lightweight for communication between the two parties.

Payload

It is the second part of the token that contains the claims. In simple words, claims are statements, which are about a user and additional data.

They are three types:

- **Registered claims:** It is part of the payload. They are predefined claims which are optional to use, but they are recommended because it provides a set of useful and interoperable claims. Some of them are `iss` (issuer), `exp` (expiration time), `sub` (subject), `aud` (audience), and others.

In the below-mentioned example `sub` is used, so that way we can use the others as well.

- **Public claims:** These can be defined at will by those using JWTs. But to avoid collisions, they should be defined in the IANA JWT Registry or be defined as a URI that contains a collision resistant namespace. In the mentioned example `admin` is the public claims.
- **Private claims:** In the mentioned example `name` is the custom claims, which is created to share information between client and server. It is based on the mutual agreement on using such sensitive information.

Notice that the claim names are only three characters long as JWT is meant to be compact.

An example of payload could be:

```
{  
    "sub": "1234567890",  
    "name": "John Doe",  
    "admin": true  
}
```

The above example has the following parts:

- `sub`: It is an example of a registered claim.
- `name`: It is an example of a private claim.
- `admin`: It is an example of a public claim.

After generating the JSON, it encodes the data in Base64Url to form the second part of the JWT.

Signature

It is the third part of the JWT. It is used for validating the token. It checks that the token is trustworthy and not modified between the communication by the third party.

Signature is the combination of an encoded header, encoded payload, a secret and the specified algorithm of the header.

For example: If we are using the HMAC-SHA256 algorithm in JWT header, then the signature will look like as follows:

```
HMACSHA256(  
    base64UrlEncode(header) + "." +  
    base64UrlEncode(payload),  
    secret)
```

It is used to check that the message wasn't changed between communication.

The output is three Base64-URL strings, which is separated by dots that can be easily passed in HTML and HTTP environments. In the below-mentioned sample, JWT is shown, which has header and payload encoded and it is signed with a secret:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZ-  
SI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36-  
P0k6yJV_adQssw5c
```

In the above example, it is the JWT, which has three parts, whereas: eyJhbGciOi-JIUzI1NiIsInR5cCI6IkpxVCJ9 is header part while eyJzdWIi0iIxMjM0NTY3ODk-wIiwibmFtZSI6Ikpvag4gRG91IiwiWF0IjoxNTE2MjM5MDIyfQ is the payload part and SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c is the signature part.

JWT working flow

JWT is used for authentication, and its flow is mentioned below. It is the first common step that is followed by any communication medium:

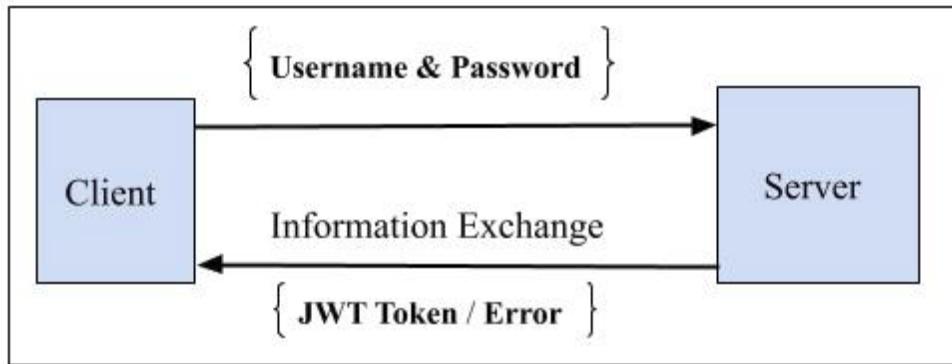


Figure 12.1 (a): JWT token generation

In the above diagram, there are two bodies called **Client** and **Server**. The client will send the request to a server. It will send the username and password in request parameters. After getting the request, the server will validate the username and password then generate the JWT token and send it back to the client as a response. If username and password are valid. Otherwise, it sends an error message. If it sends the JWT, then we can store that into session storage or client-side local storage. In the local storage, it is stored into cookies.

The following image is the second step, where JWT is going to validate:

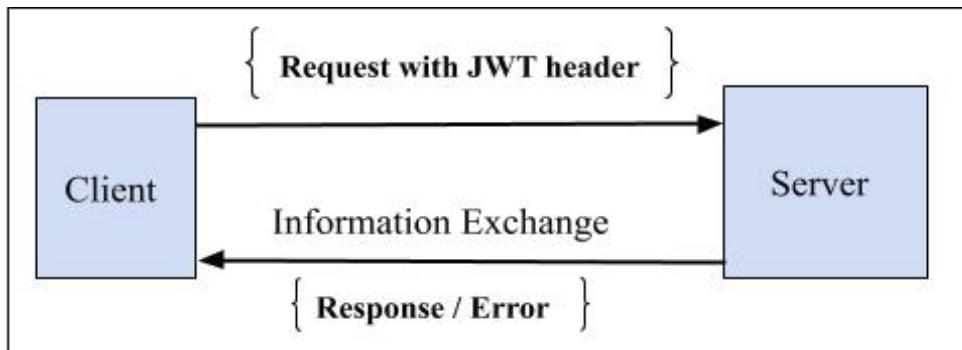


Figure 12.1 (b): JWT token validation

The above diagram has mentioned that the client is sending the request with JWT token in the request header. The server will validate the JWT token then process the request, if the token is valid then it sends a response otherwise that will give an error as the response.

So the JWT token follows the architecture explained above.

JWT deployment with Django

In the Django to deploy a JWT token. I am going to use the Django REST framework, which has some more advanced features than normal Django, and it also has one supportive library for JWT token that is `djangorestframework_simplejwt`.

Django rest framework is similar to Django, all command and settings that we did before for creating user, database, server config, migrations, model, and app configurations are the same, only a few extra features are available in that.

So, let's start with the normal Django project then install the Django REST framework. This project is for JWT token, so I selected the name for our `jwtttestpro`.

Now, project structure will look like as below:

```
jwtttestpro/
└── jwtttestpro
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── manage.py
```

The project is ready, so now we are going to create the superuser with username `test01` and password `pass12345678`. Before creating the user, it requires the database file. Commands are as follows:

```
python manage.py migrate
python manage.py createsuperuser
```

The output of the `createsuperuser` command is as follows:

```
Username (leave blank to use 'sjain'): test01
Email address: test01@gmail.com
Password:
Password (again):
Superuser created successfully.
```

Now, we will install the Django REST framework and `jwt` package command for installations:

```
pip install djangorestframework
pip install djangorestframework_simplejwt
```

To configure the Django REST framework into our Django project, we need to update our `settings.py` file,

```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
]
```

In the application definition, I added the `rest_framework`.

For JWT configuration we need to add following part in the last of our `settings.py` file:

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': ('rest_framework.permissions.
IsAuthenticated',),
    'DEFAULT_AUTHENTICATION_CLASSES': ('rest_framework_simplejwt.
authentication.JWTAuthentication',)
}
```

There is one more change required for JWT token generation. We need to add some packages and a few lines of code into our `urls.py` file, which will generate the JWT token.

Following is the code of our `views.py` file:

```
from django.contrib import admin
from django.urls import path
from rest_framework_simplejwt.views import TokenObtainPairView,
TokenRefreshView
from jwtapp.views import HelloView
```

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('api/testjwt/', HelloView.as_view(), name='tokentest'),
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
]
```

In the above code, the highlighted part is to be written by the user. To generate the JWT token, we use the `rest_framework_simplejwt` package and import the `TokenObtainPairView` and `TokenRefreshView` inbuilt function. Write the two APIs, which will generate the token, one `api/token` is used for generating the token and `api/token/refresh` is used for generating the new token when old is expired. The `api/testjwt/` is used for fetching the data, and it is a GET request.

To check the functionality of the JWT token, we required other API, which provides data for us. So I am going to create a new app, and its name is `jwtapp`. Then put the app entry into the `settings` file (inside the installed app section).

Following is the `views.py` file code, which is located on the path `jwttestpro/jwtapp/views.py`:

```
from django.shortcuts import render
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.permissions import IsAuthenticated

class HelloView(APIView):
    permission_classes = (IsAuthenticated,)

    def get(self, request):
        content = {'message': 'JWT Token test is done successfully!!!!'}
        return Response(content)
```

APIs request and response

We have created the APIs to see the flow of JWT, which is mentioned below with its screenshot and API description.

- **api/token/ API:** This API is used for generating the token for the user. It generates two types of token one is access token and refresh token.
 - o **Request:** Following is the screenshot of `api/token` request, which we are testing through Postman application.

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> username	test01	
<input checked="" type="checkbox"/> password	pass12345678	

Figure 12.2 (a): JWT token generation request param screenshot

In the above screenshot, we have mentioned the request URL and request parameters. In the request parameter, we are passing the user credentials, which are username and password.

- o **Response:** Following is the screenshot of `api/token` response, which is generated on the Postman application.

```

1 {
2   "refresh": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbl90eXBLiIjoiVmcmVzaCisImV4cCI6MTU3MTg1Mjg0MiwianRpIjoiMjI1NmQxN2ZlZDkwNDrlZjgxMTNlMzVhOGYyMTk4MjAiLCJ1c2VxX2lkIjoxf0.
3   "access": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbl90eXBLiIjoiYWNjZXNzIiwixXhwiJoxNTcxNzY2NzQyLCJqdGkiOiJiMTU1ODM2NTMyNWU0NGQ1OTUwMWNmMjE2OTBjOGRmZCisInVzZXJfaWQiOjF9.
4   "

```

Figure 12.2 (b) JWT token generation response screenshot

In the above screenshot, it is mentioned that the response of our API produces the refresh and access token. The access token is used for authorization and refresh token is used for generating the access token again if our access token is expired. An access token has some time limit for expiration.

- **api/testjwt/ API:** It is used for validating the JWT token and calling the GET request for fetching the data.
 - o **Request:** Following is the screenshot of `api/testjwt/` request, which we are testing through Postman application:

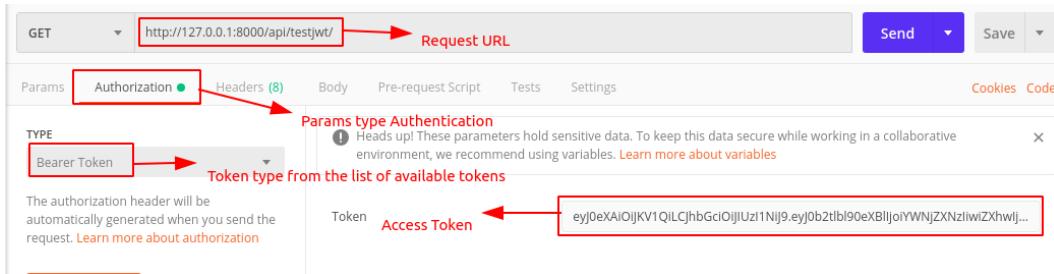


Figure 12.2 (c): JWT token validation request

In the above screenshot, We have the red marked sections, which are request URL, params type where I selected the **Authorization**, type of the token which is bearer token and the token section i put our access token for validation. These all are the important parameter for JWT validation through Postman application.

- o **Response:** Following is the screenshot of `api/testjwt/` response, which is generated on the Postman application:

```

1  {
2      "message": "JWT Token test is done successfully!!!!"
3  }
```

Figure 12.2 (d): JWT token validation response

In the above screenshot, has mentioned the response of our API, it is a success response which comes on after token validation.

If we send the wrong token, then its response is as follows:

Following is our current JWT Token:

`eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbl90eXB1I-joicmVmcmVzaC1sImV4cCI6MTU3MTg1MTU3MywianRpIjoiOGM4ZjcxZ-TM4NDMzNGRiM2JiYTU5MWEwNDEzNjEwYmYiLCJ1c2VyX2lkIjoxfQ.MqJecvKSTo4zWxm4XxS5A-liBmBq9q16sVKhZ-CxRm4`

For testing, I deleted the first character from the key, and now the JWT token is:

`yJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbl90eXB1Ijoic-mVmcmVzaC1sImV4cCI6MTU3MTg1MTU3MywianRpIjoiOGM4ZjcxZTM4ND-MzNGRiM2JiYTU5MWEwNDEzNjEwYmYiLCJ1c2VyX2lkIjoxfQ.MqJecvK-STo4zWxm4XxS5A-liBmBq9q16sVKhZ-CxRm4`

When I sent the wrong key, then its output is mentioned below:

The screenshot shows a Postman request to 'http://127.0.0.1:8000/api/token/refresh/'. The 'Headers' tab is selected, showing 'Content-Type: application/json'. The 'Body' tab is selected, showing a JSON payload with 'refresh': 'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbi90eXB...'. The 'Test Results' tab shows a status of '401 Unauthorized'. The response body is a JSON object with 'detail': 'Given token not valid for any token type', 'code': 'token_not_valid', and 'messages': [{ 'token_class': 'AccessToken', 'token_type': 'access', 'message': 'Token is invalid or expired' }].

Figure 12.2 (e): JWT token validation failed response

- **api/token/refresh/ API:** It is an important API, which comes in use when our generated access token is expired. For again generating the access token we used the refresh token. Its request is mentioned below in *Figure 12.5(f)*.
 - o **Request:** Following is the screenshot of `api/token/refresh/` request, which we are testing through Postman application:

The screenshot shows a POST request to 'http://127.0.0.1:8000/api/token/refresh/'. The 'Body' tab is selected, showing a form-data parameter 'refresh' with value 'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbi90eXB...'. Red arrows highlight the 'Request URL', the 'refresh' key, and the 'refresh' value.

Figure 12.2 (f): Refresh token API request

In the above screenshot, I have red highlighted blocks which are request URL, the value of our KEY parameter and refresh token. It is a normal POST request that we are using for generating the new access token.

- o **Response:** Following is the screenshot of `api/token/refresh/` response, which is generated on the Postman application:

```
1  {
2      "access":  
3          "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNTcxNzY1NjczLCJqdGkiOiI2N  
jg3YzRmN2UzMjIYZvkyWU0YmM2NzYyODE2MDJjYSIsInVzZXJfaWQiojF9.  
Qnx4SP5qvpe-h9YJprfIlnRwaV8Rpgdvkt0vSGxNcMU"  
4  }
```

Figure 12.5 (g): Refresh token API response

The above screenshot has mentioned the response to our request and it returns the access token.

Conclusion

In the chapter, we have seen the JWT, the perfect scenario for JWT implementation, the structure of JWT token, the flow of JWT token, and how to implement JWT with Django. There is also a basic knowledge of the Django REST framework.

In the next chapter, the reader will gain knowledge of asynchronous tasks and what are the benefits of them.

Questions

1. What is JWT?
2. Purpose of using the JWT?
3. Parts of the JWT?
4. What is a header, payload and signature in the JWT?
5. What is the access token?
6. What is a refresh token?
7. What is the flow of JWT?

CHAPTER 13

Asynchronous Tasks

There are many tasks in applications that are not required a direct action of the user. For example, if we have the acknowledgment service in the project, which sends an email to the user, so it is not important that it should trigger the email in real-time. It might be possible that email can be triggered near real-time. So in such a condition to distribute the application load, we use the asynchronous services to perform some tasks. There are many scenarios available where we can implement the asynchronous task system, for example, the generation of reports, the monitoring of devices, the maintenance and the updating of data, copying a file to a storage/remote desktop monitor system, and many more.

Structure

- Introduction of asynchronous task
- Overview of Celery, RabbitMQ and Redis
 - o Celery
 - Introduction
 - Celeryd and Celerybeat
 - Flow and use case
 - o RabbitMQ
 - Introduction
 - Flow and use case
 - o Redis

- Introduction
- Use case of Redis
- Installation and basic configuration of Celery with Django
- Tasks distribution and scheduling

Objective

In this chapter, the readers will gain knowledge of asynchronous tasks and why it is important for the project. I am including this chapter in the book because for a microservices architecture, sometimes we need to implement asynchronous tasks and it helps to improve the system performance. The chapter also includes the way of implementing the asynchronous task with the Django and other available technologies in the market. I am covering a few technologies like Celery, RabbitMQ, and Redis, which are hot topics nowadays for the market. These are the best fit for Django. In the end, we will also talk about the scheduled tasks and how we can use the Ubuntu built-in scheduler for scheduled tasks.

Introduction of asynchronous task

In today's world, many tasks are there who take a long time to perform like generating the reports, back up the database, sending the server failure logs, sending the notification for payment confirmation, failure or product shipment status. These kinds of tasks are time-taking, and it depends upon the system that how much CPU is utilized, RAM availability, and load on the server and network stability. Those are the things that are not in our hands.

So to perform these types of time taking tasks, we execute the task parallelly by using parallel programming, which allowed us to run the task on a small unit and separate thread from the primary application. When a task is completed, then it sends the notification to the main thread. If we are using parallel processing, then it improves the application performance and also increases the responsiveness.

Why we need asynchronous task?

For example, the user sends a request to the server for asking the network report, and it takes 10 min to generate than the user will get the error (timeout error) as a response. For that scenario, we can send the immediate response for the user with the message of the report that will be sent to you on your registered email id. Now let's have a look closely if we handle this request synchronously then the user has to wait till the report is not generated, and request always has the timeout limit. To solve this problem, we use the asynchronous task, which puts the task in the queue, and it will run in the background and after completing the task. It triggers the email for the user.

Overview of Celery, RabbitMQ, and Redis

These are the techniques that work well with the Django framework. Celery is designed on Python to buy its protocol can be implemented in other languages like Python, node js, and PHP clients. RabbitMQ and Redis are also working as a message broker. We can use them for asynchronous tasks. Let's talk about them in detail.

Celery

It is an asynchronous task queuing based distributed message-passing system which mainly focuses on real-time operation, but it supports the scheduling as well. The execution part is called the tasks, and they execute concurrently on a single or more server. It uses the multiprocessing or event. It depends on our choice that we want to execute the tasks asynchronously (It runs the tasks in the background) or synchronously (It waits until its response is ready). To know celery better, please understand first that what is task queue.

Task queue: It is a mechanism, which distributes the work across the threads or machines. In the task queue, dedicated process workers constantly monitor task queues for new work to perform.

Celery provides the following features:

- It is simple to learn, maintain and implement.
- It provides high availability.
- It is fast for handling the tasks.
- It is also flexible, which provide many customized facilities.

Celery has two important parts:

- **Celeryd:** It stands for celery daemon, and it is also known as workers. Which work is to execute the tasks? It executes the task from the queue in the background.
- **Celerybeat:** It is a scheduler. It keeps track of tasks, and the application tells to celery beat that when the task should be executed, which can be in every 5 seconds or in an hour or week.

Flow case

In the celery implementation, we require the message brokers. Celery requires a message transport to send and receive messages. So, now RabbitMQ and Redis come into the picture. These are the message brokers, which transport the message.

Following is the architecture of Celery:

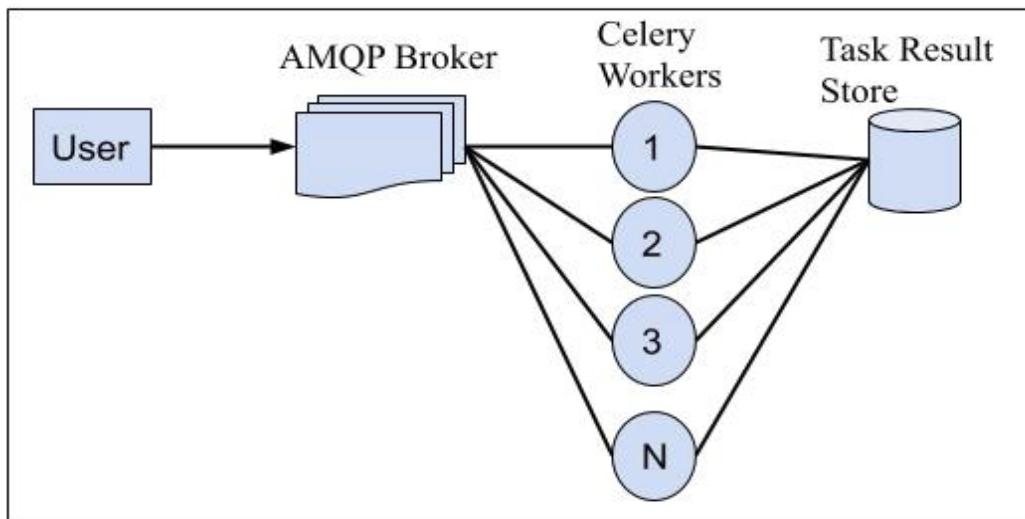


Figure13.1: Architecture of Celery

In the above diagram, four parts are defined user, AMQP broker, celery workers, and task result storage. **Advanced Message Queuing Protocol (AMQP)**, it is a broker that transports the message to the Celery workers. Celery workers are the thread that performs the tasks. Task result storage is the place, where results are stored in the database. So, that is the flow of Celery. As per the diagram application send the task to the broker then it allocated into the celery task queue. Celery workers take the tasks and after completion, it stored the results in a database. In the diagram has mentioned that Celery can create the N numbers of a worker.

Use case

For example, I have an application for expense tracking. Which reads the user SMS and returns his monthly expenses list. So if we have 1 lakh users and every user gets 5-SMS in two minutes, then it becomes 5 lakh requests every two minutes which is not easy for the server to process request synchronously. It will create a heavy load on the server. So, the solution is Celery for asynchronous tasks. It takes the user SMS then creates the task queue for processing the message then workers will execute the SMS separately.

RabbitMQ

It is message-queuing software, which is also known as a queue manager or message broker. In this software where queues are defined by the applications, which connect to transfer a message.

In the message, we can pass any information. For example, we can pass any information of a process or any task which start on another application or call for other service or just a simple text message. It stores the message in a queue until a receiving application takes it off. After retrieving, it processes the message.

RabbitMQ has three parts:

- **Producer:** It means there is a sender. The program which sends the messages is called the producer.
- **Queuing broker:** It means the place where tasks are stored in the queues. The queuing memory is bounded to the host's memory and disk limits, it has a large message buffer. At the same time, many producers can send messages, which will go to one queue and many consumers can receive the data from one queue.
- **Consumer:** It means there is a receiver. The program which waits to receive messages is called the consumer.

RabbitMQ provides the following features:

- It provides the reliability, persistence, and acknowledgements of the delivery.
- It provides the facility of clustering. We can configure many RabbitMQ servers on a local network, and it can be clustered together, in the forming of a single logical broker.
- It has highly available in queues.
- It has the facility of tracing the paths; if our messaging system is misbehaving, then we can trace out the request.

Note: The producer, consumer, and broker should be on the same host.

Flow case

To understand the flow of RabbitMQ we will take the example. For example, we have the file upload system of Passport. If you want to complete the process then you should upload the required documents. Like your Aadhaar card, PAN Card, education certificates, address proofs, election card, and many more. So while the user uploaded the document, it should save on the server-side for future verifications, and in the end, the system should check the list of uploaded documents and verify that all asked documents are uploaded successfully.

The process of uploading and acknowledgement is time consuming. That process we cannot do synchronously, so we prefer the asynchronous system.

Following is the flow diagram of the document upload system:

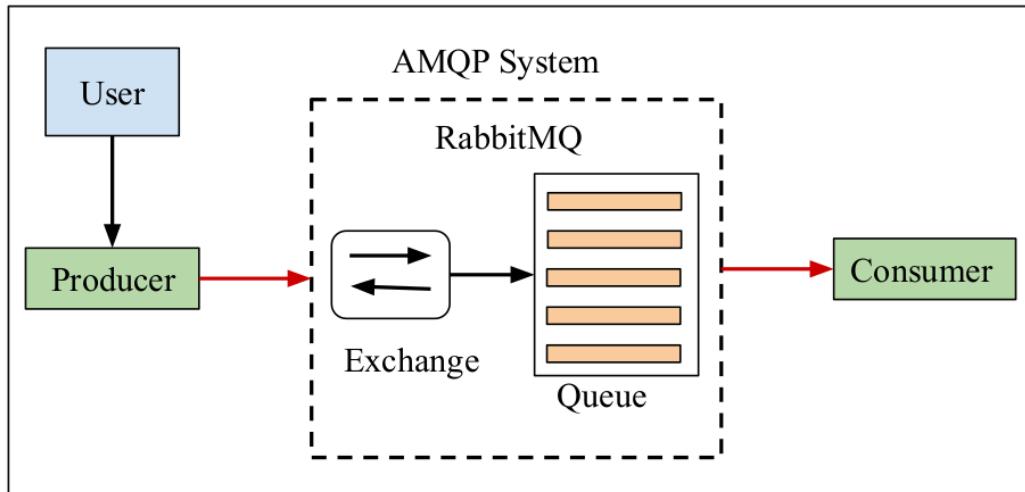


Figure 13.2: Passport application architecture

In the above diagram has mentioned the three parts producer, AMQP and consumer. AMQP is the system which has an exchange and queuing system. The queue is the list of tasks and exchange received and sends the message. The flow is started from the user request, it uploads the document through produces, and it sends a request to the queuing broker. It processes the requests then sends a response to the consumer.

Redis

Redis is an in-memory data structure store, which is used as a database, message broker and cache. We can store various types of data structures such as hashes, strings, sets, lists, and many more.

It stored the data in the key-pair format.

Cache: It is a temporary storage component, where data stored. As a result, we retrieve the data faster than other databases.

Reasons for using Redis:

- It is the fastest database because it stored the data in the cache, and it is a NoSQL database.
- It has built-in replication, Lua scripting and provides high availability.
- It also has a facility of automatic partitioning with Redis Cluster.
- It is open-source, and instead of using a cloud database, we can use Redis, which can save our money.

- Nowadays, many companies are using Redis like GitHub, Weibo, Digg, Pinterest, Craigslist, Flickr, StackOverflow, and Snapchat.
- It supports the many programming languages like Java, PHP, Go, JavaScript, C, C#, C++, Objective-C, Python, and many more.

Use case of Redis

- **Session storage:** Nowadays, it is a common use case, where people using the Redis for session storage. The advantages of Redis, it offers persistence in Memcache. To maintain a cache isn't critical with regards to consistency.
- **Session:** It is an interactive information exchange process between two or more communicating devices, which is temporary. It is established for a certain point of time, after that it expired and closed the connections of devices.

Installation and basic configuration of Celery with Django

We have two options that are available for deploying Django with Celery, which is as follows:

- Deploy Django with Celery and RabbitMQ.
- Deploy Django with Celery and Redis.

So we will go through with each other one by one.

Deploy Django with Celery and RabbitMQ

We will see how we can use Celery and RabbitMQ in our Django project. To use the Celery and RabbitMQ, we need to install some dependency package, which is as follows:

- Command for installing RabbitMQ:
`sudo apt-get install rabbitmq-server`
- Command for installing Celery:
`pip install Celery`

Now, for Django, we will create a separate APP in our Django project and any name you want to pick. I created the App, and its name is `celery_app`. In the `app` folder, we will create the `tasks.py` file, which will be part of our project.

Following is the code of `tasks.py` file:

```
from celery import Celery
```

```
app = Celery('tasks', broker='amqp://localhost')

@app.task
def add(x, y):
    return x + y
```

In the above code, first, we imported the `celery` module. Secondly, we have mentioned the connection with RabbitMQ. The `@app.task`, it means we are calling the decorator and after that create the `add` function, which will return the addition of inputs as a result. So, our task file will take the two numbers and returns the addition.

Our task file is created. So now we need to assign our task file to Celery, the command for assigning task file to Celery:

```
celery -A tasks worker --loglevel=info
```

It will start the Celery, and the output would be looks like as follows:

```
[tasks]
  . tasks.add

[2019-11-10 22:55:56,586: INFO/MainProcess] Connected to amqp://guest:**@127.0.0.1:5672// 
[2019-11-10 22:55:56,594: INFO/MainProcess] mingle: searching for neighbors
[2019-11-10 22:55:57,615: INFO/MainProcess] mingle: all alone
[2019-11-10 22:55:57,635: INFO/MainProcess] celery@sjain ready.
```

Figure 13.3: Celery logs after task file assignment

To check the working of RabbitMQ and Celery, we will create the script and run. Following is our script:

```
from tasks import add
add.delay(100, 4)
```

In the above code, firstly we imported that `add()` function from the task file. Then, we call the `add` function with a built-in delay function. The `delay` method is a shortcut of calling `apply_async()`, which gives control to the program.

For testing, I used the Django shell and ran the code mentioned above. In the following screen has input and output of our code. Following is the screenshot of the input screen:

```
>>> from tasks import add
>>> add.delay(100, 4)
<AsyncResult: 21e7d601-c3d6-442e-acc9-e1f835fa2363>
```

Figure 13.4: Screenshot of Input code

Following is the screenshot of a celery log file, which printed on the screen when we call the add function from the shell. We can also say that it is the output of code:

```
[2019-11-10 22:00:08,429: INFO/MainProcess] Received task: tasks.add[272c6286-8c05-45a7-94ad-03ad35f80484]
[2019-11-10 22:00:08,433: INFO/ForkPoolWorker-3] Task tasks.add[272c6286-8c05-45a7-94ad-03ad35f80484] succeeded in 0.00109552999857s: 104
```

Figure13.5: Screenshot of Output code

Above mentioned all the steps are for configuring the Celery and RabbitMQ with the Django project. In the next part, we will see the Celery and Redis configuration with Django.

Deploy Django with Celery and Redis

We will see how we can use Celery and Redis in our Django project. To use the Celery and Redis, we need to install some dependency package, which is as follows:

- Command for installing Redis:

```
sudo apt update
sudo apt install redis-server
pip install celery redis
```

- Command for installing Celery:

```
pip install Celery
```

Now, for Django, we will create a separate APP in our Django project and any name you want to pick. I created the App, and its name is `celery_app`. In the `app` folder, we will create the `tasks.py` file, which will be part of our project.

Following is the code of `tasks.py` file:

```
from celery import Celery

app = Celery('tasks', broker='redis://localhost')

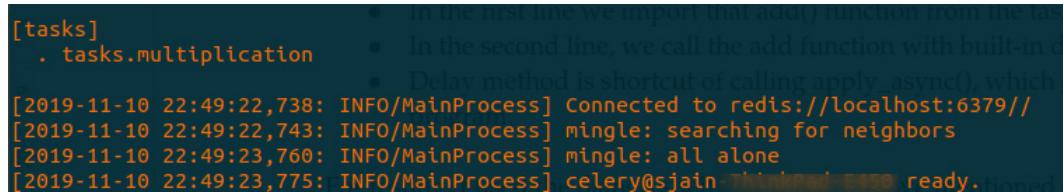
@app.task
def multiplication(x, y):
    return x * y
```

In the above code, firstly we import the `celery` module. Then, we have mentioned the connection with Redis. The `@app.task`, it means we are calling the decorator and after that create the multiplication function, which will return the multiplication of value X and Y. So, our `task` file will take the two numbers and returns the multiplication of given inputs.

Our task file is created. So now we need to assign our task file to Celery, the command for assigning task file to Celery:

```
celery -A tasks worker --loglevel=info
```

It will start the Celery, and the output would be looks like as follows:



The screenshot shows the terminal output of a Celery worker. It starts with importing the multiplication function from the tasks module. Then, it connects to a local Redis instance at port 6379. After connecting, it performs a 'mingle' operation, which finds no neighbors. Finally, it prints a message indicating the worker is ready.

```
[tasks]
  . tasks.multiplication
[2019-11-10 22:49:22,738: INFO/MainProcess] Connected to redis://localhost:6379//
```

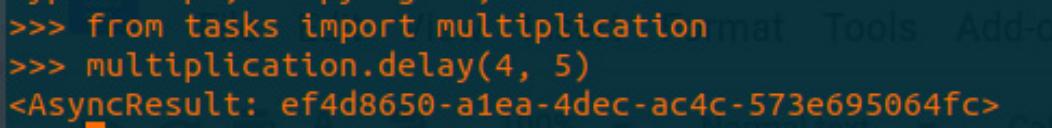
Figure13.6: Output screenshot of Celery logs

To check the working of Redis and Celery, we will create the script and run. Following is our script:

```
from tasks import multiplication
multiplication.delay(4, 5)
```

In the above code, we will import that `multiplication()` function from the task file. Then, we call the `multiplication` function with a built-in `delay` function. The `delay` method is a shortcut of calling `apply_async()`, which gives control to the program.

For testing, I used the Django shell and ran the code mentioned above. In below-mentioned screen has input and output of our code. Following is a screenshot of an input screen:



The screenshot shows the Django shell (ipython) executing the code. It imports the multiplication function and calls its delay method with arguments 4 and 5. The output is an AsyncResult object with a unique identifier.

```
>>> from tasks import multiplication
>>> multiplication.delay(4, 5)
<AsyncResult: ef4d8650-a1ea-4dec-ac4c-573e695064fc>
```

Figure13.7: Screenshot of calling multiplication function with input code

Following is the screenshot of a celery log file, which printed on the screen when we call the `multiplication` function from the shell. We can also say that it is the output of code:



The screenshot shows the terminal output of a celery worker. It shows a task being received and then completed successfully. The task ID is ef4d8650-a1ea-4dec-ac4c-573e695064fc.

```
[2019-11-10 22:50:40,215: INFO/MainProcess] Received task: tasks.multiplication[ef4d8650-a1ea-4dec-ac4c-573e695064fc]
[2019-11-10 22:50:40,216: INFO/ForkPoolWorker-1] Task tasks.multiplication[ef4d8650-a1ea-4dec-ac4c-573e695064fc] succeeded in 0.000259024000115s:
28
```

Figure13.8: Screenshot of celery log response

Above mentioned are all the steps for configuring the Celery and Redis with the Django project.

Tasks distribution and scheduling

In the simple word task distribution means to divide the task into small parts and allocate to resources. In programming, we also divide the modules into sub-modules and distributed them to the team members. So, microservices are one of the examples of task distribution where we divide our monolithic architecture into small services.

There are some benefits of task distribution, which are as follows:

- It improves the efficiency in tasks.
- It provides the scalability, flexibility on application level and also reduces the load.
- It also reduces the development time, which indirectly reduces the money.
- It also increases the team collaboration.
- It helps to improve security.

So, these are the key benefits, which increase the importance of task distribution.

Task scheduling

It is the process where the predefined task runs automatically at the scheduled time. In task scheduling, it is not required to execute the task manually. Task scheduling having many benefits, which are as follows:

- It reduces the cost of the company because it removes human efforts.
- There are a minimum percentage of chances, where mistakes will happen.
- There are many software's are available, which provides a graphical representation that helps us to track the data.
- It helps us to pre-plan our tasks.

Use case

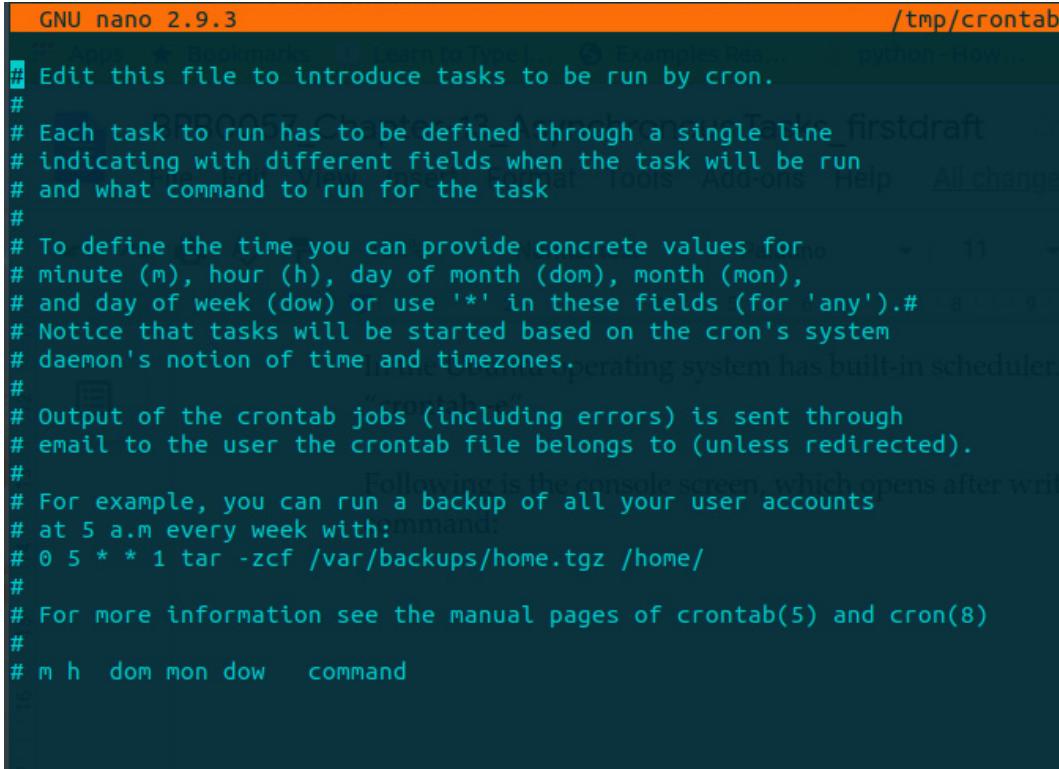
- It is used for report generation like logs report, monthly expense reports, active users reports.
- It is using to send the promotional SMS and email.
- It is used for sending the acknowledgement.
- We can use it for database backup.
- It can be used in sending notification on the network and server failure.
- It is used for executing the scripts.

- It can be used for data tally on a daily, hourly, or minute basis.
- It is also used in banking when they delivered the employee salary.

Schedule the tasks in Ubuntu OS

The Ubuntu OS has a built-in scheduler. Which starts with the command `crontab -e`. After opening the new editable file we write our task with time then save and exit the file.

Following is the console screen, which opens after the command `crontab -e` and hit enter:



The screenshot shows a terminal window with the title 'GNOME Terminal' and the path '/tmp/crontab'. The window contains the text of the crontab(5) man page, which provides instructions for defining cron tasks. The text includes sections on task definition, time specification, and output handling. It ends with examples of cron entries and manual page references.

```
GNU nano 2.9.3 /tmp/crontab
# Edit this file to introduce tasks to be run by cron.
#
# Each task to run has to be defined through a single line
# indicating with different fields when the task will be run
# and what command to run for the task
#
# To define the time you can provide concrete values for
# minute (m), hour (h), day of month (dom), month (mon),
# and day of week (dow) or use '*' in these fields (for 'any').
# Notice that tasks will be started based on the cron's system
# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h dom mon dow   command
```

Figure13.9: Screenshot of Schedule writing screen

After defining the task and time, save the file and exit. It will execute the task automatically on the defined time. So this is the way to create the scheduler in Ubuntu.

In the windows operating system has so many tools are available online which make more easy to create and track the scheduler.

Conclusion

In this chapter, readers gain a basic understanding of an asynchronous task, its benefits, and its use case. It also gets the introduction of Celery, RabbitMQ, and Redis technologies, which are the best fit for Python to perform the asynchronous tasks. This chapter also contains the flow diagram, use case, benefits and configuration codes with steps of Celery, RabbitMQ, and Redis. Readers also get the deployment and testing steps of asynchronous tasks. It also covered the basics of task distribution and task scheduling.

In the next chapter, we will learn about AWS and its serverless architecture.

Questions

1. What are asynchronous tasks?
2. What is Celery?
3. What is RabbitMQ?
4. What is Redis?
5. How to implement Celery and RabbitMQ with Django?
6. How to implement Celery and Redis with Django?
7. What is the task scheduler?
8. What is the task queue?

CHAPTER 14

AWS Serverless

AWS stands for Amazon Web Service. It is a cloud platform, which is adopted by many companies on the globe. It provides 165 services for data. Nowadays, most people know about AWS. It offers many services, which are suitable for microservice deployment. Serverless architecture is in demand; nowadays, many companies are using serverless architecture. In the serverless architecture, we can build and run the applications or services without having to manage the infrastructure of the server. The management part of the servers is done by AWS automatically. In AWS, we don't require to scale and maintain the application, database and storage of the system.

Structure

- Introduction of AWS serverless
 - AWS API Gateway and AWS Lambda
- Serverless microservice architecture on AWS
- Sample code with API Gateway and Lambda
- Django microservice architecture on AWS
 - AWS EC2 service
 - Django microservices architecture on AWS

Objective

This chapter will cover the introduction of Amazon Web Service serverless architecture. Needed service information, which is important for serverless architecture. Use case with the example. Key features of EC2, RDS, API Gateway, Lambda, and AWS DynamoDb. I also covered the Microservice architecture implementation on AWS cloud with example.

At the end of this chapter, readers will be able to create their own serverless and microservice architecture on the AWS cloud and also get a basic understanding of AWS.

Introduction of AWS Serverless

In the serverless architecture, cloud handles the operational responsibility, which helps developers to focus on the code and efficiency of the application. It also increases productivity. The basic functionality of the serverless is that it starts the server when required; otherwise, it is on the downstate. It is helpful to distribute the server load and also reduce the human efforts of monitoring and server maintenance. The most important feature is that it is very cheap in terms of server costing.

In the serverless, when we send the request to the server, it dynamically creates the new instance for serving the request and after sending the response. It destroys the instance.

We can use serverless for many purposes like As a database, for storing the information (Like S3 Bucket), for computation, for stream processing, analytics, message queueing and more.

Let's take the example to understand the serverless architecture, and it's working flow.

Suppose I want to create the API, which takes the employee Id as request parameter and as a response it provides the employee's information like the first name, last name, department, age, address, contact number, and email ID. So for storing the data, we required the database, and for request and response, we required API, which will host on any server. If we get all, then our task will be done.

But in the AWS serverless our requirement will be changed, we will use the following AWS services:

- API Gateway
- Lambda
- RDS (Any database which we want to use) or AWS DynamoDB

AWS API Gateway

It is fully managed services, which make it easy to create, maintain, monitor, publish and create secure APIs at any scale. It is also easy because it can happen in a few clicks. Through this service, we can create REST and WebSocket both types of APIs, which will act as the *first point of access* for any of access data, applications, business logic. It distributes the workload at the running time.

AWS Lambda

This service is used for computation. It means for creating the API we need a server, which will receive the request then process the request as per business logic and in the end, provide the response but in the serverless architecture. The whole process is divided into two parts one is AWS API Gateway, and second is AWS Lambda. Where AWS API Gateway will take the request and provide the response but between the request and response some logic is there, which we called the business logic so that business logic is always written in AWS Lambda. When we call API, it internally calls the API Gateway for request then AWS Lambda for calculation and for providing the response it calls the AWS API Gateway. AWS Lambda has the functionality of auto-scaling, which is helpful while the server has a high load. It is very effective because it's working flow like that if the user calls the API, then it will get activated for processing the request and AWS lambda will be shut down automatically after giving the response. It is as if I want to use the internet, then I need to activate the mobile data on my phone and after finishing my work, I will deactivate my mobile data again. So in that process only used data between the activation and deactivation of mobile data will be chargeable. So only AWS lambda processing time will be chargeable like mobile data.

RDS

Relational Database Services (RDS) is a service which provides the database storage on AWS cloud. It provides the option of selecting the database, which you required like Oracle, MSSQL, MySQL, PostgreSQL, MariaDB, and many more. When we start the RDS instance, it asks us to select the basic config like RAM, storage size and database name, username, password. It also has the in-built functionality of data auto backup and logs system, which makes us use these services. It also has a free RDS instance, which has some basic configuration of RAM and storage but if we to give any demo of any small project, then we can use it. The duration of the free instance is 720 hours. After reaching that limit, AWS charged for using the RDS service.

AWS DynamoDB

When we talk about the highly available database in the serverless architecture, then Amazon DynamoDB comes. It is a NoSQL database, which stores the data in the form of key-value and documents. It has the capability of delivering the single-digit millisecond performance at any scale. In the AWS DynamoDb, they provide the following services, which are fully automated and these services are multimaster creation, built-in security on the database, and it is durable too, restoring and backing up. It also has the ability to caching the data in-memory for internet-scale applications. In terms of performance, DynamoDB can handle 10 trillion requests in a single day and in peak time, more than 20 million requests per second.

In the AWS serverless architecture, we can choose any of the RDS or DynamoDB. It is upon us and business requirements. It also depends on how much load will come.

Following is our AWS Serverless project architecture:

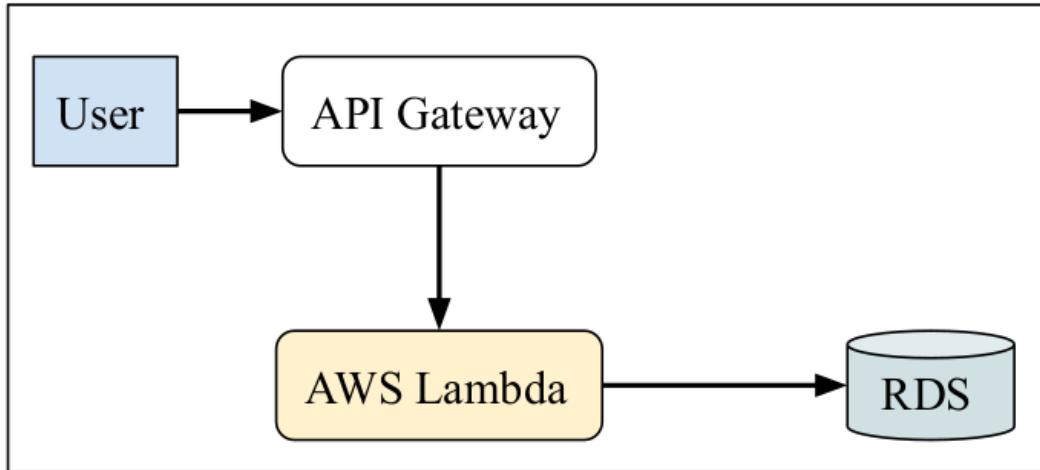


Figure 14.1: AWS serverless architecture

In the above diagram, there are four parts defined: **User**, **API Gateway**, **AWS Lambda**, and **RDS**. The user will call the API, which goes to the AWS cloud and internally it calls the AWS Lambda. AWS Lambda is the file where our code is written to connect and fetch data from the database. RDS is our storage on the AWS cloud, and it stores our data.

Sample code with API Gateway and Lambda

In the sample code, I am creating the GET API, which is based on the serverless architecture. This API request type is GET, so we are not going to use any database. In the serverless, we required the two services, API Gateway and Lambda.

Following are the steps for creating the serverless architecture:

Step-1: We will create the lambda function first then go with the API Gateway. After creating the Lambda function, we will test the lambda function.

Step-by-step guide for creating the Lambda function:

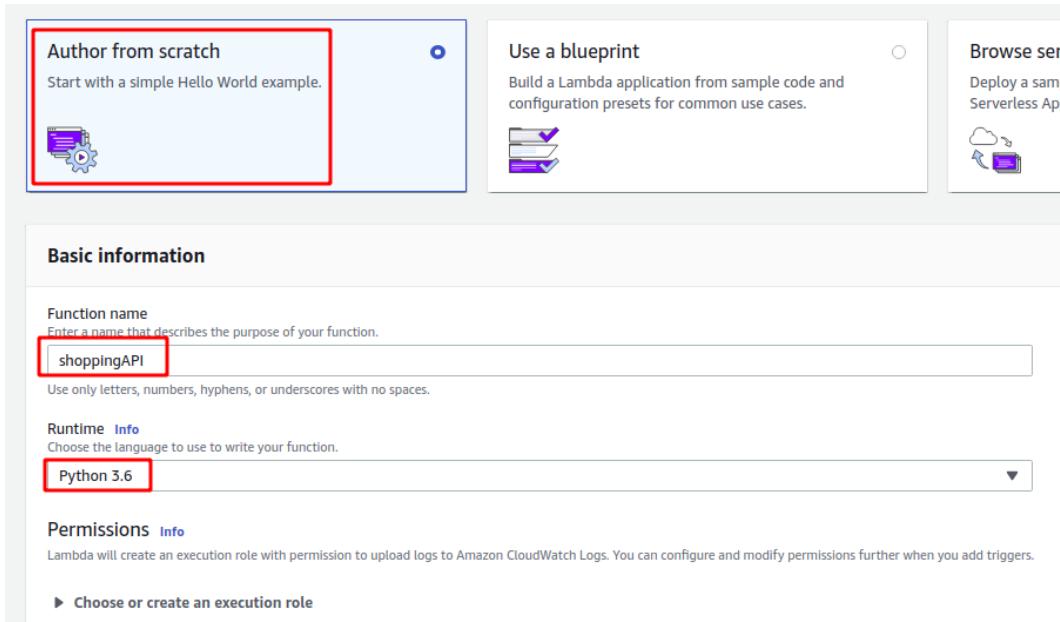


Figure14.2: Lambda function first screen

In the Lambda function has in-built templates, which we selected is shown in the first block. After that it asked for the function name, so we gave that name. In the last, it asked for language, and I choose the Python.

After selecting the options mentioned above and save it creates the default code and its open like that:

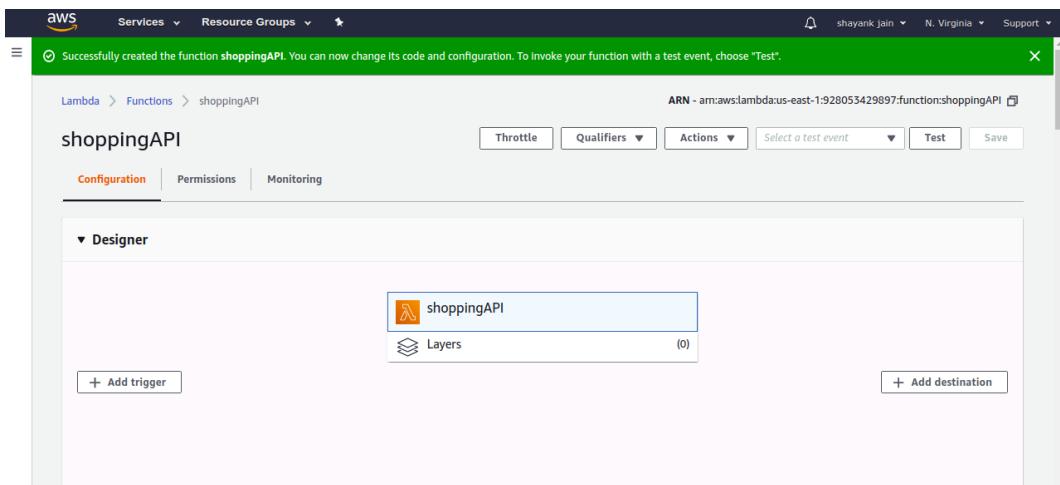
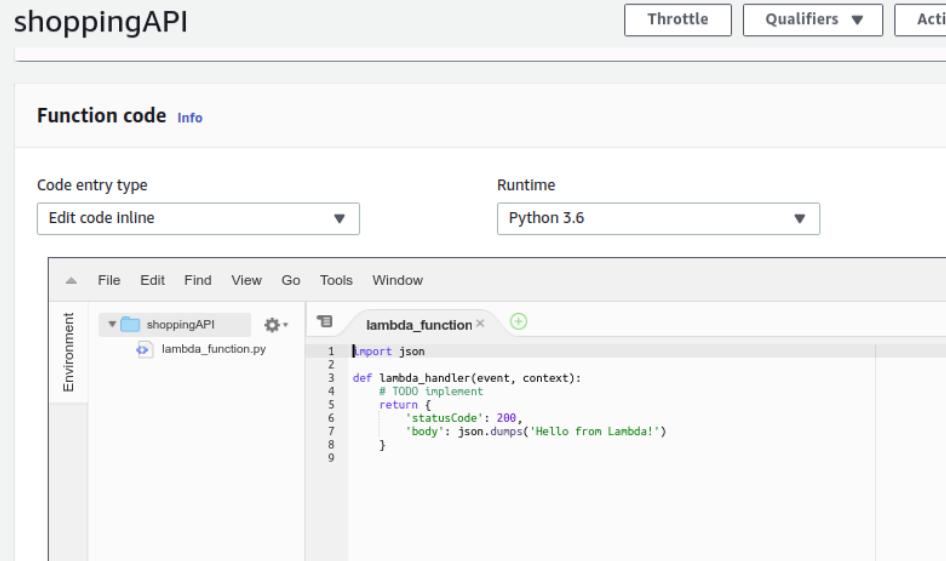


Figure14.3 Lambda function screen after creating the lambda function

Code is mentioned in the next screenshot:



```

import json
def lambda_handler(event, context):
    # TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }

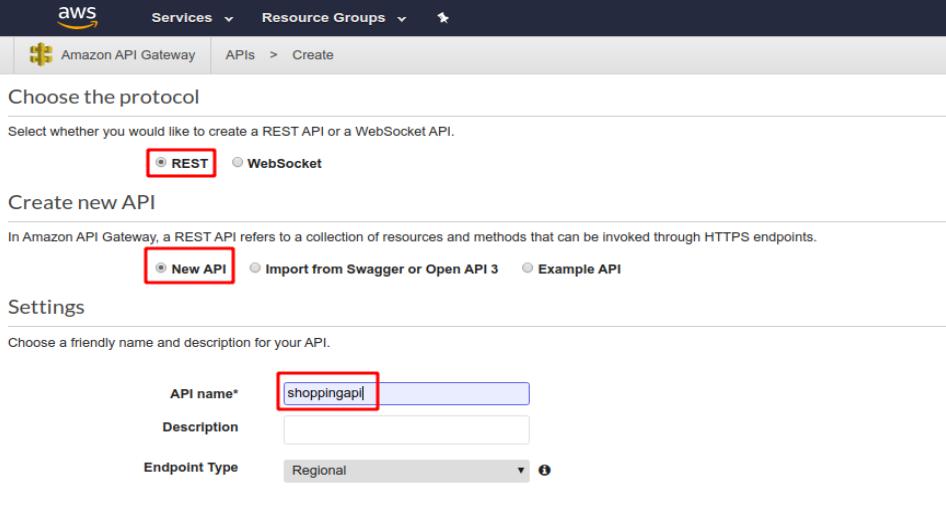
```

Figure14.4: Lambda function auto-generated code

We are creating the Get API so that we will use the same code without change.

Step-2: We will create the new API through API Gateway service then configure it to the Lambda function.

Step-by-step guidelines for creating and configure the API Gateway:



Choose the protocol

Select whether you would like to create a REST API or a WebSocket API.

REST WebSocket

Create new API

In Amazon API Gateway, a REST API refers to a collection of resources and methods that can be invoked through HTTPS endpoints.

New API Import from Swagger or Open API 3 Example API

Settings

Choose a friendly name and description for your API.

API name*	shoppingapi
Description	
Endpoint Type	Regional

Figure14.5: API Gateway first screen

We select the first screen, and it gave us the option to select a type of API, which are REST or WebSocket. So I selected the REST. In select option, it is asking for new API creation or importing or take the example. In the block, it is asking for an API name, and I gave **shoppingapi** that can be anything.

After selecting the all given options, we have to click on the create API option. Which will generate the new page and on the action button it will show the following options:

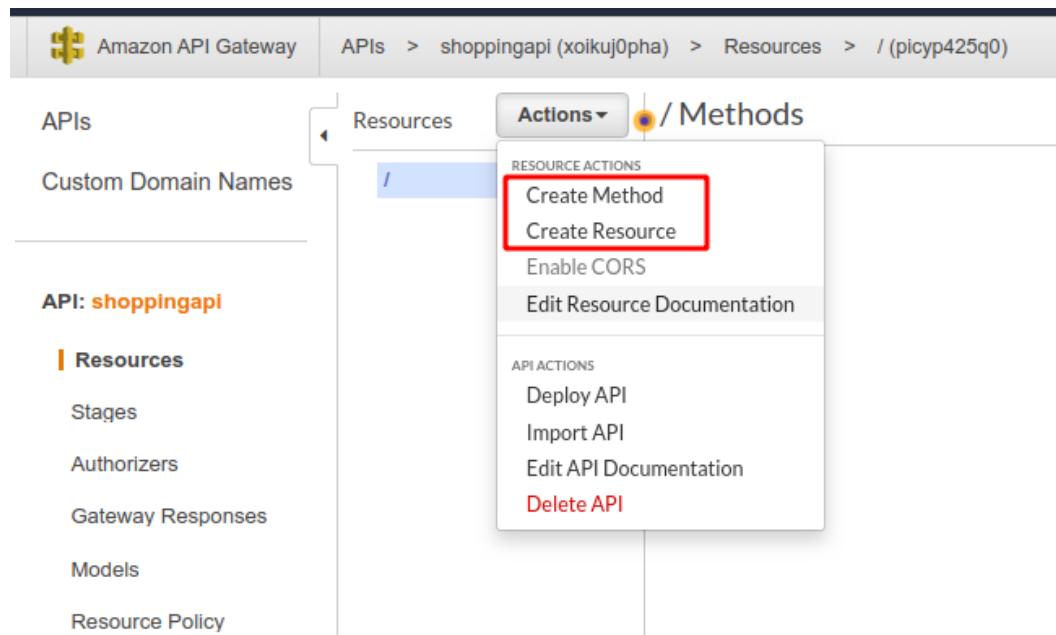


Figure 14.6 API Gateway first screen after creating the API.

In the marked block have mentioned the two options, which are created the method and create a resource. First, we will call the first resource and give the name then call the create method, it will ask for selecting the type of method which we want to create like GET, POST, PUT, and many more. I will select the GET type.

After selecting the GET method structure will look like as follows:

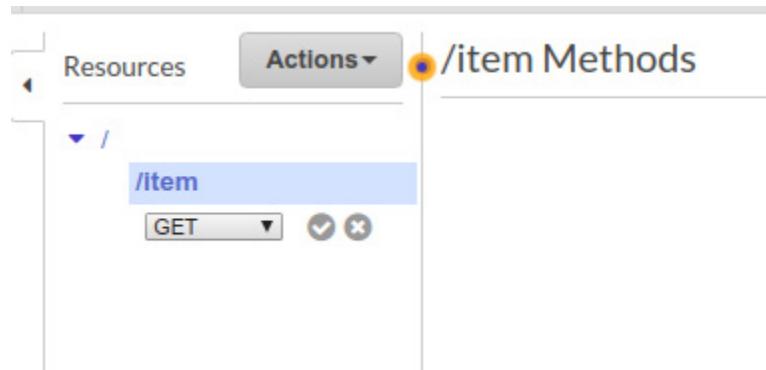


Figure 14.7 Get request structure

Now, the structure is created. When we click on the right sign, then it will create the GET method, and it will look as follows:

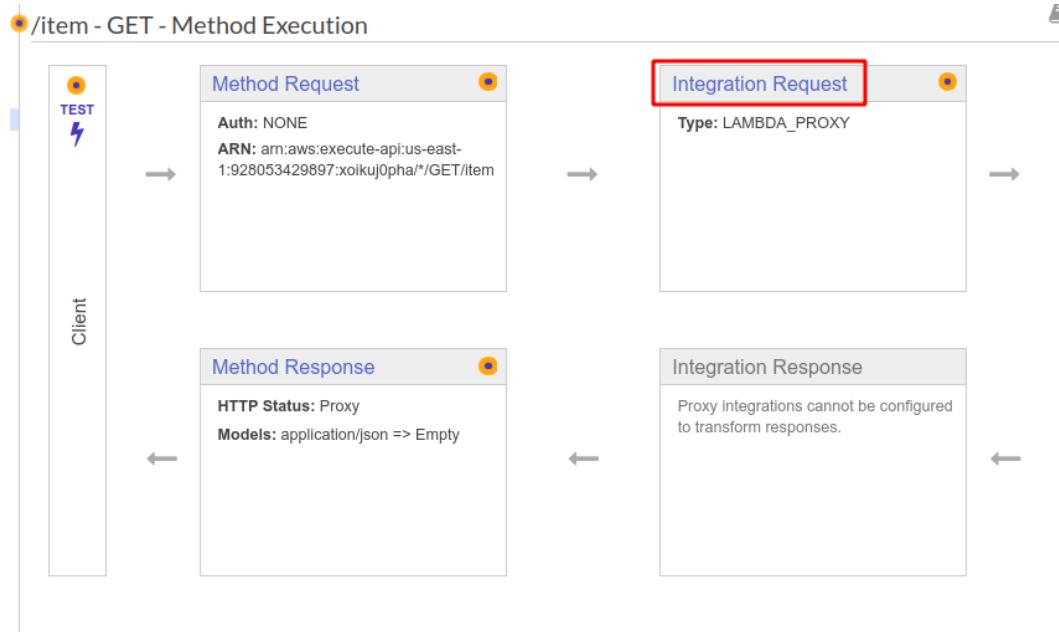


Figure 14.8 Get method configuration options

To configure the GET method, we need to click on the integration request, and then it will open the new page, which is used for lambda function integration. The page screenshot is shown below:

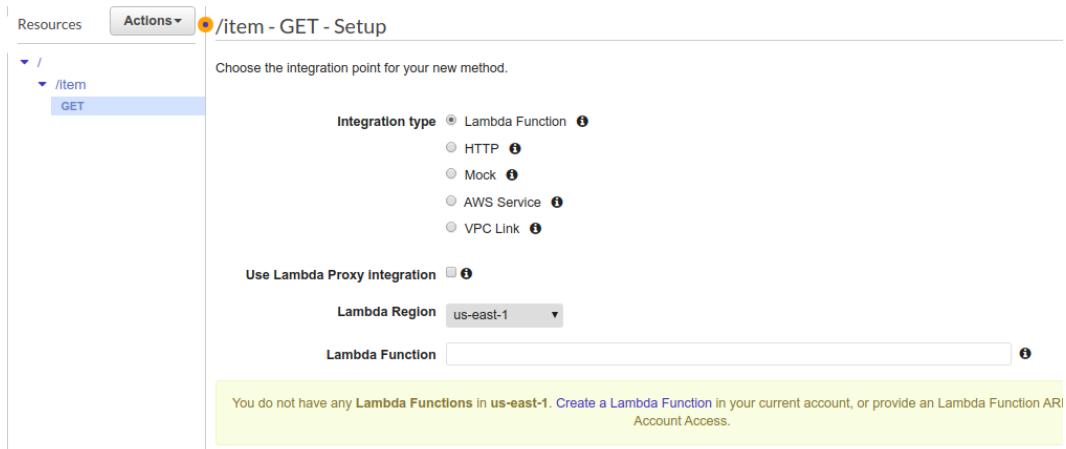


Figure14.9 Get method configuration page on the click of Integration request section

GET method is created successfully, so now we will configure our Lambda function with the method, and it will look like as follows:

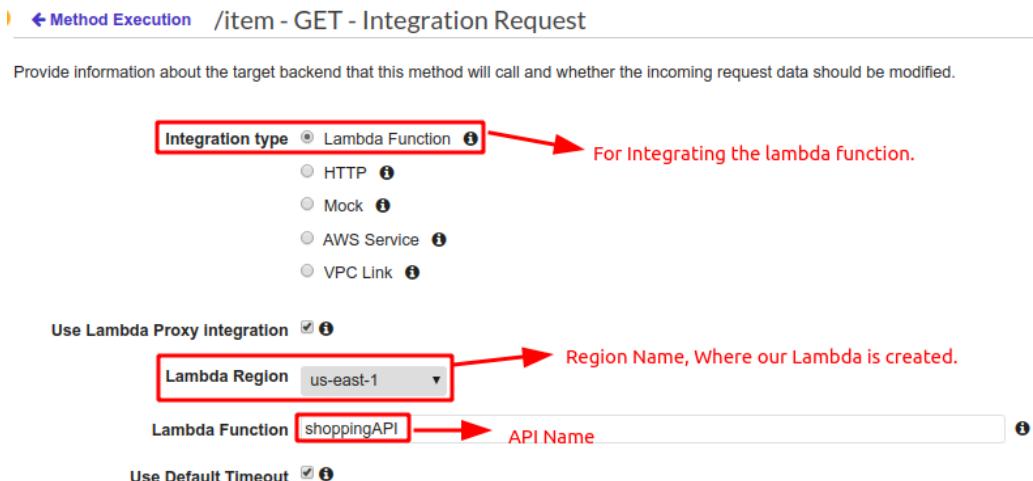


Figure14.10 Lambda function configuration with API Gateway

After filling the above options, we will save the file.

Step-3: Now, we are going to deploy our API code. Refer to the below steps.

Our integration is saved. Now we will click on the action button that provides many options, and we will select the **Deploy API** option:

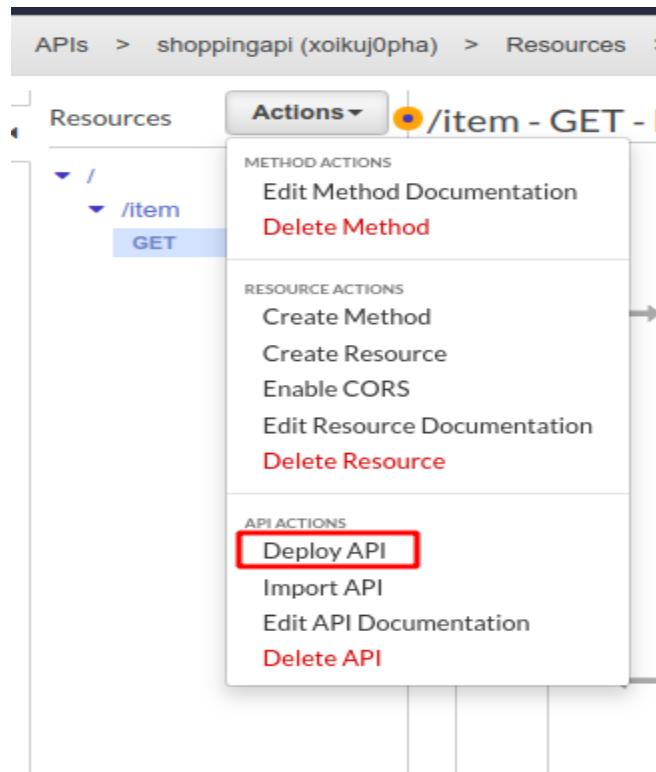


Figure 14.11 API Gateway Deployment options

When we selected the **Deploy API** option, it opens a new window, and it is shown below:

This is a configuration dialog for deploying an API stage. It includes fields for 'Deployment stage' (set to '[New Stage]'), 'Stage name*' (set to 'live', which is highlighted with a red box), 'Stage description', and 'Deployment description'. At the bottom are 'Cancel' and 'Deploy' buttons.

Figure 14.12 API Gateway Deployment Staging screen

In the marked section, we provide the stage, which can be Dev or Prod. we can give any name, which we want to select. After clicking on the **Deploy** button, it generates the URL:

The screenshot shows the AWS Lambda console interface. On the left, there's a sidebar with 'Stages' and a 'Create' button. The main area is titled 'live - GET - /item'. Below the title, there's a red box highlighting the 'Invoke URL' field, which contains the URL: <https://xoikuj0pha.execute-api.us-east-1.amazonaws.com/live/item>. Below this, there's a note: 'Use this page to override the live stage settings for the GET to /item method.' Underneath, there are 'Settings' options: 'Inherit from stage' (radio button selected) and 'Override for this method'.

Figure 14.13 Get API Deployed URL

When we copy the URL and paste it on the browser, then it will generate the following output:

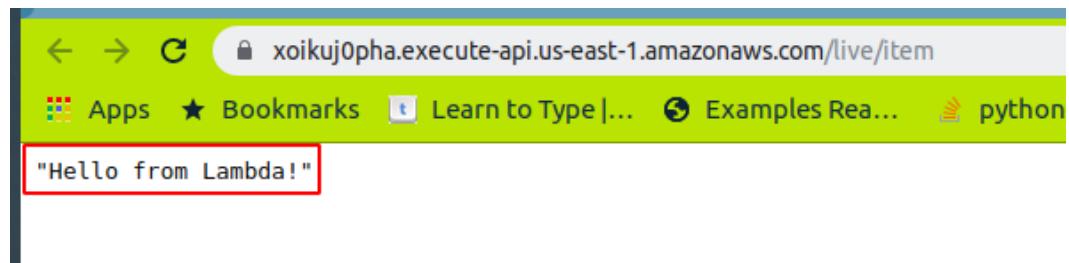


Figure 14.14 Get API Response

Our Get API is deployed successfully and working fine. If anyone follows these steps, then he/she can create and deploy easily.

Serverless microservice architecture on AWS

To understand this, we will take the example of our e-commerce project. Which we used in *Chapter 11: Microservices Deployment with Django*, for showing the deployment of microservice architecture. In our e-commerce project had four services and all have different APIs. So their names are as follows, and every API is explained very well in *Chapter 11: Microservices Deployment with Django*:

- User service
 - userregistration API
 - userlogin API
 - userinfo API

- Product service
 - getproduct API
- Payment service
 - initiate_payment API
 - payment_status API
- Shipment service
 - shipment_updates API
 - shipment_status API

Now, we have a total of eight APIs from the four services. In the serverless architecture, every API required the individual AWS API Gateway and AWS Lambda. We had four services and for microservices, we will create the four individual databases. So if we talk about the user service in terms of serverless architecture. We need to create the three API Gateway and AWS Lambda and they all three are part of user service so APIs will share the single database. For better understanding please refer the *Figure14.2*. In the figure I used the user service, for explaining the serverless architecture, the following is the user service architecture:

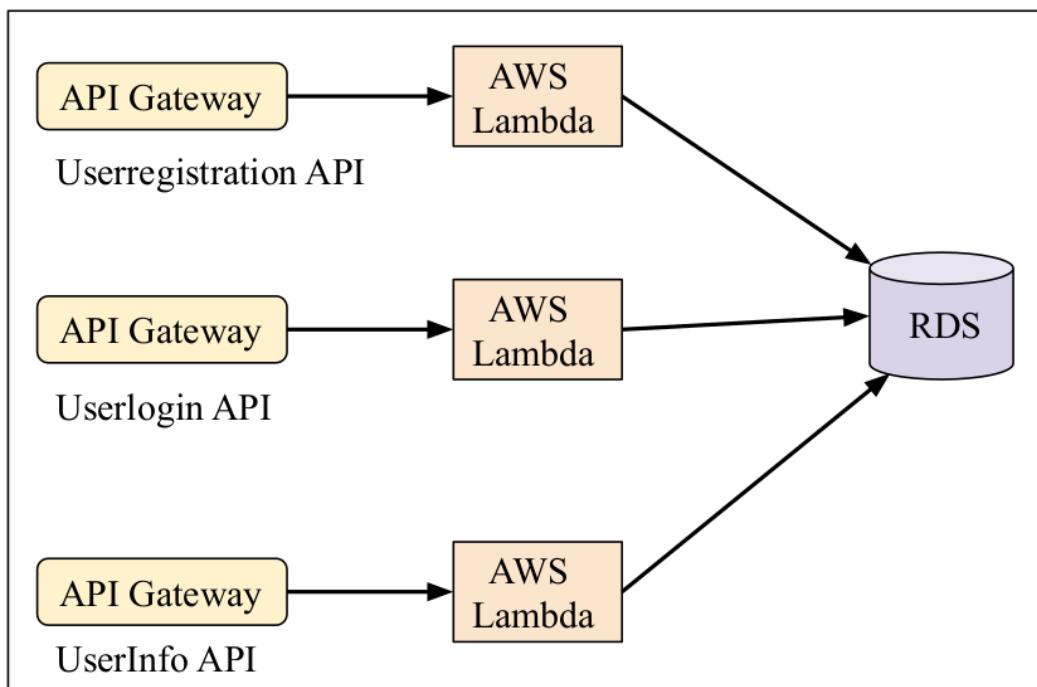


Figure14.15: Serverless Architecture of User Service

In the above diagram, we have four parts defined user, API Gateway, AWS Lambda, and RDS. We have three APIs in the user service. For serverless architecture, we required the individual API Gateway and Lambda function. These APIs are part of a single service, so I took a single RDS. We choose three different RDS or DynamoDb. The count of database service depends on our choices and requirements.

Django microservice architecture on AWS

To understand the topic, I am going to use the example of the same e-commerce website, which is used in the previous topic. As we have seen previously for the serverless architecture, it takes the one API Gateway which integrates with the lambda. So we need to create the individual API Gateway and lambda for every single API.

If we go with the non-serverless architecture, then it required the normal server to implement our services, and as we are creating the microservice architecture, we need four different servers for our user, product, payment and shipment service on the cloud. For servers, AWS provides the EC2 service, which stands for Amazon Elastic Compute Cloud. EC2 service is like a server on the cloud, so as per our microservice architecture, we will take the fours EC2 service and RDS for database storage. Every service will communicate together through APIs.

AWS EC2 service

This service provides scalable computing capacity on the AWS cloud. It is like a virtual machine but on the cloud. It provides the facility to select any available operating system like Ubuntu, Windows, Mac, Redhat, and more and with our required configuration. It also provides the free EC2 instance with minimum configuration, which we can use it for small projects or demo.

The benefit of using the EC2 service is that all hardware, which we needed for any physical machine is not required because all are available on the cloud side. It also saves our money.

The maintenance of the machine, networking, and storage are handled by cloud, and at any point in time, we can upgrade our system hardware like RAM, storage and open any port.

Many supporting services handle the load.

Django microservices architecture on AWS

In the mentioned *Figure 14.3* has shown our e-commerce project architecture with their services and databases. It is our Ecommerce project architecture on Amazon Web Service cloud:

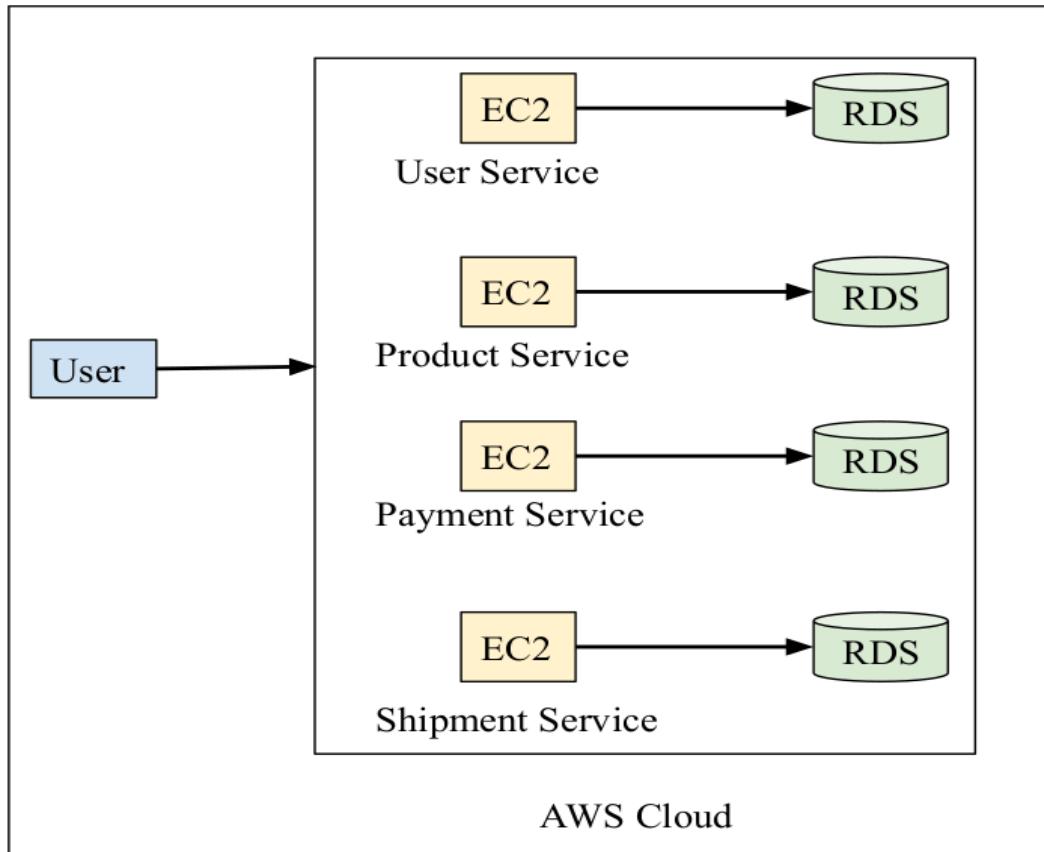


Figure 14.16: Microservice architecture of our e-commerce project on AWS

In the above diagram, there are four services defined as user service, product service, payment service, and shipment service. For each service, we create individual EC2 and RDS. All will communicate to each other through APIs. The flow of architecture is that the user will send the request to any service; for example, it calls the product service so firstly request goes to EC2. It will process the request and if it is required the data then call the RDS. After processing the request, it will send the response back to the user.

Now, we know that how we can deploy our microservice architecture on the AWS cloud. On the AWS cloud, many services are free for a limited period. After reaching the limits, it will cost you. As per AWS cost is count on the usage of the service. If we use the service, then it will cost you otherwise no charges are applied. AWS provides very good support for its customers and documentation is very good and easy. Their instruction made it easy to use any service of AWS.

Conclusion

Now, through this chapter reader has a basic understanding of Amazon web services and what is serverless architecture. It also gets the knowledge of EC2, RDS, API Gateway, Lambda, and AWS DynamoDb with their basic introduction and uses cases. With the help of the e-commerce project example, the reader gets the idea of how we can implement the microservice architecture on the AWS cloud. In the end, the reader also knows the benefits of using the AWS cloud.

In the next chapter, the reader will gain knowledge of how we can adopt the Microservice in practice.

Questions

1. What is AWS?
2. What is serverless architecture?
3. What is the EC2 service?
4. What is the RDS service?
5. Why we use the AWS Lambda?
6. What are the benefits of AWS DynamoDB?

CHAPTER 15

How to Adopt Microservices in Practice

There are so many companies that successfully implemented the microservices architecture and most of them highlight that they are using and it works pretty well like Amazon, SoundCloud, Netflix, and many more. It is good for online literature, but there might be a possibility that your business doesn't act like one of these online companies. It is not bad we can implement microservices in any organization. It means any organization which is doing business on the Internet can improve their project architecture with safety and speed at scale.

Structure

- Guidelines for creating the solution architecture
 - How to introduce microservice in the organization?
 - What are the best practices and processes for microservices?
 - What should be the count of new features or bug fixes in the single release?
 - How can we track that our project is completely transformed into microservice?
 - Is it necessary to write all microservices code in the same programming language?
 - What technologies are available for microservices?

Objective

The chapter is covering some of the most common challenges which we face while adopting a microservice architecture. I tried to put this into question and answer format, which will help you understand. I am hoping that it will help you to solve your problem.

Guidelines for creating the solution architecture

For creating the microservice architecture. Every architecture is different from the others, and every service has distinct design elements. Here is some question that always comes into our mind, which requires the solution. So I tried to include them with a suitable answer.

How to introduce microservice in the organization?

When we worked on the middle grown organization, then there might be a chance that they cannot accept the changes easily. If we make the changes in software, then it can be undone, but if we did on the architecture level, then it can be the cause of higher losses and changes cannot be undone easily. In the software organization, that term is called the refactoring. In which our goal should be to do the same thing, but improve the performance or design. So it can do better. To refactor the application, we should measure our application on the following points:

- Observation on the application performance.
- Overview of the logs file.
- Audit the application source code.
- Review the report of the continuous fall down of our application.

These are the points, and if we are covered, then we can identify the problem successfully.

Now, there is a scope for microservice architecture, which helps us to improve the efficiency of our application. It depends upon our organization's decision on how bigger or smaller changes we can do. In practice, we should start with small changes because it may affect the costing. If we focused on the exact problem then there changes could be expensive. It is our choice to select the right changes.

What are the best practices and processes for microservices?

We already talked about the principles and practices in *Chapter 9: Designing Microservice Systems* in detail.

As per my understanding, we should focus on the principles in the first place, and then it is a company call how they build microservices architecture. The architecture of our software depends on the business requirement, user request and load on the network.

For microservice architecture ideally, it uses the Agile principles, which means including continuous integration and delivery. It also includes the automation process in deployment, which increases the speed of deployment and make more focused on quality code development. Automation decreases operation complexity.

What should be the count of new features or bug fixes in the single release?

In every release, we should try to keep changes to a minimum. If our product releases are frequently, then each release change will be small. Many organizations always limited the number of changes. For example, if we talk about Netflix, They always prefer to make important changes in a single release.

Let's have a look if we are releasing a component, which contains approximately five changes, and if any falls, then it could create a problem in the production environment. Every component has a dependency on other functionality, so we go with all of them together then it can create the problem. So we should limit our changes in a single release that also increases the safety of each service.

How can we track that our project is completely transformed into microservice?

In the software field, it is hard to say some projects are done completely. After completion, there is always a scope of product improvement. In technology, creating the product and maintenance of the system is never done. In the development, experienced architects and developers spend their time identifying the better solution for their system.

It rarely works. The main benefit of microservices is that if we did changes over time to time, then it is not as costly or harmful for the system. In the comparison of tightly coupled monolithic architecture, it impacts the model. It is also important to upgrade systems from time to time because working with outdated technology creates the problem of system performance and maintenance.

Is it necessary to write all microservices code in the same programming language?

The single word answer is *NO* because the external interface is more important than the internal programming language of the component, which is APIs.

Until the two-component uses the same network protocols and message format to exchange information. So that is why internal programming should be the same is not necessary. In many companies, they stick with the same technology for major components because it makes it easy for them to provide the training and manage the resources.

What technologies are available for microservices?

Throughout this book, we have seen the many technologies, which can use in microservices and that are as follows:

- For API, we used Django and Python.
- For the Asynchronous task, we can use the Celery, Redis, and RabbitMQ.
- For the database, we can SQLite, MySQL, and PostgreSQL.
- For project deployment on the AWS Cloud, many services are available.
- For deployment, we can use Apache Tomcat, Nginx, UWSGI, Supervisor, or Gunicorn.

So these all are the technology stack, which we can use as per business requirement. We can also use the other technology that is depending on the developer's expertise and demands of the project.

Conclusion

Our microservice architecture doesn't have to be like the other companies which we heard of Netflix, Amazon, or SoundCloud. There is no formal definition of microservices, but it is easy for you to call microservice architecture and what you did. Throughout this chapter, we have seen some questions and answers which can help us to adopt the microservice in our organization.