Python Requests Essentials

Python is one of the most popular programming languages of our era; the Python Requests library is one of the world's best clients, with the highest number of downloads. It allows hassle-free interactions with web applications using simple procedures.

You will be shown how to mock HTTP Requests using HTTPretty, and will learn to interact with social media using Requests. This book will help you to grasp the art of web scraping with the BeautifulSoup and Python Requests libraries, and will then paddle you through Requests impressive ability to interact with APIs. It will empower you with the best practices for seamlessly drawing data from web apps. Last but not least, you will get the chance to polish your skills by implementing a RESTful Web API with Python and Flask!

Who this book is written for

If you are a Python administrator or developer interested in interacting with web APIs and have a passion for creating your own web applications, this is the book for you. Basic knowledge of Python programming, APIs, and web services will be an advantage.

What you will learn from this book

- Demonstrate the use of Python Requests with the help of examples
- Use the Requests module to deal with the inner sections of the request-response cycles
- Implement the RESTful Web API with Python Requests
- Authenticate Requests using different authentication methods
- Emulate server actions and interact with a mock server
- Interact with social networking sites such as Facebook, Twitter, and reddit
- Scrape the Web with Python Requests and BeautifulSoup
- Build your own web application with Flask

In this package, you will find:

- The author's biography
- A preview chapter from the book, Chapter 3 'Authenticating with Requests'
- A synopsis of the book’s content
- More information on Python Requests Essentials
About the Authors

Rakesh Vidya Chandra has been in the field of software development for the last 3 years. His love for programming first sparked when he was introduced to LOGO in his school. After obtaining his bachelor's degree in Information Technology, he worked with Agiliq Info Solutions and built several web applications using Python. Rakesh is passionate about writing technical blogs on various open source technologies. When not coding, he loves to dance to hip-hop and listens to EDM.

Bala Subrahmanyam Varanasi loves hacking and building web applications. He has a bachelor's degree in Information Technology. He has been in the software industry for the last three and a half years, where he worked with Agiliq Info Solutions and Crypsis Technologies. Bala has also built different web applications using Python, Ruby, and JavaScript. Apart from coding, he is interested in entrepreneurship and is the founder of Firebolt Labs. Currently, he is working as a software engineer at TinyOwl Technology.
Preface

Python is one of the evolving language of our era, and it's gaining a lot of attention these days. It is one of the powerful and flexible open source languages instilled with powerful libraries. For every python developer, Requests is the library that comes to mind first when he/she needs to interact with the Web. With its batteries included Requests turned the process of interacting with Web a cakewalk and stands as one of the world's best client with more than 42 million downloads.

With the rise of social media, APIs turn to be a must have part of every application, and interacting with them in the best way possible is going to be a challenge. Getting to know how to interact with APIs, building an API, scraping the web, and such stuff will help every budding web developer to reach new heights.

What this book covers

Chapter 1, Interacting with the Web Using Requests, covers topics such as why Requests is better than urllib2, how to make a simple request, different types of response content, adding custom headers to our Requests, dealing with form encoded data, using the status code lookup, locating the request redirection, location, and timeouts.

Chapter 2, Digging Deep into Requests, talks about using session objects. It discusses the structure of request and response, prepared Requests, SSL verification with Requests, streaming uploads, generators, and event hooks. This chapter also demonstrates using proxies, link headers, and transport headers.

Chapter 3, Authenticating with Requests, introduces you to the different types of procedures that are in practice for authentication. You will gain knowledge on authenticating with OAuth1, digest authentication, and basic authentication.
Chapter 4, Mocking HTTP Requests Using HTTPretty, covers HTTPretty along with its installation and usage. Then, we deal with real-time examples and learn how to mimic the actions of a server using Python Requests and HTTPretty.

Chapter 5, Interacting with Social Media Using Requests, covers significant ground. Starting with an introduction to the Twitter API, Facebook API, and reddit API, we will move on to discover ways in which we can obtain keys, create an authentication request, and work with various examples to interact with social media.

Chapter 6, Web Scraping with Python Requests and BeautifulSoup, empowers you to have a better understanding of the libraries that are used in scraping the Web. You will also be introduced to using the BeautifulSoup library, its installation, and procedures to scrape the web using Python Requests and BeautifulSoup.

We would like to thank www.majortests.com for allowing us to base the examples in this chapter around their website.

Chapter 7, Implementing a Web Application with Python Using Flask, gives an introduction to the Flask framework and moves on to discuss how to develop a simple Survey application which deals with creating, listing and voting various questions. In this chapter you will acquire all the knowledge required to build a web application using Flask.
Requests supports diverse kinds of authentication procedures, and it is built in such a way that the method of authentication feels like a cakewalk. In this chapter, we opt to throw light on various types of authentication procedures that are used by various tech giants for accessing the web resources.

We will cover the following topics:

- Basic authentication
- Digest authentication
- Kerberos authentication
- OAuth authentication
- Custom authentication

**Basic authentication**

Basic authentication is a popular, industry-standard scheme of authentication, which is specified in HTTP 1.0. This method makes use of a user-ID and password submitted by the user to get authenticated. The submitted user-ID and password are encoded using Base64 encoding standards and transmitted across HTTP. The server gives access to the user only if the user-ID and the password are valid. The following are the advantages of using basic authentication:

- The main advantage of using this scheme is that it is supported by most of the web browsers and servers. Even though it is simple and straightforward, it does have some disadvantages. Though all the credentials are encoded and transferred in the requests, they are not encrypted which makes the process insecure. One way to overcome this problem is by using SSL support while initiating a secure session.
Authenticating with Requests

- Secondly, the credentials persist on the server until the end of the browser session, which may lead to the seizure of the resources. And also, this authentication process is wide open to Cross Site Request Forgery (CSRF) attacks, as the browser automatically sends the credentials of the user in the subsequent requests.

The basic authentication flow contains two steps:

1. If a requested resource needs authentication, the server returns http 401 response containing a WWW-Authenticate header.
2. If the user sends another request with the user ID and password in the Authorization header, the server processes the submitted credentials and gives the access.

You can see this in the following diagram:

![Diagram showing basic authentication flow]

Using basic authentication with Requests

We can use the requests module to send a request to undergo basic authentication very easily. The process can be seen as follows:

```python
>>> from requests.auth import HTTPBasicAuth
>>> requests.get('https://demo.example.com/resource/path', auth=HTTPBasicAuth('user-ID', 'password'))
```

In the preceding lines of code, we performed basic authentication by creating an HTTPBasicAuth object; then we passed it to the auth parameter, which will be submitted to the server. If the submitted credentials gets authenticated successfully, the server returns a 200 (Successful) response, otherwise, it will return a 401 (Unauthorized) response.
Digest authentication

Digest authentication is one of the well known HTTP authentication schemes, which were introduced to overcome most of the drawbacks of basic authentication. This type of authentication makes use of user-ID and password just like Basic authentication, but the major difference comes in the picture, when the credentials get transferred to the server.

Digest authentication increases the security of the credentials by going an extra mile with the concept of cryptographic encryption. When the user submits the password for the sake of authentication, the browser will apply an MD5 hashing scheme on it. The crux of the process lies in using nonce values (pseudo-random numbers) while encrypting the password which decreases the replay attacks.

This type of authentication gains more strength, as the password in this encryption is not used in the form of plain text. The cracking of the password hashes becomes difficult in digest authentication with the use of a nonce, which counters the chosen plain text attacks.

Even though Digest authentication overcomes most of the drawbacks of Basic authentication, it does have some disadvantages. This scheme of authentication is vulnerable to man-in-the-middle attacks. It reduces the flexibility of storing the password in the password's database, as all the well designed password databases use other encryption methods to store them.
Using Digest authentication with Requests

Using Digest authentication with `requests` is very simple. Let us see how it's done:

```python
>>> from requests.auth import HTTPDigestAuth
>>> requests.get('https://demo.example.com/resource/path',
               auth=HTTPDigestAuth('user-ID', 'password'))
```

In the preceding lines of code, we carried out digest authentication by creating an `HTTPDigestAuth` object and setting it to the 'auth' parameter which will be submitted to the server. If the submitted credentials gets authenticated successfully, the server returns a 200 response, otherwise, it will return a 401 response.

Kerberos authentication

Kerberos is a type of Network authentication protocol, which uses a secret key cryptography to communicate between the client and the server. It was developed at MIT to mitigate many security problems like replay attacks and spying. It makes use of tickets to provide authentication for the server-side resources. It followed the idea of avoiding additional logins (single sign on) and storing the passwords at a centralized location.

In a nutshell, the authentication server, the ticket granting server and the host machine act as the leading cast in the process of authentication.

- **Authentication Server**: A server-side application which aids in the process of authentication by making the use of submitted credentials of a user
- **Ticket Granting Server**: A logical key distribution center (KDC) which validates the tickets
- **Host Machine**: A server which accepts the requests and provides the resources
You can see this in the following diagram:

Authentication with Kerberos takes place in the following steps:

1. When a person logs into his machine with the credentials, a request will be sent to ticket granting ticket (TGT).

2. If the verification of the user turns out to be true, when checked from the user database, a session key and a TGT will be created by the authentication server (AS).

3. Thus, the obtained TGT and session key will be sent back to the user in the form of two messages, in which TGT will be encrypted with the ticket granting the server's secret key. The session key will be encrypted with the client secret key and it contains a time stamp, life time, TGS name and TGS session key.

4. The user on the other end, after receiving the two messages, uses the client secret key that is, the user's password to decrypt the messages of the session key. The TGT cannot be decrypted without the TGS secret key.
5. With the available information of the session key and the TGT, the user can send a request for accessing the service. The request contains two messages and some information at this point. In the two messages, one is an encrypted message, containing a user ID and timestamp. The other is a decrypted message, containing the HTTP service name and the life time of the ticket. With the above two messages, an authenticator and TGT will be sent to the ticket granting server.

6. The messages and the information (Authenticator and TGT) will be received by the TGS, and it will check for the credibility of the HTTP service from the KDC database and decrypt both the authenticator and the TGT. Once everything goes fine, the TGS tries to verify some important parts like client ID, time stamp, lifetime of TGT and authenticator. If the verification turns out to be successful, then the TGS generates an encrypted HTTP service ticket, HTTP service name, time stamp, information about the ticket validity and the session key of HTTP service. All of the preceding ones will be encrypted by the HTTP Service session key and will be sent back to the user.

7. Now, the user receives the information and decrypts it with the TGS session key that he/she received in the earlier step.

8. In the next step, to access the HTTP service, the user sends an encrypted HTTP service ticket and an authenticator which is encrypted with the HTTP service session key to the HTTP service. The HTTP service uses its secret key to decrypt the ticket and takes hold of the HTTP service session key. With the acquired HTTP service session key, it decrypts the authenticator and verifies the client ID time stamp, lifetime of ticket, and so on.

9. If the verification turns out to be successful, the HTTP service sends an authenticator message with its ID and time stamp to confirm its identity to the user. The user's machine verifies the authenticator by making use of HTTP service session key and identifies the user as an authenticated one who accesses the HTTP service. From then onwards, the HTTP service can be accessed by the user without any bumps, until the session key expires.

Kerberos is a secure protocol as the passwords from the user can never be sent as plain text. As the process of authentication takes place with the agreement of both the client and the server through encryption and decryption, it turns out to be a rigid one to break to some extent. The other advantage comes from its capability to give server access to the user until the session key expires without reentering the password.
Kerberos does have some disadvantages:

- The server must be continuously available for the verification of the tickets which may result in blocking, if the server goes down.
- User's keys are saved on a central server. A breach of this server may compromise security for the whole infrastructure.
- Kerberos necessitates a heavy infrastructure, which means a simple web server is not sufficient.
- The setup and the administration of Kerberos requires specialized skills.

**Using Kerberos authentication with Requests**

Requests takes the support of the requests-kerberos library for the purpose of authentication. For this reason, we should first install the requests-kerberos module.

```
>>> pip install 'requests-kerberos'
```

Let's have a look at the syntax:

```
>>> import requests
>>> from requests.kerberos import HTTPKerberosAuth
>>> requests.get('https://demo.example.com/resource/path', auth=HTTPKerberosAuth())
```

In the preceding lines of code, we carried out Kerberos authentication by creating an HTTPKerberosAuth object and setting it to the auth parameter which will be submitted to the server.

**OAuth authentication**

OAuth is an open standard authorization protocol, which allows client applications a secure delegated access to the user accounts on third party services such as Google, Twitter, GitHub and so on. In this topic, we are going to introduce the two versions:- OAuth 1.0 and OAuth 2.0.
OAuth 1.0

OAuth authentication protocol came up with an idea of mitigating the usage of passwords, replacing them with secure handshakes with API calls between the applications. This was developed by a small group of web developers who are inspired by OpenID.

Here are the Key terms used in the process of OAuth authentication.

- **Consumer**: The HTTP Client who can make authenticated requests
- **Service Provider**: The HTTP Server, which deals with the requests of OAuth
- **User**: A person who has the control over the protected resources on the HTTP Server
- **Consumer Key and Secret**: Identifiers which have the capability to authenticate and authorize a request
- **Request Token and Secret**: Credentials used to gain authorization from the user
- **Access Token and Secret**: Credentials to get access to the protected resources of the user

You can see this in the following diagram:
Initially, the client application asks the service provider to grant a request token. A user can be identified as an approved user by taking the credibility of the request token. It also helps in acquiring the access token with which the client application can access the service provider’s resources.

In the second step, the service provider receives the request and issues request token, which will be sent back to the client application. Later, the user gets redirected to the service provider’s authorization page along with the request token received before as an argument.

In the next step, the user grants permission to use the consumer application. Now, the service provider returns the user back to the client application, where the application accepts an authorized request token and gives back an access token. Using the access token, the user will gain an access to the application.

**Using OAuth 1.0 authentication with Requests**

The `requests_oauthlib` is a an optional library for oauth which is not included in the Requests module. For this reason, we should install `requests_oauthlib` separately.

Let us take a look at the syntax:

```python
>>> import requests
>>> from requests_oauthlib import OAuth1
>>> auth = OAuth1('<consumer key>', '<consumer secret>',
...               '<user oauth token>', '<user oauth token secret>')
>>> requests.get('https://demo.example.com/resource/path', auth=auth)
```

**OAuth 2.0**

OAuth 2.0 is next in line to OAuth 1.0 which has been developed to overcome the drawbacks of its predecessor. In modern days, OAuth 2.0 has been used vividly in almost all leading web services. Due to its ease of use with more security, it has attracted many people. The beauty of OAuth 2.0 comes from its simplicity and its capability to provide specific authorization methods for different types of application like web, mobile and desktop.
Basically, there are four workflows available while using OAuth 2.0, which are also called grant types. They are:

1. **Authorization code grant**: This is basically used in web applications for the ease of authorization and secure resource delegation.
2. **Implicit grant**: This flow is used to provide OAuth authorization in Mobile Applications.
3. **Resource owner password credentials grant**: This type of grant is used for applications using trusted clients.
4. **Client credentials grant**: This type of grant is used in machine to machine authentication. An in-depth explanation about grant types is out of the scope of this book.

OAuth 2.0 came up with capabilities which could overcome the concerns of OAuth 1.0. The process of using signatures to verify the credibility of API requests has been replaced by the use of SSL in OAuth 2.0. It came up with the idea of supporting different types of flow for different environments ranging from web to mobile applications. Also, the concept of refresh tokens has been introduced to increase the security.

Let us take a look at the usage:

>>> from requests_oauthlib import OAuth2Session
>>> client = OAuth2Session('<client id>', token='token')
>>> resp = client.get('https://demo.example.com/resource/path')

**Custom authentication**

Requests also provides the ability to write a new or custom authentication based on the user's needs and flexibility. It is equipped with `requests.auth.AuthBase` class which is a base class for all the authentication types. This can be achieved by implementing the custom authentication in the `__call__()` of `requests.auth.AuthBase`.

Let us take a look at its syntax:

```python
>>> import requests
>>> class CustomAuth(requests.auth.AuthBase):
...     def __call__(self, r):
...         # Custom Authentication Implementation
```
>>> requests.get('https://demo.example.com/resource/path',
... auth=CustomAuth())

Summary
In this chapter, we gained knowledge of various types of authentication like Basic authentication, Digest authentication, Kerberos authentication, OAuth 1.0 authentication and OAuth 2.0 authentication which are supported by Requests. Later, we got an idea of how to use various types of authentications and the flows of the process. We also learned to use our own custom authentication and gained the knowledge of making different authentications work with Requests and the ways to use them with Requests.

In the next chapter, we will be getting to know all about a handy module, HTTPretty.
Where to buy this book

You can buy Python Requests Essentials from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.