Chapter 13

Web Services

In the previous chapter, we have learned how to use sockets and protocols to establish communication through the client-server architecture. We also could transfer data between different computers. In this chapter, we will learn how to create a communication and data transference between computers by using the world wide web.

A *Web Service* is a grouping of client-server applications that communicate through the web using a specially designed protocol. We can see this type of service as a function or a black box that can be accessed by other programs via The Web. For example, let's consider the HTTP protocol used by internet browsers to get information from a website. Each time we execute any action within a web browser, a web server send a request to the browser. The server replies sending the required information (a web page for example), then our browser interprets that page and displays it to us in a friendly format. Web Services work in a similarly way, but the main difference is that the communication occurs between applications. Client and server must know the format of the exchanged information. It also helps to develop applications according to the hardware and keep the same communication structure.

Figure 13.1 shows the example of a house controlled remotely. This house is controllable by different computers with an internet connection through a web server. This server allows interaction between the house and other devices. One of the advantages of this model is the simplicity of the interaction between the applications, because of the independence of the languages used to implement the client and server. Each node can request the server to send or modify certain parameters of the house. Nodes may publish the information using JSON or XML or using any other format as well. A protocol bound used as an interface between two or more programs is known as **Application Programming Interface**, or **API** for short. APIs consist of a set of instructions that allows the applications to access the web.

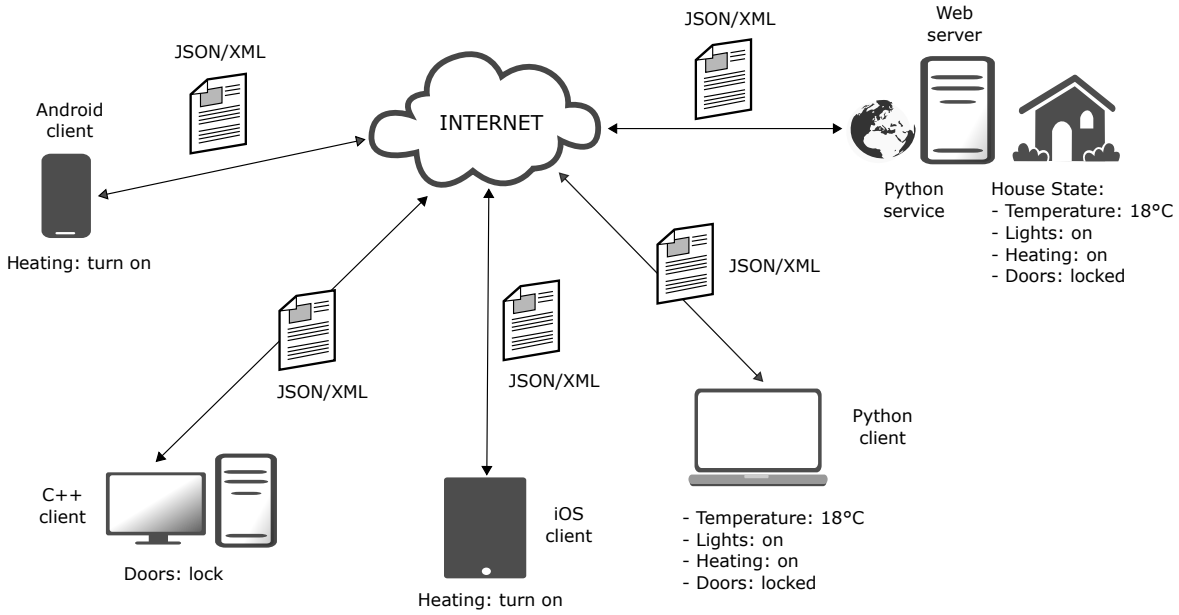

Figure 13.1: Diagram of a Web Service. Users interact with the house through clients software. In this example, clients request information and the Web Server provide a response in a JSON/XML format.

13.1 HTTP

A big part of the architecture in web services relies on the use of the HTTP or *Hypertext Transfer Protocol*. It is in charge of providing a layer to do transactions and allow communication between clients and servers. This protocol allows a higher level of communication when compared to TCP and UDP (see chapter 12). HTTP works as a *request-response* protocol in which the client performs a request and the server replies with the required information. It is a *state-less* protocol, which means that all commands are executed independently from the previous and future requests. The operation of this protocol depends on the definition of methods that indicate the action to perform on a particular resource. Resources could be existing data on the server such as files or entries in a database, or a dynamically generated output, among others. Version `HTTP/1.1` defines five methods, described in Table 13.1.

HTTP also consists of a set of **status codes** whereby they deliver information to the client about the result of its request. Table 13.2 shows an example of the HTTP message. We defer the read to the following link `http://www.w3schools.com/tags/ref\_httpmessages.asp` for more details about these codes.

Table 13.1: HTTP actions

HTTP method	Action
GET	Recovers a representation (information and meta-information) of a resource without changing anything in the server.
HEAD	Recovers only the meta-information (header) of a resource.
POST	Creates a resource.
PUT	Completely replaces a resource.
PATCH	Replaces selected attributes of a resource.
DELETE	Deletes a resource.

Table 13.2: Some common HTTP status codes

Status code	Description
200	OK. Successful request.
403	Forbidden. The request is accepted, but the server rejects it.
404	Not found. The requested resource was not found.
500	Internal server error.

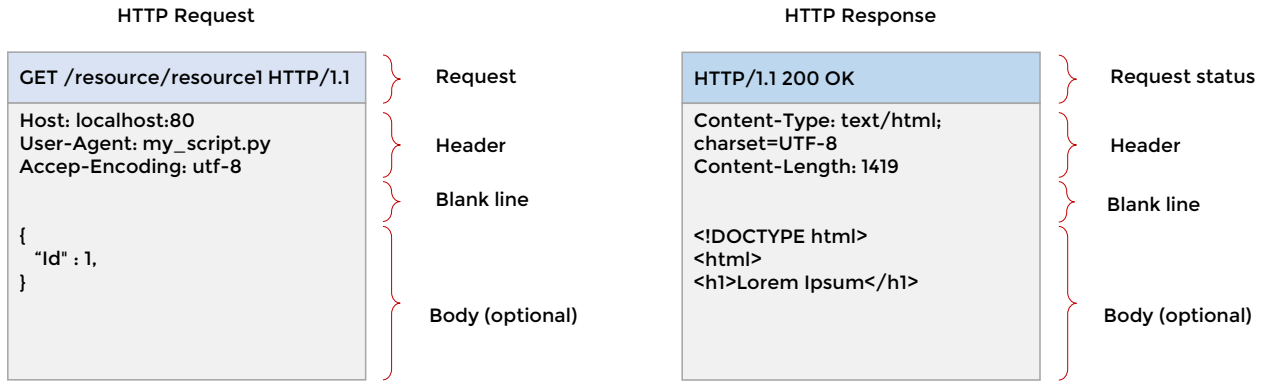

Figure 13.2: A simplified structure of the HTTP message. The blank line is intentionally added by the protocol.

13.2 REST architecture

One of the most used architectures for web service interaction is known as *Representational State Transfer* or simply **REST**. This structure uses standard HTTP methods to perform operations on the server. The calls to the server using REST reply in the format shown in the Figure 13.3, in which:

- The HTTP method corresponds to the action defined in the previous table.
- URI (Uniform Resource Identifier) is the identifier of a resource in a server.
- HTTP Version indicates the version of the protocol used by a request (`HTTP v1.1`).

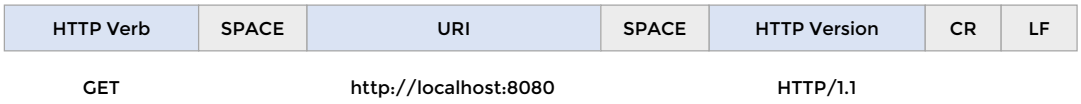

Figure 13.3: Representation of a request. Blank spaces exists between each part of the request. In the version `HTTP/1.1` the request requires the `CR` and `LF` characters at the end.

REST is straightforward and lightweight. The client and server implemented with REST and HTTP can take advantage of the entire internet infrastructure. In the Figure 13.4, we can see a diagram of the REST architecture.

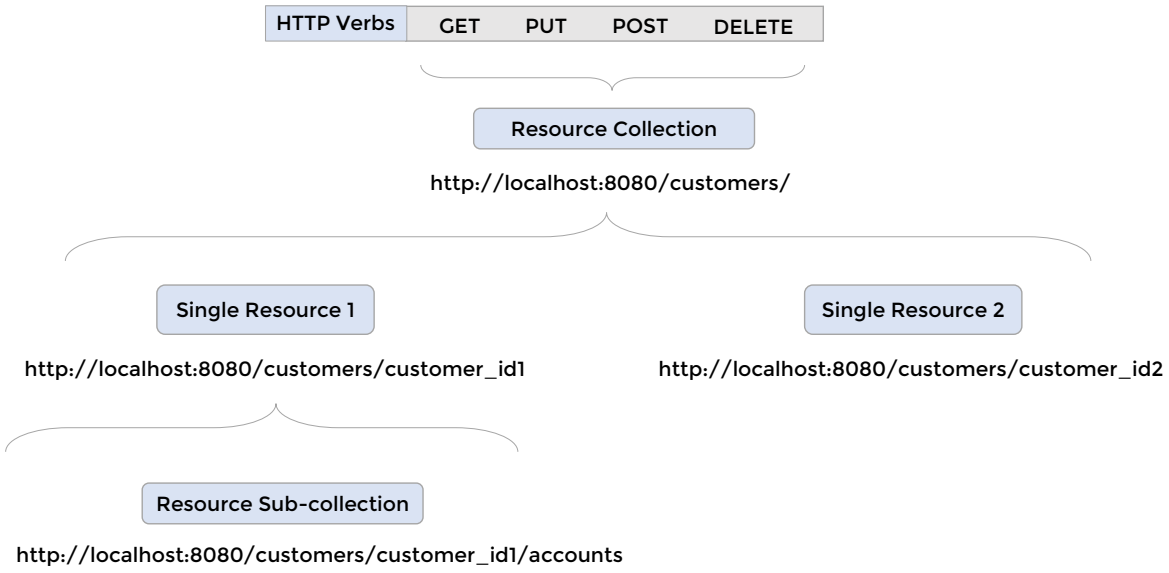

Figure 13.4: REST architecture schema. Links represent the resources. The hierarchy can be observed in the figure. There is no saved information about clients on the server. The request must contain all the necessary information to get the required resource.

13.3 Client-side Script

In this section, we see from the client viewpoint about how to perform requests to a server that hosts a web service. In Python, the `requests` library allows us to interact with several available web services. The library has the necessary HTTP methods for the REST structure. It also integrates JSON serializing methods as well.

To generate a request by means of GET, we use the `get(url)` method. For example, in the following script we generate a client that connects to the Google Image's API and recovers a query:

```
# request_google_images.py

import requests

# This URL contains the web service address
# and the required parameters

# %20 represents a 'space'

url = ('https://ajax.googleapis.com' +
       '/ajax/services/search/images?' +
       'v=1.0&q=copa%20america%20chile&as_filetype=jpg')

response = requests.get(url)
print(response.json())
```

```
{'responseData': None, 'responseDetails': 'This API is no longer available.',
'responseStatus': 403}
```

13.4 Server-side Script

To generate a web service we need to work on a *Web Framework* which lets us build dynamic websites and thus, manage the events by which the applications will interact. One of the most popular and significant framework choices for Python is *Django* (`https://www.djangoproject.com/`). However, for small applications, a micro-framework is more than enough. Here is a list of many smaller Python frameworks for web programming:

- Flask: `http://flask.pocoo.org/`
- Tornado: `http://www.tornadoweb.org/en/stable/`
- WebPy: `http://webpy.org/`
- CherryPy: `http://www.cherrypy.org/`
- Bottle: `http://bottlepy.org/docs/dev/index.html`

To exemplify the creation of a web service without loss of generality, we choose the **Flask** framework. This micro-framework is lightweight, easy to use and install, completely written in Python. Its form of encoding let us implement quickly REST-type web services.

The following example runs a server in the port `8080`. By default, the server runs locally in the IP address `127.0.0.1`, also referred as *localhost*:

```
# flask_server.py

import flask

app = flask.Flask(__name__)

# Originally, webservers used to have index.html as their main page.
#
# For example, http://mysite.com/index.html.
# By default, when no .html resource is requested
# on the URL root, it is assumed that will return
# the index.html.
#
# Nowadays this structure this folder structure
# is preserved but it is created dynamically catching the
# route and rendering anything we want.


# This route '/' determines the website root.
# Similar to the index.html document.
@app.route('/')
def index():
    return '<h1>Welcome to ur Web Service!</h1>' \
           '<p>HTML content</p>'


# Add a resource section
@app.route('/resources')
def resources_get():
```

```
30      return "<h1>Resources index</h1>"
31
32
33  # Resource section with arguments
34  @app.route('/resources/<resource_id>')
35  def resource_id_get(resource_id):
36      return '<p>Looking for resource with id: {}</p>'.format(resource_id)
37
38
39  if __name__ == '__main__':
40      # Start service as port 8080
41      app.run(port=8080)
```

Thanks to *Flask* we can start a dynamic web server using just a few lines of code. Once we run our script, we can see the result of our example in the browser, by typing in the address bar:

- `http://localhost:8080/` for the *root*.
- `http://localhost:8080/resources` to access our resources.
- `http://localhost:8080/resources/1` to access a resource created specifically with `id: 1`.

We can also execute our web service client with the following code:

```
1   # flask_client.py
2
3   import requests
4
5   r = requests.get('http://localhost:8080')
6   print('/: {}'.format(r.text))
7
8   r = requests.get('http://localhost:8080/resources')
9   print('/resources: {}'.format(r.text))
10
11  r = requests.get('http://localhost:8080/resources/1')
12  print('/resources/id: {}'.format(r.text))
```

```
/: <h1>Welcome to ur Web Service!</h1><p>HTML content</p>
/resources: <h1>Resources index</h1>
/resources/id: <p>Looking for resouce with id: 1</p>
```

13.5 Request

From the server's side, requests are calls from the client in which it is possible to receive arguments for the corresponding resource. These arguments are sent from the client through the methods: `GET`; `POST`; `PUT` and `DELETE`. *Flask* manages the sent data from the calls via the `request` class. Let's assume that we have the following service that allows sending two values to the service and define an operation using the type of resource called:

```
1  # request.py
2
3  import flask
4
5  app = flask.Flask(__name__)
6
7
8  def pow(v1, v2):
9      return v1 ** v2
10
11
12 def add(v1, v2):
13     return v1 + v2
14
15 # Server functions
16 fun_handle = {'pow': pow, 'add': add}
17
18
19 @app.route('/api/<api_id>')
20 def api_get(api_id):
21     # Request.args contains a dictionary with the
22     # arguments that were sent by the client
23     args = flask.request.args
24     if 'v1' not in args and 'v2' not in args:
25         return 'Error: No values were found'
```

```
26      else:
27          # Parse string to integer
28          v1 = int(args['v1'])
29          v2 = int(args['v2'])
30
31          return '{0}: {1}'.format(api_id, fun_handle[api_id](v1, v2))
32
33
34  if __name__ == '__main__':
35      # Start service as port 8080
36      app.run(port=8080)
```

Using our client, we can send arguments to the different functions of the service that the server is running. To transfer the parameters, we use the `params` keyword inside the `get` method.

```
1  # request_usage.py
2
3  import requests
4
5  r = requests.get('http://localhost:8080/api/pow',
6                   params={'v1': '10', 'v2': '2'})
7
8  print(r.text)
9
10 r = requests.get('http://localhost:8080/api/add',
11                  params={'v1': '10', 'v2': '2'})
12
13 print(r.text)
```

```
pow: 100
add: 12
```

13.6 Request Data

Aside from sending values as arguments in the request, it is also possible to send data to the server in JSON format, or plain text, by using the POST method. Using POST allows, aside from many things, to not leave any local information

(cache) from the sent data. It also does not let the data get exposed alongside the URI. Visually, this would be like seeing the data alongside the address in the search bar of our browser. The negotiation of the sent contents is done through the `HEADER`:

```
# request_post_server.py

import flask
import json

app = flask.Flask(__name__)


# This time only POST methods
@app.route('/upload', methods=['POST'])
def api_post():
    req = flask.request
    if req.headers['Content-Type'] == 'application/json':
        with open('post.txt', 'w') as fid:
            json.dump(req.json, fid)

        # Send echo to the client
        return "Echo: {}".format(json.dumps(req.json))


if __name__ == '__main__':
    # Start service as port 8080
    app.run(port=8080)
```

```
# request_post_client.py

import requests
import json

# Data as a form
form = {'id': 1,
        'name': 'Guido',
        'last_name': 'Van Rossum'}
```

```
10
11 # Header declares content type
12 header = {'Content-Type': 'application/json'}
13
14 # Make the request
15 r = requests.post('http://localhost:8080/upload',
16                    headers=header,
17                    data=json.dumps(form))
18
19 # Status code, 200 is 'OK'
20 print(r.status_code)
21 print(r.text)
```

```
200
Echo: {"last_name": "Van Rossum", "id": 1, "name": "Guido"}
```

13.7 Response

As we saw at the beginning of this chapter, to communicate applications, it is easier to use a format or protocol to send information back and force. Within the most used formats, JSON and XML are among the most popular. In the specific case of *Flask*, server response is managed via the `Response` class. Now, we see a serialization example of the answer of a server through JSON:

```
1  # request_response_server.py
2
3  import flask
4  import json
5
6
7  def get_id():
8      pid = 0
9      while True:
10         yield pid
11         pid += 1
12
13 pid = get_id()
```

```
14
15
16 class Person:
17     def __init__(self, name, number):
18         self.name = name
19         self.number = number
20         self.id = next(pid)
21
22
23 app = flask.Flask(__name__)
24
25
26 # Response to GET method at /api route
27 @app.route('/api', methods=['GET'])
28 def api_echo():
29     person = Person('Jason Kruger', '20.000.000-0')
30     return flask.Response(json.dumps(person.__dict__), status=200)
31
32
33 if __name__ == '__main__':
34     # Start service as port 8080
35     app.run(port=8080)
```

```
1 # request_response_client.py
2
3 import requests
4
5 r = requests.get('http://localhost:8080/api')
6 print('{}'.format(r.json()))
7
8 r = requests.get('http://localhost:8080/api')
9 print('{}'.format(r.json()))
```

```
{'name': 'Jason Kruger', 'number': '20.000.000-0', 'id': 2}
{'name': 'Jason Kruger', 'number': '20.000.000-0', 'id': 3}
```

13.8 Other architectures for Web Services

Other architectures to implement web services are XML/RCP and SOAP. Both use HTTP and XML to transfer data.

The **XML-RPC** protocol (*XML for Remote Procedure Call*) use HTTP to send requests to the server. Just like in REST, the client is an application that makes a call to a server giving arguments, which then returns a value. The use of XML allows the serialization of structures or objects as input and output of the resources. Unlike REST, which uses representations of resources, XML-RCP performs calls to methods on the server.

The **SOAP** protocol (*Simple Object Access Protocol*) uses HTTP or SMTP (Simple Mail Transfer Protocol) as transport and only XML to define the messages. SOAP allows the use of communication between different platforms, achieving the specification for *Web Services Description Language (WSDL)* and *Universal Description, Discovery, and Integration (UDDI)*. Just like XML-RCP, in the message, it defines the methods that must be executed on the server. Figure 13.5 shows the interaction between the client and server, following the SOAP architecture.

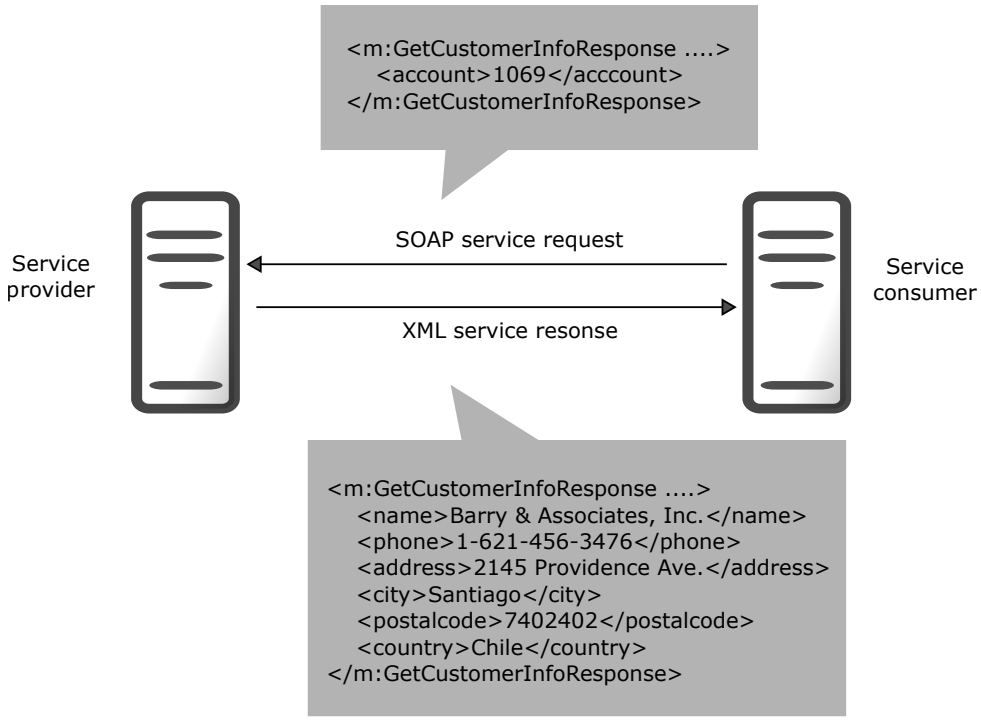

Figure 13.5: SOAP interaction

The following is an example of a SOAP message (Source: `https://en.wikipedia.org/wiki/SOAP`):

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
```

```
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice xmlns:m="http://www.example.org/stock">
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```