Data Science Fundamentals - Flask

MSc. Juan Antonio Castro Silva

April 2020

Version: 0.1 (20200424_1743)

1 Introduction

Flask is a web framework. This means flask provides you with tools, libraries and technologies that allow you to build a web application. This web application can be some web pages, a blog, a wiki or go as big as a web-based calendar application or a commercial website.

Flask is part of the categories of the micro-framework. Micro-framework are normally framework with little to no dependencies to external libraries. This has pros and cons. Pros would be that the framework is light, there are little dependency to update and watch for security bugs, cons is that some time you will have to do more work by yourself or increase yourself the list of dependencies by adding plugins. In the case of Flask, its dependencies are:

- Werkzeug a WSGI utility library

- jinja2 which is its template engine

Note WSGI is basically a protocol defined so that Python application can communicate with a web-server and thus be used as web-application outside of CGI.

2 A Minimal Application

A minimal Flask application looks something like this:

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

@app.route('/')
def hello_world():
    return 'Hello, World!'

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

Listing 1: Flask hello world application.

So what did that code do?

- First we imported the Flask class. An instance of this class will be our WSGI application.

1

- Next we create an instance of this class. The first argument is the name of the application's module or package. If you are using a single module (as in this example), you should use __name__ because depending on if it's started as application or imported as module the name will be different ('__main__' versus the actual import name). This is needed so that Flask knows where to look for templates, static files, and so on. For more information have a look at the Flask documentation.

- We then use the route() decorator to tell Flask what URL should trigger our function.

- The function is given a name which is also used to generate URLs for that particular function, and returns the message we want to display in the user's browser.

Just save it as flask_01.py or something similar. Make sure to not call your application flask.py because this would conflict with Flask itself.
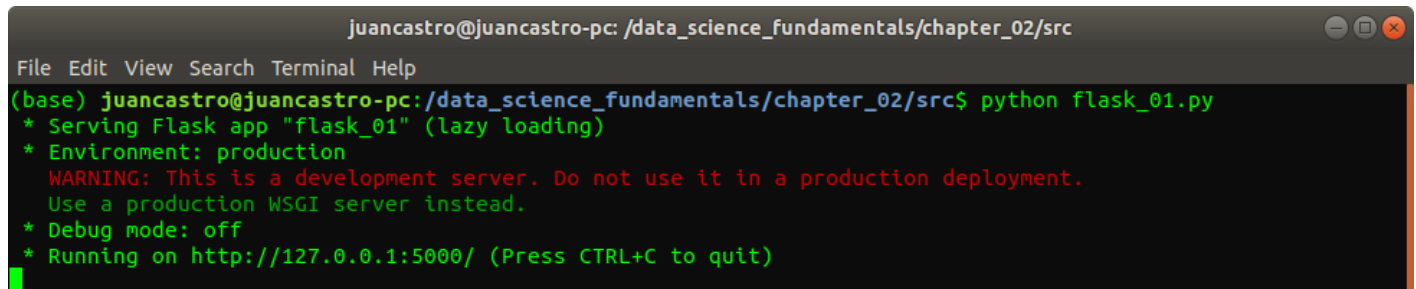
2.1   Run the application

To run the application execute the next command:

```
python flask_01.py
```

Listing 2: Run a flask server script.

The figure 1 shows the execution result in a shell terminal. The last line shows the url where the application in running (`http://localhost:5000`).

Figure 1: Variable type uuid

2.2   Test

To test the web application, open in a web browser the url `http://localhost:5000`.
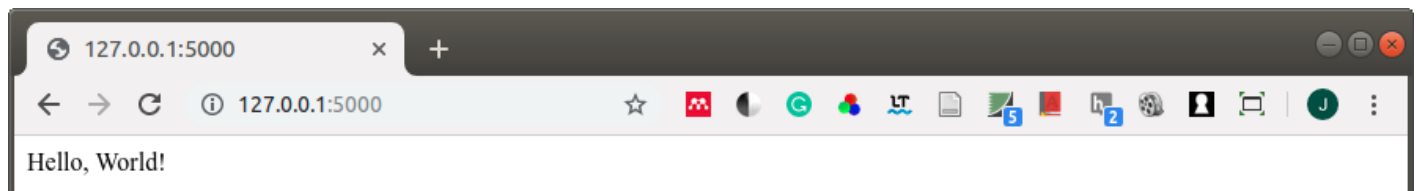Figure 2 shows how the web application running in a browser.

Figure 2: Flask Hello World

3   Routing

Modern web applications use meaningful URLs to help users. Users are more likely to like a page and come back if the page uses a meaningful URL they can remember and use to visit a page directly.

Use the route() decorator to bind a function to a URL.

```python
@app.route('/')
def index():
    return 'Index Page'

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

Listing 3: Run a flask server script.

You can do more! You can make parts of the URL dynamic and attach multiple rules to a function.

4   Variable Rules

You can add variable sections to a URL by marking sections with <variable_name>. Your function then receives the <variable_name> as a keyword argument. Optionally, you can use a converter to specify the type of the argument like <converter:variable_name>.

Converter types:

Type	Description
string	(default) accepts any text without a slash
int	accepts positive integers
float	accepts positive floating point values
path	like string but also accepts slashes
uuid	accepts UUID strings

A universally unique identifier (UUID) is a 128-bit number used to identify information in computer systems. In its canonical textual representation, the 16 octets of a UUID are represented as 32 hexadecimal (base-16) digits, displayed in five groups separated by hyphens, in the form 8-4-4-4-12 for a total of 36 characters (32 hexadecimal characters and 4 hyphens). For example: 123e4567-e89b-12d3-a456-426655440000

4.1   Example

```python
from markupsafe import escape
from flask import Flask

app = Flask(__name__)

@app.route('/user/<username>')
def show_user_profile(username):
    # show the user profile for that user
    return 'User %s' % escape(username)

@app.route('/post/<int:post_id>')
def show_post(post_id):
```

```
13      # show the post with the given id, the id is an integer
14      return 'Post %d' % post_id
15
16 @app.route('/path/<path:subpath>')
17 def show_subpath(subpath):
18      # show the subpath after /path/
19      return 'Subpath %s' % escape(subpath)
20
21 @app.route('/width/<float:width>')
22 def show_width(width):
23      # show the width, the width is a float
24      return 'Width %f' % width
25
26 @app.route('/code/<uuid:id>')
27 def show_code(id):
28      # show the code, the code is a uuid
29      return 'Code %s' % id
30
31 if __name__ == '__main__':
32      app.run()
```

Listing 4: Example of variable types.

4.2   Test

To test this example, run the `chapter_02/src/flask_03.py` python script and open in a web browser the urls.

4.2.1   string

The variable type by default is string, For this example a converter could be specified <string:username>.

```
6 @app.route('/user/<username>')
7 def show_user_profile(username):
8      # show the user profile for that user
9      return 'User %s' % escape(username)
```

Listing 5: Run a flask server script.

The figure 3 shows in a browser the result when the user open the url:
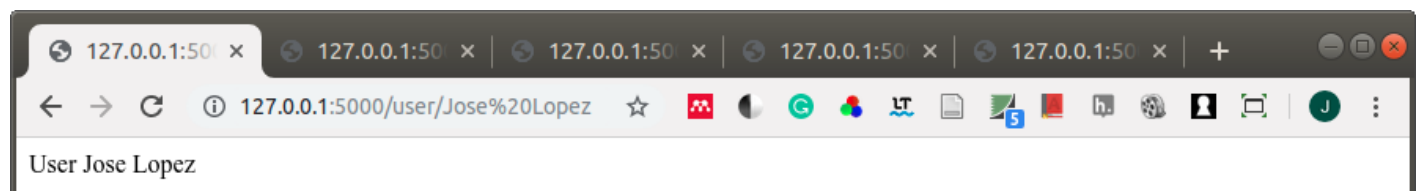`http://localhost:5000/user/Jose%20Lopez`.

Figure 3: Variable type string

4.2.2   int

The variable type int accepts positive integers. To run the example open in a web browser the url:
`http://localhost:5000/post/1235`.

```
11 @app.route('/post/<int:post_id>')
12 def show_post(post_id):
13     # show the post with the given id, the id is an integer
14     return 'Post %d' % post_id
```

<div align="center">Listing 6: Run a flask server script.</div>

Figure 4 shows the output from the python script for a variable of type int.
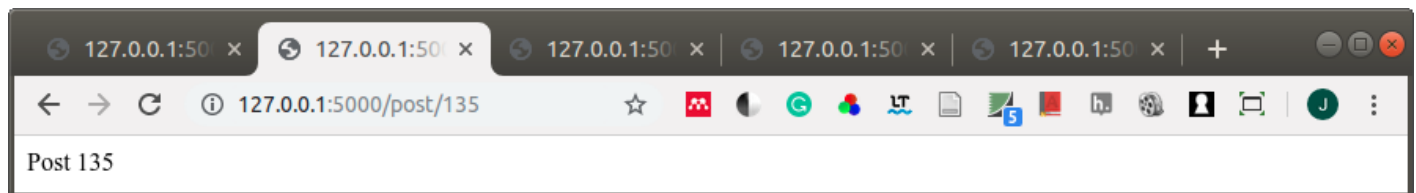

<div align="center">Figure 4: Variable type int</div>

4.2.3   path

The variable type path is like string but also accepts slashes. To run the example open in a web browser the url:

`http://localhost:5000/path/home/juan/dsf/flask.pdf`.

```
16 @app.route('/path/<path:subpath>')
17 def show_subpath(subpath):
18     # show the subpath after /path/
19     return 'Subpath %s' % escape(subpath)
```

<div align="center">Listing 7: Run a flask server script.</div>

Figure 5 shows the output from the python script for a variable of type path.
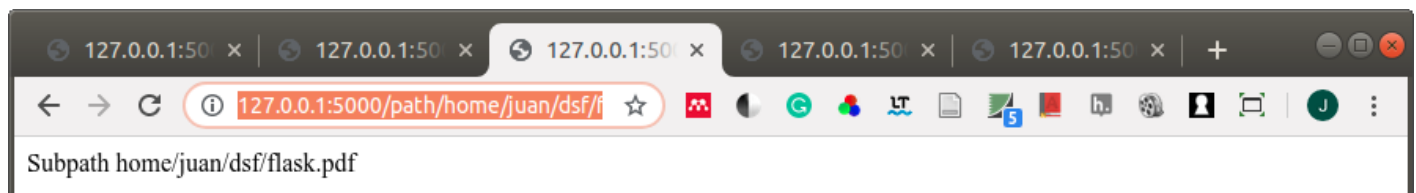

<div align="center">Figure 5: Variable type path</div>

4.2.4   float

The variable type float accepts positive floating point values. To run the example open in a web browser the url:

`http://localhost:5000/path/width/3.141592`.

```
21 @app.route('/width/<float:width>')
22 def show_width(width):
23     # show the width, the width is a float
24     return 'Width %f' % width
```

<div align="center">Listing 8: Run a flask server script.</div>

Figure 6 shows the output from the python script for a variable of type float.

Figure 6: Variable type float

4.2.5   uuid

The variable type uuid accepts UUID strings. To run the example open in a web browser the url:
`http://localhost:5000/code/123e4567-e89b-12d3-a456-426655440000`.

```
26 @app.route('/code/<uuid:id>')
27 def show_code(id):
28     # show the code, the code is a uuid
29     return 'Code %s' % id
```

Listing 9: Run a flask server script.

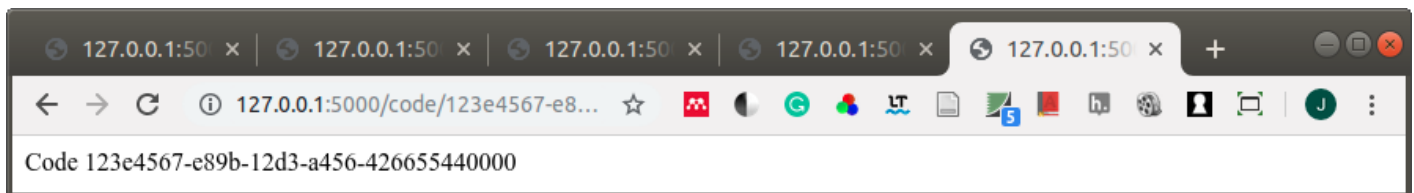Figure 7 shows the output from the python script for a variable of type uuid.

Figure 7: Variable type uuid

5 Output formats

Flask can generate different kinds of output formats like text, html, json, xml and csv among others.

5.1 Example

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

@app.route('/')
def hello_world():
    return 'Hello, World!'

@app.route('/html')
def hello_world_html():
    return '<p>Hello, World!</p>'

@app.route('/json')
def hello_world_json():
    return {"message":"Hello, World!"}

@app.route('/json/list')
def hello_world_json_list():
    data = []
    data.append({"name":"Pedro", "age":24})
    data.append({"name":"Maria", "age":18})
    data.append({"name":"Jose", "age":31})
    return {"data":data}

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

Listing 10: Example of output formats.

5.2 Test

5.2.1 Text

From lines 4 to 6 a function that return text plain is defined. To run this example open the url: `http://localhost:5000/`.

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

Listing 11: Text output format.

5.2.2 HTML

From lines 8 to 10 a function that return HTML text is defined. To run this example open the url: `http://localhost:5000/html`.

```python
@app.route('/html')
def hello_world_html():
    return '<p>Hello, World!</p>'
```

Listing 12: Run a flask server script.

Figure 8 shows the attribute Content-Type (text/html) included in the Response Header section. This information can be viewed opening the option Developer tools from the More tools menu in the chrome browser, other browser had similar tools for web developers.
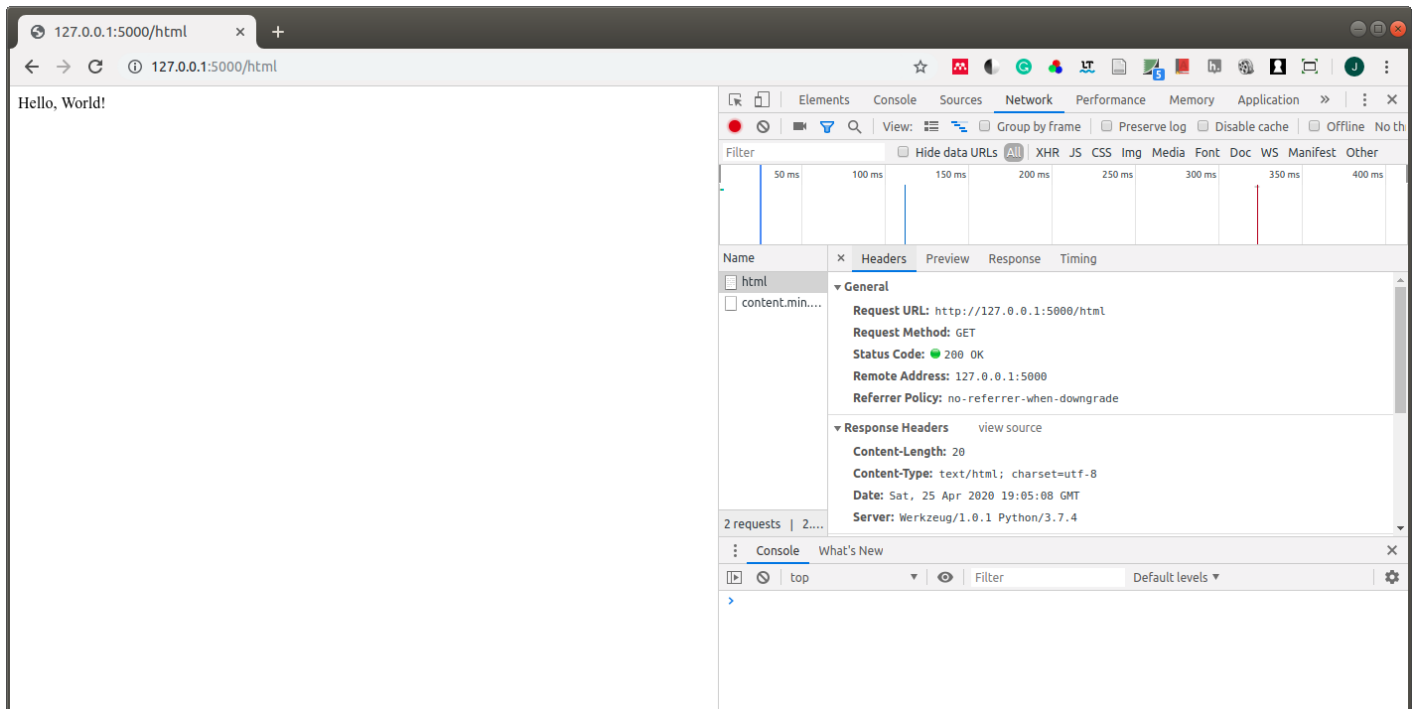

Figure 8: HTML header
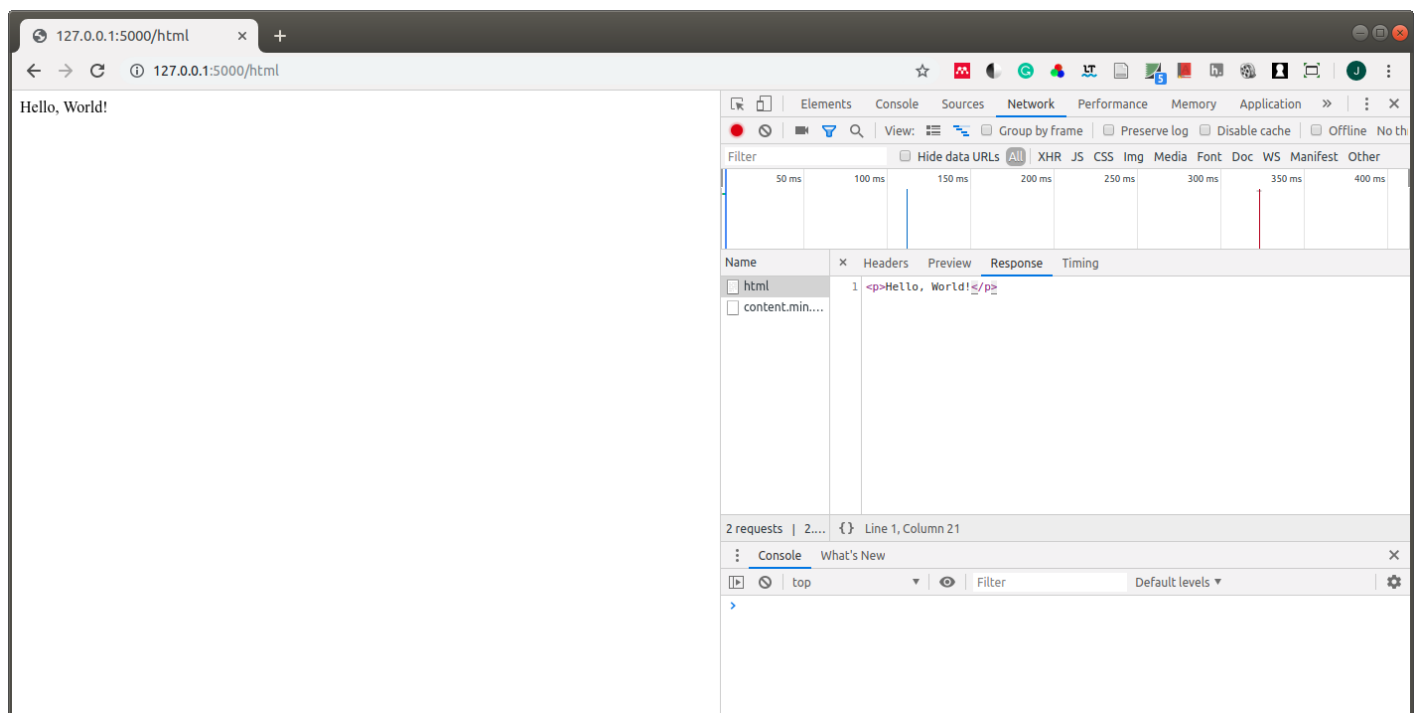
Figure 9 shows the generate html text.

Figure 9: HTML text

5.2.3   JSON

From lines 12 to 14 a function that return JSON text is defined. To run this example open the url:
`http://localhost:5000/json`.

```
12 @app.route('/json')
13 def hello_world_json():
14     return {"message":"Hello, World!"}
```

Listing 13: Run a flask server script.

Figure 10 shows chrome browser representation of the JSON text and the Content-Type = json/application in the Response Headers section.
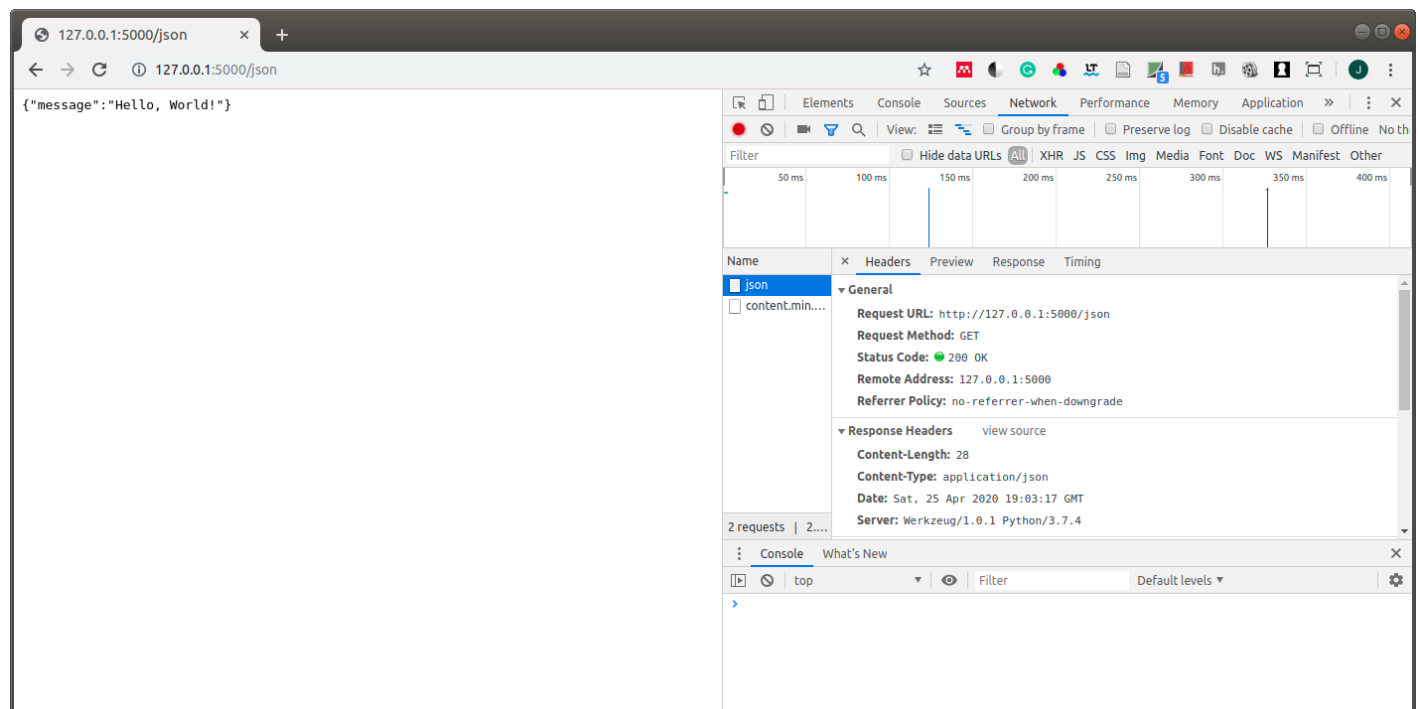

Figure 10: JSON header

From lines 16 to 22 a function that return JSON Array is defined. To run this example open the url:
`http://localhost:5000/json/list`.

```
16 @app.route('/json/list')
17 def hello_world_json_list():
18     data = []
19     data.append({"name":"Pedro", "age":24})
20     data.append({"name":"Maria", "age":18})
21     data.append({"name":"Jose", "age":31})
22     return {"data":data}
```

Listing 14: Run a flask server script.

Figure 11 shows chrome browser representation of the JSON Array and the Content-Type = json/application in the Response Headers section.

En la figura **??** se muestra como se visualiza la implentación de los enlaces (<a>) en un navegador.

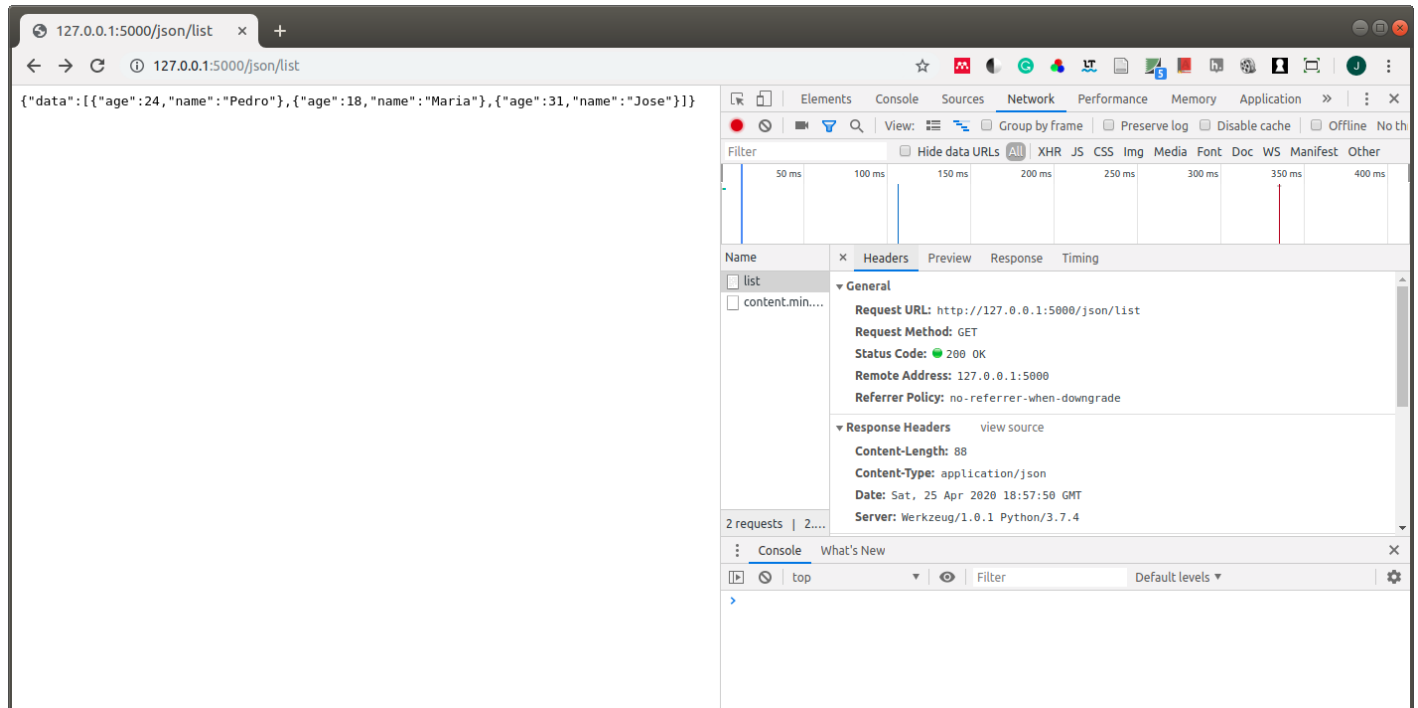Figure 12 shows the preview of the JSON Array in the chrome web browser.
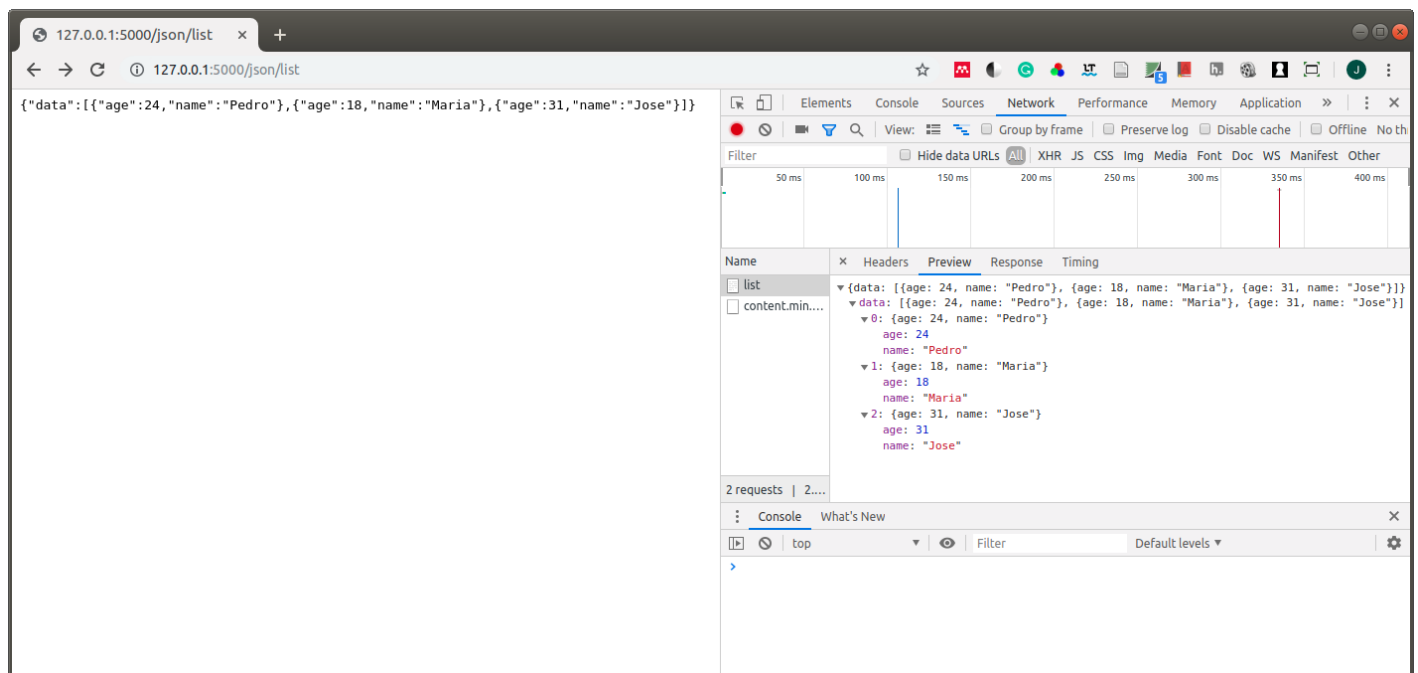
Figure 11: Variable type uuid

Figure 12: Variable type uuid

6   jsonify

The flask.json.jsonify(*args, **kwargs) turns the JSON output into a Response object with the application/json mimetype. For convenience, it also converts multiple arguments into an array or multiple keyword arguments into a dict. This means that both jsonify(1,2,3) and jsonify([1,2,3]) serialize to [1,2,3].

6.1   Example

```python
 1 from flask import Flask
 2 from flask import jsonify
 3
 4 app = Flask(__name__)
 5
 6 @app.route('/')
 7 def get_current_user():
 8     return jsonify(username="Maria",
 9                    email="maria@gmail.com",
10                    id="1298")
11
12 @app.route('/json_from_tuple')
13 def show_tuple():
14     return jsonify(1,2,3)
15
16 @app.route('/json_from_list')
17 def show_list():
18     return jsonify([1,2,3])
19
20 if __name__ == '__main__':
21     app.run()
```

Listing 15: Example usage of jsonify function.

6.2   Test

6.2.1   Jsonify example

From lines 6 to 10 a function that returns JSON text is defined. To run this example open the url:
`http.//localhost:5000/`

```python
 6 @app.route('/')
 7 def  get_current_user ():
 8     return  jsonify(username="Maria",
 9                     email="maria@gmail.com",
10                     id="1298")
```

Figure 13 shows the JSON text generated by the jsonify function().

6.2.2   Jsonify from tuple

From lines 12 to 14 a function that returns JSON text from a tuple is defined. To run this example open the url:
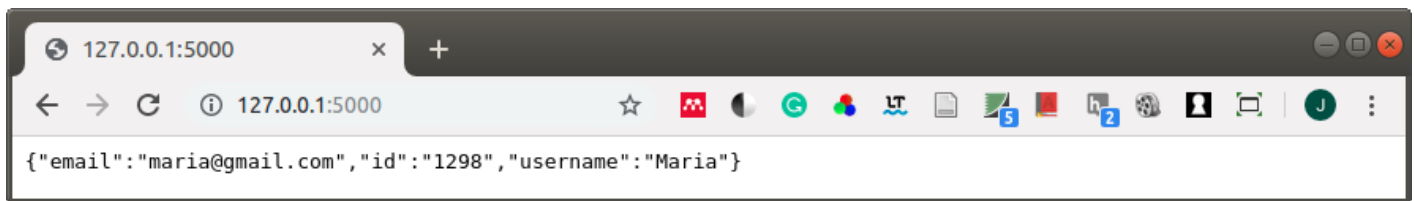`http.//localhost:5000/json_from_tuple`

Figure 13: Output of jsonify

```
12 @app.route('/json_from_tuple')
13 def  show_tuple ():
14     return  jsonify(1,2,3)
```

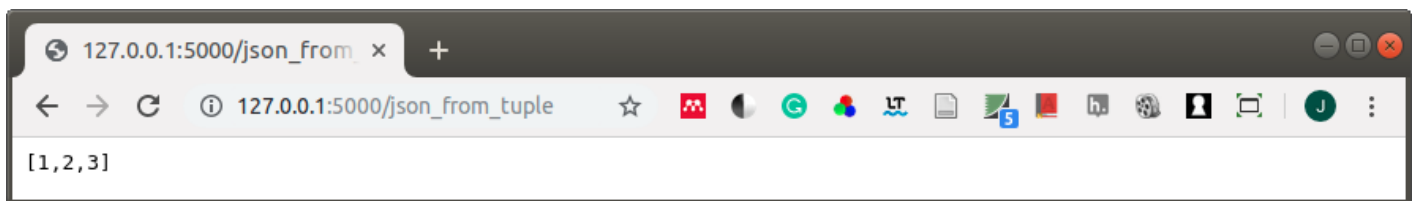Figure 14 shows the JSON text generated by the jsonify function().

Figure 14: Jsonify from tuple

6.2.3   Jsonify from list

From lines 16 to 18 a function that returns JSON text from a list is defined. To run this example open the url:
http.//localhost:5000/json_from_list

```
16 @app.route('/json_from_list')
17 def  show_tuple ():
18     return  jsonify([1,2,3])
```

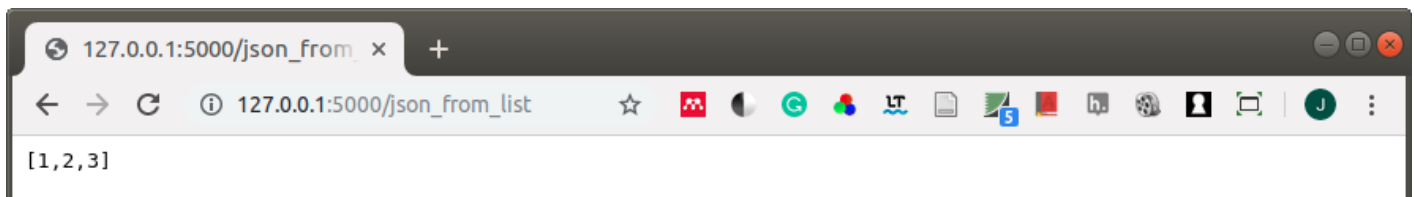Figure 15 shows the JSON text generated by the jsonify function().

Figure 15: Jsonify from list

7   Ajax + JSON

The HTTP methods (GET, POST) are covered in this section using the JSON format to exchange information in an AJAX web application written with HTML and JavaScript (Client), and Python (Server).

   The file `chapter_02/src/flask_06.py` contains the python implementation of a server script to receive and send JSON data.

```python
 1 from flask import Flask, request, jsonify
 2 from flask_cors import CORS
 3
 4 app = Flask(__name__)
 5 CORS(app)
 6
 7 users = []
 8
 9 @app.route("/", methods=["POST"])
10 def starting_url():
11     json_data = request.json
12     user = json_data["user"]
13     email = json_data["email"]
14     data = {"usuario":user,"correo":email}
15     users.append(data)
16     return {"data":data}
17
18 @app.route('/list', methods=["GET"])
19 def show_form_data():
20     return jsonify(users)
21
22 if __name__ == '__main__':
23     #app.run(host="0.0.0.0", port=8080)
24     app.run()
```

Listing 16: JSON application (GET and POST).

7.1   Sending JSON data with POST

The file `chapter_02/src/json_01.html` contains the HTML and JavaScript code to send JSON data using the POST HTTP method with AJAX.

Listing 17: Sending JSON data with POST.

```html
 1 <!DOCTYPE html>
 2 <html>
 3 <body>
 4
 5 <h2>The XMLHttpRequest Object</h2>
 6
 7 <p>Name: <input type="text" id="user" name="user"/></p>
 8 <p>Name: <input type="email" id="email" name="email"/></p>
 9 <button type="button" onclick="sendData()">Add User</button>
10
11 <p>Server response:</p>
12 <p id="response"></p>
13
14 <script>
15 function sendData() {
```

```
16    var user = document.getElementById("user").value;
17    var email = document.getElementById("email").value;
18    var data = JSON.stringify({"user": user,"email":email})
19    console.log(data);
20    var xhttp = new XMLHttpRequest();
21    xhttp.onreadystatechange=function() {
22      if (this.readyState == 4 && this.status == 200) {
23        document.getElementById("response").innerHTML = this.responseText;
24      }
25    };
26    var theUrl = "http://localhost:5000/";
27    xhttp.open("POST", theUrl);
28    xhttp.setRequestHeader("Content-Type", "application/json;charset=UTF-8");
29    xhttp.send(data);
30 }
31 </script>
32
33 </body>
34 </html>
```

7.2   Receiving JSON data with GET

The file chapter_02/src/json_02.html contains the HTML and JavaScript code to receive JSON data using the GET HTTP method with AJAX.

Listing 18: Receiving JSON data with GET.

```
 1 <!DOCTYPE html>
 2 <html>
 3 <body>
 4
 5 <h2>The XMLHttpRequest Object</h2>
 6
 7 <table id="users">
 8 <thead><th>User</th><th>Email</th></thead>
 9 <tbody id="body"></tbody>
10 </table>
11
12 <button type="button" onclick="getData()">Get Users</button>
13
14 <script>
15 function getData() {
16
17    var xhttp = new XMLHttpRequest();
18    xhttp.onreadystatechange=function() {
19      if (this.readyState == 4 && this.status == 200) {
20        var users = JSON.parse(this.responseText);
21        console.log(users);
22        var rows = "";
23        for (var i = 0; i < users.length; i++){
24          var user = users[i].usuario;
25          var email = users[i].correo;
26          var row = "<tr><td>" + user + "</td><td>" + email + "</td></tr>";
27          //alert(row);
28          rows += row;
29        }
30        document.getElementById("body").innerHTML = rows;
```

```
31        }
32      };
33      var theUrl = "http://localhost:5000/list";
34      xhttp.open("GET", theUrl, true);
35      xhttp.send();
36  }
37  </script>
38
39  </body>
40  </html>
```

7.3   Test

7.3.1   JSON AJAX POST

From lines 9 to 16, a function that receives JSON data in the server via the POST HTTP method is defined. Line 11 gets the data in JSON format. Lines 12 and 13 stores the user and email in a pair of variables. The user and email data are used to create a dictionary in Line 14. Line 15 appends the dictionary to the users list. The data JSON object is returned at line 16.

```
 9  @app.route("/", methods=["POST"])
10  def starting_url():
11      json_data = request.json
12      user = json_data["user"]
13      email = json_data["email"]
14      data = {"usuario":user,"correo":email}
15      users.append(data)
16      return {"data":data}
```

Lines 7 and 8 define two inputs to manage the user and email variables. The button to send data (Add User) is defined at line 9. When the button is clicked, the sendData() function is called. A container paragraph to store the server response is created at line 12.

```
 7  <p>Name: <input type="text" id="user" name="user"/></p>
 8  <p>Name: <input type="email" id="email" name="email"/></p>
 9  <button type="button" onclick="sendData()">Add User</button>
10
11  <p>Server response:</p>
12  <p id="response"></p>
```

The JavaScript sendData() function is defined at line 15.The user form registered data is stored in the user and email variables at lines 16 and 17. A JSON object is created at line 18 with the user and email values. Line 19 prints the data JSON object in the web browser console. A XMLHttpRequest object is created at line 20. The URL is defined at line 26. The ajax call to the URL is opened at line 27 with the POST method. The header attribute Content-Type="application/json;charset=UTF-8" is defined at line 28. The form JSON data is sent at line 29.

```
14  <script>
15  function sendData() {
16    var user = document.getElementById("user").value;
17    var email = document.getElementById("email").value;
18    var data = JSON.stringify({"user": user,"email":email})
19    console.log(data);
20    var xhttp = new XMLHttpRequest();
21    xhttp.onreadystatechange=function() {
22      if (this.readyState == 4 && this.status == 200) {
```

```
23        document.getElementById("response").innerHTML = this.responseText;
24      }
25    };
26    var theUrl = "http://localhost:5000/";
27    xhttp.open("POST", theUrl);
28    xhttp.setRequestHeader("Content-Type", "application/json;charset=UTF-8");
29    xhttp.send(data);
30  }
31  </script>
```

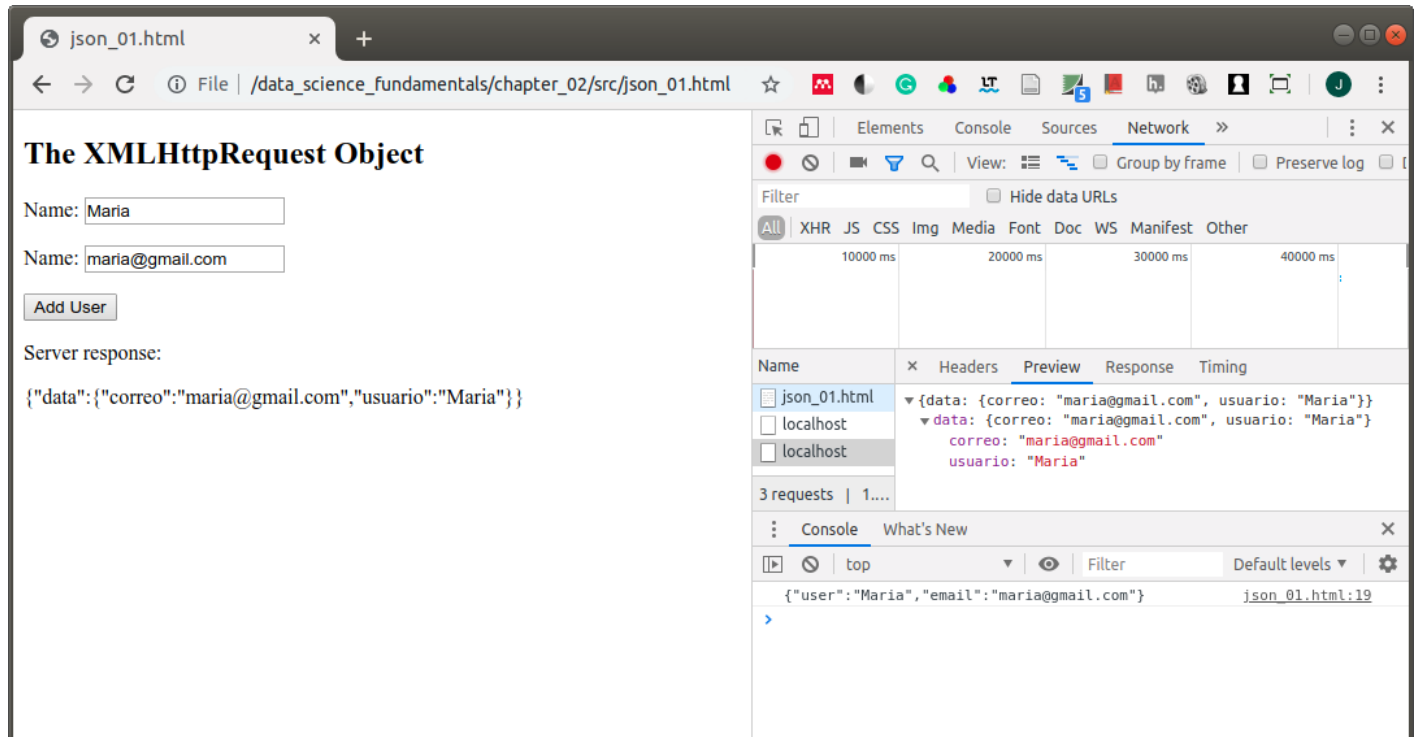Figure 16 shows the HTML form and the JSON server response in the web browser.

Figure 16: Variable type uuid

7.3.2 JSON AJAX GET

From lines 18 to 20, a server side function that send JSON data to the client via the GET HTTP method is defined. The user list is returned as a JSON array at line 20.

```
18 @app.route('/list', methods=["GET"])
19 def show_form_data():
20      return jsonify(users)
```

From lines 7 to 10, a table to store the server JSON response is created. The button to get the JSON data (Get Users) is defined at line 12. When the button is clicked, the getData() function is called.

```
 7  <table id="users">
 8  <thead><th>User</th><th>Email</th></thead>
 9  <tbody id="body"></tbody>
10  </table>
11
12  <button type="button" onclick="getData()">Get Users</button>
```

The JavaScript getData() function is defined at line 15. An instance of the XMLHttpRequest class is created at line 16. The URL is defined at line 32. The AJAX call is opened for the GET method in line 33. The AJAx request is sent at line 34.

The onreadystatechange function is defined at line 17 to receive the JSON data. The server response text is converted to JSON at line 18. A for loop to create the table rows with the user list is defined form lines 22 to 28. The HTML content of the table body is changed at line 29.

```
14  <script>
15  function getData() {
16    var xhttp = new XMLHttpRequest();
17    xhttp.onreadystatechange=function() {
18      if (this.readyState == 4 && this.status == 200) {
19        var users = JSON.parse(this.responseText);
20        console.log(users);
21        var rows = "";
22        for (var i = 0; i < users.length; i++){
23          var user = users[i].usuario;
24          var email = users[i].correo;
25          var row = "<tr><td>" + user + "</td><td>" + email + "</td></tr>";
26          //alert(row);
27          rows += row;
28        }
29        document.getElementById("body").innerHTML = rows;
30      }
31    };
32    var theUrl = "http://localhost:5000/list";
33    xhttp.open("GET", theUrl, true);
34    xhttp.send();
35  }
36  </script>
```

Figure 17 shows the JSON server response to the GET method.
`http://localhost:5000/list`

Figure 18 shows a HTML page that receive JSON data using the GET method.
`chapter_02/src/json_02.html`
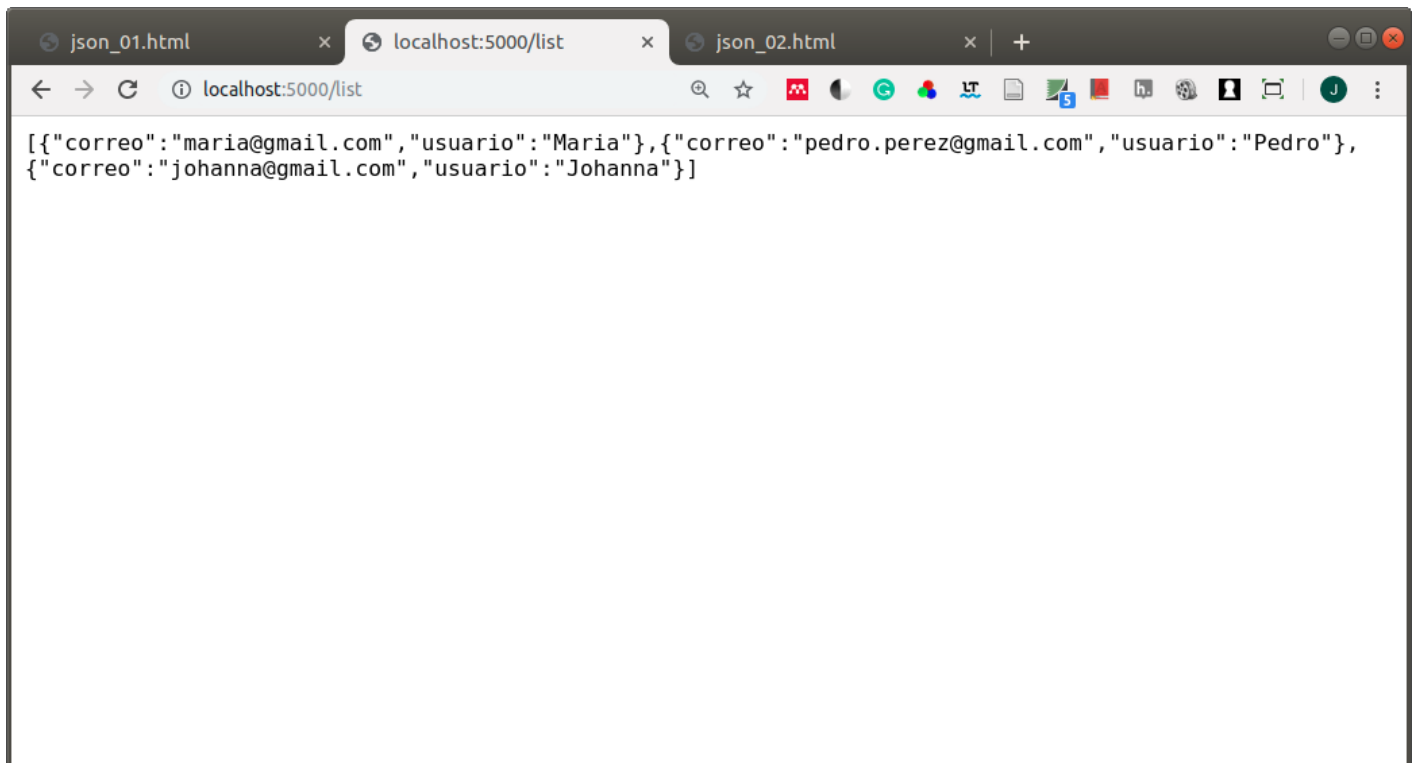
Figure 17: Receiving JSON data with GET
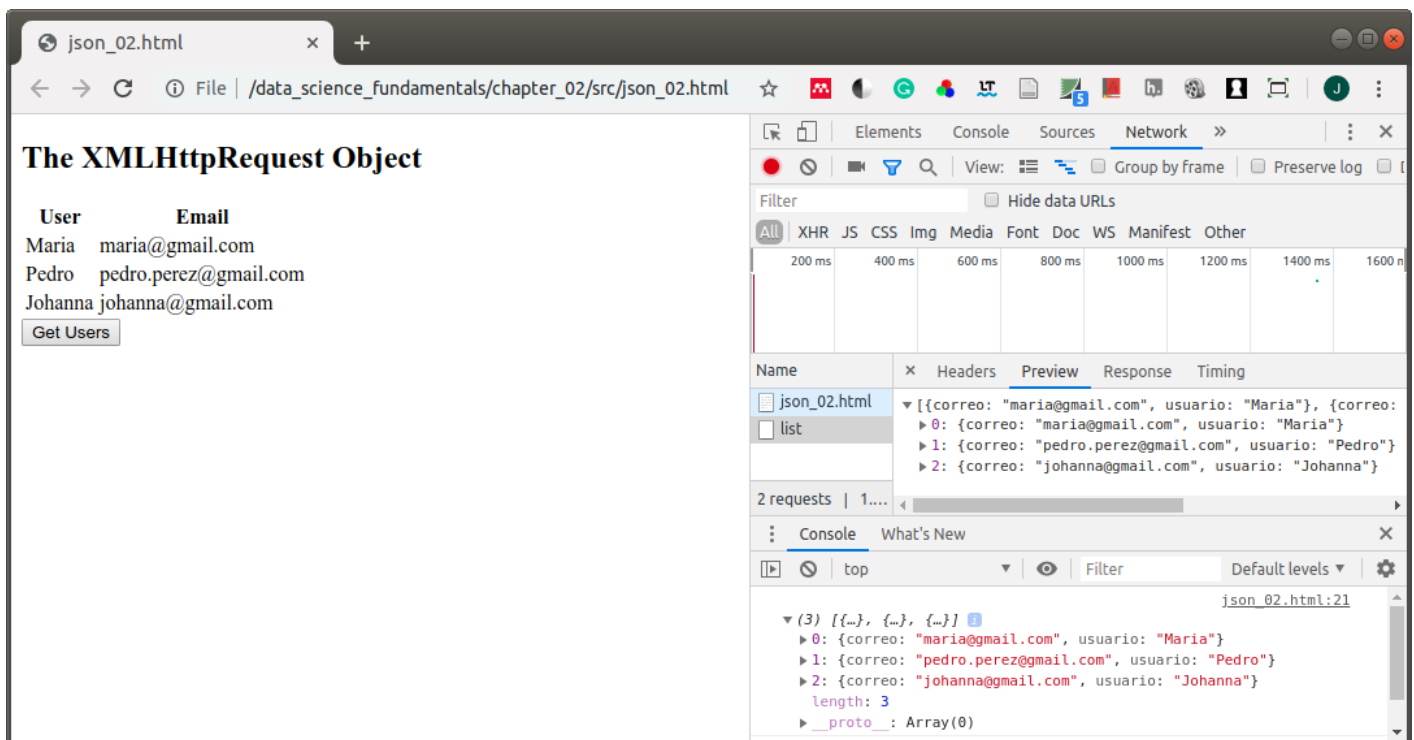
Figure 18: Receiving JSON data with GET

8   HTML Forms

The use of HTML forms with the methods GET and POST is covered in this section.

File: `chapter_02/src/flask_07.py`

```python
1 from flask import Flask, request
2 from flask import jsonify
3
4 app = Flask(__name__)
5
6 @app.route("/get", methods=["GET"])
7 def show_get_data():
8     name = request.args.get("name")
9     age = request.args.get('age')
10    return "<p>Name: {} <br/> Age :{}</p>".format(name, age)
11
12 @app.route("/post", methods=["POST"])
13 def show_post_data():
14    name = request.form.get('name')
15    age = request.form.get('age')
16    return "<p>Name: {} <br/> Age :{}</p>".format(name, age)
17
18 if __name__ == '__main__':
19    app.run()
```

Listing 19: GET and POST HTTP methods.

File: `chapter_02/src/html_forms.html`

Listing 20: Example of HTML forms.

```html
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title>HTML Forms</title>
5  </head>
6  <body>
7
8  <h2>Methods to send data</h2>
9  <fieldset>
10 <legend>Sending data (get method):</legend>
11 <form action="http://localhost:5000/get" method="get">
12 <p>Name: <br/><input type="text" name="name"/></p>
13 <p>Age: <br/><input type="number" name="age"/></p>
14 <p><input type="submit" value="Send using Get"/></p>
15 </form>
16 </fieldset>
17
18 <br/>
19
20 <fieldset>
21 <legend>Sending data (post method)</legend>
22 <form action="http://localhost:5000/post" method="post">
23 <p>Name: <br/><input type="text" name="name"/></p>
24 <p>Age: <br/><input type="number" name="age"/></p>
25 <p><input type="submit" value="Send using Post"/></p>
26 </form>
27 </fieldset>
28
```

```
29  </body>
30  </html>
```
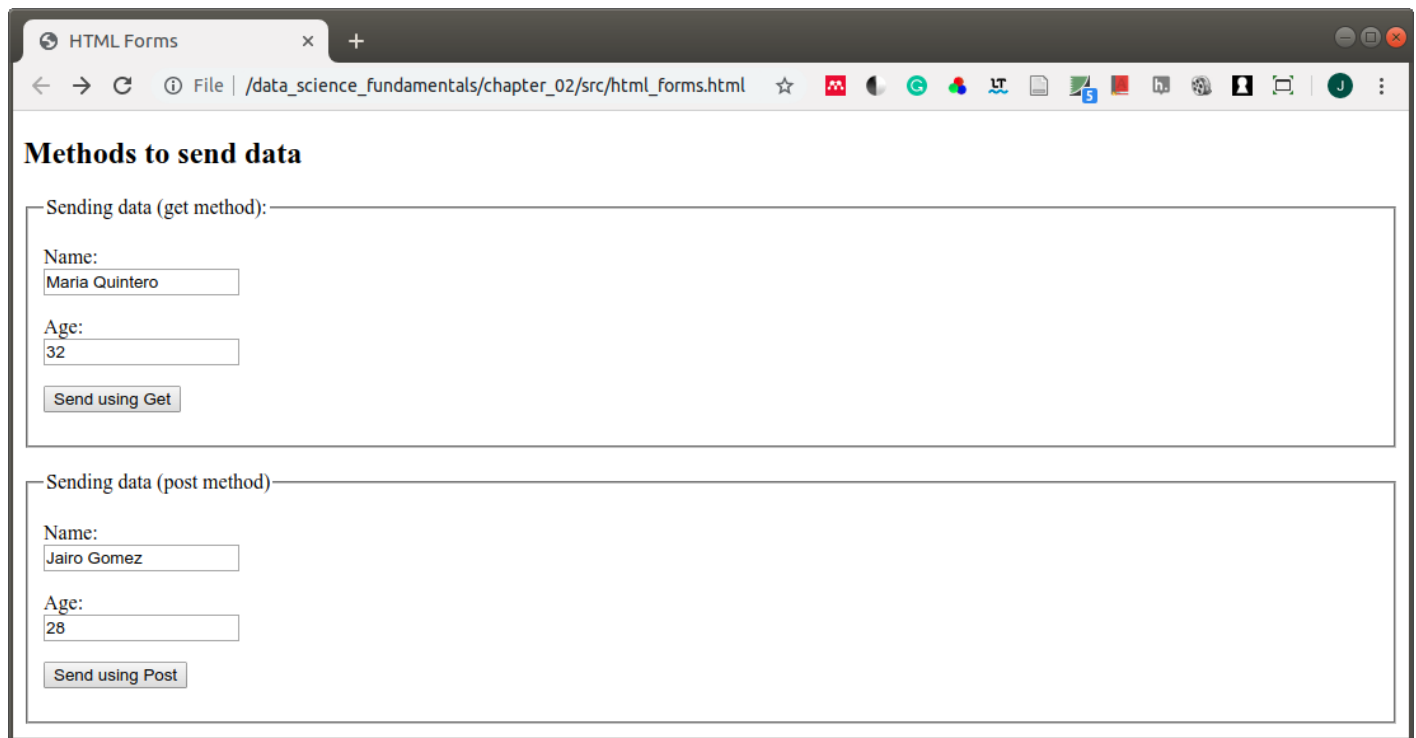
8.1 Test

The server script must be run to test this example, using the command:

```
python flask_07.py
```

The web page html_forms.html must be opened in a browser, the information for name and age variables filled, and the submit button clicked to send the data to the server.

Figure 19 shows the two HTML forms in a browser (GET and POST forms).

Figure 19: HTML forms

8.1.1 GET

From lines 6 to 10, a function that get the data send by the HTML form using the GET method is defined. It returns the response in HTML format.

```
6  @app.route("/get", methods=["GET"])
7  def show_get_data():
8      name = request.args.get("name")
9      age = request.args.get('age')
10     return "<p>Name: {} <br/> Age :{}</p>".format(name, age)
```

Line 11 defines a form with the get method. Two input elements of type text and number are defined in lines 12 and 13 to manage the name and age variables, respectively. The submit button is defined at line 14.

```
 9  <fieldset>
10  <legend>Sending data (get method):</legend>
11  <form action="http://localhost:5000/get" method="get">
12  <p>Name: <br/><input type="text" name="name"/></p>
13  <p>Age: <br/><input type="number" name="age"/></p>
14  <p><input type="submit" value="Send using Get"/></p>
15  </form>
16  </fieldset>
```

Figure 20 shows the server output for the get method in a web browser.
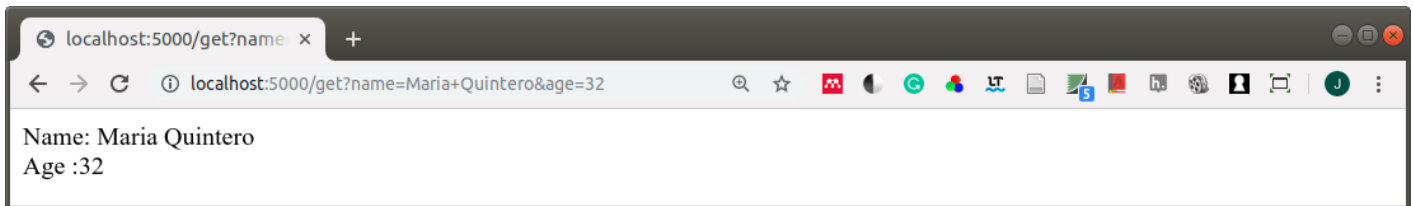
Figure 20: GET method

8.1.2   POST

From lines 12 to 16, a function that get the data send by the HTML form using the POST method is defined. It returns the response in HTML format.

```
12 @app.route("/post", methods=["POST"])
13 def show_post_data():
14     name = request.form.get('name')
15     age = request.form.get('age')
16     return "<p>Name: {} <br/> Age :{}</p>".format(name, age)
```

Line 22 defines a form with the post method. Two input elements of type text and number are defined in lines 23 and 24 to manage the name and age variables, respectively. The submit button is defined at line 25.

```
20  <fieldset>
21  <legend>Sending data (post method)</legend>
22  <form action="http://localhost:5000/post" method="post">
23  <p>Name: <br/><input type="text" name="name"/></p>
24  <p>Age: <br/><input type="number" name="age"/></p>
25  <p><input type="submit" value="Send using Post"/></p>
26  </form>
27  </fieldset>
```

Figure 21 shows the server output in a web browser.
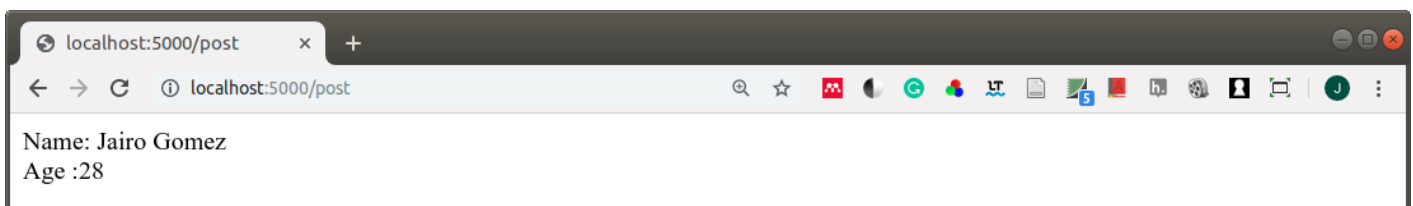
Figure 21: POST method

9   Python client to send and receive JSON data

Send and receive JSON data with a python client script using the requests library is covered in this section.

File: `chapter_02/src/requests_01.py`

```python
 1 import requests
 2 import json
 3
 4 def show_response(response):
 5     print("Status code: ", response.status_code)
 6
 7     json_response = response.json()
 8     data = json_response['data']
 9     user = data['usuario']
10     email = data['correo']
11
12     print("Printing Post JSON data")
13     print(data)
14     print("Usuario: " + user)
15     print("Correo: " + email)
16
17
18 theUrl = "http://localhost:5000"
19
20 print()
21 print("--- Method Post 1 ---")
22 data = {'user':'Sofia Perdomo',
23         'email':'sofia.perdomo@gmail.com'}
24 response = requests.post(theUrl, json=data)
25 show_response(response)
26
27 print()
28 print("--- Method Post 2 ---")
29 data = {'user':'Jorge Arevalo',
30         'email':'jorge.arevalo@gmail.com'}
31 headers = {'Content-type': 'application/json', 'Accept': 'text/plain'}
32 response = requests.post(theUrl,
33                          data=json.dumps(data),
34                          headers=headers)
35 show_response(response)
36
37 print()
38 print("--- Method Get (user list) ---")
39 theUrl = "http://localhost:5000/list"
40 response = requests.get(theUrl)
41 print("Status code: ", response.status_code)
42 json_items = response.json()
43 print(json_items)
44 for item in json_items:
45     print("user:", item['usuario'], "email:", item['correo'])
```
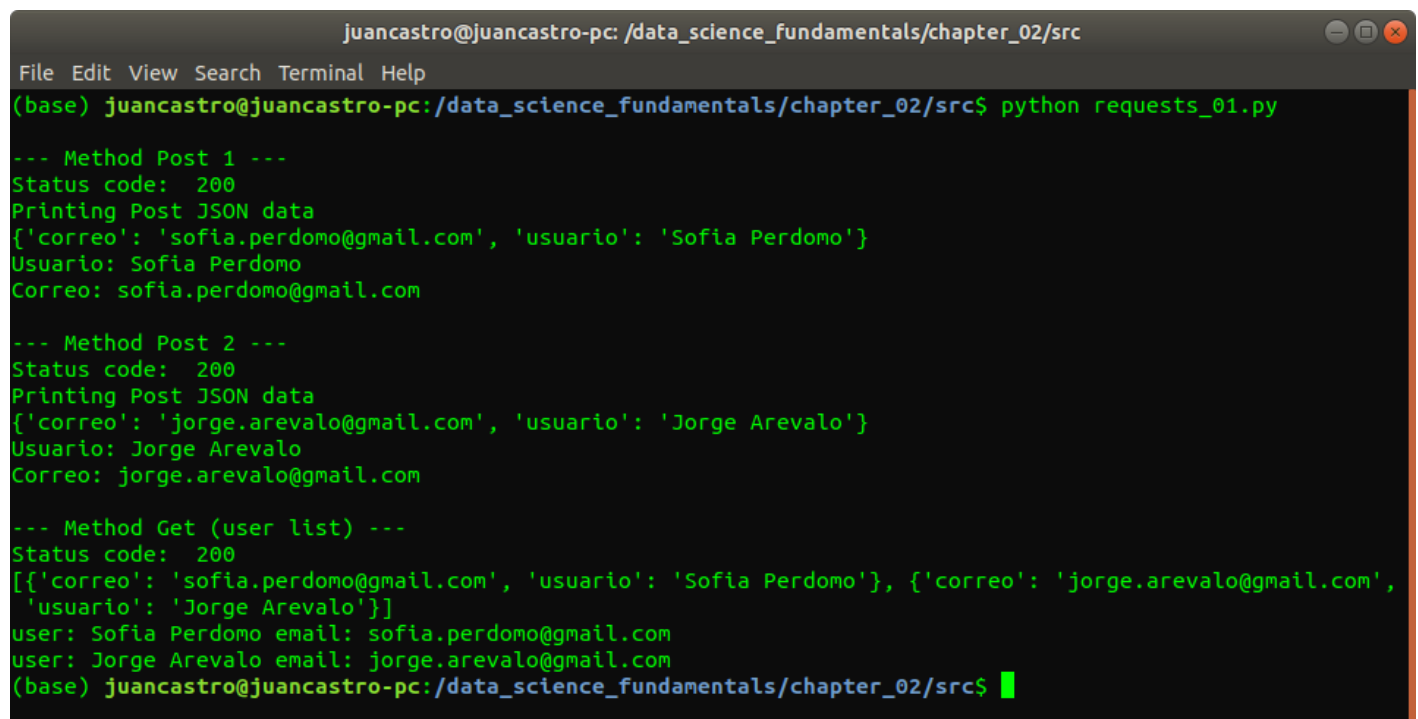
Listing 21: Example of variable types.

The URL is defined in line 18.

```python
18 theUrl = "http://localhost:5000"
```

A data dictionary is created in lines 22 and 23. The JSON data is sent, and the server JSON response is received in line 24. The function show_response() is called at line 25.

```
20 print()
21 print("--- Method Post 1 ---")
22 data = {'user':'Sofia Perdomo',
23         'email':'sofia.perdomo@gmail.com'}
24 response = requests.post(theUrl, json=data)
25 show_response(response)
```

A data dictionary is created in lines 29 and 30. The HTTP request header is created at line 31. The attribute 'Content-type':'application/json', is required. The JSON data is sent using the method post of the request library, and the server JSON response is received in line 32. The function show_response() is called at line 35.

```
27 print()
28 print("--- Method Post 2 ---")
29 data = {'user':'Jorge Arevalo',
30         'email':'jorge.arevalo@gmail.com'}
31 headers = {'Content-type': 'application/json', 'Accept': 'text/plain'}
32 response = requests.post(theUrl,
33                          data=json.dumps(data),
34                          headers=headers)
35 show_response(response)
```

From lines 4 to 15, the show_response() function is created. The server status code is printed in line 5. The server response is converted to JSON in line 7. The user and email values are obtained in lines 9 and 10. The data is printed in lines 12 to 15.

```
 4 def show_response(response):
 5     print("Status code: ", response.status_code)
 6
 7     json_response = response.json()
 8     data = json_response['data']
 9     user = data['usuario']
10     email = data['correo']
11
12     print("Printing Post JSON data")
13     print(data)
14     print("Usuario: " + user)
15     print("Correo: " + email)
```

The URL `http://localhost:5000/list` is assigned in line 39. In line 40, the GET request is called. The server status code is printed in line 41. The server response is converted to JSON in line 42. The JSON array is printed in line 43. A for loop to print the JSON array (user list) is defined in lines 44 and 45.

```
37 print()
38 print("--- Method Get (user list) ---")
39 theUrl = "http://localhost:5000/list"
40 response = requests.get(theUrl)
41 print("Status code: ", response.status_code)
42 json_items = response.json()
43 print(json_items)
44 for item in json_items:
45     print("user:", item['usuario'], "email:", item['correo'])
```

Figure 22 shows the server JSON responses to the POST and GET request from the python client.

Figure 22: Server JSON responses to POST and GET

10   JSON client applications for testing

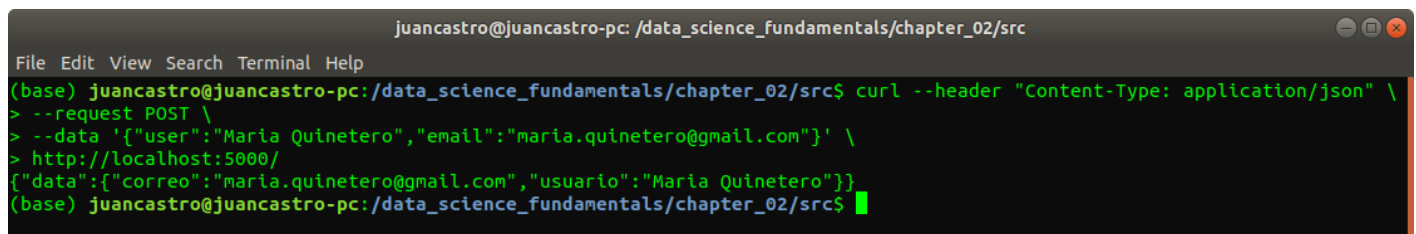## 10.1   CURL

CURL is a command line tool and library for transferring data with URLs. CURL is free and open source software. `https://curl.haxx.se/`

10.1.1   POST

```
curl --header "Content-Type: application/json" \
--request POST \
--data '{"user":"Maria Quintero","email":"maria.quintero@gmail.com"}' \
http://localhost:5000/
```
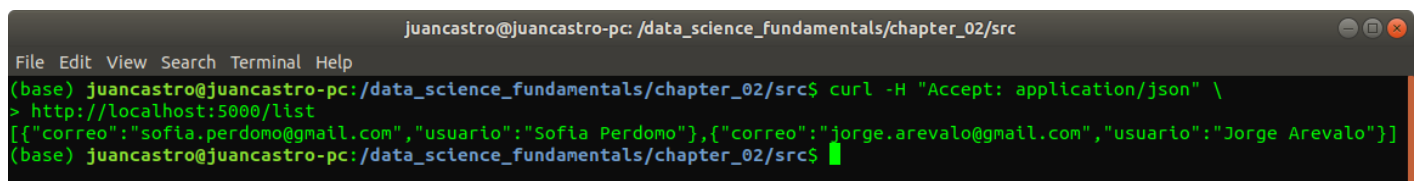
Figure 23 shows the curl POST request and the server response.

Figure 23: curl POST call

10.1.2   GET

```
curl -H "Accept: application/json" \
http://localhost:5000/list
```

Figure 24 shows the curl POST request and the server response.

Figure 24: curl GET call

10.2   POSTMAN

Postman is a collaboration platform for API development. Postman's features simplify each step of building an API and streamline collaboration so you can create better APIs—faster. `https://www.postman.com/`

10.2.1   GET

Postman is recommended to test the server response to a HTTP method call. Select the GET method, enter the URL `http://localhost:5000/list` and click the [Send] button to test the webservice.

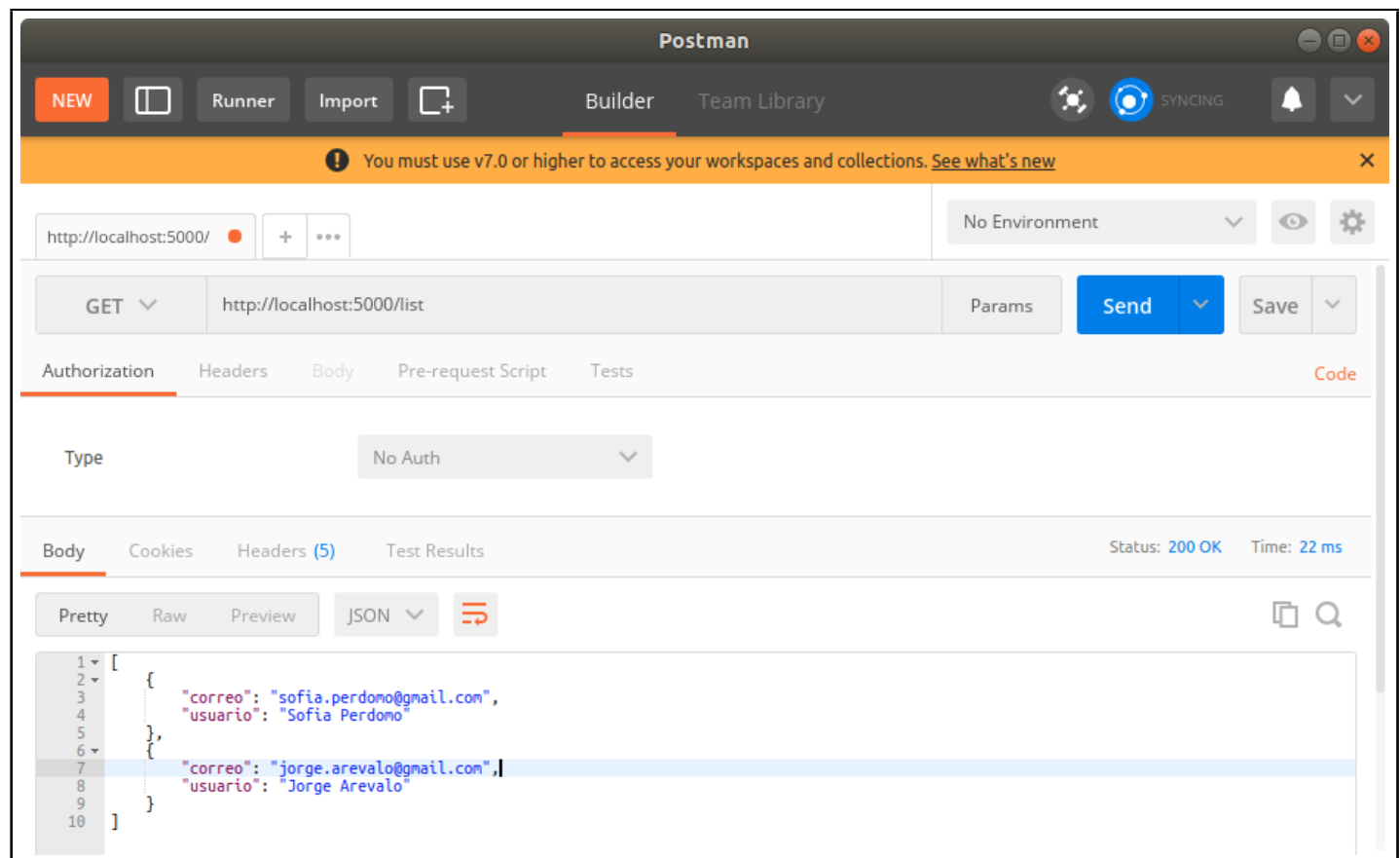Figure 25 shows the server JSON response.

Figure 25: Postman GET method test

10.2.2 POST

Select the POST method, enter the URL `http://localhost:5000/`, select the Body tab, the raw radio button, the JSON (application/json), enter the JSON data and click the [Send] button to test the server POST response.

Figure 26 shows the Postman request and the server JSON response.
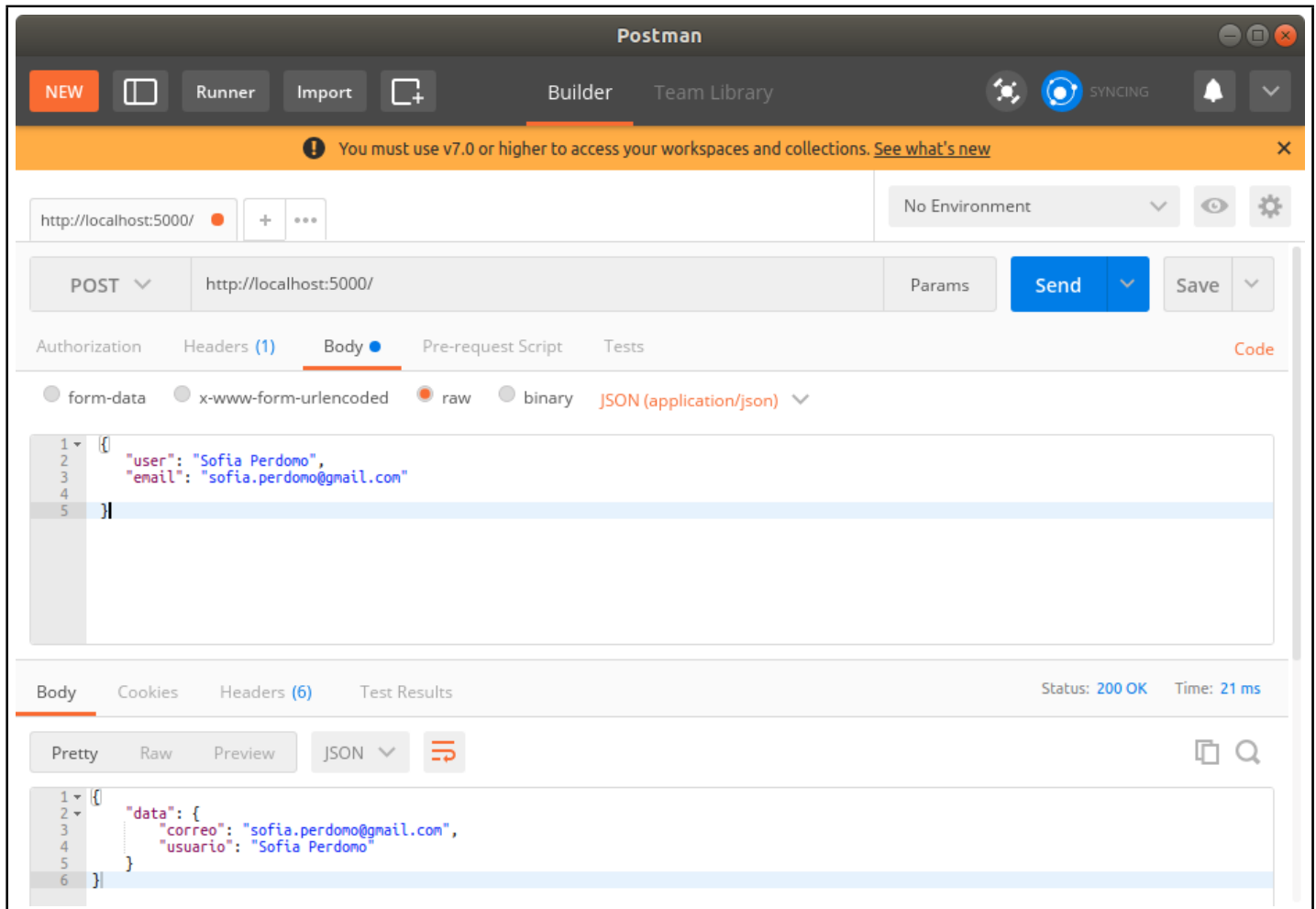
Figure 26: Postman POST method test

11   Uploading Files

An example of file uploading is covered in this section.

11.0.1   PYTHON

The server side python code is in:
File: `chapter_02/src/flask_08.py`

```python
from flask import Flask, request
from werkzeug.utils import secure_filename
import os

app = Flask(__name__)

UPLOAD_FOLDER = "/data_science_fundamentals/chapter_02/uploads"
ALLOWED_EXTENSIONS = {'txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif'}
MAX_CONTENT_LENGTH = 16 * 1024 * 1024
#Define the path to the upload folder
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
#Specifies the maximum size (in bytes) of the files to be uploaded
app.config['MAX_CONTENT_LENGTH'] = MAX_CONTENT_LENGTH

def allowed_file(filename):
    return '.' in filename and \
            filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS

@app.route('/uploader', methods = ['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        # check if the post request has the file part
        if 'file' not in request.files:
            return 'No file part'

        file = request.files['file']

        if file.filename == '':
            return 'No selected file'

        if file and allowed_file(file.filename):
            filename = secure_filename(file.filename)
            file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
            return 'file uploaded successfully'
        else:
            return 'No allowed extension'

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

Listing 22: Example of variable types.

From lines 7 to 13, the configuration variables are defined (upload folder, allowed extensions and max content length).

```python
UPLOAD_FOLDER = "/data_science_fundamentals/chapter_02/uploads"
ALLOWED_EXTENSIONS = {'txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif'}
MAX_CONTENT_LENGTH = 16 * 1024 * 1024
#Define the path to the upload folder
```

```
11 app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
12 #Specifies the maximum size (in bytes) of the files to be uploaded
13 app.config['MAX_CONTENT_LENGTH'] = MAX_CONTENT_LENGTH
```

A function that filters the allowed file extensions is defined between lines 15 to 17.

```
15 def allowed_file(filename):
16     return '.' in filename and \
17            filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
```

From lines 19 to 36, the upload file function is defined.

```
19 @app.route('/uploader', methods = ['GET', 'POST'])
20 def upload_file():
21     if request.method == 'POST':
22         # check if the post request has the file part
23         if 'file' not in request.files:
24             return 'No file part'
25
26         file = request.files['file']
27
28         if file.filename == '':
29             return 'No selected file'
30
31         if file and allowed_file(file.filename):
32             filename = secure_filename(file.filename)
33             file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
34             return 'file uploaded successfully'
35         else:
36             return 'No allowed extension'
```

11.0.2   HTML

The HTML client side is included in:
File: `chapter_02/src/upload_01.html`

Listing 23: HTML form to upload files.

```
1 <html>
2 <body>
3 <form action = "http://localhost:5000/uploader" method = "POST"
4  enctype = "multipart/form-data">
5  <input type = "file" name = "file" />
6  <input type = "submit"/>
7 </form>
8 </body>
9 </html>
```

The form element is defined in line 3. The attribute enctype="multipart/form-data" is required to upload files. The input element of type="file" is mandatory and defined at line 4.

```
3 <form action = "http://localhost:5000/uploader" method = "POST"
4  enctype = "multipart/form-data">
5  <input type = "file" name = "file" />
6  <input type = "submit"/>
7 </form>
```

11.1   Test

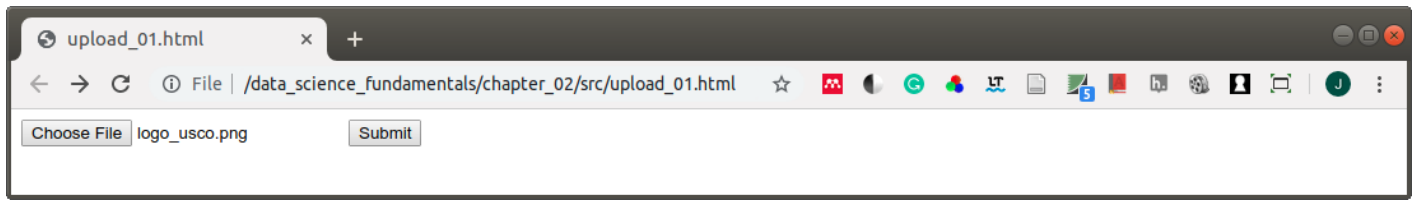Figure 27 shows the HTML form to upload files to the server.

Figure 27: Variable type uuid

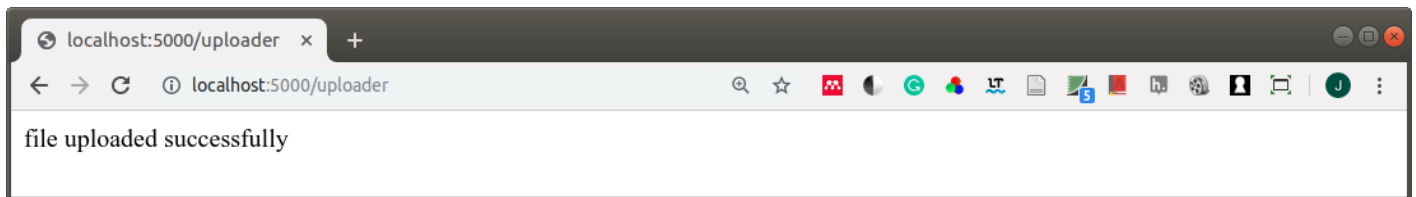Figure 28 shows a message with the server response when a file is uploaded.

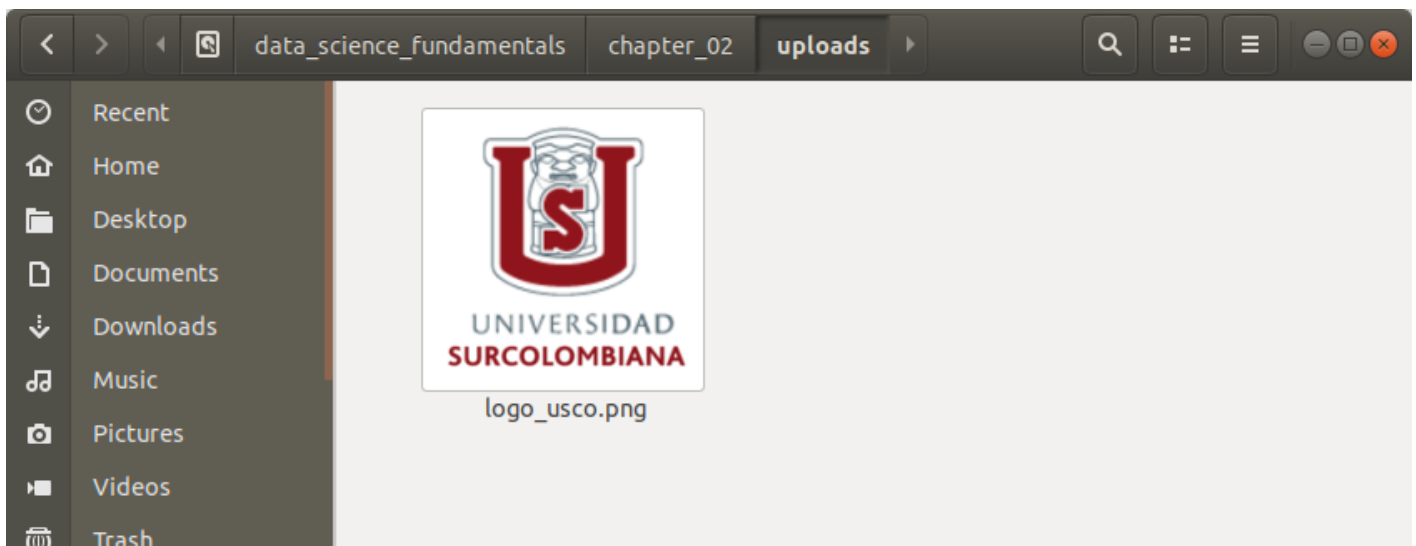Figure 28: HTML for to upñoad files

Figure 29 shows the upload file folder and its contents.

Figure 29: Uplod file folder