

---

# REpresentational State Transfer (REST)

---

PID\_00275902

Silvia Llorente Viejo

---

Recommended minimum reading time: 4 hours

---



---

Universitat  
Oberta  
de Catalunya

---

**Silvia Llorente Viejo**

The assignment and creation of this UOC Learning Resource have been coordinated by the lecturer: Joan Manuel Marquès Puig

First edition: February 2022  
© of this edition, Fundació Universitat Oberta de Catalunya (FUOC)  
Av. Tibidabo, 39-43, 08035 Barcelona  
Authorship: Silvia Llorente Viejo  
Production: FUOC  
All rights reserved

*Reproduction, copying, distribution or public communication of all or part of the contents of this work are strictly prohibited without prior authorization from the owners of the intellectual property rights.*

## Contents

<b>Introduction.....</b>	<b>5</b>
<b>Objectives.....</b>	<b>6</b>
<b>1. What is REpresentational State Transfer (REST)?.....</b>	<b>7</b>
1.1. Overview .....	7
1.2. Relationship between REST and HTTP .....	9
1.2.1. HTTP methods and REST operations .....	10
1.3. Implementation of REST-based Services .....	11
1.3.1. Format of requests .....	14
1.3.2. Javascript Object Notation (JSON) .....	18
1.3.3. Format of responses .....	19
1.3.4. eXtensible Markup Language (XML) .....	20
1.4. Comparison with other types of web services .....	21
1.4.1. SOAP-based web services .....	22
<b>2. REST services with Java language.....</b>	<b>29</b>
2.1. Server-specific operations .....	33
2.1.1. Creating the service .....	33
2.1.2. Definition of operations .....	33
2.1.3. Sending the request .....	36
2.1.4. Sending the response .....	37
2.2. Client-specific operations .....	38
2.2.1. Programming a REST service client application .....	38
<b>3. REST services with other programming languages (Node.js, Python).....</b>	<b>41</b>
3.1. Node. Js .....	41
3.1.1. Development of a REST API with Node.js and ExpressJS .....	41
3.2. Python .....	44
3.2.1. Development of a REST API with Python and Flask .....	44
<b>Summary.....</b>	<b>47</b>
<b>Bibliography.....</b>	<b>49</b>



## Introduction

Communication between systems has been evolving very rapidly in recent years. We have moved from making "tailor-made" communications between systems to defining open mechanisms based on standards and guidelines.

The objective of this module is to describe web services based on REpresentational State Transfer (REST). REST defines an architecture for how to implement Web services based on Hypertext Transfer Protocol (HTTP) methods, the supreme web protocol.

Thus, it describes REST's general operation, its relationship with HTTP, and how requests can be made and responses received. REST web services are also compared with web services based on the Simple Object Access Protocol (SOAP), one of the most commonly used mechanisms for communication between heterogeneous systems before the emergence of REST.

In addition, we will give some details of how a REST-based web service should be implemented, giving some concrete examples in different programming languages such as Java, Node.js and Python.

## Objectives

Through the study of this module, you will achieve the following objectives:

- 1.** Understand what REpresentational State Transfer (REST) is and its basic architecture.
- 2.** Understand the relationship between REST and the different methods of the HTTP protocol.
- 3.** Understand the basics for defining REST services.
- 4.** Understand other types of web services and their relationship with REST.
- 5.** Learn how to implement Java-based REST services and how to connect to them.
- 6.** Learn the basics for implementing REST clients and services based on other programming languages.

# 1. What is REpresentational State Transfer (REST)?

REpresentational State Transfer (REST) defines an architecture of how to implement distributed web services in a simpler way than other existing architectures, such as Simple Object Access Protocol (SOAP) or Remote Method Invocation (RMI).

REST uses the methods of HyperText Transfer Protocol (HTTP) <sup>1</sup> for invoking Create, Read, Update and Delete (CRUD) operations, instead of defining specific operations as it is done in other remote operation invocation mechanisms (for example, SOAP or RMI). This means that, when we want to define a query operation in a REST-based service, instead of defining an operation with the `queryResource` name, where the resource code is passed as a parameter, a Uniform Resource Locator is defined to access the resource with the HTTP GET method, such as `www.example.com/resource/code` <sup>2</sup>. In this specific case, the parameter is passed within the URL, but different mechanisms and/or formats can be used for passing parameters. The result of this operation will be sent within an HTTP response message and can have different formats, as we will see later in this module.

Finally, although REST is not a standard, it is based on standards such as HTTP, eXtensible Markup Language (XML) <sup>3</sup> or Javascript Object Notation (JSON) <sup>4</sup>. Throughout this module, we will see how to use these standards to offer REST-based web services.

## 1.1. Overview

As we have already said, REST is not a standard. Instead, it is an architectural style or design pattern of an Application Programming Interface (API) that works on HTTP. It was initially defined by Roy Fielding in his doctoral thesis, in 2000.

To understand better how REST behaves, we will briefly explain how the HTTP protocol works. HTTP is a Client-Server protocol where the client and the server communicate through requests (from the client to the server) and responses (from the server to the client) with a predetermined format, as shown in Figure 1.

### About REST, SOAP and RMI

REST: REpresentational State Transfer is a style of HTTP-based web services.

SOAP: Simple Object Access Protocol is a style of web services that makes extensive use of the XML language.

RMI: Remote Method Invocation is a mechanism for invoking remote objects in Java language.

<sup>(1)</sup> **Internet Engineering Task Force (IETF)** (2014) Hypertext Transfer Protocol (HTTP/1.1), RFC 7230 to 7235, <<https://tools.ietf.org/html/rfc7230>> <<https://tools.ietf.org/html/rfc7235>>

<sup>(2)</sup> General note: none of the links included in the text have separation hyphens

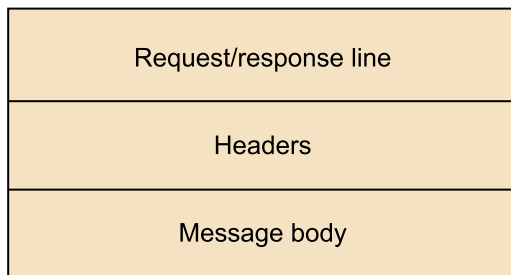
<sup>(3)</sup> **World Wide Web Consortium (W3C)** (2006). Extensible Markup Language (XML) 1.1 (Second Edition), <<https://www.w3.org/tr/xml11>>

<sup>(4)</sup> **European Computer Manufacturers Association (ECMA)** (2017). The JSON (Javascript Object Notation) Data Interchange Syntax ECMA – 404 Standard, <<http://www.ecma-international.org/publications/files/ecma-st/ecma-404.pdf>>

### Bibliographic reference

Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. [online]: <[https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding\\_dissertation.pdf](https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf)>.

Figure 1. General structure of messages exchanged over HTTP



The first line of the message changes depending on whether the message is a request or a response. For the request message, the request line contains the HTTP method, the URL of the resource (without the protocol or server name), and the HTTP protocol version being used. Below, we can see an example of a request line, where the GET method is used to request the `index.html` page, with the version 1.1 of the HTTP protocol.

```
GET index.html HTTP/1.1
```

In the case of the response message, the response line contains the HTTP version with which the server responds, the status code of the request, and a textual description associated with this code. Below, we can see what a successful response to the previous request would look like. It indicates that the protocol is the 1.1 version, and the response code is 200, corresponding to a correct answer, indicating that the `index.html` page has been found on the server, and its contents are sent in the body of the response message.

```
HTTP/1.1 200 OK
```

The headers section can be found in the two types of messages, both in requests and in responses. Some headers are common to both types of messages and others are specific to the request or the response. Below, we can find an example of a header common to the two types of messages, `Content-Length`. It tells us the length in bytes of the message body.

```
Content-Length: 2000
```

Finally, the body of the message contains information regarding the request, such as parameters to be able to make it, or the content of the response. In the specific case of the GET method, the body of the request must be empty.

In addition, HTTP is a stateless protocol, that is, the result of requests does not depend on previous requests. In case we need to store the state, it is necessary to use external mechanisms, such as cookies or sessions.



Thus, REST uses the mechanisms provided by HTTP to enable the implementation of web services. The architectural style of REST has some specific features that we define below:

- **It is stateless:** each request from the client to the server must contain all the information required to understand the request, and cannot take advantage of context stored on the server.
- **Cache:** to improve the efficiency of responses, they must be able to indicate whether or not they can be cached in intermediate systems, such as proxies, or even browsers.
- **Uniform interface:** all resources are accessible through a generic interface (for example, the one provided by the HTTP methods: GET, POST, PUT, DELETE, etc.).
- **Resources with name:** the system is composed of resources that are given a name with a URL (for example, `www.example.com/resource/code`, we give access to a resource from its code).
- **Interconnected resource representations:** resources are interconnected via URLs and allow the client to move from one state to another by "navigating" between representations. It follows the same logic as in a web application, where the web pages are interconnected with each other and we can navigate by following the links. For example, the URL `www.example.com/resource/list` allows requesting the list of all resources. The result should return a URL for each resource, to obtain its basic data or more details, with a URL of the type `www.example.com/resource/code` or `www.example.com/resource/code/details`.

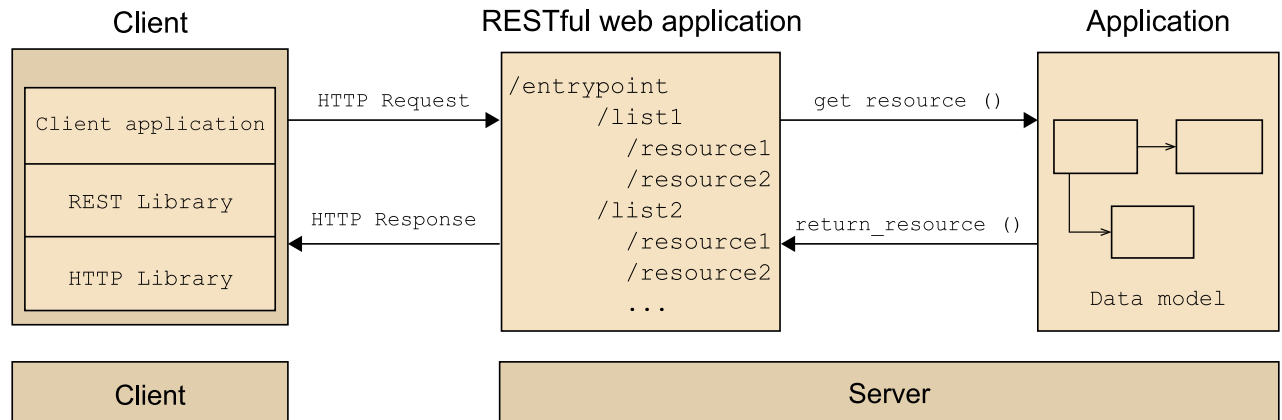
How REST uses HTTP to send and receive data, and manage resources by using the CRUD operations, is explained in greater detail in the following sections.

## 1.2. Relationship between REST and HTTP

A RESTful web application exposes its information through resources. It also gives the client options for taking action on these resources, such as creating new ones (e.g., creating a new user or product) or changing existing ones (e.g., modifying the description of a product or its price). To do this, we should use HTTP commands instead of defining specific operations. Therefore, to request information about a product, an operation such as `requestProductData(idProduct)` is not defined; instead, we will use the HTTP GET method, requesting the data in the line of this GET protocol: `GET /dataProduct?idProduct`. The result of the operation can have any format, since HTTP allows sending any type of file in response to a request.

The service implemented in an application behind a RESTful web application may not have any type of web interface, and its communication with the RESTful application can be done through other communication mechanisms, such as sockets, Remote Procedure Call (RPC), Remote Method Invocation (RMI) or the direct execution of programs from the RESTful application.

Figure 2. Architecture of a REST service



The diagram in Figure 2 shows the general architecture of a REST-based service, where a RESTful web application provides access to an application which runs on the server through the HTTP protocol. The clients will connect to the RESTful web application by using this protocol. On the client side, there must be a client application, a REST library (which can be optional) and an HTTP library, which establishes communication with the service.

### 1.2.1. HTTP methods and REST operations

The HTTP methods employed in a REST API are those listed in Table 1. Table 1 presents the HTTP methods employed in REST, the CRUD operation to which it corresponds, and a brief description of what the operation, using each method, has to do.

Table 1. HTTP methods and their function in REST

Method	CRUD Operation	Description
POST	Create	It must be employed to create a new resource that is added to the current resource collection. It can also be employed to modify, although its use is not recommended <sup>5</sup> .
GET	Read	It has to be employed to request the data of a resource. Although GET can receive parameters, it should not be employed for creation and/or modification operations.
PUT	Update	It must be employed to update an existing resource, which replaces the current resource with the one that is sent in the request message. If the resource sent by the client is a new one, this method would create it and conveniently inform the client.

<sup>(5)</sup>An *Idempotent* operation is one that always gives the same results, given the same parameters. HTTP, GET, PUT and DELETE are idempotent. On the contrary, a *Non-idempotent* operation is one that can give different results with the same parameters.

Method	CRUD Operation	Description
PATCH	Partial Update	It must be used to modify a resource, without sending all of it, just sending the data to be modified. It is similar to PUT, but must contain instructions that indicate what data from the original resource should be modified.
DELETE	Delete	It has to be used to delete a resource.

Table 2 describes the possible response codes for each of the operations in Table 1. These codes correspond to HTTP protocol codes, as defined in the Internet Engineering Task Force (2014).

Table 2. Possible HTTP response codes to each REST operation

Method	Correct answer code	Erroneous response code
POST	201 Resource created	404 Resource not found 409 Conflict, resource already exists
GET	200 Correct	404 Resource not found
PUT	200 Correct 201 Resource created	204 No more content 404 Resource not found
PATCH	200 Correct	204 No more content 404 Resource not found
DELETE	200 Correct	404 Resource not found

#### Bibliographic reference

Internet Engineering Task Force (IETF) (2014). Hypertext Transfer Protocol (HTTP/1.1), RFC 7230 a 7235. [online]: <<https://tools.ietf.org/html/rfc7230>> and <<https://tools.ietf.org/html/rfc7235>>.

### 1.3. Implementation of REST-based Services

To implement a REST-based service, it is important to define how it should be. In fact, a REST-based service exposes a series of operations on resources stored on a server, through an application programming interface (API, from now on).

Thus, one of the most relevant concepts within REST is the resource. Any information that can be given a name is a resource: a document or image, a temporary service (e.g., "the weather during the next three days in Barcelona"), a collection of other resources, a non-virtual object (e.g., a person), etc. In this way, a concept that can be represented as a reference to a hypertext can be a resource. Resources can be grouped into collections, where each collection can contain several disordered resources. Collections are also resources for themselves.

Here, there are some examples to clarify the concept of resource. *Clients* can be a collection resource and *client* a single resource (in a domain such as banking). The *clients* collection can be identified with the URI `/clients`, and we can identify a single client with the URI `/clients/{clientId}`. In addition, a resource can have sub-collections. For example, the *accounts* subcollection of a specific *client* can be identified by the URI `/clients/{clientId}/accounts`. In a similar way to collections, a single resource can also be

accessed within subcollections. Thus, a specific *account* within the *accounts* of a *client* could be accessed with `/clients/{clientId}/accounts/{accountId}`.

As we have already seen in the examples, REST APIs use Uniform Resource Identifiers (URIs) to access resources. The designer of a REST API has to define a resource model that is understandable to their potential clients' developers. When resources are logically named, an API is intuitive and easy to use. If it is badly designed, this same API can be difficult to use and understand.

Although there is no standard when it comes to defining REST API resource names, here there is a set of guidelines for doing it in the most understandable way possible.

### Guidelines for DEFINING URIs for RESOURCES in a REST API

1) Use names (things) instead of verbs (actions) to define resources. Names have characteristics that verbs do not have, such as attributes. Examples of names can be system users, user accounts, network devices, and so on. Within the names, we can distinguish between single resources (in singular) and collections (in plural).

Examples:

```
http://api.com/management-devices
http://api.com/management-devices/devices-managed
http://api.com/management-devices/devices-managed/{id}
```

2) Consistency is a key factor. Consistent naming conventions and URI formatting have to be used with minimal ambiguity and maximum readability and maintenance. Some design tips are: use the forward slash (/) to indicate hierarchical relationships; do not put the forward slash (/) at the end of the URI; do not use capital letters; and, do not use extensions in resources.

Examples:

Hierarchy

```
http://api.com/management-devices/devices-managed
http://api.com/management-devices/devices-managed/{id}
```

The use of the slash at the end is not recommended (the second option would not be correct)

```
http://api.com/management-devices/devices-managed
http://api.com/management-devices/devices-managed/
```

### Bibliographic reference

Internet Engineering Task Force (IETF) (2005). Uniform Resource Identifier (URI): Generic Syntax, RFC 3986, <<https://tools.ietf.org/html/rfc3986>>.

Use of capital letters. They can be used in the protocol and in the server, but not in the resource part (the third URI would not be valid, as the word my (My) is capitalized)

```
HTTP://API.ORG/my-folder/my-doc
```

```
http://api.org/my-folder/my-doc
```

```
http://api.org/My-folder/my-doc
```

Do not use file extensions (the second option would not be correct)

```
http://api.com/management-devices/devices-managed
```

```
http://api.com/mangement-devices/devices-managed.xml
```

**3)** Do not use the names of CRUD operations in resource names. What should be done is to use the corresponding HTTP method, such as GET to obtain the data, POST to create them, etc. The same URI is used, but with a different HTTP method, so the effect on resources is different.

Examples:

To obtain all devices

```
HTTP GET http://api.com/management-devices/devices-managed
```

To create a new device

```
HTTP POST http://api.com/management-devices/devices-managed
```

**4)** Use the URI query section to filter or sort resources instead of creating new API entries. The query section is everything that comes after the `?`, and for introducing more than one parameter, `&` is used. The format of this section is `?key1=value1&key2=value2`.

Examples:

```
http://api.com/management-devices/devices-managed
```

```
http://api.com/management-devices/devices-managed?region=cat
```

```
http://api.com/management-devices/devices-managed?region=cat&date=2020-03-18
```

### 1.3.1. Format of requests

Requests to a REST service have to go in an HTTP request message. Depending on the HTTP method, we may send the data differently. First, we will describe how the data can be sent, and then indicate with which HTTP methods each mechanism can be used.

#### URL

The data can be sent in the same request URL. Thus, when we deal with specific resources within collections, we can employ identifiers to refer to a specific resource. In a previous example, we used the `clients` collection to refer to the clients of a bank. The URI `/clients` identifies the entire collection, that is, all clients. The way to identify a single client would be a URI of the `/clients/{clientId}` type, where we have to give a client ID to make the request in the `{clientId}` field. The resulting URI would be `/clients/1111111h`, where the data of the client identified by the identifier `1111111h` are requested, corresponding to a national identity document (ID).

#### Query section of the URL

Also, within the same URL, data can be sent in the query section of the URL. The query section begins with the symbol `?` and may contain several `key=value` pairs, separated from each other by the symbol `&`. Thus, continuing with the example of clients of a bank, we could ask for the user details as follows: `/clients/1111111h?info=details`, where the `info` key can take different values; in this specific case, all the user's details are requested. If we give a user's bank account as an example, we could filter the movements by date, as follows: `/clients/1111111h/accounts/12341234?start-date=1-1-2020&end-date=31-03-2020`.

#### Request message body

There are different mechanisms for sending the data within the body of the request message:

- `application/x-www-form-urlencoded` format, which is the same format as in the query section, using the `key=value` format. With this format, all the necessary fields, separated by the symbol `&`, are sent. In this case, a URI example could be `/clients/1111111h/accounts/12341234` and the parameters within the message body would be `start-date=1-1-2020&end-date=31-03-2020`.

#### About MIME

Internet Engineering Task Force (IETF) (2015). Returning Values from Forms: multipart/form-data, RFC 7578, <<https://tools.ietf.org/html/rfc7578>>.

- `multipart/form-data` format, which is a Multipurpose Internet Mail Extensions (MIME) multipart message, where parameters are sent within the message body separated by a string of characters called *boundary*. The advantage of this format over the previous ones is that it allows sending more data, even data in binary format. This format is aimed at sending files with a form that is displayed to a user in a browser.
- Contents of `application/xml` (or `text/xml`), `application/json` type, and others supported by the REST service (which must be indicated in the service headers, as we will see later).

**XML and JSON**

XML means *eXtensible Markup Language* and its MIME types can be `application/xml` (when directed to a server) or `text/xml` (when it is aimed at being understood by a person).

JSON stands for *JavaScript Object Notation* and its MIME type is `application/json`.

Example of `multipart/form-data` message sent within an HTTP message of a POST command:

```
Content-Type: multipart/form-data; boundary=--73532303139
Content-Length: 834
--73532303139
Content-Disposition: form-data; name="text1"
text 123 abc
--73532303139
Content-Disposition: form-data; name="text2"
xyz
--73532303139
Content-Disposition: form-data; name="file1"; filename="a.txt"
Content-Type: text/plain
Content file a.txt.
--73532303139
Content-Disposition: form-data; name="file2"; filename="a.html"
Content-Type: text/html
DOCTYPE html><title>Content a.html.</title>
--73532303139
Content-Disposition: form-data; name="file3"; filename="fish.jpg"
Content-Type: image/jpeg
Binary data here in base64 format
--73532303139--
```

In this message, two text fields (`text1` and `text2`) are being sent, a plain text file named `a.txt`, an HTML file named `a.html`, and an image, named `fish.jpg`. Each file indicates what its MIME type is, with the `Content-Type` header. Each data type is separated by the field `boundary`, which has the value `--73532303139`, as defined at the beginning of the message in the `Content-Type` header of the full message.

Here there are some examples of how the requests of each of the HTTP methods can be. The request lines and the message headers are included, as well as the content of the message body, if there is any.

## POST request example

```
POST /client/json HTTP/1.1
Accept: application/json
Content-Type: application/json
Content-Length: 85
Host: example.com

{
  "Id": 12345,
  "Client": "Arnau Sola",
  "Quantity": 3,
  "Price": 10,00
}
```

In this example, we use the JavaScript Object Notation (JSON) (ECMA, 2017) format to send the data. This format is explained in detail in the section "JavaScript Object Notation", taking this example of data as a reference. In this case, the data is sent within the body of the message.

In addition, we find several request headers: `Accept`, that tells us in which format the answer can reach us, `Content-Type`, which tells us the type of content we send, `Content-Length`, which tells us the length of the message body and `Host`, which tells us which server we have to connect to.

## GET request example

```
GET /assigns/json HTTP/1.1
Accept: application/json
Host: example.com
```

In this GET request, we can see that there is no content in the message (the GET method does not allow it) and that it is accepted that the response is in JSON format (`application/json`). We can also see the server to which we make the request.

## PUT request example

```
PUT /client/json HTTP/1.1
Accept: application/json, application/xml
Content-Type: application/json
Content-Length: 85
Host: example.com

{
  "Id": 12345,
  "Client": "Arnau Sola",
  "Quantity": 1,
  "Price": 10,00
}
```



```
}
```

This PUT request is very similar to the POST request. We see that the complete client data is sent, changing the value 3 that was in POST for a 1. This will cause the entire resource on the server to be modified.

#### PATCH request example

```
PATCH /client/json HTTP/1.1
Accept: application/json, application/xml
Content-Type: application/json
Content-Length: 85
Host: example.com
{
  "Id": 12345,
  "Price": 15,00
}
```

This PATCH request is a subset of those made in POST and PUT. Here, only the Id and price are sent, which must be modified. The rest of the resource data is maintained.

#### DELETE request example

```
DELETE /client/json/1234 HTTP/1.1
Accept: application/json, application/xml
Host: example.com
```

This DELETE request contains the data of the resource that has to be deleted in the URL. In this case, we want to delete the client with Id 1234. There is no content in the message.

Table 3 shows a summary of how data can be sent based on the HTTP method used. In some cases, several options can be combined. Some HTTP methods might support more ways to send the data, but then, they would not follow the REST guidelines. That is why they do not appear in the table, even though the method specification in HTTP would support it.

Table 3. How to send requests according to the HTTP method used

Method	Request Options
POST	URL, which indicates the resource to be created. Query section, which allows passing parameters. The body of the request message. In this case, a file of the type supported by the REST service can be sent, <code>application/x-www-form-urlencoded</code> or <code>multipart/form-data</code> .

Method	Request Options
GET	URL, which indicates the resource to be queried. Query section, which allows passing parameters. In this case, nothing can be sent to the message body because the method does not support it.
PUT	URL, which indicates the resource that has to be replaced or created. The body of the request message with a file of type supported by the REST service.
PATCH	URL, which indicates the resource to be partially modified. The body of the request message with a file of type supported by the REST service.
DELETE	URL, which indicates the resource to be deleted.

### 1.3.2. Javascript Object Notation (JSON)

JavaScript Object Notation (JSON) (European Computer Manufacturers Association, 2017) is a lightweight data exchange format. It is an easy format for people to generate and for machines to interpret. In addition, it is a text format independent of the programming language, but, at the same time, it uses common conventions for languages such as C, C++, Java, Python, and many others. These properties make it very suitable as a data exchange language.

JSON is based on two structures:

- A collection of name/value pairs. In many programming languages this is represented as an object, record, structure, dictionary, hash table or list with keys, among others.
- An ordered list of values. In many languages, this is represented as a vector, matrix, list, or sequence.

```
{  
  "Id": 12345,  
  "Client": "Arnau Sola",  
  "Quantity": 3,  
  "Price": 10,00  
}
```

The example represents an object with client's data. The curly brackets ({}), indicate the start and end of the object. Inside it, we find a set of names/values. The first name we find is "Id" with the value 12345. The name is separated from the value with the symbol ":". When a name or value has the symbol "", it is a string of characters (e.g., "Id" or "Arnau Sola"). To separate name/value pairs from each other, we use the symbol ",". We can also represent numbers, both integer (e.g., 12345 or 3) and double (e.g., 10.00) values. We can include

objects within objects and also use other types of data such as Boolean values (`true`, `false`). For a more detailed description, see European Computer Manufacturers Association (2017).

### 1.3.3. Format of responses

The responses provided by a REST service can be indicated in the request headers (see subsection 1.3.1). For example, if we want the service to return a text-type file, in the request we will indicate it through the `Accept` header, where we have to put `text/plain` as a value. In this header, the client application indicates which service response formats are accepted. We can put different values separated by commas.

Example of *Accept* header in the request to the REST service, indicating which format, or formats, the client accepts.

The most common are files in JSON or XML format, which are the ones that appear in the example. In the first line, both formats are indicated, in the second line, only JSON, and in the third, only XML. Only one `Accept` header is sent in a request, even though it may contain more than one value, as seen in the first example.

```
Accept: application/json, application/xml
Accept: application/json
Accept: application/xml
```

The following is an example response from a REST service for each HTTP method.

#### POST response example

```
HTTP/1.1 200 OK
Content-Length: 19
Content-Type: application/json
{"success":"true"}
```

In this example, the response code indicates that the request was successful. In addition, we find two headers that indicate the length and the type of content. Finally, there is the content of the message, which also indicates that the request has worked correctly. In this case, the data is sent in JSON format, with a single name/value pair within the object, with a Boolean value.

#### GET response example

```
HTTP/1.1 200 OK
Content-Type: application/json
{
```

```

    "public": {
      "students": "delegates",
      "subjects": "all",
      "teachers": "all"
    },
    "private": {
      "me": "profile",
      "subjects": "XAI"
    }
  }
}

```

Here, the response with code indicates that the request went well. The data is returned in JSON format, referring to objects associated with subjects, teachers and students. There are two objects, one called "public" and another called "private", which contain several pairs of name/value objects. All values are of string type.

#### Example of PUT, PATCH, and DELETE response

```

HTTP/1.1 200 OK
Content-Type: application/xml; charset=utf-8
<?xml version="1.0" encoding="utf-8"?>
<Response>
  <ResponseCode>0</ResponseCode>
  <ResponseMessage>Success</ResponseMessage>
</Response>

```

In this example, the response code indicates that the request was successful, and the data is returned in XML format. This format is explained in greater detail in the "eXtensible Markup Language (XML)" section.

### 1.3.4. eXtensible Markup Language (XML)

eXtensible Markup Language (XML) (World Wide Web Consortium, 2006) is a format standardized by W3C<sup>6</sup>, which describes how XML documents should be. These documents are made up of storage units, called entities, which can contain different types of data. The formalized data are composed of characters, some of them defining the markup, and others defining the data. Markup encodes a description of the document's storage distribution, as well as the logical structure. XML provides a mechanism for imposing constraints on both the distribution and the logical structure of data.

<sup>(6)</sup>World Wide Web Consortium (W3C), <<https://www.w3.org/>>. [Date of access: March 2020]

#### Note

XML derives from SGML, which means *Standard Generalized Markup Language*. By definition, XML documents conform to SGML.

```

<?xml version="1.0" encoding="utf-8"?>
<Response>
  <ResponseCode>0</ResponseCode>

```

```
<ResponseMessage>Success</ResponseMessage>
</Response>
```

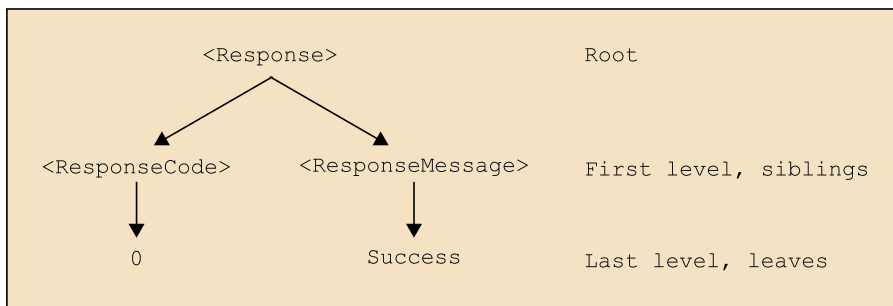
In the example, a successful response code to a PUT, PATCH or DELETE request, that we have seen in section 1.3.2, is represented. The first line indicates that this is an XML document, of 1.0 version, and it is encoded with UTF-8. This line is required in valid XML documents. Then we find the *Response* root element. The beginning of an element is marked with the symbols "<" and ">" and the end with "</" and ">". Within the root element, we find the *ResponseCode* and the *ResponseMessage* subelements, that are the children of *Response*, and siblings among themselves. The elements have a hierarchical relationship from the root to the leaves. All subelements of the same level are siblings. Finally, we find the values 0 and *Success*, which correspond to the leaves, which are final elements, have no children, and can contain string-type values, decimal or real numbers, and other basic data types defined in the XML standard.

#### UTF-8

UTF-8 stands for *8-bit Unicode Transformation Format*. It is an ISO-defined character encoding format (*International Organization for Standardization*).

Figure 3 shows the relationship between the different XML elements shown in the example.

Figure 3. Hierarchy of an XML document



In this case, we have seen that the answer is in XML format. It would also be possible to receive a response in JSON format, as in the example with the POST method seen in the "Javascript Object Notation (JSON)" section.

## 1.4. Comparison with other types of web services

The other widely used web services are those based on the Simple Object Access Protocol (SOAP) <sup>7</sup>. SOAP is a World Wide Web Consortium (W3C) standard, which defines an XML-based framework for communicating remote processes. Both the messages that are exchanged and the mechanism to define the service use XML for sending and describing the data. In this section, we give a brief overview of SOAP and other related technologies, and compare their features with REST web services.

<sup>(7)</sup>World Wide Web Consortium (W3C) (2007). Simple Object Access Protocol (SOAP) Version 1.2, <<https://www.w3.org/TR/soap12/>>

### 1.4.1. SOAP-based web services

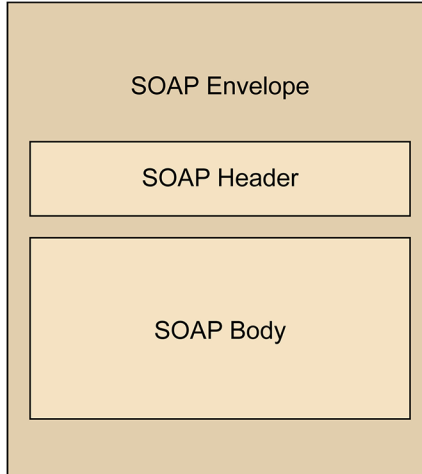
The SOAP standard is an application communication protocol that defines the format of the request and the response messages. This protocol is independent of the platform and the programming language in which the client and the server are implemented, since communication is based on messages expressed in XML format. These messages are primarily exchanged through the HTTP protocol, even though they could be sent through other application-level protocols.

In SOAP, messages between the client and the web service are always sent in XML format. The following section describes some features of SOAP services, such as sending requests and responses, or how to define the operations of a service.

#### SOAP request and response messages

Figure 4 shows a diagram of the format of SOAP requests and responses. The first element is the SOAP Envelope (<Envelope>), which encloses all the data of a request and/or response. The other elements present in a SOAP message are the SOAP Header and the SOAP Body.

Figure 4. SOAP request and response format diagram



The SOAP Header (<Header>) is an optional element of the SOAP message, and contains service-related information that has to be processed by SOAP nodes throughout the message flow. The defined information depends purely on the web service, that is, the data which is sent to the header is only understood by this specific service.

The SOAP Body (<Body>) is a mandatory element containing information intended for the ultimate recipient of the message. This element and its subelements are used to exchange information between the SOAP sender and the ultimate receiver of the SOAP message. SOAP defines a subelement, the SOAP Fault (<Fault>), which is used to send errors that have occurred. The rest of

the elements that appear in the SOAP Body are defined by the web service. They are the operations, the parameters and the results of the operations that are offered.

Below there is an example of a SOAP request message that is sent within an HTTP POST command. The body of the message is sent in `application/soap+xml` format, indicating that it is sent in XML format, following the guidelines of the SOAP standard.

Inside the SOAP message, we can see that there is a `<Header>` element, containing a subelement, which indicates that the maximum time to respond to the request is 10,000. All nodes processing this SOAP request must understand this header, as indicated by the `mustUnderstand` attribute. In case a node does not understand it, it will return a SOAP Fault, and the processing of the web service will not continue.

Next, we find the `Body` element, where the name of an action, expressed as a character string, is sent to a web service that has to return its price.

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

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
    <m:tempsMax value="10000" xmlns:m="http://www.example.org/share"
      mustUnderstand="true"/>
  </soap:Header>
  <soap:Body xmlns:m="http://www.example.org/share">
    <m:PriceShare>
      <m:NameShare>Inditex</m:NameShare>
    </m:PriceShare>
  </soap:Body>
```

The response travels within an HTTP response message, also with the `application/soap+xml` format. The SOAP response message also contains an `Envelope` element, and inside it, we find a `Body` element which carries the response of the web service that returns the value of the share; in this case, it is in real number format.

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn
```

```
<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope">

  <soap:Body xmlns:m="http://www.example.org/share">
    <m:ResponsePriceShare>
      <m:Price>34.5</m:Price>
    </m:ResponsePriceShare>
  </soap:Body>

</soap:Envelope>
```

Now that we have seen how both the request and the response of a SOAP web service travel within the messages of the HTTP protocol, we will explain in more detail how the operations of the service have to be defined. We will also explain the format of the parameters and responses, and how this service is associated with a specific application-level protocol, in this case, HTTP.

### Defining Web Services with Web Services Description Language (WSDL)

Web Services Description Language (WSDL) 2.0 is a W3C recommendation that describes an XML-based language for describing web services. To do this, an abstract model of what the service offers is defined. In addition, the conformity criteria for the documents described in this language are defined.

The main elements of a WSDL are:

- **Description.** It is the root element. The rest of the elements are inside it.
- **Documentation.** It is an optional element that can contain a human-readable description.
- **Types.** It contains the specification of the types of data exchanged between the client and the web service. By default, these data types are defined with an XML Schema<sup>8</sup>.
- **Interface.** It describes which operations the service has, and what messages are exchanged for each operation (input/output). It also allows defining possible error messages.
- **Binding.** It defines how the web service is accessed over the network. Typically, this element defines how to connect to the service over HTTP (although it is not the only possibility).

#### About WSDL

**World Wide Web Consortium (W3C)** (2007). Web Services Description Language (WSDL) Version 2.0, <<https://www.w3.org/TR/wsdl20/>>

<sup>(8)</sup> **World Wide Web Consortium (W3C)** (2012). XML Schema Definition Language (XSD) 1.1, <<https://www.w3.org/TR/xmlschema11-1/>>



- **Service.** It defines where the web service can be accessed in the network. It usually contains a URL where the service can be found.

Below, you can see an example of a WSDL where all the described elements appear.

```
<?xml version="1.0" encoding="utf-8" ?>
<description
  xmlns=           "http://www.w3.org/ns/wsd1"
  targetNamespace= "http://www.example.org/share"
  xmlns:tns=       "http://www.example.org/share"
  xmlns:stns =     "http://www.example.org/share/schema"
  xmlns:soap=      "http://www.w3.org/ns/wsd1/soap"
  xmlns:wsd1x=     "http://www.w3.org/ns/wsd1-extensions" >
<documentation>
  This is the web service documentation.
</documentation>
<types>
  <xs:schema
    xmlns:xs=       "http://www.w3.org/2001/XMLSchema"
    targetNamespace= "http://www.example.org/share/schema"
    xmlns=          "http://www.example.org/share/schema">
    <xs:element name="PeticionPriceShare" type="typePriceShare"/>
    <xs:complexType name="typePriceShare">
      <xs:sequence>
        <xs:element name="NameShare" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
    <xs:element name="ResponsePriceShare" type="typeResponsePriceShare"/>
    <xs:complexType name="typeResponsePriceShare">
      <xs:sequence>
        <xs:element name="Price" type="xs:float"/>
      </xs:sequence>
    </xs:complexType>
  </xs:schema>
</types>
<interface name = "PriceShareInterface" >
  <operation name="PriceShare"
    pattern="http://www.w3.org/ns/wsd1/in-out"
    style="http://www.w3.org/ns/wsd1/style/iri"
    wsdlx:safe = "true">
    <input messageLabel="In" element="stns:RequestPriceShare" />
    <output messageLabel="Out" element="stns:ResponsePriceShare" />
  </operation>
</interface>
<binding name="PriceShareSOAPBinding"
  interface="tns:PriceShareInterface"
```

```
    type="http://www.w3.org/ns/wsdl/soap"
    wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/"
    wsoap:mepDefault="http://www.w3.org/2003/05/soap/mep/request-response">
    <operation ref="tns:PriceShare"/>
  </binding>
  <service
    name      ="ServicePriceShare"
    interface="tns:PriceShareInterface">
    <endpoint name ="PriceShareEndpoint"
      binding ="tns:PriceShareSOAPBinding"
      address ="http://www.example.org/share"/>
    </service>
  </description>
```

The WSDL elements are related to each other. For example, the `interface` and the `binding` attributes of the `service` element are the `interface` and `binding` elements (matching the values defined in the *name* attribute), respectively. The `operation` element within the `binding` (in this case there is only one, but there could be more) matches the `operation` element defined within the `interface` element. Finally, the `input` and the `output`, defined in the `operation` element, must be the data type defined within the `types` element.

The `tns` and `stns` values, in front of some names, correspond to the references to the namespaces where the different elements are defined. The reference to namespaces is usually found in the root element of the XML document; in this specific case, in `description`.

Below are the namespaces defined in this WSDL:

```
xmlns=          "http://www.w3.org/ns/wsdl"
xmlns:tns=       "http://www.example.org/share"
xmlns:stns =     "http://www.example.org/share/schema"
xmlns:wsoap=     "http://www.w3.org/ns/wsdl/soap"
xmlns:wsdlex=    "http://www.w3.org/ns/wsdl-extensions"
```

The first is the default namespace in this document, which refers to the WSDL namespace. After `xmlns` (which means XML namespace), there is no qualified name. This is because we want WSDL namespace items not to need any prefixes to reference them within the document. The next one we find is `xmlns:tns`, which corresponds to the namespace that we define in this XML document. Thus, any element that belongs to this namespace will have to be identified as `tns: nameElement`. The same occurs with `xmlns:stns`, which refers to the data types defined in this document, and used in the operation. Finally, we have two other external namespaces, `xmlns:wsoap` and

`xmlns:wsd1x`, which refer to the SOAP namespace and the WSDL extensions, respectively. Some elements of these namespaces are required to define specific features of the service.

### Comparison between SOAP and REST web services

To compare these two types of web services, we will consider some of their main features:

- Format of requests and responses
- Application-level protocol in which data is sent
- Definition of service operations
- Security

First, we will describe the format of requests and responses in both types of services. While SOAP defines complex structures for sending these requests and responses, always in XML format, REST is much more flexible. In REST, there is no single format for sending requests or responses. In fact, in some cases, it is not even necessary to send any request message, and all the necessary data can be passed with the same URL of access to the resource. Let's look at it with an example:

```
SOAP request, to ask for details of the user with ID 12345
(it will travel in a HTTP POST command)
<soap:Envelope ...>
  <soap:body pb="http://www.example.org/agenda">
    <pb:GetDetailsUser>
      <pb:IDUser>12345</pb:IDUser>
    </pb:GetDetailsUser>
  </soap:Body>
</soap:Envelope>

REST request, to ask for the data of this same user
GET http://www.example.org/agenda/DetailsUser/12345 HTTP/1.1
```

As it can be seen, the request made with REST, where only the URL is required, is much simpler both from the point of view of the amount of data sent and the processing of these, than the SOAP request, where many XML elements, not relevant to the service, are sent.

Response messages can be equally complex in both types of web service, since REST can send any type of file, even in XML format. The main difference is that SOAP only accepts XML, while REST can accept different types of data.

Both types of web services can work through the HTTP protocol. In fact, REST only works through HTTP while SOAP could work through other protocols, as defined in SOAP Version 1.2 Part 2: Attachments, where different strategies for sending and receiving messages are described.

We have also seen that SOAP defines the characteristics of its services (format for sending and responding to messages, type of sending used, application-level protocol, service address, etc.) in a WSDL. Could the same be done with REST? The answer is yes, in the 2.0 WSDL version. The way to do it is found in the `binding` element, where we can indicate that we use the type of binding corresponding to HTTP, with `http://www.w3.org/ns/wsdl/http`, and an HTTP method as a service operation, such as GET (`whhttp:method="GET"`). Here, there is an example of what a `binding` element might look like, which defines this data for a REST service model.

```
<wsdl:binding name="ServiceHTTPBinding"
  type="http://www.w3.org/ns/wsdl/http"
  interface="tns:InterfaceService">
  <wsdl:operation ref="tns:Service" whhttp:method="GET"/>
</wsdl:binding>
```

From a security standpoint, REST works through HTTP, which supports basic authentication and communication security, using Transport Layer Security (TLS)<sup>9</sup>. However, SOAP has the WS-SECURITY<sup>10</sup>, which is a standard that defines a whole series of security mechanisms for web services.

## Conclusions of REST and SOAP services

In this section, we have seen that SOAP and REST web services share some characteristics, such as the use of HTTP as the preferred protocol for sending their data, and some quite different ones, such as the formats for sending requests and responses. This last point has made REST-based web services the most widely used today. The flexibility they provide, with much simpler requests, or the absence of obligation to use a format such as XML, have allowed this type of service to expand, and it is possible to process them easily with any type of device, especially from mobile devices. In addition, the possibility of receiving only a part of the data and linking to the rest of it through URLs also allows them to manage the transfer of data more easily.

<sup>(9)</sup>Internet Engineering Task Force (IETF) (2018). The Transport Layer Security (TLS) version 1.3, <<https://tools.ietf.org/html/rfc8446>>.

<sup>(10)</sup>Organization for the Advancement of Structured Information Standards (OASIS) (2006). Web Services Security: SOAP Message Security 1.1 (WS-Security), <<https://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf>>.

## 2. REST services with Java language

The Java language provides a specification called JAX-RS, defined in JSR 311 (Sun Microsystems, 2008). This specification defines how the Java language supports REST-based web services. It describes how to access resources by defining Java annotations<sup>11</sup> that point to resource-type classes, HTTP methods and parameters, and results to HTTP method calls.

Table 4 describes some annotations used in JAX-RS, and a brief description of them<sup>12</sup>.

Table 4. How to send requests according to the HTTP method used

Annotation	Description
@Path	The root class of access to a Java-language resource must have this annotation, to indicate the URL where this resource can be found. Example: <code>@Path ("/hello")</code> . The base address of our resource will be <code>http://server/hello</code> . This annotation can also be found within the methods, and then the value inside the annotation will be concatenated to the original @Path.
@HTTP_method	Each HTTP method has its own annotation. Thus, we can find <code>@POST</code> , <code>@GET</code> , etc. This annotation within the REST service code indicates with which HTTP method we will respond to a particular operation.
@typeParam	The annotations ending with Param tell us how the parameters can be received. Thus, <code>QueryParam</code> indicates that they are received in the query part of the URL, after the symbol <code>?</code> ; <code>FormParam</code> , that they are received within the message body, with the key-value format; or, <code>PathParam</code> , that they are part of the URL (they are identified inside curly brackets <code>{}</code> in the @Path annotation, corresponding to that method).
@Consumes, @Produces	These two annotations indicate which data are received ( <code>@Consumes</code> ) and which ones are returned ( <code>@Produces</code> ). Within these annotations, the corresponding MIME type must be put. Examples: <code>application/json</code> or <code>text/html</code> . The Java <code>MediaType</code> class defines some of these data types.

Let's look at an example of how a REST service can be implemented where the GET and POST methods are responded to with different configurations for passing parameters. We will focus on these two methods because they are the easiest to test directly from a browser (both from the address bar and from an HTML form).

The first thing we find in the service are the JAX-RS classes that we will use within our Java resource, which is called "hello". Java package names begin with `javax.ws.rs`, which identifies the Java library (`javax`) that implements web services (`ws`) based on REST (`rs`).

<sup>(11)</sup>Oracle Corporation (2014). The Java Tutorial, Annotations, <<https://docs.oracle.com/javase/tutorial/java/annotations/index.html>>.

### Bibliographic reference

**Sun Microsystems** (2008). JAX-RS: Java™ API for RESTful Web Services. [online]: <<https://download.oracle.com/otn-pub/jcp/jaxrs-1.0-fr-eval-oth-JSpec/jaxrs-1.0-final-spec.pdf>>.

### Java Annotations

Annotations in the Java programming language are in the (`@name`) format, where the `@` symbol tells the Java compiler that what it will find is an annotation. An annotation is syntactic metadata that can be applied to a class, a method, or to other elements of this language.

<sup>(12)</sup>Oracle Corporation (2013). Java Enterprise Edition 7 Specification APIs, <<https://docs.oracle.com/javaee/7/api/overview-summary.html>>.

The resources we include are some of those corresponding to the annotations that have appeared in Table 4. We have `Path`, `GET` and `POST` methods, `FormParam`, `PathParam` and `QueryParam` parameter types, and `Consumes` and `Produces`. We also have the `MediaType` class in the *core* subpackage, which allows identifying data types within `Consumes` and `Produces`.

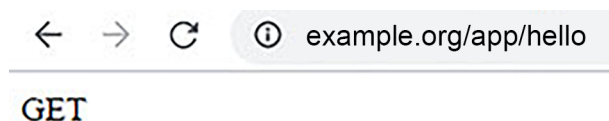
```
import javax.ws.rs.Consumes;
import javax.ws.rs.Produces;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.FormParam;
import javax.ws.rs.PathParam;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;

@Path("hello")
public class hello {
    /* Java methods that respond to GET and POST */
}
```

The first method we implement is `GET`. This method does not receive any data and produces `MediaType.TEXT_HTML` data type. The implementation of the method is very simple, since it returns an HTML page that displays the word `GET`. The URL to call this service could be `http://example.org/app/hello`. In Figure 5, we find an example of a response.

```
@GET
@Produces(MediaType.TEXT_HTML)
public String getHtml() {
    return "<html><head/><body>GET</body></html>";
}
```

Figure 5. Example of response of the `getHtml` operation in a browser

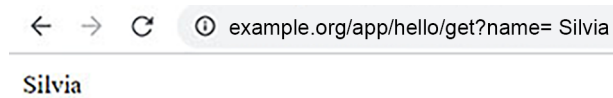


The second method we implement is also `GET`. It is not possible to have two methods that respond to `GET` with the same URL, so in this we have added the "get" path. In addition, we pass a parameter through the query section of the URL. This parameter is called "name" and is associated with a `String` type variable that is called `name`. It also produces data of the `MediaType.TEXT_HTML` type.

Implementing the method is very simple, as it returns an HTML page that displays the user's name that has been passed as a parameter. The URL to call this service could be `http://example.org/app/hello/get?name=Silvia`. In Figure 6, we find an example of a response.

```
@GET
@Path("/get")
@Produces(MediaType.TEXT_HTML)
public String getQuery(@QueryParam("name") String name) {
    return "<html><head><body>" + name + "</body></html>";
}
```

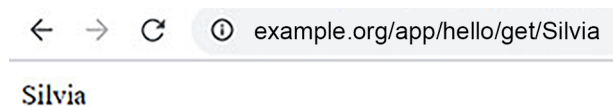
Figure 6. Example of a response to the `getQuery` operation in a browser



The third method we implement is also `GET`. In this case, as well as adding the path "get", we indicate that there is a parameter that is passed directly through the URL (`PathParam`), which gives "get/{name}" as the resulting path. This parameter is called `name`, and it associates itself with a `String` type variable, which is also called `name`. It also produces data of the `MediaType.TEXT_HTML` type. Implementing the method is very simple, as it returns a HTML page displaying the user's name, which was passed as a parameter. The URL to call this service could be `http://example.org/app/hello/get/Silvia`. In Figure 7, we find an example of a response.

```
@GET
@Path("/get/{name}")
@Produces(MediaType.TEXT_HTML)
public String getPath(@PathParam("name") String name) {
    return "<html><head><body>" + name + "</body></html>";
}
```

Figure 7. Example of `getPath` operation response in a browser



The last method we implement is `POST`. In this case, data are received in the request message (*Consumes*) and they are also returned (*Produces*). The data are passed as HTML form data (`application/x-www-form-urlencoded`) and returned as `MediaType.TEXT_HTML`. In the form, a parameter is passed as `FormParam`, called `name`. This parameter is associated with a `String` type variable that is also called `name`. Implementing the method is very simple, as it returns a HTML page that displays the user's name that has been passed as a parameter. The URL to call this service could be `http://example.org/app/hello`, but in this case, the request has to be sent in a HTTP `POST` message.

The server knows which method of the service it has to call from the URL and the HTTP method used, so it has the same URL access to the resource as the first GET method that we have seen.

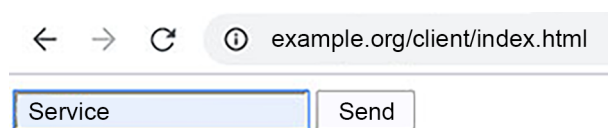
```
@POST
@Consumes("application/x-www-form-urlencoded")
@Produces(MediaType.TEXT_HTML)
public String postHtml(@FormParam("name") String name) {
    return "<html><head/><body>"+name+"</body></html>";
}
```

Finally, here is an extract of a HTML form that allows it to connect to the REST service that we have implemented in this section.

```
<form action="http://example.org/app/hello" method="POST"
    enctype="application/x-www-form-urlencoded">
    <input type="text" name="name"/>
    <input type="submit" value="Send"/>
```

In this form, we have to indicate the URL of the service (attribute *action*, `http://example.org/app/hello`), the sending method (method attribute, in this case it has to be POST) the type of encoding of the data (*enctype*, `application/x-www-form-urlencoded` attributes), and within the form, there must be a field that is called `name`. In this case, it is a text type field and the attribute *name* has the value `name`, which is the expected value for the service. If we do not define it exactly like this, the HTTP message sent will not match the one expected by the service, and the request will fail. In figures 8 and 9, we find the form and the response given by the service.

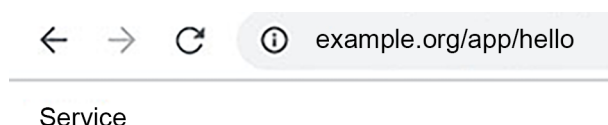
Figure 8. Form



← → ↻ ⓘ example.org/client/index.html

Service Send

Figure 9. Example of the postHtml operation response in a browser



← → ↻ ⓘ example.org/app/hello

Service

A possible error that could occur in the call with POST is that the form field is not called `name`. In this case, the service will not find the value of the corresponding variable, and will return a *null* value on the reply page. Another problem that could occur is that, if we put GET as a method instead of POST, the first GET operation of our service will be called and will show us the GET message instead of showing the data that we have written in the form. Another possible mistake is that we call the service with an undefined URL (for



example, `http://example.org/app/hello/hello3`) and this would give us a not found HTTP resource error (Code 404). Finally, if we call the service with a URL that does not correspond to the HTTP method, the error it would give us would be an unsupported method (Code 405).

## 2.1. Server-specific operations

We have shown a very simple example of how to implement a REST web service with Java language. In this section, we will see what else can be done with Java language to have a REST web service that responds to the other HTTP methods, what different types of data format we can have for requests and responses, etc.

It should be noted that what we have seen so far is independent of the type of server with which we work, since it is a Java specification that different manufacturers can implement in their own way.

### 2.1.1. Creating the service

To create a service with JAX-RS, a function that extends the `Application` abstract class (inside the `javax.ws.rs.core` Package) must be created. In this class, the root resource of the service must be defined and the classes responding to the different HTTP methods must be registered. For the service created in section 2, the class derived from `Application` could be as follows:

```
import java.util.Set;
import javax.ws.rs.core.Application;
@javax.ws.rs.ApplicationPath("app")
public class ApplicationConfig extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> resources = new java.util.HashSet<>();
        resources.add(hello.class);
        return resources;
    }
}
```

In this class, the root of the service (`app`) is defined. The `hello.class` class is added to the resources available in this service. This implements the service operations referred to in section 2.

### 2.1.2. Definition of operations

We have already seen in section 2 how the operations of a REST web service that uses JAX-RS are defined. Here, we will detail all the components that appear in it and their function, as well as the alternatives we can find.

```

@METHOD_HTTP
@Path("/URL_Access/{PATH_PARAM}")
@Consumes(TYPES_MIME_REQUEST)
@Produces(TYPES_MIME_RESPONSE)
public String nameMethod(@XParam("nameParam") String nameParam) {
    return <MESSAGE IN TYPES_MIME_RESPONSE FORMAT>;
}

```

The most general way to respond to a REST request with JAX-RS is to have an HTTP method annotation. Next, we can have an annotation with a specific `@Path` for this method, which may contain (or not, it is optional) one or more `@PathParam` type parameters, encircled "{}". They can appear in any position within the path, such as `{param1}/path/{param2}`. Each parameter of this type will appear as a parameter of the method and will be of the `@PathParam` type. We can also have parameters of the `@QueryParam` type (those sent in the URL's query section) or `@FormParam`, if the MIME type request is `application/x-www-form-urlencoded` or `multipart/form-data`. It should be noted that we could find all three types of parameters in the same method.

In addition to the aforementioned parameters, we can also have `@Consumes` and `@Produces` with MIME types that represent structured data, such as `application/xml` or `application/json`. To deal with structured data, Java support classes are required. They must have the elements corresponding to the data to be received. There are different alternatives for implementing these support classes, more or less automatically. This depends on the available libraries and the needs of the service to be implemented (store this data in files, databases, etc.). Next, we will present an example of an implementation of a method that receives and returns XML data and another that receives and returns JSON data. Both use the same Java support class, since they have the same structure.

We have named this kind of support `Item`. It contains a series of Java annotations for automatic processing of XML language. The annotations are `@XmlElement`, which identifies the root element of the document, `@XmlAccessorType`, which allows indicating how the connection between the elements of the Java class and the XML is made, and `@XmlRootElement`, which identifies the rest of the elements. There are two fields, one called `id`, of integer type and another called `name`, of the character string type. Before the `Item` class code, we see an XML file example and another JSON file with the data format and possible values.

```

<?xml version="1.0" encoding="UTF-8"?>
<item>
    <id>4</id>
    <name>Camera</name>
</item>
{

```

```
"id": 4,  
  "name": "Camera"  
}
```

The `Item` class contains the annotations already described and some support methods for modifying and querying the class fields.

```
import javax.xml.bind.annotation.XmlRootElement;  
import javax.xml.bind.annotation.XmlAccessorType;  
import javax.xml.bind.annotation.XmlAccessType;  
import javax.xml.bind.annotation.XmlElement;  
@XmlRootElement(name="item")  
@XmlAccessorType(XmlAccessType.FIELD)  
public class Item {  
    @XmlElement  
    private int id;  
    @XmlElement  
    private String name;  
    public int getId() {  
        return id;  
    }  
    public void setId(int id) {  
        this.id = id;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    @Override  
    public String toString(){  
        return "Id: " + this.id + " Name: " + this.name;  
    }  
}
```

Next, we give two examples of implementation with the HTTP PUT method, where we receive and send data in both XML and JSON format.

In the first method, the one that works with XML, it should be noted that we now receive an `Item` type parameter in the method, which automatically converts XML data to Java class fields. In this case, the result in XML is generated within the same method, but here we could add all kinds of XML data processing, storage in other media (file to disk, database, forwarding to other services, etc.).

```
@PUT
```

```

@Path("/xml")
@Consumes("application/xml")
@Produces("application/xml")
public String putXML(Item i) {
    String result;
    result = "<?xml version=\"1.0\" encoding=\"UTF-8\"?><item><id>"
        + (i.getId()+1) + "</id><name> "
        + i.getName() + " Perez</name></item>";
    return result;
}

```

In the second method, the one that works with JSON, the strategy is a little different. We receive a `java.io.InputStream` as a parameter. It contains an access channel to the JSON data sent by the client application. With this `InputStream`, using a Java class to support JSON (`com.google.gson.Gson`)<sup>(13)</sup>, we get an `Item` type object. We then modify it, to send the response to the client. The same thing that has been commented on for the XML case (databases, disk, etc.) could be done here, carrying out the corresponding treatment of the JSON data. There are many Java libraries that support the creation, reading, and writing of JSON data. This one has been chosen for its simplicity of use.

(13) **Gson Java Library**, <<https://github.com/google/gson>> [Date of access: March 2020]

```

@PUT
@Path("/json")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public String putJSON(java.io.InputStream is) {
    Item test;
    com.google.gson.Gson gson = new com.google.gson.Gson();
    java.io.BufferedReader reader = new java.io.BufferedReader
        (new java.io.InputStreamReader(is));
    test = gson.fromJson(reader, Item.class);
    test.setId (test.getId()+1);
    test.setName (test.getName() + " photo");
    return gson.toJson(test);
}

```

### 2.1.3. Sending the request

The request will be sent within a HTTP message. Below, we see an example of invoking the PUT methods, implemented in the previous section with XML and JSON, respectively. After the service request line, we find the HTTP headers. They identify the type of content that is sent, its length, the type of data we accept for the response and the name of the host to which we want to connect.

```
PUT /app/hello/xml HTTP/1.1
```

```
Accept: application/xml
Content-Type: application/xml
Content-Length: 95
Host: www.example.com
<?xml version="1.0" encoding="UTF-8"?>
<item>
  <id>4</id>
  <name>Camera</name>
</item>
```

The second request message contains the same data, but now in JSON format for the request and response.

```
PUT /app/hello/json HTTP/1.1
Accept: application/json
Content-Type: application/json
Content-Length: 35
Host: www.example.com
{
  "id": 4,
  "name": "Camera"
}
```

#### 2.1.4. Sending the response

The response will also be sent within an HTTP message. Below, there is an example of a response to the PUT method, with XML and JSON, respectively. After the response line of the service, we find the HTTP headers. They identify the type of content that is sent, and its length.

```
HTTP/1.1 200 OK
Content-Length: 95
Content-Type: application/xml
<?xml version="1.0" encoding="UTF-8"?>
<item>
  <id>5</id>
  <name>Camera photo</name>
</item>
```

The second request message contains the same data, but now in JSON format for the request and response.

```
HTTP/1.1 200 OK
Content-Length: 41
Content-Type: application/json
{
  "id": 5,
```

```
"name": "Camera photo"
}
```

## 2.2. Client-specific operations

Invoking a REST-based web service from a client application involves opening a HTTP connection with the service and sending a request in the format expected by the (HTTP method, headers, and parameters or data) service. We could do this with a connection via sockets or using support classes available in programming languages, such as Java

The client application can be of any type: a desktop application, a web application and even a mobile application. The requirements it must fulfil are to know how to send the HTTP request message with the relevant data, and then be able to understand the response sent by the server. The complexity of this client application will depend, above all, on the data-format sent in these requests and responses. Thus, the more complex the data-format (for example, sending files in XML or JSON formats), the more complicated it will be to process them. However, the existence of libraries greatly facilitates the implementation work, both for making connections and for processing data.

The rest of the section explains in a little more detail how to implement two client applications, a Java desktop application and a web application based on J2EE.

### 2.2.1. Programming a REST service client application

This section will explain how to implement a client test for our REST-based web service, but the strategy would be very similar for connecting us to another existing REST service. First, we will see how to do it from a web application with a servlet (`javax.servlet.http.HttpServlet`), and then, with a Java desktop application.

#### J2EE-based web application

A servlet is a Java class that can respond to HTTP methods made by a browser. In this sense, it is a bit like a REST web service, but limited to GET and POST, and geared towards an end user. It supports data reception of HTML forms and files. It also has access to the HTTP request that is received from the client (in this case, a browser) and the HTTP response that has to be returned, which must be in HTML language for the browser to understand it.

The declaration of a servlet type class with support for REST web services could be as follows:

```
import javax.servlet.ServletException;
```

#### Bibliographic reference

**March Hermo, M. I.** (2020). *Programación de sockets en Java*. Barcelona: UOC.

#### Bibliographic reference

**Java Community Process** (2006). JSR 88: Java™ EE (J2EE) Application Deployment, <<https://www.jcp.org/en/jsr/detail?id=88&showPrint>>

#### Bibliographic reference

**Oracle Corporation** (2017). Java Servlets Specification v4.0, <[https://javaee.github.io/servlet-spec/downloads/servlet-4.0/servlet-4\\_0\\_FINAL.pdf](https://javaee.github.io/servlet-spec/downloads/servlet-4.0/servlet-4_0_FINAL.pdf)>

```
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.core.MediaType;

@WebServlet(name = "Client", urlPatterns = {"/Client"})

public class Client extends HttpServlet {

    /* Servlet methods for contacting the service */

}
```

In the code that we see, there is the importation for the necessary Java classes and the class declaration, which derives from `HttpServlet`. We also find a Java annotation, indicating that this is a servlet (`@WebServlet`) and the URL to access it from the web application.

```
/* Extract from the connection code with the server */
Client client;
String REST_URL = "http://www.example.org/app/hello";
String result;
try {
    client = ClientBuilder.newClient();
    result = client.target(REST_URL)
        .path("json")
        .request(MediaType.APPLICATION_JSON)
        .put(Entity.json("{\"id\":4,\"name\":\"camera\"}"),
            String.class);
}
```

## Java desktop application

The desktop application code to connect to a REST-based web service is very similar to the code we implemented from a web application. The most important difference is that we have to create a Java class with the `public static void main (String [] args)` method in order to run it.

It should be noted that the only problem we can find in this case is to have all the Java libraries that support a REST-based web service client application, since the library that contains the `javax.ws.rs.client` classes has quite a few dependencies with other Java libraries that any programming environment can solve without too much trouble. We are not putting any specific version, since the Java language evolves so quickly that they would soon become obsolete.

```
/* Code to make the connection from a Java desktop application */
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.core.MediaType;

public class ClientJavaREST {

    public static void main(String[] args) {

        Client client;

        String REST_SERVICE_URL = "http://www.example.org/app/hello";

        String result;

        try {

            client = ClientBuilder.newClient();

            result = client.target(REST_SERVICE_URL)

                .path("json")

                .request(MediaType.APPLICATION_JSON)

                .put(Entity.json("{\"id\":3,\"name\":\"camera\"}"),

                    String.class);

            /* We show the result. We have had to receive data
            in JSON format, that we could process appropriately */
            System.out.println (result);

        } catch (Exception e)

        {

            System.out.println (e.getMessage());

        }

    }

}
```



### 3. REST services with other programming languages (Node.js, Python)

#### 3.1. Node. Js

Node.js is an open source server environment that runs on different operating systems and uses Javascript language on the server (the usual use of Javascript is on the client side, included in the browser).

To be able to use it as a REST server, we must use what is known as the MEAN toolstack, which includes the MongoDB, ExpressJS, Angular and Node.js (MEAN) tools.<sup>14</sup> All these tools are based on Javascript language. At a minimum, we need to have Node.js and ExpressJS, but to have a complete REST service it is advisable to have them all. Below, we briefly describe them.

MongoDB is a non-relational database that allows storing data in JSON format.

ExpressJS allows Node.js to behave like a REST server. It is a flexible environment that provides a whole series of features to support web and mobile applications, including the creation of APIs.

Angular allows developing web, mobile and desktop applications using Javascript language.

In the following subsection, we explain in a little more detail how to implement a basic REST-based web service with Node.js.

##### 3.1.1. Development of a REST API with Node.js and ExpressJS

To develop a REST service with Node.js, it must be installed from its official website. Once installed, ExpressJS must also be installed, to be able to offer the different operations via the web.

Once these two tools are installed, implementing a basic API is quite straightforward.

The first thing to do is to declare the dependencies that our API will have. This can be done with the following Javascript code:

```
/* Declaration of dependencies and variable app */
var express = require('express');
var bodyParser = require('body-parser');
var app = express();
```

#### About Node. Js

OpenJS Foundation, Node. Js, <<https://nodejs.org/es/>>. [Date of access: March 2020].

<sup>(14)</sup> IBM, MEAN (MongoDB ExpressJS Angular Node.js) stack explained, <<https://www.ibm.com/cloud/learn/mean-stack-explained>> [Date of access: March 2020]

#### References

MongoDB Inc., MongoDB, <<https://www.mongodb.com/es/>>. [Date of access: March 2020].  
StrongLoop, Inc., ExpressJS, <<https://expressjs.com/es/>>. [Date of access: March 2020].  
Google, Angular, <<https://angular.io/>>. [Date of access: March 2020].

```
app.use(bodyParser.urlencoded({ extended: true }));  
app.use(bodyParser.json());
```

Here it is indicated that there are two dependencies, `express` and `body-parser`, which we will use later. Then, we declare the `app` variable which will be the web application that ExpressJS will use.

In the following piece of code, we see that the application uses some features of `body-parser`. In addition, it declares the `port` variable which defines the port where this server will listen. If no other port has been defined in the configuration files, the port to be used is the 8080. Then, a constant that will contain the router that ExpressJS provides is declared. With this router, the different operations provided by the service are defined.

The first one is to reply to GET in the base URL of the service and return the data `{"message": "API responds to GET!"}`. The next one is to reply to POST in the base URL of the service and return the data `{"message": "API responds to POST!"}`.

```
app.use(bodyParser.urlencoded({ extended: true }));  
app.use(bodyParser.json());  
var port = process.env.PORT || 8080;  
const router = express.Router();  
router.get('/', function(req, res) {  
  res.json({ message: 'API responds to GET!' });  
});  
router.post('/', function(req, res) {  
  res.json({ message: 'API responds a POST!' });  
  res.end("end");  
});
```

Below, we see an operation that this API offers. Given a letter, it returns the same uppercase letter as a result. If it is already capitalized, it returns the same letter that the user typed. The access URL (which has to be concatenated to the URL of the service) has the form `/letters/a` and the result it returns is `{"result": "A"}`.

Finally, the web application is launched in the URL `/api` that will listen on the port indicated above. Thus, to access the functionality of the letters operation with this application, we would have to use the following URL: `http://example.org/api/letters/a`. The base URL of the service would be `http://example.org/api` and it would give a different response to the GET and POST methods, as we have seen above.

```
router.route('/letters/:letter').get((req, res) => {  
  res.json({result: req.params.letter.toUpperCase()})  
})
```

```
});  
app.use('/api', router);  
app.listen(port);  
console.log('Listening to the port ' + port);
```

The service can be launched from a command line, indicating the name of the node command and then the name of the file with extension .js where we have our code. We can see an example just below.

```
line_commands> node name_file.js
```

### Full REST service code with Node. Js

```
/* Code for making a basic REST API with Node.js */  
var express = require('express');  
var bodyParser = require('body-parser');  
var app = express();  
app.use(bodyParser.urlencoded({ extended: true }));  
app.use(bodyParser.json());  
var port = process.env.PORT || 8080;  
const router = express.Router();  
router.get('/', function(req, res) {  
    res.json({ message: 'API responds to GET!' });  
});  
router.post('/', function(req, res) {  
    res.json({ message: 'API responds to POST!' });  
    res.end("end");  
});  
router.route('/letters/:letter').get((req, res) => {  
    res.json({result: req.params.letter.toUpperCase()})  
});  
app.use('/api', router);  
app.listen(port);  
console.log('Listening to the port ' + port);
```

## 3.2. Python

Python is an interpreted programming language (it needs an interpreter to be able to run). It allows deploying many types of applications, including REST-based services. To do this, it requires a web environment, such as Flask or Django. In the following examples, we will use Flask as a web support.

Flask is a web *framework* written in Python language. It is called a microframework because it does not need extra tools or libraries for its operation. Even though, extensions can be easily added to provide extra functionality.

Django is another web *framework* written in Python. It is free and open source, and follows the Model-Template-View (MTV) architectural model.

In the following subsection, we explain in greater detail how to implement a basic REST-based web service with Python.

### 3.2.1. Development of a REST API with Python and Flask

To develop a REST service with Python, it has to be installed from its official website. Once installed, Flask must also be installed to be able to offer the different operations via the web.

Once these two tools are installed, implementing a basic API is quite straightforward.

The first thing to do is to declare the dependencies that our API will have. This can be done with the following Python code:

```
from flask import Flask, jsonify, request
app = Flask(__name__)
accounts = [
    {'name': "Anna", 'balance': 450.0},
    {'name': "Pau", 'balance': 250.0},
]
```

The dependencies we have are `Flask`, `jsonify` and `request`. All these dependencies come from `Flask`. This is indicated in `from Flask`. Then we declare the `app` variable, which will contain our web application. Finally, we declare the `accounts` variable, which will contain the details of some bank accounts in JSON format.

Next, we declare the methods of our API. We will first implement the functions that respond to GET. In this case we have two, one that will return all the accounts we have in our application and another that will allow us to know the data of an account from its identifier.

#### References

**Python Software Foundation**, Python. [online]: <<https://www.python.org/>> [Date of access: March 2020]

**Pallets**, *Flask*. [online]: <<https://flask.palletsprojects.com/en/1.1.x/>> [Date of access: March 2020]

**Django Software Foundation**, *Django*. [online]: <<https://www.djangoproject.com/>> [Date of access: March 2020]

**Django Software Foundation**, *Model-Template-View (MTV) architectural model in Django*. [online]: <<https://docs.djangoproject.com/en/3.0/faq/general/>> [Date of access: March 2020]

To define operations, we use the previously declared `app` variable and we call the `route` method, where we define the access URL to the operation and the HTTP method to which it will respond. In the first case, the URL is `/accounts`, and in the second case, the URL is `/accounts/<id>` where `id` is a parameter of the URL that indicates the identifier of the account we want to query. Then, the code of the operation has to be implemented, which returns the data requested by the user.

```
@app.route("/accounts", methods=["GET"])
def getAccounts():
    return jsonify(accounts)

@app.route("/accounts/<id>", methods=["GET"])
def getAccount(id):
    id = int(id) - 1
    return jsonify(accounts[id])
```

The last operation responds to the POST method and allows us to register a new account. The access URL will be `/account` and the data will be passed in JSON format to the body of the HTTP message. Finally, the web application is launched on port 8080.

```
@app.route("/account", methods=["POST"])
def addAccount():
    name = request.json['name']
    balance = request.json['balance']
    data = {'name': name, 'balance': balance}
    accounts.append(data)
    return jsonify(data)

if __name__ == '__main__':
    app.run(port=8080)
```

The service can be launched from a command line, indicating the name of the `python3` command (corresponding to Python version 3) and then the file name with the extension `.py` where we have our code. We can see an example just below.

```
line_commands> python3 name_file.py
```

## Python Full Service Code

```
/* Code to make a basic API REST with Python */
from flask import Flask, jsonify, request
app = Flask(__name__)
accounts = [
    {'name': "Anna", 'balance': 450.0},
    {'name': "Pau", 'balance': 250.0},
]
```

```
@app.route("/accounts", methods=["GET"])
def getAccounts():
    return jsonify(accounts)

@app.route("/accounts/<id>", methods=["GET"])
def getAccount(id):
    id = int(id) - 1
    return jsonify(accounts[id])

@app.route("/account", methods=["POST"])
def addAccount():
    name = request.json['name']
    balance = request.json['balance']
    data = {'name': name, 'balance': balance}
    accounts.append(data)
    return jsonify(data)

if __name__ == '__main__':
    app.run(port=8080)
```

## Summary

The module has introduced web services based on REST (REpresentational State Transfer), which define an architecture of how to access and manipulate remote resources, using the HTTP methods (POST, GET, PUT, PATCH and DELETE).

We have also seen the comparison between web services based on SOAP and REST, focusing on the characteristics of each of them, and explaining some principal differences between them.

Finally, we have discussed how to develop web services in Java language, also giving some guidelines on how it could be done with other programming languages, such as Node.js and Python.





## Bibliography

**Django Software Foundation.** *Django*. [online]: <<https://www.djangoproject.com/>> [Date of access: March 2020].

**Django Software Foundation.** *Model-Template-View (MTV) architectural model in Django*. [online]: <<https://docs.djangoproject.com/en/3.0/faq/general/>> [Date of access: March 2020].

**European Computer Manufacturers Association (ECMA)** (2017). The JSON (JavaScript Object Notation) Data Interchange Syntax ECMA – 404 Standard. [online]: <<http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>>.

**Fielding, R.** (2000). *Architectural Styles and the Design of Network-based Software Architectures*. [online]: <[https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding\\_dissertation.pdf](https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf)>.

**Google.** *Angular*. [online]: <<https://angular.io/>> [Date of access: March 2020].

**Gson Java library.** [online]: <<https://github.com/google/gson>> [Date of access: March 2020].

**IBM.** *MEAN (MongoDB Express Angular Node) stack explained*. [online]: <<https://www.ibm.com/cloud/learn/mean-stack-explained>> [Date of access: March 2020].

**Internet Engineering Task Force (IETF)** (2005). Uniform Resource Identifier (URI): Generic Syntax, RFC 3986. [online]: <<https://tools.ietf.org/html/rfc3986>>.

**Internet Engineering Task Force (IETF)** (2014). Hypertext Transfer Protocol (HTTP/1.1), RFC 7230 a 7235. [online]: <<https://tools.ietf.org/html/rfc7230>> and <<https://tools.ietf.org/html/rfc7235>>.

**Internet Engineering Task Force (IETF)** (2015). Returning Values from Forms: multipart/form-data, RFC 7578. [online]: <<https://tools.ietf.org/html/rfc7578>>.

**Internet Engineering Task Force (IETF)** (2018). The Transport Layer Security (TLS) Protocol Version 1.3. [online]: <<https://tools.ietf.org/html/rfc8446>>

**Introducing JSON.** [online]: <<https://www.json.org/json-en.html>> [Date of access: March 2020].

**Java Community Process** (2006). JSR 88: Java™ EE (J2EE) Application Deployment. [online]: <<https://www.jcp.org/en/jsr/detail?id=88&showPrint>>.

**March Hermo, M. I.** (2020). *Java Socket Programming*. Barcelona: UOC.

**MongoDB Inc.** *Mongodb*. [online]: <<https://www.mongodb.com/es>> [Date of access: March 2020].

**OpenJS Foundation.** *Node.js* [online]: <<https://nodejs.org/es/>> [Date of access: March 2020].

**Oracle Corporation** (2013). Java Enterprise Edition 7 Specification APIs. [online]: <<https://docs.oracle.com/javase/7/api/overview-summary.html>>

**Oracle Corporation** (2014). *The Java Tutorial, Annotations*. [online]: <<https://docs.oracle.com/javase/tutorial/java/annotations/index.html>>.

**Oracle Corporation** (2017). Java Servlets Specification v4.0. [online]: <[https://javaee.github.io/servlet-spec/downloads/servlet-4.0/servlet-4\\_0\\_FINAL.pdf](https://javaee.github.io/servlet-spec/downloads/servlet-4.0/servlet-4_0_FINAL.pdf)>.

**Organization for the Advancement of Structured Information Standards (OASIS)** (2006). Web Services Security: SOAP Message Security 1.1 (WS-Security). [online]: <<https://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf>>.

**Pallets.** *Flask*. [online]: <<https://flask.palletsprojects.com/en/1.1.x/>> [Date of access: March 2020].

**Python Software Foundation.** *Python*. [online]: <<https://www.python.org/>> [Date of access: March 2020].

**StrongLoop, Inc.** *ExpressJS*. [online]: <<https://expressjs.com/es/>> [Date of access: March 2020].

**Sun Microsystems** (2008). JAX-RS: Java™ API for RESTful Web Services. [online]: <<https://download.oracle.com/otn-pub/jcp/jaxrs-1.0-fr-eval-oth-JSpec/jaxrs-1.0-final-spec.pdf>>.

**World Wide Web Consortium (W3C)**. [online]: <<https://www.w3.org/>> [Date of access: March 2020].

**World Wide Web Consortium (W3C)** (2006). Extensible Markup Language (XML) 1.1 (Second Edition). [online]: <<https://www.w3.org/TR/xml11>>.

**World Wide Web Consortium (W3C)** (2007). Simple Object Access Protocol (SOAP) Version 1.2. [online]: <<https://www.w3.org/TR/soap12/>>.

**World Wide Web Consortium (W3C)** (2007). SOAP Version 1.2 Part 2: Adjuncts (Second Edition). [online]: <<https://www.w3.org/TR/soap12-part2/>>.

**World Wide Web Consortium (W3C)** (2007). Web Services Description Language (WSDL) Version 2.0. [online]: <<https://www.w3.org/TR/wsdl20/>>.

**World Wide Web Consortium (W3C)** (2012). XML Schema Definition Language (XSD) 1.1. [online]: <<https://www.w3.org/TR/xmlschema11-1/>>.