Introducton to Data Engineering 10

Paul Blondel

UTSEUS, Shanghai

June 1st, 2021

PAUL BLONDEL

PAUL BLONDEL

1. Recall
2. API
3. REST API
4. Flask
5. Streamlit

PAUL BLONDEL

We have seen together how to **store**, **manage** and **process** data...

Today, we will see how to make your data available to others, we will see:

- How to **make your own API** to share data with flask
- How to **make an application** with streamlit

PAUL BLONDEL

1. Recall
2. API
3. REST API
4. Flask
5. Streamlit

PAUL BLONDEL

What is an API?

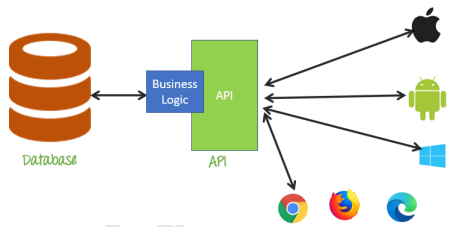
An API is a **set of definitions and protocols facilitating the creation and integration of software applications**.
API = "**Application Programming Interface**"

APIs...

- allow your product or service to **communicate with other products and services without knowing implementation details**
- allow to **save time and money** (simplify development)
- give **flexibility**, **simplify design**, **administration**, and **empower** API users to innovate
- are perfect to **make available your data** in a controlled and safe way

PAUL BLONDEL

Users **do not need to worry about how to query the database and knowing the business logic** to transforming the data, they just need to know how to call the API to get the data:



There are mainly **four main types of APIs**:

- **Open APIs**: **Publicly available** APIs, there are o restriction to use them (called Public APIs)
- **Partner APIs**: Specific rights or licenses permit to access this type of API, they are **not available to the public**
- **Internal APIs**: Internal or private APIs developed by companies to use in their internal systems, to **enhance work productivity and reduce dependency between software**

When dealing with APIs you will frequently see these terms:

- **HTTP (Hypertext Transfer Protocol):** Protocol permitting communicating data on the web. It implements a number of methods (two most common: GET and POST to push new data to a server
- **URL (Uniform Resource Locator):** An address for a resource on the web. A URL consists of a protocol (http://), a domain (website.com), and optional path (/about). A URL describes the location of a specific resource, such as a web page
- **JSON (JavaScript Object Notation):** A text-based data storage format that is **designed to be easy to read for both humans and machines**. JSON is generally the most common format for returning data through an API, XML being the second most common
- **REST (REpresentational State Transfer):** A philosophy that **describes some best practices for implementing APIs**. APIs designed with some or all of these principles in mind are called REST APIs

The book distributor example:

- But: Distributor may want to provide an application to check the availability of the books
- Remark: making an application might be expensive and time-consuming (and require ongoing maintenance)
- Alternatively: **the distributor can just provide an API** to check inventory availability

Opting for an API in this case has numerous advantages:

1. Customers have the **ability to centralize their inventory** information by accessing the API
2. Distributor can change the internal system of the API and **not impact the customer experience**
3. It is **easy for others to develop applications plugged to it**, easy to spread data to other systems

In this course we are interested in **remote APIs**:

- Remote APIs: designed to **interact via a communication network**
- Resources manipulated by the **API are not located on the computer making the request**
- Most remote APIs are designed **based on Web standards**
- Not all remote APIs are Web APIs, but it can be assumed that all Web APIs are remote
- Web APIs typically use the **HTTP protocol**
- Response messages are most often in the form of **XML** or **JSON**

There are two main way to build remote APIs:

- **SOAP (Simple Objects Access Protocol):**
  - APIs designed according to the SOAP protocol use the XML format for their messages and receive requests via HTTP or SMTP
  - SOAP aims to simplify the exchange of information between applications that run in different environments or that were written in different languages
- **REST (Representational State Transfer)**[1]**:**
  - Another attempt of API standardization
  - Web APIs meeting all the constraints of the REST architecture are called RESTful APIs
  - REST is an architectural style (contrary to SOAP, which is a protocol)

[1] In this course we will only talk about this type of API

PAUL BLONDEL

List of principles of a RESTful API:

1. **Client-server**: Composed of clients servers, and resources and it processes requests via the HTTP protocol
2. **Stateless**: Client content is never stored on the server between requests (session state information is stored on the client)
3. **Cacheable**: Caching brings performance improvement for the client-side and reduce the load
4. **Layered system**: Additional layers can perform additional functions: load balancing, cache sharing or security
5. **Code on demand (optional)**: Can extend the functionality of a client by returning executable code (permitted)
6. **Uniform interface**: Resources should be accessible through a **common approach** and modified using a **consistent approach**

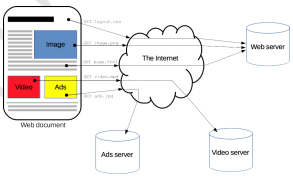
REST or RESTFUL?

RESTful services means it **follows all the above principles**
REST based services **follow some of the above principles and not all**

- REST APIs permit all possible **CRUD** operations:
  - **C**reate
  - **R**etrieve
  - **U**pdate
  - **D**elete
- REST guidelines suggest...
  - ...**a specific HTTP method**
  - ...for **a particular type of API call**
- Technically possible to violate the guidelines, but **highly discouraged**

What is **HTTP**?

- HTTP is a protocol allowing the **fetching of resources**, such as HTML documents
- At the foundation of any data exchange on the Web, it's a **client-server protocol**
- Requests are **initiated by the recipient**, usually the Web browser
- Full document **rebuilt from different sub-documents** (text, images, videos, etc.)



How **HTTP Communication between clients and servers** is handled:

- A client sends a HTTP request to the web
- A web server receives the request
- The server runs an application to process the request
- The server returns an HTTP response (output) to the browser
- The client (the browser) receives the response

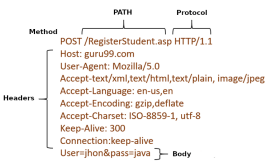
HTTP and REST APIs

REST APIs use HTTP protocol to transmit requests and get results but:

- not necessarily through the Web
- not necessarily from a Web browser

A request to fetch content from a server contains:

- A **Path** to requested web page
- A **Protocol Version**
- A **Method** (GET, POST, other)
- **Headers** containing info about the client
- (Eventually) A **Body** containing data to send



The response:

When the server responds, it returns a **requested content**, or a **result**, plus a **code**.
(404 error code: content not found)

There are many others code:

- Code 500: an error occurred on the server
- Code 401: access to this web page is not allowed
- Code 400: client's request is badly formed
- Code 200: everything went well
- ...

HTTP Method	CRUD	Entire Collection (e.g. /users)	Specific Item (e.g. /users/123)
POST	Create	201 (Created)	Avoid using POST on single resource
GET	Read	200 (OK) + list of users	200 (OK) + a user or 404 (Not Found)
PUT	Update Replace	405 (Should not be allowed)	200 (OK), 204 (No Content) or 404 (Not Found)
PATCH	Partial Update	405 (Should not be allowed)	200 (OK), 204 (No Content) or 404 (Not Found)
DELETE	Delete	405 (Should not be allowed)	200 (OK) or 404 (Not Found)

What is Flask?

- Flask is a **web framework** for Python
  - it provides functionality for **building web applications**
  - it can **manage HTTP requests** and **render templates**
- Flask is easy: it can **save time and effort** for programmers
- One can easily build a HTTP REST/RESTful API with it

Let's analyze this address before starting:
"http://www.siteduzero.com/forum-81-407-langage-python.html"

- "http" is the **protocol**
- "www.siteduzero.com": is the **domain name**
- "/forum-81-407-language-python.html": is the path to the requested content (also called **route**)
- The full address is often called a "**endpoint**"

In this course the HTTP server will be hosted on our own computer:

- The "domain name" of our own computer is called "**localhost**" by convention
- With Flask, the default port is 5000
- The route "hello" is thus accessible through the URL: "http://loclahost/hello"

Let's begin with a simple code example (we will name our file "hello.py"):

```
from flask import Flask
app = Flask(__name_)

@app.route('/hello/', methods=['GET', 'POST'])
def welcome():
    return "Hello World!"

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

1. Execute the code by simply typing: "**python hello.py**" in the terminal
2. You should see something like this:
   `* Running on http://localhost:5000/ (Press CTRL+C to quit)`
3. Then open your web browser and go to **http://localhost:5000/hello/**
4. You should see a "Hello World!" appearing...

PAUL BLONDEL

Let's analysis this code line-by-line:

- **from flask import Flask** → Import the Flask class
- **app = Flask(__name__)** → Create an instance of the class
- **@app.route('/hello/', methods=['GET', 'POST'])** → the route() decorator tell Flask what URL should trigger the function and the HTTP methods that are allowed (default is ['GET'])
- **if __name__ == '__main__':** → This line ensures that our Flask app runs only when it is executed as the main file and not when imported in some other file
- **app.run()** → Run the Flask application host specifies the server on which we want our flask application to run (default host value is localhost or 127.0.0.1)

Adding a route is easy using the following Python decorator before a function definition:

```
@app.route('/ROUTENAME/', methods=['GET'])
def my_function :
    ...
    return ...
end
```


JSON serializable output:

- The return value of a function in a Flask app should be a JSON object
- You can use **jsonify** to make your output a JSON
  - This returns a JSON output as a "**Response object**" with **application/json mime-type**

Example:

```
from flask import jsonify
@app.route('/person/')
def hello():
    return jsonify({'name':'Jimit',
                    'address':'India'})
This will send a JSON response like this:
{
  "address": "India",
  "name": "Jimit"
}
```


Example of a RESTful API:

Goal: An API to get book(s), add (a) book(s), update book(s) and remove (a) book(s) from a DB

We must have two routes:

- **/books**: to perform action on book collection: list books, add a book, etc
- **/books/id**: to perform action on a book item: show, update or delete

References

A similar example here:

- https://www.kite.com/blog/python/flask-restful-api-tutorial/

Official documentation for Flask 1.1:

- https://flask.palletsprojects.com/en/1.1.x/


```
@app.route('/books', methods = ['GET', 'POST'])
def booksFunction():
   if request.method == 'GET':
       return getBooks()
   elif request.method == 'POST':
       title = request.args.get('title', '')
       author = request.args.get('author', '')
       genre = request.args.get('genre', '')
       return makeANewBook(title, author, genre)

@app.route('/books/', methods = ['GET', 'PUT', 'DELETE'])
def bookFunctionId(id):
   if request.method == 'GET':
       return geABook(id)

   elif request.method == 'PUT':
       title = request.args.get('title', '')
       author = request.args.get('author', '')
       genre = request.args.get('genre', '')
       return updateABook(id,title, author,genre)

   elif request.method == 'DELETE':
       return deleteABook(id)
```

Recall

getBooks, makeANewBook, getABook, updateABook, deleteABook should **return a JSON** and a **HTTP status or code** (200, 404, 500, etc.)

Now, how to interrogate our API?

For example, to get all the list of all books we should do the following GET API call with curl:

```
$ curl -v http://localhost:5000/books
```

If you want to add a book to the database through the API and with curl:

```
$ curl -d '{"title":'Book',"author' :'Jean',"genre" :'science-fiction'}'
-H 'Content-Type: application/json' http://localhost:5000/books/
```

If you want to delete a book:

```
$ curl -X DELETE http://localhost:5000/books/9
```

If you want to get informaiton about a specific book:

```
$ curl -v http://localhost:5000/books/9
```

If you want to update the information of a book:

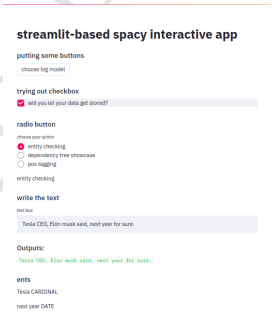
```
$ curl -d '{"title":'Book',"author' :'Jean-Jaques',"genre" :'science-fiction'}'
-H 'Content-Type: application/json' -X PUT http://localhost:8082/books/9
```

Note that we can also call the API using the Python requests module:

```
# !/usr/bin/python
import requests
response = requests.get('http://localhost:5000/books/10')
print(response.json())

# Create a new resource
data_1 = {'title':'titre 2', 'author' : 'Patrick', 'genre' : 'History'}
response = requests.post('http://localhost:5000/books', data = data_1)
print(response.json())

# List of books
response = requests.get('http://localhost:5000/books')
print(response.json())

# Update an existing resource
data_2 = {'title':'Titre 2', 'author' : 'Patrick', 'genre' : 'History'}
requests.put('http://localhost/books/454', data = data_2)
print(response.json())

response = requests.delete('http://localhost:5000/books/10')
print(response.json())
```


What is Steamlit ?

- The "**fastest way to build and share data apps**"
- Works with **Python** (plain Python code)
- Extremely **easy**
- Streamlit data apps can be created just by using the **streamlit module**
- Run your app in your terminal like a Python script: **streamlit run script_name.py**



Streamlit support different representations (seaborn, matplot, etc) and much can be written using the following functions:

- **st.title()**: Create a title element in the data app (useful method to give a proper heading to your app)
- **st.write()**: The "swiss army knife" of streamlit: you can use it to print or show or display elements in your app using this method
- **st.subheader()**: Create different sections in an app (create big and bold texts and displays them properly helps creating sections in your app)

Example of code:

```
import streamlit as st
import numpy as np
import pandas as pd
import altair as alt

st.title("The first basic testing app")
st.write("\frac{1}{2} does it support lateX?")
st.write("looks like it doesn't")
st.write("a normal string")
st.write("<b>food</b> is great here; checking if simple html works", unsafe_allow_html = True)
st.write("thanks, its a *good* day", unsafe_allow_html = True)
st.write("displaying that numbers can be represented")
st.write(1234)
st.write("showing that dataframes can be printed")
st.write(pd.DataFrame({"first_column":[10,20,30,40],"second_column":[105,405,905,1605]}))
st.write("we can show headers also")
st.header("it shows chart too")
```


Rest of the code:

```
df = pd.DataFrame(np.random.randn(200, 3),
                  columns=['a', 'b', 'c'])
c = alt.Chart(df).mark_circle().encode(x='a', y='b',size='c',color='c',tooltip=['a', 'b', 'c'])
st.write(c)
st.subheader("showing a codeblock using st.code")
code = """for i in range(20):
          print("this is a count of" + str(i))
       """
st.code(code,language = "python")
df = pd.DataFrame(np.random.randn(20,8),
                  columns = ["columns_"+str(i) for i in range(1,9)])
st.subheader("showing dataframe as table")
st.table(df)
```

Once you created your python file, run the Streamlit application like this:

```
$ streamlit run app.py
```

And open the application in your browser...

PAUL BLONDEL

It is also possible to add **checkboxes**, **radios**, **text inputs**, **buttons**, etc.

st.button(): take a label in input. It return a boolean value, if the button is clicked in the app then it returns True otherwise False.

```
if st.button("choose big model"):
    nlp = spacy.load("en_core_web_lg")
else:
    nlp = spacy.load("en_core_web_sm")
```


st.checkbox() is a similar to st.button(), it takes a label and returns a bool return value (but a box is showed in the UI):

```
st.subheader("trying out checkbox")
check = st.checkbox("will you let your data get stored?")
if check:
    print("fake notion! nothing recording now!")
else:
    pass
```


st.text_input(): returns the text written in the text box... It is a very powerful tool. see below how we implemented it:

```
st.subheader("write the text")
text = st.text_input("text box","Example: write here")
```

Official documentation

More information and content in the official Steamlit documentation : https://docs.streamlit.io/en/stable/

PAUL BLONDEL