Flask

Advanced Computer Programming

Prof. Luigi De Simone

Summary

- **Argomenti**
  - Web services, RESTful web services, REST e HTTP
  - Cenni HTML
  - Flask
    - Installazione
    - Route e Dynamic Route
    - Request e Response Object
  - Requests library

- **Riferimenti**
  - RESTful Web Services, L. Richardson e S. Ruby, O'Reilly
  - https://flask.palletsprojects.com/en/3.0.x/
  - https://flask.palletsprojects.com/en/3.0.x/api/
  - https://flask.palletsprojects.com/en/3.0.x/quickstart/
  - https://flask.palletsprojects.com/en/3.0.x/tutorial/
  - Miguel Grinberg, Flask Web Development – Developing web applications in Python, O'Reilly
  - https://requests.readthedocs.io/en/latest/

Web Service e REST

Web Service (WS)

- *"A Web service is a software system identified by a **URI** [RFC 2396], whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols"*

https://www.w3.org/TR/wsa-reqs/

Servizi

- Moduli software che espongono delle **funzionalità invocabili dai client**
- Il fornitore del servizio ne produce l'**implementazione**, e ne fornisce la ***descrizione***:
  - la descrizione include, per esempio, l'interfaccia del servizio.
- Un servizio può essere *riusato* ed è *componibile*
- Un servizio è tipicamente offerto ed invocabile dai client tramite Internet ed acceduto con protocolli standard:
  - per esempio, HTTP, URI, XML, SOAP;
  - in linea generale, la nozione di "web service" si riferisce alla possibilità di usare richieste HTTP per avviare l'esecuzione di un programma (da non confondere con i *web server*).

RESTful web services

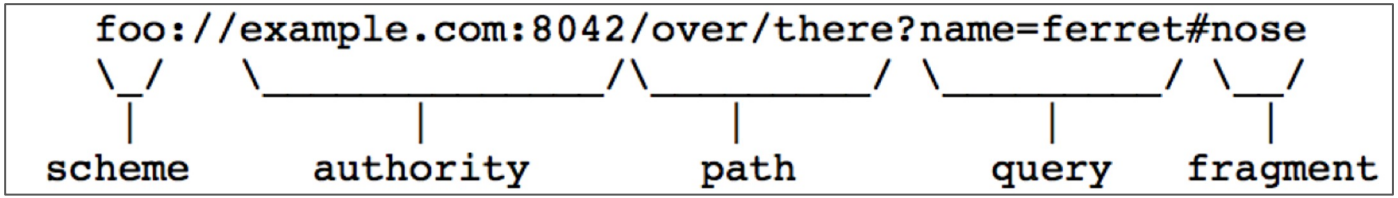
- **Representational State Transfer (REST)** è “uno stile architetturale che definisce gli attributi di qualità architetturale del World Wide Web, visto come un sistema ipermediale aperto, accoppiato lascamente, massicciamente distribuito e decentralizzato”.
- Rispetto ai meccanismi RPC il focus è sulle ***risorse*** e non sulle *procedure.*

REST: concetti chiave

- Il web service espone i propri *dati* e *funzionalità* tramite **risorse** identificate da **URI**:
  - **risorsa**: entità indirizzabile tramite Web, ovvero accessibile e trasferibile tra client e server.
- **Interfaccia uniforme**: i clienti interagiscono con le risorse tramite un insieme *fissato* di metodi:
  - per esempio, nel caso HTTP, essi sono tipicamente GET (read), POST (create), PUT (update), DELETE.
- **Stateless**: ciascun ciclo di richiesta-risposta rappresenta un'interazione *completa* tra client e server; non esiste il concetto di sessione.

Uniform Resource Identifier (URI)

- Si tratta di uno standard di Internet per l'identificazione delle risorse.



- Un **URI Template** specifica in che modo costruire e leggere una determinata URI.

- Template:

http://www.myservice.com/order/{oid}/item/{iid}

- Example URI:

http://www.myservice.com/order/XYZ/item/12345

REST su HTTP

- Lo scenario più comune è costituito dall'implementazione dell'architettura REST sul protocollo **HTTP**.

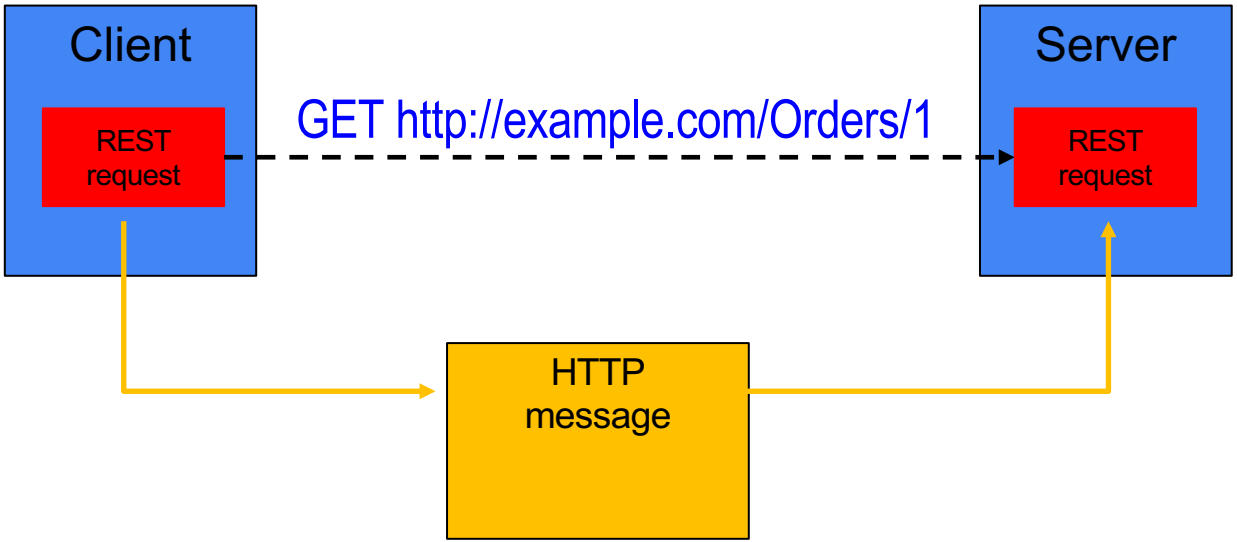
- **Hypertext Transfer Protocol (HTTP)** è un protocollo di livello applicativo per il trasferimento di pagine web:
  - ogni pagina web è identificata da un **URL (Uniform Resource Locator)**;
  - ogni oggetto all'interno di una pagina web (per es., sfondi, immagini, links, …) è identificato a sua volta da un URL;
  - prevede un insieme di **metodi** per la gestione delle risorse (per es., POST, GET, PUT, DELETE) che consentono di implementare in maniera diretta le tipiche operazioni **CRUD** (create, retrieve, update, delete).

HTTP methods

HTTP		SAFE	IDEM POTENT
POST	Create a sub resource	NO	NO
GET	Retrieve the *current state* of the resource	YES	YES
PUT	Initialize or update the state of a resource at the given URI	NO	YES
DELETE	Clear a resource, after the URI is no longer valid	NO	YES

- Un metodo HTTP è **safe** quando non altera lo stato del server, i.e., implica operazioni *read-only*
- Un metodo HTTP è **idempotente** quando richieste identiche producono lo stesso risultato

Comunicazione RESTful



Alcuni esempi

http://example.com/Orders/{oid}

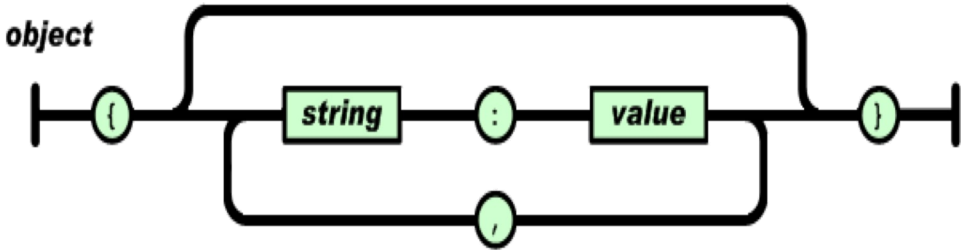
POST http://example.com/Orders/8	CREATE
GET http://example.com/Orders/1	READ
PUT http://example.com/Orders/12	UPDATE
DELETE http://example.com/Orders/12	DELETE

Richieste-risposte HTTP

Method	Entity-Body (o Representation) della richiesta	Entity-Body (o Representation) della risposta
GET	Di solito il client invia un entity-body/representation vuoto.	Il sever tipicamente restituisce nell' entity-body la rappresentazione della risorsa.
DELETE	Di solito il client invia un entity-body/representation vuoto.	Il sever può restituire un entity-body vuoto o con un messaggio di stato.
PUT	Di solito il client invia nell'entity-body la rappresentazione proposta per la risorsa.	Il sever può restituire un entity-body vuoto o un messaggio di stato o una copia della reppresentazione della risorsa.
POST	il client invia nell'entity-body la rappresentazione proposta per la risorsa.	Il sever può restituire un entity-body vuoto o un messaggio di stato o una copia della reppresentazione della risorsa.

XML e JSON

- Sono formati tipicamente usati per la rappresentazione dei dati scambiati tra client-servizi (external data representation) :
  - **XML** (Extensible Markup Language);
  - **JSON** (JavaScript Object Notation);
- Sono rappresentazioni testuali (a differenza, per esempio, della **serializzazione gRPC**, che invece è binaria).
- Un oggetto JSON è una serie non ordinata di nomi/valori ed è caratterizzato dalla seguente sintassi:



XML e JSON

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<DatabaseInventory>
  <DatabaseName>

<GlobalDatabaseName>production.cubicrace.com</
GlobalDatabaseName>
    <OracleSID>production</OracleSID>
    <Administrator EmailAlias="piyush"
Extension="6007">Piyush
Chordia</Administrator>
    <DatabaseAttributes Type="Production"
Version="9i" />
    <Comments>All new accounts need to be
approved.</Comments>
  </DatabaseName>
  <DatabaseName>

<GlobalDatabaseName>development.cubicrace.com<
/GlobalDatabaseName>
    <OracleSID>development</OracleSID>
    <Administrator EmailAlias="kalpana"
Extension="6008">Kalpana
Pagariya</Administrator>
    <DatabaseAttributes Type="Development"
Version="9i" />
  </DatabaseName>
</DatabaseInventory>
```

JSON

```
{
   DatabaseInventory: {
      DatabaseName: [
         {
            GlobalDatabaseName: "production.cubicrace.com",
            OracleSID: "production",
            Administrator: [
               {
                  EmailAlias: "piyush",
                  Extension: "6007",
                  value: "Piyush Chordia"
               }
            ],
            DatabaseAttributes: {
               Type: "Production",
               Version: "9i"
            },
            Comments: "All new accounts need to be approved."
         },
         {
            GlobalDatabaseName: "development.cubicrace.com",
            OracleSID: "development",
            Administrator: [
               {
                  EmailAlias: "kalpana",
                  Extension: "6008",
                  value: "Kalpana Pagariya"
               }
            ],
            DatabaseAttributes: {
               Type: "Development",
               Version: "9i"
            }
         }
      ]
   }
}
```

RPC vs REST

- Ciascun web service *RPC-style* espone un proprio “vocabolario” di metodi (funzioni con nomi differenti).
- Nel caso REST il vocabolario è fissato (sono i metodi HTTP):
  - tutti i servizi REST espongono quindi una stessa interfaccia di base
- Esempio gestione di *orders identificati* da un “id”:

RPC-style:	**RESTful:**
`insertOrder(id);`	`POST` *`Order/{id}`*
`getOrder(id);`	`GET` *`Order/{id}`*
`updateOrder(id);`	`PUT` *`Order/{id}`*
`deleteOrder(id);`	`DELETE` *`Order/{id}`*

Principali passi progettazione REST

- Identificare le risorse da esporre come servizi.
- Per ogni risorsa:
  - definire le URI;
  - esporre un subset *adeguato* dell'interfaccia uniforme (ossia ragionare su "cosa significa" fare GET, POST, etc. di certe risorse, e se/quali operazioni debbono essere consentite);
  - progettare le rappresentazioni fornite dal client (al server) o servite dal server (al client):
    - JSON, XML;
    - messaggi di stato;
    - …

	GET	PUT	POST	DELETE
/client	✓	✓	✓	✗
/book	✓	✓	✓	✓
/order	✓	?	✓	✗

Note sulle URI

- Meglio preferire nomi a verbi:
  - `DELETE /book/15` OK
  - `GET /book?isbn=15&action=delete` NO

- GET vs. POST:
  - GET è *read-only* (idempotente)
  - POST è *read-write* e può modificare lo stato della risorsa.

- POST vs. PUT:
  - POST: tipicamente usata per creare una risorsa *figlia* rispetto ad una padre;
  - PUT: usata per modificare risorse esistenti.

Cenni di HTML

Cenni di HTML

- HTML è l'acronimo di **HyperText Markup Language**
- Tecnologia che permette di specificare la struttura degli elementi visivi che compongono una web application
- Un documento HTML può essere descritto da un set di **HTML tags**
- Ogni HTML tag è caratterizzato da un opening ed un closing:

  <name>...</name>

- I contenuti sono solitamente inseriti tra i tag di apertura e chiusura, e possono essere testo o altri tag HTML

The Hello world example

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example</title>
    </head>
    <body>
        <p>This is an example of a simple HTML page with one paragraph.</p>
    </body>
</html>
```

Headings, Anchors e List

- **Heading levels**: <h1>, <h2>, <h3>, <h4>, <h5>, <h6> tags
  - Utilizzati solitamente per contenuti importanti all'interno della pagina

- **Anchor**: <a>
  - utilizzati per creare link
  - Utilizzati principalmente in ***hypertext*** dato che possono portare ad altre informazioni, anche all'interno della stessa pagina o presenti su altre pagine

- **List**: Permettono di specificare
  - **ordered** list (i.e., enumeration): `<ol>`
  - **unordered** list (i.e., bullet list): `<ul>`

```
<!doctype html>
<html>
        <head>
                <title>List Examples</title>
        </head>

        <body>
                <h1>List Examples!</h1>
                <!-- We'll wrap the links in an ul tag -->
                <ul>
                        <li>
                                Here is a <a href="http://www.google.com">link</a> to Google!
                        </li>
                        <li>
                                <a href="http://www.example.com">
                                        And this is a link that is a little longer </a> </li>
                        <li>
                                And here is a link to <a href="http://www.facebook.com"> www.facebook.com </a>
                        </li>
                </ul>
                <!-- We can also create an ordered list tag -->
                <h3>How to make an ordered list</h3>
                 <ol>
                        <li>Start by opening your ol tag</li>
                        <li>Then add several list items in li tags</li>
                        <li>Close your ol tag</li>
                </ol>
        </body>
</html>
```

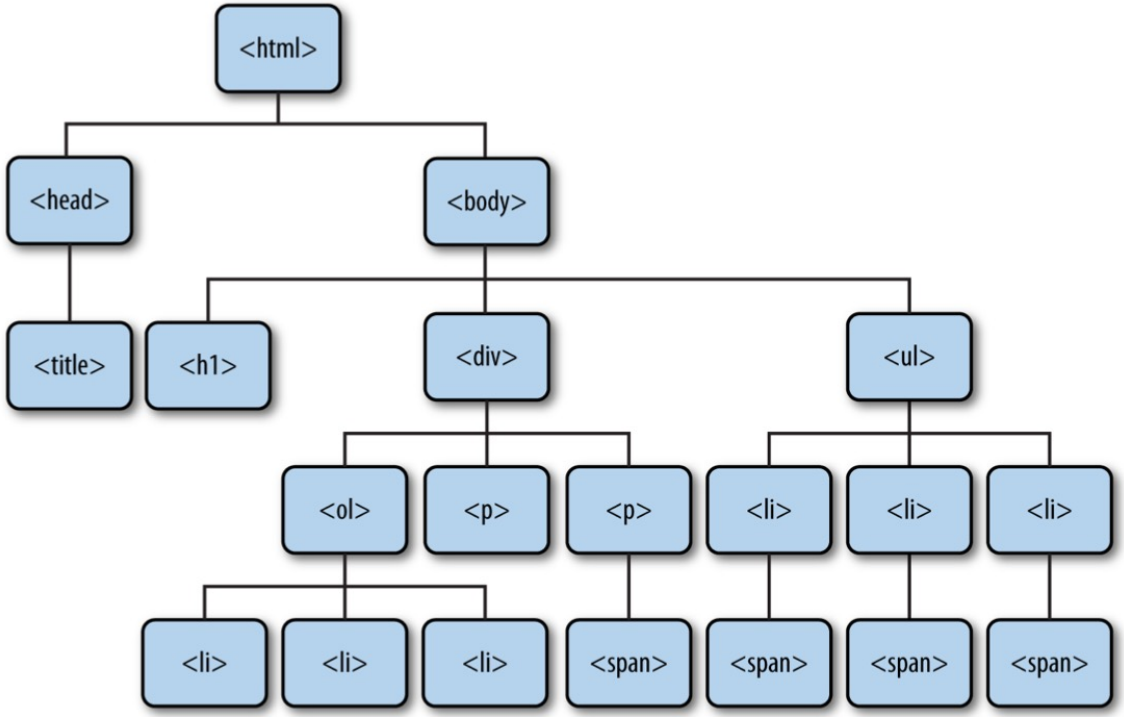
The Document Object Model and Trees

- I tag HTML definiscono una architettura gerarchica chiamata ***Document Object Model***, or DOM
- Il DOM è un modo di **rappresentazione degli oggetti** che può essere definito attraverso l'HTML e con il quale si può interagire attraverso scripting language, e.g., JavaScript
- I tag HTML definiscono i DOM *elements*, i quali sono le entità che vivono all'interno del DOM
- Tutti i tags possono essere sempre organizzati come un **tree**

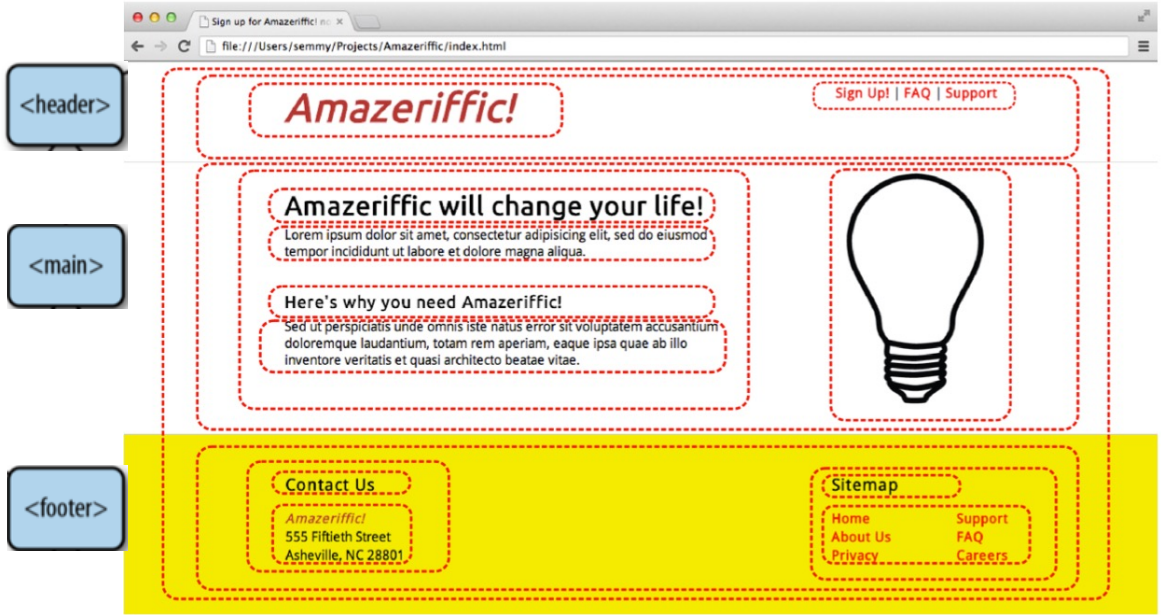
DOM Example

```
<!doctype html>
<html>
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <h1>Hello World!</h1>

    <div>
      <ol>
        <li>List Item</li>
        <li>List Item</li>
        <li>List Item</li>
      </ol>

      <p>This is a paragraph.</p>

      <p>This is a <span>second</span> paragraph.</p>

    </div>

    <ul>
      <li>List Item <span>1</span></li>
      <li>List Item <span>2</span></li>
      <li>List Item <span>3</span></li>
    </ul>
  </body>
</html>
```

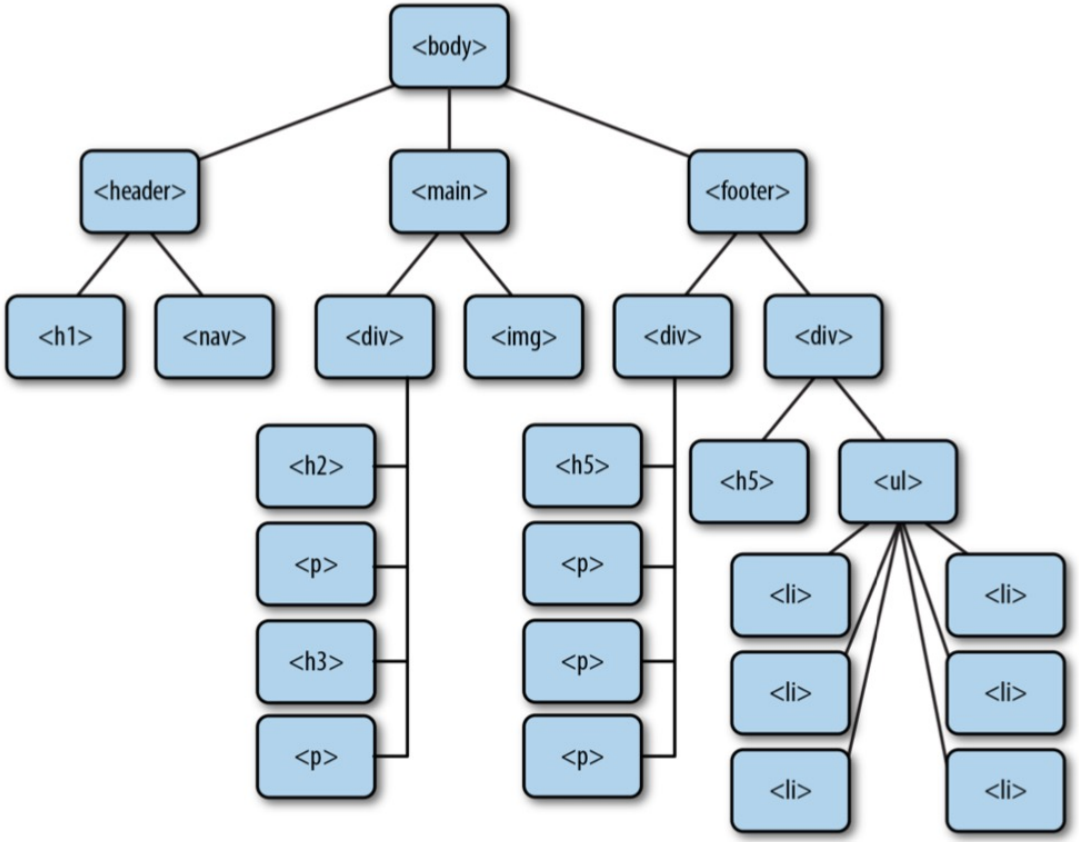
DOM Example



DOM elements relationships



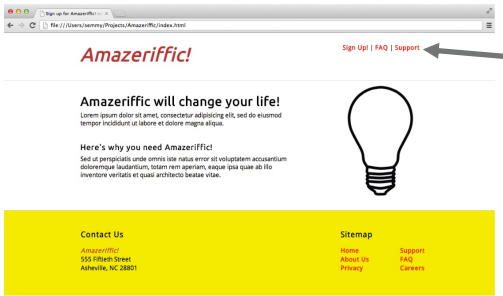
Amazerrific DOM



Header

```
<!doctype html>
<html>
  <head>
    <title>Amazeriffic</title>
  </head>

  <body>
    <header>
      <h1>Amazeriffic</h1>
    </header>

    <main>
    </main>

    <footer>
    </footer>
  </body>
</html>
```

- The upper-right corner of the page has a small **navigation** section with links to a *Sign Up* page, a *FAQ* page, and a *Support* page
- We can use the HTML navigation element **<nav>**
- Let's add it to our header. Note that the nav element contains several links that are separated by the | symbol
- we use the # symbol as a temporary placeholder for the link




```
<header>
      <h1>Amazeriffic</h1>
      <nav>
        <a href="#">Sign Up!</a> |
        <a href="#">FAQ</a> |
        <a href="#">Support</a>
      </nav>
</header>
```

Main

```
<h2>Amazeriffic will change your life!</h2>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit,
sed do
eiusmod tempor incididunt ut labore et dolore magna
aliqua.</p>
<h3>Here's why you need Amazeriffic</h3>
<ul>
          <li>It fits your lifestyle</li>
          <li>It's awesome</li>
          <li>It rocks your world</li>
</ul>
<img src="lightbulb.png" alt="an image of a lightbulb">
```

Footer

```
<footer>
        <div class="contact">
                <h5>Contact Us</h5>
                <p>Amazeriffic!</p>
                <p>555 Fiftieth Street</p>
                <p>Asheville, NC 28801</p>
        </div>
<div class="sitemap">
        <h5>Sitemap</h5>
        <ul>
                <li><a href="#">Home</a></li>
                <li><a href="#">About Us</a></li>
                <li><a href="#">Privacy</a></li>
                <li><a href="#">Support</a></li>
                <li><a href="#">FAQ</a></li>
                <li><a href="#">Careers</a></li>
        </ul>
</div>
</footer>
```

HTML tag Summary

Tag	Description
`<html>`	*The main container for an HTML document*
`<head>`	*Contains meta-information about the document*
`<body>`	*Contains the content that will be rendered in the browser*
`<header>`	*The header of the page*
`<h1>`	*Most important heading (only one per document)*
`<h2>`	*Second most important heading*
`<h3>`	*Third most important heading*
`<main>`	*The main content area of your document*
`<footer>`	*The footer content of your document*
`<a>`	*Anchor, a link to another document or a clickable element*
`<ul>`	*A list of things where order doesn't matter*
`<ol>`	*A list of things where order matters*
`<li>`	*An element of a list*
`<div>`	*A container for a substructure*

Il web framework Flask

Web framework

- Un web framework è un insieme di tecnologie utili per lo sviluppo di web app
- I web frameworks tipicamente forniscono:
  - **Routes**, che mappano le URLs a server file o funzioni
  - **Template**, che inseriscono automaticamente server-side data in pagine HTML
  - Meccanismi di **autenticazione** e **autorizzazione**
  - **Sessioni**, che tengono traccia di un utente durante una singola visita
  - **...**

Che cos'è Flask

- Flask è web application framework WSGI leggero (*microframework*)
  - Progettato per consentire la realizzazione rapida e semplice di applicazioni web, con la possibilità di scalare verso applicazioni complesse
  - Il core mette a disposizione tutti i servizi di base, che possono essere estesi attraverso il meccanismo delle estensioni
- Nato come *wrapper* di *Werkzeug* and *Jinja2*, ed è diventato uno dei web application framework per Python più utilizzati

Werkzeug e Jinja2

- **Werkzeug** mette a disposizione le funzionalità di routing, debugging, e Web Server Gateway Interface (WSGI)
  - WSGI (pronunciato whiskey) è una specifica che descrive come un web server comunica con le applicazioni web, e come tali applicazioni possono essere interconnesse per processare una richiesta

- **Jinja2** mette a disposizione il supporto ai *template* (maggiori dettagli più avanti)

Installazione

- E’ possibile installare *Flask*, *Werkzeug* e *Jinja2* attraverso il comando:

```
pip install Flask
```

- Per verificare se l’operazione è andata a buon fine:
  - Aprire il terminale Python e provare il seguente import

```
>>> import flask
```

  - Se non appare alcun errore, l’installazione è andata a buon fine

Hello World in Flask

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def index():
    return "<h1>Hello, World!</h1>"

@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, %s!</h1>' % name

if __name__ == "__main__":
    app.run(debug=True)
```

Inizializzazione

Routes

Server startup

Inizializzazione

- Tutte le applicazioni Flask devono creare una ***application instance***:

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

- L’argomento al costruttore Flask è il nome del main module o del package dell’applicazione

Routes

- I client web (ad es. un browser) di una web application inviano **richieste** al web server, il quale le **instrada** verso l'applicazione Flask

- L'applicazione Flask necessita di sapere quale codice eseguire per ogni URL richiesta
  - Necessità di mantenere il **mapping tra le URL e le funzioni Python**

- **Route:** mapping tra una URL e la funzione Python che la gestisce

Routes e View functions

- Flask permette di definire una rotta attraverso il *decoratore* **`app.route`**

```
@app.route("/")
def index():
    return "<h1>Hello World!</h1>"
```

- Il decoratore registra la funzione definita subito dopo (***index( )*** nell'esempio) come **gestore della URL** specificata (**"/"** nell'esempio)
  - La funzione prende il nome di **view function**
  - Ad ogni richiesta ricevuta sulla URL specificata, sarà invocata la *view function*
  - La funzione può ritornare come risposta codice HTML o formati più complessi (ad es. JSON); la risposta sarà fornita al client

Dynamic Routes

- Flask permette di definire dynamic route, cioè route che contengono porzioni dinamiche (variabili) all'interno della URL
  - porzioni che possono cambiare ad ogni richiesta o essere assenti

```
@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, %s!</h1>' % name
```

- Nell'esempio:
  - **`/user/`** rappresenta la parte statica dell'URL
  - **`<name>`** rappresenta la parte dinamica: match del testo che appare dopo la parte statica fino al prossimo "/"

- Quando la view function (`user` nell'esempio) è invocata, Flask passa la parte dinamica dell'URL come parametro della funzione (parametro `name` nell'esempio)

Server Startup

- L’istanza di una applicazione Flask prevede un metodo `run` che avvia il web server di sviluppo integrato in Flask
  - Una volta avviato, il server entra in un loop in cui si mettere in attesa di richieste
  - Il ciclo termina solo forzandone la chiusura da terminale

```
if __name__ == "__main__":
    app.run(debug=True)
```

- Di default il server è:
  - Raggiungibile su **http://127.0.0.1:5000/**
  - Raggiungibile solo da localhost

- Può essere configurato:
  - `app.run (host=None, port=None, debug=None, **options)`

```
app.run(host='0.0.0.0', port=80)
```

Hello World in Flask: esecuzione

```python
# hello.py
from flask import Flask

app = Flask(__name__)

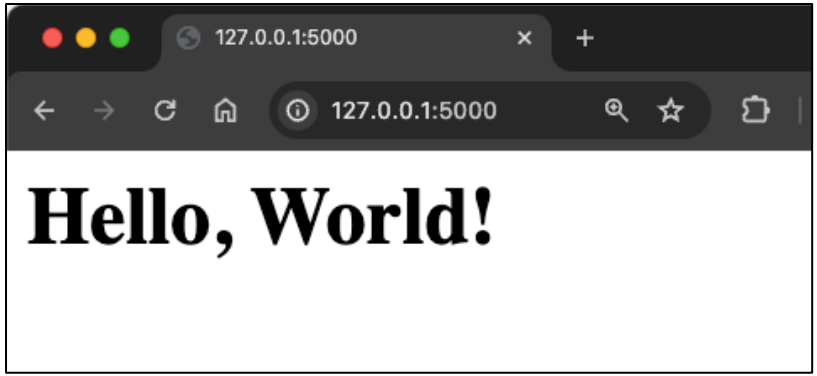
@app.route("/")
def index():
    return "<h1>Hello, World!</h1>"

@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, %s!</h1>' % name

if __name__ == "__main__":
    app.run(debug=True)
```

Hello World in Flask: esecuzione

```
user@hostname % python hello.py
 * Serving Flask app 'hello'
 * Debug mode: on
WARNING: This is a development server.
Do not use it in a production deployment.
Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 350-815-848
127.0.0.1 - - [08/May/2024 00:44:44]
     "GET / HTTP/1.1" 200 -
127.0.0.1 - - [08/May/2024 00:46:21]
     "GET /user/pippo HTTP/1.1" 200 -
```



HTML templating: **Jinja2**

- Negli esempi precedenti le view function ritornavano stringhe in HTML **generate direttamente nelle funzioni**
  - Mix tra *business logic* e *presentation logic*
  - Codice più difficile da comprendere e da manutenere
- Il modo migliore è affidarsi ad un ***template engine*** per spostare la *presentation logic in un template*
  - **Template:** file che contiene il testo di una risposta (parte statica), con dei *placeholder* per la parte dinamica che saranno popolati opportunamente dalla richiesta
  - **Rendering:** Processo che sostituisce i placeholder con i valori attuali e ritorna la risposta finale al client
- Flask utilizza il **template engine Jinja 2**

HTML templating: **Jinja2**

- Di default, i template devono essere inseriti nella sotto cartella `./templates`
  - E' possibile specificare un percorso diverso come parametro nell'application instance Flask

```
import os
template_dir = os.path.abspath('./path_to_templates/')
app = Flask(__name__, template_folder=template_dir)
```

- **I template sono file HTML** (estensione .html)

Esempio:

- File **templates/index.html**

```
<h1>Hello World!</h1>
```

- File **templates/user.html**

```
<h1>Hello, {{ name }}!</h1>
```

HTML templating: **Jinja2**

- E’ possibile effettuare il rendering dei template con la funzione `render_template(template_name_or_list, **context)`
  - **template_name_or_list:** Il nome del template di cui fare il rendering. Se è fornita una lista, il primo file disponibile sarà utilizzato per il rendering
  - **Context:** Le variabili da mettere a disposizione nel template

```
from flask import Flask, render_template
…
@app.route("/index")
def index():
    return render_template('index.html')

@app.route('/user/<name>')
def user(name):
    return render_template('user.html', name=name)
```

Template variables

- Jinja2 riconosce ogni tipo di variabile, compresi tipi complessi come liste, dizionari ed oggetti
- Alcuni esempi di variabili in template:

```
<p>A value from a dictionary: {{ mydict['key'] }}.</p>
<p>A value from a list: {{ mylist[3] }}.</p>
<p>A value from a list, with a variable index: {{ mylist[myintvar] }}.</p>
<p>A value from an object's method: {{ myobj.somemethod() }}.</p>
```

Strutture di controllo

Jinja2 offre anche **strutture di controllo** che possono essere utilizzate per alterare il flusso del template

Alcuni esempi:

- Costrutto condizionale

```
{% if user %}
    Hello, {{ user }}!
{% else %}
    Hello, Stranger!
{% endif %}
```

- Ciclo for:

```
<ul>
    {% for comment in comments %}
        <li>{{ comment }}</li>
    {% endfor %}
</ul>
```

curl

- `curl` è un command-line tool che permette di generare richieste HTTP (e non solo) da terminale
  - Installazione sistemi Linux: `sudo apt-get install curl`
  - Installazione MacOS: `brew install curl`
  - Installazione Windows: `https://curl.se/windows/`
- `curl https://www.google.com/`
  - Genera una GET verso l'URL specificato (metodo utilizzato di default, se i parametri non suggeriscono altro)
- Alcuni parametri utili*:
  - **`-I`** permette di recuperare solo gli Headers della risorsa richiesta, generando una richiesta HEAD
    - `curl -I https://example.com`
  - **`-d`** permette di specificare il campo data per una richiesta POST (curl considera automaticamente il metodo come POST, senza necessità di -X)
    - `curl -d "name=curl" https://example.com`
  - `--json` permette di inviare dati in formato JSON in una richiesta POST (come nel caso precedente -X non necessario)
    - `curl --json '{ "name": "pippo", "surname": "pippozzo"}' https://example.com`
  - -X permette di specificare il tipo di metodo HTTP da utilizzare per la richiesta (per sovrascrivere quello usato di default da curl)
    - `curl -X DELETE https://example.com`

* Riferirsi al manuale per altri: https://curl.se/docs/manpage.html

Request object

- Mette a disposizione le informazioni delle richieste HTTP inviate dai client, Flask utilizza i **context** e gli oggetti **request**
  - I context permettono di rendere temporaneamente accessibili gli oggetti request come variabili globali
  - I context permettono di rendere accessibili le variabili ad un thread senza interferire con gli altri thread
  - Attraverso gli oggetti request è possibile accedere ai campi della richiesta HTTP del client

```
from flask import Flask, request
…

@app.route('/')
def index():
    user_agent = request.headers.get('User-Agent’)
    return '<p>Your browser is %s</p>' % user_agent
```

Request object

- Attraverso l'oggetto request è possibile **accedere al body della richiesta**
  - `get_data(cache=True, as_text=False, parse_form_data=False)`
    - Legge i dati provenienti dal client in un unico bytes object
    - `cache`: specifica se i dati devono essere cached
    - `as_text`: specifica se i dati devono essere decodificati come stringa
    - `parse_form_data`: specifica se i dati devono essere parsati come form data
  - `get_json(force=False, silent=False, cache=True)`
    - Fa il parsing dei dati JSON presenti nella richiesta se il mimetype è application/json
    - `force`: specifica se ignorare il mimetype ed effettuare sempre il parsing JSON
    - `silent`: silenzia i mimetype e parsing errors, e ritorna None
    - `cache`: memorizza il parsed JSON per successive chiamate
- E' possibile **accedere ai parametri passati attraverso la URL (querysting)**
  - request.args: permette l'accesso ai parametri nella URL (la parte dell'URL dopo "?")

```
from flask import Flask, request
…

@app.post('/json')
def json():
   json = request.get_json()
   print(json)
   return json

@app.post('/data')
def json():
   data = request.get_data(as_text=True)
   print(data)
   return json
```

```
from flask import Flask, request
…
# test: curl http://127.0.0.1:5000/hello?name=pippo&surname=pippozzo
@app.get('/hello')
def hello():
   params = request.args
   print("name:", params['name']) # pippo
   print("surname:", params['surname']) # pippozzo
   return params
```

Route methods

- Di default, le route rispondono solo a richieste di tipo GET
- E' possibile utilizzare l'argomento **`methods`** del decoratore `route()` per gestire differenti metodi HTTP

```
@app.route('/print', methods=['GET', 'POST'])
def print_method():
   if request.method == 'POST':
      return '<h1>This is a POST</h1>'
   else:
      return '<h1>This is a GET</h1>'
```

Test
GET: `curl http://127.0.0.1:5000/print`
POST: `curl -X POST http://127.0.0.1:5000/print`

- E' possibile anche separare la gestione dei metodi HTTP in differenti funzioni
  - Flask mette a disposizione delle **shortcut** per decorare tali rotte
  - Esiste una shortcut per ogni metodo HTTP, ad esempio **`get()`** e **`post()`**

```
@app.get('/print')
def print_post():
   return '<h1>This is a POST</h1>'

@app.post('/print')
def login_post():
   return '<h1>This is a GET</h1>'
```

Response Object

- In generale, Flask utilizza la seguente logica per la **conversione dei valori di ritorno in un Response object**:

Se il valore di ritorno è:

- Oggetto Response, tale oggetto viene direttamente ritornato dalla view function
- Stringa, l'oggetto Response conterrà
  - la stringa come **body**
  - `200 Ok` come **status code**
  - `text/html` come **mimetype**
- Iterator o **generator** che ritornano stringhe o byte, allora viene trattato come una *streaming response*
- Dizionario o una **lista**, viene invocato `jsonify()` per la creazione dell'oggetto Response
- Tupla, è possibile includere varie informazioni extra. La tupla dovrà essere nel formato:
  - `(response, status)`
  - `(response, headers)`
  - `(response, status, headers)`

Da notare che:
  - `status:` sovrascrive lo status code della risposta
  - `header:` può essere una lista o dizionario con header aggiuntivi

- Nessuno dei casi precedenti, Flask assume che il valore di ritorno sia un applicazione WSGI valida, e la converte in un oggetto Response

Response Object: dizionario e list

- Il dizionario o la lista dovranno contenere dati che siano JSON-serializable

```
@app.route("/me")
def me_api():
    user = {
        "username": "pippo",
        "age": 20,
    }
    return user

@app.route("/users")
def users_api():
    users = get_all_users() #return a list of dictionaries
    return users
```

Response Object: tupla

- Nel caso in cui sia necessario ritornare uno **status code** differente dal 200 OK (ritornato di default), è possibile includerlo come secondo valore di ritorno (ritornando quindi una tupla):

```
@app.route('/error')
def error():
   return '<h1>Bad Request</h1>', 400
```

- Le risposte possono anche includere un terzo argomento che contiene un dizionario di headers che sono aggiunti alla risposta HTTP

```
@app.route('/errorh')
def errorh():
   headers = {'' : 'Flask Application'}
   return '<h1>Bad Request</h1>', 400, headers
```

1XX Information	
100	Continue
101	Switching Protocols
102	Processing
103	Early Hints
2XX Success	
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
205	Reset Content
206	Partial Content
207	Multi-Status (WebDAV)
208	Already Reported (WebDAV)
226	IM Used (HTTP Delta Encoding)
3XX Redirection	
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
306	Unused
307	Temporary Redirect
308	Permanent Redirect
4XX Client Error	
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
Compiled by Ivan Tay.	

4XX Client (Continue)	
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Payload Too Large
414	URI Too Large
415	Unsupported Media Type
416	Range Not Satisfiable
417	Exception Failed
418	I'm a teapot
421	Misdirected Request
422	Unprocessable Entity (WebDAV)
423	Locked (WebDAV)
424	Failed Dependency (WebDAV)
425	Too Early
426	Upgrade Required
428	Precondition Required
429	Too Many Requests
431	Request Header Fields Too Large
451	Unavailable for Legal Reasons
499	Client Closed Request
5XX Server Error Responses	
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported
507	Insufficient Storage (WebDAV)
508	Loop Detected (WebDAV)
510	Not Extended
511	Network Authentication Required
599	Network Connect Timeout Error

Response Object: make_response

- Flask permette ad una view function di costruire un oggetto Response all'interno della view function stessa attraverso la funzione **make_response()**
- Tale funzione accetta più argomenti, che saranno utilizzati per la costruzione del Response object
  - Il Response object sarà generato come nel caso di view function che ritorna una tupla di valori
- Tale modalità è utilizzata quando si vuole creare esplicitamente il Response object nella view function e configurare opportunamente tale oggetto
- In questo esempio viene creato un Response object e viene settato un cookie su di esso:

```
from flask import Flask, time, make_response
…
@app.route('/')
def index():
    response = make_response('<h1>This document carries a cookie!</h1>')
    response.set_cookie('Last-access', str(time.time()))
    return response
```

Web client Python: la libreria **`requests`**

La libreria **requests**

- Requests è una libreria HTTP per Python che permette di creare facilmente richieste HTTP/1.1
- Basata sulla libreria urllib3, attraverso la quale Requests supporta i meccanismi di Keep-alive ed HTTP connection pooling
- Supporta tutti metodi HTTP, ad es. GET, POST, DELETE, HEAD
- Installazione:
  - `pip install urllib3==1.26.6`
  - `pip install requests`

```
import requests

get_resp = requests.head('https://httpbin.org/get’)
post_resp = requests.post('https://httpbin.org/post', data={'key': 'value’})
del_resp = requests.delete('https://httpbin.org/delete’)
head_resp = requests.head('https://httpbin.org/get’)
```

La libreria **requests**

- ```
  requests.head(url, **kwargs)
  requests.get(url, params=None, **kwargs)
  requests.post(url, data=None, json=None, **kwargs)
  requests.put(url, data=None, **kwargs)
  requests.delete(url, **kwargs)
  ```

  - **url**: URL della richiesta
  - **params**: (opzionale) Dizionario, lista di tuple o bytes da inviare nella ***query string*** della richiesta
  - **data**: (opzionale) Dizionario, lista di tuple, bytes, o file-like object da inviare nel body della richiesta (utilizzabile nella POST e PUT)
  - **json**: (opzionale) Un oggetto JSON serializable da inviare nel body della richiesta (utilizzabile nella POST)
  - ****kwargs**: (opzionale) keyword arguments eventualmente necessari nella richiesta
  ```

Requests library: **query string**

- Requests permette di utilizzare le query string, cioè passare parametri all'interno dell'URL
  - `httpbin.org/get?key=val`
- La libreria permette di aggiungere i parametri attraverso dei keyword arguments

```
payload = {'key1': 'value1', 'key2': 'value2'}
r = requests.get('https://httpbin.org/get', params=payload)
print(r.url)
```

```
https://httpbin.org/get?key2=value2&key1=value1
```

- Da notare che:
  - Se una delle chiavi del dizionario ha un valore nullo allora, il parametro non sarà aggiunto alla query string dell'URL
  - E' possibile anche passare una lista come valore

```
payload = {'key1': 'value1', 'key2': ['value2', 'value3']}
r = requests.get('https://httpbin.org/get', params=payload)
print(r.url)
```

```
https://httpbin.org/get?key1=value1&key2=value2&key2=value3
```

Requests library: **POST payload**

- Il metodo `requests.post` permette di popolare il payload della richiesta attraverso l'argomento **`data`** o **`json`**
- **Dati come dizionario**. I dati saranno inseriti nella richiesta nel formato form-encoded

```
>>> payload = {'key1': 'value1', 'key2': 'value2'}
>>> r = requests.post('https://httpbin.org/post', data=payload)
>>> print(r.text)
{
 ...
 "form": {
    "key2": "value2",
    "key1": "value1"
 },
 ...
}
```

Requests library: **POST payload**

- **E’ possibile prevedere anche più valori per ogni chiave!**

```
>>> payload_tuples = [('key1', 'value1'), ('key1', 'value2’)]
>>> r1 = requests.post('https://httpbin.org/post', data=payload_tuples)
>>> payload_dict = {'key1': ['value1', 'value2’]}
>>> r2 = requests.post('https://httpbin.org/post', data=payload_dict)
>>> print(r1.text)
{
  ...
  "form": {
    "key1": [
      "value1",
      "value2"
    ]
  },
  ...
}
```

- **C’è un’equivalenza tra dati che sono trattati usando tipi di dato differenti:**

```
>>> r1.text == r2.text
True
```

- **E’ possibile aggiungere un payload in formato JSON**

```
url = 'https://api.github.com/some/endpoint'
payload = {'some': 'data'}
r = requests.post(url, json=payload)
```

Requests library: **Response object**

- Il riferimento ottenuto dall'invio di una richiesta è un **Response object** (diverso dal Response object di Flask) da cui è possibile ottenere informazioni sulla risposta
  - La libreria requests effettua automaticamente la decodifica delle informazioni generate dal server

```
r = requests.get('https://api.github.com/events')
r.text
```

```
'[{"repository":{"open_issues":0,"url":"https://github.com/...
```

- Requests effettua il rilevamento dell'*unicode charsets* se non specificato nella risposta, e lo sfrutta quando si accede a `r.text`
  - `r.encoding` permette di visualizzare l'encoding utilizzato, e di settarne uno nuovo
  - Il nuovo encoding sarà utilizzato ogniqualvolta si invoca `r.text`

Requests library: **Response object**

- **`r.content:`** mostra il contenuto della risposta in byte
- **`r.text:`** mostra il contenuto della risposta in unicode
- **`r.encoding:`** mostra l'encoding utilizzato dal server per la risposta (utilizzato con la `r.txt`)
- **`r.status_code:`** mostra lo status code della risposta
- **`r.url:`** mostra la final URL location della risposta
- **`r.headers:`** mostra gli headers della risposta
- **`r.headers['header name']:`** mostra uno specifico header,
  - Esempio: `r.headers['Content-Type']`

Requests library: **Response con contenuto JSON**

- Requests prevede anche un decoder JSON *built-in*, che permette di gestire eventuali risposte con contenuti nel formato JSON

```
r = requests.get('https://api.github.com/events')
r.json()
```

```
[{'repository': {'open_issues': 0, 'url': 'https://github.com/...
```

- Se il decoding JSON fallisce, la `r.json()` solleva una eccezione `requests.exceptions.JSONDecodeError`
  - Alcuni esempi sono: contenuto vuoto (204 – No Content) oppure se la risposta ha un contenuto che non rispetta il formato JSON

Requests library: **Response con contenuto JSON**

- Da notare che `r.json()` decodifica solo il contenuto della risposta, pertanto se la chiamata va a buon fine non è detto che la richiesta abbia avuto successo
  - Molti server ritornato in risposta un oggetto JSON anche a fronte di fallimenti (ad esempio per dare dettagli sul problema avuto lato server nel caso di un HTTP 500)
- Per verificare lo stato di una risposta è possibile utilizzare `r.status_code`
- Esiste anche la funzione `raise_for_status()` che solleva un `HTTPError`, se la risposta contiene un HTTP error come status code

```
>>> bad_r = requests.get('https://httpbin.org/status/404’)
>>> bad_r.status_code

404

>>bad_r.raise_for_status()

Traceback (most recent call last):
  File "requests/models.py", line 832, in raise_for_status
    raise http_error
requests.exceptions.HTTPError: 404 Client Error
```