



# API REST e OpenAPI

*Angelo Di Iorio*  
*Università di Bologna*



# API Web

- Un API Web descrive un'interfaccia HTTP che permette ad applicazioni remote di utilizzare i servizi di dell'applicazione
- Queste possono essere:
  - Applicazioni automatiche che utilizzano i dati dell'applicazione
  - Applicazioni Web che mostrano all'utente un menù di opzioni, magari anche un form, e gli permettono di eseguire un'azione sui dati dell'applicazione
- Un esempio: Twitter API v1.1  
<https://developer.twitter.com/en/docs/twitter-api/v1/>



# REST

- REST è l'acronimo di **REpresentational State Transfer**, ed è il **modello architetturale** che sta dietro al World Wide Web e in generale dietro alle applicazioni web “ben fatte” secondo i progettisti di HTTP.
- Applicazioni non REST si basano sulla generazione di un API che specifica le funzioni messe a disposizione dell'applicazione, e alla creazione di un'interfaccia **indipendente** dal protocollo di trasporto e ad essa completamente **nascosta**.
- Viceversa, un'applicazione REST si basa fundamentalmente sull'uso dei protocolli di trasporto (HTTP) e di naming (URI) per generare interfacce **generiche** di interazione con l'applicazione, e **fortemente connesse** con l'ambiente d'uso.



# Il modello CRUD

- Un pattern tipico delle applicazioni di trattamento dei dati
- Ipotizza che tutte le operazioni sui dati siano una di:
  - **Create** (inserimento nel database di un nuovo record)
    - Crea un cliente il cui nome è "Rossi SpA", il telefono "051 654321", la città "Bologna" e restituisce il codice identificatore che è 4123.
  - **Read** (accesso in lettura al database)
    - **individuale**: dammi la scheda del cliente con id=4123,
    - **contenitore**: dammi la lista dei clienti la cui proprietà *città* è uguale al valore "Bologna"
  - **Update**
    - Cambia il numero di telefono del cliente il cui id=4123 in "051 123456"
  - **Delete**
    - Rimuovi dal database il cliente con id=4123



# REpresentational State Transfer

L'architettura REST si basa su quattro punti :

1. Definire **risorsa** ogni concetto rilevante dell'applicazione Web
2. Associargli un **URI** come l'**identificatore** e selettore primario
3. Usare i verbi HTTP per esprimere ogni **operazione** dell'applicazione secondo il modello CRUD:
  - creazione di un nuovo oggetto (metodo PUT)
  - visualizzazione dello stato della risorsa (metodo GET)
  - cambio di stato della risorsa (metodo POST)
  - cancellazione di una risorsa (metodo DELETE)
4. Esprimere in maniera parametrica ogni **rappresentazione dello stato interno della risorsa**, personalizzabile dal richiedente attraverso un **Content Type** preciso



# Esempio REST: crea cliente


Il metodo specifica  
l'operazione eseguita

```
PUT clients/1234 HTTP/1.1  
Host: http://www.sito.com:80  
Content-Type: text/xml; charset=utf-8  
Content-length: 474
```

L' URI dell'oggetto  
coinvolto

```
<client xmlns:m="http://www.myAp  
  <nome>Rossi S.p.A.</nome>  
  <tel>051 654321</tel>  
  <citta>Bologna</citta>  
</client>
```

Il body contiene  
una rappresentazione  
(in questo caso XML)  
dell'oggetto da creare



# Esempio REST: aggiorna cliente

Il metodo PUT è usato sia per creare che per sostituire una risorsa

```
PUT clients/1234 HTTP/1.1
Host: http://www.sito.com:80
Content-Type: application/json; charset=utf-8
Content-length: 176
```

```
{
  "nome": "Rossi S.p.A.",
  "tel" : "051 654321",
  "citta" : "Bologna"
}
```

Il body contiene una rappresentazione JSON dell'oggetto da sovrascrivere



# Individui e collezioni

- REST identifica due concetti fondamentali: **individui e collezioni**
  - un cliente vs. l'insieme di tutti i clienti
  - un esame vs. l'insieme di tutti gli esami superati
  - ...
- Fornisce URI ad entrambi
- Ogni operazione avviene su uno e uno solo di questi concetti.
- Su entrambi si possono eseguire operazioni CRUD. A seconda della combinazione di verbi e risorse otteniamo l'insieme delle operazioni possibili.
- Ciò che passa come corpo di una richiesta e/o risposta **NON E'** la risorsa, ma una *rappresentazione* della risorsa, di cui gli attori si servono per portare a termine l'operazione.





# Gerarchie

- Le collezioni possono "contenere" individui o altre collezioni
- E' consigliabile strutturare gli URI in modo gerarchico, per esplicitare queste relazioni
- API più leggibile e *routing* semplificato in molti framework di sviluppo
- Ad esempio:
  - Tutti i clienti  
`/clients/`
  - Il cliente 1234  
`/clients/1234`
  - Tutti gli ordini del cliente 1234  
`/clients/1234/orders/`



# Linee guida degli URI in REST

Le **collezioni** sono intrinsecamente **plurali**. Gli **individui** sono intrinsecamente **singolari**.

Le collezioni debbono essere visivamente distinguibili dagli individui. Per questo usiamo un termine plurale e uno slash in fondo all'URI

URI	Rappresentazione
<code>/customers/</code>	collezione dei clienti
<code>/customers/abc123</code>	cliente con id=abc123
<code>/customers/abc123/</code>	collezione delle sotto-risorse del cliente con id=abc123 (es. indirizzi, telefoni, ecc.)
<code>/customers/abc123/addresses/1</code>	primo indirizzo del cliente con id=abc123



# Filtri e search negli URI REST

Un filtro genera un **sottoinsieme** specificato attraverso una regola di qualche tipo. La gerarchia permette di specificare i tipi più frequenti e rilevanti di filtro.

Altrimenti si usa la parte query dell'URI di una collezione:

URI	Rappresentazione
/regions/ER/customers/	collezione dei clienti dell'Emilia Romagna
/status/active/customers/	collezione dei clienti attivi
/customers/?tel=0511234567 oppure /customers/? search=tel&value=0511234567	collezione dei clienti che hanno telefono = 051 1234567
/customers/? search=sales&value=100000&op=gt	collezione dei clienti che hanno comprato più di 100.000€



# Uso dei verbi HTTP in REST

- Elencare tutti i clienti  
`GET /customers/`
- Accedere ai dati del cliente id=abc123  
`GET /customers/abc123`
- Creare un nuovo cliente (il client **non** decide l'identificatore)  
`POST /customers/`
- Creare un nuovo cliente (il client **decide** l'identificatore)  
`PUT /customers/abc123`
- Modificare (tutti) i dati del cliente id=abc123  
`PUT /customers/abc123`
- Modificare alcuni dati del cliente id=abc123  
`POST /customers/abc123/telephones/`
- Cancellare il cliente id=abc123  
`DELETE /customers/abc123`

Attenzione a questa differenza!  
Operazioni su collezione  
o individuo



# Semantica del POST

Nelle vecchie versioni di HTTP (ad es. RFC2616), si diceva:

- *"The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line. It essentially means that POSTrequest-URI should be of a collection URI."*

Nel 2014 è stata approvata una modifica e chiarificazione ad alcuni testi del documento di HTTP (RFC7231), e in particolare:

- *"The POST method requests that the target resource process the representation enclosed in the request according to the resource's own specific semantics."*

In pratica, il POST può essere usato in una moltitudine di situazioni secondo una semantica decisa localmente, purché non sovrapposta a quella degli altri verbi



# Altri consigli e linee guida

- Adottare una convenzione di denominazione coerente e chiara negli URI
- Usare gerarchie ma valutare i livelli necessari (chiarezza vs. carico sul server)
- Evitare di creare API che rispecchiano semplicemente la struttura interna di un database
- Fornire meccanismi – parametri nelle query – per filtrare e paginare le risposte
- Supportare richieste asincrone e in questo caso restituire codice HTTP 202 (Accettato ma non completato) e informazioni (URL) per accedere allo stato della risorsa



# Descrivere API con OpenAPI



# Descrivere una RESTful API

- Una API è RESTful se utilizza i principi REST nel fornire accesso ai servizi che offre
- Per documentare un API è necessario definire:
  - **end-point** (*URI / route*) che supporta
    - separando collezioni e elementi singoli
  - **metodi** HTTP di accesso
    - Cosa succede con un GET, un PUT, un POST, un DELETE, ecc.
  - **rappresentazioni in Input e Output**
    - Di solito non si usa un linguaggio di schema, ma un esempio fittizio e sufficientemente complesso
  - **condizioni di errore** e i **messaggi** che restituisce in questi casi





# Swagger e Open API

- Swagger è un ecosistema di tool per la creazione, costruzione, documentazione e accesso ad API soprattutto in ambito REST.
- In particolare ha creato un linguaggio per la documentazione di API REST e strumenti per l'editazione e la documentazione e il test di queste API.
- Nel 2016, il linguaggio è stato reso di pubblico dominio ed è diventato Open API
- Open API può essere serializzato sia in JSON che in YAML
- Standard industriale per API REST
- Generazione automatica di documentazione, modelli e codice



# Apriamo una parentesi: YAML

- YAML (Ain't a Markup Language) è una linearizzazione di strutture dati con sintassi ispirata a Python:
  - simile a JSON (in realtà un superset)
  - indentazione come modello di annidamento
  - supporto di tipi scalari (stringhe, interi, float), liste (array) e array associativi (coppie <chiave>:<valore>)

```
nome: Angelo
cognome: Di Iorio

ufficio:
  città: Bologna
  civico: 14
  via: Ranzani

corsi:
  - Programmazione
  - "Tecnologie Web"
```

```
name: Sagre
news:
  - id: 1
    titolo: Sagra del ...
    articolo: Lo stand ...
    immagine: sagra.jpg
  - id: 2
    titolo: Tortellini per tutti
    articolo: Bologna la patria...
    immagine: tortelli.jpeg
```



# OpenAPI in YAML

```
12 host: "petstore.swagger.io"
13 basePath: "/v2"
14 tags:
15 - name: "pet"
16   description: "Everything about your Pets"
17 - externalDocs:
18   description: "Find out more"
19   url: "http://swagger.io"
20 schemes:
21 - "https"
22 - "http"
23 paths:
24   /pet:
25     post:
26       tags:
27       - "pet"
28       summary: "Add a new pet to the store"
29       description: ""
30       operationId: "addPet"
31       consumes:
32       - "application/json"
33       - "application/xml"
34       produces:
35       - "application/xml"
36       - "application/json"
37       parameters:
38       - in: "body"
39       name: "body"
40       description: "Pet object to add to the store"
```



# Sezione paths

- La parte centrale di un'API descrive i percorsi (URL) corrispondenti alle operazioni possibili sull'API
- Seguono la struttura: `<host>/<basePath>/<path>`
- Per ogni percorso (path o endpoint) si definiscono tutte le possibili operazioni che, secondo i principi REST, sono identificate dal metodo HTTP corrispondente
- Per ogni `path` quindi ci sono tante sottosezioni quante sono le operazioni e per ognuna:
  - Informazioni generali
  - Parametri di input e di output



# Struttura di un path

Risorsa

```
/pet/{petId}:  
  get:  
    summary: "Find pet by ID"  
    description: "Returns a single pet"  
    operationId: "getPetById"  
    produces:  
      - "application/xml"  
      - "application/json"  
  
    parameters:  
      ...  
      ...  
  
  post:  
    summary: "Updates a pet in the store with form data"  
    description: ""  
    operationId: "updatePetWithForm"  
  
    parameters:  
      ...  
      ...
```

Formati in Output

Parametri in Input

Operazioni  
(metodi HTTP)



# Parametri in input

- I parametri in input sono descritti nella sezione **parameters** e per ogni parametro è possibile definire:
  - tipo del parametro: keyword **in** che può assumere valori **path**, **query** o **body**
  - nome (**name**) e descrizione (**description**)
  - se è opzionale o obbligatorio (**required**)
  - formato del/i valore/i che il dato può assumere (**schema**)
    - Il tipo può essere scalare (interi, stringhe, date, ecc.), o un oggetto o un vettore di valori scalari o oggetti



# Esempi di parametri path e query

```
/pet/{petId}:  
get:  
  summary: Find pet by ID  
  description: Returns a single pet  
  operationId: getPetById  
  parameters:  
    - name: petId  
      in: path  
      description: ID of pet to return  
      required: true  
      type: integer  
      format: int64
```

Parametro <petId> nell'URI

Parametro <status> nella parte query dell'URI /pet/?status=ready

```
/pet/:  
get:  
  summary: Finds Pets by status  
  operationId: findPetsByStatus  
  parameters:  
    - name: status  
      in: query  
      description: Status values that need to be considered for filter  
      required: true  
      type: array  
      items:  
        type: string
```



# Esempi di parametri nel body

Oggetto <User> nel body

```
/user/{username}:  
  put:  
    tags:  
      - user  
    summary: Updated user  
    description: This can only be done by the logged in user.  
    operationId: updateUser  
    parameters:  
      - name: username  
        in: path  
        description: name that need to be updated  
        required: true  
        type: string  
      - in: body  
        name: body  
        description: Updated user object  
        required: true  
        schema:  
          $ref: '#/definitions/User'
```





# Oggetti e definizioni

- Nell'esempio precedente il body contiene un oggetto di tipo **User**; viene infatti passata un'intera risorsa (o meglio la sua rappresentazione) come parametro
- La sezione **definitions** permette di definire i tipi degli oggetti, le loro proprietà e possibili valori
- Questi tipi possono essere referenziati (tramite **schema** -> **\$ref**) sia delle richieste che delle risposte



# Esempi di schemi

```
User:
  type: object
  properties:
    id:
      type: integer
      format: int64
    username:
      type: string
    firstName:
      type: string
    lastName:
      type: string
    email:
      type: string
    password:
      type: string
    phone:
      type: string
    userStatus:
      type: integer
      format: int32
      description: User Status
```

```
Order:
  type: object
  properties:
    id:
      type: integer
      format: int64
    petId:
      type: integer
      format: int64
    quantity:
      type: integer
      format: int32
    shipDate:
      type: string
      format: date-time
    status:
      type: string
      description: Order Status
      enum:
        - placed
        - approved
        - delivered
    complete:
      type: boolean
```



# Output

- L'output (dati e codici e messaggi di errore) sono definiti attraverso la keyword **responses**
- Si specifica il tipo di output atteso nel body della risposta
- Inoltre ogni risposta ha un id numerico univoco, associato al codice HTTP corrispondente
  - **200** viene usato per indicare che non c'è stato alcun errore
  - da **400** in su vengono in genere usati per indicare messaggi di errore



# Esempio di risposta

```
/pet/:  
  get:  
    summary: Finds Pets by status  
    operationId: findPetsByStatus  
    parameters:  
      - name: status  
        in: query  
        description: Status values that need to  
        required: false  
        type: array  
        items:  
          type: string  
    responses:  
      '200':  
        description: successful operation  
        schema:  
          type: array  
          items:  
            $ref: '#/definitions/Pet'  
      '400':  
        description: Invalid status value
```

Codici  
HTTP

Vettore di  
oggetti <Pet>



# Swagger Editor

<https://editor.swagger.io/>

The screenshot displays the Swagger Editor interface. On the left, a code editor shows the Swagger JSON definition for a Petstore API. The JSON includes fields for version (2.0), title (Swagger Petstore), description, contact information, license (Apache 2.0), host (petstore.swagger.io), and several API endpoints under the 'paths' section, such as /pet (POST), /pet (PUT), /pet/findByStatus (GET), and /pet/findByTags (GET).

On the right, the rendered Swagger UI is visible. It features the title 'Swagger Petstore 1.0.0' and a description of the API. Below the description, there are links for 'Terms of service', 'Contact the developer', 'Apache 2.0', and 'Find out more about Swagger'. A 'Schemes' dropdown menu is set to 'HTTPS', and an 'Authorize' button is present. The API endpoints are listed in a table-like format, each with a colored header (POST, PUT, GET) and a lock icon.



# Esercizio

- Progettare un API REST (parziale) per la gestione di un ristorante e descriverla in OpenAPI (JSON o YAML). Il ristorante offre menù diversi, ognuno caratterizzato da un ID e una descrizione testuale; ogni menù include diversi piatti, ognuno caratterizzato da un ID, una descrizione testuale e un prezzo. Tutti gli attributi sono obbligatori.
- L'API permette di:
  - ottenere l'elenco di tutti i menù
  - ottenere le informazioni di uno specifico menù (ID e descrizione, senza elenco piatti)
  - aggiungere un nuovo piatto ad un menù
- Specificare: URL di accesso, metodi HTTP, parametri e risposte con esempi.



# Conclusioni

- REST considera ogni applicazione come un ambiente di cui si cambia lo stato attraverso un insieme limitato di comandi (i metodi HTTP) applicati alle risorse (esprese attraverso URI) e visualizzate attraverso una esplicita richiesta di rappresentazione (attraverso un content Type MIME).
- REST ha il pregio di sfruttare completamente ed esattamente tutti gli artifici del web, ed in particolare caching, proxying, sicurezza, ecc.
- Inoltre l'aprirsi all'uso sistematico di URI permette ad applicazioni sofisticate basate su logica ed inferenza si sfruttare le tecniche del Semantic Web per creare funzionalità ancora più sofisticate e intelligenti con applicazioni create su architettura REST.