RESTful Web Services

Giuseppe Della Penna

Università degli Studi di L'Aquila
giuseppe.dellapenna@univaq.it
http://people.disim.univaq.it/dellapenna

> *Questo documento si basa sulle slide del corso di Sviluppo Web Avanzato, riorganizzate per migliorare l'esperienza di lettura. Non è un libro di testo completo o un manuale tecnico, e deve essere utilizzato insieme a tutti gli altri materiali didattici del corso. Si prega di segnalare eventuali errori o omissioni all'autore.*

1. Representational State Transfer

Il **Representational State Transfer** (REST) è l'architettura alla base del web.

Lo "stile architetturale" REST è stato introdotto dal *W3C Technical Architecture Group* parallelamente al rilascio della versione 1.1 del protocollo HTTP.

L'architettura **RESTful** è di tipo client/server, stateless, e i dati in essa scambiati sono opportune rappresentazioni di risorse, ottenute solitamente usando codifiche e formati standard del web.

Molte delle caratteristiche dello stile architetturale RESTful combaciano con i requisiti della web services architecture, per questo sono nati i servizi web "in stile REST" o **RESTful web services**.

Un servizio web RESTful

- Utilizza il **protocollo HTTP** per il trasporto delle risorse,
- Può utilizzare una qualsiasi codifica standard per le risorse, **tipicamente XML o JSON**,
- Dispone di un **insieme fissato di metodi**, corrispondenti ai verbi HTTP (`GET`, `POST`, `DELETE`, ecc.).

Data la sua semplicità, un servizio RESTful può essere implementato facilmente usando un server HTTP e un qualsiasi linguaggio di programmazione server side.

Tuttavia, **non si tratta di una tecnologia standardizzata nell'ecosistema dei web services, né supportata dai suoi formalismi di base, come il WSDL**.

Inoltre, **l'insieme fissato di metodi può non essere adatto a tutte le necessità**.

2. Struttura delle URL

2.1. Il Paradigma Collezione-Risorsa

I servizi web RESTful vengono spesso utilizzati per manipolare (tramite le classiche *CRUD operations*) risorse, reali o virtuali, come le tabelle di un database.

Le **risorse** da manipolare sono organizzate in una strutture gerarchiche tipo directory (che è spesso una vista sull'effettiva organizzazione dei dati), i cui elementi sono detti **collezioni**.

Ogni risorsa ha un **identificativo unico** all'interno della propria directory di appartenenza.

Una tipica URL per l'accesso a questo tipo di servizio inizierà con

```
http://server.net/servizio/rest(/interfaccia)
```

- *servizio* identifica univocamente il servizio a cui si accede. Può essere anche un path di più passi.
- La componente *rest* viene solitamente aggiunta per identificare l'interfaccia RESTful verso il servizio (e differenziarla da altre interfacce che usano tecnologie diverse, ivi compresa quella web).
- Opzionalmente, è possibile aggiungere un segmento di path che identifica la specifica *interfaccia* con la quale si accede al servizio: lo stesso servizio può avere interfacce diverse e/o versioni diverse della stessa interfaccia.

La parte restante della URL è di solito una sequenza di segmenti che identificano collezioni, sotto-collezioni o risorse all'interno di esse:

- `http://server.net/servizio/rest/utenti`
  Indica la collezione di tutti gli utenti (come la tabella di un database)
- `http://server.net/servizio/rest/utenti/U01`
  Indica l'utente con identificativo U01 nella la collezione di tutti gli utenti (come il record di una tabella)
- `http://server.net/servizio/rest/utenti/U01/permessi`
  Indica la sotto-collezione di tutti i permessi dell'utente con identificativo U01 nella la collezione di tutti gli utenti (in un database, corrisponde a percorrere la relazione tra la tabella utenti e quella permessi relativamente all'utente U01)
- `http://server.net/servizio/rest/utenti/U01/permessi/scrittura`
  Indica il permesso di scrittura (sotto-risorsa nella sotto-collezione permessi) preso tra quelli dell'utente U01
- ...e così via.

Sintassi alternative

È possibile anche trovare (prevalentemente per motivi di efficienza nell'accesso ai dati) URL con uno schema leggermente diverso da quello "collezione-risorsa-collezione..." appena esposto.

In particolare, l'ultimo segmento di una URL che rappresenta una risorsa può indicarne un attributo, ad esempio

```
http://server.net/servizio/rest/utenti/U01/nome
```

Qui non indichiamo una collezione *nome* relazionata con l'utente *U01*, ma piuttosto consideriamo la risorsa *utente U01* come una *collezione di attributi* e ne estraiamo uno chiamato "*nome*" (in un database, questo corrisponderebbe a selezionare solo una certa colonna di un record).

2.2. Query string

Infine, è possibile utilizzare una *query string* per simulare un'operazione di filtro, tipicamente quando la URL indica una collezione:

```
http://server.net/servizio/rest/utenti?n=pinco&c=pallino
```

Questa URL non dovrebbe rappresentare tutta la collezione degli utenti, ma solo quelli che si chiamano "Pinco Pallino".

La *query string* può essere inoltre utilizzata per scopi particolari come limitare il numero di record in output (emulando una clausola LIMIT), come vedremo.

3. Operazioni

Un servizio web RESTful può utilizzare come operazioni solo i verbi forniti da HTTP (con eventuali estensioni proprietarie).

Tipicamente, i verbi HTTP vengono mappati su delle *CRUD operations* applicate sulle risorse o collezioni identificate dalla URL. Se necessario, è possibile passare altre informazioni a queste operazioni tramite il *payload* del messaggio, ove previsto.

L'interpretazione dei verbi HTTP come metodi di manipolazione non è del tutto standard, ma nelle slide successive vedremo quella più comune.

Per ciascuna operazione indicheremo quali dati dovrebbero comparire nella richiesta e nella risposta, sia nel *payload* che negli *header*, e i più comuni codici di stato restituibili.

3.1. GET

Il metodo `GET` si usa come una **SELECT** in un database.

Su URL rappresentanti **collezioni**

```
http://server.net/servizio/rest/utenti?nome=pinco&cognome=pallino
```

```
SELECT * FROM utenti WHERE nome="pinco" AND cognome="pallino"
```

Input

- Eventuale **query string**, con la quale filtrare la collezione.
- La query string è a volte usata anche per passare **altri parametri utili** all'esecuzione della selezione, ad esempio "*start*" e "*end*" per specificare una clausola **LIMIT** *start,end* (limitazione/paginazione dei record restituiti): in questo caso, ha senso che il valore di ritorno sia una lista di record *completi*.
- Header `Accept`: *media type(s)* accettabili per la risposta.

Output

- Payload: una lista di url delle risorse (non le risorse stesse!) nella collezione, eventualmente filtrate.
- Header `Content-Type`: uno dei tipi elencati nell'Accept della richiesta.
- Return status: `OK` se la richiesta è stata correttamente servita.

Su URL rappresentanti **risorse**

```
http://server.net/servizio/rest/utenti/U01
```

```
SELECT * FROM utenti WHERE ID="U01"
```

Input

- Header `Accept`: *media type(s)* accettabili per la risposta.

Output

- Payload: la rappresentazione della risorsa nel media type scelto.
- Header `Content-Type`: uno dei tipi elencati nell'Accept della richiesta.
- Return status: `OK` se la richiesta è stata correttamente servita.

Su URL rappresentanti **attributi** di risorse

```
http://server.net/servizio/rest/utenti/U01/nome
```

```
SELECT nome FROM utenti WHERE ID="U01"
```

Input

- Header `Accept`: *media type(s)* accettabili per la risposta.

Output

- Payload: la rappresentazione dell'attributo della risorsa nel media type scelto.
- Header `Content-Type`: uno dei tipi elencati nell'Accept della richiesta.
- Return status: `OK` se la richiesta è stata correttamente servita.
  Altri return status possibili:
  - `METHOD_NOT_ALLOWED`: operazione non supportata su questa URL
  - `UNSUPPORTED_MEDIA_TYPE`: il server non sa come restituire il risultato in alcuno dei media type richiesti

    - `NOT_FOUND`: risorsa, collezione o attributo inesistente
    - `INTERNAL_SERVER_ERROR`: ogni errore generico nell'esecuzione dell'operazione

3.2. PUT (e PATCH)

Il metodo `PUT` si usa come una **UPDATE** in un database: aggiorna risorse preesistenti in una collezione.

Su URL rappresentanti **risorse**

```
http://server.net/servizio/rest/utenti/U01
```

```
UPDATE utenti SET ... WHERE ID="U01"
```

Input

- `Content-Type` header: *media type* con cui si sta per trasmettere la risorsa da aggiornare.
- Payload: rappresentazione della risorsa che verrà sovrascritta a quella indicata dalla URL, codificata nel tipo dichiarato.
  La semantica di `PUT` richiede che il payload **contenga l'intera risorsa**, anche se solo una parte di essa viene effettivamente aggiornata.
  Il metodo `PATCH`, se supportato dal server, può essere usato per trasmettere una rappresentazione **parziale**, per sostituire solo gli attributi specificati. *In assenza di supporto per questo metodo, per limitare il traffico, è comunque utile "forzare" la semantica del* `PUT` *ammettendo anche la trasmissione di risorse parziali.*

Output

- Return status: `NO_CONTENT` se la richiesta è stata correttamente servita.

3.3. PUT

Su URL rappresentanti **attributi** di risorse

```
http://server.net/servizio/rest/utenti/U01/nome
```

```
UPDATE utenti SET nome=… WHERE ID="U01"
```

Input

- `Content-Type` header: *media type* con cui si sta per trasmettere la risorsa da aggiornare.
- Payload: rappresentazione dell'attributo che verrà sovrascritto a quello della risorsa indicata dalla

URL, codificato nel tipo dichiarato.

Output

- Return status: `NO_CONTENT` se la richiesta è stata correttamente servita.

Su URL rappresentanti **collezioni** di risorse

```
http://server.net/servizio/rest/utenti
```

Input

- `Content-Type` header: *media type* con cui si sta per trasmettere le risorse da aggiornare.
- Payload: rappresentazione di una lista di risorse, codificata nel tipo dichiarato. L'intero contenuto della collezione verrà cancellato e sostituito con gli item indicati.

Output

- Return status: `NO_CONTENT` se la richiesta è stata correttamente servita.
  Altri return status possibili:
  - `METHOD_NOT_ALLOWED`: operazione non supportata su questa URL
  - `UNSUPPORTED_MEDIA_TYPE`: il server non sa come decodificare il payload con il media type dichiarato
  - `NOT_FOUND`: risorsa o attributo inesistente
  - `INTERNAL_SERVER_ERROR`: ogni errore generico nell'esecuzione dell'operazione

3.4. POST

Il metodo `POST` si usa come una **INSERT** in un database: aggiunge una risorsa a una collezione.

Su URL rappresentanti collezioni

```
http://server.net/servizio/rest/utenti
```

```
INSERT INTO utenti VALUES(...)
```

Input

- `Content-Type` header: *media type* con cui si sta per trasmettere la nuova risorsa.
- Payload: rappresentazione della risorsa da aggiungere alla collezione, codificata nel tipo dichiarato.

Output

- Payload: URL utilizzabile per accedere alla risorsa appena inserita (*opzionale*).
- Header `Location` impostato alla stessa URL restituita.
- Return status: `CREATED` se la richiesta è stata correttamente servita.
  Altri return status possibili:
  - `METHOD_NOT_ALLOWED`: operazione non supportata su questa URL
  - `UNSUPPORTED_MEDIA_TYPE`: il server non sa come decodificare il payload con il media type dichiarato
  - `NOT_FOUND`: collezione inesistente
  - `INTERNAL_SERVER_ERROR`: ogni errore generico nell'esecuzione dell'operazione

3.5. DELETE

Il metodo `DELETE` si usa come una `DELETE` in un database: elimina una risorsa da una collezione o svuota la collezione.

Su URL rappresentanti **risorse**

```
http://server.net/servizio/rest/utenti/U01
```

```
DELETE FROM utenti WHERE ID="U01"
```

Output

- Return status: `NO_CONTENT` se la richiesta è stata correttamente servita.

Su URL rappresentanti **collezioni**

```
http://server.net/servizio/rest/utenti
```

```
DELETE FROM utenti
```

Output

- Return status: `NO_CONTENT` se la richiesta è stata correttamente servita.
  Altri return status possibili:
  - `METHOD_NOT_ALLOWED`: operazione non supportata su questa URL
  - `NOT_FOUND`: risorsa inesistente
  - `INTERNAL_SERVER_ERROR`: ogni errore generico nell'esecuzione dell'operazione

3.6. HEAD

Il metodo `HEAD` viene implementato raramente. Ha la stessa semantica del metodo `GET`, ma **non restituisce il corpo della risposta**. In questo senso, è utile **per verificare l'esistenza di una risorsa senza prelevarla**.

Su URL rappresentanti **risorse**

```
http://server.net/servizio/rest/utenti/U01
```

Output

- Come per `GET`, omettendo il corpo (la rappresentazione della risorsa).
- Return status: `OK` se la risorsa esiste ed è leggibile.
  Altri return status possibili:
  - `METHOD_NOT_ALLOWED`: operazione non supportata su questa URL
  - `NOT_FOUND`: risorsa inesistente
  - `INTERNAL_SERVER_ERROR`: ogni errore generico nell'esecuzione dell'operazione

3.7. OPTIONS

Il metodo `OPTIONS` viene implementato raramente. Il suo scopo è quello di indicare quali metodi sono ammessi sulla risorsa.

Può essere utile per verificare le "autorizzazioni di accesso" su una risorsa (specialmente quando l'accesso avviene all'interno di una sessione con autenticazione dell'utente).

Su URL rappresentanti **risorse**

```
http://server.net/servizio/rest/utenti/U01
```

Output

- Header `Allow`: la lista (separata di virgole) dei metodi ammessi sulla risorsa.
- Return status: `OK` se la richiesta è stata correttamente servita.
- *Opzionalmente*, il payload della risposta può contenere una descrizione più dettagliata delle opzioni di comunicazione disponibili per la risorsa. In tal caso, l'header `Content-Type` andrà impostato opportunamente per riflettere il formato della risposta.
  Altri return status possibili:
  - `METHOD_NOT_ALLOWED`: operazione non supportata su questa URL
  - `NOT_FOUND`: risorsa inesistente
  - `INTERNAL_SERVER_ERROR`: ogni errore generico nell'esecuzione dell'operazione

4. Cross-Origin Resource Sharing (CORS)

Per ragioni di sicurezza, le richieste HTTP originate da una pagina web tramite AJAX possono essere dirette solo verso lo stesso dominio della pagina stessa (**same-origin policy**).

Con i servizi web, però, è molto comune che un client Javascript acceda a un servizio offerto da terze parti, quindi su un dominio differente.

CORS è l'acronimo di **cross-origin resource sharing**. Si tratta di un meccanismo utilizzabile per evitare il blocco delle richieste "*cross-domain*" normalmente forzato dai browser.

Tramite gli header specificati da CORS, il server può comunicare al browser se e quali richieste cross-domain permettere.

In generale, un servizio web RESTful dovrebbe inserire i seguenti header in *tutte* le sue risposte:

```
Access-Control-Allow-Origin "*"
Access-Control-Allow-Headers "origin, x-requested-with, content-type"
Access-Control-Allow-Methods "PUT, GET, POST, DELETE, OPTIONS"
```

Questi header garantiscono l'accesso al servizio da qualunque dominio ("*") e per tutti i metodi HTTP. Ovviamente è possibile personalizzare gli header per irrobustire il proprio servizio, proteggendolo da accessi indesiderati.

4.1. CORS e OPTIONS

Gli header CORS andrebbero restituiti insieme a tutte le risposte del server.

Spesso, comunque, prima di effettuare la chiamata vera e propria (con il metodo specificato dall'utente), **le librerie client HTTP inviano sulla stessa URL una chiamata `OPTIONS` per verificare, nel modo *meno costoso* possibile, se e in che misura la chiamata effettiva sarebbe accettabile, *anche verificando gli header CORS* restituiti**.

In questo scenario, ogni chiamata, o quantomeno la prima chiamata verso ciascuna risorsa, **viene preceduta da una chiamata `OPTIONS` generata automaticamente**, attraverso la quale la libreria HTTP client decide se proseguire con la reale richiesta dell'utente oppure bloccarla a priori invocando la *same origin policy*.

Quindi attenzione: se nel vostro framework il metodo `OPTIONS` ha, ad esempio, l'implementazione di default prevista dalle Servlet Java (*405 – Method not Allowed*), potreste ricevere errori imprevisti, che non hanno a che vedere con l'implementazione del metodo che state chiamando, ma con quella dell'`OPTIONS` implicito che lo precede!

5. Operazioni RESTful generiche

In alcuni casi è necessario esporre **operazioni che non corrispondono alle *CRUD operations*** appena esposte (`http://server.net/servizio/rest/utenti/U01/inviaEmail` per inviare una email a un utente dato il testo).

In questi casi, si utilizzano spesso URL del tipo

```
http://server.net/servizio/rest/contesto/metodo
```

- *contesto* può essere una collezione, una risorsa o altro che "contestualizzi" in qualche modo l'esecuzione del metodo;
- *metodo* è il nome dell'operazione da invocare.

La chiamata si effettua con una `POST`.

`POST` su URL rappresentanti un **metodo**, possibilmente contestualizzato

```
http://server.net/servizio/rest/utenti/U01/inviaEmail
```

```
utenti.get("U01").inviaEmail(testo) //invocazione metodo
```

Input

- Header `Content-Type`: *media type* usato per il payload, solitamente text/plain o application/json.
- Payload: rappresentazione dei parametri del metodo (se necessari) codificati nel tipo dichiarato.
- Header `Accept`: *media type(s)* accettabili per la codifica del valore di ritorno del metodo, se previsto.

Output

- Payload: rappresentazione del valore di ritorno del metodo, se previsto, codificato nel tipo dichiarato.
- Header `Content-Type`: uno dei tipi elencati nell'Accept della richiesta.
- Return status: `OK` se la richiesta è stata correttamente servita.
  Altri return status possibili:
  - `METHOD_NOT_ALLOWED`: metodo non supportata (in questo contesto)
  - `INTERNAL_SERVER_ERROR`: ogni errore generico nell'esecuzione dell'operazione

5.1. Lettura di dati derivati

Tuttavia, non tutto ciò che esula dalla struttura dati "di base" associata al servizio deve per forza essere modellato come metodo tramite `POST`. In alcuni casi, quando la chiamata può essere vista come una lettura di dato "derivato", ad esempio la dimensione di una collezione, è ugualmente lecito (anzi più chiaro) usare la `GET`.

`GET` su URL rappresentanti un **metodo di lettura (dato sintetico o derivato)**, possibilmente contestualizzato e privo di parametri

```
http://server.net/servizio/rest/utenti/count
```

```
utenti.count() //invocazione metodo
```

Input

- Header `Content-Type`: *media type* usato per il payload, solitamente text/plain o application/json.
- Payload: rappresentazione dei parametri del metodo (se necessari) codificati nel tipo dichiarato.
- Header `Accept`: *media type(s)* accettabili per la codifica del valore di ritorno del metodo, se previsto.

Output

- Payload: rappresentazione del valore di ritorno del metodo, se previsto, codificato nel tipo dichiarato.
- Header `Content-Type`: uno dei tipi elencati nell'Accept della richiesta.
- Return status: `OK` se la richiesta è stata correttamente servita.
  Altri return status possibili:
  - `METHOD_NOT_ALLOWED`: metodo non supportato (in questo contesto)
  - `INTERNAL_SERVER_ERROR`: ogni errore generico nell'esecuzione dell'operazione

6. Tecnologie

6.1. Server: Java/Servlet

È possibile realizzare un servizio RESTful molto semplice a partire dalle servlet. Solitamente si usa uno schema come quello che segue:

- Una servlet di dispatching viene mappata (tramite un URL pattern terminante con un "*") su tutte le URL del servizio.
- La servlet di dispatching analizza il primo segmento utile della URL effettiva e smista la chiamata alla servlet delegata a elaborare quel segmento, tramite il metodo `forward` del suo `RequestDispatcher`.

- Ciascuna servlet chiamata potrà, se la URL contiene altri segmenti, delegare a sua volta l'elaborazione a una servlet più specifica.
- L'ultima servlet di questa catena, invece, sovrascrivendo opportunamente i metodi `doX` (con X= Get, Put, Post,...) gestirà la richiesta effettiva.

Avremo quindi in generale una servlet per ogni segmento accodabile a una URL valida del servizio.

Nel gestire la richiesta, la servlet dovrà prestare attenzione a leggere/scrivere e interpretare correttamente anche gli header `Accept` e `Content-Type`. In particolare, per gestire la codifica e la decodifica JSON, è molto utile **Google GSON** (https://github.com/google/gson).

6.2. Server: Java/JAX-RS

JAX-RS è una tecnologia per la realizzazione di servizi web RESTful in applicazioni web Java estremamente avanzata, che permette di realizzare servizi molto compatti e modulari.

L'implementazione di riferimento di JAX-RS è **Jersey**. La struttura di un servizio realizzato con Jersey si basa sul concetto di risorse e *sotto-risorse*, mappate su classi tramite un uso estensivo di annotazioni che permettono di mettere in corrispondenza i relativi metodi con segmenti di URL e verbi HTTP.

Attenzione: esistono due rami di sviluppo per Jersey: la 2.x è compatibile con i server che utilizzano la Java EE, mentre la 3.x è da utilizzarsi sui server che utilizzano la nuova Jakarta EE.

6.3. Server: PHP

Il problema principale dei servizi RESTful realizzati con PHP è che spesso rendono necessario **programmare sistemi di mappatura delle URL esterni**, come il **MOD_REWRITE** di Apache HTTPD.

Con PHP è possibile realizzare servizi web RESTful utilizzando normali script. In questo caso bisogna gestire manualmente gli header (`Accept`, `Content-Type`). L'accesso al payload e la gestione di metodi diversi da `GET` e `POST` può risultare macchinosa (in quanto PHP è più orientato a questi due metodi).

Esistono anche vari framework che incorporano librerie per la realizzazione semplificata di servizi web RESTful, ma non c'è alcuno standard al riguardo. Ad esempio segnaliamo **Slim** (http://www.slimframework.com).

6.4. Client

Essendo i servizi web RESTful basati sugli standard web e su codifiche semplici delle risorse, realizzare un client è molto facile:

- In Java, si possono usare le classi del package **java.net**, ma è consigliabile (soprattutto per gestire più facilmente i metodi e gli header) usare librerie avanzate come l' **Apache HTTP Client** (http://hc.apache.org/httpcomponents-client-ga). Inoltre, per gestire la codifica e la decodifica

JSON, è molto utile **Google GSON** (https://github.com/google/gson).

- In PHP si possono usare le estensioni built-in per le connessioni HTTP e la codifica JSON, oppure poggiarsi a utili librerie di terze parti come **HTTPFUL** (https://github.com/nategood/httpful).
- In Javascript è possibile usare *Web API* di base come la **XMLHttpRequest** (https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest) o più moderne come le **Fetch API** (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) o ancora basarsi sugli helper AJAX di librerie come JQuery (https://api.jquery.com/Jquery.ajax/). Infine, i framework per applicazioni web client side fanno solitamente largo uso di chiamate RESTful.

7. Riferimenti

Representational state transfer
https://en.wikipedia.org/wiki/Representational_state_transfer

Java API for RESTful Services (JAX-RS)
https://jax-rs-spec.java.net

Jersey – RESTful Web Services in Java
https://eclipse-ee4j.github.io/jersey

Slim Framework
http://www.slimframework.com/

Cross-Origin Resource Sharing
https://www.w3.org/TR/cors/
https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

Same Origin Policy
https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

HTTP Response Codes
https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

Title: RESTful Web Services

Author: Giuseppe Della Penna*, University of L'Aquila*

Version: 20240530