

RESTful API

GET PUT POST DELETE

Indice generale

REST.....	3
Perchè usare REST.....	4
Risorse.....	5
Sicurezza.....	6
Sviluppo.....	7
Web Server.....	7
SSH.....	7
Apache.....	8
PHP.....	9
MySQL.....	10
Netfilter.....	10
Risorse.....	12
Api.....	14
Web Application.....	16
Storia – Prima Guerra Mondiale.....	16
Italiano – Giuseppe Ungaretti.....	17
Storia – Caduti Lombardi.....	19
Open Database.....	20
Conclusione.....	20

REST

REpresentational State Transfer (REST) è un tipo di architettura software introdotto da Roy Fielding nel 2000. Si basa su pochi principi di progettazione che rendono la struttura, alla quale si applicano, più scalabile ovvero in grado di crescere senza intervenire sull'intero sistema. Per capire cosa possa significare scalabile è forse meglio concentrarsi su cosa non lo è. Un televisore ad esempio non è scalabile perché se lo volete più grande dovete buttarlo tutto e comprarne uno nuovo, al contrario un'auto è scalabile perché se volete cambiare le ruote o aggiungere un alettone non dovete per forza comprare una macchina nuova. Il World Wide Web, ovvero quel servizio di Internet che ci permette di accedere a un insieme vastissimo di contenuti collegati tra loro attraverso link, è considerato il più grande sistema RESTful funzionante.

Con la sigla RESTful si catalogano tutti i sistemi che rispettano questi principi:

- ogni risorsa è unica e indirizzabile usando una *sintassi universale*
- tutte le risorse sono condivise :
 - tramite un insieme vincolato di operazioni ben definite
 - tramite un protocollo che rispetta questi vincoli strutturali:
 - **Client-server.** Devono esistere almeno un'entità client, un semplice dispositivo che accede alle risorse (il vostro smartphone), e almeno un'entità server, un computer che fornisce servizi, separati da una o più interfacce. Questa separazione di ruoli significa che, per esempio, il client non si deve preoccupare del salvataggio delle informazioni, che rimane all'interno di ogni singolo server, in questo modo la portabilità del codice del client ne trae vantaggio. I server non si devono preoccupare dell'interfaccia grafica o dello stato dell'utente, in questo modo i server sono più semplici e maggiormente scalabili. Server e client possono essere sostituiti e sviluppati indipendentemente fintanto che l'interfaccia non viene modificata.
 - **Stateless.** Ogni richiesta da ogni client contiene tutte le informazioni necessarie per richiedere il servizio.
 - **Cacheable.** Le risposte devono in ogni modo definirsi implicitamente o esplicitamente conservabili o no, in modo da prevenire che i client possano riusare risorse vecchie o dati errati. Una gestione ben fatta della cache può ridurre le comunicazioni client-server, migliorando scalabilità e le performance.
 - **Layered system.** Un client non può dire se è connesso direttamente ad un server o se passa attraverso intermediari.
 - **Uniform interface.** Un'interfaccia di comunicazione omogenea tra client e server permette di semplificare e disaccoppiare l'architettura, la quale si può evolvere separatamente.

Possiamo quindi notare come il World Wide Web sia a tutti gli effetti un sistema RESTful in quanto né rispetta tutti i vincoli.

1. Ogni risorsa presente nel WWW (un file di testo, una pagina web, un video ecc...) è unica, identificata univocamente da un URI (Uniform Resource Identifier) universale. Questo è semplicemente il testo che si digita nella barra degli indirizzi del browser.

2. Tutti le risorse sono condivise con un protocollo (HTTP) che rispetta i vincoli strutturali e possiede un insieme vincolato di operazioni (GET,POST,PUT,DELETE)

Roy Fielding ha partecipato alla stesura delle specifiche di HTTP 1.1, protocollo fondamentale per la navigazione in rete nel WWW, rendendolo così conforme alla sua architettura e rendendo il World Wide Web un sistema più scalabile.

Un importante modifica apportata è quella di introdurre le quattro operazioni fondamentali sopracitate e definirle chiaramente.

Il protocollo HTTP deve essere visto come una busta, come quella si spedisce con le poste, dove dentro c'è la nostra richiesta e fuori, sulla busta, c'è il mittente e il destinatario.

Con la versione 1.1 del protocollo ora sulla busta viene scritto, oltre all'URI della risorsa, anche l'operazione che si desidera effettuare su di essa (GET,POST,PUT,DELETE) che nell'ordine significano (prendi,crea,modifica,elimina). Colui che possiede le risorse ora sa come comportarsi già prima di aprire la busta e può eventualmente scartarla.

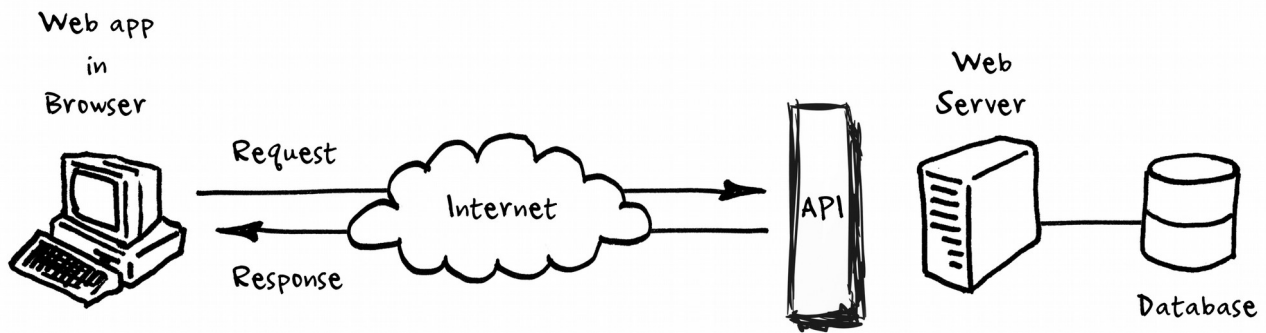
L'architettura REST rimane ampiamente utilizzata in molti sistemi informatici perché aiuta molto il processo di sviluppo e mantenimento.

Con la nascita degli smartphone il suo utilizzo ha subito una forte impennata questo a causa della necessità di avere molti dispositivi diversi che hanno bisogno di accedere alle stesse risorse. Un esempio eclatante sono i social network che hanno tipicamente un database pieno di risorse e diverse tipologie di client (App per Android, iOS, FirefoxOs, Windows Phone ecc..) che devono accedervi. Questo problema viene risolto da questi fornitori di servizi sviluppando il loro software seguendo i principi REST.

Perchè usare REST

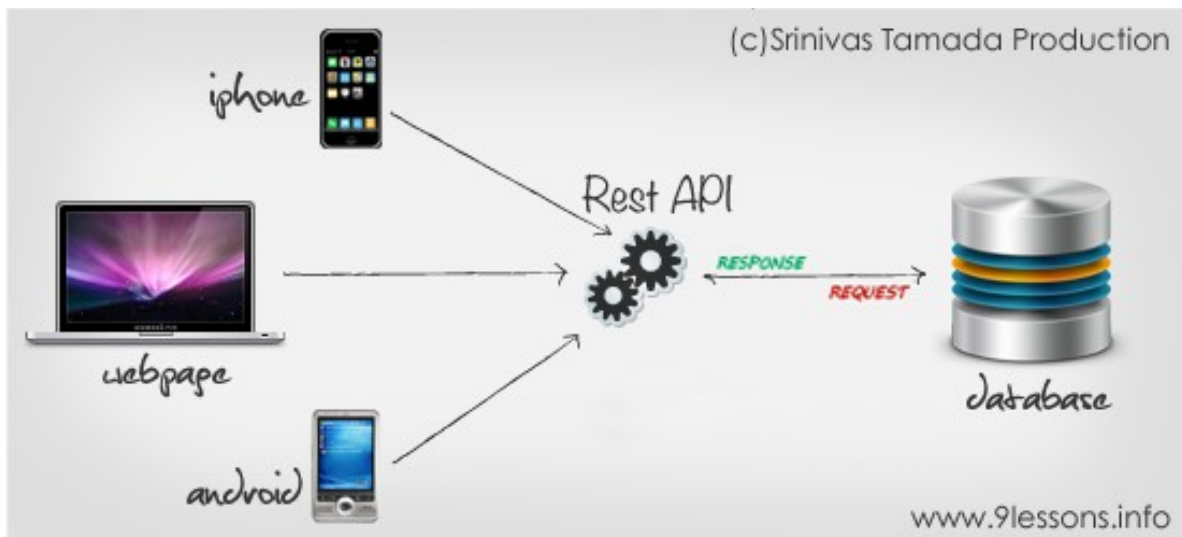
Twitter ad esempio è un social network che si basa sull'architettura REST. Il motivo per cui questa società ha scelto un sistema RESTful, come la maggior parte della aziende che forniscono servizi cloud, è dettata dal fatto che l'utilizzo di un servizio RESTful permette la divisione della architettura la quale si può evolvere separatamente. Il sistema diventa quindi più stabile, performante, di facile manutenzione e crescita. In questo modo le diverse applicazioni possono accedere a dati presenti sul server facendo riferimento ad una singola interfaccia comune che si frappone tra loro e il database (le API).

Possiamo considerare queste API come dei traduttori. Immaginiamo che nel mondo dei computer ogni macchina sappia parlare un lingua come l'italiano e anche l'inglese. Se il nostro database parla solo il russo per far comunicare il database con i dispositivi ci sono due soluzioni; Possiamo insegnare ad ogni macchina il russo ma sarebbe un lavoro lungo e faticoso, l'altra opzione è di creare uno o più traduttori che conoscono sia l'inglese che il russo. I dispositivi contatteranno questi traduttori (le API) che parlano con il database e poi rispondono al dispositivo.



Difficile spiegare ai non tecnici come questo sia un enorme miglioramento. Provate ad esempio a pensare allo scandalo che ha colpito la Volkswagen sui motori diesel. Per rimediare all'errore la società deve ora riportare in sede tutte le autovetture e cambiargli il motore per poi restituirle, un enorme dispendio di soldi e tempo da entrambe le parti. Se potessimo applicare un sistema RESTful al problema è come se alla Volkswagen avessero un motore uguale a quello su tutte le vetture e una semplice modifica a questo motore comporterebbe istantaneamente la modifica di tutti i motori di tutte le auto. Fantastico.

Questa architettura apre poi infiniti sviluppi ad esempio twitter utilizza un sistema di questo tipo per dare accesso al suo database anche ad applicazioni di terze parti (si veda il bottone 'accedi con twitter' presente in alcune applicazioni).

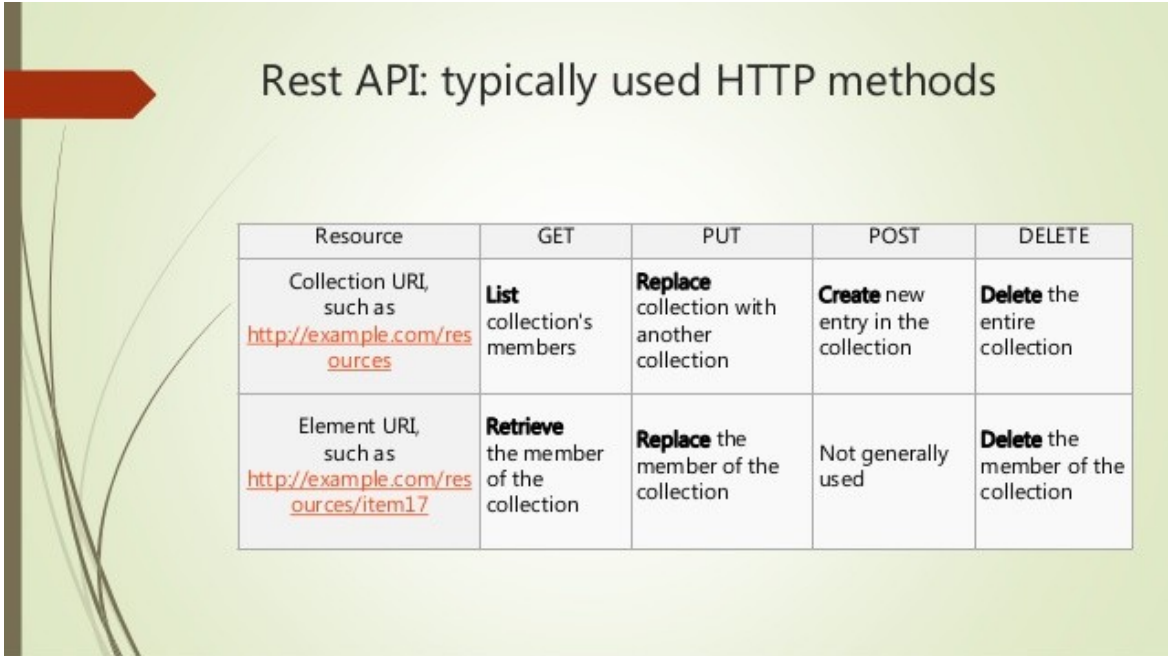


Risorse

Alla base di ogni architettura REST sono presenti le risorse ovvero dati, informazioni. Senza alzare il livello di astrazione possiamo dire che queste risorse sono individuabili tramite URI univoche e universali e accessibili solo tramite i verbi HTTP (GET,POST,PUT,DELETE). Rimanendo nell'ambito del Word Wide Web, è utile conoscere che esistono solo due tipi di

risorse, quelle “originali” (file, video, immagini) ovvero quelle fisicamente presenti sul server e quelle “astratte” (post, commenti, inserzioni ecc..), derivate, nella maggior parte dei casi dalle tabelle principali di un database .

I fornitori di servizi (twitter, ebay etc..) quindi creano solo le risorse astratte derivate dai loro database e forniscono poi API per accedervi.



Rest API: typically used HTTP methods

Resource	GET	PUT	POST	DELETE
Collection URI, such as http://example.com/resources	List collection's members	Replace collection with another collection	Create new entry in the collection	Delete the entire collection
Element URI, such as http://example.com/resources/item17	Retrieve the member of the collection	Replace the member of the collection	Not generally used	Delete the member of the collection

Queste API, che tecnicamente sono degli script, costruiscono queste risorse astratte ad ogni richiesta e le inviano come risposta in uno specifico formato (tipicamente JSON o XML)

Un esempio finale:

Prendiamo l'applicazione di subito.it. Quando noi selezioniamo un annuncio (risorsa astratta) questa applicazione effettuerà una chiamata HTTP di tipo GET alle api del server richiedendo quello specifico annuncio. Il server risponderà con la risorsa richiesta in formato JSON. Il client quindi visualizzerà in modo carino le informazioni ricevute. Un annuncio contiene però anche immagini perciò dentro la risorse ricevuta in risposta saranno contenuti anche i link alle immagini dell'annuncio (risorse “originali”). L'applicazione effettuerà quindi una nuova chiamata HTTP di tipo GET, non alle api, ma direttamente al web server che fornirà quindi l'immagine in risposta.

Sicurezza

Essendo REST un'architettura e non un protocollo o una specifica non contempla la sicurezza delle risorse. Chiaramente un sistema RESTful è più sicuro perché il database è accessibile solo a file che risiedono sul server e neanche gli stessi sviluppatori conoscono la struttura del database.

Il concetto di utenza non è perciò presente ma, nel mondo reale, è necessario che delle risorse siano visibili, modificabili o eliminabili solo da alcune persone autorizzate. Molti servizi hanno quindi adottato metodi di autenticazione e autorizzazione per le loro RESTful API. Ad ogni applicazione è quindi associato un Token, una specie di numero di serie che

identifica la stessa. Esso ha una validità limitata nel tempo ed è associato ad alcuni permessi. Esistono due tipi di token utilizzati maggiormente:

- JWT
- Bearer

Il primo consente di introdurre il concetto di utenza, ovvero il server può capire chi sta chiedendo una determinata risorsa e se ha i permessi per farlo, rimanendo fedele ai principi REST.

Il secondo invece viola il principio di stateless, ha bisogno di un server Auth per funzionare, ma consente l'accesso anche ad applicazioni di terze parti.

Sviluppo

Per dimostrare le potenzialità di questa architettura ho proposto un piccolo progetto sulla prima guerra mondiale. Trovo che la grande guerra sia l'argomento più interessante che ho affrontato quest'anno per via del nuovo modo di combattere e delle ripercussioni sul futuro. Ho sviluppato una piccola web app dove è possibile avere informazioni sulla battaglie svoltesi durante la prima guerra mondiale e un database pubblico sui caduti lombardi della grande guerra. Sono state sviluppate e implementate API RESTful solo di tipo GET e senza contemplare la sicurezza in puro stile REST.

Web Server

Il primo passo è quello di costruire un repository web sulla prima guerra mondiale e delle API per accedervi. Per ottenere questo risultato dovrei configurare una macchina in modo che possa ospitare un semplice web server. Ho utilizzato una macchina virtuale con 512MB di RAM. E' stato scelto Ubuntu 14 Server come sistema operativo.

La macchina è accessibile tramite indirizzo 192.168.1.70 solo dall'interno della LAN dell'istituto

Sul server dovranno essere installati e configurati i seguenti servizi:

- SSH
- Apache
- PHP
- MySQL
- NetFilter

e configurata la seguente utenza:

- Un utente **root** abilitato
- Un utente **tech** con accesso alla console e via SSH
- Un utente **web** con accesso solo via SFTP

SSH

Ssh permette l'accesso alla macchina da remoto, ovvero da altre macchine, in maniera sicura questo perché il canale viene criptato. Il servizio *Secure Shell* dovrà essere configurato con queste specifiche

1. Impedire l'accesso diretto come root

2. Utilizzare un banner di avviso
3. Forzare l'utilizzo della versione 2 del protocollo
4. Accettare connessioni solo per l'utente **tech**
5. Attivare la funzionalità SFTP per l'utente **web**

Dopo l'installazione di ssh configuro il file di configurazione */etc/ssh/sshd.conf*

```
#
# Fedeli 04/03/16 Modifiche significative
#
#Basic Setting
Port 22
Protocol 2
Banner /etc/ssh/banner
PrintMotd yes
AddressFamily inet

# Security
PermitEmptyPasswords no

#Restriction
PasswordAuthentication yes
PermitRootLogin No
AllowUsers tech web
MaxAuthTries 2

# Connection Setting
AcceptEnv LANG LC_*
LoginGraceTime 60

#Log Information SSH
LogLevel DEBUG
SyslogFacility AUTH

# Display
X11Forwarding no

#SFTP Setting
Subsystem sftp internal-sftp
Match User web
    ForceCommand internal-sftp
    ChrootDirectory /var/www
    PasswordAuthentication yes
```

Apache

Apache è una piattaforma server Web modulare. E' in grado di operare su una grande varietà di sistemi operativi. Ho scelto di appoggiarmi a questo software principalmente per la sua elevata diffusione e il fatto che è open source. La sua architettura modulare permette di avere un servizio completamente personalizzabile. Per questo progetto utilizzerò un suo modulo non attivo di default, ovvero `mod_rewrite`. Grazie a questo modulo è possibile reindirizzare, in maniera invisibile all'utente, un link ad una risorsa astratta ad un file presente sul server. Con questo modulo è possibile avere quindi URI tipo

`www.facebook.com/users/stefano.fedeli`

invece di

[www.facebook.com/profile.php?user=stefano fedeli](http://www.facebook.com/profile.php?user=stefano%20fedeli)

Il servizio Web deve rispettare queste specifiche:

1. Consentire solo richieste GET
2. Reindirizzare URL (/api, /documentation)
3. Filtrare API con parametri scorretti

Dopo l'installazione di apache attivo il modulo (*mod_rewrite*) infine inserisco nel file di configurazione */etc/apache2/apache2.conf*:

```
#
# Fedeli 04/03/16 Modifiche significative
#
<Limit PUT POST DELETE>
    order allow, deny
    Deny from all
</Limit>

<FilesMatch \.(inc|conf)>
    order allow, deny
    deny from all
</FilesMatch>

<Files ~ "\.ht">
    Order allow,deny
    Deny from all
    Satisfy All
</Files>

<Location />
    Options -Indexes
</Location>

ServerTokens Prod

AccessFileName .htaccess
```

Creo il mio VirtualHost su Apache

```
<VirtualHost 0.0.0.0:80>
    [...]
    RewriteEngine On

    RewriteRule ^/api/v1/(soldiers|events|battles)/?$ /API/APIv1.php [L]
    RewriteRule ^/api/v1/(soldiers|events|battles)/[0-9]+/?$ /API/APIv1.php [L]
    RewriteRule ^/api/v1/(soldiers|events|battles)/[0-9]+/info/?$ /API/APIv1.php
[L]
    RewriteRule ^/documentation$ /docs.php [L]

</VirtualHost>
```

PHP

Come linguaggio lato server ho optato per il linguaggio PHP, questo perché si integra benissimo con apache ed è il linguaggio più diffuso su server Linux. Una volta installato è

stato necessario configurarlo in modo da:

1. Disabilitare funzioni potenzialmente pericolose
2. Evitare che mostri errori nel browser
3. Disabilitare l'inclusione di file esterni al webserver

Aggiungo al *php.ini*

```
#
# Fedeli 04/03/16 Modifiche significative
#
expose_php=Off
display_errors=Off
log_errors=On
error_log=/var/log/apache2/php_scripts_error.log

file_uploads=Off
allow_url_fopen=Off
allow_url_include=Off

max_execution_time = 30
max_input_time = 30
memory_limit = 40M

; Limits the PHP process from accessing files outside
open_basedir="/var/www/html/"

default_charset="utf-8";
```

MySQL

Come motore del mio repository ho scelto il freeware MySQL, ovvero un database relazionale basato su SQL. Il database verrà ospitato sulla stessa macchina e deve quindi essere configurato per accettare connessioni solo da *localhost*.

```
#
# Fedeli 04/03/16 Modifiche significative
#
[client]
port                = 3306
socket              = /var/run/mysqld/mysqld.sock

[...]
bind-address        = 127.0.0.1
```

Netfilter

Grazie al programma iptables di linux applichiamo un piccolo firewall che rispetti queste specifiche:

1. Bloccare tutte le connessioni in entrata
2. Consentire le connessioni in entrata e uscita con SSH,HTTP
3. Prevenire attacchi DoS e SYN
4. Permettere la risposta ai ping

```
#
# Fedeli 30/05/16 Modifiche significative
```

```

#
# Generated on Mon May 30 09:50:40 2016
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [23:5847]
:ATTACKS - [0:0]
:BLACKLIST - [0:0]
:LOGDROP - [0:0]
:TCP - [0:0]

#####
# INPUT CHAIN #
#####
-A INPUT -i lo -j ACCEPT
-A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
# Prevent DoS and SYN Attacks
-A INPUT -j ATTACKS
# Filter Packet by Source
-A INPUT -j BLACKLIST
# Accept Traffic
-A INPUT -j TCP
# Log and Drop the rest of packet
-A INPUT -j LOGDROP

#####
# ATTACKS CHAIN #
#####
-A ATTACKS -m limit --limit 1/s --limit-burst 3 -j RETURN
-A ATTACKS -p tcp --dport 80 -m limit --limit 50/minute --limit-burst 200 -j
RETURN
-A ATTACKS -j DROP

#####
# BLACKLIST CHAIN #
#####

#####
# TCP CHAIN #
#####
-A TCP -m state --state NEW -p tcp --dport 22 -j ACCEPT
-A TCP -m state --state NEW -p tcp --dport 80 -j ACCEPT
-A TCP -m state --state NEW -p tcp --dport 1024 -j ACCEPT
-A TCP -p icmp -m limit --limit 1/sec -m icmp --icmp-type 8 -j ACCEPT
-A TCP -j RETURN

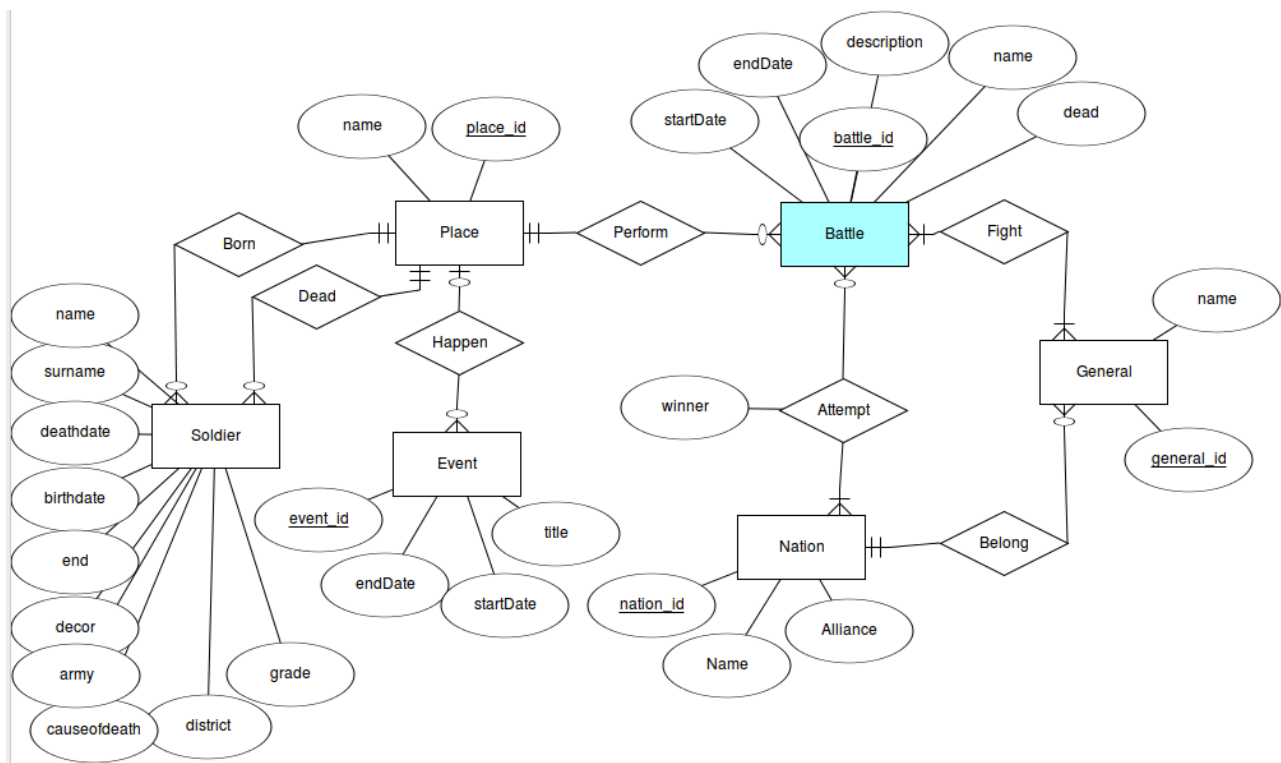
#####
# LOGDROP CHAIN #
#####

-A LOGDROP -j LOG --log-prefix "INPUT:DROP:" --log-level 6
-A LOGDROP -j DROP

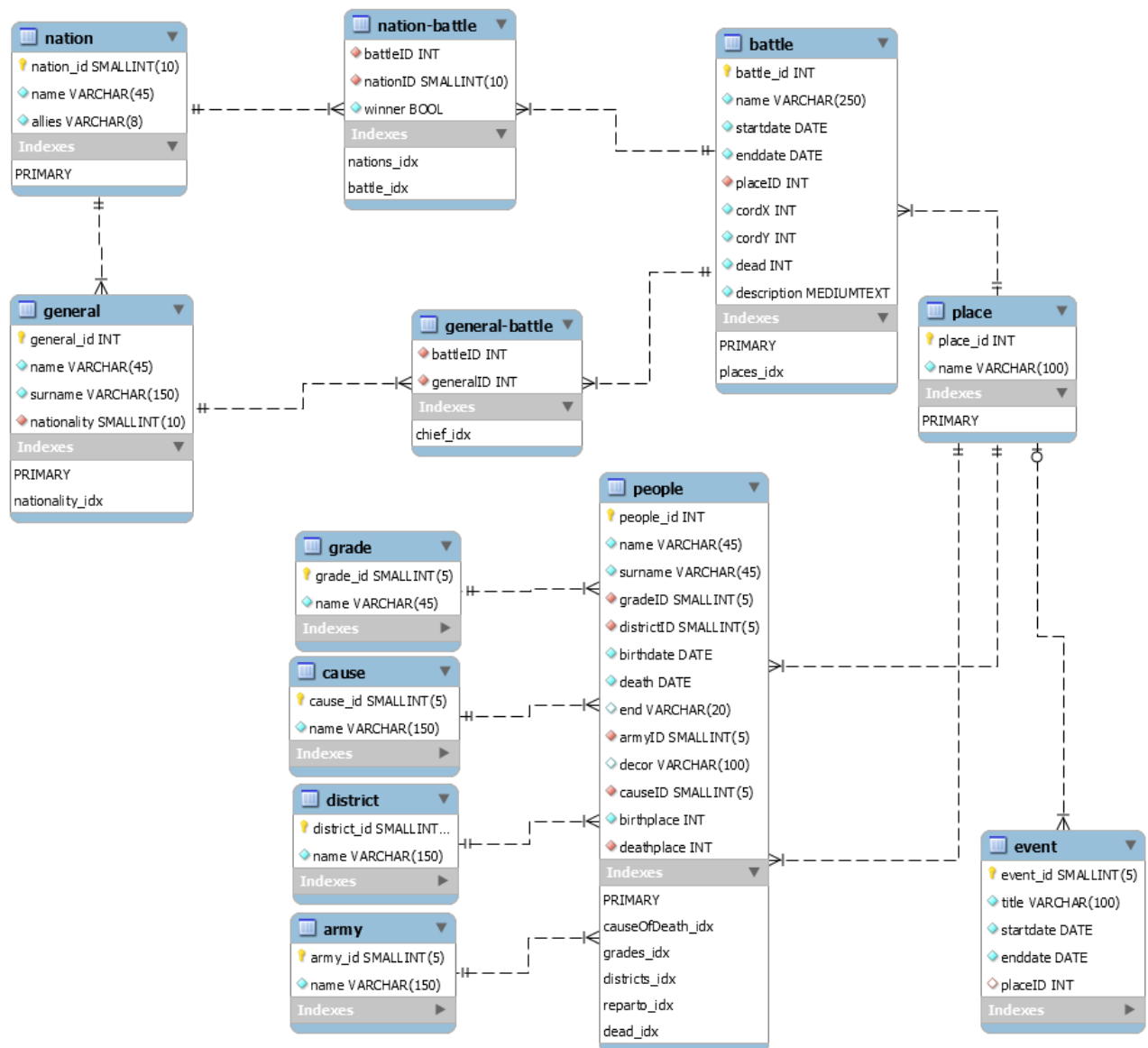
```

Risorse

Ho deciso quindi di creare un repository che contenga diverse informazioni sulle battaglie che si sono svolte durante la prima guerra mondiale e informazioni sui caduti lombardi della grande guerra. Partendo dal diagramma concettuale



sono poi arrivato al diagramma relazionale



Essendo il mio un sistema RESTful ho deciso di rendere disponibili tramite API tre tipi di risorse astratte:

- ◆ Battle
- ◆ Event
- ◆ Soldier

Queste risorse sono certamente derivate dalle entità presenti nel database ma potrebbero non avere la stessa struttura delle relative tabelle. Ad esempio la risorsa soldier può avere una struttura diversa dalla tabella people e, ad esempio, non avere un campo per la data di nascita.

La mie risorse verranno fornite in formato JSON, più leggero e compatto rispetto ad XML.

Api

Sono poi passato alla creazione delle API in linguaggio PHP.

Tutte le URI con prefisso `/api` saranno quindi reindirizzate ad un file presente sul mio server web che contiene il codice PHP che genererà su richiesta la risorsa.

Il servizio accetta richieste HTTP formulate in questo modo

Header	Value
Accept	<i>application/json oppure text/html</i>
Accept-Encoding	<i>gzip</i>
Accept-Language	<i>It oppure en</i>
Host	<i>192.168.1.70</i>
Range (optional)	<i>items=n-m</i>
If-None-Match (optional)	<i>ETag value</i>

Il server risponderà con questo formato

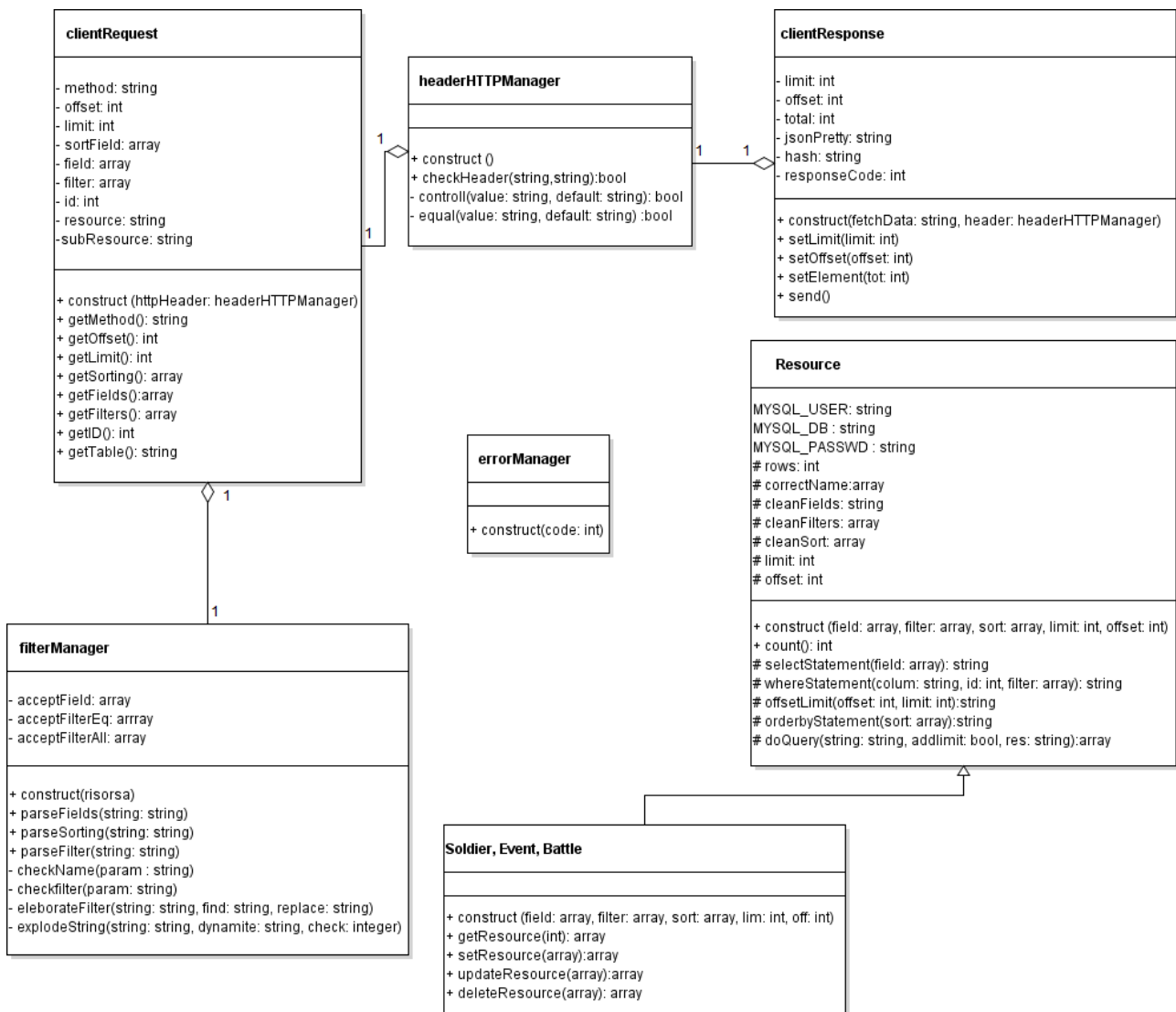
Header	Value
Cache-Control	<i>max-age=3600, public</i>
Connection	<i>close</i>
Content-Encoding	<i>gzip</i>
Content-Language	<i>it</i>
Content-Length	<i>n</i>
Content-Range	<i>items=0-n/tot</i>
Content-Type	<i>application/json</i>
Content-Encoding	<i>gzip</i>
ETag	<i>Hash della risorsa</i>

Le risorse sono accessibili da chiunque e utilizzano il formato

```
192.168.1.70/api/v1/{tipo risorsa}/{id}/{sotto-risorsa}}?  
[field={nomecampo,nomecampo}&filter={filtro1=valore1,filtro2>valore2}&sort={+-  
campo}]
```

E' possibile perciò richiedere specifici campi, ordinare, filtrare e personalizzare l'output.

Il codice PHP è strutturato in classi così definite:



Lavorare con un PHP a classi mi permette di mantenere inalterata la struttura delle API anche se dovessero cambiare le risorse.

Arrivati a questo punto è stato creato un sistema RESTful a tutti gli effetti.

Ora è necessario sviluppare dei client che appoggiandosi alle API appena costruite visualizzino le risorse.

La mia scelta è ricaduta su una web application perché mi permetteva di sfruttare conoscenze che ho imparato in questi anni di scuola (avrei potuto creare un'applicazione

Android o iOS senza problemi di compatibilità).

Web Application

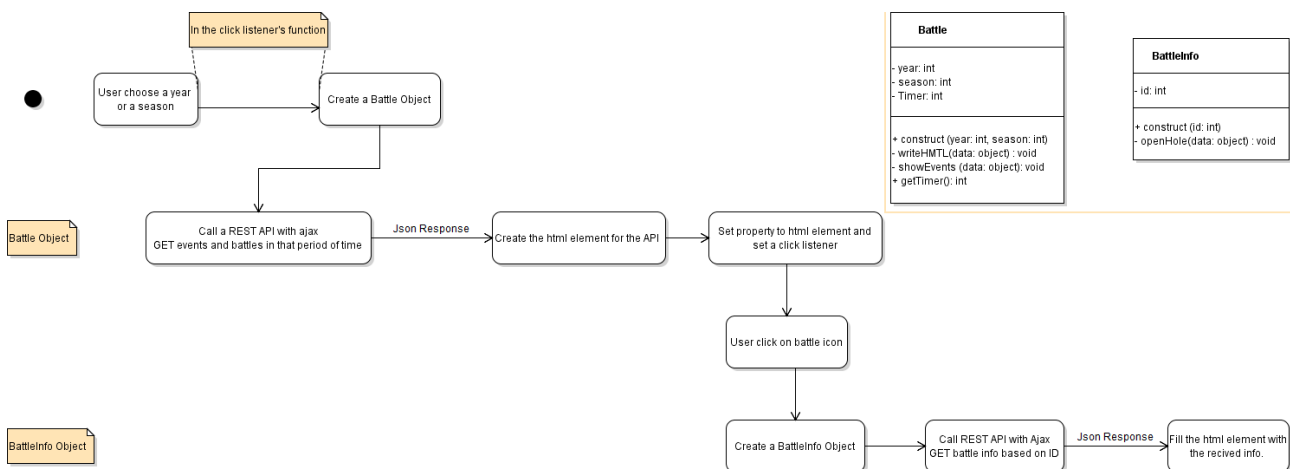
Una web application è un applicazione che è accessibile per mezzo di un browser appoggiandosi ai consueti protocolli di rete. Esempi di web-app sono ad esempio Office 365 di Microsoft o Google Docs. Avendo a disposizione tre tipi di risorse ho scelto di creare:

- Una sezione dove saranno mostrati gli eventi e dopo sia possibile navigare tra le varie battaglie con le informazioni a riguardo.
- Una feature che permette di sapere quante persone sono morte nelle vicinanze.

Storia – Prima Guerra Mondiale

La sezione principale è quindi quella legata alle battaglie. Il progetto è quello di mostrare una mappa dove verranno visualizzate le battaglie relative al periodo selezionato. Se viene selezionata una battaglia verranno mostrate tutte le informazioni disponibili a riguardo.

Ho sviluppato l'intera interfaccia (immagini e animazioni) in HTML5 e CSS3 lasciando la gestione degli eventi e l'interazione con le API a javascript sfruttando la libreria JQuery e tecnologia AJAX. Sono state create due classi, una per la gestione delle battaglie proiettate sulla mappa e una per la gestione dell'evento selezionato. Il workflow complessivo è rappresentato in figura:





Il sistema si può rivelare molto utile a scopo didattico perché permette di avere una visione grafica e geografica della guerra, approfondendo le singole battaglie con un semplice click.

Italiano – Giuseppe Ungaretti

Per rendere più interessante l'esperienza dell'utente ho deciso di inserire una sezione dedicata a Giuseppe Ungaretti. Al centro di questa sezione ho inserito una piccola chat testuale con un chatterbot (AI) che simula il poeta italiano Giuseppe Ungaretti.

Un **chatterbot** è un software progettato per simulare una conversazione intelligente con esseri umani tramite l'uso della voce o del testo, e vengono usati soprattutto come assistenti virtuali (Siri, Cortana, Doretta).

Ho scelto di utilizzare ChatScript come motore del mio chatterbot. Questo motore è stato sviluppato da Bruce Wilcox ed è giunto alla versione 6.2 grazie all'aiuto della comunità open source. Le capacità di questo motore sono molteplici, ma io utilizzerò solo le funzioni basilari. Il processo di elaborazione è il seguente:

1. Il sistema riceve una stringa in input e la suddivide nelle singole parole.
2. Ogni parola è processata per individuare il suo significato e la sua forma canonica.
3. Vengono poi associati ad ogni parola i *concepts* collegati a essa.
4. Vengono processati i *patterns* disponibili in cerca di una risposta adeguata.
5. Il motore restituisce una stringa in risposta.

Alle base di questo motore si trova un dizionario (attualmente solo inglese) dove il motore va a cercare il significato e la forma canonica delle parole ricevute in input (1-2).

I *concepts* sono i concetti astratti legati alla parola. Ad esempio la parola *dog* sarà collegata ai concetti *~pets ~mammals*. Gli stessi concetti possono essere collegati ad altri concetti creando quindi una imponente struttura. Esempio *~mammals* fa parte del *concept ~animals*. Esistono già dei *concepts* caricati nel sistema ma è possibile crearne di nuovi. (3).

Un *pattern* è uno schema, una struttura, che verrà consultata dal motore per trovare un corrispondenza con la frase in input. Un *pattern* potrebbe essere composto da una sequenza di parole o concetti. Se la frase in input segue lo stesso schema allora verrà utilizzata la risposta legata a quel *pattern*. I *patterns* sono l'elemento fondamentale delle regole. Ogni regola è formata da un *pattern* e da una o più risposte. Le regole vengono analizzate in sequenza, esattamente come le regole di un firewall, appena c'è corrispondenza il processo si blocca e si invia la risposta. Le regole sono raggruppate in file detti *topic* (.top). Ogni *topic* è associato ad un *concept*, se quindi nella frase è presente una parola collegata a quel *concept* il *topic* verrà attivato. Il bot andrà a controllare solo le regole dei *topic* attivi. Nel caso non si trovi nessuna corrispondenza il motore reagisce scegliendo a caso una regola preceduta da *t:*; Quest'ultime sono regole speciali che non possiedono un *pattern* ma solo una risposta. (4-5)

Un esempio di Topic

TOPIC: ~introductions keep repeat (~emogoodbye ~emohello ~emohowzit name here)

t: I am very happy to talk to someone in a long time

a:(~emohappy) You're nice

a:(~agree) It is a pleasure

a:(~disagree) You're rude.

t: Do you like this website?

a: (~agree) I think that too, It was designed and developed by a student named Stefano Fedeli

a: (~disagree) Why you don't like it?

b:(*) \$feedback = * Thank for tell me that

t:(\$feedback) Did you change your mind about this site?

u: WHOAMI (what is your name) [My name is Giuseppe Ungaretti] [I am Giuseppe Ungaretti].

u: (who are you) ^reuse(WHOAMI)

?: (Ungaretti) ^reuse(WHOAMI)

?: (Giuseppe) ^reuse(WHOAMI)

?: ((you are)>) The only thing I know is that I'm Giuseppe Ungaretti

s: (~emohello) [Hi] [What's up?] [Hello] [Glad to hear you]

u: STATUS (~emohowzit) [Well, as far as it could be a dead man] [I'm fine but I'm dead]

?: ({how} can *1 speak) I live inside the computer networks. I haven't a body.

u: (what {are} (you do)) [I stay here waiting for new people] [I'm talking to you] [None of your business]

u: USERNAME (I am _*1) \$user_name=_0 Nice to meet you.
u: (my name is _*1) ^reuse(USERNAME)
u: (I_am _*1) ^reuse(USERNAME)
?: ({what} is *~2 name) You are \$user_name. I got some good memory.
u: (I love I) So, you are a little egocentric.
u: (what are you >) [I am a bot] [I am a poet, a unanimous cry. I am a lump of dreams]
s: (you are [smart intelligent brave good]) [A lot of people told me that] [You're sweet] [You too]
u: (~agree) [I'm really smartahaha] [I'm glad you share my thoughts]
u: (~disagree) [everyone has their own opinion] [The world is beautiful because various]
?: (when are you dead) I left this world in June 1970.
?: ([what where]) Good question.

Grazie a questo motore è stata sviluppata Suzette, un chatbot che nel 2010 ha vinto il premio The Loebner una competizione annuale di intelligenza artificiale che premia il bot il cui comportamento è più simile al pensiero umano tramite un semplice test di Turing. Suzette aveva alle spalle 16000 regole, il bot sviluppato da me ne possiede meno di 200. E' chiaro che dopo un allenamento di un solo mese non è capace di reggere un discorso complesso ma è comunque in grado di rispondere ad alcune domande sulla sua vita e sulla prima guerra mondiale.

Il servizio una volta attivato crea un demone in ascolto sulla porta 1024 che si aspetta stringhe in questo formato

```
{userip}\x00{botname}\x00{message}\x00
```

la documentazione fornita da ChatScript fornisce già il codice per una pagina PHP che si collegherà al server e stamperà la risposta in un form.

Essendo la mia applicazione sviluppata interamente in JavaScript ho scritto un pezzo di codice che si occupa di richiamare asincrona-mente la pagina php già disponibile inviando i dati necessari e di stampare i dati ricevuti da essa in modo opportuno.

Storia – Caduti Lombardi

L'ultima sezione della web-application è dedicata ai caduti della prima guerra mondiale. Quando viene ultimato il caricamento della pagina, un piccolo pezzo di codice javascript forza i moderni browser a richiedere l'accesso alla posizione geografica dell'utente. Se viene consentito l'accesso verranno utilizzate le coordinate ricevute per individuare il numero dei caduti nati nelle vicinanze.

Open Database

E' presente anche una seconda pagina che ho dedicato alla documentazione sull'utilizzo delle REST API relative ai soldati lombardi caduti durante la prima guerra mondiale. Questo per evidenziare un'altra implementazione di REST ovvero rendere disponibili risorse e informazioni pubblicamente in un formato comune e universale processabile da macchine.

Conclusione

Lo sviluppo di applicazioni con un architettura REST è chiaramente un vantaggio per tutti i soggetti coinvolti. Nonostante possa risultare ostico ad un primo impatto, ritengo sia molto importante per la crescita di un programmatore confrontarsi con queste realtà. Prendendo poi in considerazione la mia applicazione, penso possa essere un buon strumento di integrazione allo studio con le opportune modifiche. Ho lasciato il chatterbot come argomento secondario per il poco lavoro effettuato da me. Allenato come si deve può diventare un fantastico aiuto agli studenti, anche per avvicinarli alla materia.