REST WEB API: LINEE GUIDA

Introduzione

Nello sviluppo di applicazioni web è sempre più frequente l'utilizzo di API offerte da soggetti e società esterne: API per il social network, API per pagamenti, API per recuperare dati che non detieni o per usufruire di servizi offerti da altri. Anche se tu non stai scrivendo API per altri programmatori, è sempre buona norma per la tua applicazione avere delle API coerenti che seguono determinate regole.

Esistono delle linee guida e regole da seguire quando si progettano delle API in modo che siano logicamente comprensibili anche da chi non le ha sviluppate. L'obiettivo di questo articolo è quello di fornire un insieme di linee guida attraverso punti chiave / suggerimenti su come progettare API. Il target previsto sono invece programmatori o software architect, e l'unico prerequisito per poter comprendere è avere una conoscenza base del protocollo HTTP, e i concetti base dell'architettura RESTful e API WEB. L'articolo non richiederà la conoscenza di nessun particolare linguaggio di programmazione o infrastruttura di rete: tutti i principi e le indicazioni descritte sono applicabili a qualsiasi linguaggio o framework per lo sviluppo di API RESTful.

DOCUMENTAZIONE

Quando cerco sul web delle API da utilizzare in un mio progetto la prima cosa su cui faccio attenzione è se le API che voglio utilizzare hanno una documentazione di ottima qualità; se quel progetto non offre una documentazione chiara e con esempi di utilizzo scarto subito quel progetto di API. Quando si progetta delle API occorre assolutamente fornire

agli utilizzatori la documentazione delle API:

1. Descrizione del servizio offerto dalla API
2. Url della API
3. Elenco degli eventuali parametri o body payload accettati in input, riportando dettagliatamente la struttura che devono avere e il significato semantico del parametro
4. Struttura della risposta della API sia in caso di successo che di errore
5. Significato dei diversi stati HTTP della risposta della API. Ad esempio se un API che utilizzo mi risponde con l'errore 404 NOT FOUND, mi aspetto che nella documentazione sia riportato in quali occasioni viene sollevato questo errore.

Esistono diversi Tool riconosciuti come standard dalla community per la produzione di documentazione di API. Riporto quelli che a mio avviso sono i migliori:

Io personalmente uso Swagger, in quanto recentemente ha lanciato la **Specifica OpenAPI** (conosciuta originariamente come la Specifica Swagger), una specifica per file di interfaccia leggibili dalle macchine per descrivere, produrre, consumare e visualizzare servizi web RESTful.[1] Con i tool di Swagger hai a disposizione una serie di tool per:

- generare il codice dei tuoi model, e non solo, a partire dalla specifica delle API
- generare la documentazione
- generare test case

Ultimamente utilizzo molto Swagger, il quale è adatto anche all'uso interno in un team di sviluppo, per avere delle specifiche delle API utili sia per chi sviluppa sia al team frontend che le utilizza. Tratterò di Swagger in uno dei prossimi articoli.

Alcuni esempi di API con una grande documentazione sono:

Nome delle risorse e url design

TERMINOLOGIA

Riportiamo i termini ricorrenti nel mondo delle REST API:

- **Risorsa:** è un oggetto o la rappresentazione di qualcosa di significativo nel dominio applicativo. È possibile interagire con le risorse attraverso le API. Esempi di entità? Un Libro, un Ordine, un Post e qualsiasi altra entità che si può astrarre da un determinato contesto. Il concetto di risorsa è quindi molto simile a quello di oggetto nel mondo della programmazione ad oggetti. Prima di mostrare esempi voglio innanzitutto chiarire che il nome delle risorse, documentazione e nome URL sono da standard e linee guida in INGLESE; vi chiedo quindi rispettare questa regola.
- **Collezioni:** sono insiemi di risorse. Ad esempio *People* è una collezione di risorse Person.
- **URL (Uniform Resource Locator)**: è il path attraverso la quale una risorsa può essere ottenuta e dove possono essere eseguite azioni su di essa.

Facciamo in dettaglio l'esempio di una rappresentazione dell'entità Persona **(Person)**. Una persona ha un nome, un cognome , un sesso e un'età. Una rappresentazione JSON della risorsa Person potrebbe quindi essere

```
{ "id":1, "firstname":"Marco", "lastname":"Rossi", "sex":"M",
"age":28,
...
}
```

DAI UN ID UNIVOCO ALLA TUA RISORSA

E' assolutamente buona norma includere un campo id la quale

rappresenta in modo univoco un'istanza di una determinata risorsa. Tale campo, suggerisco di chiamarlo id, deve essere il primo campo in ordine in una rappresentazione JSON della risorsa.

USA IL PLURALE PER L'URL DELLA RISORSA

È possibile interagire con le risorse tramite delle API identificate da un URL. È assolutamente buona norma utilizzare una convenzione riconosciuta globalmente per il nome dell'url: gli URL devono chiamarsi con il nome plurale della risorsa che permettono di interagire, nel nostro esempio gli url per interagire con la risorsa Person dovrebbero chiamarsi **/people** (in italiano "persone"). Altri esempi di URL: **/orders, /students, /books.**

Sono assolutamente quindi vietati url del tipo getOrders/, getBooks/, ecc. L'URL corretto è orders/. I più inesperti quindi si chiederanno: se chiamo sia l'URL per creare una risorsa che l'URL per recuperare una risorsa con lo stesso nome, non è motivo di confusione? Assolutamente NO! Il tipo di operazione lo identifica il verbo del metodo (ad es. GET per ottenere la risorsa e POST per crearla). Questo tema verra' approfondito sotto.

USA L'ID PER ACCEDERE ALLA SINGOLA RISORSA

Per il design dell'url di una singola risorsa è buona norma utilizzare l'id univoco nella forma /{collection}/{resource id} . Ad esempio:

- /books/abcbc
- /people/1
- /users/dario_frongi
- /people/dhdhdhd

url base e versioning

Supponiamo che tu sia una sviluppatore di API e che tu abbia pubblicato delle API. Dopo alcuni mesi decidi di rinominare il nome di un parametro della tua API e di inserirne altri per rivoluzionare il comportamento della tua API. Cosa accade? Chiunque utilizzava le tue API non sarà più in grado di utilizzarle. La soluzione a questo problema è quello di utilizzare il versioning delle API come base dell'url. Supponiamo che al lancio del mio progetto le mie api abbiano tutte come URL base: https://api.myservice.it/v1. Ad esempio https://api.myservice.it/v1/orders per ottenere gli ordini dalla mia API. Se io introduco una **breaking change** (una modifica impattante alla API che non garantisce il corretto funzionamento se invocata come nella vecchia maniera) non cancellerò dal mio progetto la API v1/orders, ma creerò un altra API v2/orders comunicando nella documentazione del progetti di migrare al più presto alla versione v2/orders della API.

METODI HTTP

Il protocollo HTTP fornisce diversi metodi per implementare il prinicipio CRUD (Creae Read Update Delete); ogni diversa funzionalità che voglio fornire per permettere l'interazione con le risorse del mio dominio deve essere mappata con il giusto metodo HTTP:

- Usa POST per creare
- Usa GET per leggere
- usa PUT Per aggiornare
- usa DELETE per eleminare

codici HTTP di risposta

Quando un client effettua una richiesta al server attraverso una API, il client dovrebbe saper interpretare il feedback. Il server risponde sempre con un codice HTTP: codici standard che

hanno un determinato significato. È importante quindi utilizzare i codici HTTP in modo consono ed universale. Riepilogo i codici HTTP di solito utilizzati:

2xx (categoria CODICI DI SUCCESSO)

I codici inizianti con il numero 2 sono codici utilizzati per confermare che la richiesta è stata ricevuta e processata con successo dal server.

- **200 Ok:** La risposta standard per rappresentare una operazione di tipo GET, PUT or POST (per azioni e non creazione risorsa) avvenuta con successo.
- **201 Created:** Questo codice andrebbe utilizzato lato server per comunicare che la nuova istanza è stata creata con successo. Ad esempio nella creazione di una nuova risorsa usando il metodo POST.
- **204 No Content:** significa che la richiesta è stata correttamente processata, ma non ha ritornato nessun contenuto. DELETE è un ottimo esempio di utilizzo di questo codice.

3xx (categoria di Redirection)

- **304 Not Modified:** indica che il client ha già in cache la risposta e non è necessario reinviarla.

4xx (categoria di errore client)

Questa categoria di errore include errori correttamente gestiti dal server e causati da uno scorretto input della API.

- **400 Bad Request:** indica che la richiesta non è stata processata perché il server non riconosce la richiesta come una tra quelle attese: ad esempio url errata o nome parametro errato.
- **401 Unauthorized:** indica che il client non ha i privilegi per accedere a tale risorsa (la API richiesta richiede l’autenticazione o un token di autenticazione).

- **403 Forbidden:** indica che la richiesta effettuata è valida e il client è correttamente autenticato, ma non ha i privilegi per poter utilizzare questa API
- **404 Not Found:** indica che la risorsa richiesta non esiste o è momentaneamente non recuperabile.
- **410 Gone:** indica che la risorsa richiesta non è disponibile e intenzionalemente spostata.

5xx (categoria di errore server)

Questa cateoria di errori rappresentano situazioni non correttamente gestite come eccezioni lato server oppure servizi momentaneamente non disponibili.

- **500 Internal Server Error:** indica che la richiesta è valida, ma il server ha riscontrato un errore non gestito (ad es. in java una eccezione non correttamente catturata).
- **503 Service Unavailable:** indica che il server è offline e non può quindi processare richieste.

Dopo aver riportato le linee guida per la naming convention delle URL e dei codici HTTP di risposta riportiamo alcuni esempi di API.

crea una risorsa persona (post http)

Per creare una nuova risorsa sul server le tue API dovrebbero prevedere un endpoint di metodo POST. Il body della richiesta di solito contiene le info dell'oggetto che si vuole creaee ad eccezione dell'id, il quale solitamente è valorizzato lato server.

Un esempio di richiesta POST:

```
POST /v1/people HTTP/1.1
Host: https://api.myapi.it
Content-Type: application/json
{ "firstname": "Dario", "lastname": "Frongi", "age":30
}
```

La risposta a una richiesta POST dovrebbe essere di codice HTTP 201 per confermare la creazione, con body uguale alla rappresentazione della risorsa creata.

LEGGI UNA RISORSA CON GET

Per ottenere una risorsa le tue API dovrebbero prevedere un endpoint di metodo GET.

Un esempio di richiesta GET:

```
GET /v1/people HTTP/1.1 RISPOSTA
HTTP/1.1 200 OK
Content-Type: application/json
[
{ "id":1, "firstname": "Dario", "lastname": "Frongi", "age":30
},
{ "id":2, "firstname": "Mario", "lastname": "Rossi", "age":30
}
]
```

aggiorna UNA RISORSA CON put

Per aggiornare una risorsa le tue API dovrebbero prevedere un endpoint di metodo PUT.

La risposta a una richiesta PUT dovrebbe essere di codice HTTP 200 per confermare l'aggiornamento avvenuto con successo, con body uguale alla rappresentazione della risorsa aggiornata.

Un esempio di richiesta PUT per aggiornare il nome della risorsa con id '1' a "Mario"

```
PUT /v1/people/1 HTTP/1.1
Host: https://api.myapi.it
Content-Type: application/json
{ "firstname": "Mario",
}
```

elimina UNA RISORSA CON delete

Per eliminare una risorsa le tue API dovrebbero prevedere un endpoint di metodo DELETE.

La risposta a una richiesta DELETE dovrebbe ressere di codice HTTP 204 per confermare l'avvenuta eliminazione e e un body vuoto.

Un esempio di richiesta DELETE per eliminare la risorsa person con id 1.

```
DELETE /v1/people/1 HTTP/1.1
Host: https://api.myapi.it
```

parametri di ricerca, ordinamento dati e paginazione

Il client potrebbe voler effettuare delle vere e proprie query utilizzando le URL della API. In fase di design delle API occorre quindi prevedere dei parametri query che possono arrivare sulla URL, e che quindi vanno correttamente gestiti. Di solito con tali parametri si gestisce:

- **Ordinamento:** qualora il client volesse ordinare le risorse ottenute dalla API come ad esempio la lista delle persone ordinate per nome, l'API GET /people accettera' il parametro query sort nella URL. Ad esempio: GET v1/people?sort=firstname_asc
- **Filtri:** per filtrare il dataset le API di solito prevedono alcuni parametri query nella URL. Ad esempio per ottenere le persone con il nome dario di età 30 anni: GET v1/people?name=dario&age=30
- **Pagination:** quando il numero di risorse di una collezione è molto grande si dividono le risorse di una collezione in porzioni chiamate "pagine", in modo da ridurre l'overhead sulla rete (immaginate di dover far passare sulla rete mobile 2000 risorse!). Un esempio:

> `GET /people?page=2&size=20` significa che le risorse vengono divise in pagine di 20 persone e io voglio ottenere i dati della seconda pagina (o terza a seconda se consideriamo 0 come indice di pagina).

CONCLUSIONE

In questo articolo abbiamo dato delle semplici linee guida che andrebbero sempre seguite nel design di API REST. In un prossimo articolo approfondiremo alcuni temi trattati solo di sfuggita come l'autenticazione e il caching.

Stay tuned.

https://italiancoders.it/rest-web-api-linee-guida/