Crea API che non odierai

By Phil Sturgeon

tradotto da
Damiano Venturin

Crea API che non odierai

Ormai tutti vogliono utilizzare le API quindi é decisamente una buona idea imparare a costruirle.

Phil Sturgeon and Damiano Venturin

This book is for sale at http://leanpub.com/crea-api-che-non-odierai

This version was published on 2014-05-06

Tweet This Book!

Please help Phil Sturgeon and Damiano Venturin by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Ho appena comprato ”Crea API che non odierai”. Non vedo l’ora di leggerlo! http://bit.ly/1bqkhri

The suggested hashtag for this book is #creaapichenonodierai.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#creaapichenonodierai

Indice

1 HATEOAS

1.1 Introduzione

HATEOAS è un argomento difficile da spiegare, ma, di per sé, è piuttosto semplice. É l'acronimo di "Hypermedia as the Engine of Application State" ("Hypermedia come Motore dello Stato dell'Applicazione"), e si pronuncia "hat-ee-os" oppure "hate O-A-S" oppure ancora "hate-ee-ohs"- che suona un po' come una marca di cereali per sviluppatori di API.

Comunque lo si voglia chiamare, significa fondamentalmente due cose per la tua API:

1. Negoziazione dei contenuti (Content negotiation)
2. Controlli Hypermedia (Hypermedia controls)

Secondo la mia esperienza la negoziazione dei contenuti è una delle prime cose che gli sviluppatori API implementano. Quando ho costruito la mia estensione Rest-Server per CodeIgniter è stata la prima cosa che ho fatto, perché … è divertente! Modificare l'intestazione `Accept` e veder il `Content-Type` nella risposta passare da JSON a XML o CSV è fico e anche molto facile da fare.

1.2 Negoziazione del contenuto

Alcune API che si auto proclamano RESTful (Twitter, ce l'ho con te) gestiscono la negoziazione dei contenuti mediante l'estensione file. Fanno cose tipo `/statuses/show.json?id=210462857140252672` e `/statuses/show.xml?id=210462857140252672` invece di `/statuses/210462857140252672` e lasciare che sia l'intestazione `Accept` a fare il lavoro.

Gli URI non dovrebbero essere una manciata di cartelle e di nomi di file , e un'API non è un elenco di file JSON o file XML. Si tratta di un elenco di risorse che possono essere rappresentate in diversi formati a seconda dell'intestazione `Accept`, **e nient'altro!**

Un semplice esempio di una corretta negoziazione RESTful di contenuti che richiede JSON

```
GET /places HTTP/1.1
Host: localhost:8000
Accept: application/json
```

La risposta dovrebbe quindi contenere JSON, se l'API supporta JSON come formato di output.

Un breve esempio di risposta HTTP con dati JSON

```
HTTP/1.1 200 OK
Host: localhost:8000
Connection: close

{
    "data": [
        {
            "id": 1,
            "name": "Mireille Rodriguez",
            "lat": -84.147236,
            "lon": 49.254065,
            "address1": "12106 Omari Wells Apt. 801",
            "address2": "",
            "city": "East Romanberg",
            "state": "VT",
            "zip": 20129,
            "website": "http://www.torpdibbert.com/",
            "phone": "(029)331-0729x4259"
        },
        ...
    ]
}
```

Molte API RESTful supporteranno JSON come impostazione predefinita (default), o semplicemente *solo* JSON come ha fatto finora l'applicazione di esempio di questo libro anche se non è molto realistico (è stato fatto soprattutto per semplicitá).

XML è abbastanza difficile da produrre poiché esige dei file di vista (view), argomento che è al fuori del campo di applicazione di questo capitolo.

YAML è, invece, piuttosto facile da produrre, e posso mostrare come funziona la negoziazione dei contenuti, con un piccolo aggiustamento della la nostra app. Guarda in `~/apisyouwonthate/chapter12/` per vedere in dettaglio le modifiche che ho apportato.

La modifica principale, oltre ad includere il componente YAML per Symfony[1] é stata quella di aggiornare l'`ApiController::respondWithArray()` per controllare l'intestazione `Accept` e reagire di conseguenza.

[1] http://symfony.com/doc/current/components/yaml/introduction.html

```php
protected function respondWithArray(array $array, array $headers = [])
{
    // É possibile che tu voglia fare qualcosa di intelligente con il charset, \
 se fornito.
    // Questo capitolo, semplicemente, lo ignora e considera solo il valore de\
l mime-type

    $mimeParts = (array) explode(';', Input::server('HTTP_ACCEPT'));
    $mimeType = strtolower($mimeParts[0]);

    switch ($mimeType) {
        case 'application/json':
            $contentType = 'application/json';
            $content = json_encode($array);
            break;

        case 'application/x-yaml':
            $contentType = 'application/x-yaml';
            $dumper = new YamlDumper();
            $content = $dumper->dump($array, 2);
            break;

        default:
            $contentType = 'application/json';
            $content = json_encode([
                'error' => [
                    'code' => static::CODE_INVALID_MIME_TYPE,
                    'http_code' => 415,
                    'message' => sprintf('Content of type %s is not supported.\
', $mimeType),
                ]
            ]);
    }

    $response = Response::make($content, $this->statusCode, $headers);
    $response->header('Content-Type', $contentType);

    return $response;
}
```

Molto semplice e specificando un altro mime-type, si avrá un formato diverso:

Una richiesta HTTP che specifica il mime-type preferito

```
GET /places HTTP/1.1
Host: localhost:8000
Accept: application/x-yaml
```

La risposta sarà in YAML.

Un breve esempio di risposta HTTP con dati YAML

```
HTTP/1.1 200 OK
Host: localhost:8000
Connection: close

data:
    - { id: 1, name: 'Mireille Rodriguez', lat: -84.147236, lon: 49.254065, ad\
dress1: '12106 Omari Wells Apt. 801', address2: '', city: 'East Romanberg', st\
ate: VT, zip: 20129, website: 'http://www.torpdibbert.com/', phone: (029)331-0\
729x4259 }
    ...
```

Eseguire queste richieste programmaticamente è semplice.

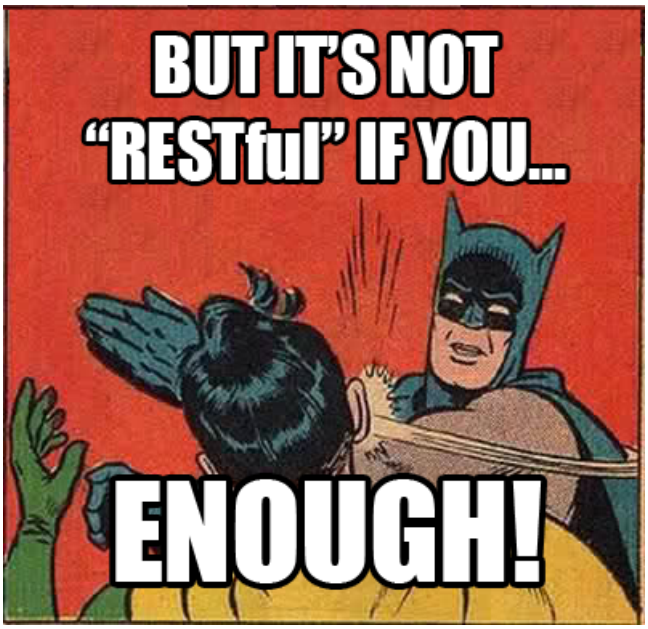
Utilizzo di PHP e del pacchetto Guzzle per richiedere una risposta personalizzata

```
use GuzzleHttp\Client;

$client = new Client(['base_url' => 'http://localhost:8000']);

$response = $client->get('/places', [
    'headers' => ['Accept' => 'application/x-yaml']
]);

$response->getBody(); // YAML, ready to be parsed
```

Questa non esaurisce la conversazione sulla negoziazione dei contenuti, perché c'è ancora qualcosa da dire sui vendor-based mime-type per le risorse, che possono essere soggetti a versioning. Per mantenere questo capitolo in tema, questa discussione verrà ripresa nel Capitolo 13: API versioning.

1.3 Controlli hypermedia (ipermediali)

La seconda parte di HATEOAS, però, è drasticamente sottoutilizzata, ed è il tassello mancante per rendere la tua API una vera API RESTful.



Batman ha una risposta pronta alle osservazioni inutili del tipo "Ma non è RESTful se..." di Troy Hunt (@troyhunt)

Mentre si sentono spesso lamentele del tipo "ma non è RESTful!" che tirano in ballo cose stupide, questa è invece un caso fondato. Senza controlli hypermedia hai solo un'API, non un'API RESTful. Questo è un aspetto in cui il 99% di tutte le API falliscono.

RESTful Nirvana

1. **"La Palude di POX"**. Stai usando HTTP per effettuare chiamate RPC. HTTP è utilizzato, in realtá, come tunnel.
2. **Risorse**. Invece di fare ogni chiamata ad un endpoint del servizio, hai più endpoint che vengono utilizzati per rappresentare le risorse, e per comunicare con loro. Questo è solo l'inizio del vero supporto REST.
3. **verbi HTTP** Questo è ció che Rails, ad esempio, offre nativamente: interagisci con le risorse utilizzando i verbi HTTP, anziché utilizzare sempre POST.
4. **Hypermedia Controls**. HATEOAS. Sei 100% compatibile con REST. –**Fonte**: Steve Klabnik, "Haters gonna HATEOAS"[2]

Quell'articolo è basato su di un altro articolo scritto da Martin Fowler[3] chiamato "Richardson Maturity Model"[4] che spiega un modello scritto da Leonard Richardson[5] che copre ció che lui considera i quattro livelli di maturitá REST.

Allora che cosa sono i controlli hypermedia? Sono solo dei link, alias "collegamenti ipertestuali", ció che hai probabilmente utilizzato per anni in HTML. Sin dall'inizio del libro ho detto che REST utilizza solamente HTTP e le stesse convenzioni usate in internet (anziché inventarne di

[2] http://timelessrepo.com/haters-gonna-hateoas
[3] http://martinfowler.com/
[4] http://martinfowler.com/articles/richardsonMaturityModel.html
[5] http://www.crummy.com/

nuove), quindi ha effettivamente senso che il collegamento ad altre risorse avvenga in un'API esattamente come avviene in una pagina web.

Il tema di fondo di HATEOAS, in generale, è che l'API deve avere perfettamente senso per applicazione client e per l'umano che guarda le risposte, tutto senza dover andare a caccia nella documentazione per capire cosa sta succedendo.

Ho state subdolamente intriso questo libro di concetti HATEOAS, dal suggerire che i codici di errore debbano essere combinati con messaggi di errore leggibili e da collegamenti alla documentazione, all'aiutare l'applicazione client evitandole di far conti quando si interagisce con contenuto paginato. Il tema di fondo è sempre quello di rendere i controlli, come successivo, precedente o qualsiasi altro tipo di interazione, chiari sia ad un essere umano o che a un computer.

Comprendere i controlli hypermedia

Questa è la parte più facile della costruzione di un'API RESTful, e mi sforzeró davvero di non concludere questa sezione scrivendo: "Basta che aggiungi i link, ciccio" - che sarebbe ció che direi a chiunque mi chiedesse di HATEOAS.

Il nostro solito blocco di dati viene ritornato in output in modo che rappresenti una o più risorse. Di per sé, questo blocco dati è completamente isolato dal resto dell'API. A questo punto lo sviluppatore per continuare a interagire con l'API dovrebbe aver letto la documentazione, aver capito quali dati possono essere correlati e dove sono. Non é esattamente il massimo della comoditá.

Legare un `Luogo` alle relative risorse, sotto-risorse o collezioni è facile.

```
{
    "data": [
        "id": 1,
        "name": "Mireille Rodriguez",
        "lat": -84.147236,
        "lon": 49.254065,
        "address1": "12106 Omari Wells Apt. 801",
        "address2": "",
        "city": "East Romanberg",
        "state": "VT",
        "zip": 20129,
        "website": "http://www.torpdibbert.com/",
        "phone": "(029)331-0729x4259",
        "links": [
            {
                "rel": "self",
                "uri": "/places/2"
            },
            {
                "rel": "place.checkins",
                "uri": "/places/2/checkins"
```

```
22              },
23              {
24                  "rel": "place.image",
25                  "uri": "/places/2/image"
26              }
27          ]
28      ]
29  }
```

Qui ci sono tre semplici elementi, il primo é autoreferenziale. Tutti contengono un `uri` (Universal Resource Indicator) e un `rel` (Relazione).

URI vs URL

L'acronimo "URI" è spesso usato per riferirsi al solo contenuto dopo il protocollo, ovvero il nome host e la porta (ovvero URI è: percorso + estensione + stringa di query), mentre "URL" è usato per descrivere l'indirizzo completo. Anche se questa distinzione non è rigorosa, si perpetua in molti progetti software come CodeIgniter. Wikipedia[6] e il W3[7] si contraddicono, ma mi sento dire che un URI è semplicemente un qualsiasi tipo di identificatore della posizione di una risorsa su internet.

Un URI può essere parziale o assoluto. URL è considerato da alcuni come un termine del tutto inesistente, ma questo libro usa URL per descrivere un URI assoluto, che è quello che vedi nella barra degli indirizzi. Giusto o sbagliato che sia. Capito?

Alcuni credono che `self` sia inutile. É ovvio che tu sappia qual é l'URL che hai appena chiamato, ma non é detto che quell'URL coincida sempre quello indicato da `self`. Se, per esempio, hai appena creato una risorsa `Luogo`, avrai chiamato `POST /places` che non è certamente l'URI che vuoi chiamare di nuovo per avere informazioni aggiornate sulla stessa risorsa. Indipendentemente dal contesto, l'output di un `place` necessita sempre di avere una relazione `self` e quel `self` non coincide sempre con l'URL della barra degli indirizzi. In sostanza, la relazione `self` identifica il luogo in cui vive la risorsa, non il corrente indirizzo.

Per quanto riguarda le altre voci `rel`, si tratta di collegamenti alle risorse correlate. Il contenuto dei tag é arbitrario, basta che sia coerente. La convenzione utilizzata in questo esempio è quella di usare namespace univoci per le relazioni. Due diversi tipi di risorse potrebbero avere una relazione `checkin` (ad esempio: `users` e `luoghi`); mantenerle uniche é un beneficio quantomeno per la documentazione. Magari preferisci rimuovere il namespace, ma questo sta a te.

Tali relazioni personalizzate hanno nomi abbastanza univoci, ma per relazioni più generiche puoi considerare di usare il Registry of Link Relations[8] definito da IANA, che é utilizzato da Atom (RFC 4287[9]) e da molte altre cose.

[6] http://wikipedia.org/wiki/Uniform_Resource_Identifier
[7] http://www.w3.org/TR/uri-clarification/
[8] http://www.iana.org/assignments/link-relations/link-relations.xhtml
[9] http://atompub.org/rfc4287.html

Creazione di controlli hypermedia

In questo caso sono stati sparati alcuni link in output. Comunque tu decida di farlo, puoi includerli nel layer di “trasformazione” o “presentazione”.

Se utilizzi il componente Fractal - che è stato utilizzato come un esempio per tutto il libro - puoi fare cosí:

PlaceTransformer con collegamenti inclusi nella risposta.lang=php

```php
public function transform(Place $place)
{
    return [
        'id'           => (int) $place->id,
        'name'         => $place->name,
        'lat'          => (float) $place->lat,
        'lon'          => (float) $place->lon,
        'address1'     => $place->address1,
        'address2'     => $place->address2,
        'city'         => $place->city,
        'state'        => $place->state,
        'zip'          => (float) $place->zip,
        'website'      => $place->website,
        'phone'        => $place->phone,

        'links'        => [
            [
                'rel' => 'self',
                'uri' => '/places/'.$place->id,
            ],
            [
                'rel' => 'place.checkins',
                'uri' => '/places/'.$place->id.'/checkins',
            ],
            [
                'rel' => 'place.image',
                'uri' => '/places/'.$place->id.'/image',
            ]
        ],
    ];
}
```

Alcuni usano automatismi per sembrare piú intelligenti e ritornano le relazioni in base alle impostazioni di `$_SERVER` o sulla base di relazioni ORM, ma questo causa solo problemi. Se usi i transformers, devi solo scrivere qualche riga di codice ed eviterai di esporre la logica del database e otterrai un codice leggibile e comprensibile.

Dopo aver inserito i collegamenti, gli utilizzatori hanno bisogno di sapere come interagirci. Potresti pensare "Ci devo mettere `GET` o `PUT` così la gente sa che cosa fare" Sbagliato. Si tratta di collegamenti a risorse, non di azioni. Abbiamo un'immagine per Luogo, e possiamo supporre ciecamente che possiamo fare determinate azioni su di essa, oppure possiamo chiedere alla'API quali azioni sono possibili e memorizzare il risultato nella cache.

Scoprire le risorse programmaticamente

Se hai un output come questo:

```
{
    "data": [
        ...
        "links": [
            {
                "rel": "self",
                "uri": "/places/2"
            },
            {
                "rel": "place.checkins",
                "uri": "/places/2/checkins"
            },
            {
                "rel": "place.image",
                "uri": "/places/2/image"
            }
        ]
    ]
}
```

Puoi supporre che un `GET` funzionerà sia con l'endpoint `self` che con l'endpoint `place.checkins`, ma cos'altro ci puoi fare? Inoltre, che ci fai con l'endpoint `place.image`?

HTTP corre in nostro aiuto, per rispondere a quelle domande, mediante un verbo semplice ed efficace di cui non ho ancora parlato: `OPTIONS` (`OPZIONI`).

Una richiesta HTTP utilizzando il verbo OPTIONS

```
OPTIONS /places/2/checkins HTTP/1.1
Host: localhost:8000
```

La risposta alla richiesta HTTP precedente

```
HTTP/1.1 200 OK
Host: localhost:8000
Connection: close
Allow: GET,HEAD,POST
```

Ispezionando l'intestazione `Allow`, noi esseri umani (come pure un'applicazione client) siamo in grado di capire quali opzioni sono disponibili per un qualsiasi endpoint. Questo è ciò che spesso fa JavaScript, senza che nessuno lo sappia, per eseguire le richieste AJAX.

La maggior parte dei client HTTP ti permette di fare una chiamata `OPTIONS` con la stessa facilitá con cui effettui una chiamata `GET` o `POST`. Se il tuo client HTTP non te lo consente buttalo e trovane un altro.

Fare una richiesta HTTP OPTIONS utilizzando PHP ed il pacchetto Guzzle

```
use GuzzleHttp\Client;

$client = new Client(['base_url' => 'http://localhost:8000']);
$response = $client->options('/places/2/checkins');
$methods = array_walk('trim', explode(',', $response->getHeader('Accept'));
var_dump($methods); // Outputs: ['GET', 'HEAD', 'POST']
```

Analizzando il risultato, sappiamo che possiamo ottenere un elenco di checkin per un luogo usando `GET`, e possiamo aggiungerne di nuovi facendo una richiesta HTTP `POST` a tale URL. Possiamo anche fare controlli con `HEAD`, che è la stessa cosa di un `GET` mancante del corpo HTTP. `HEAD` è utile per controllare se una risorsa o una raccolta esiste senza dover scaricare l'intero corpo (i.e: vedi se ottieni un `200` o un `404`).

Forse ti puó sembrare pazzesca l'idea di interagire in questo modo con un'API, ma in realtà é molto più facile che scartabellare la documentazione. Pensaci, vai sul sito della documentazione, cerca il microscopico link "Developers" ("Sviluppatori"), passati tutta la documentazione per l'API corretta (perché sono così fighi che hanno 3 API), chiediti se hai la versione giusta... non è divertente. Pensa invece ad un'API auto-documentante, che si aggiorna ed espande automaticamente col tempo: non c'é paragone, fidati.

Se sei certo che un API segue i principi RESTful allora *dovresti* essere sicuro che segue HATEOAS - perché se viene sbandierata come RESTful senza che segua HATEOAS allora é menzogna puzzolente. Purtroppo, dietro alla maggior parte delle API piú popolari si nascondono bugiardi puzzolenti.

> GitHub risponde con un 500, Reddit con 501 Not Implemented, le mappe di Google con 405 Method Not Allowed. Hai capito cosa intendo no?. Ne ho provate molte altre, ed i risultati sono generalmente simili. Talvolta ritornano qualcosa di simile ad una risposta GET. Nessuna API si comporta in modo corretto. – **Fonte**: Zac Stewart, "The HTTP OPTIONS method and potential for self-describing RESTful APIs"[10]

[10] http://zacstewart.com/2012/04/14/http-options-method.html

Se stai costruendo la tua API, allora puoi decidere come si deve comportare e farla come dio comanda cosí i tuoi utenti sapranno che sai uno che sa come costruire un'API decente.

Questo é tutto quanto c'é da sapere su HATEOAS. Ora ne dovresti sapere abbastanza da costruire un'API che non odierai (in teoria). Purtroppo, a prescindere da tutto, dovrai probabilmente scriverne una nuova versione nel giro di pochi mesi (succede sempre cosí) quindi non resta che affrontare il capitolo sulle versioni.