Quick Reference Card

RESTful – API – Design

OCTO Technology

There is a better way

WWW.OCTO.COM

We believe that API IS THE ENGINE OF DIGITAL STRATEGY WE KNOW *that the Web infiltrates* AND *transforms* COMPANIES WE WORK TOGETHER, *with passion,* TO CONNECT BUSINESS & IT *We help you* CREATE OPPORTUNITIES AND EMBRACE THE WEB *Inside & Out*

© OCTO Technology 2015 - All rights reserved

tribu.woapi@octo.com

WWW.OCTO.COM

HTTP Status codes.

HTTP STATUS CODE | **DESCRIPTION**

SUCCESS

200 OK
- Basic success code. Works for the general cases.
- Especially used on successful first GET requests or PUT/PATCH updated content.

201 Created
- Indicates that a resource was created. Typically responding to PUT and POST requests.

202 Accepted
- Indicates that the request has been accepted for processing.
- Typically responding to an asynchronous processing call (for a better UX and good performances).

204 No Content
- The request succeeded but there is nothing to show. Usually sent after a successful DELETE.

206 Partial Content
- The returned resource is incomplete. Typically used with paginated resources.

CLIENT ERROR

400 Bad Request

General error for a request that cannot be processed.

```
GET /bookings?paid=true
→ 400 Bad Request
→ {"error":"invalid_request", "error_description":"There is no 'paid' property"}
```

401 Unauthorized

I do not know you, tell me who you are and I will check your permissions.

```
GET /bookings/42
→ 401 Unauthorized
→ {"error":"no_credentials", "error_description":"You must be authenticated"}
```

403 Forbidden

Your rights are not sufficient to access this resource.

```
GET /bookings/42
→ 403 Forbidden
→ {"error":"protected_resource", "error_description":"You need sufficient rights"}
```

404 Not Found

The resource you are requesting does not exist.

```
GET /hotels/999999
→ 404 Not Found
→ {"error":"not_found", "error_description": "The hotel '999999' does not exist"}
```

405 Method Not Allowed

Either the method is not supported or relevant on this resource or the user does not have the permission.

```
PUT /hotels/999999
→ 405 Method Not Allowed
→ {"error":"not_implemented", "error_description":"Hotel creation not implemented"}
```

406 Not Acceptable

There is nothing to send that matches the Accept-* headers. For example, you requested a resource in XML but it is only available in JSON.

```
GET /hotels
Accept-Language: cn
→ 406 Not Acceptable
→ {"error": "not_acceptable", "error_description":"Available languages: en, fr"}
```

SERVER ERROR

500 Internal Server Error

The request seems right, but a problem occurred on the server. The client cannot do anything about that.

```
GET /users
→ 500 Internal server error
→ {"error":"server_error", "error_description":"Oops! Something went wrong…"}
```

RESTful API Design.

As soon as we start working on an API, design issues arise. A robust and strong design is a key factor for API success. A poorly designed API will indeed lead to misuse or – even worse – no use at all by its intended clients: application developers.

Creating and providing a state of the art API requires taking into account:
- RESTful API principles as described in the literature (Roy Fielding, Leonard Richardson, Martin Fowler, HTTP specification…)
- The API practices of the Web Giants

Nowadays, two opposing approaches are seen.

"Purists" insist upon following REST principles without compromise. "Pragmatics" prefer a more practical approach, to provide their clients with a more usable API. The proper solution often lies in between.

Designing a REST API raises questions and issues for which there is no universal answer. REST best practices are still being debated and consolidated, which is what makes this job fascinating.

To facilitate and accelerate the design and development of your APIs, we share our vision and beliefs with you in this Reference Card. They come from our direct experience on API projects.

Why an API strategy ?

"Anytime, Anywhere, Any device" are the key problems of digitalisation. API is the answer to "Business Agility" as it allows to build rapidly new GUI for upcoming devices.

An API layer enables
- Cross device
- Cross channel
- 360° customer view

Open API allows
- To outsource innovation
- To create new business models

Embrace WOA
"Web Oriented Architecture"
- Build a fast, scalable & secured REST API
- Based on: REST, HATEOAS, Stateless decoupled µ-services, Asynchronous patterns, OAuth2 and OpenID Connect protocols
- Leverage the power of your existing web infrastructure

DISCLAMER

This Reference Card doesn't claim to be absolutely accurate. The design concepts exposed result from our previous work in the REST area. Please check out our blog http://blog.octo.com, and feel free to comment or challenge this API cookbook. We are really looking forward to sharing with you.

General concepts.

KISS

Anyone should be able to use your API without having to refer to the documentation.

- Use standard, concrete and shared terms, not your specific business terms or acronyms.
- Never allow application developers to do things more than one way.
- Design your API for your clients (Application developers), not for your data.
- Target main use cases first, deal with exceptions later.

`GET /orders, GET /users, GET /products, ...`

CURL

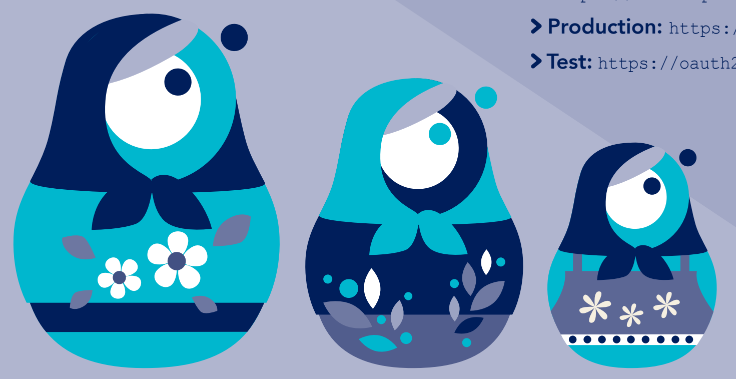
You should use CURL to share examples, which you can copy/paste easily.

```
CURL -X POST \
-H "Accept: application/json" \
-H "Authorization: Bearer at-80003004-19a8-46a2-908e-33d4057128e7" \
-d '{"state":"running"}' \
https://api.fakecompany.com/v1/users/007/orders?client_id=API_KEY_003
```

GRANULARITY

Medium grained resources
You should use medium grained, not fine nor coarse. Resources shouldn't be nested more than two levels deep:

```
GET /users/007
{ "id":"007",
  "first_name":"James",
  "name":"Bond",
  "address":{
     "street":"Horsen Ferry Road",
     "city":{"name":"London"}
  }
}
```



SECURITY

OAuth2/OIDC & HTTPS

You should use OAuth2 to manage Authorization. OAuth2 matches 99% of requirements and client typologies, don't reinvent the wheel, you'll fail. You should use HTTPS for every API/OAuth2 request. You may use OpenID Connect to handle Authentication.

API DOMAIN NAMES

You may consider the following five subdomains:

- **Production:** `https://api.fakecompany.com`
- **Test:** `https://api.sandbox.fakecompany.com`
- **Developer portal:** `https://developers.fakecompany.com`
- **Production:** `https://oauth2.fakecompany.com`
- **Test:** `https://oauth2.sandbox.fakecompany.com`

URLs.

NOUNS

You should use nouns, not verbs (vs SOAP-RPC).
`GET /orders not /getAllOrders`

HIERARCHICAL STRUCTURE

You should leverage the hierarchical nature of the URL to imply structure (aggregation or composition). Ex: an order contains products.
`GET /orders/1234/products/1`

CONSISTENT CASE

You may choose between snake_case or camelCase for attributes and parameters, but you should remain consistent.

```
GET /orders?id_user=007
or GET /orders?idUser=007
POST/orders {"id_user":"007"}
or POST/orders {"idUser":"007"}
```

If you have to use more than one word in URL, you should use spinal-case (some servers ignore case).
`POST /specific-orders`

PLURALS

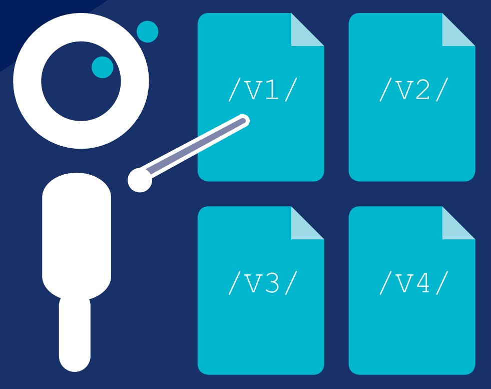
You should use plural nouns, not singular nouns, to manage two different types of resources:

- Collection resource: `/users`
- Instance resource: `/users/007`

You should remain consistent.
`GET /users/007 not GET /user/007`

VERSIONING

You should make versioning mandatory in the URL at the highest scope (major versions).
You may support at most two versions at the same time (Native apps need a longer cycle).
`GET /v1/orders`



CRUD-LIKE OPERATIONS

Use HTTP verbs for CRUD operations (Create/Read/Update/Delete).

HTTP VERB	COLLECTION: /ORDERS	INSTANCE : /ORDER/{ID}
GET	Read a list of orders. 200 OK.	Read the details of a single order. 200 OK.
POST	Create a new order. 201 Created.	-
PUT	-	Full Update: 200 OK./ Create a specific order: 201 Created.
PATCH	-	Partial Update. 200 OK.
DELETE	-	Delete order. 204 OK.

POST is used to Create an instance of a collection. The ID isn't provided, and the new resource location is returned in the "Location" Header.

```
POST /orders {"state":"running", «id_user":"007"}
201 Created
Location: https://api.fakecompany.com/orders/1234
```

But remember that, if the ID is specified by the client, PUT is used to Create the resource.

```
PUT /orders/1234
201 Created
```

PATCH is commonly used for partial Update.

```
PATCH /orders/1234 {"state":"paid"}
200 Ok
```

PUT is used for Updates to perform a full replacement.

```
PUT /orders/1234 {"state":"paid", "id_user":"007"}
200 Ok
```

GET is used to Read a collection.

```
GET /orders
200 Ok
[{"id":"1234", "state":"paid"}
{"id":"5678", "state":"running"}]
```

GET is used to Read an instance.

```
GET /orders/1234
200 Ok
{"id":"1234", "state":"paid"}
```

Query strings.

SEARCH

You should use /search keyword to perform a search on a specific resource.
`GET /restaurants/search?type=thai`
You may use the "Google way" to perform a global search on multiple resources.
`GET /search?q=running+paid`

FILTERS

You ought to use '?' to filter resources
`GET /orders?state=payed&id_user=007`
or (multiple URIs may refer to the same resource)
`GET /users/007/orders?state=paied`

PAGINATION

You may use a range query parameter. Pagination is mandatory: a default pagination has to be defined, for example: range=0-25.
The response should contain the following headers: Link, Content-Range, Accept-Range.
Note that pagination may cause some unexpected behavior if many resources are added.

```
/orders?range=48-55
206 Partial Content
Content-Range: 48-55/971
Accept-Range: order 10
Link : <https://api.fakecompany.com/v1/orders?range=0-7>; rel="first",
<https://api.fakecompany.com/v1/orders?range=40-47>; rel="prev",
<https://api.fakecompany.com/v1/orders?range=56-64>; rel="next",
<https://api.fakecompany.com/v1/orders?range=968-975>; rel="last"
```

PARTIAL RESPONSES

You should use partial responses so developers can select which information they need, to optimize bandwidth (crucial for mobile development).

```
GET /users/007?fields=firstname,name,address(street)
200 OK
{ "id":"007",
  "firstname":"James",
  "name":"Bond",
   address:{"street":"Horsen Ferry Road"}
}
```

SORT

Use ?sort =atribute1,atributeN to sort resources.
By default resources are sorted in ascending order.
Use ?desc=atribute1,atributeN to sort resources in descending order
`GET /restaurants?sort=rating,reviews,name;desc=rate,reviews`

URL RESERVED WORDS : FIRST, LAST, COUNT

Use /first to get the 1st element

```
GET /orders/first
200 OK
{"id":"1234", "state":"paid"}
```

Use /last to retrieve the latest resource of a collection

```
GET /orders/last
200 OK
{"id":"5678", "state":"running"}
```

Use /count to get the current size of a collection

```
GET /orders/count
200 OK
{"2"}
```

Other key concepts.

I18N

Use ISO 8601 standard for Date/Time/Timestamp:
`1978-05-10T06:06:06+00:00` or `1978-05-10`
Add support for different Languages.
`Accept-Language: fr-CA, fr-FR` not `?language=fr`

CONTENT NEGOTIATION

Content negotiation is managed only in a pure RESTful way. The client asks for the required content, in the Accept header, in order of preference. Default format is JSON.
`Accept: application/json, text/plain not /orders.json`

CROSS-ORIGIN REQUESTS

Use CORS standard to support REST API requests from browsers (js SPA…).

But if you plan to support Internet Explorer 7/8 or 9, you shall consider specifics endpoints to add JSONP support.

- All requests will be sent with a GET method!
- Content negotiation cannot be handled with Accept header in JSONP.
- Payload cannot be used to send data.

```
POST /orders       and  /orders.jsonp?method=POST&callback=foo
GET /orders        and  /orders.jsonp?callback=foo
GET /orders/1234   and  /orders/1234.jsonp?callback=foo
PUT /orders/1234   and  /orders/1234.jsonp?method=PUT&callback=foo
```

Warning: a web crawler could easily damage your application with a method parameter. Make sure that an OAuth2 access_token is required, and an OAuth2 client_id as well.

"NON RESOURCE" SCENARIOS

In a few use cases we have to consider operations or services rather than resources.

You may use a POST request with a verb at the end of the URI.

```
POST /emails/42/send
POST /calculator/sum  [1,2,3,5,8,13,21]
POST /convert?from=EUR&to=USD&amount=42
```

However, you should consider using RESTful resources first before going this way.

HATEOAS

Your API should provide Hypermedia links in order to be completely discoverable. But keep in mind that a majority of users wont probably use those hyperlinks (for now), and will read the API documentation and copy/paste call examples.

So, each call to the API should return in the Link header every possible state of the application from the current state, plus self.

You may use RFC5988 Link notation to implement HATEOAS :

```
GET /users/007
< 200 Ok
< { "id":"007", "firstname":"Mario",...}
< Link : <https://api.fakecompany.com/v1/users>; rel="self"; method:"GET",
<https://api.fakecompany.com/v1/addresses/42>; rel="addresses"; method:"GET",
<https://api.fakecompany.com/v1/orders/1234>; rel="orders"; method:"GET"
```