RESTFul API - Good practices

# RESTful services

## Definition

**REST** stands for **Re**presentational **S**tate **T**ransfer. It is an architectural style for network applications.

Its main characteristics are :

* Simplicity
* Stateless : the server does not persist any client related state -> survive server restarts -> no client to server binding, the request can be handled by a cluster of servers -> the server scales easier, by not dealing with multiple session memory management
* Cacheability

Most of the time it relies on the HTTP protocol, using its verbs : GET, POST, PUT, DELETE, PATCH. Although it does not offer built-in security features, encryption, session managemeent, QoS guarantee, those services can be added on top of HTTP.

Notes about the document :

* JSON : requests content are written using JSON. It is the go-to description langage nowadays but XML could also be used.
* "Should" : I also often use the word should because in the REST approach, even if we can pretty much do whatever we want, there seems to be an empirical better way, provided mostly by conventions.

## Typical usage

REST is resource-centric as opposed to SOAP which is action-centric. Therefore a Restful API typically exposes its functionality via **ressource identifiers** like the following :

<http://localhost/r3/transparency/api/products>

Besides the fact that this endpoint is on localhost, we can see that it can be used to execute actions on the *products* resources.

Pro-tip : The **plural form** is typically prefered to fulfil the need to request a list of products which would not be semantically correct via a singular formed endpoint.

This resource can be used in conjunction with HTTP verbs...

* To return a list of products :

GET [host]/api/products

* To return a specific product :

GET [host]/api/products/ 25

* To create a new product :

POST [host]/api/products {

"product\_type" : "Vegetable",  
 "product\_name" : "Tometo"

}

* To replace a product :

PUT [host]/api/products/25 {

"product\_type" : "Vegetable",  
 "product\_name" : "Tomato"

}

* To modify a product :

PATCH [host]/api/products/25 [

{   
 "action" : "replace",  
 "field" : "product\_type",  
 "value" : "Fruit"   
 }

]

* To delete a product :

DELETE [host]/api/products/324

Note : anybody can decide to implement such functionnality by using only GET and POST verbs for the sake of simplicity, this would not be standard REST though.

**HTTP verbs :**

* GET : reading, this should not modify the data.
* PUT : replace a data
* PATCH : update some part of the data
* DELETE : delete the data
* POST : everything else

**Consistency**

Usage of standards format like the one detailed above makes the Restful API consistent and also self-documentary for a big part. Let's imagine that a developper enters the project to implement the **Contacts** functionnalities, a typical CRUD. And it goes like this :

* To get a list of contacts POST [host]/api/getcontactlist {

"page" : 4,  
 "page\_size": 20

}

* To get a specific contact GET [host]/api/getcontact/23
* To delete a contact POST [host]/api/contacts/delete/23

A second developper seeing the products implementation could not deduce any of the contacts usage because of the lack of consistency. Following the REST standards would have make the job easier.

**HTTP status**

HTTP protocol has plenty of return codes that can be used to give a client maximal data in a minimal number of characters. 200, 500 and 404 are not the only possible status, a complete list of those can be found at<https://en.wikipedia.org/wiki/List_of_HTTP_status_codes> with their corresponding meaning, which are used as the enumeration values in System.Net.HttpStatusCode

The status are organized by hundreds

* 1\*\* : informational
* 2\*\* : success
* 3\*\* : redirection
* 4\*\* : client error
* 5\*\* : server error

Most of the standard status are quite specific and can be used to describe typical scenario.

The usage of specific values inside those categories seems not to be mandatory, so in doubt 200, 300, 400 and 500 are meaningful enough. Typically though :

* Creations completed should return *201 - Created*,
* Searches yielding no results should return *204 - No Content*
* Creations with duplicates detected should return *409 - Conflict*
* Teapot servers should always return *418 - I'm a teapot*

And so on.

**Exception handling**

Whenever an error occurs, there is to decide what data are returned and how.

**400 or 500 ?**

When confronted to an error, the server has first to answer one basic question : is the client responsible (400) or not (500) ? As 500 errors can be yielded by different application layers, it can be tricky to have a consistent way of dealing with what is returned. For security reasons, a server should not return too much information about how it failed. In the case of 400 errors, the educated client knows that the API has handled the exception because the mistake was expected in some way. In other words, unhandled exceptions coming from the client should return a 500 error as long as the implementation of the API ignores the possible client-made mistake.

**Examples**

Let's say I forgot to validate the input data of the product, but the business layer needs the product\_name not to be null or empty. By default, the unhandled exception will return a 500, and something like "product\_name" cannot be empty with the full stack trace. This is correct in a way, because it means that the server (aka the developper) has made the error of not implementing the validation so it returns a 500. So testers will create a story very fast about it. Then the validation will be made, the exception handled and a proper 400 error returned.

In another case, the developper can expose an API with "Not yet implemented" content (because he had no time, because of specflows). Therefore, it should return a 500 status but this time with the full knowledge of the facts.

**Error format**

When the error is handled at the API's application level, it is good to create a generic exception format, giving the client specific data about the mistake(s) he made. We can even define domain-specific codes :

{  
 "ErrorCode" : "NEWBIE\_MISTAKE"   
 "Details" : "Do you really think tomato is a vegetable ? I mean... really ?"  
}

**Searches and lists**

Returning lists should be done with performance in mind. There should be a default and a maximum pagination in order to avoid returning too many data, even if the client asks for it. Especially in the cases where returned data contain embedded subtypes.

**Documentation :**

Self documentation and usage of HATEOAS or HAL depends on the specificity of the API and on its clients.

**the N+1 selects issue**

<http://stackoverflow.com/questions/97197/what-is-the-n1-selects-issue>

**Security of REST**

<http://stackoverflow.com/questions/454355/security-of-rest-authentication-schemes>

**Server response**

REST is not bound to any description langage. A RESTful service can return XML, Json, CSV...

**Guidelines**

* URLS should be logical, not physical
* lists should always be paginated
* the result of the API must be well documented and stick to the documentation. Changing the API should be made in coordination with the clients.
* good design is supposed to contain hypermedia links to related resources. For example in a list of products, each product should have a link pointing to its detail URL.
* GET should never cause a state change

**ASP.NET implementation**

**Exceptions handling**

**Routing**

**Configurations**

**Documentation (swagger)**

**Validations of forms**

**Documenting the API :**

Swagger

**Bibliography**

* Learn REST : A Tutorial (a tad old)

-> <http://rest.elkstein.org/2008/02/what-is-rest.html>

* Best Practices for Designing a Pragmatic RESTful API

-> <http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api>

* HAL hypermedia Type and other stuff

-> <http://sookocheff.com/post/api/on-choosing-a-hypermedia-format/>

* understanding HATEOAS

-> <https://spring.io/understanding/HATEOAS>

* JSon schema

-> <https://brandur.org/elegant-apis>