Introduction to the REST API – Part II

This is part II of the introduction to the REST API blog post. In part one we covered the basic aspects of what REST is really all about, and introduced some of the key concepts such as the REST constraints and REST elements which are comprised of resources, and which make up the building blocks of the web etc…

In part II we are going to look at the **REST Request Methods and Responses.**

To start with when we refer to **Request Methods,** the Clients specify the desired interaction **method** in the Request-Line part of an HTTP request message. RFC 2616 defines the Request-Line syntax as shown below:

Request-Line = **Method** SP Request-URI SP HTTP-Version CRLF

Each HTTP **method** has specific, well-defined semantics within the context of a **REST API's** resource model. The purpose of GET is to retrieve a representation of a resource's state. HEAD is used to retrieve the metadata associated with the resource's state. PUT should be used to add a new resource to a store or update a resource. DELETE removes a resource from its parent. POST should be used to create a new resource within a collection and execute controllers.

In summary, and when someone asks you about the REST Request Methods you can think of it in these terms :

- Clients specify the desired interaction (HTTP request message as defined by RFC 2616)
- Each HTTP method has specific, well-defined semantics within the context of a REST API's resource model.
- GET – (The purpose of GET is to retrieve a representation of a resource's state).
- HEAD is used to retrieve the metadata associated with the resource's state.
- PUT should be used to add a new resource to a store or update a resource.
- DELETE removes a resource from its parent.
- POST should be used to create a new resource.

So logic dictates if there is a *Request* then there needs to be a *Response.* Right? Indeed, that is the case with the REST API and is handled through **REST Response Status Codes.**

HTTP defines forty standard status codes that can be used to convey the results of a client's request.

- The REST APIs use a HTTP response message to inform clients of their request's result (as defined in RFC 2616).
- 5 Categories are Defined
    - 1xx: Informational Communicates transfer protocol-level information.
    - 2xx: Success Indicates that the client's request was accepted successful.
    - 3xx: Redirection Indicates that the client must take some additional action in order to complete their request.
    - 4xx: Client Error This category of error status codes points the finger at clients.
    - 5xx: Server Error The server takes responsibility for these error status codes.

This following section concisely describes how and when to use the subset of codes that apply to the design of a **REST API**. Here is a simple list of the **REST Response Status Codes** as a starting point for you to look at. These are the HTTP response message(s) to inform clients of their request's result (As Defined in RFC 2616).

- 200 OK
- 201 Created
- 202 Accepted
- 204 No content
- 400 Bad request
- 401 Unauthorized
- 403 Forbidden
- 404 Not found
- 405 Method Not Allowed
- 409 Conflict
- 500 Internal Server Error

- 501 Not implemented
- 503 Service unavailable

Let us now look at these **REST Response Status Codes** in more detail beginning with 200.

Rule: 200 ("OK") should be used to indicate nonspecific success

In most cases, 200 is the code the client hopes to see. It indicates that the **REST API** successfully carried out whatever action the client requested, and that no more specific code in the 2xx series is appropriate. Unlike the 204 status code, a 200 response should include a response body.

Rule: 200 ("OK") must not be used to communicate errors in the response body

Always make proper use of the HTTP response status codes as specified by the rules in this section. In particular, a **REST API** must not be compromised in an effort to accommodate less sophisticated HTTP clients.

Rule: 201 ("Created") must be used to indicate successful resource creation

A **REST API** responds with the 201 status code whenever a collection creates, or a store adds, a new resource at the client's request. There may also be times when a new resource is created as a result of some controller action, in which case 201 would also be an appropriate response.

Rule: 202 ("Accepted") must be used to indicate successful start of an asynchronous action

A 202 response indicates that the client's request will be handled asynchronously. This response status code tells the client that the request appears valid, but it still may have problems once it's finally processed. A 202 response is typically used for actions that take a long while to process.

Controller resources may send 202 responses, but other resource types should not.

Rule: 204 ("No Content") should be used when the response body is intentionally empty

The 204 status code is usually sent out in response to a PUT, POST, or DELETE request, when the **REST API** declines to send back any status message or representation in the response message's body. An **API** may also send 204 in conjunction with a GET request to indicate that the requested resource exists, but has no state representation to include in the body.

Rule: 301 ("Moved Permanently") should be used to relocate resources

The 301 status code indicates that the **REST API's** resource model has been significantly redesigned and a new *permanent* URI has been assigned to the client's requested resource. The **REST API** should specify the new URI in the response's Location header.

Rule: 302 ("Found") should not be used

The intended semantics of the 302 response code have been misunderstood by programmers and incorrectly implemented in programs since version 1.0 of the HTTP protocol.[24] The confusion centers on whether it is appropriate for a client to always automatically issue a follow-up GET request to the URI in response's Location header, regardless of the original request's **method**. For the record, the intent of 302 is that this automatic redirect behavior only applies if the client's original request used either the GET or HEAD **method**.

To clear things up, HTTP 1.1 introduced status codes 303 ("See Other") and 307 ("Temporary Redirect"), either of which should be used instead of 302.

Rule: 303 ("See Other") should be used to refer the client to a different URI

A 303 response indicates that a controller resource has finished its work, but instead of sending a potentially unwanted response body, it sends the client the URI of a response resource. This can be the URI of a temporary status message, or the URI to some already existing, more permanent, resource.

Generally speaking, the 303 status code allows a **REST API** to send a reference to a resource without forcing the client to download its state. Instead, the client may send a GET request to the value of the Location header.

Rule: 304 ("Not Modified") should be used to preserve bandwidth

This status code is similar to 204 ("No Content") in that the response body must be empty. The key distinction is that 204 is used when there is nothing to send in the body, whereas 304 is used when there *is* state information associated with a resource but the client already has the most recent version of the representation.

This status code is used in conjunction with conditional HTTP requests, discussed in Chapter 4.

Rule: 307 (“Temporary Redirect”) should be used to tell clients to resubmit the request to another URI

HTTP/1.1 introduced the 307 status code to reiterate the originally intended semantics of the 302 (“Found”) status code. A 307 response indicates that the **REST API** is not going to process the client’s request. Instead, the client should resubmit the request to the URI specified by the response message’s Location header.

A **REST API** can use this status code to assign a *temporary* URI to the client’s requested resource. For example, a 307 response can be used to shift a client request over to another host.

Rule: 400 (“Bad Request”) may be used to indicate nonspecific failure

400 is the generic client-side error status, used when no other 4xx error code is appropriate.

NOTE

For errors in the 4xx category, the response body may contain a document describing the client’s error (unless the request **method** was HEAD). See Error Representation for error response body design.

Rule: 401 (“Unauthorized”) must be used when there is a problem with the client’s credentials

A 401 error response indicates that the client tried to operate on a protected resource without providing the proper authorization. It may have provided the wrong credentials or none at all.

Rule: 403 (“Forbidden”) should be used to forbid access regardless of authorization state

A 403 error response indicates that the client’s request is formed correctly, but the **REST API** refuses to honor it. A 403 response is *not* a case of insufficient client credentials; that would be 401 (“Unauthorized”).

REST APIs use 403 to enforce application-level permissions. For example, a client may be authorized to interact with some, but not all of a **REST API’s**resources. If the client attempts a resource interaction that is outside of its permitted scope, the **REST API** should respond with 403.

Rule: 404 (“Not Found”) must be used when a client’s URI cannot be mapped to a resource

The 404 error status code indicates that the **REST API** can’t map the client’s URI to a resource.

Rule: 405 (“Method Not Allowed”) must be used when the HTTP method is not supported

The **API** responds with a 405 error to indicate that the client tried to use an HTTP **method** that the resource does not allow. For instance, a read-only resource could support only GET and HEAD, while a controller resource might allow GET and POST, but not PUT or DELETE.

A 405 response must include the Allow header, which lists the HTTP **methods**that the resource supports. For example:

Allow: GET, POST

Rule: 406 (“Not Acceptable”) must be used when the requested media type cannot be served

The 406 error response indicates that the **API** is not able to generate any of the client’s preferred media types, as indicated by the Accept request header. For example, a client request for data formatted as *application/xml* will receive a 406 response if the **API** is only willing to format data as *application/json*.

Rule: 409 (“Conflict”) should be used to indicate a violation of resource state

The 409 error response tells the client that they tried to put the **REST API’s**resources into an impossible or inconsistent state. For example, a **REST API**may return this response code when a client tries to delete a non-empty store resource.

Rule: 412 (“Precondition Failed”) should be used to support conditional operations

The 412 error response indicates that the client specified one or more preconditions in its request headers, effectively telling the **REST API** to carry out its request only if certain conditions were met. A 412 response indicates that those conditions were not met, so instead of carrying out the request, the **API** sends this status code.

See Rule: Stores must support conditional PUT requests for an example use of the 412 status code.

Rule: 415 (“Unsupported Media Type”) must be used when the media type of a request’s payload cannot be processed

The 415 error response indicates that the **API** is not able to process the client’s supplied media type, as indicated by the Content-Type request header. For example, a client request including data formatted as *application/xml* will receive a 415 response if the **API** is only willing to process data formatted as *application/json*.

Rule: 500 (“Internal Server Error”) should be used to indicate API malfunction

500 is the generic **REST API** error response. Most web frameworks automatically respond with this response status code whenever they execute some request handler code that raises an exception.

A 500 error is never the client’s fault and therefore it is reasonable for the client to retry the exact same request that triggered this response, and hope to get a different response.

Finally, let us turn our attention to the important **HTTP Request Headers** that you should be aware of :

1. **Authorization :** The user credential in Base64 encoded formats
2. **Content-Type**: “application/xml” or “application/json”. Tells us that the request body/payload from user is in xml or json format.
3. **Accept :** “application/xml” or “application/json”. Tells us that the response body/payload from server is in xml or json format.
4. **Location :** It gives the URL/resource to check for the status of the current call on success.
5. **HTTP Request Headers** : For the details on these HTTP request headers check the HTTP RFC 2616: http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html which is part of the Hypertext Transfer Protocol — HTTP/1.1and which can be found here: http://www.w3.org/Protocols/rfc2616/rfc2616.html

As a reference more information can be found in the Fielding, Roy T., Tim Berners-Lee, et al. HTTP/1.1, RFC 2616, RFC Editor, 1999 (http://www.rfc-editor.org/rfc/rfc2616.txt).