**Error Handling in RESTful API**

RESTful API

RESTful API is a common tool of building web services, especially in front and back-end separated application. It is based on HTTP protocol, which is simple, text-oriented, and well supported by various languages, browsers or clients. However, REST is not yet standardized, so that the developers need to decide how to design their APIs. One of the decisions is error handling. Should I use HTTP status code? How to handle form validation errors, etc. This article will propose an error handling mechanism for RESTful API, based on my daily work and understanding of this technique.

**Types of Errors**

I tend to categorize errors into two types, global and local. Global errors include requesting an unknown API url, not being authorized to access this API, or there’s something wrong with the server code, unexpected and fatal. These errors should be caught by the web framework, no customized handling in individual API function.

Local errors, on the other hand, are closely related to the current API. Examples are form validation, violation of unique constraint, or other expected errors. We need to write specific codes to catch these errors, and raise a global error with message and payload for framework to catch and respond with.

Flask, for instance, provides a mechanism to catch exceptions globally:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | class BadRequest(Exception):  """Custom exception class to be thrown when local error occurs."""  def \_\_init\_\_(self, message, status=400, payload=None):  self.message = message  self.status = status  self.payload = payload   @app.errorhandler(BadRequest) def handle\_bad\_request(error):  """Catch BadRequest exception globally, serialize into JSON, and respond with 400."""  payload = dict(error.payload or ())  payload['status'] = error.status  payload['message'] = error.message  return jsonify(payload), 400   @app.route('/person', methods=['POST']) def person\_post():  """Create a new person object and return its ID"""  if not request.form.get('username'):  raise BadRequest('username cannot be empty', 40001, { 'ext': 1 })  return jsonify(last\_insert\_id=1) |

**Error Response Payload**

When you post to /person with an empty username, it’ll return the following error response:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | HTTP/1.1 400 Bad Request Content-Type: application/json  {  "status": 40001,  "message": "username cannot be empty",  "ext": 1 } |

There’re several parts in this response: HTTP status code, a custom status code, error message, and some extra information.

**Use HTTP Status Code**

HTTP status code itself provides rich semantics for errors. Generally 4xx for client-side error and 5xx server-side. Here’s a brief list of commonly used codes:

* 200 Response is OK.
* 400 Bad request, e.g. user posts some in valid data.
* 401 Unauthorized. With Flask-Login, you can decorate a route with @login\_required, and if the user hasn’t logged in, 401 will be returned, and client-side can redirect to login page.
* 403 Access is forbidden.
* 404 Resource not found.
* 500 Internal server error. Usually for unexpected and irrecoverable exceptions on the server-side.

**Custom Error Code**

When client receives an error, we can either open a global modal dialog to show the message, or handle the errors locally, such as displaying error messages below each form control. For this to work, we need to give these local errors a special coding convention, say 400 for global error, while 40001 and 40002 will trigger different error handlers.

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | fetch().then(response => {  if (response.status == 400) { // http status code  response.json().then(responseJson => {  if (responseJson.status == 400) { // custom error code  // global error handler  } else if (responseJson.status == 40001) { // custom error code  // custom error handler  }  })  } }) |

**More Error Information**

Sometimes it is ideal to return all validation errors in one response, and we can use payload to achieve that.

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | {  "status": 40001,  "message": "form validation failed"  "errors": [  { "name": "username", "error": "username cannot be empty" },  { "name": "password", "error": "password minimum length is 6" }  ] } |

**Fetch API**

For AJAX request, [Fetch API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) becomes the standard library. We can wrap it into a function that does proper error handling. Full code can be found in GitHub ([link](https://github.com/jizhang/rest-error/blob/master/src/request.js)).

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 | function request(url, args, form) {  return fetch(url, config)  .then(response => {  if (response.ok) {  return response.json()  }   if (response.status === 400) {  return response.json()  .then(responseJson => {  if (responseJson.status === 400) {  alert(responseJson.message) // global error handler  }  // let subsequent "catch()" in the Promise chain handle the error  throw responseJson  }, error => {  throw new RequestError(400)  })  }   // handle predefined HTTP status code respectively  switch (response.status) {  case 401:  break // redirect to login page  default:  alert('HTTP Status Code ' + response.status)  }   throw new RequestError(response.status)  }, error => {  alert(error.message)  throw new RequestError(0, error.message)  }) } |

This method will reject the promise whenever an error happens. Invokers can catch the error and check its status. Here’s an example of combining this approach with MobX and ReactJS:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | // MobX Store loginUser = flow(function\* loginUser(form) {  this.loading = true  try {  // yield may throw an error, i.e. reject this Promise  this.userId = yield request('/login', null, form)  } finally {  this.loading = false  } })  // React Component login = () => {  userStore.loginUser(this.state.form)  .catch(error => {  if (error.status === 40001) {  // custom error handler  }  }) } |

中文

OR

REST APIs use the Status-Line part of an HTTP response message to inform clients of their request’s overarching result.   
[RFC 2616](https://www.ietf.org/rfc/rfc2616.txt) defines the Status-Line syntax as shown below:

Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF

A great amount of applications are using Restful APIs that are based on the HTTP protocol for connecting their clients. In all the calls, the server and the endpoint at the client both return a call status to the client which can be in the form of:

* The success of API call.
* Failure of API call.

In both the cases, it is necessary to let the client know so that they can proceed to the next step. In the case of a successful API call they can proceed to the next call or whatever their intent was in the first place but in the case of latter they will be forced to modify their call so that the failed call can be recovered.

Top of Form



Bottom of Form

To enable the best user experience for your customer, it is necessary on the part of the developers to make excellent error messages that can help their client to know what they want to do with the information they get. An excellent error message is precise and lets the user know about the nature of the error so that they can figure their way out of it.

**A good error message also allows the developers to get their way out of the failed call.**

Next step is to know what error messages to integrate into your framework so that the clients on the end point and the developers at the server are constantly made aware of the situation which they are in. in order to do so, the rule of thumb is to keep the error messages to a minimum and only incorporate those error messages which are helpful.

HTTP defines over 40 standard status codes that can be used to convey the results of a client’s request. The status codes are divided into the five categories presented here:

* **1xx: Informational** - Communicates transfer protocol-level information
* **2xx: Success** -Indicates that the client’s request was accepted successfully.
* **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.

If you would ask me 5 years ago about HTTP Status codes I would guess that the talk is about web sites, status 404 meaning that some page was not found and etc. But today when someone asks me about HTTP Status codes, it is 99.9% refers to [REST API web services development](http://www.restcase.com/). I have lots of experience in both areas (Website development, REST API web services development) and it is sometimes hard to come to a conclusion about what and how use the errors in REST APIs.

There are some cases where this status code is always returned, even if there was an error that occurred. Some believe that returning status codes other than 200 is not good as the client did reach your REST API and got response.

Proper use of the status codes will help with your [REST API management](http://www.restcase.com/) and [REST API workflow management](http://www.restcase.com/).

If for example the user asked for “account” and that account was not found there are 2 options to use for returning an error to the user:

* Return **200 OK** Status and in the body return a json containing explanation that the account was not found.
* Return **404 not found** status. The first solution opens up a question whether the user should work a bit harder to parse the json received and to see whether that json contains error or not.
* There is also a third solution: Return **400 Error - Client Error**. I will explain a bit later why this is my favorite solution.

It is understandable that for the user it is easier to check the status code of 404 without any parsing work to do.

**I my opinion this solution is actually miss-use of the HTTP protocol**

We did reach the REST API, we did got response from the REST API, what happens if the users misspells the URL of the REST API – he will get the 404 status but that is returned not by the REST API itself.

I think that these solutions should be interesting to explore and to see the benefits of one versus the other.

There is also one more solution that is basically my favorite – this one is a combination of the first two solutions, he is also gives better [Restful API services automatic testing](http://www.restcase.com/) support because only several status codes are returned, I will try to explain about it.

##### Error handling Overview

Error responses should include a common HTTP status code, message for the developer, message for the end-user (when appropriate), internal error code (corresponding to some specific internally determined ID), links where developers can find more info. For example:

'{ "status" : 400,   
"developerMessage" : "Verbose, plain language description of the problem. Provide developers suggestions about how to solve their problems here", "userMessage" : "This is a message that can be passed along to end-users, if needed.", "errorCode" : "444444", "moreInfo" : "<http://www.example.gov/developer/path/to/help/for/444444>,<http://tests.org/node/444444>", }'

How to think about errors in a pragmatic way with REST?   
Apigee's [blog post](http://apigee.com/about/blog/technology/restful-api-design-what-about-errors) that talks about this issue compares 3 top API providers.

###### Facebook

No matter what happens on a Facebook request, you get back the 200 status code - everything is OK. Many error messages also push down into the HTTP response. Here they also throw an #803 error but with no information about what #803 is or how to react to it.

###### Twilio

Twilio does a great job aligning errors with HTTP status codes. Like Facebook, they provide a more granular error message but with a link that takes you to the documentation. Community commenting and discussion on the documentation helps to build a body of information and adds context for developers experiencing these errors.

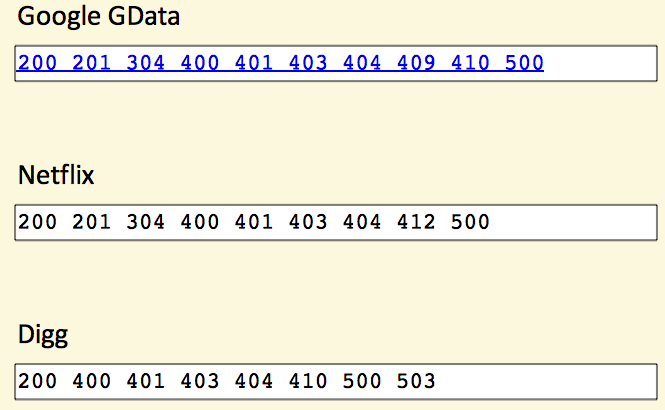
###### SimpleGeo

Provides error codes but with no additional value in the payload.

##### Error Handling - Best Practises

First of all: **Use HTTP status codes!** but don't overuse them.   
Use HTTP status codes and try to map them cleanly to relevant standard-based codes.   
There are over 70 HTTP status codes. However, most developers don't have all 70 memorized. So if you choose status codes that are not very common you will force application developers away from building their apps and over to wikipedia to figure out what you're trying to tell them.

Therefore, **most API providers use a small subset**.   
For example, the Google GData API uses only 10 status codes, Netflix uses 9, and Digg, only 8.



###### How many status codes should you use for your API?

When you boil it down, there are really only 3 outcomes in the interaction between an app and an API:

* Everything worked
* The application did something wrong
* The API did something wrong

Start by using the following 3 codes. If you need more, add them. But you shouldn't go beyond 8.

* 200 - OK
* 400 - Bad Request
* 500 - Internal Server Error

Please keep in mind the following rules when using these status codes:

###### 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.

###### 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. For errors in the 4xx category, the response body may contain a document describing the client’s error (unless the request method was HEAD).

###### 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.

If you're not comfortable reducing all your error conditions to these 3, try adding some more but do not go beyond 8:

* 401 - Unauthorized
* 403 - Forbidden
* 404 - Not Found

Please keep in mind the following rules when using these status codes:

###### 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.

###### 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.

###### 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.





Bottom of Form

Top of Form

##### Conclusion

I believe that the best solution to handle errors in a REST API web services is the third option, in short:   
**Use three simple, common response codes indicating (1) success, (2) failure due to client-side problem, (3) failure due to server-side problem:**

* 200 - OK
* 400 - Bad Request (Client Error) - A json with error \ more details should return to the client.
* 401 - Unauthorized
* 500 - Internal Server Error - A json with an error should return to the client only when there is no security risk by doing that.

I think that this solution can also ease the client to handle only these 4 status codes and when getting either 400 or 500 code he should take the response message and parse it in order to see what is the problem exactly and on the other hand the REST API service is simple enough.

The decision of choosing which error messages to incorporate and which to leave is based on sheer insight and intuition. For example: if an app and API only has three outcomes which are; everything worked, the application did not work properly and API did not respond properly then you are only concerned with three error codes. By putting in unnecessary codes, you will only distract the users and force them to consult Google, Wikipedia and other websites.

Most important thing in the case of an error code is that it should be descriptive and it should offer two outputs:

* A plain descriptive sentence explaining the situation in the most precise manner.
* An ‘if-then’ situation where the user knows what to do with the error message once it is returned in an API call.

The error message returned in the result of the API call should be very descriptive and verbal. A code is preferred by the client who is well versed in the programming and web language but in the case of most clients they find it hard to get the code.

As I stated before, 404 is a bit problematic status when talking about Restful APIs. Does this status means that the resource was not found? or that there is not mapping to the requested resource? Everyone can decide what to use and where :)

#### [Guy Levin](https://blog.restcase.com/author/guy/)

Read [more posts](https://blog.restcase.com/author/guy/) by this author.

[*http://www.restcase.com*](http://www.restcase.com/)

#### Share this post