Learning Objectives: Error Handling

Learners will be able to…

- **recognize the HTTP error codes used in RESTful APIs**
- **support the errors on the client side**
- **deploy various retry strategies**

info

Make Sure You Know

This assignment assumes a basic knowledge of JavaScript, HTML and CSS and everything that has been covered in the previous assignments.

Limitations

We will not explain simple programming concepts, it is assumed that the user is familiar with these concepts.

REST status codes

Status codes are returned as part of the HTTP response to a REST API request. The status codes returned by a REST API call fall into one of the following categories:

Range	Category
`100-199`	Informational
`200-299`	Success
`300-399`	Redirection
`400-499`	Client Error
`500-599`	Server Error

In the previous assignment we covered the following successful status codes and one unsuccessful code. We will cover client and server errors in more detail in this assignment.

- `200 Ok` The request succeeded, and the item requested is being returned.
- `201 Created` Returned as the result of a POST request and, occasionally of a PUT request, the resource was successfully created.
- `202 Accepted` when the request was accepted for processing, but the processing has not been completed. E.g. it could be queued. May be used with POST, PUT, DELETE, PATCH.
- `204 No Content` similar to `200`, but is used when the server does not need to return any content.

And one unsuccessful:
* `404 Not Found` when the URI that was used in the request does not correspond to a known entity.

Clarification about `404`

The most common use of this status is when the client made an error in the URL.

When the requested URI points to a particular entity with a non-existent ID or its sub-entities, `404` should be used.

When the requested URI points to a collection of entities (e.g. /pets) `404 Not Found` should not be used. Even if it is a GET method and the result of the query after filtering is empty. `200 Ok` and a representation of an empty collection should be used for consistency.

important

404 Not Found

The most common use of this status is when the client made an error in the URL.

When the requested URI points to a resource with a non-existent ID or its sub-entities, `404` should be used.

When the requested URI points to a collection of entities (e.g. /pets) `404 Not Found` should not be used. Even if it is a GET method and the result of the query after filtering is empty. `200 Ok` and a representation of an empty collection should be used for consistency.

Another use of `404 Not Found` is when the requester is not authorized to access the information requested but saying so would confirm that the information exists. In this case using the `404` error can help maintain privacy.

Common error codes

The most common error codes

The 4xx error codes are handled by the client. For the 5xx error codes, there isn't much the client can do other than inform the user.

Error Code	Meaning
`400 Bad Request`	The request was not properly formed
`401 Unauthorized`	The client was not authenticated
`403 Forbidden`	Access to requested resource is forbidden
`404 Not Found`	Resource does not exist
`412 Precondition Failed`	Precondition provided in a header has failed
`500 Internal Server Error`	Situation encountered that server does not know how to handle
`503 Service Unavailable`	Server is not ready to handle the request, could be overloaded or down for maintenance

View a complete listing of HTTP status codes and their meanings.

We can generate some of these errors using the `curl` command. Paste the following commands into the terminal window.

- This one will give you a `400` - notice the typo in the time zone region.

```
curl -i https://timeapi.io/api/Time/current/zone?
        timeZone=Europe/Hamsterdam
```

- This one will give you a `401` - We haven't added a legitimate **Bearer Token**.

```
curl --request GET
        'https://api.twitter.com/2/tweets/search/recent?
        query=from:twitterdev' --header 'Authorization: Bearer
        $BEARER_TOKEN'
```

- This one will give you a `404` - the word `fake` at the end makes it a nonexistent page.

```
curl -I http://www.example.com/fake
```

Error codes by category

Access errors

- **401 Unauthorized**
  The required authentication credentials have either not been supplied or authorization is denied for those credentials.

info

Authentication vs Authorization

- Authentication is the process of verifying the identity of the user.
- Authorization verifies what the user has access to.

For example, you can sign into your health portal with your username and password, that is **authentication**. You only have access to your own medical records on the portal, those are requests you are **authorized** to access.

- **403 Forbidden**
  In some cases, for example where it might confirm the existence of a user, the server should not return this error. In situations like this `404 - Not Found` is the safer error to return because it gives nothing away about whether the resource exists or not.

Parameter errors

These are sometimes the result of a client being developed for a previous version of the server. In these cases using an incorrect URL is often the problem.

- **400 Bad Request**
  This is an error on the client end, the server cannot or will not process the request. This is often due to an incorrectly formed request.

- **405 Method Not Allowed**
  The method requested is not supported by the targeted resource. The server will return the methods currently supported by the resource.

- **406 Not acceptable**
  Client is requesting an unavailable data representation. In some cases the server may send a list of available data representations.
- **415 Unsupported Media Type**
  Client is sending data in a format that is not acceptable to the server. The server detects the data type by either looking at the Content-Type or Content-Encoding fields or by inspecting the data directly.
- **422 Unprocessable Entity**
  An example of this is when an XML request body is syntactically correct but the instructions are incorrect.

Concurrency errors

These are often the result of concurrent access to the same resource. For example if you have two browser windows open to the same data and you try to edit data in one that you had deleted in another before it had time to refresh.

- **424 Failed Dependency**
  Indicates that the requested operation can not be performed because it depends on another action that had failed.
- **409 Conflict**
  This indicates that there is conflict between the request and the current state of the resource. This can occur for example when you try to upload an older version of a file that already exists on the server.

Denial of service errors

These are often related to requests that may overload the server.

- **408 Request Timeout**
  This can happen when the client does not complete sending the request in a timely fashion.
- **429 Too Many Requests**
  The user has made too many requests in a given amount of time. This count could be user based or resource based - only allowing a certain number of requests per user or limiting the number of times a specific resource can be accessed. The response may also give an amount of time after which the request may be sent again.
- **413 Payload Too Large**
- **414 Request-URI Too Long**

- **431 Request Header Fields Too Large**
  These three status codes are all related to requests that are too large in one way or another.

Unexpected errors

- **500 Internal Server Error**
- **503 Service Unavailable**
  These both relate to the server possibly not anticipating certain conditions. There isn’t much that the client can do on their end, they can just inform the user about the error.
- **502 Bad Gateway**
- **504 Gateway Timeout**
  In both cases the server is informing the client that they could not complete the request because of proxy issues. Also not something that can be handled by the client.

Handling errors

The following are examples of how you might handle specific errors.

`400 Bad Request`

In this example, the error is handled in the request processing code and it clearly informs the caller about what is missing - the query.

```
app.get('/', (request, response) => {
  if (request.query.text === undefined) {
    return response.status(400).json('Query text is required');
  }
  response.json(request.query.text);
});
```

`404 Not Found`

In the example on the left you can see how to handle a **Not Found** error

info

Try it out in the pane on the left:

1. Click on the middle tab labeled “Terminal” to start the server.
2. Click on the tab labeled “app.js” to view the code.
3. Click on the remaining tab, which is the server URL, to try the application.

Things to try:

1. Click on the first **Get** and then **Try it out** and then **Execute** that will return all the items in the list.
2. Click on the **Get** where you can specify the id `/{id}` and then **Try it out**, then enter an id that does not exist, 5 for example. Click **Execute** and you can then see the Server response after this is executed.

Retry strategies

API calls to a network server can fail for many reasons, we looked at the range of possible errors on previous pages. Some errors require fixing the request on the client side, others may be transient and the request needs to be retried.

What can cause failure status codes

Rate Limiting

Services that provide access to multiple clients need to ensure that no single client monopolizes all the services, this is called rate limiting. A rate limit is the maximum number of calls that can be made to an API in a defined time interval. Rate limiting can also be used to reduce access in cases where there are tiered services, such as when you have unlimited access to resources with a paid subscription but a free subscription might allow only a single access per day.

Limiting the number of connections

Some services such as databases may start to fail or bog down if there are too many simultaneous connections. To prevent this, the service will limit the number of connections.

Why are retry strategies necessary?

In the case of errors that might require another try, it is important to have a **strategy** that will increase the likelihood of success on future calls. The goal is not to overwhelm the server with retry attempts as that can exacerbate the problem. The other important condition to consider when deciding whether to retry is whether a request is **idempotent**. Retrying requests that are not **idempotent** can lead to conflicts.

These are the errors that would likely benefit from a retry:

Status Code	Likely reason	Recommended action
`408 Request Timeout`	Client is taking too long to send the request	Could be a request URL error or a connection issue - check before retrying
`429 Too Many Requests`	Client sent too many requests in a range of time	

| Wait at least one second before retrying |
| `5xx Server Errors` | Something went wrong on the server end | Use one of the strategies below |

Retry Strategies

A failure in one location for a resource may cause requests to be directed to other locations and this could cause them to overload as well, this is called a **cascading failure**. A client developer should be aware that requesting retries increases the load on the server and may result in more harm to a system that is already overloaded.

Linear Backoff

This strategy pauses for a fixed interval between each retry. The problem with this system is choosing the wait time. If it's too short, the issue on the server end may not have been resolved. If it's too long, the response time may be annoying to the user. This strategy may exacerbate overload issues on the server end.

Exponential Backoff

This is a retry system that uses ever increasing wait times between requests. Initially the wait time is short and this works well for issues that may have been quickly resolved. Eventually though the wait time grows too large so there needs to be a cut off in retries once the wait time hits a certain threshold.

Jitter

A **jitter** strategy is often used in combination with **exponential backoff**. Adding **jitter** to a retry strategy introduces randomness to the time intervals between requests and helps spread retries more evenly so they don't all come in at the same time and overwhelm the server.

Circuit Breaker Pattern

The **Circuit Breaker Pattern** was designed to prevent repeated retries of operations that are likely to fail. It also provides a way for an application to determine whether the problem has been resolved. Retries are routed through the **Circuit Breaker** which keeps track of the number of failures and if that number is too high, it may block the retry. A throttled down number of retries are permitted and are used to determine whether the problem has been fixed. More information about circuit breakers.