REST API General Guidelines

DRAFT

Definitions:

Authentication vs Authorization: Authentication is "I am user abc". Authorization is "As user abc, I have permission to do xyz"

Entity: Request or Response bodies (i.e. JSON).

Headers:

Whenever data can be immediately actionable by a client, server, or a proxy[1] between them, it may be useful to add this data to a header.

Examples:

- Caching: Include headers that support caching directives and data freshness.
- Authentication: Use Authorization header, or a user-defined header if using a custom authentication scheme.
- Request preconditions: Useful when modifying (i.e. POST, PUT, or PATCH) an endpoint/resource; Requests which might lead to 409, 412 typically use preconditions.
- Server, Application, or other Response Metadata: Typically user-defined; useful for application name and versioning, for example.

[1] A proxy is anything which may read and act based on data in a request or response, such as a load balancer, gateway, or filter.

Request methods:

GET - Get a resource. In general, you should not include an entity for GET and HEAD requests, because servers are allowed to ignore entities. GET and HEAD requests are also safe, and should not modify a resource.

HEAD - Same semantics as get, but no entity returned.

POST - Create a resource or sub-resource.

PUT - Replace a resource. Typically this should include a full representation of what you plan on replacing. It can be used similar to how PATCH (i.e. modify instead of replace) might be used, as PATCH is newer and isn't necessarily as well supported as PUT.

PATCH - Modify a resource. Typically this can include a diff of what you plan on modifying. There's several different ways of performing a diff with JSON, including JSON-Patch and JSON-Merge. JSON-Merge is likely most simple and useful.

See also:
https://tools.ietf.org/html/rfc6902 (JSON patch)
https://tools.ietf.org/html/rfc7396 (JSON merge)

DELETE - Delete a resource. Typically this is for deleting full objects, and "deleting" parts of objects should actually use PATCH. The reason why is some clients, and as far as I know, some servers, can choose to ignore request entities sent with a delete request.

HTTP status codes

200 - OK

Use this when a request succeeds and you want to send a message body, except for the rare case where the request method was HEAD.

201 - Created

Usually only returned from POST requests. A response body may be returned to reflect the new state of the resource. This is useful in the case that a resource may have auto-generated fields or keys, for example.

204 - No Content

Usually returned when a PUT/PATCH or a DELETE request is successful and an entity does not need to be returned. By definition, there must be no body.

301, 302, 303

There are reasons why these should be used, but you should use them with caution because some clients may handle them automatically while others may not.

One example is, if a user goes to create something that already exists, instead of sending them a 409, for example, you can send them a 303 instead pointing to the resource. The caveat of this is that many clients and browsers will automatically redirect to that link in See Other, so it's sometimes hard to convey additional logic to the client/user that their original request was not performed and why.

304 - Not modified

For cache-enabled clients, or requests that just want to see if a resource has been modified, return this on conditional requests. If you aren't implementing client-side caching, this isn't very useful.

400 - Bad Request

Typically a client provided improper parameters in a GET request. It may have also provided a request entity which was unprocessable (i.e. malformed JSON, or JSON fails validation).

401 - Unauthorized

This is worded a bit poorly. Really, this means "Unauthenticated". It's important to note that, according to the HTTP spec, the server is required to include a WWW-Authenticate header. The WWW-Authenticate header allows for different types of authentication mechanisims.

403 - Forbidden

This is really "Unauthorized". Use this when an resource requires autheticated client/user does not have permission to perform the requested action.

404 - Not Found

When a RESTful resource does not exist, this is the default status code to return.

409 - Conflict

When updating a resource, if the payload is found to be conflicting with the current representation (concurrent modification, constraint violation), of the resource in the database, this error code is returned. This indicates that the request will not succeed, even if the payload is modified. See 412 also.

412 - Precondition Failed

For requests which may update a resource, such as POST,PUT/PATCH, and DELETE requests, an If-unmodified-since header MAY used to prevent lost update and write/write conflicts. Basically, the 412 can let the client know a resource has been modified since it has last seen it.

For 409 and 412, see also: https://www.ietf.org/rfc/rfc5789.txt

Respones:

There are two default responses. One is for responses which return a singular object, and one is for response which return multiple objects. If a response is known to return one and only one object, you should use the scalar response object. If a response may contain zero or more objects, you should always use the Vector response object. There are cases where you should return a 404 if no objects are found, but there are also cases where an empty list (with the response metadata) is also useful, namely queries. In general, a scalar response should always return a non-empty object, because an empty object should always be a 404. This makes client code which is merely performing an existence check a bit simpler.

Scalar Response:

```
{
    "type": "object",
    "properties": {
        "result": {
            "type": "object",
            "required":true
        },
        "metadata": {
            "type": "object",
            "required":false
        }
    }
}
```

Vector/Array Response:

```
{
    "type": "object",
    "properties": {
        "results": {
           "type": "array",
            "required":true
        },
        "metadata": {
            "type": "object",
            "required":false
        },
        "range": {
            "type": "object",
            "required":false
        }
    }
}
```

Response Ranges, Pagination, and Intervals:

There are several possibilites to use for "range" metadata. They keywords I've used are somewhat arbitrary, but I just wanted to show the several ways you can represent a "range" or responses. In practice, I believe the first slice format is easiest to use across different types (timestamps, integers) and easy translate to database notation (even if it requires a tiny bit of math). The alternatives have the benefit that they make more sense if you are using timestamps or strings. The interval type could potentially use ISO 8601 interval notation instead.

Pagination

```
"range":{
    "type": "pagination",
    "page": 2,
    "per_page": 20,
    "pages": 24      # optional
}
```

Slice

```
"range": {
    "type":"slice",
    "offset":200,
    "max": 20,       # optional, alternatively "count"
    "length":400     # optional
}
```

Slice Alternative 1:

```
"range": {
    "type":"slice",
    "offset": 200,
    "limit": 220,    # optional
    "length": 400    # optional
}
```

Slice Alternative 2:

```
"range": {
    "type": "slice",
    "start": 200,
    "stop": 220,     # optional
    "length": 400    # optional
}
```

Interval

```
"range":{
    "type":"interval",
    "from":"2014-12-01",
    "to":"2015-01-01",
    "latest":"2015-03-01" # optional
}
```

Errors:

HTTP responses with error details should use a 4XX status code to indicate a client-side failure (such as invalid authorization, or an invalid parameter), and a 5XX status code to indicate server-side failure (such as an uncaught exception).

For a given client, there is a caveat in this: When an application is behind a gateway, proxy, or resides in some sort of application container, the intermediary itself may also respond with a 4xx or 5xx error.

Because of this, the response body should include enough information for the client to recognize the source of the error. When you include a response entity, your client can evaluate the status code (first and foremost), and then it can attempt to evaluate the response. If it discover the error response isn't what it expected, the client can make additional assumptions about the error response in general, such as treating a 404 as a server error.

Schema

```
{
    "type": "object",
    "properties": {
        "exception": {
            "type": "string",
            "required":true
        },
        "message": {
            "type": "string",
            "required":false # Might be required
        },
        "cause": {
            "type": "string",
            "required":false # Might be required
        }
    }
}
```

Example

```
{
    "exception":"ClientException",
    "message":"Description of the error.",
    "cause":"[Detailed Stack Trace]"
}
```

Examples Illustrating different types of Client Errors

```
GET /users/bvan
```

Resource not Found

```
    Status: 404
    Content-Type: application/json
    {
        "exception":"UserNotFound"
        "message":"The user 'bvan' does not exist"
    }
```

Internal Application Error

```
    Status: 500

    Content-Type: application/json
    {
        "exception":"DatabaseException"
        "message":"Unable to acquire a connection to the database"
        "cause":"[stack trace]"
    }
```

Intermediary -> Application Error

```
    Status: 500

    Content-Type: text/html


    <html>
      <body>
        Your application is configured correctly but isn't responding.
      </body>
    </html>
```

Applications not confiured or Application Server Down

```
    Status: 404
    Content-Type: text/html
```

```
<html>
  <body>
    I don't think you configured your application is running or confgiured
  </body>
</html>
```

See Also:

https://github.com/18f/api-standards