Good routes versus bad routes

## Introduction

Naming your API properly is the first step in designing a good API. When the API name follows a convention, it provides lots of information about the API and its purpose. To create a meaningful API endpoint, you need to follow some simple guidelines and rules.

In this reading, you will learn about API naming conventions and familiarize yourself with good API endpoints vs. bad API endpoints, or good and bad routes.

## Rule 01: Everything in lowercase, with hyphens and not abridged

The URI of your API should always be in lowercase. Do not use camelCase, PascalCase or Title case when you design your API. Also, separate multiple words using hyphens, not underscores. Do not keep abridged, or shortened, words in your URI; always use the full and meaningful form.

If your API accepts a variable, you should always represent it in camelCase, and wrap it inside a set of curly braces {}.

Let’s examine the following examples.

| **URI** | **Status** | **Why** |
| --- | --- | --- |
| /customers | Good | Plural and lowercase |
| /customers/16/phone-number | Good | Lowercase and hyphen used to separate words |
| /customers/16/address/home | Good | Lowercase, no trailing slash, displays the hierarchical relationship with forward slashes. |
| /users/{userId} | Good | Variable in camelCase, and no hyphen in the variable name |
| /Customers | Bad | Title case |
| /generalMembers | Bad | camelCase, no hyphens to separate words |
| /MenuItems  /GeneralMembers | Bad | Pascal case |
| /customers/16/tel-no | Bad | Abbreviation |
| /customers/16/phone\_number | Bad | Underscores |
| /customers/16/phonenumber | Bad | No separation of words |
| /users/{user-id} | Bad | Variable should be in camelCase, and hyphen between words |

## Rule 02: Use a forward slash to indicate a hierarchical relationship

In your API URI, always use the forward slash to indicate a hierarchical relationship. To understand this rule, consider the following scenarios and the relationship from the API endpoints.

A store can have customers who have placed many orders and each of these orders can have delivery addresses, menu items and bills.

| **URI** | **Status** |
| --- | --- |
| /store/customers/{customerId}/orders | Good |
| /store/orders/{orderId} | Good |
| /store/orders/{orderId}/menu-items | Good |

Similarly, a library can have books from many authors. Each of these books has an ISBN number.

| **URI** | **Status** |
| --- | --- |
| /library/authors/books | Good |
| /library/book/{bookId}/isbn | Good |

## Rule 03: Use nouns for resource names, not verbs

One of the most prominent features of REST APIs is that it uses nouns to indicate resources, not verbs. And you should always stick to this rule when designing your API. You should also use plural nouns to indicate a collection.

| **URI** | **Expects** | **Status** | **Why** |
| --- | --- | --- | --- |
| /orders | Collection | Good | Uses a noun, not a verb |
| /users/{userId} | Single user | Good | Uses a noun and proper hierarchical relationship and naming convention |
| /order | Collection | Bad | Uses plural nouns for collections |
| /getOrder | Single resource | Bad | Uses a verb, camelCase |
| /getUser{userId} | Single user | Bad | Uses a verb, camelCase |

## Rule 04: Avoid special characters

You should always avoid special characters in your API endpoints. They can be confusing and technically complex for your users. Consider the following bad examples.

| **URI** | **Status** | **Why** |
| --- | --- | --- |
| /users/12|23|23/address | Bad | Special character | |
| /orders/16/menu^items | Bad | Special character ^ |

If your API can accept multiple user ids, then they should be separated using a comma, as demonstrated below.

| **URI** | **Status** | **Why** |
| --- | --- | --- |
| /users/12,23,23/address | Good | Uses a comma for separation |

## Rule 05: Avoid file extensions in URI

You should always avoid file extensions in your API names. For example, if your API can deliver an output in both JSON and XML format, it should never look like this.

| **URI** | **Status** | **Why** |
| --- | --- | --- |
| /sports/basketball/teams/{teamId}.json | Bad | File extension at the end |
| /sports/basketball/teams/{teamId}.xml | Bad | File extension at the end |

Instead, your client should be able to indicate its expected format in a query string, just like this.

| **URI** | **Status** | **Why** |
| --- | --- | --- |
| /sports/basketball/teams/{teamId}?format=json | Good | No file extension |
| /sports/basketball/teams/{teamId}?format=xml | Good | No file extension |

Similarly, if your API is serving a static file, for example, CSS or JavaScript files, you should use endpoints like the following to deliver the minified and original source file. You can also use a query string to get the minified or original version. Some API developers use the output format like file extension at the end of the regular API endpoints, which is also bad practice.

| **URI** | **Status** | **Why** |
| --- | --- | --- |
| /assets/js/jquery/3.12/min | Good | No file extension |
| /assets/js/jquery/3.12/source | Good | No file extension |
| /assets/js/jquery/3.12/?format=min | Good | No file extension |
| /assets/js/jquery/3.12/?format=source | Good | No file extension |
| /menu-items?format=json | Good | Perfectly named endpoint with expected output format in a query string |
| /menu-items.json | Bad | Uses the expected output format as the file extension |

## Rule 06: Use query parameters to filter when necessary

When designing your API, you should always perform data filtering using a query string. This is the same when you expect some extra parameters, like the number of items per page and page number.

Consider this example of a travel site. You want to find which locations a particular user has traveled to. And then you want to know which locations in the USA the user has already seen.

| **URI** | **Status** | **Why** |
| --- | --- | --- |
| /users/{userId}/locations | Good | Hierarchical |
| /users/{userId}/locations?country=USA | Good | camelCase, no separation of words |
| /articles?per-page=10&page=2 | Good | Proper use of query string |
| /users/{userId}/locations/USA | Bad | Doesn't use a query string to filter data |
| /articles/page/2/items-per-page/10 | Bad | Redundant and obscure |

## Rule 07: No trailing slash

When sharing your API endpoint with others in your team, or in public, avoid using a trailing slash at the end of your API endpoints. Consider the following examples.

| **URI** | **Status** | **Why** |
| --- | --- | --- |
| /users/{userId} | Good | No trailing slash |
| /articles/{articleId}/author | Good | No trailing slash |
| /users/{userId}/ | Bad | Trailing slash |
| /articles/{articleId}/author/ | Bad | Trailing slash |

## Conclusion

Now you understand how to create REST API endpoints with good names. Remember, a consistent naming strategy for your API is one of the most important design decisions for the whole project!

RESTful APIs are considered to be stateless. What this means is the state is saved

Only with the client.

**REST best practice**s

You should implement caching for your APIs

Kiss(keeping things simple)

filtering, ordering and pagination,

versioning,

cacheing, and rate limiting and monitoring.

**Security and authentication in REST API**

**SSL**(Secure Socket Layer) - SSL encrypts your data and protect your data when it leaves your browser and the web server. When you set up the SSL certificates properly your APIs can be served over HTTPS.

**Signed URLs -** Signed URLs give a client application limited access to a specific resource for a brief period of time. With a signed URL, every time an API is called, a particular piece of text called a signature is included with the URL.