


DRAFT: REST Guidelines

Created by Alexander Daubler, last modified by Natalya Bormusova 48 minutes ago

 Some of the content of this page has been taken from here (where additional information can be found): [API Best Practice Guide](#)

- [Introduction](#)
- [API Resource Naming Conventions](#)
 - [Use Nouns, not Verbs](#)
 - [Use Plural Nouns](#)
 - [Use concrete domain-specific resource names \(nouns\)](#)
 - [Multi-word Resource Names](#)
 - [Resource Path Depth](#)
- [HTTP Verb Conventions](#)
 - [Identifying Actions](#)
 - [Map actions to HTTP verbs](#)
 - [GET](#)
 - [POST](#)
 - [PUT](#)
 - [PATCH](#)
 - [DELETE](#)
- [Query Parameter Conventions](#)
 - [Resource Retrieval](#)
 - [Filtering Parameters for Resource Narrowing](#)
 - [Pagination Parameters](#)
 - [Sorting Parameters](#)
- [URI Parameter Conventions](#)

Introduction

The key principles of REST support separating an API into logical resources and manipulating these using HTTP methods, where each method has a specific meaning. A resource represents object types within your domain. For example, **/customers** could represent all customers of an organization.

The only allowed HTTP methods for your API requests are the following:

HTTP Method	Description
GET	Retrieve an entity from the service
POST	Create an entity on the service side
PUT / PATCH	Implements a persistent change of the entity on the service
DELETE	Removes an entity from the service

API Resource Naming Conventions

Use Nouns, not Verbs

An API contract describes resources, so the only place where actions should appear is in the HTTP methods. Instead of thinking of actions (verbs), it's often helpful to think about putting a message in a letter box, e.g., instead of having the verb cancel in the URL, think of sending a message to cancel an order to the cancellations letter box on the server side.

The inclusion of actions, such as **getOrders** or **createOrder**, within the resource name must be avoided. The type of action to be performed on the resource is implied by the HTTP method i.e. GET or DELETE. This practice is important because it helps to keep the total number of interfaces down, promotes a consistent and easy to understand approach, and helps to encapsulate common functionality under a single resource.

Examples:

GET /customers
POST /customers/cancellation-requests
GET /getCustomers
POST /getCustomers
POST /customers/cancel

Use Plural Nouns

It is common practice to standardize on using plural nouns over a mixture of both singular and plural nouns in URIs. This makes it consistent and predictable for API Consumers.

Think of a resource as a container of elements you can interact with - so a **POST /customers** request means that you add one more element to the set of customers. To specify a singleton resource, use the URN. For example, **/customers** relates to all customer resources, whereas **/customers/123** relates to the customer with the identity of 123.

Examples:

GET /customers
GET /customers/123
POST /customers/cancellation-requests
GET /customer
POST /customers/cancellation-request

User concrete domain-specific resource names (nouns)

Avoid making a resource name too abstract. API resources represent elements of the service's domain model. Using domain-specific resource names helps developers in understanding the functionality and basic semantics of your resources. It also reduces the need for further documentation outside the API definition.

For example, **/sales-order-items** is preferable to **/order-items** in that it clearly indicates which business object it represents. Tunnelling a number of objects through an abstract resource name makes it difficult to understand what the resource actually represents or how the API resource should be used. Alternatively, you can also use sub-resources for improving the comprehensibility of your API, e.g. instead of having a resource **/gdpr-consent-documents** you could also use **/documents/gdpr-consents** (this becomes especially relevant in case that you expect other types of documents to be sent in the future).

Examples:

GET /sales-order-items
POST /gdpr-consent-documents
POST /documents/gdpr-consents
GET /items

Multi-word Resource Names

For resource names composed of 2 words or more (e.g. meter + readings) the convention is to use kebab-case syntax

Examples:

GET /meter-readings
GET /meterreadings
GET /meter_readings
GET /meterReadings

Resource Path Depth

Similar to resource tunnelling, a URI formed by a long chain of resources increases complexity and reduces the ease of consumption of the API. As a maximum depth, a URI should adhere to **/primaryresource/{id}/subresource**, i.e. there should be just a single URI parameter.

In case you use sub-resources "just" for structuring purposes (e.g. **/documents/gdpr-consents**) there is no dedicated max value for the depth but it must be ensured that the API and its functionality is still comprehensible to the potential consumer.

Exceptions to this recommendation need to be documented explicitly.

Examples:

POST /customers/123/orders	
POST /customers/123/orders/987/status	typically it should be sufficient to call POST /orders/987/status to get the same result

HTTP Verb Conventions

Identifying Actions

It is important to identify the actions that can be performed upon the resources during their lifecycle. Using the example of a **/purchase-orders** resource, it could be created, updated with a new status when paid, retrieved as part of an accounting process and, finally deleted to meet data retention requirements.

Map actions to HTTP verbs

GET

Use Case(s):

- Read single element from the backend resource
- Read collection from the backend resource

Usage:

- When trying to retrieve a single resource use a URI parameter
 - Response should be a single element then
- When trying to query for specific elements within the resource use one or more query parameters
 - Response should be a collection then
 - In case the query params are very complex in nature (i.e. an array needs to be provided for the query), restrain from using query parameters and design a dedicated **queries** endpoint for the resource and send a POST request to it
- GET requests must NOT have a request body payload, and GET body must be ignored if present

Typical HTTP Response Codes:

- GET requests for a single element generate a 200 in case the element can be retrieved successfully
- GET requests for a single element generate a 404 if the element does not exist in the backend system
- GET requests for collections will return a 200 (response collection may be empty)

Examples:

GET /customers	
GET /customers/12345	
GET /customers? lastname=Huber&firstname=Alexander	
GET /customers?id=12345	use a URI param instead
GET /customers/12345?lastname=Huber	combining a URI param with one or more query params must be avoided

POST

Use Case(s):

- Create a single element on the backend resource
- Create a collection on the backend resource
- POST should be used for scenarios that cannot be covered by any of the other methods sufficiently. Examples for this could be complex queries (which would typically be designed using a GET endpoint, but in case you want to provide the query params within the request body a POST endpoint is preferable) and complex deletions. In such cases, it is fundamental to document the fact that the usage of POST is actually intended.

Usage:

- Details on the element(s) to be created are sent in the request body payload
- Query parameters are not allowed for POST requests and will be ignored if present
- On a successful POST request, the server will create one or multiple new resources and provide their IDs in the response
- Custom headers to manipulate a backend resource are not allowed for POST requests and will be ignored if present

Typical HTTP Response Codes:

- Successful POST requests will usually generate 201 (if resources have been created), and 202 (if the request was accepted but has not been finished yet).

Examples:

POST /customers	create a new customer (or a set of customers)
POST /customers/queries	a special case to make a GET on the "customers" resource with a very complex filter criteria provided in body
POST /customers/deletion-requests	creates a new deletion request for customers (e.g. in case that several customers need to be deleted with a single call)
POST /customers/12345	typically no ID is provided when creating a new customer (should be done by the backend system), but if that a case provide ID in the request body
POST /customers? lastname=Huber&firstname=Alexander	provide data in the request body

PUT

Use Case(s):

- Update single entire element on the backend resource (this should be the default approach)
- Update entire collections on the backend resource
- Do not use PUT for creating elements
 - prefer POST instead
 - in case an API is implementing UPSERT functionality (i.e. a new object is created in case that the provided ID is not yet available in the backend application) then PUT is the right verb to use (please make sure that this is properly documented)

Usage:

- When trying to update a single element use a URI parameter to identify it
- When trying to update a collection, provide the ID(s) of the element as part of the request body payload
- Query parameters are not allowed for PUT requests and will be ignored if present
- On successful PUT requests, the server will replace the entire **element** addressed by the URL with the representation passed in the payload
- Custom headers to manipulate a backend resource are not allowed for PUT requests and will be ignored if present

Typical HTTP Response Codes:

- Successful PUT requests will usually generate 200

Examples:

PUT /customers/12345	update an existing customer
PUT /customers	update a collection of customers (IDs must be provided within body request)
PUT /customers?id=12345	provide ID as URI param
PUT /customers? ids=12345;23466	IDs must be provided within body request when updating a collection of customers

PATCH

Use Case(s):

- Update parts of a single element on the backend resource (this should be the default approach), i.e. where only a specific subset of resource fields should be replaced
- Update collections on the backend resource

Usage:

- When trying to update a single element partially use a URI parameter to identify it
- When trying to update a collection partially, provide the ID(s) of the element as part of the request body payload
- In PATCH payloads there are typically no mandatory fields
- Query parameters are not allowed for PATCH requests and will be ignored if present
- Custom headers to manipulate a backend resource are not allowed for PATCH requests and will be ignored if present

Typical HTTP Response Codes:

- Successful PATCH requests will usually generate 200

Examples:

PATCH /customers/12345	update an existing customer
PATCH /customers	update a collection of customers (IDs must be provided within body request)
PATCH /customers?id=12345	provide ID as URI param
PATCH /customers? ids=12345;23466	IDs must be provided within body request when updating a collection of customers

DELETE

Use Case(s):

- Delete single element from the backend resource
- Do not use DELETE for deleting collections of elements from the backend resource
 - prefer POST instead (by sending a POST request to a dedicated **/bulk-deletions** or **/deletion-requests** endpoint of the resource); it should be clear from the name of the endpoint that this something which needs to be used with highest caution

Usage:

- To delete a single element use a URI parameter to identify it
- Query parameters are not allowed for DELETE requests and will be ignored if present
- DELETE requests must NOT have a request body payload, and DELETE body must be ignored if present

Typical HTTP Response Codes:

- Successful DELETE requests will usually generate 200
- Failed DELETE requests will usually generate 404 (if the resource cannot be found)

Additional Remarks:

- After deleting a resource with DELETE, a GET request on the resource is expected to return 404 (not found). Under no circumstances the resource must be accessible after this operation on its endpoint.

Examples:

DELETE /customers/12345	delete an existing customer
DELETE /customers?id=12345	provide ID as URI param
DELETE /customers? firstname=Alexander&lastname=Huber	in case params are required consider a POST /customers/deletion-requests endpoint

Query Parameter Conventions

Query parameters are not applicable for POST, PUT, PATCH and DELETE methods so this section applies for GET requests only.

Resource Retrieval

Often a GET request offers a wide variety of query params with the aim to find and retrieve a single element. However, by default a GET endpoint using query params should always return a collection of elements - if you decide to go with just a single element in the response this needs to be documented explicitly.

Examples:

GET: /customers? firstname=Alexander&lastname=Huber&birthdate=14.08.1966	return customers matching the provided query params
GET: /customers	do you actually want to return all customers?

Filtering Parameters for Resource Narrowing

Exposing a collection of resources through a single URI can lead to applications fetching large amounts of data when only a subset of the information is required. In order to provide efficiency, the API should allow passing a set of filter params.

Examples:

GET: /contracts?status=active	return active contracts only
--------------------------------------	------------------------------

Pagination Parameters

Other way to reduce large amounts of data in a response is using pagination functionality. It can be achieved by passing two query parameter offset and limit.

Offset - specify an arbitrary offset at which to start retrieving resource elements

Limit - specify the number of elements to return in one API call

Examples:

GET: /purchase-orders?offset=1&limit=1000	return 1000 purchase orders starting from first
--	---

Sorting Parameters

Sorting parameters can be utilised to specify how the matching resources are provided to the API consumer without having an influence on the result set.

Examples:

GET: /purchase-orders?order-by=date&order-type=desc	return all purchase orders sorted by date (descending)
--	--

Filtering capability and the use of parameters in this regard requires implementation effort, so it needs to be considered during the API design phase.

URI Parameter Conventions

- For PUT, PATCH and DELETE methods having a URI parameter as the last part of the URI is the default approach
 - In case that such an parameter can't be provided (as the field(s) identifying the resource to be changed / deleted are part of the payload) this must be explicitly documented
- For POST methods the URI parameter must not be the last part of the URI - it can be used to identify the parent element when using a sub-resource, however
- For GET methods both approaches are possible - by using a URI parameter as the last part of the URI you define that the response will contain a single element only

Good / Not so good examples:

PUT /customers/123	
PATCH /customers/123	
DELETE /customers/123	
POST /customers/123/orders	
GET /customers/123	
GET /customers/123/orders	
PUT /customers	update a collection of customers (IDs must be provided within body request)
PATCH /customers	partially update a collection of customers (IDs must be provided within body request)
DELETE /customers	
POST /purchase-orders/123	

No labels