Table of Contents

HTTP API Design Guide

This guide describes a set of HTTP+JSON API design practices, originally extracted from work on the Heroku Platform API.

This guide informs additions to that API and also guides new internal APIs at Heroku. We hope it's also of interest to API designers outside of Heroku.

Our goals here are consistency and focusing on business logic while avoiding design bikeshedding. We're looking for *a good, consistent, well-documented way* to design APIs, not necessarily *the only/ideal way*.

We assume you're familiar with the basics of HTTP+JSON APIs and won't cover all of the fundamentals of those in this guide.

Available for online reading and in multiple formats at gitbook.

We welcome contributions to this guide.

See Summary for Table of Contents.

Translations

- Portuguese version (based on fba98f08b5), by @Gutem
- Spanish version (based on 2a74f45), by @jmnavarro
- Korean version (based on f38dba6), by @yoondo
- Simplified Chinese version (based on 337c4a0), by @ZhangBohan
- Traditional Chinese version (based on 232f8dc), by @kcyeu
- Turkish version (based on c03842f), by @hkulekci

Foundations

The Foundations section outlines the design principles upon which the rest of the guide builds.

Separate Concerns

Keep things simple while designing by separating the concerns between the different parts of the request and response cycle. Keeping simple rules here allows for greater focus on larger and harder problems.

Requests and responses will be made to address a particular resource or collection. Use the path to indicate identity, the body to transfer the contents and headers to communicate metadata. Query params may be used as a means to pass header information also in edge cases, but headers are preferred as they are more flexible and can convey more diverse information.

Require Secure Connections

Require secure connections with TLS to access the API, without exception. It’s not worth trying to figure out or explain when it is OK to use TLS and when it’s not. Just require TLS for everything.

Ideally, simply reject any non-TLS requests by not responding to requests for http or port 80 to avoid any insecure data exchange. In environments where this is not possible, respond with `403 Forbidden`.

Redirects are discouraged since they allow sloppy/bad client behaviour without providing any clear gain. Clients that rely on redirects double up on server traffic and render TLS useless since sensitive data will already have been exposed during the first call.

Require Versioning in the Accepts Header

Versioning and the transition between versions can be one of the more challenging aspects of designing and operating an API. As such, it is best to start with some mechanisms in place to mitigate this from the start.

To prevent surprise, breaking changes to users, it is best to require a version be specified with all requests. Default versions should be avoided as they are very difficult, at best, to change in the future.

It is best to provide version specification in the headers, with other metadata, using the `Accept` header with a custom content type, e.g.:

```
Accept: application/vnd.heroku+json; version=3
```

Support ETags for Caching

Include an `ETag` header in all responses, identifying the specific version of the returned resource. This allows users to cache resources and use requests with this value in the `If-None-Match` header to determine if the cache should be updated.

Provide Request-Ids for Introspection

Include a `Request-Id` header in each API response, populated with a UUID value. By logging these values on the client, server and any backing services, it provides a mechanism to trace, diagnose and debug requests.

Divide Large Responses Across Requests with Ranges

Large responses should be broken across multiple requests using `Range` headers to specify when more data is available and how to retrieve it. See the Heroku Platform API discussion of Ranges for the details of request and response headers, status codes, limits, ordering, and iteration.

Requests

The Requests section provides an overview of patterns for API requests.

Accept serialized JSON in request bodies

Accept serialized JSON on `PUT` / `PATCH` / `POST` request bodies, either instead of or in addition to form-encoded data. This creates symmetry with JSON-serialized response bodies, e.g.:

```
$ curl -X POST https://service.com/apps \
    -H "Content-Type: application/json" \
    -d '{"name": "demoapp"}'

{
  "id": "01234567-89ab-cdef-0123-456789abcdef",
  "name": "demoapp",
  "owner": {
    "email": "username@example.com",
    "id": "01234567-89ab-cdef-0123-456789abcdef"
  },
  ...
}
```

Resource names

Use the plural version of a resource name unless the resource in question is a singleton within the system (for example, the overall status of the system might be `/status`). This keeps it consistent in the way you refer to particular resources.

Actions

Prefer endpoint configurations that don't require special actions. In cases where actions are needed, clearly dileanate them with the `actions` prefix:

```
/resources/:resource/actions/:action
```

e.g. to stop a particular run:

```
/runs/{run_id}/actions/stop
```

Actions on collections should also be minimized. Where needed, they should use a top-level actions dilenation to avoid namespace conflicts and clearly show the scope of action:

```
/actions/:action/resources
```

e.g. to restart all servers:

```
/actions/restart/servers
```

Use consistent path formats

Downcase paths and attributes

Use downcased and dash-separated path names, for alignment with hostnames, e.g:

```
service-api.com/users
service-api.com/app-setups
```

Downcase attributes as well, but use underscore separators so that attribute names can be typed without quotes in JavaScript, e.g.:

```
service_class: "first"
```

Support non-id dereferencing for convenience

In some cases it may be inconvenient for end-users to provide IDs to identify a resource. For example, a user may think in terms of a Heroku app name, but that app may be identified by a UUID. In these cases you may want to accept both an id or name, e.g.:

```
$ curl https://service.com/apps/{app_id_or_name}
$ curl https://service.com/apps/97addcf0-c182
$ curl https://service.com/apps/www-prod
```

Do not accept only names to the exclusion of IDs.

Minimize path nesting

In data models with nested parent/child resource relationships, paths may become deeply nested, e.g.:

```
/orgs/{org_id}/apps/{app_id}/dynos/{dyno_id}
```

Limit nesting depth by preferring to locate resources at the root path. Use nesting to indicate scoped collections. For example, for the case above where a dyno belongs to an app belongs to an org:

```
/orgs/{org_id}
/orgs/{org_id}/apps
/apps/{app_id}
/apps/{app_id}/dynos
/dynos/{dyno_id}
```

Responses

The Responses section provides an overview of patterns for API responses.

Return appropriate status codes

Return appropriate HTTP status codes with each response. Successful responses should be coded according to this guide:

- `200` : Request succeeded for a `GET` , `POST` , `DELETE` , or `PATCH` call that completed synchronously, or a `PUT` call that synchronously updated an existing resource
- `201` : Request succeeded for a `POST` , or `PUT` call that synchronously created a new resource. It is also best practice to provide a 'Location' header pointing to the newly created resource. This is particularly useful in the `POST` context as the new resource will have a different URL than the original request.
- `202` : Request accepted for a `POST` , `PUT` , `DELETE` , or `PATCH` call that will be processed asynchronously
- `206` : Request succeeded on `GET` , but only a partial response returned: see above on ranges

Pay attention to the use of authentication and authorization error codes:

- `401 Unauthorized` : Request failed because user is not authenticated
- `403 Forbidden` : Request failed because user does not have authorization to access a specific resource

Return suitable codes to provide additional information when there are errors:

- `422 Unprocessable Entity` : Your request was understood, but contained invalid parameters
- `429 Too Many Requests` : You have been rate-limited, retry later
- `500 Internal Server Error` : Something went wrong on the server, check status site and/or report the issue

Refer to the HTTP response code spec for guidance on status codes for user error and server error cases.

Provide full resources where available

Provide the full resource representation (i.e. the object with all attributes) whenever possible in the response. Always provide the full resource on 200 and 201 responses, including `PUT` / `PATCH` and `DELETE` requests, e.g.:

```
$ curl -X DELETE \
  https://service.com/apps/1f9b/domains/0fd4

HTTP/1.1 200 OK
Content-Type: application/json;charset=utf-8
...
{
  "created_at": "2012-01-01T12:00:00Z",
  "hostname": "subdomain.example.com",
  "id": "01234567-89ab-cdef-0123-456789abcdef",
  "updated_at": "2012-01-01T12:00:00Z"
}
```

202 responses will not include the full resource representation, e.g.:

```
$ curl -X DELETE \
  https://service.com/apps/1f9b/dynos/05bd

HTTP/1.1 202 Accepted
Content-Type: application/json;charset=utf-8
...
{}
```

Provide resource (UU)IDs

Give each resource an `id` attribute by default. Use UUIDs unless you have a very good reason not to. Don't use IDs that won't be globally unique across instances of the service or other resources in the service, especially auto-incrementing IDs.

Render UUIDs in downcased `8-4-4-4-12` format, e.g.:

```
"id": "01234567-89ab-cdef-0123-456789abcdef"
```

Provide standard timestamps

Provide `created_at` and `updated_at` timestamps for resources by default, e.g:

```
{
  // ...
  "created_at": "2012-01-01T12:00:00Z",
  "updated_at": "2012-01-01T13:00:00Z",
  // ...
}
```

These timestamps may not make sense for some resources, in which case they can be omitted.

Provide standard response types

This document describes the acceptable values for each of JSON's basic data types.

String

- Acceptable values:
  - string
  - `null`

e.g:

```
[
  {
    "description": "very descriptive description."
  },
  {
    "description": null
  },
]
```

Boolean

- Acceptable values:
  - true
  - false

e.g:

```
[
  {
    "provisioned_licensese": true
  },
  {
    "provisioned_licenses": false
  },
]
```

Number

- Acceptable values:
  - number
  - `null`

Note: some JSON parsers will return numbers with a precision of over 15 decimal places as strings. If you need precision greater than 15 decimals, always return a string for that value. If not, convert those strings to numbers so that consumers of the API always know what value type to expect.

e.g:

```
[
  {
    "average": 27.123
  },
  {
    "average": 12.123456789012
  },
]
```

Array

- Acceptable values:
  - array

Note: Return an empty array rather than `NULL` when there are no values in the array.

e.g:

```
[
  {
    "child_ids": [1, 2, 3, 4],
  },
  {
    "child_ids": [],
  }
]
```

Object

- Acceptable values:
  - object
  - null

e.g:

```
[
  {
    "name": "service-production",
    "owner": {
      "id": "5d8201b0..."
     }
  },
  {
    "name": "service-staging",
    "owner": null
  }
]
```

Use UTC times formatted in ISO8601

Accept and return times in UTC only. Render times in ISO8601 format, e.g.:

```
"finished_at": "2012-01-01T12:00:00Z"
```

Nest foreign key relations

Serialize foreign key references with a nested object, e.g.:

```
{
  "name": "service-production",
  "owner": {
    "id": "5d8201b0..."
  },
  // ...
}
```

Instead of e.g.:

```
{
  "name": "service-production",
  "owner_id": "5d8201b0...",
  // ...
}
```

This approach makes it possible to inline more information about the related resource without having to change the structure of the response or introduce more top-level response fields, e.g.:

```
{
  "name": "service-production",
  "owner": {
    "id": "5d8201b0...",
    "email": "alice@heroku.com"
  },
  // ...
}
```

When nesting foreign key relations, use either the full record or just the foreign keys. Providing a subset of fields can lead to surprises and confusion, makes inconsistencies between different actions and endpoints more likely.

To avoid inconsistency and confusion, serialize either:

- **foreign keys** only - values the full record can be looked up with, like `id`, `slug`, `email`.
- **full record**, all fields (this would be an "embedded record")

Generate structured errors

Generate consistent, structured response bodies on errors. Include a machine-readable error `id`, a human-readable error `message`, and optionally a `url` pointing the client to further information about the error and how to resolve it, e.g.:

```
HTTP/1.1 429 Too Many Requests
```

```
{
  "id":      "rate_limit",
  "message": "Account reached its API rate limit.",
  "url":     "https://docs.service.com/rate-limits"
}
```

Document your error format and the possible error `id`s that clients may encounter.

Show rate limit status

Rate limit requests from clients to protect the health of the service and maintain high service quality for other clients. You can use a token bucket algorithm to quantify request limits.

Return the remaining number of request tokens with each request in the `RateLimit-Remaining` response header.

Keep JSON minified in all responses

Extra whitespace adds needless response size to requests, and many clients for human consumption will automatically "prettify" JSON output. It is best to keep JSON responses minified e.g.:

```
{"beta":false,"email":"alice@heroku.com","id":"01234567-89ab-cdef-0123-456789abcdef","last_login":"2012-01-01T12:00:0
0Z","created_at":"2012-01-01T12:00:00Z","updated_at":"2012-01-01T12:00:00Z"}
```

Instead of e.g.:

```
{
  "beta": false,
  "email": "alice@heroku.com",
  "id": "01234567-89ab-cdef-0123-456789abcdef",
  "last_login": "2012-01-01T12:00:00Z",
  "created_at": "2012-01-01T12:00:00Z",
  "updated_at": "2012-01-01T12:00:00Z"
}
```

You may consider optionally providing a way for clients to retrieve more verbose response, either via a query parameter (e.g. `?pretty=true`) or via an `Accept` header param (e.g. `Accept: application/vnd.heroku+json; version=3; indent=4;`).

Artifacts

The Artifacts section describes the physical objects we use to manage and discuss API designs and patterns.

Provide machine-readable JSON schema

Provide a machine-readable schema to exactly specify your API. Use prmd to manage your schema, and ensure it validates with `prmd verify`.

Provide human-readable docs

Provide human-readable documentation that client developers can use to understand your API.

If you create a schema with prmd as described above, you can easily generate Markdown docs for all endpoints with `prmd doc`.

In addition to endpoint details, provide an API overview with information about:

- Authentication, including acquiring and using authentication tokens.
- API stability and versioning, including how to select the desired API version.
- Common request and response headers.
- Error serialization format.
- Examples of using the API with clients in different languages.

Provide executable examples

Provide executable examples that users can type directly into their terminals to see working API calls. To the greatest extent possible, these examples should be usable verbatim, to minimize the amount of work a user needs to do to try the API, e.g.:

```
$ export TOKEN=... # acquire from dashboard
$ curl -is https://$TOKEN@service.com/users
```

If you use prmd to generate Markdown docs, you will get examples for each endpoint for free.

Describe stability

Describe the stability of your API or its various endpoints according to its maturity and stability, e.g. with prototype/development/production flags.

See the Heroku API compatibility policy for a possible stability and change management approach.

Once your API is declared production-ready and stable, do not make backwards incompatible changes within that API version. If you need to make backwards-incompatible changes, create a new API with an incremented version number.