BASIS API GUIDELINES (DRAFT)

Version:

- **0.1.1** - January 2015 - Draft

Authors:

- Felix Fennell (BASIS)

Status:

- **These guidelines have not yet been adopted**. When they are, they will automatically become version 1.0.0.
- **Adopted versions** of these guidelines will be tracked in the **master** branch of the API Guidelines repository, tagged by version.
- **Draft versions** of these guidelines will be tracked in the **develop** branch of the API Guidelines repository.

Contents

Conventions & Terms

- **MUST**, **MUST NOT**, **SHOULD**, **SHOULD NOT** - As defined by RFC 2119
- BASIS - British Antarctic Survey Information Services

Audience

These guidelines are aimed at API authors or those working closely with them.

You do not need to know these guidelines to consume an API based upon them. The documentation for the respective API **SHOULD** contain any information you require.

These guidelines are written for a technical audience and use technical terms where appropriate. The *Conventions & Terms* section will document any terms that may not be generally known by such an audience, or where a term is used to mean something other than its common definition.

Feedback

Feedback and discussion is strongly encouraged, either formally through revisions [1] to this document, or informally by contacting any of the authors.

[1] Via forking the API Guidelines repository and submitting a pull-request with changes.

Acknowledgements

As noted later, **BAS APIs are not significantly unique**. Therefore large parts of relevant guidelines produced by others, will be directly applicable to us. Consequently, large parts of these guidelines are based on other peoples work, namely:

- Government Digital Service - Service Manual
- 18F - 18F API Standards [1]
- Interagent - HTTP API Design [2]

[1] 18F are the US equivalent of the GDS.
[2] AKA Heroku

Aims & Objectives

The aims of these guidelines are:

1. To ensure APIs created by BASIS follow best practice in a way that makes them easy to maintain and use.
2. To ensure, where appropriate, APIs created by BASIS are of a consistent design with one another, so as to be able to share a similar end user experience, and allow multiple APIs to be maintained effectively.

These aims will be achieved through these objectives:

1. Provide a series of guidelines focusing on the *design* and *behaviour* of APIs [1].
2. Promote the use of standards and best practice and how these create better APIs.
3. Ensure technologies and approaches used to create APIs are justified [2].
4. Provide, real world, non-abstract resources and implementations which creators can use to provide functionality that meet these guidelines [3].
5. In some cases, make opinionated decisions, with explanations why these have been made.
6. Ensure any relevant requirements that apply to BASIS through BAS, NERC, Government, etc. are followed.
7. Ensure these guidelines are updated in a timely fashion to ensure guidance remains relevant and APIs remain useful.

[1] These guidelines provide one way of doing things, this will not be the **only** way, or the **best** way depending on your requirements.

[2] These guidelines provide criteria on how choices in technology **SHOULD** be made and how these **SHOULD** be justified, but do not specify what languages, frameworks, techniques and methodologies **SHOULD** be used.

[3] These are not required implementations, where others are more suitable they **SHOULD** be used.

Scope

- These guidelines apply to APIs created by members of BAS Information Services (BASIS) only [1].
- These guidelines apply to both **new** and **existing** APIs [2].
- These guidelines apply to both **internal** and **external** APIs [3].
- These guidelines apply to HTTP based 'RESTful' APIs only. Web and data services in general are out of scope [4].
- The specific technology stack used within an API is out of scope.

[1] This does not mean that other groups within BAS, NERC or the wider Community cannot use, build on or constructively criticise these guidelines. This is simply to indicate that these guidelines make no claims to apply to these other groups and their APIs.

[2] This may require existing APIs to be modified and therefore reasonable provision must be given to allow these APIs to come into compliance. APIs where this cannot be achieved are still in scope of these guidelines, but **MUST** choose not to implement them.

[3] Internal in this context refers to APIs used solely within BASIS, BAS or NERC.

[4] Whilst these guidelines deal with APIs specifically other guidelines may apply to web and data services more generally. These guidelines would effectively become additional guidelines that apply in a wider context.

Requirements

- APIs implementing these guidelines **MUST** indicate the version used.
- APIs **MUST** state whether they follow or no not follow these guidelines in their documentation.
- APIs that do not implement these guidelines **MUST** be aware of the potential repercussions of doing so, such as security and interoperability issues, etc.

General Principles

The guidelines in this section are not objectively measurable. However they summarise the spirit and intention of these guidelines as a whole. Later sections and guidelines give targeted, measurable, guidance on specific aspects or issues. Each relates back to one or more of these general guidelines.

[1.1] APIs **SHOULD** be consistent but not uniform, using the most appropriate solution for each need

Consistency, within these guidelines, refers to providing a similar end user experience, as well as the *stability* (amount of change) and *sustainability* (long term maintenance and availability) of APIs.

This is the ninth GDS Design Principle, echoed here for convenience.

> "Wherever possible we should use the same language and the same design patterns -- this helps people get familiar with our services. But, when this isn't possible, we should make sure our underlying approach is consistent. So our users will have a reasonable chance of guessing what they're supposed to do.
>
> This isn't a straitjacket or a rule book. We can't build great services by rote. We can't imagine every scenario and write rules for it. Every circumstance is different and should be addressed on its own terms. What unites things, therefore, should be a consistent approach -- one that users will hopefully come to understand and trust -- even as we move into new digital spaces."

Where appropriate, APIs **SHOULD** use consistent terminology and meaning, including field names, methods, error formats, etc. Where appropriate, these **SHOULD** be commonly approaches used elsewhere.

APIs will not all serve the same purpose. However, there are a number of common types that **SHOULD**, where appropriate, be consistent with one another. Some examples are listed in the 18F API Standards, echoed here for convenience:

> "*Bulk data*. Clients often wish to establish their own copy of the API's dataset in its entirety. For example, someone might like to build their own search engine on top of the dataset, using different parameters and technology than the "official" API allows. If the API can't easily act as a bulk data provider, provide a separate mechanism for acquiring the backing dataset in bulk.
>
> *Staying up to date*. Especially for large datasets, clients may want to keep their dataset up to date without downloading the data set after every change. If this is a use case for the API, prioritise it in the design.

> *Driving expensive actions*. What would happen if a client wanted to automatically send text messages to thousands of people or light up the side of a skyscraper every time a new record appears? Consider whether the API's records will always be in a reliable unchanging order, and whether they tend to appear in clumps or in a steady stream. Generally speaking, consider the "entropy" an API client would experience."

Sources:

- Ninth Principle - GDS Design Principles
- APIs - Names Reinforce Conventions - GDS Service Manual
- Naming Principles - Microformats
- 18F API Standards - Design For Common Use Cases - I8F

[1.2] APIs **SHOULD** be as simple as possible

Simple APIs, and code, are easier to understand than complex APIs and code. Greater simplicity increases:

- The maintainability of APIs and, through 'fewer moving parts', reduces the opportunities for bugs to hide
- The speed of development and documentation, both during rapid iteration and future updates/maintenance
- The speed at which end users can understand and start using our APIs and, where alternatives exist, the chances of using our APIs.

APIs **SHOULD NOT** do more than they need to. If someone else already provides an appropriate service we **SHOULD** use or recommend this, we **SHOULD NOT** duplicate it ourselves.

Doing this keeps our API as simple as possible and lets us focus on making what we do need to provide as useful as possible. We should let others concentrate on solving other needs.

For example, an API for providing the locations of BAS ships could include a current weather report from a third party. Since this weather information can be requested directly from the third party, rather than bundled by us, it **SHOULD NOT** be included in our API. Instead information on where to get this weather data from, and preferably an example of how to get it, **SHOULD** be provided in the our API's documentation.

This is essentially the UNIX philosophy i.e. 'do one thing and do it well', and the second GDS Design Principle.

Sources:

- UNIX philosophy - Wikipedia
- Keep It Simple Stupid (KISS) Principle - Wikipedia
- Second Principle - GDS Design Principles
- APIs - Consuming and Using APIs - GDS Design Principles

[1.3] APIs **SHOULD** be clear and explicit in their operation, returning helpful errors

APIs **SHOULD NOT** try to "guess" or make assumptions about what what the user wants. If a request doesn't make sense, or is unclear, APIs **SHOULD** say so by returning an error/warning etc.

"Guessing" introduces unnecessary complexity, increasing the chances of bugs. Instead APIs **SHOULD** be "helpful".

For example, an API method takes a start date as a filter for some dataset, the method expects a date in the form YYYY-`MM-DD`. A request is made using a date `2004/10/14`.

This request **SHOULD** fail, stating the start date given is not recognised as valid.

It is clear this is a date, however in the wrong format. Whilst it is possible (and likely trivial) to detect this and convert the date into the right format, the API **SHOULD NOT** do so.

In most cases, it would be necessary to add code to check for this. If this code is refactored later and there are not adequate tests (manual or automatic) in place this request will now fail when previously it succeeded.

In this case, the delimiter, `/` used is common and would typically be included in any "guessing" code. However if a `;` was used instead, and not coded for, that request would fail. This would mean the two requests, which contain the same fault, would be treated differently.

This will likely have an impact on any service using the API and cause confusion (i.e. not be clear and unambiguous) Debugging, and more code changes will then be needed to fix the error.

Instead, a simpler `if date == format` approach **SHOULD** be used. If the statement fails, for any reason, an error **SHOULD** be returned. This error **SHOULD** explain:

- Why the request failed, possibly with links to documentation (the date wasn't valid)
- The syntax that was expected (YYYY-`MM-DD`, e.g. `2004-07-23`)
- The value used in the request ('2004/10/14')

This shows, clearly and unambiguously, why the request failed and how to correct the error. Importantly, regardless of the input used, this method will, from first use, it will only succeed where the date is correctly formatted.

In addition, an appropriate status code **SHOULD** be used to indicate, more generally, the type of error that has occurred. Errors caused by us (the API) **SHOULD** use a status code in the 5XX range, user errors **SHOULD** use a status code in the 4XX range.

Sources:

- APIs Explicitly Set Expectations - GDS Service Manual
- 18F API Standards - Error Handling - I8F

[1.4] APIs SHOULD make things as easy as possible, whilst remaining clear and consistent

If there is something we can do to make things easier for the end user we **SHOULD** do so, providing this is clearly explained, either in the response or in documentation.

This is Principle 4 of the GDS Design Principles, echoed here for convenience.

> "Making something look simple is easy; making something simple to use is much harder -- especially when the underlying systems are complex -- but that's what we should be doing.
>
> With great power comes great responsibility -- very often people have no choice but to use our services. If we don't work hard to make them simple and usable we're abusing that power, and wasting people's time."

Sources:

- Fourth Principle - GDS Design Principles

[1.5] APIs SHOULD be as resilient as possible

It is infeasible to provide all services ourselves, credit card processing for example **SHOULD** be left to a payment gateway, however we **SHOULD** create APIs in such a way that reliance on a particular service is avoided through abstraction. This ensures we can swap out services, potentially for newer, better ones, without having to change core logic.

Where we create APIs that rely on third parties, we **MUST** ensure we have plans in place for when these services are either temporally or permanently unavailable in the future.

When evaluating whether to use an external service factors such as the reputation of the service, the cost (and if this likely to change), availability (can the service scale to the same degree we can) and support all need to considered. This applies both to services provided internally (i.e. by ourselves, BAS, NERC) or by third parties.

These factors need to weighed against the importance of the functionality such services provide. For services providing core functionality, a more defensive, conservative approach **SHOULD** be followed.

Sources:

- APIs - Service Agreements and Resilience - GDS Service Manual

[1.6] Where appropriate, APIs **SHOULD** use best practice for solving common problems

Our APIs **SHOULD NOT** "reinvent the wheel". The types of API BAS will commonly create will not be unique. They therefore don't require unique functionality or solutions to common problems.

Using a common solution can:

- Utilise prior knowledge of developers or of the community, reducing implementation and support costs
- Utilise existing tooling and frameworks, increasing interoperability and reducing development time
- Utilise prior knowledge of end users decreasing the effort to learn and start using our APIs

For example, content negotiation for determining response data types is a solved problem. Rolling our solution would take significant time, not cover all edge cases and would require all support to be provided by us. Using the *common* solution means we benefit from:

- A proven technique with millions of hours of use in production environments [1]
- The experience of others in the form of blog posts on optimal implementations, faster development cycles and security exploit detection/patching through larger pools of developer effort and support for common problems (e.g. Stack Overflow etc.)
- Support within other tools and services allowing our APIs to be more easily consumed, including by end users

A distinction should be made between *best* practice, which **SHOULD** be followed, and *common* practice, which may not be "a good idea". Examples of *common* practice which are justifiable to avoid include:

- Solutions which break or misuse standards
- Solutions which not sufficiently secure, in situations where a more secure alternative exists, or, for sensitive information, where any solution exists.
- Solutions which are "hacks" or "lazy" approaches, where a "proper" approach is available, or that rely on "quirks" or "side effects" to function.

These approaches, especially the latter "quirk" kind, are typically not sustainable as their behaviour could change at any time, often without warning. The results of this could, at best, break the approach used, or at worst, alter the result of the approach. This could unexpectedly introduce new security flaws or expose information presumed to be safe.

A good example of a *common* practice solution is JSONP, which is essentially a hack compared to the *best* practice solution, CORS.

[1] More likely billions or trillions of man hours.

Sources:

- HTTP API Design - Generate Structured Errors - Interagent

[1.7] Where appropriate, APIs **SHOULD** be open, and made available publicly

Given that BAS is a Government Organisation our work **SHOULD** be visible and accessible by as wider a audience as possible. Providing APIs aids this by offering data in a, hopefully, well organised, logical, way in convenient formats, and ideally without the involvement of BAS Staff [1].

In addition to any outputs our API may provide, the research, designs and source code of APIs themselves may prove useful to others. We benefit enormously from the work of others, through open sourced ideas and technologies, it is therefore right we "pay back" by making whatever we can available to others.

Source code **SHOULD** be listed within the BAS (Antarctica) organisation account on GitHub. This provides consistency for end users to easily find all projects released by BAS Staff. If another location is more suitable (i.e. domain or project specific) this **SHOULD** be used. Projects **SHOULD NOT** be split or mirrored across multiple locations as this adds confusion and may lead to one version falling out of sync.

In addition, and where appropriate, issue trackers, wiki's and other sources of information associated with APIs **SHOULD** be made available under reasonable restrictions [2]. This gives end users an increased understanding of the wider context of an APIs which may aid in their evaluation of our APIs.

This is the tenth GDS Design Principle.

[1] Which is a benefit for both parties.

[2] This may include:

- Read-only access
- Redaction of information that could place services at risk of attack or misuse
- Redaction of any personally identifying information for privacy reasons
- Redaction of information for ethical, professional, commercial or regulatory reasons (such as data that may be misleading)

Sources:

- Tenth Principle - GDS Design Principles
- APIs - Be Public By Default - GDS Service Manual
- 18F API Standards - Point of Contact - I8F

Resources:

- GitHub
- BitBucket
- Antarctica Organisation - Github

[1.8] Where appropriate, APIs **SHOULD** use established standards and open formats

Good standards encourage interoperability through consistency between different systems that implement the same standard.

Different standards target different *layers*, technologies or designs used within an API. Conformance to appropriate standards, at whatever level, can be used to justify technology choices for example.

Following standards, or using standards compliment technologies, typically bring benefits such as community/hive knowledge (e.g. StackOverflow etc.) and compatibility with tooling, saving time and generally giving a better experience than non-standard alternatives.

The HTTP Protocol is a good example of a standard, using it means we can use a variety of tools (web browsers, caching layers, load balancers etc.) and frameworks provided by others with a high confidence these will understand each other. End users will also typically understand, if not expect, the use of this standard.

Another good technology specific, example, is the PSR-4 Autoloader standard for automatically loading PHP classes. Using this standard reduces the effort to use unfamiliar technologies, encouraging experimentation and rapid iteration. Using a common standard encourages the creation of associated tooling, such as package managers and IDE/editor support.

Data formats **SHOULD** be chosen based on there suitability for the data being represented and their interoperability.

In general, data formats **SHOULD** be as open as possible. Similar to technologies, proprietary data formats can become obsolete or expensive to maintain, limiting their long term availability.

Sources:

- APIs - Just Use the Web - GDS Service Manual
- APIs - Choosing Appropriate Formats - GDS Service Manual

[1.9] Where appropriate, APIs **SHOULD** favour proven and open source alternatives over experimental, or proprietary, alternatives

An experimental or immature technology may be discontinued. If resources are not available to reimplement the features that technology provided, the API may be degraded or cease to function. Similarly, if proprietary technology is used and is no longer supported or open sourced, it will become obsolete and incompatible with newer technologies.

Ongoing, variable licensing costs may force the removal of a proprietary technology. If resources are not available to reimplement the features that technology provided, the API may be degraded or cease to function.

Each time a technology is changed it may introduce new dependencies and constraints, such as needing to remove a feature that the new technology cannot provide. Such changes impact on the consistency of the API and may introduce new bugs.

Sources:

- Choosing Technology - GDS Service Manual
- Technology Code of Practice - GDS Service Manual

Need Driven

[2.1] APIs **MUST** meet the needs of their users

Ideally all stakeholders **SHOULD** be involved in the development of APIs, including:

- Data providers
- API creators [1]
- Project mangers [1]
- Significant end users
- Representative end users.

For simple APIs, this may not be many people and simple to organise. Others may require more extensive research and engagement with many users, both internal and external to BAS, and require negotiation to resolve problems and constraints.

Testing is important to ensure APIs we create meet the needs of end users. Logging and monitoring **SHOULD** be used to track this over time, and clear mechanisms for end users to submit feedback **SHOULD** be documented.

Feedback and retrospectives **SHOULD** be used to ensure that we don't repeat mistakes in the future, and support and emphasise things that worked well.

This is the first of the GDS Design Principles.

[1] This is likely to be the same person.

Sources:

- First Principle - GDS Design Principles
- APIs - Testing - GDS Service Manual

[2.2] APIs **MUST** allow user feedback, and this **MUST** be taken seriously

Ideally the individual responsible for each API **SHOULD** be contactable directly.

Feedback **MUST** be treated seriously, particularly in regards to usability problems.

Examples of this type of feedback include:

- A user has made a reasonable attempt to achieve a task the API claims to support, using provided documentation/examples and limited experimentation.
- A user reports that something doesn't work as expected or is unclear about how something should be used.

Sources:

- 18F API Standards - Point of Contact - I8F

[2.3] APIs **SHOULD NOT** be created in isolation, and we **SHOULD** use them ourselves

API creators **SHOULD** also be API end users. This is the quickest and easiest way to validate our APIs do what they say they should and are easy to use.

API creators **SHOULD** take part in any discussions on the wider context of APIs. This reinforces *why* the API is needed and helps ensures these needs are met.

For example if data processing is performed in other systems that feed APIs, API creators **SHOULD** be able to feed in their requirements such as attributes or statistics that may not be needed in these other systems but would be useful in the API.

Sources:

- APIs - Build an API By Building With the API
- 18F API Standards - Using One's Own API - I8F

API Development

[3.1] APIs **SHOULD** have a single owner responsible for its delivery and support

This person **SHOULD**:

- Be reasonable for making sure the API follows these guidelines
- Be involved in the day to day creation and running of the API, they **SHOULD NOT** be a symbolic figurehead
- Be responsible for the allocation of resources available to that API
- Be contactable by users of the API, and **SHOULD** personally review and act on any feedback provided
- Be accountable for the API including its design decisions and ease of use to its end users and other stake holders

Sources:

- 18F API Standards - Point of Contact - I8F

[3.2] APIs **SHOULD** be loosely coupled

This means the use of abstraction to separate logic or functionality from specific implementations of technologies or services and data sources.

This ensures the APIs we create are modular and can adapt to change. This applies at multiple levels, from whole services to specific functionality and requires considered, appropriate, design to ensure clear separation of concerns and approaches such as SOLID.

Sources:

- APIs - Code Integration - GDS Service Manual
- Seperation of Concerns - WikiPedia
- Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion (SOLID)

[3.3] APIs **MUST** use source control

Git **SHOULD** be used. For internal APIs the NERC Stash repository, provided by CEH, is recommended. For collaborative efforts or where access from outside the NERC firewall is needed, multiple public offerings exist.

If a project hosted on Stash is to be open sourced, it **SHOULD** be listed within the BAS (Antarctica) organisation account on GitHub. This provides consistency for end users to easily find all projects released by BAS Staff. If another location is more suitable (i.e. domain or project specific) this **SHOULD** be used.

APIs **SHOULD** adopt a suitable branching model for separating development and production codebases, and for using tools such as continuous integration and processes such as code reviews.

By convention, a *develop* branch should reflect the version of the API under active development, and a *master* branch reflect the latest stable version.

Wherever possible, API source code **SHOULD** be made publicly available. This increases the number of people who can review our code for bugs and mistakes and maybe offer suggestions on how to do things better. It also forces us to write better code, by avoiding "hacks", using proper structuring and ensuring code is readable and well explained.

Sources:

- Why Should I use Source Control - Stack Overflow

Resources:

- Stash - CEH
- GitHub
- Bitbucket
- Antarctica Organisation - Github
- Comparing Workflows - Atlassian Tutorials

[3.4] APIs **SHOULD** use issue tracking

These guidelines have no preference on how issues are managed, or the system used to do so. The type of issue tracker you choose will depend on the "size" of the API [1].

For small projects released to the public, issue tracking provided by GitHub or BitBucket offer good accessibility for users and contributors and are free to use. However they lack more complex features such as custom issue states, workflows and integration with other systems.

These tools **MUST NOT** be used to discuss details that could place BAS or NERC infrastructure at risk or disclose personally identifying or commercial information, or for regulatory reasons where these apply.

For larger projects or those focused on an internal or NERC audience, you **SHOULD** use the NERC Jira installation, provided by CEH.

[1] The "size" of an API is largely subjective but these factors can be used for a rough estimation:

- The size of an APIs audience
- The number and range of people contributing to its development
- The complexity of the issues that will need to be handled.

Sources:

- Why Use Bug Tracking Software - Stack Overflow

Resources:

- Comparison of Issue Tracking Software - Wikipedia
- Github Issues 2.0 - Github Blog
- Jira - CEH
- Stash - CEH

[3.5] APIs **SHOULD** be robustly tested

This takes a number of forms:

- Automated testing using linters, test suites & continuous integration, mocked services etc.
- Human testing through passive and active feedback during alpha/beta/live stages of development

Sources:

- APIs - Testing - GDS Service Manual

API Design

[4.1] APIs **SHOULD** use `UTF-8` for character encoding

The 18F API Standards summarise the benefits of using this, echoed here for convenience:

> "Expect accented characters or 'smart quotes' in API output, even if they're not expected. An API should tell clients to expect UTF-8 by including a charset notation in the Content-Type header for responses. An API that returns JSON should use: `Content-Type: application/json; charset=utf-8`"

Sources:

- 18F API Standards - Just Use UTF-8 - 18F

Resources:

- UTF-8 Everywhere

[4.2] APIs **SHOULD** use `ISO 8601` and `UTC` for dates and times

The 18F API Standards summarise the benefits of using this, echoed here for convenience:

> "For just dates, that looks like 2013-02-27. For full times, that's of the form 2013-02-27T10:00:00Z. This date format is used all over the web, and puts each field in consistent order -- from least granular to most granular."

Sources:

- 18F API Standards - Use A Consistent Date Format - I8F
- HTTP API Design - Use UTC Times Formatted in ISO8601 - Interagent

[4.3] APIs **SHOULD** use unique IDs for resources

Each resource **SHOULD** be identified by a unique identifier, UUID's **SHOULD** be used [1]. This ID **SHOULD** be unique across all instances of an API and ideally other APIs as well.

[1] Version 4 or 5 is recommended, represented in lower case.

Sources:

- HTTP API Design - Provide Resource (UU)IDs - Interagent
- Additional information on why UUIDs are preferable - StackOverflow

[4.4] Where Appropriate, resources **SHOULD** use `created_at` and `updated_at` timestamps

A "created at" timestamp **SHOULD** be set when the object is first created and **MUST NOT** change. An "updated at" timestamp **SHOULD** change whenever a resource is modified.

There may be resources where using these attributes doesn't make sense, in such cases they **SHOULD NOT** be used, as this may cause confusion.

Sources:

- HTTP API Design - Provide Standard Timestamps - Interagent

Documentation

[5.1] APIs **MUST** provide end user documentation

It isn't feasible for these guidelines to state exactly how all APIs **SHOULD** be documented as this will depend on its purpose, intended audience and other factors. However in general APIs **MUST** be well documented so it is clear to users what the API can do and how they should use it to achieve what they want.

The standard of documentation for both internal and external APIs **SHOULD** be the same, i.e. you **SHOULD** assume an end user has never used any BAS API before. For internal APIs, it is reasonable to assume basic familiarity with the BAS domain.

That said, it is possible to identify some key requirements for API documentation:

- It **MUST** be clear, concise and well organised
- It **MUST** cater to two key audiences, new users, looking to evaluate the API, and existing users, looking for specific information on how to use the API

How you decide to write and structure your documentation is up to you. However, as an aid to ensure that common elements are not forgotten, a list of suggested sections/topics is given here.

You do not have to include all of these items, and you **SHOULD NOT** include any that don't make sense for an API, as this may cause confusion. These suggestions are not exhaustive and do not take into account the context or specific nature of any single API. Therefore you will need to add other sections as needed to document unique, API specific information. This includes an APIs resources and endpoints which are naturally API specific.

It is suggested to divide an APIs documentation into two parts:

1. Overview information, aimed primarily at new/evaluating users.
2. Endpoint information, aimed primarily at existing users.

Overview information **SHOULD** include:

- The version of these guidelines used
- The status of the API (such as Alpha, Beta, Live, etc.)
- API stability, versioning and deprecation policy, including how to select the desired API version
- Changelog highlighting changes to end user aspects of the API divided into major versions
- Roadmap highlighting upcoming changes to end user aspects of the API divided into non-breaking and breaking changes to allow for end users to plan accurately
- Project management, including details of the API Maintainer and how to submit feedback
- Mailing list - i.e. some way to stay up to date with any changes

- Contribution policy
- Data sources, accuracy and limitations
- Security and authentication/authorisation, including acquiring and using authentication tokens
- Rate limiting
- Common request and response headers
- Response serialisation format, including errors, warnings, notices, etc.
- Real world examples
- Copyright and licensing

Endpoint information **SHOULD** include, for each endpoint:

- The purpose of the endpoint
- Deprecation status
- How to access the endpoint including any required authentication/authorisation and/or restrictions
- Details of supported data-types
- Details of request parameters and any supported options, including value range limitations or lists of acceptable values
- Sample/example response headers and output

Sources:

- APIs - Explicitly Set Expectations - GDS Service Manual
- APIs - Practice Service Evolution - GDS Service Manual
- Security - Atlassian REST API Design Guidelines (V1)
- 18F API Standards - Notifications of Updates - I8F
- HTTP API Design - Provide Human Readable Docs - Interagent
- HTTP API Design - Describe Stability - Interagent

Resources:

- Good Examples
  - Stripe
  - New York Times
  - Twilio
- Templates
  - Changelogs

[5.2] APIs **SHOULD** provide real world examples

In addition to through and accurate documentation, examples of how to make requests for different scenarios and responses which would be returned **SHOULD** be provided.

This aids users in evaluating if the API will provide the information they are looking for, more so than verbose descriptions of responses.

When implementing an API end users can use examples to get started quickly. Therefore examples **SHOULD** be functional (i.e. use real endpoints and resources) using safe, sensible defaults for real world, typical, requests.

Documentation **SHOULD** clarify any extreme values or lists of acceptable values, these **SHOULD NOT** be defined within examples.

Examples **SHOULD** be useable with a minimum of effort, ideally from the browser or the command line. Provision for features such as authentication/authorisation **SHOULD** be made, such as:

- Using public methods (where available)
- Using test credentials which may return fake, but realistic, data, that is sufficiently similar to real data to be useful, for example, fake ship positions

Complex APIs, where requests cannot be tested within a browser, **SHOULD** still offer some way to experiment. Commonly this is done using an API "explorer" or "playground". In addition to self hosted solutions, there are a number of hosted options to provide this functionality, though these are not all free.

Sources:

- APIs Document by discovery... and Example - GDS Service Manual
- HTTP API Design - Provide Executable Examples - Interagent

Resources:

- Examples/Generators:
  - Self hosted:
    - Swagger
    - Slate
  - SASS:
    - Apiary
    - Mashape
  - Bespoke/Custom:
    - Facebook API Tools
    - Guardian API Explorer
    - Google Search for API Explorer

[5.3] APIs **SHOULD** provide developer documentation

Developer documentation **SHOULD** contain a description of the technology stack used by an API and how this can be provisioned to allow new instances of the API to be created, either for further development or deployment [1]

[1] This is crucial for new developers, especially where non-standard or custom technologies are used.

[5.4] APIs **SHOULD** be backed by a schema

This ensures all elements of an API are defined in a structured, testable, way and can be validated either by hand or using automatic tools.

Sources:

- HTTP API Design - Provide Machine Readable JSON Schema - Interagent

Resources:

- Understanding JSON Schema - Michael Droettboom, Space Telescope Science Institute

Versioning

[6.1] APIs **MUST** use versioning using whole integers

New API versions **SHOULD** replace older versions, therefore older versions **SHOULD** be deprecated.

[6.2] API versions **SHOULD** map to major releases

API versions are not alternatives, they are linear progressions, hopefully improving over time.

[6.3] The API version **MUST** be the first token in a URL path, prefixed by a "v"

For example `https://api.bas.ac.uk/v2/ships`.

Sources:

- Version Control For APIs - Atlassian REST API Design Guidelines (V1)

[6.4] APIs **MUST NOT** assume a version for requests that do not specify one

APIs **MUST NOT** use conventions such as `/latest/`, `/current/`, `/head/` etc. as pointers to the latest API version as this behaviour is not clear or unambiguous. Requests made in this way may suddenly fail if a breaking change is introduced in a new version.

Instead, users **MUST** explicitly state the version of the API they wish to use. If no version is provided, or the version is not valid, the API **MUST** return a fatal error a valid version is provided.

[6.5] APIs **MUST** preserve backwards compatibility within a version

APIs **MUST NOT** introduce breaking changes *within* an API version. If you need to do this, create a new version.

Examples of changes that can be implemented within the current version:

- New resources (with associated methods)
- New methods for existing resources
- Support for new data formats
- New properties or attributes for existing resources

Examples of changes that **MUST** be implemented in a new version:

- Removed or renamed methods/resources/endpoints, including changes to the HTTP verb used
- Changes to the data returned by a method or the structure of this data

Sources:

- APIs - Practice Service Evolution - GDS Service Manual
- Version Control For APIs - Atlassian REST API Design Guidelines (V1)
- 18F API Standards - I8F

[6.6] APIs **SHOULD** try to be compatible with older versions, providing they remain clear and unambiguous

For example, an API endpoint is moved from `/ships` to `/fleet/ships` to be better organised. No other aspects of the endpoint are changed.

This represents a breaking change, as requests made to `/ships` will no longer return what they did previously. In this case a request made to the earlier `/ships` and newer `/fleet/ships` will be identical (other than the URL used to access the endpoint obviously). Backwards compatibility can therefore be preserved by using a `301 Moved Permanently` header to redirect requests made to the old method to the new.

This remains clear and unambiguous by informing the user, "this has moved to here", the user can then update their requests in their own time.

Sources:

- APIs - Practice Service Evolution - GDS Service Manual

[6.7] Where appropriate, APIs **SHOULD** support old versions

It is not reasonable to assume all users will upgrade to newer API versions at the same time. Therefore old API versions **SHOULD** continue to be supported, for a limited period, to allow users to upgrade in their own time.

The length of this limited period will depend on the audience of the API. For internal APIs the period can be quite short, for external APIs a longer period **SHOULD** be given. The number of users an API has **SHOULD** be taken into account, which **SHOULD** be judged from API analytics and logs.

Wherever possible users **SHOULD** be notified of new API versions in advance, increasing the likelihood of a quicker adoption by allowing testing before upgrading. Additionally, requests made to deprecated API versions **SHOULD** inform the user that API version is deprecated and how they can use the new version.

APIs **SHOULD** clearly state their deprecation policy within their documentation, this allows users to evaluate if they can commit to this schedule before committing to the API.

The number of API versions that **SHOULD** be supported will depend on the same factors as above, plus the rate at which new versions are released. Ultimately these are judgement calls with internal resources for supporting older versions usually being the limiting (and deciding) factor.

A deprecated API version **SHOULD** remain accessible and bugs and faults **SHOULD** be addressed (as these will likely exist in newer versions as well). Security issues **MUST** be addressed, if this is not possible the older version **MUST** be withdrawn.

Improvements and new features, regardless of whether they maintain compatibility within a version, **SHOULD NOT** be supported in deprecated versions.

Sources:

- APIs - Practice Service Evolution - GDS Service Manual

[6.8] APIs **SHOULD** be forward compatible, providing they remain clear and unambiguous

APIs **SHOULD** be designed to be robust and forward thinking, given the speed of evolution on the web.

For example, if a new header is added to later API versions, older API versions **SHOULD** ignore it, rather than fail. Documentation **SHOULD** emphasise where different versions may give different responses for the same request, however these situations **SHOULD** be avoided wherever possible.

Sources:

- APIs - Practice Service Evolution - GDS Service Manual

Security

[7.1] APIs **MUST NOT** rely on "security through obscurity"

It is a dangerous fallacy and **MUST NOT** be relied upon.

Sources:

- Security Through Obscurity - Wikipedia
- Why is Security Through Obscurity a Bad Idea - Stack Overflow

[7.2] APIs **MUST NOT** use non-standard cryptography

Cryptography is hard and we **SHOULD NOT** design it ourselves. Cryptography, encryption and related technologies are all excellent examples of when it's better to use something by someone else, or more specifically by someone that knows what they're doing. Even *implementing* cryptography is difficult, as a quick internet search will prove.

It would insane for us to attempt to "roll our own" cryptography or cryptography implementation. If you think you do need to do this **STOP!**. Consult widely with others (both internally and online) to test your reasoning and ensure you haven't missed something.

Sources:

- Why shouldn't we roll our own? - Information Security - Stack Exchange
- Even if You Don't Invent Your Own Cryptography its Still Hard - Neohapsis Labs

Resources:

- OWASP

[7.3] Where appropriate, APIs **SHOULD** allow anonymous access

APIs **SHOULD NOT** require authentication or authorisation, unless it is necessary.

This ensures APIs are easier to experiment and integrate with, especially for end users such scientists who want to get the information they want as quickly and easily as possible.

Anonymous responses can be externally cached much more easily and effectively. Where the same resources are commonly requested this can lead to significant performance increases for accessing these resources.

Clearly there will be many cases where anonymous access isn't appropriate (or useful). As a general rule GET methods, for most resources, can be made accessible anonymously, whereas other methods would typically require authentication/authorisation.

In some cases authentication is a useful feature for the end user, such as user "favourites" or usage histories. This is perfectly acceptable and **SHOULD** be offered if they help the user.

Sources:

- APIs - Be Public By Default - GDS Service Manual
- 18F API Standards - API keys - I8F

[7.4] Where appropriate, APIs **MUST** restrict access to sensitive information

This includes any information that is personally identifying, confidential or privileged, including anything rated as 'OFFICIAL' under the Government Security Classifications.

Authorisation levels **SHOULD** be created according to need, using the principle of least privileged access.

Sources:

- APIs - Be Public By Default - GDS Service Manual
- Government Security Classifications - GOV.UK
- Principle of Least Privileged Access

[7.5] Where appropriate, APIs **SHOULD** use common authentication sources

APIs **SHOULD NOT** maintain their own directory of users for the purposes of authentication.

Many APIs will be accessible to the same group of people (i.e. BAS/NERC staff). It would therefore make sense to centralise and delegate user authentication to a central service. This reduces the number of passwords users have to remember and reduces complexity in our APIs.

For internal APIs, this central service **SHOULD** be all or part of the NERC Active Directory. This allows the authentication of users to be delegated, saving considerable time and effort. Integration with Active Directory is possible through a range of technologies.

It is reasonable to assume users will be able to readily provide their NERC AD account details, and if not we can delegate password resets to an established process (i.e. the IT Helpdesk). This ensures where authentication is needed, it is as painless as possible.

For external (to NERC) users, a similar approach **SHOULD** be adopted where a common authentication source is created to contain these users, to which our APIs verify credentials. This may take the form of a formal internal LDAP service or users database or, for academic users, Shibboleth, or for the general public, OAuth for providers such as Google, Facebook and Twitter.

These external authentication sources offer similar benefits as internal sources, plus much larger audience sizes and access to additional user information, subject to the provider's or user's approval.

There will likely be times where APIs may need to store information about a user that is specific to that API. In these cases, APIs **SHOULD** store this extra information within the API so as not to 'pollute' the authentication service.

Sources:

- APIs - Be Public By Default - GDS Service Manual

Resources:

- The UK Access Management Federation
- OAuth
- HTTP Basic authentication

[7.6] APIs **SHOULD** use HTTPS

The 18F API Standards summarise the benefits of using HTTPS, echoed here for convenience:

> "Any new API should use and require HTTPS encryption (using TLS/SSL). HTTPS provides:
>
> *Security*. The contents of the request are encrypted across the Internet.
>
> *Authenticity*. A stronger guarantee that a client is communicating with the real API.
>
> *Privacy*. Enhanced privacy for apps and users using the API. HTTP headers and query string parameters (among other things) will be encrypted.
>
> *Compatibility*. Broader client-side compatibility. For CORS requests to the API to work on HTTPS websites -- to not be blocked as mixed content -- those requests must be over HTTPS.
>
> HTTPS **SHOULD** be configured using modern best practices, including ciphers that support forward secrecy, and HTTP Strict Transport Security."

BAS IT and the Web & Applications team maintain a number of SSL certificates on behalf of BAS. These include:

- api.bas.ac.uk (DV certificate)

Where appropriate, APIs can be hosted under domains covered by these certificates to benefit from SSL. For `api.bas.ac.uk` specifically, mulitple APIs can be hosted using a load balancer/reverse proxy approach [1].

For other domains, SSL certificates can be purchased through JANET at a flat rate. This service is available through NERC via BAS IT. SSL certificates are currently not purchased from a central budget, therefore you will need to arrange financing separately through BAS IT.

[1] This system is outside the scope of these guidelines but more information can be found here.

Sources:

- 18F API Standards - Always Use HTTPS - I8F

Resources:

- SSL Labs

[7.7] Where appropriate, APIs **MUST** use HTTPS for sensitive information

This includes anything information that is personally identifying, confidential or privileged, including anything rated as 'OFFICIAL' under the Government Security Classifications.

Sources:

- APIs - Be Public By Default - GDS Service Manual
- Government Security Classifications - GOV.UK

[7.8] Where HTTPS is used, APIs **SHOULD** reject HTTP requests

It is clearer and simpler to support only one protocol and ensures all requests are made securely. Ideally insecure requests **SHOULD NOT** be accepted, i.e. end users should receive a "connection refused" error.

Insecure requests **SHOULD NOT** be silently 'upgraded' or redirected to secure alternatives. This introduces ambiguity and requires additional complexity to support. Taking a hard line approach is not as permissive as redirecting but ensures 'lazy' behaviours are not tolerated.

Sources:

- HTTP API Design - Require TLS - Interagent

Logging & analytics

[8.1] APIs **SHOULD** collect information about themselves to determine user needs and identify problems

Aggregating and analysing request data **SHOULD** inform the future development of our APIs.

Information on which methods and features are used most provides an evidence base for planning future development with the needs of users in mind. This data is unlikely to explain what *new* functionality might be required, but is invaluable in identifying features that are not used.

At a technical level this data is useful at both a macro and micro scale. For example, frequent errors can be highlighted to ensure common problems can be identified or fixed swiftly. Or an issue with a specific resource or even request can be flagged for manual review later. Whilst testing may identify some of these types of issues in advance, it's not feasible to test every condition and request permutation possible.

By recording factors such as the time taken to generate requests and logging queries/requests to data sources or services, the performance of the API can be measured objectively. Knowing which types of requests result in slow database queries, for example, can only feasibly be determined through this type of analysis.

Key to the success of this staggery lies not only in the raw data being available, but in the tools used to collect, store, analyse and present it in a useful fashion. Fortunately, a multitude of third party and self-hosted solutions are available to assist with this. Given the availability of these tools, and high value to effort ratio in terms of the insight they can provide, our APIs **SHOULD** use such an approach.

Sources:

- APIs Document by discovery... and Example - GDS Service Manual
- APIs - Testing - GDS Service Manual

[8.2] APIs **SHOULD** log requests using an unique identifier

Each request made to an API **SHOULD** be given a UUID. This **SHOULD** be returned in the response to each request as a `x-request-id` header allowing the end user to easily store and reference this value.

APIs **SHOULD** log each request ID, along with an other relevant information, such as which endpoint was called, its parameters and options and an indication of what was returned (an error, some data etc.).

This information is invaluable when resolving bugs or issues that are reported.

Sources:

- HTTP API Design - Support Caching with Etags - Interagent

[8.3] Where appropriate, API logs **SHOULD** be anonymous

Unless required, identifying information such as full IP addresses, location data, etc. **SHOULD** not be stored.

Caching & version control

[9.1] Where appropriate, API responses **SHOULD** be able to be cached

For the purposes of these guidelines, there are two forms of caching:

- *External* caching, caches endpoint responses and can provide information even when the API itself is unavailable.
- *Internal* caching takes place within the API application, e.g. database query caching.

External caching has much greater potential impact compared to internal caching and **SHOULD** be used for APIs with large (thousands) of users. However this method of caching is only truly effective for anonymous requests that return the same information. If authentication is used, its effectiveness is decreased.

Caching, of any kind, **SHOULD** only become a focus after an API has been launched and is proven, through metrics, to have performance problems. That said, techniques that would make adding caching more difficult, such as a authenticating all methods, **SHOULD** be taken into consideration during development, so as not to make adding caching more difficult if needed later.

[9.2] Where appropriate, APIs **SHOULD** use Etags for cached resources

The Etag for a resource **SHOULD** be the same regardless of the data-type used for the response. I.e. if both JSON and XML data-types are available they should both have the same Etag.

Sources:

- Rest Resources - Version Control for Entities section - Atlassian REST API Design Guidelines (V1)
- HTTP API Design - Support Caching with Etags - Interagent

[9.3] Where appropriate, APIs **SHOULD** use sensible time limits for cached responses

Where caching is used, suitable measures **SHOULD** be used to ensure stale or invalid data is removed in a timely fashion.

Techniques such as `Cache-Control` headers can be used to instruct caching systems, and end users, how long a response should be cached before a new request is made.

End users **SHOULD** be able to query this information in subsequent requests using an `If-None-Match` header.

For example, an endpoint may update once an hour with a new ship position. Responses from this endpoint **SHOULD** include a `Cache-Control` header, this informs the end user the information will not change until the next update. This also ensures any request made after this period has expired will automatically receive any updated information.

Sources:

- Rest Resources - Version Control for Entities section - Atlassian REST API Design Guidelines (V1)
- HTTP API Design - Support Caching with Etags - Interagent

[9.4] Where appropriate, APIs **SHOULD** use rate limiting to ensure the API is used fairly

In some cases [1] rate limiting can be used to ensure end users do not (inadvertently) overwhelm the API or cause performance/access problems for other users.

[1] For endpoints that may update frequently or which are expensive to process.

Sources:

- HTTP API Design - Show Rate Limit Status - Interagent

Resources:

- Token Bucket - Wikipedia

API requests

[10.1] API methods **MUST** be available at unique endpoints

Within these guidelines, an *endpoint* is said to consists of 4 elements:

1. HTTP verb (e.g. GET)
2. HTTP headers (e.g. `Accept: application/json`)
3. URL path (e.g. `/ships`)
4. URL parameters (e.g. `?operator=BAS`)

The HTTP verb (1) and URL Path (3) **MUST** be specified in each API request, these are referred to as *required* elements.

HTTP headers (2) and URL parameters (4) can be specified in an API request, these are referred to as *optional* elements.

An API endpoint is considered unique if its required elements are unique.

For example,

- `[GET]v2/ships` and `[GET]v2/ships/{id}` are unique, as the URL path is different.
- `[GET]v2/ship/{id}` and `[PUT]v2/ship/{id}` are unique, as the HTTP verb is different.
- `[Get]v2/ships` and `[Get]v2/ships?id={id}` are **NOT** unique, as the required elements are the same.

This ensures each API endpoint can be bookmarked and referenced easily and that each API request refers to a unique API method, which is needed for analytics.

Sources:

- https://www.gov.uk/service-manual/making-software/apis.html#give-each-thing-a-bookmarkable-url
- 18F API Standards - API Endpoints - I8F

[10.2] API endpoints **SHOULD** not change

Where endpoints do change location and perform the same function, redirects (e.g. `301 Moved Permanently`) **SHOULD** be used to ensure backwards compatibility. You **MUST NOT** do this if the method an endpoint refers to has changed, or where doing so would introduce confusion and ambiguity.

Sources:

- APIs - Practice Service Evolution - GDS Service Manual

[10.3] API endpoints **SHOULD** use an appropriate HTTP verb

For example, mapping to CRUD these would be:

- `GET` for retrieving/reading
- `POST` for creating
- `PUT` for updating
- `DELETE` for destroying/deleting

Sources:

- https://www.gov.uk/service-manual/making-software/apis.html#use-http-methods-as-tim-intended

[10.4] API endpoints **SHOULD NOT** rely on HTTP verbs other than `GET` and `POST`

A mechanism for using `POST` requests with some way to specify the intended verb **SHOULD** be supported.

For example, a `_method` parameter could be used to fake other verbs.

APIs **SHOULD NOT** use verbs such as PATCH which are unambiguous. APIs **SHOULD** treat less common verbs with caution and **SHOULD** always offer a well supported alternative if used.

Sources:

- https://www.gov.uk/service-manual/making-software/apis.html#use-http-methods-as-tim-intended

[10.5] API endpoints **MUST NOT** allow GET requests to alter resources

GET endpoints **MUST** be safe and read-only.

Sources:

- https://www.gov.uk/service-manual/making-software/apis.html#use-http-methods-as-tim-intended

[10.6] API endpoints **SHOULD** be self descriptive

It **SHOULD** be possible to gather what an endpoint does from its URL alone. Therefore endpoints **SHOULD** be as self descriptive as possible, whilst being concise and ensuring endpoints remain well structured.

The easiest way to do this is to use descriptive and logical names for endpoints, you **SHOULD NOT** use verbs.

Endpoints **SHOULD** be structured logically and intuitively. Methods for a resource **SHOULD** be nested under the resource.

For example, `[GET]v2/ships/{id}/voyages` indicates the resource is `ships`, `{id}` the specific resource and the method is `voyages`. From this alone, it should be reasonably obvious that this endpoint returns a list of voyages for a ship specified by `{id}`.

Sources:

- 18F API Standards - API Endpoints - I8F

[10.7] API endpoints **SHOULD** be succinct

Ideally endpoints **SHOULD** be snappy. Longer URLs require more effort to understand and increase the chance of making a mistake (in the spelling, order, etc.).

Endpoints do not need to reflect the underlying relationship between resources, providing this full relationship 'chain' is not relevant for the endpoint.

The Interagent HTTP API Design guidelines provide a good example of when this technique makes sense. In this case there is no need to understand what the resources used in this example are, other than their relationship (shown in the first URL example).

"

```
/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}
```

"

Sources:

- HTTP API Design - Minimize Path Nesting - Interagent

[10.8] API endpoints **SHOULD** be case insensitive using hyphens as separators

This aids consistency and removes ambiguity, it also preserves compatibility with hostnames.

For example, `ships` not `Ships`, `cruise-reports` not `cruise_reports` or `cruiseReports`.

Sources:

- HTTP API Design - Downcase Paths and Attributes - Interagent

[10.9] Where appropriate, API endpoints **SHOULD** use the plural term for a resource

The same endpoint **SHOULD** be used for both "a thing" and "a collection of things". This helps keep endpoints consistent as resources change and are more predictable by not splitting methods over multiple URL paths.

For example, `[GET]v2/ships/{id}` and `[GET]v2/ships` both use `ships` even though the first returns a single *ship* and the latter multiple *ships*.

You **SHOULD NOT** have URLs such as:

- `[GET]v2/ships/method_A`
- `[GET]v2/ship/method_B`
- `[GET]v2/ships/method_C`

Where a resource is referred to elsewhere (i.e. as a parameter in another method) its plural name **SHOULD** be used.

For example `[GET]/v2/search/ships/{call_sign}` not `[GET]/v2/search/ship/{call_sign}`.

Sources:

- https://www.gov.uk/service-manual/making-software/apis.html#give-each-thing-a-bookmarkable-url
- HTTP API Design - Use Consistent Path Formats - Interagent

[10.10] Where appropriate, APIs **SHOULD** support aliases for a resource

Typically IDs will have no relation to the resource to which they are assigned, whether this is a randomly assigned numeric ID or UUID. Whilst this ensures resources are uniquely identified, they are not very user friendly.

In many cases resources will contain one or more "pseudo-identifiers" such as a username that are unique for a resource, thus could act as an identifier, whilst being much easier to remember or understand. Where appropriate, APIs **SHOULD** support using pseudo-identifiers as aliases.

Aliases **SHOULD** guarantee the correct resource will be returned, i.e. pseudo-identifiers **SHOULD** be as unique as regular identifiers are. A judgement should be made to ensure a balance between ease of referring to resources, and accuracy of results.

Whilst supporting aliases does increase the complexity of an API, providing this behaviour is documented properly, there should be no significant loss of clarity or unambiguousness. Clearly there will be *some* loss which highlights the judgement aspect of this guideline. In general, the cases where this behaviour **SHOULD** be used will be obvious. If in doubt, you **SHOULD NOT** use an alias.

For example, the call sign of BAS ship's are not likely to change and are an attribute by which a ship is commonly known. In this case, it would be appropriate to use the ship callsign as an alias for a ship resource.

Sources:

- HTTP API Design - Support Non-Id Dereferencing for Convenience - Interagent

[10.11] URL parameters **MUST NOT** be required

URL Parameters (query strings) **MUST** be optional.

Generally URL parameters are used for expressing preferences or filtering information in a request, for example setting sorting preferences or specifying a date range. API requests made without these parameters would be the same in both cases, though in a different order or a super set of the same information.

Sources:

- https://www.gov.uk/service-manual/making-software/apis.html#give-each-thing-a-bookmarkable-url

[10.12] The order of URL parameters **MUST NOT** be significant

It **MUST NOT** matter in what order URL parameters are given.

For example,

`?from=1991-12-22&until=20010-07-21` and `?until=20010-07-21&from=1991-12-22` **MUST** return the same response if all other factors are the same.

Sources:

- https://www.gov.uk/service-manual/making-software/apis.html#give-each-thing-a-bookmarkable-url

[10.13] Where appropriate, API endpoints which accept JSON **SHOULD** support serialised JSON within the request body

Where JSON is used for both the request and response data-type this provides a symmetry, simplifying requests for the end user.

This also prevents any errors when converting JSON data into HTTP key-value pairs. This makes API integrations simpler and clearer without introducing ambiguity in our API.

For example,

```
$ curl -X POST https://ships-api/V2/ship \
-H "Content-Type: application/json" \
-d '{"name": "RRS James Clarke Ross"}'
```

Sources:

- HTTP API Design - Accept Serialised JSON in Request Bodies - Interagent

API responses

[11.1] API responses **SHOULD** use an appropriate status code

It isn't practical to give guidance on specific status codes for every situation, look for the common consensus elsewhere.

4XX responses **SHOULD** be used to indicate client errors, i.e. that the user must fix, 5XX responses **SHOULD** be used to indicate server errors, i.e. that we must fix.

Sources:

- 18F API Standards - Error Handling - I8F

[11.2] Where appropriate, API responses **MUST NOT** use a 2XX status code for an unsuccessful request

If the API is unable to return a response to a request according to its documented output a 2XX status code **MUST NOT** be used.

For example, a request is made for a resource that cannot be found, the API is unable to return the expected response meaning a 2XX status code **MUST NOT** be used.

If, for the same request, the resource can be found but the method used has been deprecated, the API has been able to return the expected output, but has some additional information to include (the method is deprecated). In this case a 2XX status code **SHOULD** be used, and the additional information included in the request (i.e. as a warning).

Sources:

- 18F API Standards - Error Handling - I8F

[11.3] Where appropriate, APIs **SHOULD** support CORS

CORS is an industry standard for allowing access to APIs from multiple origins from within web browsers.

All major browsers, web servers (such as TomCat) and frameworks (such as jQuery) support this standard providing wide interoperability.

Depending on how CORS is configured, access can be restricted at the origin level. Outside of browsers these restrictions are meaningless (its just a header) but CORS provides good protection against some web based attacks.

APIs **SHOULD NOT** use JSONP. It is a hack and we **SHOULD NOT** condone its use. The 18F API Standards provide good reasons for this, they are echoed for convenience:

"JSONP is not secure or performant. If IE8 or IE9 must be supported, use Microsoft's XDomainRequest object instead of JSONP. There are libraries to help with this."

Sources:

- 18F - - 18F
- Why JSONP Is a Terrible Idea and I Will Never Use it Again - tmcw

Resources:

- Enable CORS
- Corslight - Mapbox

[11.4] API responses **SHOULD** support appropriate data formats, defaulting to the simplest

The default format **SHOULD** be the most useful for the given response. Where multiple data formats are offered, at least one **SHOULD** be a widely interoperable, ideally open, format.

Often more complex, specialised formats may be more useful for particular purposes. These **SHOULD** be offered, where feasible and based on user need, alongside simpler but more interoperable alternatives.

Sources:

- https://www.gov.uk/service-manual/user-centred-design/choosing-appropriate-formats.html

Resources:

- List of common formats

[11.5] APIs **SHOULD** use HTTP content negotiation to decide which supported data format to use

Content type (data type) negotiation using the `Accept` and `Content-Type` headers is a best practice which is widely supported and tested.

Sources:

- Content Negotiation - Mozilla Developer Network

[11.6] API responses **SHOULD** offer JSON as a supported data format

Where a single data format is offered this **SHOULD** be JSON, unless another, more suitable, format is available.

The 18F API Standards summarise the benefits of using JSON, echoed here for convenience:

"JSON is an excellent, widely supported transport format, suitable for many web APIs.

Supporting JSON and only JSON is a practical default for APIs, and generally reduces complexity for both the API provider and consumer."

Sources:

- 18F API Standards - Just Use JSON - I8F

[11.7] Where appropriate, API responses using JSON **SHOULD** use objects and underscored attributes in lower case

Using an array to return data limits the ability to include metadata (including errors/warnings/notices etc.) and limits the ability to add additional top-level keys in the future.

Different languages use different case conventions. JSON uses underscored properties, and ensures attributes can be used without quotes in JavaScript.

For example, use `cruise_reports` not `cruise-reports` or `cruiseReports`.

Sources:

- 18F API Standards - Just Use JSON - I8F
- HTTP API Design - Downcase Paths and Attributes - Interagent

[11.8] Where appropriate, API responses **SHOULD** return a full representation of a resource

This prevents additional requests (for example a POST followed by a GET request for the same resource), which reduces load on the API and improves the experience for the user.

Where a response may contain nested resources, judgement should be used whether to include all or part of each nested resource.

Where appropriate, nested resources **SHOULD** be returned in full, as requesting each nested resource separately would create significantly more requests than simply making the original request alone.

Sources:

- HTTP API Design - Provide Full Resources Where Available - Interagent

[11.9] Where appropriate, API responses **SHOULD** nest related resources

For a resource containing relations to other resources, these **SHOULD** be nested within the resource returned.

This guideline applies regardless of the number of attributes each related resource has.

The Interagent HTTP API Design guidelines provide a good example of why this technique makes sense.

> " 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...",
>         "name": "Alice",
>         "email": "alice@heroku.com"
>     },
>     ...
> }
> ```
>
> "

Sources:

- HTTP API Design - Nest foreign key relations - Interagent

[11.10] Where appropriate, API responses **SHOULD** support pagination for managing large numbers of items

Where a response involves returning a large number of resources, pagination **SHOULD** be used to split these items into a number of "chunks". Only one chunk **SHOULD** be returned in a response, with additional requests used to request other chunks.

The API **SHOULD** provide the mechanism for managing this process and communicating with the user using meta-data. This meta-data **SHOULD** include, where chunking has been used, which chunk is present and how to request other chunks.

There are numerous implementations of pagination, most have similar features but expose these to the end user in different ways. Some use HTTP headers whilst others data structures within response bodies themselves.

These guidelines don't make a preference as to which implementation **SHOULD** be used, and there is no fixed point where pagination needs to be used. However any solution **SHOULD** allow the user to determine the size of each chunk.

Pagination is often combined with sorting to return the most useful information first.

Sources:

- HTTP API Design - Paginate With Ranges - Interagent

Resources:

- Ranges - Heroku Platform API

Unreleased information

The information in this section is still being researched, written-up or waiting to be included to existing sections of the guidelines.

You **MUST NOT** implement any guidelines contained in this section.

You **MUST NOT** rely on any information in this section, it may be incomplete (and therefore misleading by omission), inaccurate, or otherwise wrong.

[11.11] Where appropriate, API responses **SHOULD** use a standard response object

This object **SHOULD** from the body of the response and be the top level object. It **SHOULD** support multiple, distinct, "aspects" of a response.

These aspects may include:

- *Errors* - Used for reporting fatal errors
- *Warnings* - Non-fatal errors, or information with a negative context such as deprecation warnings
- *Notices* - Information of any other context, related to the API in general or to specific responses, such as new feature releases etc.
- *Data* - A general or default aspect, usually this will contain resource representations
- *Meta* - Meta-data about the *Data* aspect, such as the number of resources returned

Note: The purpose (and list) of these aspects is yet fully decided and may change in future revisions of these guidelines. This applies mainly to the *notices* aspect.

Aspects **SHOULD** always be used in plural. If used, the *Errors* aspect **MUST** be used alone.

[11.12] Where appropriate, API error responses **SHOULD** use a standard response format

APIs should use a standard response format to increase interoperability with other tools and services.

There are a range of standards available, and these guidelines currently have no preference on which error standard should be used. Critically we **SHOULD NOT** come up with our own standard.

Resources:

- JSON API error format
- Draft HTTP problem specification