API Design Guide Documentation

Release 0.1

AusDTO

Jun 02, 2017

Contents

CHAPTER 1

Preface to the API Design Guide

A design guide with developer empathy.

Like websites, shop-fronts and call centres, APIs (Application Programming Interfaces) could be viewed as simply another channel for delivering services online. APIs power many of today's websites and mobile apps, and are becoming increasingly important in the digital economy. This is because APIs can be incorporated into downstream applications, whereby third-parties reuse and recombine them in creative ways. Good APIs strip a digital service down to it's simplest possible form, so that the value of the service can be amplified through third-party reuse.

The principle of user-centred design is at the core of website and application development. It's about ergonomics; can a user "consume a digital service" without bad design getting in their way? Getting this right requires empathy for users, backed by evidence regarding user-preferences. Although a well known principle, it is not always well practised.

In some ways, developer empathy is exactly the same; developers are people too, and the users of web APIs. The key difference is that when developers use an API, they often create additional value for many people. A well designed end-user experience results in a single good user experience. A well-designed API on the other hand can result in many good user experiences, often enjoyed by those the original service designer did not envision.

> We have developer empathy. If our interfaces are hard to use, we consider that a bug. We are an engineer-led company. Everything we do tries to be compatible with our API.
>
> —Steve Mayzak, ElasticSearch (around 6.20 into https://www.youtube.com/watch?v=uxfvNwl_MGc).

User empathy is focused on ergonomics at the point of delivery. Developer empathy is focused on painless systems integration. Good API design is still fundamentally about ergonomics, but the context is different. As a user, successful design has the quality of affordance (obviousness, "don't make me think"). For developers, good API design also incorporates this quality (through idioms and good documentation). Further, good API design allows developers to integrate their applications with your systems in a decoupled manner; they are able to reuse the resources exposed by your API in previously unimagined ways. Lastly, good APIs are stable and adhere the 'principle of least surprise': developers are able to rely on your API to behave in a predictable manner.

The gold standard in developer empathy is found in thriving open source projects. Collaboration, peer review and a responsive community are the hallmarks of such projects. Organisations that do well in this space nurture their relationship with developers as though their survival depends on it (quite often because it does). This does not describe Australian Government IT of today, but hopefully describes the "government as a platform" of tomorrow. Some things are certain: APIs are important, they can generate additional and significant value, developer empathy is critical to successs and we need all the help we can get.

The DTO planned to release a draft API design guide describing contemporary best practices in API design and developer empathy. The final document (API Design Guide) will inform assessment of conformance against mandatory policy (the Digital Service Standard) that all Commonwealth agencies must follow, and that citizens could be entitled to expect. The purpose of releasing a draft was to get feedback, but at the last minute we had a better idea...

We want feedback from developers, right? So we figured: why not convert it to a developer-friendly format, published in a version control system with a public ticket system for issues and support, and invited our users to help us make it better?

Done.

- read https://apiguide.readthedocs.org/
- discus https://github.com/AusDTO/apiguide/issues
- kanban https://waffle.io/AusDTO/apiguide
- fork https://github.com/AusDTO/apiguide.git

Pull requests welcome!

CHAPTER 2

Copyright

CHAPTER 3

Principles

Developer empathy

Design with developer empathy

When it comes to APIs, developers are your users. The same principles of user-centred-design apply to the development and publication of APIs (simplicity, obviousness, fit-for-purpose etc). Most of the other criteria in this guide are traceable to the principles of good design. Perhaps the most important criteria to be mindful of is *simplicity*: as with any other product, people simply won't use something if it is hard to use.

Hard-to-use APIs create bad public policy outcomes:

- Potentially significant economic value will remain undiscovered or unrealised
- Secondary markets that form around APIs (e.g. automated tax reporting software) will be less competitive, and therefore less efficient (API complexity creates barriers to market entry)

Recommended

An API that is consistently rated as hard to use should be remediated. Ensure less than 20% of feedback rates the API as hard to use.

Granularity

Design granular, re-useable APIs

A common misconception is that there is a one-to-one mapping between a service in the paper world and a corresponding API. In reality, this is almost never the case. APIs should be designed at the lowest practical level of granularity because this makes each service simpler and allows them to be combined in ways that suit the consumer. The key principle is to design services that can be re-used and combined in different ways.

For example, an agency paper form often combines multiple separate functions on the one document to provide a simpler experience for users of the paper form. Such as, a tax form might collect address data as well as tax information.

In the electronic world, it is better to treat each of the functions as a separate API and have a separate service for each. An address change is logically a separate event that is not related to tax-time reporting. If the address update

is part of the tax return then there's no way to advise government of an address change without also lodging a tax return.

A useful method to determine the right service granularity is to identify the key entities that the service impacts and to model their life cycle/s. There is typically one API operation for each entity life cycle state transition.

Recommended

There should be only one API function for one business outcome (e.g. change an address).

Functionality

Design APIs to provide full coverage

When your API service is being integrated (or 'mashed up') into another service then it's important to ensure you provide APIs that cover the full process life cycle. Failure to do so would seriously impact the user experience, in the consumer service, because users would need to jump between applications to complete a process.

For example, a payroll tax lodgement API that can be consumed by a business payroll system is much less useful if the user must still visit the State Revenue Office portal to register for payroll tax, query obligation calendar, query payment balances, download assessments, and so on.

The same entity life cycle modelling approach that you use to identify service granularity will also help you to understand which services are necessary to support the full business process.

Recommended

Any function that is available via a (retail) UI should also be available as an API (wholesale).

Ubiquitous standards

Use ubiquitous web standards

Consider API development to be just part of building a website. When choosing technology standards, government should be a follower, not a leader. Just use the same standards and methods that are widely used on the web, following examples set by the leading cloud service platforms. This does not mean use the same technology for everything, but it does mean use the most universally accepted technology solution for each business pattern or requirement.

When choosing technology solutions, the key principles are to:

- pick the technical solution that minimises implementation effort for API consumers
- make no assumptions about the software development technologies used by consumer applications.

Any agency that builds an API service interface using a niche technology that is not natively supported by client software development languages will face considerable adoption challenges. Either a number of clients will be commercially or technically excluded, or the agency will need to build, supply, and maintain developer kits for all client technology platforms.

Follow the links in the table for further guidance on the best practice use of each technology.

Table 1: API Technology Selection Guide

Type	Business Pattern	Qualities	Preferred Technology	Data Format	Potential Examples
1	Public bulk data file	Offline, public	Upload to data.gov.au	CSV, XML, or JSON	Bulk ASIC register, 2015 budget data.
2	Content syndication	Online, public	ATOM	XML	ABC radio programs
3	Public data service	Synchronous, public	REST	JSON and XML	ABR query service
4	Private query or transactional service	Synchronous , authenticated	REST with OpenID Connect	JSON and XML	Grant application & acquittal reporting
5	Peer-to-peer message exchange	Asynchronous, routable, secure, reliable	ebMS3/AS4 with SAML2.0	XML	SuperStream Rollover, UBL e-Invoice

It is expected that the vast majority of government API services will be type 3 or 4. Therefore we provide further detailed information on REST/JSON APIs in Building and publishing APIs and further detailed information on OpenID Connect in Securing APIs.

Recommended

All new or refreshed API development should conform to the standards in this guide.

Special circumstances

Use specialised technologies only in special circumstances

There are some cases where a specialised technology is the right choice because it solves a specific problem that would otherwise be very complex and/or there is no other reasonable technical option.

May

When a case can be made that specialised technology is the best fit for solving a speficic technical problem, an agency may chose to use that technology over adopting a more ubiquitous standard.

Should

When an agency chooses to apply a specialised technology, they should only use that technology for the specific problem it's designed to solve.

For example, ebMS3 / AS4 is an advanced messaging technology that is built upon XML/SOAP. It is designed for the specific problem of peer-to-peer messaging. Unlike the 'client-server' model where an agency provides an API service and many clients can consume the service, the peer-to-peer model has no sense of a service provider or consumer. Instead it is characterised as a community of members that need to exchange electronic documents with each other (such as, B2B e-invoicing).XBRL is a niche technology that is specifically designed for corporate regulatory financial reporting. It is a 'fact distribution' model where the provider of data can report any fact so long as it exists in a reference taxonomy (that is, a dictionary of terms). This offers a lot of flexibility for the provider of financial reports that essentially need to provide a machine-readable version of their corporate annual report. But it's a poor fit for most service APIs that normally need to specify what data must be provided by the consumer.

Although these niche technologies **can** be used outside their design scope, this does not mean that they **should**. The vast majority of government service APIs are a simple client-server model and overloading them with niche or complex technologies will cause unnecessary barriers to uptake.

Should

Agencies should be prepared to provide a rationale for diverging from the standards specified in this guide.

Community standards

Follow consistent information standards

Consistent use of information standards and naming conventions makes your APIs much more usable and interoperable.

In some cases, the complete information structure is externally imposed because you are participating in an information sharing community that has already defined data standards. For example:

- An agency as an employer must send superannuation member contributions as defined by the SuperStream specification.
- An agency as a buyer may support electronic invoicing using a standard such as the UN/CEFACT cross industry invoice.

But in most cases the API will be implementing an agency specific service so there won't be any externally imposed standard that completely describes the service. In these cases, agencies should still make every effort to construct their service using industry standard information components. For example:

- iCalendar for events
- vCard (or jCard) for name & address
- KML geospatial data
- Microformats when embedding structured data in HTML.

Some useful resources for standard terms and codes that can be used to construct more understandable and interoperable APIs are:

- Australian Reporting Dictionary www.dictionary.sbr.gov.au
- US government National Information Exchange Model www.niem.gov/technical/Pages/current-release.aspx

Recommended

Where an established and widely used information standard exists then it should be used.

Clear error messages

Provide useful error messages

Graceful handling of failure conditions is an essential part of delivering a high quality API.

Recommended

Error messages should conform to the standards relevant for the technology in use.

Recommended

Error messages should provide a human-readable error message that is designed to be read and understood by the user.

Recommended

Error messages should include a diagnostic message that contains technical details for use by the developers/maintainers of the application that consumes the API.

May

Diagnostic messages may include links to documentation and other 'hints' that might assist developers resolve issues that may have resulted in the error condition.

Further information on error messaging for REST APIs is provided in the Build and publish APIs guidance.

Endpoint stability

Provide appropriate stability and availability

When a third-party software application integrates a government API, then it may become dependent on the continued availability of that API for the software to function correctly. The software package must also depend on the stability of the API so that changes can be planned to fit within a normal software product release cycle. These requirements impose the following service management principles on agency APIs:

- The availability of the API must be no less than that availability of the equivalent agency online service or website.
- Where government APIs are essential components of national processes, then they must be designed and operated for continuous availability. Examples are ABN and TFN verification service APIs.
- Changes to APIs must always be deployed as fully backwards compatible upgrades. If they are not backwards compatible, the old API version must be maintained alongside the new version for an appropriate period to allow all consumers to transition.
- Alpha or Beta versions of APIs may have lower availability and stability but the service level should still be clearly specified.

Further information on this topic is provided in the Operate & Improve APIs guidance.

Recommended

All APIs should have a published SLA and behave accordingly.

Privacy and security

Provide appropriate security and privacy

The level of protection required for a specific API depends on a risk assessment (e.g. of the consequences of an information leak).

Should

Agencies should match each API to the relevant assurance level so that the controls applied can be comapared to the recommendations for that level.

Table 2: API assurance level implementation guidance

Assurance Level	Identity Confidence	Example information set	Recommended controls
0	None	Open data - for example, all data sets on data.gov.au	None
1	Minimal	Public Data validation services - for example, ABN Lookup	Authentication GUID issued to any self-registered entity
2	Low	Parking permit application, tax return, balance enquiry, private data validation (for example, TFN validation)	OpenID Connect to accredited identity provider with single factor authentication and level 2 identity confidence (for example, online registration process with shared secret)
3	Moderate	Create or update bank details (for payments & refunds), personal medical records	OpenID Connect to accredited identity provider with multi-factor authentication and level 3 identity confidence – for example, face-to-face registration with 100 point Document Verification Service (DVS) verified Proof of Identity (POI)
4	High	Not applicable to public API services	Physical identity verification including biometrics

It is very important to apply the right assurance level to each granular API operation. Failure to do so can seriously impact user experience or service risk.

Guidance on the best practice usage of the OpenID Connect protocol for securing government API services that require NEAF assurance level 2 or 3 is provided in the Securing APIs guide.

For peer-to-peer messaging using standards such as ebMS3/AS4, the same assurance level is achieved but with different protocols (WS-Security and SAML2.0).

Recommended

All APIs should specify their assurance level when applying the relevant controls.

Documentation

Ensure APIs are discoverable and documented

For APIs to be used, they must be discoverable and documented. Developers often cite the requirement for good documentation as the single most important quality of APIs.

Recommended

Publish API documentation, and provide a link to the documentation from the API endpont.

Recommended

Provide feedback and support channels for API users.

May

Consider using some sort of *literate system* that generates the API documentation from API code (or vica verca), to ensure so that there is always a precise alignment between the API and it's documentation.

May

Consider using open source API specification tools such as http://swagger.io/, to simplify and standardise the means by which APIs are documented.

Recommended

All APIs that are published should be adequately documented and supported by the agency that produces them.

Dogfooding

Consume your API for your online services

Agencies that publish APIs must use their own APIs for the providing the equivalent online service. By consuming your own API you will ensure that your API (wholesale) service:

- remains aligned with your online (retail) service
- maintains at least the same availability level as your online (retail) service.

For example, an agency has an online web form for a grant application. The web form is completed by a grant applicant and the 'submit' action sends the form data to the grant application API.

Another applicant has chosen to use the services of an intermediary that specialises in grant applications. The intermediary has their own application for grants management and so the application is lodged directly to the agency API, bypassing the online form.

In both these cases, the agency back-end grants assessment system receives the data from the same API interface. If the agency wants to change the online form, they must also change the API – and so both remain aligned.

Recommended

User interfaces (reatail) should consume the corresponding (wholesale) API.

CHAPTER 4

Building and using APIs

Recommended

Agencies are encouraged to make all digital services available as an API,

Recommended

Agencies are encouraged to deliver all APIs following 'Pragmatic REST' (Representational state transfer) principles detailed below.

Use RESTful service URLs

Under REST principles, a URL identifies a resource. The following URL design patterns are considered REST best practices:

- URLs should include nouns, not verbs.
- Use plural nouns only for consistency (no singular nouns).
- Use HTTP methods (HTTP/1.1) to operate on these resources:
- Use HTTP response status codes to represent the outcome of operations on resources.

Should

Agencies should consistently apply RESTful design patterns for API URLs.

Versioning

Example of an API URL that contains a version number:

```
GET /v1/path/to/resource HTTP/1.1
Host: www.example.gov.au
Accept: application/json, text/javascript
```

May

An API URL may contain a version number.

See *versioning* for more details.

Should

If an API URL does not contain a version number (anonymous version), then it should be understood that it always refers to the latest version.

Should Not

Anonymous API versions should not be considered stable, because the latest version changes with each release.

Formats

Allow users to request formats like JSON or XML, for example:

- http://example.gov/api/v1/magazines.json
- http://example.gov/api/v1/magazines.xml

URL Depth

The resource/identifier/resource URL pattern is sufficient for full *attribute visibility* between any resources. Therefor, this URL depth is usually sufficient to support any arbitrary resource graph. If your URL design goes deeper than resource/identifier/resource, it may be evidence that the granularity of your API is too coarse.

Recommended

Avoid URL designs deeper than resource/identifier/resource.

May

If your API has URLs deeper than resource/identifier/resource, consider revisiting the granularity of your API design.

API Payload formats

To interact with an API, a consumer needs to be able to produce and consume messages in the appropriate format. For a sucessful interaction both parties would need to be able to process (parse and generate) these messages.

Should Not

Agency APIs should not produce or consume messages in a propietary format. This is because open formats maximise interoperability and reduce costs and risks associated with API utilisation.

May

Agency APIs may support multiple (open) payload formats. For example, it is not unusual for an API endpoint to support both JSON and XML formats.

API Payload format encoding

To interact with an API, the consumer needs to know how the payload is encoded. This is true regardless of how many encoding formats the endpoint supports.

Should Not

Agencies should not rely on documentation alone to inform consumers about payload encoding. This is generally poor *affordance*.

The three patterns of payload format encoding most frequently found in the wild are:

- HTTP headers (e.g. *Content-Type:* and *Accept:*)
- GET parameters (e.g. *&format=json*)
- resource label (e.g. */foo.json*)

Using HTTP headers to specifying payload format can be convenient, however unfortunately not all clients handle headers consistently. Using HTTP headers alone will create issues for buggy clients.

Using GET parameters to specify format is another common pattern for specifying the encoding of API payloads. This results in slightly longer URLs than resource label technique, and can occasionally create problems with caching behavior of some proxy servers.

Resource label specification of API payload format, such as */foo/{id}.json*, are functionally equivalent to GET parameter encoding but without the (admittedly rare) proxy caching issues.

Should

Agency APIs should consider supplimenting URL-based format speficications with HTTP header based format specification (e.g. *Content-Type:* and *Accept:*).

Should

Agency APIs should consider use resource labels to indicate payload format, e.g. */foo/{id}.json*.

Should

If GET parameter based payload format specification is chosen, the potential impact of proxy caching and URL length issues should be evaluated.

Good RESTful URL examples

List of magazines:

```
GET /api/v1/magazines.json HTTP/1.1
Host: www.example.gov.au
Accept: application/json, text/javascript
```

Filtering and sorting are server-side operations on resources:

```
GET /api/v1/magazines.json?year=2011&sort=desc HTTP/1.1
Host: www.example.gov.au
Accept: application/json, text/javascript
```

A single magazine in JSON format:

```
GET /api/v1/magazines/1234.json HTTP/1.1
Host: www.example.gov.au
Accept: application/json, text/javascript
```

All articles in (or belonging to) this magazine:

```
GET /api/v1/magazines/1234/articles.json HTTP/1.1
Host: www.example.gov.au
Accept: application/json, text/javascript
```

All articles in this magazine in XML format:

```
GET /api/v1/magazines/1234/articles.xml HTTP/1.1
Host: www.example.gov.au
Accept: application/json, text/javascript
```

Specify query parameters in a comma separated list:

```
GET /api/v1/magazines/1234.json?fields=title,subtitle,date HTTP/1.1
Host: www.example.gov.au
Accept: application/json, text/javascript
```

Add a new article to a particular magazine:

```
POST /api/v1/magazines/1234/articles.json HTTP/1.1
Host: www.example.gov.au
Accept: application/json, text/javascript
```

Bad RESTful URL examples

Non-plural noun:

```
GET /magazine HTTP/1.1
Host: www.example.gov.au
Accept: application/json, text/javascript
```

```
GET /magazine/1234 HTTP/1.1
Host: www.example.gov.au
Accept: application/json, text/javascript
```

Verb in URL:

```
GET /magazine/1234/create HTTP/1.1
Host: www.example.gov.au
Accept: application/json, text/javascript
```

Filter outside of query string:

```
GET /magazines/2011/desc HTTP/1.1
Host: www.example.gov.au
Accept: application/json, text/javascript
```

Use HTTP methods and status codes

Meaningful status codes helps consumers utilise your API.

Should

HTTP methods and status codes should be used in compliance with their definitions under the HTTP/1.1 standard.

The action taken on the representation will be contextual to the media type being worked on and its current state.

Examples

`GET /magazines`
List all magazines contained within the /magazines resource

```
GET /magazines HTTP/1.1
Host: example.gov.au
Accept: application/json, text/javascript
```

A response of "200 OK" indicating that the request has succeeded.

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: text/javascript

{...}
```

Status Codes

- 200 OK – no error
- 404 Not Found – the "magazines" resource does not exist

`POST /magazines`
Create new magazine within the /magazines resource

```
POST /magazines HTTP/1.1
Host: example.gov.au
Accept: application/json, text/javascript

{...}
```

A response of "201 Created" indicates that the request has been fulfilled.

The URI of the new resource is provided in the response

```
HTTP/1.1 201 Created
Vary: Accept
Content-Type: text/javascript

{ "id": "1234" }
```

However, no effect if the resource already exists.

```
HTTP/1.1 405 Method Not Allowed
Vary: Accept
Content-Type: text/javascript

{
  "developerMessage" : "Unable to create a magazine with ID of 1234 because a␣
↪magazine with that ID already exists",
```

```
  "userMessage" : "Unable to create duplicate magazine 1234",
  "errorCode" : "444444",
  "moreInfo" : "http://api.example.gov/v1/documentation/errors/444444.html"
}
```

Status Codes

- 201 Created – magazine created
- 405 Method Not Allowed – unable to create "magazine" resource

PUT /magazines

Replace all magazines in the /magazines resource with those in the request

```
PUT /magazines HTTP/1.1
Host: example.gov.au
Accept: application/json, text/javascript

[ ... ]
```

200 indicates that the request has succeeded.

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: text/javascript

{ "id": "1234" }
```

Status Codes

- 200 OK – magazines replaced

PUT /magazines/1234

Replace the /magazines/1234 resource with the representation in the request

```
PUT /magazines/1234 HTTP/1.1
Host: example.gov.au
Accept: application/json, text/javascript

[ ... ]
```

200 indicates that the request has succeeded.

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: text/javascript

{ "id": "1234" }
```

Status Codes

- 200 OK – magazine 1234 replaced

DELETE /magazines

Delete all magazines from the /magazines resource

```
DELETE /magazines HTTP/1.1
Host: example.gov.au
Accept: application/json, text/javascript
```

200 indicates that the request has succeeded.

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: text/javascript

{ "id": "1234" }
```

Status Codes

- 200 OK – all magazines deleted

DELETE /magazines/1234

Delete the magazine resource /magazines/1234

```
DELETE /magazines/1234 HTTP/1.1
Host: example.gov.au
Accept: application/json, text/javascript
```

200 indicates that the request has succeeded.

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: text/javascript
```

Status Codes

- 200 OK – magazine 1234 deleted

HEAD /magazines

List metadata about the /magazines resource, such as last-modified-date.

```
HEAD /magazines HTTP/1.1
Host: example.gov.au
Accept: application/json, text/javascript
```

200 indicates that the request has succeeded.

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: text/javascript

{...}
```

Status Codes

- 200 OK – magazines metadata found

HEAD /magazines/1234

List metadata about /magazines/1234, such as last-modified-date.

```
HEAD /magazines HTTP/1.1
Host: example.gov.au
Accept: application/json, text/javascript
```

200 indicates that the request has succeeded.

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: text/javascript

{...}
```

Status Codes

- 200 OK – metadata about magazine 1234 found

HTTP response principles

No values in keys – for example, {“125”: “Environment”} is bad, {“id”: “125”, “name”: “Environment”} is good. Note that in the first (bad) example, the key is “125” and the value is “Environment”. This is a problem because the key is supposed to be the name of the value. In the second example (good) the keys are descriptions of their coresponding values.

No internal-specific names (for example, “node” and “taxonomy term”)

Metadata should only contain direct properties of the response set, not properties of the members of the response set

Provide error responses

Error responses should be as descriptive and specific as possible. They should also include a message for the end-user (when appropriate), internal error code (corresponding to some specific internally determined ID) and links where developers can find more information. For example:

```
{
  "developerMessage" : "Verbose, plain language description of the problem.
            Provide developers suggestions about how to solve their problems here
↪",
  "userMessage" : "This is a message that can be passed along to end-users, if␣
↪needed.",
  "errorCode" : "444444",
  "moreInfo" : "http://api.example.gov/v1/documentation/errors/444444.html"
}
```

Use versions

Never release an API without a version number.

Versions should be integers, not decimal numbers, prefixed with ‘v’. For example:

Good: v1, v2, v3

Bad: v-1.1, v1.2, 1.3

Maintain APIs at least one version back.

Batching results

If no limit is specified, return results with a default limit.

To get records 51 through 75 do this:

```
GET /magazines?limit=25&offset=50 HTTP/1.1
Host: example.gov.au
Accept: application/json, text/javascript
```

offset=50 means, ‘skip the first 50 records’

limit=25 means, ‘return a maximum of 25 records’

Information about record limits and total available count should also be included in the response. Example:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: text/javascript

{
  "metadata": {
    "resultset": {
      "count": 227,
      "offset": 25,
      "limit": 25
    }
  },
  "results": [...]
}
```

Mock behaviour

It is suggested that each resource accept a 'mock' parameter on the testing server. Passing this parameter should return a mock data response (bypassing the back-end).

Implementing this feature early in development ensures that the API will exhibit consistent behaviour, supporting a test-driven development methodology.

Mock features compliment interface specification documents, facilitating development of applications that reuse an API. When supplying client libraries or reference implementations for use with an API, mocking features can be implemented at that layer as an alternative to the API itself.

Example 1 - GET mock list of magazines

Request mock list of magazines

```
GET /api/v1/magazines.json?mock=True HTTP/1.1
Host: example.gov.au
Accept: application/json, text/javascript
```

Response list of mock magazines

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: text/javascript

{
  "metadata": {
    "resultset": {
      "count": 123,
      "offset": 0,
      "limit": 10
    }
  },
  "results": [
    {
      "id": "1234",
      "type": "magazine",
      "title": "Public Water Systems",
      "tags": [
        {"id": "125", "name": "Environment", "url": "http://api.example.gov.au/v1/
↪tags/125"},
        {"id": "834", "name": "Water Quality"}
      ],
```

```
      "created": "1231621302"
    },
    {
      "id": 2351,
      "type": "magazine",
      "title": "Public Schools",
      "tags": [
        {"id": "125", "name": "Elementary"},
        {"id": "834", "name": "Charter Schools"}
      ],
      "created": "126251302"
    }
  ]
}
```

Example 2 - GET individual mock magazine

Request mock magazine

```
GET /api/v1/magazines/1234?format=json&mock=True HTTP/1.1
Host: example.gov.au
Accept: application/json, text/javascript
```

Response mock magazine

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: text/javascript

{
  "id": "1234",
  "type": "magazine",
  "title": "Public Water Systems",
  "tags": [
    {"id": "125", "name": "Environment"},
    {"id": "834", "name": "Water Quality"}
  ],
  "created": "1231621302"
}
```

Example 3 - POST article to mock magazine

Post an article to mock magazine 1234

```
POST /api/v1/magazines/1234/articles?mock=True HTTP/1.1
Host: example.gov.au
Accept: application/json, text/javascript

{
  "title": "Raising Revenue",
  "author_first_name": "Jane",
  "author_last_name": "Smith",
  "author_email": "jane.smith@example.gov",
  "date": "2014-06-22",
  "text": "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam eget␣
↪ante ut augue..."
}
```

Note: This method would not result in an article being posted, it is only a simulation.

Support cross-domain mashups with CORS

It is likely that many of your API consumers will want to mashup your API service with services from other agencies or private sector domains using purely client-side applications (for example, mobile apps or single page apps). Agencies should support this model by delivering Cross-Origin Resource Sharing (CORS) enabled services by default.

There's a good description of CORS with examples from Mozilla under 'Overview'

Management state with HATEOAS

"Hypermedia As The Engine Of Application State" (HATEOAS) is a RESTful technique that can make consumer applications simpler and more robust. In many applications, the allowed actions on a resource depend on the state of that resource. Rather than require the consumer to understand and code for the allowed states, HATEOAS provides a means for the server to say what is allowed. The concept is best explained by example.

Consider a bank account number 12345 with a positive balance of $100. A REST query on that resource might return a response indicating that subsequent allowed actions are deposit, withdraw, or transfer:

```
{
  "account_number":"12345"'
  "balance": 100.00,
  "links":[
    {"rel": "deposit",  "href":"/account/12345/deposit"},
    {"rel": "withdraw", "href":"/account/12345/withdraw"},
    {"rel": "transfer", "href":"/account/12345/transfer"}
  ]
}
```

But if the same account is overdrawn by $25 then the only allowed action is deposit:

```
{
  "account_number":"12345"'
  "balance": -25.00,
  "links":[
    {"rel": "deposit",  "href":"/account/12345/deposit"}
  ]
}
```

It is easy to see how many government interactions also have a similar idea of allowed actions depending on state.

Should

Agencies should adopt HATEOAS designs for their REST implementations where practical.

Document your API

Should

All APIs must include documentation targeted at the developer that will consume your API.

The best way to ensure that your API documentation is current and accurate is to embed it within your API implementation and then generate the documentation using literate programming techniques, or a framework such as http://apidocjs.com/, http://swagger.io/, or http://raml.org/index.html.

There are a number of sites offering guidance on providing high quality API documentation:

- http://www.programmableweb.com/news/web-api-documentation-best-practices/2010/08/12
- http://bradfults.com/the-best-api-documentation/

Some good examples of API documentation include:

- https://dev.twitter.com/overview/documentation
- https://stripe.com/docs/api#charges
- https://www.twilio.com/docs/api/rest

Register your API

All APIs must be registered so that they are consistently discoverable. To register your API, send an email to apiregister@dto.gov.au with the following data (sample data provided for data.gov.au API).

```
API Name : data.gov.au query API
API Version : v2
API Protocol : REST
Payload formats: json, xml
API Authentication : none
API Description : An API to query the data.gov.au repository
API Owner Agency : Department of Finance
API Owner contact : data.gov@finance.gov.au
API Documentation URL : http://data.gov.au/api/v2/documentation
API Audience (individual, business, agency, third party) : third party
API tags : data, open, ..
```

The content in this guide has been adapted from: https://github.com/WhiteHouse/api-standards

CHAPTER 5

Secure APIs

TODO

CHAPTER 6

Operate and improve APIs

TODO

RFC 2119 keywords

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.

HTTP Routing Table

/magazines