DPC/S7.3

ACROSS GOVERNMENT POLICY

API Management technical standards

Authority

This policy is issued under the authority of the Chief Technology Officer for the Government of South Australia.

Purpose

This document is intended to provide a minimum set of technical standards for APIs to be followed by government agencies wishing to participate in the MySAGov project and in future in the State's API program.

Uniform implementation of the standards will enable us to

- provide consistent and high quality digital services
- spur Innovation
- minimise duplication
- strengthen Governance
- facilitate quick deployment of APIs
- apply privacy and security standards uniformly and consistently
- focus resources on developing innovative, mission-oriented solutions rather than reinventing the wheel

Conventions

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.

Dates are represented with numbers and can be interpreted in different ways. All dates in this document and in the standard, are to be interpreted as described in ISO 8601.

Background

During 2016, Department of Premier & Cabinet (DPC) embarked on the “MySAGov” project as part of the Government of South Australia’s Digital Transformation journey.

DPC’s Digital Transformation journey is guided by an Enterprise Architecture Roadmap. Central to that Roadmap is the notion of an Application Programming Interface (API) façade. The MySAGov project is the first to rely on this central API platform. APIs provide an efficient and flexible way for sharing agency services and information with the public.

This document provides the minimum set of standards an API should meet in order for it to be exposed through the API façade not only for the MySAGov initiative, but also for the future projects. These standards will improve the usability, quality, accessibility and timeliness of our services and information.

Scope

The scope of this document is:

- South Australian Government Agencies. The API Management is targeted at all state government agencies that provide online services to individuals.
- API Management. The API Management Platform is concerned with publishing of existing and/or new APIs. Individual Agencies are still responsible for the identity management of users of Agency services.
- APIs. The API Management Platform supports online services provided via the APIs.

Policy detail

Security - Secure Connections

Any new API **MUST** secure connections using HTTPS encryption using TLS 1.1 and above to access the API, without exception.

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

Redirects are **NOT RECOMMENDED**. Redirects double up on server traffic and render TLS useless since sensitive data will already have been exposed during the first call.

HTTPS provides:

- **Security** – HTTPS encrypts the request/response which can only be decrypted by the client and the server. It provides protection from sniffing or “Man in the Middle Attack”
- Confidentiality – Information remains confidential as only the client and the server can decrypt the data
- 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 not to be blocked as mixed content, these requests must be over HTTPS to work with APIs.

HTTPS should be configured using modern best practices, including ciphers that support forward secrecy, and HTTP Strict Transport Security.

For an existing API that runs over plain HTTP, the first step is to add HTTPS support, and update the documentation to declare it the default.

Then, evaluate the viability of disabling or redirecting plain HTTP requests.

Security - Authentication and Authorisation

All APIs **MUST** support OAuth 2.0 protocol. OAuth2 provides a standard mechanism for allowing users to authorize 3rd Party applications without the need for those applications to request the user's credentials (username/password) or requiring the user to copy & paste API Keys.

It allows the API provider to revoke tokens for an individual user, for an entire app, without requiring the user to change their original password. This is critical if a mobile device is compromised or if a rogue app is discovered. Above all, OAuth 2.0 will mean improved security and better end-user and consumer experiences with Web and mobile apps.

APIs **SHOULD** use session-based authentication, either by establishing a session token via a POST or by using an API key as a POST body argument or as a cookie.

Usernames, passwords, session tokens, and API keys **SHOULD NOT** appear in the URL, as this can be captured in web server logs, which makes them intrinsically valuable.

Examples:

Good:

```
https://example.com/resourceCollection/<id>/action
https://twitter.com/david/lists
```

Bad:

```
https://example.com/controller/<id>/action?apiKey=d456feb431cde12
```

(API Key in URL)

```
http://example.com/controller/<id>/action?apiKey= d456feb431cde12
```

(transaction not protected by TLS; API Key in URL)

Security - API Keys

API keys **SHOULD** never be treated as secret.

API uses the concept of application identity called API key. This key is replicated across every instance of an application.

Security - Validate Inputs

All APIs **MUST** validate inputs against a strict schema.

It is RECOMMENDED that the schema should be as restrictive as possible, using typing, ranges, sets & explicit white listing whenever possible.

It is NOT RECOMMENDED to use automatically generated schemas as they tend to be too broad.

All APIs which deal with file transfers SHOULD decode attachments by using server grade virus scanning before persisting in the file system.

The above helps in minimizing the scope for exploitation through the data sent to an API including URL, query parameters, HTTP Header and/or POST control.

Design - Keep It Simple

One of the objectives of an API strategy is to reach out to as many developers as possible, opening one's system to the Internet. It is therefore critical that the API be self-describing and as simple as possible, so that developers barely need to refer to the documentation.

When designing an API, it is **RECOMMENDED** that the following principles should be kept in mind:

- the API semantics must be intuitive. URI, payload, request or response: a developer should be able to use them without referring to the API documentation
- the terms must be common and concrete, rather than emanate from a functional or technical jargon. Customers, Orders, Addresses, Products are all good examples. There should not be different ways to achieve the same action
- the API is designed for its clients, the developers, and should not be a simple access layer above the domain model. The API must provide simple features that fit developers requirements. A common mistake is to base the design of an API on an existing data model, which is usually too complex
- in the early design phase, focus on the main use-cases and leave exceptional ones for later phases
- prioritise simplicity. It should be easy to guess what an endpoint does by looking at the URL and HTTP verb, without needing to see a query string.

Design - API Endpoints

An "endpoint" is a combination of two things:

The verb (e.g. GET or POST)
The URL path (e.g. /licenses)

Information can be passed to an endpoint in either of two ways:

The URL query string (e.g. ?year=2016)

HTTP headers (e.g. X-Api-Key: my-key)

It is **RECOMMENDED** that endpoint URLs should advertise resources and avoid verbs.

It is **RECOMMENDED** to avoid single-endpoint APIs.

It is **NOT RECOMMENDED** to assign multiple operations into the same endpoint with the same HTTP verb.

Design - Versioning

All APIs **MUST** be versioned.

A version **MUST** be specified with all requests. Default version **SHOULD** be avoided as they are very difficult to change in the future.

It is **RECOMMENDED** that

- specify the version with a 'v' prefix
- specify the version in the URL, if it changes the logic of handling responses
- alternatively specify the version in the headers, with other metadata using the ACCEPT header with a customer content type.
- maintain APIs at least one version back.

Examples:

Good: v1, v2, v3

Bad: v-1.1, v1.2, 1.3

Request and response - Multiple Format Support

An API **MUST** support JSON by default and it is **RECOMMENDED** to support more than one format.

Direct Data Formats are designed to handle data directly between machines. These languages are often called machine readable, as they tend to be dense and compact. This means they are great for machine-machine integration, and/or manipulation with other APIs.

Direct data formats are best used when additional APIs or services require a data stream from your API in order to function. The three most common formats in this category are JSON, XML, and YAML.

The connection to the resources in an API must be rendered in a way that is both usable to the requesting party, and recognizable in a way relevant to the type of data being presented.

This is why data format support choices are so important — an otherwise amazing API, with wonderful architecture, implementation, and marketing strategy will be wasted if the data format support is incorrect. The initial choice of API data format will determine how effective the API is,

affecting use rates, the success of routine or specific calls, and the long-term adoption and retention curve over the duration of its lifecycle

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

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

General JSON guidelines:

- responses should be a JSON object (not an array). Using an array to return results limits the ability to include metadata about results and limits the API's ability to add additional top-level keys in the future.
- don't use unpredictable keys. Parsing a JSON response where keys are unpredictable (e.g. derived from data) is difficult.
- use consistent case for keys. Whether you use under_score or camelCase for your API keys, make sure you are consistent.

Request and response - Keep JSON minified in all responses

All responses **MUST** be for machine to machine interaction and **SHOULD NOT** include extra whitespace which adds needless response size to requests.

It is best to keep JSON responses minified e.g.:

```
{"firstName":"Joe","lastName":"Smith","age":
36,"address":{"streetAddress":"21 2nd
Street","city":"Adelaide","state":"SA","postalCode": "5000"}}
```

Instead of e.g.:

```
{
  "firstName": "Joe",
  "lastName": "Smith",
  "age": 36,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "Adelaide",
    "state": "SA",
    "postalCode": "5000"
  }
}
```

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

Request and response - Pagination

It is **RECOMMENDED** to use pagination to navigate datasets if required, use the method that makes the most sense for the API's data.

- page and per_page – intuitive for many use cases. Links to "page 2" may not always contain the same data.
- offset and limit – this standard comes from the SQL database world and is a good option when you need stable permalinks to result sets.
- since and limit – get everything "since" some ID or timestamp. Useful when it's a priority to let clients efficiently stay "in sync" with data. Generally, requires result set order to be very stable.

A server **MAY** choose to limit the number of resources returned in a response to a subset (“page”) of the whole set available.

A server **MAY** provide links to traverse a paginated data set (“pagination links”).

Pagination links **MUST** appear in the links object that corresponds to a collection. To paginate the primary data, supply pagination links in the top-level links object. To paginate an included collection returned in a compound document, supply pagination links in the corresponding links object.

The following keys **MUST** be used for pagination links:

- first: the first page of data
- last: the last page of data
- prev: the previous page of data
- next: the next page of data

Keys **MUST** either be omitted or have a null value to indicate that a particular link is unavailable.

Request and response - Metadata

It is **RECOMMENDED** to include enough metadata so that clients can calculate how much data there is, and how and whether to fetch the next set of results.

Example of how that might be implemented:

```
{
  "results": [ ... actual results ... ],
  "pagination": {
    "count": 2340,
    "page": 4,
    "per_page": 20
  }
}
```

Request and response - Use UTF-8

All APIs **MUST** use UTF-8.

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`

Request and response - Date Format

All APIs **MUST** use the ISO 8601 date format.

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.

Request and response - Cross Origin Resource Sharing (CORS)

For clients to be able to use an API from inside web browsers, the API **MUST** enable CORS.

Many 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

For the simplest and most common use case, where the entire API should be accessible from inside the browser, enabling CORS is as simple as including this HTTP header in all responses:

Access-Control-Allow-Origin: *

It's supported by every modern browser, and will just work in many JavaScript clients, like jQuery.

For more advanced configuration, see the W3C spec or Mozilla's guide.

Error handling

All errors **MUST** be handled (including otherwise uncaught exceptions) and return a data structure in the same format as the rest of the API.

For example, a JSON API might provide the following when an uncaught exception occurs:

```
{
  "message": "Description of the error.",
  "exception": "[detailed stacktrace]"
}
```

HTTP responses with error details **SHOULD** use a 4XX status code to indicate a client-side failure (such as invalid authorization, or an invalid parameter), and a 5XX status code to indicate server-side failure (such as an uncaught exception).

Error handling - Processing Errors

A server **MAY** choose to stop processing as soon as a problem is encountered, or it **MAY** continue processing and encounter multiple problems. For instance, a server might process multiple attributes and then return multiple validation problems in a single response.

When a server encounters multiple problems for a request, the most generally applicable HTTP error code **SHOULD** be used in the response. For instance, 400 Bad Request might be appropriate for multiple 4xx errors or 500 Internal Server Error might be appropriate for multiple 5xx errors.

Error handling - Error Objects

Error objects provide additional information about problems encountered while performing an operation.

An error object **MUST** contain the following members:

`status`	the HTTP status code applicable to this problem, expressed as a string value.
`code`	an application-specific error code, expressed as a string value.
`message`	a short, human-readable summary of the problem that SHOULD NOT change from occurrence to occurrence of the problem, except for purposes of localization.
`detail`	a human-readable explanation specific to this occurrence of the problem. Like title, this field's value can be localized.

An error object **MAY** have the following members:

`id`	a unique identifier for this particular occurrence of the problem.
`links`	a links object containing the following members:
`about`	a link that leads to further details about this particular occurrence of the problem.

`source`	an object containing references to the source of the error, optionally including any of the following members:
`pointer`	a JSON Pointer [RFC6901] to the associated entity in the request document [e.g."/data" for a primary data object, or "/data/attributes/title" for a specific attribute].
`parameter`	a string indicating which URI query parameter caused the error.
`meta`	a meta object containing non-standard meta-information about the error.

Error handling - Documentation

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

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

- authentication, including acquiring and using authentication tokens
- API stability and versioning, including how to select the desired API version
- common request and response headers
- error serialisation format
- examples of using the API with clients in different languages.

Abbreviations

API	Application Programming Interface
REST	Representational State Transfer
JSON	JavaScript Object Notation
HTTP	Hyper Text Transfer Protocol
CORS	Cross-Origin Resource Sharing
URL	Uniform Resource Locator
SOAP	Simple Object Access Protocol
SAML	Security Assertion Markup Language
XML	Extensible Markup Language
HTML	Hyper-Text Markup Language

Related documents

- DPC Enterprise Architecture Roadmap

References

- Digital Government: Building a 21st Century Platform to Better Serve the American People, White House
- Application Programming Interfaces (APIs), Digital Transformation Agency, Australian Government
- API Design Guide, Australian Government
- 18F API Standards, 18F
- White House API Standards, White House
- Five Simple Strategies for Securing APIs, Scott Morrison, CA Technologies
- The Definitive Guide to API Management, Apigee

Document Control

ID	DPC/S7.3
Version	1.0
Classification/DLM	Public-I1-A1
Compliance	Mandatory
Original authorisation date	January 2017
Last approval date	February 2019
Next review date	February 2021