Australian Government
Department of Human Services

RESTful Services Standards

Digital Services Branch

myGov REST API Profile - External

Produced by: Citizen Access
Last Edited: 12/08/2019
Version: v1.0 - Final
Classification: OFFICIAL

humanservices.gov.au

Table of Contents

STAKEHOLDER LIST

Name	Role	Team/Branch	Reason(s)
Lorraine Hollis	National Manager	Digital Services Branch	Implementation Owner
Marnix Zwankhuizen	Director - myGov and GovPass Delivery	Citizen Access Section	Implementation Owner
Stephen Kirk	Director - Architecture	Citizen Access Section	Implementation Owner

SIGNATORIES LIST

Branch	**Digital Services Branch**
Name	**Lorraine Hollis**
Position	**National Manager**
Signature	

1. EXECUTIVE SUMMARY

1.1. Background

Currently DHS exposes a large number of web services via an external ESB using both SOAP and REST. REST has increasingly become the de facto standard for web services. In 2017, 83% of APIs were REST based compared to 15% SOAP (Cloud Elements – The State of API Integration 2017).

DHS service consumers such as Member Services have indicated that they would like to use a REST API for system-to-system interactions, for example to initiate the unlinking of a customer account in myGov.

One of the biggest problems with using REST for the Enterprise is that it is not a standard (like SOAP), but an architectural style. Because of this it does not have the same degree of prescriptive implementation standards especially in areas such as security. Over recent years this shortcoming has been adequately addressed through solutions such as JSON Web Tokens (JWT), combined with existing patterns like Asymmetric signing, and the use of standardised REST API specifications such as OpenAPI.

As the need for RESTful services will only continue to grow, myGov is defining the use of these technologies into a REST Profile to define the governance of delivering secured RESTful services. This profile may find further use in future as a wider standard in the department, but is currently only aimed at the myGov system.

1.2. Purpose

This document provides a best-practise based specification to enable secure access to myGov services that are exposed via RESTful API methods for both external and internal consumers.

The purpose of this document is to:

1. **Specify the RESTful API messaging semantics:** Concretely define how RESTful API requests should be formatted, and how responses and errors should be returned along with constituent elements such as headers and body.
2. **Specify the RESTful API security protocol**: Specifies how the client should present identity and assertion information, and how client requests should be authenticated and authorised.
3. **Specify the RESTful API object schemas**: Specified the structure of the objects involved in the use of RESTful services such as JWT, JSON Error Objects etc.

1.3. The Role of myGov

myGov is a very large consumer and provider of services at DHS and as such has a large role to play in the concepts described in this document. Examples and explanations may be myGov specific but could be expanded on over time to include examples from other domains.

myGov also has complex requirements with regards to securing its services. System-to-system interactions are required and catered for as an “Internal Profile” set, however most customer interactions occur within a myGov context, using OAuth/OIDC and the role of the Access Token may be significant in terms of authorisation required to call methods on a RESTful endpoint as part of an “External Profile” set. The interaction of OIDC is covered in section 5.5 OIDC Context.

1.4. Scope

The scope of this document is to describe RESTful integration semantics for the following scenarios:

1. **Consumer to Government APIs**: This category describes situations where an individual customer of myGov makes API calls either on their own behalf or as a delegate. This typically involves a rich-web or native client, and possibly Vendor software, making requests over the internet.
2. **Government to Government APIs**: This category describes situations where a trusted Australian Government Agency system makes API calls to myGov.

1.5. Out of Scope

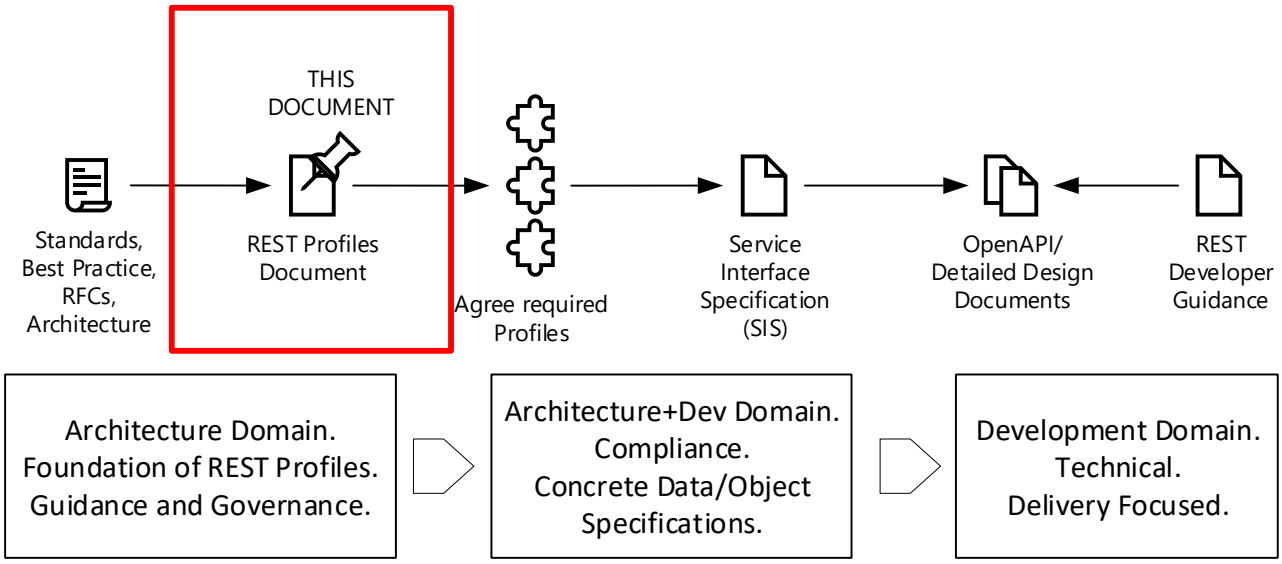
This document does not address the following aspects of RESTful APIs:

- Business Logic
- Decision to implement REST (vs SOAP)
- Network Design
- Implementation Details
- JWT/JWKS Provisioning

1.6. Usage

This document should serve as a base messaging specification for RESTful APIs fitting the scope and purpose defined above, and as a guide to help understand when and how RESTful APIs should be developed.

The REST Profiles document is shown below in the context of other documentation. As defined in the "out of scope" above, this document does not provide detail for any specific implementation. Subsequent to this document being absorbed and understood by a service creator, agreement on the required profiles is needed along with a Service Interface Specification, which **MUST** outline the applicable profiles and justification for selection along with specific details about the service. After this, one or more OpenAPI documents and detailed design document(s) may be created. It is recommended that development teams have a REST Developer Guidance document that is crafted around the requirements of the profiles, technologies and implementations considerations contained in this document.



As stated earlier, it is important to note that REST is an architecture, and not a specification like SOAP. Therefore, the utility in this specification is in concretely defining how request, responses and errors should be formatted such that clients and providers can create systems and services using REST in a consistent manner, across projects and time.

1.7. Status

This document is at Final stage.

2. KEY FEATURES OF SPECIFICATION

The key features of the REST API Profile are:

1. **Best-practise based**: Use of industry best practises for messaging and security, including adoption of existing open standards for REST APIs.
2. **Interoperability**: The profiles are designed to be interoperable across different technology platforms, however in this document it will be constrained to HTTP as there are no requirements to support any other protocols at the moment.
3. **Support for non-functional requirements**: Support (not implementation) is built in via common headers for requirements such as logging, helpdesk, rate-limiting, audit and so on.

2.1. Notational Conventions

The keywords "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in RFC2119.

2.2. Terminology / Acronyms

In the text the use of square brackets denotes a reference e.g. [REFERENCE]. See Appendix C – References for more information.

Name	Description
HTTP	Hyper Text Transfer Protocol
JSON	JavaScript Object Notation
JWA	JSON Web Algorithms
JWE	JSON Web Encryption
JWK	JSON Web Key
JWKS	JSON Web Key Store
JWS	JSON Web Signature
JWT	JSON Web Token
OAUTH	Open Standard for Authorisation. OAuth is an Authorisation Framework that works by means of tokens to provide authorisation after authentication has occurred.
OIDC	OpenID Connect. OIDC is an identity layer built on top of OAuth 2.0.
OpenAPI	OpenAPI is a specification for machine-readable interface files for describing, producing, consuming, and visualizing RESTful web services.
OpenID	OpenID is an open standard and decentralized authentication protocol.
REST	Representational State Transfer
SAML	Security Assertions Mark-up Language
SOAP	Simple Object Access Protocol. A protocol for using XML-based messages to exchange structured data in a distributed computing environment.
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
UUID	Universally Unique Identifier
XML	Extensible Mark-up Language

3. PROFILE OVERVIEW

REST is a widely-adopted architecture that defines how clients should consume resources of a server, typically using HTTP, in a decoupled, distributed fashion. However, there is no standardised REST specification that spells out precisely how to format requests, and so on. As a result, there are a variety of implementations. This profile seeks to specify the standard for REST services.

3.1. OpenAPI

OpenAPI [OPENAPI] is the most widely adopted specification for describing RESTful web services in terms of interfaces, schemas, security and so on. The specification was called “Swagger” up until its second release (2.0) when it moved under new sponsorship and formally became OpenAPI. The current major version of OpenAPI is 3.0.0.

To avoid confusion and maintain forwards compatibility, this document will use the term “OpenAPI” to refer to whatever is the latest released specification.

All new RESTful web services **MUST** use OpenAPI to describe their interfaces and **MUST** use the 3.0 or higher specification.

The OpenAPI specification is not prescriptive, however this document will prescribe how the elements of the specification are to be applied in line with 2.1 Notational Conventions.

3.2. Compliance

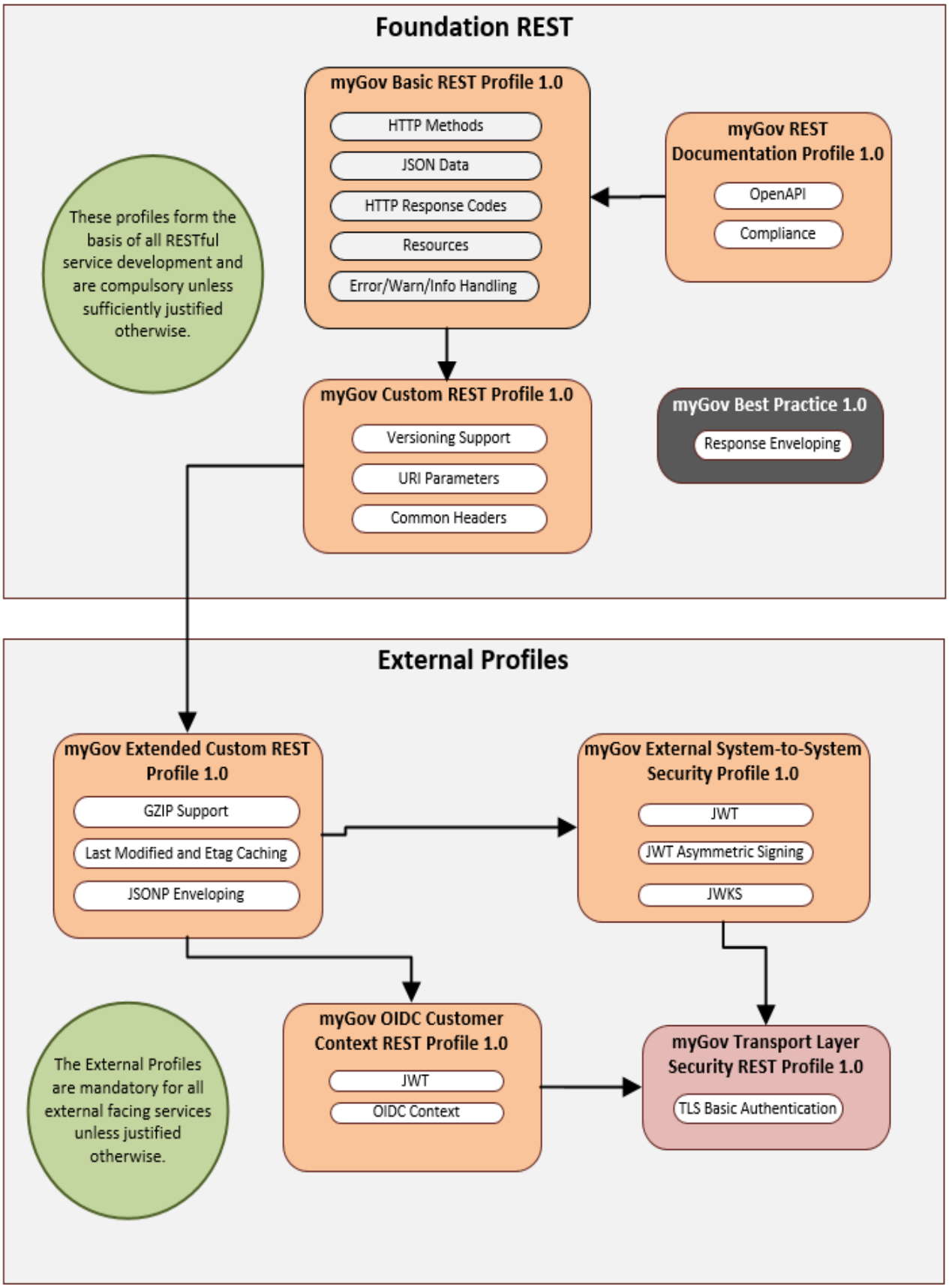
Services **MUST** state their agreed applicable profiles and the degree of profile compliance, noting any exemptions and/or deviations which must be agreed by all partied involved in both providing and consuming the service. These compliance statements must form part of the SIS (System Interface Specification). A compliance table for use in a SIS can be found in Appendix E – Compliance Statement.

The profiles used by a service may vary depending on the context of the service process workflow, for example an external service may comply fully with the “JWT Asymmetric Signing” profile but internally may only comply with the “Unsecured JWT” profile.

3.3. Profiles

This Profile is a set of guidelines defining the use of REST service specifications beyond the core principles. These guidelines are necessary because as already stated, REST is merely a general-purpose approach and is not enough by itself to satisfy enterprise level requirements. These Profiles also aim to resolve ambiguities in areas where REST is not clear enough to ensure that all implementations process REST messages in the same way.

3.3.1. Profiles Overview



3.3.2. Actors

The actors used in this document are:

Actor	Description
Service Consumer	Also referred to as "consumer" or "the consumer". This is the client of the service or resource at runtime. This is expected to be the software component that creates *request* messages.
Service Provider	Also referred to as "service", "server", "the service" or "the server". This is the actor that provides, at runtime, a service or resource for consumption. This actor may be composed of different layers – a security layer, a business layer, and so on. The Service Provider creates *response* messages.

3.3.3. myGov REST Documentation Profile 1.0

The "myGov REST Documentation Profile 1.0" is a profile designed to ensure that any service interface is properly documented. This profile **MUST** be used as part of writing an SIS (System Interface Specification).

Section Ref	Standard	Description
3.1	OpenAPI	The use of OpenAPI to specify the interface.
3.2	Compliance	The use of JSON for request and response elements.

3.3.4. myGov Basic REST Profile 1.0

The "myGov Basic REST Profile 1.0" is a set of basic standards needed for every RESTful service transaction. This profile includes all of the standards from the "myGov REST Documentation Profile 1.0" listed above, and additionally mandates the use of the following standards references.

This profile **MUST** be used for all RESTful services and for all consumers and providers. For intra-system RESTful services this is the only mandatory profile.

Section Ref	Standard	Description
4.1	HTTP Methods	The use of the particular HTTP verb (GET, POST, PUT, DELETE, PATCH) that best matches the intention of the call.
4.2	JSON Data	The use of JSON for request and response elements.
4.3	HTTP Response Codes	The use of HTTP response codes (200 OK, 404 Not Found, etc.) that best match the nature of the result.
4.6	Resources	Resources in RESTful URIs should be used to represent data elements using Nouns for Objects and Verbs for Controllers.
4.7	Error, Warning and Informational Handling	The provider has to handle all errors, warning and informational data with a standard payload structure.

3.3.5. myGov Custom REST Profile 1.0

The "myGov Custom REST Profile 1.0" is a set of standards that **MUST** be used for every RESTful service transaction that will operate across systems (inter-system) whether external or internal. This profile includes all of the standards from the "myGov Basic REST Profile 1.0" listed above, and additionally mandates the use of the following standards references.

Section Ref	Standard	Description
4.4	Versioning Support	The use of client-visible versioning of the resource/service being consumed.
4.5	URI Parameters	The use of client-specified parameters, through URI paths and query parameters, to signal to the server how to process the request. For example, GET /mygov/coredata/AA123456/?fields=links,firstName,lastName would retrieve the links, First Name and Last Name for a myGov coredata related to myGov id AA123456.
4.8	Common Headers	This allows for the implementation of a variety of functional and non-functional requirements by providing appropriate request metadata.

3.3.6. myGov Extended Custom REST Profile 1.0

The "myGov Extended Custom REST Profile 1.0" profile is a set of best practices standards needed for RESTful service transactions. This profile includes all of the standards from the "myGov Custom REST Profile 1.0" listed above, and additionally mandates the use of the following standards references. This profile **MUST** be used for any external facing services, whether or not the consumer is a backend system or a user agent.

Section	Standard	Description
4.9	GZIP Support	Where the client indicates that they can accept gzip encoding for responses, the server should comply.
4.10	Last Modified and ETag Caching	Where applicable and when the client specifies an ETag or Last Modified value, the server should honour the request in a cache-aware manner.
4.11	JSONP Enveloping	The enveloping of JSON responses. This also covers the JSONP standard.

3.3.7. myGov Transport Layer Security REST Profile 1.0

The "myGov Transport Layer Security REST Profile 1.0" describes the standards needed to secure the transaction at the transport layer and this profile mandates the use of the following standards. This profile **MUST** be used for all external facing services, whether or not the consumer is a backend system or a user agent, and is recommended for all internal services also.

Section	Standard	Description
5.1	TLS Basic Authentication	This profile mandates the use of TLS for transport layer security. I.e. the use of HTTPS instead of HTTP.

3.3.8. myGov External System-to-System Security Profile 1.0

The "myGov External System-to-System Security Profile 1.0**"** includes those standards from "myGov Extended Custom REST Profile 1.0", "myGov Transport Layer Security REST Profile 1.0**"** and additionally mandates the use of the following standards references. Services provided by either external parties or internally and consumed by the other party **MUST** use this security profile.

The on the wire communication between the two parties will be protected by HTTPS / TLS encryption and hence the "myGov Transport Layer Security REST Profile 1.0" in this profile is a **MUST**.

Even though the profile is based on a trust between two parties, signing of the token is a **MUST** as it will provide security for man in the middle attack scenarios as well as provide authentication. The JWT signing will be done with the provider's X.509 certificate with the public key exposed via a JWKS endpoint.

The JWT **MUST** carry all the context information that is needed to establish and identify the caller and the transaction subject.

Section	Standard	Description
5.2	JWT	This profile mandates the use of a JWT to carry the payload of data.
5.3	JWT Asymmetric Signing	The JWT must be signed using an Asymmetric algorithm.
5.4	JWKS	A JWKS endpoint must be used to provide the public key used in signing.

3.3.9. myGov OIDC Customer Context REST Profile 1.0

The "myGov OIDC Customer Context REST Profile 1.0" includes those standards from "myGov Extended Custom REST Profile 1.0", "myGov Transport Layer Security REST Profile 1.0" and additionally may use the standards from "myGov External System-to-System Security Profile 1.0". It also mandates the use of the following standards references.

This profile **MUST** be used by external clients accessing REST APIs where a customer context is already established via the OIDC pattern used in sites such as myGov, and targets customers intending to consume services over the internet using the tokens created as part their login.

Even though customer context is established by successfully authenticating the user session, OIDC token(s) to carry complete customer context information is a **MUST**, as it is central to authorising user-to-system scenarios where a user context is required. For example only an end user should request access to their inbox messages and would need this profile, otherwise it would be possible for a pure system-to-system transaction to read these messages.

Possible Use Cases:

1) Member Service requesting myGov Inbox messages for a Customer.
2) Member Service requesting customer profile information after customer successfully logged in.

Section	Standard	Description
5.2	JWT	This profile mandates the use of a JWT to carry the payload of data.
5.5	OIDC Context	The inclusion of this profile means that the request must have suitable Tokens (e.g. Access Token, ID Token).

3.4. Best Practices

This set of "Best Practices" is not a standard but is a preferred approach for implementation.

3.4.1. Response Enveloping 1.0

This practice **SHOULD** be implemented by all REST services and it is critical that if used it **MUST** be clearly documented in the interface specification(s).

Section	Best Practice Name	Description
4.12	Response Enveloping	Response body must be enveloped with the "meta", "data" and "info" objects.

4. MESSAGING PROFILES

What follows are best practises about how messages should be governed between service providers and service consumers.

4.1. HTTP Methods

4.1.1. Principle

Note that HTTP Methods are often called “verbs” when referring specifically to RESTful implementations and in this context they are used synonymously. This document uses the technically correct term “method” to avoid any potential ambiguity.

The server **MUST** respond to requests for resources and services according to the most relevant HTTP method and obey the rules of that method. Note that the use of a method may have a different implementation and response depending if the operation is on a collection or individual item (see 4.6 Resource).

Services **MUST NOT** implement actual CRUD methods (e.g. “/api/deleteRecord”), and should instead rely on correctly designed HTTP Methods.

4.1.2. Explanation

There are a number of standard HTTP methods most relevant to RESTful APIs – GET, POST, PUT, DELETE and PATCH – and each of them has a different intent [IETF-RFC7231]. By conforming to the intent as closely as possible, the intention of the request will be clear from the verb alone. In addition, certain HTTP methods enable platform-level capabilities, such as GET allowing cache use.

Idempotence is a key term when dealing with RESTful services. The technical definition is “unchanged when multiplied by itself” however it is used in RESTful context to mean that repeating the same call will produce the same result from the perspective of the client. E.g. a GET call to a resource should always return that same resource (even if the resource attributes have changed it’s still the same resource), but a POST will always create something new and so is not idempotent.

Idempotence **MUST** be implemented as described below.

Safety is another key consideration for RESTful services. A method is *safe* if it does not change the state on the server at the request of the client (i.e. the client has not requested a change and cannot be held responsible for one). This is critically important with methods such as PATCH, where collisions must be handled appropriately (see below).

Safety **MUST** be implemented as described below.

VERB	CRUD Advice	Usage	Idempotent	Safe
GET	Read	Retrieve an individual object or collection of objects from a resource or service.	Yes	Yes
POST	Create	Create an individual subordinate object.	No	No
PUT	Update→Replace	Update an individual subordinate object by replacing it.	Yes	No

VERB	CRUD Advice	Usage	Idempotent	Safe
DELETE	Delete	Deletes an individual object from a resource or service.	Yes[1]	No[2]
PATCH	Update→Modify	Update an individual subordinate object by modifying one or more properties. PATCH requests	No[3]	No

Collision handling refers to handling situations where multiple conflicting requests are made for the same resource or service where the server state is intended to be changed.

Collision handling **MUST** be implemented and documented in the interface specification as described below.

VERB	Collision Handling
POST	Where it is possible to identify that the new object is identical to a previous request, and the service does not support identical object insertion, the service **MUST** respond with an appropriate non-2xx status for example 422 (Unprocessable Entity). Where it is not possible to identify that the new object is identical, the system **MUST** process the request as normal.
PUT	When a PUT is received from a client that is potentially not synchronised (i.e. expects the server to be in a different state than it is actually in), the requests **SHOULD** be processed in First-In-First-Out (FIFO) order. Where an ETag is available (e.g. from the previous GET request), it **MAY** be provided as part of the PUT request, and **MUST** be rejected if it is provided and the ETag does not match with the current state of the object. If an ETag is not provided as part of the PUT request, then FIFO processing does not guarantee that an update collision will not occur and the interface specification **SHOULD** inform the consumer that a subsequent GET request should be made to verify the update.
PATCH	PATCH should be handled in the same way as PUT, described above.

4.1.3. Example

Retrieve all inbox elements for a given myGov User.

```
GET /myGov/inbox
```

Retrieve myGov inbox message details with message id ‘123’.

```
GET /myGov/inbox/123
```

Create a new empty myGov inbox message.

```
POST /myGov/inbox
```

[1] DELETE is technically idempotent, however calling this method more than once will return a different status code (e.g. 404) since the item was already removed.

[2] DELETE is technically not safe as it changes server state, however there is no need to consider collision handling as they are not conflicting requests.

[3] PATCH requests can technically be made idempotent by updating the entire set of properties however this should never be done and a PUT should be used in such instances.

Update all details associated with myGov inbox message 123.

```
PUT /myGov/inbox/123
```

Update specific details associate with myGov inbox message '123'card 4247588971, leaving all other details unchanged.

```
PATCH /myGov/inbox/123
```

Delete myGov inbox message with id '123'

```
DELETE /myGov/inbox/123
```

4.1.4. Caveat

One exception to this rule is where legacy software platforms that only support GET and POST method are explicitly targeted. In such scenarios, the best practise is to use GET as above, and POST otherwise (with http header HTTP-Method-Override supplied to override POST with a replacement verb specified in the value field).

4.2. JSON Data

4.2.1. Principle

Offer REST APIs using JSON content only, by default. This includes the use of JSON specifications such as JSON Web Tokens (JWT).

4.2.2. Explanation

JSON has better support on a number of consumer-centric platforms, including web browsers. As a result, JSON should be the primary and only content type used for client requests and server responses in the REST profile.

The correct Content-Type from JSON requests and responses that **MUST** be used is: application/json

4.2.3. Example

```
GET /mygov/coredata
```

RETURN BODY (formatted for readability):

```
{
    "name":"John",
    "age":30,
    "DOB":"10/10/1984"
}
```

4.2.4. Caveat

None.

4.3. HTTP Response Codes

4.3.1. Principle

Use the HTTP response code as the primary mechanism for providing a client feedback. Use the HTTP response code as closely to its specified use as is reasonable.

4.3.2. Explanation

A variety of HTTP response codes have been specified, and each has a precise meaning. The use of these HTTP response codes means that the server does not have to specify proprietary means of conveying these common situations.

A full list of common HTTP response codes can be found in Appendix A – HTTP RESPONSE CODES. It is not an exhaustive list – business applications may introduce HTTP response codes beyond the scope of this generic set as long as they are within industry standards as agreed by architectural review.

The table below provides advice on implementing status codes for various methods across both collection and individual requests. Services **SHOULD** use "Common Responses" where possible and **SHOULD NOT** use "Invalid Responses" without providing an explanation as part of the service documentation. This table is not comprehensive, and other response codes are also acceptable with sufficient documented justification.

Key ☑ = Common Responses ✓ = Valid Responses × = Invalid Responses		Success				Client Error									Server Error		
		200 (OK)	201 (Created)	202 (Accepted)	204 (No Content)	400 (Bad Request)	401 (Unauthorized)	403 (Forbidden)	404 (Not Found)	405 (Method Not Allowed)	406 (Not Acceptable)	408 (Request Timeout)	409 (Conflict)	410 (Gone)	500 (Internal Server Error)	501 (Not Implemented)	503 (Service Unavailable)
GET	Collection	☑	×	×	×	✓	✓	✓	☑	✓	✓	✓	×	✓	✓	✓	✓
GET	Individual	☑	×	×	×	✓	✓	✓	☑	✓	✓	✓	×	✓	✓	✓	✓
POST	Collection	✓	☑	✓	✓	✓	✓	✓	☑	☑	✓	✓	☑	✓	✓	✓	✓
POST	Individual	✓	☑	✓	✓	✓	✓	✓	☑	✓	✓	✓	☑	✓	✓	✓	✓
PUT	Collection	☑	×	✓	✓	✓	✓	✓	☑	☑	✓	✓	✓	✓	✓	✓	✓
PUT	Individual	☑	×	✓	☑	✓	✓	✓	☑	✓	✓	✓	✓	✓	✓	✓	✓
DELETE	Collection	✓	×	✓	☑	✓	✓	✓	☑	☑	✓	✓	×	✓	✓	✓	✓
DELETE	Individual	✓	×	✓	☑	✓	✓	✓	☑	✓	✓	✓	×	✓	✓	✓	✓
PATCH	Collection	✓	×	✓	☑	✓	✓	✓	☑	☑	✓	✓	✓	✓	✓	✓	✓
PATCH	Individual	✓	×	✓	☑	✓	✓	✓	☑	✓	✓	✓	✓	✓	✓	✓	✓

Content can be used in combination with a HTTP response code to provide supporting or clarifying information.

4.3.3. Example

Call	Scenario	HTTP Status Code
GET /myGov/coredata	Security token invalid.	403 Forbidden
DELETE /myGov/coredata	Successful call.	204 No Content
GET /myGov/coredata	Internal server error occurs.	500 Internal Server Error
PATCH /myGov/account/123	Item has changed since client retrieved data (determined by ETag), so will not be patched.	409 Conflict

4.3.4. Caveat

None.

4.4. Versioning Support

4.4.1. Principle

The service **MUST** explicitly support versions through the URL mechanism, and versions **MUST** be classified and documented as either backwards compatible or incompatible changes. The use of a "major" version is a **MUST**, and an additional "minor" version **MAY** be used if justified. For external services only a major version **SHOULD** be used where appropriate so as to avoid external consumers having to update their code base due to backwards compatible changes.

4.4.2. Explanation

As functionality is added and modified within an API suite, it is crucial that versions be identified and managed to ensure seamless operation over time, and reduce maintenance support for legacy version.

A new version (e.g. 2.0) of an API is considered backwards compatible if a client using an existing version (e.g. 1.0) could utilise version 2.0 simply through changing the version parameter in the URL. Note that individual operations or resources within a suite are not versioned – rather, an API suite is versioned as a whole.

Backwards compatible changes typically fit into the following categories:

1. New operations are added to an API suite
2. New response data is added to existing operations
3. New optional request parameters or data is added to existing operations

Any other change is likely to be backwards incompatible. Examples include:

1. Removing or modifying existing request or response data attributes
2. Removing existing data objects or attributes
3. New mandatory request parameters or data is added to existing operations

A major version is considered backwards incompatible with all earlier versions.
A minor version is considered backwards compatible within other versions of the same major version.
For example 2.1 is backwards compatible with 2.0, but not with 1.6.

How many concurrent versions are supported is a business decision, made on an API suite basis.

The mechanism through which the client specifies the version they wish to invoke is the URL pattern below, where X is the major version, and Y is the minor version.

```
GET /apisuite/X.Y
```

For externally facing services, the major version only **SHOULD** be used unless there is good reason to use a minor version also, e.g.

```
GET /apisuite/X
```

It is common to use a "v" prefix to denote that part of the path which holds the version, e.g.

```
GET /apisuite/v1
```

4.4.3. Example

Consider an API suite – myGovAPI - with two resources – coredata and inbox.

Build 1 – 1.0 – Original

```
GET /mygovapi/v1/coredata
GET /mygovapi/v1/inbox
```

Build 2 – 2.0 – Data representation changes

```
GET /mygovapi/v2/coredata
GET /mygovapi/v2/inbox
```

4.4.4. Caveat

None.

4.5. URI Parameters

4.5.1. Principle

Use URI parameters to allow the client to specify how processing should be done – including filtering and sorting.

4.5.2. Explanation

Allowing a client to specify filter, search and sorting options can allow the client to integrate more easily, and the server to perform less work – a win-win situation.

Sensitive data such as User Identifiers or Passwords **MUST NOT** be placed in the URI. The information should be passed as part of a signed JWT token.

Selection – Selection allows a client to specify a specific object and is essential for most methods. Selection **MUST** be implemented by using the final URI path.

Filtering – Filtering allows a client to specify criteria that a result object must meet in order to be returned. This can be very helpful when a request without a filter would result in a large number of objects being returned. The way to implement a filter **MUST** be via a query parameter.

Sorting – Sorting allows a client to specify which results they are most interested in, such that top to bottom processing follows a client specified priority. Sorting should be via a query parameter called *sort*, whose value is a list of properties on which to sort. The default sort order is suite specific, but **SHOULD** be ascending.

Searching – By default, only implement search (string searching) when the client could significantly benefit.

4.5.3. Example

Show all links for the current myGov account in ascending “member service” order

```
GET /mygovapi/1.0/agencyinfo?sort=memberservice
```

Select a specific inbox message in myGov

```
GET /mygovapi/1.0/inbox/123
```

Delete an account (unlink) with link id 123 for a member service abcd

```
DELETE /mygovapi/1.0/clients/abcd/accounts/links/123
```

4.5.4. Caveat

None.

4.6. Resources

4.6.1. Principle

Resources in RESTful URIs **SHOULD** be used to represent data elements using Nouns for Items and Collections (objects), and Verbs for Controllers (functions).

4.6.2. Explanation

Resources and Controllers are fundamental to REST and API managed services, and are used to represent the objects and actions exposed by the service. Resources may be any of the following archetypes:

- Item – Represents a singular instance of an object or record.
- Server Collection – Represents a collection of resources that are governed by the server.
- Client Collection – Represents a collection of resources that are governed by the client.
- Controller – Represents a procedure or function.

Note that "Controller" is often considered separate from a "Resource" although technically it is a sub-type.

A generic Resource URI for HTTPS looks like the following and the structure is broken down below where angled brackets donate a variable portion of the URI, and square brackets denote a portion of the URI which may repeat several times with differing parameters.

```
https://<authority>/[<path>/{id}]
```

Where resources have a parent/child or hierarchical relationship then the constituent <path> section should clearly infer that relationship by separating the relationship with the forward slash such as /clients/abcd/accounts which clearly infers that accounts belong to an instance of a client.

Rule Name	Implication
Nouns	If the resource is a collection, a plural noun **SHOULD** be used, otherwise singular.
Verbs	For controllers, verbs **SHOULD** be used.
No Trailing Slash	The final character in a URI path **MUST NOT** be a forward slash.
Hyphenation	Hyphens **SHOULD** be used in place of spaces for readability improvements.
No Underscores	Underscores **SHOULD NOT** be used in a URI, consider using hyphens instead.
Lowercase	Lowercase letters **SHOULD** be used for the entire URI path.
No File Extensions	File extensions **SHOULD NOT** be used, consider media types (Content-Type header).

4.6.3. Example

A collection of inbox messages

```
/messages
```

A resource with information about a specific message with ID 123

```
/messages/123
```

A specific account with ID 123 belonging to a customer of “abcd” of client “qwerty”

```
/clients/qwerty/customers/abcd/accounts/123
```

The singular mother of specific person with ID 123

```
/people/123/mother
```

A controller to search for accounts with the name “John”

```
/clients/qwerty/customers/findByName/John
```

4.6.4. Caveat

None.

4.7. Error, Warning and Informational Handling

4.7.1. Principle

The server **MUST** handle all errors, warnings and informational messages with a standard payload structure defined in section “7 - Appendix B – Additional Response Information Specification“.

4.7.2. Explanation

A response code other than in the 2xx (success) range **MUST** return additional error information in the form of a JSON document in the body as specified below. Warnings and information messages **MAY** also be returned in a similar fashion however the response code for these can be any valid code.

The information returned **MAY** be singular object or an array of objects (for example, when multiple errors have occurred).

By using a consistent standard for such messages, operational management and incident/problem management can be improved and consumers can integrate much easier when integrating with multiple compliant services.

JSON Error/Warn/Info Body Object			
Attribute	**Mandatory?**	**Format[length]**	**Description**
status	No	integer[3]	The HTTP status code of the response. Whilst this is also sent in the header, recording it here allows the full message to be logged in one place.
code	Yes	string[15]	A combination of the originator system code, error/warn/info code, severity and layer in the format SYSCODE CODE SEVERITY LAYER (with no spaces). For example MGV0001ESE identified an issue from myGov (MGV), code 0001 (documented in interface specification), Error (E) which occurred at the Security Layer (SE). SYSCODE is the system identifier, up to 8 characters. CODE is the identifying code, unique to the originating system, which should be clearly documented in the interface specification (SIS). SEVERITY is one of E=Error, W=Warning or I=Informational.

JSON Error/Warn/Info Body Object			
			LAYER tells the client where the error occurred. BU=Business Layer (e.g. business rules), IN=Infrastructure Layer (e.g. network or ESB) and SE=Security Layer (e.g. JWT validation). The four digit sub-code (e.g. 0001 from above example) can be used to identify the issue that occurred. The code field can be parsed by clients to determine what action to take and the meaning should be clearly documented.
message	Yes	string[1024]	A string literal that gives a human-understandable explanation of what the issue was. The message string for a given code is not guaranteed to remain consistent, and is prone to change. The message field should not be parsed.
details	No	string [1024]	This optional attribute provides more detailed information about the issue to aid understanding of what has occurred. It may for example include a stack trace (for internal API only).
reference	No	UUID	The message identifier (header "mgv-messageid") this information relates to.
url	No	string [1024]	The url that is related to this message, e.g. the request url or resource url.

All codes and corresponding descriptions **MUST** be documented for consumers.

4.7.3. Example

Formatted for readability. This example includes ALL fields.

```
{
    "status": 400,
    "code": "MGV0001EBU",
    "message": "Validation failed.",
    "details": "There was a validation problem with one or more fields, please refer to
                child errors for more information.",
    "reference": "f4138df5-c6e9-4586-b964-5352c2515d82",
    "url": "https://test.rest.my.gov.au/mygovapi/1.0/accounts/email%40myaddress.gov.au"
}
```

Formatted for readability. This example includes MANDATORY fields only.

```
{
    "code": "MGV0002EBU",
    "message": "An error occurred in processing."
}
```

Formatted for readability. This example includes MANDATORY fields only and returns an array of errors.

```
[{
    "code": "MGV0002EBU",
    "message": "An error occurred in processing."
},
{
    "code": "MGV0034EBU",
```

```
    "message": "Another, different error occurred in processing."
}]
```

Formatted for readability. This example includes a warning message.

```
{
    "code": "MGV0003WBU",
    "message": "The provided data for field X was invalid and has been ignored, record
                creation completed without this data."
}
```

4.7.4. Caveat

None.

4.8. Common Headers

4.8.1. Principle

The client **MUST** provide meta-details about each request as a series of headers to allow the server to correctly process the transaction and support non-functional requirements.

4.8.2. Explanation

A DHS service is required to capture a number of details about the nature of the client and request in order to correctly process the transaction and support the API suite. These details **MUST** be provided by the client as described below.

The mandated format for UUID follows the UUID format specified in [IETF-RFC4122], which most software platforms provide native support for.

Custom Headers			
Name	**Format [max length] / Value**	**Mandatory**	**Description**
mgv-jwt	JWT	Usually Yes. See Description.	When a JWT is required by the chosen profile, it must be passed as a header called "mgv-jwt". Refer to section 5 Security for the information, specifically 5.2 JWT.
mgv-messageid	UUID	Yes	A unique identifier through which the message can be identified.
mgv-relatedmessageid	UUID	No	A unique identifier of a previously exchanged message. This creates a relationship between two messages (e.g. request/response).
mgv-globalsessionkey	UUID	Yes	A unique identifier through which a group of related messages can be logically grouped (a "session"). If there is no session, each request should contain a unique session key.

			This allows a transaction to be tracked and audited as it flows through the systems.
mgv-productid	string[128]	Yes	The value for mgv-productid represents the application or software component that is sending the request. The mgv-productid value to use should be documented and **MUST** be used.
xxx	*Varies*	*Varies*	*Custom headers may be used by specific API Suites and should be documented in their interface specifications.*
Standard HTTP Headers			
Authorization	Varies. See description.	No	The format to be used **MUST** be specified in the relevant API suite documentation. The Authorization header contains the security credential value that the server uses to permit access to the API suite. For example the Authorization value will often be a bearer token if in a user context, or a base64 encoded user name and password for HTTP Basic Authentication. See section 5 Security for more information.

The common HTTP headers allow the following functions to be performed:

1. Replay attack prevention – messageid has to be unique with each request/response.
2. Troubleshooting/auditing of messages – allows for simpler tracking of individual messages and end-to-end transactions occurring within a session. Allows a "picture" of what has occurred through the systems and users involved.
3. Authentication – the signed JWT and/or Authorization header token provide a means of determining that the sender is who they claim to be.
4. Authorisation – with the "Authorization" header information combined with the JWT and/or other headers, most authorisation checks can be done on these values alone.
5. Quick identification - Fast end-user and subject identification allows for improved analytics and ability to troubleshoot. E.g. get all staff who have touched X record in Y system in the last Z weeks.
6. Attribute based message handling – Functions such as throttling and rejection can be based on specific users, products, systems and so on.

4.8.3. Example

The JWT provided in the "mgv-jwt" header is the most complex element and is described here first. Formatted for readability, this is an example JWT (before being signed).
JWT Header:

```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

JWT Body:

```
{
    “iss”: “https://example.gov.au”,
    “sub”: “044240b8-067d-44f1-840f-b8a9898a569d”,
    “iat”: 1456786800,
    “nbf”: 1456786800,
    “exp”: 1457996400,
    “aud”: “https://example.gov.au”,
    “jti”: “81f22e43-075c-4d70-aac5-31dbda62d7c7”,
    “squ”: “XYZ”,
    “sty”: “MBUN”,
    “uid”: “044240b8-067d-44f1-840f-b8a9898a569d”,
    “uty”: “SYSTEM”
}
```

Once signed the JWT becomes:

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJodHRwczovL2V4YW1wbGUuZ292LmF1Iiwic3ViIjoieHl6IiwiaWF0IjoxNDU2Nzg2ODAwLCJuYmYiOjE0NTY3ODY4MDAsImV4cCI6MTQ1Nzk5NjQwMCwiYXVkIjoiaHR0cHM6Ly9leGFtcGxlLmdvdi5hdSIsImp0aSI6IjgxZjIyZTQzLTA3NWMtNGQ3MC1hYWM1LTMxZGJkYTYyZDdjNyJ9.0-rODIeKyVQ477GJ2Fhgzv1s1dKRbnwGT9zsooDVUIOR-XCDSp3ADKn4POhXSPCp_4gxOp6-ouxnVrtkSK6R95t_EqureUdPqVyWKsauVs3PwfmsAEFsFgq3Ud6ONFEYXDrnOonP-ZJsW2m1HSbpr-qP6dCUhGFcfCvHHa8mHxU
```

And therefore the in the header would be passed as: (where <signed JWT> is the encoded block above)

```
GET /mygovapi/1.0/coredata
mgv-jwt: <signed JWT>
mgv-messageid: f4138df5-c6e9-4586-b964-5352c2515d82
mgv-relatedmessageid: f4138df5-c6e9-4586-b964-5352c2515d82
mgv-globalsessionkey: 035ee2fd-b465-4141-bb1c-d5b0bebb5d4b
mgv-productid: ABC
Authorization: Bearer 49fad390491a5b547d0f782309b6a5b33f7ac087
```

4.8.4. Caveat

None.

4.9. GZIP Support

4.9.1. Principle

The server **SHOULD** support the return of compressed text responses through GZIP.

4.9.2. Explanation

Significant performance benefits can be achieved in both XML and JSON scenarios when GZIP compression is used. Most client software platforms support GZIP compression. On-wire compression can save more than 80% bandwidth in some situations, which is particularly important over mobile networks.

In order to use GZIP compression, the client should sent the following HTTP header:

```
Accept-Encoding: gzip
```

4.9.3. Example

Request:

```
GET /mygovapi/inbox/123
Accept-Encoding: gzip
```

Response:

```
200 OK
Content-Encoding: gzip
```

4.9.4. Caveat

None.

4.10. Last Modified and ETag Caching

4.10.1. Principle

The server **SHOULD** support both Last Modified and ETag forms of caching for GET requests.

4.10.2. Explanation

The processing of some responses can be expensive to access from a backend data store and send over the wire. The use of caching mechanisms built into the HTTP protocol can significantly reduce the work performed by the server, as well as transmission times (particularly over mobile networks). The server should always support these two common caching mechanisms where feasible.

The method to generate an ETag is not specified by the HTTP standard, however for DHS services it **MUST** be a hash of the content. The ETag solution involves the client sending a hash of the last representation they received from a GET operation, and the server performing the same hash of the current representation. A client wishing to leverage ETag caching must pass the "If-Non-Match" header, and if the server hash matches the ETag value supplied in that header then the server must return 304 Not Modified. Hash algorithms supported by the server should be documented, and should be CRC32C and MD5 by default.

ETags come in two forms – strong, which relates to identical objects, and weak, which relates to objects being functionally equivalent. The use of strong ETags is preferred, and the use of weak ETags is only appropriate to make on a resource by resource basis and should be clearly documented. A weak ETag begins with the string 'W/' so for example this is a weak ETag:

```
ETag: W/"3764"
```

And this is a strong ETag:

```
ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4"
```

The Last Modified solution involves the client requesting a resource by using the If-Modified-Since HTTP header with the timestamp of the last representation they have (obtained from the previous GET response's Last Modified timestamp). If the server determines the object has not changed since the client timestamp, the server should respond with 304 Not Modified. A number of timestamp formats exist, and each must be supported by the server.

4.10.3. Example

ETag caching:
Request A

```
GET /mygovapi/1.0/coredata
```

Response A

```
200 OK
ETag: "15f0fff99ed5aae4edffdd6496d7131f"
```

Request B

```
GET /mygovapi/1.0/coredata
If-None-Match: "15f0fff99ed5aae4edffdd6496d7131f"
```

Response B - if the hash matches the hash of the current object

```
304 Not Modified
```

Response B - if the hash does NOT match the hash of the current object

```
200 OK
ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4"
```

Last Modified caching:
Request A

```
GET /mygovapi/1.0/coredata
```

Response A

```
200 OK
Last-Modified: Mon, 03 Jan 2011 17:45:57 GMT
```

Request B

```
GET /mygovapi/1.0/coredata
If-Modified-Since: Sat, 03 Nov 2018 17:45:57 GMT
```

Response B - if the object has not more recently been changed on the server

```
304 Not Modified
```

Response B - if the object HAS more recently been changed on the server

```
200 OK
Last-Modified: Sun, 04 Nov 2011 17:45:57 GMT
```

4.10.4. Caveat

None.

4.11. JSONP Enveloping

4.11.1. Principle

By default, the server **SHOULD NOT** envelope the JSON data requested of it in a function, to allow to support compatibility issues.

4.11.2. Explanation

JSONP "callback" enveloping is a pattern to request data by loading a script tag, enabling sharing of data bypassing same-origin policy.

In some circumstances, an application may not have access to HTTP headers, or may need a particular envelope format in order to parse the data (such as JSONP). In these circumstances, it may be necessary to envelope the data with metadata about the response – such as HTTP headers, and for JSONP the "callback" name.

However, these scenarios have been greatly diminished through advances in CORS (cross origin resource sharing) and client software platforms. As a result, not enveloping data should be the default solution, with enveloping used only when absolutely necessary.

The service documentation **MUST** specify if enveloping is supported/required, and provide the schema and usage information for the metadata contained within.

4.11.3. Example

Standard resource representation:

```
{
    “id” : jdoe,
    “name” : “John Doe”
}
```

JSONP Enveloping example, formatted for readability:

```
callbackFunction(
{
    “id” : 123,
    “name” : “John Doe”
})
```

Alternative simple enveloping example showing how response metadata can be included, formatted for readability (also see 4.12 Response Enveloping):

```
{
    “meta”: {
        “code”: 200
    },
    “data”: {
        “id” : 123,
        “name” : “John Doe”
    }
})
```

4.11.4. Caveat

None.

4.12. Response Enveloping

4.12.1. Principle

As a Best Practice, a response body should be enveloped within a standard JSON object with three possible sub-objects of “meta”, “data” and “info”.

4.12.2. Explanation

The response body, where appropriate, **MUST** consist of a JSON object containing one or more sub-objects named “meta”, “data” and “info”. See 4.12.4 Caveat for exemptions, such as HTTP code 204.

"meta" **MUST** contain an attribute "code" which contains the HTTP response code. Additional attributes may be included but **MUST** be clearly documented.

```
{
    "meta": {
        code: "200"
        }
}
```

"data" **MUST** contain the actual response body information if a response body is required, otherwise the data object **MAY** not be provided. The content of "data" is usually a JSON object as specified in 4.2 JSON Data. The content of the "data" can be read as the standard response body.

```
{
    "data": {
        "id": "ABC123",
        "name": "ABCM SS ICM OK",
        "dateOfRegistration": "2018-01-01"
    }
}
```

"info" **MUST** contain information as described and specified in 4.7 Error, Warning and Informational Handling and Appendix B – Additional Response Information Specification whenever the HTTP response code is *not* a success code in the 2xx range. Warnings or information can still be present with a 2xx result, and **MUST** be documented so the consumer knows when to check these values and what to expect.

```
{
    "info": {
        "code": "MGV0001EBU",
        "message": "Could not do the thing."
    }
}
```

The service documentation **MUST** specify if this this method of response enveloping is supported/required, and provide the schema and usage information for the metadata contained within.

4.12.3. Example

Successful response example with data returned.

```
{
    "meta": {
        code: "200"
    }
    "data": {
        "id": "ABC123",
        "name": "ABCM SS ICM OK",
        "dateOfRegistration": "2018-01-01"
    }
}
```

Client error response example.

```
{
    “meta”: {
         code: “400”
    }
    “info”: {
         “code”: “MGV0003EBU”,
         “message”: “Could not do the thing.”
    }
}
```

4.12.4. Caveat

Responses which either have no response body, or for which the integration layer/concrete service will not accept one, are exempt from this profile, but **MUST** clearly indicate that this is the expected behaviour in the integration specification.

5. SECURITY PROFILES

5.1. TLS Basic Authentication

5.1.1. Principle

This profile uses the TLS version 1.3 standard which **SHOULD** be used unless it is unavailable in which case justification must be provided and TLS 1.2 **SHOULD** be used. Similarly, if TLS 1.2 is unavailable justification must be provided and then TLS 1.1 **MUST** be used. TLS 1.0 **MUST NOT** be used under any circumstances as it is being deprecated by all major companies.

By default every RESTful transaction **MUST** use this profile for all external profiles.

5.1.2. Explanation

Transport Layer Security (TLS) is a protocol to provide security over the network. TLS provides the encryption capability for systems at the transport layer.

TLS is a Standard RFC, more information can be found at [RFC2246]. NOTE: Do not confuse TLS Basic Authentication with HTTP Basic Authentication. When talking about TLS, “Basic” is used to differentiate from “Mutual”. TLS Mutual Authentication is not required as part of any profile in this document.

5.1.3. Example

N/A. As only HTTP is catered for as a protocol, the simplest example is that endpoints are https://example.api.endpoint instead of http://example.api.endpoint.

5.1.4. Caveat

Some entities may have to use TLS 1.1. When this is the case the reasons should be documented along with the timeframe for moving to an acceptable version.

5.2. JWT

5.2.1. Principle

A signed JWT **MUST** be used to provide authentication and authorisation elements along with any sensitive claims as agreed by both API provider and consumer.

5.2.2. Explanation

JWT is a standard means of representing claims to be transferred between two parties [IETF-RFC7519]. A JWT consists of three elements, a Header, a Payload and a Signature, defined below. JWT are usually signed to provide authentication. If the JWT is not signed, or decrypted into plain JSON it is “unsecured”.

The JWT header follows the JOSE specification (JSON Object Signing and Encryption) [RFC7515]. Note that other optional headers (such as “jwk”, “jku”, “x5c” etc.) are not part of this profile. These other headers **MAY** be used with justification.

JWT Header			
Name	**Description**	**Format / Value**	**Mandatory**
alg	Algorithm. Identifies the cryptographic algorithm used to secure the JWT.	RFC 7518 "JSON Web Algorithms" defines allowed values. [IETF-RFC7518]. Most common values are "HS256" and "RS256" where HS256 is Symmetric/Shared Secret and RS256 is Asymmetric. Therefore when using Asymmetric signing this value **MUST** be "RS256". An unsecured JWT **MUST** use the value "none". The service documentation **MUST** specify accepted types.	YES
typ	Type. This declares the media type of the JWT.	**SHOULD** be "JWT". If other, it **MUST** be described in the service documentation and agreed between parties.	YES
kid	Key ID. This should hint/indicate the key which was used for the signature.	There is no standard format, however it **SHOULD** be in an understandable string. For example "myGovKeyA" might be more informative than an UUID or other complex format. When "alg" is "none", the kid value **MAY** be removed.	NO

The JWT payload consists of a standard set of fields and optional custom private claims which are implementation specific. Where any field is not mandatory but supplied, it **MUST** be processed, for example if "nbf" is provided, the logic to implement it must be completed.

JWT Payload			
Name	**Description**	**Format / Value**	**Mandatory**
iss	Issuer. The system that issued the token. This is a mandatory header for this profile. The issuer **MUST** be validated and an error thrown if absent or the issuer is not trusted.	StringOrURI representing the issuer. For example in myGov it may be "https://my.gov.au".	YES
sub	Subject. Identifies the token subject, the principal that is the subject of the JWT.	Varies across services. For example this may be an MBUN, or an email address. This field represents the entity on whom the API call is intended to operate.	YES
iat	Issued At. Identifies the time at which the token was issued.	UNIX Epoch time [UNIXTIME].	YES
nbf	Not Before. Identifies the time before which the token **MUST NOT** be accepted for processing.	UNIX Epoch time [UNIXTIME].	NO
exp	Expiration. Identifies the token expiration time. A JWT **MUST NOT** be accepted if it has expired.	UNIX Epoch time [UNIXTIME].	YES
aud	Audience. Identifies the recipients that token is intended for.	StringOrURI representing the intended recipient. For example a	YES

		token sent to a ABCD endpoint might be “https://abcd.gov.au”	
jti	JWT ID. A unique identifier for the token. This provides additional audit/logging capability as well as preventing replaying of tokens.	UUID.	YES
squ*	Subject Qualifier. A private claim to qualify the value in the “sub” field. This claim is often critical in a DHS context, e.g. myGov required. The documentation **MUST** specify valid qualifiers.	String which qualifies the “sub” in context. For example the qualifier may be “ABC”, “XYZ” or any other relying party identifier.	NO
sty*	Subject Type. A private claim to clarify the type of value in the “sub” field. This claim is often critical in a DHS context, e.g. myGov required. The documentation **MUST** specify valid types.	String which describes the type of value in “sub”, e.g. “EMAIL” or “MBUN” would be popular values.	NO
uid*	User ID. Private claim that represents the entity (which could be a system user) who is making the API request. The precise value to use **MUST** be specified in the relevant API suite documentation.	String value. Could be a system user id or a system identifier (e.g. “ABC”, “XYZ”).	NO
uty*	User Type. Private claim that identifies the type of user.	Value **MUST** be one of “USER”, “STAFF” or “SYSTEM”.	NO
<claim>	Additional private claim(s). These **MUST** be documented between service provider and consumers to avoid name collisions, ambiguity or confusion.	To be specified in service documentation.	Varies

* Represents a non-standard *private claim* within the JWT.

The JWT signature varies depending on the algorithm implemented. For an unsecured JWT this value **MUST** be an empty string.

JWT Signature		
alg	**Example (formatted for reasability)**	**Description**
HS256	HMACSHA256(base64UrlEncode(header) + “.” + base64UrlEncode(payload), mySharedSecret)	In this example, the HMAC shared secret is “mySharedSecret”.
RS256	RSASHA256(base64UrlEncode(header) + “.” + base64UrlEncode(payload), <public key or certificate>, <private key>)	In this example the parameters in angles brackets are used to replace full certificate/key information. See examples below for more detail.

5.2.3. Example

Also see section 4.8.3 Example for another example of a JWT. The below example is of an asymmetrically signed JWT.

JWT Header:

```
{
  “alg”: “RS256”,
  “typ”: “JWT”,
  “kid”: “KeyAlpha”
}
```

JWT Body:

```
{
    “iss”: “https://example.gov.au”,
    “sub”: “044240b8-067d-44f1-840f-b8a9898a569d”,
    “iat”: 1456786800,
    “nbf”: 1456786800,
    “exp”: 1457996400,
    “aud”: “https://example.gov.au”,
    “jti”: “81f22e43-075c-4d70-aac5-31dbda62d7c7”,
    “squ”: “XYZ”,
    “sty”: “MBUN”,
    “uid”: “044240b8-067d-44f1-840f-b8a9898a569d”,
    “uty”: “SYSTEM”
}
```

Once signed the JWT becomes something like this:

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6IktleUFscGhhIn0.eyJpc3MiOiJodHRwczovL2V4YW1wbGUuZ2
92LmF1Iiwic3ViIjoiMDQ0MjQwYjgtMDY3ZC00NGYxLTg0MGYtYjhhOTg5OGE1NjlkIiwiaWF0IjoxNDU2Nzg2ODAwLCJu
YmYiOjE0NTY3ODY4MDAsImV4cCI6MTQ1Nzk5NjQwMCwiYXVkIjoiaHR0cHM6Ly9leGFtcGxlLmdvdi5hdSIsImp0aSI6Ij
gxZjIyZTQzLTA3NWMtNGQ3MC1hYWM1LTMxZGJkYTYyZDdjNyIsInNxdSI6IkVYQ0hHRSIsInN0eSI6Ik1CVU4iLCJ1aWQi
OiIwNDQyNDBiOC0wNjdkLTQ0ZjEtODQwZi1iOGE5ODk4YTU2OWQiLCJ1dHkiOiJTWVNURU0ifQ.1UG0XeLBZ5m4mXaaQFq
Uo6KYetKjP1D7CVwp5IIJvcR2gvW0wVRD7bZlP1usMVjWfRLueN0ar9WUPCWe1XaeCkMNGiH3uKTNR0oSC35Y_uxKTenvL
DWvdf5Vjwp56r-BA5UzPitZ_uWZfjsKejQj44BiQJTvovp4QkeyOPdnEoA
```

Note that a JWKS endpoint would provide the public key to decrypt the JWT. See 5.4 JWKS.

The signed JWT would then be sent as a custom header called “mgv-jwt” along with the other common headers (see 4.8 Common Headers):

```
mgv-
jwt: eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6IktleUFscGhhIn0.eyJpc3MiOiJodHRwczovL2V4YW1wb
GUuZ292LmF1Iiwic3ViIjoiMDQ0MjQwYjgtMDY3ZC00NGYxLTg0MGYtYjhhOTg5OGE1NjlkIiwiaWF0IjoxNDU2Nzg2ODA
wLCJuYmYiOjE0NTY3ODY4MDAsImV4cCI6MTQ1Nzk5NjQwMCwiYXVkIjoiaHR0cHM6Ly9leGFtcGxlLmdvdi5hdSIsImp0a
SI6IjgxZjIyZTQzLTA3NWMtNGQ3MC1hYWM1LTMxZGJkYTYyZDdjNyIsInNxdSI6IkVYQ0hHRSIsInN0eSI6Ik1CVU4iLCJ
1aWQiOiIwNDQyNDBiOC0wNjdkLTQ0ZjEtODQwZi1iOGE5ODk4YTU2OWQiLCJ1dHkiOiJTWVNURU0ifQ.1UG0XeLBZ5m4mX
aaQFqUo6KYetKjP1D7CVwp5IIJvcR2gvW0wVRD7bZlP1usMVjWfRLueN0ar9WUPCWe1XaeCkMNGiH3uKTNR0oSC35Y_uxK
TenvLDWvdf5Vjwp56r-BA5UzPitZ_uWZfjsKejQj44BiQJTvovp4QkeyOPdnEoA
```

5.2.4. Caveat

N/A.

5.3. JWT Asymmetric Signing

5.3.1. Principle

All external facing services **MUST** use asymmetric encryption.

Whilst historically PKI certificates were generated and distributed as part of member service onboarding for Asymmetric Signing, JWKS **MUST** now be used for all new or enhanced services unless sufficient justification is provided. See 5.4 JWKS for more information on JWKS.

5.3.2. Explanation

Asymmetric Signature Algorithms makes it possible to verify the message without needing to have access to the signing key. This pattern is very well suited for DHS and member service communication.

With Asymmetric Signing, the Public keys are distributed or otherwise made publically available (JWKS) between trusted partners and the tokens are signed using the sender's Private Key. The receiving party can verify the signature with the corresponding senders Public Key. The Public Key would come from the JWKS endpoint, or for older services or where JWKS is not possible, via the already exchanged PKE certificate. The benefit of a JWKS endpoint is that the key can change without having to notify the consumer. The choice of implementation should be decided and documented as part of onboarding, with justification if not using JWKS.

5.3.3. Example

See 5.2 JWT.

5.3.4. Caveat

N/A.

5.4. JWKS

5.4.1. Principle

When using JWT Asymmetric signing, a JWKS endpoint **MUST** be used for exposing the public key used in signing.

5.4.2. Explanation

JWKS is an alternative method to PKI certificates of accessing the public signing key and has some significant advantages over the certificate approach, chiefly that there is no requirement to distribute certificates. Distributing certificates is both an initial burden to perform and operationally a burden to manage over time (as certificates expire etc.).

The location of the JWKS **MAY** be provided directly to the consumer, but **SHOULD** be retrieved with the standard OIDC metadata endpoint (see reference [OIDC-Discovery]). The details of the OIDC metadata are not in the scope of this document however the first few lines from an example myGov development OIDC metadata endpoint is shown below. Note that the JWKS is exposed with the "jwks_uri" value.

```
{
        "issuer": "https://auth.my.gov.au",
        "authorization_endpoint": "https://{environment}.auth.my.gov.au/mga/sps/oauth/oauth20/authorize",
        "token_endpoint": "https://{environment}.auth.my.gov.au/mga/sps/oauth/oauth20/token",
        "userinfo_endpoint": "https://{environment}.auth.my.gov.au/mga/sps/oauth/oauth20/userinfo",
        "jwks_uri": "https://{environment}.auth.my.gov.au/mga/sps/oauth/oauth20/jwks/MYGOV-OIDC-RS256",
        …etc…
```

The components of a JWK within the JWKS are show below (also see Example). Any other supplied parameter within the JWK **MUST** be ignored. A JWKS endpoint is usually generated via an ESB or similar software and **SHOULD NOT** be hand crafted.

JWK Fields			
Name	**Description**	**Format / Value**	**Mandatory**
alg	Algorithm. Identifies the cryptographic algorithm intended for use with this key.	See [IETF-RFC7517]. For DHS JWK this value **MAY** be provided however as can be seen in the "jwks_uri" above, the algorithm type has been determined by the endpoint URL itself i.e. "MYGOV-OIDC-RS256" signifying that the algorithm is RS256.	NO
kty	Key Type. Identifies the algorithm family used.	**MUST** be "RSA".	YES
use	Use. Intended use of the key.	**MUST** be "sig".	YES
x5c	X.509 certificate chain.	Used to expose the chain of certificates. See [IETF-RFC7517] for further information. As DHS keys will not usually have a chain this value will not be present but **MAY** be provided when needed.	NO
e	Exponent. The "Exponent" value of the RSA key.	When the kty is "RSA", this value is required, and therefore in this profile it **MUST** be provided.	YES
n	Modulus. The "Modulus" value of the RSA key.	When the kty is "RSA", this value is required, and therefore in this profile it **MUST** be provided.	YES
kid	Key ID. The case-sensitive unique identifier for the key, especially useful in a JWKS with multiple keys.	The kid **MUST** be provided.	YES
x5t	X.509 Thumbprint (digest).	The x5t **MAY** be provided as an additional means of certificate identification.	NO

An individual JSON Web Key (JWK) is defined by [IETF-RFC7517]. JWKS is a set of JWK's. Often the JWKS contains just a single JWK however multiple keys are allowed. For example if a private key is compromised a key rotation could mean that a replacement new key is added which would have a different identifier (kid).

The service documentation **MUST** specify the endpoint information for the JWKS endpoint and if applicable, the OIDC metadata endpoint for discovery.

5.4.3. Example

Sample myGov JWK:

```
{
    "kty": "RSA",
    "kid": "d0z_bjSa2MRXdaGM--XJmB5_m7bPQLEpptrVa-zJkJ8",
```

```
    "use": "sig",
    "n": "3-0EpXsa1crJ9UuJtCC4_Lg413fvmfnAoHCcF5ZP0CN-
6kK0HQeWD57iL3NXUM8IKxPcP91XylIqjYZA3ehfV_dWkTHSW3zsuBPPHxnf37_KUOi_QR8CjPS-
ZFWmShrpqaja0anC3Sby19k3YaUemTDdubk0BZaV0osQvWBo5hhzHT_qDfmRjkOGOID2Bu16k6MlYyksq-
s63fe6iraglsll0Jq1BQIddku5x7P1XDMqexSxrE7j8KYdPJNQil96G4ts5HLmoycb1tKHzaHs_jQ5lQsWbeEYSONfKtVH
H4P983T2Wgb-zdd5ZYEIeE9NP2oELSGthZdxGPo4JbNzHQ",
    "e": "AQAB"
}
```

Sample JWKS containing a single JWK with a certificate chain:

```
{
    "keys": [
    {
        "alg": "RS256",
        "kty": "RSA",
        "use": "sig",
        "x5c": [
"MIIC+DCCAeCgAwIBAgIJBIGjYW6hFpn2MA0GCSqGSIb3DQEBBQUAMCMxITAfBgNVBAMTGGN1c3RvbWVyLWRlbW9zLmF1d
GgwLmNvbTAeFw0xNjExMjIyMjIyMDVaFw0zMDA4MDEyMjIyMDVaMCMxITAfBgNVBAMTGGN1c3RvbWVyLWRlbW9zLmF1dGg
wLmNvbTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMnjZc5bm/eGIHqO9N9HKHahM7Y31POul+A2wwP4lSpIw
FrWHzxw88/7Dwk9QMc+orGXX95R6av4GF+Es/nG3uK45ooMVMa/hYCh0Mtx3gnSuoTavQEkLzCvSwTqVwzZ+5noukWVqJu
MKNwjL77GNcPLY7Xy2/skMCT5bR8UoWaufooQvYq6SyPcRAU4BtdquZRiBT4U5f+4pwNTxSvey7ki5Oyc1tG49Per/OzA4
O6Tlpv8x7Red6m1bCNHt7+Z5nSl3RX/QYyAEUX1a28VcYmR41Osy+o2OUCXYdUAphDaHo4/8rbKTJhlu8jEcc1KoMXAKjg
aVZtG/v5ltx6AXYOCAwEAAaMvMC0wDAYDVR0TBAUwAwEB/zAdBgNVHQ4EFgQUQxFG6O2h1cG+pnyvJoy9pGJJoCswDQYJK
oZIhvcNAQEFBQADggEBAGvtCbzGNBUJPLICth3mLsXOZ4z8T8iu4tyoiuAshP/Ry/ZBnFnXmhD8vwgMZ2lTgUWwlrvlgN+
fAtYKnwFO2G3BOCFw96Nm8So9sjTda9CCZ3dhoH57F/hVMBB0K6xhklAcOb5ZxUpCIN92v/w+xZoz1XQBHe8ZbRHaP1HpR
M4M7DJk2G5cgUCyu3UBvYS41sHvzrxQ3z7vIePRA4WF4bEkfX12gvnyORsPkrbVMXX1Rj9t6V7QXrbPYBAO+43JvDGYawx
YVvLhz+BJ45x50GFQmHszfY3BR9TPK8xmMmQwtIvLu1PMttNCs7niCYkSiUv2sc2mlq1i3IashGkkgmo="
        ],
        "n":
"yeNlzlub94YgerTO3OcodqEztjfU_S6X4DbDA_iVKkjAWtYfPHDzz_sPCT1Axz6isZdf3lHpq_gYX4Sz-
cbe4rjmigxUxr-FgKHQy3HeCdK6hNq9ASQvMK9LBOpXDNn7mei6RZWom4wo3CMvvsY1w8tjtfLb-yQwJPltHxShZq5-
ihC9irpLI9xEBTgG12q5lGIFPhTl_7inA1PFK97LuSLnTJzWObjO96v_TMDg7pOWm_zHtF53qbVsIOe3v5nmdKXdFf9BjI
ARRfVrbxVxiZHjU6zL6jY5QJdh1QCmENoejj_ytspMmGW7yMRxzUqgxcAqOBpVm0b-_mW3HoBdjQ",
        "e": "AQAB",
        "kid": "NjVBRjY5MDlCMUIwNzU4RTA2QzZFMDQ4QzQ2MDAyQjVDNjk1RTM2Qg",
        "x5t": "NjVBRjY5MDlCMUIwNzU4RTA2QzZFMDQ4QzQ2MDAyQjVDNjk1RTM2Qg"
    }
]}
```

5.4.4. Caveat

There may be scenarios where variations in the JWK parameters is required, but they should be by exception and must be documented and agreed by all parties (consumers and provider).

5.5. OIDC Context

5.5.1. Principle

Existing OIDC Tokens **MUST** be used where end user authentication and/or authorisation is required and the tokens are available in the current context.

5.5.2. Explanation

Once the customer successfully logs into and creates a valid session with a relevant system (e.g. myGov), the replying party system requesting the user access will get Access and Refresh Tokens for that user. The access token is short lived and entitles specific access for that user in myGov system. A Refresh token will have a longer time to live and it can be used for getting a new Access Token for that user.

This profile is fully compatible with the profiles in this document and can continue to be used for identity/authentication/authorisation by passing the bearer token in the Authorization header.

.

5.5.3. Example

This is an example of an Access Token being passed as the Authorization header:

```
Authorization: Bearer 49fad390491a5b547d0f782309b6a5b33f7ac087
```

5.5.4. Caveat

N/A.

6. APPENDIX A – HTTP RESPONSE CODES

6.1. Successful Client Response

HTTP Response Code	Meaning
200 – OK	The operation was performed successfully. The response depends on the verb used for invocation.
201 – Created	The request has successfully executed and a new resource has been created in the process. The response body may be empty or contain a representation containing URIs for the resource created. The Location header in the response may point to the URI as well.
202 – Accepted	The request was valid and has been accepted but has not yet been processed. The response may include a URI to poll for status updates on the request. This would allow asynchronous REST requests but should be clearly documented if any service interface supports this.
204 – No Content	The request was successfully processed but the server did not have any response.

6.2. Redirected Client Requests

HTTP Response Code	Meaning
301 – Moved Permanently	The requested resource is no longer located at the specified URL. The new Location should be returned in the response header. Only GET or HEAD requests should redirect to the new location. The client should update its bookmark if possible.
302 – Found	The requested resource has temporarily been found somewhere else. The temporary Location should be returned in the response header. Only GET or HEAD requests should redirect to the new location. The client need not update its bookmark as the resource may return to this URL.
303 – See Other	This response code has been reinterpreted by the W3C Technical Architecture Group (TAG) as a way of responding to a valid request for a non-network addressable resource. This is an important concept in the Semantic Web when we give URIs to people, concepts, organizations, etc. There is a distinction between resources that can be found on the Web and those that cannot. Clients can tell this difference if they get a 303 instead of 200. The redirected location will be reflected in the Location header of the response. This header will contain a reference to a document about the resource or perhaps some metadata about it.
304 – Not Modified	A GET operation was performed but the data has not changed since the client's cached copy, or an update was performed that does not result in modification of the resource.

6.3. Invalid Client Requests

HTTP Response Code	Meaning
400 – Bad Request	The client made a poorly formed or otherwise invalid request. Support information should be provided through content in the response, as specified in the 'Error' section.

401 – Unauthorized	A client's request could not be correctly authenticated, such as when a security token was not supplied or is invalid. This response code should be supported with error content indicating the nature of the authentication problem, but not information that could assist an attacker.
403 – Forbidden	A client's request was authenticated but could not be authorised.
404 – Not Found	The client request was well formed, but the resource specified does not exist.
405 – Method Not Allowed	The client request does not specify the correct HTTP verb for the URL and content.
406 – Not Acceptable	The response type is not compatible with the "Accept" provided by the client.
408 – Request Timeout	The server took too long to process the request.
409 – Conflict	The request has conflicted with another request. Usually a request to update a resource has failed because it has already changed state.
410 – Gone	The client's request relates to a decommissioned resource or service. Similar to 301 Moved Permanently except that the service has been decommissioned, rather than replaced with a newer version.

6.4. Server Failed to Handle Requests

HTTP Response Code	Meaning
500 – Internal Server Error	The client request is correct, but the server encountered a technical problem during processing.
501 – Not Implemented	The request cannot be performed by the server.
503 – Service Unavailable	The server encountered a technical problem that is likely to be temporary. The client should try again later.

7. APPENDIX B – ADDITIONAL RESPONSE INFORMATION SPECIFICATION (ERROR, WARNING, INFORMATION)

7.1. Codes

What follows is a list of common, reserved codes that may be returned as part of the "integration" layer like an ESB. These codes are common across multiple REST API suites. These are in the range of 0001 to 0999.

In addition to these codes, the business application may return codes in the 1000 to 9999 range. Business errors are specified in the relevant SIS.

#	Error Code	Message
Infrastructure Layer Errors – layercode "IN"		
1.	0001	Unexpected URL format
2.	0002	Poorly formed request content
3.	0003	Missing header
4.	0004	Service temporarily unavailable
5.	0005	Service permanently unavailable
6.	0006	Version not supported
7.	0007	Product not accredited
Security Layer Errors - layercode "SE"		
1.	0001	User not authenticated
2.	0002	User not authorised
3.	0003	Poorly formed security credential
Business - layercode "BU"		
1.	1000	*Sample Business Error Code (no BU codes are standard)*

8. APPENDIX C – REFERENCES

8.1. Normative

A Normative reference is *prescriptive* i.e. they must be adhered to in order to comply with the profile/standard. Abbreviated terms in square brackets used in this document refer to the corresponding document set out below.

Reference	Description
[IETF-JWT-DRAFT]	IETF: http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html
[IETF-RFC2119]	https://www.ietf.org/rfc/rfc2119.txt
[IETF-RFC7231]	https://tools.ietf.org/html/rfc7231
[IETF-RFC4122]	https://www.ietf.org/rfc/rfc4122.txt
[IETF-RFC7519]	https://tools.ietf.org/html/rfc7519
[IETF-RFC2246]	https://www.ietf.org/rfc/rfc2246.txt
[IETF-RFC7517]	https://tools.ietf.org/html/rfc7517
[IETF-RFC7518]	https://tools.ietf.org/html/rfc7518
[OIDC-Discovery]	https://openid.net/specs/openid-connect-discovery-1_0.html

8.2. Informative

Informative references are *descriptive* and should help to understand some of the concepts described in the document.

Reference	Description
[GADBUY]	Google Ad Buyer API Reference: https://developers.google.com/ad-exchange/buyer-rest/v1.3/
[THTTPRESP]	Twitter HTTP response codes: https://dev.twitter.com/overview/api/response-codes
[WHTTPRESP]	WikiPedia list of HTTP response codes: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes
[VBESTPRAC]	Collated list of Rest best practises: http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api#versioning
[GPARAM]	Google custom search parameter list: https://developers.google.com/custom-search/json-api/v1/reference/cse/list
[GHEADERS]	Google common HTTP headers: https://cloud.google.com/storage/docs/reference-headers
[GGZIP]	Google's stance on GZIP: https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/optimize-encoding-and-transfer
[HCACHE]	Heroku recommendations about caching: https://devcenter.heroku.com/articles/increasing-application-performance-with-http-cache-headers#conditional-requests

Reference	Description
[MB10BTA]	10 Design Tips For APIs (Manuel Boy CTO at PhraseApp) https://phraseapp.com/blog/posts/best-practice-10-design-tips-for-apis
[PHRADBPNS]	RESTful API Design. Best Practices in a Nutshell (Philipp Hauer's Blog) https://blog.philipphauer.de/restful-api-design-best-practices
[GHAPIV3]	GitHub API v3 - GitHub Developer Guide https://developer.github.com/v3
[SJ10BPBRA]	10 Best Practices for Better RESTful API (Stefan Jauker) http://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api
[OPENAPI]	The OpenAPI Specifications https://github.com/OAI/OpenAPI-Specification
[UNIXTIME]	Unix time (aka UNIX Epoch time). https://en.wikipedia.org/wiki/Unix_time

9. APPENDIX E – COMPLIANCE STATEMENT

Compliance to this profile is to be recorded by using the table below, and placing it in the interface specification.
Profile – The name of the profile from this document.
Compliance – One of "Full", "Partial", "None" or "N/A".
Description/Justification – A short description of how/why compliance was rated as it was, and if not "Full" then a justification of why the profile has not been fully adhered to.
Reference – The intra-document reference to where the implementation of that profile is described.

Profile	Compliance	Description/Justification	Reference
HTTP Methods			
JSON Data			
HTTP Response Codes			
Versioning Support			
URI Parameters			
Resources			
Error/Warn/Info Handling			
Common Headers			
GZIP Support			
Last Modified and ETag Caching			
JSONP Enveloping			
JWKS			
TLS Basic Authentication			
JWT			
myGov OAuth/OIDC			
JWT Asymmetric Signing			
No Algorithm JWT			
HTTP Basic Authentication			
OpenAPI			
Compliance			
Best Practice: Response Enveloping			