REST API Guidance - Version 1.1

This wiki captures standards, best practices and guidance compiled from industry's thought leaders for developing RESTful APIs. This guidance is intended to provide project teams with information that will result in the development of APIs that conform to a common style and that present service consumers with familiar paradigms across services developed by multiple project teams. There are no mandatory components to this guidance but readers are encouraged to consider the implications of not following this guidance, e.g. making it harder for API consumers to use their services, requiring custom integration code to deal with new data formats and causing unnecessary API versioning issues.

Version History

Version	Description
1.0	Initial published guidance incorporating feedback from SDP Tech Panel
1.1	Updated version incorporating • Review feedback from Gartner • New guidance on error response formats • New guidance on JSON schema usage

RESTful API Design

This page provides guidance on the design of RESTful APIs, refer to API Implementation Considerations for guidance on implementing APIs at CDC.

A service is a logical component that provides functionality to service consumers through an application programming interface (API). APIs decouple the implementation of a service from that service's users by defining a contract between the two, the service implementation can change drastically, e.g. switch the implementation programming language or framework, provided the API remains unchanged.

Service APIs that follow the constraints of the REST architectural style are called RESTful APIs. RESTful APIs define a set of related resources. Each resource is identified by a uniform resource identifier (URI), and clients interact with resources using a common set of operations (GET, POST, PUT, DELETE, etc) which are HTTP methods (aka REST verbs) that all follow a request/response interaction pattern. Operations on resources often involve exchanging representations of those resources (JSON, XML, HTML, etc) and operations are governed by security and

operational policies. RESTful APIs are ideally navigated using hypermedia via links embedded in representations or in operation request or response header fields, however this can be viewed as an "advanced" design pattern that brings additional loose coupling to mature service consumers.

Resource

A resource is an abstract concept that defines how data is exposed to consumers of the API. Resources do not necessarily map directly to service implementation constructs such as a database entity or a row in a database though the two are often related. A single resource might be backed by many entries in one or more data stores such as a database. E.g. a customer account resource on an eCommerce site could be represented by a JSON structure that includes customer information, recent orders, and an account balance. This resource might be backed by a row in a customers database table for identity information, multiple rows in an orders table for recent order information and a computed value based on multiple rows in a transactions table for the account balance. The mapping from an abstract resource to the concrete data that backs it can change over time, e.g. a database backend might be refactored and the customer information split over two tables. The service that provides the RESTful API is responsible for mapping between the abstract resource and the concrete data that backs it, e.g. by aggregating data from multiple internal APIs or other sources.

In general, resources should be coarse-grained to avoid creating overly "chatty" APIs which do not function well over networks (see the fallacies of distributed computing). E.g. a customer account would be an example of a coarse-grained resource, a customer first name resource would be too fine-grained. However, API designers need to take careful account of the needs of consumers to avoid creating resources that are too coarse, make too much use of available network bandwidth, or are too computationally demanding to service or consume.

Resource Identifier (URI)

Each resource in a RESTful API is identified by a unique resource identifier (URI). A URI that can be dereferenced is also called a uniform resource locator (URL) since it provides information about the mechanism used to obtain a representation of that resource. URIs used in RESTful APIs are typically HTTPS URIs (e.g. `https://example.org/`). HTTPS is used in preference to HTTP for anything that requires that interactions between clients and services remain confidential; HTTP and HTTPS URIs are URLs since they both define identifiers that can be used to fetch a representation of that resource by providing the network address of that resource.

Here are guidelines for URLs:

1. A resource is an abstraction of information or a concept. Choose a URI that incorporates the resource name to identify the information or concept. Resource names should be nouns. For example, the SDP Vocabulary Service includes the concept of a Survey to define data collection instruments. The resource name should include "survey" to indicate its meaning, e.g. a resource that provides a list of surveys could be `https://sdp-v.services.cdc.gov/api/surveys`
2. Use plural nouns for consistency. For example, `https://sdp-v.services.cdc.gov/api/surveys` and `https://sdp-v.services.cdc.gov/surveys/S-11`
3. Do not encode API version numbers in the URL. Cool URIs don't change, see the section on versioning for more information.
4. Support content negotiation (selection of the representation format - XML, JSON, etc.) via the HTTP Accept header. For convenience it is also acceptable to *additionally* support URL-based format specification, e.g. `http://example.org/docs/1.xml` to specify an XML representation of the `http://example.org/docs/1` resource.
5. Do not include operation or action names in the URL, e.g. `http://www.acme.com/products/`**`new`**. While this style of URI is common in Web applications, where the example URI might identify a Web form that aids in creation of a new product, Web APIs should instead use URIs like `http://www.acme.com/products` in conjunction with a HTTP POST that includes a representation of the new resource to be created.
6. Filtering, sorting, searching, pagination and resource versions (not API versions) should be specified using query parameters. For example, `https://sdp-v.services.cdc.gov/api/surveys?limit=100` where the limit query parameter is used to limit the number of items returned. This allows multiple such qualifiers to be used together without requiring complex hierarchies within the URI.

Examples of Bad URLs

- `https://www.acme.com/`**`product`** (singular product)
- `https://www.acme.com/products/`**`filter/cats`** (encodes qualifier in URI path)
- `https://www.acme.com/`**`product`**`/1234` (singular product)
- `https://www.acme.com/photos/products/`**`new`** (encodes operation name in URI)
- `https://www.acme.com/`**`api/v1/products`** (encodes API version in URI)

Examples of Good URLs

- `https://www.acme.com/`**`products`** (plural product)
- `https://www.acme.com/products/`**`?filter=cats`** (encodes qualifier in URI query parameter)
- `https://www.acme.com/`**`products`**`/1234` (plural product)
- `https://www.acme.com/photos/`**`products`** (in conjunction with POSTing a representation of the resource to be created)
- `https://www.acme.com/`**`products`** (do not encode API version in URI, instead follow guidance in the section on versioning)

Representations

Representations encode the actual or intended state of a resource at a point in time. The state of a resource can change over time and it can be captured in different formats (e.g. XML or JSON). This means that a resource can have multiple representations that differ over time and differ in format. Representation metadata is included in HTTP headers and is used for various purposes by both clients and servers (e.g. determining the format).

REST operations on a resource proceed via the exchange of representations that capture the current or intended state of that resource. A representation is a sequence of bytes that encode the state, plus metadata that describes those bytes. Representation format should be decided based on the use-case. Typical representations are HTML for human consumption and XML or JSON for machine-to-machine data exchange based on the needs of anticipated API consumers.

The format of a representation is known as its media type. It is recommended that APIs should define the media types they use in detail including:

- The data format, e.g. XML or JSON
- The specific dialect of the data format, e.g. Atom Feed or Entry or a particular XML or JSON schema as a subset of the generic format specified above
- The use of hypertext within the format, in particular any link relationships used

Registering custom media types is encouraged; registration under the vendor subtree (application/vnd.xyz, e.g. application/vnd.gov.cdc.sdp.surveys+xml) provides a lightweight framework for registering a new media type.

Media types are used in several HTTP header fields, e.g. a request to obtain the current representation of a resource in the application/vnd.gov.cdc.sdp.surveys+xml media type would like this

```
GET https://sdp-v.services.cdc.gov/surveys
Host: sdp-v.services.cdc.gov
Accept: application/vnd.gov.cdc.sdp.surveys+xml
Accept-Language: en-US, en
Accept-Charset: utf-8
```

Media types can include parameters and it is recommended that the version of a media type is specified using a parameter (by convention the letter v), e.g. application/vnd.gov.cdc.sdp.surveys+xml**;v=2.0**, for further details see the guidance in the section on versioning.

XML vs JSON

The two most common data formats used for RESTful Web services are Extensible Markup Language (XML) and JavaScript Object Notation (JSON). Each has advantages and disadvantages and choosing between them requires careful consideration of the use case and capabilities of each format.

XML

XML was originally envisioned as a more easily implemented subset of SGML that would enable structured documents and data to be exchanged over the Web. Unlike HTML, which was focused on a document structure (sections, subsections, paragraphs etc) and presentation, XML focused on semantic markup and provided a means for XML vocabulary authors to define their own data elements and structures. XML was rapidly adopted and spawned a set of associated technologies including:

- **XML Namespaces**: provide a mechanism to qualify the names of elements to provide modularity and distributed developed of markup vocabularies
- **XML Schema**: provides a mechanism to define the data types and structures used in a markup vocabulary
- **XPath**: provides a mechanism to identify content in a specific part (or parts) of an XML document
- **XSLT**: builds on XPath to provide a mechanism to transform XML documents into other formats of XML and other document formats
- **XQuery**: builds on XPath to define a query language for XML documents similar in capability to SQL for relational databases

Advantages of XML include

- Most modern programming languages have libraries for consuming and producing XML
- The XML ecosystem includes a mature set of associated technologies for defining types and structures, querying documents and transforming documents
- Rich data types including explicit support for hypermedia, encoded binary data
- Good support for representing both data and prose documents

Disadvantages of XML include

- The XML information set does not typically map directly to native programming language constructs. Instead, some form of data binding is used to map between the two
- Large domains tend to spawn complex schema definitions
- XML can be somewhat verbose
- Strictly defined schemas can be fragile and cause difficulty when evolving APIs

JSON

JSON is a text format for the serialization of structured data. It is derived from the object literals of JavaScript. Like JavaScript, JSON can represent four primitive types (strings, numbers, booleans, and null) and two structured types (objects and arrays). Like XML, JSON has also spawned a set of associated technologies including:

- **JSON-LD**: a method for encoding linked data using JSON that also includes similar capabilities to XML namespaces
- **JSON Schema**: provides a mechanism to define JSON data structures that can be used to validate that a particular JSON instance matches the expectations of the producer or consumer
- **JSONiq**: a query language for JSON, similar in capability to XQuery
- **JSON Hypertext Application Language**: a media type for representing resources and their relations with hyperlinks

Advantages of JSON include

- Most modern languages support parsing of JSON to native language constructs and serialization of native language constructs to JSON
- The data model of JSON is similar to that of most programming languages so a direct mapping between the two is generally possible, particularly for loosely-typed languages
- Simple type system
- Concise serialization
- Most modern Web APIs now use JSON

Disadvantages of JSON include

- Associated technologies are less mature than their XML counterparts, e.g. JSON Schema is still evolving while XML Schema was a W3C Recommendation in 2001
- Simple type system requires extensions for common requirements like hypermedia and binary data
- Not suitable for representing prose documents

Recommendations

Use JSON if:

1. You can get by using only four primitive types (strings, numbers, booleans, and null) and two structured types (objects and arrays)
2. Your primary clients are in-browser JavaScript
3. You don't expect clients to rely on schema-based validation
4. You don't need to capture rich text
5. Your clients prefer JSON

Use XML if:

1. Your API deals with structured prose documents (mixed text and semantic markup)
2. You expect clients to use schema-based validation
3. You expect clients to use standards-based tooling to transform and extract information from your data
4. You need a rich set of specific data types
5. Your clients prefer XML

REST Request

Operations

Operations on the resource should be performed using HTTP methods. Here are the guidelines for using typical HTTP methods:

HTTP method	Operation	Idempotent	Example	Notes
GET	Retrieve a representation of a resource. Resource can be a collection or single item	Yes	All the surveys in a vocabulary Service can be retrieved by: GET *https://sdp-v.services.cdc.gov/surveys* Filtering can be controlled using query parameters.	Control metadata should be used to optimize requests, e.g. If-None-Match to only retrieve a representation if it has changed since the last time the client retrieved it.
POST	Create or perform an operation on the resource	No	A new survey can be created using Vocabulary Service by POST *https://sdp-v.services.cdc.gov/surveys* The new survey URI will be assigned automatically	The initial state of the new resource should be included in the request.

PUT	Update a resource by replacing its state entirely with that conveyed by the supplied representation	Yes	A survey can be updated using Vocabulary Service by PUT *https://sdp-v.services.cdc.gov/surveys/{id}*	Full representation of the object should be sent in the request. Control metadata should be used to perform optimistic locking. e.g, use an If-Match header with an etag to avoid overwriting updates from others.
PATCH	Update part of resource using the delta conveyed by the supplied representation	No	A survey could be updated using the Vocabulary service by PATCH *https://sdp-v.services.cdc.gov/surveys/{id}*	A delta from the current state to the desired state should be sent in the request. Control metadata should be used to perform optimistic locking, e.g. If-Match to ensure that the resource has not changed since the delta was created.
DELETE	Delete a resource	Yes	A survey can be deleted using Vocabulary Service by DELETE *https://sdp-v.services.cdc.gov/surveys/{id}*	Control metadata should be used to perform optimistic locking, e.g. If-Match to ensure that the resource being deleted has not changed since the last time it was retrieved.
OPTIONS	Obtain the supported methods and CORS on a resource	Yes	Supported methods and control information can be retrieved by OPTIONS *https://sdp-v.services.cdc.gov/surveys* Some browsers use this method to support CORS by querying the resource for Access-Control-Allow-Origin header	

Control Data

Control data defines the purpose of a message between components, such as the action being requested or the meaning of a response. For example, cache behavior can be modified by control data specified as HTTP Cache-Control header included in the request or response message. Etags are useful to help prevent simultaneous updates (optimistic locking) of a resource from overwriting each other. Control data using HTTP headers should be used appropriately. Below is an example of control data in a Request and Response

Request

```
GET /surveys HTTP/1.1
Host: sdp-v.services.cdc.gov
If-None-Match: "0eaac798-cb6a-41fd-ac7a-78f1a40945c6"
```

The `If-None-Match` header makes the request conditional on the current state of the resource being different to the state when the client last retrieved a representation - essentially "only perform this request if the state of the resource has changed since the last time I retrieved it". This type of request can be used to optimize operations that could result in potentially large representations.

Response

```
HTTP/1.1 200 OK
Cache-Control: max-age=86400
ETag: "fc187038-2147-4b53-acef-5230f1d5b0aa"
```

In this example the `If-None-Match` precondition was met so the response includes a `200 OK` status code, an updated etag that captures the current state of the resource and (not shown) a representation of the resource. If the precondition had not been met the response would have indicated a `412 Precondition Failed` status and would not have included a representation.

REST Response

Status Codes

When the client makes a request to the server through an API, the response to that request should indicate the status of the request, e.g. whether it was successful, incorrect or caused an error. HTTP status codes provide a standard set of response codes for this purpose. Use of the correct status code is important to integrate the function of the API with the underlying HTTP infrastructure and to respect the layering of the API on HTTP, e.g. an API would not want error responses to be cached for other users and may want to use HTTP facilities to redirect clients to alternate resources as the API evolves. APIs that do not layer correctly on HTTP (e.g. by sending error responses using a HTTP 200 status code) are said to tunnel over HTTP rather than layer on it.

Use of the following HTTP codes is recommended:

2xx (Success category)

These status codes represent that the request was received and successfully processed by the server.

- **200 Ok** The standard HTTP response representing success for GET, PUT or POST.
- **201 Created** This status code should be returned whenever the new resource is created. E.g on creating a new resource, using POST method, should always return 201 status code. The server should also include a Location header with this response to inform the client of the the identifier of the new resource.
- **202 Accepted** This status code indicates that the request was accepted for future processing (queued), this status is particularly useful for asynchronous APIs. The server should include a Location header that provides the identifier of a resource that can provide a update on the status of the request. The server may also include a Retry-After header to indicate a suitable pause before obtaining the status.
- **204 No Content** indicates that the request was successfully processed, but has not returned any content.
  DELETE can be a good example of this.
  The API `DELETE /companies/43/employees/2` will delete the employee 2 and in return we do not need any data in the response body of the API, as we explicitly asked the system to delete. If there is any error, like if `employee 2` does not exist in the database, then the response code would be not be of `2xx Success Category` but around `4xx Client Error category`.

3xx (Redirection Category)

- **301 Move Permanently** indicates that the URI of the requested resource has been changed and the new URI is provided in the Location header.
- **302 Found** indicates that the resource is *temporarily* identified by a different URI (supplied in the Location header). New changes in the URI might be made in the future. Therefore, this same URI should be used by the client in future requests.
- **304 Not Modified** indicates that request preconditions (e.g. If-Modified-Since or If-None-Match) were not met or the client has the response already in its cache. In both cases there is no need to transfer the same representation again. This response should be used for requests with `GET` or `HEAD` methods.

4xx (Client Error Category)

These status codes represent that the client submitted a faulty request.

- **400 Bad Request** indicates that the request by the client was not processed, as the server could not understand what the client is asking for.
- **401 Unauthorized** indicates that the client is not allowed to access resources, and should re-request with the required credentials.
- **403 Forbidden** indicates that the request is valid and the client is authenticated, but the client is not allowed to access the page or resource for any reason. E.g sometimes the authorized client is not allowed to access the directory on the server.
- **404 Not Found** indicates that the requested resource is not available now.
- **406 Not Acceptable** indicates that the server was unable to return a representation of the resource in the format requested by the client (in the request Accept header).
- **410 Gone** indicates that the requested resource is no longer available which has been intentionally moved.
- **412 Precondition Failed** indicates that access to the target resource has been denied. This happens with conditional requests on methods other than `GET` or `HEAD` when the condition defined by the `If-Unmodified-Since` or `If-None-Match` headers is not fulfilled. In that case, the request, usually an upload or a modification of a resource, cannot be made and this error response is sent back.
- **415 Unsupported Media Type** indicates the media format of the requested data is not supported by the server, so the server is rejecting the request.

5xx (Server Error Category)

- **500 Internal Server Error** indicates that the request is valid, but the server encountered an error. All the runtime unhandled errors and exceptions should return this status code.
- **503 Service Unavailable** indicates that the server is down or unavailable to receive and process the request. Mostly if the server is undergoing maintenance.

Error Handling

For all the errors and exceptions on the server, return an error response along with the correct HTTP status code. Use of the JSON and XML formats defined by RFC 7807 is recommended.

The type of the error response should be communicated using the Content-Type response header. For RFC 7807, the content type is `application/problem+json` or `application/problem+xml`.

Example RFC 7807 Error Response

```
HTTP/1.1 400 Bad Request
Content-Type: application/problem+json
Content-Language: en

{
   "type": "https://example.net/validation-error",
   "title": "Invalid request parameters.",
   "invalid-params": [
      {
         "name": "age",
         "reason": "must be a positive integer"
      },
      {
         "name": "color",
         "reason": "must be 'green', 'red' or 'blue'"
      }
   ]
}
```

The above example indicates that a client sent an invalid request so the response is returned with a 400 HTTP response code.

Hypermedia As The Engine Of Application State (HATEOAS)

HATEOAS introduces discoverability and evolvability, providing a way of making a RESTful API self-documenting. Rather than coding a client to use specific service URIs, it instead relies on a generic understanding of hypermedia and a detailed knowledge of the representation format in use and the link relationships used in that format.

The client enters a RESTful HATEOAS API through a simple fixed or previously bookmarked URL. All future actions the client may take are discovered within resource representations or HTTP headers returned from the server. The media types used for these representations, the link relations they may contain, and operations available on the resource are standardized. The client transitions through application states by selecting from the links within a representation or by manipulating the representation in other ways afforded by its media type. In this way, RESTful interaction is driven by hypermedia, rather than out-of-band information.

Benefits of HATEOAS include:

1. The client is informed of what actions can be taken using the contents of responses
2. The server can change its URI scheme without breaking clients, provided clients behave according to the expectations of HATEOAS
3. New capabilities can be advertised by putting new links in the response

Guidelines:

1. Use links to allow clients to discover locations and operations
2. Use Hypermedia Link Relations to specify the relationship
3. Use the atom:link structure to capture links in XML representations
4. Use JSON HAL to capture links in JSON representations

Sample XML HTTP Response with Hypermedia

```
HTTP/1.1 200 OK
Content-Type: application/vnd.com.bank.account+xml
Content-Length: ...

<?xml version="1.0"?>
<account>
    <account_number>12345</account_number>
    <balance currency="usd">100.00</balance>
    <link rel="self" href="https://bank.example.com/accounts/12345" />
    <link rel="transfer"
href="https://bank.example.com/accounts/12345?amount={amount}&from={sour
ceAccount}" />
    ...
</account>
```

Note the use of URI templates in the above to create parameterized URIs.

Sample JSON HTTP Response with Hypermedia

```
Request:
GET /surveys/1 HTTP/1.1
Accept: application/vnd.sdp-v-services.v1+json

Response:
HTTP/1.1 200 OK
Content-Type:  application/vnd.sdp-v-services+json;v=1.0;charset=utf-8

{
  "id":1,
  "name":"Salmonella outbreak",
  "creator":"John Doe",
  "_links": {
    "self": {
      "href": "https://sdp-v.services.cdc.gov/surveys/1"
    },
    "program": {
      "href": "https://sdp-v.services.cdc.gov/programs/17"
    }
  }
}
```

In the example the `_links` object contains a `self` link that defines its canonical URI.

Headers

Enable CORS

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

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: *
```

HTTP Strict Transport Security

Enable HTTP Strict Transport Security (HSTS) by adding the following header in all the responses:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

Caching

HTTP Caching eases the load on a service since it doesn't need to serve all clients directly, and it can improve performance if it takes less time to transmit the resource back to the client. Caching is use case dependent. Every request and response can define its own caching policies via the `Cache-Control` HTTP header. For requests, the directive specifies whether cached responses are permitted and under what circumstances. For responses, the directive specifies whether it can be cached, under what conditions and for how long. Below is a list of *Cache-Control* directive that can be used in HTTP requests and responses.

HTTP Request:

HTTP Request Cache Control Headers

```
Cache-Control: max-age=<seconds>
Cache-Control: max-stale[=<seconds>]
Cache-Control: min-fresh=<seconds>
Cache-Control: no-cache
Cache-Control: no-store
Cache-Control: no-transform
Cache-Control: only-if-cached
```

HTTP Response:

HTTP Response Cache Control Headers

```
Cache-Control: must-revalidate
Cache-Control: no-cache
Cache-Control: no-store
Cache-Control: no-transform
Cache-Control: public
Cache-Control: private
Cache-Control: proxy-revalidate
Cache-Control: max-age=<seconds>
Cache-Control: s-maxage=<seconds>
```

As described by Irakli Nadareishvili, there are two main use cases for cache use in RESTful APIs:

1. **Rapidly changing data**: even rapidly changing data can benefit from caching and the benefit scales with the volume of requests. E.g. a service responding to hundreds of requests per second can benefit greatly from a cache where data expires after only a few seconds. When the cache expiry time is short, clock drift between client and server becomes an issue so clients should refrain from using time-stamp dependent headers such as `Expires`, `Last-Modified` and `If-Modified-Since` and instead use time-period dependent controls such as `Cache-Control: max-age`.
2. **Mostly static data**: more static data can be cached for potentially long periods using time-stamp dependent headers such as `Expires`, `Last-Modified` and `If-Modified-Since` since clock skew between client and server is unlikely to be an issue for longer cache expiry times.

When a client maintains its own cache of information, use of eTags as described above can also provide a similar benefit. E.g. a client can include an `If-None-Match` header to make a request conditional upon a different representation of the resource being available. Note that eTags serve to shortcut requests that would return the same data as a previous request whereas caching serves to distribute a time-limited response to intermediate servers that can offload request processing from the source of truth. eTags work well to reduce the number of duplicate requests from a single client, caches work well to share the result of one client's request with other subsequent clients.

The following example shows an HTTP response that specifies a two second cache timeout

HTTP response with cache control header

```
HTTP/1.1 200 OK
Content-Type: application/vnd.sdp-v-services+json;v=1.0;charset=utf-8
Cache-Control: max-age=2
```

A client can override the response cache policy using its own cache control headers as shown below

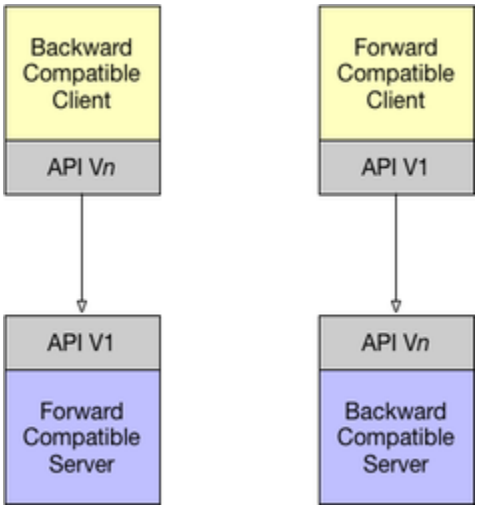
HTTP request with cache control header

```
GET /surveys/1 HTTP/1.1
Accept: application/vnd.sdp-v-services+json;v=1.0
Cache-Control: max-age=1
```

Versioning and Compatibility

Requirements change over time and, in turn, lead to changes in APIs. Since APIs define a contract between a provider (server) and consumer (client) it is important to consider the impact that changes will have on compatibility. Compatibility is typically categorized as forwards and backwards compatibility and can be stated from the context of the API provider or API consumer. The diagram below illustrates compatibility

concepts within the context of APIs.



- An API provider is backwards compatible if it can interoperate with consumers of older versions of the API
- An API provider is forwards compatible if it can interoperate with consumers of newer versions of the API
- An API consumer is backwards compatible if it can interoperate with providers of earlier versions of the API
- An API consumer is forwards compatible if it can interoperate with providers of newer versions of the API

Compatibility is an important consideration since clients and servers are often the responsibility of different parties, there are often many more clients of an API than providers, and it can be difficult to upgrade all parties at the same time. The rest of this section describes approaches to versioning of APIs that follows Postel's Law. Additional material on application of Postel's law to the design of APIs is available in Jose Luis Ordiales's blog.

Versioning RESTful APIs

A RESTful API's contract is defined by the constraints specified in the Uniform Interface. The Uniform Interface comprises of these four constraints

1. Identification of resources or resource identifier
2. Manipulation through representations
3. Self-descriptive messages
4. Hypermedia as the engine of application state (HATEOAS)

The following guidelines focus on resource identifier (URL) and representation (XML, JSON, HTML, etc) versioning and these are from Howard Dierking's blog:

- If additional fields are being added to the representation, then just add it to the representation. Clients and servers should ignore what they don't understand and you don't need to version the API. Refer to Postel's Law.
- If there are breaking changes to the representation, version the representation and use content negotiation to serve the right representation version to clients. For example, PIN code is changed to ZIP code in Address resource representation, then the Address representation's version should be increased to the next version. The same guideline will apply when the hypermedia links are modified. The example of how content negotiation can be done using the `Accept` header is described in the media type versioning section. Handling changes using media type versioning strategy is the recommended approach.
- If you want to change the meaning of a resource then resist the urge to change an existing resource, instead create a new one with a different URI.

Refer to Howard Dierking's blog for additional details with examples.

Note that this section relates to versioning of APIs, not versioning of the underlying information that is represented as resources in the API. How a service exposes the revision history of a particular logical resource is a different problem that can often be solved by offering revision-specific identifiers along with a floating identifier that identifies the latest revision, e.g.:

Identifying the version of a resource in the URI

```
https://sdp-v.services.cdc.gov/api/surveys/10
https://sdp-v.services.cdc.gov/api/surveys/10/versions/2
https://sdp-v.services.cdc.gov/api/surveys/10?version=2
```

In the above there is a single resource (a survey with an id of 10) that has multiple versions. The first URI is a version-independent identifier that floats to the latest version of a resource while the second and third URIs show two alternative approaches to identifying a specific version of the survey.

Recommended approaches to versioning RESTful APIs are described below:

1. No Versioning: New resources and functionality can be added without requiring any versioning provided existing clients can continue to work based on the pre-existing functionality.

Adding new content to resource representations does not represent a breaking change if it does not impact existing clients. For example, the representation of a survey resource might be:

```
HTTP/1.1 200 OK
Content-Type:  application/json; charset=utf-8

{
  "id":1,
  "name":"Salmonella outbreak",
  "creator":"John Doe"
}
```

If a new dateCreated field is added to the representation of a survey resource, then the response would look like this:

```
HTTP/1/1 200 OK
Content-Type:  application/json; charset=utf-8

{
  "id":1,
  "name":"Salmonella outbreak",
  "creator":"John Doe",
  "dateCreated:"2018-08-08"
}
```

If existing clients have built in logic to ignore unrecognized fields they will continue to operate normally while new client applications can use the new data field. Similarly, adding new resources and new link relation types should not cause breaking changes provided existing clients can continue to work without understanding the new functionality.

Breaking changes to the representation of a resource such as removing, renaming or restructuring fields (e.g. restructuring the *creator* field in the example above into sub-fields such as *firstname* and *lastname*) requires other approaches as described below.

The W3C TAG published a guide to extending and versioning XML schemas that contains practical advice on the use of XML features for versioning XML-based representations with a focus on forwards and backwards compatibility. An equivalent guide for versioning of JSON Schemas could not be located at the time of authoring this document, however general guidance to preserving forwards and backwards compatibility between clients and services is to

- Preserve existing object structure and only add new properties that are safe to ignore
- Preserve stated bounds of values, e.g. do not add new values to enums or extend the range of numbers
- Preserve stated lengths of arrays and uniqueness constraints
- Preserve string validations

Adding new query parameters does not represent a breaking change if it does not impact existing clients. For example, in the first version of an API, a survey might be identified using the following URI:

```
https://sdp-v.services.cdc.gov/api/surveys/10
```

In a future version of the API, it may be desirable to limit the size of the returned response by adding a `verbose` boolean parameter, e.g.:

```
https://sdp-v.services.cdc.gov/api/surveys/10?verbose=false
```

Addition of this new `verbose` query parameter can be done in a non-breaking way by defaulting its value so that omitting it would produce the same result as the prior version of the API. Existing clients that don't know about the parameter (and don't include it the URIs they use) can continue to function without change, new clients that are aware of the parameter can use it to control the size of the returned response. This approach also complies with the guidance that cool URIs don't change since existing URIs for a resource continue to work unchanged.

The same kind of approach can be taken to renaming query parameters or supporting new values for those parameters. Rather than removing existing parameters or changing the meaning of existing parameters or parameter values, continue to support the existing parameters and parameter values and add support for new parameters or parameter values.

2. Media Type Versioning: When consuming a RESTful API a client specifies the format of content that it expects as part of HTTP GET request using the Accept header. The client can specify XML, JSON, or some other format that it expects in the body of the response. Additionally, the Accept header can also be used to specify a custom media type that allows the client application to specify the version of the resource representation it expects in the response. For a breaking change to the survey representation, this approach is shown below.

Version 1:

```
Request:
GET /surveys/1 HTTP/1.1
Accept: application/vnd.sdp-v-services+json;v=1.0

Response:
HTTP/1.1 200 OK
Content-Type:  application/vnd.sdp-v-services+json;v=1.0; charset=utf-8
{"id":1, "name":"Salmonella outbreak","creator":"John Doe"}
```

Version 2:

```
Request:
GET /surveys/1 HTTP/1.1
Accept: application/vnd.sdp-v-services+json;v=2.0

Response:
HTTP/1.1 200 OK
Content-Type:  application/vnd.sdp-v-services+json;v=2.0; charset=utf-8
{"id":1, "name":"Salmonella outbreak","creator":{"firstname":"John",
"lastname": "Doe"}}
```

In both the examples above the *application/vnd.sdp-v-services+json;v=x* element in the Accept header tells the API to return version x (1 OR 2) of the media type (the *json* component of the media type is a convention that indicates that the underlying format is JSON). The service should return the requested representation version if possible and confirm this using the *Content-Type* header in the response. In case no media type is specified in the request it is recommended that the service returns a default media type response message.

This approach is the recommended approach for implementation of a RESTful API as it aligns itself with the principle of REST where content negotiation is handled using media types and it lends naturally to HATEOAS where server can send in response all the related data in resource link along with the their MIME type. It should be noted that this versioning scheme requires the RESTful API code to look at the Accept header to determine content to return in response. When using this approach it is important to include a `Vary` header in responses that includes the `Content-Type` header to ensure that caches use both the URL and media type of the response to determine whether a request can be served by a cached response.

Note that the above change could have been made in a non-breaking fashion (i.e. following the No Versioning approach) by adding the restructured field (e.g. *creator_structured* in addition to the existing *creator* field).

Access Control

It is recommended that APIs that require authentication and authorization utilize OAuth for that purpose. Use of OAuth to secure CDC APIs is detailed in Using OAuth to Secure CDC APIs.

API Definition

APIs need to be described using a standard, language-agnostic interface description which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. Consumers of an API should be able understand and interact with the service with a minimal amount of implementation logic. The OpenAPI Specification is the industry standard for defining and describing RESTful APIs. APIs should be defined and described using the latest version of OpenAPI Specification.

The API definition should be made available using an endpoint, https://*service*/api. This endpoint should render HTML, JSON and YAML representations.

Note that there is currently a divergence between JSON Schema and Open API, full details and a proposed approach to work around the incompatibility is described in this article by Phil Sturgeon.

Advertising API Availability

In addition to creating an API definition and offering it at the standard endpoint as described above, it is recommended that consideration be given to advertising API availability via API registries. CDC currently hosts two such registries:

- open.cdc.gov is a catalog of publicly accessible APIs hosted by CDC
- easi.cdc.gov is a catalog of internal CDC APIs

References

1. Fielding's dissertation on REST
2. Richardson Maturity Model
3. Designing HTTP Interfaces and RESTful Web Services
4. Securing Microservices APIs, Sustainable and Scalable Access Control - Matt McLarty, Rob Wilson & Scott Morrison
5. Hypermedia Link Relations
6. Media Types
7. HTTP resources and specifications
8. HTTP Status codes
9. HTTP Caching