HTTP-based API guidelines

These are a set of guidelines for building network APIs that take advantage of HTTP features and semantics.

General guidelines

- URLs should represent "nouns". HTTP methods should represent "verbs" (HTTP verbs)
  - GET for side effect-free operations
  - HEAD for getting metadata about a resource (caching information, content type, content length, etc)
  - POST for non-idempotent operations with side effects (e.g., create, append, apply delta)
  - PUT for idempotent operations with side effects (e.g., update, replace)
  - DELETE for deletion operations
- If you need to support multiple content types (e.g., JSON, XML, HTML, etc), use content negotiation (via Accept headers).

```
GET /someurl HTTP/1.1
Host: example.com
Accept: application/json
```

- Compress responses when client requests so via Accept-Encoding and Content-Encoding headers
  - Request

```
GET /someurl HTTP/1.1
Host: example.com
Accept-Encoding: gzip
```

  - Response

```
HTTP/1.1 200 OK
Content-Encoding: gzip
Content-Length: 1234
Content-Type: application/json

... gzipped data ...
```

- Use HTTP auth for authentication (basic, digest, oauth)
- Use SSL (HTTPS) for channel encryption
- Use HTTP status codes to indicate request outcome
- Return a meaningful response body in case of an error (secondary error code, short explanation, etc). Respect expected content type based on content negotiation if possible.

```
POST /someurl HTTP/1.1
Host: example.com
Content-Type: application/json
Accept: application/json
Content-Length: 18

{"field": "value"}
```

```
HTTP/1.1 400 Bad Request
Content-Type: application/json

{"code": "10", "message": "required field2 is missing"}
```

- Use Cache-Control header to provide hints to clients on the cacheability of a resource.
- Use caching-related headers (ETag, Last-Modified, If-Modified-Since, etc) and conditional GET for cacheable resources. Prefer etag to timestamp based lase-modified to avoid clock skew production problems.

```
GET /someurl HTTP/1.1
Host: example.com
```

```
HTTP/1.1 200 OK
Date: Wed, 25 Nov 2009 18:08:47 GMT
Server: Apache/2.2.11 (Ubuntu)
Last-Modified: Fri, 20 Nov 2009 02:21:09 GMT

... data ...
```

```
GET /someurl HTTP/1.1
Host: example.com
If-Modified-Since: Fri, 20 Nov 2009 02:21:09 GMT
```

```
HTTP/1.1 304 Not Modified
```

- Always use request body for sending the entity during POST or PUT. Don't encode it in url parameters.
- In a POST, only use url parameters for algorithm and response hints.
    - Example: add `pretyPrint=true` to get a pretty printed JSON or XML response.
    - Example: add `UseAvScan=true` to enable an expensive antivirus scanning module which is normally disabled
- In a GET, url parameters should only be used in conjunction with algorithm resources such as search.
    - Example: `/search?from=[date]&to=[date]`
- Use links in resource representations where it makes sense. They make it easier for clients to navigate from resource to resource without having to know URL patterns in advance.

Practical concerns

- Prefer HTML for human interaction, json for machine interaction (faster and easier to parse and manipulate than XML). User other formats when it makes sense, though.
- Allow for overriding method via url parameter (e.g., ?_method=PUT) to support browser-based interaction.
- Allow for overriding Accept headers via url parameter (e.g., ?_format=application/json) to make it easier for browser or command line-based testing
- Prefix urls with a version identifier. It makes it easier to support multiple incompatible versions as the system evolves (e.g, /v1/device)
- API should be explicit, not a side-effect of how classes/methods/variables in the implementation are structure and named. Otherwise, the definition is brittle and it becomes hard refactor the code while maintaining the same semantics and syntax. Frameworks like Jersey, Rails, etc. have mechanisms to help with decoupling interface from implementation.

Versioning and backward-compatibility

- Backward compatibility in the context of client-server architectures: An API S2 is backward compatible with S1 if a client C that was written to work against S1 can operate against S2 unchanged.
- RESTful APIs should be designed to be backward compatible in the common case. Backward compatibility should be broken only an exception basis and should be avoided to the extent possible.
- Advantages:
    - It simplifies rollout of services that require high uptime by allowing for servers to be updated individually while the service continues to operate.
- Constraints on how the API needs to evolve:
    - The new API must be a superset of the old one.
    - New operations. Existing operations must still be supported, with the **same** semantics.
    - New data types & formats. Existing formats must still be supported, with the **same** semantics.
    - New optional fields for existing data types, with assumed defaults if they are missing.
- Constraints on how clients interact with servers:
    - Unknown fields in responses from the server must be ignored.
    - Clients should avoid sending data that the server does not expect. Otherwise, future versions of the API might start using them break existing clients.
- Resources should be versioned to accomodate future changes that need to break backward compatibility
    - URLs should be prefixed with a version number. E.g., "/v1/user", "/v2/customer"
        - Not pure within the REST architectural style, but simpler to work with than the alternative approach (i.e., versioned media types)
    - If a new version of the service introduces a backward-incompatible API, the old API must still be supported. Old APIs can be dropped when no more clients use them.
- Testing requirements

        - APIs should have a comprehensive suite of unit tests. These should be used to validate that changes to the server implementation maintains backwards compatibility
        - API clients should have unit tests that validate that the client interacts with the API in the way the API was designed to work.
        - Tests should include extra bogus data in requests and responses to verify that clients and servers are properly ignoring or rejecting extra data.
        - Note: tests are not going to catch all possible bugs, so services must be designed to be fault-tolerant and clients must be smart enough to deal with mis-behaving servers (retries, defer requests, etc).
- Documentation requirements
    - Each client and server API must be clearly documented for backwards and forwards compatibility. This is necessary for operations to understand the restrictions of rolling out and back the new versions of the software.

Useful HTTP codes

- 200 OK: successful request
- 201 Created: used in response to a POST. The response should return a **Location** header with a url where the newly created resource can be accessed.
- 202 Accepted: the request has been accepted but not yet processed. The response should contain a **Location** header with a url where the client can check status

- 301 Moved Permanently: the resource is no longer found under the given location. The response should contain a **Location** header with the new location. Clients should no longer use the old url.
- 303 See Other: the response to a request can be found by looking at another url (via GET). Generally used after a POST or PUT.
- 304 Not Modified: in response to a **conditional GET**, when the server determines the resource hasn't changed.
- 307 Moved Temporarily: similar to 301, but reflecting a temporary move. This status code is used to tell a client that they should repeat their request against an alternate url, instead (i.e., they should preserve their original request method)

- 400 Bad Request: the request contains bad syntax. Generally an indication of a **client** error.
- 401 Unauthorized: the client is not authorized to access the given resource without proper authorization.
- 403 Forbidden: similar to 401, but indicating that authentication will make no difference.
- 404 Not Found: the given resource cannot be found (but may be available later)
- 405 Method Not Allowed: the resource doesn't support the requested method (e.g., POST not allowed on a read-only resource). The server should return an **Allow** header with the list of supported methods for the given resource.
- 406 Not Acceptable: the resource doesn't support the requested content type (e.g., application/xml was requested but only application/json or application/xml+xhtml is supported).
- 408 Request Timeout: the request could not be fulfilled in the allotted time. Useful for implementing server-side SLA constraints and load shedding.
- 409 Conflict: used in response to a conditional PUT to indicate that the state of the resource in the server has changed and doesn't match what the client expects it to be. Generally used for implementing optimistic transactions. Last-Modified timestamps or ETags can be used to represent the "version" to check against.
- 410 Gone: the resource is no longer available at this location. This reflects a permanent condition, unlike **404 Not Found**.
- 411 Length Required: the request cannot be accepted without a Content-Length header.
- 412 Precondition Failed: may be used to indicate a "resource already exists" error.
- 413 Request Entity Too Large: the content of the request is larger than what the server is willing to accept (e.g., due to insufficient disk space to spool the request entity). If this is a temporary condition, the server can return a **Retry-After** header.
- 415 Unsupported Media Type: the server does not support the submitted content type or encoding (e.g., gzip) for the given resource. Note that this refers to the type or encoding of the request body, not to what the client expects in the response. For the latter, see **406 Not Acceptable**.

- 500 Internal Server Error: used for indicating that an unknown error occurred on the server – a **catch all** for any unexpected errors.
- 503 Service Unavailable: the server cannot fulfill the request at this time. This can be used to provide an indication that the server is currently overloaded or unwilling to respond for arbitrary reasons. Clients can be told to wait before retrying via a **Retry-After** header.
- 505 HTTP Version Not Supported

Examples

Create a user

POST /api/v1/user
Accepts: application/json
Returns: 201 on success
Supported return types: application/json, application/xml
Return headers: "Location", pointing to the url for the newly created user

Get a user

GET /api/v1/user/{id}
Returns: 200 on success
Supported return types: application/json, application/xml, application/xhtml+xml

Get a specific attribute of a user

GET /api/v1/user/{id}/{attribute}
Returns: 200 on success
Supported return types: text/plain, application/json

Get a user, override content negotiation headers

GET /api/v1/user/{id}?_format=application/xml

Search for users with a given name

GET /api/v1/user?name={value}

Search for users with multiple attributes

GET /api/v1/user?name={value1}&age={value2}

Search using advanced expression

GET /api/v1/user?query={expression}

Saved searches

POST /api/v1/user/search
Request body: expression
Returns: 201 on success
Return headers: "Location" of saved search (e.g., /api/v1/user/search/1234)

GET /api/v1/user/search/{id}

Update a user

PUT /api/v1/user/{id}
Returns: 200 on success

Update a specific attribute of a user

PUT /api/v1/user/{id}/{attribute}
Returns: 200 on success

Update a user, url-based method override

POST /api/v1/user/{id}?_method=PUT

Delete a user

DELETE /api/v1/user/{id}

Delete a user, url-based method override

POST /api/v1/user/{id}?_method=DELETE

Resources

- RFC 2616 - HTTP 1.1
- RESTwiki
- RESTful Web Services Book
- RFC 5023 - Atom Publishing Protocol

- Case Study: RESTful Web Services at Orbitz
- How to GET a cup of coffee
- Amazon SQS "REST" API