**API Best Practices**

Below points should be considered to make an API more interactive and self serviceable.

1. Resource Definition

2. Resource URI definition

3. RAML Practices

4. API Portal

5. Working with Collection of Resources

6. API Security

7. Data Volumes

8. Pagination / Resource Linking

9. Bounding Queries

10. Status Codes and Error Responses

Resource Definition

The fundamental concept in any RESTful API is the resource. A resource is an object with a type, associated data, relationships to other resources, and a set of methods that operate on it. It is similar to an object instance in an object-oriented programming language, with the important difference that only a few standard methods are defined for the resource (corresponding to the standard HTTP GET, POST, PUT and DELETE methods), while an object instance typically has many methods. Resource definition plays an important role in any API design.

Resource URI Definition

Each resource in a service suite will have at least one URI identifying it. And it's best when that URI makes sense and adequately describes the resource. URIs should follow a predictable, hierarchical structure to enhance understandability and, therefore, usability: predictable in the sense that they're consistent, hierarchical in the sense that data has structure—relationships. This is not a REST rule or constraint, but it enhances the API.

RESTful APIs are written for consumers. The name and structure of URIs should convey meaning to those consumers. It's often difficult to know what the data boundaries should be, but with understanding of your data, you most-likely are equipped to take a stab and what makes sense to return as a representation to your clients. Design for your clients, not for your data.

RAML Practices

RESTful API Modeling Language (RAML) makes it easy to manage the whole API life-cycle from design to sharing. It's concise - you only write what you need to define - and reusable. It is machine readable API design that is actually human friendly.

Working with Collections of Resources

Resources can be grouped into collections. Each collection is homogeneous so that it contains only one type of resource, and unordered. Resources can also exist outside any collection. In this case, we refer to these resources as singleton resources. Collections are themselves resources as well.

Collections can exist globally, at the top level of an API, but can also be contained inside a single resource. In the latter case, we refer to these collections as sub-collections. Sub-collections are usually used to express some kind of “contained in” relationship.

API Security

RESTful web services should use session-based authentication, either by establishing a session token via a POST or by using an API key as a POST body argument or as a cookie. Usernames, passwords, session tokens, and API keys should not appear in the URL, as this can be captured in web server logs, which makes them intrinsically valuable.

Resource Linking

Linking can help with discoverability and use-case scalability. Presenting the fully qualified URL for every resource aids discoverability by allowing for new resources to be consumed by just embedding a new reference (link). This works well for JSON because JSON is schema-less; adding new fields will not break functionality and will maintain backwards compatibility.

Fortunately, if a resource’s endpoint changes, the new URL will be returned and consumed as though nothing happened, which aids scalability. Additionally, providers should support 302 redirects for deprecated URLs to ensure clients with older URLs still work. Since virtually every type of web client can read a URL, developers do not need to use any additional libraries to create or parse the URLs.

Pagination is a great example for when to use linking when returning resource collections. Developers working with your API might not know how pagination works or know how to build the URL for the next set of results. Instead, you can provide them links that automatically support pagination.

Example:

HTTP/1.1 200 OK

{

"href" : "https://api.mycompany.com/v1/users?offset=50&limit=50"

"offset": 50,

"limit": 50,

“first”: {

“href”: "https://api.mycompany.com/v1/users"

},

“prev”: {

“href”: "https://api.mycompany.com/v1/users"

},

“next”: {

“href”: "https://api.mycompany.com/v1/users?offset=100&limit=50"

},

“last”: {

“href”: "https://api.mycompany.com/v1/users?offset=50&limit=50"

},

"items": [

{

... user 51 name/value pairs ...

},

...,

{

... user 100 name/value pairs ...

}

}

}

Status Codes and Error Responses

HTTP defines a bunch of meaningful status codes(https://en.wikipedia.org/wiki/List\_of\_HTTP\_status\_codes) that can be returned from your API. These can be leveraged to help the API consumers route their responses accordingly. Below are the short list of the ones that we definitely should be using:

• 200 OK - Response to a successful GET, PUT, PATCH or DELETE. Can also be used for a POST that doesn't result in a creation.

• 201 Created - Response to a POST that results in a creation. Should be combined with a Location header pointing to the location of the new resource

• 204 No Content - Response to a successful request that won't be returning a body (like a DELETE request)

• 304 Not Modified - Used when HTTP caching headers are in play

• 400 Bad Request - The request is malformed, such as if the body does not parse

• 401 Unauthorized - When no or invalid authentication details are provided. Also useful to trigger an auth popup if the API is used from a browser

• 403 Forbidden - When authentication succeeded but authenticated user doesn't have access to the resource

• 404 Not Found - When a non-existent resource is requested

• 405 Method Not Allowed - When an HTTP method is being requested that isn't allowed for the authenticated user

• 410 Gone - Indicates that the resource at this end point is no longer available. Useful as a blanket response for old API versions

• 415 Unsupported Media Type - If incorrect content type was provided as part of the request

• 422 Unprocessable Entity - Used for validation errors

• 429 Too Many Requests - When a request is rejected due to rate limiting

Further Reading

Apart from the conceptual practices there are few code build practices that needs to be enforced to make the API rich in terms of usability.

https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.30