API Design Cheat Sheet

1. Build the API with consumers in mind–as a product in its own right.
   - Not for a specific UI.
   - Embrace flexibility / tunability of each endpoint (see #5, 6 & 7).
   - Eat your own dogfood, even if you have to mockup an example UI.
2. Use the Collection Metaphor.
   - Two URLs (endpoints) per resource:
     - The resource collection (e.g. /orders)
     - Individual resource within the collection (e.g. /orders/{orderId}).
   - Use plural forms ('orders' instead of 'order').
   - Alternate resource names with IDs as URL nodes (e.g. /orders/{orderId}/items/{itemId})
   - Keep URLs as short as possible. Preferably, no more-than three nodes per URL.
3. Use nouns as resource names (e.g. don't use verbs in URLs).
4. Make resource representations meaningful.
   - "No Naked IDs!" No plain IDs embedded in responses. Use links and reference objects.
   - Design resource representations. Don't simply represent database tables.
   - Merge representations. Don't expose relationship tables as two IDs.
5. Support filtering, sorting, and pagination on collections.
6. Support link expansion of relationships. Allow clients to expand the data contained in the response by including additional representations instead of, or in addition to, links.
7. Support field projections on resources. Allow clients to reduce the number of fields that come back in the response.
8. Use the HTTP method names to mean something:
   - POST - create and other non-idempotent operations.
   - PUT - update.
   - GET - read a resource or collection.
   - DELETE - remove a resource or collection.
9. Use HTTP status codes to be meaningful.
   - 200 - Success.
   - 201 - Created. Returned on successful creation of a new resource. Include a 'Location' header with a link to the newly-created resource.
   - 400 - Bad request. Data issues such as invalid JSON, etc.
   - 404 - Not found. Resource not found on GET.

- 409 - Conflict. Duplicate data or invalid data state would occur.

10. Use ISO 8601 timepoint formats for dates in representations.
11. Consider connectedness by utilizing a linking strategy. Some popular examples are:
    - HAL
    - Siren
    - JSON-LD
    - Collection+JSON
12. Use OAuth2 to secure your API.
    - Use a Bearer token for authentication.
    - Require HTTPS / TLS / SSL to access your APIs. OAuth2 Bearer tokens demand it. Unencrypted communication over HTTP allows for simple eavesdroppping and impersonation.
13. Use Content-Type negotiation to describe incoming request payloads.

    For example, let's say you're doing ratings, including a thumbs-up/thumbs-down and five-star rating. You have one route to create a rating: **POST /ratings**

    How do you distinguish the incoming data to the service so it can determine which rating type it is: thumbs-up or five star?

    The temptation is to create one route for each rating type: **POST /ratings/five_star** and **POST /ratings/thumbs_up**

    However, by using Content-Type negotiation we can use our same **POST /ratings** route for both types. By setting the *Content-Type* header on the request to something like **Content-Type: application/vnd.company.rating.thumbsup** or **Content-Type: application/vnd.company.rating.fivestar** the server can determine how to process the incoming rating data.
14. Evolution over versioning. However, if versioning, use the Accept header instead of versioning in the URL.
    - Versioning via the URL signifies a 'platform' version and the entire platform must be versioned at the same time to enable the linking strategy.
    - Versioning via the Accept header is versioning the resource.
    - Additions to a JSON response do not require versioning. However, additions to a JSON request body that are 'required' are troublesome–and may require versioning.
    - Hypermedia linking and versioning is troublesome no matter what–minimize it.
    - Note that a version in the URL, while discouraged, can be used as a 'platform' version. It should appear as the first node

in the path and not version individual endpoints differently (e.g. api.example.com/v1/...).

15. Consider Cache-ability. At a minimum, use the following response headers:
    - ETag - An arbitrary string for the version of a representation. Make sure to include the media type in the hash value, because that makes a different representation. (ex: ETag: “686897696a7c876b7e”)
    - Date - Date and time the response was returned (in RFC1123 format). (ex: Date: Sun, 06 Nov 1994 08:49:37 GMT)
    - Cache-Control - The maximum number of seconds (max age) a response can be cached. However, if caching is not supported for the response, then no-cache is the value. (ex: Cache-Control: 360 or Cache-Control: no-cache)
    - Expires - If max age is given, contains the timestamp (in RFC1123 format) for when the response expires, which is the value of Date (e.g. now) plus max age. If caching is not supported for the response, this header is not present. (ex: Expires: Sun, 06 Nov 1994 08:49:37 GMT)
    - Pragma - When Cache-Control is ‘no-cache’ this header is also set to ‘no-cache’. Otherwise, it is not present. (ex: Pragma: no-cache)
    - Last-Modified - The timestamp that the resource itself was modified last (in RFC1123 format). (ex: Last-Modified: Sun, 06 Nov 1994 08:49:37 GMT)

16. Ensure that your GET, PUT, and DELETE operations are all idempotent. There should be no adverse side affects from operations.