22 Best Practices to Take Your API Design Skills to the Next Level

Here are some best practices to follow.

First, Some Terminology
Any API design follows something called **Resource Oriented Design** It consists of three key concepts

- **Resource:** A resource is a piece of data, For example, **a User**.
- **Collection:** A group of resources is called a collection. Example: **A list of users**
- URL: Identifies the location of resource or collection. Example: /user

1. Use kebab-case for URLs

For example, if you want to get the list of orders.

Bad:

```
/systemOrders or /system_orders
```

Good:

```
/system-orders
```

2. Use camelCase for Parameters

For example, if you want to get products from a particular shop.

Bad:

```
/system-orders/{order_id} or /system-orders/{OrderId}
```

Good:

```
/system-orders/{orderId}
```

3. Plural Name to Point to a Collection

If you want to get all the users of a system.

Bad:

```
GET /user or GET /User
```

Good:

```
GET /users
```

4. URL Starts With a Collection and Ends With an Identifier

If want to keep the concept singular and consistent.

Bad:

```
GET /shops/:shopId/category/:categoryId/price
```

This is bad because it’s pointing to a property instead of a resource.

Good:

```
GET /shops/:shopId/ or GET /category/:categoryId
```

5. Keep Verbs Out of Your Resource URL

Don’t use verbs to express your intention in the URL. Instead, use proper HTTP methods to describe the operation.

Bad:

```
POST /updateuser/{userId} or GET /getusers
```

Good:

```
PUT /user/{userId}
```

6. Use Verbs for Non-Resource URL

You have an endpoint that returns nothing but an operation. In this case, you can use verbs. For example, if you want to resend the alert to a user.

Good:

```
POST /alerts/245743/resend
```

Keep in mind that these are not our CRUD operations. Rather, these are considered functions that do a specific job in our system.

7. Use camelCase for JSON property

If you’re building a system in which the request body or response is JSON, the property names should be in `camelCase`

Bad

```
{
   user_name: "Mohammad Faisal"
   user_id: "1"
}
```

Good

```
{
   userName: "Mohammad Faisal"
   userId: "1"
}
```

8. Monitoring

RESTful HTTP services MUST implement the `/health` and `/version` and `/metrics` API endpoints. They will provide the following info.

/health

Respond to requests to `/health` with a `200 OK` status code.

/version

Respond to request to `/version` with the version number.

/metrics

This endpoint will provide various metrics like average response time.

`/debug` and `/status` endpoints are also highly recommended.

9. Don’t Use table_name for the Resource Name

Don’t just use the table name as your resource name. In the long run, this kind of laziness can be harmful.

Bad:

```
product_order
```

Good:

```
product-orders
```

This is because exposing the underlying architecture is not your purpose.

10. Use API Design Tools

There are many good API design tools for good documentation, such as:

- API Blueprint
- Swagger

Having good and detailed documentation results in a great user experience for your API consumers.

11. Use Simple Ordinal Number as Version

Always use versioning for the API and move it all the way to the left so that it has the highest scope. The version number should be `v1`, `v2` etc.

Good:

```
http://api.domain.com/v1/shops/3/products
```

Always use versioning in your API because if the API is being used by external entities, changing the endpoint can break their functionality.

12. Include Total Number of Resources in Your Response

If an API returns a list of objects always include the total number of resources in the response. You can use the `total` property for this.

Bad:

```
{
  users: [
     ...
  ]
}
```

Good:

```
{
  users: [
     ...
  ],
  total: 34
}
```

13. Accept limit and offset Parameters

Always accept `limit` and `offset` parameters in `GET` operations.

Good:

```
GET /shops?offset=5&limit=5
```

This is because it’s necessary for pagination on the front end.

14. Take fields Query Parameter

The amount of data being returned should also be taken into consideration. Add a `fields` parameter to expose only the required fields from your API.
Example:
Only return the name, address, and contact of the shops.

```
GET /shops?fields=id,name,address,contact
```

It also helps to reduce the response size in some cases.

15. Don’t Pass Authentication Tokens in URL

This is a very bad practice because often URLs are logged and the authentication token will also be logged unnecessarily.

Bad

```
GET /shops/123?token=some_kind_of_authenticaiton_token
```

Good

Instead, pass them with the header:

```
Authorization: Bearer xxxxxx, Extra yyyyy
```

Also, authorization tokens should be short-lived

16. Validate the Content-Type

The server should not assume the content type. For example, if you accept `application/x-www-form-urlencoded` then an attacker can create a form and trigger a simple POST request.
So, always validate the `content-type` and if you want to go with a default one use `content-type: applicaiton/json`

17. Use HTTP Methods for CRUD Functions

HTTP methods serve the purpose of explaining CRUD functionality.
`GET`: To retrieve a representation of a resource.
`POST`: To create new resources and sub-resources.
`PUT`: To update existing resources.
`PATCH`: To update existing resources. It only updates the fields that were supplied, leaving the others alone
`DELETE`: To delete existing resources.

18. Use the Relation in the URL For Nested Resources

Some practical examples are:

- `GET /shops/2/products` : Get the list of all products from shop 2.
- `GET /shops/2/products/31`: Get the details of product 31, which belongs to shop 2.
- `DELETE /shops/2/products/31` , should delete product 31, which belongs to shop 2.
- `PUT /shops/2/products/31` , should update the info of product 31, Use PUT on resource-URL only, not the collection.
- `POST /shops` , should create a new shop and return the details of the new shop created. Use POST on collection-URLs.

19. CORS

Do support CORS (Cross-Origin Resource Sharing) headers for all public-facing APIs.
Consider supporting a CORS allowed origin of “*”, and enforcing authorization through valid OAuth tokens.
Avoid combining user credentials with origin validation.

20. Security

Enforce HTTPS (TLS-encrypted) across all endpoints, resources, and services.
Enforce and require HTTPS for all callback URLs, push notification endpoints, and webhooks.

21. Errors

Errors, or more specifically service errors, occur when a client makes an invalid or incorrect request to a service or passes invalid or incorrect data to a service, and the service rejects the request.
Examples include invalid authentication credentials, incorrect parameters, unknown version IDs, etc.

- Do return `4xx` HTTP error codes when rejecting a client request due to one or more Service Errors.
- Consider processing all attributes and then returning multiple validation problems in a single response.

22. Golden Rules

If you are ever in doubt about an API formatting decision, these golden rules can help guide us to making the right decision.

- Flat is better than nested.
- Simple is better than complex.
- Strings are better than numbers.
- Consistency is better than customization.

That’s it — congratulations if you’ve made it this far! I hope you learned a thing or two.