Swagger
Supported by SMARTBEAR

Best Practices in API Design

Good API design is a topic that comes up a lot for teams that are trying to perfect their API strategy. In a previous blog post, I briefly discussed the importance of API design. The benefits of a well-designed API include: improved developer experience, faster documentation, and higher adoption for your API . But what exactly goes into *good* API design? In this blog post, I will detail a few best practices for designing RESTful APIs.

Characteristics of a well-designed API

In general, an effective API design will have the following characteristics:

- **Easy to read and work with**: A well designed API will be easy to work with, and its resources and associated operations can quickly be memorized by developers who work with it constantly.
- **Hard to misuse:** Implementing and integrating with an API with good design will be a straightforward process, and writing incorrect code will be a less likely outcome. It has informative feedback, and doesn't enforce strict guidelines on the API's end consumer.
- **Complete and concise:** Finally, a complete API will make it possible for developers to make full- fledged applications against the data you expose. Completeness happens over time usually, and most API designers and developers incrementally build on top of existing APIs. It is an ideal which every engineer or company with an API must strive towards.

For the purpose of illustrating the concepts listed below, I'll be taking the example of a photosharing app. The app allows users to upload photos, characterizing them with the location where these photos were taken and hashtags that describe the emotions associated with them.

Collections, Resources, and their URLs

Understanding resources and collections

Resources are fundamental to the concept of REST. A resource is an object that's important enough to be referenced in itself. A resource has data, relationships to other resources, and methods that operate against it to allow for accessing and manipulating the associated information. A group of resources is called a collection. The contents of collections and resources depend on your organizational and consumer requirements. If, for example, you believe the market will benefit from obtaining basic information about your product's user base, then you could expose this as a collection or resource. A Uniform Resource Locator (URL) identifies the online location of a resource. This URL points to the location where your API's resources exist. The base URL is the consistent part of this location. In the case of the photosharing app, we could expose data about the users who use the app through collections and resources, accessed by the appropriate URL.

1. /users: a collection of users
2. /users/username1: a resource with information about a specific user

Nouns describe URLs better

The base URL should be neat, elegant, and simple so that developers using your product can easily use them in their web applications. A long and difficult-to-read base URL is not just bad to look at, but can also be prone to mistakes when trying to recode it. Nouns should always be trusted. There's no rule on keeping the resource nouns singular or plural, though it is advisable to keep collections plural. Having the same plurality across all resources and collections respectively for consistency is good practice. Keeping these nouns self explanatory helps developers understand the kind of resource described from the URL, which can eventually enable them to become self sufficient while working with your API . Coming back to the photosharing app, say it has a public API with /users and /photos as collections. Notice how they're all plural nouns?They're also self explanatory and we can infer that /users and /photos gives information about the product's registered userbase, and shared photos respectively.

Describe resource functionality with HTTP methods

All resources have a set of methods that can be operated against them to work with the data being exposed by the API. RESTful APIs comprise majorly of HTTP methods which have well defined and unique actions against any resource. Here's a list of commonly used HTTP methods that define the CRUD operations for any resource or collection in a RESTful API.

Method Description

PUT Used to update existing resources
PATCH Used to update existing resources
DELETE Used to delete existing resources

(If you want to know the difference between PUT and PATCH, check out this feed on StackOverflow.) Keeping verbs out of your URLs is also a good idea. The operations GET, PUT, POST and DELETE are already used to operate on your resource described by the URL, so having verbs instead of nouns in the URL can make working with your resources confusing. In the photosharing app, with /users and /photos as end points, an end consumer of your API can easily work with them intuitively using the RESTful CRUD operations described above.

Responses

Give feedback to help developers succeed

Providing good feedback to developers on how well they are using your product goes a long way in improving adoption and retention. Every client request and server side response is a message and, in an ideal RESTful ecosystem, these messages must be self descriptive. Good feedback involves positive validation on correct implementation, and an informative error on incorrect implementation that can help users debug and correct the way they use the product. For an API, errors are a great way to provide context to using an API. Align your errors around the standard HTTP codes. Incorrect, client side calls should have 400-type errors associated with them. If there are any server side errors, then a suitable 500 response must be associated with them. A successful method used against your resource should return a 200-type response. There are a lot of response codes. For additional information, check out this REST API tutorial. In general, there are three possible outcomes when using your API: -

1. The client application behaved erroneously (client error - 4xx response code)
2. The API behaved erroneously (server error - 5xx response code)
3. The client and API worked (success - 2xx response code)

Hand holding your end consumer to success whenever they hit a road block working with your API will go a long way in improving developer experience and preventing API misuse. Describe these error responses well, but keep them concise and neat. Provide enough information in the error codes for an end user to start work on fixing the cause, and, if you feel there's more information needed, provide links to additional documentation.

Give examples for all your GET responses

A well- designed API also has an example of the kind of response one can expect on successful call against a URL. This example response should be simple, plain, and quick to comprehend. A good rule of thumb is to help developers understand exactly what a successful response would give them in under five seconds. Coming back to our photosharing app, we've defined a /users and a /photos URL. The /users collection would give the username and date of joining of all the users who have registered with the app in an array. You can use an API design tool to define your API in the Swagger (OpenAPI) specification as follows:

```
responses:
200:
description: Successfully returned information about users
schema:
type: array
items:
type: object
properties:
username:
type: "string"
example: "kesh92"
created_time:
type: "dateTime"
example: "2010-01-12T05:23:19+0000"
```

Notice the data types and an example described in each response item an end user can expect from a successful GET call. The successful response an end user would receive in JSON would look as follows.

```
{
“data”:[
{
```

},
{
“username”:“example_user2”,
“created_time":“2015-3-19T17:51:15+0000 ”
}
....
]
}

If the end user successfully calls the end point with a GET method, the user should obtain the above data along with a 200 response code to validate the correct usage. Likewise, an incorrect call should produce an appropriate 400 or 500 response code with relevant information to help user better operate against the collection.

Requests

Handle complexity elegantly

The data you’re trying to expose can be characterized by a lot of properties which could be beneficial for the end consumer working with your API. These properties describe the base resource and isolate specific assets of information that can be manipulated with the appropriate method. An API should strive towards completion, and provide all the required information, data and resources to help developers integrate with them in a seamless manner. But completion means taking into account common use cases for your API. There could be numerous such relationships and properties, and it’s not good practice to define resources for each of them. The amount of data the resource exposes should also be taken into account. If you’re trying to expose a lot, there can be negative implications on the server, especially with regards to load and performance. The above cases and relationships are important considerations in the design of the API, and can be handled using the appropriate parameters. You can sweep properties and limit responses behind the ‘?’ in a query parameter, or isolate specific component of the data the client is working with using a path parameter. Let's take the example of our photosharing app. It could be of use to developers to get information on all the photos shared in a particular location and a specific hashtag. You also want to limit the number of results to 10 per API call to prevent server load. If the end user wants to find all photos in Boston with a hashtag #winter, the call would be:

```
GET /photos?location=boston&hashtag=winter&limit=10
```

Notice how the complexities have now been reduced to a simple association with a query parameter. If you want to provide information about a specific user depending on the client’s request, the call would be:

```
GET /users/kesh92
```

Where kesh92 is the username of a specific user in the users collection, and will return the location and date of joining for kesh92. These are just some of the ways you could design parameters that strive towards API completion and help your end developers use your API intuitively.

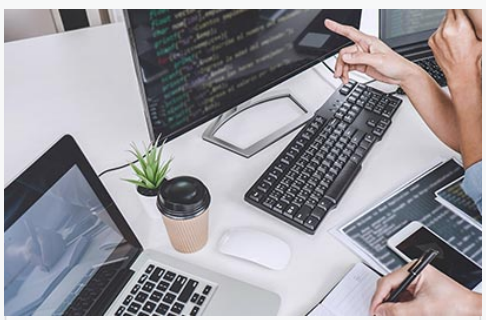
Finally, when in doubt, leave it out. If you’re having second thoughts about a specific resource or collection’s functionality, then leave it for the next iteration. Developing and maintaining APIs is a continuous process, and waiting for the feedback from the right users can go a long way in building a robust API that enables users to integrate and develop applications in creative ways.

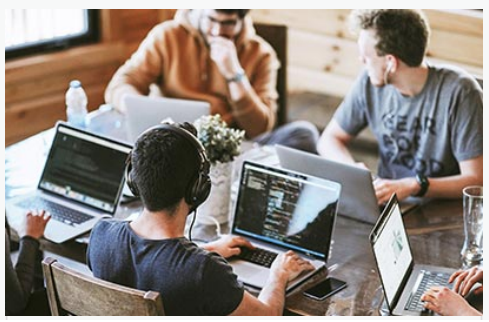
Get started with API design

There isn’t a single approach to API design that will work for every organization. The above suggestions are just that — advice and recommendations which can be used or discarded depending on your user case and requirement. One of the main reasons why API design is crucial is to help the end consumer use your API. Their needs should be the guiding light towards designing and building a great API.

Interested in getting started with API design for REST APIs? Find out how Swagger API design tools can help.

Related Resources

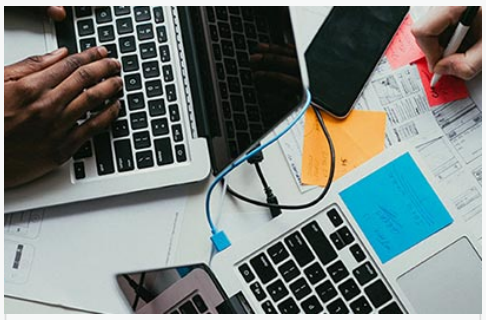




Developer experience is an extension of general User Experience, which emphasizes the...

1 hour

house from the ground up, you expected them to...



Definition Driven API Development: How OAS & Swagger Help Teams Streamline Their API...

We cover trends in the API landscape that have led to the adoption of API definitions...

1 hour