Design Effective and Secure REST APIs

ALEX XU
MAY 17, 2023 · PAID

215 6 Share

In the last issue, we explored various API architectural styles, each with its unique strengths. Despite the many options, REST remains the most popular. However, its popularity doesn't imply simplicity. REST merely defines resources and the use of HTTP methods. To master the art of crafting REST APIs, we need to follow certain guidelines to ensure that we design efficient, user-friendly APIs.

In this issue, we cover the finer details of REST API design. This includes:

- Sniffing out API issues. We learn to identify the telltale signs of inefficient APIs. The "bad smells" hint at a need for a redesign or improvement.
- Understanding API maturity. We dive into the Richardson Maturity Model (RMM), a model that helps us distinguish good from bad API design by assessing how closely an API aligns with the REST framework.
- Back to basics. To ensure we are all on the same page, we revisit the core components of REST APIs - HTTP verbs and status codes.

To bring these concepts to life, we'll embark on the first of the three practical examples in this issue with the design of a sign-up and sign-in component. In our next issue, we'll continue and explore how to construct a shopping cart API and study the Stripe payment API redesign.

Detecting API Issues

How can we tell when an API isn't living up to its potential and needs a redesign? Much like code, APIs can give off a distinct "smell" when they are not performing as they should. Recognizing these red flags is crucial. Let's look at some concrete scenarios that suggest an API might not be up to par.

For example, suppose we have thoroughly studied the API documentation, but we are still struggling to understand the nuances of its functionality. This needs for constant clarification from API owners is a clear indication of an API that could be more user-friendly.

Consider another example where API parameters and results are vaguely defined. This lack of clarity can lead to confusion and potential errors. It slows down development and hinders effective collaboration between teams.

Also, think about a situation where our front-end and back-end teams find themselves needing to collaborate extensively just to test and validate API behaviors. This level of coordination overhead suggests an API that isn't as intuitive or well-documented as it should be.

As API owners, we may notice different sets of red flags. If we find ourselves constantly fielding API usage queries, it is like we are wearing a part-time customer service hat. This scenario points to an API that might benefit from more detailed documentation and possibly a more intuitive design.

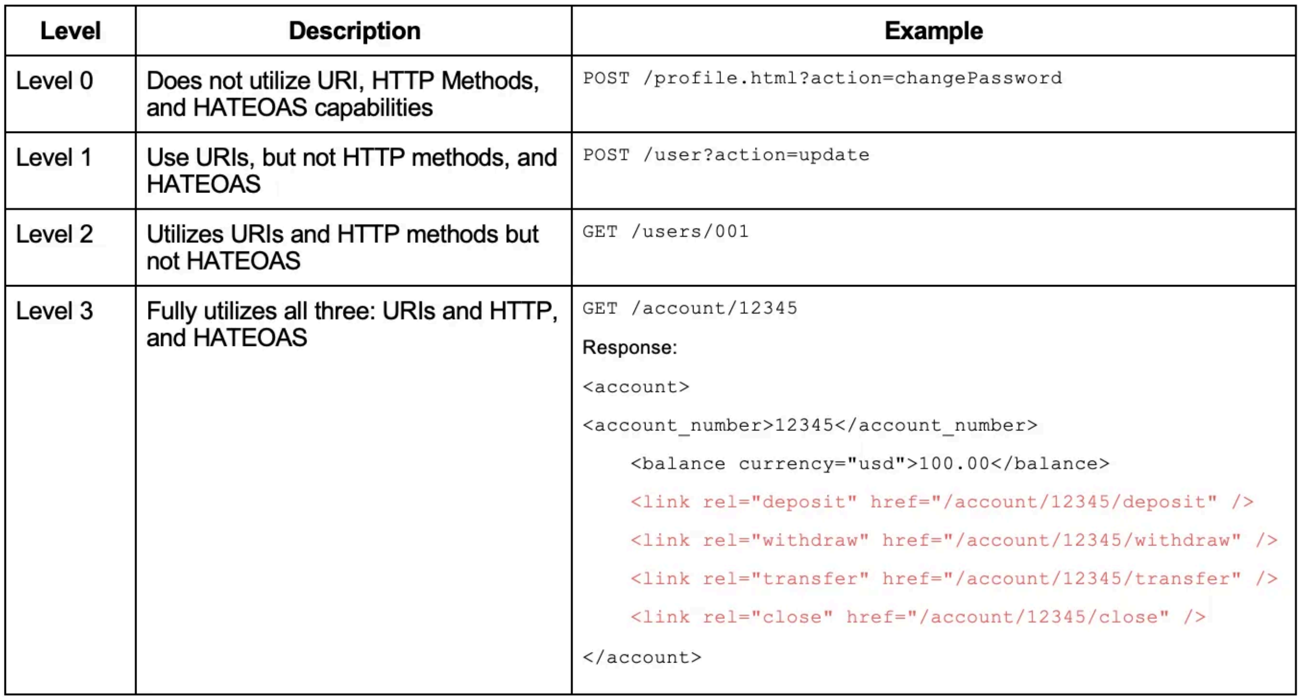
Another telltale sign can be an influx of requests for minor enhancements. If it feels like we are always tweaking and adjusting our APIs, it might suggest that they are not as robust or flexible as they need to be.

These real-world examples highlight the kind of challenges that indicate our APIs could use some fine-tuning.

Understanding API Maturity Levels

The Richardson Maturity Model (RMM) [3], introduced by Leonard Richardson in 2008, serves as a valuable tool to better understand the concept of API maturity and how well an API conforms to the REST concepts. This model helps us identify the strengths and weaknesses of our API design.

The RMM defines four levels to assess how closely an API conforms to the REST framework. The main factors that decide the maturity of a service are its URI, HTTP methods, and HATEOAS (Hypermedia as the Engine of Application State).



Level	Description	Example
Level 0	Does not utilize URI, HTTP Methods, and HATEOAS capabilities	`POST /profile.html?action=changePassword`
Level 1	Use URIs, but not HTTP methods, and HATEOAS	`POST /user?action=update`
Level 2	Utilizes URIs and HTTP methods but not HATEOAS	`GET /users/001`
Level 3	Fully utilizes all three: URIs and HTTP, and HATEOAS	`GET /account/12345` Response: `<account>` `<account_number>12345</account_number>` `<balance currency="usd">100.00</balance>` `<link rel="deposit" href="/account/12345/deposit" />` `<link rel="withdraw" href="/account/12345/withdraw" />` `<link rel="transfer" href="/account/12345/transfer" />` `<link rel="close" href="/account/12345/close" />` `</account>`

Level 0: The Swamp of POX

At Level 0, APIs use a single URI and a single HTTP method, typically POST. This approach does not leverage the true capabilities of the HTTP protocol and lacks a uniform way to interact with system resources. Martin Fowler famously called this level "The Swamp of POX (Plain Old XML)" due to its simplistic, RPC-style system.

Level 1: Resources

Level 1 introduces the concept of resources, a cornerstone of RESTful design. Each resource is uniquely identified by a URI, creating an easier way to manage and interact with different elements of a system. However, it still uses only one HTTP method, POST, limiting the full potential of REST.

Level 2: HTTP Verbs

Level 2 represents an advancement in RESTful design. The services at this level not only use unique URIs for resources but also take advantage of different HTTP methods (like GET, POST, PUT, DELETE) that correspond to operations on these resources. This approach makes our APIs more intuitive and aligns them more closely with the principles of the web. This maturity level is the most popular.

Level 3: HATEOAS

Level 3 brings in the concept of HATEOAS (Hypermedia as the Engine of Application State). HATEOAS makes our APIs self-descriptive, improving their usability and discoverability. When a client interacts with a resource, the API provides information not just about the resource itself, but also about related resources and possible actions, all represented through hypermedia links.

In the example above, when we request to query account 12345, not only do we receive the account balance ($100), but the response also guides us on the next steps and how to execute them via URIs. For instance, we could deposit more money into account 12345 by navigating to /account/12345/deposit.

The RMM offers an effective framework to help us better understand and implement RESTful principles in our API design. It is essential to remember, as we strive to improve our APIs, that Level 2 is a prerequisite for REST [3].

Basics

HTTP Verbs

When designing a system, we often encounter two layers: the web service layer, which is responsible for handling web requests, and the service layer, where the real work occurs, such as interacting with databases, handling message queues, and communicating with other services.

At the web service layer, we leverage HTTP verbs. These verbs define the operations we can perform on various resources. At the service layer, we utilize CRUD operations (Create, Read, Update, Delete) to define these operations.

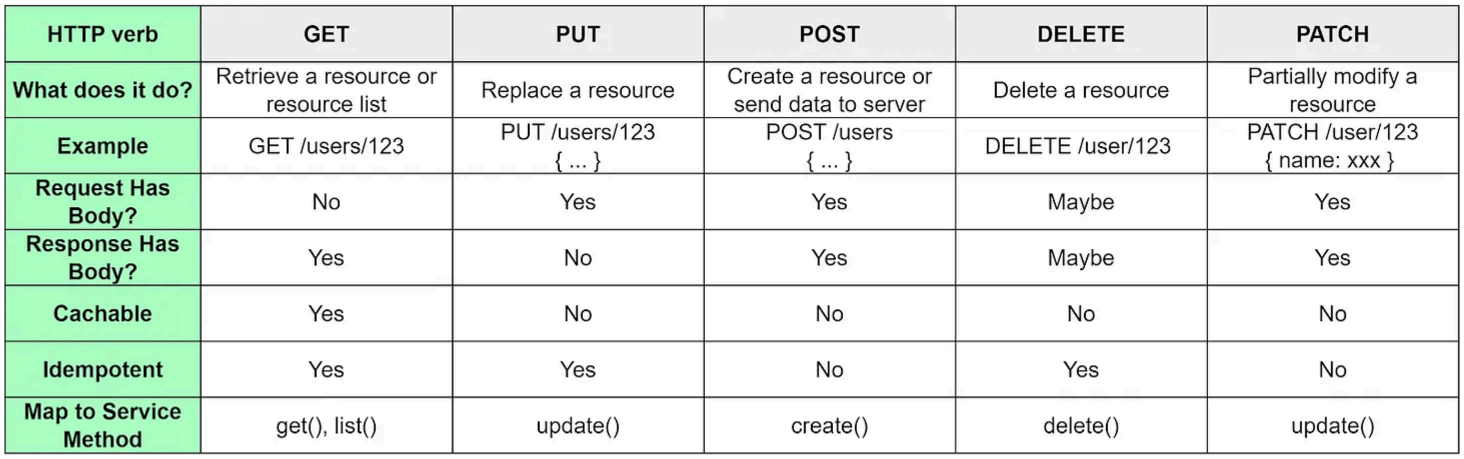
The diagram below lists the common HTTP verbs and their mappings to service methods. It is essential to understand that there isn't always a direct one-to-one mapping between HTTP verbs and service methods. For example, the GET verb can be used to retrieve a single resource or an entire list of resources. Similarly, both PUT and PATCH verbs can be used to modify a resource. However, the PATCH verb is specifically used when we want to make partial modifications to a resource.

While the five HTTP methods should satisfy most of our needs, there might be scenarios where we need to define custom operations. We have two ways to approach this:

1. Map custom operations to standard HTTP verbs. For example, we could map a *search* operation to the GET verb. However, this might lead to some confusion as the verb might

not entirely align with the operation’s purpose. It is vital to have comprehensive API documentation to clarify these mappings.

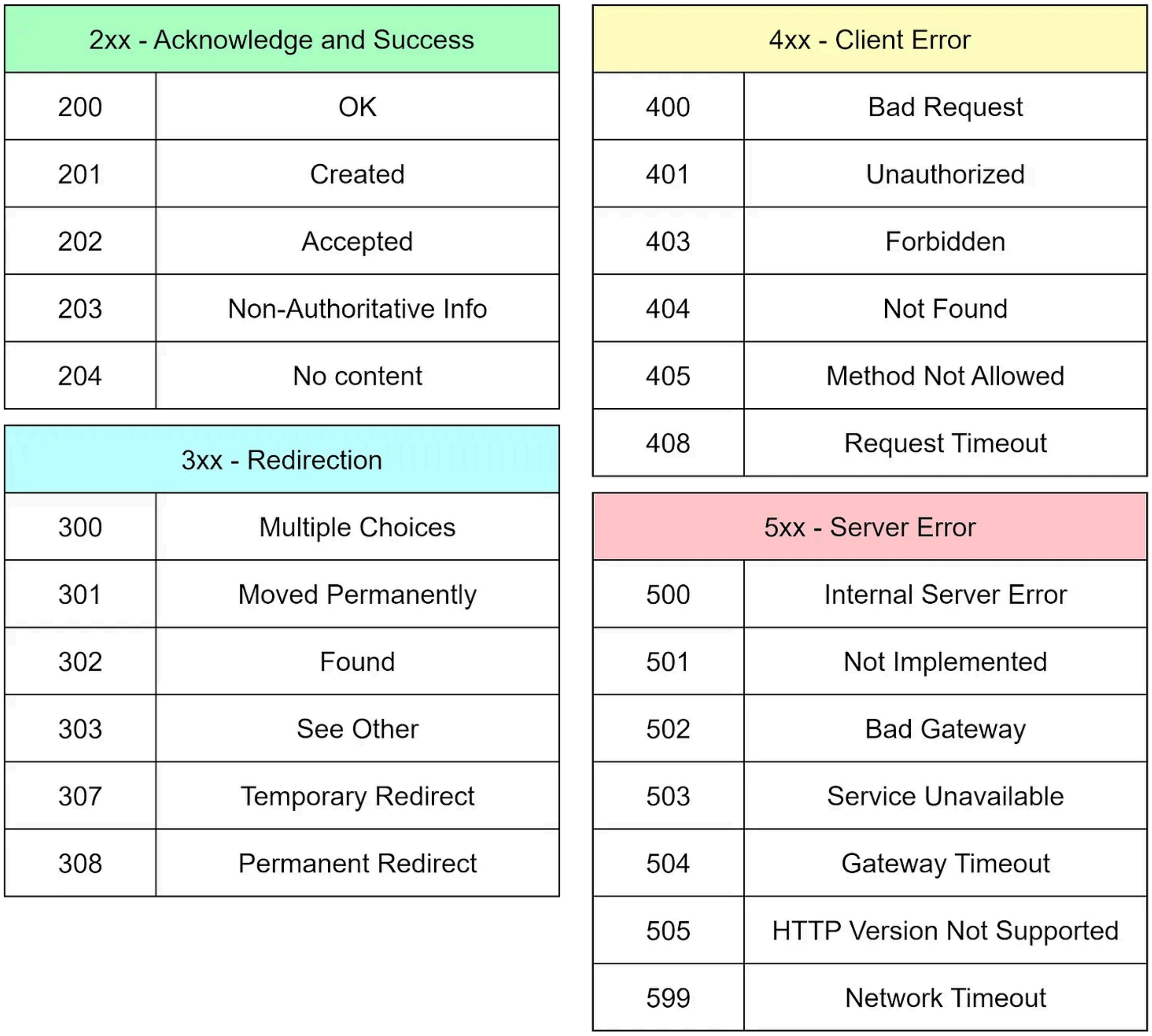
2. Define a custom HTTP method. In some cases, we might need a specific operation not covered by standard HTTP verbs. For example, in online games, we might need to reset a match. We can define a custom method, such as RESET, for this purpose.



HTTP verb	GET	PUT	POST	DELETE	PATCH
What does it do?	Retrieve a resource or resource list	Replace a resource	Create a resource or send data to server	Delete a resource	Partially modify a resource
Example	GET /users/123	PUT /users/123 { ... }	POST /users { ... }	DELETE /user/123	PATCH /user/123 { name: xxx }
Request Has Body?	No	Yes	Yes	Maybe	Yes
Response Has Body?	Yes	No	Yes	Maybe	Yes
Cachable	Yes	No	No	No	No
Idempotent	Yes	Yes	No	Yes	No
Map to Service Method	get(), list()	update()	create()	delete()	update()

Status Codes

REST is built on the HTTP protocol. Therefore, our APIs should use HTTP status codes to ensure consistent and predictable behavior. The diagram below shows some common HTTP status codes.



2xx - Acknowledge and Success	
200	OK
201	Created
202	Accepted
203	Non-Authoritative Info
204	No content

3xx - Redirection	
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
307	Temporary Redirect
308	Permanent Redirect

4xx - Client Error	
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
405	Method Not Allowed
408	Request Timeout

5xx - Server Error	
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported
599	Network Timeout

However, HTTP status codes are not exhaustive for all the possible outcomes of an API call. We should also establish our own set of internal status codes or error codes to communicate specific service-related issues. One efficient way of managing these codes is by maintaining a common library for all the codes in the system. It facilities easy registration and sharing across different code repositories.

For example, let's assume the user service uses message codes in the range of 50000 to 59999, while the shopping cart service uses codes from 60000 to 69999. We can wrap these message details in an HTTP response and send them back to the clients. This practice greatly benefits customer service, as they can easily identify issues based on the returned code.

A golden rule we should always follow is to return a code for every response, even if it is a timeout. Adhering to this rule ensures predictable and consistent behavior. It is crucial for maintaining a high-quality service.