**14 Best Practices for Designing RESTful APIs | .NET Core Web API**

In today’s interconnected environment, RESTful APIs serve as the backbone of most modern applications. Virtually all major cloud-based services, such as Amazon, Twitter, Facebook, Google, and YouTube, rely on REST APIs. Companies across various industries, including BestBuy, Target, Macy’s, Walmart, and Walgreens, incorporate REST APIs into their B2B and B2C transactions.

In this article, I’ll explore best practices for designing RESTful APIs to ensure clarity, maintainability, and ease of use.😍🧑‍💻

**Table of Contents🫙**

RESTful web API design  
What is the REST API?  
#1: Semantics of HTTP methods  
#2: API Versioning  
#3: Plural Nouns  
#4: Accept and respond with JSON  
#5: Use Nouns  
#6: Nesting resources  
#7: Return Status Code  
#8: Allow filtering, sorting, and pagination  
#9: Security Best Practices  
#10: Caching  
#11: Use HATEOAS  
#12: API Documentation  
#13: Rate limiting  
#14: Authentication and Authorization

**RESTful web API design**

In the realm of modern web development, the importance of a well-designed and robust web API cannot be overstated. A successful API should not only facilitate seamless communication between clients and servers but also adhere to certain principles that ensure ***platform independence***, ***service evolution***, and ***overall usability***.

**1. Platform Independence**

A fundamental principle of RESTful API design is platform independence. This means that any client, regardless of its technology stack or implementation details, should be able to interact with the API. Achieving this independence involves the following considerations:

**Standard Protocols:**Use standard protocols such as HTTP/HTTPS for communication. This allows clients built on diverse platforms, including web browsers, mobile devices, and server applications, to seamlessly consume the API.

**Data Exchange Format:**Establish a clear mechanism for data exchange between clients and the API. JSON (JavaScript Object Notation) has become the de facto standard due to its simplicity, human-readability, and broad support across various programming languages.

**Consistent Resource Naming:**Adopt a consistent and meaningful naming convention for resources (e.g., URIs) that is independent of the underlying implementation. This ensures clients can easily navigate and interact with the API without being tightly coupled to server details.

**2. Service Evolution**

A well-designed Web API should be capable of evolving independently of client applications. This principle is crucial to ensuring that the API can introduce new features or modify existing ones without disrupting existing client functionality.

**Versioning:**Implement a versioning strategy to manage changes in the API. This allows different versions to coexist, enabling clients to migrate gradually and giving them time to adapt to changes.

**Backward Compatibility:**Ensure backward compatibility whenever possible. Existing clients should continue to function seamlessly, even as the API undergoes updates. Avoid making changes that would necessitate modifications in all existing client applications.

**Discoverability:**Make all API functionality discoverable. Clients should be able to explore available resources, actions, and representations dynamically. This is often achieved through consistent resource linking and documentation.

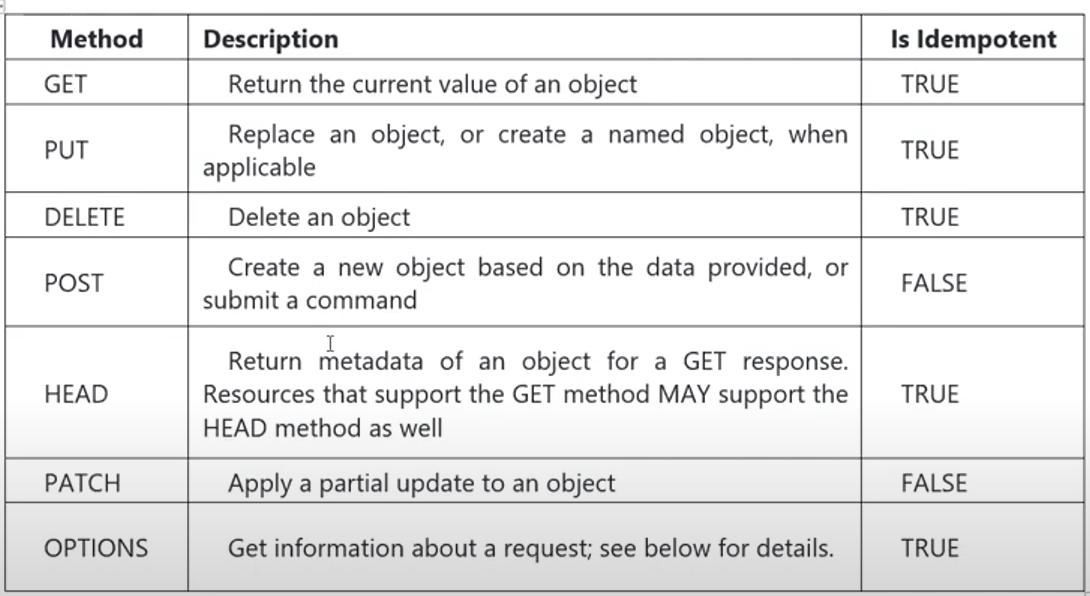
**Note:** *This guidance describes issues that you should consider when designing a web API.*

🔗Read this article to understand [**What is the REST API?**](https://medium.com/@jeslurrahman/understand-the-web-rest-api-asp-net-core-web-api-in-c-8236e2bcb0f1)

**The constraints of the REST architectural style affect the following architectural properties:**

* Performance in component interactions can be the dominant factor in user-perceived performance and network efficiency.
* Scalability**,** allowing the support of large numbers of components and interactions among components;
* The simplicity of a uniform interface.
* Modifiability of components to meet changing needs (even while the application is running).
* visibility of communication between components by service agents.
* portability of components by moving program code with the data.
* Reliability is the resistance to failure at the system level in the presence of failures within components, connectors, or data.

**#1: Semantics of HTTP methods**



**GET methods**  
A successful GET method typically returns HTTP status code 200 (0K). If the resource cannot be found, the method should return 404 (Not Found).  
  
**POST methods**  
If a POST method creates a new resource, it returns HTTP status code 201 (Created). The URI of the new resource is included in the Location header of the response. The response body contains a representation of the resource.

If the method does some processing but does not create a new resource, the method can return HTTP status code 200 and include the result of the operation in the response body. Alternatively, if there is no result to return, the method can return HTTP status code 204 (No Content) with no response body.

If the client puts invalid data into the request, the server should return HTTP status code 400 (Bad Request). The response body can contain additional information about the error or a link to a URI that provides more details.

**PUT methods**  
If a PUT method creates a new resource, it returns HTTP status code 201 (Created), as with a POST method. If the method updates an existing resource, it returns either 2OO (0K) or 204 (No Content). In some cases, it might not be possible to update an existing resource. In that case, consider returning HTTP status code 409 (Conflict).

Consider implementing bulk HTTP PUT operations that can batch updates to multiple resources in a collection. The PUT request should specify the URI of the collection, and the request body should specify the details of the resources to be modified. This approach can help to reduce chattiness and improve performance.

**PATCH methods**  
With a PATCH request, the client sends a set of updates to an existing resource, in the form of a patch document. The server processes the patch document to perform the update. The patch document doesn’t describe the whole resource, only a set of changes to apply.

**DELETE methods**  
If the delete operation is successful, the web server should respond with HTTP status code 204 (No Content), indicating that the process has been successfully handled, but that the response body contains no further information. If the resource doesn’t exist, the web server can return HTTP 404 (Not Found).

**#2: API Versioning**

A diagram of a computer program

Description automatically generated with medium confidence

Versioning is one of the REST API's best development practices as it allows developers to introduce any changes in the data structure or specific actions. As your project increases in size and passing time, you might manage more than only one API version. But, on the upside, this will allow you to introduce more modifications and improvements in your service while holding a part of your API users that are reluctant to change or slow in adapting to the new changes.  
  
**No Versioning**  
This is the simplest approach and may be acceptable for some internal APIs. Significant changes could be represented as new resources or new links. Adding content to existing resources might not present a breaking change as client applications that are not expecting to see this content will ignore it.  
For example, a request to the URI  
**https://adventure works.com/customers/3**

**URI versioning**  
Each time you modify the web API or change the schema of resources, you add a version number to the URI for each resource. The previously existing URIs should continue to operate as before, returning resources that conform to their original schema.

**/api/vl/feedbacks  
/api/vl/feedbacks/{id}  
/api/v2/feedbacks  
/api/v2/feedbacks/{id}**

**Query string versioning**  
Rather than providing multiple URIS, you can specify the version of the resource by using a parameter within the query string appended to the HTTP request, such as [https://api.com/customers/3?version](https://api.com/customers/3?version=)

The version parameter should default to a meaningful value such as 1 if it is omitted by older client applications.

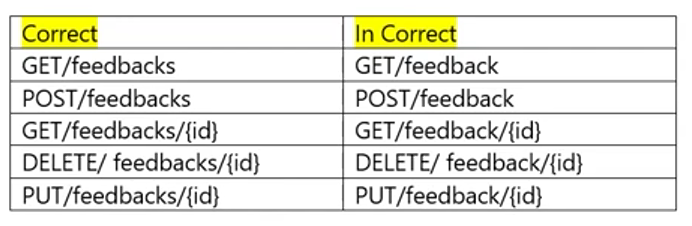
**Header versioning**  
Rather than appending the version number as a query string parameter, you could implement a custom header that indicates the version of the resource. This approach requires that the client application adds the appropriate header to any requests, although the code handling the client request could use a default value (version 1) if the version header is omitted. The following examples use a custom header named Custom-Header. The value of this header indicates the version of the web API.

**GET https://api.com/customers/3 HTTP/ 1.1**  
**Custom-Header. am-version= 1**

**#3: Plural Nouns**

Name the collection using Plural Nouns

As a generally accepted norm, REST API developers use plural nouns for collections. It is one of the REST service's best practices that makes it easier for consistency in global code sharing and for normal people to understand that those clubbed APIs are a collection



**#4: Accept and respond with JSON**

JSON (JavaScript Object Notation) is the easiest programming language and format to read and use. However, still, not all APIs make use of JSON. It is considered one of the REST API development best practices. Many developers are replacing or substituting their existing SOAP (Simple Object Access Protocol) in favor of REST.

**Key Features of JSON —**  
✔️Easy to parse  
✔️Support for most frameworks  
✔️Can be used by any programming language

REST APIs should accept JSON for request payload and also send responses to JSON. JSON is the standard for transferring data. Almost every networked technology can use it

To make sure that when our REST API app responds with JSON clients interpret it as such, we should set ***Content-Type*** in the response header to ***application/json***after the request is made

**#5: Use Nouns**

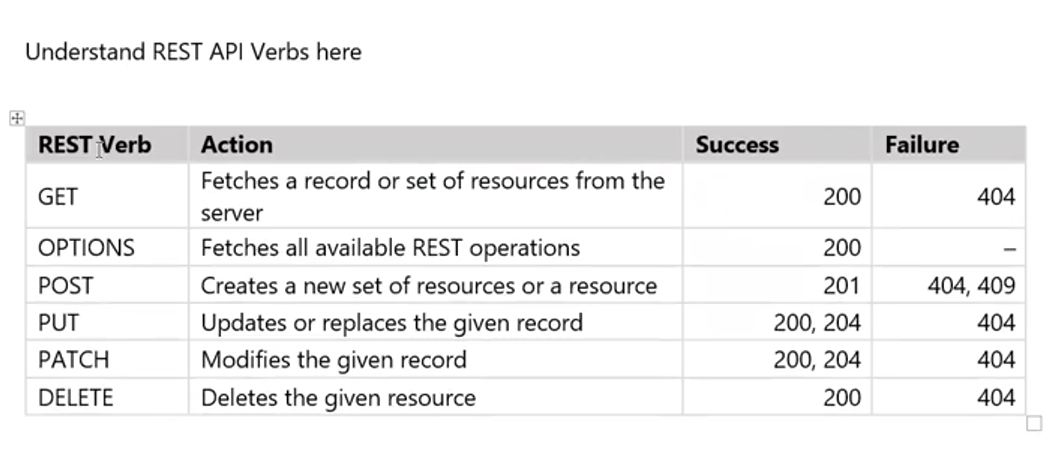
Use nouns instead of verbs in endpoint paths

We shouldn’t use verbs in our endpoint paths. Instead, we should use the nouns that represent the entity that the endpoint that we’re retrieving or manipulating as the pathname.

This is because our HTTP request method already has the verb. Having verbs in our API endpoint paths isn’t useful and it makes it unnecessarily long since it doesn’t convey any new information.

The action should be indicated by the HTTP request method that we’re making. The most common methods include GET, POST, PUT, and DELETE.

✔️GET retrieves resources.  
✔️POST submits new data to the server.  
✔️PUT updates existing data.  
✔️DELETE removes data.



**#6: Nesting resources**

**Nesting resources for hierarchical objects**  
When designing endpoints, it makes sense to group those that contain associated information. That is, if one object can contain another object, you should design the endpoint to reflect that. This is good practice regardless of whether your data is structured like this in your database. It may be advisable to avoid mirroring your database structure in your endpoints to avoid giving attackers unnecessary information.  
For example:**/api/feedbacks/12/comments**

**#7: Return Status Code**

Handle errors gracefully and return standard error codes

To eliminate confusion for API users when an error occurs, we should handle errors gracefully and return HTTP response codes that indicate what kind of error occurred. We don’t want errors to bring down our system, so we can leave them unhandled, which means that the API consumer has to handle them.

Common error HTTP status codes include:  
**✅ 200 0K —** 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)  
**✅ 400 Bad Request —**This means that client-side input fails validation.  
**✅ 401 Unauthorized —** This means the user isn’t authorized to access a resource. It usually returns when the user isn’t authenticated.  
**✅403 Forbidden —** This means the user is authenticated, but it’s not allowed to access a resource.  
**✅404 Not Fouhd** — This indicates that a resource is not found.  
**✅422 Unprocessable Entity —** This should be used if the server cannot process the entity, e.g. if an image cannot be formatted or mandatory fields are missing in the payload.  
**✅500 Internal server error —** This is a generic server error. It probably shouldn’t be thrown explicitly.  
**✅502 Bad Gateway —**This indicates an invalid response from an upstream server.  
**✅503 Service Unavailable —**This indicates that something unexpected happened on the server side (It can be anything like server overload, some parts of the system failed, etc.).

**#8: Allow filtering, sorting, and pagination**

The databases behind a REST API can get very large. Sometimes, there’s so much data that it shouldn’t be returned all at once because it’s way too slow or will bring down our systems. Therefore, we need ways to filter items.

We also need ways to paginate data so that we only return a few results at a time. We don’t want to tie up resources for too long by trying to get all the requested data at once.

Filtering and pagination both increase performance by reducing the usage of server resources. As more data accumulates in the database, the more important these features become.

Here’s a small example where an API can accept a query string with various query parameters to let us filter out items by their fields:

REST API offers different types of filtering options  
🟥Filtering  
🟧Sorting  
🟨Paging  
🟩Field Selection  
Filtering, sorting, paging, and field selection are important features for efficiently using REST API.

🟥**Filtering**  
It helps narrow down results using specific search parameters like creation date, country, and more.  
✔️**GET /users?country=uk**  
✔️**GET /users?creation\_date=2024–01–15**

🟧 **Sorting**  
Allows you to sort the results in ascending or descending format by your preferred parameter or parameters like dates.  
**✔️GET /users?sort=birthdate date:asc**  
✔️**GET /users?sort=birthdate date:desc**

🟨**Paging**  
RESTful APIs that return collections MAY return partial sets. Consumers of these services MUST expect partial result sets and correctly page through to retrieve an entire set.

Two forms of pagination MAY be supported by RESTful APIs. Server-driven paging mitigates denial-of-service attacks by forcibly paginating a request over multiple response payloads. Client-driven paging enables clients to request only the number of resources that they can use at a given time.

Sorting and Filtering parameters MUST be consistent across pages because both client- and server-side paging are fully compatible with both filtering and sorting.

**Server-driven paging**  
Paginated responses MUST indicate a partial result by including a continuation token in the response. The absence of a continuation token means that no additional pages are available.

**Client-driven paging**  
Clients MAY use Stop and Sskip query parameters to specify a number..of results to return and an offset into the collection.

The server SHOULD honor the values specified by the client; however, clients MUST be prepared to handle responses that contain a different page size or contain a continuation token.

When both Stop and Sskip are given by a client, the server SHOULD first apply Sskip and then Stop on the collection.

**Note:**If the server can’t honor Stop and/or skip, the server MUST return an error to the client informing about it instead of just ignoring the query options. This will avoid the risk of the client making assumptions about the  
data returned.

🟩**Field Selection**  
Field selection is a convenient REST API development function that allows developers to request only a specific part of the available data for a particular object. Hence, if the thing you are querying about has too many fields and you need only a few specific ones, you can use field selection to specify the ones you want to include in the response.

For one specific user  
✔️**GET/ users/123?fields=name,birthdate,email**

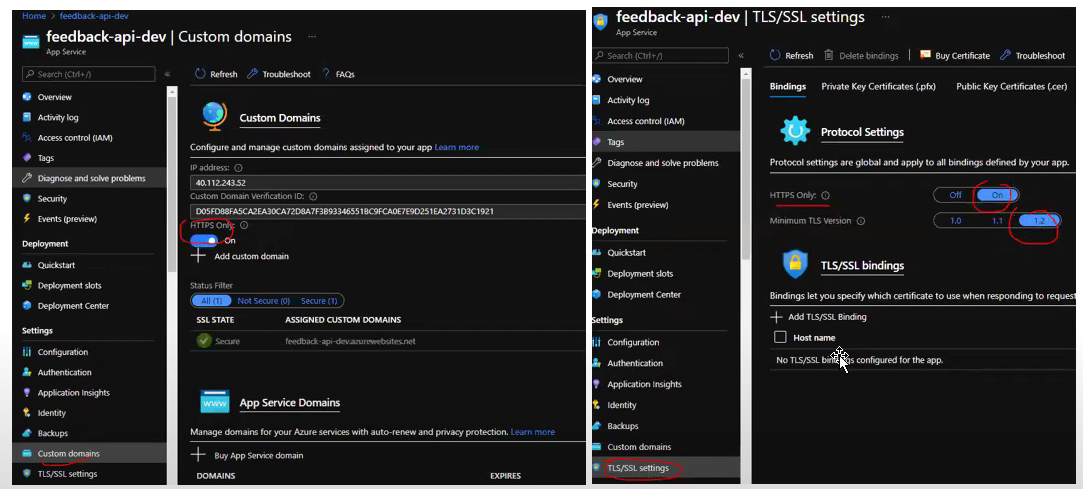
For the full list of users  
✔️**GET/ users?fields=name,birthdate,email**

**#9: Security Best Practices**

Maintain Good Security Practices

Most communication between client and server should be private since we often send and receive private information. Therefore, using SSL/TLS for security is a must.

**A SSL certificate** isn’t too difficult to load onto a server and the cost is free or very low. There’s no reason not to make our REST APIs communicate over secure channels instead of in the open.



**Authorization -**People shouldn’t be able to access more information than they requested.  
To enforce the principle of least privilege, we need to add role checks either for a single role or have more granular roles for each user.

**#10: Caching**

Cache data to improve performance

We can add caching to return data from the local memory cache instead of querying the database to get the data every time we want to retrieve some data that users request. The good thing about caching is that users can get data faster.

However, the data that users get may be outdated. This may also lead to issues in production environments when something goes wrong as we keep seeing old data.

There are many kinds of caching solutions like Redis, in-memory caching, and more. We can change the way data is cached as our needs change.

**#11: Use HATEOAS**

Use HATEOAS to enable navigation to related resources

Hypermedia as the Engine of Application State (HATEOAS)

One of the primary motivations behind REST\ is that it should be possible to navigate the entire set of resources without requiring prior knowledge of the URI scheme.

Each HTTP GET request should return the information necessary to find the resources related directly to the requested object through hyperlinks included in the response, and it should also be provided with information that describes the operations available on each of these resources. This principle is known as HATEOAS, or Hypertext as the Engine of the Application State. The system is effectively a finite state machine, and the response to each request contains the information necessary to move from one state to another; no other information should be necessary.

**#12: API Documentation**

When you make a REST API, you need to help clients (consumers) learn and figure out how to use it correctly. The best way to do this is by providing good documentation for the API.

The documentation should contain:  
✔️relevant endpoints of the API  
✔️example requests of the endpoints  
✔️implementation in several programming languages  
✔️messages listed for different errors with their status codes

One of the most common tools you can use for API documentation is Swagger. You can also use Postman, one of the most common API testing tools in software development, to document your APIs.

**#13: Rate limiting**

To prevent abuse, it is a standard practice to implement rate limiting in an API, commonly signaled by the HTTP status code 429 Too Many Requests. However, proactively notifying the consumer about their limits before reaching them can be highly beneficial. Although this area lacks universal standards, there are several popular conventions involving the use of HTTP response headers.

**#14: Authentication / Authorization**

For a RESTful API to adhere to the stateless principle, authentication should not rely on cookies or sessions. Instead, each request should include explicit authentication credentials to ensure a secure and state-independent communication process.

**Conclusion**

In conclusion, the key to designing high-quality REST APIs lies in maintaining consistency through adherence to web standards and conventions. JSON, SSL/TLS, and HTTP status codes serve as foundational elements for modern web development.

Performance is a critical consideration. Limiting the amount of data returned and implementing caching strategies can significantly enhance response times and overall efficiency.

Ensure that endpoint paths follow a consistent pattern, using nouns to represent resources. Nesting resource paths logically after the parent resource contributes to a clear and intuitive API structure. The goal is to provide meaningful insights into the API’s functionality without the need for excessive documentation.