**REST API Design Best Practices**

Document Information

Document Title: REST API Design Best Practices

Version: 1.1

Date: 23/11/2023

Table of Contents

Introduction

1.1 Purpose

1.2 Scope

1.3 Audience

API Development Guidelines

2.1 RESTful API Best Practices

2.1.1 Use JSON as the Format for Sending and Receiving Data

2.1.2 Use resource nesting to show relations or hierarchy

2.1.3 Name Collections with Plural forms

2.1.4 Use SSL for Security

2.1.5 Handling exceptions and return a meaningful response to clients

2.2 Versioning

2.3 Error Handling

2.4 Endpoint Naming Conventions

2.5 Name abbreviation

2.5 Rate Limiting

2.6 Data Validation

API Testing Guidelines

3.1 Tools that we have used for testing APIs

3.2 Unit Testing

3.3 Functional Testing

3.4 Performance Testing

3.5Test Data

3.6 Reference

**Document Revision History**

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Description** |
| 1.0 | 21/11/2013 | Initial version |
| 1.1 | 23/11/2023 | Updated version |

# Introduction

**1.1 Purpose**

The purpose of this document to help developer & tester about how to design REST API, what are the main aspects while designing as well as testing APIs. Also handle exceptions with meaningful response to clients that will help developer exactly what went wrong and how to fix it. It will also reduce development, debugging time.

**1.2 Scope**

This document contains API development & testing guidelines. Also how to maintain versioning of API, error handling with meaningful response to clients.It contains different types of testing as functional testing, performance testing, unit testing etc.

**1.3 Audience**

· Development Team

· Testing Team

**API Development Guidelines**

* **What is a REST API?**

REST stands for Representational State Transfer. REST relies on a client/server approach that separates front and back ends of the API and provides considerable flexibility in development and implementation.

In other words, I should be able to make changes to my mobile application without impacting either the data structure or the database design on the server. At the same time, I should be able to modify the database or make changes to my server application without impacting the mobile client .REST is stateless, which means the API stores no data or status between requests.

Simply put, a REST API is a medium for two computers to communicate over HTTP (Hypertext Transfer Protocol), in the same way clients and servers communicate.

* 1. **REST API Design Best Practices**

**1. Use JSON as the Format for Sending and Receiving Data**

XML represents data in a tree pattern, while JSON uses key-value pairs JSON has come to be one of the most popular standards for data interchange, being easy for humans to read while being lightweight to ensure small transmission size.

This is because, with XML for example, it's often a bit difficult to decode and encode data – so XML isn’t widely supported by frameworks anymore. XML uses end tags, which makes it longer than JSON.

Example: JSON Format

{

"clientID":"hello",

"sLocale":"eng",

"tLocale":"ori",

"sText":" 4th Floor, CDAC, near Dominos , Aundh, Pune, MH"

}

**2. Use resource nesting to show relations or hierarchy**

Resource objects have some kind of functional hierarchy or are related to each other. For example in the online store, we have ‘users’ and ‘orders’. Orders always belong to some user. Also in case of gistlangserver we have leveraging & GetSupportedLocales.this api is having hierarchy/resource nesting & designed according to this guideline.

<http://gistlangserver.in/api/Leveraging/GetSupportedLocales>

http://gistlangserver.in/api/Leveraging/GetLeverage

**3. Name Collections with Plural forms**

When you have to develop the collection in REST API, just go with plural nouns. It makes it easier for humans to understand the meaning of collection without actually opening it. Let’s go through this example:

GET /cars/123

It is clear from the example that ‘car’ is referred to as number 123 from the entire list of "cars". The usage of a plural noun is indicating that this is a collection of different cars.

GET /car/123

This example doesn’t clearly show whether there is more than one car in the system or not. For a human reader, it might be challenging to understand, as well.

**4. Use SSL for Security**

SSL stands for secure socket layer. This will secure your API and make it less vulnerable to malicious attacks.

The clear difference between the URL of a REST API that runs over SSL and the one which does not is the “s” in HTTP:

<https://gistlangserver3.in/api/Service/GetLongestStringMatch> runs on SSL.

<http://gistlangserver3.in/api/Service/GetLongestStringMatch> does not run on SSL

**5. Handling exceptions and return a meaningful response to clients:**

The exception should also include the appropriate HTTP status code rather than simply returning status code 500 for every situation. It is convenient for the API to return error details in the JSON or response body to help a user.

{

"error": {

"id": "firstName",

"status": "400",

"title": "User first name cannot be empty",

"detail": "The first name field is required to have a value",

"href":"http://docs.domain.ext/users/post"

}}

**2.2 Versioning**

REST APIs should have different versions, so you don’t force clients (users) to migrate to new versions. This might even break the application if you're not careful.

An example of semantic versioning is 1.0.0, 2.1.2, and 3.3.4. The first number represents the major version, the second number represents the minor version, and the third represents the patch version.

Example:

[http://gistlangserver3.in/api/leveraging/GetULCAMT/3.0/](https://us6.api.mailchimp.com/3.0/) (major + minor version indication)

[http://gistlangserver3.in/api/leveraging/GetULCAMT/v1/](https://api.stripe.com/v1/) (major version 1 indication only)

[http://gistlangserver3.in/api/leveraging/GetULCAMT/2010-04-01/](https://api.twilio.com/2010-04-01/) (date based indication)

**Versioning of REST API**

1. **Versioning through URI Path**

e.g. <http://gistlangserver3.in/api/1/leveraging/GetULCAMT>

The internal version of the API uses the **1.2.3** format, so it looks as follows:

**MAJOR.MINOR.PATCH**

* **Major version:** The version used in the URI and denotes breaking changes to the API.
* **Minor and Patch versions**: These are transparent to the client and used internally for backward-compatible updates. They are usually communicated in change logs to inform clients about a new functionality or a bug fix.

1. **Versioning through Query Parameters**

e.g.http://gistlangserver3.in/api/leveraging/GetULCAMT?version=1

Another option for versioning a REST API is to include the version number as a query parameter.

This is a straightforward way of versioning an API from an implementation point of view.

* **Pros:** It’s a straightforward way to version an API, and it’s easy to default to the latest version

**2.3 Use Status Codes in Error Handling**

You should always use regular HTTP status codes in responses to requests made to your API. This will help your users to know what is going on – whether the request is successful, or if it fails, or something else. So, in API development, correct error handling should be considered as one of the best practices as well as testing API as the sooner a problem is detected.

Below is a table showing different HTTP Status Code ranges and their meanings:

**STATUS CODE RANGE** **MEANING**

100 – 199 **Informational Responses.**

For example, 102 indicates the resource is being processed

300 – 399 **Redirects**

For example, 301 means Moved permanently

400 – 499 **Client-side errors**

400 means bad request and 404 means resource not found

500 – 599 **Server-side errors**

For example, 500 means an internal server error

**Error handling Code**

|  |  |  |
| --- | --- | --- |
| **HTTP** | **Error** | **Description** |
| 200 | OK | No error. |
| 400 | INVALID\_ARGUMENT | Client specified an invalid argument. Check error message and error details for more information. |
| 400 | FAILED\_PRECONDITION | Request can not be executed in the current system state, such as deleting a non-empty directory. |
| 400 | OUT\_OF\_RANGE | Client specified an invalid range. |
| 401 | UNAUTHENTICATED | Request not authenticated due to missing, invalid, or expired OAuth token. |
| 403 | PERMISSION\_DENIED | Client does not have sufficient permission. This can happen because the OAuth token does not have the right scopes, the client doesn't have permission, or the API has not been enabled. |
| 404 | NOT\_FOUND | A specified resource is not found. |
| 409 | ABORTED | Concurrency conflict, such as read-modify-write conflict. |
| 409 | ALREADY\_EXISTS | The resource that a client tried to create already exists. |
| 429 | RESOURCE\_EXHAUSTED | Either out of resource quota or reaching rate limiting. The client should look for google.rpc.QuotaFailure error detail for more information. |
| 499 | CANCELLED | Request cancelled by the client. |
| 500 | DATA\_LOSS | Unrecoverable data loss or data corruption. The client should report the error to the user. |
| 500 | UNKNOWN | Unknown server error. Typically a server bug. |
| 500 | INTERNAL | Internal server error. Typically a server bug. |
| 501 | NOT\_IMPLEMENTED | API method not implemented by the server. |
| 502 | N/A | Network error occurred before reaching the server. Typically a network outage or misconfiguration. |
| 503 | UNAVAILABLE | Service unavailable. Typically the server is down. |
| 504 | DEADLINE\_EXCEEDED | Request deadline exceeded. This will happen only if the caller sets a deadline that is shorter than the method's default deadline (i.e. requested deadline is not enough for the server to process the request) and the request did not finish within the deadline. |

**2.4 Endpoint Naming Conventions**

When you're designing a REST API, you should not use verbs in the endpoint paths. The endpoints should use nouns, signifying what each of them does.

This is because HTTP methods such as GET, POST, PUT, PATCH, and DELETE are already in verb form for performing basic CRUD (Create, Read, Update, and Delete) operations.

GET, POST, PUT, PATCH, and DELETE are the commonest HTTP verbs.

For example:

http://gistlangserver.in/api/leveraging/GetWebsiteSettings

GET is only used to request data from a specified resource.

POST is used to send data to a server to create/update a resource.

DELETE removes the specified resource.

PUT means insert, replace if already exists.

PATCH is used to make a partial update to an existing resource. PATCH means replace only specified fields.

**2.6 Name abbreviation**

For well-known name abbreviations among software developers, such as config and spec, the abbreviations should be used in API definitions instead of the full spelling. This will make the source code easy to read and write. In formal documentations, the full spelling should be used. Examples:

config (configuration)

id (identifier)

spec (specification)

stats (statistics)

* 1. **Rate Limiting**

Rate Limiting is a technique used to control the rate of incoming requests to an API.

In API Connect, rate limits can be defined as unlimited, or with a specified number of calls per second, minute, hour, day, or week.

For example, let’s say a developer only wants to allow a client to call the API a maximum of 10 times per minute. In this case, the developer would apply a rate limit to their API expressed as “10 requests per 60 seconds”. This means that the client will be able to successfully call the API up to 10 times within any 60-second interval. After that, if they call the API an 11th time within that time frame, the user will get an error stating they have exceeded their rate limit.

**Private APIs:**

Private APIs are designed for internal use. The goal is to communicate services inside the company. For example DOLR api developed only for Department of Land Record which cannot be used by anyone except DOLR.

e.g.: <https://localhost:44345/transliteration.aspx>?

<http://transcoretest.in/getsingleresult.aspx>?

**Public APIs:**

Public APIs are exposed to anyone in the internet. A public API is open and available for use by any outside developer or business. These are also called open APIs or external APIs. Public

**API Testing Guidelines**

**3.1Tools that we have used for testing APIs:**

1. **Postman**

Postman is a popular tool for API testing that allows developers to create and execute HTTP requests and test API responses.

1. **Jmeter**

The Apache JMeter application is open source software used for Performance & Stress testing. In Jmeter we can save the summary of test report in csv file.

**Performance testing**: It determines how many requests it can handle per unit time.

**Stress testing**: evaluates the response time and throughput with an increasing number of users sending requests to the server.

* During a Load test, it is important to know the health of the servers we performed server monitoring. We can monitor CPU, Memory utilization of the server.

**3.2 Unit Testing**

Unit testing is a type of software testing that focuses on individual units or components of a software system. The purpose of unit testing is to validate that each unit of the software works as intended and meets the requirements. Unit testing is typically performed by developers, and it is performed early in the development process. This allows developers to quickly identify and fix any issues early in the development process.

**3.3 Functional Testing**

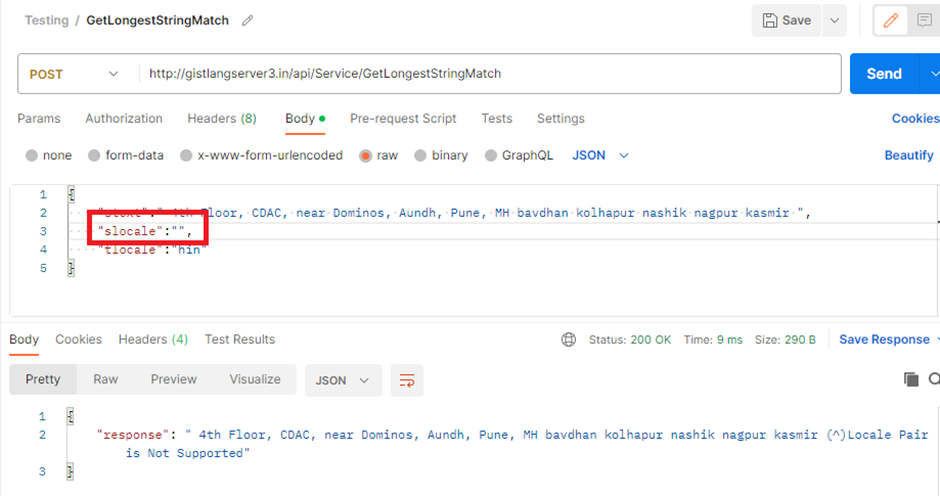
API functional testing used to validate if the API is functioning as expected and can handle different data types, requests, and responses. Using the test results, developers can easily identify and fix bugs or issues in the API.

In the case of the gistlangserver, you need to identify the different parameters, and expected response formats.

Tested RESTful API functionality using the Postman App. Postman can support multiple parameters, including GET, POST, PUT, and DELETE. Testing is done by entering the GET parameter to display some data. Then Postman will display the results in the form of response time. GET function can function properly and the response time obtained for displaying data is 137 ms. Similarly testing will be carried out for remaining methods.

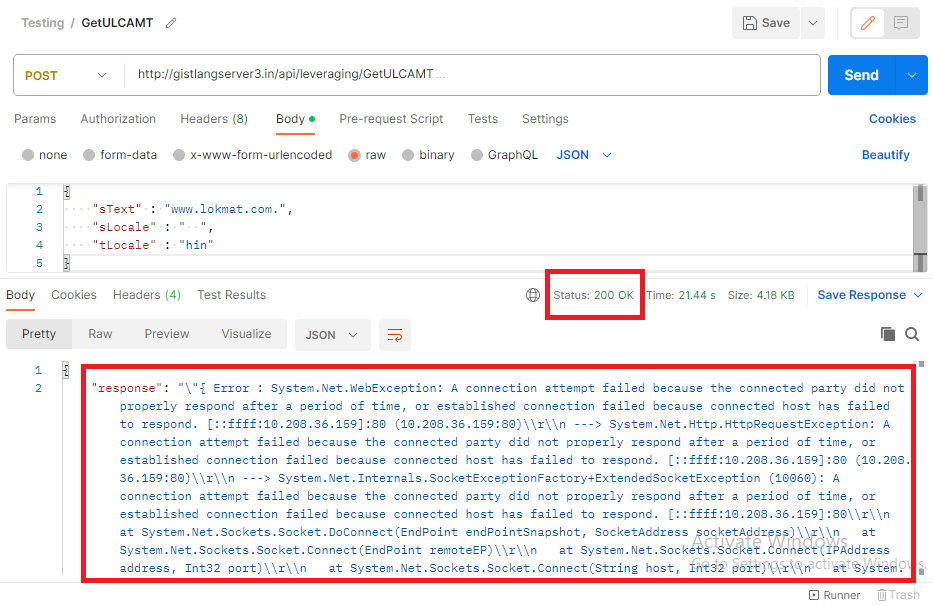
Also check for language supported. For example gistlangserver api designed for 27 languages so we checked input & output language’s accordingly.

Some fields are required so we kept these fields empty & check result accordingly. Also check response error.as shown in below we kept slocale field empty & still get output for api:[**http://gistlangserver3.in/api/Service/GetLongestStringMatch**](http://gistlangserver3.in/api/Service/GetLongestStringMatch)



Also check response & error codes. In some API Status code showing 200 but we are getting invalid response as shown below.

[**http://gistlangserver3.in/api/leveraging/GetULCAMT**](http://gistlangserver3.in/api/leveraging/GetULCAMT)



If a date must be represented as a string, it should be in the date format YYYY-MM-DD, e.g. 2014-07-30.

If a time of day must be represented as a string, it should be in the 24-hour time format HH:MM:SS [.FFF], e.g. 14:55:01.672.

**3.4 Performance Testing**

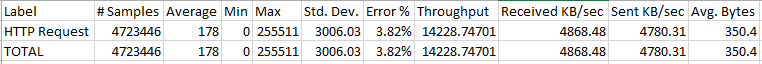
Performance Testing ensures software applications to perform under the certain workload. It determines how many requests it can handle per unit time. It also determines the average response time, percentage of failed requests.

We used Jmeter for Performance testing of API. For that we kept http request for certain number of users in Thread group for specific time & check the response. Evaluate the response time and throughput with an increasing number of users sending requests to the server.

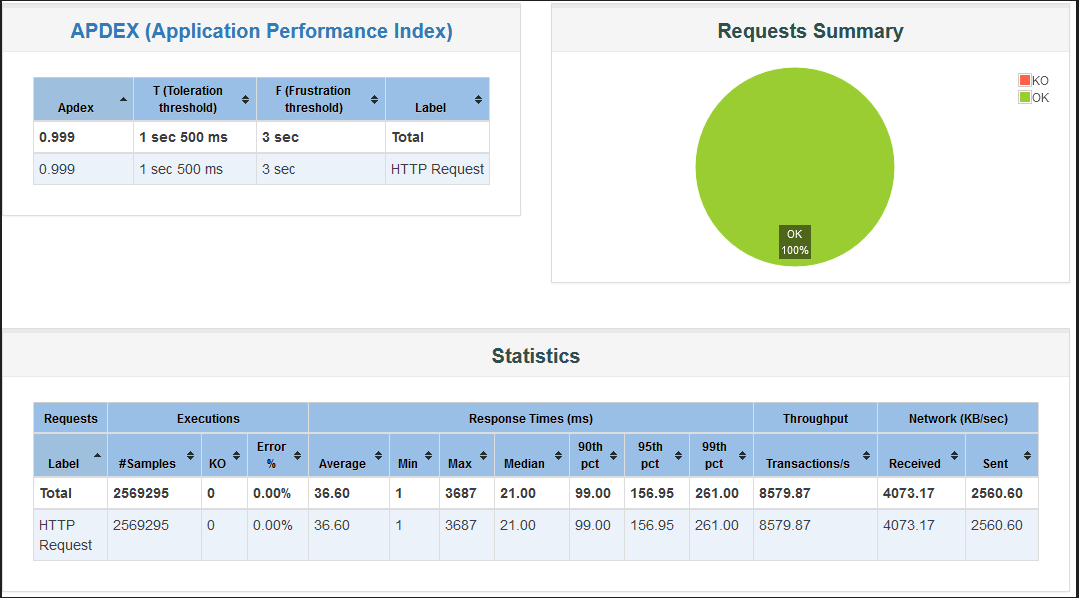
In Jmeter we can save the summary of test report in csv file. Throughput is calculated as no.of request processed per sec. The summary report also gives us Error%. i.e. percentage of failed requests as shown below.

API: IL to IL & English to IL [<https://localhost:44345/transliteration.aspx>?]

No. of Users & Time: 500 users & 15 minutes.



To generate the result graph we have to run the test in Non-GUI Mode. i.e. using command prompt.



**References:**

<https://www.merixstudio.com/blog/best-practices-rest-api-development/>

<https://www.partech.nl/nl/publicaties/2020/07/9-trending-best-practices-for-rest-api-development>

ebook-UndisturbedREST\_v1.pdf

<https://www.postman.com/product/tools/>

<https://jmeter.apache.org/index.html>

<https://testsigma.com/blog/api-functional-testing/>

https://tyk.io/blog/api-management-101-rate-limiting/#:~:text=Rate%20limit%20refers%20to%20the,of%2010%20times%20per%20minute.