July API Batch 2024 Notes

API Terminologies:

The document "API Terminologies" includes definitions and explanations for a variety of terms essential in understanding and working with APIs. These terms include:

1. **API Request**: A message sent from an API client to a server to request data or perform an action.
2. **API Response**: A message sent by the API server to the client in response to a request.
3. **API Endpoint**: A URL representing a specific resource or action in an API.
4. **Methods**: HTTP methods indicating desired actions on a resource.
5. **Resource**: A specific entity or object within the API.
6. **Parameters**: Inputs provided to the API to process requests.
7. **Payload**: Data transmitted as part of a request or response.
8. **API Gateway**: A server acting as an entry point for clients to access multiple services.
9. **API Key**: A unique identifier used for authenticating and authorizing API access.
10. **cURL**: A command-line tool for making HTTP requests.
11. **CRUD**: Operations (Create, Read, Update, Delete) performed on data.
12. **Cache**: Temporary storage of API data or responses.
13. **Client**: A software application sending API requests and receiving responses.
14. **JSON**: A data interchange format used in APIs.
15. **XML**: A markup language for structuring data.
16. **REST**: A set of architectural principles for creating APIs.
17. **SOAP**: A protocol for API communication.
18. **Authentication**: Verifying client identity before allowing API access.
19. **API Documentation**: Instructions describing the functionality and usage of an API.
20. **API Security**: Protecting an API from unauthorized access and attacks.
21. **Environment**: The combination of hardware, software, and network configurations for API deployment.
22. **CI/CD**: Practices for automating software development and release processes.
23. **Webhook**: Notifications or updates sent to clients when specific events occur.
24. **Mock Server**: A simulated server used for testing and development.

What is REST and best practices:

Topic	Summary	Comments

What is REST API?

REST (Representational State Transfer) is an architectural style for designing networked applications.
RESTful APIs (Application Programming Interfaces) adhere to the principles and constraints defined by the REST architectural style.
While REST itself does not dictate specific standards, there are several commonly accepted practices and guidelines that developers follow when designing RESTful APIs.

Here are some REST API standards and best practices:

1. Use HTTP Verbs: RESTful APIs leverage HTTP methods (GET, POST, PUT, DELETE, etc.) to perform operations on resources. Use the appropriate HTTP verb to reflect the intended action on the resource.

2. Resource Naming: Use descriptive, plural nouns to represent resources. For example, "/users" for a collection of users or "/users/{id}" for a specific user identified by an ID.

3. Use Proper HTTP Status Codes: Return appropriate HTTP status codes to indicate the success or failure of an API request. For example, 200 OK for a successful request, 404 Not Found for a non-existent resource, or 400 Bad Request for invalid input.

4. Versioning: If you need to make backward-incompatible changes to your API, consider versioning it. You can include the version in the URL (e.g., "/v1/users") or use request headers to specify the version.

5. Use Proper Error Handling: When an API request fails, return meaningful error messages in a consistent format, along with the appropriate HTTP status code. Include error codes, error descriptions, and any additional relevant information.

6. Pagination: For resource collections that may return a large number of results, provide pagination options to limit the number of items returned per page and offer navigation links (e.g., "next," "previous") for traversing the result set.

7. Query Parameters: Use query parameters to filter, sort, or search resources. For example, "/users?role=admin" or "/users?sort=name".

8. Content Negotiation: Use HTTP headers like "Accept" and "Content-Type" to allow clients to specify the desired response format (JSON, XML, etc.) and to indicate the format of the data being sent in the request body.

9. Authentication and Authorization: Implement proper authentication and authorization mechanisms to secure your API. Use standard protocols like OAuth or JWT (JSON Web Tokens) to handle authentication and authorization.

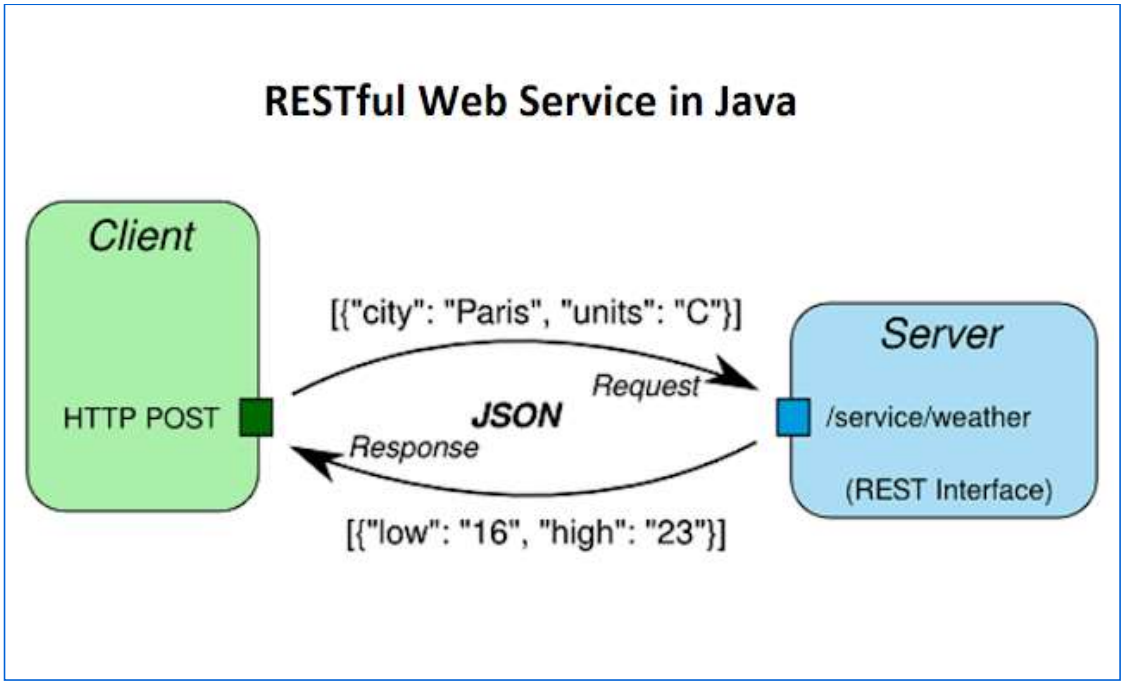
10. HATEOAS: Consider adopting HATEOAS (Hypermedia as the Engine of Application State) to make your API self-descriptive. Include links in the API responses that allow clients to navigate and discover related resources.

These are some of the commonly followed REST API standards and best practices. Adhering to these practices helps create consistent, predictable, and easy-to-use APIs that promote interoperability and scalability.

Simple example of REST WebService in Java:

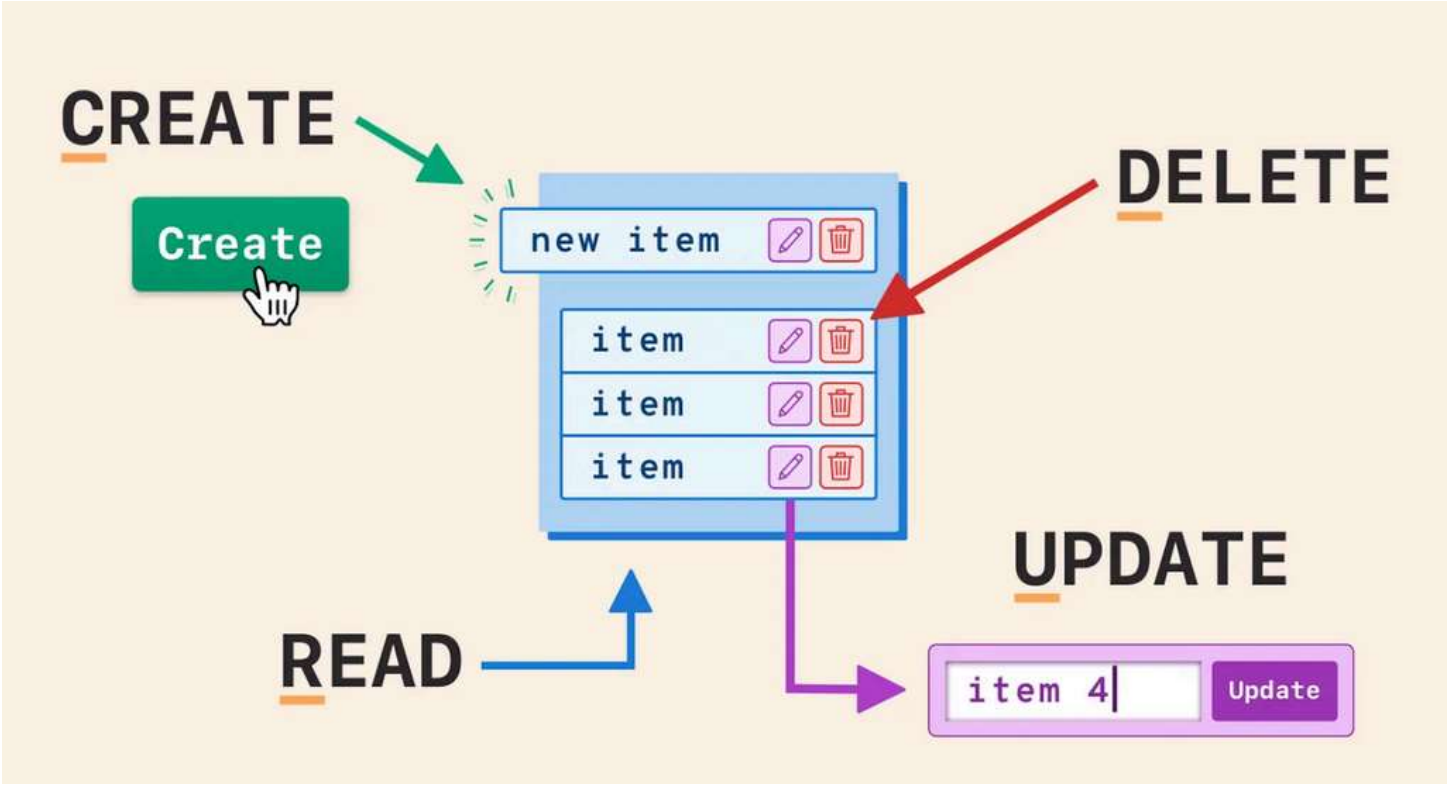
IETF - RFC 7231 Doc: https://www.rfc-editor.org/rfc/rfc7231

RFC - Request For Comments
IETF: Internet Engineering Task Force

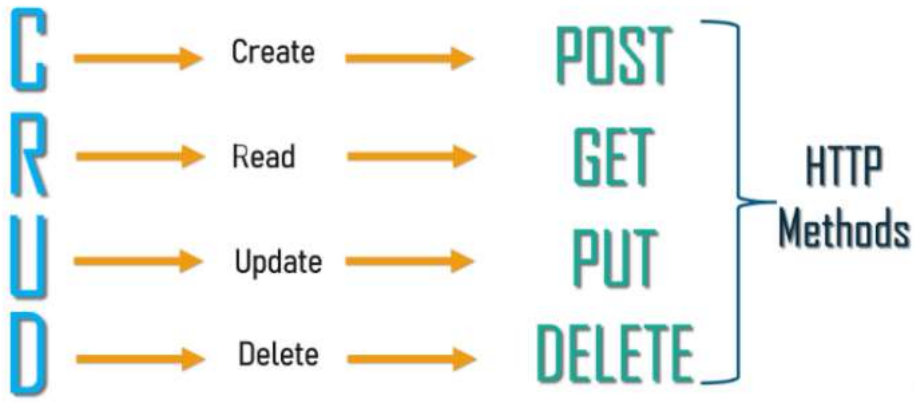


HTTP Methods:	**HTTP Method:** • GET • POST • PUT • PATCH • DELETE • HEAD • OPTIONS	**To perform CRUD operation:** **C- Create** **R- Read/Retrieve** **U - Update** **D - Delete**

CRUD:



CRUD against HTTP METHODS:



Safety, cacheability, and idempotency of common HTTP methods:

HTTP Method	Safe	Cachable	Idempotent
GET	Yes	Yes	Yes
POST	No	Yes*	No
PUT	No	No	Yes
DELETE	No	No	Yes
PATCH	No	No	No

OPTIONS	Yes	Yes	Yes
HEAD	Yes	Yes	Yes

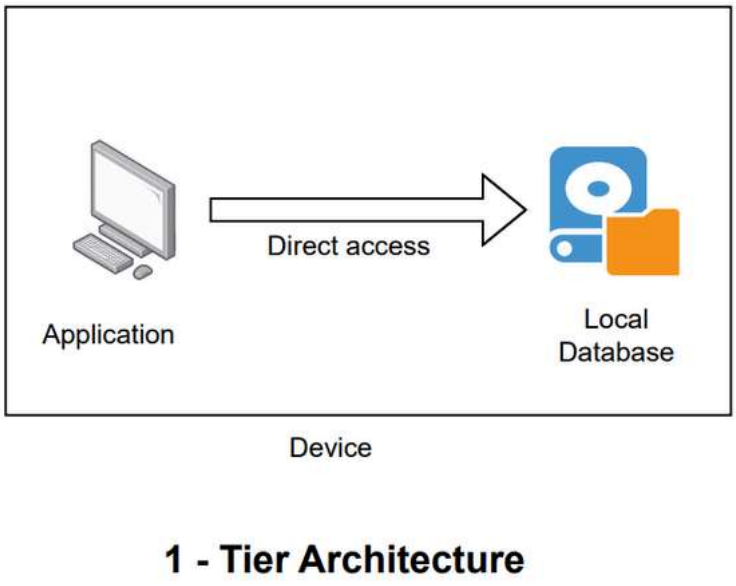
* Note: POST requests can be cached if they meet specific criteria defined by the server, though this is not common practice.

Definitions:

- **Safe**: An HTTP method is considered safe if it does not modify any resources on the server. Safe methods are intended only for retrieving data.
- **Cachable**: An HTTP method is considered cachable if the responses to requests using that method can be stored and reused.
- **Idempotent**: An HTTP method is idempotent if multiple identical requests have the same effect as a single request.

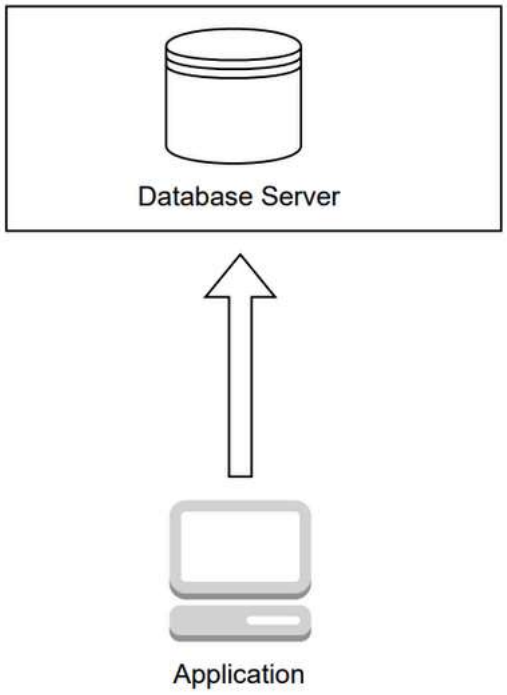
System Architecture:

1. **Single Tier Architecture**: In this basic structure, the client, server, and database are all on the same machine. This architecture puts the user directly in contact with the database itself, so the user can create, modify, or delete data within the database. The user sits directly on the database, without any intermediary layer.



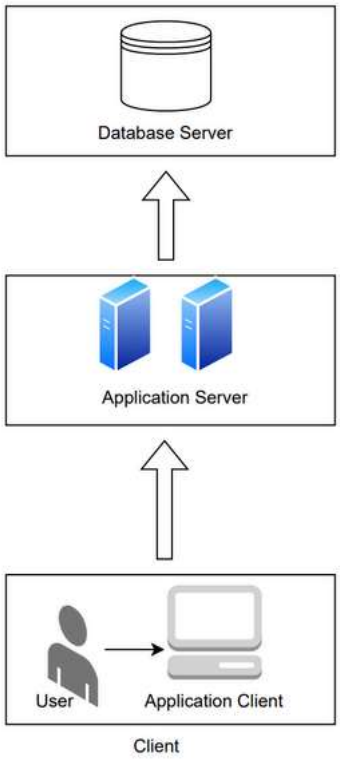
1 - Tier Architecture

2. **Two Tier Architecture**: This client-server model involves user interfaces and application programs on the client side, interacting with the database on the server side. It's efficient for handling multiple users simultaneously. Two tier architecture provides added security to the DBMS as it is not exposed to the end-user directly. It also provides direct and faster communication.



2 - Tier Architecture

3. **Three Tier Architecture**: In this widely used model, an application layer or API is added between the client and db server layers. This prevents exposing the direct data access to public. The APIs does some authentications, then provide access. It enhances security, data integrity, and scalability by abstracting interactions.



3 - Tier Architecture

Each architecture has its pros and cons. Single tier is simple but not scalable; two-tier is efficient but has security risks; three-tier offers enhanced security and scalability but introduces complexity.

Topics	Details	Comments

API vs WebServices

API (Application Programming Interface):

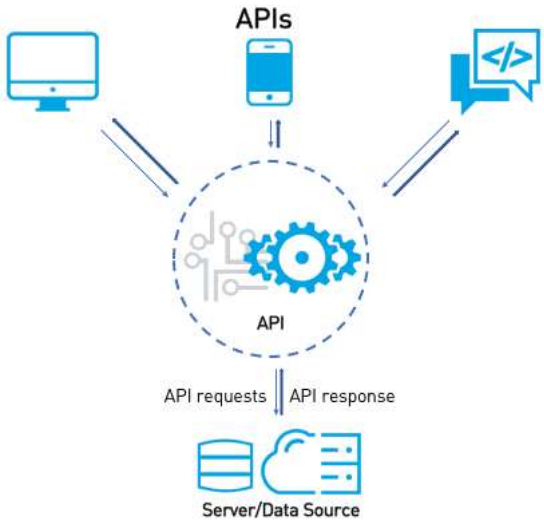
- **An API is a set of rules and protocols that allows different software applications to communicate and interact with each other.**
- **It defines how requests and responses should be structured, what data formats to use, and what functionalities can be accessed.**
- **APIs enable developers to access and use the functionalities of an application, service, or platform to build new applications or integrate existing systems.**

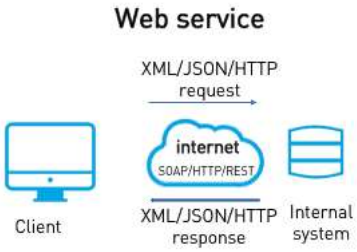
Examples:

- APACHE POI API
- Log4j API
- Selenium API
- Java Collections API

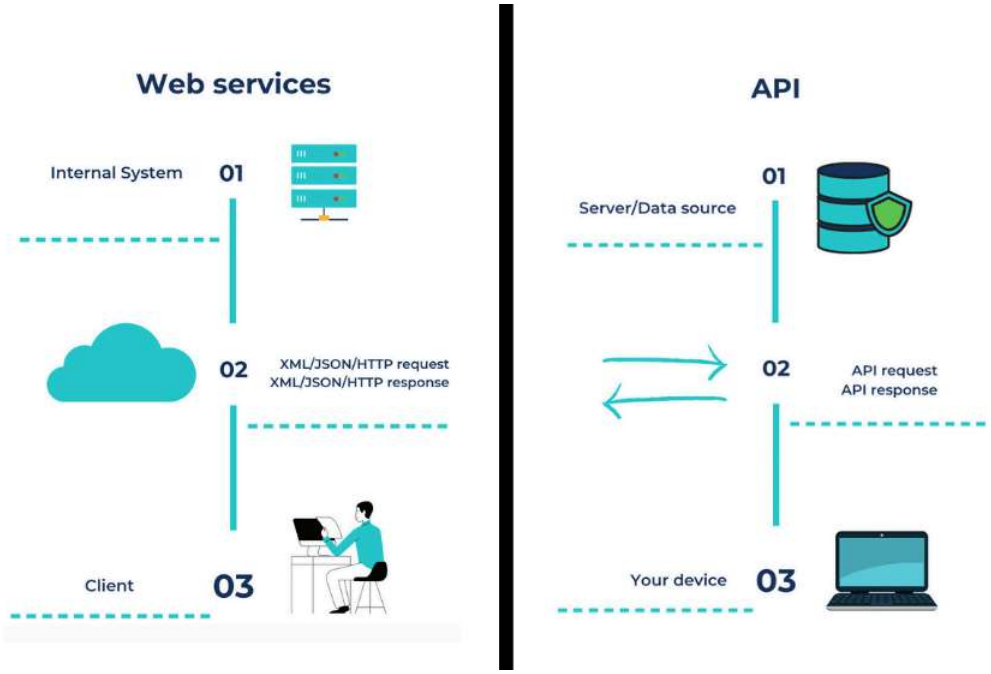
Web Service:

- **A web service is a technology method of communication between different software applications over a network/internet, typically using standard web protocols such as HTTP.**
- **It allows systems built on different platforms and programming languages to interact with each other by exchanging data in a structured format, often using XML or JSON. Web services are typically based on a client-server architecture and can be accessed remotely over the internet.**





APIs define the rules and protocols for communication between applications, while web services are a specific implementation of APIs that use web technologies to enable interoperability between different systems. APIs can be implemented using various technologies, not just limited to web services, but web services are a common and popular form of API implementation.



Examples:

1. OpenWeatherMap web service provides weather data by exposing APIs that allow developers to retrieve weather forecasts and current conditions for specific locations. Developers can make HTTP requests to the OpenWeatherMap API endpoint, receive weather data in a structured format like JSON, and integrate it into their applications.

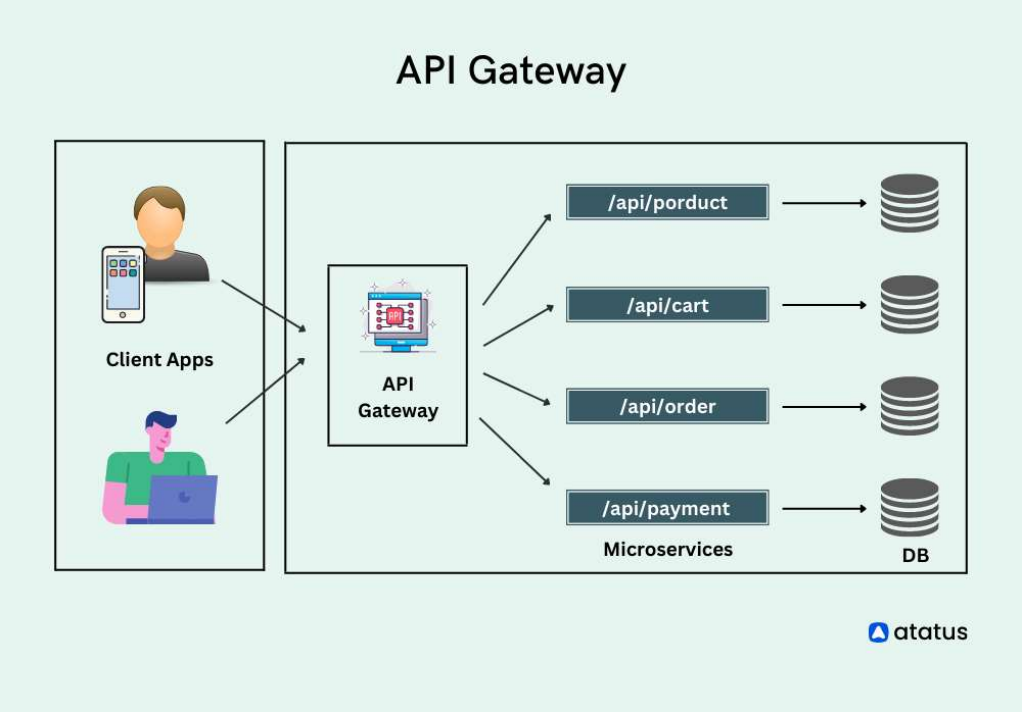
2. SOAP (Simple Object Access Protocol): SOAP is a protocol for implementing web services. It uses XML to structure requests and responses and typically operates over HTTP or other application layer protocols. SOAP web services are widely used in enterprise applications and can be accessed using a WSDL (Web Services Description Language) file.

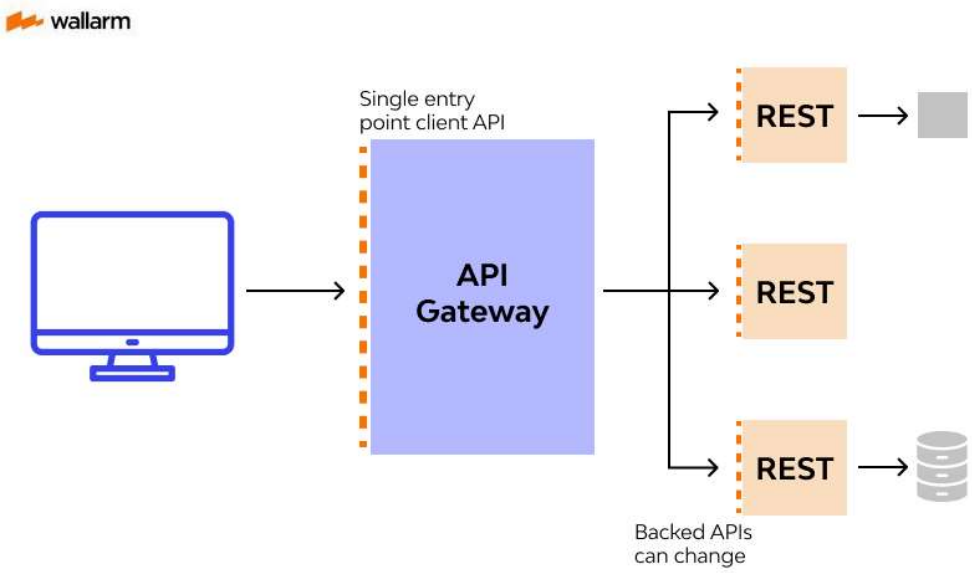
3. RESTful API (Representational State Transfer): REST is an architectural style for designing networked applications. RESTful APIs use standard HTTP methods like GET, POST, PUT, and DELETE to perform operations on resources. They often utilize JSON or XML for data exchange and are widely used in web and mobile applications.

4. Amazon Web Services (AWS) API: AWS provides a vast array of web services covering cloud computing, storage, databases, machine learning, and more. AWS offers APIs for each service, allowing developers to manage resources, provision infrastructure, and access various cloud-based functionalities programmatically.

What is API Gateway?

An API Gateway is a centralized entry point for managing, securing, and monitoring APIs. It acts as an intermediary entity between clients (such as web or mobile applications) and a collection of backend services or APIs.





The API Gateway provides several key features and benefits:

1. API Management: API Gateways offer a comprehensive set of management capabilities for APIs. This includes defining and managing API endpoints, handling versioning, enforcing access controls, rate limiting, and throttling to ensure the stability and security of the APIs.
2. Security: API Gateways provide security features such as authentication and authorization mechanisms, allowing you to control who can access the APIs and what actions they can perform. It can handle authentication protocols like OAuth, JWT, or API keys, and enforce security policies across all the APIs.
3. Request Routing: The API Gateway can route requests to the appropriate backend services based on defined rules and configurations. It acts as a reverse proxy, directing traffic to the correct service or backend system. It can also aggregate data from multiple services and compose responses to fulfill client requests in the form of JSON/XML.

An API Gateway often incorporates load balancing capabilities, which help distribute incoming requests across multiple backend servers or services.

Load Balancing: An API Gateway can act as a load balancer by distributing incoming API requests across multiple backend servers or services.

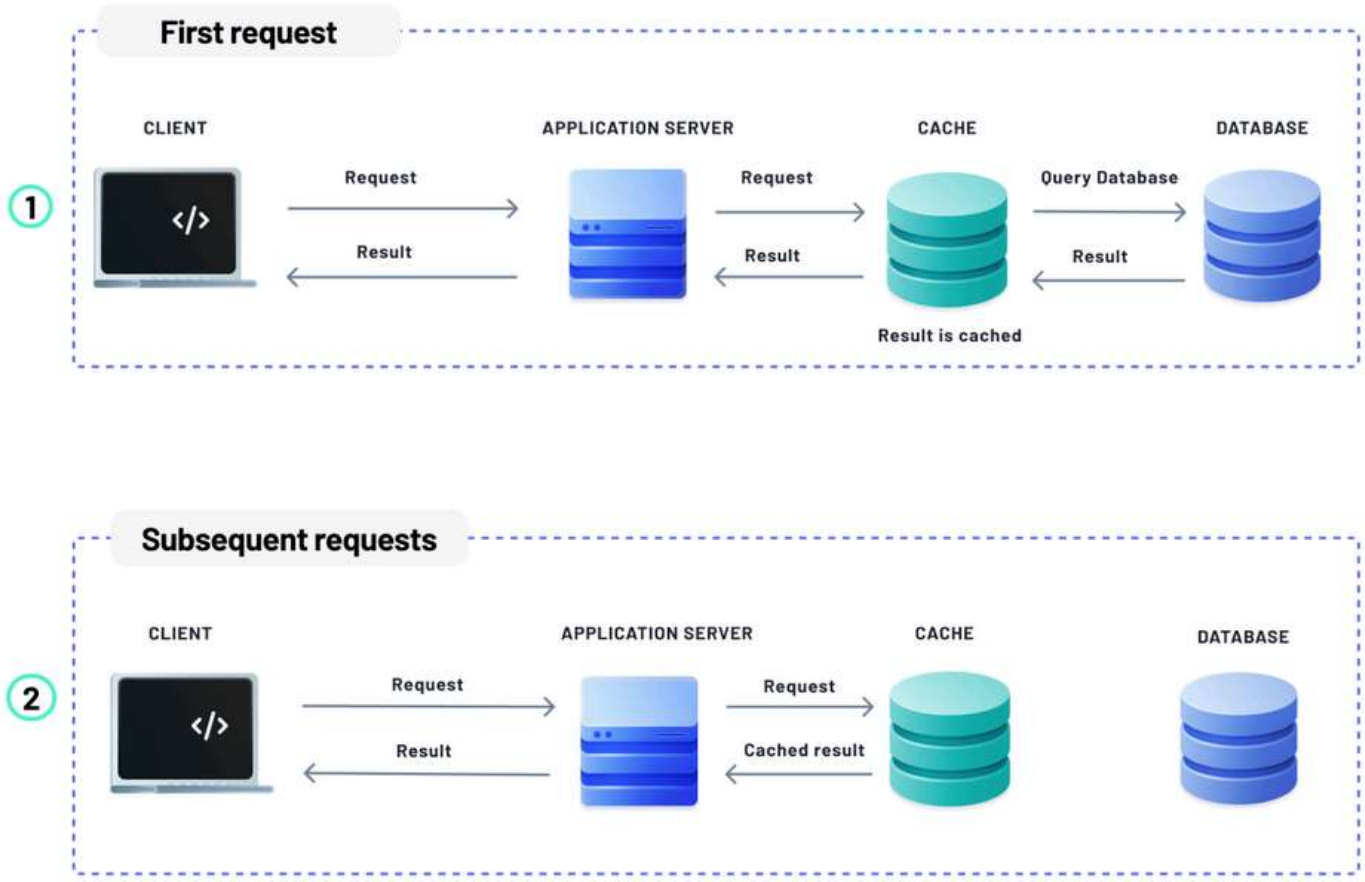
This helps achieve high availability, scalability, and efficient resource utilization.

4. Transformation and Adaptation: API Gateways can transform the structure or format of requests and responses to match the needs of the client or backend services. This allows for protocol translation, data mapping, payload manipulation, or response formatting to provide a seamless integration between different systems.
5. Monitoring and Analytics: API Gateways collect and provide valuable insights into API usage, performance, and health. They offer logging, monitoring, and analytics capabilities, enabling you to track API usage, identify bottlenecks, and diagnose issues in real-time.
6. Caching and Performance Optimization: API Gateways can implement caching mechanisms to store and serve frequently requested data or responses. This reduces the load on backend systems, improves response times, and enhances overall performance.
7. Scalability and Load Balancing: API Gateways can distribute incoming requests across multiple instances or backend services, ensuring high availability and scalability. They can perform load balancing to evenly distribute traffic and manage backend service instances dynamically.
8. Developer Portal: API Gateways often include a developer portal or documentation platform, providing developers with information about available APIs, documentation, usage guidelines, and code samples. This helps streamline API discovery, onboarding, and developer collaboration.

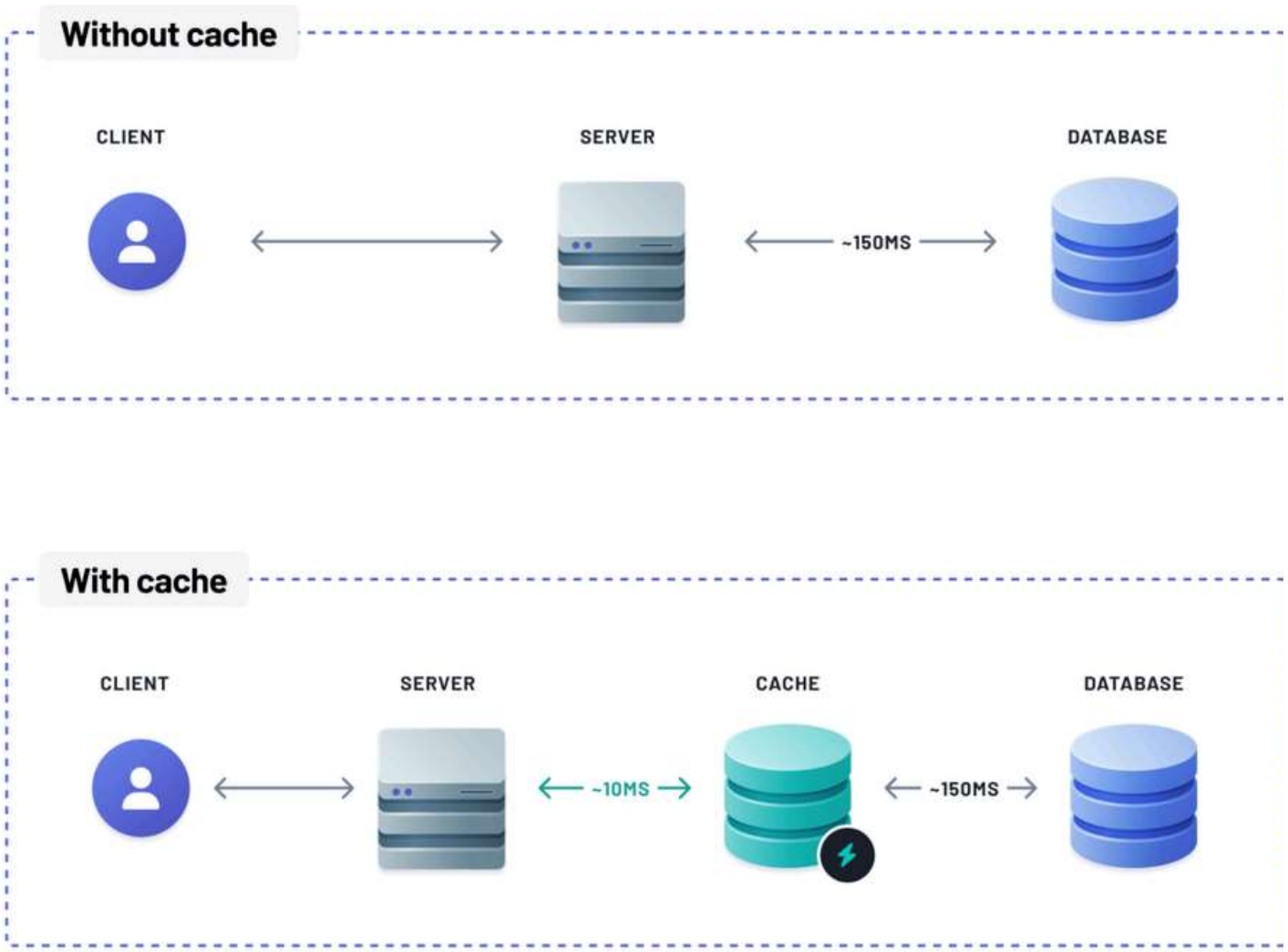
What is Caching?

Caching significantly improves performance

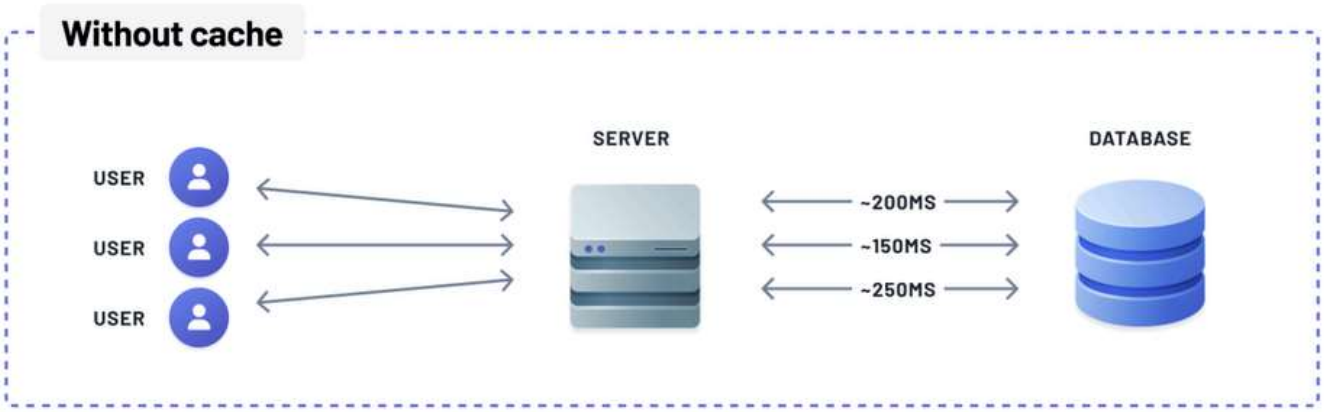
Using a cache to store database query results can significantly boost the performance of your application.
A cache is much faster and usually hosted closer to the application server, which reduces the load on the main database, accelerates data retrieval, and minimizes network and query latency.

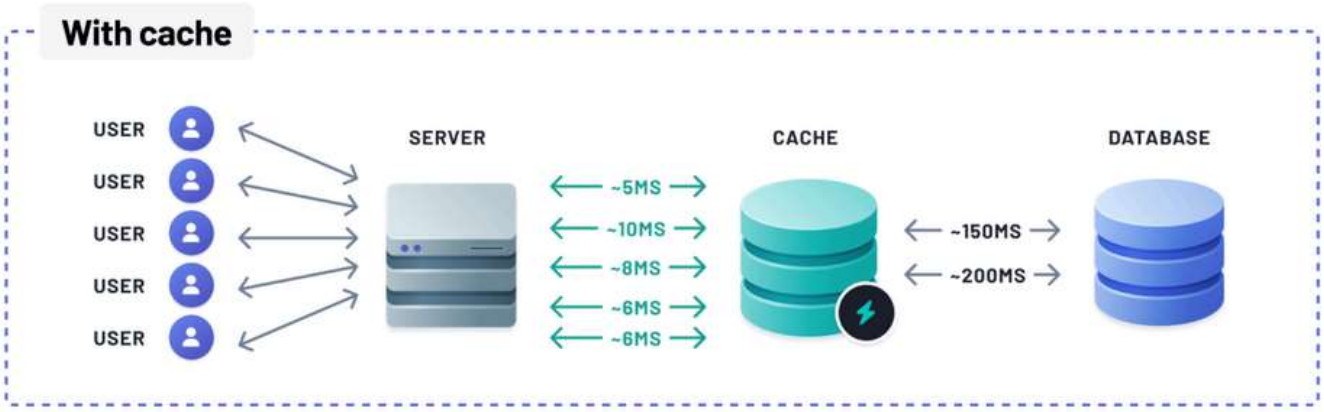


Caching reduces CPU usage, disk access, and network utilization by quickly serving frequently accessed data to the application server, bypassing the need for a round trip to the database.



Caching also plays a crucial role in improving the scalability of your application, allowing it to handle increased loads and accommodate higher user concurrency and more extensive data volumes.

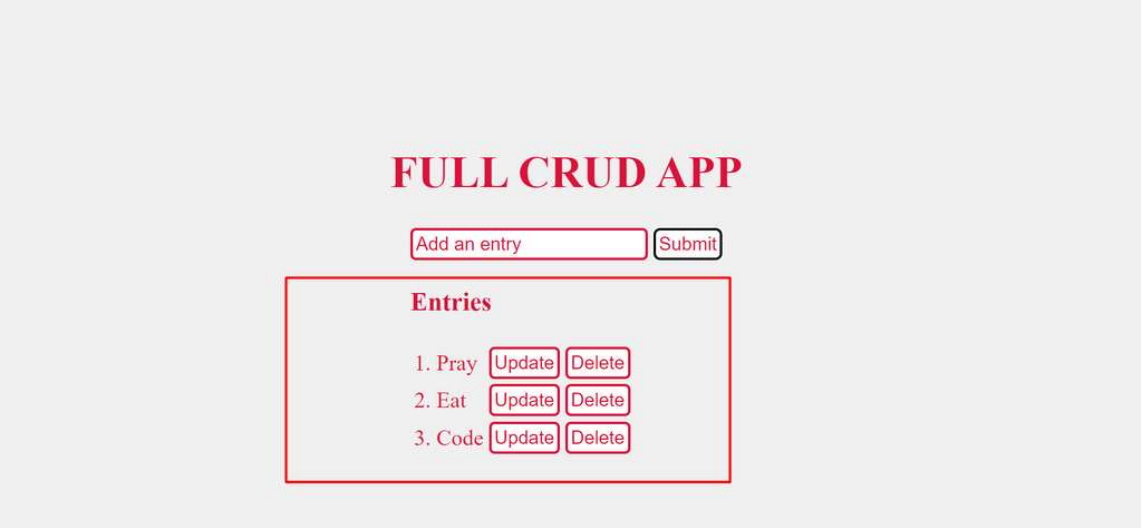




CREATE CALL:

Image could not be loaded due to removal or insufficient permissions.

GET CALL:



UPDATE CALL:

Image could not be loaded due to removal or insufficient permissions.

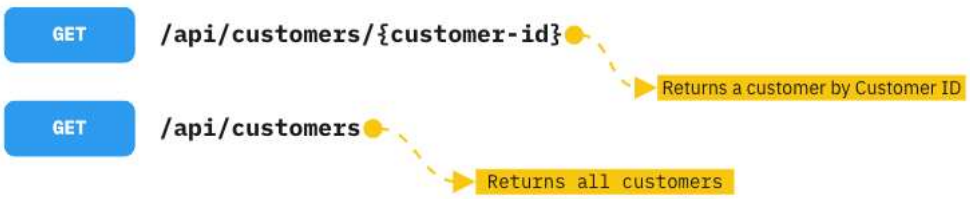
DELETE Call:

Image could not be loaded due to removal or insufficient permissions.

Explanation of various HTTP Methods/Verbs:

GET Method

If we want to retrieve data from a resource like websites, servers or APIs, we send them a GET Request. For example, we send a GET request to the server if we want a list of our customers or a specific customer.



Since the GET method should never change the data on the resources and just read them(read-only), it is considered a Safe Method. Additionally, the Get method is idempotent.

How to test an API with a GET method?

When we want to test an API, the most popular method that we would use is the GET method. Therefore, We expect the following to happen.

- If the resource is accessible, the API returns the 200 Status Code, which means OK.
- Along with the 200 Status Code, the server usually returns a response body in XML or JSON format. So, for example, we expect the [GET] /members endpoint to return a list of members in XML or JSON.
- If the server does not support the endpoint, the server returns the 404 Status Code, which means Not Found.
- If we send the request in the wrong syntax, the server returns the 400 Status Code, which means Bad Request.

POST Method

The POST method creates a new resource on the backend (server). The request body carries the data we want to the server. It is neither a safe nor idempotent method. We don't expect to get the same result every time we send a POST request. For example, two identical POST requests will create two new equivalent resources with the same data and different resource ids.
When sending a POST request to a server, we expect the following to happen:

- Ideally, if the POST request has created a new resource on the other side, the response should come with 201 Status Code which means Created.
- Sometimes, performing a POST request doesn't return a resource at the given URL; in this case, the method will return 204 status code which means No content.

How to test a POST endpoint

Since the POST method creates data, we must be cautious about changing data; testing all the POST methods in APIs is highly recommended. Moreover, make sure to delete the created resource once your testing is finished.
Here are some suggestions that we can do for testing APIs with POST methods:

- Create a resource with the POST method, and it should return the 201 Status Code.
- Perform the GET method to check if it created the resource was successfully created. You should get the 200 status code, and the response should contain the created resource.
- Perform the POST method with incorrect or wrong formatted data to check if the operation fails.

PUT Method

With the PUT request method, we can update an existing resource by sending the updated data as the content of the request body to the server. The PUT method updates a resource by replacing its entire content completely. If it applies to a collection of resources, it replaces the whole collection, so be careful using it. The server will return the 200 or 204 status codes after the existing resource is updated successfully.

How to test an API with a PUT method?

The PUT method is idempotent, and it modifies the entire resources, so to test that behavior, we make sure to do the following operations:

- Send a PUT request to the server many times, and it should always return the same result.
- When the server completes the PUT request and updates the resource, the response should come with 200 or 204 status codes.
- After the server completes the PUT request, make a GET request to check if the data is updated correctly on the resource.
- If the input is invalid or has the wrong format, the resource must not be updated.

PATCH Method

PATCH is another HTTP method that is not commonly used. Similar to PUT, PATCH updates a resource, but it updates data partially and not entirely. For example, to make it more precise, the request [PUT] customers/{customerid} would update the fields in the Customers entity on the resource entirely. However, the PATCH method does update the provided fields of the customer entity. In general, this modification should be in a standard format like JSON or XML.

How to test an API with a PATCH method?

To test an API with the PATCH method, follow the steps discussed in this article for the testing API with the PUT and the POST methods. Consider the following results:

- Send a PATCH request to the server; the server will return the 2xx HTTP status code, which means: the request is successfully received, understood, and accepted.
- Perform the GET request and verify that the content is updated correctly.

- If the request payload is incorrect or ill-formatted, the operation must fail.

DELETE Method

As the name suggests, the DELETE method deletes a resource. The DELETE method is idempotent; regardless of the number of calls, it returns the same result. Most APIs always return the 200 status code even if we try to delete a deleted resource but in some APIs, If the target data no longer exists, the method call would return a 404 status code.

How to test a DELETE endpoint?

When it comes to deleting something on the server, we should be cautious. We are deleting data, and it is critical. First, make sure that deleting data is acceptable, then perform the following actions.

- Call the POST method to create a new resource. Never test DELETE with actual Data. For example, first, create a new customer and then try to delete the customer you just created.
- Make the DELETE request for a specific resource. For example, the request [DELETE] /customers/ {customer-id} deletes a customer with thee specified customer Id.
- Call the GET method for the deleted customer, which should return 404, as the resource no longer exists.

Testfully's Multi-step tests allow you to create resources on the fly and use them for testing DELETE endpoints.

HEAD Method

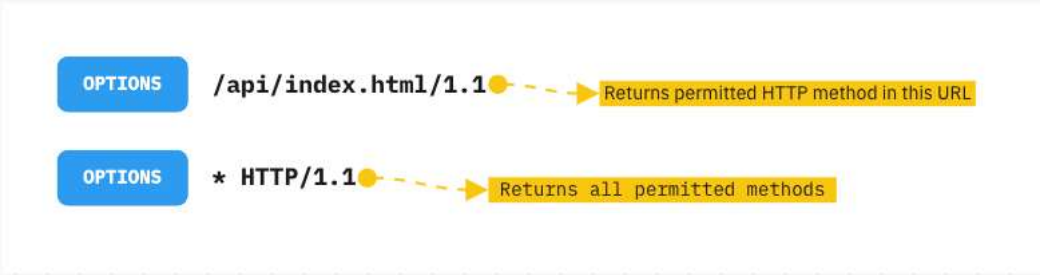
The HEAD method is similar to the GET method. But it doesn't have any response body, so if it mistakenly returns the response body, it must be ignored. For example, the [GET] /customers endpoint returns a list of customers in its response body. Also, the [HEAD] /customers do the same, but it doesn't return a list of customers. Before requesting the GET endpoint, we can make a HEAD request to determine the size (Content-length) of the file or data that we are downloading. Therefore, the HEAD method is safe and idempotent.

How to test a HEAD endpoint

One of the advantages of the HEAD method is that we can test the server if it is available and accessible as long as the API supports it, and it is much faster than the GET method because it has no response body. The status code we expect to get from the API is 200. Before every other HTTP method, we can first test API with the HEAD method.

OPTIONS Method

We use This method to get information about the possible communication options (Permitted HTTP methods) for the given URL in the server or an asterisk to refer to the entire server. This method is safe and idempotent.
Various browsers widely use the OPTIONS method to check whether the CORS (Cross-Origin resource sharing) operation is restricted on the targeted API or not.



How to test an OPTIONS endpoint

Depending on whether the server supports the OPTIONS method, we can test the server for the times of FATAL failure with the OPTIONS method. To try it, consider the following.

- **Make an OPTIONS request and check the header and the status code that returns.**
- **Test the case of failure with a resource that doesn't support the OPTIONS method.**

==========

POSTMAN:

Download link : https://www.postman.com/downloads/

POSTMAN

Postman is a versatile tool that goes beyond these features, and its capabilities continue to expand with new updates and integrations.
It serves as a comprehensive API development, testing, and collaboration platform, helping streamline the API workflow and improve productivity for developers and teams.

==========

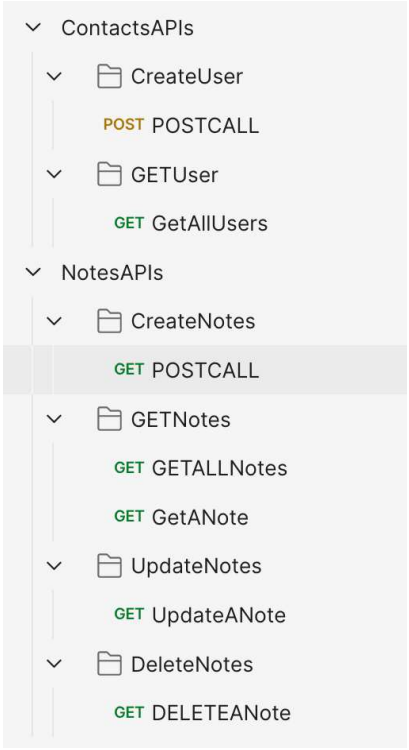
Postman Feature	Explanation
API Testing	Postman is primarily known as an API testing tool. It allows you to send HTTP requests and verify the responses, making it easy to test API endpoints and functionalities.
Request Builder	Postman provides a user-friendly interface for building HTTP requests. You can specify the request method (GET, POST, etc.), URL, headers, parameters, and request body. It simplifies the process of constructing requests with the desired configurations.
Request History	Postman keeps a history of all the requests you have made, making it convenient to revisit and reuse previous requests. You can access the request history, view the details of past requests, and quickly resend them without having to recreate them from scratch.
Collections	Postman allows you to organize related requests into collections. Collections provide a way to group requests, making it easier to manage and execute them together. You can create folders, add requests to collections, and share collections with team members.
Environment Variables	Postman supports the use of environment variables, which enable you to store and reuse dynamic values across requests. You can define variables for different environments (e.g., development, staging, production), making it simple to switch between environments and maintain consistency in your API testing.

Testing and Automation	Postman provides a powerful testing framework that allows you to write tests in JavaScript. You can write test scripts to verify the expected behavior of the API responses, perform assertions, and automate testing workflows. Additionally, you can create test suites, run tests in bulk, and generate reports to track test results.
Mock Servers	With Postman, you can create mock servers that simulate API endpoints. This feature allows you to define custom responses for different requests, making it possible to test and develop against an API even before the actual backend implementation is ready.
Documentation	Postman offers a documentation feature that automatically generates interactive API documentation based on your requests and their descriptions. This documentation includes details about the API endpoints, request examples, response formats, and can be shared with others to provide clear and accessible API documentation.
Collaboration	Postman supports collaboration features, enabling team members to work together on API-related tasks. You can share collections, environments, and documentation with teammates, making it easier to collaborate, provide feedback, and ensure consistency in API development and testing.

Step-by-step guide on creating a workspace, organizing collections into folders, and adding HTTP requests to those folders in Postman:

1. Creating a Workspace:
   - Open Postman and click on the "Workspace" dropdown at the top-left corner.
   - Click on "Create a Workspace" and give it a name.
   - Choose the appropriate visibility (private or public) and click "Create."
2. Creating Folders and Collections:
   - Inside the newly created workspace, click on "Create a collection" to create a collection.
   - Give the collection a name (e.g., "API Requests").
   - Click on the "..." next to the collection name and select "Add Folder."
   - Create four folders: "GET," "POST," "PUT," and "DELETE" within the collection.
3. Adding Requests to Folders:
   - Click on the desired folder (e.g., "GET") to select it.
   - Inside the folder, click on "Create a request" and give it a name.
   - Select the HTTP method as GET and enter the URL for the GET request.
   - Repeat the above steps for the other HTTP methods (POST, PUT, DELETE) in their respective folders.

Actual Structure:



Example Structure:

markdown Copy code

```
- Workspace
  - API Requests (Collection)
    - GET (Folder)
      - GET Request 1
      - GET Request 2
    - POST (Folder)
      - POST Request 1
      - POST Request 2
    - PUT (Folder)
      - PUT Request 1
      - PUT Request 2
    - DELETE (Folder)
      - DELETE Request 1
      - DELETE Request 2
```

API End Points for the practice with DataBase (My SQL Server):

The app defines following CRUD APIs.

- GET /api/notes
- POST /api/notes
- GET /api/notes/{noteId}
- PUT /api/notes/{noteId}
- DELETE /api/notes/{noteId}

API/ End point URL	JSON Payload	Response
• Base URL : http://3.110.31.34:8080 • Data Base URL: https://auth-db351.hstgr.io/index.php?route=/sql&server=1&db=u811712038_notes_app • Table Name: **Notes** • DB: MY SQL Server	• **UserName: u811712038_naveenautomate** • **Password: Spring@1234554321**	

1. **POST: /api/notes**
Create a new note
	{
	"title" : "my selenium title",
	"content" : "this is my selenium content"
	}
		{
		"id": 41,
		"title": "my selenium title",
		"content": "this is my selenium content",
		"createdAt": "2023-06-09T06:09:38.877+00:00",
		"updatedAt": "2023-06-09T06:09:38.877+00:00"
		}

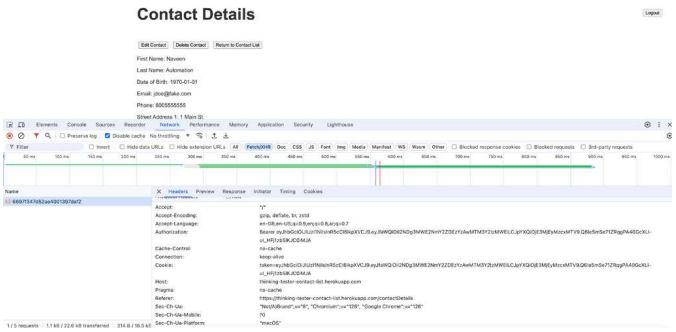
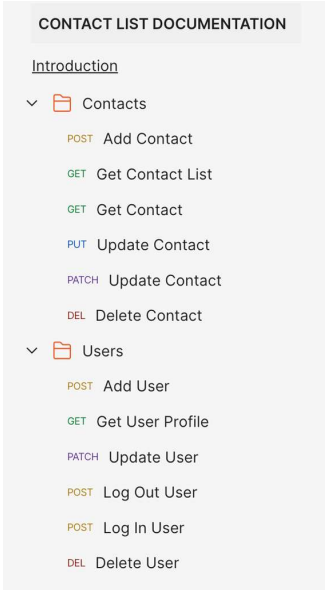
2. **GET: /api/notes**
Get all the notes
	NA	[
		{
		"id": 3,
		"title": "article",
		"content": "this is my first article",
		"createdAt": "2023-06-06T09:04:38.000+00:00",
		"updatedAt": "2023-06-06T09:04:38.000+00:00"
		},
		{
		"id": 4,
		"title": "code",
		"content": "this is my first code",
		"createdAt": "2023-06-06T09:04:44.000+00:00",
		"updatedAt": "2023-06-06T09:04:44.000+00:00"
		},
		{
		"id": 5,
		"title": "codes",
		"content": "this is my second code",
		"createdAt": "2023-06-06T10:11:39.000+00:00",
		"updatedAt": "2023-06-08T18:37:40.000+00:00"
		}
		]

3. GET: /api/notes/5
Get the specific note
	NA	{
		"id": 5,
		"title": "codes",
		"content": "this is my second code",
		"createdAt": "2023-06-06T10:11:39.000+00:00",
		"updatedAt": "2023-06-08T18:37:40.000+00:00"
		}

3. PUT: /api/notes/3
Update existing note
	{
	"title": "my api title course",
	"content": "this is my api content"
	}
		{
		"id": 3,
		"title": "my api title course",
		"content": "this is my api content",
		"createdAt": "2023-06-06T09:04:38.000+00:00",
		"updatedAt": "2023-06-09T06:14:22.107+00:00"
		}

4. DELETE: /api/notes/4
Delete a note
• Perform GET call for the notes id=4 using GET URL after delete:
• URL: http://3.110.31.34:8080/api/notes/4
	NA	NA
		
		Response after GET call:
		{
		"timestamp": "2023-06-09T06:16:03.024+00:00",
		"status": 404,
		"error": "Not Found",
		"path": "/api/notes/4"
		}

Contact List API with Browser Network Calls:

Documentation URL : https://documenter.getpostman.com/view/4012288/TzK2bEa8 --Please check the API docs here. Web Application URL: https://thinking-tester-contact-list.herokuapp.com/ --Use this web url to get the apis from Browser Network tab. --Please get the Bearer token from the network tab: 		

What is Bearer Token?	## Bearer Authentication **Bearer authentication** (also called **token authentication**) is an HTTP authentication scheme that involves security tokens called bearer tokens. The name “Bearer authentication” can be understood as “give access to the bearer of this token.” The bearer token is a cryptic string, usually generated by the server in response to a login request. The client must send this token in the Authorization header when making requests to protected resources:	**Authorization: Bearer <token>**

Test Cases for Contacts APIs:

Endpoint	Test Case	Description	Response Body
GET /contacts	Status Code	Verify that the status code is 200.	
	Response Time	Verify that the response time is less than 2 seconds.	
	Response Format	Verify that the response is in JSON format.	

	Response Structure	Verify that the response body contains an array of contact objects with keys id, firstName, lastName, email, phone, postalCode etc..	[{ "_id": "6697f347d52ae4001397daf2", "firstName": "Naveen", "lastName": "Automation", "birthdate": "1970-01-01", "email": "jdoe@fake.com", "phone": "8005555555", "street1": "1 Main St.", "street2": "Apartment A", "city": "Anytown", "stateProvince": "KS", "postalCode": "12345", "country": "USA", "owner": "64871a66f6d13c00137cb31a", "__v": 0 }, { "_id": "6697f393d52ae4001397daf4", "firstName": "Naveen", "lastName": "Automation", "birthdate": "1970-01-01", "email": "jdoe@fake.com", "phone": "8005555555", "street1": "1 Main St.", "street2": "Apartment A", "city": "Anytown", "stateProvince": "KS", "postalCode": "12345", "country": "USA", "owner": "64871a66f6d13c00137cb31a", "__v": 0 }, { "_id": "6697f7c23155840013ce96be", "firstName": "Naveen", "lastName": "Automation", "birthdate": "1970-01-01", "email": "haley_batz@gmail.com", "phone": "8005555555", "street1": "1 Main St.", "street2": "Apartment A", "city": "Anytown", "stateProvince": "KS", "postalCode": "12345", "country": "USA", "owner": "64871a66f6d13c00137cb31a", "__v": 0 }, { "_id": "6698d468e4d2cf00134c078a", "firstName": "Naveen", "lastName": "Automation", "birthdate": "1970-01-01", "email": "jdoe@fake.com", "phone": "8005555555", "street1": "1 Main St.",

			"street2": "Apartment A",
			"city": "Anytown",
			"stateProvince": "KS",
			"postalCode": "12345",
			"country": "USA",
			"owner": "64871a66f6d13c00137cb31a",
			"__v": 0
			},
			{
			"_id": "6697e8763155840013ce966c",
			"firstName": "tom",
			"lastName": "peter",
			"owner": "64871a66f6d13c00137cb31a",
			"__v": 0
			},
			{
			"_id": "6697e7133155840013ce9665",
			"firstName": "Anna",
			"lastName": "sharma",
			"birthdate": "1990-12-15",
			"email": "kavya@gmail.com",
			"phone": "9999999999",
			"owner": "64871a66f6d13c00137cb31a",
			"__v": 0,
			"city": null,
			"country": null,
			"stateProvince": null,
			"street1": null,
			"street2": null
			}
			]

GET /contacts/{id}	Valid ID	Verify that a valid id returns status code 200 and correct contact details.	
	Invalid ID	Verify that an invalid id returns status code 404.	
	Response Time	Verify that the response time is less than 2 seconds.	

	Response Structure	Verify that the response body contains the keys:	{
			"_id": "6698d468e4d2cf00134c078a",
			"firstName": "Naveen",
			"lastName": "Automation",
			"birthdate": "1970-01-01",
			"email": "jdoe@fake.com",
			"phone": "8005555555",
			"street1": "1 Main St.",
			"street2": "Apartment A",
			"city": "Anytown",
			"stateProvince": "KS",
			"postalCode": "12345",
			"country": "USA",
			"owner": "64871a66f6d13c00137cb31a",
			"__v": 0
			}

POST /contacts	Status Code	Verify that the status code is 201.	

	Response Time	Verify that the response time is less than 2 seconds.	

	Request Body	Verify that a valid request body:
		{
		"firstName": "Naveen",
		"lastName": "Automation",
		"birthdate": "1970-01-01",
		"email": "jdoe@fake.com",
		"phone": "8005555555",
		"street1": "1 Main St.",
		"street2": "Apartment A",
		"city": "Anytown",
		"stateProvince": "KS",
		"postalCode": "12345",
		"country": "USA"
		}	

	Response Structure	Verify that the response body contains the keys id, firstName, lastName, email, phone, createdAt.	
	Invalid Data	Verify that sending an invalid request body (e.g., missing firstName) returns status code 400.	
PUT /contacts/{id}	Status Code	Verify that the status code is 200.	
	Response Time	Verify that the response time is less than 2 seconds.	
	Valid ID and Data	Verify that a valid id and request body (e.g., {"firstName": "Jane", "lastName": "Doe", "email": "jane.doe@example.com", "phone": "0987654321"}) updates the contact.	
	Invalid ID	Verify that an invalid id returns status code 404.	
	Invalid Data	Verify that sending an invalid request body (e.g., missing firstName) returns status code 400.	
	Response Structure	Verify that the response body contains the updated id, firstName, lastName, email, phone, createdAt.	
DELETE /contacts/{id}	Status Code	Verify that the status code is 200.	
	Response Time	Verify that the response time is less than 2 seconds.	
	Valid ID	Verify that a valid id deletes the contact and subsequent GET request to the same id returns status code 404.	
	Invalid ID	Verify that an invalid id returns status code 404.	

Endpoint	Test Case	Description
All Endpoints	Authorization Header Present	Verify that the Authorization header is present in the request.

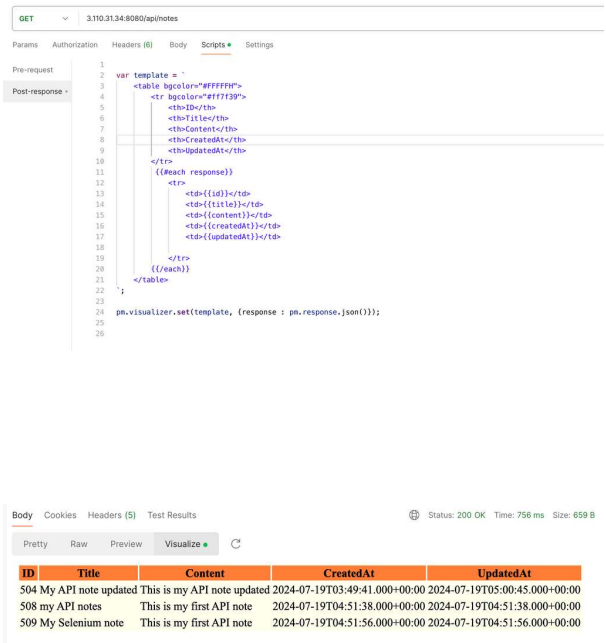
	Bearer Token Format	Verify that the Authorization header uses the format Bearer {{token}}.
	Valid Token	Verify that requests with a valid token return the expected status codes (200, 201, etc.).
	Invalid Token	Verify that requests with an invalid token return status code 401 (Unauthorized).
	Missing Token	Verify that requests without the Authorization header return status code 401 (Unauthorized).
	Expired Token	Verify that requests with an expired token return status code 401 (Unauthorized).

Stateless vs Stateful

Stateless vs Stateful	**Stateless and stateful are terms used to describe systems or protocols based on how they manage and store information.** 1. Stateless: a. In a stateless system or protocol, no information about previous interactions or requests is retained. b. Each request is handled independently, without any knowledge of the past. c. The server or system does not maintain any session or context between requests. d. The requests are self-contained and include all the necessary information for processing. e. Stateless systems are often simpler and more scalable since they don't require storing and managing session data. f. Examples of stateless protocols include HTTP, where each request is independent, and RESTful APIs that follow the stateless constraint. 2. Stateful: a. In a stateful system or protocol, information about the current state or context is maintained and associated with each client or session. b. The system keeps track of the client' s state, such as session data, preferences, or transactional information. c. The server stores and references this state for subsequent requests, allowing for continuity and maintaining a connection or session. d. Stateful systems require managing session data and maintaining synchronization between the client and server. e. Examples of stateful protocols include TCP, which establishes a connection and maintains a session until explicitly closed, and session-based authentication systems that require session tokens or cookies to track user state. Another examples are WebSocket, chat based applications. Stateless architectures, like RESTful APIs, are often preferred for scalability, ease of caching, and loose coupling between components. Stateful architectures are suitable when maintaining state or session information is necessary, such as in real-time applications or scenarios requiring session management, transaction tracking, or complex workflows.

Stateless Examples:	**Stateless Example:** 1. HTTP: The Hypertext Transfer Protocol (HTTP) is a stateless protocol widely used for communication between web browsers and web servers. Each request sent by the browser contains all the necessary information for the server to process it, such as the HTTP method, headers, and payload. The server does not retain any knowledge of past requests from the same client. Each request is treated independently. 2. RESTful APIs: Representational State Transfer (REST) is an architectural style for building web services. RESTful APIs follow the stateless constraint of HTTP, where each request from a client contains all the necessary information for the server to process it. The server does not store any session or context between requests. Authentication is typically handled through tokens or credentials provided with each request.
Stateful Examples:	**Stateful Example:** 1. TCP: The Transmission Control Protocol (TCP) is a stateful protocol used for reliable and ordered communication between networked devices. TCP establishes a connection between the client and server, creating a session. The connection is maintained until explicitly closed. TCP keeps track of the current state of the connection, including sequence numbers, acknowledgments, and window sizes, to ensure reliable delivery of data. 2. Session-based Authentication: Many web applications use session-based authentication, where a user's session is established upon successful login. The server assigns a unique session ID to the client and stores session data on the server-side. The session ID is typically stored in a cookie or included in each request. The server references the session data to authenticate subsequent requests and maintain user state throughout the session. 3. WebSocket is considered a stateful protocol. Unlike traditional HTTP, which is stateless, WebSocket maintains a persistent, bidirectional connection between the client and the server. This connection is established through an initial HTTP handshake and then upgraded to the WebSocket protocol.

Postman Template for Visualize:

Postman Template for Visualize:	var template = ` <table bgcolor="#FFFFFH"> <tr bgcolor="#ff7f39"> <th>ID</th> <th>Title</th> <th>Content</th> <th>CreatedAt</th> <th>UpdatedAt</th> </tr> {{#each response}} <tr> <td>{{id}}</td> <td>{{title}}</td> <td>{{content}}</td> <td>{{createdAt}}</td> <td>{{updatedAt}}</td>   </tr> {{/each}} </table> `; pm.visualizer.set(template, {response : pm.response.json()});	

Important_REST_API_Content-Types

Content-Type	Description
application/json	JSON format data
application/xml	XML format data
application/x-www-form-urlencoded	Form submissions with key-value pairs
multipart/form-data	Forms with file uploads
text/plain	Plain text content
application/octet-stream	Binary data
text/html	HTML content
application/pdf	PDF documents
application/zip	ZIP archive files
image/png	PNG image files
image/jpeg	JPEG image files
application/vnd.api+json	JSON API responses following JSON API specification
application/graphql	GraphQL queries
application/x-ndjson	Newline Delimited JSON
text/csv	CSV (Comma-Separated Values) content
application/vnd.ms-excel	Excel files
application/vnd.openxmlformats-officedocument.spreadsheetml.sheet	Excel files in the newer Office Open XML format
application/javascript	JavaScript code
application/x-yaml	YAML data
application/ld+json	Linked Data in JSON format

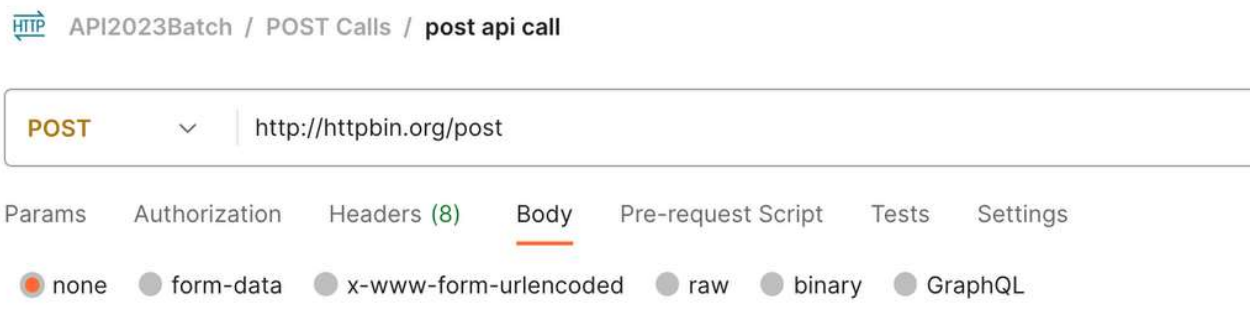
Various options while selecting the Request Body in postman:

Topics	Details

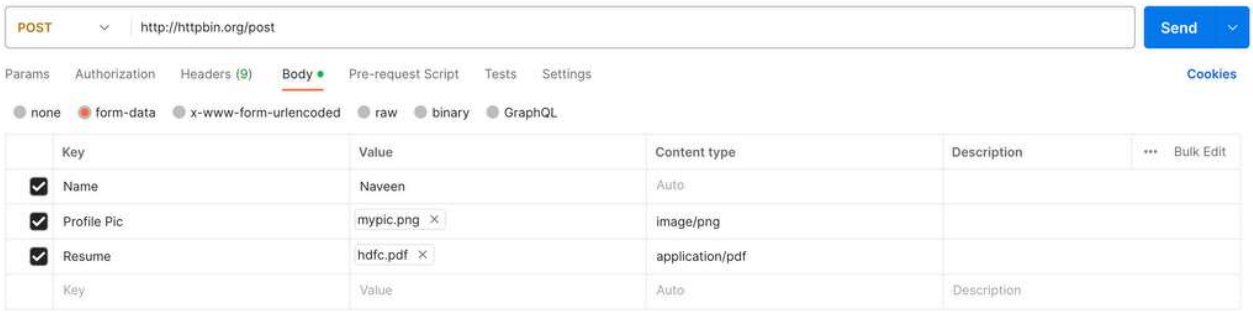
Various options while selecting the Request Body in postman:

Options for the request body in Postman, along with explanations and examples for each:

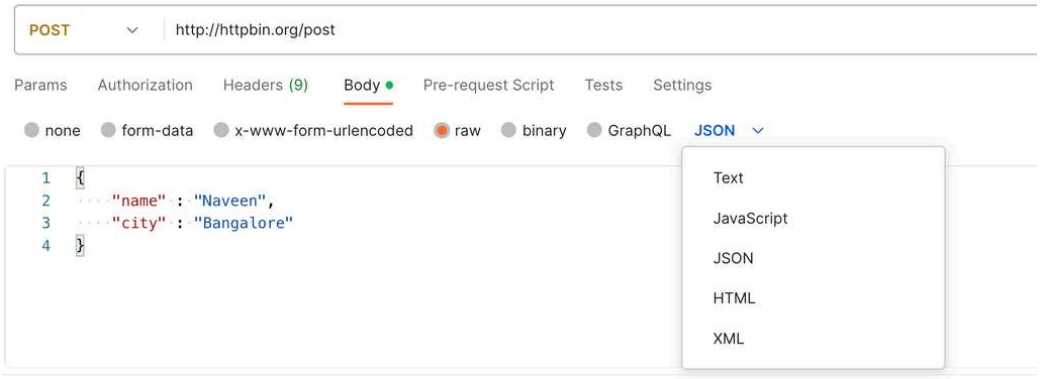
1. **No Body: none**
   - Explanation: This option is used when the request does not require a body, such as for GET or DELETE requests.
   - Example: For a GET request to retrieve a list of users, you don't need to include a request body.



2. **Form Data: form-data**
   - Explanation: Form Data is typically used when submitting data through HTML forms. It consists of key-value pairs where the values can be text or files.
   - Example: When submitting a form that includes fields like name, email, profile picture and resume pdf, you can use Form Data to send these key-value pairs in the request body.



3. **x-www-form-urlencoded:**
   - Explanation: This option is similar to Form Data and is commonly used for encoding simple key-value pairs in the request body.
   - Example: When sending data to an API that expects URL-encoded parameters, such as a search query with parameters like "q" and "sort", you can use x-www-form-urlencoded to send the data.
4. Raw: raw
   - Explanation: The Raw option allows you to send the request body in various data formats, such as JSON, XML, plain text, or other custom formats.
   - Example: When sending a POST request with JSON data, you can choose the Raw option and specify the data in JSON format, like {**"name": "naveen", "age": 30**}.
5. Binary: binary
   Explanation: The Binary option is used when you need to send binary data as the request body, such as files or images. Example: When uploading a profile picture to an API, you can choose the Binary option and attach the image file as the request body.
6. GraphQL:
   Explanation: The GraphQL option is specifically designed for making requests to GraphQL APIs. Example: When sending a GraphQL query or mutation to a GraphQL API, you can choose the GraphQL option and provide the query/mutation in the request body.
7. File:
   Explanation: The File option allows you to send files as the request body.
   Example: When uploading a file to an API endpoint that expects the file as the request body, you can choose the File option and select the file to include in the request.

In the Raw section of the request body in Postman, you have the following options for different data formats:	1. JSON: ◦ Explanation: Allows you to send data in JavaScript Object Notation (JSON) format. ◦ Example: { "name": "naveen", "age": 30, "email": "naveen@nal.com" } 2. XML: ◦ Explanation: Allows you to send data in Extensible Markup Language (XML) format. ◦ Example: <user> <name>John</name> <age>30</age> <email>john@example.com</email> </user> 3. HTML: ◦ Explanation: Allows you to send data in Hypertext Markup Language (HTML) format. ◦ Example: Hello World This is a sample HTML content. 4. JavaScript: Explanation: Allows you to send data in JavaScript format.Example: var person = { name: "John", age: 30, email: "john@example.com" }; 5. Text: Explanation: Allows you to send plain text data. ◦ Example: "This is a plain text message."  ◦