Jan API Batch 2025 Batch Notes

What is an API?

An API (Application Programming Interface) is like a waiter at a restaurant. Just as a waiter takes your order and brings you food from the kitchen, an API helps different software applications communicate with each other. It's a set of rules that allows one application to access the features or data of another application.

Example of an API

Think about ordering food using a food delivery app:

1. You open the app and browse restaurants (the app is using the GPS API to find your location)
2. You check the weather before ordering (the app is using a Weather API to show current conditions)
3. You pay for your order (the app is using a Payment API to process your credit card)

In each case, the app is talking to different services through their APIs to get specific information or perform specific tasks.

What is a Web Service?

A Web Service is a specific type of API that uses web protocols (typically HTTP) to communicate over the internet. Think of it as an API that specifically works through the web. All web services are APIs, but not all APIs are web services.

Example of a Web Service

Let's say you're building a travel booking website:

1. Your website needs flight information from multiple airlines
2. Each airline provides a web service that your website can connect to
3. Your website uses these web services to fetch real-time flight prices, schedules, and availability
4. The communication happens using standard web protocols (HTTP/HTTPS)

Key Differences

1. Communication Method

- **API**: Can use any communication method (local files, databases, hardware, etc.)
- **Web Service**: Must use web protocols (HTTP/HTTPS)

2. Format

- **API**: Can transfer data in any format (binary, custom protocols)
- **Web Service**: Usually transfers data in standard web formats (XML, JSON)

3. Access

- **API**: Can be used locally within a single system or over a network
- **Web Service**: Must be accessed over a network (usually the internet)

Real-World Analogy

Think of it this way:

- An API is like having access to all the services in a building (elevator, security system, parking gate)
- A Web Service is specifically like having access to only the services that you can use through the internet

Side-by-Side Comparison

Aspect	API	Web Service
Definition	A general interface for software communication	A specific type of API that operates over the web
Protocol	Can use any protocol or method	Must use web protocols (HTTP/HTTPS)
Network Requirement	Can work offline or online	Requires internet connectivity
Data Format	Any format (binary, text, custom)	Usually XML, JSON, or SOAP
Bandwidth Usage	Can be very lightweight	Generally requires more bandwidth due to HTTP overhead
Security	Varies based on implementation	Always needs web security measures (SSL/TLS)
Usage Scope	Can be public or private	Typically public
Implementation Cost	Can be low (for simple APIs)	Generally higher due to web infrastructure needs
Speed	Can be very fast (especially locally)	Dependent on internet connection and server response
Platform Dependency	May be platform-specific	Platform-independent
Integration Complexity	Can be simple or complex	Generally more complex due to web protocols
Examples	File system API, Database API	RESTful services, SOAP services

Common Use Cases

API Examples

1. **Operating System APIs**
   - Windows API (WinAPI) for developing Windows applications
   - iOS Core APIs (CoreLocation, CoreGraphics) for iOS app development
   - Android APIs for accessing device features (Camera, Sensors, Bluetooth)
2. **Database APIs**
   - MongoDB's native drivers for different programming languages
   - MySQL Connector APIs
   - PostgreSQL libpq C library
3. **Browser APIs**
   - DOM API for manipulating web page content
   - Geolocation API for getting user's location
   - WebRTC API for real-time communication
   - Canvas API for drawing graphics
   - Web Audio API for sound processing
4. **Hardware APIs**
   - Printer APIs for print management
   - USB API for device communication
   - Graphics card APIs (OpenGL, DirectX)
5. **Programming Language APIs** Java Collections API Python's Standard Library APIs JavaScript's Math and Date APIs

Web Service Examples

1. **Payment Services**
   - PayPal REST API for payment processing
   - Stripe API for online transactions
   - Square API for point-of-sale integration
   - Braintree API for payment gateway services
2. **Cloud Services**
   - Amazon Web Services (AWS)

        - S3 for storage
        - EC2 for computing
        - Lambda for serverless functions
    - Google Cloud Platform
        - Cloud Storage
        - Cloud Functions
        - Cloud Vision API
    - Microsoft Azure
        - Azure Blob Storage
        - Azure Functions
        - Azure Cognitive Services
3. **Social Media Services**
    - Twitter API for tweets and user data
    - Facebook Graph API for social integration
    - Instagram API for media sharing
    - LinkedIn API for professional networking
    - YouTube API for video integration
4. **Map and Location Services**
    - Google Maps API
        - Directions API
        - Places API
        - Geocoding API
    - Mapbox API for custom maps
    - OpenStreetMap API for geographic data
    - Here Maps API for navigation
5. **Business and Communication Services** Salesforce API for CRM integration Twilio API for SMS and voice services SendGrid API for email services Slack API for workspace communication Zoom API for video conferencing
6. **Data and Analytics Services** Weather APIs (OpenWeatherMap, WeatherAPI) Financial APIs (Alpha Vantage for stock data) Currency conversion APIs (Fixer.io) Analytics APIs (Google Analytics) Machine Learning APIs (IBM Watson)

In Practice: Code Examples

Using an API (Java File API Example)

```java
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class FileAPIExample {
    public void writeToFile(String fileName, String content) {
        try {
            // Using Java's File API to create and write to a file
```

Using a Web Service (cURL Example)

```bash
# GET request to a weather web service
curl -X GET "https://api.weatherapi.com/v1/current.json?key=YOUR_API_KEY&q=London" \
     -H "Content-Type: application/json"

# POST request to create a resource
curl -X POST "https://api.example.com/users" \
     -H "Content-Type: application/json" \
     -H "Authorization: Bearer YOUR_TOKEN" \
```

> The Java example shows how to use a local API (File API) to perform file operations, while the cURL examples demonstrate how to interact with web services using HTTP requests. The key difference is that the API example works with local resources directly, while the web service examples require network communication and follow HTTP protocols.

Summary

While all web services are APIs, not all APIs are web services. The main difference is that web services specifically operate over the web using standard web protocols, while APIs can work in any

environment and use any communication method. Think of web services as a subset of APIs that are specifically designed for web-based communication.

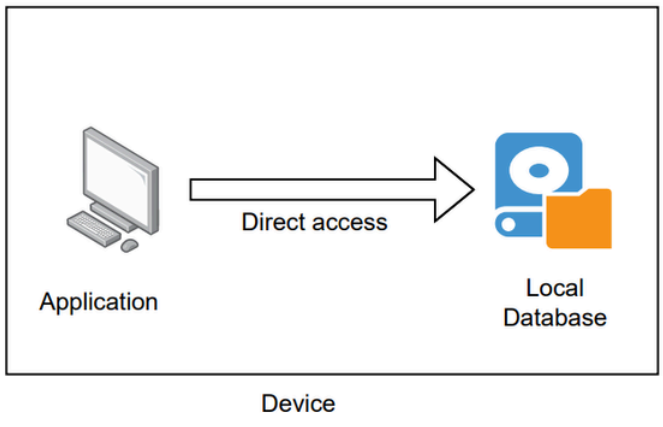
API Terminologies:

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.

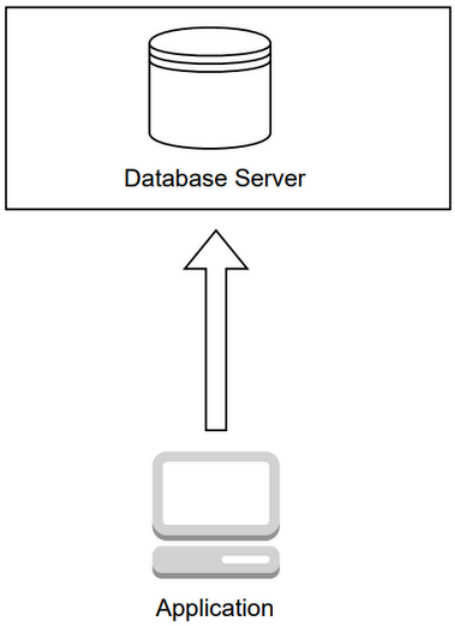
System Architecture:

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.



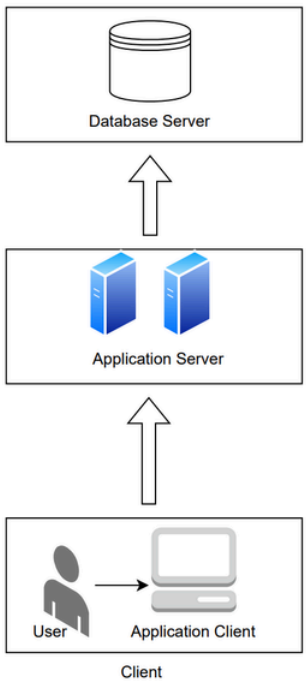
1 - Tier Architecture

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

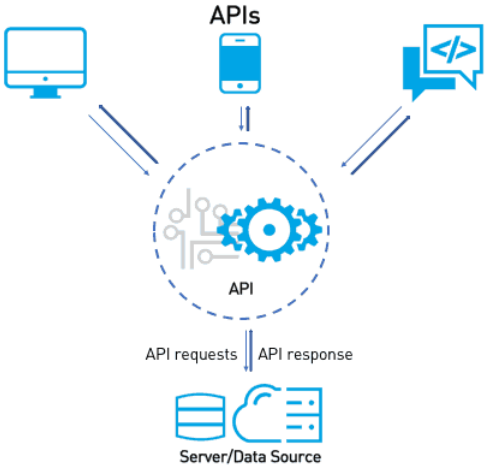
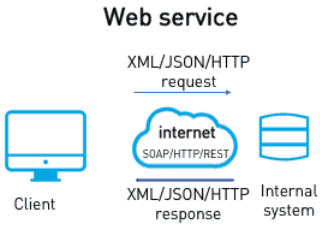
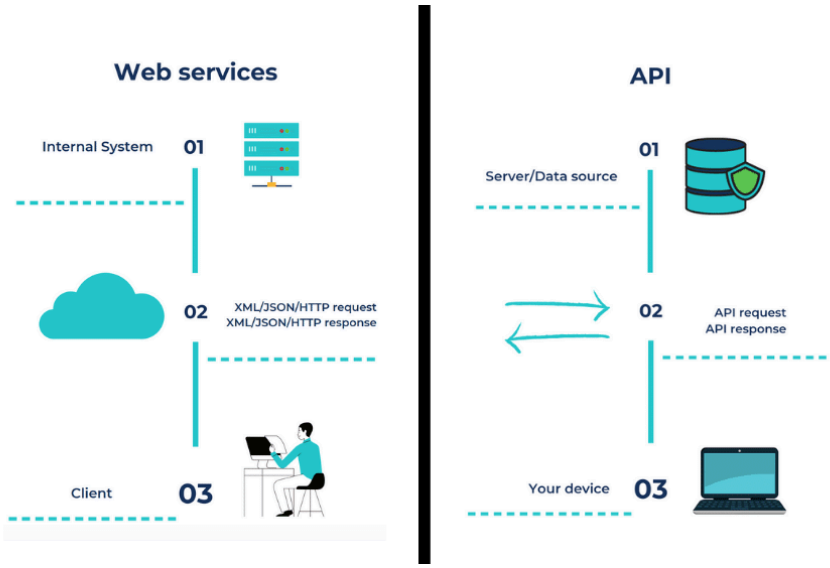
3. **Three Tier Architecture**: In this widely used model, an application layer or API is added between the client and db server layers. This prevents exposing the direct data access to public. The APIs does some authentications, then provide access. It enhances security, data integrity, and scalability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

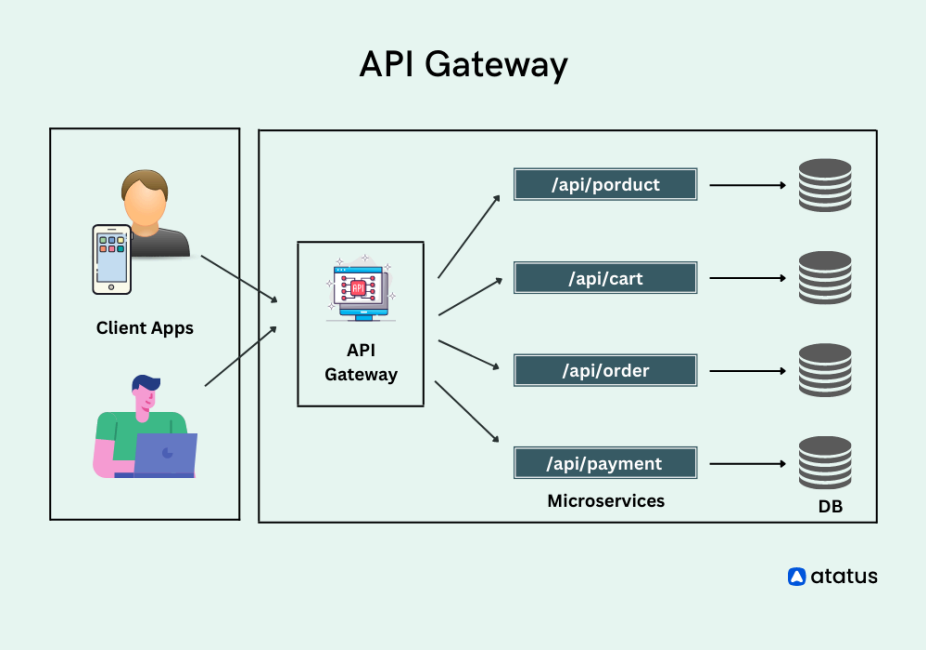
Topics	Details	Comments

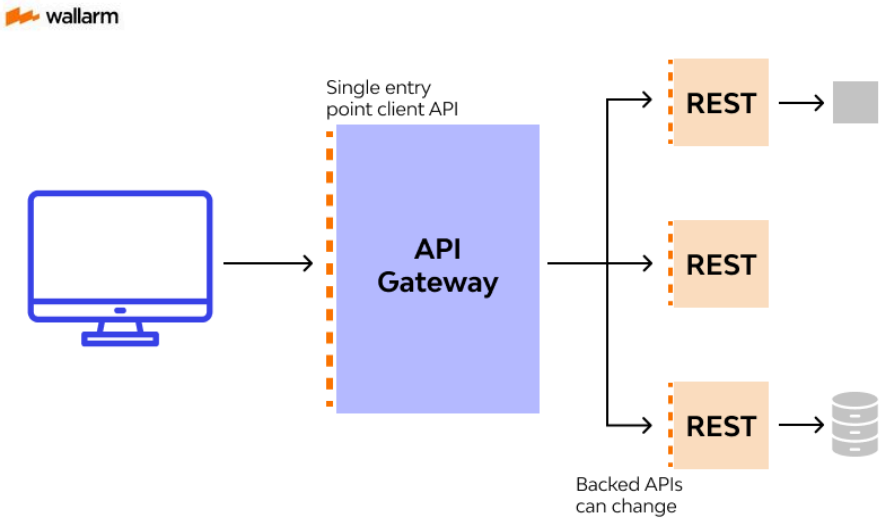
API vs WebServices	**API (Application Programming Interface):** • **An API is a set of rules and protocols that allows different software applications to communicate and interact with each other.** • **It defines how requests and responses should be structured, what data formats to use, and what functionalities 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.**    **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.	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.

	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.
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

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.

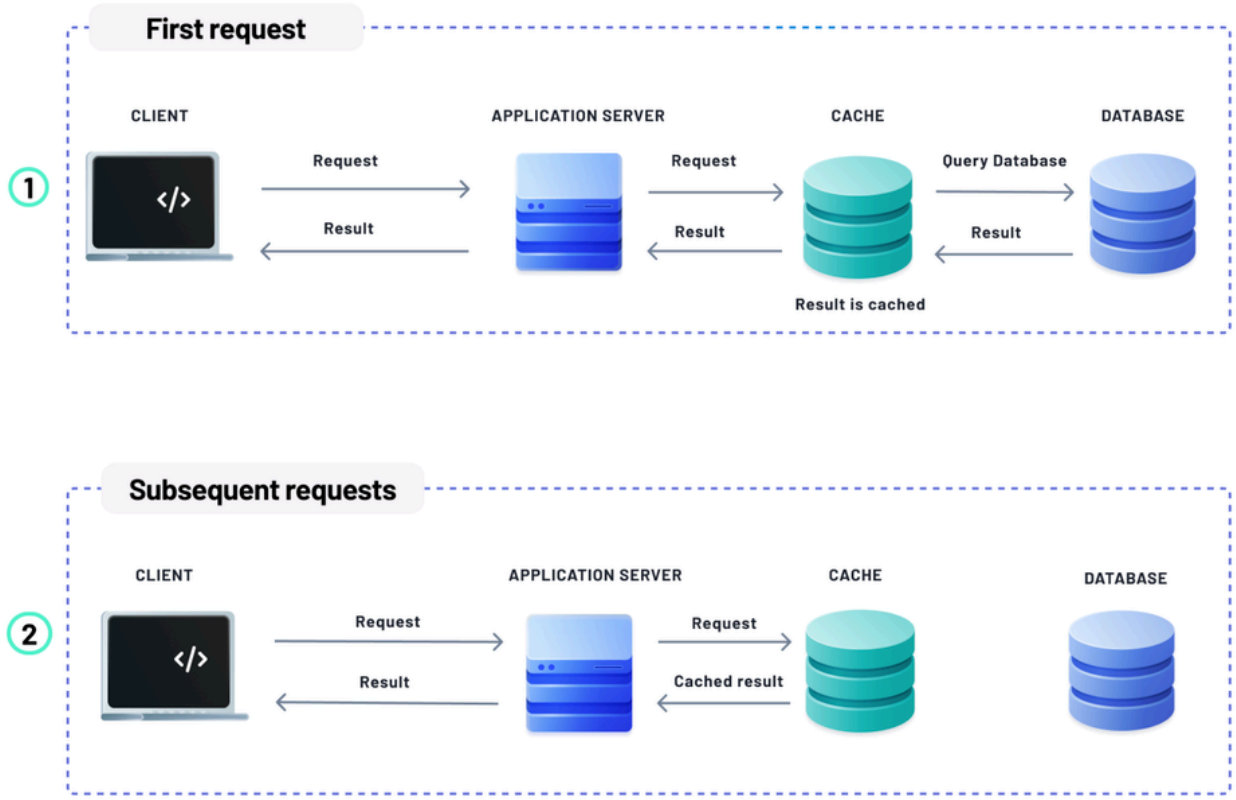
This helps achieve high availability, scalability, and efficient resource utilization.

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.

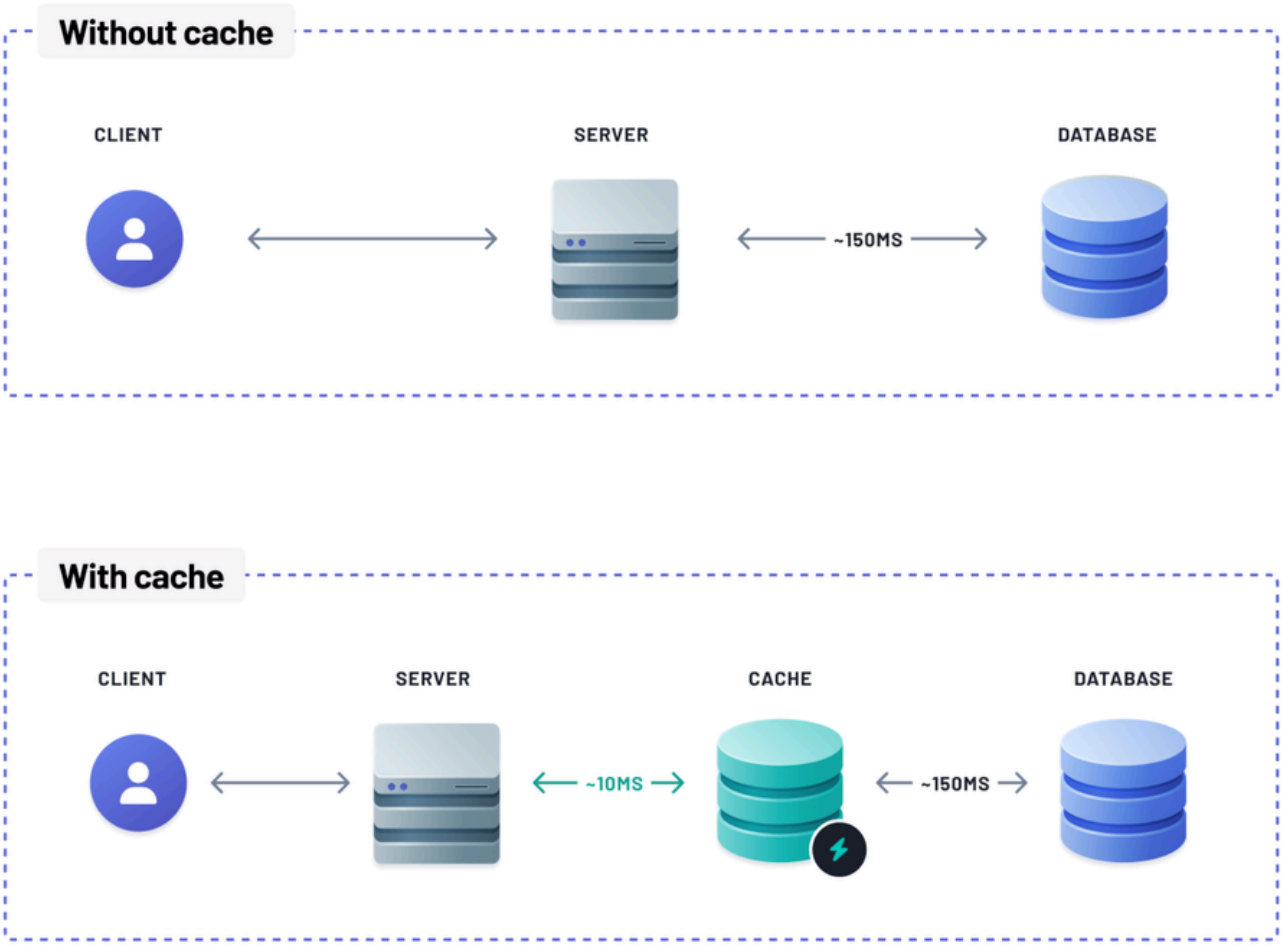
What is Caching?

Caching significantly improves performance

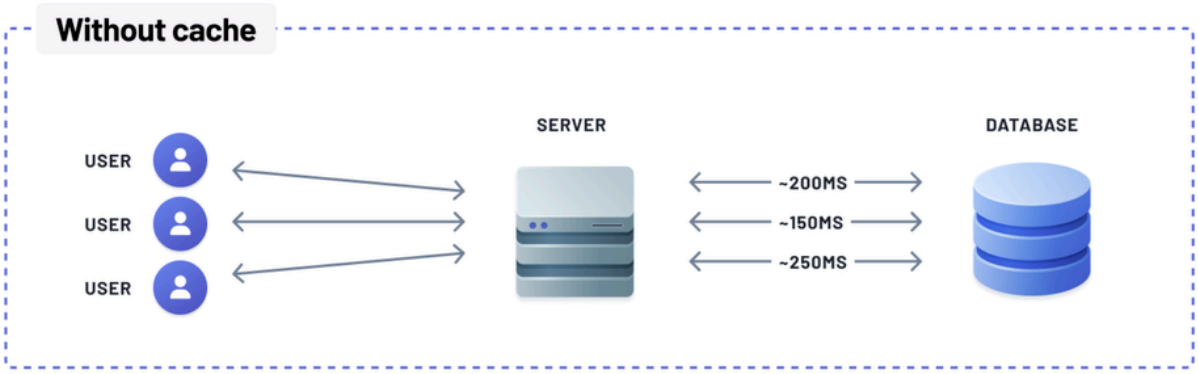
Using a cache to store database query results can significantly boost the performance of your application.
A cache is much faster and usually hosted closer to the application server, which reduces the load on the main database, accelerates data retrieval, and minimizes network and query latency.

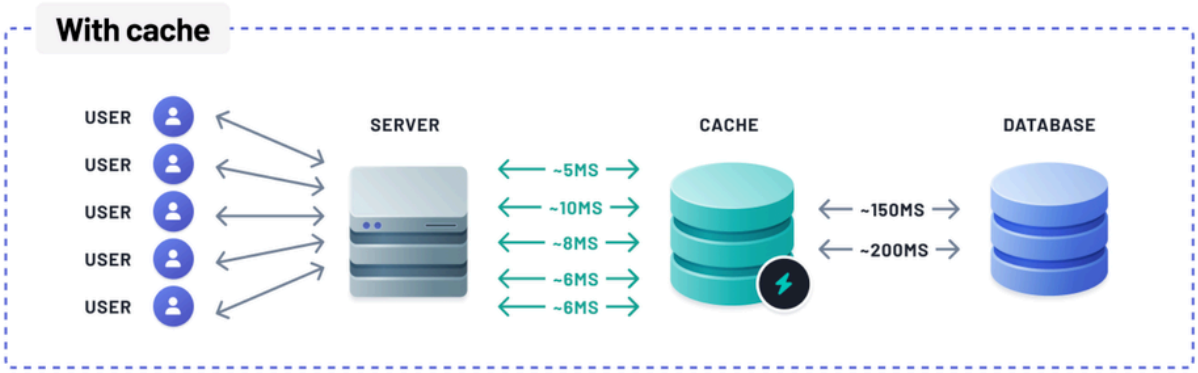


Caching reduces CPU usage, disk access, and network utilization by quickly serving frequently accessed data to the application server, bypassing the need for a round trip to the database.



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.





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 (webservices APIs).
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: CRUD
C - Create POST : email id is same: 400 - error message
R-Read GET
U-Update PUT/PATCH
D-Delete DELTE

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. /products/{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.

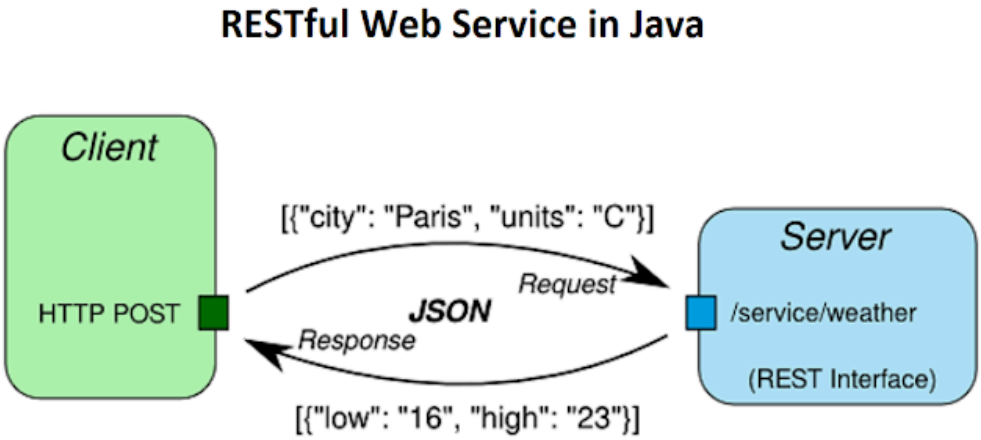
10. HATEOAS: Consider adopting HATEOAS (Hypermedia as the Engine of Application State) to make your API self-descriptive. Include links in the API responses that allow clients to navigate and discover related resources.

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.

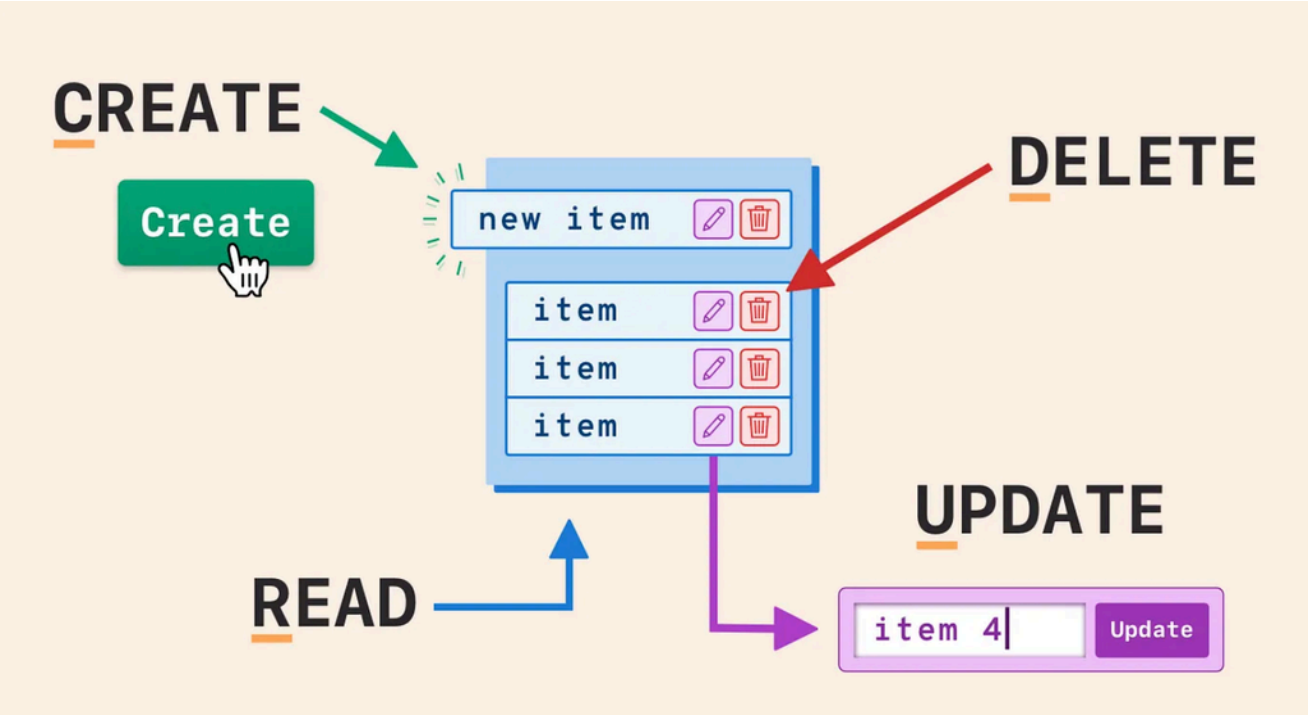
Simple example of REST WebService in Java:

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

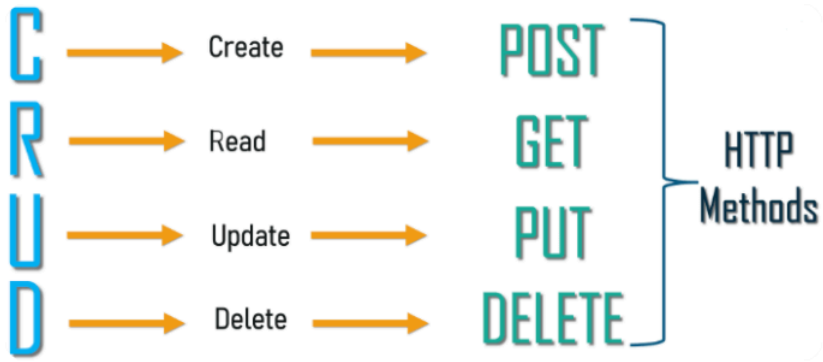
RFC - Request For Comments
IETF: Internet Engineering Task Force

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

CRUD:



CRUD against HTTP METHODS:



Safety, cacheability, and idempotency of common HTTP methods:

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

- PATCH is generally NOT idempotent. Unlike PUT which replaces an entire resource, PATCH performs partial modifications, and repeated PATCH operations with the same payload might produce different results depending on the current state of the resource.
- For POST cacheability, while the note is technically correct that POST responses can be cached under specific circumstances, POST is generally considered "Not Cacheable" by default according to the HTTP specification. The asterisk and note might be misleading since it's an edge case rather than a general characteristic.

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.

Idempotency in HTTP Methods

HTTP defines several methods (verbs) for different types of requests.
These methods can be categorized by whether they are idempotent or non-idempotent, influencing how a system handles retries and preventing unintended side effects.

Idempotent Methods:

- **GET:** Retrieves data from a resource. GET requests are inherently idempotent because they only read data and do not alter the server's state.
  - **Example:** Accessing a blog post by making a GET request to /posts/123 will simply retrieve that post, without modifying any server data. Whether you retrieve it once or a thousand times, the post remains unchanged.
- **PUT:** Update or completely replace an existing resource. PUT requests are idempotent because the final state is the same whether the PUT request is executed once or multiple times.
  - **Example:** Updating user information by making a PUT request to /users/45 with updated user details will overwrite the user's data with the new information provided. Executing the same PUT

request repeatedly results in the same final user data on the server.

- **DELETE:** Removes a resource from the server. DELETE requests are idempotent because deleting a resource that's already been deleted has no further effect.
  - **Example:** Deleting an item by making a DELETE request to /items/678 will remove the item. If you attempt the DELETE request again, it will have no effect since the item no longer exists.

Non-Idempotent Methods:

- **POST:** Creates a new resource on the server. POST requests are non-idempotent because each request usually results in the creation of a new resource.
  - **Example:** Creating a new order by making a POST request to /orders with order details will generate a new order each time the request is made.

==========

POSTMAN:

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

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.

==========

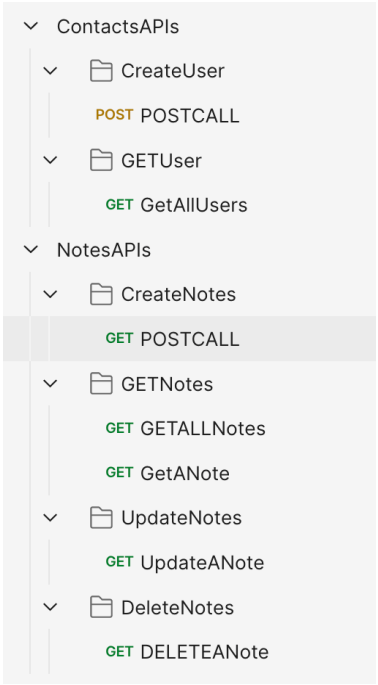
Postman Feature	Explanation
API Testing	Postman is primarily known as an API testing tool. It allows you to send HTTP requests and verify the responses, making it easy to test API endpoints and functionalities.
Request Builder	Postman provides a user-friendly interface for building HTTP requests. You 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://65.0.80.58:8080** • Data Base URL: https://auth-db696.hstgr.io/index.php?route=/table/sql&db=u811712038_notes_app&table=notes • Table Name: **Notes** • DB: MY SQL Server	• **UserName: u811712038_naveenautomate** • **Password: Spring@1234554321**	Please do not delete the table name in DB.

1. **POST: /api/notes**
Create a new note
• **URL:** http://65.0.80.58:8080/api/notes
	{
	"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
• **URL:** http://65.0.80.58:8080/api/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
• **URL:** http://65.0.80.58:8080/api/notes/5
	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
• **URL:** http://65.0.80.58:8080/api/notes/3
	{
	"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://65.0.80.58:8080/api/notes
	NA	NA
		
		Response after GET call:
		{
		"timestamp": "2023-06-09T06:16:03.024+00:00",
		"status": 404,
		"error": "Not Found",
		"path": "/api/notes/4"
		}

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.	

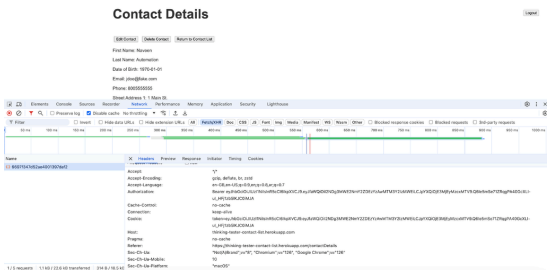
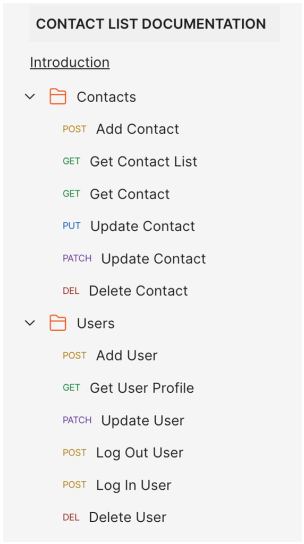
	Request Body	Verify that a valid request body:
		{
		"firstName": "Naveen",
		"lastName": "Automation",
		"birthdate": "1970-01-01",
		"email": "jdoe@fake.com",
		"phone": "8005555555",
		"street1": "1 Main St.",
		"street2": "Apartment A",
		"city": "Anytown",
		"stateProvince": "KS",
		"postalCode":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

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 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/Websocket, 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()});	

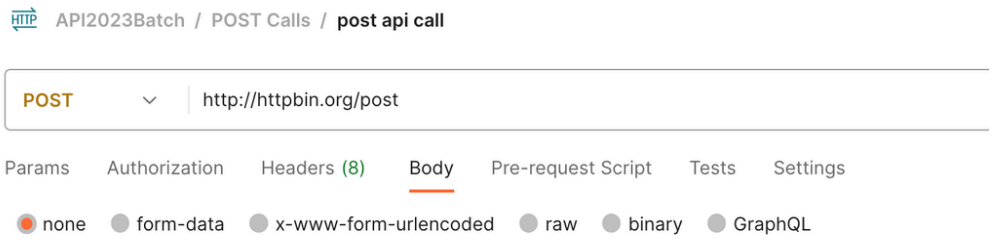
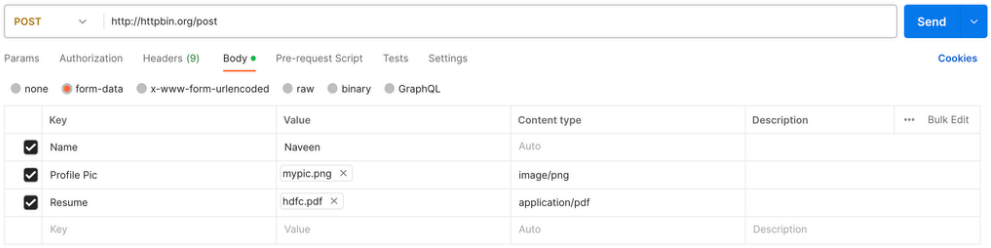
Important_REST_API_Content-Types

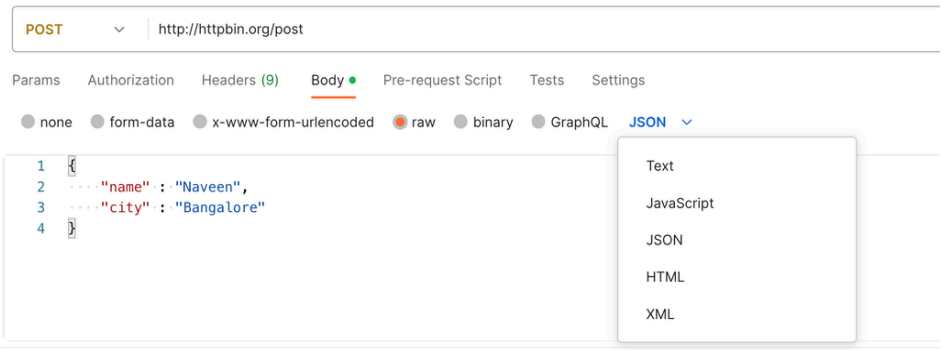
Content-Type	Description
application/json	JSON format data
application/xml	XML format data
application/x-www-form-urlencoded	Form submissions with key-value pairs
multipart/form-data	Forms with file uploads
text/plain	Plain text content
application/octet-stream	Binary data
text/html	HTML content

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."  ◦

Important Interview Questions:

Important Interview Questions	Answer	Comments

POST vs PUT?	**According to the RFC 7231, which defines the semantics and syntax for HTTP/1.1, there is a distinction between the POST and PUT methods based on their intended purposes:** 1. **POST (HTTP POST):** Purpose: The POST method is used to submit data to be processed by the identified resource. It is generally used when creating a new resource on the server or submitting data for further processing. Idempotence: POST requests are not required to be idempotent. Multiple identical POST requests may result in different outcomes each time they are submitted. Safety: POST requests are not considered "safe" because they can have side effects on the server or modify resources. Example: Submitting a form, creating a new user account, or adding a new entry to a database are common use cases for the POST method. 2. **PUT (HTTP PUT):** Purpose: The PUT method is used to upload or replace the entity identified by the Request-URI. It is typically used to update or overwrite an existing resource with the provided representation. Idempotence: PUT requests are expected to be idempotent. Sending the same PUT request multiple times should have the same effect as sending it once. Safety: PUT requests are not considered "safe" as they can modify or replace the target resource. Example: Updating an existing user's profile information or replacing an existing document with a new version are typical use cases for the PUT method.	**The main different between POST and PUT, as per the RFC, is that POST is used for creating or submitting data, while PUT is used for updating or replacing existing resources. Additionally, POST requests are not required to be idempotent, whereas PUT requests should be idempotent, meaning repeated identical requests should have the same effect as a single request.**
Can I use PUT to create a new Resource?	Technically, according to the HTTP specifications defined in RFC 7231, the PUT method is primarily intended for updating or replacing an existing resource, rather than creating a new resource. However, in practice, some APIs and frameworks may allow or support using PUT for creating new resources. While it's not recommended to use PUT for creating new resources from a strict adherence to the HTTP specifications, the decision ultimately depends on the API design. Some APIs may choose to interpret a PUT request on a non-existing resource as an indication to create a new resource at the specified URI. If you're working on an API, it is generally better to follow the conventional usage of the HTTP methods to ensure consistency and clarity. In most cases, creating new resources is better achieved using the POST method. If you're working with an existing API or framework that allows using PUT for resource creation, make sure to consult its documentation or guidelines to understand the expected behavior.	

Other important interview questions:	1. **Q: What is the difference between GET and POST requests?** A: GET is used to retrieve data from a server, while POST is used to send data to a server to create a new resource or perform an action. 2. **Q: When should you use GET versus POST?** A: Use GET when you want to retrieve data, and use POST when you want to send data to create or update a resource on the server. 3. **Q: What is the idempotence property of HTTP methods, and how does it relate to PUT and DELETE?** A: Idempotence means that making multiple identical requests should have the same effect as a single request. PUT and DELETE are expected to be idempotent, meaning sending the same request multiple times should have the same outcome as sending it once. 4. **Q: What are the potential security risks of using GET requests to modify or delete resources?** A: GET requests should not modify or delete resources as they can be cached, bookmarked, or preloaded by browsers or proxies. This can lead to unintentional modifications or deletions. It's recommended to use POST, PUT, or DELETE methods for modifying or deleting resources. 5. **Q: What is the purpose of the OPTIONS HTTP method?** A: The OPTIONS method is used to retrieve the communication options available for a given resource or server. It can provide information about supported methods, headers, and other capabilities. 6. **Q: What is the significance of the HTTP status codes in response to a request?** A: HTTP status codes provide information about the success or failure of a request. They indicate whether a request was processed successfully (2xx), redirected (3xx), had a client error (4xx), or encountered a server error (5xx). 7. **Q: How can you pass data in a POST request?** A: Data can be passed in a POST request through the request body. The body can contain data in various formats such as JSON, form data, or URL-encoded parameters. 8. **Q: How can you pass data in a GET request?** A: In a GET request, data can be passed through query parameters appended to the URL. For example, **https:/gorest.co.in/public/v2/users?id=123&name=John**. 9. **Q: Can you have a request body in a DELETE request?** A: Technically, the HTTP specifications do not prevent a request body in a DELETE request. However, it is not a common practice, and DELETE requests typically do not have a request body. Deletions are usually based on the resource identification provided in the URL or request parameters.

Steps to Call Selenium Endpoints from Postman

1. Start Selenium Standalone Server :
If using Selenium Standalone, start it using:
java -jar selenium-server.jar standalone

Once the server is running, it exposes a REST API (typically at http://localhost:4444).

2. Create a New Session
Method: POST
URL: http://localhost:4444/session
Headers: Content-Type: application/json
Body (JSON):

```
{
  "capabilities": {
    "alwaysMatch": {
      "browserName": "chrome"
    }
  }
}
```

Response: It will return a sessionId which you'll use for further API calls.

3. Navigate to a URL
Method: POST

URL: http://localhost:4444/session/{sessionId}/url
Body (JSON):

```
{
  "url": "https://www.google.com"
}
```

4. Find an Element
Method: POST
URL: http://localhost:4444/session/{sessionId}/element
Body (JSON):

```
{
 "using": "xpath",
"value": "//textarea[@name='q']"
}
```

Response: Returns an elementId, which you can use for actions.

5. Send Keys to an Input Field
Method: POST
URL: http://localhost:4444/session/{sessionId}/element/{elementId}/value
Body (JSON):

```
{
 "text": "Hello, Naveen Automation Labs!"
}
```

6. Click an Element
Method: POST
URL: http://localhost:4444/session/{sessionId}/element/{elementId}/click
Body (JSON:
{}

7. Close the Browser Session
Method: DELETE
URL: http://localhost:4444/session/{sessionId}

==========

CURL:

Curl - Client URL :
Install Curl on windows: https://linuxhint.com/install-use-curl-windows/
Curl is already available on MAC machine (by default)

List of some important options commonly used with curl: Check Curl version:
curl --version
Curl options:
-X, --request <command>: Specifies the HTTP request method (GET, POST, PUT, DELETE, etc.). (By defauly Curl uses GET method)
-H, --header <header>: Adds a custom header to the request.
-d, --data <data>: Sends data in the request body (used with POST or PUT requests).
-i, --include: Includes the response headers in the output.
-o, --output <file>: Saves the response body to a file.
-L, --location: Follows redirects.
302 redirection - moved to some other server premamnently client -- server 1,--> 2,3, 4, 5
-u, --user <user:password>: Specifies the user and password for basic authentication.
-k, --insecure: Allows insecure SSL connections.
-x, --proxy <[protocol://]host[:port]>: Specifies a proxy to use for the request.
-v, --verbose: Provides detailed debugging information.

Imp Points to remember:

1. Don't forget to include URL inside single quotes if you include more than one query parameter. ex:
curl 'https://gorest.co.in/public/v2/users/?status=active '
2. By defauly curl uses GET method
3. use the curl -m option to provide a timeout for HTTP requests. If the server will not return any response within a specified time period then curl will exit.
ex: curl -m 1 'http://httpbin.org/get'
4. send a file via HTTP POST: use @
curl --location 'http://httpbin.org/post' \
--form 'file=@"/Users/naveenautomationlabs/Desktop/hdfc.pdf"'
5. Basic Auth:
https://the-internet.herokuapp.com/basic_auth
Ex:
curl -i -u "admin:admin" -X GET "https://the-internet.herokuapp.com/basic_auth"
6. Basic Auth:
If you dont want to maintain the password in your shell history. Leave password blank.curl will prompt for a password when it tries to authenticate on the server
Ex:
curl -i -u "admin" -X GET "https://the-internet.herokuapp.com/basic_auth"

cUrl Examples for REST API List users:

curl -i -H "Accept:application/json" -H "Content-Type:application/json" -H "Authorization: Bearer e4b8e1f593dc4a731a153c5ec8cc9b8bbb583ae964ce650a741113091b4e2ac6" -X GET "https://gorest.co.in/public/v2/users"

Create user

curl -i -H "Accept:application/json" -H "Content-Type:application/json" -H "Authorization: Bearer e4b8e1f593dc4a731a153c5ec8cc9b8bbb583ae964ce650a741113091b4e2ac6" -XPOST "https://gorest.co.in/public/v2/users" -d '{"name":"Tenali Ramakrishna", "gender":"male", "email":"tenali.ramakrishna@15ce.com", "status":"active"}'

Update user

curl -i -H "Accept:application/json" -H "Content-Type:application/json" -H "Authorization: Bearer e4b8e1f593dc4a731a153c5ec8cc9b8bbb583ae964ce650a741113091b4e2ac6" -XPATCH "https://gorest.co.in/public/v2/users/1932" -d '{"name":"Allasani Peddana", "email":"allasani.peddana@15ce.com", "status":"active"}'

Delete user

curl -i -H "Accept:application/json" -H "Content-Type:application/json" -H "Authorization: Bearer e4b8e1f593dc4a731a153c5ec8cc9b8bbb583ae964ce650a741113091b4e2ac6" -XDELETE "https://gorest.co.in/public/v2/users/1932"

==========

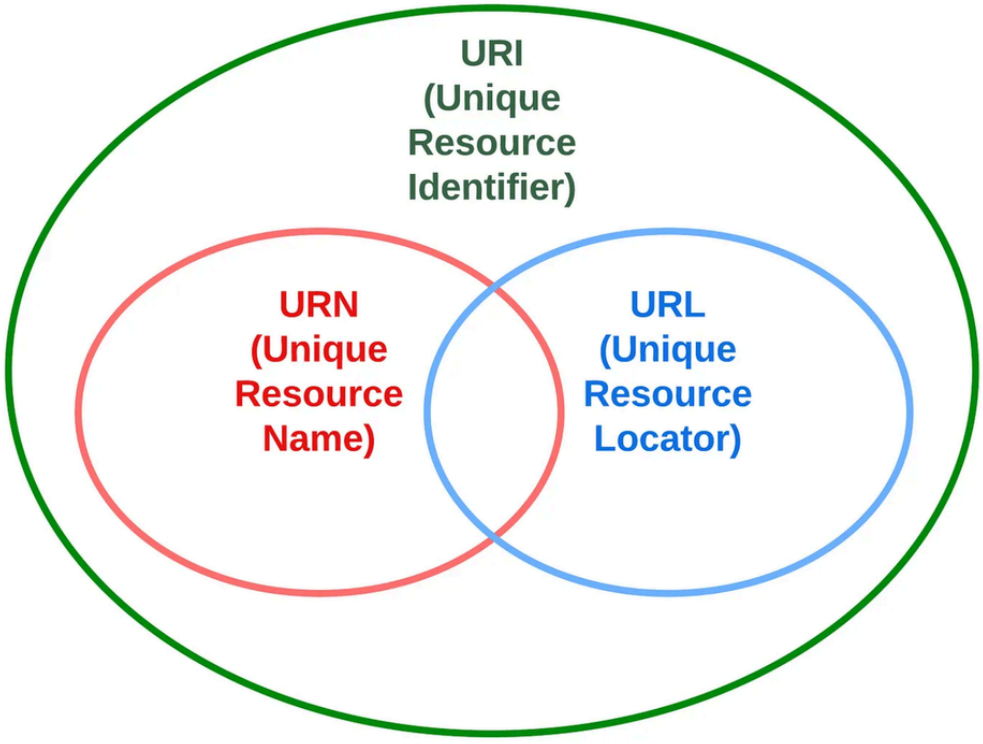
URI vs URL vs URN:

- **URI (Uniform Resource Identifier)**: A general term that encompasses both URLs and URNs, as well as other schemes that identify resources.
- **URL (Uniform Resource Locator)**: A specific type of URI that provides a means to locate the resource. All URLs are URIs.
- **URN (Uniform Resource Name/Number)**: Another specific type of URI that provides a unique name for a resource within a namespace but does not provide location information.

> **a URL is like a home address, a URI is like an identity card, and a URN is like a social security number.**

- **URL as a Home Address**:
  URL: Just like a home address provides specific information on where you can find a house, a URL provides the details needed to locate a resource on the internet.
  - For example, https://www.naveenautomationlabs.com/path/to/resource is a URL that tells you where to find a specific web page.
- **URI as an Identity Card**:
  URI: An identity card provides information to identify a person without specifying their exact location. Similarly, a URI identifies a resource but may not necessarily provide information on how to locate it.
  - All URLs and URNs are types of URIs, but a URI could be more general and include other types of identifiers.
- **URN as a Social Security Number**:
  URN: Just like a Social Security Number uniquely identifies a person within a specific system without providing location information, a URN uniquely identifies a resource by name within a specific namespace.
  - For example, urn:isbn:0451450523 identifies a book by its ISBN but doesn't provide a means to locate it.

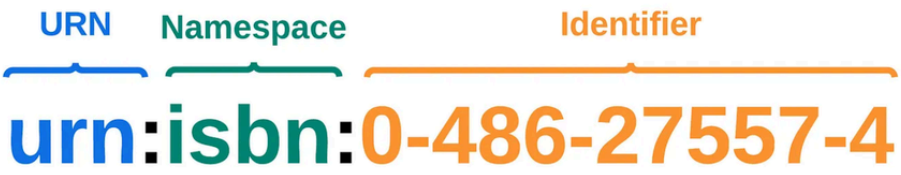
ISBN (International Standard Book Number)
ISSN (International Standard Serial Number)
https://www.bookfinder.com/search/?isbn=9780763695880&st=xl&ac=qr



Relationship between URI, URN and URL

Typical structure of a URL

A URN typically follows the following structure:

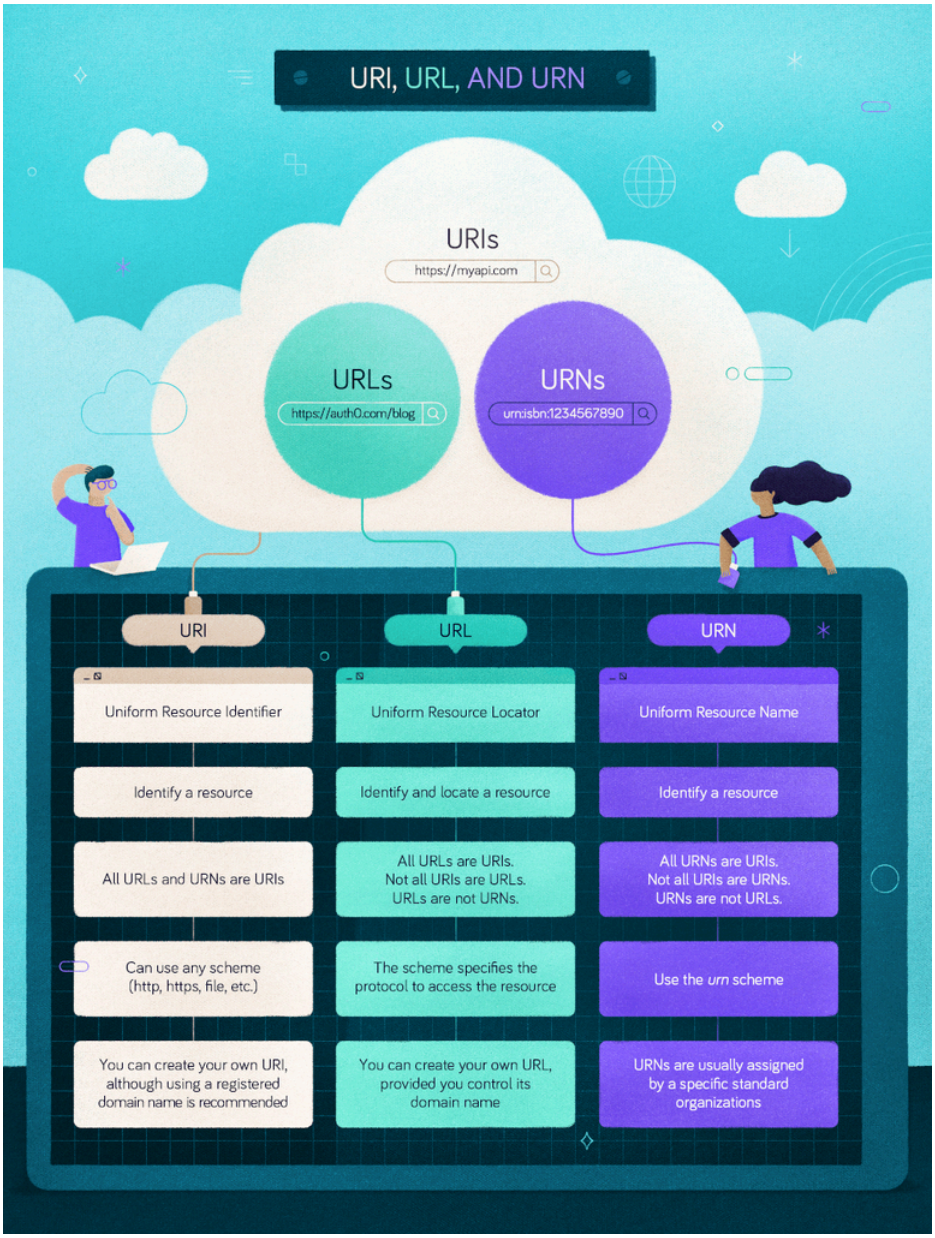


An Example with URI + URL + URN

naveen.co.uk

http://www.naveenautomatiolabs.com:8080/path/to/resource?urn=urn:isbn:0451450523#fragment

```
Protocol (Scheme)   : http
Subdomain           : www
Domain              : naveenautomationlabs
TLD (Top Level Domain): com
Port                : 8080
Path                : /path/to/resource
Query               : urn=urn:isbn:0451450523
Fragment            : fragment
```



URI vs URL vs URN	Examples	

URI vs URL vs URN	URLs (location + access method): • http://university.edu/students/profile?id=12345 (web resource) • sftp://fileserver.company.com/documents/report.pdf (secure file transfer) • ldap://ldap.company.com:389/cn=John%20Doe,dc=example,dc=com (directory service) • jdbc:mysql://localhost:3306/mydatabase (database connection) • git://github.com/user/project.git (git repository) • file:///C:/Users/username/Documents/file.txt (local file) • rtsp://streaming.company.com/live/stream1 (streaming media) URNs (unique names without location): • urn:ietf:rfc:2648 (identifies an IETF specification) • urn:nid:12345-67890 (national identity number) • urn:doi:10.1000/182 (digital object identifier for academic papers) • urn:lex:eu:council:directive:2010-03-09;2010-19-EU (legal document identifier) • urn:epc:id:sgtin:0614141.107346.2017 (product code) • urn:oasis:names:specification:docbook:dtd:xml:4.1.2 (technical specification) • urn:service:sos (standardized service identifier) Complex URI examples: • mongodb://username:password@host:port/database?options (database connection string) • redis://user:secret@localhost:6379/0?timeout=10s (Redis connection) • s3://mybucket/folder/file.jpg (Amazon S3 object) • spotify:track:2TpxZ7JUBn3uw46aR7qd6V (Spotify resource) • tel:+1-816-555-1212 (telephone number) • news:comp.infosystems.www.servers.unix (newsgroup) • magnet:?xt=urn:btih:c12fe1c06bba254a9dc9f519b335aa7c1367a88a (BitTorrent hash)	**The key difference remains: URLs tell you how to find and access the resource, while URNs just give it a persistent unique name. Both are types of URIs.**

==========

Postman Collection: GET-Put Chaining:

Collection Information:

- **Name:** GET-Put Chaining

Important Points:

- **Authentication:**
  - Each request is authenticated using a Bearer token: e4b8e1f593dc4a731a153c5ec8cc9b8bbb583ae964ce650a741113091b4e2ac6.
- **Variable Management:**
  - user_ids: Stores all user IDs fetched from the Get Users request.
  - singleUserId: Stores the current user ID to be updated.
- **Chaining Mechanism:**
  The Get Users request fetches all user IDs and stores them in user_ids.The Update User request uses a pre-request script to update each user one by one.The script shifts the first user ID from user_ids to singleUserId and re-runs the Update User request until all user IDs are processed.

API Endpoints and Details:

1. **Get Users (1-GetUsers)**
   - **Method:** GET
   - **URL:** https://gorest.co.in/public/v2/users
   - **Headers:**
     - Authorization: Bearer e4b8e1f593dc4a731a153c5ec8cc9b8bbb583ae964ce650a741113091b4e2ac6
   - **Test Script:**

- **Description:** This request fetches all users and extracts their IDs, storing them in a variable user_ids.

2. **Update User (2-UpdateUser)**

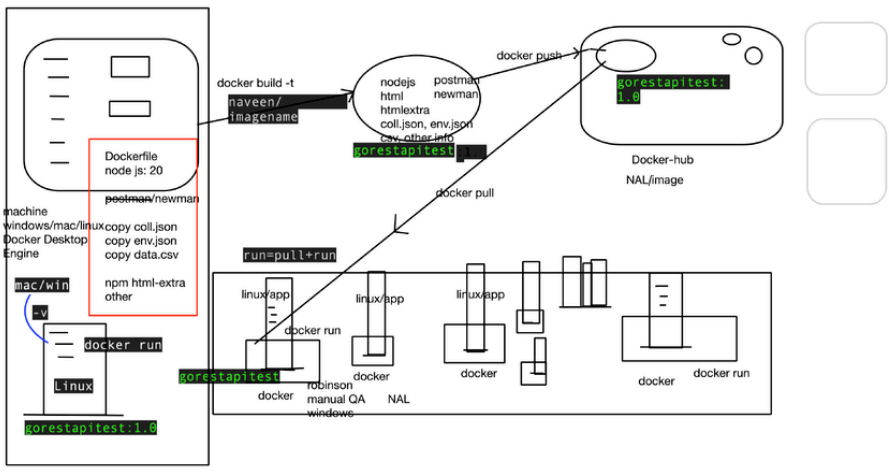
- **Method:** PUT
- **URL:** https://gorest.co.in/public/v2/users/{{singleUserId}}
- **Headers:**Authorization: Bearer e4b8e1f593dc4a731a153c5ec8cc9b8bbb583ae964ce650a741113091b4e2ac6
- **Body:** Raw JSON

- **Pre-request Script:**

- **Description:** This request updates a user with a new name and status. The pre-request script handles chaining, ensuring the next user is updated by re-running the request until all users are processed.

==========

Run Postman Collection Using Docker



==========

Dockerfile

==========

```
# Use the official Node.js 20 image as the base image
FROM node:20

# Set the working directory in the container(linux)
WORKDIR /app
```

```
# Install Newman globally
RUN npm install -g newman

#Install Newman html report
RUN npm install -g newman-reporter-html

#Install Newman Reporter HTMLExtra
RUN npm install -g newman-reporter-htmlextra


# Copy your Postman collection and environment files to the working directory
COPY GoRestAPIWorkflowColl.json .
COPY GoRestEnv.json .

# Set the command to run Newman and execute your Postman collection
CMD ["newman", "run", "GoRestAPIWorkflowColl.json", "-e", "GoRestEnv.json", "-r", "cli,json,html,htmlextra"]
```

==========

How to build docker image and push it to Docker Hub:

----Please create your docker account on Docker HUB: https://app.docker.com/signup?

Run the following commands:

1. Create image : docker build -t naveenkhunteta/mybookingapi:1.0 .
2. Push the image to hub: docker push naveenkhunteta/mybookingapi:1.0
3. Pull the image: docker pull naveenkhunteta/mybookingapi:1.0
4. Run the image and start container:

docker run -v naveenkhunteta/mybookingapi:1.0

==========

==========

How to do volume mapping to fetch files from docker linux container directory to local:

command:

//mac/linux:

docker run -v $(pwd)/newman:/app/newman newman-image

//windows:

docker run -v %cd%/newman:/app/newman naveenkhunteta/bookingapp:1.0

//windows powershell:

docker run -v ${PWD}/newman:/app/newman naveenkhunteta/bookingapp:1.0

//Git bash:

docker run -v $(pwd)/newman:/app/newman naveenkhunteta/bookingapp:1.0

========== ==========

==========

Run postman/newman using docker and postman api url:

docker run -t postman/newman run "https://api.getpostman.com/collections/243c4c87-937d-4bf2-803b-b7455b8c472c?apikey=PMAK-67ced9b809d8340001825584-2b14f04f1507c38a8e908816f8ad719366" --environment "https://api.getpostman.com/environments/012f09c1-a383-4205-bcaa-d9c2c60266a5?apikey=PMAK-67ced9b809d8340001825584-2b14f04f1507c38a8e908816f8ad719366"

==========

Script to maintain old html history in Jenkins Job win/mac:

----------for windows---------

- **Go to Jenkins Dashboard → Your Job → Configure**
- **In the "Build" section**, click **"Add build step" → "Execute Windows batch command"** (since you're on Windows)
- **Paste your batch script** directly in the command field:

```
@echo off
setlocal enabledelayedexpansion

:: Run Newman and generate report with timestamp
set TIMESTAMP=%DATE:~10,4%%DATE:~4,2%%DATE:~7,2%%TIME:~0,2%%TIME:~3,2%%TIME:~6,2%
set TIMESTAMP=%TIMESTAMP: =0%
newman run "C:\Users\YourUser\Documents\Jan2025_PostmanCollections\GoRestAPIWorkflow.json" ^
  -e "C:\Users\YourUser\Documents\Jan2025_PostmanCollections\GoRestEnv.json" ^
  --verbose -r cli,htmlextra --reporter-htmlextra-export newman\report-%TIMESTAMP%.html

:: Wait to ensure the file is fully written
timeout /t 10 /nobreak
echo hello world

:: Find the latest report
for /f "delims=" %%F in ('dir /b /o-d newman\report-*.html') do (
  set LATEST_REPORT=%%F
  goto :found
)
echo No report found!
goto :cleanup

:found
echo Latest report: %LATEST_REPORT%

:: Rename the previous "NewmanReportLatest.html" to keep history
if exist "newman\NewmanReportLatest.html" (
  set TIMESTAMP=%DATE:~10,4%%DATE:~4,2%%DATE:~7,2%%TIME:~0,2%%TIME:~3,2%%TIME:~6,2%
  set TIMESTAMP=%TIMESTAMP: =0%
  move /y newman\NewmanReportLatest.html newman\NewmanReport_%TIMESTAMP%.html
  echo Previous latest report renamed to: newman\NewmanReport_%TIMESTAMP%.html
)

:: Copy the latest report to NewmanReportLatest.html
copy /y "newman\%LATEST_REPORT%" newman\NewmanReportLatest.html
echo Latest report copied to: newman\NewmanReportLatest.html

:cleanup
:: Cleanup: Keep only the last 5 reports
for /f "skip=5 delims=" %%F in ('dir /b /o-d newman\report-*.html') do del "newman\%%F"
for /f "skip=5 delims=" %%F in ('dir /b /o-d newman\NewmanReport_*.html') do del "newman\%%F"
echo Old reports deleted, only the last 5 reports are kept.

endlocal
```

----------for mac----------

- **Go to Jenkins Dashboard → Your Job → Configure**
- **In the "Build" section**, click **"Add build step" → "Execute shell"** (since you're on Windows)
- **Paste your batch script** directly in the command field:

```
#!/bin/bash
# Run Newman and generate report with timestamp
newman run /Users/naveenautomationlabs/Documents/Jan2025_PostmanCollections/GoRestAPIWorkflow.json \
  -e /Users/naveenautomationlabs/Documents/Jan2025_PostmanCollections/GoRestEnv.json \
  --verbose -r cli,htmlextra --reporter-htmlextra-export newman/report-$(date +%Y%m%d%H%M%S).html

# Wait to ensure the file is fully written
sleep 10
echo "hello world"

# Find the latest report
LATEST_REPORT=$(ls -t newman/report-*.html | head -n 1)
```

```
echo "Latest report: $LATEST_REPORT"

if [ -f "$LATEST_REPORT" ]; then
  # Rename the previous "NewmanReportLatest.html" to keep history
  if [ -f "newman/NewmanReportLatest.html" ]; then
    TIMESTAMP=$(date +%Y%m%d%H%M%S)
    mv newman/NewmanReportLatest.html newman/NewmanReport_$TIMESTAMP.html
    echo "Previous latest report renamed to: newman/NewmanReport_$TIMESTAMP.html"
  fi

  # Copy the latest report to NewmanReportLatest.html
  cp "$LATEST_REPORT" newman/NewmanReportLatest.html
  echo "Latest report copied to: newman/NewmanReportLatest.html"
else
  echo "No report found!"
fi

# Cleanup: Keep only the last 5 reports
cd newman
ls -t report-*.html | tail -n +6 | xargs rm -f
ls -t NewmanReport_*.html | tail -n +6 | xargs rm -f
echo "Old reports deleted, only the last 5 reports are kept."
```