API Testing

API testing is the process of testing an Application Programming Interface (API) to validate its **functionality, reliability, security, and performance**.

Instead of testing the **User Interface (UI)**, API testing is performed at the **application layer** to verify how different applications or systems communicate with each other by sending requests and receiving responses.

Postman is one of the most widely used API testing tools that allows testers and developers to send API requests and validate the responses without using the application's user interface. It supports testing **REST and SOAP APIs** and helps verify **response status codes, response body, response time, authentication, headers, and error handling**.

Postman also allows requests to be organised into **collections**, automated using **JavaScript test scripts**, and integrated into **CI/CD pipelines** using Newman.

Commonly used HTTP Methods

The commonly used HTTP methods are GET, POST, PUT, and DELETE.
GET is used to retrieve data from a server.
POST is used to create or add new data to the server.
PUT is used to update or replace an existing resource on the server.
DELETE is used to remove data or resources from the server.
Each method performs a different operation depending on the business requirement.

REST & SOAP

REST (Representational State Transfer) and **SOAP (Simple Object Access Protocol)** are both used for communication between **Applications**, but they work differently.

REST is an **Architectural Style**, whereas **SOAP** is a **Protocol**.

My unnderstaning is that
REST APIs commonly exchange data in **JSON** format over **HTTP** or **HTTPS**.
SOAP APIs use **XML** messages and follow strict rules for **Security** and **Messaging**.

REST APIs are commonly tested using **Postman**, **Swagger**, and **Insomnia**, while **SOAP APIs** are commonly tested using **SoapUI** and **Postman**.

REST is commonly used for **Web Applications**, **Mobile Applications**, and **Microservices** because it is simple and fast.
SOAP is commonly used in **Enterprise Applications**, such as **Banking** and **Healthcare**, where **Security** and **Reliability** are important.

Environment

Environment: A set of **key-value variables** specific to an environment, such as **Development**, **QA**, **Staging**, or **Production**.
• Lets you switch between **environments** without manually changing the **request URLs**.
• Stores reusable values such as **base URL**, **authentication token**, and **user credentials**.
Example:
base_url = https://dev.api.com (**Development**)
base_url = https://prod.api.com (**Production**)

Collection

Collections allow multiple **API requests** to be organised into a single **reusable** group. e.g., all requests for the **Worker Screening API**

How to perform API Testing

In **API testing**, we send a **request** to the **API** using known **input data** and then analyse the **response** returned by the **server**.

○ My role during **API testing** is to verify that the **API** behaves correctly and returns the **expected results**.

○ To address this, I verify the accuracy of the ***response data,*** **HTTP status code**, ***response time,*** **error codes** returned by the **API**, ***authentication*** *and* ***authorization*** *behaviour,* **non-functional aspects** such as * **performance** and **security**.

○ Once I send a **request** to an **endpoint** in **Postman** and click the **Send** button, I receive the **response** in the **response body** together with the **HTTP status code**.

○ There are several ways to verify that the returned data is correct.
--The simplest method is checking whether the **API** returns the **expected status code**.
--Another method is writing **assertions**.
--I commonly create **assertions** to validate the **response time**, **status code**, **status code description**, **response type**, **response headers**, and specific **values** or **strings** within the **response body**.

○ **Postman** also provides many **built-in snippets** using the **Chai JavaScript library**. These snippets allow me to quickly validate **status codes**, verify that the **response** contains **expected text**, or compare the **response body** against **expected values**.

HTTP Response Status Codes

200 OK	**Request** successful.
201 Created	**Resource** created successfully.
400 Bad Request	Invalid **request** or **input data**.
401 Unauthorized	Missing, invalid, or expired **authentication token**.
403 Forbidden	**Authenticated** but **permission denied**.
404 Not Found	**Resource** or **endpoint** not found.
500 Internal Server Error	Unexpected **server** error.
503 Service Unavailable	**Server** temporarily unavailable.

Positive and Negative API testing

Positive Testing verifies that an **API** behaves correctly when **valid data** is provided.
Negative Testing verifies that the **API** handles **invalid requests** correctly without crashing or exposing **unexpected behaviour**.

My role while performing **API Testing** is to ensure both **Positive** and **Negative Scenarios** are fully covered.
To address this, I execute requests using **valid input** to confirm successful processing and expected responses. I also test **missing mandatory fields**, **invalid data types**, **incorrect parameters**, **expired authentication tokens**, **unsupported HTTP methods**, and **malformed payloads**.
For every scenario, I validate the returned **HTTP Status Code**, **Response Body**, and **Error Messages**.
As a result, **Positive** and **Negative Testing** help ensure the **API** is both **functional** and **resilient**.

Can a POST request create a resource?

Yes.
POST is primarily used to **create new resources** on the **Server**.
My role while testing **POST APIs** is to verify that the **resource** is successfully created and that the **API** returns the correct **Response**.
To address this, I validate the **Request Payload**, execute the **API**, verify the **HTTP Status Code**, confirm the **Response Body**, and check that the newly created **data** exists within the **Application** or **Database**, where applicable.
As a result, I can confirm that the **API** successfully creates new **resources** and behaves according to the **Business Requirements**.

Path paramerter & Query Parameter

Path Parameter: A value included in the **API URL path** to identify a **specific resource**.
Example: /users/123
Query Parameter: A value added after the **?** in the **URL** to **filter**, **search**, **sort**, or **control the response**.
Example: /users?status=active

Payload

A **Payload** is the actual **data** sent from the **Client** to the **Server** as part of an **API Request**.
My role while testing **APIs** is to ensure that the **Payload** contains the correct information before sending the request.
To address this, I validate the **Payload Structure**, **Field Names**, **Mandatory Fields**, **Data Types**, and **Business Values** before executing the request.
I also verify that the **API** processes the **Payload** correctly and returns the expected **Response**.
As a result, I can confirm that the **API** correctly accepts, validates, and processes the incoming **data**.

Components of HTTP request

An **HTTP Request** consists of several important components.

1. **HTTP Request Method**, such as **GET**, **POST**, **PUT**, **PATCH**, or **DELETE**, which defines the action to be performed.

2. **Uniform Resource Identifier (URI)**, which represents the **endpoint** where the API is hosted.

3. **Resources** and **Parameters**, including **Path Parameters** and **Query Parameters**, which are used to pass information to the API.

4. **Request Header**, which carries **metadata** as **key-value pairs**, such as **Content-Type**, **Authorization** tokens, **Accept**, and other request information.

5. **Request Body**, also known as the **Payload**, which contains the data being sent to the server. It is commonly used with **POST**, **PUT**, and **PATCH** requests.

--Together, these components make up a complete **HTTP Request**.

Challenges in API Testing?

Some of the common challenges involved in API testing include **API documentation, database access, and authorization**.
--**API documentation** is important because incomplete or outdated documentation can make it --difficult to understand the API behaviour and expected responses.
--**Obtaining access to the database** when backend data validation is required.
--**Authorization** can also be challenging because APIs often require authentication tokens, credentials, or different user roles before requests can be executed successfully.

Authentication and Authorization

Authentication and **Authorization** are both used to secure **APIs**, but they serve different purposes.
Authentication = Who are you? (Identity Verification)
Authorization = What can you access? (Permission Verification)

Authentication verifies the **identity** of the user or application. It answers the question, **"Who are you?"**
Common authentication methods include **Bearer Tokens**, **OAuth**, **Basic Authentication**, and **API Keys**.

Authorization determines what an **authenticated user** is allowed to access or perform. It answers the question, **"What are you allowed to do?"**
It checks the user's **roles**, **permissions**, or **access rights** before allowing access to a **resource**.

Example:
401 Unauthorized → Authentication failed (invalid or missing token).
403 Forbidden → Authentication succeeded, but the user doesn't have permission.

Authentication technique

Some of the commonly used **Authentication** techniques include
○**Session or Cookie-Based Authentication**,
○**Basic Authentication**,
○**Digest Authentication**, and
○**OAuth**.

○**Session or Cookie-Based Authentication** is commonly used in traditional **Web Applications**.

○**Basic Authentication** uses a **Username** and **Password** encoded within the **Request**.

○**Digest Authentication** is similar to **Basic Authentication** but provides additional **Security** by encrypting the **Credentials**.

○**OAuth** is widely used for modern **Web Applications** because it provides secure **Token-Based Authentication** without exposing **User Credentials**.

Variables

Variables are used to store **reusable values** that can be referenced throughout a **Postman Collection**.
• Defined using **double curly braces**: **{{variable_name}}**
• Can be used in **URLs**, **Headers**, **Request Body**, and **Scripts**.

--Variables allow values to be stored and reused throughout **API requests**.
--My role while using **variables** is to eliminate **hardcoded values** and improve **maintainability**.
--To address this, I use **Global Variables**, **Collection Variables**, **Environment Variables**, **Local Variables**, and **Data Variables** depending on the testing scenario.
--For example, instead of hardcoding the **Base URL** in every request, I store it as an **Environment Variable**. If the environment changes from **Development** to **UAT**, I simply switch the **Environment** without modifying each request individually.
--As a result, **variables** make **API requests** easier to maintain, more **reusable**, and less prone to **manual errors**.

Different types of Variables

Global Variables are accessible across **every Collection** and **Environment** within the **workspace**.

Environment Variables are available only within the currently selected **Environment**, such as **Development**, **QA**, or **UAT**.

Collection Variables are available only within a specific **Collection**, regardless of which **Environment** is selected.

Postman follows a **Variable Precedence** order when resolving variables. The priority from highest to lowest is **Local Variables** → **Data Variables** → **Environment Variables** → **Collection Variables** → **Global Variables**.

Different types of Variables (cont)

As a result, understanding **Variable Scope** helps prevent **conflicts**, improves **maintainability**, and keeps **API requests** organised and **reusable**.

Pre-request Script

Pre-request Script: A JavaScript script that runs before an API request is sent.
They are written in the Pre-request Script tab.
Commonly used for:
○ **Setting dynamic variables**, such as timestamps or generated tokens.
○ Generating **authentication tokens** before calling an API endpoint.
○ **Chaining data** from a previous request.
○ **Preparing request data** before execution.
Example: pm.environment.set("timestamp", Date.now());