10. OnlineStoreAPI - Project

So far, we have learned various Postman concepts. Now, we will put this knowledge into practice by automating end-to-end API scenarios using a sample Online Store application.

API Automation Flow: From Understanding to Documentation

1. **Understanding the Starting Point of API Testing**
    a. Get a clear idea of the application features.
    b. Know what APIs are available.
    c. Understand which functionality is handled via Frontend (UI) and which via Backend (APIs).
    d. Review user stories created by project managers.
    e. Since Developers build APIs in parallel with the UI, based on these user stories.
2. **Requirement Gathering**
    a. Coordinate with developers or BA team to collect API-related requirements.
    b. Ask for Postman Collections, Swagger documentation, or any available API references.
3. **Understanding the System**
    a. Read **technical documents** to understand
        i. API endpoints (URLs) (endpoint <–> resource)
        ii. Request/response formats (JSON, XML)
        iii. Authentication mechanisms
        iv. Database structure
        v. Tools and Technologies used
4. **Test Case Preparation**
    a. Identify all positive and negative test scenarios.
    b. Use a standard test case template (e.g., Test Case ID, Endpoint, Method, Input, Expected Output, etc).
    c. Include headers, request body, and expected status codes.
5. **Postman Setup**
    a. Create a Postman Workspace.
    b. Create a Collection for grouping related APIs. Use separate collections for different modules.
    c. Add requests with necessary headers, body, and environment variables.
    d. Write test scripts in the "Tests" tab to automate validations.

6. **Execution**
    a. Run APIs manually in Postman or using Collection Runner.
    b. Validate the response status, body content, headers, etc.
    c. Integrate with Jenkins/Newman for CI execution (if needed).
7. **Documentation**
    a. Documentation is an Inbuilt feature provided in postman itself. So no need to use ms-word, google docs, etc
        i. Generate API documentation.
        ii. Publish and share the documentation with the team.

Online Store API -Technical Design

➔ **E-commerce Application Overview**

- ◆ An e-commerce app typically consists of two main parts
    - **Frontend Application**
        - The user-facing part (website or app) where customers
            - ◆ Login
            - ◆ Browse or search products
            - ◆ Add items to cart
            - ◆ Checkout and make payments
    - **Backend Application**
        - Handles
            - ◆ Business logic
            - ◆ Data management
            - ◆ API responses

➔ **Most Common Functionalities of e-commerce applications**

- ◆ **Product Management** – Browse various products.
- ◆ **Cart Functionality** – Add or remove products from the cart.
- ◆ **User Management** – Login, registration, and profile handling.
- ◆ **Payment Flow** – Checkout and payment processing.

➔ Refer **Online Store API -Technical Design.pdf** for high level information

➔ **Project Scope**

- ◆ Although our Technical Document includes multiple modules such as **Products, Cart,**

and User, in our API Automation project, we will focus solely on the Products module.

- ◆ This module covers key real-world use cases and serves as an ideal foundation for learning and hands-on practice.

Note

- ➔ The APIs are publicly available, we can start testing them in Postman.
- ➔ The Base URL is usually created as a global or environment variable because it remains the same for all requests. It can also change between environments (e.g., development, testing, production), so defining it as a variable makes it easy to update in one place.
- ➔ On the other hand, endpoints (like /login or /products) are defined during development and usually do not change.
- ➔ An endpoint (or resource path) is a specific URL that represents a resource on the server. It defines the address where a client can access or manipulate data using HTTP methods like GET, POST, PUT, PATCH, and DELETE.
  - ◆ For the endpoint path /users/123, the endpoint path is /users/123, and the resource is the user with ID 123.
- ➔ A **Data Model** in API Testing defines the structure, relationships, and format of the data that the API accepts and returns. Purpose of Data Model is **Ensures data consistency, Simplifies data validation, Helps define API request and response formats**.
- ➔ Testers need a Data Model to
  - ◆ **Check API Responses:** It helps testers verify if the data in the response matches the expected structure.
  - ◆ **Ensure Correct Data:** It ensures the data types and fields are correct in the response.
- ➔ Swagger is an interactive document, not a static one. It is created during or after API development. it allows developers and testers to view, test, and interact with API endpoints directly from the documentation.

Test Case Preparation or Test Case Design

- ➔ We create test cases based on the high-level technical document, but it doesn't provide all the details. During test case writing, we have meetings with developers and product managers to clarify questions and add more information to the test cases as part of the process.
- ➔ Once the test cases are created, we review them with the Developer and Product Owner (PO) or Product Manager (PM). After the review, we can proceed with execution once the application is deployed on the server.
- ➔ Finally, we upload the test cases to a Test Case Management Tool for tracking and reporting.
- ➔ Popular Test Case Management Tools
  - ◆ TestRail
  - ◆ JIRA (with Zephyr plugin)

- Quality Center (ALM)
- Xray (for JIRA)
- qTest
- TestLink
- Azure DevOps (with Test Plans)

➔ Refer **Product API TestCases, Cart TestCases, User Test Cases sheets** in **StoreAPI_Testcases.xlsx**

➔ When creating test cases we need to categorize the required elements like query parameters, global variables (e.g., baseUrl, authToken), and collection variables (e.g., productId) to organize and manage the tests.

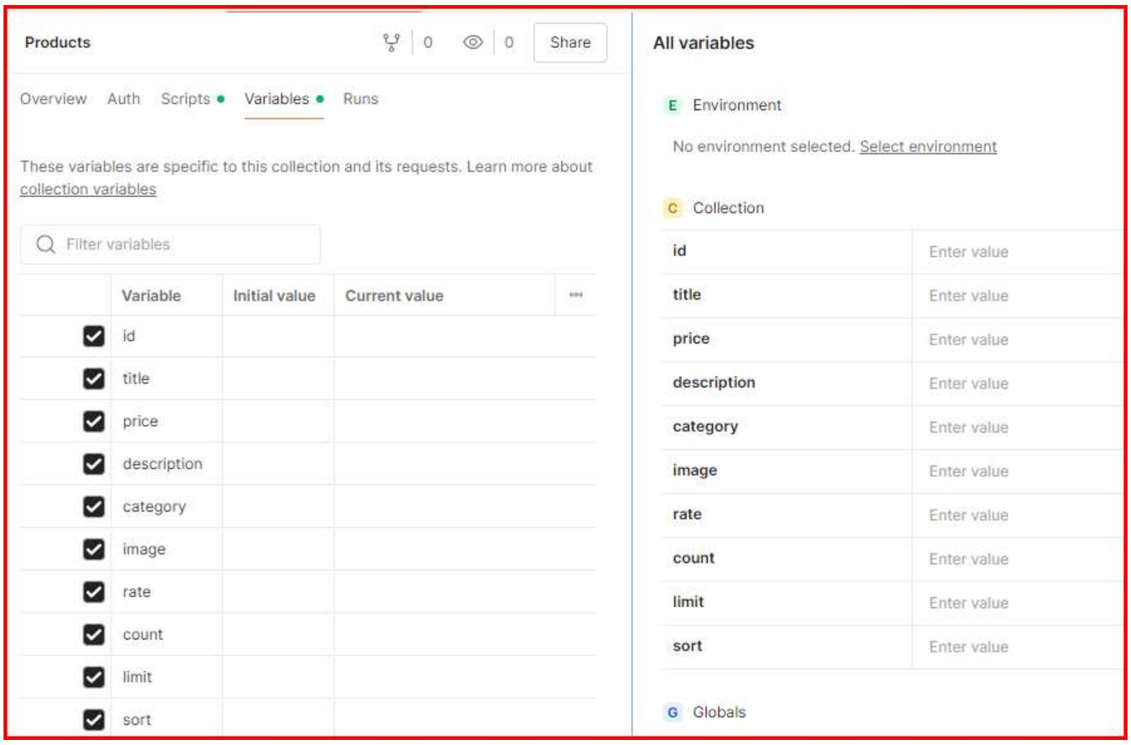
➔ Refer **Postman Variables Required** sheet in **StoreAPI_Testcases.xlsx**

Workout for Product API TestCases in Postman

➔ Create a Blank workspace → **OnlineStoreAPI**

➔ Import a ready-made **Products.postman_collection** that already contains API requests, validations, and test scripts for quick setup and testing. In our case, you will create the collection manually.

- Observe Collection Level Variables



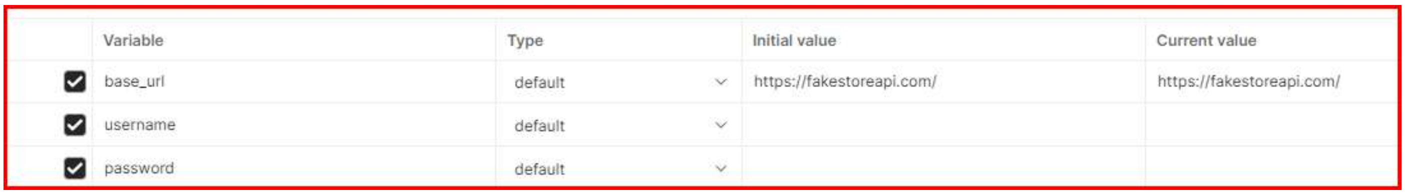
These variables are specific to this collection and its requests. Learn more about collection variables

	Variable	Initial value	Current value
✓	id		
✓	title		
✓	price		
✓	description		
✓	category		
✓	image		
✓	rate		
✓	count		
✓	limit		
✓	sort		

No environment selected. Select environment

id	Enter value
title	Enter value
price	Enter value
description	Enter value
category	Enter value
image	Enter value
rate	Enter value
count	Enter value
limit	Enter value
sort	Enter value

- Create Global variables as in the **Postman Variables Required** sheet



	Variable	Type	Initial value	Current value
✓	base_url	default	https://fakestoreapi.com/	https://fakestoreapi.com/
✓	username	default		
✓	password	default		

➔ Observe Validations on Post Response Script Tab. Response contains all Items so we need to do below validations because both schemas will be different.

  ◆ Entire response schema (for all items)

  ◆ Every item schema (for one object) in the response.

➔ We should not hardcode any data in the request. we always take the data from the variables. Most of the times variable values are taken from external files.

➔ We use 2 external. files while executing a collection

  ◆ A JSON file with one set of common data used across all requests.

    ● **product_data.json**

  ◆ A CSV file with multiple sets of data (15–20 rows) for data-driven testing of each request.

    ● **products_data.csv**

➔ If any authentication is required we will create at collection level in authorization tab since all requests uses same type of authentication.

➔ **Run collection** with single set of data(json file) and multiple sets of data(csv file) in postman manually.

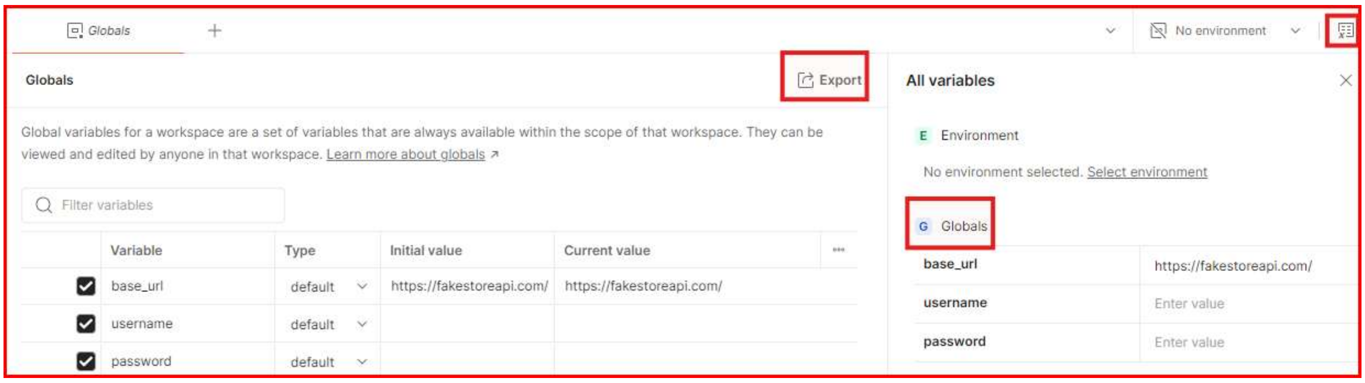
➔ After Running the collection we can share the results by clicking Export Results.

➔ When we run a collection inside Postman, global variables are used automatically.

➔ But when we export the collection to run outside Postman like in Command Prompt, Jenkins, or GitHub, only the collection along with collection variables is exported not the global variables.

➔ So, if our collection depends on global variables we must also export the global variables separately as an environment or global variable JSON file.

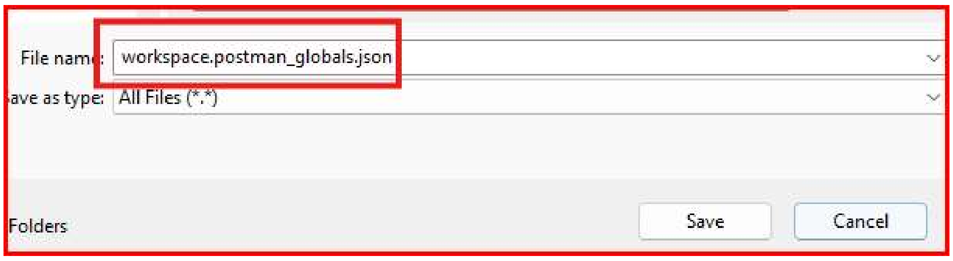
  ◆ Variables → Globals → Export



Global variables for a workspace are a set of variables that are always available within the scope of that workspace. They can be viewed and edited by anyone in that workspace. Learn more about globals ↗

	Variable	Type	Initial value	Current value
✓	base_url	default	https://fakestoreapi.com/	https://fakestoreapi.com/
✓	username	default		
✓	password	default		

base_url	https://fakestoreapi.com/
username	Enter value
password	Enter value



File name: workspace.postman_globals.json
Save as type: All Files (*.*)

➔ Without global variables, the collection may fail to run correctly outside Postman.

Note

➔ Postman does not provide a built-in detailed report.To generate a proper report automatically, you need to run the collection using **Newman (Postman's command-line tool).**

➔ When doing data-driven testing in Postman with 4 IDs in a CSV file, where one ID returns a 204 No Content status and others return 200 OK with data, we can add dynamic validation in **Post-response Scripts tab** using JavaScript.

validation.js

```
// Extract the current ID from the CSV file

const currentId = pm.iterationData.get("id");

// Check the status code

if (pm.response.code === 204) {

    // Expect 204 for specific ID (e.g., 101)

    pm.test(`No data found for ID: ${currentId}`, function () {

        pm.expect(currentId).to.eql("101");

        pm.expect(pm.response.text()).to.eql("");

    });

} else if (pm.response.code === 200) {

    // Expect 200 for all other IDs

    pm.test(`Data found for ID: ${currentId}`, function () {

        pm.expect(currentId).not.to.eql("101");

        const responseData = pm.response.json();

        pm.expect(responseData).to.be.an("object");

        pm.expect(responseData).to.have.property("id", currentId);

    });

} else {

    // Fail if status code is neither 200 nor 204

    pm.test(`Unexpected status code for ID: ${currentId}`, function () {

        pm.expect(pm.response.code).to.be.oneOf([200, 204]);

    });

}
```