API TESTING- Please refer the notes in Phone too.

API (Application Programming Interface) testing is a crucial part of software testing that focuses on validating the functionality, reliability, performance, and security of APIs. Even for manual testers, understanding API testing is becoming increasingly important as applications become more interconnected.

Here's a breakdown of common API testing interview questions for manual testers, along with comprehensive answers:

**Core Concepts**

**1. What is an API?**

* **Answer:** An API (Application Programming Interface) is a set of rules and protocols that allows different software applications to communicate and interact with each other. It acts as an intermediary, defining how one piece of software can request services from another. For example, when you use a weather app, it uses an API to get weather data from a weather service.

**2. Why is API testing important?**

* **Answer:** API testing is important because:
  + **Early Bug Detection:** It allows testing the core business logic and functionality at the integration layer, before the UI is even built, leading to earlier bug detection and cheaper fixes.
  + **Improved Test Coverage:** It provides better test coverage for the backend systems, which might not be fully exposed through the UI.
  + **Faster Feedback:** API tests are generally faster to execute than UI tests, providing quicker feedback to developers.
  + **Independent of UI Changes:** If the UI changes, API tests are often unaffected, making them more stable and maintainable.
  + **Performance and Security:** It allows for testing performance (load, stress) and security vulnerabilities directly at the API level.

**3. What is the difference between API testing and UI testing?**

* **Answer:**
  + **API Testing:** Focuses on the business logic layer of an application, testing the communication between different software components. It validates if the API functions correctly, handles data as expected, and returns the correct responses. It's "headless" and doesn't involve a graphical interface.
  + **UI Testing:** Focuses on the presentation layer, verifying that the user interface works as expected, looks correct, and interacts properly with the user. It simulates user actions on the actual application screen.

**4. What are the common types of APIs you might encounter?**

* **Answer:** The most common types are:
  + **REST (Representational State Transfer) APIs:** These are widely used, lightweight, and typically use HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources. They are stateless.
  + **SOAP (Simple Object Access Protocol) APIs:** Older, XML-based, and more structured/protocol-driven. They are typically used in enterprise environments.
  + **GraphQL APIs:** A newer query language for APIs that allows clients to request exactly the data they need, avoiding over-fetching or under-fetching.

**Manual API Testing Process**

**5. How do you perform API testing manually?**

* **Answer:** Manual API testing involves using tools to send requests to API endpoints and analyzing the responses. The general steps are:
  1. **Understand API Documentation:** Review the API documentation (Swagger/OpenAPI spec, Postman collection, etc.) to understand endpoints, request methods, parameters (path, query, header, body), and expected responses.
  2. **Choose an API Client Tool:** Select a tool like Postman, Insomnia, or a browser's developer tools.
  3. **Construct API Requests:**
     + **Endpoint URL:** Enter the correct URL.
     + **HTTP Method:** Select the appropriate method (GET, POST, PUT, DELETE).
     + **Headers:** Add necessary headers (e.g., Content-Type, Authorization tokens).
     + **Parameters:** Provide query parameters, path parameters, or request body (JSON, XML).
  4. **Send Request:** Execute the request.
  5. **Analyze Response:**
     + **Status Code:** Verify the HTTP status code (e.g., 200 OK, 201 Created, 400 Bad Request, 500 Internal Server Error).
     + **Response Body:** Check the content and structure of the response data (JSON, XML). Validate data types, values, and completeness.
     + **Headers:** Examine response headers for expected values.
     + **Response Time:** Observe the time taken for the API to respond.
  6. **Validate Against Requirements:** Compare the actual response with the expected behavior defined in the requirements or API documentation.

**6. What are the key components of an HTTP request that you would focus on during API testing?**

* **Answer:** The key components of an HTTP request are:
  + **HTTP Method (Verb):** Defines the action to be performed (e.g., GET for retrieving, POST for creating, PUT for updating, DELETE for removing).
  + **URI/URL (Endpoint):** The Uniform Resource Identifier/Locator that specifies the resource to be accessed.
  + **Headers:** Key-value pairs that carry metadata about the request (e.g., Content-Type, Authorization, Accept).
  + **Request Body (Payload):** The actual data being sent with the request (most commonly used with POST, PUT).
* NOTE: GET request doesn’t have a pay load/request body.

**7. What are the key components of an HTTP response that you would verify?**

* **Answer:** The key components of an HTTP response are:
  + **Status Code:** A three-digit number indicating the outcome of the request (e.g., 200 OK, 404 Not Found).
  + **Response Body (Payload):** The data returned by the API in response to the request.
  + **Headers:** Key-value pairs providing metadata about the response (e.g., Content-Type, Date).

**8. Can you explain common HTTP status codes and what they indicate?**

* **Answer:**
  + **2xx (Success):**
    - 200 OK: The request was successful.
    - 201 Created: The request has been fulfilled and resulted in a new resource being created.
    - 204 No Content: The server successfully processed the request, but is not returning any content.
  + **3xx (Redirection):**
    - 301 Moved Permanently: The requested resource has been permanently moved to a new URL.
  + **4xx (Client Error):**
    - 400 Bad Request: The server cannot process the request due to client error (e.g., malformed syntax, invalid request message framing).
    - 401 Unauthorized: The request requires user authentication.
    - 403 Forbidden: The server understood the request but refuses to authorize it.
    - 404 Not Found: The requested resource could not be found on the server.
    - 405 Method Not Allowed: The HTTP method used is not supported for the requested resource.
    - 429 Too Many Requests: The user has sent too many requests in a given amount of time ("rate limiting").
  + **5xx (Server Error):**
    - 500 Internal Server Error: A generic error message, given when an unexpected condition was encountered and no more specific message is suitable.
    - 503 Service Unavailable: The server is currently unable to handle the request due to a temporary overload or scheduled maintenance.

**9. What are some common authentication methods you might encounter in APIs? How would you handle them in manual testing?**

* **Answer:**
  + **API Keys:** Often sent as a header (e.g., X-API-Key) or query parameter. You would typically paste the provided API key into the tool's header or query parameter section.
  + **Basic Authentication:** Sends username and password encoded in the Authorization header. Most API tools have a dedicated "Basic Auth" section where you enter credentials, and the tool handles the encoding.
  + **Bearer Tokens (OAuth 2.0, JWT):** A common method where a token (usually a JWT - JSON Web Token) is obtained after login and sent in the Authorization header as Bearer [token]. You would typically make an initial login request to get the token, then copy and paste it into the Authorization header for subsequent requests.

**10. How do you handle dynamic data or dependencies in API testing (e.g., an ID generated by one API call needed for another)?**

* **Answer:** This is a common scenario. For manual testing, you would:
  1. **Execute the first API call:** Send the request and get the response.
  2. **Manually extract the dynamic value:** Locate the generated ID or other dynamic data from the response body.
  3. **Copy and paste:** Copy this extracted value.
  4. **Paste into the subsequent API call:** Use the copied value as a parameter or part of the request body for the next API call.
  5. (In automated testing, this process would be scripted, but for manual, it's a manual copy-paste operation.)

**Test Case Design for APIs (Manual)**

**11. What are the key aspects you would cover when designing manual API test cases?**

* **Answer:**
  + **Functionality:** Does the API do what it's supposed to do? (e.g., GET /users returns a list of users).
  + **Input Validation:**
    - **Positive Scenarios:** Valid inputs (data types, formats, ranges).
    - **Negative Scenarios:** Invalid inputs (missing parameters, incorrect data types, out-of-range values, malformed requests).
  + **Error Handling:** Does the API return appropriate error codes and messages for invalid requests or internal server issues?
  + **Data Integrity:** Is the data created, updated, or deleted correctly and consistently?
  + **Authorization & Authentication:** Are only authorized users able to access specific resources? Are authentication mechanisms working?
  + **Performance (Basic Checks):** Observe response times.
  + **Edge Cases/Boundary Value Analysis:** Test minimum/maximum values, empty strings, nulls, very long strings.
  + **Schema Validation:** Does the response body adhere to the expected JSON or XML schema?
  + **Chaining/Workflow:** Test sequences of API calls that represent a complete business flow (e.g., create user -> get user -> update user -> delete user).

**12. Can you give an example of a positive and a negative test case for a POST /users API endpoint that creates a new user?**

* **Answer:**
  + **Positive Test Case:**
    - **Description:** Verify that a new user can be successfully created with valid user data.
    - **Endpoint:** POST /users
    - **Request Body (JSON):** {"name": "John Doe", "email": "john.doe@example.com", "password": "securepassword"}
    - **Expected Status Code:** 201 Created
    - **Expected Response Body:** Contains the newly created user's details, possibly with a generated ID.
    - **Post-condition:** Verify the user exists by making a GET /users/{id} request.
  + **Negative Test Case (Missing Required Field):**
    - **Description:** Verify that user creation fails when a required field (e.g., "email") is missing.
    - **Endpoint:** POST /users
    - **Request Body (JSON):** {"name": "Jane Doe", "password": "anotherpassword"}
    - **Expected Status Code:** 400 Bad Request
    - **Expected Response Body:** Contains an error message indicating the "email" field is missing or invalid.
  + **Negative Test Case (Invalid Email Format):**
    - **Description:** Verify that user creation fails with an invalid email format.
    - **Endpoint:** POST /users
    - **Request Body (JSON):** {"name": "Peter Pan", "email": "invalid-email", "password": "weakpassword"}
    - **Expected Status Code:** 400 Bad Request
    - **Expected Response Body:** Contains an error message indicating invalid email format.

**Tools and Best Practices**

**13. What tools would you use for manual API testing?**

* **Answer:**
  + **Postman:** Very popular and versatile. Offers a user-friendly GUI to send requests, organize collections, manage environments, and view responses.
  + **Insomnia:** Another excellent GUI-based API client with similar functionalities to Postman.
  + **Curl:** A command-line tool for transferring data with URLs. Useful for quick checks and understanding the raw HTTP request/response.
  + **Browser Developer Tools:** Browsers like Chrome and Firefox have built-in developer tools (Network tab) that allow you to inspect requests and responses made by web applications, which can be helpful for understanding API calls.
  + **Swagger UI / OpenAPI Generator:** Often used for API documentation, it can also act as a basic interface to test API endpoints directly from the documentation.

**14. What are some challenges you might face as a manual tester doing API testing?**

* **Answer:**
  + **Complexity of Request Bodies:** Crafting complex JSON/XML payloads can be time-consuming and error-prone.
  + **Managing Test Data:** For scenarios requiring a lot of varied input data, managing it manually can be difficult.
  + **Handling Dependencies/Sequencing:** Manually extracting values from one response and pasting them into the next request for chained API calls can be tedious.
  + **Lack of Automation:** Repetitive tests are inefficient to perform manually.
  + **Debugging:** Without automated assertions, verifying every aspect of a large response body can be challenging.
  + **Authentication/Authorization Complexity:** Dealing with complex authentication flows (e.g., OAuth tokens that expire quickly) can be tricky.
  + **Poor Documentation:** If API documentation is incomplete or outdated, it's hard to know what to test.

**15. What are your thoughts on API automation vs. manual API testing?**

* **Answer:** While manual API testing is valuable for initial exploration, understanding, and ad-hoc testing, automation is crucial for efficiency and regression testing.
  + **Manual Testing:** Good for exploratory testing, understanding API behavior, quick validation, and testing new/unstable APIs.
  + **Automated Testing:** Essential for repetitive regression testing, comprehensive test coverage, faster feedback in CI/CD pipelines, and handling complex test data/scenarios.
  + **Conclusion:** A hybrid approach is often best, using manual testing for initial checks and automating stable, critical test cases.

**16. How do you ensure the reliability and consistency of your manual API tests?**

* **Answer:**
  + **Clear Test Cases:** Document test cases with clear steps, expected inputs, and expected outputs.
  + **Organized Collections:** Use features in tools like Postman to organize requests into collections, folders, and environments.
  + **Environment Variables:** Use environment variables in tools to switch between different environments (dev, UAT, prod) without changing request details.
  + **Pre-request Scripts (in Postman/Insomnia):** For more complex scenarios, you can use basic scripting to set up data or authentication tokens, even as a manual tester.
  + **Regular Review:** Periodically review and update test cases as the API evolves.

By understanding these questions and providing thoughtful, practical answers, manual testers can demonstrate their proficiency in API testing and their value to a QA team.

API Student programs:

Multiple API types may be used to achieve a task. For example, uploading a photo to Instagram makes use of various APIs:

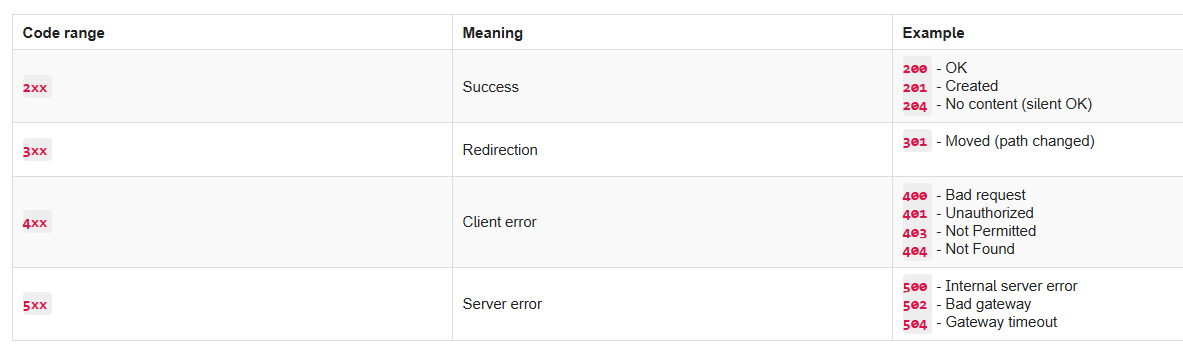
1. Hardware API for the app to talk to your camera
2. Software library API for the image to be processed with filters
3. Web API for sending your image to Instagram's servers so your friends can like it!

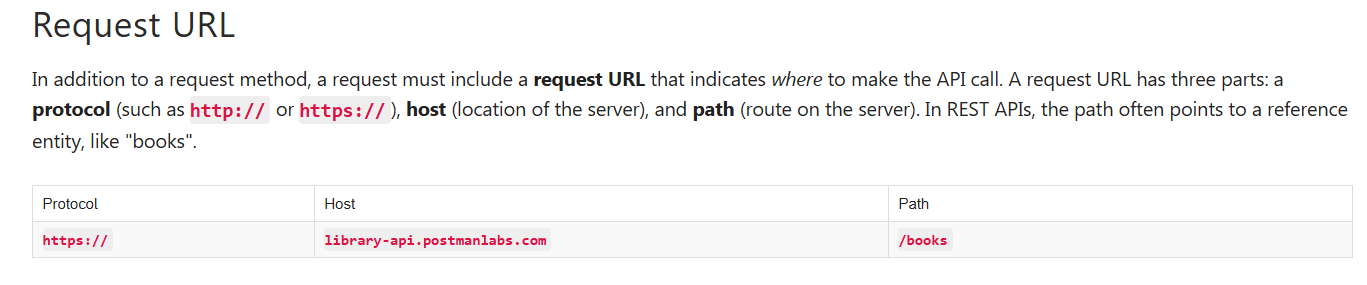
**REST APIs**  
Some traits of REST APIs include not storing session state between requests, the ability to cache, and the ability to send and receive various data types. Still confused? Don't worry; we will learn hands-on very soon in this course!

Access

APIs also vary in the scope of who can access them.

* **Public APIs (aka Open APIs)**Consumed by anyone who discovers the API
* **Private APIs**Consumed only within an organization and not made public
* **Partner APIs**Consumed between one or more organizations that have an established relationship





Variables in Postman:

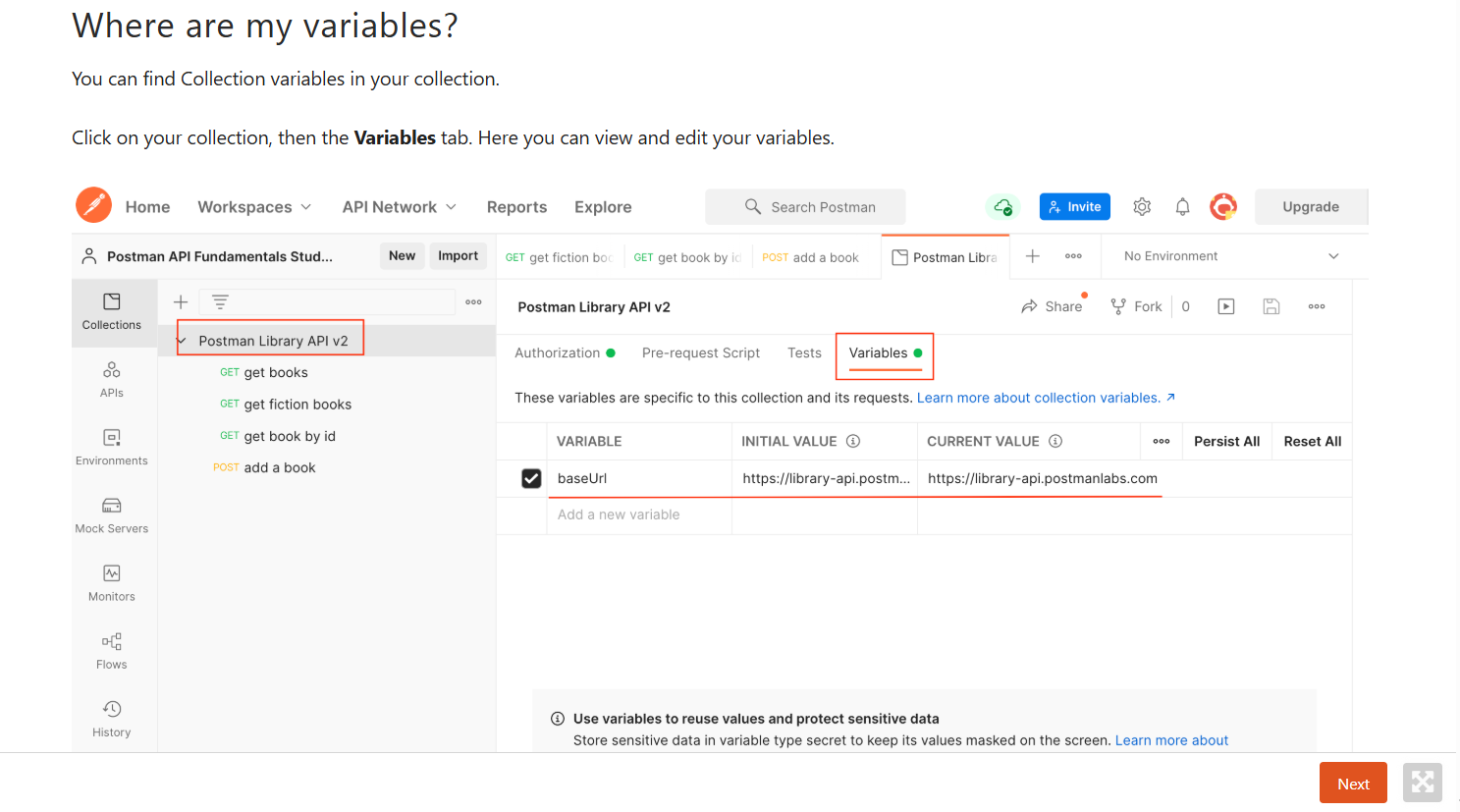
Postman allows you to save values as [variables](https://learning.postman.com/docs/sending-requests/variables/) to reuse them and easily hide sensitive information like API Keys.

We will use a variable to replace our base URL so that we don't have to type that repeatedly. Once a variable is defined, you can access its value using double curly brace syntax like this: **{{variableName}}**

Paths and complete URLs are also sometimes called **API endpoints**.

Real time Workshop:

1. Create a Workspace.
2. Create a collection.
3. Get books from Library api.
4. Variables- **Set the "baseUrl" variable – Remember variable names are CASESENSITIVE- should be as is- baseUrl. Also, we only need to select the url till.com only don’t include/and the path parameter after that.**



Note that there are two columns:  
  
**Initial Value** - the value initially set when someone forks or imports your collection. Note that if you share your collection with others, they will see this value, so don't put any secrets here!

**Current Value** - Postman always resolves the variable to this value. This is local to your Postman account, and not public. It is good to keep secrets like API Keys ONLY in this column and not include them in the Initial Value column.

1. Query parameters:

Query parameters

Remember that the minimum ingredients you need to make a request are:

* a request method (**GET**/**POST**/**PUT**/**PATCH**/**DELETE**, etc)
* a request URL

Some APIs allow you to refine your request further with key-value pairs called **query parameters**.

Query parameter syntax

Query parameters are added to the end of the path. They start with a question mark **?**, followed by the key-value pairs in the format: **<key>=<value>**. For example, this request might fetch all photos that have landscape orientation:

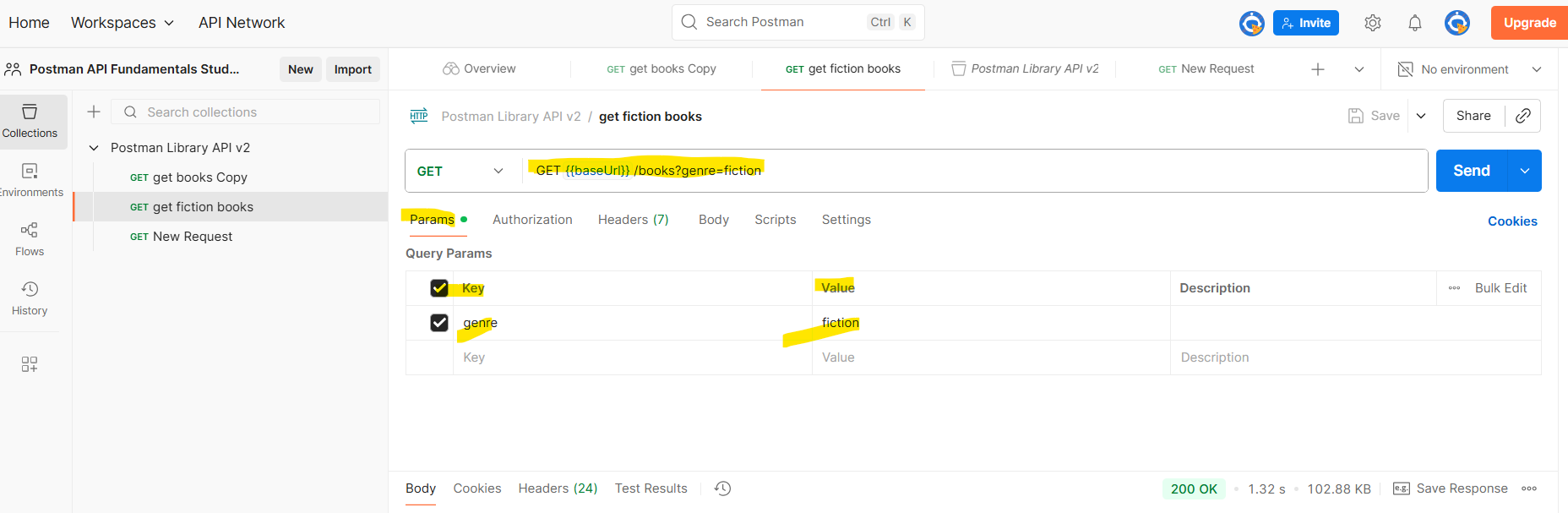
**GET https://some-api.com/photos?orientation=landscape**

If there are multiple query parameters, each is separated by an ampersand **&**. Below two query parameters to specify the orientation and size of the photos to be returned:

**GET** [**https://some-api.com/photos?orientation=landscape&size=500x400**](https://some-api.com/photos?orientation=landscape&size=500x400)

**Basically, Query parameters are used as filters when we want filtered responses.**

**5, To add query parameters to our get request in postman, we need to go to params tab and add filter there- i.e. the key value pair there(this will be fetched from the api documentation). Notice that our get request url is autoupdated with the key value pair provided. Sve and hit send . Filtered results will be populated.**



1. How to add multiple Query parameters? – Same way in Params section go and add another key value pair- Save and Send request.

## **Path Variable**

Another way of passing request data to an API is via **path variables** (a.k.a. "path parameters"). A path variable is a dynamic section of a path and is often used for IDs and entity names such as usernames.

### **Path Variable syntax**

The path variable comes immediately after a slash in the path. For example, the [GitHub API](https://docs.github.com/en/rest/reference/users#get-a-user) allows you to search for GitHub users by providing a username in the path in place of **{username}** below:

**GET https://api.github.com/users/{username}**

Making this API call with a value for **{username}** will fetch data about that user:

**GET https://api.github.com/users/postmanlabs**

You can have multiple path variables in a single request, such as this endpoint for getting a user's GitHub code repository:

**GET https://api.github.com/repos/{owner}/{repoName}**

For example, to get information about the **newman** code repository from **postmanlabs**:  
  
**GET https://api.github.com/repos/postmanlabs/newman**

|  |  |
| --- | --- |
| **Path Variable** | **Query parameters** |
| ex: **/books/abc123** | ex: **/books?search=borges&checkedOut=false** |
| Located **directly after a slash** in the path. It can**be anywhere on the path** | Located only at the **end of a path**, right after a question mark **?** |
| Accepts **dynamic values** | Accepts **defined query keys** **with potentially dynamic values**. |
| \* Often used for IDs or entity names | \* Often used for options and filters |

### **Path vs. query parameters**

At first, it is easy to confuse these two parameter types. Let's compare them side by side.

\* These are just conventions! Some APIs might ask you to pass an ID or username in a query parameter like this: **/users?username=getpostman**

### **When to use path variable?**

**Always read the API documentation**! If a path parameter is required, the documentation will mention this.

Note that some API documentation uses **colon syntax** to represent a wildcard in the path like **/users/:username**, while some use curly braces like **/users/{username}**. They both mean the same thing: that part of the path is dynamic!

Path parameter- placeholder in the url to accept a value.- usually defined within curly braces or : as shown above.

Path variable- the actual value provided in the url.

Path variable-

1. Get a book by id?

In the Request Url go and add :id after books/ as below. GET{{baseUrl}}/books/:id

Once this is done in the params tab, automatically the path parameter tab will be opened as follows. Go and give the id of the book we want to search there as value. Save and send.



## **Task: Add an authorization header**

Some APIs require [**Authorization**](https://learning.postman.com/docs/sending-requests/authorization/authorization/) (aka **Auth**) for certain endpoints in order to permit a request.

### **Authorization**

Think about why you might not want an API to have completely open endpoints that anyone can access publicly. It would allow unauthorized people to access data they shouldn't see, or allow bots to flood an API with thousands of calls per second and shut it down.

There are multiple methods for authorizing a request. Some examples are **Basic Auth** (username and password), **OAuth** (delegated authorization), and **API Keys** (secret strings registered to a developer from an API portal).

### **Getting an API Key**

APIs that use API Key auth usually allow developers to sign up in a developer portal, where they will receive a random API Key that can be used to authorize their requests to the API. The API Key allows the API to track who is making calls and how often.

The Postman Library API v2 uses very light protection and does not require you to register for an API Key. You simply have to know it:

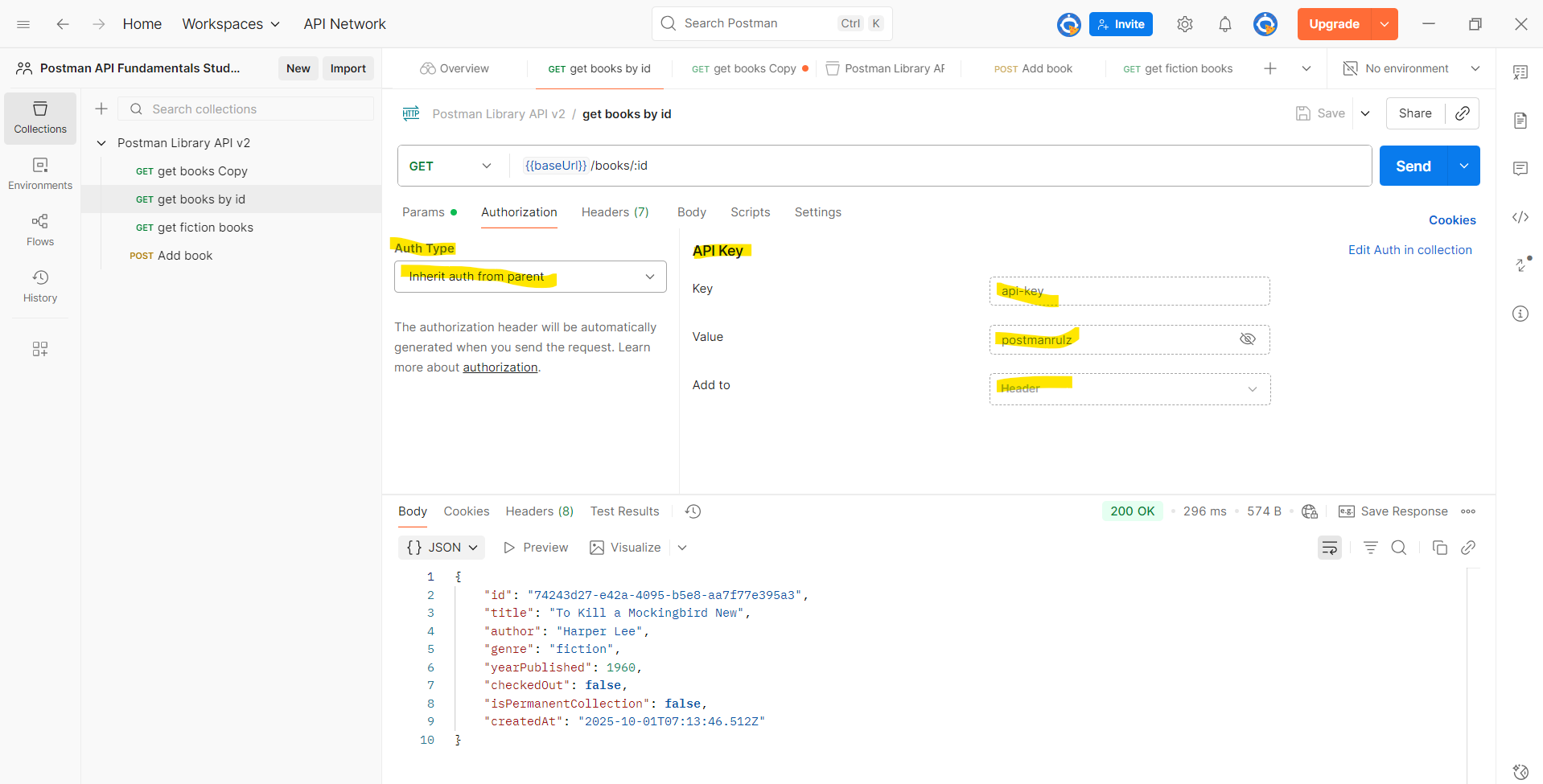
Header name: **api-key**  
Header value: **postmanrulz**  
  
As the [documentation](https://documenter.getpostman.com/view/15567703/UVyxRtng#e0dbc14c-371d-4813-97c3-281de36fad0c) shows, the Postman Library API v2 requires adding this **header** to any requests for adding, updating and deleting books, since these operations change data in the database instead of simply reading them.

For Authorization through api key, we need to add the api key header name and value in Header section.

1. POST- create a new entry. Request always need an authorization and a body.
2. Postman Auth- Instead of the auth method mentioned in step 8, we can also use Postman Auth.

### **Add Auth to the Collection**

The Postman Auth helper can help you add authorization at the request, folder or collection level. Let's add the api-key to our entire collection so that all requests will send the key.



NOTE: Step 10 is the most commonly used and better way to add Auth than Step 8.

NOTE: Always use Console window at the bottom left corner to see the logs/errors and debug.

1. Variables in Postman (Continued)

Previously in the "Request Parameters" section of this course, we saw how using a variable saved us time and helped reduce redundant copy-paste of the request URL using the double curly brace syntax like this: **{{variableName}}**.

Remember, Postman allows you to save values as [variables](https://learning.postman.com/docs/sending-requests/variables/) so that you can:

* 1. Reuse values to keep your work [DRY](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself#:~:text=%22Don't%20repeat%20yourself%22,data%20normalization%20to%20avoid%20redundancy.) (Don’t Repeat Yourself)
  2. Hide sensitive values like API keys from being shared publicly

### **Variable scopes**

You can set variables that live at various [scopes](https://learning.postman.com/docs/sending-requests/variables/#variable-scopes). Postman will resolve to the value at the nearest and narrowest scope.  
  
From broadest to narrowest, these scopes are **global**, **collection**, **environment**, **data**, and **local**.

If a variable with the same name is declared in two different scopes, the value stored in the variable with narrowest scope will be used. For example, if there is a global variable named **username** and a local variable named **username**, the local value will be used when the request runs.

We will work with **collection variables** today, which live at the collection level and can be accessed anywhere inside the collection.

1. Setting variables programmatically

Scripting in Postman:

Postman allows you to add automation and dynamic behaviors to your collections with [scripting](https://learning.postman.com/docs/writing-scripts/intro-to-scripts/).  
Postman will automatically execute any provided scripts during two events in the request flow:

Immediately before a request is sent: [pre-request script](https://learning.postman.com/docs/writing-scripts/pre-request-scripts/) (**Pre-request Script** of Scripts tab).

Immediately after a response comes back:[post-response script](https://learning.postman.com/docs/writing-scripts/test-scripts/) (**Post-response** of Scripts tab).

In this lesson, we will focus on writing scripts in the **Post-response** tab, which are executed when a response comes back from an API.

The **pm** object

Postman has a [helper object named **pm**](https://learning.postman.com/docs/writing-scripts/script-references/postman-sandbox-api-reference/#the-pm-object) that gives you access to data about your Postman environment, requests, responses, variables and testing utilities.

For example, you can access the JSON response body from an API with:   
  
**pm.response.json()**

You can also programmatically get collection variables like the value of **baseUrl** with:

**pm.collectionVariables.get(“baseUrl”)**

In addition to getting variables, you can also set them with **pm.collectionVariables.set("variableName", "variableValue")** like this:  
  
**pm.collectionVariables.set(“myVar”, “foo”)**  
Get ready!In the next task, we will use scripting and the **pm** object to set a new book's automatically **id** as a collection variable so we can use the **id** in other requests.

1. Task: Your first script:

If you are new to JavaScript - have no fear! Here are some basics.

Logging data

In JavaScript, you can print data for a value to the console using this syntax:

**console.log("Hello world!")**

**// => Hello world!**

Comments

In JavaScript you can add comments to your code. Comments are skipped by the interpreter, so you can use them to explain things in your code if you like.

**// Single line comments start with two slashes. I am not code!**

**/\* You can write multi-line comments by**

**opening and closing with slash and asterisk.**

**I am not code!**

**\*/**

Add a script to your request

* 1. In your "**add a book**" request, change the book data in your **Body** to a new book you like.
  2. From the **Scripts**tab of your request, open the **Post-res** tab (short for Post-response)
  3. Inside the Script editor, **add this JavaScript code** to log the JSON response from the API:

**console.log(pm.response.json())**

* + 1. **Save** your request
    2. **Send** your request. This will trigger the script in the Post-response script tab to run after the response comes back from the API
    3. **Open**the**Postman Console** in the lower left of the window.
    4. Scroll to the bottom of the logs in the console. You will see your most recent request **POST https://library-api.postmanlabs.com/books**

The response data from the API is logged in the console because of the code in our **Scripts** tab! You can **expand the data** by clicking on the small arrow to the left.



1. Write a script to automatically grab the newly created book id from the post Response into the get request and fetch details of the newly added book.

Saving a value as a variable allows you to use it in other requests. Using a **Post-response** script, let's grab the **id** of a newly added book and save it so we can use it in future requests.

Setting and getting collection variables

The **pm** object allows you to set and get collection variables.

To **set** a collection variable, use the **.set()** method with two parameters: the variable name and the variable value

**pm.collectionVariables.set("variableName", value)**

To **get** a collection variable use the **.get()** method and specify the name of the variable you want to retrieve:

**pm.collectionVariables.get("variableName")**

Local variables

We can also store local variables inside our scripts using JavaScript.

There are two ways to define a variable in JavaScript: using the **const** or **let** keywords. **const** is for variables that won't change value, whereas **let** allows you to reassign the value later.

Set the new book **id** as a variable:

1. In the **Body** tab of the "**add a book**" request, **change the book's details to add a new book**!
2. In the **Post-response** tab in **Scripts** of the "add a book" request, replace the console.log() statement with this code:

**// save the "id" value from the response to a variable named "id"**

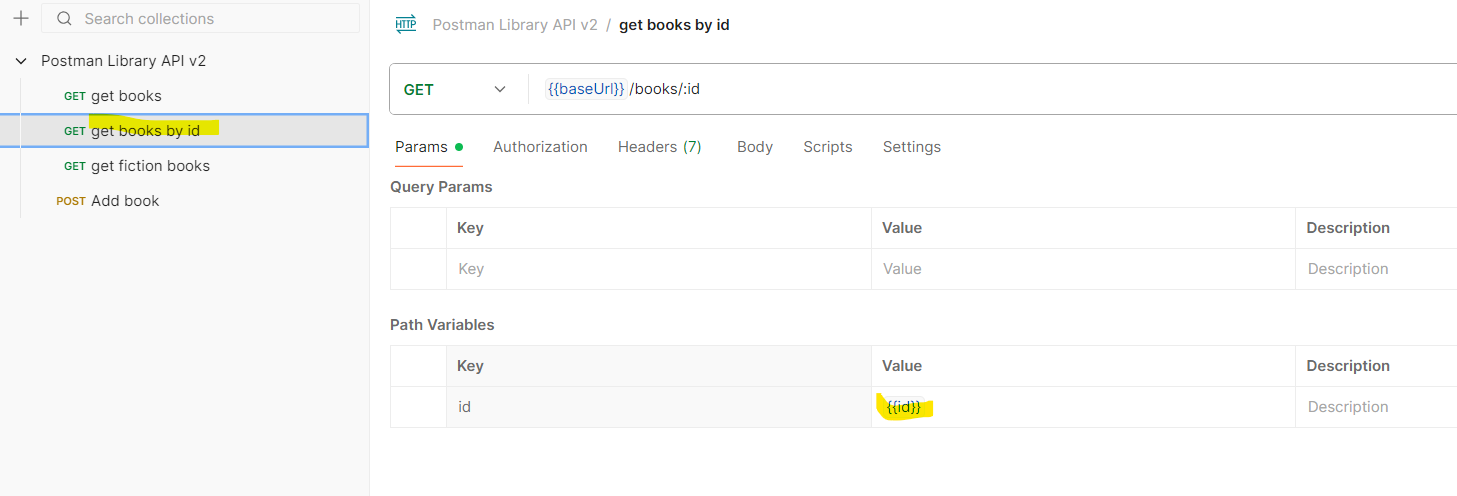
**const id = pm.response.json().id**

**// save the id as a collection variable named "id"**

**pm.collectionVariables.set("id", id)**

The comments above that start with **//** help explain what the code is doing, and are ignored when we run the script.

1. **Save and send**your request.  
     
   When the **201** the response comes back from the API with your newly created book, and the test script will run and save the book's **id** as a collection variable automatically.
2. View your collection variables by clicking on your **Postman Library API v2** collection, then the **Variables** tab.  
     
   The **id** variable has been automatically assigned the id of your new book as its Current Value!
3. You can now use **{{id}}** anywhere in your collection to access this value! This will come in handy for our next request. – Use it in the value pair of Get request as shown below.



Note: If there is no Collection variable named "id" Postman will create a new variable named id and assign the value.