GOREST API TESTING

Made by: Leon Asanovski 221007 and Gorjan Bogoevski 221193

GitHub Link

Postman Collection

Introduction

API (Application Programming Interface) testing is a type of software testing that focuses on verifying the functionality, reliability, performance, and security of application interfaces. Unlike UI testing, which tests the graphical interface, API testing ensures that the underlying business logic and data handling layers work as expected.

APIs serve as the communication bridge between different software systems, enabling them to exchange data and services.

By performing thorough API testing, we aim to catch integration bugs early, improve system reliability, and accelerate release cycles through automation and consistent test coverage.

Setting the environment for testing

We are going to use the application called Postman. This application can be downloaded from the internet on the following link → Postman.

Then you install the application, and it is recommended to login or register. With the student accounts, we can have a GitHub account, and that is the way how we logged in to use this software so we can perform the api testing.

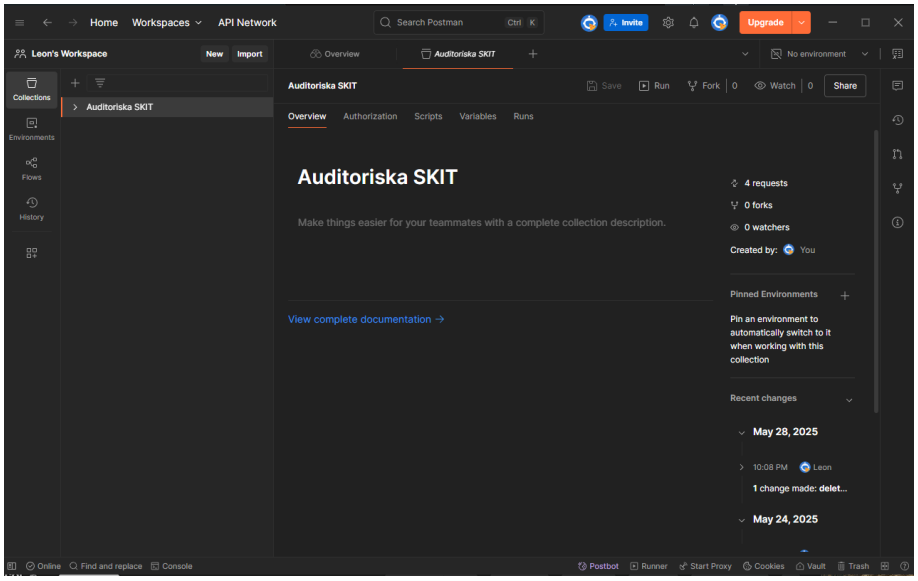


Figure 1 Postman App

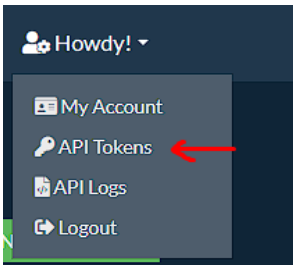
Now we are ready to start our project and start writing the scenarios and their tests.

Setting the project up

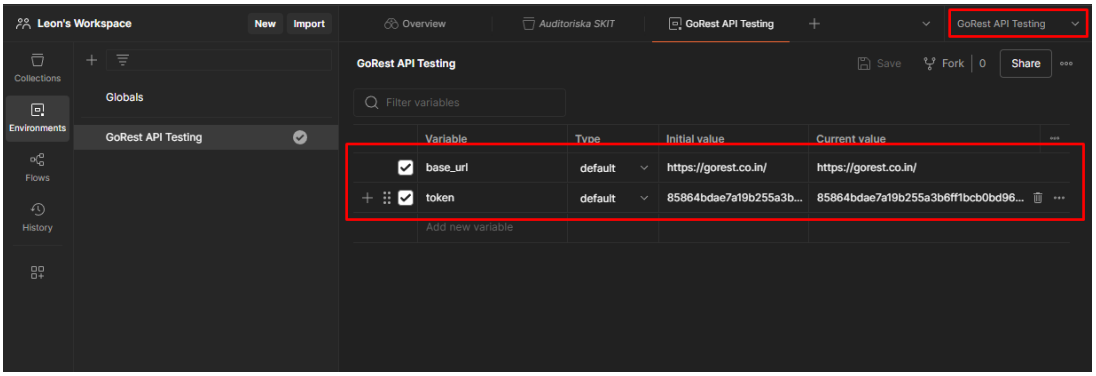
First, we need to mention that we are going to test the following web page GoRest and i it is meant to be used for performing api calls.

Below, there are some steps that need to be followed, so we can set a working environment with the needed variables and a collection of tests for it.

1. Go to: https://gorest.co.in
2. Signup/Login
3. After signing up, go to my account and go to API Tokens.

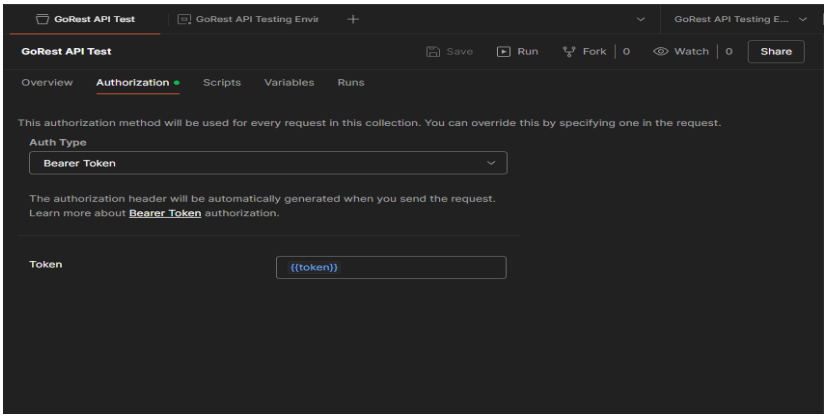


4. Generate a new token, set a descriptive label and get the token copied.
5. In Postman, you need to create a new environment, that is going to have the base URL and token as variables in it. Give the new environment a name, and then from the top right corner select it as default.



Why do we need an environment? -> Because it contains a set of key-value pairs (variables) that you define and reuse across your collection.
For example, the base_url, instead of writing it over and over again, you define it with a name of the variable and just use it in curvy brackets {{base_url}}/...

6. Then we create a new Collection that is going to have a name, and in the authentication, choose bearer token and set it to {{token}}. That is going to fetch the token value from the environment and allow using the api of the web page for testing.



What is the collection? -> It is a folder that contains API calls (GET, POST, PUT, DELETE, etc.) and test scripts.

After setting everything, make one request, to prove that it works.

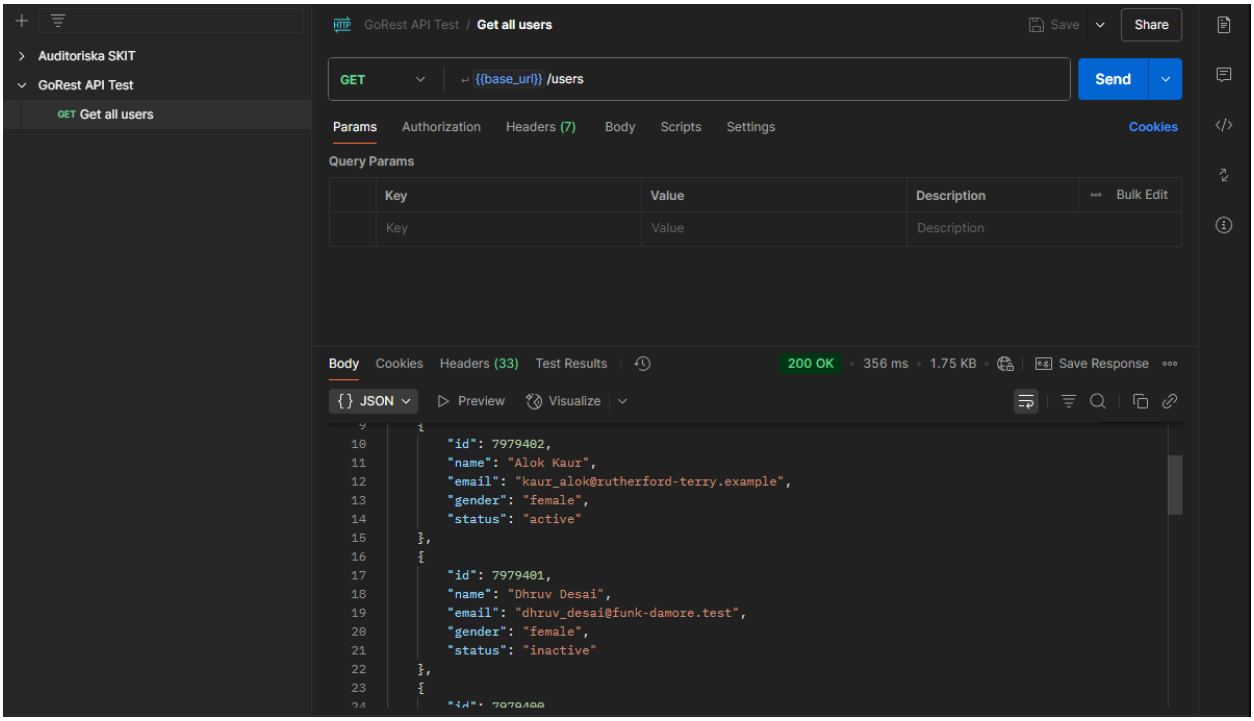
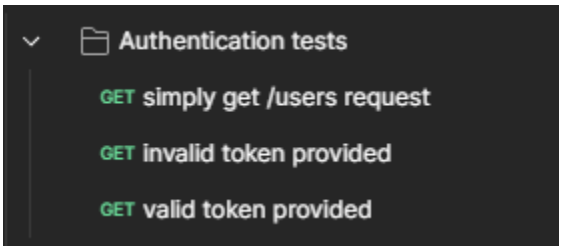


Figure 2 Performing a simple /get request

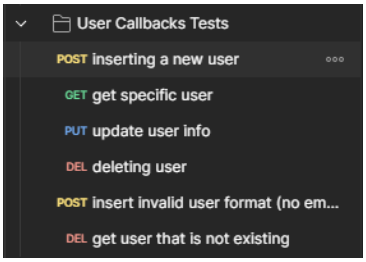
Now when everything is set up, and when we have tested that we have a functional collection with functional environment, we can finally start writing our api calls and their tests, so we can test their functionality.

From this moment on, we are going to explain the usage of the tests we made. We separated them into groups, for better readability, and for better scalability.

1. Authentication Tests
   - simply get /users request – verifying that the /users endpoint returns a 200 OK status with or without a token (publicly accessible resource).
   - invalid token provided - ensure that the API returns a 401 Unauthorized response when an invalid token is provided.
   - valid token provided - correct token allows access and returns a 200 OK status for the /users endpoint.

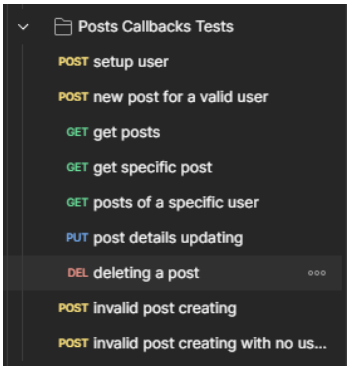


2. User Callbacks Tests
   - inserting a new user - verifies user creation and stores userId for the chain of tests in the folder section.
   - get specific user - ensures that the previously created user exists and was stored correctly.
   - update user info - updates the user's details with dynamic email to avoid duplication errors.
   - deleting user – deletes a created user with DELETE.
   - insert invalid user format (no email provided) - should return a 422-status code.
   - get user that is not existing - confirms the API returns 404 Not Found when attempting to delete a user that doesn't exist.

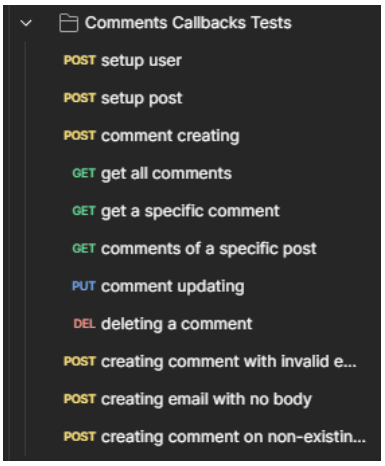


3. Posts Callbacks Tests
   - setup user – sets up a new user that is going to be tested with manipulating its posts later in the tests.
   - new post for a valid user – sets up a new post for the user that is created.

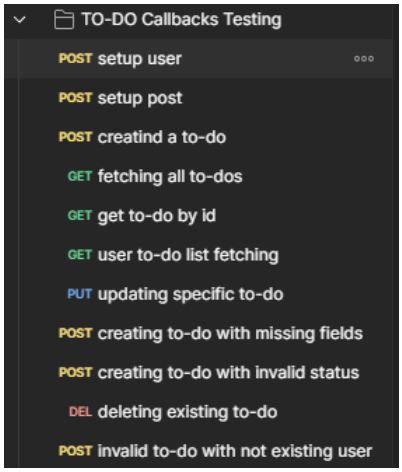
- get posts – here it tests if the posts are retrieved correctly (if the api call returns posts in a Json format).
- get specific post – we are getting a specific post provided and the tests are checking if the post is successfully retrieved.
- posts of a specific user – retrieves the posts of a specific provided user.
- post details updating - verifies that existing post details can be updated successfully.
- deleting a post - deleting a specific post created earlier.
- invalid post creating - ensures that invalid post creation fails with a 422- status code and proper validation error messages.
- invalid post creating with no user – testing if a user is not provided in the body of the post creation.



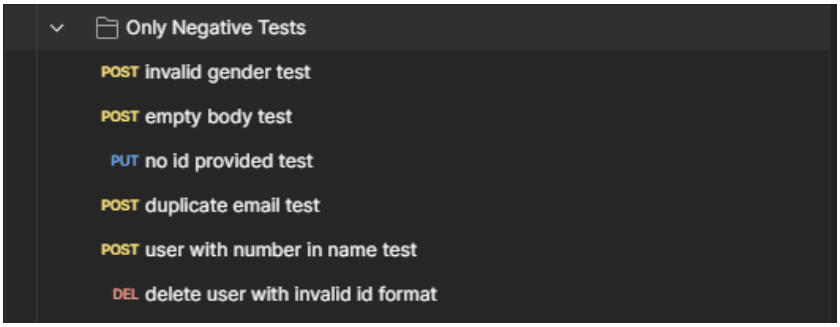
4. Comments Callbacks Tests
    - setup user – sets up a user for further usage.
    - setup post - setting up a post for further usage.
    - comment creating - validates successful comment creation.
    - get all comments – validates the functionality of fetching all comments.
    - get a specific comment - validates the functionality of fetching a specific comment.
    - comments of a specific post – retrieves all comments linked to a specific post.
    - comment updating – updates details of existing comment.
    - deleting a comment - deletes the previously created comment.
    - creating comments with invalid email - ensures that invalid email formats are rejected.
    - creating email with no body - validates that comment body is required.
    - creating comments on non-existing post - tests that the API rejects comments for posts that don't exist.



5. TO-DO Callbacks Testing
    - setup user – sets up a user for further usage.
    - setup post - setting up a post for further usage.
    - creating a to-do – test checks if todo task is successfully created.
    - fetching all to-dos – fetches all to-do tasks that are created.
    - get to-do by id – fetches a specific to-do that is provided and tests its existence.
    - user to-do list fetching – tests if it can fetch all user to-do list.
    - updating specific to-do – updates a specific provided to-do and checks if it is successful
    - creating to-do with missing fields – this test needs to catch a failure.
    - creating to-do with invalid status – this test needs to catch a test fail in the invalid status provided in the body.
    - deleting existing to-do – this test deletes existing to-do and checks if it is successful.
    - invalid to-do with non-existing user - try to create a TODO for a non-existent user.



6. Only Negative Tests
    - invalid gender test - tests the response when an unsupported value is used for the gender field.
    - empty body test - verifies that the API correctly rejects requests with no body.
    - no id provided test - ensures the API returns an error when an id is required but not provided in the endpoint.
    - duplicate email test - tries to create a user with an email that already exists in the system.
    - user with number in name test - checks whether the API allows names that include numeric characters (depending on business rules).
    - delete user with invalid id format – deleting user that has invalid id format (providing a string for example).



7. Chained Testing

    The chained testing folder in the Postman collection performs a complete end-to-end scenario that validates how the GoRest API handles dependencies between resources. The flow simulates the lifecycle of a user, their associated data, and the consequences of removing that user. Each request is dependent on the output of the previous one.

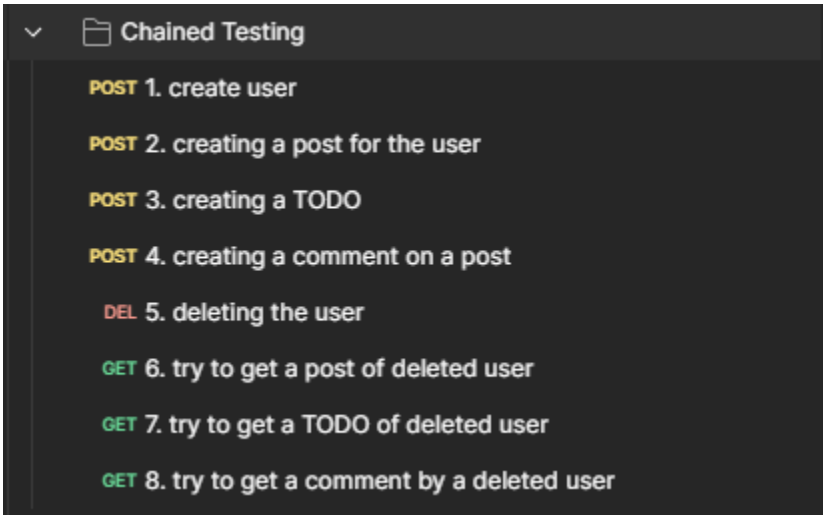
    The test begins with the creation of a new user using a POST request. Once the user is successfully created, their user ID is stored for later use. The second test uses this user ID to create a new post for the user. This post creation is also a POST request and depends on the user being valid and active.

    In the third step, a TODO item is created for the same user using the /todos endpoint. This again uses a POST method and is linked to the previously created user. After that, the fourth test creates a comment on the post made in step two, completing the creation chain. The fifth test then deletes the user using a DELETE request. This action is critical because it sets up the conditions to test how the API behaves when dependent resources are left behind or potentially removed along with the parent user.

Following the deletion, the next three tests use GET requests to attempt retrieval of the post, TODO item, and comment associated with the now-deleted user. These tests are intended to check whether the API:

- Still retains the related data,
- Returns a 404 Not Found,
- Or throws validation errors due to orphaned references.

The entire chain validates the integrity and cleanup behavior of the system when cascading deletions or orphaned data scenarios are introduced. This kind of chained test is valuable for identifying weaknesses in data integrity and ensuring the API behaves as predicted.



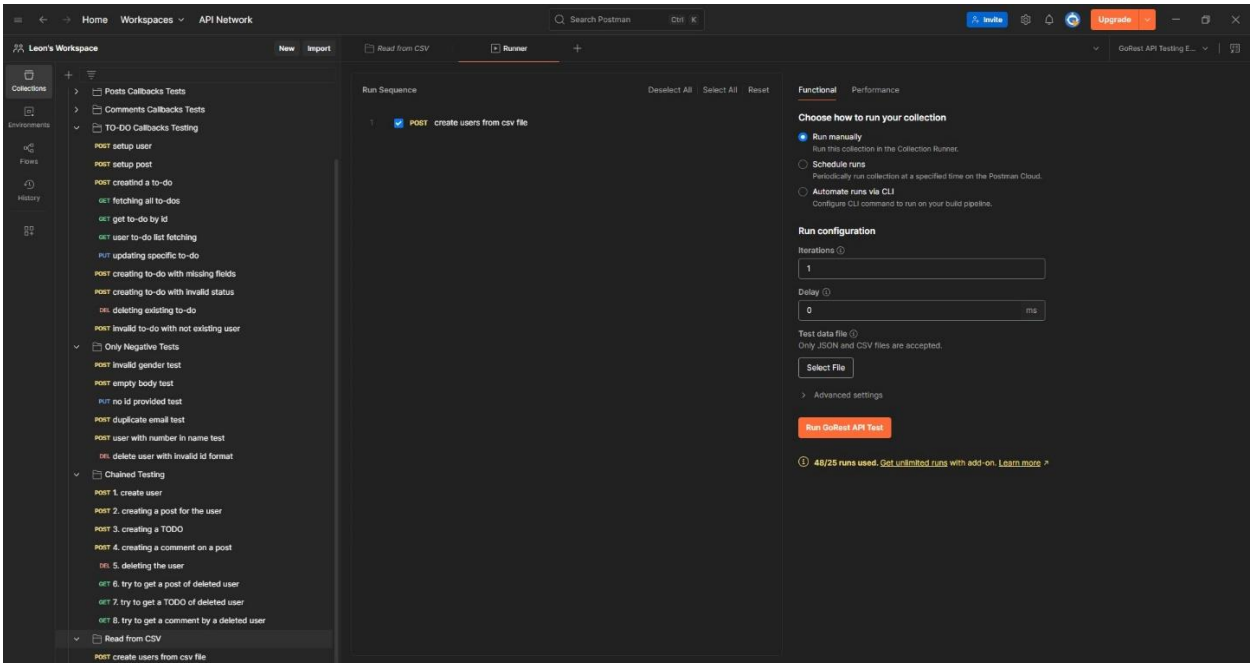
8. Read Object from CSV and Insert Them

   This test is checking if the reading of an object from a csv file is working. To be correct, we need to create a csv file with the proper format of the object we want to insert (in our case we insert several user objects in the database). The second thing we need to keep an eye on is inserting objects that does not exist in the database.

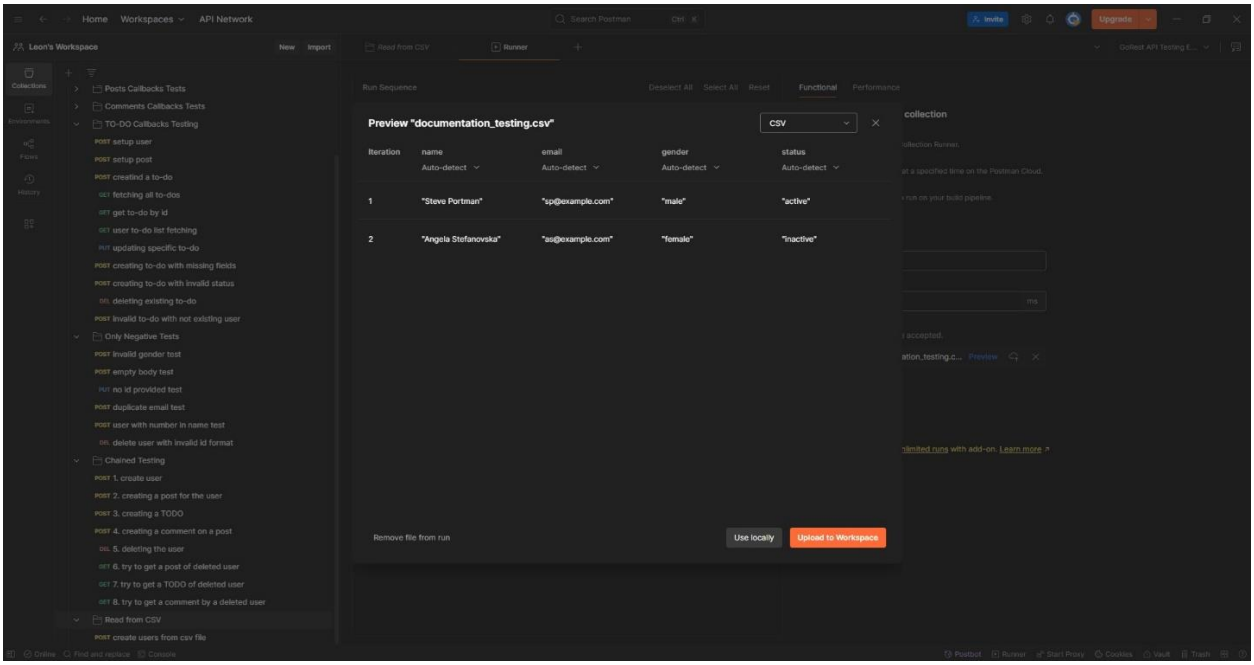
   Example of a csv provided:

```
name,email,gender,status
Alice Smith,alice@example.com,female,active
Gorjan Bogoevski,gorjan@example.com,male,inactive
Leon Asanovski,leon@example.com,male,active
```

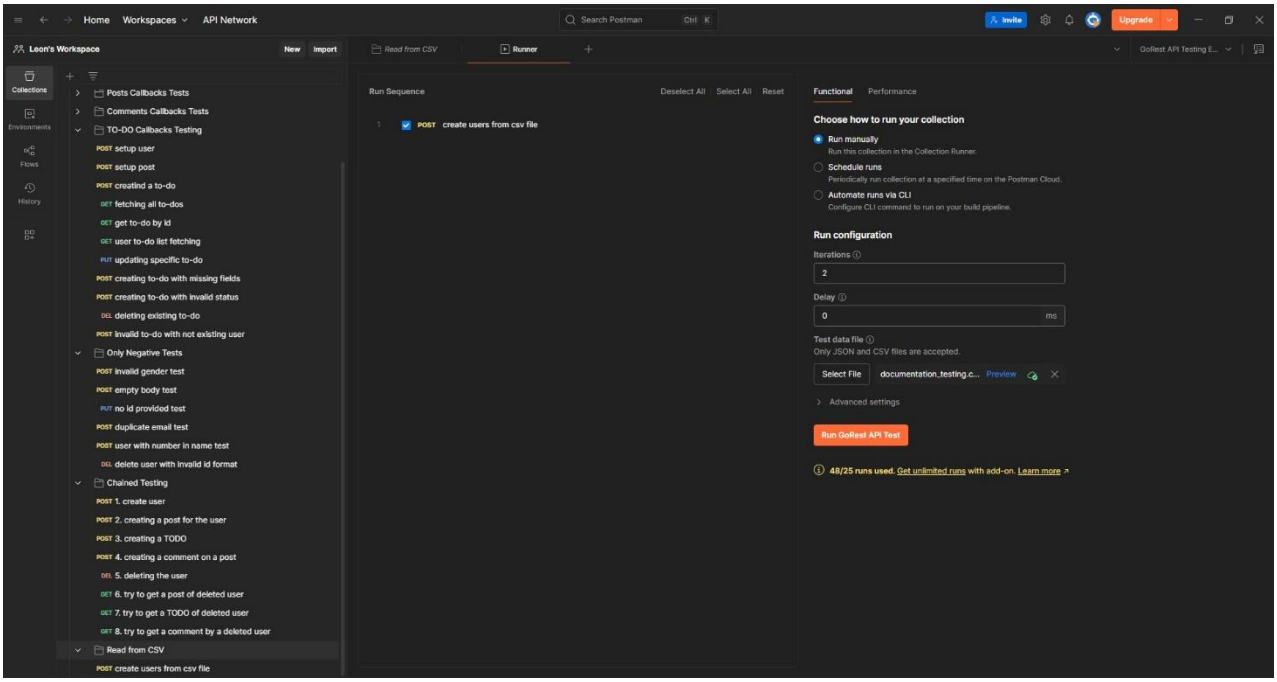
   After that on the folder of the test, we right click and then click Run. We upload the csv file, and if everything is well defined and structured, the test will be successful.



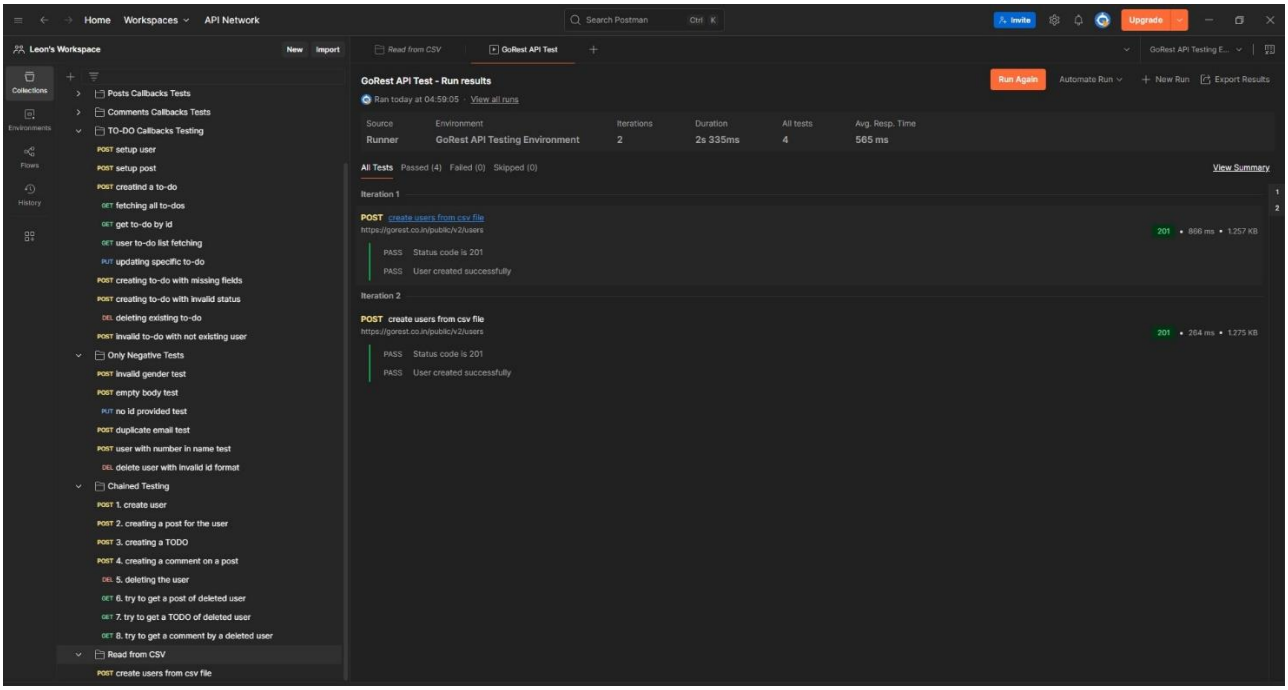
Screenshot 1 Activating Run Pane



Screenshot 2 Importing the .csv file



Screenshot 3 Everything set up for test Run



Screenshot 4 We see the tests passed, the objects are created successfully

Conclusion

Postman is an essential tool for modern API testing, offering a simple yet powerful platform to validate the behavior, reliability, and security of RESTful services. By organizing tests into structured collections and folders, you can cover both positive and negative scenarios, simulate real-world user workflows, and ensure that your API handles edge cases correctly. With the API Testing as a tool that can be used, everyone can test its API callbacks on the application and make the product much more stable, reliable, with a higher success percentage overall.

Link Provided from the Postman Collection → link

The Environment JSON is provided in the zip.