Documentation for the project

Databaser - Inlämningsuppgift 2 - PVT23 - utveckling och testning av API

Prepared by
Kira Popko

April 2024

Content:

Introduction:

API documentation is essential for any developer using an API, as it helps them understand its functionality, parameters, and expected responses. Proper documentation ensures that the API is easy to use, maintain, and scale. The following guidelines will help students create clear, concise, and comprehensive API documentation.

1. Overview:

For the project Inlämningsuppgift 2 was created a database with MongoDB and API should be tested manually and automatically using Postman.

General instruction on how to prepare the project. List of actions:

Nr	Action	Status	Comment
1.	Create GIT HUB repository, it should be public	done	https://github.com/KiraPopko/Inl-mningsuppgift2.PVT23utveckling-testningAvAPI
2.	Create server for the project using MongoDB	done	
3.	Create Mock data (books)	done	
4.1	Performa Manual/Automation testings	partly	
5	Skapa en Postman mock server med exempeldata för ditt API	done	
6	Git repo lämna in until 17.04.2024	done	Make it public
7	Add readme file		
8	Postman collection should be created	done	should be public
9	Lämna in uppgift in time (28.04.2024)		https://kyh.omniway.se/studentcourses/2387536

2. Authentication and Authorization:

Accounts and authorization needed:

- GitHuB repository is public- there can be found javascript script, postman collection, documentation, readme file with special information. Personal account is required.
- Postman collection - collection can be imported or used request examples from section 4 from this documentation. Personal account is required.
- MongoDB- js script will be used and personal username and password should be used. Personal account is required.

3. Preparations:

How to prepare project for work with it:

- Connect Mongo DB Atlas
- Start your server using server.js. Don't forget to change username and password for MongoDB.
- Create a database using seedDB.js script. Create needed amounts of users/objects.
- Check the request in Postman

4. Request and Response Examples:

GET request to see all users

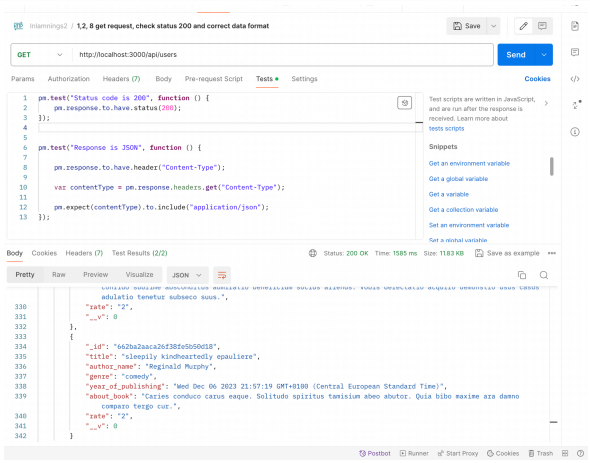
Request	Url endpoint	tests

GET	http://localhost:3000/api/users	pm.test("Status code is 200", function () {
		  pm.response.to.have.status(200);
		});
		
		pm.test("Response is JSON", function () {
		  pm.response.to.have.header("Content-Type");
		  var contentType = pm.response.headers.get("Content-Type");
		  pm.expect(contentType).to.include("application/json");
		});

Request is used to see all objects available in the created database. In response body will be able to see all objects available in database.

First test shows that after sending a request the expected status code is 200 and shows that the request was successful.

The second test verifies that the response is in JSON format.



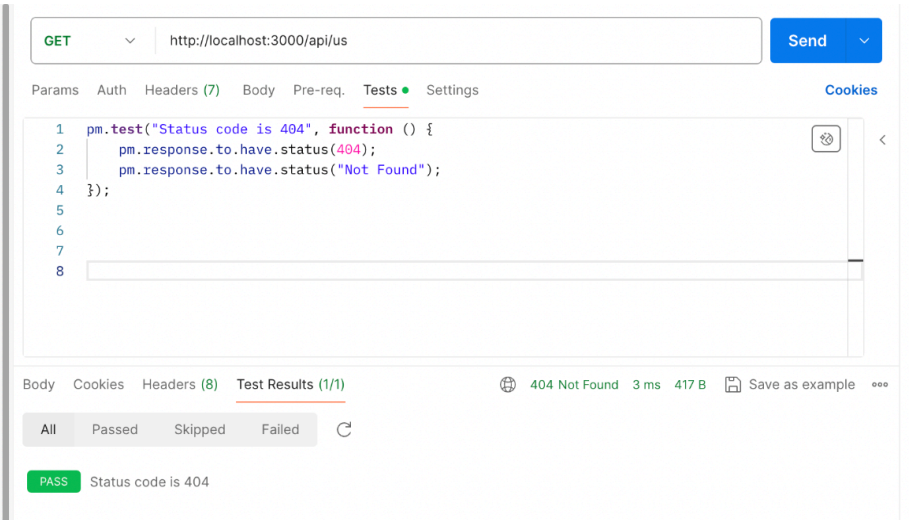
GET request with invalid endpoint

Request	Url endpoint	tests

GET	http://localhost:3000/api/us	`pm.test("Status code is 404", function () {`
		`pm.response.to.have.status(404);`
		`pm.response.to.have.status("Not Found");`
		`});`

Request is used to see that after sending invalid request response returns status 404.

Test shows that after sending request status code is 404, it means that request reached the server, but did not find that was requested, some error in url.



GET request with search criteria

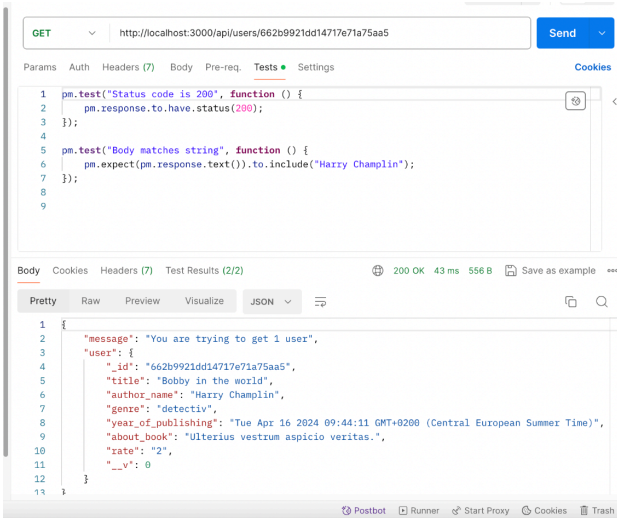
Request	Url endpoint	Tests

GET	http://localhost:3000/api/users/:id
	
	Example:
	
	http://localhost:3000/api/users/662b9921dd14717e71a75aa5
		pm.test("Status code is 200", function () {
		pm.response.to.have.status(200);
		});
		
		pm.test("Body matches string", function () {
		pm.expect(pm.response.text()).to.include("Harry Champlin");
		}); * change include(“”) with your data

Request is used to find a special user/object from the database. In the response body the target object will be available.

First test shows that after sending a request the expected status code is 200. It shows that the request was successful and the user was found.

The second test shows the response contains specific data from the target object.



Pagination

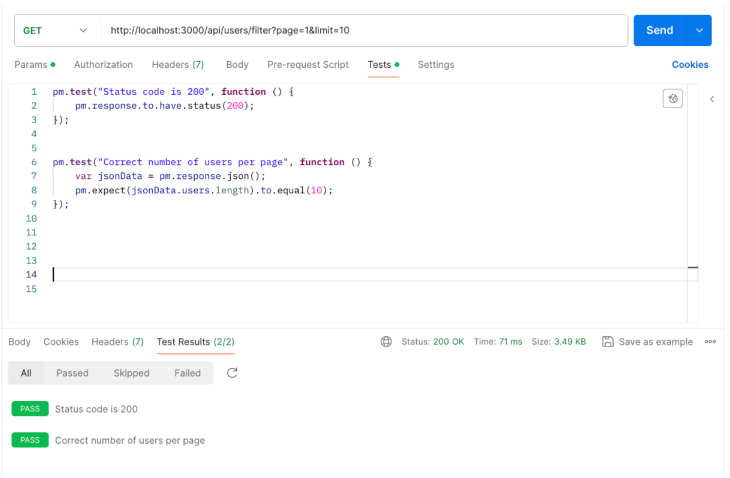
Request	Url endpoint	tests

GET	http://localhost:3000/api/users/filter?page=1&limit=10	pm.test("Status code is 200", function () {
		pm.response.to.have.status(200);
		});
		pm.test("Correct number of users per page", function () {
		var jsonData = pm.response.json();
		pm.expect(jsonData.users.length).to.equal(10);
		});

Request is used to limit the quality of the object on the specific page. In the response body will be available pagination list of users/objects where each page contains up to users 10 from the mentioned page

First test shows that after sending a request the expected status code is 200 and shows that the request was successful.

The second test is used to verify that the correct amount of users per page is available.



Special characters and non-English text

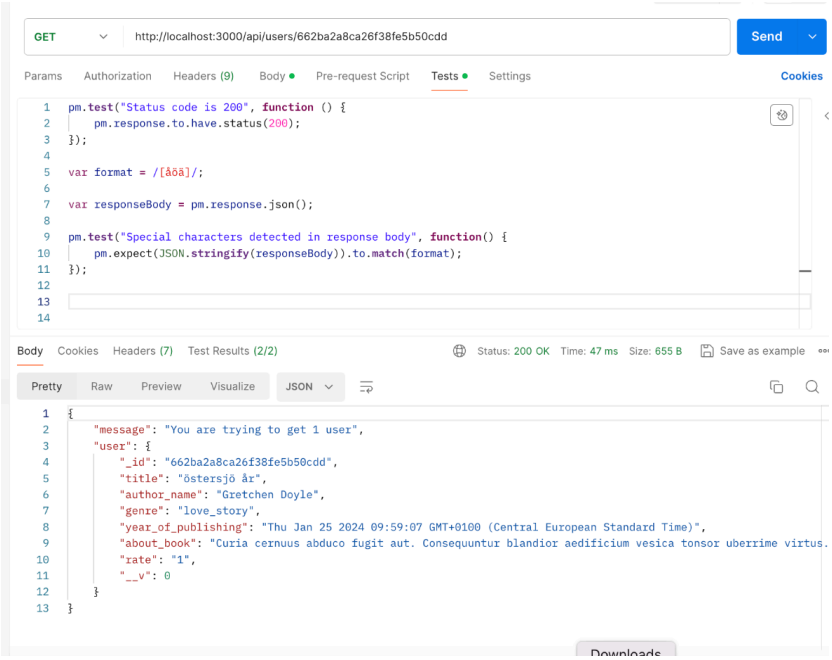
Request	Url endpoint	tests

GET	http://localhost:3000/api/users/:id
	
	Example:
	http://localhost:3000/api/users/662ba2a8ca26f38fe5b50cdd
	
	To create user with non english text was used PUT request
		pm.test("Status code is 200", function () {
		pm.response.to.have.status(200);
		});
		
		var format = /[åöä]/;
		var responseBody = pm.response.json();
		
		pm.test("Special characters detected in response body", function() {
		pm.expect(JSON.stringify(responseBody)).to.match(format);
		});

User mentioned in request contains non-english characters.

First test shows that after sending a request the expected status code is 200 and shows that the request was successful.

The second test API handles special characters correctly. Special symbols are displayed correctly and do not cause issues with object reading



POST request

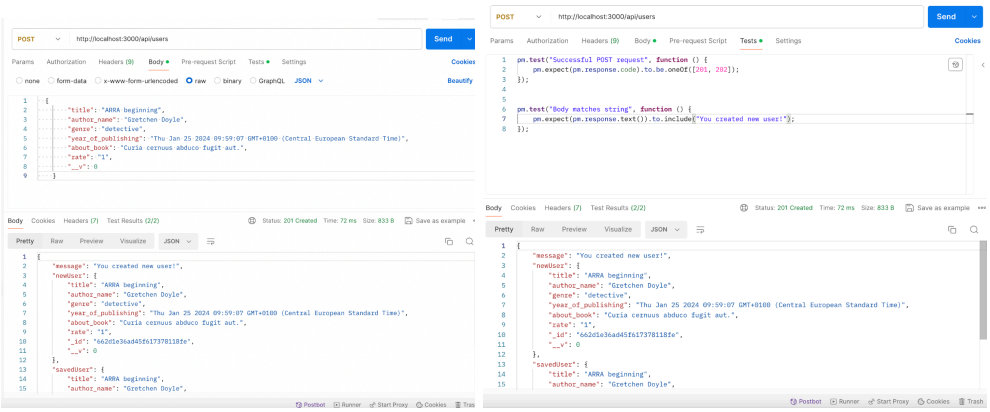
Request	Url endpoint	tests

POST	http://localhost:3000/api/users	pm.test("Successful POST request", function () {
		pm.expect(pm.response.code).to.be.oneOf([201, 202]);});
		pm.test("Body matches string", function () {
		pm.expect(pm.response.text()).to.include("You created new user!");});

Request is used to create a new user/object with a new id.

First test shows that after sending a request the expected status code is 201 and shows that a new user was created successfully.

The second test shows that the response body contains text that a new user has been created.



PUT request

Request	Url endpoint	tests

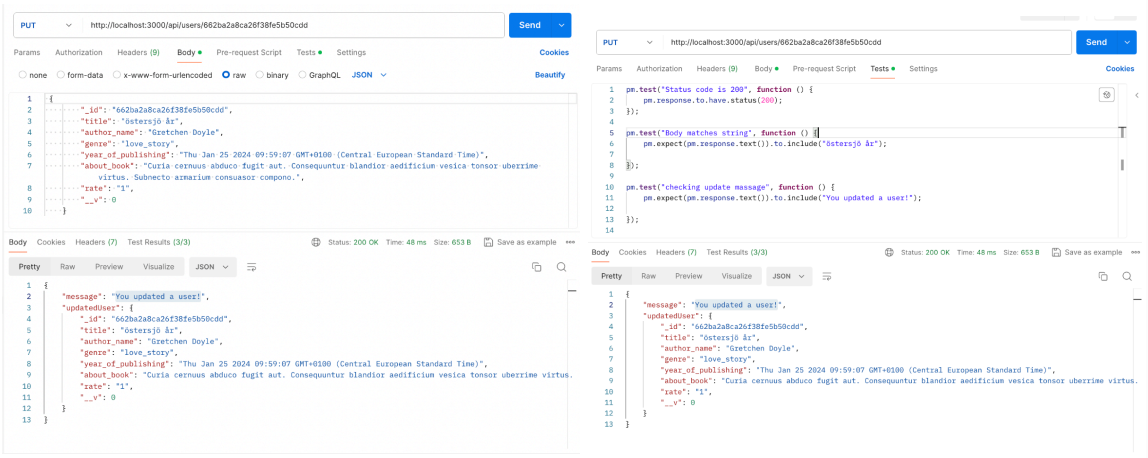
PUT	http://localhost:3000/api/users/:id
	
	Example:
	http://localhost:3000/api/users/662ba2a8ca26f38fe5b50cdd
		pm.test("Status code is 200", function () {
		pm.response.to.have.status(200);});
		
		pm.test("Body matches string", function () {
		pm.expect(pm.response.text()).to.include("östersjö år");
		}); * in include() use your changes in the object
		
		pm.test("checking update massage", function () {
		pm.expect(pm.response.text()).to.include("You updated a user!");});

Request is used to update data in an already existing user.

First test shows that after sending a request the expected status code is 200 and shows that the request was successful and in this case user was updated.

The second test shows that the response body contains data that has been changed.

The third test shows that the response body contains text that the user has been updated.



DELETE request

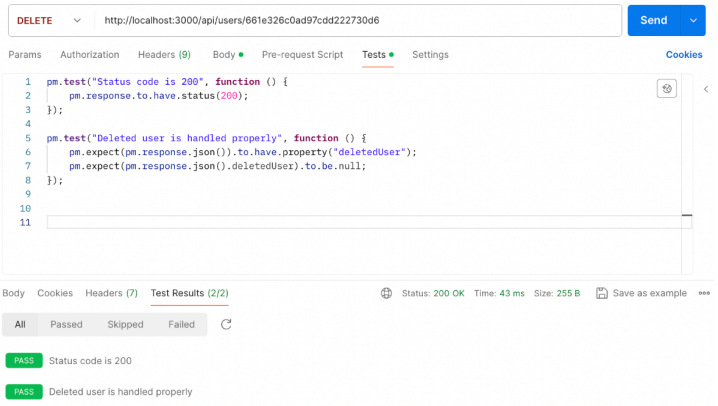
Request	Url endpoint	tests

DELETE	http://localhost:3000/api/users/:id
	
	Example:
	http://localhost:3000/api/users/661e326c0ad97cdd222730d6
		pm.test("Status code is 200", function () {
		pm.response.to.have.status(200);
		});
		pm.test("Deleted user is handled properly", function () {
		pm.expect(pm.response.json()).to.have.property("deletedUser");
		});

Test is used to delete existing user/object from the database.

First test shows that after sending a request the expected status code is 200 and shows that the request was successful, in this case the user was deleted.

The second test shows that in the response body is property deleteUser, that shows that user was deleted.



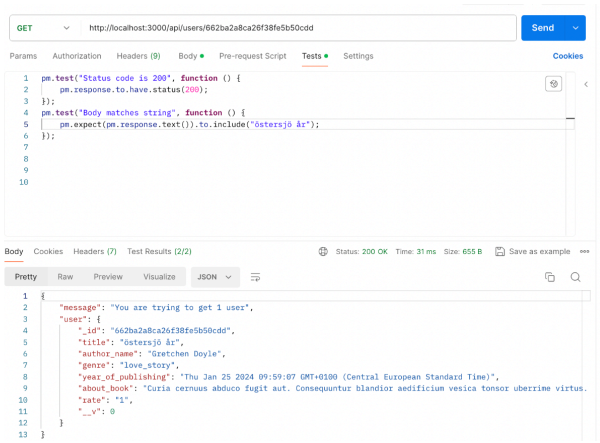
Correct updates handling

Request	Url endpoint	tests

GET	http://localhost:3000/api/users/:id
	
	Example:
	http://localhost:3000/api/users/662ba2a8ca26f38fe5b50cdd
		pm.test("Status code is 200", function () {
		pm.response.to.have.status(200);
		});
		
		pm.test("Body matches string", function () {
		pm.expect(pm.response.text()).to.include("östersjö år");
		}); *in include() use your data

First test shows that after sending a request the expected status code is 200 and shows that the request was successful and in this case user was updated.

The second test shows that the response body contains data that has been changed.

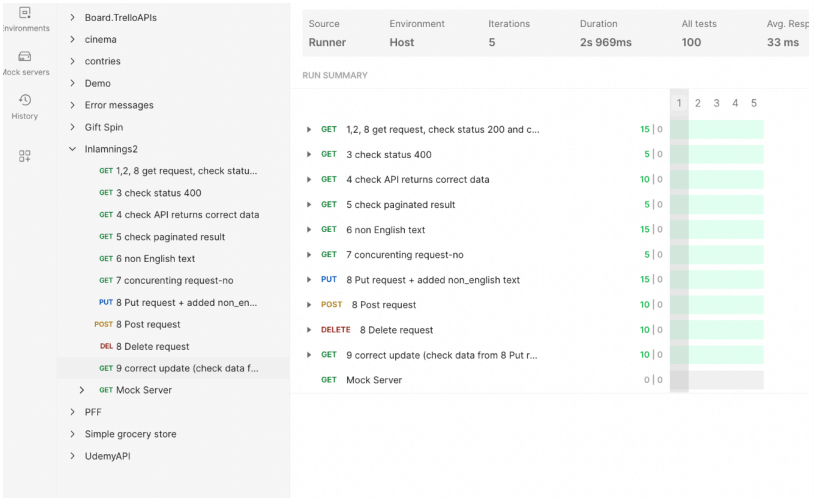


Concurrent request

By pressing Run collection, choose the proper collection and run it. Check the result.

In example was performed 5 iterations of collection and all tests passed.

For proper data for concurrent requests should be used to automate runs via CLI, it was not possible to install it on my local machine because of limited access and lack of available passwords.

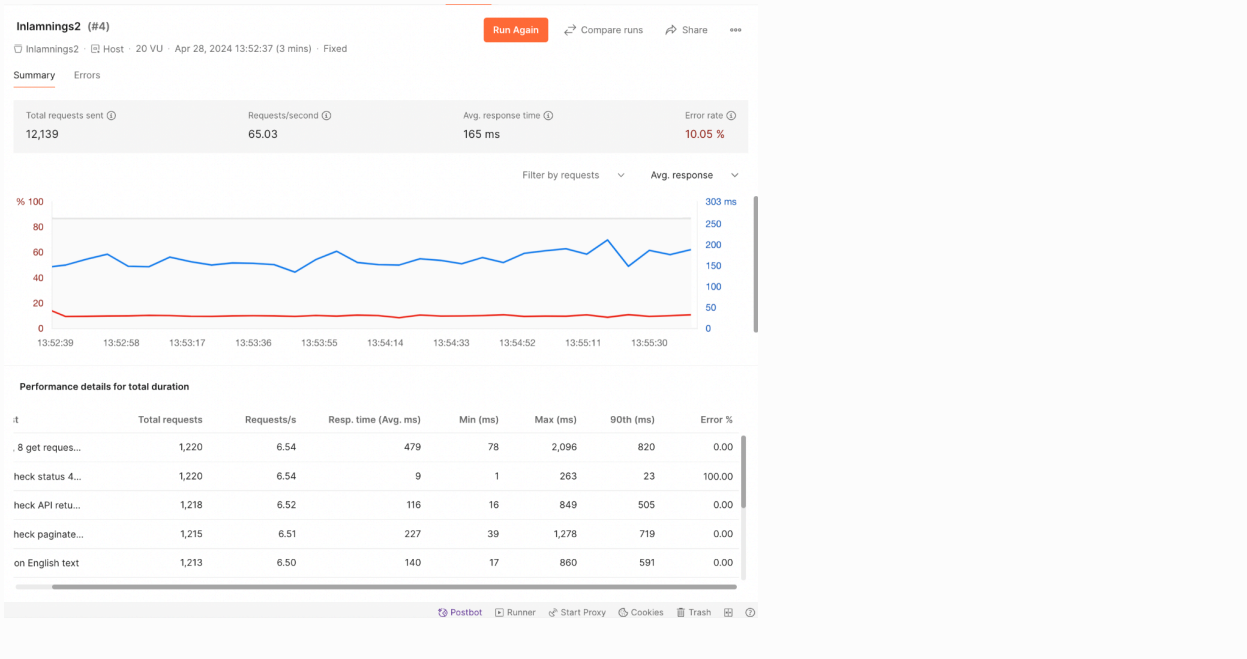


Test the API's performance under heavy load

Open run collection, choose correct collection and instead functional choose performance testing. Choose load profile, virtual users and test duration.

For my testing was chosen Fixed load profile, that means that the amount of users will not change under testing time, 20 virtual users and 3 min test duration.

Total number requests sent to the endpoint was 12 139,and 65.03 requests sent per second with average response time 165ms. There was a 10.05% error rate, because in the collection is an example that returns status 404. That error rate is not 0% is an expected result.



9. Error Handling

Error that was detected under testing:

- 404-not found. The requested URL was not found. For example, instead of users is user, instead of api is apo.
- 500- internal server error. The request reached the server, but did not find proper information. For example check does ID in request match that ID that should be found.