l'm not a robot



Share copy and redistribute the material in any medium or format for any purpose, even commercially. Adapt remix, transform, and build upon the material for any purpose, even commercially. The licensor cannot revoke these freedoms as long as you follow the license terms. Attribution You must give appropriate credit, provide a link to the license, and indicate if changes were made . You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use. ShareAlike If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. No additional restrictions You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits. You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation. No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material. API testing is a software testing type that tends to validate the application programming interfaces. As per Postman API, API testing is confirming that an API is working as expected. It sends requests to an API and monitors the responses to check its behavior to assess the functionality, reliability, performance, and security of an API. It is usually considered to be a crucial part of the API development lifecycle. What is Postman recently ranked on top for best API Platform, by G2 in its 2023 Spring Reports. It is easy and user-friendly and offers various courses and projects. To use it, one must register in the Postman platform. As per Postman, it is an API platform for building and using APIs that simplifies each step of the API lifecycle and streamlines collaboration so one can create better APIs. It is trusted by over 25 million users worldwide. It has a friendly community too and allows API testing with REST History of Postman:Just like every origin story, the origin, purpose, development till date has many theories but web based APIs tend to be more recent and the example can be Google Maps, something we use pretty often. Postman API was created by Abhinav Asthana, Ankit Sobti and Abhijit Kane in Bangalore, India (2012). Firstly, it was developed as a plugin for Google Chrome, then a rich client, and finally a thin client and now, Postman Inc., originally from India, has its headquarters in San Francisco. Syntax: Example of API Testing using Postman How about a bookstore or maybe Library API? We'll test that in this example. Firstly, get acquainted with the basics of Postman API and API testing fundamentals first like HTTP methods, mostly. You need to create an account in the Postman API platform. Download and create an account on Postman API. You can also use their workspace online. Create a new workspace and name it accordingly like postman API. I wanted to add books so I created this request. Choose POST as the request method. Enter the API endpoint URL in the address bar. You can also save it as baseURL. You need to save it as a variable first. Let's assume the baseURL has a current value as in the following code snippet in body. Click the "Send" button to execute the request. Enter any book details in the code snippetAnd then check the response in the "Response" window. You should be able to view book details. Now for API testing, first, go to the "Tests" tab.Response window at bottom showing book detailsLook for the code bracket icon. Select JavaScript Fetch. Write JavaScript code for fetch response is, like: JavaScript const myHeaders = new Headers();myHeaders.append("Content-Type", "application/json");myHeaders.append("api-key", "postmanrulz"); const raw = JSON.stringify({ "title": "To Kill a Mockingbird New", "author": "Harper Lee", "genre": "fiction", "yearPublished"; 1960}); const requestOptions = { method: "POST", headers: myHeaders, body: raw, redirect: "follow"}; fetch(" requestOptions) .then((response) => response.text()) .then((result) => console.log(result)) .catch((error) => console.error(error)); For test body, write a code snippet like:const id = pm.response.json().idpm.collectionVariables.set("id", id)Click the "Run" button in the "Tests" tab to execute the code snippet. The results for the test are displayed almost instantly. Check and evaluate accordingly.Output: Since the API Testing was successful. It shows the response status and body. The code 201 ishighlighted in green. According to Postman API, the table contains the status code that it offers. Code Range Description Example 2-- Success 200 - OK201 - Created 204 - No content but OK 3-- Redirection 4--Client Error 400 - Bad request401 - Unauthorized403 - Not Found 5-- Server Error 500 - Internal server error502 - Bad gateway504 - Gateway timeoutKey Features: It can build various API requests like, GET, POST, PUT, DELETE, etc. with user-friendly interfaces for defining URLs, headers, body parameters, and authentication methods.It can send, analyze and execute requests easily.It has simple and neat documentations.Write tests using JavaScript or Collections/Environments that validates response codes, body content, headers, and other aspects. You can group them, and execute entire collections of tests at once using the collection runner to streamline automated testing. Share collections and/or test scripts with team members via workspaces or public links to enable collaborative API development. Manage environment variables to store and reuse values like API keys, base URLs, across requests and tests, enabling maintainability and reuse values like API keys, base URLs, across requests and tests. development and testing purposes, allowing you to work on API integration without relying on external endpoints. It can track API performance metrics like, response times, errors over time to identify potential issues and optimize performance. Advantages: It has neat, user friendly interface. It tends to organize and group API requests (in Postman API collections) efficiently. It can monitor and analyse by tracking API performance, response times etc. Disadvantages: There might be few paid tools or features in this platform. They always need to add an additional server to the user's local environment. Storing tokens is not safe in Postman API. This page provides post-response script examples for various API testing scenarios in Postman. You can use these post-response scripts to validate response scripts to validate response scripts to validate response scripts to validate response scripts in your first post-response scripts to validate response scripts in your first post-response scripts in your first post-response scripts to validate response scripts in your first post-response scripts in your first post-respons tab. Enter the following JavaScript code:pm.test("Status code is 200", function () { pm.response.to.have.status(200); }); This code uses the pm library BDD syntax, which provides options to optimize how readable your tests are to you and your collaborators. In this case, the code uses a to.have chain to express the assertion. This test checks the response code is 200, the test will pass, otherwise it will fail. Select Send and go to the Test Results tab in the response area. To learn what test results look like when they pass or fail, change the status code in the assertion code and send the request again. You can structure your test assertions in a variety of ways, depending on how you want the results to output. The following code is an alternative way of achieving the same test as the one before using the expect syntax:pm.test("Status code"). is 200", () => { pm.expect(pm.response.code).to.eql(200); }); Refer to the Chai Assertion Syntax options. Use multiple assertions as part of a single test. Use this to group together related assertions; pm.test("The response has all properties", () => { const responseJson = pm.responseJson.id).to.have.lengthOf(1);}); ff any of the contained assertions must be successful for the test to pass.Parse response body dataTo carry out assertions on your responses, you will first need to parse the data into a JavaScript object that your assertions can use. To parse JSON data, use the following:const responseJson = xml2Json(pm.responseJson = xml2Json();To parse XML, use the following:const responseJson = xml2Json();To parse XML, use the following:const responseJson();To parse XML logging useful.To parse CSV, use the CSV parse (csv-parse/lib/sync) utility:const parse = require('csv-parse/lib/sync');const response.text()); To parse HTML, use cheerio:load(pm.response.text()); Console.log(\$.html()); Handle responses that don't parse for the response body into JavaScript because it's not formatted as JSON, XML, HTML, CSV, or any other parsable data format, you can still make assertions on the data.Test if the response body has a string:pm.test("Body contains string",() => { pm.expect(pm.response.text()).to.include("customer_id");});This doesn't tell you where the string was encountered because it carries out the test on the whole response body. Test if a response matches a string:pm.test("Body is string", function () { pm.response.to.have.body("whole-body-text");});Make assertions on the HTTP response.to.have.body("whole-body-text");});Make assertions on the HTTP response.to.have.body("whole-body-text");}); bodyCheck for particular values in the response body:pm.test("Person is Jane", () => { const responseJson.age).to.eql(23);});Test status code:pm.test("Status code is 201", () => { pm.responseJson.age).to.eql(23);});Test status code:pm.test("Status code is 201", () => { pm.responseJson.age).to.eql(23);});Test status code:pm.test("Status code:p want to test for the status code being one of a set, include them all in an array and use
oneOf:pm.test("Successful POST request", () => { pm.expect(pm.response.code).to.be.oneOf([201,202]); }); Check the status code text:pm.test("Status code name has string", () => { pm.response.to.have.status("Created"); }); Check that a response header is present:pm.test("Content-Type header is application/json", () => { pm.expect(pm.response.headers.get('Content-Type')); }); Test for a response header is application/json", () => { pm.expect(pm.response.headers.get('Content-Type')); }); Test for a response header is present in the response:pm.test("Cookie isLoggedIn is present", () => { pm.expect(pm.cookies.has('isLoggedIn')).to.be.true; });Test for a particular cookie value:pm.test("Response time to be within a specified range:pm.test("Response time to be within a specified range:pm.test(" is less than 200ms", () => { pm.expect(pm.response.responseTime).to.be.below(200); }); Common assertions might help you write your post-response scripts. For a more comprehensive overview of what you can include in your assertions, refer to the Chai Assertian tesponse assertions might help you write your post-response scripts. For a more comprehensive overview of what you can include in your assertions might help you write your post-response scripts. For a more comprehensive overview of what you can include in your assertions might help you write your post-response scripts. For a more comprehensive overview of what you can include in your assertions might help you write your post-response scripts. For a value against a variableCheck if a response property has the same value as a variable (this example uses an environment variable):pm.test("Response property matches environment.get("name"));});See Using variables to learn more about using variables in your postresponse scripts.Assert a value typeTest the type of any part of the response:const jsonData = pm.response.json();pm.test("Test data type of the response", () => { pm.expect(jsonData.name).to.be.an("object"); pm.expect(jsonData.name).to.be.an("ob pm.expect(jsonData.email).to.be.undefined; pm.expect(jsonData.email).to.be.null;});Assert array properties("Test array properties", () => { pm.expect(jsonData.errors).to.be.empty; pm.expect(jsonData.areas).to.include("goods"); const notificationSettings = jsonData.settings.find (m => m.type === "notificationSettings.detail).to.have.members(["email", "sms"]); }); The order in .members doesn't affect the test.Assert object", "Could not find the setting"); pm.expect(notificationSettings.detail).to.have.members(["email", "sms"]); }); The order in .members doesn't affect the test.Assert object", "Could not find the setting"); pm.expect(notificationSettings.detail).to.have.members(["email", "sms"]); }); The order in .members doesn't affect the test.Assert object", "Could not find the setting"); pm.expect(notificationSettings.detail).to.have.members(["email", "sms"]); }); The order in .members doesn't affect the test.Assert object", "Could not find the setting"); pm.expect(notificationSettings.detail).to.have.members(["email", "sms"]); }); The order in .members doesn't affect the test.Assert object", "Could not find the setting"); pm.expect(notificationSettings.detail).to.have.members(["email", "sms"]); }); The order in .members doesn't affect the test.Assert object", "Could not find the setting"); pm.expect(notificationSettings.detail).to.have.members(["email", "sms"]); }); The order in .members doesn't affect the test.Assert object", "Could not find the setting"); pm.expect(notificationSettings.detail).to.have.members(["email", "sms"]); }); The order in .members doesn't affect the test.Assert object", "Could not find the setting"); pm.expect(notificationSettings.detail).to.have.members(["email", "sms"]); }); The order in .members doesn't affect the test.Assert object", "Could not find the setting"); pm.expect(notificationSettings.detail).to.have.members(["email", "sms"]); }); }); The order in .members doesn't affect the test.Assert object", "Could not find the setting"); }); }); }); } propertiesAssert that an object has keys or properties:pm.expect({a: 1, b: 2}).to.have.any.keys('a', 'b');pm.expect({a: 1, b: 2}).to.have.any.keys('a', 'b');pm .keys is run without .all or .any, the expression defaults to .all. As .keys behavior varies based on the target type, it's recommended to check the type before using .keys with .a.Assert that a value is in a setCheck a response value against a list of valid options: pm.test("Value is in valid list", () => { pm.expect(pm.response.json().type) .to.be.oneOf(["Subscriber", "Customer", "User"]);});Assert that an object is containedCheck that an object is part of a parent object: pm.test("Object is contained", () => { const expectedObject = { "created": true, "errors": [] }; pm.expect("Object is contained", () => { const expectedObject;});});The .deep assertion causes all .equal, .include, .members .keys, and .property assertions that follow in the chain to use deep equality instead of strict (===) equality.Assert the current environment.name).to.eql("Production");});Troubleshoot common test errors. () => { pm.expect(pm.environment.name).to.eql("Production");});Troubleshoot common test errors. unexpected behavior in your post-response scripts, the Postman Console can help you to identify the source. By combining console.log(), console.info(), console.info(), console.info(), console.error() debug statements with your test assertions, you can examine the content of the HTTP requests and responses, and Postman data items such as variables. You can also use the console.clear() method to clear information from the console.log(pm.response.json().name);Log the type of variable or response property:console.log(pm.response.json().id);Use Console logs to mark code execution, sometimes known as "trace statements": if (pm.response.json().id) { console.log("id was found!"); } else { console.log("id was found!"); } else { console.log("id was found!"); } code:pm.expect(1).to.eql("1"); This happens because the test is comparing a number to a string value. The test will only return true if both the type and value are equal. Variable not defined errorYou might encounter the ReferenceError: is not de or is outside the scope of your test code.In the following example, a JSON object is the value of a variable in the first test. The second test's code.pm.test("Test 1", () => { const jsonData = pm.response.json(); pm.expect(jsonData.name).to.eql("John");}); pm.test("Test 2", () => { pm.expect(jsonData.age).to.eql(29); });Make sure variables are available at the global scope if test functions needs to reference it. In the previous example, moving const jsonData = pm.response.json(); before the first pm.test ("Test 2", () => { pm.expect(jsonData.age).to.eql(29); });Make sure variables are available at the global scope if test functions.Assertion undefined errorYou might encounter the AssertionError: expected undefined to deeply equal error. Typically this happens when you are referring to a property that doesn't exist or is out of scope.const jsonData = pm.response.json();pm.expect(jsonData.name).to.eql("John");In this example, if you get the error AssertionError: expected undefined to deeply equal 'John', this indicates that the name property isn't defined in the jsonData object. Test not failing There may be occasions where you expect a test to fail, and it doesn't. Make sure your test code is syntactically correct, then resend your request. In the following example, the test is expected to fail because true doesn't equal false. The test actually passes because the pm.test function isn't correctly defined. The pm.test function is missing the first parameter, which is a text string that displays in the test result output. You can validate your JSON Schema with Ajv.In the following example, the test will pass only when the request includes a query parameter named alpha.const schema = { required: ["args"], properties: { args: [Schema validator, which supersedes the deprecated tv4.Send an asynchronous requestYou can send a request from your test code and log the response.json());});Previous style of writing Postman tests (deprecated)This section refers to deprecated script syntax used in earlier versions of Postman. If you are writing new scripts, use the current syntax. The previous style of writing Postman tests relies on setting values for the tests object. Set a descriptive key for an element in the object and then assert if it's true or false. For example, the following will check if the response body has the user id string tests["Body contains user id"] = responsebody.has("user id"); If you use the previous style of writing Postman tests, the syntax will appear in the code editor with a strikethrough to indicate that the style is deprecated. A warning will log to the Postman tests, the syntax for the current style that's recommended instead. Add as many keys as needed, depending on how many things you want to test for. View your test results in the response tab. The tab header shows how many tests passed, and the keys that you set in the tests variable are listed there. If the value evaluates to true, the test passed.postman.setEnvironmentVariable("key", "value"); const array = [1, 2, 3, 4]; postman.setEnvironmentVariable("array", JSON.stringify(array, null, 2)); const obj = { a: [1, 2, 3, 4], b: { c: 'val' } }; postman.setEnvironmentVariable("obj", JSON.stringify(obj)); postman.getEnvironmentVariable("key"); const array = [1, 2, 3, 4]; postman.setEnvironmentVariable("key", "value"); const array = [1, 2, 3, 4]; postman.setEnvironmentVariable("key", "value"); const array = [1, 2, 3, 4]; postman.setEnvironmentVariable("key", "value"); const array = [1, 2, 3, 4]; postman.setEnvironmentVariable("key", "value"); const array = [1, 2, 3, 4]; postman.setEnvironmentVariable("key"); const array = [1, 2, 3, 4]; postman.setEnvironmentVariable("key JSON.parse(postman.getEnvironmentVariable("key");
postman.getEnvironmentVariable("key"); postman.getGlobalVariable("key"); postman.getGlobalVa string"] = responseBody.has("string_you_want_to_search"); tests["Body is correct"] = responseBody === "responseBody];tests["Your test name"] = data.value === 100; tests["Content-Type is present"] = postman.getResponseHeader("Content-Type");tests["Content-Type is present"] = postman.getResponseHeader("Content-Type"); tests["Response time is less than 200ms"] = responseCode.code == 200; tests["Status code is 200"] = code name has string"] = responseCode.code === 201 || responseCode.code === 202;Need help debugging a script? Connect with developersAsk the communityLast modified: 2025/05/12 Postman is a tool that can help you develop APIs. From capturing and validating to testing requests and responses! You need to perform API testing to ensure that your Application Programming Interface or API is working correctly and as it should. This article shows you how API testing is done using Postman and JavaScript.IntroductionIn todays software development ng the guality and functionality of APIs is crucial. Modern software applications are built around APIs, which make sure they function properly and follow the required standards, APIs go through extensive testing, just like any software component. This is where API testing comes into play.API testing involves testing the APIs directly to verify their functionality, reliability, performance, and security. In the long term, it helps save time and comprehensive API testing. In this guide, I will explain how API testing in Postman works using JavaScript and provide you with the knowledge needed to create more effective API testing is a type of software testing that focuses on verifying that APIs function as expected. Unlike traditional UI testing, which tests the graphical interface of an application, API testing examines to talk to each other. This involves sending requests to API endpoints and analyzing the responses to ensure they meet the expected outcomes. Testing APIs early is crucial to catch any issues before they affect the rest of the application. It helps ensure that the core functionality of the application is reliable and performs well before any front-end components are even developed. This approach helps catch bugs early in the development process, leading to a more robust and stable software product.API testing covers various aspects of an APIs functionality, including:Functionality Testing: Ensuring that the API is secure dresults.Performance Testing: Verifying that the API is secure dresults.Performance Testing: Assessing the API is secure dresults.Performance Testing: Ensuring that the API is secure dresults.Performance Testing: Assessing the API is secure dresults.Performance Testing: Ensuring that and protected against unauthorized access and potential vulnerabilities. Reliability Testing: Ensuring that the API consistently performs well under various scenarios and does not fail unexpectedly. Integration of automated and manual testing techniques. Automated tests help in running repetitive tasks and regression tests efficiently, while manual testing allows for more exploratory and in-depth testing of complex scenarios. By incorporating API testing into your development workflow, you can make sure that your APIs are reliable, performant, and secure, ultimately and in-depth testing into your development workflow. leading to better software quality and user experience. Why is API Testing Important? API testing is essential for several reasons. At its core, it ensures that your API works correctly and meets the intended requirements. Beyond that, though, it has a big impact on your applications overall performance and quality. Here are some key reasons why API testing is so important: Ensuring Functionality The primary goal of API testing is to verify that the API functions as expected. This includes checking that each endpoint returns the correct responses, the data formats are accurate, and the logic is implemented correctly. By thoroughly testing the functionality, you can catch issues early and prevent bugs from reaching the production environment. Performance VerificationAPIs are expected to perform well under various conditions. Performance testing helps assess how your API handles different loads and stress levels. It ensures that the API can manage a high number of requests efficiently and responds quickly, providing a smooth experience for the end-users. Security AssuranceAPIs often handle sensitive data, making security testing a critical aspect of API testing. You can make sure that your API is safe from threats like unauthorized access, data breaches, and other malicious attacks by testing for security vulnerabilities. protection laws. Reliability and Stability Reliability testing ensures that your API consistently performs well under different scenarios and does not fail unexpected inputs. By ensuring the reliability and stability and stability and stability and stability resting for various edge cases, error conditions, and unexpected inputs. users.Integration ValidationAPIs often act as the glue between different software components or systems. Integration testing checks that these components or systems and that any dependencies or interactions are handled properly. By catching bugs and issues early in the development process, API testing helps reduce the cost of fixing problems later on. It is much cheaper and easier to address issues during the development cycle.Improved User ExperienceUltimately, API testing contributes to a better user experience. By ensuring that your API is functional, performant, secure, reliable, and well-integrated, you provide a high-quality service to your users. This leads to higher user satisfaction. Types of API testing, each serving a specific purpose in ensuring the overall quality and functionality of an API Heres an overview of the different types of API testing and what they aim to achieve:Unit TestingUnit testing checks that an applications smallest components, usually individual functions or methods, function as intended. For APIs, this means testing the individual functions or methods, function as intended. issues at an early stage in the development process.Example:// Example of a unit test for an API endpointpm.test("Status code is 200", function () { pm.response.to.have.status(200);});Integration TestingIntegration testing involves testing multiple components or services together to ensure they work as a whole. For APIs, this means checking how different endpoints interact with each other and with external services. It verifies that data is correctly passed between components and that they function () { var jsonData = pm.response.json() pm.expect(jsonData.authenticated).to.be.true;});End-to-end testing End-to-end testing simulates real user scenarios and validates the entire workflow of the application. For APIs, this means testing the user. It ensures that all parts of the application work together seamlessly.Example:// Example of an end-to-end testpm.test("Complete user workflow", function () { pm.sendRequest({ url: ", method: "POST", body: { mode: "raw", raw: JSON.stringify({ name: "John Doe" }), }, }, function () { pm.sendRequest({ url: ", method: "POST", body: { mode: "raw", raw: JSON.stringify({ name: "John Doe" }), }, }, } evaluates how well an API performs under different loads. It helps identify bottlenecks and ensures that the API can handle the expected load efficiently.Example of a performance testpm.test("Response time is less than 200ms", function () { pm.expect(pm.responseTime).to.be.below(200); });Load TestingAPI Load testing is a subset of performance testing that specifically focuses on how the API can handle before its performance starts to decrease. Contract testing API contract testing involves verifying that the API can handle before its performance starts to decrease. services. This includes checking the request and response formats, data types, and required fields. It ensures that any changes to the API do not break the agreed-upon contract.Example:// Example of a contract ("Response adheres to contract," function () { var jsonData = pm.response.json(); pm.expect(jsonData).to.have.property("id"); pm.expect(jsonData).to.have.property("name");});Benefits of API testing There are lots of advantages to API testing, which can significantly improve the overall qualityand efficiency of your software development process. Here are some key benefits of incorporating API testing into your workflow:Quality AssuranceAPI testing ensures that your APIs meet the required standards and function correctly. By testing various aspects, such as functionality, performance, and security, you can catch issues early and prevent them from impacting the end users. This thorough testing process helps maintain high-quality standards throughout the development lifecycle. Early Issue Detection and ResolutionBy integrating API testing early in the development process, you can identify and fix issues before they escalate. By reducing the need for debugging and troubleshooting later, this proactive strategy saves time and resources. Early detection also allows for faster iterations and smoother development cycles. Resource Conservation Automated API testing reduces the need for manual testing efforts, saving valuable time and resources. As a result, your staff may focus on more important tasks, like creating new features and enhancing current ones. Additionally, automated tests can run continuously, providing constant feedback and ensuring that your API remains stable and functional. Rapid IterationBy giving quick and reliablefeedback on code changes, API testing supports and upgrades may be released more quickly because of this quick feedback loop, which speeds up continuous integration and deployment. Improvec CollaborationAPI testing promotes better collaboration between development, testing, and operations
teams. Together, the team members may more efficiently discover and results. This collaborative approach fosters a culture of quality and accountability, leading to a more cohesive and productive development process. Enhanced Security An essential component of API testing your APIs for vulnerabilities and making sure they are safe from any attacks. Regular security testing helps identify and mitigate risks, ensuring that your API complies with industry standards and regulations. API testing provides comprehensive coverage of various aspects of your API, including functionality, performance, and security. By testing different users. This comprehensive approach helps deliver a robust and reliable API.Ultimately, the goal of API testing is to provide a seamless and enjoyable user experience. By ensuring that your API works correctly, performs well, and is secure, you can deliver a high-quality product that meets user expectations. A well-tested API contributes to higher user satisfaction and retention. Challenges in the secure expectation and retention are expected as a seamless and enjoyable user expectations. A well-tested API contributes to higher user satisfaction and retention. Challenges in the secure expectation are expected as a seamless and enjoyable user expectation are expected as a seamless and enjoyable user expectation. API TestingWhile API testing offers numerous benefits, it also comes with its own set of challenges. You can implement better testing procedures and be more preparedby being aware of these challenges. Here are some common challenges faced during API testing and potential solutions to address them: Complexity of API EndpointsAPIs can have complex endpoints with various parameters, headers, and payloads. Testing all possible combinations and ensuring that every endpoint works correctly can be challenging. This complexity increases with the number of endpoints and the intricacies of the data they handle. Solution: Use comprehensive test cases and data-driven testing techniques to cover as many scenarios as possible. Tools like Postman allow you to create parameterized tests and use data files to automate the efficient testing of multiple combinations. Handling Asynchronous Processes can be difficult because it requires checking the APIs behavior over time and ensuring that asynchronous events are handled correctly. Solution: Implement test scripts that can wait for specific conditions before proceeding with the next steps in your test. Managing Dependencies APIs often depend on other services, databases, or external APIs. Testing and simulating these dependencies during testing can be challenging. Solution: Use mock servers and service virtualization to simulate dependent services. Postmans mock server feature allows you to create mock responses for your API endpoints, enabling you to test your API endpoints as pect of API testing, but it can be difficult to identify all potential vulnerabilities. injection attacks, data breaches, and unauthorized access, can affect APIs.Solution: Implement comprehensive security testing that includes authentication, authorizet access, can affect APIs are often updated with new features, changes to existing endpoints, or bug fixes. Keeping your tests up-to-date with these changes can be challenging, especially in a fast-paced development environment. Solution: Integrate your API testing with your CI/CD pipeline to ensure tests are automatically updated and run with every code change. Postmans integration with CI/CD tools allows you to automate test execution and ensure your tests are always in sync with your API testing by following the best practices. These practices help ensure that your tests are thorough, reliable, and maintainable, leading to higher quality APIs and a smoother development process. Here are some key best practices for API testing: Create a Dedicated testing environment helps you test your APIs without affecting the production system. This testing environment helps you test your APIs without affecting the production system. same configurations, databases, and network conditions. Automate Your API Tests Automating API tests allows you to run them frequently and consistently, catching issues early and ensuring that new changes do not introduce regressions. Use tools like Postman to create and automate your tests. pipeline to run with every code change, providing immediate feedback. Run Tests Throughout the API Lifecycle Testing should not be a one-time activity. Run your tests throughout the API lifecycle, from development and staging to production. Continuous testing helps catch issues at every stage and ensures that your API remains reliable and functional over time. Creating reusable subtests for common test scenarios can save time and effort. For example, if multiple endpoints require authentication, write a reusable test for the authentication process. This practice reduces redundancy and makes it easier to maintain your test suite. Document API Requests Documenting your API requests helps you and your team understand the purpose and functionality of each test. Postman provides excellent documentation. This documentation and collaboration. Practice API Security TestingIncludes, improving communication and collaboration. security testing as part of your API testing strategy. Test for common security issues like SQL injection, cross-site scripting (XSS), and unauthorized access. Ensure that your API handles authentication and authorization correctly. Postman can help automate these security tests and integrate them into your regular testing process. API Load testing evaluates how your API performs under heavy traffic. It helps identify bottlenecks and ensure that your API can handle the expected load. Use tools like Postman is a tool for developing and testing APIs. Its features simplify the process and make it more efficient for various team members involved in software development. Here are the key features of Postman: User-Friendly Interface. Collections: Group related API requests into collections: Group related API requests easily with a visual interface. variables for different environments like development, staging, and production to streamline testing. Automation with Newman: Run Postman collections from the command line, which is useful for integrating with CI/CD pipelines. Mock Servers: Create mock responses to simulate APIs for testing purposes. Monitoring: Schedule and run API tests regularly, get alerts on failures, and ensure API performance. Documentation: Generate and share detailed API documentation directly from your Postman is a popular choice for API testing? Postman is a popular choice for API testing? and manage API requests. You can quickly set up new requests, add parameters, headers, and body data, and send them to see the responses. Because of its simplicity, even people who are not familiar with API testing: With Postman, you can write test scripts in JavaScript to automate your API tests. This saves time and ensures consistency. Collaboration: Postman allows teams to share collections, environments, and test results, fostering better collaboration: with environment variables.Mock Servers: Create mock servers to simulate API responses and test your applications without needing the actual backend.Continuously tested.How to Use Postman to Test APIs Testing APIs with Postman is straightforward, thanks to its user-friendly interface and robust features. Heres a step-by-step guide on how to get started: Step 1: Sign Up for a Postman if you havent already. You can sign up for a free account, which allows you to save your requests and collections in the cloud. Step 2: Create a New RequestOpen Postman and click the New button. Select your request protocol to create a new API request (We will use HTTP for this guide). Name your request and choose the HTTP method (GET, POST, PUT, DELETE, etc.) from the dropdown. Enter the URL for your API endpoint. If needed, add parameters, headers, and body data. You can do this in the respective tabs under the URL field. Step 4: Send the Request Click the Send button to send your API request to ensure the API request was successful (e.g., 200 OK). Review the response body to verify the returned data. Check the headers for additional information about the response. Step 6: Create a new collection. Name your collection and add a description if needed.Step 7: Add Requests to the Collection Save your individual requests into the appropriate collection Runner by clicking on the Runner button in the bottom-right corner. Select the collection you want to run and configure any necessary settings (e.g., environment, iterations). Click Schedule Run to execute the requests. Postman will display the results for each request in the collection. Postman Automation: How to Do It? Automating your API tests with Postman can save time and ensure consistency in your testing process. Heres a step-by-step guide on how to set up automated tests in Postman. Step 1: Select the Scripts TabWithin your API request in Postman, navigate to the Scripts TabWithin your API request in Postman. Step 2: Write Test ScriptsUse JavaScript along with Postmans pm object to write your tests. Common assertions include checking status codes, response times, and response times, and response times into a collection. Open the Collection Save your requests with tests into a collection your set. want to run.Configure any necessary settings, such as environment and number of iterations. Click Schedule Run to execute the tests. The Collections from the command line. This allows you to integrate your tests into CI/CD pipelines. Step 5: Monitor Test ResultsPostmans monitoring feature allows you to schedule and run tests at regular intervals. Set up monitors to run specific collections and receive alerts on test failures. This helps ensure your API remains functional and performs well over time. Incorporating Postman into Your
CI/CD Pipelines. This helps ensure your API remains functional and performs well over time. Incorporating Postman into Your CI/CD Pipelines. This helps ensure your API remains functional and performs well over time. Incorporating Postman into Your CI/CD Pipelines. This helps ensure your API remains functional and performs well over time. Incorporating Postman into Your CI/CD Pipelines. This helps ensure your API remains functional and performs well over time. Incorporating Postman into Your CI/CD Pipelines. This helps ensure your API remains functional and performs well over time. Incorporating Postman into Your CI/CD Pipelines. This helps ensure your API remains functional and performs well over time. Incorporating Postman into Your CI/CD Pipelines. This helps ensure your API remains functional and performs well over time. Incorporating Postman into Your CI/CD Pipelines. This helps ensure your API remains functional and performs well over time. Incorporating Postman into Your CI/CD Pipelines. This helps ensure your API remains functional and performs well over time. Incorporating Postman into Your CI/CD Pipelines. This helps ensure your API remains functional and performs well over time. Incorporating Postman into Your CI/CD Pipelines. This helps ensure your API remains functional and performs well over time. Incorporating Postman into Your CI/CD Pipelines. This helps ensure your API remains functional and performs well over time. Incorporating Postman into Your CI/CD Pipelines. This helps ensure your API remains functional and performs well over time. Incorporating Postman into Your CI/CD Pipelines. This helps ensure your API remains functional and performs well over time. Incorporating Postman into Your CI/CD Pipeli (Continuous Integration/Continuous Deployment) pipeline allows you to automate your API tests, ensuring that they run with every build or deployment. This continuous testing approach helps catch issues early and maintain the reliability of your APIs. Heres how you can incorporate Postman into your CI/CD pipeline:Step 1: Set Up Your CollectionEnsure that your API requests and tests are organized into collections within Postman. Collection Runner allows you to execute a series of requests in a collection. You can access the Collection Runner by clicking the Runner button in the bottom-right corner of the Postman app. Configure your collection to include all necessary requests and tests. Step 3: Automate with Postman CLI for running collections from the terminal. This is useful for integrating with CI/CD tools. Install Postman CLI: First, download and install the Postman CLI.Authenticate: Use the provided command to log in with your API key:postman login --with-api-key Run the Collection: use the following command to run your collection: use the following command to run your collection: use the following command to log in with your API key:postman login --with-api-key Run the Collection: use the following command to run your co shell commands as part of the build process. You can add the Postman CLI commands to your pipeline configuration to run your API tests automatically. Step 5: Monitor Test ResultsPostmans monitoring feature allows you to schedule and run tests at regular intervals, independent of your CI/CD pipeline. Set up monitors to run specific collections and receive alerts on test failures. This helps ensure your API remains functional and performs well over time. Postman Best Practices Following bestpractices to enhance your use of Postman: Organize Collections and EnvironmentsUse Collections: Group related requests into collections, organize requests into folders, and requests easy to identify, use consistent and obvious naming conventions. Use Environment Variables: Create environment variables for items like base URLs, API keys, and tokens. This allows you to switch environments (development, staging, production) without changing the actual requests. Pre-request scripts to set up environment variables dynamically before a request is sent.Write Test ScriptsReusable Tests: Write reusable test scripts that can be included in multiple requests. Assertions: Use robust assertions to check various aspects of the response, such as status codes, headers, and body content. Version Control Version Collections: Use version control systems (e.g., Git) to track changes to your Postman collections and share them with your team. Postman API to programmatically manage your collections and environments, integrating them with your version control workflows. Documentation for your API collections and environments integrating them with your team. This makes it easier for team members and external users to understand and use your APIs. Annotations: Add detailed descriptions and comments to your requests and test scripts for better clarity. Practice API Security TestingAuthorization: Test various authorization mechanisms (e.g., OAuth, API keys) to ensure they are implemented correctly.Vulnerability Testing: Check for common security vulnerabilities like SQL injection, XSS, and improper error handling.Simulate Load: Use tools like Postman and Newman to simulate high volumes of requests and optimize your APIs performance.Postman Monitors and Their BenefitsPostman Monitors allow you to automate the execution of your API tests at scheduled intervals. This feature is essential for continuous monitoring of your API tests at scheduled intervals. Postman Monitors WorkSet Up a Monitor. Go to the Postman app and select the collection you want to monitor. Click on the Monitor stab and then Create a Monitor. Click on the Monitor by setting the schedule (e.g., every hour, daily) and selecting the environment you want to use. Schedule Runs: Define the frequency of the test runs, such as hourly, daily, or weekly.Postman will automatically execute the tests at the specified intervals.Monitor Execution:Monitors execute your predefined tests, simulating real-world usage of your API. They check for performance, functionality, and reliability issues. Receive Alerts: Set up alerts to notify you if any tests fail. You can receive alerts via email. Detailed reports are generated for each run, showing the status of each test and any errors encountered. Benefits of Using Postman MonitorsContinuous testing without manual intervention, freeing up resources for other tasks. Performance testing without manual intervention, freeing up resources for other tasks. MonitoringResponse Times: Monitors track response times to identify performance degradation. Scalability: Ensure your API can handle the expected load consistently. ReliabilityUptime Tracking: Monitors help ensure your API can handle the expected load consistently. Reliability Uptime Tracking: Monitors help ensure your API can handle the expected load consistently. Reliability Uptime Tracking: Monitors help ensure your API can handle the expected load consistently. Reliability Uptime Tracking: Monitors help ensure your API can handle the expected load consistently. Reliability Uptime Tracking: Monitors help ensure your API can handle the expected load consistently. Reliability Uptime Tracking: Monitors help ensure your API can handle the expected load consistently. Reliability Uptime Tracking: Monitors help ensure your API can handle the expected load consistently. Reliability Uptime Tracking: Monitors help ensure your API can handle the expected load consistently. Reliability Uptime Tracking: Monitors help ensure your API can handle the expected load consistently. Reliability Uptime Tracking: Monitors help ensure your API can handle the expected load consistently. Reliability Uptime Tracking: Monitors help ensure your API can handle the expected load consistently. Reliability Uptime Tracking: Monitors help ensure your API can handle the expected load consistently. Reliability Uptime Tracking: Monitors help ensure your API can handle the expected load consistently. Reliability Uptime Tracking: Monitors help ensure your API can handle the expected load consistently. Reliability Uptime Tracking: Monitors help ensure your API can handle the expected load consistently. Reliability Uptime Tracking: Monitors help ensure your API can handle the expected load consistently. Reliability Uptime Tracking: Monitors help ensure your API can handle the expected load consistently. Reliability Uptime Tracking: Monitors help ensure your API can handle the expected load consistent well approximately. Reliability Uptime Tracking: Checks: Monitors can run security tests at regular intervals to ensure ongoing protection. Compliance: Ensure your API continues to meet security standards and regulations. Detailed Reports: Each monitor run generates a detailed report showing the results of each test. Trends and Insights: Analyze trends over time to identify patterns and potential issues. Alerts and Notifications Real-Time Alerts: Receive immediate notifications: Configure alerts to suit your needs, choosing how and where you receive them. Setting Up a Monitor: Create a Monitor: Create a Monitor: Create a Monitor: Create a Monitor Example of setting up a Postman Monitor: Create a Monitor: Configure alerts to suit your needs, choosing how and where you receive them. Setting up a Monitor Example of setting up a Postman Monitor: Create a Monitor Example of setting up a Postman Monitor: Create a the Monitors tab, and select Create a Monitor.Name your monitor and choose the environment (e.g., development, production).Configure email notifications for test failures.Review and Save:Review your settings and save the monitor.Writing Tests in PostmanThe Scripts tab has been merged into the Scripts tab has been merged into the Scripts tab.Best Practices, you can make your tests better and more understandable so that your tests become more robust and reliable. Lets check out some of these best practices: 1. Descriptive and Clear Test NamesWhy Its Important: Clear names make it easier to understand what each test
checks at a glance, especially when reviewing test results or sharing tests with colleagues. Example: Instead of naming a test Test 1 or Status Check, use descriptive names like Verify Status Code is 200 for User Endpoint or Ensure Response Time is Below 500ms.2. Testing One Concern Per TestWhy Its Important: Focusing on one assertion per test simplifies troubleshooting and understanding test results. If a test fails, you know just what went wrong. Example: Separate them instead of combining status code and response time checks in one test:// Test for status code is 200", function () { pm.response.to.have.status(200); }); // Test for response time pm.test("Response time pm. them more readable and intention-driven. It clarifies the purpose of the tests. Example: Use assertive phrases like expect(response).to.contain... or response.should.have..., clearly stating the tests expectations.4. Organize Tests LogicallyWhy Its Important: Grouping related tests or organizing them logically can make your testing suite more understandable and maintainable. Example: If testing various aspects of a user API, group tests related to user creation, user data retrieval, and user deletion together. 5. Handle Different Test Scenarios why Its Important: Testing only the happy path can leave critical bugs in edge cases. Its essential to test various scenarios, including potential error conditions.Example: Alongside testing a successful API call, write tests for scenarios like invalid inputs, unauthorized access, or server errors.// Test for invalid inputs for invalid inputs, unauthorized access, or server errors.// Test for invalid inputs and Reusability and Reusabil and reusable for different scenarios. This practice saves time and effort in the long run. Example: Create reusable functions for common test assertions. Call these functions with different parameters as needed rather than writing the same code in multiple tests. 7. Commenting and Documentation Why Its Important: Good comments and documentation make it easier for others (and your future self) to understand the purpose and mechanics of your tests. Example: Add comments to clarify complex logic or the reason behind specific test cases, especially when testing less obvious or intricate API parts. Using expect for AssertionsIntroduction to Chai LibraryChai is an assertion library used in JavaScript for test-driven development (TDD) and behaviour-driven development (BDD). In API testing using Postman, Chai offers a set of assertions for validating the API requests and responses by ensuring they meet the expected criteria. Purpose of Assertion Libraries and responses by ensuring they meet the expected criteria. Purpose of Assertion Libraries and responses by ensuring they meet the expected criteria. Purpose of Assertion Libraries and responses by ensuring they meet the expected criteria. Purpose of Assertion Libraries and responses by ensuring they meet the expected criteria. Purpose of Assertion Libraries and responses by ensuring they meet the expected criteria. Purpose of Assertion Libraries and responses by ensuring they meet the expected criteria. Purpose of Assertion Libraries and responses by ensuring they meet the expected criteria. Purpose of Assertion Libraries and responses by ensuring they meet the expected criteria. Purpose of Assertion Libraries and responses by ensuring they meet the expected criteria. Purpose of Assertion Libraries and responses by ensuring they meet the expected criteria. Purpose of Assertion Libraries and responses by ensuring they meet the expected criteria. Purpose of Assertion Libraries and responses by ensuring they meet the expected criteria. Purpose of Assertion Libraries and response and the expected criteria. Purpose of Assertion Libraries systematic way to check whether the output of a code block (or, in this case, an API response) matches the expected result. Readability: Chais syntax is designed to be human-readable, making tests covering various aspects of the API response.Using expect in PostmanWithin Postman, expect statements allow you to perform detailed checks on your response.json()).to.be.an("object");}); Status Code: Ensure your API returns the correct status code, indicating the requests success or failure.Response Body: Validate the structure and data of the response body to ensure your API returns the expected data.Response Time: Ensuring your API response Body: Validate the structure and data of the response in Postman for API Testingpm.response is an important object in Postman scripting that gives you much information about the response correctly and effectively, you can improve your API testing because this object allows you to access and validate various aspects of the response data. Here a more detailed look at utilizing pm.response in your tests: Accessing Response contains several properties and methods that give you access to different parts of the API response to verify if the API request was successful.let statusCode = pm.response.code; pm.expect(statusCode).to.eql(200); Response Time: Check how long the API took to response.code; pm.expect(statusCode); // time in millisecondsHeaders: Examine the response headers for important metadata like content type, caching policies, and more.let contentTypeHeader = pm.response.headers.get("Content-Type");pm.expect(contentTypeHeader).to.include("application/json");Body: The response body contains the data returned by the API. You can parse this data and make assertions based on your APIs expected output.let responseBody = pm.response.json(); // For JSON responseBody).to.have.property("name", "John Doe"); Using pm.response for Complex validations: Valida structure. Conditional Testing: Perform different tests based on certain response conditions. For example, if the status code is 200, check another. Dynamic data (like timestamps or unique IDs). Use pm. response to validate the format of these dynamic elements without hardcoding the values. Best Practices with pm.response Readability: Keep your tests readable and straightforward. Complex logic can make tests more complicated to understand and maintain. Error Handling: Include error handling: Include error handling: Include error handling in your tests. consistent in how you use pm.response across different tests. This consistency helps in maintaining and scaling your test suite.Validation ExamplesValidating Response Status code", () => { // change 200 to the response code you expect const expected StatusCode = 200; pm.response.to.have.status(expectedStatusCode);});Validate multiple status codes:// change 200 or 201", function () { pm.expect(pm.response.code).to.be.oneOf([200, 201]);});Validating Response timepm.test("Response time is less than 500ms", function () { pm.expect(pm.response.to.have.header("Content-Type is application/json"); }); Validating Response BodyPostman test to check field value in response.to.have.header("Content-Type is application/json"); }); Validating Response BodyPostman test to check field value in response.to.have.header("Content-Type is application/json"); }); Validating Response BodyPostman test to check field value in response.to.have.header("Content-Type is application/json"); }); Validating Response BodyPostman test to check field value in response.to.have.header("Content-Type is application/json"); }); Validating Response.to.have.header("Content-Type is application/json"); }); Validating Response BodyPostman test to check field value in response.to.have.header("Content-Type is application/json"); }); Validating Response test below.pm.test("API response contains the expected fields", () => { const response.json(); // the line below checks value of the id field is 1 (number). pm.expect(response).to.have.property("id", 1); // the line below checks value of the id field is 1 (number). pm.expect(response).to.have.property("id", 1); // the line below checks value of the id field is 1 (number). pm.expect(response).to.have.property("id", 1); // the line below checks value of the id field is 1 (number). pm.expect(response).to.have.property("id", 1); // the line below checks value of the id field is 1 (number). pm.expect(response).to.have.property("id", 1); // the line below checks value of the id field is 1 (number). pm.expect(response).to.have.property("id", 1); // the line below checks value of the id field is 1 (number). pm.expect(response).to.have.property("id", 1); // the line below checks value of the id field is 1 (number). pm.expect(response).to.have.property("id", 1); // the line below checks value of the id field is 1 (number). pm.expect(response).to.have.property("id", 1); // the line below checks value of the id field is 1 (number). pm.expect(response).to.have.property("id", 1); // the line below checks value of the id field is 1 (number). pm.expect(response).to.have.property("id", 1); // the line below checks value of the id field is 1 (number). pm.expect(response).to.have.property("id", 1); // the line below checks value of the id field is 1 (number). pm.expect(response).to.have.property("id", 1); // the line below checks value of the id field is 1 (number). pm.expect(response).to.have.property("id", 1); // the line below checks value of the id field is 1 (number).pm.expect(response).to.have.property("id", 1); // the line below checks value of the id field is 1 (number).pm.expect(response).to.have.property("id", 1); // the line below checks value of the id field is 1 (number).pm.expect(response).to.have.property("id", 1); // the line below checks value of the id field is 1 (number).pm.expect(response).to.have.property("id Sanchez"); }); Test if Response Body matches schemaTesting if the response body matches a specific schemapm.test ("Body matches schema" }, name: { type: "string" }, status: { type: "string" }, species: { type: "string" }, type: { type: "string" }, type: { type: "string" }, gender: { type: "string" }, gender: { type: "string" }, type: { type: "string" }, type: {
type: "string" }, type: { type: "string" }, gender: { type origin: { type: "object", properties: { name: { type: "string" }, y, required: ["name", "url"], // Added required property for origin }, location: { type: "string" }, y, required: ["name", "url"], // Added required property for location }, image: { type: "string" }, episode: { type: "string" }, y, required: ["name", "url"], // Added required property for location }, image: { type: "string" }, episode: { type: "string" }, y, required: ["name", "url"], // Added required property for location }, image: { type: "string" }, episode: { ty "array", items: { type: "string" }, }, url: { type: "string" }, }; reated: { type: "string" }, }; required: ["id", "name", "status", "species", "type", "gender", "origin", "location", "image", "episode", "url", "created",], }; pm.expect(pm.response.json()).to.be.jsonSchema(schema);}); Test if nested field value is available in responseThe script below step works for fields at the root of the response. What if we wanted to test the name field under the origin field. We can tweak the script to support fields, () => { const response = pm.response.json(); // the line below checks value of the id field is 1 (number). pm.expect(response).to.have.nested.property("id", 1); // the line below checks value of the origin.name", "Rick Sanchez"); // the line below checks value of the origin.name", "Earth (C-137) (string). pm.expect(response).to.have.nested.property("origin.name", "Earth (C-137) (string).pm.expect(response).to.have.nested.property("atting).pm.expect(response).to.have.nested.pm.expect(response).to.have.nested.pm.expect(response).to.have.nested.pm.expect(response).to.have.nested.pm.expect(response).to.have.nested.pm.expect(response).to.have.nested.pm.expect(response).to.have.nested.pm.expect(response).to.have.nested 137)");});Check nested array value in response contains the same technique to validate the value of items in the array. For example, we can use the same technique to validate the value of the second item in the episode array of the endpoint.pm.test("API response contains the expected fields", () => { const response = pm.response.json(); // the line below checks the value of the episode field at index 0 is ". pm.expect(response).to.have.nested.property("episode.0", ");});No code API testing & monitoring tool and a great Postman alternative for API testing. The below video is a quick demo of TestfullyTestfully and how you can use it to test your APIs without writing code. ConclusionAs you saw, you can significantly improve your API testing process by using JavaScript in your Postman workflow. The examples and practices that we went through can help you develop comprehensive and reliable API tests. Try them and tailor them to fit your specific testing needs. Happy testing! Frequently Asked Questions We got an answer for your questions What is API testing? API testing is a type of software testing that focuses on verifying that APIs function as expected. It involves sending requests to API endpoints and analyzing the responses to ensure they meet the expected outcomes, covering aspects like functionality, performance, and security. Why is API testing important? API testing is essential because it ensures that your APIs work correctly and meet the required standards. It helps catch issues early in the development process, improves quality assurance, enhances security, and ensures the performance and reliability of APIs. How do you use Postman for API testing? To use Postman for API testing, create and manage API requests, organize them into collections, use environment variables for different settings, write test scripts in the "Tests" tab, and automate the tests using Postman CLI or integrate them into CI/CD pipelines. What are Postman Collections? Postman Collections are groups of related API requests that you can save and organize together. Collections help manage and run tests efficiently, keeping your API tests in Postman by writing test scripts in JavaScript, using the Postman Collections, and integrating Postman CLI with CI/CD tools like Jenkins, GitLab CI, or CircleCI to run tests automatically with each build or deployment. What is the Postman CLI? The Postman CLI is a command-line tool that allows you to run Postman CLI with each build or deployment. process. Can Postman be used for performance testing? While Postman can be used for basic performance testing are recommended for extensive performance testing. For more detailed performance testing, tools like Apache JMeter or Gatling are recommended What are Postman Monitors? Postman Monitors are a feature that allows you to schedule and run API tests at regular intervals. Monitors help ensure your APIs are always functional and performant by providing continuous testing and alerts for any test failures. you to define and manage variables for different environments, such as development, staging, and production. This makes it easier to switch settings without changing the actual requests, ensuring flexibility and efficiency in testing. What are some best practices for API testing with Postman? Best practices for API testing with Postman include organizing requests into collections, using environment variables, writing clear and reusable test scripts, automating tests, integrating with CI/CD pipelines, and regularly monitoring API performance using Postman Monitors. Design, secure, and ship higher-quality APIsall in one platform.Learn more Prototype, document, test, and demo all your APIs in one place. Get early feedback by having conversations in the context of any API internal, public, or partnernot scattered across tools. Transform API development from an individual to a team sport. Get to that first API call faster, improve development from an individual to a team sport. expedite self-serve API consumption across the org, so devs can start testing APIs and building workflows. Organize API collections in to workspaces where API consistency across teams, and strong partner collaboration. Move fast to build quality APIs without breaking anythingby managing every phase of the API workflow, together, on a single platform. Avoid building from scratch or writing lengthy descriptions about API issues by storing and sharing APIs in Collections. Seamlessly update, edit, deprecate, and communicate changes on APIs so even the simplest API change doesn't bring havoc to your workflow. Use AI in Postman to write test scripts from scratch, generate tests for an entire collection, or visualize API responses with graphs to better understand API output.

Salesforce rest api postman example. Postman api key authentication example. Dummy api for testing in postman example. Rest api upload file example postman. Postman example api test. Api chaining in

postman with example. Postman automation api testing example. Postman example api post. Postman api collection example. Postman jira api example. Postman mock api example. Postman tutorial for beginners with api testing example. Postman rest api example. Splunk rest api search postman example. Postman api documentation example.