POSTMAN

Automated Testing

Understanding, designing, and setting up an effective automated testing strategy

Table of Contents

Abstract 02
Introduction 03
Background 04
Challenges & Solutions 05
Extended Example 09
Conclusion 16
References 17

Abstract

Progress in automation is causing a shift left in testing behavior. Shifting testing to earlier and often in the software lifecycle is a direct response to the lag time that now exists between development and testing. Developers and engineers are using automated tests to bridge this gap, reduce test development time, and ultimately save resources.

Transitioning from manual to automated testing requires a shift in strategy and in resources. However, the trajectory of testing is overwhelmingly leaning toward automation **(Bhamaret, Lalitkumar, and Motvelisky, Joel)**. Increased availability of testing tools has allowed developers and engineers to speed-up testing cycles, scale-up, and improve program quality. It is easier than before to leverage the benefits of automated testing while continuing necessary manual testing at an overall lower cost.

This white paper addresses the background of manual and automated testing, discusses the pros and cons of test automation, and highlights a few important considerations for implementing automated testing. This paper also includes an extended example of consumer-driven contract testing (CDC testing). This example will dive into how automated testing is beneficial for software built with a microservices architecture.

As automated testing continues to shape software development, more and more companies are investing in test automation strategies. Automated testing is equipping developers with the tools to make higher quality, extensible products with longer shelf-lives. Implementing automated testing provides a significant advantage to software developers.

Introduction

Today, top companies leverage automated testing to increase product longevity, reduce costly and repetitive build-out, and improve iteration quality. This whitepaper will provide a brief introduction to automated testing. It will also address the benefits and limitations of automated testing and give an in-depth example of consumer-driven contract testing.

What is Manual Testing?

Manual testing is generally executed by an individual or group of individuals who are trained in QA testing. The goal of manual testing, like automated testing, is to find errors in code, potential bugs, and to ensure performance. Any test can be manual, but manual testing takes more time and money than automated testing long term. Manual testing generally decreases return on investment (ROI) because it requires replicating investments on the same tests over and over again. Automated testing does not cause this issue because test suites can be saved, duplicated, and reused. The benefit of manual testing is that it allows testers to execute certain types of tests that are either impossible or very difficult to automate. Some common manual tests include ad hoc testing, exploratory testing, and user interface testing **(K. Elena, and Z. Maryana)**.

What is Automated Testing?

Automated tests are not run by humans (as you might have already guessed). They are, however, set up by humans. Tests are automated by creating test suites that can run again and again. Generally, these test suites are made up of a collection of individual tests with specific boundaries. Each individual test can deliver one piece of information, but when run together, will give you an overview of successful and failed tests. Many types of tests can be automated, including unit tests, integration tests, end-to-end tests, regression tests, and mock tests. The process of automated testing is more efficient in the long run, but can take time to strategize and set up.

Automated tests suites focus on repeatable processes with known results. For example A+B=C should always be the case when the variable is D. Using test automation to identify known risks decreases time spent on monotonous tests and frees up time for developers to perform other manual tests like exploratory testing. Exploratory testing is designed to identify and prevent potential risks rather than to confirm or deny known risks. Your test suites can be easily maintained if you include exploratory testing and other manual tests in with your automated workflow. This allows you to not only test for the expected, but keep a sharp eye on the unexpected. In this way, implementing automated tests extends all of your testing capabilities - manual and automated.

MANUAL TESTING WORKFLOWS

The process of manual testing is involved and costly for all types of developers and engineers. Some typical manual testing workflows are listed below.

- **Developers:** Using requests and history, developers continuously query the service they are building. Then, they must watch each response to verify that the expected data is being received.
- **Test Engineers:** With a service URL and product documentation, test engineers must manually test each program capability individually. They must update the product documentation by hand as they find and fix errors.
- **Systems Engineers:** Systems engineers query multiple integration points while building an application delivery pipeline. This means they must manually query multiple integration points by hand by following several documentation sources. This process becomes more complex as the number of services with which they integrate increases.

TEST SUITES

Test Suites are made up of test cases. Test cases are individual validations for some software function. For example, one test case might to ensure that your API is pulling the correct data from your database.

Test Suites organize groups of test cases into categories. Generally, each function of a program has many commands that achieve the function. It is a good practice to group test cases in categories. This allows you to group, save, and re-run common functionality tests whenever you need to validate a certain function. Test suites typically run in a series, one after the other, so they become the basis for many automated tests.

Background

Not long ago, most tests were performed locally (on someone's laptop). This meant that they would replicate an ideal environment and run a set number of known requests on an entirely complete product (known as an end-to-end test). This was common practice because software wasn't necessarily relying on micro-services or independent units of an entire program **(Sridharan, Cindy)**.

A more granular philosophy for testing is now the norm. Microservices must be tested for functionality independently and with other services. One of the main reasons for using microservices is that they can run independently, so one part breaking does not necessarily mean that the whole program will break. A granular testing philosophy helps ensure that microservices work independently before putting the whole system together. This philosophy is also the foundation for consumer-driven contract tests, which will be addressed in the extended example.

If you are working in software development, you may be aware of some of the different forms of testing that take place during the development process (and in post-production). Unit testing, integration testing, and end-to-end testing are the most common types of automated software tests, and they each serve different purposes **(Colantonio, Joe)**.

Unit Testing

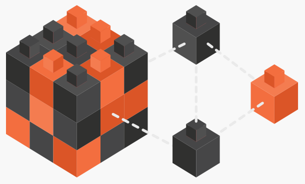
Just as the name implies, unit tests deal explicitly with testing the components that make up an entire program. These units are fragments of a larger ecosystem that need to be tested in isolation for code errors -- before adding other dependencies. Unit tests verify that the smallest pieces of a program are functioning properly. For example, when testing APIs, unit testing generally refers to verifying that a single HTTP request is delivering the expected results. Unit tests are valuable because they are relatively simple to build and run. They find bugs that would be more difficult to find once the fragments are added to the rest of a program. You can think of unit tests as the first step in a full testing strategy.

Integration Testing

Integration tests deal with two or more units at once. This ensures that the units of your program are compatible together and functioning properly with dependencies. Dependencies can either exist in your own software or in third-party tools like an API access key or login credentials. Building an integration test is more complex than building a unit test because it involves more moving parts. The benefit of integration testing is that it allows you to take a systematic approach to testing units with other

MICROSERVICES

Microservices are small, independently working units that can be stacked to create a more complex program. The trend of using microservices is growing because they provide more flexibility, improve stability, and allow for greater scalability.



Above, a network of connected microservices makes one larger product.

Learn more about Microservices in our blog post »

EXAMPLE

Let's say you are testing a program that sends you a Slack message with a list of trending Twitter topics in a specific location.

In order to unit test, you will need to test each part of the program. You might begin by testing the Twitter API with the goal of ensuring that it will send your request for trending topics and return the correct information. Next, you might do an integration test with the Twitter API and the Slack API to make sure that the Twitter API sends a list of trends to your Slack address when requested. Lastly, after making sure that your unit tests and integration tests passed, you might test the whole program with an end-to-end test to see if any unexpected bugs come up. This could include testing different variables like location, different Slack accounts, and different API keys, etc.

units. If you test everything at once, it can be difficult to isolate issues. By testing a few units at a time, you can easily identify the pieces that aren't fitting together properly.

End-to-end Testing

End-to-end tests target a complete ecosystem (that is, your newly developed software and all of its dependencies). End-to-end testing ensures that each use case for your software delivers the expected results. In addition, these tests will verify that your software works with different variables, networks, or operating systems, etc. These are complex tests to create and run, but are important to the process of releasing new software.

Automated Testing Challenges and Solutions

Despite the many benefits of implementing an automated testing framework, companies and individuals struggle to transition from manual to automated testing. These difficulties vary from product to product, and depend on what goals you have for testing. Below are a few of the most common questions and some suggestions for answering them.

Which tests do I automate?

Not all tests can or should be automated. Investing in an automated test takes resources, so it is important to be selective when deciding where to spend those resources.

Test Frequency

Tests that are run frequently are great candidates for automation, as long as they can be automated. Each time you run the same workflow, you are duplicating an investment unnecessarily and lowering your ROI. On the other hand, tests that are only run periodically do not necessarily need to be automated. Especially when you are starting out with an automated testing practice, these infrequent tests should not be the first priority.

Number of Configurations to Test

Some programs have multiple possible configurations like different networks, operating systems, or device types. In manual testing, the same test would be performed for each different configuration. With automated testing, you can significantly reduce the time spent on testing by setting up a test to run against multiple configurations one after the other. However, a program with just one or two configurations would not necessarily be worth testing automatically.

Quality Gate Testing
Running certain tests on every build is a great quality measure, especially when releasing changes on high-traffic programs. Typically the rule is: any test that is repeated frequently is worth automating. Many companies use automated testing to ensure that tests like smoke tests, sanity tests, and regression tests are performed before any major release.

Amount of Data
Inputting large amounts of data takes a long time. If you have a test that requires inputting a ton of data, you can save your QA or DevOps engineer (or yourself) the trouble and time by automating these types of tests.

Length of Test
Some tests can be very easy to perform, but just take a long time to get through. These types of tests might be worth automating so you can run them during off hours, without needing someone to actively initiative and monitor the test.

These characteristics can help you figure out whether or not to automate a test and can also help you prioritize your tests. If you are new to test automation, it may be best to start by automating a few tests and then move onto the next highest priorities.

What tool(s) should I use?

Not all tools are created equal. You can find many different tools by simply searching, "automated testing." However, some tools are specialized for one or a few types of automated tests. Not every tool will give you the full testing capabilities that you are looking for.

When you are looking for the best tool for your product, consider the following:

Compatibility
One of the first things to consider is whether or not a testing tool supports the coding languages in your product. If you have multiple products in multiple languages, ideally you can find a tool that supports a wide range of languages.

Product Characteristics
Your product may need to be tested in various browsers, on different device types, or with other third-party scripts. Your tooling choice will depend on your product's unique characteristics. To ensure that your requirements can be met by a certain tool, check the tooling documentation.

Integration Capability

You may need more than one tool to set up complete automated testing on your product. In that case, you will want to ensure that any tools you look into are capable of integrating with each other and also with any internal tools you plan on using.

Budget

This is a somewhat obvious qualification, but still very important. A basic budget for automated testing will include the cost of any paid tool, cost of employment for developers or QA engineers, and cost for future test fixes and additions. There are many free tools and free trials out there that can help you get the most out of your budget, so it is worth it to research your options before investing in a costly automated testing tool.

Testing community and resources

Resources and community support are invaluable tools for any testing engineer or developer. The better the resources and community support are, the more effective you can be with that particular testing tool. To gauge the quality of resources, you can review the documentation for clarity, check any support boards and third-party support threads (ie. on GitHub), and check the company website for how-to blog posts, tutorial videos, etc.

Is automated testing worth the start-up cost?

Cost is a common concern with individuals and companies alike. Costs can come from investing in new testing tools, time spent strategizing, or development time. Starting an automated testing practice can be expensive. You'll want to consider where you are as a company or with your product. Larger companies can easily absorb costs and justify long term benefits, but smaller companies, startups, and individuals are less likely to be able to.

The best way to justify new costs is to measure effectiveness on a small scale. A common way to measure automated test effectiveness is by calculating return on investment (ROI). When beginning a transition to automated testing, it is important to keep records of cost, time spent and saved, and number of employees using an automated testing strategy so you can calculate and visualize the effectiveness of your automated testing strategy. In general, when companies move from manual to automated testing, measure and optimize their strategy, automated testing saves money and time.

Testing takes about 50% of all development time (Damm et. al., 2005), so increasing the efficiency of your testing strategy can save you a significant amount of money.

Finding and fixing defects in software differs in cost depending on the stage of a program's development. According to testing expert Paul Grossman, finding and fixing a defect costs about five times more after a product release than it does to fix a defect during development (Paul Grossman, 2009). Automated testing can run repeatedly without taking extra developer time. By implementing automated testing, bugs and defects can be caught much earlier in development without taking extra time from developers.

How do I get started with automated testing?

We've considered which tests to automate, how to identify effective tools, and start up costs, but how do you actually begin planning an automated test?

Automated testing requires a shift in testing strategy and a thorough plan with enough flexibility and reliability to last over time. This is very different from the test-as-you-go philosophy of manual testing.

When setting up a successful test suite, consider the following:

Purpose: There can be many different goals to testing like to find bugs, to ensure usability with third party products (even if they break), to test the variables, etc.

Variables: Any program will have changing variables that may be dependant on some database or that may change based on user, etc. Variables are always an important piece of your program to test.

Limits: A test suite will usually have many tests that work together to deliver the information you need. Each piece of the test suite will need to be limited to delivering a specific piece of that knowledge. How will you define your testing blocks so they are as informative as possible?

Assumptions: If your program has any dependencies like an excel database or a third-party API, these should be recorded in a library so they can be tested and the library can be updated and reused in future testing.

Extensibility: Programs change over time (if they're any good), so any test suite you design should be able to adjust for changes in a program, in a library, or in dependencies.

Once you lay out the details of what you have in your program and what you expect to get out of your test, you are in a great position to begin building your test suite. While there are many things to consider, the time and resource investment is well worth it if you create successful, reusable, and flexible test suites.

Consumer-Driven Contract Testing with Postman

In this section, we'll address some of these missing links and see where Postman fits in. Postman has a broad range of automated testing capabilities. In our extended example, we'll go through the process of creating a consumer-driven contract test with Postman.

What is Consumer-Driven Contract Testing?

Consumer-driven contract testing occurs between two services that need to interact. The two parties are generally considered a consumer (like an API client) and a provider (like an authorization service). Contracts are written lists of dependencies between the consumer and the provider. Based on these lists of dependencies, providers create test suites to run at every new build. Testing at every build ensures product performance for consumers. Joyce Lin, a developer advocate at Postman, explained CDC testing clearly in her blog post, **Acing Your APIs: What You Need to Know for Test Automation**.

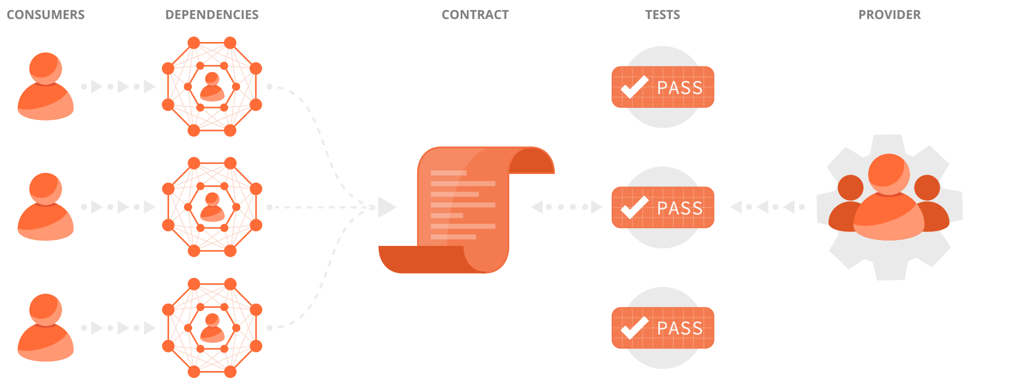
> "Consumer-Driven Contract Testing (CDC Testing) ensures that your service doesn't break when a service that you rely on is updated. This type of contract testing requires the consumer of the service to define the format of the expected responses, including any validation tests. The provider must pass these tests before releasing any updates to the service."

Implementing consumer-driven contract testing is a great way to maintain growing microservices stacks. It prevents API discrepancies and ensures that services properly function together. In addition to guiding development and testing, CDC testing helps side-step the need for large-scale and complex integration testing.

What are Contracts?

Documentation can be likened to a contract. Documentation typically lists available endpoints, request data, expected response data, etc. A developer might look at API documentation and create a product that depends on a certain command and response pattern. However, documentation changes without signaling potential issues to providers, unlike contracts.

A contract is a more explicit and formal way to communicate consumer dependencies to a provider. When a contract is created, tests are created alongside. These tests are created specifically to test consumer dependencies on provider services. This creates a simple workflow. If a provider makes a change in their service, then they must run and pass the contract tests before releasing any updates to their service. This ensures that consumer products do not break.



In CDC tests, consumers define dependencies, which make up the contract, and providers perform necessary tests to ensure that the contract is held up any time the service is updated.

Using Postman for Consumer-Driven Contract Testing

In a recent survey, Postman found that **52% of APIs are internal**. Internal APIs are generally used to create one larger, customer-facing product. With this kind of microservice architecture happening at such a large scale, it is imperative to have communication between microservices.

Postman has all the tools you need in place to start implementing contract testing in your organization.

We provide a collaborative platform that allows you to automatically share your Postman Collections and maintain a single source of truth with your other team members. Collections are executable specifications of an API. You can run a collection locally on your Postman app, on the command line, on CI systems using **Newman**, and via the cloud using **Monitors**. Requests in your collection are executed sequentially. These tools allow you to easily share and execute collaborative contract collections.

Now, let’s run through an example use case on how Postman’s features come together to implement consumer-driven contracts.

Use Case: Log Retrieval Service

Let’s say we need to build a hypothetical service that returns a list of log entries from a database. This service exposes one endpoint that returns latest five log entries with the name of the service that created the entry, a timestamp, and a description string.

The endpoint resides at `/api/v1/logs`. Sending a GET request to this endpoint should return JSON data in this structure:

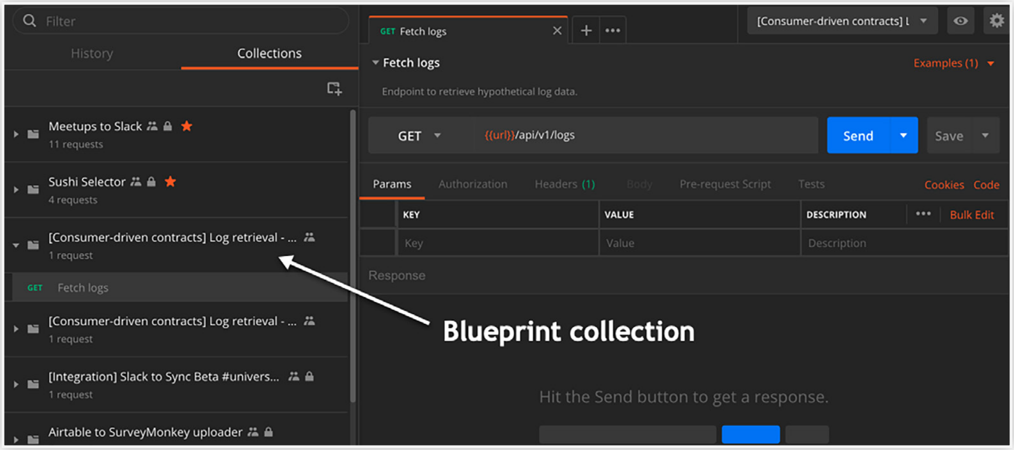
```
{
  “Count”: Number,
  “Entries”: Array[object]
}
```

The `entries` array contains an object for each log entry. Here’s what that should look like:

```
{
 “serviceName”: String,
 “timestamp”: Number,
 “description”: String
}
```

Blueprint Collection

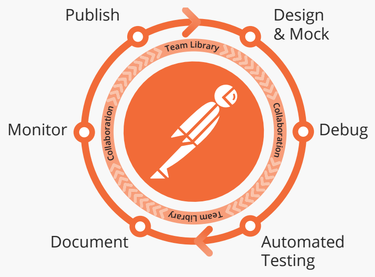
To begin, we need to create a blueprint collection. A blueprint collection lays out the API structure. This is created by the provider of the service.



Sample blueprint collection

WHAT IS POSTMAN?

Postman is the only complete API development platform. Postman’s built-in tools support every stage of the API lifecycle. You can design and mock, debug, automatically test, document, monitor, and publish your APIs -- all from one place.



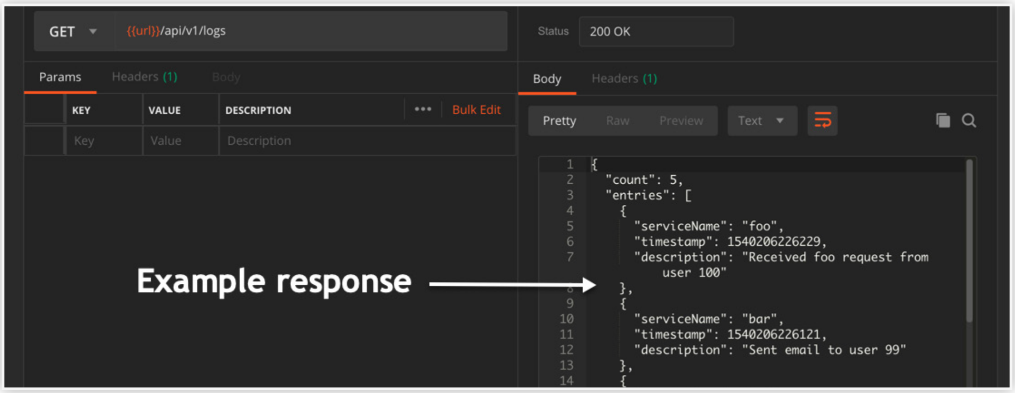
You can access Postman through **native apps** for MacOS, Windows, and Linux. Postman’s complete API development platform allows you to create requests, build and manage collections, and govern the lifecycle of your APIs.

Next, we need to add **examples for the request**. Examples allow such blueprint collections to describe response data. They show up in Postman's automatically generated documentation.

Here is an example of the response data for the default output of this service:

```
{
 “count”: 5,
 “entries”: [
   {
     “serviceName”: “foo”,
     “timestamp”: 1540206226229,
     “description”: “Received foo request from user 100”
   },
   {
     “serviceName”: “bar”,
     “timestamp”: 1540206226121,
     “description”: “Sent email to user 99”
   },
   {
     “serviceName”: “foo”,
     “timestamp”: 154020622502,
     “description”: “Received foo request from user 10”
   },
   {
     “serviceName”: “baz”,
     “timestamp”: 1540206223230,
     “description”: “Activated user 101”
   },
   {
     “serviceName”: “bar”,
     “timestamp”: 1540206222126,
     “description”: “Error sending email to user 10”
   }
 ]
}
```

Each example has a name and specific request path. Here is how it looks in the Postman app:



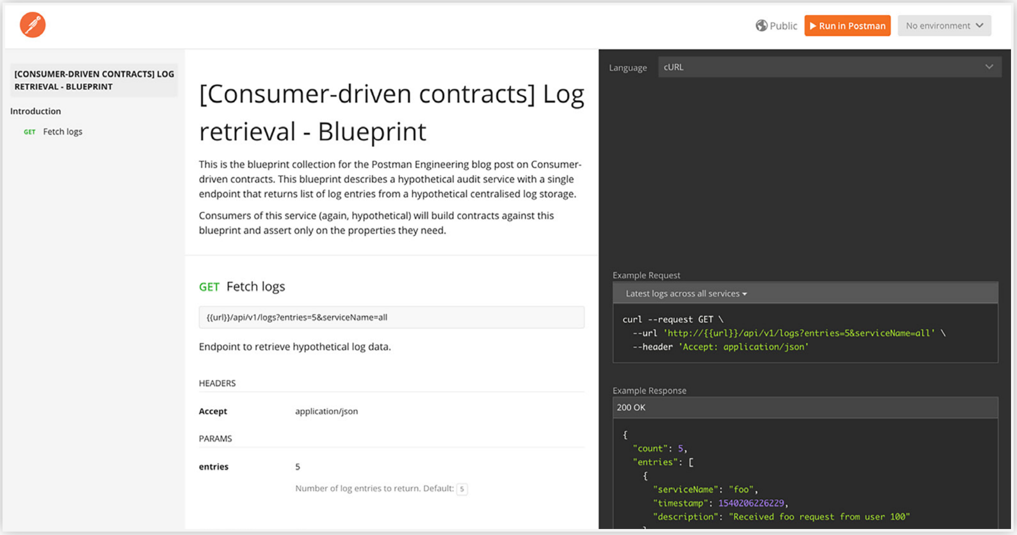
```
GET {{url}}/api/v1/logs
Status 200 OK
```

```
{
  "count": 5,
  "entries": [
    {
      "serviceName": "foo",
      "timestamp": 1540206226229,
      "description": "Received foo request from
        user 100"
    },
    {
      "serviceName": "bar",
      "timestamp": 1540206226121,
      "description": "Sent email to user 99"
    },
    {
```

Adding example to sample blueprint collection's request

With the example responses in place, Postman will automatically generate web-based documentation for the blueprint.

Here is what Postman's published documentation looks like:



[CONSUMER-DRIVEN CONTRACTS] LOG RETRIEVAL - BLUEPRINT

[Consumer-driven contracts] Log retrieval - Blueprint

This is the blueprint collection for the Postman Engineering blog post on Consumer-driven contracts. This blueprint describes a hypothetical audit service with a single endpoint that returns list of log entries from a hypothetical centralised log storage.

Consumers of this service (again, hypothetical) will build contracts against this blueprint and assert only on the properties they need.

GET Fetch logs

`{{url}}/api/v1/logs?entries=5&serviceName=all`

Endpoint to retrieve hypothetical log data.

HEADERS

Accept	application/json

PARAMS

entries	5 Number of log entries to return. Default: 5

Example Request

Latest logs across all services

```
curl --request GET \
  --url 'http://{{url}}/api/v1/logs?entries=5&serviceName=all' \
  --header 'Accept: application/json'
```

Example Response

200 OK

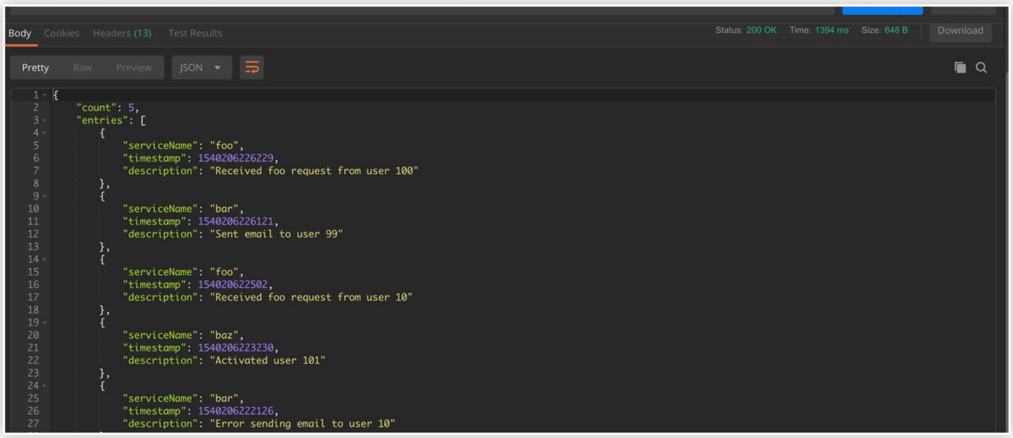
```
{
  "count": 5,
  "entries": [
    {
      "serviceName": "foo",
      "timestamp": 1540206226229,
      "description": "Received foo request from user 100"
```

Published documentation generated by Postman from the sample blueprint collection

Next, you will need to create a mock server in Postman based on this collection. The example responses added in the request will be sent as part of the mock server response. You can make requests to the mock server using the endpoint Postman generates for you as a URL.

Note: If you are using a team workspace, all members can view the documentation, comment on the documentation, and access the collection.

Making a request to `https://<mock-server-id>.pstmn.io/api/v1/logs` will return the following response:



Status: 200 OK Time: 1394 ms Size: 648 B

```
{
    "count": 5,
    "entries": [
        {
            "serviceName": "foo",
            "timestamp": 1540206226229,
            "description": "Received foo request from user 100"
        },
        {
            "serviceName": "bar",
            "timestamp": 1540206226121,
            "description": "Sent email to user 99"
        },
        {
            "serviceName": "foo",
            "timestamp": 154020622502,
            "description": "Received foo request from user 10"
        },
        {
            "serviceName": "baz",
            "timestamp": 1540206223230,
            "description": "Activated user 101"
        },
        {
            "serviceName": "bar",
            "timestamp": 1540206222126,
            "description": "Error sending email to user 10"
```

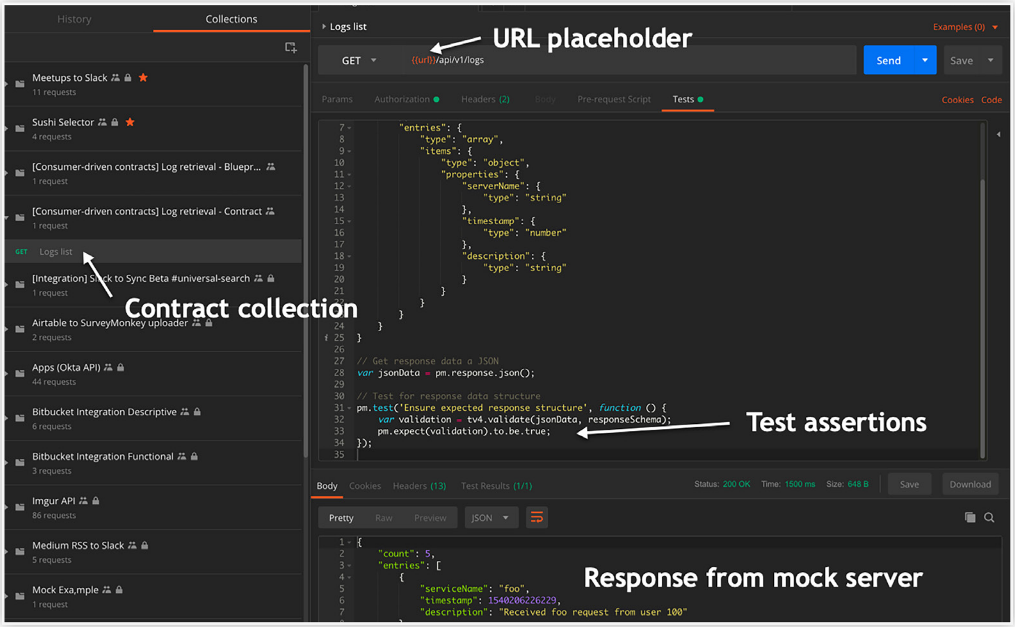
Response returned from mock server created from sample collection

Writing Contract Collections

Consumers of a service can build contract collections based on a blueprint collection and mock server. Postman tests allow you to assert on every aspect of the response, including response headers, body, and response time.

For this example, let's assume there is only one consumer of this service. Our contract collection example will have one request and it will assert only on the response data structure.

Note: A real-world contract would assert on the data structure as well as the data received in the response.



```
{{url}}/api/v1/logs
```

```
        "entries": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                    "serverName": {
                        "type": "string"
                    },
                    "timestamp": {
                        "type": "number"
                    },
                    "description": {
                        "type": "string"
                    }
                }
            }
        }
    }
}

// Get response data a JSON
var jsonData = pm.response.json();

// Test for response data structure
pm.test('Ensure expected response structure', function () {
    var validation = tv4.validate(jsonData, responseSchema);
    pm.expect(validation).to.be.true;
});
```

```
{
    "count": 5,
    "entries": [
        {
            "serviceName": "foo",
            "timestamp": 1540206226229,
            "description": "Received foo request from user 100"
```

Consumer contract collection using tests to assert on response data

Below is an example of a test script that the contract collection can use to test the data structure:

Note: It uses the tv4 library, which is included in the Postman Sandbox:

```
// Define the schema expected in response
var responseSchema = {
   “type”: “object”,
   “properties”: {
       “count”: {
           “type”: “number”
       },
       “entries”: {
           “type”: “array”,
           “items”: {
               “type”: “object”,
               “properties”: {
                   “serverName”: {
                       “type”: “string”
                   },
                   “timestamp”: {
                       “type”: “number”
                   },
                   “description”: {
                       “type”: “string”
                   }
               }
           }
       }
   }
}
// Get response data as JSON
var jsonData = pm.response.json();
// Test for response data structure
pm.test(‘Ensure expected response structure’, function ()
{
   var validation = tv4.validate(jsonData,
responseSchema);
   pm.expect(validation).to.be.true;
});
```

The contract collection is published **here**. You can use the “Run in Postman” button to load the collection in your Postman app and play around with the configurations. Note the use of the {{url}} variable placeholder in the contract collection. When a service is in its early phase, consumers can use the mock server URLs to make the requests. When the service has been built, the environment variable can be switched to point to a hosted instance of the service. This way, development of consumer applications or services can happen in parallel.

Continuous Testing

Contracts need to be continuously tested to ensure validity over time. There are two ways this can be achieved.

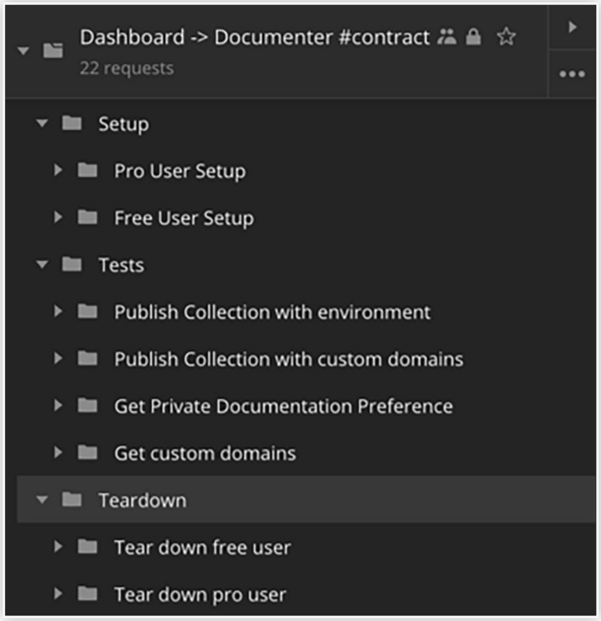
If you have an existing continuous integration system, you can export collection files and environments from Postman and **run them from the command-line** using Newman. Refer to Newman’s documentation for steps to set up continuous builds for **Jenkins** and **Travis CI**. Your build pipelines can be triggered every time there is a change in the specification or for every new version of the upstream service.

Organizing Contract Tests

Real-world tests have setup and teardown phases. Contract tests are no different. A common use case for contract tests is to populate the system being tested with some data, perform operations on it, then remove the test data.

A neat pattern of emulating this in Postman is to have a Setup folder as the first folder in the collection and a Teardown folder as the last folder of your collection. All contract tests can then be situated in between the Setup and Teardown folders. Postman runs collection sequentially, so this will ensure that Postman will always run the requests in the Setup folder first, then the contract tests, and will lastly run the requests in the Teardown folder.

We make use of this pattern when writing our own internal contracts:



Dashboard -> Documenter #contract
22 requests

- Setup
  - Pro User Setup
  - Free User Setup
- Tests
  - Publish Collection with environment
  - Publish Collection with custom domains
  - Get Private Documentation Preference
  - Get custom domains
- Teardown
  - Tear down free user
  - Tear down pro user

Setup and Teardown folders in an actual contract collection used by the Postman engineering team.

Consumer-driven contracts provide the surface area for testing microservices and negotiating changes. For even more on testing, **visit our Learning Center**.

Conclusion

Automated testing is shifting the landscape of software development. Test automation allows for faster iterations, more reliable test results, and ultimately higher quality products. There are many aspects to consider before and during the transition from manual to automated testing. There is no catch-all method for testing. Manual testing is necessary for qualitative results and for tests that require human verification. Investing in the initial setup for automated testing can be time-consuming and costly.

When you strategically consider cost, time, test characteristics, etc., you can optimize your testing tactics to be cost-effective and efficient. Creating a successful testing strategy means utilizing both manual and automated testing to the fullest potential. Automated testing tools are becoming more common and readily available.

Building an automated testing suite saves a significant amount of company time and money and improves overall ROI **(Palamarchuk, Sofia)**. Setting up an automated test involves more planning and up-front development time than manual testing. Once the initial setup is complete, the result is that less planning and development will need to be spent on testing in the future. Test automation offers greater scalability, saves time and resources in the long run, and simplifies your workflow. Automated testing provides a huge advantage for software developers.

References

1. Bhamaret, Lalitkumar, and Motvelisky, Joel. *State of Testing: Report 2017.* QA Intelligence, 2017, http://qablog.practitest.com/wp-content/uploads/2017/03/State_of_testing_2017_final_report.pdf
2. Colantonio, Joe. "Automation Testing Ultimate Guide (How, What, Why)?" *Automation Awesomeness,* https://www.joecolantonio.com/automation-testing-resources/
3. Damm, L-O. & Lundberg, L. & Olsson, D. (2005). *Introducing Test Automation and Test-Driven Development: An Experience Report. Electronic Notes in Theoretical Computer Science.* https://core.ac.uk/download/pdf/39962016.pdf
4. Grossman, Paul (2009). *Automated testing ROI: fact or fiction?* HP Technosource. https://docplayer.net/20448512-Table-of-contents-automated-testing-roi-fact-or-fiction-a-customer-s-perspective-what-real-qa-organizations-have-found.html
5. K., Elena, and Z., Maryana. "Pros and Cons of Automated and Manual Testing." *RubyGarage,* https://rubygarage.org/blog/automated-and-manual-testing
6. Palamarchuk, Sofia. "The True ROI of Test Automation." Abstractaca, https://abstracta.us/blog/test-automation/the-true-roi-of-test-automation/
7. Sridharan, Cindy. "Testing Microservices, the sane way." *Medium*, 30 Dec. 2017, https://medium.com/@copyconstruct/testing-microservices-the-sane-way-9bb31d158c16