

APIs in Action

A Guide to Monitoring APIs
for Performance

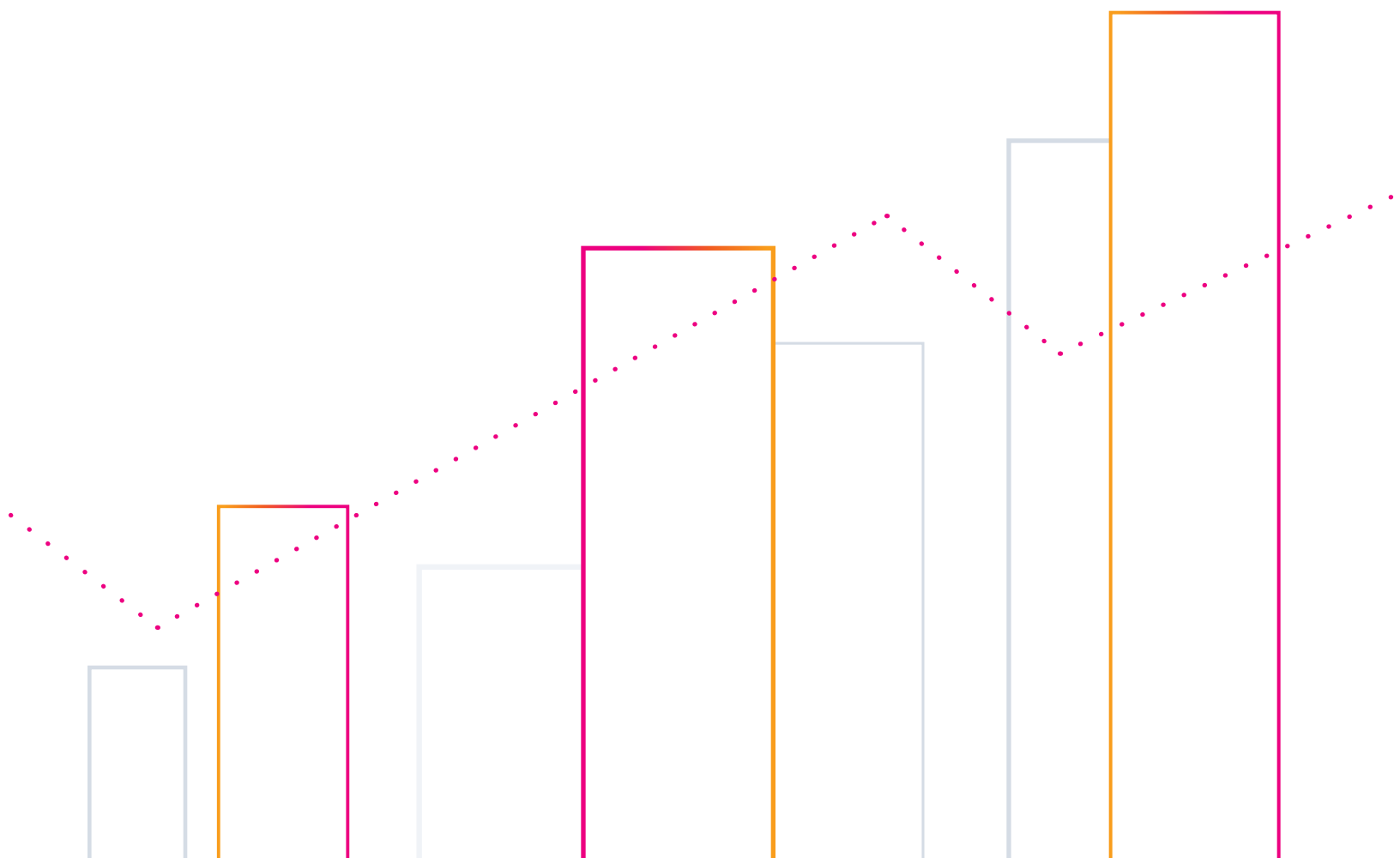


Table of Contents

Introduction.....	3
Understanding APIs.....	3
APIs in Action.....	4
Chapter One: Soap, Rest and Json	5
Chapter Two: Why Monitor APIs.....	5
Chapter Three: How to Monitor APIs.....	7
Chapter Four: Monitoring for SLAs	11
Chapter Five: Splunk’s API Checks	13

Introduction

Thanks to the rise of microservices, the spread of single page web apps and the continued dominance of native mobile apps, APIs are the unsung heroes of the modern web.

Early on, APIs proved a way for businesses to differentiate themselves and integrate with other systems, but some argue that they didn't become completely mainstream until the rise of social media. As social media networks grew and gained more consumer users, businesses began building and leveraging APIs to allow users to share and embed content into streams of social media interactions.

Today we use APIs to connect between multiple social media platforms — for example, posting from Instagram to Facebook or Twitter — or to connect to sets of data, such as pinning our location on a map so that we can check in or hail a ride. Many web and mobile apps that we use today are built on top of other APIs, such that they're tightly dependent on those APIs performing and returning data correctly, quickly, securely and reliably.

Because APIs drive modern applications and infrastructure, it's critical that we monitor APIs for performance.

In this white paper we'll help you understand the basic functions of APIs, why it's important to monitor APIs, and what types of features you should look for when implementing systems to monitor API performance.

Understanding APIs

An API is a set of programming instructions and standards for accessing a web-based software application or service. APIs give developers instructions about how to interact with services and can be used to connect data between different systems. An API describes the available functionality from a service, how it can be used and accessed, and what formats it will access for inputs and outputs.

Today entire businesses and applications are built on open APIs, relying on the ability to pass data back and forth between systems. Think of an API as a waiter at a restaurant. At a restaurant you have a menu of items you can order. Maybe you have specific instructions about how you'd like your dinner order to be prepared. When the waiter asks for your order you say, "I would please like the filet mignon, medium-rare, with a side of creamed spinach and fully-loaded baked potato with no chives."

Your waiter writes down your order, delivers to the kitchen, picks up your food when it's ready and serves it to you at your table.

In this simple scenario the waiter's role mirrors the role of an API, except instead of delivering an exquisite meal, an API delivers data. Like a waiter, the API takes a set of instructions (a request) from a source (an application or a developer), takes that request to a database, fetches the data or facilitates some actions and then returns a response to the source. APIs are messengers that keep systems connected.

When it comes to sending and receiving API messages, it's important to know that APIs may be private programmer APIs used within an internal organization or they may be public, consumer-facing APIs. Private APIs make businesses more agile, flexible and powerful.

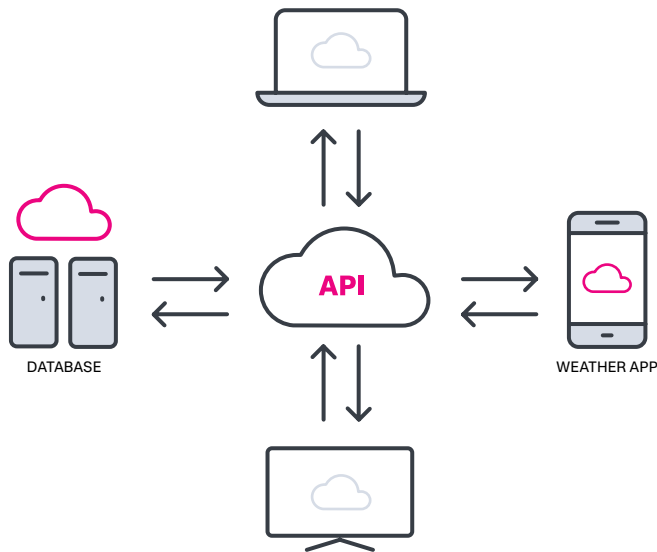
Public APIs help businesses connect to offer new integrations and build new partnerships.

APIs in Action

Let's look at a well-known company that relies on public APIs.

The Weather Company collects weather data from millions of endpoints and sources around the globe. They store this data and make it accessible to consumer-facing applications via hundreds of different APIs.

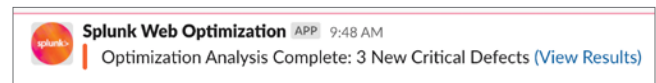
Consider a widget on your home screen or desktop that always displays current temperature and weather conditions based on your location. Your device may not have a thermometer or a barometer or any baked-in technology that detects or predicts weather patterns, but your device can send information about your location to an API and return the correct data such as the temperature and forecast based on your location. Even if you're not on the Weather Company website or using one of the Weather Company's native mobile applications you may be interacting with data from a Weather Company API responding to simple requests.



Now let's compare that public API scenario to an example of how APIs can be used to trigger tasks to improve internal business processes.

Splunk Web Optimization captures a number of performance metrics for digital businesses and provides smart suggestions for how to improve site performance. The optimization tool can help teams identify when any changes to a website introduce a regression in performance, meaning a site or web app has some content that's making it load slower than it could.

Using the optimization tool's API, the team at Splunk can integrate performance testing into deployments and identify any performance regressions introduced. A request was built into the deployment process for the optimization API that triggers a performance scan to Splunk's staging environment and posts the details of the scan into our Slack chat channel. Now we can use our existing ChatOps setup to notify the team when anything we build in staging will make our application slower.



In this scenario the API doesn't simply return data based on a request, but it also takes action based on a request. Both the Weather Company API or the Splunk API could be public or consumer-facing, meaning that other systems could rely on these APIs to return data or trigger specific actions.

CHAPTER ONE

SOAP, REST and JSON

At their core, APIs are about requesting and receiving data from a remote service. In this eBook, we are focusing on web-based APIs, where the request and response happen over HTTP. There are many different ways to format these requests and responses. When reading about APIs, you will see a wide variety of acronyms or terms. Largely these are unimportant to the broader concepts of APIs, their importance and how to test and monitor them for performance.

Here are a few high-level points that will help you find your way:

- Different API formats may use different ways of structuring request and response bodies; SOAP is very structured, using XML. REST can use anything, but usually uses JSON.
- Different API formats may use different HTTP methods when making requests. Something like SOAP might always use POST, while REST can use GET, POST or even less common methods like PUT or DELETE.
- Different formats may pass credentials or authentication information in different ways. Cookies could be used, or special HTTP headers, or even query string parameters.

If you find yourself getting lost or confused, just remember: APIs are all about sending a request and getting data back. Everything else flows from this simple idea.

CHAPTER TWO

Why Monitor APIs

Earlier we covered some examples of systems that are dependent on data from APIs or that have core functionality built on top of third-party APIs.

APIs are wonderful because they connect services and let us pass data back and forth between uniquely managed systems, but that connectivity and interdependence creates vulnerabilities. When it comes to the performance and availability, it's important to monitor the APIs that we depend on and the APIs that we provide for others.

Why API performance matters

When an API fails and disrupts the performance or user experience on your site, this failure reflects on your company. End users and customers likely won't recognize that a third-party could be at fault. And depending on how critical that API is to a transaction process, this failure could impact your bottom line right away.

For example, if a key component of the checkout process on your website is a location-based search and you rely on a third-party API to provide the search by location, when that API doesn't work correctly, your potential customers cannot check out successfully.

Or, imagine that you developed an application that requires authentication from a social media platform. If the API for that social media platform goes down, your users might not be able to log into your system.

As a developer or a site owner you may decide that the benefit of relying on the third-party service outweighs the risk of these types of failures. In order to accurately assess the risk and have visibility into the impact of these services over time, it's crucial that you monitor the part of your site's user flow that relies on an API.

If you have an open API that you've made available to partners or developers, then you have a responsibility to ensure that the API is available and working as expected.

Or, let's say that you've developed a new internal API that passes order data from a mobile device to a system in your product warehouse. Maybe it's critical that the data passes to the warehouse within two minutes, or the entire production schedule will be off. When you developed the API and tested it in staging it always passed data successfully within one minute, but when you launch the API and start processing real requests you notice that the real response time is creeping up closer and closer to that two-minute threshold. Without active monitoring on the API in production, your team might assume that the developed API is fast enough based on pre-production tests.

Note: If you host an API that other people rely on, be sure to actively monitor that API in both pre-production and production environments.

Whether it's to keep tabs on your own APIs or see the impact of external services that you rely on, it's important to monitor APIs for availability, functionality, speed and performance. If you know you have an API that's been unreliable in the past and you're not actively monitoring that API, start the conversation with your team and develop a plan to begin monitoring that API today.

What to monitor

So now that you understand why it's crucial to monitor API performance, you also know that you should consider both:

- APIs that your website or native application relies on for critical data or processes, and
- APIs that you manage that customers, end users or developers rely on for data or processes.

When monitoring both of these types of APIs, it's important to test:

- **Availability:** Is this API endpoint up? Is it returning an error?
- **Response time:** How quickly is the API returning responses? Is the response time degrading over time? Is the response time worse in production than in pre-production?
- **Data validation:** Is the API returning the correct data in the right format?
- **Multi-step processes:** Can I successfully save and reuse a variable from this API? Does authentication work as expected? Can I complete a transaction with data from this API?

These are just the basic concepts that your team should be looking for when it comes to monitoring API performance. In the next section we'll cover how to technically implement monitoring for APIs and what types of features are important to build out robust, flexible performance tests.

CHAPTER THREE

How to Monitor APIs

If you have access to an external, proactive monitoring system, monitoring a response from an API for availability can be pretty simple and easy to execute with basic uptime or ping-type checks. Define the protocol or put in an endpoint, set the test to run frequently from locations around the globe, and alert on bad response codes or latency.

But what if you need to monitor more than availability?

There are a number of key features that your monitoring solution will need to provide in order to fully test API transactions. These are components of typical API requests that you will need to configure in your tests beyond, “What endpoint should I hit?”

Request headers

Depending on how a site or application works, request headers may be a critical part of requirements for an active monitoring test’s configuration in order to effectively simulate a transaction. For example, if we need to actively test the way that a checkout process works when a visitor is cookie’d, then we’ll need some way to set that cookie with a Request Header when we build an active test on that transaction.

When we talk about request headers, we’re referring to fields passed along in the header sections of HTTP requests. Request headers can include rules and settings to define how an HTTP transaction should operate.

There is a standard set of supported request header types that have specific names and purposes. Some common examples of request headers would be:

- **Authorization:** Send credentials for basic HTTP authentication to give permission for access.
- **Cache-Control:** Tell the browser how long a resource is eligible to be cached and reused.
- **Content-Type:** Tell a server the MIME type of the body of a request so that the server knows how to parse the data.
- **Cookie:** Set a cookie to be stored in the browser so we can track state or sessions.

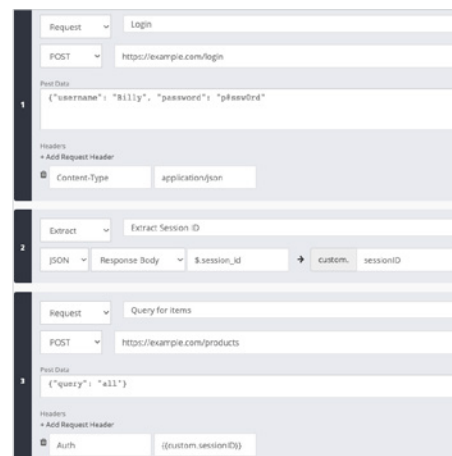
Some developers may also implement custom request headers with custom names. It’s common to see custom request headers with a prefix of “X,” for example: X-Http-Method-Override could override the request method from something like POST to another method like PUT or DELETE.

Request headers in API checks

Splunk’s API Check helps us monitor the availability, response time and data quality for transactions with APIs. With an API check, we can set request headers with each request as part of a transaction.

Consider a scenario where we need to POST username and password credentials to access some information. Then once we’re logged in at that endpoint we need to store and set a session ID in order to pre-populate other components specific to our session.

When building the steps for an API check, we can click + Add a Request Header to supply one or more headers at each request step.



In the example above, we’re using Splunk’s API check to:

1. Make a request to POST a username and password to an endpoint to log in.
2. Extract a session ID from the response body using JSON path and save that ID as a variable that can be reused in future steps.
3. Make a request to POST to a different endpoint with the Session-id in the request headers.

We could continue to add more functional steps to this transaction or add an Assert step to confirm that the session ID is set as expected.

Handling authentication

In the above example we used request headers to send over a username and password for authentication. Let's take a minute to focus on the security aspect of APIs and how we can consider this as we build performance tests.

Authentication for an API defines who has permission to access secure data or endpoints. This is especially important for APIs sharing sensitive information, APIs that allow end users to make changes, or for companies that charge some cost for accessing data via API. And while securing an API for an individual human end user is one undertaking, there are additional considerations as we authenticate systems for an increasing number of non-human entities.

As APIs become more secure, proactive monitoring systems are adapting to make it possible to access secure systems externally.

Direct authentication

A good example of direct authentication is HTTP basic authentication. HTTP basic authentication is a standard part of HTTP, and it can be used for API endpoints or any HTTP URL. You simply send a username and password — encoded together in base64 — as part of your request to the API. The benefit of HTTP basic authentication is that it's easy to implement and is typically included in standard frameworks. On the downside, HTTP basic authentication offers no advanced options and may be easily decoded.

Another example of direct authentication would be using API keys or tokens. API keys are just a long string of hexadecimal digits, i.e. 34d83d84f28d146aeae0e32f7803c88d, that can be sent instead of a username or password to authenticate access to an API endpoint. API keys are essentially the same as a set of username and password credentials, but they provide a layer of abstraction that is useful. For example, multiple end users could share a single API key.

When using any type of direct authentication, it's important that you also use SSL/TLS or

https:// at the start of the API endpoint URL. Using SSL/TLS will ensure that the HTTP basic authentication credentials or API keys aren't exposed in the URL.

Interested in learning more about SSL/TLS and how to optimize for performance? There are [a number of excellent articles on the Zoompf blog](#).

HTTP basic authentication

If you are using basic authentication to secure your APIs, it's super simple to include that authentication when configuring an external monitor to check for API performance.

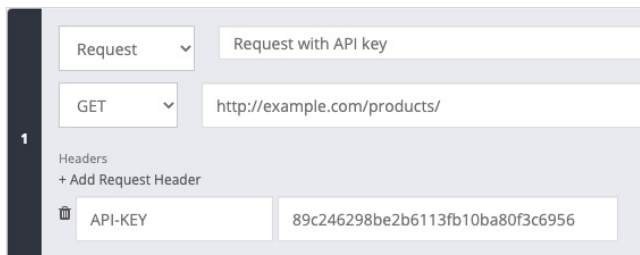
The most common and reliable way to set up a monitoring request with HTTP basic authentication is to go ahead and encode that username:password value in base64 and send that value over in an authorization header:



It's important to note that while it's easy to encode usernames and passwords into base64, it's also very easy to reverse or decode so that a system can authenticate a request. You can try this yourself with [an online base64 encoder/decoder](#). Because base64 is so accessible, it's important to protect this type of direct authentication with SSL/TLS.

API keys

From a monitoring standpoint, it's fairly simple to replicate the process of hitting an endpoint with an API key in the URL or with request headers. Supply the key and just remember that if it ever changes you'll need to update your monitoring test's configuration as well.



1

Request with API key

GET http://example.com/products/

Headers

+ Add Request Header

API-KEY 89c246298be2b6113fb10ba80f3c6956

Note that different systems may accept API keys in different ways — for example, as part of the POST data instead of as a request header — so check with the API you are monitoring to understand how to properly transmit the API Key.

Ticket-based authentication

While there are certainly some conveniences to implementing direct authentication, we may need to add an additional layer of security to our APIs. Ticket-based authentication systems rely on central authentication servers that act as intermediaries, accepting credentials from end users and then sending back tickets, tokens or keys that allow the specific end user to access only specific secured data. Ticket-based authentication is ideal for any scenario where you're protecting sensitive information, allowing an API to create objects or make changes or if you're charging some cost for use of your API.

Understanding ticket-based systems

We might think of ticket-based authentication as similar to how we might obtain keys to test-drive vehicles.

Imagine that I'm selling my junker of a car on Craigslist. You meet me at the local diner so you can give my car a test drive. You show me your driver's license. I say, "Hey! You seem to be this nice person who I just met online. I trust you absolutely. Here's the key. Give her a spin and

bring her back in a few minutes." That would be similar to direct authentication in the sense that I'm giving the car key directly to you based on your name.

Now imagine that I'm selling you a luxury vehicle from a dealership. You meet me at the dealership and give me your driver's license. I don't have one key that will work for all of the cars on the lot. I have to take your driver's license and use it to register your information to a computer system that will verify that you're an upstanding gentleman. This will then unlock a box where I can take out a key that will only work for the vehicle you're now registered to test drive. This is similar to a ticket-based system in the sense that I'm relying on a centralized system to distribute a key that's now connected to you through the ticket.

OAuth, Kerberos, single sign on and webforms are all examples of ticket-based authentication systems. You may even develop your own custom authentication system. While there are many different ways to implement this type of protocol, most ticket-based systems share a similar structure in the sense that you first make a request for a ticket or a token and then use that ticket or token to access secured data or endpoints.

The tickets in ticket-based authentication systems look very similar to the API keys we discussed above. One main difference is that the tickets are ephemeral. They are only valid a short period of time and can be easily revoked, which provides an extra layer of security.

Monitoring with ticket-based authentication

In order to effectively monitor an API that uses ticket-based authentication you must be able to complete multiple steps and save the ticket or token in a variable that can be reused in future steps.

A simple example of this would be to make a request with a username and password and some type of specification in the header, then retrieve a token from the system, save that token as a variable and then make another request to an endpoint with that token as a header.

If you're not already implementing some authentication for your API, it's critical that you start doing so to protect your data and your systems. And, as you increase security make sure that your external monitoring systems also have the permission and ability to monitor the performance and reliability of your system. If you're only monitoring your API performance on the application side, you could miss all sorts of connectivity problems preventing your end users from accessing data or making changes through your API.

Monitoring and validating data

When we're monitoring a website in a browser we want to go beyond checking the response code and confirm that some content or images load on the page. If the page returns a "200 OK" but it's completely blank, that's something we'll want to investigate right away. The same concept applies when we're monitoring API endpoints. When monitoring API endpoints we want to not only confirm that the response code is expected but that the right data comes back in the right format, too. Let's walk through a simple use case for how to use basic Extract and Assert options to validate that an API returns data in the correct format.

A data validation example

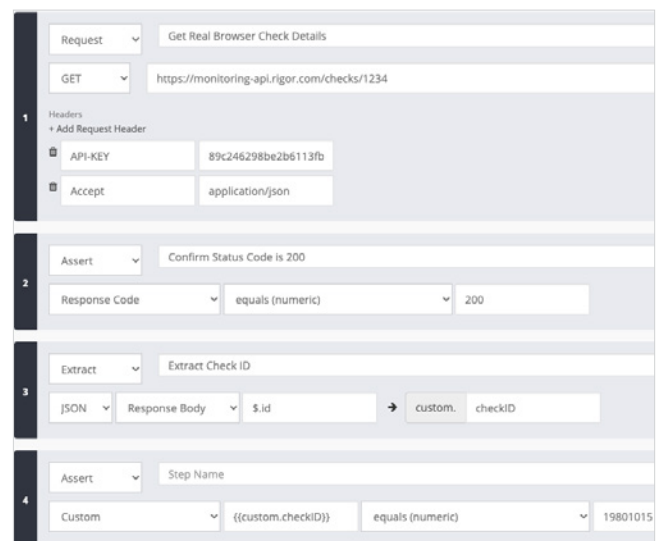
Splunk Optimization has [an open API](#) that Splunk users rely on to regularly pull data for reporting. It's important that when a Splunk user hits the endpoint for their check with an API key that we return a "200 OK" response code and the data set for the correct ID that matches the endpoint.

We can create an external, synthetic test to hit the check endpoint at a set frequency from multiple locations and confirm that:

1. Response Code = 200, and
2. The check ID included in the JSON output matches the URL endpoint we're hitting.

In the example that follows we're using a Splunk API Check to:

1. Make a request with an API key to Splunk API's endpoint for real browser check data.
2. Assert that the response code contains the value '200.'
3. Extract the check ID from the JSON using JSON path.



The screenshot displays the configuration for a Splunk API Check, organized into four numbered steps:

- Step 1: Request** - Method: GET, URL: `https://monitoring-api.rigor.com/checks/1234`. Headers: API-KEY (89c246298be2b6113fb), Accept (application/json).
- Step 2: Assert** - Confirm Status Code is 200. Response Code equals (numeric) 200.
- Step 3: Extract** - Extract Check ID. JSON Response Body, path `$.id` to custom `checkID`.
- Step 4: Assert** - Step Name Custom, value `{{custom.checkID}}` equals (numeric) 19801015.

4. Assert that the check ID extracted from the JSON path is the expected value. This very simple user flow helps us test:

- **Availability:** The check will fail if the API returns a response code that's not 200 OK.
- **Data Format:** If the data comes back from the API in a format other than JSON then the step to extract a value using JSON path will fail the check.
- **Data Quality:** If we're able to extract a value for the ID but it doesn't match the expected value, then the assert step will fail the check.

For example, if we receive an alert that our external monitor was unable to extract the check ID in JSON we would want to visit our alert and inspect the output or response body from the API endpoint. By looking at the response body we could quickly see whether the format was incorrect or whether the id value was missing from the output. This information would help us start troubleshooting right away.

This is just one simple example of how to implement robust monitoring for an API. If your current API tests only monitor for response code and response time, it might be time to consider adding some additional criteria for data format and quality.

Write performance tests to assume failure

So far we've looked at how to monitor with request headers, authentication and data validation in mind. When it comes to writing performance tests, one

strategy is to write tests in a way that allows a system to call an API and not receive data.

When writing code with lots of local calls, a wrapper that calls to an external API often goes unnoticed with the context of an application. If your test is designed to alert when no data is present, this will help make sure you don't miss critical errors. Remember to make your code resilient so that when it receives an error message, mangled data or no response at all, it will continue to function.

CHAPTER FOUR

Monitoring for SLAs

Earlier we touched on the vulnerabilities that arise from interdependent systems that must pass data back and forth on the web. As more and more applications are structured on top of third-party APIs it's critical to businesses that partner APIs perform quickly, safely, securely and reliably.

Businesses have developed a way to manage this risk, and we typically refer to these contracts as "SLAs."

What is an SLA?

A service-level agreement (commonly called an SLA) is an agreement between two parties about what services will be provided from one party to another. In a broad sense this agreement could include any number of services — everything ranging from custom support replies times to product delivery.

Often when SLAs are established between two technology or software providers, the agreement will outline both:

- **Availability:** What uptime percentage can be guaranteed by the partner? How much time in advance is required to notify a partner of planned downtime or maintenance?
- **Responsiveness:** How quickly can my system expect reply times from a partner's system?

And one party might be entitled to a credit, refund, or freedom to back out of a contract depending on whether those SLAs are met and upheld.

For example, let's imagine there's a ride-sharing web app that relies heavily on data from a third-party that specializes in mapping. When this ride-sharing web app agrees to work exclusively with one excellent mapping provider, that mapping provider may guarantee, "Your ride-sharing app will have access to our map data 99.99% of the time and we will notify your team at least three weeks in advance of upcoming planned maintenance."

The team managing the ride-sharing web app might say, "Nice! That sounds like a great deal. Our users tolerate some glitchiness and they never complain about it on Twitter, so 99.99% uptime is more than enough. Three weeks is plenty of time to let our customers know about upcoming downtime. We agree to the terms, but if your API is available less than 99.99% of the time we'll need to be refunded in full."

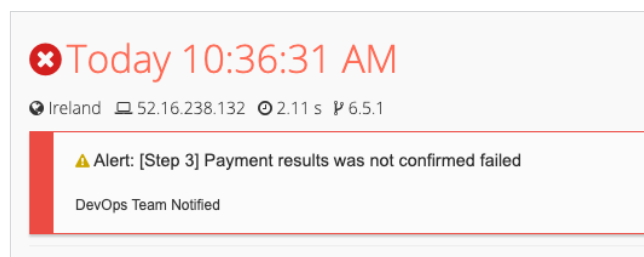
Everyone signs on the dotted lines and shakes hands and the ride-sharing web app builds a new feature that hooks to pull data from the mapping provider's API.

How to ensure that you're upholding your SLAs

As business owners for the mapping provider we might say, "Hey, we need some data to get ahead of issues so that we can make sure that we're upholding our end of the bargain. And, it would be nice if we could share that data publicly with our partners at the ride-sharing web app so they know they can trust our service."

We could rely on the internal monitoring of our application, but that might only give us part of the picture. How do we know whether our map data is available from our API to the end user outside of our system? How do we confirm that data isn't just available but in the right format?

We can build a synthetic, external monitor to test pulling data from our own API and put alerting in place so that our engineers know right away if there's any type of issue that might be putting us close to breach of our agreement.



The example above shows an alert from an API check in Splunk Synthetic Monitoring.

With the proactive data that simulates real end users interacting with our mapping system we can:

1. Get ahead of performance issues before they affect our real users, and
2. Share reports with our partner to demonstrate that our uptime is exactly what we promised.

SLA Report	
Jan 18 2021, 10:46AM - Feb 17 2021, 10:46AM	
SLA Report: New Panel 1 Last 30 Days (Jan 18 2021, 10:46AM to Feb 17 2021, 10:46AM) Checks: About , Checkout and 6 more checks Metrics: Uptime	
Name	Uptime (Mean)
About	99.907%
Checkout	97.847%
Homepage (Desktop)	99.988%
Homepage (mobile)	100%
Search Flow (mobile)	99.969%
Search Flow	100%
Product page (mobile)	100%
Product page	100%

How to enforce SLAs with your partners

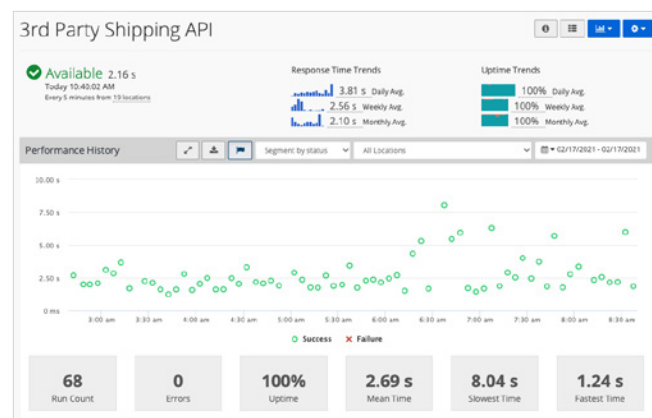
As business owners of the ride-sharing web app we may say, "Hey, that's nice that you're giving us these reports, mapping provider friends, but we really need to do our own due diligence and compare some external data to your reports." We could use external API checks to monitor the performance of the mapping API and confirm that the mapping API is in fact up 99.99% of the time.

In the event that we see availability fall under the SLA or if we see prolonged downtime that wasn't communicated three weeks in advance, we could use our reports to start a conversation with our partner about rectifying the breach of the agreement. We may also use API checks to better understand how issues with our partner's API might affect our real users.

API checks can be used by partners on both sides of service-level agreements to confirm that the agreement is being met according to the terms. For API owners, proactive monitoring can help you catch issues before they impact your partners and empower you with reporting that's easy to share with your partners. For API end users, proactive monitoring can alert you of third-party issues affecting your users and also help offer an extra level of confidence that your partners are upholding their agreement.

In the example that follows we can see a multi-step API check on the mapping provider's API that confirms that data falls in an expected range and tracks the availability of these requests.

Remember: API checks can be used to monitor both availability and responsiveness. Any use cases we build for API checks should match our SLAs. If our agreement is based on availability alone, we can configure a check to simply hit the end point and then rely on the uptime %. If our agreement is based on how quickly an API returns data, then we can build a multi-step check that pulls data from the API and then compare the average response time to our agreed standards.



CHAPTER FIVE

Splunk's API Checks

At Splunk we've developed a framework for performance tests that can help businesses understand whether APIs are performing as expected.

With API checks we can:

- Track the availability of critical APIs.
- Capture and trend an API's response time.
- Ensure API functionality by validating the API's response values and structure.
- Alert on any conditions indicating a broken or poorly performing API, allowing teams to get ahead of issues before they affect users or breach SLAs.

Splunk's API checks are built upon four simple concepts:

1. Making HTTP requests
2. Extracting data from responses
3. Saving data for use in additional requests or analysis
4. Making assertions about data and its format

The components are simple, but powerful. Like a fractal, sometimes beautiful and complicated things can be made by arranging the most basic building blocks. Splunk's API check uses simple pieces that can be assembled together for test cases capable of monitoring and verifying some of the most complicated API flows.

Engineers worked closely with customers during the development and beta testing of the API check to make sure this simple approach would make the check easy to understand while covering customer needs. One of our beta testers was a company called Lookout, a leader vendor of mobile security solutions. Here is what Dean Ross-Smith, a Cloud Operations Engineer at Lookout, had to say:

“The flexibility of Splunk’s new API Check allowed Lookout to configure a check and monitor a critical piece of infrastructure. We weren’t able to do this before with other solutions.”

Designed with businesses in mind

Whether an API is powering a web property, a mobile app or even the core infrastructure of your business, Splunk’s API check ensures that API is available, performing and functional in ways that users of any technical level can understand.

Here are just a few of the business needs that our API check can solve:

- Testing complex, multi-step API flows
- Monitoring availability and response time from geographies around the world
- Tracking and enforcing performance SLAs of third-party APIs
- Verifying correctness of API responses
- Testing the entire CRUD lifecycle of a data object via an API
- Handling complex token-based API authentication systems
- Monitoring application status pages

Takeaways

Do you have an application dependent on first-party or third-party APIs? Do you provide data to your customers via an API? Do you need great transparency into the availability, functionality, and performance of an API? The answer to any of those questions is the new API check for Splunk Synthetic Monitoring. To learn more about Splunk’s web performance products or have a demo, contact us today.

To get the full details and technical walk through, read more about API Check on [Splunkbase](#).



Learn more: www.splunk.com/asksales

www.splunk.com