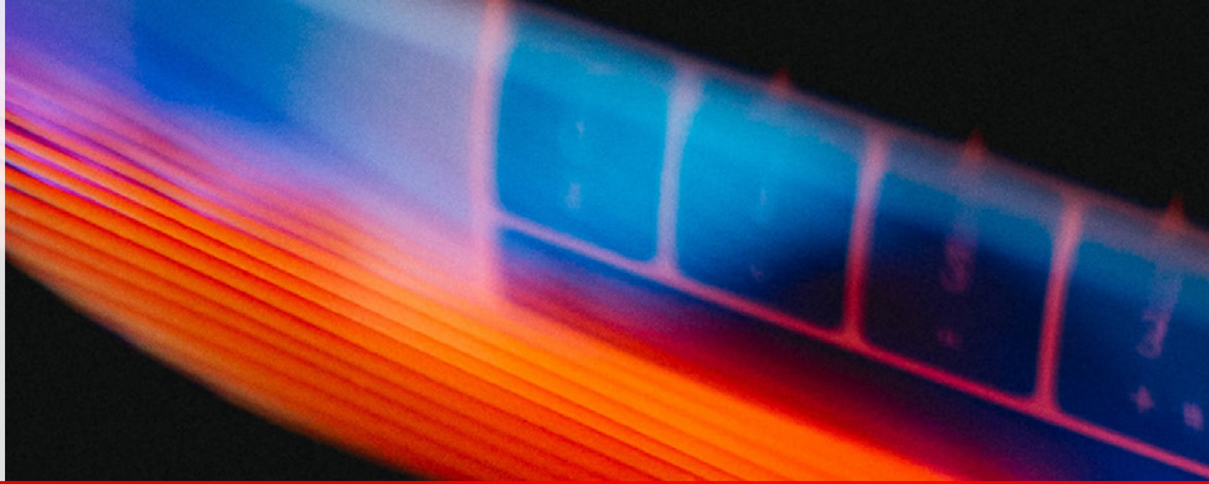
Special Report

Getting Started with API Testing: A Crucial Step in Microservices and Container Development

mindfire
BURN IGNORANCE



As monolithic architecture gives way to microservices and containers, software developers are embracing APIs to provide continuity and communication between complex applications and as a bridge between modern and legacy systems.

APIs are ideally suited for agile's short release cycles and DevOps' continuous deployment demands, but to keep pace, software testers must adopt new API testing strategies to ensure the functionality, reliability, security, and performance of their organizations' software systems.

This API testing eGuide provides a roadmap for starting your API testing journey. In this eGuide you will find background information on why API testing is a crucial part of creating high-quality software, resources to help you kick start your API test automation, and methods and tools for implementing a data-driven API testing strategy that is ideal for testing microservices and container APIs.

In this API Testing Special Report

mindfire BURN IGNORANCE

How to Get Started Automating Your API Tests

By Elise Carmichael

There are a variety of reasons testing teams decide it's time to introduce test automation. Maybe someone on your team has been building their technical chops, or perhaps a new manager or CTO has gotten more involved and wants to see QA advance. Or maybe your organization is one of the 74 percent of companies transitioning to DevOps and you want to support the transition by shortening your test cycles with automation, which can make continuous integration and delivery possible by increasing testing efficiency.

Today, you'd be hard-pressed to find a QA team that isn't doing test automation, or at least gearing up for it. Yet automation is still under-exploited in QA, according to Capgemini's 2017–18 World Quality Report.

So what's the best way to get started or expand your existing strategy? An area that is often overlooked is automating API testing.

API test scripts are faster and easier to write than other types of scripts and can be fairly simple tests. And while you need someone who's tech-savvy to write them, you don't necessarily need an experienced developer.

An added benefit is that API testing can be included in your continuous integration tools for execution and quick feedback on failures to the developers, giving you a quick win for providing more continuous feedback to your development team.

So what's the best way to get started or expand your existing strategy? An area that is often overlooked is automating API testing.

API testing has marked benefits for security-focused organizations. If you're not testing APIs for vulnerabilities and you jump straight into testing the UI instead, you could be missing a whole suite of basic security tests focused on authentication and authorization. And if there is a security breach in the services layer, it could expose the entire database, rather than just an individual user's data.

If you don't have a tester with development experience who can write custom test frameworks, there are wonderful tools like Postman and Newman that can help you write API tests, but you'll still be able execute the tests from a command line so you can include the tests in your build pipeline. Other good options include SmartBear's SoapUI and Apache JMeter. If you're practicing continuous integration, you can set up your tools to kick off API tests as soon as new code is committed to the repository.

Here's a rundown of how API tests could be integrated into the software development lifecycle:

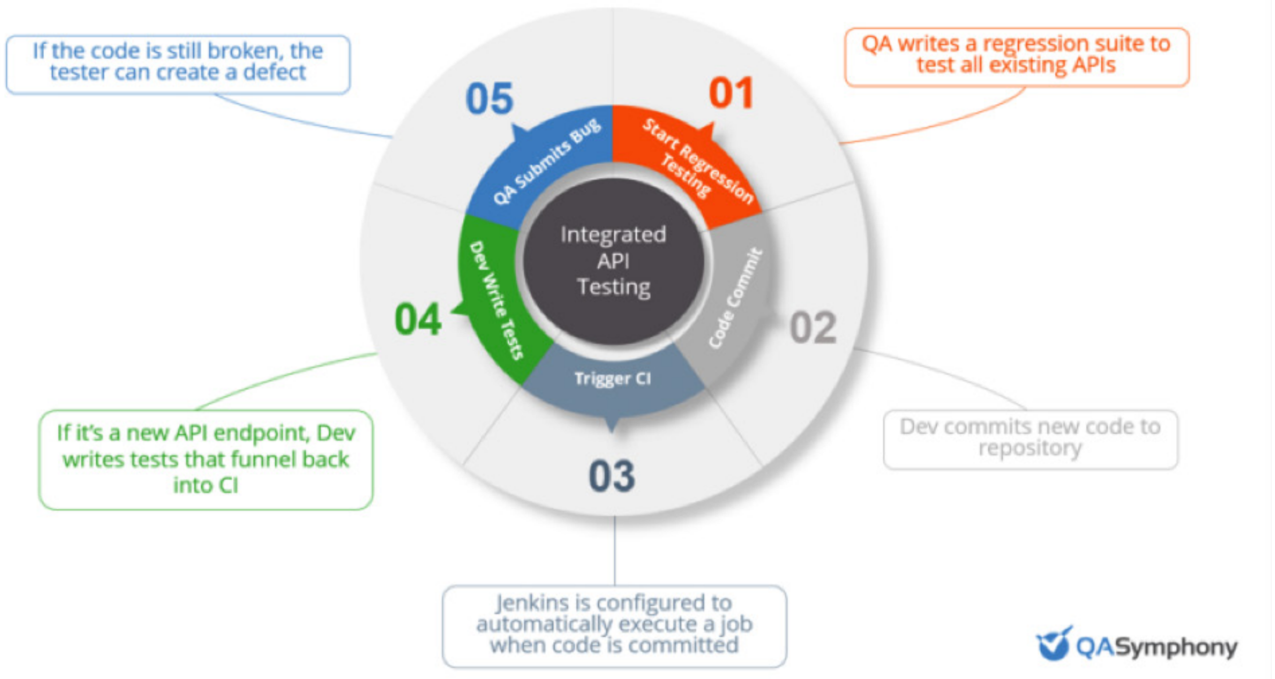
1. QA writes a regression suite to test all APIs.

2. Dev commits new code to the repository.

3. Configure Jenkins to automatically execute a job when code is committed.
- Code builds in Jenkins
- Code is deployed to a test environment
- API tests are executed against the environment to which code was just deployed
- Developers and testers are notified via email if the test fails

4. If this is a new API endpoint that the developer worked on, new tests get written and committed, then Jenkins builds and runs the tests again. (Note: Because APIs can have really detailed specifications, the tests can be written at the same time as the code.)

5. If the code is still broken, the tester can create a defect or fail the ticket.


If the code is still broken, the tester can create a defect
05
QA Submits Bug
01
Start Regression Testing
QA writes a regression suite to test all existing APIs
Integrated API Testing
Code Commit
02
Dev commits new code to repository
04
Dev Write Tests
If it's a new API endpoint, Dev writes tests that funnel back into CI
Trigger CI
03
Jenkins is configured to automatically execute a job when code is committed
QASymphony

mindfire
BURN IGNORANCE

Where to Begin with Test Automation

By Justin Rohrman

My introduction to test automation was a manager handing us a UI automation tool. The company I was working for at the time had hired an outsourcing firm to build an API using Ruby, and we were supposed to use it to drive our product in Internet Explorer.

That project was a failure. While we automated test case after test case, the nightly test suite runs were failing at a rate of 60 percent or higher. No one trusted the test results, and soon, developers didn't even bother to open the test report emails.

We had made a bad choice about where to start automating.

The user interface for many products is constantly in flux. The last company I worked for full-time was building a platform for advertisers. Each week, the development team added new pages to the UI that affected how other pages worked. A little less often, we took feedback from users to improve the usability and flow. And, every once in a while, the team would play "chase the trend" with JavaScript libraries and jump from Bootstrap to Angular and then to React.

Our manager wanted some UI automation. I built a few small tests that ran consistently for a week before needing to be refactored in order to cope with the changing user interface. Luckily, our manager got the point, and we didn't build much past a small smoke test.

If you are wondering where to start automating, the answer is usually as close to the code as you can possibly get.

The UI wasn't a good place for us, but we did have a REST API.

We ran another experiment, where I worked closely with a back-end developer. While she was writing code, I was stubbing automated tests in a JavaScript library. Each time I got to an assertion, I'd ask questions about data types, data formats, characters that should be allowed, and even user experience. These questions would guide the developer to make better choices about how to implement an API change, and that would help me write more relevant tests. By the time the feature was ready to check in, we had some test automation and the change had been exploratory tested.

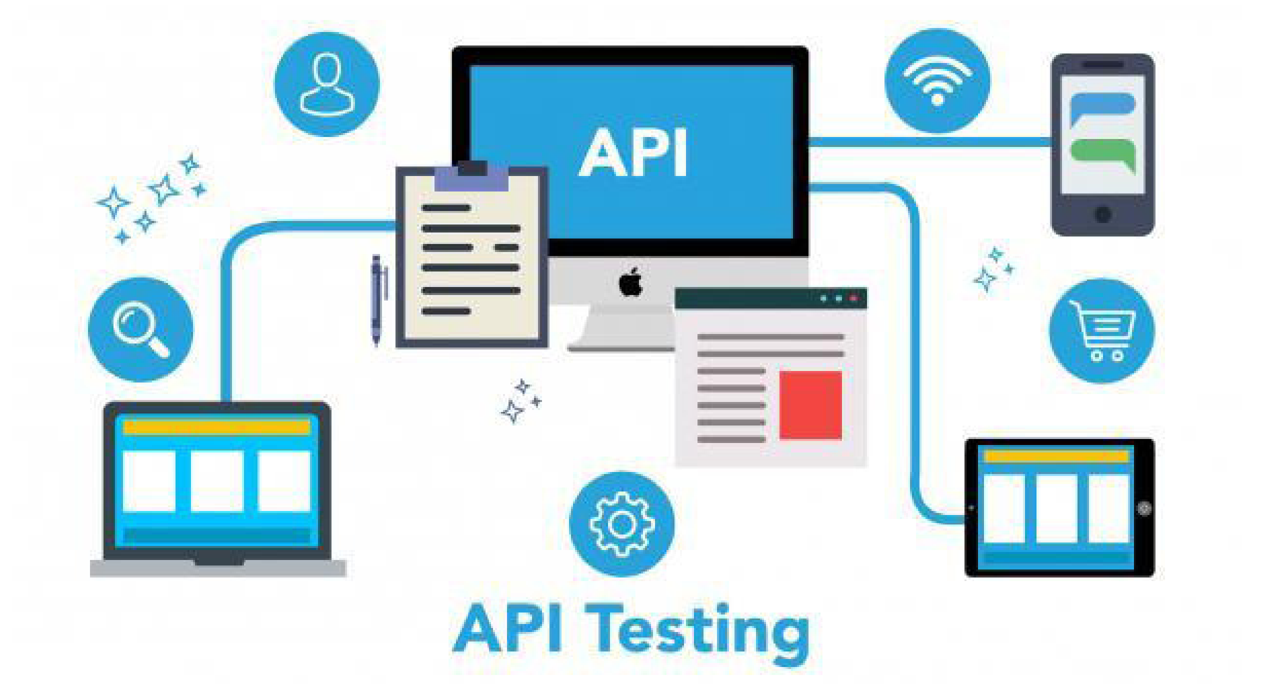
Responsibilities changed once the tests were live and running in our continuous integration system. Our development team was invested in the API tests, which ran several times a day and provided important information. When a test failed, the developer that broke code or the test would fix it.

Compare that to the normal UI automation flow. When a test breaks, someone on the test team is assigned to discover whether the failure was caused by a bug in the software, a bug in the test code, or a legitimate UI change. Next, they document the bug and wait for it to be prioritized and fixed. The test failure is completely detached from the development of new code.

If you are wondering where to start automating, the answer is usually as close to the code as you can possibly get. The farther you get from the code, the more you expose yourself to issues from change. How much of your UI automation project could be done more effectively—and faster—at the API or lower?

Benefits of API Testing: How It Assists in Faster and Better Delivery of Software Products

The development of software and web services continues to adapt and evolve to meet modern demands. Within digital platforms, we're currently witnessing an important transition from monolithic architecture to microservices. This new modular approach to software development enables better continuity between services and more efficient inter-process communication (IPC) within complex applications.



The evolution of microservices has a number of implications for software creation and testing, with an application programming interface (API) often used to manage data exchange and allow microservices to coexist with existing legacy systems. While the evolution of web services is a constant process, occasional leaps manage to change the landscape and disrupt the development cycle.

Instead of building monolithic applications as a single unit, developers are taking advantage of microservice capabilities, which are often formally expressed through containerization with APIs. The existence of modular building blocks with clearly defined communication and data exchange protocols allows for faster and more efficient software development and more robust testing procedures.

Monolithic Architecture vs. Microservices

Conventional monolithic architecture typically consists of an independent client-side user interface, server-side business logic, and data access layer. While this single logical executable approach can be extremely powerful, development, scalability and testing are all compromised.

In contrast, microservices break down individual applications into a number of functional and modular services that can be completely hidden and reused depending on context and need. While managing this approach can be challenging, it provides enhanced scalability, faster modification, and more transparent testing with minimal impact downstream.

API Testing and Integration with Microservices

In order to understand the relationship between microservices and APIs, it's important to grasp the significance of modularity within software development. If microservices are individual components within applications or container ecosystems, then APIs are needed to enable request-response messages and other forms of communication between modules.

API testing is necessary in order to ensure the functionality, reliability, security, and performance of software systems.

Along with enhanced communication, APIs also enable better integration between systems. Real-world scenarios are never as clear cut as they are in text books, with companies needing microservices to exist alongside existing processes and architectural patterns. Microservices are often coupled with APIs in containers, with this necessary union allowing businesses to implement new modular architectures without risking redundancy and increasing complexity.

API testing is necessary in order to ensure the functionality, reliability, security, and performance of software systems. Testing procedures are especially critical in order to integrate microservices with surrounding systems and ensure cohesion with third-party service-oriented architecture (SOA).

Differences between microservices, APIs, and SOAs are fluid and dependent on the surrounding context. While these concepts all employ similar principles of modularity, they have a different architectural scope. Microservices relate to application architecture, SOA typically operates at the enterprise level by exposing functions as service interfaces, and APIs are used to integrate systems and provide clients with access to back-end functions.

APIs vs. SOA Services

An API is a low-level programming interface consisting of subroutine definitions, communication protocols, and additional tools for building software. Despite the accuracy of this general definition, the term is mostly used today in reference to specific REST interfaces for web services provided over HTTP. Basically, APIs have become products in their own right. The key difference between APIs and SOA services is in terms of their scope and economics. While SOA programs are about reusing functions on the interface level, modern APIs are about creating new functional and useable interfaces that can be branded as products.

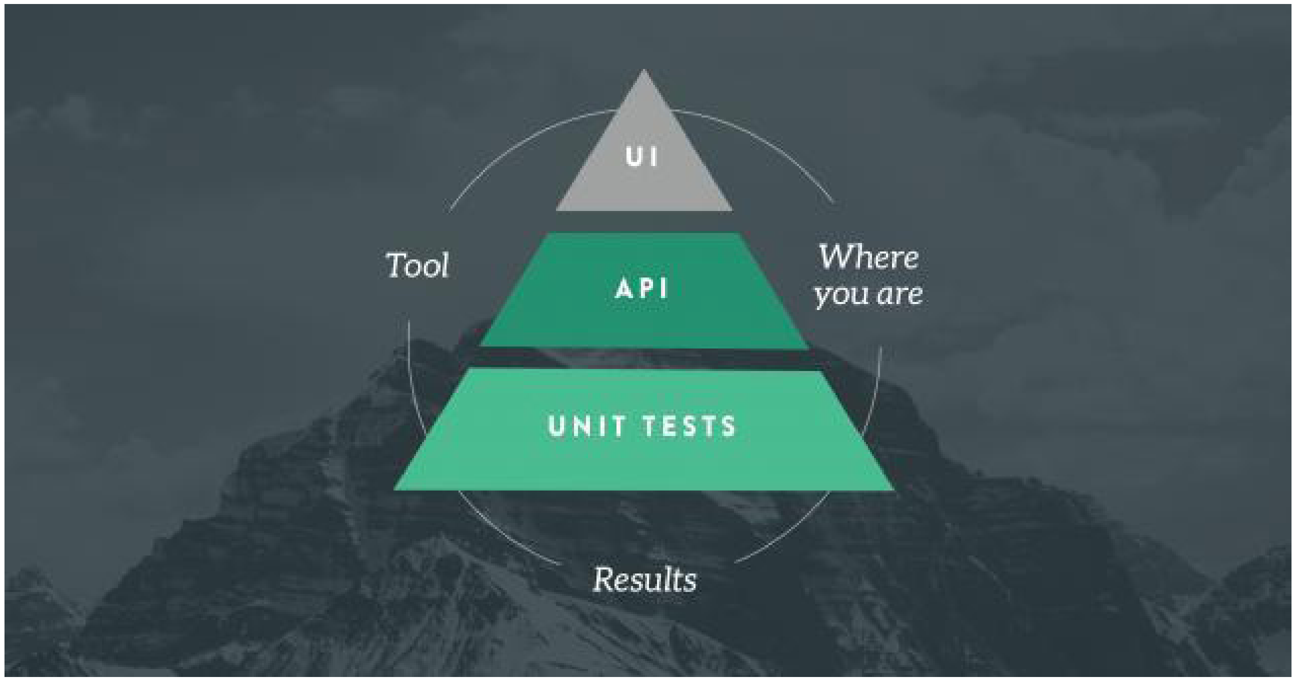
The Benefits of Early API Testing

The successful use of APIs depends on the automation and early adoption of testing procedures. While micro-services are internal and single-function applications designed to maximize efficiency, APIs can be exposed both internally and externally depending on the context. In order to maximize results and speed up delivery rates, it's important to automate the design and development of APIs and conduct testing from an early stage. API testing allows you to test the business logic of an application in a way that's independent from the user interface (UI).

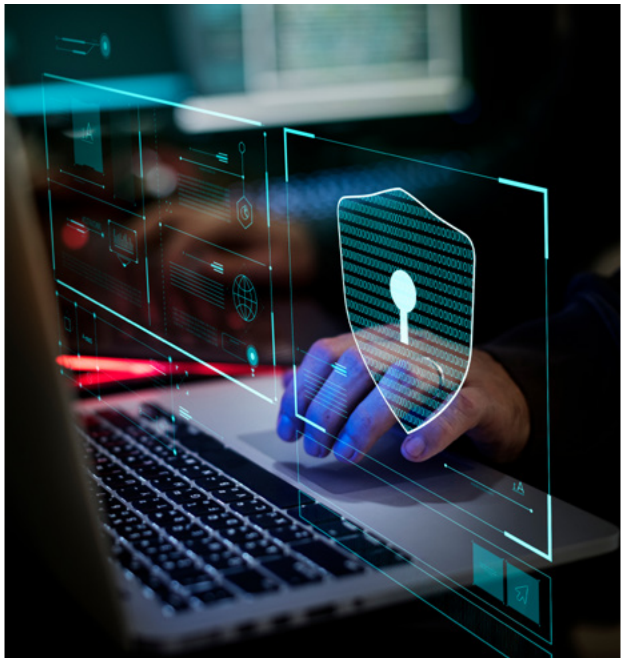
API Automated Testing vs. UI Automated Testing

Early testing allows you to identify security issues and bugs early in the software cycle before they lead to additional problems and spiraling costs. The ideal testing scenario involves the test pyramid, with unit tests followed by component tests, integration tests, API tests, and finally, GUI tests. While programmers are comfortable conducting unit tests during development and application testers work at the GUI level, API testing is often under-resourced or completely ignored.

By working from the bottom of the pyramid up and testing APIs early in the development cycle, software kinks can be ironed out before they snowball and affect the end user. Also known as test automation but not synonymous with it, UI automated testing is designed to mimic the actions of the end user. By scripting common actions involving the writing and reading of pages, fields and elements, UI tests attempt to identify issues starting from the front-end and going backwards.



While UI tests are a practical way to put yourself in the shoes of the end user, they are also slow, inefficient, expensive and cumbersome. The code base of a UI is typically much larger than that of an API, with UI elements also more fragile and prone to failure. Along with being an inefficient and expensive way to work, automated UI testing is also highly impractical compared to API testing. Instead of working from the outside-in, API testing allows you to test business logic from the inside-out in a way that ensures 100 percent test coverage.



API Testing for Improved Security and Performance

API testing offers numerous advantages over UI testing and related approaches, especially when it comes to security and performance. While an API can still be hacked, dealing with security and access issues at this level allows you to identify vulnerabilities before they affect the end user. There are lots of ways to identify attack vectors, including fuzzing or fuzz testing, command injection, testing for unauthorized endpoints and methods, and parameter tampering. Automated API testing suites need to be comprehensive enough to handle any eventuality and flexible enough to adapt to changing scenarios.

Automated testing at the API level also offers a number of advantages in terms of performance. Whether you're conducting functional tests or turning up the dial and running performance tests, API testing allows you to work through known issues in a way that's divorced from the user experience. While this disconnection can be dangerous, the ability to isolate API load testing and monitoring from the UI helps testers to check the overall speed and performance of software in a way that ensures accurate test results.

Continuous API Testing in DevOps and Containers Environments

API testing is relevant to a wide range of industries and software environments, including DevOps development that focuses on continuous delivery, continuous deployment, and continuous integration. Continuous delivery involves small build cycles with almost immediate feedback between development and delivery. The aim of this development paradigm is for code to be ready and waiting at any given time. Continuous deployment involves similar principles of speed and efficiency, only this time the focus is on automatic deployment with every change to code. Continuous integration involves merging code from multiple developers to one branch in order to avoid future conflicts.

Automated API testing suites need to be comprehensive enough to handle any eventuality and flexible enough to adapt to changing scenarios.

Robust API testing has an important effect on DevOps practices and containers environments, especially those that integrate microservices with APIs. Whether you're focused on improved efficiency, faster development, improved data exchange management, or the ability for microservices to co-exist with existing legacy systems, API testing ensures the faster and better delivery of software products.

To Test APIs Effectively—Build an API Regression Suite

By Sanjay Zalavadia

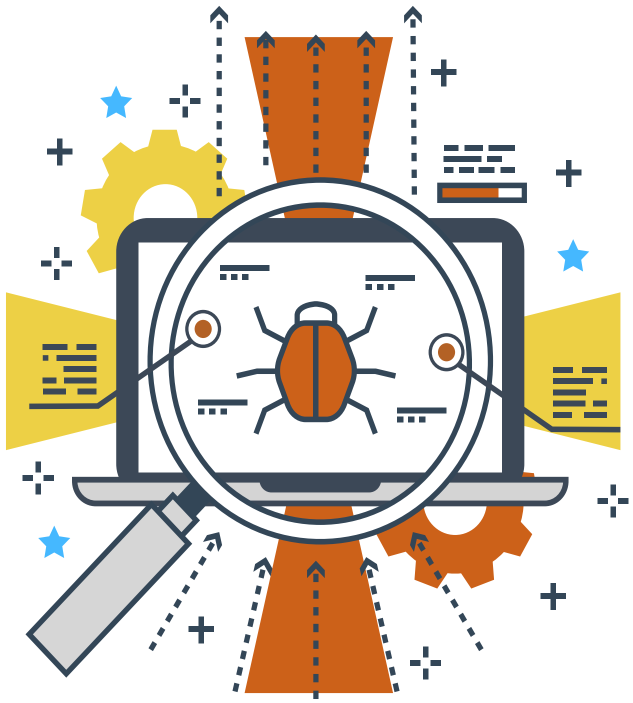
All of the ingenuity, innovation, and hard work that go into application development is essentially worthless if the finished product doesn't do what it was supposed to do. Alternatively, if the solution is not comprehensively vetted for potential bugs or errors in the code, security flaws and other issues that devalue the user experience may arise. For all of these reasons and more, establishing an efficient test management strategy is an essential step in creating great software. Herein lies the value of building an API regression suite.

The ABCs of Regression Testing

As each phase of software development occurs, and as subsequent changes are made either to enhance functionality, patch security flaws, or amend a glitch, certain test cases must be run in order to ensure that the alterations are compatible with everything else that came before them. This is not to be confused with simply testing again. Many test cases will have to be repeated, but as TechTarget contributor Mike Kelly notes, that definition misses the point. The purpose of regression testing is to make sure that the thing still works.

"When I think about regression testing, I think about any testing that involves the reuse of tests (manual or automated) or test ideas (regression charters for example – a regression test does not necessarily need to be the exact same test) to manage the risks of change," Kelly writes.

This means that an API regression suite will not necessarily be static. New tests can be added on an as-needed basis, and in some cases, manual testing may become necessary. Keeping this in mind is especially important when it comes to agile software development, considering feedback loops are much faster, and iteration cycles are shorter. This also means that testing teams must have a streamlined approach to regression testing since it will be occurring with more frequency. Anything less than meticulous test management here can pose the risk of an API that doesn't serve the end user.



Where Test Automation Tools Fit In

In theory, DevOps culture, agile development, and continuous development are aimed at greater flexibility in API development. The benefits are speedier time to market, quicker patching, and the ability to respond to consumer demands and changing market conditions at a breakneck pace. To achieve this, testers have begun relying on new testing methodologies, not the least of which is automation integration.

Regression tests are the perfect candidate for software testing automation, mainly because they can be so redundant. By not having to manually run the same test cases time and again, QA teams can save a significant amount of time and reduce the likelihood of introduced errors.

It makes so much sense to run regression tests at the API level in particular because, as noted by TechTarget contributor Yvette Francino, automated testing at the graphical user interface level is somewhat impractical. When it comes to GUI, functionality is a slightly more subjective matter. But at the API level, test management teams can ensure that everything that happens at the GUI level is supported by critical functionality at the base layer. Some exploratory tests can still be automated—and agile testing methodologies are becoming increasingly accommodating on this front—but some regression tests, especially for user experience, may still have to be run manually.

Nevertheless, the contributions of automated testing to an API regression suite are invaluable.

Long-Term Benefits

In the long run, building an API regression suite with a well-rounded test management system that supports collaboration between testers and developers allows for methodical test tracking, and features such as automation integration can enhance the development process as a whole. In addition to better software, the deferred benefits include the ability to reuse test cases for new projects. It also provides QA management with a golden opportunity to restructure their internal processes to make them inherently more agile. In chorus, all of this improves the chances that end users will be satisfied with the API, and this means a better bottom line for your business.

In chorus, all of this improves the chances that end users will be satisfied with the API, and this means a better bottom line for your business.

Methods and Tools for Data-Driven API Testing

By Albert Gareev

Software testing has many forms and breeds, but one major distinction has always been based on the approach—either working with the code or interacting with the product. The former was typically a prerogative of programmers while testers have concerned themselves with the latter. The industry even came up with the famous analogies of glass-box testing and black-box testing, meaning that you either see what's going on inside the box or you have no idea, observing only input and output.

However, modern software systems are boxes within boxes within boxes. Programmers use third-party libraries without knowledge of their code. Products interact with each other through integration protocols. At the same time, testing feedback is needed way before a product gets a user interface. Some products, like internet of things devices and wearables, do not have a UI at all.

While programmers take responsibility for more testing, testers may want to increase their value by becoming more technical. This includes using developer tools and learning to do a little coding on their own in order to test programs at the API level.

A Quick Introduction to API Testing

An application program interface, or API, is a set of rules specifying interaction protocols between software "boxes." API testing is a form of black-box testing—i.e., interacting with a function without knowing what's going on inside, through feeding inputs and evaluating outputs.

Internet applications use web services APIs in the form of either SOAP (Simple Object Access Protocol) or REST (Representational State Transfer). There are technical and architectural differences between these two communication methods that would be of interest to programmers, but they can remain within the black box for testers. In terms of input-output, both protocols use text-based format; SOAP uses XML notation and REST uses JSON notation. Both formats are easy to learn. Testers need to be able to read and compose

them in order to prepare test inputs and evaluate outputs.

Imagine a web form with data fields First Name, Last Name, and Date of Birth. Instead of manually typing an input, the tester would need to prepare data snippets, like below.

Data snippet in XML:

```
<firstname>John</ firstname>

<lastname>Smith</lastname>

<dob>1980/06/04</dob>

Data snippet in JSON:

{

“firstname”: “John”,

“lastname”: “Smith”,

“dob”: “1980/06/04”

}
```

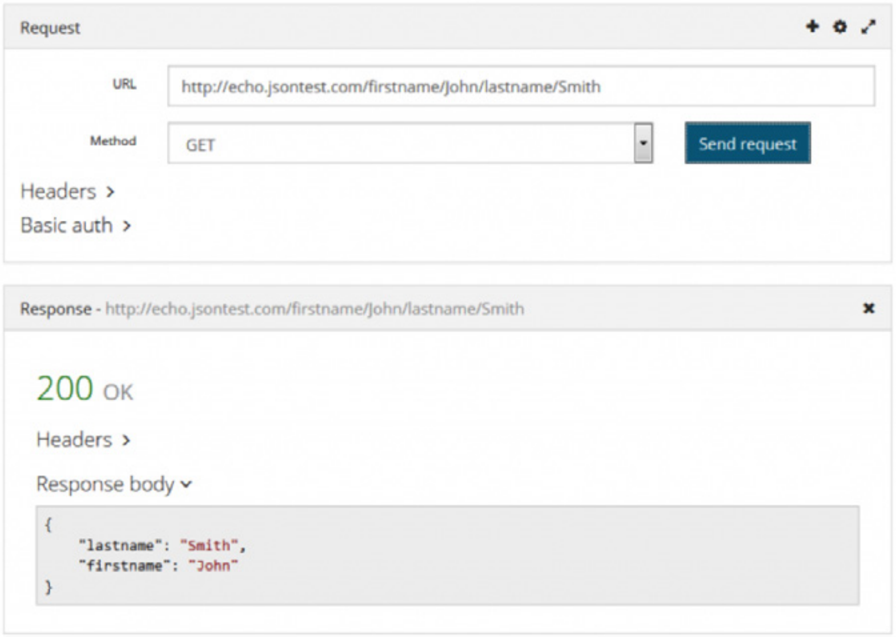
As you can see, the examples are easy to get started with.

Methods and Tools for API Testing

There are a few ways to perform API testing.

At the simplest, you can just put the web service URI—that is, the path and parameters—into the browser address line, and the response will be displayed within the browser window. There are basic and sophisticated add-ons available for free for all popular browsers, and there are free and commercial standalone tools. Finally, there are automation frameworks supporting different languages.

Here's an example using the RESTED Firefox add-on to place a request and review the response from the free online service JSON Test.


Request
URL
http://echo.jsontest.com/firstname/John/lastname/Smith
Method
GET
Send request
Headers >
Basic auth >
Response - http://echo.jsontest.com/firstname/John/lastname/Smith
200 OK
Headers >
Response body

```
{
    "lastname": "Smith",
    "firstname": "John"
}
```

Each method has its value. Quite often, there's a need for a few tests to be done quickly. They don't need to be repeated, so simple methods and tools are right on. If there's a need for continuous testing after each code change and deployment, advanced tools and automation suites fit better here.

The common denominator is risks. All testing is guided by explicit or implicit risk assessment. Risks assess the likelihood of something unwanted or negatively impacting the product due to a direct or indirect result of some code changes or build and promotion issues.

The most basic question about an API component, like a web service, would be, is it up and running? That question can be answered by making just one API call. If answered positively, developers may ask, does it properly handle correct and incorrect input and return the expected results? These questions might be further refined case by case with different data combinations.

Test Design and Execution

Test design is about answering these questions. We design and simulate various interactions with an API to explore and investigate its behavior.

If we organize our test questions from basic to deep coverage, we will end up with something like the following categories.

Availability
Checking whether a particular API function (or a few of them) is down is the most basic question. That can be answered with a single API call.

Functionality
Checking whether a sequence of API calls intended to work in a scenario is unavailable or behaving incorrectly in some way would be considered shallow but broad coverage.

One example is continuously scrolling a map while the app visualizes points of interest around your point of view. This operation is simulated by a sequence of API calls feeding a geolocation.

Another example would be sending a search request in an online store, reviewing search results, and retrieving details of a found item. This operation is simulated by a sequence of API calls where data retrieved from the previous call might be used as parameters for the next one (for example, “item id”).

Broad coverage also involves checking whether typical calls, such as valid data or a generic response, yield an invalid response or is incorrect in some way.

Input parameters
Input is what is sent into an API function when the call is made. Testing from the input perspective requires covering all variations of data input, including all equivalence classes of valid input and invalid input.

For example, boundary testing for a numeric field would include testing at the min boundary, one lower than min, way lower than min, one higher than min, one lower than max, at the max boundary, one higher than max, and as much as possible higher than max.

We design and simulate various interactions with an API to explore and investigate its behavior.

You also would include data variations to test valid and invalid format, precision, encoding, and any other applicable business rules. For example, for a web service “GetUserByID” with “ID” as a parameter, a thorough testing would require a sequence of calls like the following:

- Provide a legitimate ID
- Provide no ID (null input)
- Provide a non-numeric ID
- Provide a valid but nonexistant ID
- Provide a legitimate ID with the lowest number available
- Provide a legitimate ID with the highest number available
- Provide an ID with heading or trailing spaces

Output parameters
Output is data that were returned from the API call. Data may come in different forms, like response code, response header, and response body returned by a web service.

Testing the output aspects requires reproducing all variations of output. To provoke a desired output, you need to know what to request. A peek into a database would be of great help, as you can find and inject peculiar field values there. For web services, you have to reproduce all intended HTTP response codes.

Whenever a data set is returned, reproduce cases like none, one, and max. For example, consider an online

store returning a list of matching items. There could be none, or just one, or the returned data set may exceed the maximum number that can be displayed on one page—or the returned data set may be so big that it will slow down or crash the front-end.

System state
Have you seen things like “most popular article” or “most searched merchandise”? As customers use software, certain statistics are collected that affects the behavior of the system. There are also other effects on the system state, including memory usage and possible leakage, as well as filling in and overflow of buffers and queues.

To test this aspect, reproduce all intended consequences and investigate if there are unintended ones.

Flow testing
A special testing aspect is simulating continuous usage of the API. Imagine users constantly searching for merchandise in that online store. The intent of such a simulation is not a verification of something particular, but rather observing if the software responds consistently over time or begins to manifest different errors.

Flow testing also includes simulation of timed-out and interrupted calls. If your system has functionalities for rejection and rollback of transactions, make sure to reproduce them, and verify what happens with data records. For example, let’s say you’re booking a movie ticket online. Typically, it’s a sequence of screens for movie selection, date and time selection, seat selection, and payment. As soon as seats are selected, they become reserved. But what if a payment was never made or didn’t come through? How many seats will be lost due to a fake or failed booking? To manage that problem, the booking sequence has a timeout. If the final call doesn’t succeed, all reservations should be released, and the customer should not be billed. While testing through the API, make sure to reproduce situations like this and investigate how the system handles a timeout and performs a rollback.

Another aspect is testing with a high volume of data in input and output. That might be a request for an unusually large data record or set of records, or continuous throughput of data. This is similar to testing the system state, but the focus is on stressing amounts of data that the system should be able to handle.

Structure Your API Testing Needs

The lists of test design ideas above might look frighteningly long, but that’s because we reviewed very broad examples aiming for many kinds of risks. In practice, your level of coverage probably will be more constrained and specifically guided.

It is wise to test more important functions sooner and to spend more time on crucial aspects of a product. That calls for having a structured model and for having risks and priorities identified and monitored as the software development project unfolds.

Data-driven API testing can enable feedback much sooner and more often during development while being just as comprehensive as classic functional black-box testing. Testers looking to diversify their skills should consider learning some coding in order to test their programs at the API level.

Insight from around the Industry

On Testing in a DevOps Environment
"In environments where [DevOps] isn't happening, a tester might not even know what the APIs are called, or how to invoke them, or what information is being sent to and from. By embracing this [DevOps] culture, you're really getting into all the nitty-gritty for how an application is developed and architected. It absolutely will open up more doors from a testing perspective to really ensure that you're producing a quality application."
—Adam Auerbach

On Why Automated Testing Should be Ubiquitous
"If you look at any modern application, you will see it has a back end and several front ends. Web, mobile, and soon even wearable, like watches and clothes. At any point in time, you will need to be able to test API, regression, performance, and also obviously user interface. You will need to test the UI on various operating systems and devices. Organizations should strive to have all testing become obviously fully automated."
—Alon Girmonsky

On Modeling and API Testing
"So when you're designing, people say 'Oh, we'll just convert a couple programs to job and then we'll make them a microservice.' I'm like 'Stop, no no no. No no no no.' You need to fundamentally change the way ... you need to model exactly what this API is going to do. You need to model the roll backs, and then, once you have done that, you can then build those test cases automatically."
—Huw Price

On Automated Testing vs. Automated Tests
"You don't want to have automated tests, you want to have automated testing, and the more things around the execution of tests that you can automate, the better."
—Dorothy Graham

On Black Box API Testing
"The vast majority of the testing I do can be done without having any expressed knowledge of what the code behind the API is doing. The same things that a tester would do in the UI, you can do with the parameters of an API. "
—Jonathan Cooper

Additional Resources

MORE INFORMATION FOR SOFTWARE PROFESSIONALS

API TESTING SERVICES

Automation Testing Services

Software Testing Services

WHITE PAPERS

QA Dashboard Automation—The DevOps Way

Finding the Right Product Development Partner

Overcoming Critical Offshoring Pain Points in Software Development

BLOG

Puppeteer: Is It Time to Ditch Selenium?