Revolutionize your API testing practice (or bring it in for the first time) by leveraging artificial intelligence

INTRODUCTION

In service development, an Application Program Interface (API) is a way for various applications to communicate with each other using a common language, often defined by a contract (e.g. a Swagger document for RESTful services or a WSDL for SOAP services). Even databases have an interface language (e.g. SQL). Much like how a UI allows a human to effectively interact with an application, APIs allow machines to efficiently communicate with each other. In this paper, we look at how to effectively use APIs in development, the benefits of testing at the API layer, and how to lower the technical skills required for API testing by using artificial intelligence.

USING APIS

APIs are powerful because they are building blocks that developers can use to easily assemble all sorts of interactions, without having to rewrite an interface every time they need machines to communicate. Since APIs have contracts, applications that want to communicate with each other can be built in completely different ways, as long as they communicate in accordance with the API contract. This allows different developers from different organizations in different parts of the world to create highly-distributed applications, all while re-using the same APIs.

When a user interacts with the front-end of an application (i.e. a mobile app), that front-end makes API calls to back-end systems, simplifying the development process in two main ways:

1. The developer doesn't have to worry about making a customized application for every mobile device or browser.

2. Different backend systems can be updated or modified without having to redeploy the entire application every time.

As a result, developers can save time by focusing an individual service on accomplishing a discrete task, instead of spending time writing extensive logic into their application.

A GOOD EXAMPLE OF STANDARD API USE

Amazon shopping services' documented APIs enable developers to interface with Amazon shopping as they create their applications. The developer can use the Amazon APIs at the appropriate times in their user experience to create a seamless customer journey.

For example, using Amazon APIs might look something like this:

User Experience	Corresponding API Calls
1. Search for a good videogame	1. ItemSearch
2. Amazon suggests Minecraft	2. ItemLookup
3. Add Minecraft to my cart	3. CartCreate 4. CartAdd

In this case, the user interacts with the user interface, while the application interacts with back-end Amazon APIs, as defined by the developer. Everything works very well as long as the underlying APIs behave as expected. However, if the API doesn't behave as expected, there's much cause for concern and, potentially, a significant quality, security, or privacy issue. Thus, the importance of testing these APIs.

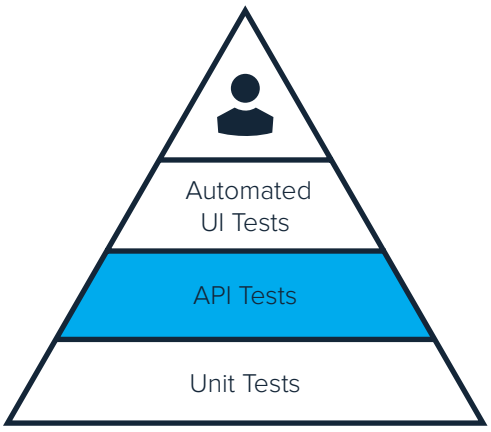
WHY PERFORM API TESTING?

Unlike the user, who interacts with the application only at the UI level, the developer/tester must ensure the reliability of any underlying APIs. Without testing the APIs themselves, developers and testers would be stuck, just like a user, testing the application at the UI level, waiting until the entire application stack was built before being able to start testing.

Fortunately, testers and developers can instead perform API testing of the application at the API level, designing test cases that interact directly with the underlying APIs, and gaining numerous advantages, including the ability to test the business logic at a layer that is easy to automate in a stable manner. Unlike UI testing, which is limited to validating a specific user experience, API testing provides the power to bulletproof applications against the unknown.

HOW TO APPROACH API TESTING

The best way to approach API testing is to build a solid testing practice from the bottom up. To this end, a great way to design a test strategy is to follow Martin Fowler's testing pyramid. The pyramid approach recommends building a wide array of API tests (e.g. contract, scenario, performance, etc.) on top of a solid foundation of unit tests with UI tests. The API tests allow for testing application logic at a level that the unit tests cannot.



These testing strategies are complementary. Testing earlier, at the lower levels of the application, helps to "fail fast and fail early," catching defects early at their source, rather than later in the SDLC. So what kinds of API tests can be implemented? Why are they important? How are they done? This paper addresses those questions and more. We'll reference Parasoft SOAtest, the functional test automation tool that is easy to use and adopt, which helps users create and scale automated API tests of all kinds. The different types of tests done at the API level are enumerated below.

CONTRACT TESTS

An API represents a contract between 2 or more applications. The contract describes how to interact with the interface, what services are available, and how to invoke them. This contract is important because it serves as the basis for the communication. If there's something wrong with the contract, nothing else really matters.

The first and most basic type of API tests are contract tests, which test the service contract itself (Swagger, PACT, WSDL, or RAML). This type of test validates that the contract is written correctly and can be consumed by a client. This test works by creating a series of tests that pull in the contract and validate that:

- The service contract is written according to specifications
- A message request and response are semantically correct (schema validation)
- The endpoint is valid (HTTP, MQ/JMS Topic/Queue, etc)
- The service contract hasn't changed

Consider these as the first "smoke tests." Should these tests fail, there's really no reason to continue testing this particular service. The software supplying the service needs fixing or the contract needs amendment or both. Should these tests pass, testing can progress to the actual functionality of the API.

COMPONENT TESTS

Component tests are like unit tests for the API – take the individual methods available in the API and test each one of them in isolation. These tests are created by making a test step for each method or resource that is available in the service contract.

The easiest way to create component tests is to consume the service contract and let it create the clients, then data-drive each individual test case with positive and negative data to validate that the responses that come back have the following characteristics:

- The request payload is well-formed (schema validation)
- The response payload is well-formed (schema validation)
- The response status is as expected (200 OK, SQL result set returned, or even an error if that's what you're going for)
- The response error payloads contain the correct error messages
- The response matches the expected baseline. This can take two forms:
    - Regression/diff - the response payload looks exactly the same from call to call (a top-down approach where you essentially take a snapshot of the response and verify it every time). This can also be a great catalyst to identify API change (more about that later).
    - Assertion - the individual elements in the response match your expectations (this is a more surgical, bottom-up approach targeted at a specific value in the response).
- The service responds within an expected timeframe

These individual API tests are the most important tests to be built because they leverage all of the subsequent testing techniques. Why rebuild test cases when it's possible to simply reference these individual API calls in all of the different types of tests going forward? This not only promotes consistency but also simplifies the process of approaching API testing.

SCENARIO TESTS

Scenario testing tends to be what most people think about when they think about API testing. In this testing technique, the individual component tests are assembled into a sequence, much like the example described above for the Amazon service. There are two techniques for obtaining the sequence:

1. Review the user story to identify the individual API calls that are being made.

2. Exercise the UI and capture the traffic being made to the underlying APIs.

Scenario tests determine if defects might be introduced by combining different data points together.

Consider the following example: A company employs a series of services to call a customer's financial profile, available accounts, credit cards, and recent transactions. Each of these API calls work individually, but when put together in a sequence, the tests fail. A likely reason for the failure in this case would be an API call within the sequence that's providing different results from one API call to the next. Unlike unit testing or smoke testing, which could easily miss this type of behavior, scenario testing will help pinpoint the problem.

Another benefit of scenario testing is the ability to validate expected behavior when APIs are being used in unexpected ways. When an API is released, a series of building blocks are being provided for the world to use. There may be prescribed techniques for combining these blocks together, but customers can have unpredictable desires, and unexpectedly combine APIs together to expose a defect in an application. To safeguard against this, it's recommended to create as many scenario tests as possible, with different combinations of APIs, to bulletproof an application against a critical breakdown.

Since the component tests form the backbone of the scenario tests, an organization usually has a wide number of scenario tests. They are built when a new functionality is introduced, to model the customer's journey for the new feature. Creating scenarios for new functions as they are added reduces the amount of time spent on testing because only tests for the new functionality will be built, and there will be a reliable library of underlying tests to catch anything unexpected.

PERFORMANCE TESTS

Performance testing is usually relegated to the end of the testing process, in a performance-specific test environment. This is because performance testing solutions tend to be expensive, require specialized skillsets, and require specific hardware and environments. This is a big problem because APIs have service level agreements (SLAs) that must be met in order to release an application. Waiting until the very last moment to do performance testing often leads to failures to meet the SLAs, which in turn causes huge release delays.

Doing performance testing earlier in the process allows for the discovery of performance-related issues early in the software development lifecycle. Following the recommendations of this paper, performance testing will be relatively easy because of all of the underlying test cases already created.

For example, you can take the existing scenario tests, load them up into Parasoft SOAtest, and run them with a higher number of users. If these tests fail, the failure can be traced back to the individual user story, with a better

level of understanding for what is affected. Managers can then use this understanding to make a go or no-go decision about releasing the application.

SECURITY TESTS

Security testing is important to all stakeholders in your organization. If a security vulnerability is exposed and exploited, it can lead to significant reputation loss and financial penalties. Much like how a user can accidentally use your APIs in ways you wouldn't expect, a user can also intentionally try to exploit an API, otherwise known as a hacker. A hacker can attack an API, discover vulnerabilities, and take advantage of them.

To safeguard against this type of behavior, it's important to build test cases that attempt to perform these types of malicious attacks. Using Parasoft SOAtest, you can leverage existing test cases to do so, because a scenario test provides an attack vector into the application. These tests can be re-used to penetration attacks. A good example of this is combining different types of parameter fuzzing or SQL injection attacks with scenario tests. That way, any changes that propagate through the application are picked up by these security tests.

OMNI-CHANNEL TESTS

Modern applications interact with a plethora of interfaces (e.g., mobile, web, databases, etc.), and there will be gaps in test coverage if each channel is tested in isolation, missing the subtleties of the complex interactions between these interfaces.

This is where "omni-channel" tests come in. Omni-channel tests refer to tests that comprehensively cover the application's many interfaces to ensure thorough test coverage, by interweaving API and database tests into the validation of mobile and web UI interactions. This means taking a test that is exercising one of the interfaces and coordinating it with another. For example, executing UI tests such as the web interface or the mobile app interface and interlacing these with existing API or database tests, exchanging data points from the system through the test execution. With effective omni-channel testing, you can create stable, reusable test cases that are easily automated.

MANAGING CHANGE

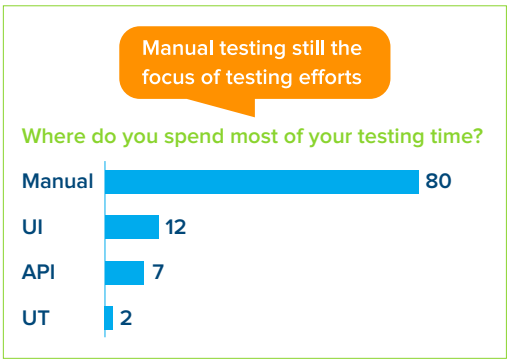
Change is one of the most important indicators of risk in a development project. Change can occur in many forms, including:

- Protocol message format change for a service
- Elements added or removed from an API
- Underlying code change affecting the data format returned
- Re-architecture of a service to break it down into multiple parts (extremely prevalent as organizations move to microservices)

As changes occur, you must build test cases to verify the functionality of these changes. You can use a data-driven approach, leveraging the smart analytics from Parasoft DTP, which integrates into Parasoft SOAtest and the rest of your testing tooling, to understand the true impact of change and target the specific tests that are affected.

BARRIERS TO API TESTING ADOPTION

Automated API testing gives teams the ability to easily test an application at the earlier stages of development, while providing an effective communication mechanism between developers and testers that is highly resistant



to change. Organizations that adopt API testing as a fundamental piece of their testing strategy can leverage the agility they provide to overcome their testing challenges.

But even with all of the benefits that come from API testing, the industry is still largely focused on UI testing. Ostensibly, this is because testers believe they don't know how to test APIs and/or don't know how their application is using the APIs. It's not necessarily apparent where to get started with API testing an application, and understanding how to assemble all of the "puzzle pieces" together in a meaningful way requires domain knowledge of the application.

Since organizations still tend to leverage a centralized testing practice, testers need intimate knowledge of all the different application interfaces and know how to stitch them together properly. It's not a trivial task.

Add to this the fact that API testing is still considered no man's land. In a recent survey, we asked a series of developers and testers to tell us who is responsible for API testing in their organization.

- 70% of testers said that **development** is responsible for API testing, since developers are also responsible for creating the APIs and should therefore build the API tests to validate that they work as described.

- 80% of developers said that **test** is responsible for API testing, because they specifically created the APIs to be externally facing, documented them with a service contract, and the testing team should be able to come in and validate that the APIs work as described.

Because of this disconnect about who is ultimately responsible for API testing, we see low API test coverage and automated API testing adoption. Solving this problem, therefore, is about connecting those dots.

Testing at the API level, manually, requires specialized skills and tools in order to get comprehensive test coverage. It's not intuitive. There are tools that exist in the market that are trying to help organizations build an API testing strategy, but the vast majority of them require a high degree of technical expertise to build comprehensive API tests. Additionally, testers still need to understand how the APIs work, which requires domain knowledge. As a result, organizations tend to do the bare minimum for API testing, which is counter to where agile teams need to go.

ARTIFICIAL INTELLIGENCE FOR TEST AUTOMATION

The only way to solve this industry problem is to build tools that take the complexity out of API testing. Parasoft has been developing software testing automation tools for three decades, and in that time, amassed a lot of data to understand what's required for building a comprehensive API test. Today, artificial intelligence is helping to leverage that expertise to simplify the challenge of API testing for testers across the industry.

Patterns observed in API dataflow form the basis of the decision making and pattern matching algorithms used to group associated API calls together. The decision making "brain" is based on years of experience that has been translated into heuristics to detect these API patterns as scenarios. Over time, these heuristics are trained in order to improve the pattern matching capabilities. Using sophisticated heuristics to automatically detect API test scenarios greatly simplifies API test creation.

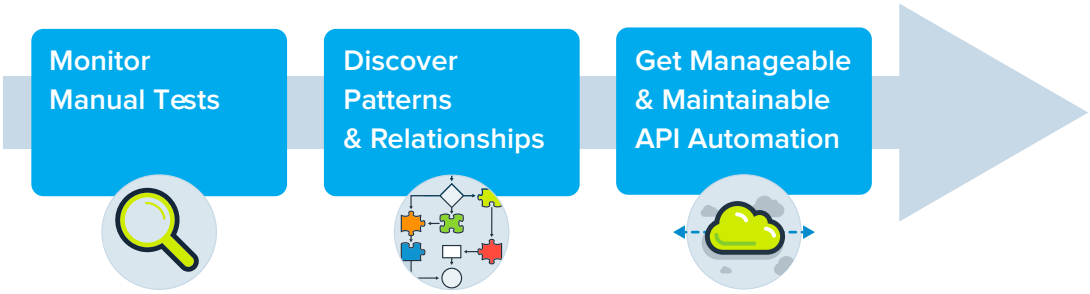
THE SMART API TEST GENERATOR

Parasoft SOAtest's Smart API Test Generator was built from the ground up to make it easier to adopt API testing. The Smart API Test Generator captures traffic during UI tests and feeds it to the artificial intelligence engine, to translate manual testing activities into meaningful, automated API testing scenarios. Deployed by a simple plug-in for Chrome, the Smart API Test Generator lowers the technical skills required to adopt API testing and helps organizations build a comprehensive API testing strategy that scales.

HOW DOES THE SMART API TEST GENERATOR WORK?

The Smart Generator monitors background traffic while testers are executing manual tests, analyzes that traffic, and uses artificial intelligence to automatically build not just a series of API tests, but a meaningful set of API test scenarios. The artificial intelligence engine discovers patterns and analyzes the relationships between them, so it can generate these complete API testing scenarios.

To avoid the difficult activities associated with manually building API tests, the Smart Generator does all of the heavy lifting automatically, based on activity it observes while the tester is using the UI. This helps novice users get a greater understanding of API testing in general because they can map the activities they performed in the UI to the API tests that were created, and build a greater understanding of the relationship between the UI and the underlying API calls, helping drive future API testing efforts.



To help organizations continue to adopt and scale a comprehensive API testing practice, Parasoft SOAtest provides visual tools that are easy to use, enabling API testing beginners to start creating powerful API scenarios in less time than with other tools. The Smart Generator bridges the gap, bringing novice users into the API testing world.

CONCLUSION

Agile development helps organizations deliver quality software to market faster, but without needed technology to help organizations fully test their applications at speed, the risks associated with accelerated delivery erode Agile's potential benefits. Now is the time for organizations to get smart about API testing. Start to leverage artificial intelligence via SOAtest's Smart API Test Generator to bring in a manageable, maintainable, and scalable API testing strategy.

ABOUT PARASOFT

Parasoft helps organizations perfect today's highly-connected applications by automating time-consuming testing tasks and providing management with intelligent analytics necessary to focus on what matters. Parasoft's technologies reduce the time, effort, and cost of delivering secure, reliable, and compliant software, by integrating static and runtime analysis; unit, functional, and API testing; and service virtualization. With developer testing tools, manager reporting/analytics, and executive dashboarding, Parasoft supports software organizations with the innovative tools they need to successfully develop and deploy applications in the embedded, enterprise, and IoT markets, all while enabling today's most strategic development initiatives — agile, continuous testing, DevOps, and security.

www.parasoft.com

Parasoft Headquarters:
+1-626-256-3680

Parasoft EMEA:
+31-70-3922000

Parasoft APAC:
+65-6338-3628

PARASOFT®
Automated Software Testing