API Testing: Challenges and Best Practices

Who invented the light bulb?

Thomas Edison is usually the first name that comes to mind…but he didn't actually invent the incandescent light bulb. Rather, he extended the contributions that other inventors made over three quarters of a century. Nevertheless, he garners all the glory because he's the one who made the light bulb practical for everyday use in real-world environments.

In much the same way, APIs are now getting all the glory for revolutionizing software development—with little recognition of the fact that the foundation was established years ago by the concept of SOA and web services. Without this foundation, today's API-driven development environment could not exist. APIs takes the flexible "building block" application-building approach that SOA targeted for internal or business-to-business usage and extends it to a broad base of potential API consumers—known or unknown—across the world-wide development ecosystem.

This extreme exposure of services as APIs undeniably brings with it a new set of challenges. For example:

- A significantly increased attack surface area.
- High potential for unintentional or malicious misuse.
- The need to provide API consumers a test environment.
- The challenge of validating whether performance will satisfy SLAs in the event of the erratic or surging demand commonly associated with API exposure.

However, considering APIs' heritage, it makes little sense to approach API testing as a brand new beast. From a technical perspective, API testing is not a new skill. Many proven strategies for testing SOA are equally effective for API testing—and are actually extremely well-poised to address the challenges that surface as APIs expose services to the outside world.

This paper explores the difference between API testing and SOA testing, then outlines how many of the core tenets of SOA testing can be leveraged and extended to ensure that your APIs are built to survive in the extreme conditions they will face "in the wild."

4 Unique Challenges with API Exposure

API exposure gives rise to several unique challenges…

1. Broader Attack Surface Area

Simply exposing an API through an internal infrastructure undeniably increases your application's attack surface area from a security perspective. You could be vulnerable to API-level attacks (injections, payload-based attacks, etc.) as well as exploits that take advantage of ineffective authentication, encryption, and access control. The challenge of broader attack surface area is compounded if your API is hosted by a public cloud service. In traditional computing, you're aware of (and have full control over) the parameters of the network security. With cloud services, this level of control is significantly diminished. If you're now leveraging a third-party set of services, the onus is upon you to ensure that your APIs will provide the level of security that your organization expects.

2. Elevated Potential for Unexpected Misuse

Considering the range and number of people who will have access to your published APIs, it's virtually inevitable that they will be exercised in a number of unexpected ways: both by people innocently using them in ways you never anticipated and by attackers maliciously trying to exploit them. In the early days of SOA—when services were exposed internally through controlled networks—you could be fairly certain that your services would be used by colleagues or partners who were familiar with their intended uses cases. Now, when you expose an API to the public, you surrender all control and certainty over how those APIs are consumed.

3. Exceptionally Unpredictable Demand

Like SOA, APIs typically need to meet established performance SLAs. However, validating performance vs. SLAs is more complicated with APIs because it's so difficult to predict how and when your API might be accessed. Thus, it's important to validate performance SLAs against a broad range of performance scenarios, including the sudden surges that could occur if your API garners unexpected attention. Moreover, if you're testing an API that interacts with additional layers of services, the potential for variable or unacceptable performance increases exponentially. This makes it both more critical —and more challenging—to execute a robust set of performance testing scenarios and pinpoint whether the system under test satisfies expectations.

4. Potential API Consumers Need Test Environments

To promote widespread adoption of your APIs, it's often desirable to provide consumers test environments (a.k.a. sandboxes) that enable them to develop and test against your services without zero impact on your production system. Such test environments are also critical when you want to jumpstart adoption by allowing integration to begin before your API is actually implemented or before new features are fully completed. Another key driver for providing a test environment is to give consumers easy access to the broad range of behavior, data, and performance profiles they might want to access for testing purposes.

5 API Testing "Must Haves"

Ensuring that your APIs are delivering the necessary level of security, reliability, and performance that's vital to success in today's API ecosystem inevitably involves developing, continuously executing, and religiously maintaining a broad array of complex tests. Following are several key API testing "must haves" that will help you achieve those goals in light of the above challenges.

1. Intelligent Test Creation and Automated Validation

Exposing services as APIs without proper hardening is like bringing your child to a "developing" country without first obtaining the recommended immunizations. Fortunately, the tools and strategies for achieving the proper hardening are an extension of those that have proven effective for SOA testing—just as the process of administering the immunizations required for foreign travel is essentially an extension of administering the recommended domestic ones.

As the extreme exposure and misuse potential of APIs makes testing a broad range of conditions and corner cases critical, automation comes to the forefront. The creation and execution of simple automated tests with limited or manual validation might have sufficed given the internal scope of SOA, but more sophisticated and extensive automation is required to be confident that your APIs are robust enough to survive in the wild. You need a level of automation that gives you a comprehensive set of functional test cases that can be repeated in a systematic manner.

Recommended capabilities for this goal include an intuitive interface for automating complex scenarios across the messaging layer, ESBs, databases, and mainframes:

- Defining automated test scenarios across the broad range of protocols and message types used in APIs: REST, WADL, JSON, MQ, JMS, EDI, fixed-length messages, etc.
- Automating rich multilayer validation across multiple endpoints involved in end-to-end test scenarios.
- Parameterizing test messages, validations, and configurations from data sources, values extracted from test scenarios, or variables.
- Defining sophisticated test flow logic without requiring scripting.
- Visualizing how messages and events flow through distributed architectures as tests execute.

These are all capabilities that should—or at least could—have been applied to SOA testing. In fact, most of these capabilities were invented, tested, and refined in the context of SOA testing. APIs—with their extreme exposure and myriad opportunities for misuse—brings us to the tipping point that makes these automated testing and validation capabilities a “must have” for organizations serious about delivering APIs that satisfy user needs and expectations.

2. Change Management for Test Assets and Environments

Continuously evolving APIs helps organizations stay a step ahead of the competition while responding to business demands. Yet, this frequent change presents significant quality risks if the automated test suite fails to keep pace with the evolving API.

A system for fast, easy, and accurate updating of test assets is critical for keeping test assets in sync with the changing API. If you can automatically assess the impact of changes to existing tests and then quickly update existing tests (or create new ones) in response to the identified change impacts, you can vastly reduce the amount of time required to ensure that your tests don’t fail due to expected changes…or overlook critical new functionality.

3. Service Virtualization for Simulated Test Environments

Service Virtualization technology creates simulated test environments that provide anytime, anywhere access to the behavior of dependent resources that are unavailable, difficult to access, or difficult to configure for development or testing. “Dependent resources” might include mainframes, mobile app front-ends, databases, web services, third-party applications, or other systems that are out of your team’s direct control. Service virtualization can be used in conjunction with hardware/OS virtualization to access the environments you need to test earlier, faster, or more completely.

In the context of API testing, service virtualization can be applied in two key ways:

- To provide access to the dependent resource behavior (e.g., from a mobile app, database, legacy system, or third-party service) that you need in order to thoroughly validate your API.
- To mimic the behavior of your APIs, creating a test environment that API consumers can develop and test against without impacting your production environment—or to enable development and testing to occur before your APIs are even completed.

4. Extensive Performance Testing—Ideally, with Service Virtualization

Due to the highly-exposed nature of APIs, there's a high potential for unpredictable and often volatile traffic volumes. To determine whether your API will satisfy SLAs in the event of the erratic or surging demand that APIs commonly face, it's essential to ramp up the scope of performance testing. You can use service virtualization (covered above) to create simulated test environments that help you test against different performance scenarios that would otherwise be difficult to create in the test environment.

For instance, you can easily set performance conditions (e.g., timing, latency, delay) to emulate peak, expected, and slow performance—perhaps to help you plan for cloud bursts or determine how the API might respond when someone is accessing it from China. You can also configure various error and failure conditions that are difficult to reproduce or replicate with real systems—for instance, if your APIs rely on Amazon Web Services, you can easily simulate a scenario where AWS is down. This ability to rapidly configure a broad range of conditions in dependent systems is essential for determining if your APIs provide reasonable responses—or at least fail gracefully—under exceptional conditions.

One final way that adopting service virtualization helps performance testing: you can "virtualize" any connections to third-party systems, reliably eliminating the risk that your stress tests might impact services you aren't permitted (or budgeted) to barrage with test messages.

5. Extensive Security Testing—Ideally, with Service Virtualization

Considering APIs' increased attack surface area, a multi-faceted security testing strategy is essential for ensuring that development has built the appropriate level of security into your application. This includes:

- Executing complex authentication, encryption, and access control test scenarios.
- Generating a broad range of penetration attack scenarios involving parameter fuzzing, injections, large payloads, etc.
- Running penetration attack scenarios against your existing functional test scenarios.
- Monitoring the back-end during test execution in order to determine whether security is actually compromised.

In addition, if you're adopting service virtualization (covered above) you can leverage it to take your security testing to the next level:

- It provides rapid ways to emulate attack scenarios as well as emulate different security behaviors of dependencies. This lets you derive more value from your existing functional test scenarios (since you can run them vs. different security scenarios that would otherwise be difficult to configure and unfeasible to test against).

- It enables extensive security testing to be performed without a security expert. Existing test scenarios can be easily executed against a broad set of preconfigured security scenarios.
- It helps you isolate and zero in on your APIs response to various attack scenarios and different security behaviors of dependencies.

About Parasoft

For 25 years, Parasoft has researched and developed software solutions that help organizations define and deliver defect-free software efficiently. By integrating Development Testing, cloud/API testing, and service virtualization, we reduce the time, effort, and cost of delivering secure, reliable, and compliant software. Parasoft's enterprise and embedded development solutions are the industry's most comprehensive—including static analysis, unit testing, requirements traceability, functional & load testing, dev/test environment management, and more. The majority of Fortune 500 companies rely on Parasoft in order to produce top-quality software consistently and efficiently. For more information, visit the Parasoft web site and ALM Best Practices blog.

Contacting Parasoft

USA
101 E. Huntington Drive, 2nd Floor
Monrovia, CA 91016
Toll Free: (888) 305-0041
Email: info@parasoft.com
URL: www.parasoft.com

Europe
France: Tel: +33 (1) 64 89 26 00
UK: Tel: + 44 (0)208 263 6005
Germany: Tel: +49 731 880309-0
Email: info-europe@parasoft.com

Other Locations
See http://www.parasoft.com/contacts

Author Information

This paper was written by:

- Wayne Ariola (wayne.ariola@parasoft.com), VP of Strategy at Parasoft
- Cynthia Dunlop (cynthia.dunlop@parasoft.com), Lead Technical Writer at Parasoft