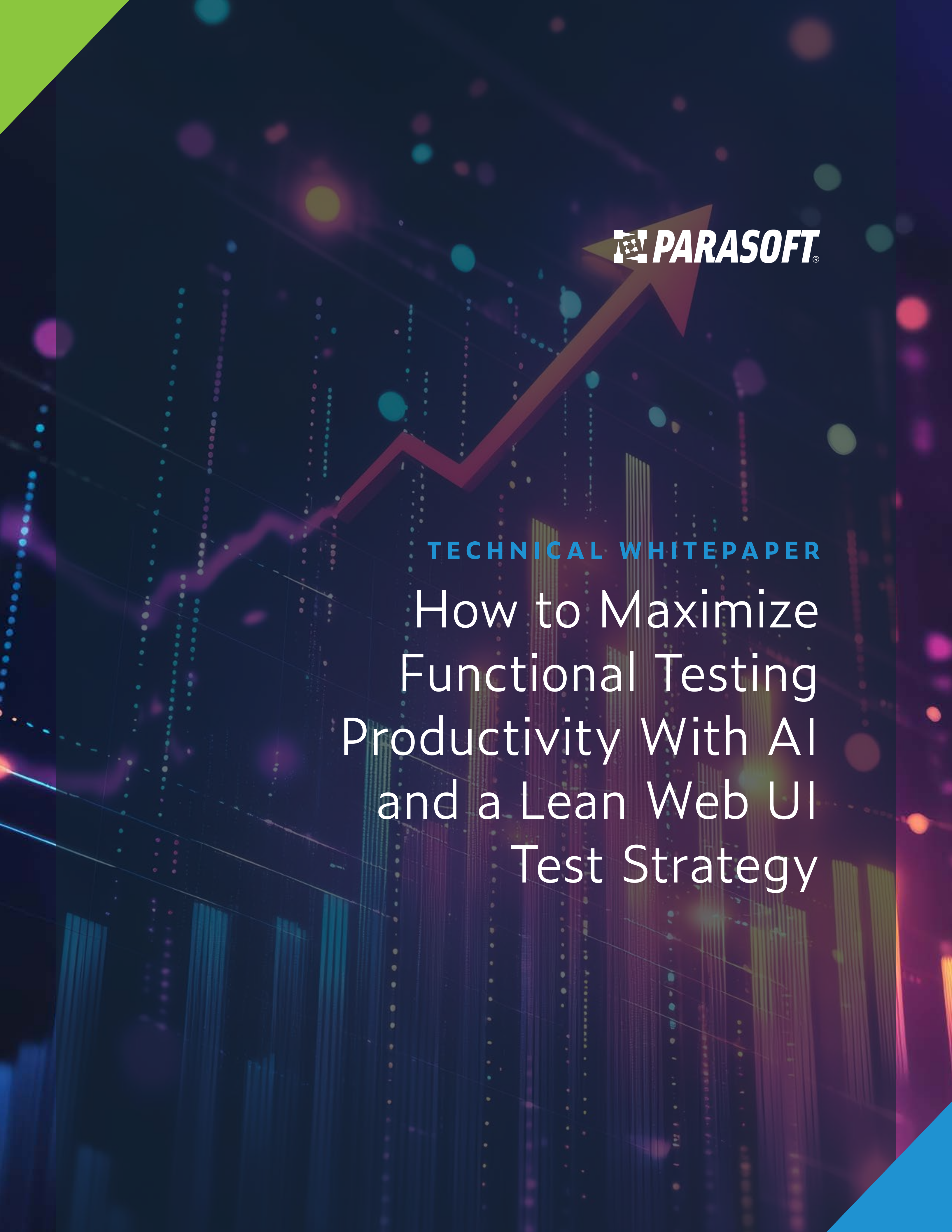


TECHNICAL WHITEPAPER

How to Maximize Functional Testing Productivity With AI and a Lean Web UI Test Strategy

Overview

As applications become more complex with underlying microservices and distributed architectures, testing them becomes more challenging. Each use case or test scenario increases in complexity to achieve end-to-end validation.

Today, many organizations still rely on web UI testing as the primary form of validation. However, as applications modernize and move to microservice architectures or highly distributed systems, efficient complements to web UI testing with its scalability challenges offer big productivity boosts in functional test strategies.

Teams must look beyond web UI testing and focus more testing efforts on the API layers to achieve a scalable and maintainable test automation strategy that keeps in-sprint testing velocity in focus.

How the Limitations of Web UI Testing Impact Team Velocity

When it comes to adopting a test automation strategy, there are usually four primary business goals that motivate the investment to adopt test automation.

1. Increase the speed of software delivery.
2. Increase the quality of software.
3. Reduce costs.
4. Reduce risks associated with software releases.

Web UI testing helps validate end-to-end functionality, cross-browser compatibility, client-side logic, and the user experience. However, it does have limitations that can impact a QA team's testing velocity and ability to scale as applications under test mature, including the following:

- Application changes have a high impact on web UI tests, making them brittle and high test maintenance burdens.
- Web UI tests require a significant amount of time to execute, resulting in long regression runs and delayed testing feedback to development.
- Issues are difficult to diagnose, which slows defect remediation and reduces team velocity.
- Use cases are time-consuming and difficult to create, leading to test complexity challenges.

Limitations often become more pronounced as application teams try to scale their automated testing practices, impacting their ability to reach goals around increasing speed and quality while reducing costs and risks.

High Test Maintenance Burdens

Application teams that depend too heavily on web UI testing for functional validation face a high test maintenance burden since tests break frequently when development releases new versions and the application changes. When first starting out, this may not seem like a big problem, but as the test suite grows, QA teams will need to allocate a lot of time to diagnose test failures and update tests that have broken due to application change.

To address the increasing workload and maintain the same level of testing productivity, management must do one or more of the following:

- **Increase staffing numbers to keep up with testing requirements.** However, expanding the team through hiring is both time-consuming and costly. In addition to factoring in the salary or hourly rate of the new team members, there are also costs associated with training, tech stack, and internal processes.
- **Release with lower confidence and risk more defects in production.** This is a judgement call for the application owner's tolerance for risk and is not an option for business-critical or safety-critical applications.

» **Delay the release to allocate more time for testing.** There can be real business ramifications involved in delaying releases. It puts extra pressure on the team to catch up. Delaying the time to market can impact customer satisfaction and increase project costs. It also impacts longer-term roadmap plans, which affects the development team's ability to frequently iterate and keep up with changing requirements.

While there are some commercial solutions on the market today that include capabilities to heal or mitigate the test maintenance burden, many open source and commercial solutions do not offer these capabilities. Without them, scalability challenges are a common factor for QA teams that overrely on UI testing as a functional testing strategy.

Long Regression Runs and Slow Testing Feedback

When teams first start their test automation journey, the time it takes to execute web UI tests may not have a heavy impact on the velocity of testing cycles. However, when the test suite has grown to thousands of test cases, regression testing is going to take considerably longer to complete. This, combined with frequent UI changes, means it takes longer than one would expect to run a full, automated regression suite that accounts for maintenance issues and re-running tests that need to be stabilized. This leads to

» Delays in providing feedback to development.

» Slower defect remediation.

» Increased project costs.

» Decreased developer productivity due to context switching.

Slow Defect Remediation

The primary goal of testing should always be to find and fix defects as quickly as possible. However, with UI testing, when an issue is found, it can be difficult to pinpoint the exact location of the defect in the application. This can cause a slowdown, as development must first succeed in reproducing the issue before they can identify the root cause to remediate. This impacts the time to remediation, resulting in a higher cost of fixing the defect.

Test Complexity Challenges

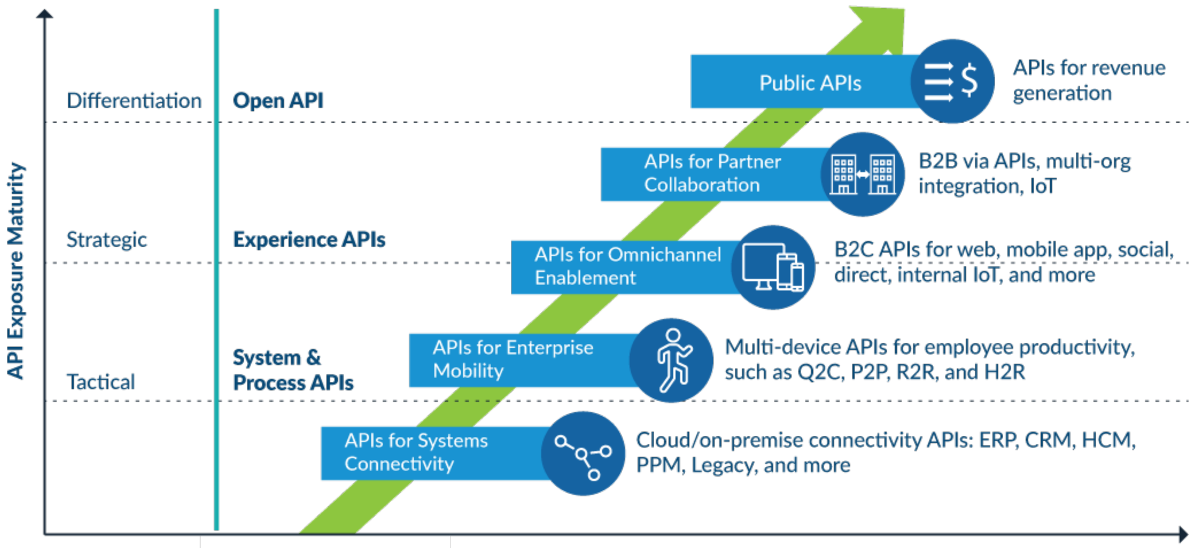
As applications and websites today become more feature-rich, distributed, and interconnected, the necessary tests will also become more complex and difficult to automate. This leads to scalability challenges where end-to-end tests can take hours to create and maintain. To mitigate these challenges, testing teams need to expand beyond web UI testing and focus more of their efforts on API test automation, adopting a lean web UI test strategy where the underlying business logic of the application is validated through the APIs and web UI testing is narrowly focused on validating the presentation and interactions on the user interface.

How Modern API Architecture Is Changing Testing Strategies

When it comes to expanding beyond web UI testing, it's important to understand where most modern software architectures are heading, the increased reliance on APIs, and the different ways they're being used.

There are three primary types of API exposure maturity.

1. **System and process APIs.** Fundamental layers that expose core assets and business logic via a consistent contract. System APIs represent all the functions, features, and resources a system exposes to outside clients, while process APIs represent middleware and orchestration of functions across multiple systems. Using an API's contract, thorough testing should be conducted against these endpoints to ensure proper functionality, security, and performance before advancing to a higher scope end-to-end testing.
2. **Experience APIs.** Sometimes called frontend or browser-based, this type of API can be thought of as a "functionality wireframe" for the resulting user experience in the UI. It's the Experience API that delivers dynamic chunks of data that allow frontend engineers to enrich the user experience while not requiring a page reload. Popular frontend frameworks like Angular and React support this paradigm of UI/UX built on top of Experience APIs. These APIs are often undertested or assumed to be (indirectly) tested by UI tests, which leads to gaps in test coverage and difficulty keeping automated regression testing caught up within each sprint. Within modern system architectures, Experience APIs open the door for higher testing velocity while regression testing end-to-end use cases.
3. **Open APIs.** The final level of API exposure maturity is where you publicly document and encourage public use of your software. Many successful businesses have been born out of an open API monetization strategy, and generating revenue with use cases like partner integration also necessitates comprehensive API testing.


API Exposure Maturity
Differentiation
Open API
Strategic
Experience APIs
Tactical
System & Process APIs
Public APIs
APIs for revenue generation
APIs for Partner Collaboration
B2B via APIs, multi-org integration, IoT
APIs for Omnichannel Enablement
B2C APIs for web, mobile app, social, direct, internal IoT, and more
APIs for Enterprise Mobility
Multi-device APIs for employee productivity, such as Q2C, P2P, R2R, and H2R
APIs for Systems Connectivity
Cloud/on-premise connectivity APIs: ERP, CRM, HCM, PPM, Legacy, and more

Sources: Panagiotis Kriaris, Capgemini & Mulesoft

Expanding beyond web UI testing to increase testing productivity isn't referring to testing system, process, or open APIs. It's a given that these APIs must have thorough regression tests. As a "functionality wireframe," Experience APIs represent the underlying end-to-end business logic of the application. By shifting the focus in functional testing strategy from the web UI layer to the Experience API layer, teams can:

» Increase testing productivity.

» Maintain Agile development velocities.

» Create a more scalable and maintainable test automation strategy.

What Is a Lean Web UI Test Strategy?

QA teams can adopt a more maintainable and scalable test automation strategy by refocusing a substantial amount of their testing efforts on the API layers. Web UI testing is still a crucial part of quality assurance and vital to ensuring a positive user experience. However, due to web UI testing scalability barriers, teams must become more strategic in their testing practice with a lean web UI test strategy.

A lean web UI test strategy urges QA teams to think critically about what type of test to create to validate functionality or meet testing requirements. In many cases, web UI tests are the appropriate type of test case. For instance, teams should opt to create a web UI test over an API test in the following scenarios:

» **User interface validation.** If the primary focus is on validating the user interface, including layout, styling, and user interactions, creating a web UI test is appropriate. This is especially important for applications where the user experience is critical, such as customer-facing websites and web applications.

» **Navigation testing.** Web UI testing is valuable in navigation testing, which ensures that the user interface navigation menus function as expected and that links and buttons direct users to the intended destination.

» **Blackbox testing.** In scenarios where the internal implementation details of backend services are not accessible or relevant to the testing objectives, creating web UI tests allows testers to interact with the application as an external user would, without needing knowledge of the underlying APIs.

» **Cross-browser and cross-device testing.** If testing needs to ensure compatibility across different browsers, devices, and screen sizes, then web UI tests are essential for validating the application's behavior in various environments.

» **Input validation and error handing testing.** Check that only valid data can be entered into specific fields and ensure that text fields do not allow inputs to exceed a specific character limit.

Web UI testing plays a crucial role in preserving user satisfaction and loyalty by identifying and rectifying potential issues before they impact end users. Through rigorous testing of the user interface across various browsers and devices, businesses can ensure consistent performance and accessibility, thus maximizing their reach and driving traffic and engagement. The problem lies in overreliance on web UI testing and its challenges around scalability. Teams that practice a lean web UI test strategy practice both API and web UI testing.

Increase Your Productivity: Directly Test Experience APIs

Let's examine the benefits of shifting your test strategy to directly test the Experience APIs that your end users interact with from the browser. API testing has a higher testing velocity than web UI testing. It lends itself to be more scalable and maintainable over time.

Here are some benefits of API testing.

- Lowers test maintenance costs due to API tests being more resilient to application changes than web UI tests.
- Provides faster feedback for development.
- Accelerates test failure diagnostics and remediation processes.
- Facilitates high levels of test automation.
- Scales easily across teams and physical machines.

While web UI tests often break due to application changes and require dedicated engineering time for test maintenance during each sprint, API tests are more resilient to change because APIs change less often. By shifting your testing approach to directly test your applications' Experience APIs versus indirectly testing them with web UI testing, QA teams can amplify efforts by creating more test cases to cover new features and functionality.

Since API testing can begin before the user interface finishes, application teams can start testing much earlier in their development cycles, giving them more time to create tests, and increase test coverage.

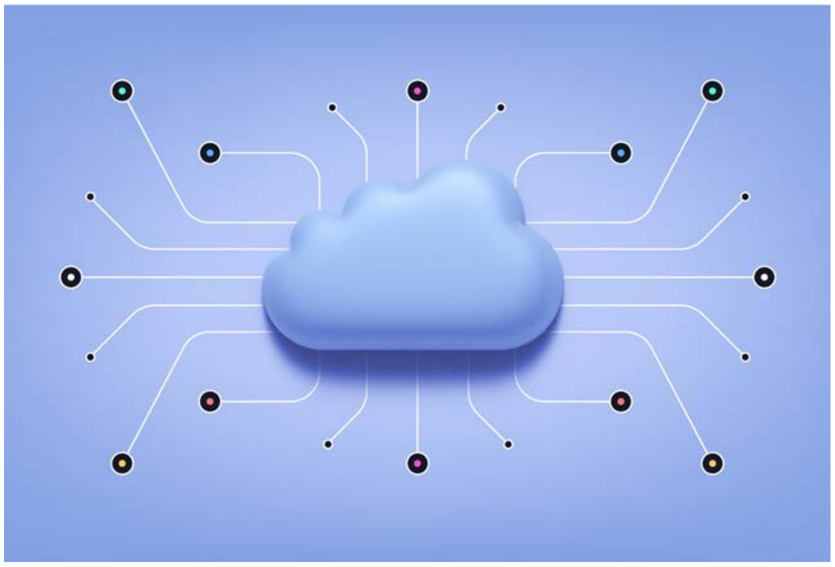
API test cases are also much faster to execute. By adopting a lean web UI and API first testing approach, QA teams can accelerate testing cycles, enabling faster feedback to development. As a result, they stay on track better with delivery schedules, increase the quality of the resulting software, and reduce risks associated with low test coverage or lack of negative testing performed through UI testing only.

As API testing can test applications with and without user interfaces, supports complex architectures and use cases, and allows testing to begin earlier, teams benefit from easy scalability across the applications they're testing. While it can be a cultural change, enabling more QA activities at the API layer allows you to scale test automation more easily while minimizing the typical test maintenance burdens of web UI testing.

Using AI Test Generation to Directly Test Your Experience APIs

With many QA teams having invested heavily in automated web UI testing, the challenge becomes how to get QA teams to adopt API testing for the Experience APIs under their applications' user interface to reap the benefits that API testing offers.

QA teams often struggle with how to effectively test APIs end-to-end because it requires an understanding of the underlying business logic from the frontend of the application. This can be daunting because it's viewed as more technical with an assumption that one must have developer-level knowledge to be successful in testing APIs. While a QA team may recognize the benefits of API testing, API testing often remains a challenge.



This is where AI-powered API test generation comes into play. Solutions like Parasoft SOAtest enable QA teams to leverage their existing web UI test suite, regardless of the test framework, to automatically generate a complimentary suite of API scenario tests with its AI-enhanced Smart API Test Generator.

Teams can more easily transition their functional testing practice so that the underlying business logic of the application is validated with API testing and refocuses web UI testing on validating the interactions and presentation of the user interface. Over time, as old web UI test cases break, QA teams can determine if the test cases are necessary and should be maintained or if the business logic is sufficiently tested with API testing.

Eventually, the size of the web UI test suite will reduce along with the QA team's test maintenance burden. The QA team will prioritize API testing to validate the underlying business logic, while the web UI test suite will strategically focus on validating the user interface elements.

By leveraging AI to create complementary API tests to go along with your web UI testing practice, teams can benefit from the higher ROI that API testing offers and adopt a strategy that scales across their applications, reduces test maintenance burdens, and increases their test velocity for faster software validation and deliveries.

Double Your Test Suite: Leverage Existing Test Cases

QA teams can use Parasoft SOAtest's Smart API Test Generator, an AI-driven solution, to record API traffic with a proxy while performing actions on a UI. By executing existing web UI tests with the Smart API Test Generator's traffic recorder capturing the underlying API calls, teams can potentially double their test suite by codelessly generating a complementary suite of API scenario tests.

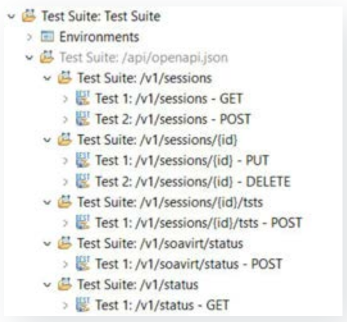
This technology is purpose-built for modern frontends that heavily rely on REST or GraphQL to enrich the UI/UX of their application. While this technical whitepaper focuses on leveraging existing automated UI tests, a Chrome extension is also available to record manual test sessions.

Parasoft offers two ways for application teams to leverage existing web UI test suites to generate complementary API scenario tests that directly test their Experience APIs.

1. Integrate the Parasoft Recorder REST API into the existing UI test framework.
2. Use Parasoft Selenic to automate the injection of the Parasoft Recorder REST API into existing Selenium-based tests during their execution.

How to Integrate Parasoft's Recorder REST API Into Your UI Test Automation Framework

The Parasoft Recorder REST API has a straightforward set of resources that your test framework can call to set up the proxy and recording session. Here's a visual breakdown of the REST API:

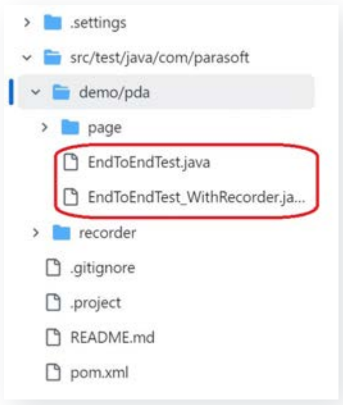


```
Test Suite: Test Suite
  Environments
  Test Suite: /api/openapi.json
    Test Suite: /v1/sessions
      Test 1: /v1/sessions - GET
      Test 2: /v1/sessions - POST
    Test Suite: /v1/sessions/{id}
      Test 1: /v1/sessions/{id} - PUT
      Test 2: /v1/sessions/{id} - DELETE
    Test Suite: /v1/sessions/{id}/tsts
      Test 1: /v1/sessions/{id}/tsts - POST
    Test Suite: /v1/soavirt/status
      Test 1: /v1/soavirt/status - POST
    Test Suite: /v1/status
      Test 1: /v1/status - GET
```

The REST API will return a proxy port that you must leverage with your UI test framework so that all the HTTP traffic flowing through the browser or mobile device on test playback is captured by the proxy.

For a runnable example of a web UI test that integrates the Parasoft Recorder REST API, see the following GitHub repo. This project leverages Selenium in Java but setting up a proxy with the UI is a common feature in most UI test frameworks. This repository can be a useful reference for doing a similar integration regardless of the framework or language used for test automation. For example, both Cypress and Playwright have similar proxy control capabilities:

- » https://docs.cypress.io/guides/references/proxy-configuration
- » https://playwright.dev/docs/network#http-proxy



```
> .settings
v src/test/java/com/parasoft
  v demo/pda
    > page
      EndToEndTest.java
      EndToEndTest_WithRecorder.ja...
  > recorder
  .gitignore
  .project
  README.md
  pom.xml
```

The UI test in our example is testing a web application called Parasoft Demo App, which is a logistics application that Parasoft often uses when demonstrating UI and API testing concepts. The Parasoft Demo App is open sourced and can be deployed locally to give the test a spin for yourself. We've even published a docker image to make deploying it easy.

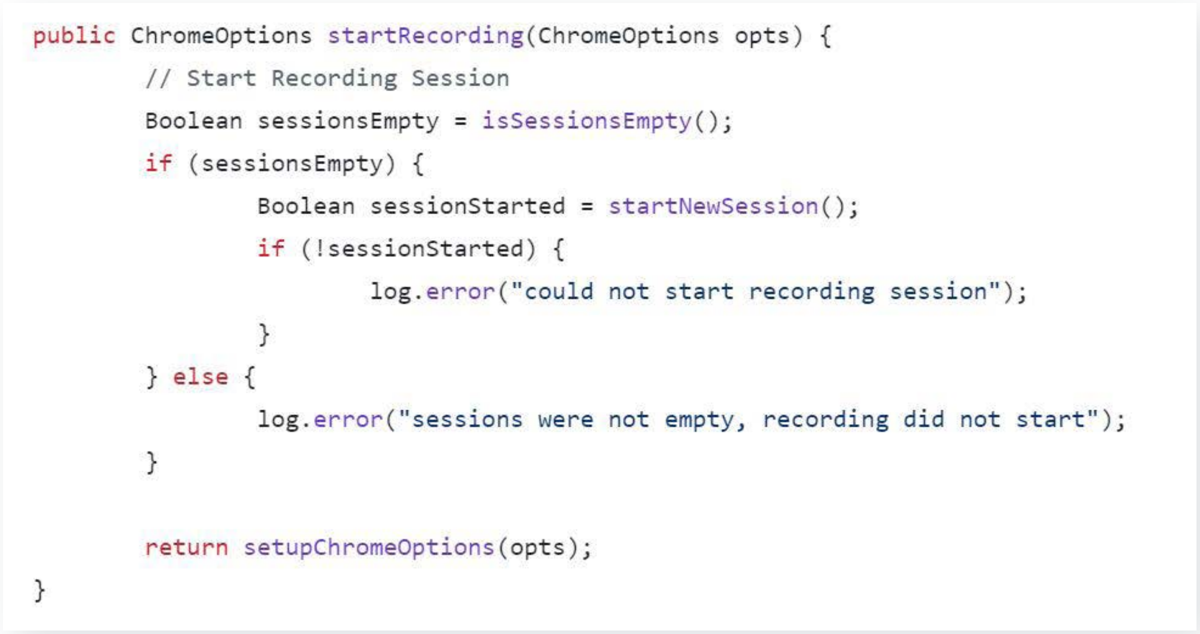
The GitHub repo has two versions of the same test class, an "original" test without integrating the Recorder REST API, and another that does, so you can compare the differences between them.

Before running EndToEndTest_WithRecorder.java, you must have:

- Parasoft Recorder installed.
- Parasoft SOAtest installed with a valid SOAtest Desktop license that includes the “Advanced Test Generation Desktop” license feature enabled.

This example leverages JUnit’s @Before and @After annotations to run some code before and after each test. These beforeTest() and afterTest() methods are where we will call the Parasoft Recorder REST API to set up the test for recording API traffic. The code being called from these methods includes setting up the Selenium WebDriver with a proxy connection from the Parasoft Recorder and is contained in the ParasoftRecorder.java class.

The following method describes example logic for starting a new recording session.



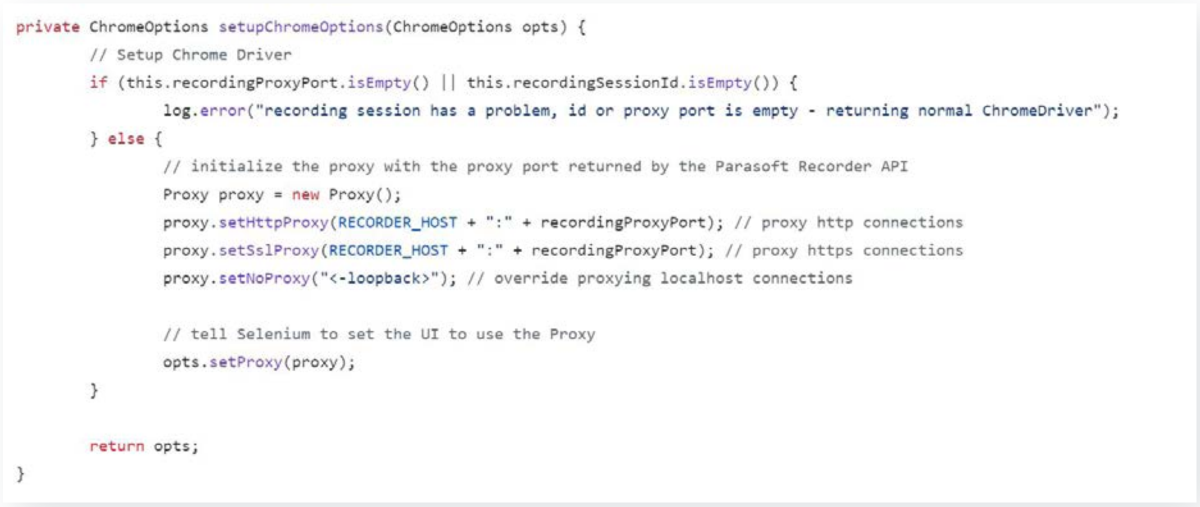
```
public ChromeOptions startRecording(ChromeOptions opts) {
        // Start Recording Session
        Boolean sessionsEmpty = isSessionsEmpty();
        if (sessionsEmpty) {
                Boolean sessionStarted = startNewSession();
                if (!sessionStarted) {
                        log.error("could not start recording session");
                }
        } else {
                log.error("sessions were not empty, recording did not start");
        }

        return setupChromeOptions(opts);
}
```

There are some noteworthy private method calls being made in the logic for starting a recording:

» isSessionsEmpty() – calls the Recorder REST API to see if there are any currently active sessions.

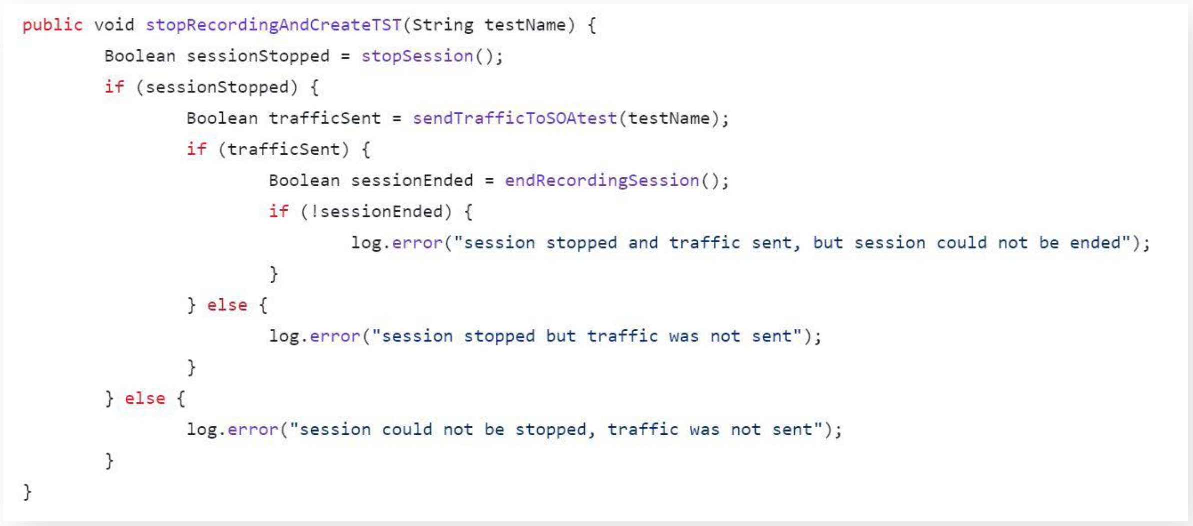
» startNewSession() – calls the Recorder REST API to start a new recording session.

» setupChromeOptions(opts) - demonstrates how to configure a proxy in Chrome with Selenium.



```
private ChromeOptions setupChromeOptions(ChromeOptions opts) {
        // Setup Chrome Driver
        if (this.recordingProxyPort.isEmpty() || this.recordingSessionId.isEmpty()) {
                log.error("recording session has a problem, id or proxy port is empty - returning normal ChromeDriver");
        } else {
                // initialize the proxy with the proxy port returned by the Parasoft Recorder API
                Proxy proxy = new Proxy();
                proxy.setHttpProxy(RECORDER_HOST + ":" + recordingProxyPort); // proxy http connections
                proxy.setSslProxy(RECORDER_HOST + ":" + recordingProxyPort); // proxy https connections
                proxy.setNoProxy("<-loopback>"); // override proxying localhost connections

                // tell Selenium to set the UI to use the Proxy
                opts.setProxy(proxy);
        }

        return opts;
}
```

Then to stop recording, we have the following method:



```
public void stopRecordingAndCreateTST(String testName) {
        Boolean sessionStopped = stopSession();
        if (sessionStopped) {
                Boolean trafficSent = sendTrafficToSOAtest(testName);
                if (trafficSent) {
                        Boolean sessionEnded = endRecordingSession();
                        if (!sessionEnded) {
                                log.error("session stopped and traffic sent, but session could not be ended");
                        }
                } else {
                        log.error("session stopped but traffic was not sent");
                }
        } else {
                log.error("session could not be stopped, traffic was not sent");
        }
}
```

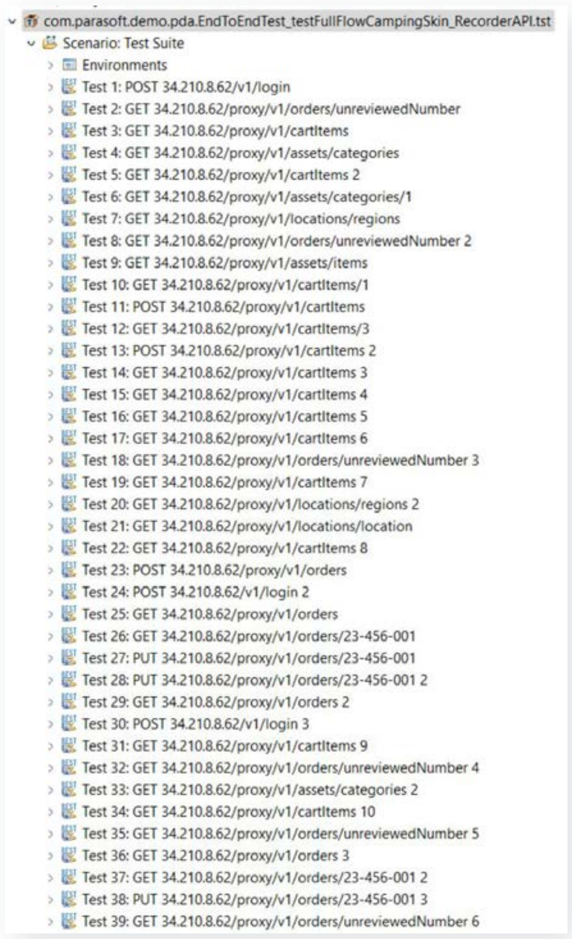
There are some noteworthy private methods being called here as well:

» stopSession() – calls the Recorder REST API to stop the session.

» sendTrafficToSOAtest(testName) – transmits its recorded traffic to SOAtest so its AI can generate an API test.

» endRecordingSession() – calls the Recorder REST API to terminate the recording session.

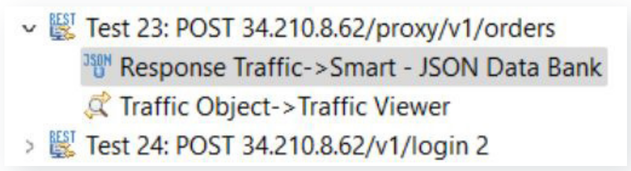
After running EndToEndTest_WithRecorder.java, you will see the following test automatically created in your SOAtest workspace!



```
com.parasoft.demo.pda.EndToEndTest_testFullFlowCampingSkin_RecorderAPI.tst
  Scenario: Test Suite
    Environments
    Test 1: POST 34.210.8.62/v1/login
    Test 2: GET 34.210.8.62/proxy/v1/orders/unreviewedNumber
    Test 3: GET 34.210.8.62/proxy/v1/cartItems
    Test 4: GET 34.210.8.62/proxy/v1/assets/categories
    Test 5: GET 34.210.8.62/proxy/v1/cartItems 2
    Test 6: GET 34.210.8.62/proxy/v1/assets/categories/1
    Test 7: GET 34.210.8.62/proxy/v1/locations/regions
    Test 8: GET 34.210.8.62/proxy/v1/orders/unreviewedNumber 2
    Test 9: GET 34.210.8.62/proxy/v1/assets/items
    Test 10: GET 34.210.8.62/proxy/v1/cartItems/1
    Test 11: POST 34.210.8.62/proxy/v1/cartItems
    Test 12: GET 34.210.8.62/proxy/v1/cartItems/3
    Test 13: POST 34.210.8.62/proxy/v1/cartItems 2
    Test 14: GET 34.210.8.62/proxy/v1/cartItems 3
    Test 15: GET 34.210.8.62/proxy/v1/cartItems 4
    Test 16: GET 34.210.8.62/proxy/v1/cartItems 5
    Test 17: GET 34.210.8.62/proxy/v1/cartItems 6
    Test 18: GET 34.210.8.62/proxy/v1/orders/unreviewedNumber 3
    Test 19: GET 34.210.8.62/proxy/v1/cartItems 7
    Test 20: GET 34.210.8.62/proxy/v1/locations/regions 2
    Test 21: GET 34.210.8.62/proxy/v1/locations/location
    Test 22: GET 34.210.8.62/proxy/v1/cartItems 8
    Test 23: POST 34.210.8.62/proxy/v1/orders
    Test 24: POST 34.210.8.62/v1/login 2
    Test 25: GET 34.210.8.62/proxy/v1/orders
    Test 26: GET 34.210.8.62/proxy/v1/orders/23-456-001
    Test 27: PUT 34.210.8.62/proxy/v1/orders/23-456-001
    Test 28: PUT 34.210.8.62/proxy/v1/orders/23-456-001 2
    Test 29: GET 34.210.8.62/proxy/v1/orders 2
    Test 30: POST 34.210.8.62/v1/login 3
    Test 31: GET 34.210.8.62/proxy/v1/cartItems 9
    Test 32: GET 34.210.8.62/proxy/v1/orders/unreviewedNumber 4
    Test 33: GET 34.210.8.62/proxy/v1/assets/categories 2
    Test 34: GET 34.210.8.62/proxy/v1/cartItems 10
    Test 35: GET 34.210.8.62/proxy/v1/orders/unreviewedNumber 5
    Test 36: GET 34.210.8.62/proxy/v1/orders 3
    Test 37: GET 34.210.8.62/proxy/v1/orders/23-456-001 2
    Test 38: PUT 34.210.8.62/proxy/v1/orders/23-456-001 3
    Test 39: GET 34.210.8.62/proxy/v1/orders/unreviewedNumber 6
```

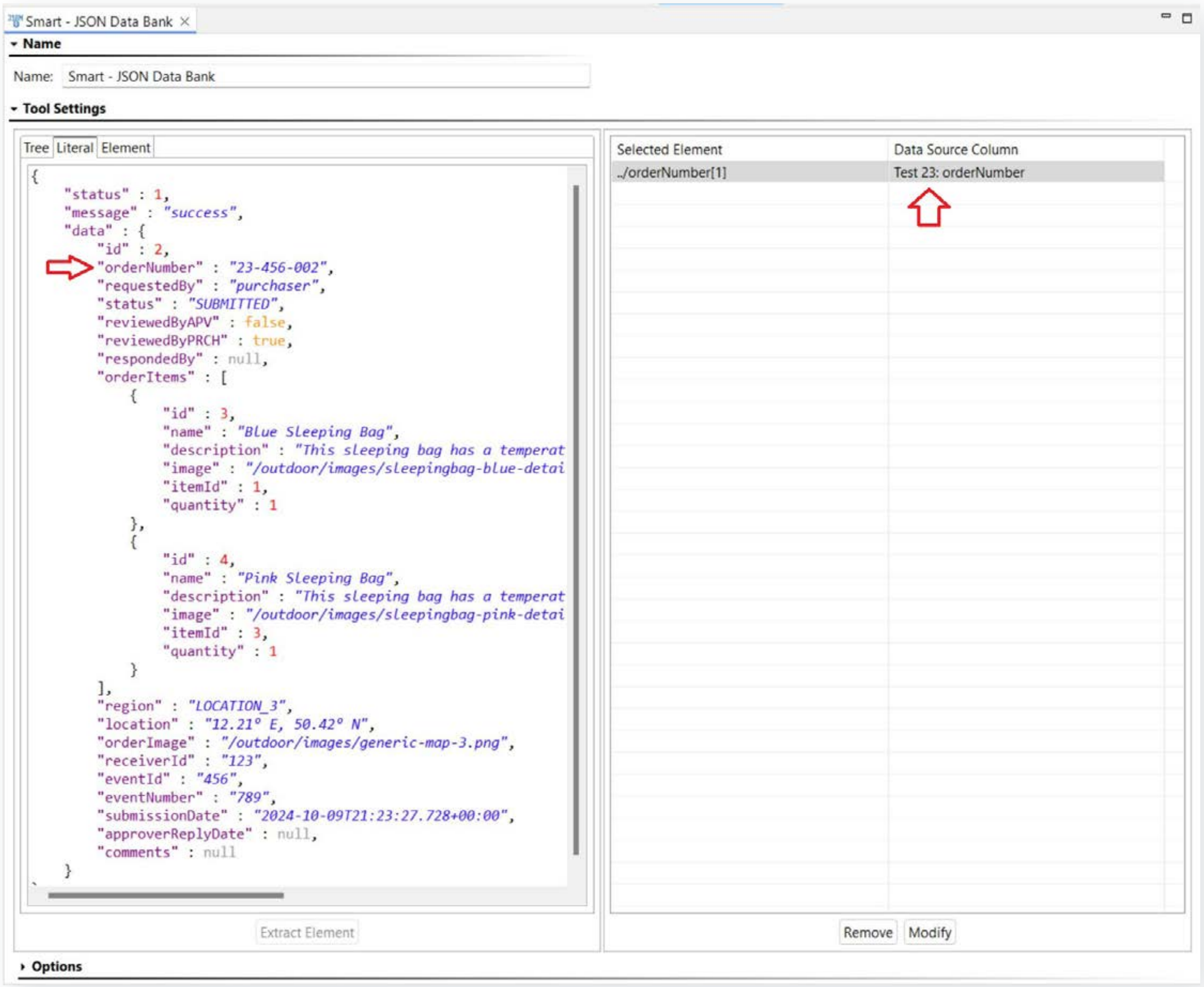
Even a relatively simple UI playback in a responsive web application can lead to many calls to Experience APIs, as shown here with the Parasoft Demo App. Capturing the calls and creating the API test scenario is only part of what this recording workflow does. SOAtest's AI also analyzes all the recorded request/response messages and configures the needed parameterizations for dynamic data that make this long sequence of API calls replay ready!

Notice test step 23 is a POST request that happens right before logging in as a different user in the end-to-end flow.



```
Test 23: POST 34.210.8.62/proxy/v1/orders
  Response Traffic->Smart - JSON Data Bank
  Traffic Object->Traffic Viewer
Test 24: POST 34.210.8.62/v1/login 2
```

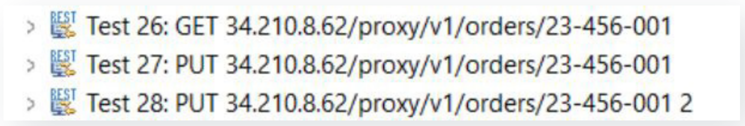
SOAtest's AI figured out that there is data in the response of this API call that is necessary in subsequent test steps. In this case, the AI automatically configured an extraction to grab a new order number.



```
{
    "status" : 1,
    "message" : "success",
    "data" : {
        "id" : 2,
        "orderNumber" : "23-456-002",
        "requestedBy" : "purchaser",
        "status" : "SUBMITTED",
        "reviewedByAPV" : false,
        "reviewedByPRCH" : true,
        "respondedBy" : null,
        "orderItems" : [
            {
                "id" : 3,
                "name" : "Blue Sleeping Bag",
                "description" : "This sleeping bag has a temperat
                "image" : "/outdoor/images/sleepingbag-blue-detai
                "itemId" : 1,
                "quantity" : 1
            },
            {
                "id" : 4,
                "name" : "Pink Sleeping Bag",
                "description" : "This sleeping bag has a temperat
                "image" : "/outdoor/images/sleepingbag-pink-detai
                "itemId" : 3,
                "quantity" : 1
            }
        ],
        "region" : "LOCATION_3",
        "location" : "12.21° E, 50.42° N",
        "orderImage" : "/outdoor/images/generic-map-3.png",
        "receiverId" : "123",
        "eventId" : "456",
        "eventNumber" : "789",
        "submissionDate" : "2024-10-09T21:23:27.728+00:00",
        "approverReplyDate" : null,
        "comments" : null
    }
```

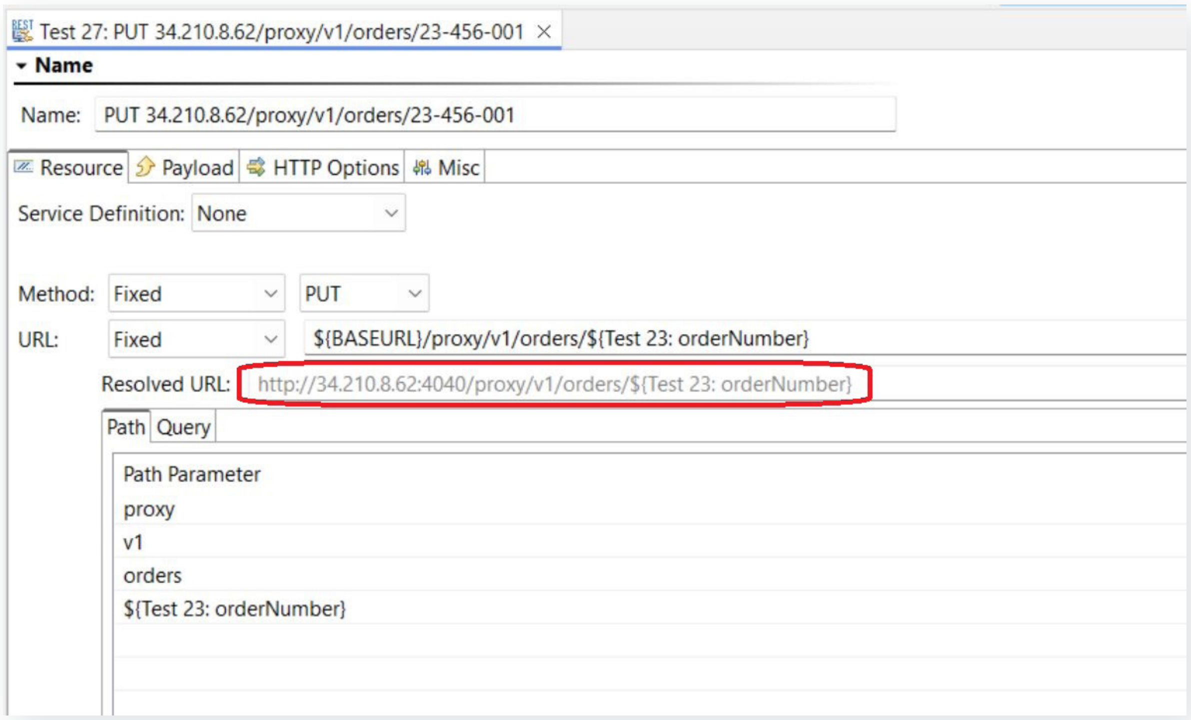
Selected Element	Data Source Column
../orderNumber[1]	Test 23: orderNumber

This is important for the entire end-to-end test scenario to work because there are API calls that require the order number as a request parameter, like test steps 26-28:



Test 26: GET 34.210.8.62/proxy/v1/orders/23-456-001
Test 27: PUT 34.210.8.62/proxy/v1/orders/23-456-001
Test 28: PUT 34.210.8.62/proxy/v1/orders/23-456-001 2

SOAtest's AI made sure not to hardcode the order number from the traffic recording and instead correctly configures the request parameters for these API clients to use the extracted order number from test step 23.



Test 27: PUT 34.210.8.62/proxy/v1/orders/23-456-001

Name

Name: PUT 34.210.8.62/proxy/v1/orders/23-456-001

Service Definition: None

Method: Fixed PUT

URL: Fixed ${BASEURL}/proxy/v1/orders/${Test 23: orderNumber}

Resolved URL: http://34.210.8.62:4040/proxy/v1/orders/${Test 23: orderNumber}

Path Parameter
proxy
v1
orders
${Test 23: orderNumber}

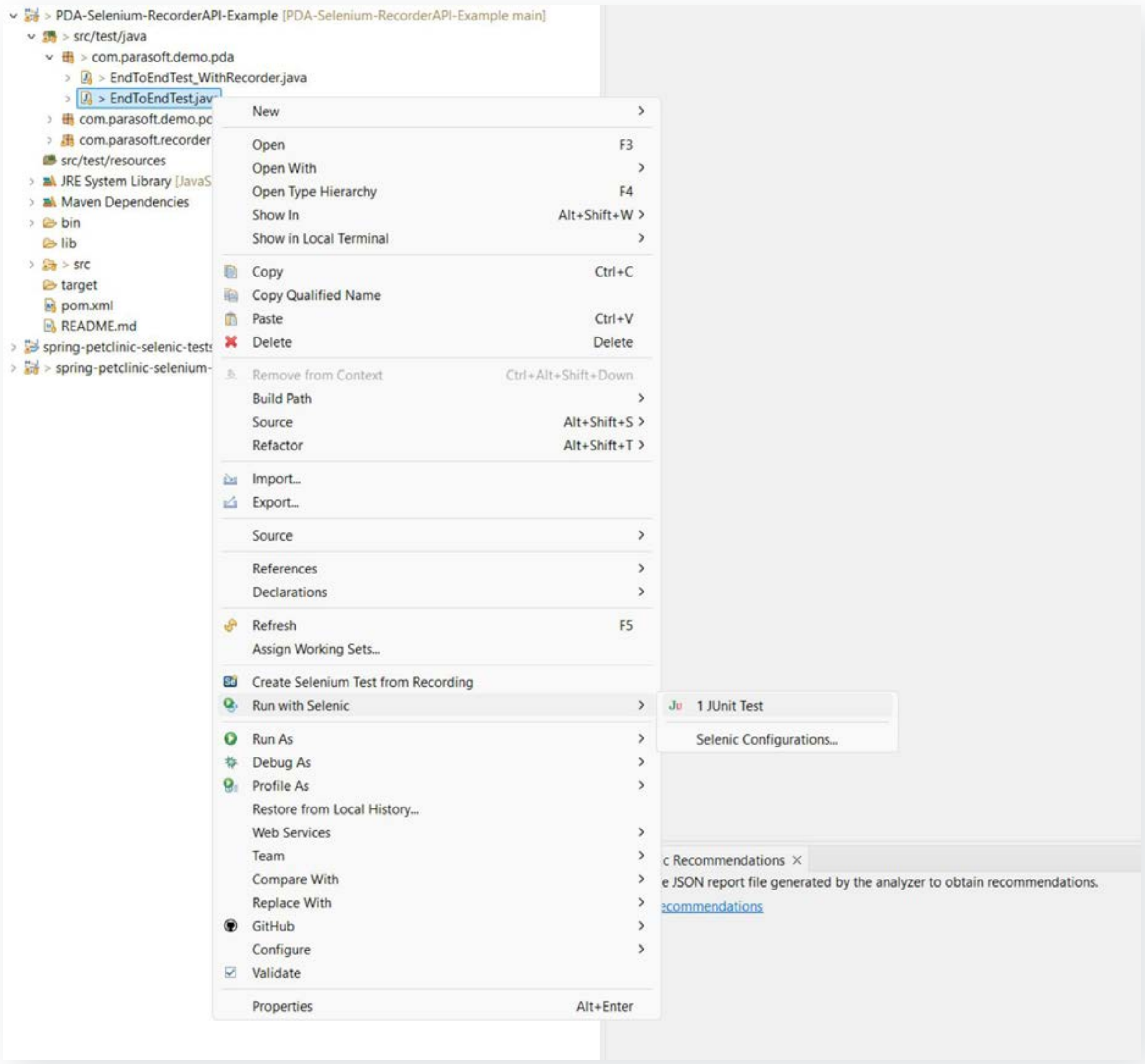
Feel free to review the rest of the example code to see how the full integration with the Recorder REST API works. In the next section of this whitepaper, you will see how Parasoft Selenic eliminates the need for any modifications to your Java-based Selenium projects to accomplish the same workflow in those environments.

Custom-Built Workflows

Selenic is designed to seamlessly plug in to custom-built Java Selenium frameworks that use JUnit, TestNG, and Cucumber with its agent-based approach. In addition to Selenic's AI features to self-heal tests and assist with test creation, users don't have to manually integrate the Parasoft Recorder REST API into their test framework to benefit from SOAtest Smart API Test Generator. Selenic takes care of this for them.

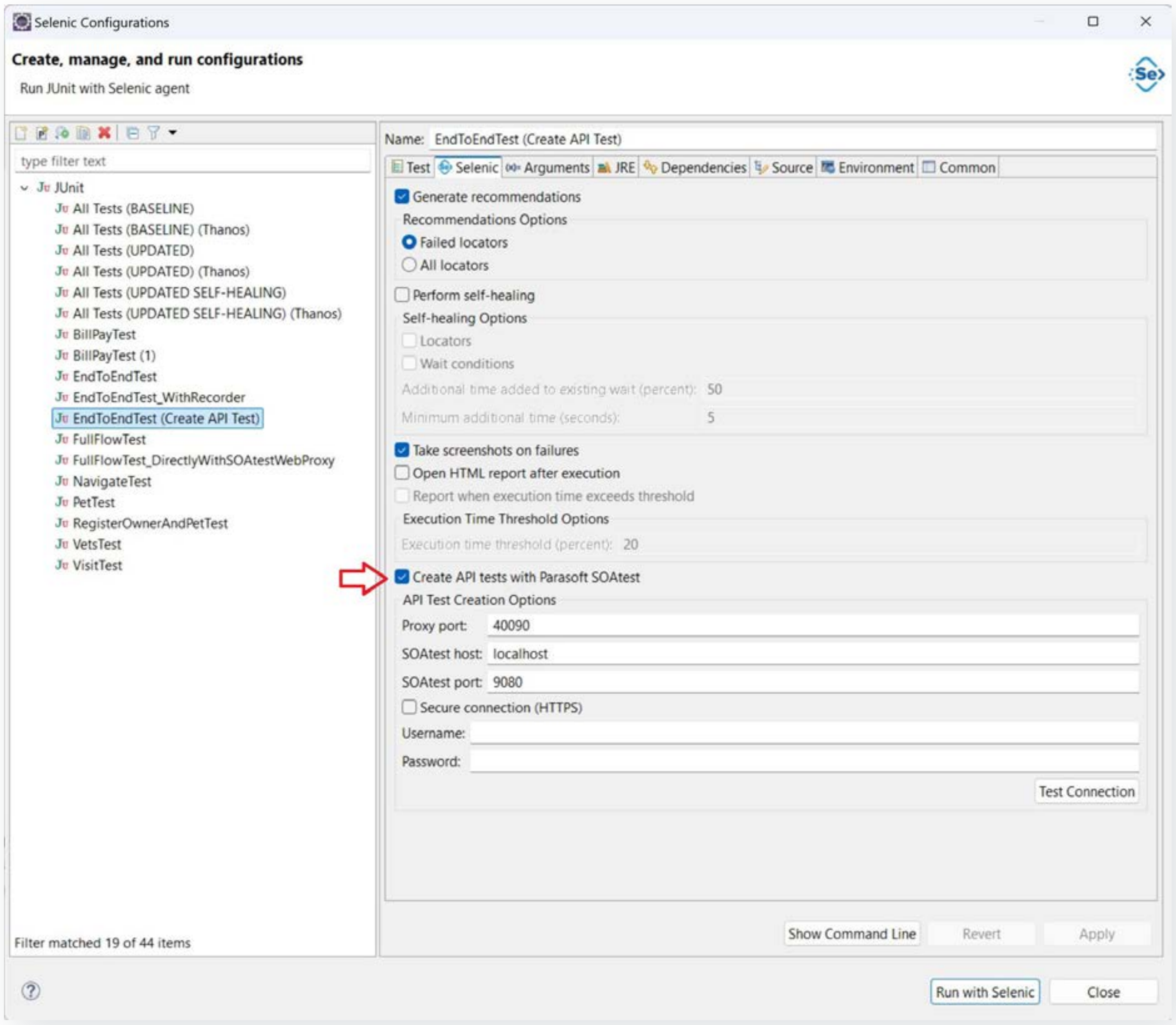
In this section, we'll run the unaltered EndToEndTest.java test class and achieve the same result in SOAtest Desktop without needing to change any code.

Selenic has IDE plugins for both Eclipse and IntelliJ. If you want to replay your suite of Selenium tests from the IDE and get the resulting API scenario tests in your SOAtest workspace, the IDE plugins are a great way to do it. Try it with the Selenic 30-day free trial.



e JSON report file generated by the analyzer to obtain recommendations.

Selenic's IDE integration wraps the JUnit or TestNG test runner with some additional configuration settings. Parasoft can provide the Recorder proxy port and connection information to the SOAtest instance that will receive the recorded HTTP traffic and create API tests from it.


Selenic Configurations

Create, manage, and run configurations

Run JUnit with Selenic agent

type filter text

JUnit
- All Tests (BASELINE)
- All Tests (BASELINE) (Thanos)
- All Tests (UPDATED)
- All Tests (UPDATED) (Thanos)
- All Tests (UPDATED SELF-HEALING)
- All Tests (UPDATED SELF-HEALING) (Thanos)
- BillPayTest
- BillPayTest (1)
- EndToEndTest
- EndToEndTest_WithRecorder
- EndToEndTest (Create API Test)
- FullFlowTest
- FullFlowTest_DirectlyWithSOAtestWebProxy
- NavigateTest
- PetTest
- RegisterOwnerAndPetTest
- VetsTest
- VisitTest

Filter matched 19 of 44 items

Name: EndToEndTest (Create API Test)

Test | Selenic | Arguments | JRE | Dependencies | Source | Environment | Common

- [✓] Generate recommendations
- Recommendations Options
  - [✓ Failed locators]
  - All locators
- [ ] Perform self-healing
- Self-healing Options
  - [ ] Locators
  - [ ] Wait conditions
  - Additional time added to existing wait (percent): 50
  - Minimum additional time (seconds): 5
- [✓] Take screenshots on failures
- [ ] Open HTML report after execution
- [ ] Report when execution time exceeds threshold
- Execution Time Threshold Options
  - Execution time threshold (percent): 20
- [✓] Create API tests with Parasoft SOAtest
- API Test Creation Options
  - Proxy port: 40090
  - SOAtest host: localhost
  - SOAtest port: 9080
  - [ ] Secure connection (HTTPS)
  - Username:
  - Password:

If you want to execute Selenium tests from a command line, Selenic instruments the JVM for easy integration with your Maven or Gradle builds that run your tests.

Benefits of a Lean Web UI and Shift to API Test Strategy

QA teams who have heavily invested in web UI testing don't need to start over to shift their testing focus to the API layers. Instead, leverage AI-enhanced SOAtest to generate API scenario tests that complement your web UI test cases.

By shifting focus to the API layers, teams can begin validating most of the core functionality before web UI test automation can be stabilized. This will result in decreased feedback time to development while increasing feedback frequency.

Fully testing for functional misbehavior via API testing enables a lean web UI test strategy where web UI test problems aren't exacerbated with application defects that could have been caught with API testing. Teams can also target UI tests for specific purposes or reasons, such as cross-browser validation, user acceptance testing, web accessibility testing, or testing frontend logic that isn't built through APIs.

Conclusion

UI testing has clear benefits. However, overinvesting in UI testing can lead to high maintenance burdens on testing teams if the applications under test are dynamic and consistently experiencing change. By diversifying the types of tests that make up the functional test strategy and shifting testing to the API layers, teams can benefit from a more scalable, maintainable, and effective testing practice. Easily make this shift to the API layers and start reaping the benefits by applying Parasoft SOAtest's AI-enhanced Smart API Test Generator to your test strategy.

TAKE THE NEXT STEP

Request a demo to see how your team can increase the scalability and velocity of your functional test strategy with AI-powered Parasoft SOAtest.

About Parasoft

Parasoft helps organizations continuously deliver high-quality software with its AI-powered software testing platform and automated test solutions. Supporting the embedded, enterprise, and IoT markets, Parasoft's proven technologies reduce the time, effort, and cost of delivering secure, reliable, and compliant software by integrating everything from deep code analysis and unit testing to web UI and API testing, plus service virtualization and complete code coverage, into the delivery pipeline. Bringing all this together, Parasoft's award-winning reporting and analytics dashboard provides a centralized view of quality, enabling organizations to deliver with confidence and succeed in today's most strategic ecosystems and development initiatives—security, safety-critical, Agile, DevOps, and continuous testing.