Definitive Guide

to Data-Driven Testing for SOAP & REST APIs

SoapUI NG Pro

Table of Contents

Introduction

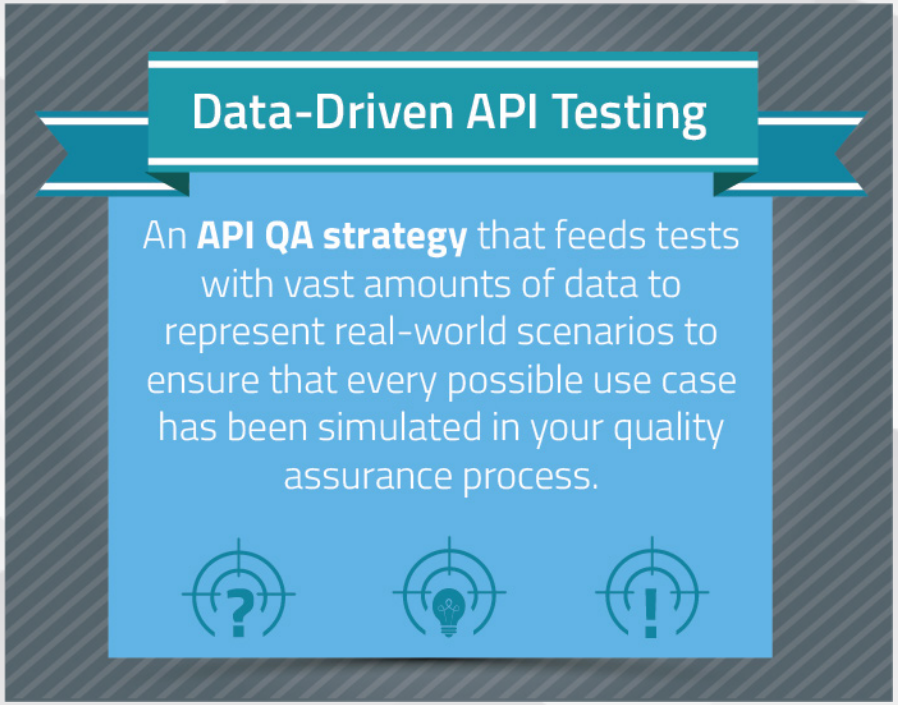
Over the past several years, the role of the API in the modern enterprise has grown exponentially. Long gone are the days when an API was treated solely as an afterthought – a mere set of tacked-on calls to an existing application, with little additional value to anyone other than a narrow developer community.

Today, businesses - regardless of whether or not they operate in the technology industry - place increased emphasis on delivering APIs. In many cases, the API is now the primary mechanism by which partners and consumers interact with an enterprise's business systems.

The API has now evolved into absolute fact of life in nearly every organization, from mid-size to global giant. Whether they're employed internally, externally, or both, APIs are assets that connect systems, streamline workflows, and make every type of integration possible. In fact, beyond improving operational efficiency and enabling cross-system communication, APIs now serve as competitive differentiators for many organizations. It's no exaggeration to state that technology-driven businesses such as Uber, AirBnB or eBay live and die on the strength of their APIs, and this degree of reliance is spreading across every industry.

With all this in mind, it's more important than ever to treat the API with the same amount of attention and care as you would for any other mission-critical enterprise asset. This means that API quality and performance are absolutely non-negotiable.

But how do you ensure that your API is ready to handle the seemingly infinite possible conditions that it will encounter in the real world? One powerful, yet surprisingly simple technique is to feed your tests with vast amounts of representative data to help ensure that every conceivable use case has been simulated in your quality assurance process.

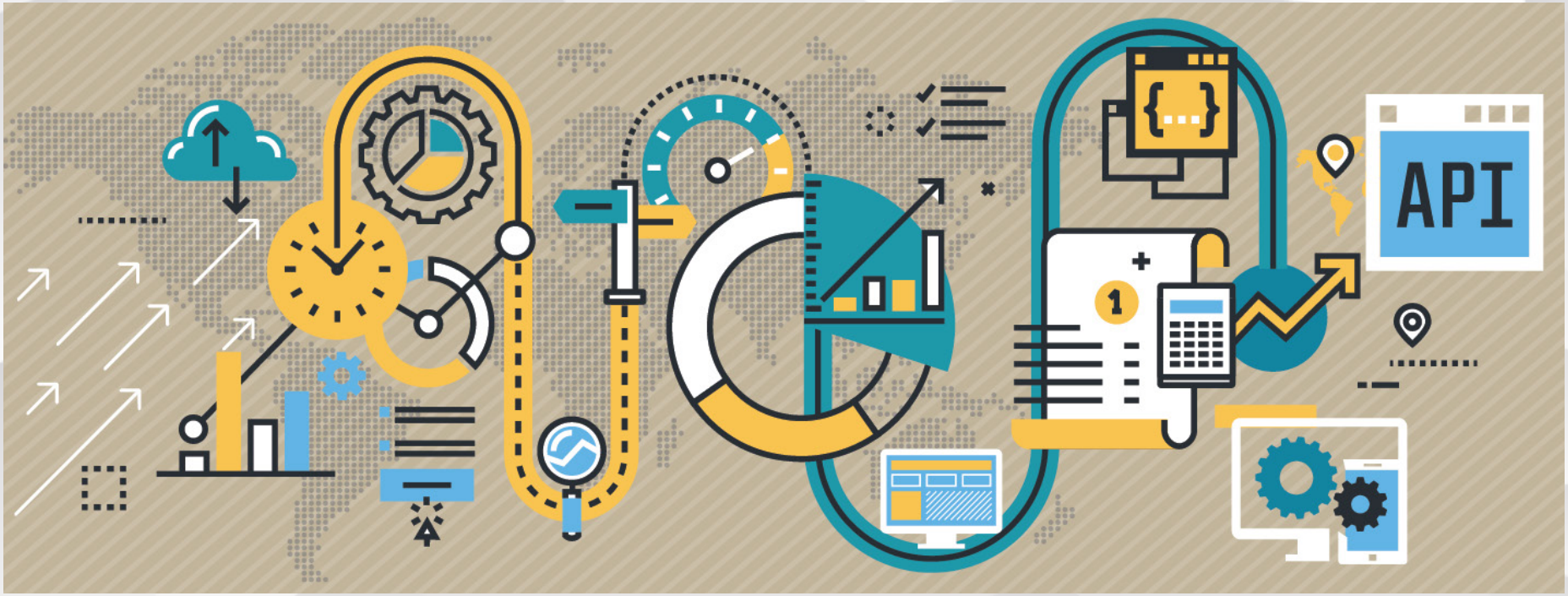


This solution is known as data-driven API testing, and it's a well-proven quality assurance strategy that separates the data that powers the functional test from the underlying test logic.

This autonomy permits a tester to quickly design a series of scenarios that execute the same steps - which are performed repeatedly in identical order - but with a variety of test data supplied at run time from information sources such as spreadsheets, flat files, or relational databases.

This approach is in direct contrast to the common practice where testers hard code – or manually enter – input parameters and then eyeball the responses to ensure they're correct. It's also possible for testers to store predicted responses alongside the messages that will be communicated to their API. This makes it feasible to configure automated assertions to compare the expected results with the actual responses.

Adopting a data-driven API testing methodology is an essential prerequisite for ascertaining that your APIs are of production quality. Beyond validating functionality, these tactics also serve as a foundation for confirming that your APIs will deliver on their performance and security obligations.

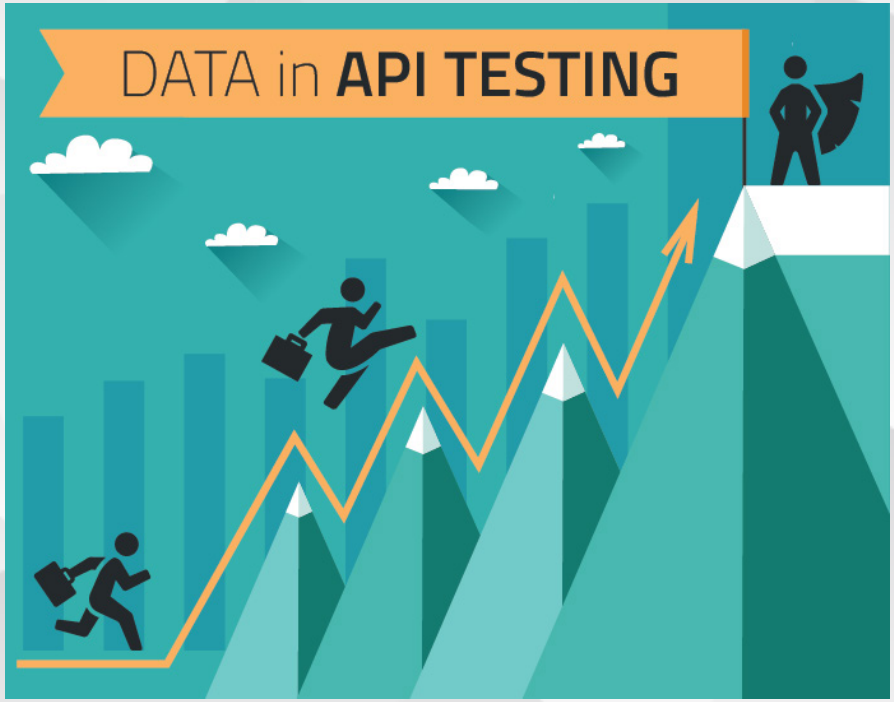


Obstacles to Employing Data in API Testing

At first glance, it's tempting to simply resurrect the same methodologies and best practices that have been so successful in evaluating graphical user interface (GUI) applications and apply them to SOAP and REST API testing. This is a mistake: APIs are very different types of assets, with their own distinct set of design patterns, access methods, usage characteristics, and risk exposures.

For example, when testing an application through its standard user interface, the array of potential inputs are greatly constrained by what's physically possible in a GUI - after all, it's not easy to hand-type a 2 GB video file into a text box in a browser or Windows application. On the other hand, it takes no extra effort to attempt to transmit that video file to a SOAP service or REST API. The people or software that interact with the API can submit any kind of request; it's the specification and application logic of the API itself that will determine whether or not the request content is permissible, and what type of response will be returned.

The allure of APIs is that they're open and encourage interoperability, but these attractions also multiply testing complexity. Even the simplest API must be tested using colossal numbers of permutations, far beyond what a person or even large team can manage by hand. Simply stated, fruitful API testing mandates that the enterprise apply a specialized collection of quality assurance practices that recognize the inherent complexity found in distributed computing. Data-driven testing is a major part of this methodology, but many enterprises encounter difficulty trying to properly incorporate it into their day-to-day operations. Let's look at a few factors that contribute to this shortfall.



Time pressures

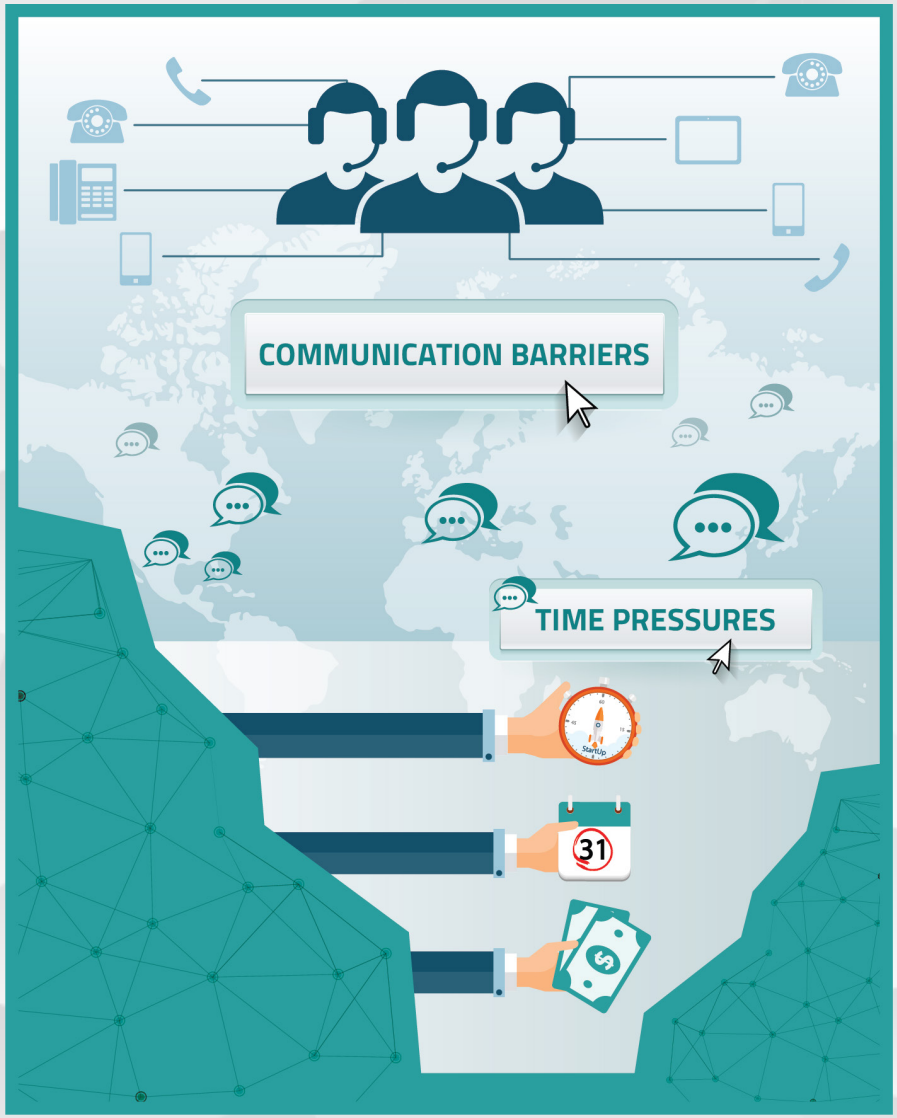
Diverse influences such as agile delivery practices, competitive business pressures, outdated testing methodologies, and unyielding schedules team up to place enormous pressure on software developers and quality assurance professionals. The relentless drive to continually release new API functionality means that there's rarely sufficient time to properly architect reusable tests capable of effectively incorporating data.

Instead, testers tend to reuse the same handful of static, sample records. This tactic is easily replicable (and after all, consistency is important), but falls far short of accurately representing real-world usage or the full set of API capabilities. For example, cutting corners by using test records from a collection of five customers misses out on the nearly infinite combinations of data that will ensue in production.

Communication barriers

Many organizations segregate the business analysts and users that specify the functionality to be supplied by an API from the QA team that is tasked with testing it. By the time the testing phase begins, the specification has gotten lost in translation and the testers – who may even be employees of a third party outsourcing firm working on the other side of the globe – aren't quite clear on the exact purpose of the API itself, much less the nuances of its inbound and outbound parameters.

This unawareness makes it impossible to design effective, far-reaching data-driven tests. Without precise knowledge of the API's business goals, it's natural for the QA team to design test cases based on a relatively limited set of hard-coded data.



Inadequate tooling

Even with the best of intentions, it's impossible to quickly and easily create data-driven API tests using much of the software tooling on the market.

With the exception of techno-logies such as SmartBear's Ready! API, most API evaluation tools have been produced for software developers. These utilities frequently require extensive scripting to conduct even the most rudimentary API interactions, much less a full set of automated data-driven probes.

Faced with tight schedules, functionally poor testing tools, and the job of writing copious scripts, it's no wonder that many testers take the easier path of hand-typing sample API input data and glancing at the results to detect anomalies.

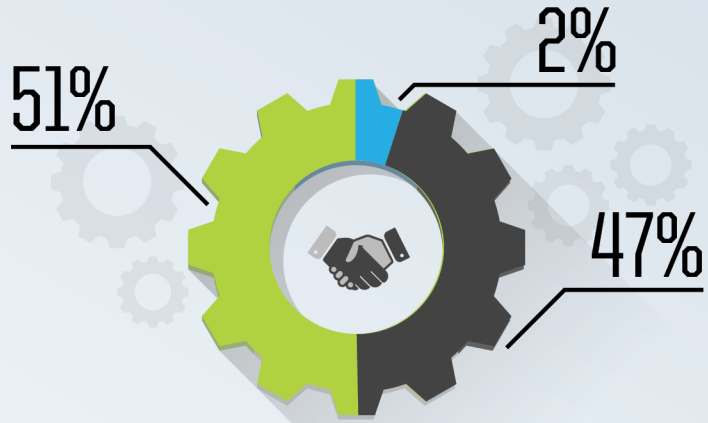
The Gap between API Testing Goals and Reality

The vast majority of enterprises recognize the wisdom of powering their API tests with data. In a live webinar on the topic of data-driven API testing, attendees were polled on their perception of the importance of data-driven testing, along with how much of this testing strategy they actually employed in their own organization.

The results were illuminating:

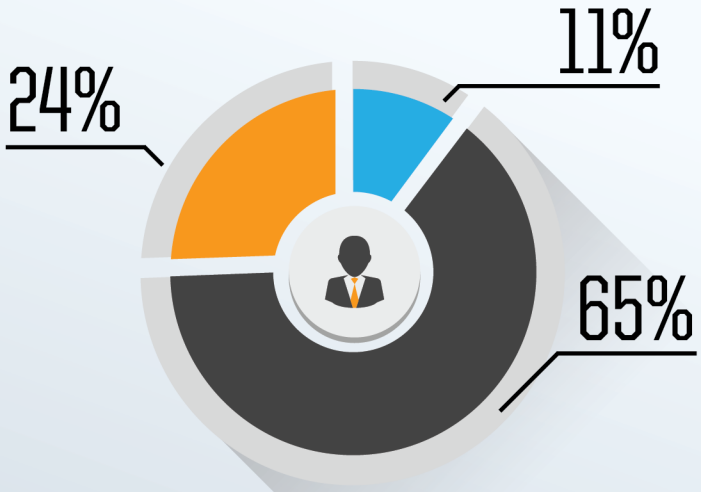
Although over half of respondents indicated that data-driven API testing is a "very important" priority, merely 24% of the audience was employing this strategy extensively. This deficiency exposes APIs – and the business that relies on them – to plenty of risks.

How important is Data-Driven Testing to your organization?



- Not important at all
- Very important
- Growing importance

How much Data-Driven API Testing do you do today?



- None
- Some
- Extensive

51% of attendees for this webinar responded that **Data-Driven API Testing** is "very important", and yet only **24%** are doing extensive **Data-Driven Testing**

Business logic isn't fully exercised

Although every API is unique, it's common for inbound messages to incorporate dozens of parameters, each with its own range of potential values. There are also uncountable would-be interactions among these parameters. A handful of hard-coded test cases can't possibly explore all of these permutations, meaning that only a fraction of the underlying application code gets evaluated. The upshot is that the API appears to have higher quality than it really does.

It's even worse when you recall that business users are frequently separated from the QA team. For example, consider what happens when a tester encounters a date field to be sent to an API. With no knowledge of the distinct input requirements for that field – and a lack of supporting software tooling to make it easy to supply a variety of values – the tester will likely just plug in a random entry. The end result is that the business logic that was so carefully encoded in the API will not be properly tested, which guarantees hard-to-diagnose problems in production.

Latency isn't accurately measured

An API is not an island: a complex, multilayered stack of technology – including database servers, web servers, application servers, and other infrastructure – join forces to bring it to life. Consequently, it's imperative to consider the impact of your testing strategy on the entire API stack. For example, if testers repeatedly feed the same minimal, static set of input values to the API, its responses will become cached and performance will appear to be blindingly fast – at least during testing.

This provides yet another false sense of security: in production, the API will suddenly start performing unpredictably, and may even seem to be plagued by hard-to-reproduce bugs.

Automation isn't possible

Today, many software development organizations have adopted agile delivery methodologies. Release cycles have shortened from weeks to hours, and the only credible way to keep up with this blistering pace is to apply automation – for builds as well as testing. Unfortunately, a lack of data-driven testing means manually invoking APIs using a limited set of values coupled with time-consuming inspection of individual request/response messages.

These monotonous, error-prone maneuvers are diametrically opposed to what's possible using automation.

Luckily, the quality, robustness, and affordability of modern API testing platforms such as Ready! API have made data-driven testing possible for every organization: there's no longer a need – or justification – for a tester to painstakingly type in API-bound data and then eyeball the results to confirm their accuracy.

Best Practices for Data-Driven API Testing

Once you've decided to take the plunge and begin applying data-driven testing procedures to your API quality assurance efforts, you'll quickly start reaping the rewards of this highly flexible strategy. These advantages include:

- Comprehensive application logic validation.
- Accurate performance metrics.
- Efficient technology stack utilization.
- Synchronization with Agile software delivery techniques.
- Automation-ready tests.

To get the most value from your data-driven tests, try following as many of these best practices as you can.

Use lots of realistic data

This point may seem intuitive, but the closer that your test data reflects the conditions that the API will encounter in production, the more comprehensive and accurate your testing process will be. The best way to ensure that your test data is realistic is to start at the source – the business procedures that your API was designed to support. It's incumbent upon you to be mindful of the gaps between business users and API testers that were depicted earlier in this guide: make it a priority to understand the rationale behind the API, along with the information being sent to it, both in design and in practice.

It's also important to consider that there may be numerous yet non-obvious interrelationships among data: certain input values may be contingent on other information that is be transmitted to the API, and the same conditions may apply for returned data. With the proper set of tools, you should be able to accurately represent these relationships in your test data.

Test both positive and negative outcomes

Most people only think of monitoring positive responses from their APIs: transmitting valid data should result in a server-side operation being successfully completed and a reply to that effect returned to the API's invoker. But this is just the start - it's equally important to confirm that sending incorrect

or otherwise invalid parameters to the API triggers a negative outcome, which is commonly an error message or other indication of a problem. Furthermore, functional tests can be configured to gracefully cope with error conditions that would normally halt the test.

This method of API evaluation frees the tester from having to wade through the full set of results to hone in on a point of failure. Instead, only those cases where an anticipated positive outcome came back negative - or vice versa - need to be investigated.

Use data to drive dynamic assertions

Assertions are the rules that express the projected response from any given API request. They're used to determine whether the API is behaving according to its specification, and are thus the primary metrics for quality. Many testers make the mistake of hard-coding these guidelines, which introduces unnecessary maintenance overhead and brittleness to the API evaluation process.

On the other hand, dynamic assertions are flexible, and can very from one API request to another. For example, an e-commerce shipping calculator API probe may transmit one row of test data for a sale of $50, which should result in free shipping (i.e. a reply stating that the shipping charge will be $0). The next row of test data may be for an order valued at $49.99, which should result in a shipping charge of $5.99. A dynamic assertion will let the tester store the expected response of $0 alongside the input order of $50, and $5.99 for the $49.99. New test scenarios can then easily be added to the set of input data without requiring any changes to the functional test itself. And if the shipping policy changes, only the test's data and assertions need to change – everything else will remain the same.

Track API responses

Many testers fixate on the success or failure of each API invocation, and discard the set of responses after they've finished running their functional tests. That's a shame, because replies from an API are very useful artifacts. Without recording these test results, important history is lost.

If an API undergoes multiple changes and a new error is uncovered during the regression testing process, it can be a monumental task to determine precisely which modification caused the flaw. Consulting a library of stored API requests and responses makes identifying the moment that the new problem occurred – and correcting it – much less of a hassle.

Repurpose data-driven functional tests for performance and security

Throughout this book we've been talking about functional tests but it's worthwhile to note that many organizations use highly unrealistic, narrowly focused performance and security tests that are also hamstrung by narrow sets of hard-coded test data.

Since it takes significant time and effort to set up a properly configured, adaptable data driven functional test, once you've made that investment, why not utilize it for more than just one objective? Reusing a data-driven functional test introduces a healthy dose of reality to the performance and security evaluation processes, and technology such as Ready! API makes it easy to make this transition.

Three Levels of Data-Driven Testing

This section introduces three examples of data driven testing to help bring the recommendations we've made in this book to life. We've performed the scenarios using SoapUI NG Pro; you can download an evaluation copy to try on your own APIs from <link>. These samples are for a car rental company that offers a SOAP Web service and REST API to help with the ve-hicle acquisition process. The SOAP Web service is designed to provide vehicle availabi-lity details for specific locations.

See table 1 for the range of sample input values that this Web service uses. The REST API is intended to supply details about individual airport locations – such as their GPS coordinates and hours of operation.

Table 2 lists these airport office lookup values.

LOCATION	VEHICLE TYPE	DURATION
Beijing (PEK)	Subcompact	1 day
Chicago (ORD)	Compact	...
Dubai (DXB)	Intermediate	30 days
London (LHR)	Full-size	
Los Angeles (LAX)	Van	
Mumbai (BOM)	Luxury	
New York (JFK)		
San Francisco (SFO)		

Table 1: Parameters for the SOAP Web service

LOCATION
Beijing (PEK)
Chicago (ORD)
Dubai (DXB)
London (LHR)
Los Angeles (LAX)
Mumbai (BOM)
New York (JFK)
San Francisco (SFO)

Before we present these scenarios, it's important to acknowledge that thanks to scheduling, budget, tooling, or other pressures, a significant percentage of enterprises deploying APIs typically conduct the most rudimentary interactions with their APIs.

As shown in figures 1 and 2, these usually involve a tester manually issuing a handful of requests and then double-checking the responses.

Table 2: Parameters for the REST API

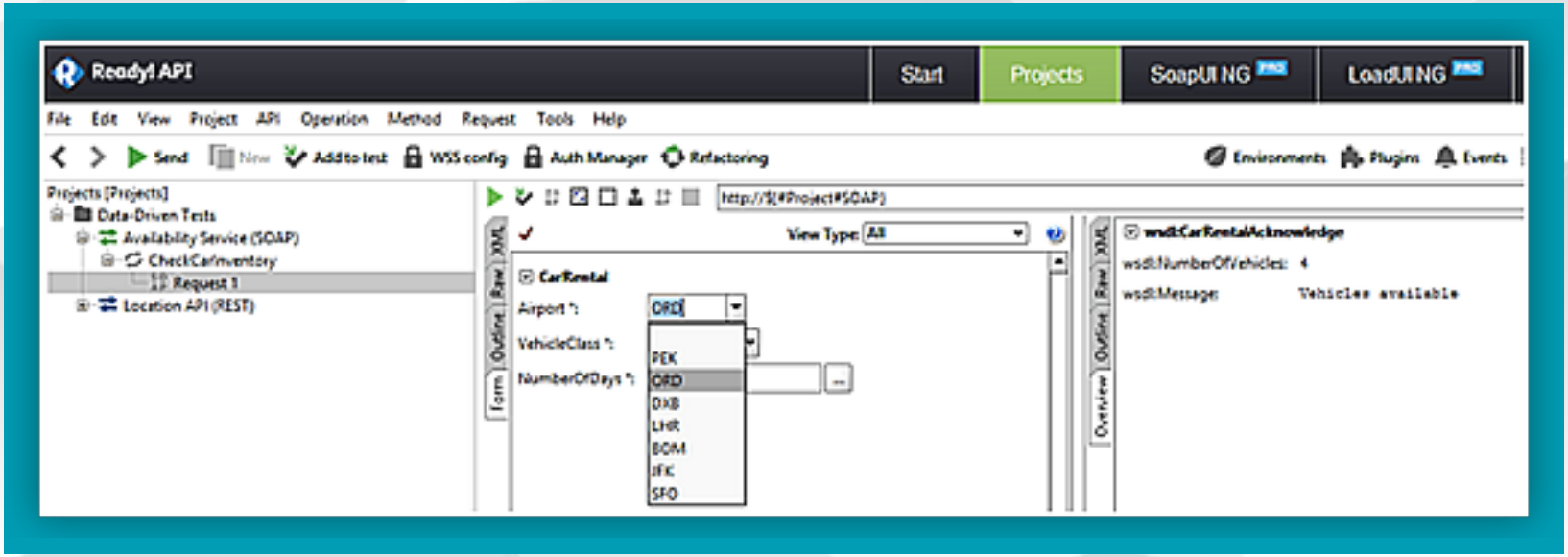


Figure 1: A manual Web service interaction

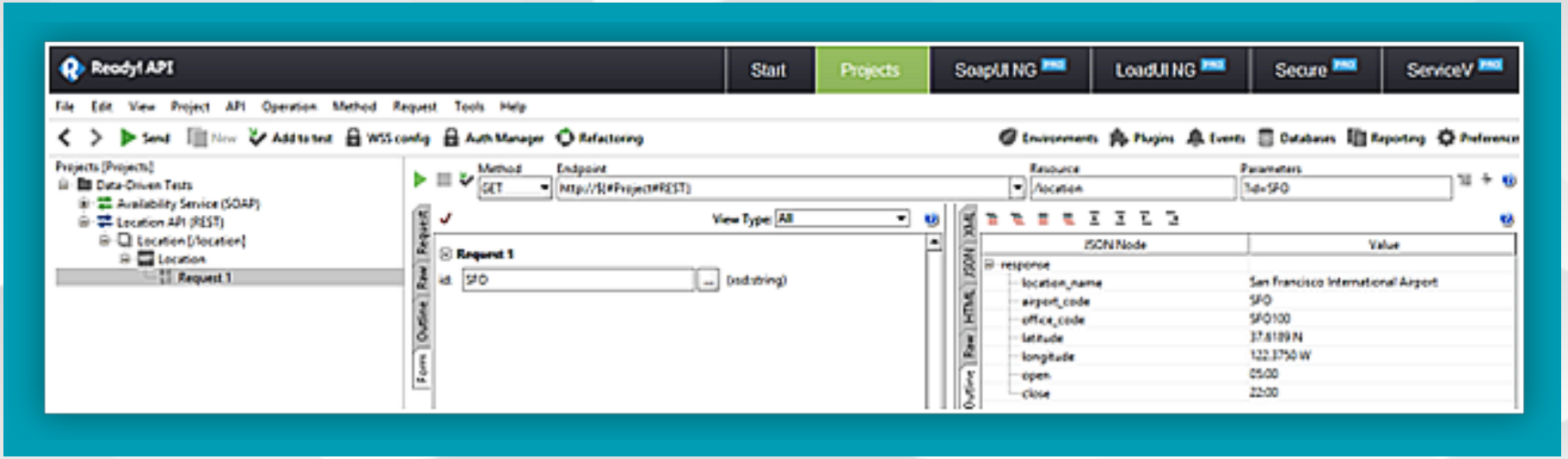


Figure 2: A manual REST API interaction

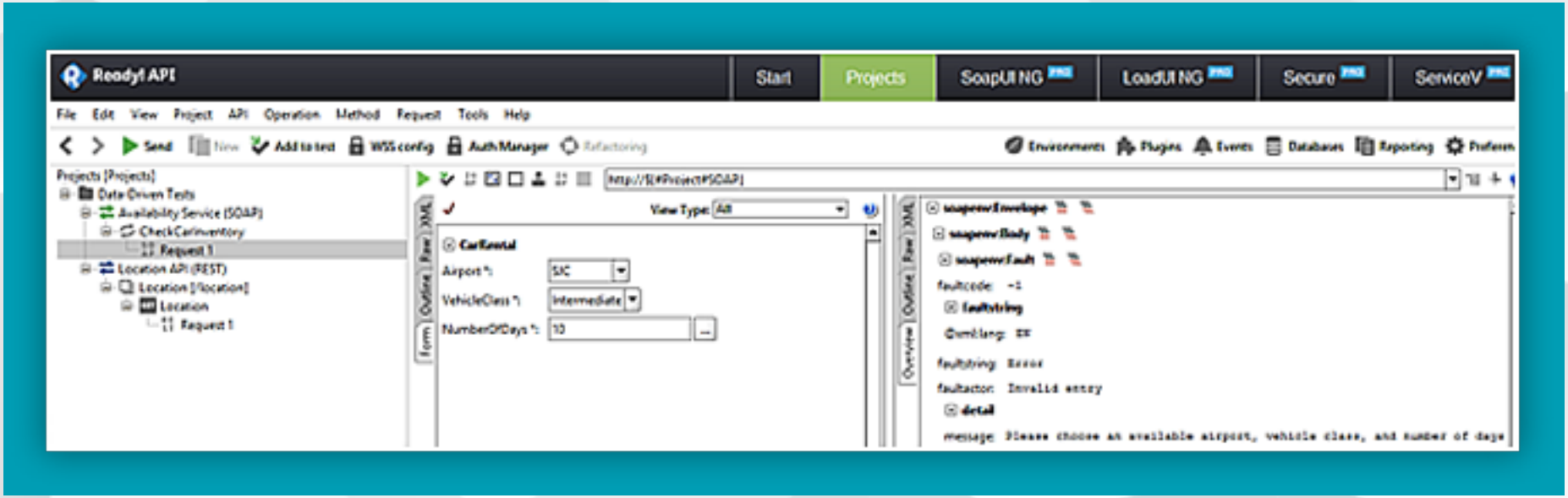


Figure 3: A manually triggered SOAP fault

This haphazard, time-consuming strategy means that agile delivery and test automation are out of the question.

With the widespread example of manual testing out of the way, let's examine three increasingly sophisticated and effective approaches to data-driven testing.

Scenario 1: Basic data transmission

In this example, the tester is utilizing the SoapUI NG Pro Data Generator to feed randomly chosen values from pre-defined lists to a SOAP Web service. Figure 4 shows how these lists can be defined and maintained, while figure 5 displays linking the test data with the input parameter.

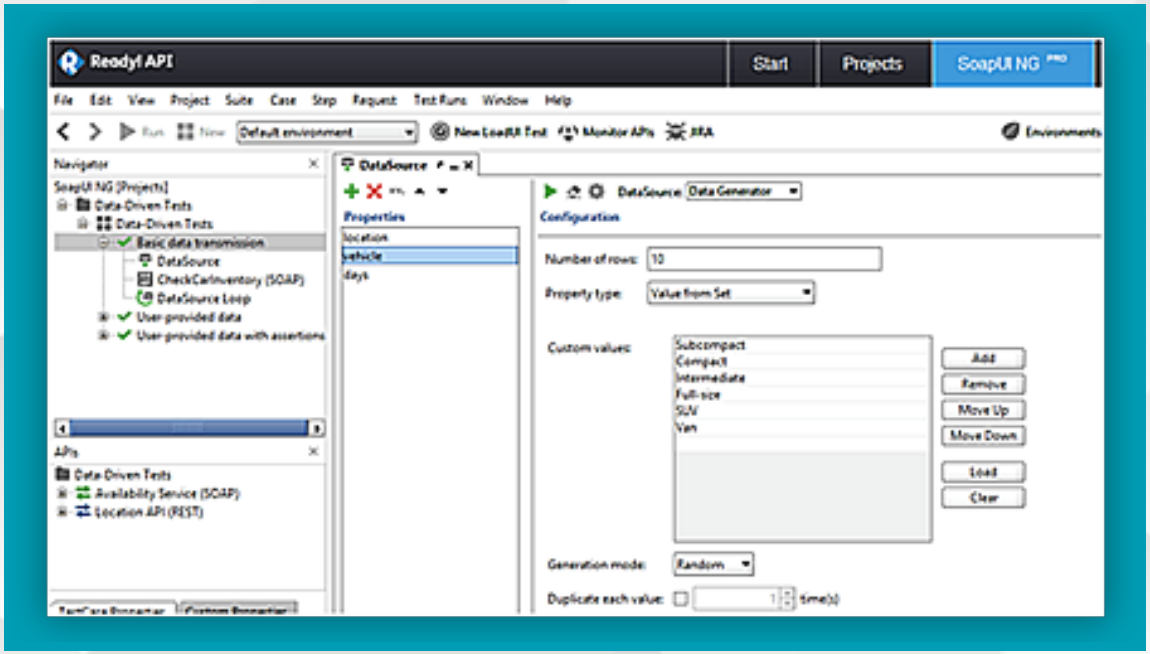


Figure 4: Configuring the SoapUI NG Pro data generator

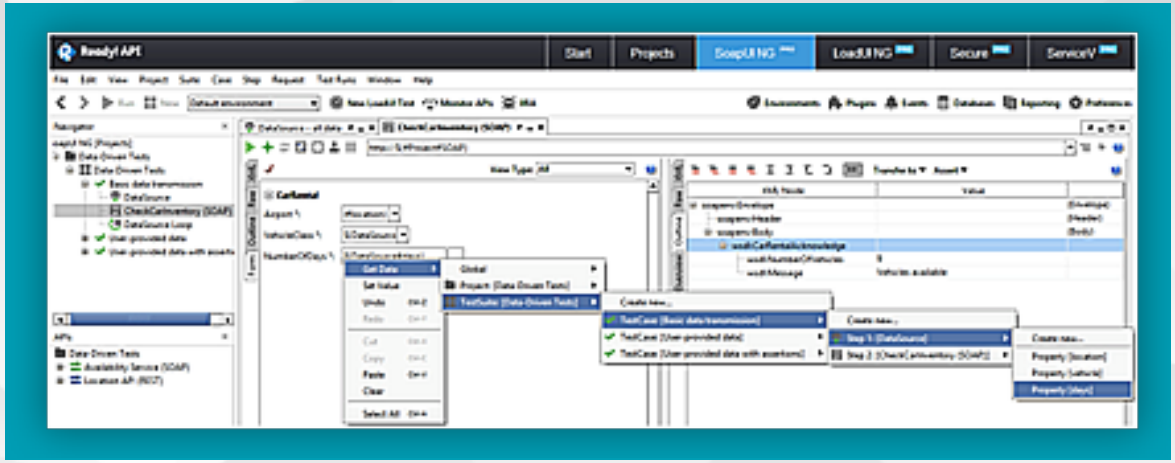


Figure 5: Associating data with an element in a request

Once all of the communications with the Web service have concluded, the tester can scrutinize each interaction in the transaction log.

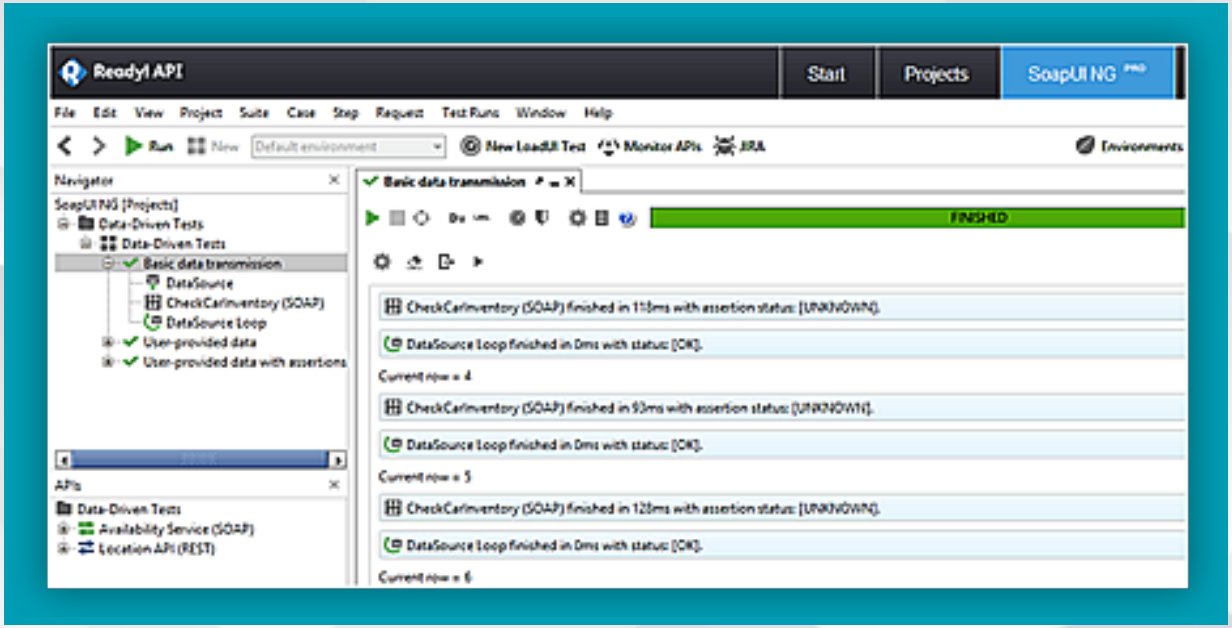


Figure 6: Transaction log for a completed functional test using data generation

While this is a great start, it still requires the tester to peruse the entire transaction log to evaluate each message request/response pair. Figure 7 presents one of these instances; imagine the amount of effort to manually evaluate hundreds of much more complex, real-world interactions.

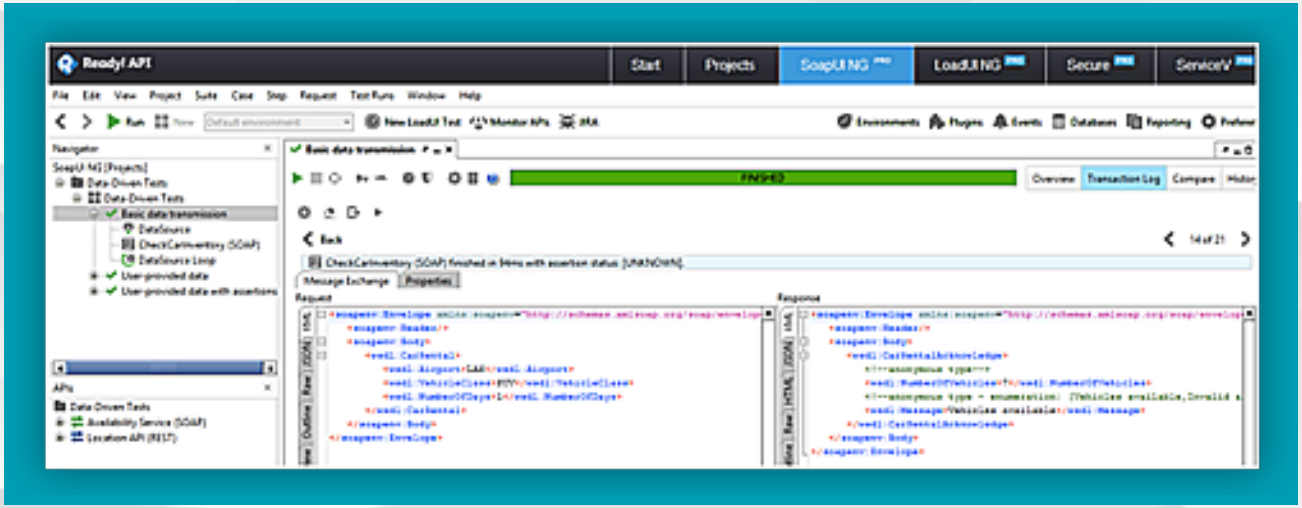


Figure 7: Message detail for a data-generated request

Scenario 2: User-provided data with manual result evaluation

In the previous example, the tester relied on the data generator to randomly select values from a pre-defined group of lists to transmit to the API. However, this approach had no awareness of any inter-element relationships, which was of limited worth when attempting to evaluate the underlying API business logic.

In this scenario, the tester goes one step further and supplies business-aware parameters - including encoding cross-element dependencies - to a test that probes a REST API. Note that these values may be maintained inline, or in a separate sources such as a flat file, spreadsheet, or database, and often include values that should precipitate an error.

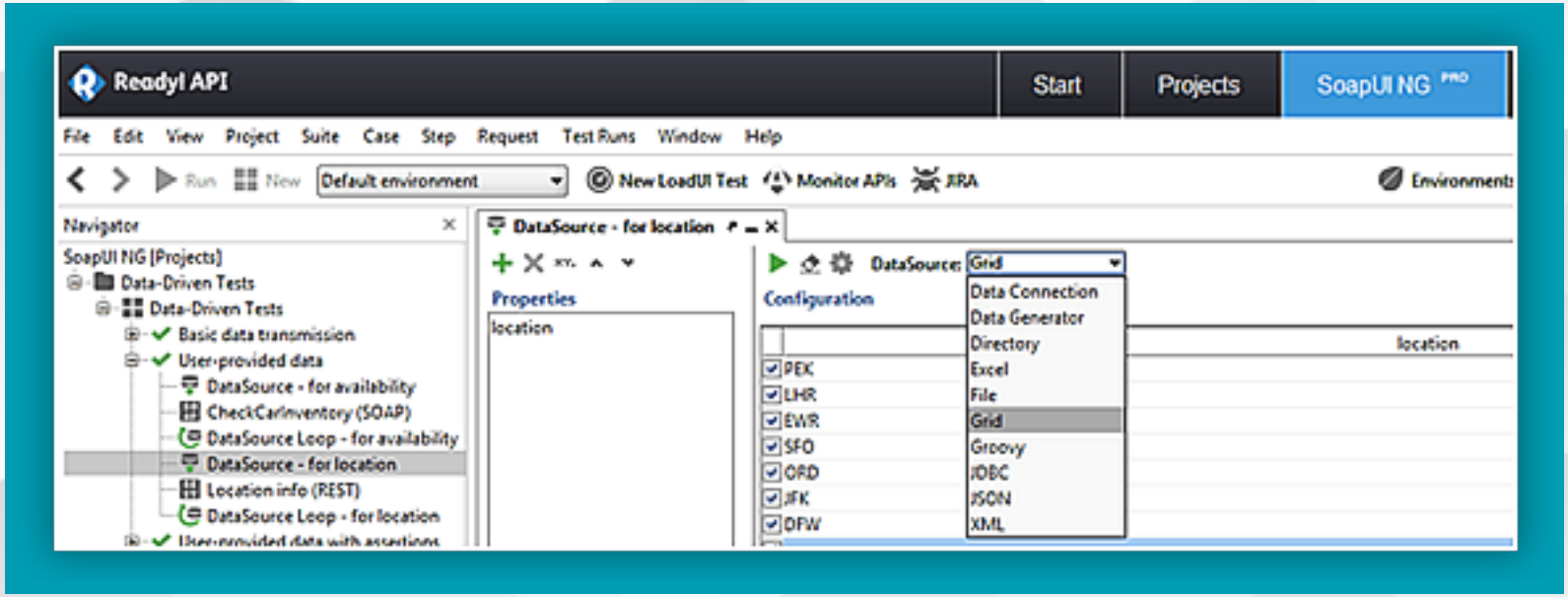


Figure 8: Creating in-lined data for transmission to API

Once again, the burden is on the tester to manually review the transaction log (figure 9) and cross-reference the input parameters with the response messages, especially for those records that should have triggered an error - as shown in figure 10.

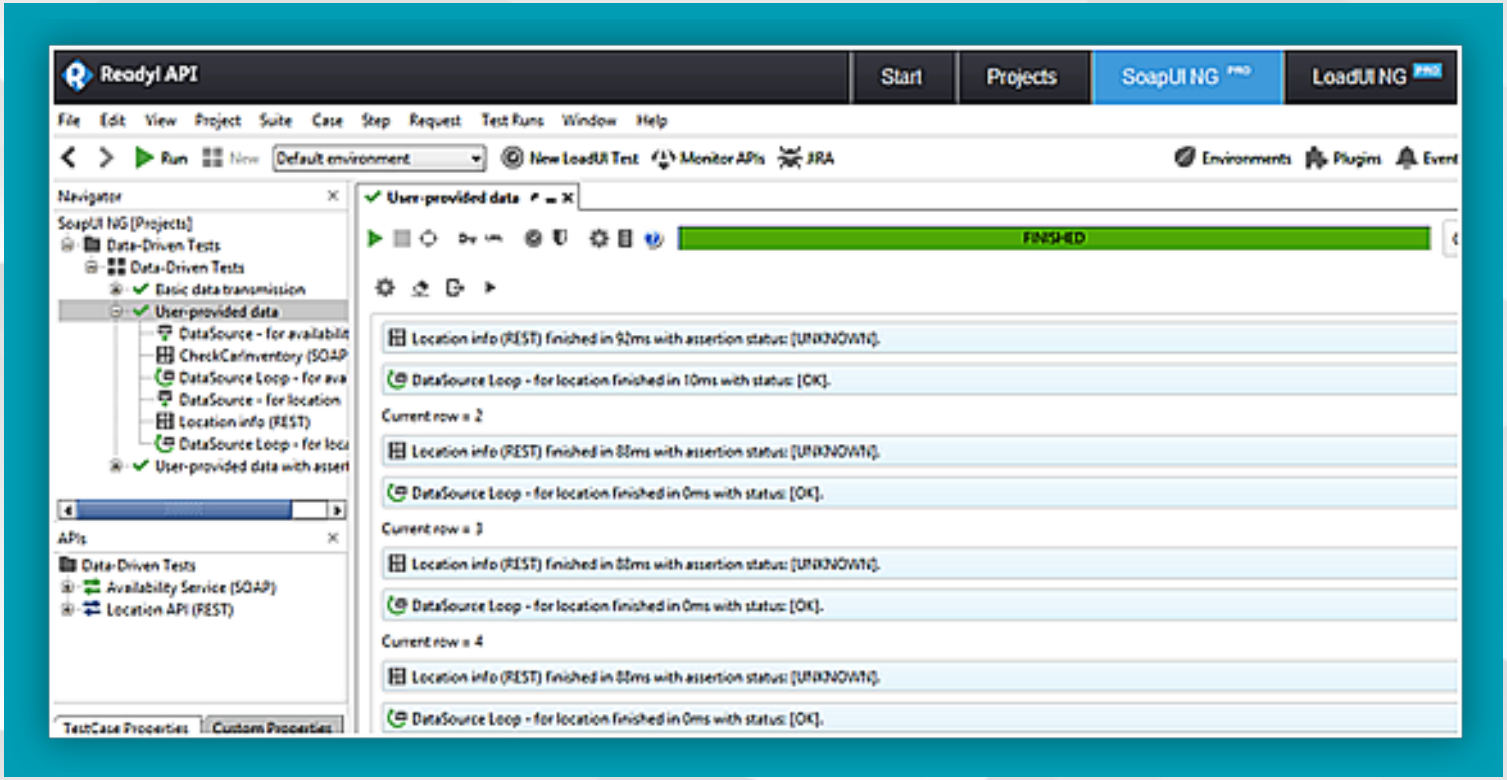


Figure 9: Transaction log for manual inspection

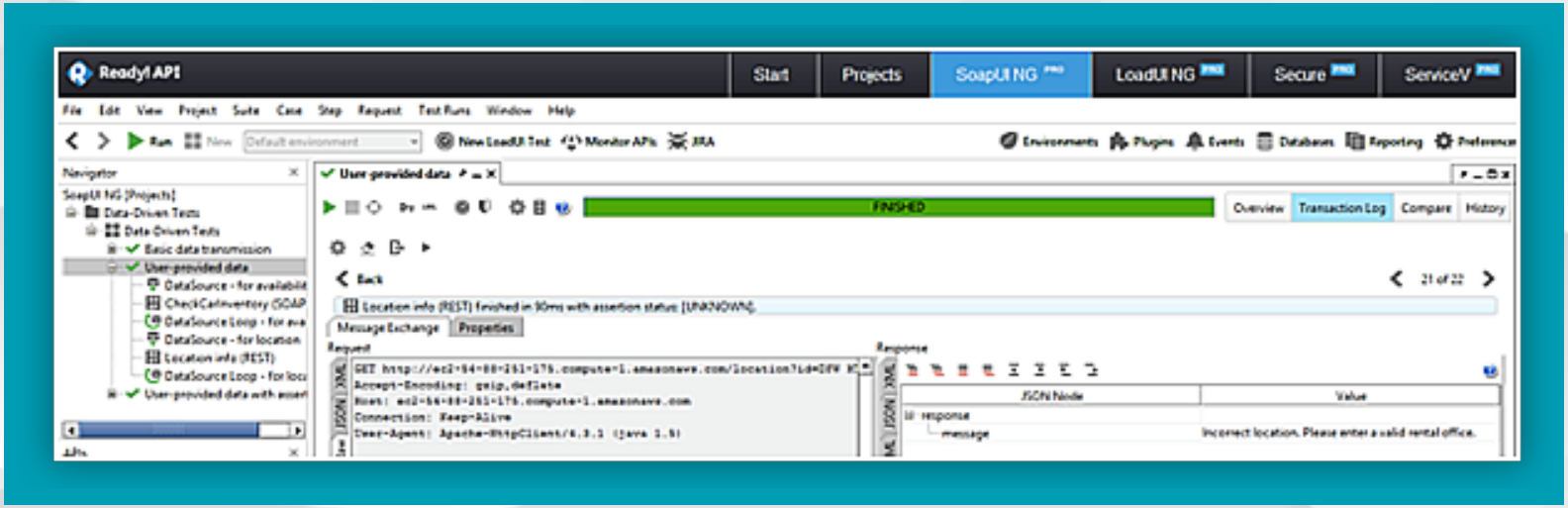


Figure 10: Confirming an error response for a faulty input parameter

Scenario 3: User-provided data with automated assertions

As we described earlier in the section dedicated to dynamic assertions, unlocking the full potential of data-driven testing usually means storing expected responses - either in full or partial - alongside the inbound test data.

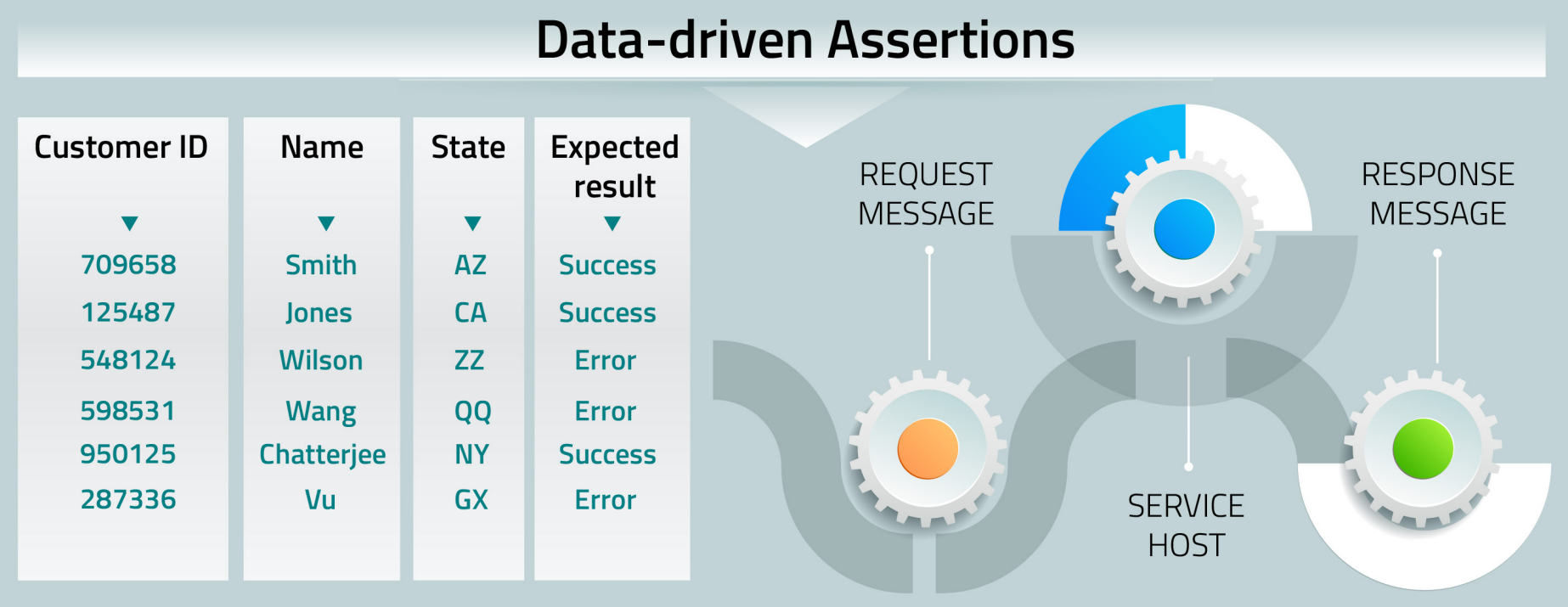


Figure 11: Storing inbound test data and expected results together

In this example - shown in figure 12 - the tester has created a comprehensive set of data to be transmitted, as well as what should be found in the responses from the SOAP (expected_ inventory) and REST (expected_location) API calls.

This data source can be utilized for multiple API calls, helping to reduce the overall number of moving parts in the test.

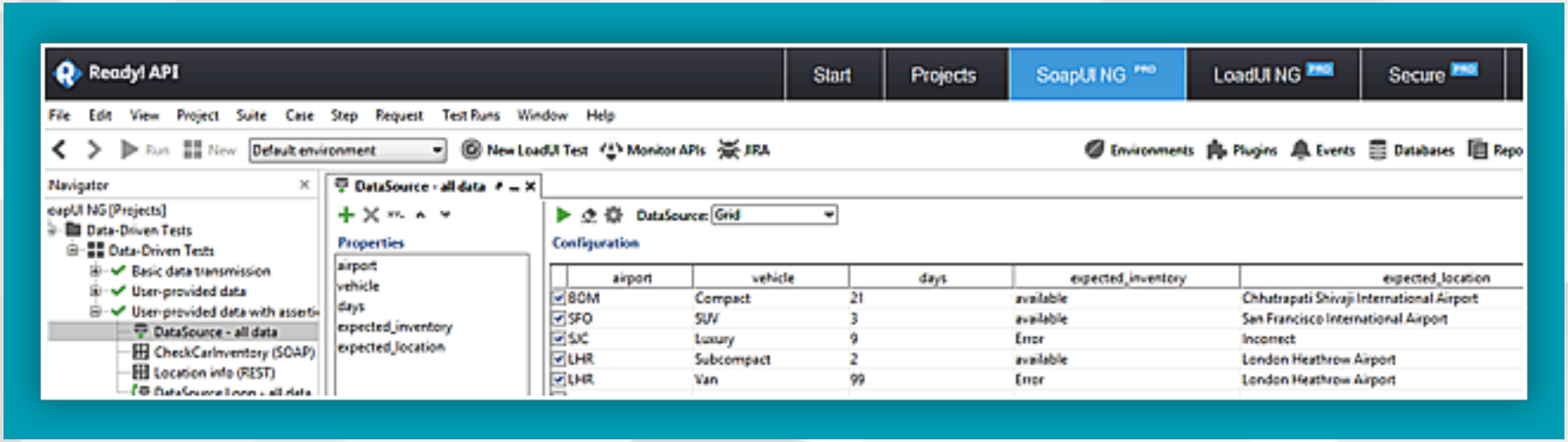


Figure 12: Configuring a data source with request and expected response values

Rather than needing to manually review responses, the tester sets up dynamic assertions that are automatically updated and correlated with the data source for each request/response pair. Figure 13 shows the syntax for one of these assertions.

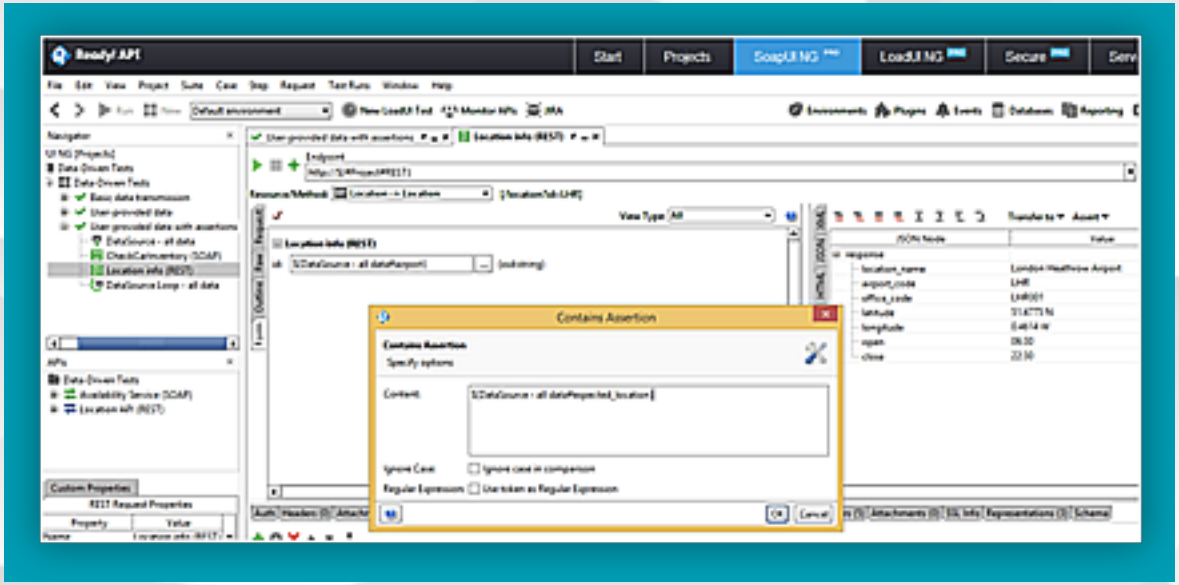


Figure 13: Creating a dynamic assertion

Once again, the transaction log (figure 14) provides a history of the test, but this time the tester can rely on a report, data extract, or other automated mechanism to uncover only those messages that have unexpectedly failed.

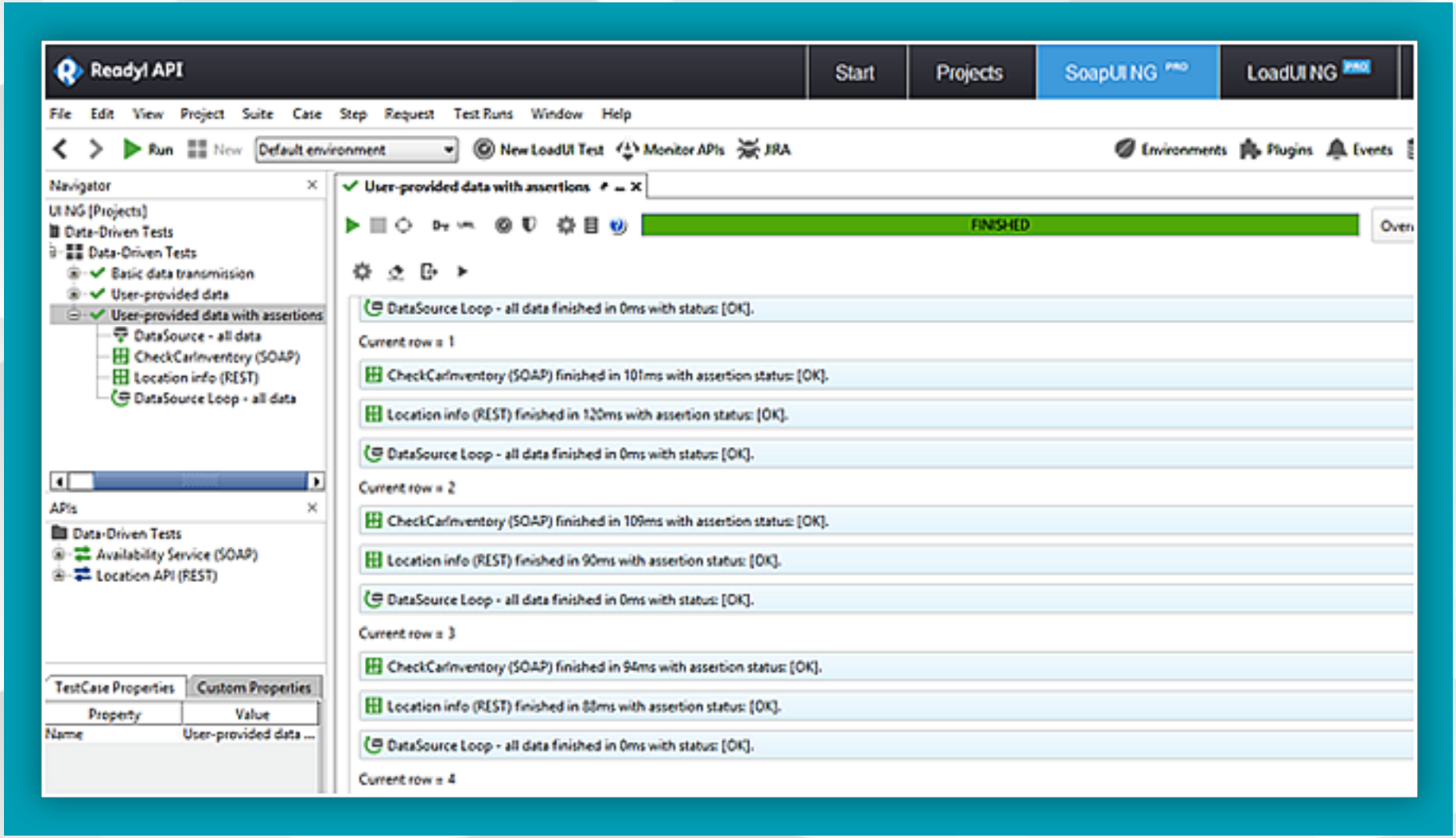


Figure 14: Transaction log for fully automated test

Figure 15 shows details of an error - which was anticipated - that has been returned from the REST API. Since the actual response matches the expected response ("Incorrect"), this test evaluates both positive and negative conditions.

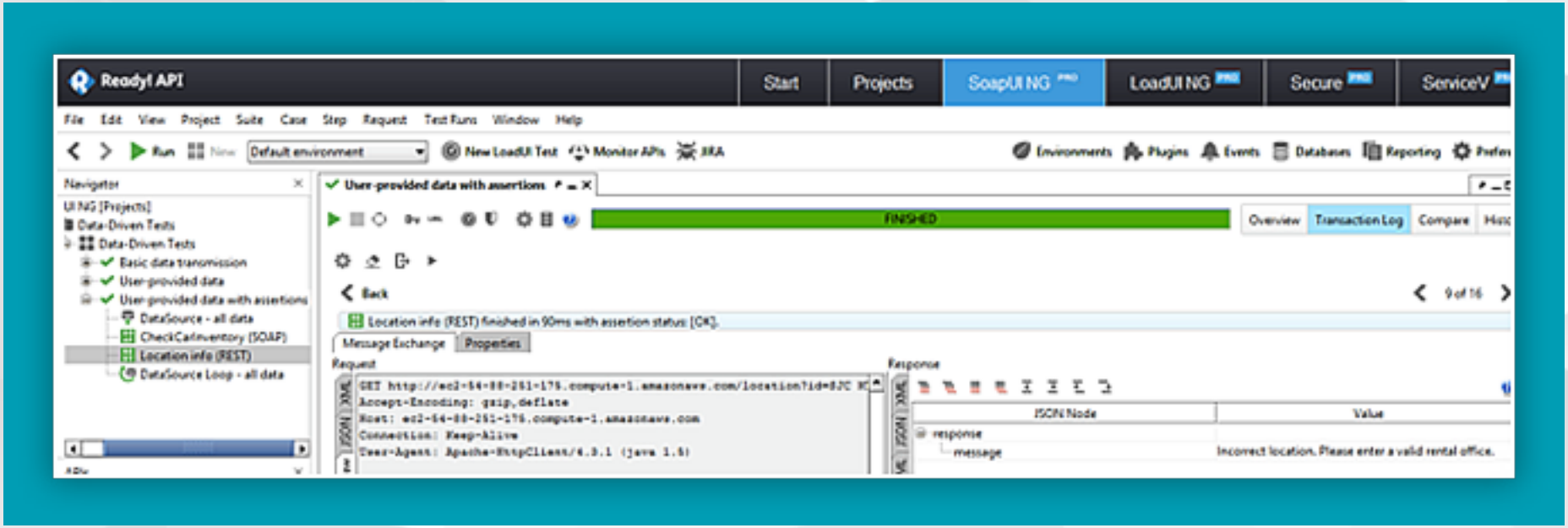


Figure 15: Details of an error message handled by a dynamic assertion

About the Author

Robert D. Schneider is a Silicon Valley–based technologist and author, and managing partner at WiseClouds – a provider of strategic API training and professional services.

He has supplied distributed computing design/testing, database optimization, and other technical expertise to Global 500 corporations and government agencies around the world. Clients have included Amazon.com, JP Morgan Chase & Co, VISA, Pearson Education, S.W.I.F.T., and the governments of the United States, Brazil, Malaysia, Mexico, Australia, and the United Kingdom.

Robert has written eight books and numerous articles on complex technical subjects such as APIs, Big Data, cloud computing, business intelligence, and security. He is also a frequent organizer and presenter at major international technology industry events. Robert blogs at rdschneider.com.

e professionals and
across 194 countries
ear tools

K+
ions

194
countries

sful Customers >>

PERFORMANCE
MONITORING

Synthetic monitoring for API, web, mobile, SaaS, and Infrastructure

SEE MONITORING PRODUCTS

CODE
COLLABORATION

Peer code and documentation review

SEE COLLABORATION PRODUCTS


SoapUI NG Pro