Technical white paper

Getting started with API testing

Test all layers of your composite applications, not just the GUI

Table of contents

Executive summary 3
Introduction 3
Who should read this document? 3
  Developers 3
  Test automation engineers 3
  Test architects 3
  Managers 3
Types of API 4
  Web services 4
  REST 4
  Object-oriented languages 4
  Database 5
  Proprietary APIs 5
Types of test 5
  Unit tests 5
  Integration or system tests 5
  Functional tests 5
  Load or stress tests 5
The world of API testing 5
  Early testing 5
  Collaborating with other developers 6
  Service availability 6
  Managing change 6
  Versioning 6
  Data 6
Getting started 7
  1. Engage an API testing engineer 7
  2. Find a suitable API testing tool 7
  3. Automate your tests 7
  4. Run your tests 7

Click here to verify the latest version of this document

Practicalities of writing a test 8
Import the API 8
Define the test's flow 8
Configure input properties 8
Define checkpoints 8
Run the test 8
Roll the tests out to production 9
API Testing with HP 9
What's next? 10
For more information 10

Executive summary

People use GUIs to interact with their applications, but software uses APIs to talk to the application or its services. Your application almost certainly exposes APIs to the outside world—especially if your application is in "the cloud" and the subsystems your application uses internally communicate with each other via APIs. Just as you need to test the application's GUI, you need to test the APIs; and you need to test them at least as extensively as the GUI, and often even more so, as you have little or no control over how they may be used. This document discusses how you can introduce API test automation in your organization.

Introduction

Composite applications and service-oriented architectures (SOA) call services and components through each service and component's API. The API layer contains any number of services, message queues, database abstraction layers, and other mechanisms that supply important business logic and data to applications. Without proper functional testing of the APIs, poorly designed and functioning services and components can negatively impact the quality of the overall system.

Despite the inherent risk of depending on functionality provided through the API layer, the testing of services and components is often limited to unit tests created by developers, if it's done at all. There might not be a GUI; the service might come from the cloud; it requires an understanding of the API; tools and practices focus on testing after the GUI (if there is one) becomes stable, late in the application lifecycle.

It is simply incorrect to assume that business logic can be adequately validated through the application's GUI. This approach has the obvious disadvantage of waiting until late in the application lifecycle to discover bugs when they're more expensive to fix, but has other shortcomings as well. The GUI might exercise only a subset of the service's functionality. What works for one application may fail later for another that exposes different paths in the business logic. And some aspects of the functionality are not exposed through a GUI, such as performance and security.

This document presents some best practices that you can use to introduce automated API testing into your organization.

Who should read this document?

The intended audience is people who are looking to introduce API testing into their organization. Examples of roles include:

Developers

Developers will learn how to perform unit testing and integration testing of their APIs.

Test automation engineers

Test automation engineers will understand how they can add API testing to their test suites.

Test architects

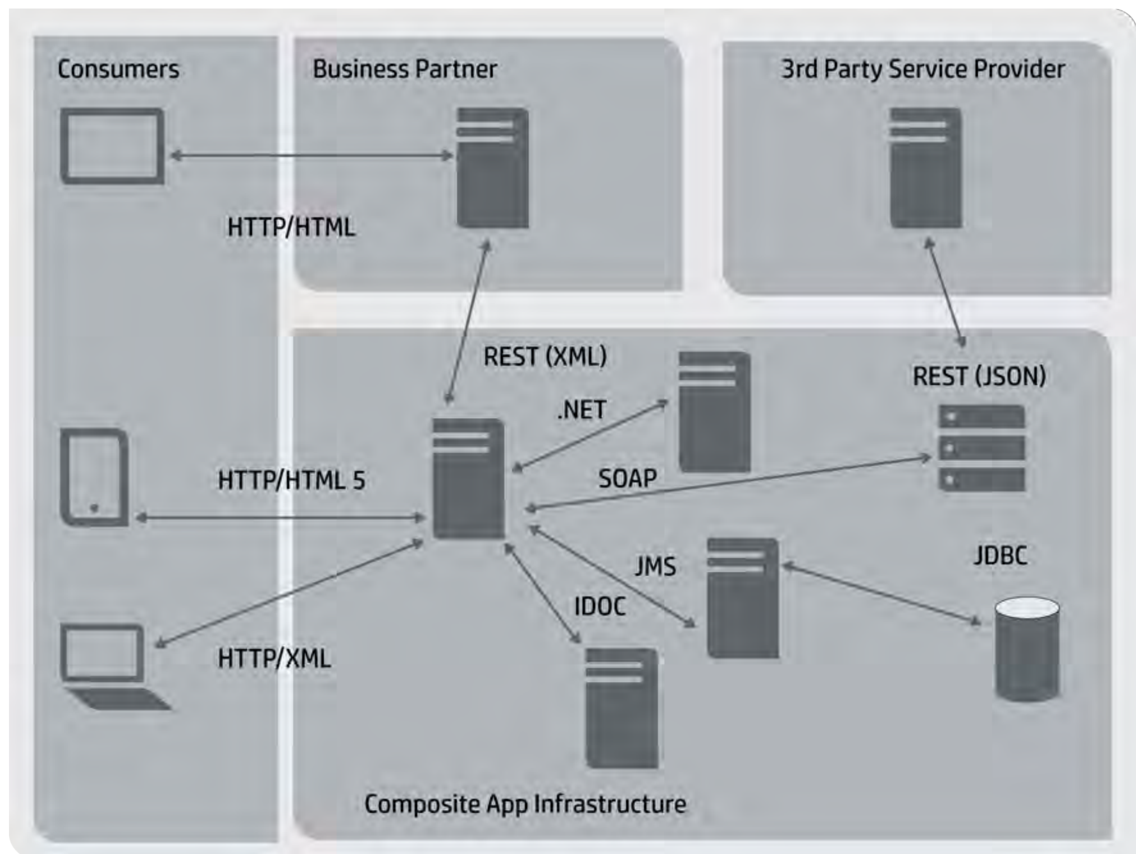
The test architect is typically responsible for designing and implementing the testing framework. Test architects will learn how to include API testing in their frameworks.

Managers

As the most senior stakeholders, managers need to have visibility into the quality of the whole system, including the APIs. They will learn how to introduce API testing as an integral part of the application development lifecycle.

Types of API

Figure 1. A simple composite application



Let's take a quick tour of some of the types of APIs in use today.

Web services

Web services allow computers to talk to each other over a network and are commonly found in SOA environments. They are accessed by sending an XML message called a SOAP request to the server, which responds with a SOAP response.

They are typically described by a WSDL file, which is an XML file (and therefore easily processed by software) detailing the service, its methods and the schemas of the SOAP requests and responses, and other information required to access the service.

Many standards exist around Web services. For example, WS-security specifies how a SOAP message can be encrypted or signed, and WS-addressing specifies where SOAP messages can be sent.

WSDL files can be listed in standard registries which permit easy discovery by software. Universal Description Discovery and Integration (UDDI) is an example of this kind of registry.

REST

REST also allows computers to talk to each other over a network. It involves using HTTP methods to send and receive messages, and does not require a strict message definition, unlike Web services. REST messages often take the form of XML, or JavaScript Object Notation (JSON).

Object-oriented languages

Object-oriented languages such as C# or Java usually expose APIs as classes in a package. Each class contains the methods of the API.

Database

Databases are usually accessed using Open Database Connectivity (ODBC) or Java Database Connectivity (JDBC). These allow interaction with the database using SQL statements.

Proprietary APIs

Some frameworks provide their own APIs, protocols, and data structures. Examples might be SAP's Remote Function Call (RFC) and Intermediate Document (IDoc).

Types of test

There are different types of test that can be performed on an API, and each type is designed to validate a different aspect of the API. This section discusses some of the tests that you should consider as part of the test planning process.

Unit tests

A unit test is a small, atomic test that runs whenever a build of the software is performed, and is ideally part of a continuous integration (CI) system. Developers usually create unit tests at the same time as they develop the APIs, although they can be written beforehand as well, especially in organizations that adopt test-driven development methodologies.

Integration or system tests

Unit tests generally test a single module, and if that module is dependent on other modules, they are emulated using mocks. Integration tests, also called system tests, perform similar tests to unit tests, but instead of working against mock objects and emulated services, they work against real implementations.

These tests can only be performed on a system where all of the modules have been deployed and configured correctly.

Functional tests

Functional tests execute scenarios from a user's perspective by simulating the user's behavior. If we take the previous example of transferring money between accounts, we might want to create a battery of functional tests to check transferring money between different accounts belonging to the same customer, and between accounts belonging to different customers. Functional testing includes negative testing, which ensures the functionality is stable even when invalid values are supplied, and security testing, which highlights vulnerabilities and ensures that the system cannot be abused.

Functional tests are performed directly against the user interface of the application being tested, and should be performed on the API layer.

Load or stress tests

Load tests are designed to ensure that the system performs correctly when subjected to different loads. The system is usually tested by generating a surge or drop in the number of concurrent users, or by running different scenarios concurrently. These tests are also referred to as stress tests when abnormally high loads are placed on the system, which is expected to fail gracefully.

The world of API testing

While API testing has much in common with GUI testing, there are some significant challenges that need to be addressed, as well as some unique opportunities. We'll explore some of these in this section.

Early testing

You can't test an application through its GUI until the GUI is available. But you can test APIs well before the GUI is ready. In fact, many services and APIs are often designed for use by programmatic consumption, and not necessarily by a specific GUI—or indeed by any GUI. These types of APIs must be tested directly. This provides the opportunity to begin the testing process early, because you can start creating tests based on the API's published interface, which is often designed early in the development process. The obvious advantage is that early testing leads to early defect detection, and the earlier defects are detected, the cheaper and easier they are to fix.

Collaborating with other developers

API testing requires a certain amount of technical knowledge, although most API testing tools allow testers with minimal technical knowledge to become productive very quickly. Even if you are very technical, you still need to understand the API's documentation so that you can determine how to test it. This often means talking to other developers, particularly if the documentation is incomplete, as is often the case when testing starts early. Your API testers need to work closely with the developers. In fact, many development teams now include API testers as part of the team and not as part of a separate testing team.

Service availability

Composite applications use services that can be developed in parallel, accessed from third parties, and reused in multiple projects. But there is a downside to these dependencies on other services, because there are times when they are not available, and some services cost money to access. This can be a painful barrier to testing, but it can be mitigated by virtualizing these services. Service virtualization allows these services to be simulated, and allows testing to continue on the dependent service.

Managing change

During development, you often have to work with APIs that are not yet stable, and which are still changing. The same challenges are present when you start testing early; you might find that a test you wrote and ran successfully yesterday doesn't run today because the API changed. Sometimes this might indicate a bug in the API. But when we're testing early in the development cycle, it is often a simple case of a refactoring of the API, which necessitates an equivalent refactoring of the test.

API testing tools offer features that make it easy to manage the changes in APIs, by detecting these changes and automatically updating tests as necessary, taking care to preserve the test's flow.

Versioning

Although APIs change frequently during the early phases of development, there comes a point where the API stabilizes and is released to customers and clients. The customers and clients build their own services and applications on top of these APIs, and are dependent on them. If you then change the API, your clients' services and applications break. This is potentially disastrous and must be managed properly.

When you release an API, you must think very carefully about subsequent changes you make to it. If the change is simply a bug fix to the implementation that doesn't affect the public interface, and doesn't change the behavior of the implementation in any significant way, then you can consider releasing this fix so that existing clients benefit from it, because it shouldn't break their application. But if you change the API, or its behavior, you must make it clear to your customers that this is a new version. You should also ensure that you continue to meet existing contractual obligations around the existing version. When a significant API change is released, it should be released as a new version, which does not automatically override existing versions. Customers must have the choice to proactively consume the new version, or continue to use the existing versions.

This requires you to run your API tests on every fix that is released. You should run tests from previous versions to ensure that the same tests run on the new version. This will ensure that backwards compatibility has been preserved.

To encourage customers to move away from older versions, it is common practice to limit the lifetime of service versions, and require clients to use the new version.

Data

APIs usually work with data structures that define the API's input and output. In most cases, these are not flat lists of properties, but a hierarchical structure that includes substructures, arrays, optional elements, and other data types.

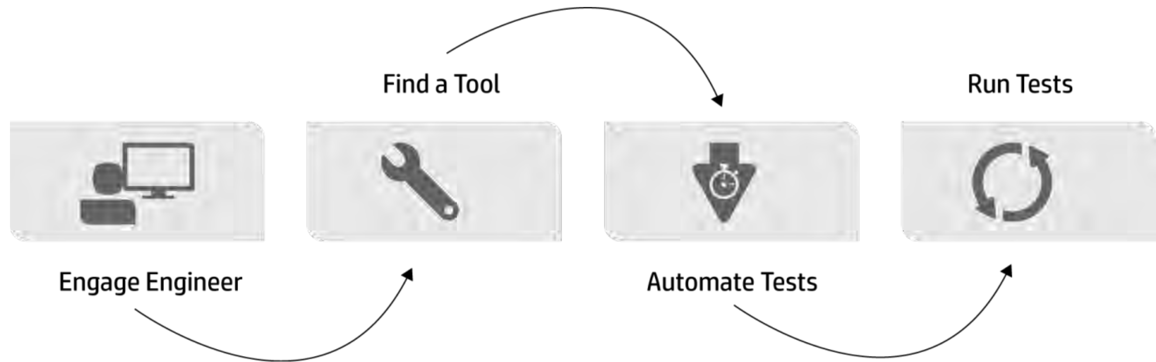
API tests work by supplying data to the API as input and then comparing the values in the API's output to some expected values in a checkpoint. Although APIs may send and receive large structures with many data elements, sometimes the interesting data for the test is only a small subset of what you see. It is important to identify the relevant data you want to deal with. The rest may be static or even optional.

To get data, you need access to production databases or to sample databases that are specially prepared for testing. Either way, it is critical to use actual data values and structures, and not hard-coded values.

Your API testing tool should be able to access these databases and other data sources, and map these data sources to the API being tested. It should allow the same test to be run repeatedly, or iterated, for different inputs and expected outputs, as supplied by the data sources.

Getting started

Figure 2. Getting started with API testing



API testing can be introduced gradually into your organization. Here are some steps that you can take to ease the process.

1. Engage an API testing engineer

To be successful at API testing, you need to have someone on board who is comfortable working with low-level APIs. Even though API test automation tools take a lot of the pain out of the process of testing, you'll get the best results if you understand what you're testing and how the underlying technology works.

2. Find a suitable API testing tool

You'll need to get hold of an API testing tool which to help you create, maintain, and run tests and which supports the technologies you need to test. An example of such a tool is HP Unified Functional Testing 11.50, which allows you to work on API, GUI, and business process tests within a single application, and includes a wide range of technology support. If you only need the API testing capabilities, HP Service Test is also suitable. In order to manage the tests properly, you should consider a system such as HP Application Lifecycle Management, which gives you everything you need to manage your tests, requirements and defects, as well as provide visibility to all stakeholders.

3. Automate your tests

Now you have to decide which tests to automate. To get a quick return on your investment, you should be looking at tests which meet the following criteria:

Stable APIs
The first consideration when starting to automate API testing is how much the API is going to change. If the API changes frequently, you will be spending your time maintaining the tests, rather than writing new tests. So start with APIs that are not expected to change much, and write small tests. As you increase the cover of your stable APIs, you can start testing the less stable APIs, and getting more value out of early testing.

Data-intensive tests
We noted above that APIs could have complex, hierarchical data structures. These data structures can be filled in essentially infinite combinations of values. With a data driven test, you can exercise an API with as many combinations of data as you want, by running the test once for each set of data. API testing tools refer to each of these runs as iterations, where each iteration is a different test case. If you need to test a new situation, you don't need to touch the test—all you need to do is add a new row of data to the test's data source.

4. Run your tests

Once you've written your test, you want it to work for you. The most effective way to do that is to run them whenever the API or service is built, preferably as part of your CI system if you have one. Alternatively, you can run them as part of a test set in a test management system such as HP Application Lifecycle Management.

However, when you run the tests ensure that the results are published to the relevant stakeholders' dashboards, so that they get an indication as soon as any of the tests fail.

Practicalities of writing a test

Figure 3. Stages of writing a test

Let's look at how to actually write an API test. We'll examine each of the steps required.

Import the API

Many APIs are defined by some kind of contract. For example, as we saw above, Web services are described by a WSDL file. The testing tool can parse the WSDL file to discover the Web services and operations it contains, as well as essential information about how to access the service. Once the WSDL file has been imported, the Web service and its operations will be represented within the testing tool and be available to the test.

Define the test's flow

Now that the testing tool knows about our API, you can add an operation from the API to the test flow as a step in the test.

If the operation depends on other operations, you can add these other operations as steps in the flow.

Configure input properties

Each of the steps in the flow needs to be configured with data. While some of the data, such as endpoint addresses are imported along with the contract, you need to provide additional details such as the input data, and perhaps security credentials.

Some of the data required for the step might be hard coded, but to make the test as flexible as possible, you should be able to connect data from other places, such as:

- Previous steps
- Environment or test variables
- Deployment parameters
- Data sources such as Microsoft® Excel files, databases, XML files, and such.

Input properties are typically linked to data sources such as databases, Excel files, and XML files. Your tool should offer the following ways to achieve this.

Data first

You can prepare an Excel file or prime a database with the data your test needs. You can then add the file to the test and map each of the step's input properties to the appropriate column in the file or database. However, it is often impractical to manually prepare an Excel file which can represent the hierarchical structure prevalent in modern APIs.

Step first

Some tools allow you to data drive the step by automatically constructing a data source based on the hierarchical data structure in the step, and adding multiple data sheets, linked together to represent the data structure, and mapped to the step's input properties. All you need to do is add the actual data values.

Define checkpoints

The checkpoint is the validation of the data. It compares the response from the API to an expected response and it is the result of this comparison that determines the success of the test.

As with the input properties, the expected values can be data driven, using the "data first" or the "step first" approach.

Run the test

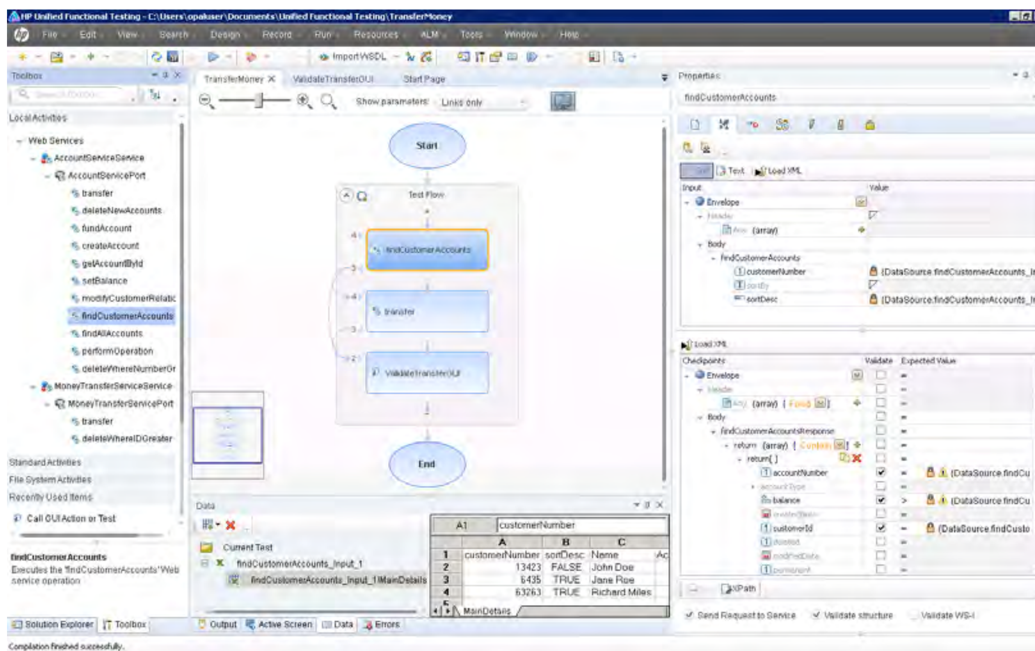
Run the test from the testing tool to make sure it works correctly. You might need to make some corrections to the test's flow, or to the data files that the test uses. Once your test is performing as expected, you can add, integrate it into your development processes.

Roll the tests out to production

Add the tests to your CI suite or to your test management suite and schedule it to run regularly. You and your stakeholders will be able to see the status of each test by looking at the appropriate dashboard, or even by getting an automated email or alert if a failure is detected.

API Testing with HP

Figure 4. HP Unified Functional Testing



In addition to full GUI testing and Business Process Testing, HP Unified Functional Testing (UFT) software provides a full set of API testing capabilities. You can begin testing with a few clicks of a mouse. Simply drag an activity from the toolbox and drop it on to the central canvas where it becomes part of the flow of the test. Common service types supported include Web services, HTTP, REST, JMS, SAP, IBM MQ, Java, C#, and FTP. If you don't need the GUI testing capabilities, HP also provides HP Service Test, which contains everything you need to create and execute API tests.

Each step in the test is configured using the property sheet where you define the flow of execution as well as the flow of data; the output from one step can be the input of another. This approach hides the implementation details from the user, who is no longer required to use a programming language to begin testing. The result is a codeless approach for most testing needs.

HP UFT provides powerful data handling and supports text-based data tables as well as Microsoft Excel files, which can be created to match the format of the input and expected output of each step. XML data sources also provide a natural way of editing and supplying XML structures to steps.

Advanced users and developers can customize the behavior of a test by implementing event handlers for events exposed by a test step, or they can add their own custom code modules to the test flow. Event handlers and custom code modules are written in C#.

A plug in for the Jenkins and Hudson CI systems has also been developed, to allow you to run HP UFT tests as part of your automated build process.

What's next?

The effort required to include API tests in your application development processes may only be a matter of a few hours or a few days in many cases. This section summarizes four easy steps to get up and running with HP UFT and HP Service Test.

Step 1: Get the tools	You need a tool capable of designing API tests. HP UFT provides full testing capabilities for GUI testing, API testing, and business process testing. If you only need API testing capabilities, HP Service Test is suitable.
Step 2: Get additional training	To jumpstart your productivity using HP UFT or HP Service Test, you may need some additional training, which is offered by HP or third-party training vendors.
Step 3: Create the API tests	You are now ready to start creating the tests. Make sure the tests are data driven and that there are checkpoints in the right places. Provide the input and expected responses. Ensure that your tests work correctly.
Step 4: Roll the tests out to production	Add the test to the CI system so that it runs whenever the service is built. HP has a plug in for Jenkins and Hudson that you can use to run HP UFT and HP Service Test API tests.

For more information

To read more about API testing with HP Unified Functional Testing, go to: hp.com/go/uft

Sign up for updates
hp.com/go/getupdated

Rate this document

© Copyright 2013 Hewlett-Packard Development Company, L.P. The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Java is a registered trademark of Oracle and/or its affiliates. Microsoft is a U.S. registered trademark of Microsoft Corporation.

4AA4-4815ENW, January 2013