PROFESSIONAL TESTER

Essential for software testers

October 2012 | £4 / €5 | v2.0 | number 17

Including articles by:

Stephen Janaway

Ole Lensmar
SmartBear Software

Wolfgang Platz
Tricentis

Jeanne Hofmans and
Erwin Pasmans
Improve Quality Services

Martin Mudge
Bugfinders.com

Closing the TestDevOps gap

by Ole Lensmar

From test reuse to test unification

Ole Lensmar explains the special aims and requirements of testing web APIs

Most software is unfinished. As long as it remains in service it is subject to change, and probably will change, in response to new requirements and newly discovered defects. Changes made before deployment are considered good because they reduce the need to make changes after deployment where they are considered bad because they can cause regression. The tension between good and bad change is especially high in the rapidly growing domain of web APIs, which inhabit a much more dynamic environment than most software, including non-web APIs. They are more sensitive to environmental change because they usually have many external dependencies, and their rate of change is higher due to market change, technology change and other forces.

That's why functional and performance monitoring, tuned to the end user experience, is now part of pre-deployment development testing, and why pre-deployment development testing is closely tied to post-deployment functional and performance monitoring. The concept of pre-versus-post deployment testing has already become meaningless. The quality attributes tested before deployment are chosen for their importance after it, and are the same as those tested when subsequent change occurs. The difference is between the testing practices and tools used at different points in the lifecycle. I think of this as testing's own “DevOps gap”. In this article I will discuss how to close it.

The key is to use exactly the same test assets both pre- and post-deployment. Otherwise, there is no way to assess the realism of the lab tests and, if any of them fail in the production environment, no way to tell which. The test environment and tools must make it possible to:

- reuse pre-deployment test assets for post-deployment monitoring
- monitor key functional transactions and performance metrics continuously during pre-deployment testing
- add assertions to tests, for example defining rules about the validity of an element in an XML-encoded response to a given input
- add realism to tests, for example by data-driven testing, inserting varying “think time” and simulating attempted SQL injection.

Assert yourself

Luckily, the nature of web APIs helps with the task of designing a gap-free methodology. By definition, any API-accessible web application comes with its own “command line”, ie the API itself. That removes one reason why pre-deployment software test assets might need to be

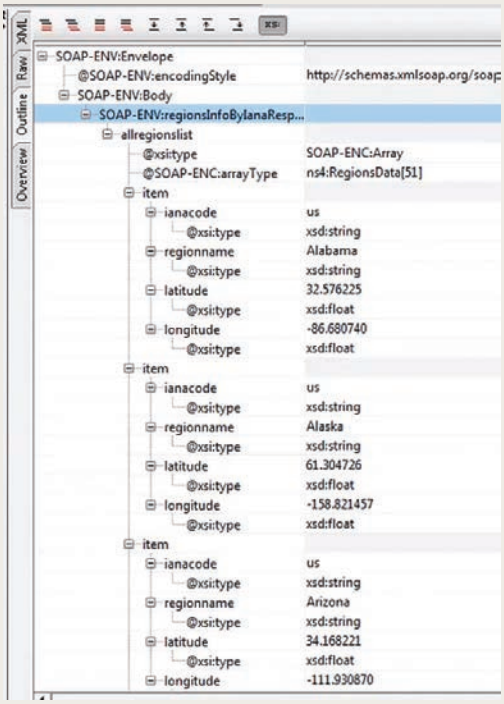

Figure 1: Response message

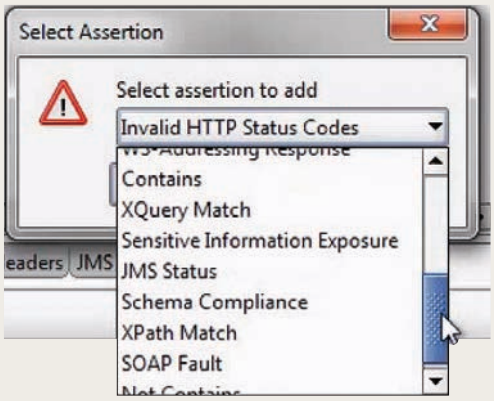

Figure 2: Assertion types

different from post-deployment software test assets. In many non-web situations, developers create a different version of the software specifically to add command lines and create custom scripts whose only purpose is to test against those command lines. After deployment, different test assets are needed to monitor the "real" software from an end-user's perspective, for example additional custom scripts to run against a user interface.

Moving from pre-deployment to post-deployment testing the API doesn't need to change, so there is also no need to change the test script. Every web API exposes operations and/or resources, each of which responds to a particular request: SOAP, REST, HTTP, AMF etc. The response should not change when the API is deployed so neither should test steps.

In functional testing, the correctness of the response is checked using assertions. Here is how that is done using the Open Source tool soapUI. SoapUI generates the requests automatically from the WSDL file. Clicking a request displays it and the API's response message to it (see figure 1). To create an assertion, right-click any element in the response and select the assertion type (figure 2), then use the configuration panel which appears (figure 3) to define your assertion. For example, an "XPath Match" would be used to assert that a target element contains a particular value: otherwise, the test will fail.

Be realistic

Assertions don't change at deployment because the logic exposed by the APIs does not change. What does change is the environment. That includes the underlying data layer, so it is important to craft test assets so that they are not affected by eventual differences between pre- and post-deployment data environments.

The tester's role of course is to design tests that are realistic: in other words, that predict production conditions with sufficient accuracy to detect all important defects before deployment. For web APIs, completely vulnerable to the unknown behaviour of the systems that use them, that's especially difficult to achieve and so it's common, even typical, for web APIs to pass testing but fail in production. But the more realistic the tests, the fewer the unexpected failures and therefore the better their cost and risk (both project and product) are contained. Experience shows that the most likely causes of failure despite effective functional testing are related to unexpected volume, load and security. Therefore realistic testing must include these test types.

Volume testing is best achieved by the data-driven approach. Whereas in functional testing that is used to improve coverage and exhaustiveness, here the aim is to discover requests that cause the APIs to read, process and/or update huge numbers of records from an external data source, and measure how well APIs cope with them.

Testing under load aims to measure how well APIs perform when the numbers of simultaneous requests, and of those with various characteristics, fluctuate: especially when they rise. The most stringent tests are often engineered to include the high demand requests discovered during volume testing. LoadUI

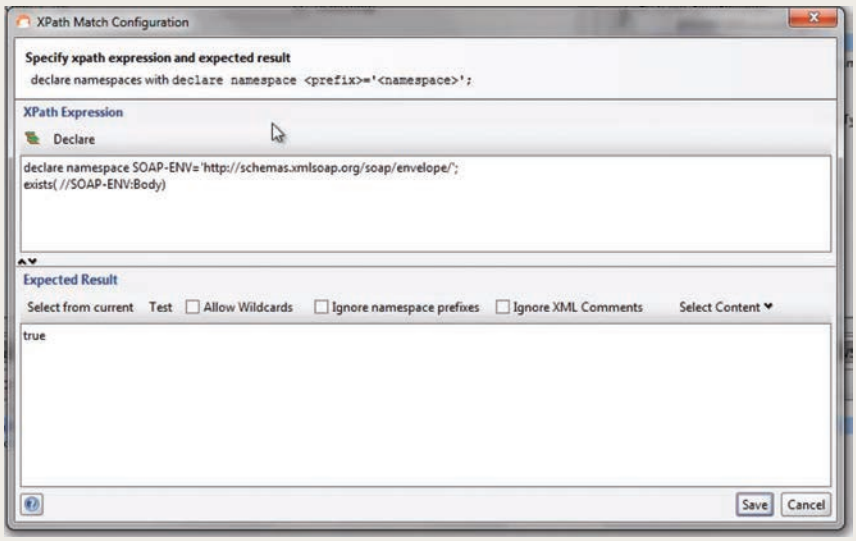

Figure 3: Assertion configuration

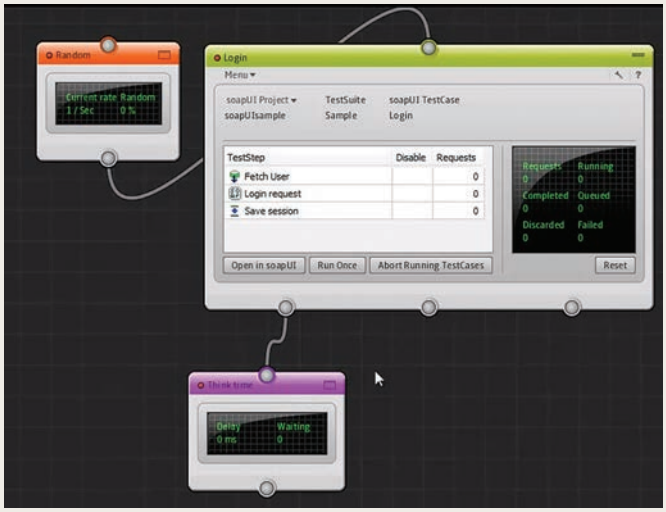

Figure 4: Meter and controller added to a functional test in loadUI

reuses soapUI's functional tests for this purpose and allows meters (for better result capture) and control modules (to improve realism) to be connected to them (figure 4).

As well as executing tests derived from predicted deliberate attacks, for example well-known SQL injection strings such as *' or '1'='1*, web API security testing should aim to detect the presence of inappropriate data in responses to innocent, but failure-causing, requests: common examples include stack traces and third-party software version/configuration information. Both are easier when testing web APIs than other software types, because all inputs are encapsulated in requests rather than more complicated things such as user navigation behaviour, and because negative as well as positive assertions can be used. For example, all tests can include the assertion that the string "Microsoft Windows" does *not* appear in the response. This might lead to some false positives, but they will usually simply confirm that the assertion is working as intended to protect against actual failure.

Turn on, tune in and don't drop out

Unifying pre- and post-deployment testing brings benefit that works in the opposite direction. The results of post-deployment testing are fed back to pre-deployment test design, especially usefully when requirements and/or design change. Understanding the behaviour of APIs in the actual transactional context can also drive operational decisions such as how to re-cluster infrastructure or provision cloud resources. Using the same tests created during development for post-production monitoring gives testers and developers a head start on using and reacting to monitoring because, having designed the tests and resolved past incidents raised from them, they know the context and meaning of its results.

To achieve this, the tests are deployed to the production environment, that is the web. For example, soapUI tests can be executed on more than 80 monitoring sites worldwide comprising SmartBear's AlertSite network, providing full step- and assertion-level results plus response times and other statistics, from actual production conditions. Applying load in production for testing purposes can cause performance or even reliability failure, so careful load design and scheduling are necessary.

Whether or not tests fail, the comparison of results of the same tests executed pre-deployment, post-deployment and after operational change offers insight into what causes variation, for example infrastructure and network issues, usage patterns, user behaviour or unrealistic test data. Ultimately of course what matters is the service delivered to whatever consumes the service provided by the API and how it affects users. A unified test methodology creates a feedback loop whose resonant frequency is user experience. Everything else, including application, test and monitoring design is tuned to that. It never stops ■

Ole Lensmar is CTO and co-founder of SmartBear Software in Sweden, formerly known as Eviware Software, which created soapUI and was acquired by SmartBear in 2011.

The free, Open Source soapUI, and free trials of soapUI pro, loadUI and AlertSite, are available at http://smartbear.com