6 do's and don'ts of service virtualization for developers and QAs

Table of Contents

Three things I always do....1
Do: Focus on keeping it simple....1
Do: Remember what is the System Under Test (SUT) when virtualizing a service....2
Do: Use productized solutions in most cases....3
Three things I avoid doing....4
Avoid: Virtualizing unstable APIs....4
Avoid: Replacing one bottleneck with a new one....4
Avoid: Using wrong metrics....5
Next steps....6

Three things I always do

Do: Focus on keeping it simple

I have worked in environments where systems have been designed for testability from day 1 and where there was no focus on that.

What I have found was that creating stateful virtual services or even simulators was sometimes a necessary short-term solution, especially when you are working in an environment where the System Under Test has not been designed for testability. What I have found was that systems not designed for testability often use external APIs in a way that makes them hard to virtualize.

I have also been on projects where system testability was part of the design process from the beginning. In those environments, stubbing the external APIs was very simple. The virtual services are almost always stateless and never have any logic.

I have always focused on avoiding creating complicated virtual services. That fact that you can simulate a backend system in your virtual service does not mean you should.

Remember that service virtualization is only a part of the big picture of creating software to deliver business impact. What you want to do is help reduce the costs of software delivery as a whole, not only QA is isolation.

Create the simplest virtual services possible. If you need to create a stateful or simulator virtual service, communicate well with architects and developers to make sure they know how much effort goes into it. It might be a cheaper long term option to change/refactor the use of the APIs and the APIs themselves to make them more testable and eliminate the need for complex simulation.

Action for you: find an often used virtual service that is complex and talk about it with developers and architects about it to understand if it a good tradeoff to keep on maintaining it.

P.S. I have seen situations where a lot of effort went into building emulators when you would like to get your builds times from minutes to seconds. If all your automated CI builds run in less than 20 minutes, and you are doing not much manual exploratory testing, and you would still like to reduce the build times, you could explore this option.

Do: Remember what is the System Under Test (SUT) when virtualizing a service

Your system under test is the group of components or applications you are testing. It defines the set of functionalities you are testing.

Let us say we are testing a web application connecting to a backend system. The backend system exposes an API for users to log in. After three unsuccessful logins, the backend system will lock the account and start returning an error indicating that the account has been deactivated.

If I am testing the web application, I would like to see three messages rendered to the user: a successful login, an unsuccessful login, and an account locked message. It can be done with three stateless responses from a virtual service. Then I would proceed to test the backend system API. I would test that after 3 login attempts I get an account locked message. After, that I would proceed to test both systems in an integration environment, but I would not repeat all the tests. I would only test if I can successfully log in. The number of tests in integration environments should be reduced to a minimum giving enough confidence to proceed to production (more on the testing pyramid by Google). So we had three stages of testing, in the first one the SUT was the web application. In the second one, the SUT was the backend system. In the third one, the SUT was the web application together with the backend system. When the SUT is the web application, it does not make sense to create a stateful virtual service that will return an account locked message after three unsuccessful logins. It would create unnecessary complexity. You have to test that functionality in the backend system anyway, so you know it works. All you test in the web application is whether the messages get rendered correctly.

Action for you: what is the system under test you are testing today? What does your testing pyramid look like?

Do: Use productized solutions in most cases

You can write your own stubs, mocks, virtual services, simulators, and emulators. While this is an option to consider, you should take a pragmatic approach to this decision.

Ask yourself questions like:

- What are the functionalities we need be able to use the tool?
- How long will it take us to develop the new tool?
- How much maintenance can we afford to do further down the line?

Then you can compare the tools available on the market:

- Are there any tools available on the market that would satisfy our needs?
- How much do they cost?
- Will it be cheaper or faster to implement the tools in-house and then maintain them long term in-house as well?

You would be surprised to find that there are plenty of open source and commercial tools on the

market. They should satisfy requirements of most teams. I have compiled a list of 40+ tools you can download.

If you want to know more, I have prepared a video introduction for developers.

Action for you: look at the service virtualization tools comparison.

Three things I avoid doing

Avoid: Virtualizing unstable APIs

If you are virtualizing an API, you freeze it in time. What happens if the original API in under heavy development and changes every week significantly enough to cause breaking changes on your side? You need integration contract tests, consumer-driven contract tests or manual exploratory tests to make sure the original API has not drifted from the virtualized version. If you cannot afford development of those automated tests or do not have resources to spend time manually testing it you should consider either using the real version of the API or postponing consumption the APIs. You could be working on other parts of the system as a priority and come back to the unstable API once is a bit more stable.

Action for you: how often do the APIs you are developing against change? Once a week? Once a month? Once a year?

Avoid: Replacing one bottleneck with a new one

You will find many shared virtual service environments in many companies. Many teams will be using the same server with multiple virtual services deployed. While this is an option that works, I have found enabling individual teams to have their instances to be more efficient.

A virtual service or stub per developer or QA, deployable on-demand is an even more attractive option.

I have found that while a shared service virtualization environment is a good option for removing test data setup problems, environment availability, etc. it creates a strong dependency between teams using that instance. So one team doing their testing impacts many other teams. You need to assign port ranges to teams, manage credentials, make sure the hardware is fast enough to support

multiple teams, control the availability of the environment, etc. You suddenly need a new team to manage the virtual service environment

So, instead of having one environment that all teams in the organisation use, I prefer every team or every developer and tester having their environment. You let people use service virtualization however they want on their hardware or laptops. It scales very well.

If you would like to do that, it might become very expensive depending on the tool you use; you need to choose a tool that can run on existing hardware and whose licensing model allows for distributed deployment like Traffic Parrot, or use a cloud offering like GetSandbox.

Avoid: Using wrong metrics

The ideal situation would be to measure, change one thing, and measure again. It is very hard, though.

What I have seen is teams measuring in different ways. Estimating cost savings after the fact (very inaccurate and subjective), counting the number of services virtualized, number of transactions on the virtual service environment. Those things could be a measure of the complexity of the service virtualization implementation you have chosen, but they do not tell you much about the state of your software delivery lifecycle.

I don't measure anything with regards to service virtualization alone. I would look at the state of the software delivery process as a whole. For example, the number of bugs in late stages of development, release lead times, the number of teams blocked, etc. When you want to measure something in the service virtualization space think about for example:

Total amount of time spent waiting for APIs to be delivered (lower is better)

Total amount of time spent waiting for test data set up

Average time to find a defect with big business impact (lower is better)

One exception here is measuring third party transaction costs. It is easy to assess costs savings when you use service virtualization in this case.

Action for you: read about the use of metrics in SDLC.

The end

Thanks for reading. If you disagree, please comment! If you enjoyed this and wanna see more, please like!

Next steps

- Want to see more articles like this? Subscribe to Traffic Parrot on http://blog.trafficparrot.com/
- Request a free 20-minute service virtualization consultation: http://trafficparrot.com/free_consultation_form.html