

# Best Practices: API Testing

Last Updated October 10, 2007

Copyright © Microsoft Corporation. All Rights Reserved.

Created by Josh Poley

## Introduction

If you ask several testers how to write good API tests you will get several different answers. Not only are peoples' styles and approaches different, but often API sets will lend themselves to various methodologies. That being said, my goal here is to describe some of the practices that have worked well for me while testing the Xbox Operating System. Hopefully the information detailed here will be beneficial and useful in your own tests.

## Planning

Before jumping into writing some test code, you will want to spend some time up-front and think about *what* and *how* you will be doing the testing. Depending on the complexity of the task, you will want to create a test specification or design document and have it reviewed by your peers. This document will serve as a road map to your testing, thus making your tests easier to write while providing better test coverage.

## Organization

How you organize your tests will influence your productivity, the effectiveness, and eventually, the maintenance of your tests. It pays to do some initial planning and spend the up-front cost to ensure you have a clean foundation to build upon. Use the below tips to fuel your thinking in this area.

All the tests for a given API should be in a single file, and it makes sense to name the file after the API under test. This makes it easier for others to find and locate all the test cases for a specific function. Some methods should be tested in pairs or groups, so there are times when you will cover multiple APIs in a single file.

In a comment at the top of the file, you should include the declaration of the APIs. This will allow you to quickly reference the parameters and return type. If there are nuances of the API (undocumented or otherwise), it is a good idea to also note them here.

Each test case should be as self-contained and as isolated from dependencies as possible. Make it as easy as possible for the developer to copy an individual test case out and insert it into a stand-alone application or their personal unit test framework. This also implies that test cases should be as simple as possible. You will want to break individual cases down into the lowest layer. You don't want to be using or writing a bunch of custom framework code to wrap or group the APIs under test.

Test cases should be grouped by the test category (straight line cases, boundary cases, null inputs, etc.). This makes it easier for readers and maintainers to scan for a specific test case. It also provides for a logical and understandable means to segment the tests. If you have a large collection of tests, it can be confusing to have all the tests listed one right after another. It is beneficial to group them visually in the code by using comment blocks to break them apart and signify the current test category.

The order of the test cases should also match the test plan (test specification). This makes it easier to ensure that all cases documented in the specification are, in fact, represented in the test code. Alternately, it allows the tester to ensure the two are kept in sync. All too often, the test plan is ignored after it is initially written; by keeping the order the same, it is easier to sync back and forth between them and keep both up to date.

Try to avoid "test chaining" in your development. A chain is where test N does not perform any cleanup because test N+1 relies on the result of the first test case, typically as a setup step. This practice increases the maintenance cost and can lead to a fragile test framework. If a test case needs to perform some setup actions, have the test do that work, even if it is a duplication of work done in a previous test case. If it is easy to do, you should run your tests in a different order, or even with some tests run multiple times. Doing so can uncover additional bugs, but test chains will prevent you from doing this effectively.

## Priorities

When you first start developing your test cases, it is fairly common to jump right into some of the invalid or "evil" tests that will (hopefully) cause crashes and other interesting problems. Unfortunately, this is not necessarily the best use of a tester's time. It is actually better to focus on the mainline cases first. Ask yourself "what are the test cases that must work, and work well?" These are the tests that should be focused on first. A basic guideline is to identify the most common parameters and conditions that an end developer will use when calling the API and test these scenarios extensively.

## Development

The standard for your test code should be as high as that of the product code. As such, be sure to follow your team's coding convention or style guidelines for source code. If your team doesn't currently have one, then actively work with your development counterparts to create a comprehensive guide. Having all the tests follow the same convention is important for maintenance and readability, and is especially helpful if it is the same guideline as the main product code. Working with the development team here provides a good avenue to collaborate on improving the overall quality of the software.

Since your tests will likely be inherited by someone else, be sure to write good clean and well documented code. If it is not immediately clear what the test code is attempting to accomplish or verify, then it needs more comments. As new test cases are added to verify explicit bugs, be sure to add a reference to the bug number which documents the error. If necessary, add comments to describe what the actual fixes were in the product code, this will make it very clear what needs to be covered in the new tests.

The quality of your test code can directly affect the quality of the product. If there are bugs in your test code, then you may be incorrectly testing the product and missing important issues. When compiling your test code, convert all warnings to errors and use the most aggressive warning level possible (for example, Visual Studio's /W4 and /WX, or GCC's -W (and family) and -Werror compiler switches). In addition, if you have the tools available, static code analysis applications can also help find common development problems.

Depending on your test, it can be very beneficial to store state information in easily accessible (via the debugger) variables. If your test breaks during stress, you will want to investigate the machine and look at these variables to quickly glean important information without having to dig very deep. Keeping previous state information around is a good idea too, especially if the test case relies on a sequence of events dealing with arbitrary data. That way, when a test fails at a later point, you can still discover some information about the events that that transpired in the past.

## Configuration

Changing the run-time behavior of your tests is a fairly common practice. Here are some pointers to keep in mind when designing your tests.

At the start of your test, log out any system settings and test parameters that pertain to the run-time environment. From a log file you should be able to recreate the test's exact configuration and parameters; you never want to try and guess how the environment was setup when an error occurs. If each test case does its own configuration setup/initialization, be sure that you are actually testing real world scenarios as your setup could be placing the test in an unrealistic configuration.

If you have a concept of BVTs (build verification tests), regression tests, stress, etc. It is best to have these tests for a given API all located in the same source file/project and use a configuration setting to define which set of tests to run. The BVT tests are often just a few key test cases, whereas regressions and stress tests will encompass a much more extensive set of scenarios.

To aid in tracking down resource leaks, you should make it easy to configure your test project to run a specific set of test cases in a tight loop. If it is not easy to run your tests in this manner, then they *won't* be run in this manner, leaving your components open to leaks. And as a bonus, it makes it straightforward to help isolate an observed leak.

Randomized inputs can help you easily reach a wide variety of states, but make sure your test is smart about the limits, types, formats, etc. of the inputs. It is also very important that the seed (which you need to log) is configurable to allow reproducibility when an error is encountered. And be sure to log out the exact data passed into the API, preferably in a format that makes it easy to create a standalone test case with the exact inputs/states used.

## Validation

Validating the test's behavior is typically considered pretty straight forward. Unfortunately, lack of attention here can often lead to false positives/negatives, or non-actionable bugs. When approaching each test case, make the initial assumption that it currently fails, **and then prove through testing that it works or doesn't work correctly**. To do so, you will want to verify (and then log) a good set of detailed conditions.

To begin with, every time you encounter a failure condition, be sure to log out everything you need to investigate the failure and file a meaningful bug. For example, the below sample provides much more information than just logging out a basic "test failed" string:

```
LogFailure("Unexpected result. Received %lu, expected %lu", GetLastError(), result);
```

A good logging methodology will let you provide a rich collection of useful state information. Consider adding some of the following items to your log output:

- Write out the file name and line number of the test's source code where the error occurred. This allows users to easily jump to the exact location in the test code where an error was first identified. The compiler's `__FILE__` and `__LINE__` macros make this very easy.
- Dump out any resource utilization metrics that make sense for your tests. Memory usage is a usual suspect, but CPU utilization, network bandwidth, disk drive IO, etc. may be useful in the context of your tests.
- Use the logging infrastructure to record performance timing for key scenarios, which can later be analyzed over a period of time to track trends.

Most tests follow a typical "setup, test, cleanup" pattern. Every step in the setup phase should be validated with as much enthusiasm as the actual test itself. If a pre-condition fails, you should record the error (typically as a "this test is blocked" message) and exit out of the test case. Failing to pay attention to initial failures can lead to false negatives. For example, if you expect your API under test to fail, it may be failing for the wrong reason if a pre-condition is not setup correctly.

You should periodically run your tests in an environment/configuration where you expect them to fail. This is a good way to identify tests that provide false positives. For example, if a test is attempting to access a known resource (file path, registry setting, memory location, etc.) then run the test in an environment where the resource does not exist (or the test does not have permission to access it). If a test case "passes" in this type of setup, then it may be making some false assumptions and could be hiding valid bugs.

Testing for assertions and crashes in the API are something that should be done periodically. But these test cases should be coded up and configured so they do not run by default, as you don't want to halt or cause down-stream errors when running a series of automated tests.

If an API has out parameters, set the initial variable's value to a hard-coded but non-standard pattern. This will enable you to easily check whether or not the API modified the out parameter. It will obviously depend on the specified behavior, but for every test case, you should check to see if the value was altered and then verify if it was set to the correct value.

Be sure to also check for over and underflows of buffers and memory locations modified by the API under test. You can create a larger buffer than the API will be modifying, and place known values before and after the expected modified region. Then verify these values are not altered. Alternately, you can use read-only memory pages to trap edits that extend beyond the expected range.

Read-only memory is also good for ensuring that APIs do not alter memory they aren't supposed to. An input buffer marked as `"const"` in the C/C++ language is not guaranteed to preserve the memory (it is a compiler-time check, not a run-time enforcement). By marking memory read-only before passing it into an API, you can quickly discover conditions where the code is misbehaving.

## From Here On Out

Be very careful about when you run tests that have adverse side effects. For example, a test to call `CloseHandle()` twice on one file handle is a good thing to test, but in an isolated environment. Running code like this in conjunction with other tests could be disastrous. If another person's test was issued the handle immediately after the first close (as it was released back into the population), then the second call to `CloseHandle()` will invalidate it and cause the other test to start acting on bad data.

Make it easy for the developers to run your tests. Do whatever it takes to provide all the necessary scripts and configuration files so that all it takes is a double click. If it is super easy for the developer to run your tests, he/she will be more likely to do so before checking in changes.

Now that you have coded up your tests, schedule a full code review and get feedback from your peers. If your test code is buggy itself, then it will be harder to find bugs in the product code. Especially watch out for cut and paste errors, as test cases often follow a similar pattern from one case to the next.

## About the Author

Josh Poley has been a tester at Microsoft since 1998. He initially worked on the very first version of the Passport authentication service (currently called Windows Live ID). Then, in the spring of 2000, Josh moved over and joined a small handful of people who were starting to work on a project code-named Xbox. His initial responsibilities covered

various pieces of the low level operating system (file systems, peripheral communication, etc.). Shortly after the Xbox game console launched in 2001, Josh took over as lead of the Kernel Test Team and remained in charge of validating the core operating system throughout the development and launch of the Xbox 360. Then, in the spring of 2007, Josh joined the Zune team where he is currently working on new media devices.