**[blog.izs.me](https://blog.izs.me/)**

* [tweets](https://twitter.com/izs)
* [gh](https://github.com/sponsors/isaacs)
* [resume](https://izs.me)
* [archive](https://blog.izs.me/archive/)
* [tags](https://blog.izs.me/tags/)
* [about](https://blog.izs.me/about/)

**Test Assertion Styles in JavaScript** [**#**](https://blog.izs.me/2023/09/software-testing-assertion-styles)

[Wes Todd](https://wesleytodd.com) asked me:

reminding myself of [tap](https://node-tap.org)’s api, can you sell me on why people like a test runner to have built in assertions?

I answered and he said it should be a blog post, and I obviously agreed. So here's the infodump that is even more verbose than the one he got in slack.

A lot of it is down to subjective taste, and like a lot of subjective stuff, you kind of have to give yourself time to acquire a taste for a thing in order to really appreciate it. And once you do, you might still eventually decide it's just not for you. I have [my own preferences](https://node-tap.org/#why-tap%3F) in the matter, obviously, but I try to be fair when explaining the pros and cons.

**There are no wrong answers as long as you write tests that help you make good software.** But in general, the reason for having an assertion lib in the test framework (rather than just throwing on any failure) is so that you can have *multiple* assertions in a given batch with less boilerplate, and fully decouple the *test* from the *runner*.

There are two general approaches that are common in JavaScript testing APIs, which I think of as "the **spec family**" and "the **tap family**". These aren't firm categories; it's a bimodal distribution, with quite a bit of overlap.

These families are not unique to JavaScript, and you often see one or the other being more popular with a given language community. JavaScript being what it is, of course, we do everything in every *possible* way, with several different implementations of each.

The most common complaint about JavaScript is the embarassment of riches in open source tooling options, that it moves too fast and there are too many choices. For a language that is both functional and object oriented, running on such a wide array of environments, with the more developers than any other language by a wide margin, this should be no surprise.

In this case, the fundamental divide comes down to where a test framework places its boundary of agnosticism; where it is *a la carte* and where it is *omakase*.

Frameworks in the spec family tend to be agnostic about the means of declaring a test function has passed or failed (any throw will do, do nothing and it passes). Frameworks in the tap family tend to be agnostic about the boundary between the runner and the test suite process (anything emitting TAP on stdout and exiting with 0 or not-0 status will do). Each choice has its pros and cons.

**Spec Family** [**#**](https://blog.izs.me/2023/09/software-testing-assertion-styles/#spec-family)

In the spec family (eg: mocha, lab, jest), you have a block that defines a suite, and within that, you create a test block function. If that test function throws, the test fails, otherwise, it passes. They often (but not always) are paired with assertion libraries that provide a "literate programming" API.

describe('account', () => {  
 it('has a balance of zero when first created', () => {  
 // if this throws, the test fails  
 expect(new Account().balance).to.equal(0)  
 })  
 it('has an empty list of initial addresses', () => {  
 expect(new Account().addresses).to.match([])  
 })  
})

Historically, these are largely inspired by RSpec, and much of the current popularity of this style can be traced back to TJ Holowaychuk's development of ruby-inspired testing tools in the early days of Node.js, eventually culminating in Mocha.

RSpec.describe Account do  
 it "has a balance of zero when first created" do  
 expect(Account.new.balance).to eq(Money.new(0))  
 end  
end

The spec family of JS testing tools rely heavily on decorating the thrown Error object, and building up a tree of suites and tests which are then evaluated. The "protocol" boundary *between the test and harness* is based on direct access to shared (or directly serialized) object structures, and is thus highly opinionated.

Within a given framework, these opinionated parts of the interface can be formalized, so that a variety of reporters can be created to turn the tree of Suite and Test objects into human-readable output, but the barrier to doing so is higher.

Also, the describe/it/context/etc functions (what Mocha calls its "BDD" and "TDD" interfaces) are injected as globals, but only when run with the dedicated runner. Jest takes this a step further, parsing tests and running them in a fully virtualized environment, meaning that they can effectively *only* be run with the test framework's runner.

However, the boundary between the *test functions themselves* and the framework as a whole is completely agnostic. Any throw will do. So, there can be quite a lot of innovation and variety in how those errors are generated and thrown.

**Tap Family** [**#**](https://blog.izs.me/2023/09/software-testing-assertion-styles/#tap-family)

The tap family of tools are traditionally built around the [Test Anything Protocol](http://testanything.org), a text based protocol which describes the results of a test run. TAP is human-readable and machine-parseable, meaning that the results can be inspected directly, turned into reports, stored, aggregated, and so on. Most test frameworks today have the capacity to output TAP, even if they are not built around the protocol as a first-class concern.

Testing tools in this family include tap, tape, and (at least in the presented API, if not all of the internal mechanisms) ava.

Rather than always throwing on a failed assertion and a set of global functions to create suites and test blocks, frameworks in this family expose a single t object with assertion methods that define "test points". Passing test points are reported along with failures in the TAP stream, though these are often omitted from reports.

t.equal(new Account().balance, 0, 'balance of zero when first created')  
t.match(new Account().addresses, [], 'empty list of addresses')

Suites of test points can be grouped using the test method on these objects, nested indefinitely, with each subsequent test blocking getting a t object relating to the subtest.

t.test('account', async t => {  
 const a = new Account()  
 t.equal(a.balance, 0, 'balance of zero when first created')  
 t.match(a.addresses, [], 'empty list of addresses')  
 t.test('credits', async t => {  
 a.credit(100)  
 t.equal(a.balance, -100, 'negative balance after credit')  
 a.debit(200)  
 t.equal(a.balance, 100, 'positive balance after debit')  
 })  
})

TAP started with Larry Wall's perl test scripts, evolving into CPAN's various [Test](https://metacpan.org/dist/Test-Simple/view/lib/Test/Tutorial.pod) libraries, and which in turn spawned an ecosystem of TAP producers in various languages.

Because failing test points don't throw, and passing test points are reported, there's no need for as many layers of enclosing block functions, reducing the boilerplate. Ie, instead of describe(groupName, () => { it(thingName, () => { assert(cond) }) }), the minimum "just test a thing" that shows some output on success is just t.ok(cond, message).

This is less boilerplate for writing tests, but of course noisier output, which is pretty much always addressed by having a reporter that parses the standard TAP output and doesn't bother showing you all the passing tests (unless you want it to).

Because the boundary of agnosticism is between the *test file and the runner*, test files can be anything, and can be run by any any TAP-consuming harness. Or, they can even be piped to a file, and their results reported long after the fact. It's not uncommon to use a runner for tests that are using some other TAP producing test framework, perhaps even in another language. I've seen the perl prove tool used to run node-tap tests, and written tests in bash that are run by node-tap. This means that the framework typically doesn't *have* to create a specially crafted environment for tests, and they can be run any which way, just as (more or less) "normal" programs.

The trade-off is that these tools typically *don't* have as fully agnostic a way to define tests themselves. Because the expectation is that the test generates a TAP stream, and you typically want that to capture passes as well as failures, you see these collections of assertion methods that come along with the framework itself, because they have to be hooked into the structure to generate valid TAP (even though this "in-test" framework can be fully decoupled from the runner).

**It's Software, you can do anything with anything** [**#**](https://blog.izs.me/2023/09/software-testing-assertion-styles/#it's-software-you-can-do-anything-with-anything)

As I said, these are not strict categories, but more of a bimodal distribution of styles and approaches, informed by what the preferences of the creators of these frameworks about what is important to keep flexible, vs what was important to keep consistent.

Pretty much every popular test framework *can* output a TAP stream. And a thrown error in any popular framework will be treated as a test failure and handled reasonably.

But the focus and features of a framework will be designed around its goals, and that tends to show. The TAP stream output by mocha or jest tends to be much less expressive than that emitted by tap or tape; and the level of integration between the runner, reporter, and framework is much reduced in tap family frameworks, which can sometimes limit what they are capable of providing.

***Aside: TAP Family Cousins, \*Unit*** [**#**](https://blog.izs.me/2023/09/software-testing-assertion-styles/#aside:-tap-family-cousins-*unit)

TAP is not the only popular cross-platform protocol for expressing test results as a text stream. Another popular option is the XML format popularized by JUnit, PHPUnit, and xUnit. (Technically xUnit is a somewhat different schema from JUnit, but the principle is the same.)

It's an old joke that Java is a DSL for turning XML into stack traces. JUnit is a DSL for turning stack traces into XML. Because the test results are captured in a formalized text-based format, the results can be saved to a file, reported after the fact, and so on. Anything that generates this text format can interoperate with anything that can consume it.

That's why I consider these test frameworks related to the TAP family; possibly by convergent evolution as much as by direct inspiration, I don't know enough of the history of both to say one way or the other. I've heard that JUnit borrowed a lot from SUnit, but I've also read that SUnit was more of an idea that multiple smalltalk teams tended to adopt, rather than a specific format or piece of software.

**Personally...** [**#**](https://blog.izs.me/2023/09/software-testing-assertion-styles/#personally...)

I like the tap family tools better. I'm not saying that they *are* better, they just appeal to my preferences and style. I *often* want to replay tests, look at the raw data, etc. And my first instinct when I want to zoom in on a test is to just run it as a normal program to be able to poke at it when I'm debugging something or working on a new feature. I get frustrated and annoyed when I can't just run a node program with node and see the results right there. And I often work on libraries and programs where things just won't work properly in a sandboxed testing VM, no matter how cleverly implemented.

Yes, providing a nicely integrated experience can be more challenging with that approach, but I find the separation of concerns beneficial enough to make it worthwhile.

[2023-09-30](https://blog.izs.me/2023/09/software-testing-assertion-styles/)[oss](https://blog.izs.me/tagged/oss/)[testing](https://blog.izs.me/tagged/testing/)[javascript](https://blog.izs.me/tagged/javascript/)[tap](https://blog.izs.me/tagged/tap/)[infodump](https://blog.izs.me/tagged/infodump/)[special interest](https://blog.izs.me/tagged/special-interest/)

* previous: [How to Get Started in Open Source (as a Developer)](https://blog.izs.me/2023/02/how-to-get-started-in-open-source/)