Running 1000 tests in 1s

written by @**marvinhagemeist** 02 October 2022

📖 tl;dr: Most code doesn't require the amount of test isolation modern test runners apply by default. If you only opt into the amount of isolations you need, you can easily run 1000 tests in 1s.

Recently, I **tweeted** that the whole test suite for **Preact**, a modern framework to build web apps, runs in approximately 1s. It's composed of 1003 tests at the time of this writing that run in the browser and assert against the real DOM.

Here is a video of what that looks like:

That tweet struck a nerve, because those numbers aren't even close to be achievable in any of the popular test runners in the frontend space right now. Many newer web developers have never experienced a fast test suite that runs in 1s. Naturally, the question popped up as to what it is that made Preact's test suite so fast.

Establishing a baseline

Before we begin, we should have a rough baseline of how fast it could be in theory. To do that we need to know what kind of tests we want to run. In Preact's case it's mostly about asserting that it rendered the correct output. For that we prefer to test against the real DOM inside one or potentially many browsers to ensure compatibility. The majority of test cases look similar to this:

```
// Create a DOM container element to render into
const container = document.createElement("div");

// Render `<p>foo</p>` into `<div id="app />`
render(<p>foo</p>, container);

// Check if the rendered HTML is correct.
expect(container.innerHTML).toEqual("<p>foo</p>");
```

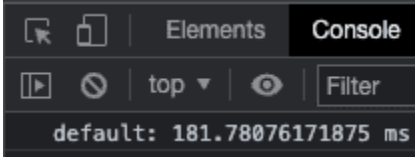
Let's simplify this even further so that we could copy and paste this snippet into the browser. And you know what: We'll flex the browser's muscle a little and run this test 100000 times.

```
import { render, h } from "https://esm.sh/preact";

console.time();
for (let i = 0; i < 100_000; i++) {
  const container = document.createElement("div");
  render(h("p", null, "foo"), container);

  if (container.innerHTML !== "<p>foo</p>") {
    throw new Error("fail");
  }
}
console.timeEnd();
```

Result:



On my machine (MacBook Air M1) all 100000 tests complete in about `~182ms`. Sure, the test case is a bit simplified and we definitely have a few more complex tests in our suite, but these timings are what we should keep in the back of our minds as a baseline of how fast things could be. Anything that we add to our setup from this point on is merely overhead.

Blast from the past

Those that have been around for a while, know that we the most popular test runners were different back in the day. There was **Mocha**, **Jasmine**, **tap**, **tape** and many more. What they all had in common was speed. Let's check that for ourselves by adding Mocha to the mix. A few script tags later after hitting refresh we get some a new number:

~210ms

Adding Mocha added about `~35ms` of overhead. That's plenty fast for a test runner that has been around since 2011. Jasmine is even older, being published in 2008 and runs similarly quick.

This poses the question: Why are those older test runners so much faster?

Living in an isolated world

The answer to that question is: Test isolation. With the rise of `Promises` and later `async/await`, writing asynchronous code became more popular and with that the drive for test runners to invest into test isolation. With every test having their own environment they can reliably ensure that no test interferes with another by mutating globals or other forms of shared state.

```js
// Module has a global variable
let count = 0;

export function increment() {
  count += 1;
  return count;
}
export function decrement() {
  count -= 1;
  return count;
}
```

And here is how we use that module in our test file:

```js
import { increment, decrement } from "./my-app";

it("increment", () => {
  expect(increment()).toEqual(1);
});

it("decrement", () => {
  // The assertion fails if `count` isn't reset between
  // test runs.
  expect(decrement()).toEqual(-1);
});
```

Ensuring that developers don't run into this trap can be a good thing. But it comes at a cost: speed. Creating and tearing down environments for each single test takes a lot of time. So much time in fact, that the majority of time in today's test suites is spent on exactly this. Running the tests itself barely shows up when you trace your test runner. It pales in comparison to the overhead of test isolation.

Thing is, most tests don't need that level of isolation. If we have a function that entirely operates on its inputs and doesn't mutate shared state, then any form of isolation is unnecessary. Yet, we run the whole environment creation and tear down cost regardless, because the test runner applies these by default. Let's illustrate that with an example:

```js
// Example code that doesn't rely on shared state at all.
// The function only relies on its inputs.
function add(a, b) {
  return a + b;
}
```

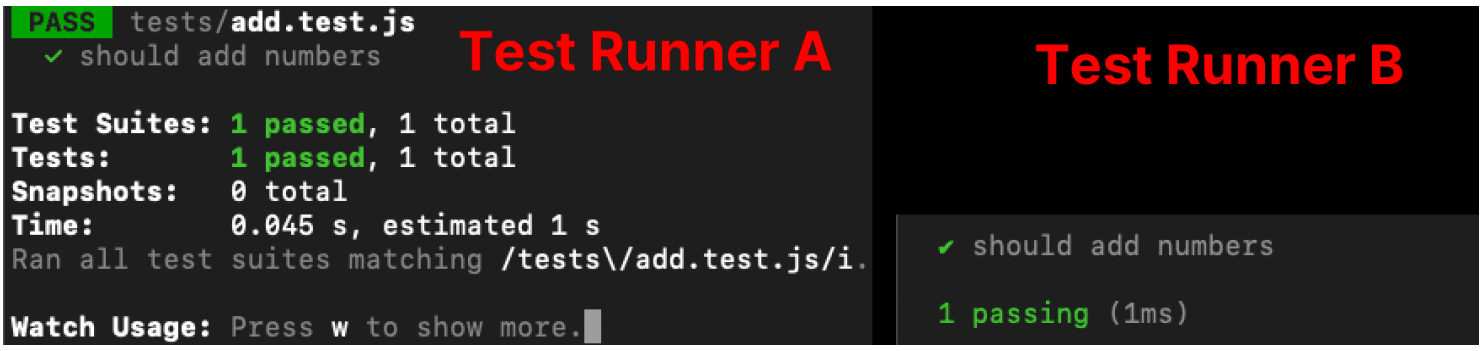
This simple `add()` function doesn't need isolation at all. It can be safely called as often as you want, regardless if it's called in a synchronous context or an asynchronous one.

When can visualize the cost of isolating across test runners, by running a simple addition test and measuring their time.

```js
function add(a, b) {
  return a + b;
}

it("should add numbers", () => {
  // We intentionally don't use an assertion library here
  // to avoid tainting timings by that.
  if (add(1, 2) !== 3) {
    throw new Error("fail");
  }
});
```

Let's run those tests:



We can already see some difference, but let's scale that up to a 100 tests by duplicating the `add.test.js` test file a hundred times.

Checking if the addition of two numbers is correct for a hundred times takes a full `1.133s` in test runner A and just `10ms` in runner B. That's about **114x** slower than test runner B.

If we think about how many computations modern processors can do per second, even the `10ms` result is super slow. But despite that it's still so much faster than test runner A due to not having to constantly create and tear down test environments.

What about the DOM?

For Preact though, we *have* to write to the DOM which is a form of shared state. Creating and destroying the whole document for every test is immensely expensive, so we want to avoid doing that. Instead we leverage a bit of knowledge about Preact to speed up our tests. Preact, like other frontend frameworks, renders *into* a single element. So we only need to supply a unique element per test case.

```
describe("my test suite", () => {
  let container;

  // Runs before every single test case
  beforeEach(() => {
    container = document.createElement("div");
  });

  // Runs after every single test case, regardless if it
  // failed or succeeded.
  afterEach(() => {
    container.remove();
  });
```

```
  it("should render", () => {
    render(<p>hello world</p>, container);
    expect(container.innerHTML).toEqual("<p>hello world</p>");
  });

  //...more tests
});
```

In doing so we set the amount of isolation we need ourselves and avoid spending time unnecessarily setting up isolation containers.

Global state

Some aspects of Preact, like hooks, require a global variable to track the component that's currently being rendered. This is not something unique to Preact's implementation, but rather how the hook API is designed. We could add a way to reset that state similar to how we do with the DOM, but that would introduce a code-path into Preact that's solely for testing purposes. That's something we don't want to do, as we'd rather test Preact in a way that is as close as possible to how it's used by real users.

For the longest time in Preact's history we just dealt with failures caused by global variables not being reset. And to be honest, it was totally fine. It rarely happened and when it did, it pointed to a bug in Preact itself that we would've missed otherwise. Because whenever something fails in Preact when users use it, it should not end up in an invalid state anyway. So essentially by us not adding any form of isolation for global state, we got free testing out of that.

To make debugging a little easier, we added a way to scope globals to a single test file. That way the terminal isn't spammed with hundreds of failing test cases, but rather to only a few. We do this by creating a separate bundle for every single test file. Everyone of them gets their own copy of Preact and everything else.

```
/tests
  render.test.jsx // <- has their own copy of Preact
```

```
component.test.jsx // <- has their own copy of Preact too
```

This works because our global variables aren't truly global in the sense that they *have* to dangle off of the `window` -object. Instead they're more akin to module scoped variables. So a new instance of the module, creates a new instance of the variable. Creating a unique bundle per test file takes care of that.

Preact's test setup

We achieve this through the amazing bundling speed by **esbuild** and then cache every bundle that isn't invalidated by file changes during the development process. So whenever you edit a file and hit save, we'll re-generate only the bundles that included that file, ping `karma` (coordinates the browser) when we're done and `karma` just reloads the browser.

All this wouldn't be possible without the compilation speed of `esbuild` . It's basically so fast that it barely makes a difference, if you create one or more bundles. So we happily traded off the slight hit in test run time in favor of a little nicer debugging experience.

Our whole testing stack basically consists of `karma` as the browser coordinator, `esbuild` as the bundler and `mocha` as the test runner. If there's one thing I'd love to replace from that stack it would be `karma` , because the APIs feel a little dated. But then again I'd happily deal with its oddities, when it allows me to run a 1000 tests in 1s.

Let's make test suites fast

In summary, the problem with applying the maximum possible isolation is that everything becomes painfully slow. Knowing which level of isolation you need is a sure way to speed up any test suite. The defaults the currently popular test runners cover have them all turned on to accommodate the widest range of possible use cases. That's why they are so slow.

Them being so popular also introduces more of a cultural problem. By now we have a generation of developers who have grown up with those tools being the default ones to pick. They don't know that their tests could run faster, because they've never experienced it. As Andrew puts it on twitter:

And they're right. There is nothing novel about our approach or how we set up our test suite. Both `karma` and `mocha` have been around for what feels like forever in web development. They are not new. Only `esbuild` is somewhat newer. But then again the usage of `esbuild` isn't unique to our stack either.

What's unique to the Preact team though, is that we always go with what the best option is for ourselves and we fill any blind spots in gluing or creating tooling to achieve that. This requires us to be aware of which amount of isolation we need and want from our test suite, and then only applying that. Nothing more, nothing less.

Because one thing is for sure: Working on Preact is way more fun if all tests complete nearly instantaneous, in 1 second.