Version: 27.1

Timer Mocks

The native timer functions (i.e., `setTimeout()`, `setInterval()`, `clearTimeout()`, `clearInterval()`) are less than ideal for a testing environment since they depend on real time to elapse. Jest can swap out timers with functions that allow you to control the passage of time. Great Scott!

> **NOTE**
>
> The default timer implementation changed in Jest 27 and is now based on @sinonjs/fake-timers.

Enabling fake timers

/examples/timer/modern/timerTest.js

```
function timerTest(callback) {
  setTimeout(() => callback('Timer finished!'), 10000);
}

module.exports = timerTest;
```

/examples/timer/modern/__tests__/timerTest.spec.js

```
test('should invoke callback after timer ends', () => {
  const timerTest = require('../timerTest');
  const callback = jest.fn();

  // Enable mocking of native timer functions
  jest.useFakeTimers();

  // Add a 10 second timer to invoke a mocked callback function
  timerTest(callback);

  // The callback should not have been called yet
  expect(callback).not.toHaveBeenCalled();

  // Fast-forward until all timers have been executed
```

```
  jest.runAllTimers();

  // Assert successfully without having to wait for the 10 second delay
  expect(callback).toBeCalledWith('Timer finished!');
});
```

Here we call `jest.useFakeTimers()` to replace `setTimeout()` and other native timer functions with mock functions. And since we use `jest.runAllTimers()` to fast forward in time, the callback has a chance to run before the test completes.

You are in the driver's seat

Fake timers will not automatically increment with the system clock. When they are activated, time is essentially frozen until you say otherwise (or until Jest times out).

In the previous example, we use `jest.runAllTimers()` to fast forward and run all tasks queued by the mocked timer functions. Without telling Jest to advance time like this, our callback would never be invoked.

You can also have more fine-grained control with for example `jest.advanceTimersByTime()` or `jest.advanceTimersToNextTimer()`. See the API documentation for more information.

Faking timers is a global operation

Calling `jest.useRealTimers()` will turn on fake timers for all tests within the same file, until normal timers are restored with `jest.useRealTimers()`.

The fake timers also have a global state that only resets each time you call `jest.useFakeTimers()`.

So while `jest.useFakeTimers()` and `jest.useRealTimers()` can be called from anywhere (top-level, inside a `test` block, etc.), you need to be careful in order to avoid unexpected behavior.

```
test('do something with fake timers', () => {
  jest.useFakeTimers();
  // ...
});

test('do something with real timers (?)', () => {
  // This would still use fake timers
```

```
});
```

In this example, since we never call `jest.useRealTimers()`, both tests would end up using fake timers when run synchronously. And the state (time/internal counters) of the fake timers would leak across since we did not reset it.

A better approach could be:

```
afterEach(() => {
  // Reset to real timers after each test
  jest.useRealTimers();
});

test('do something with fake timers', () => {
  jest.useFakeTimers();
  // ...
});

test('do something else with fake timers', () => {
  jest.useFakeTimers();
  // ...
});

test('do something with real timers', () => {
  // ...
});
```

Here we use an `afterEach` to call `jest.useRealTimers()` after every single test.

This pattern can help keep behavior predictable as your test file grows.

- It will establish a clear baseline -> "timers are real unless otherwise is set".
- The state of the fake timers is always reset between tests, since all tests wanting to use them have to call `jest.useFakeTimers()` first.

Handling recursive timers

Scenarios exist where you might have a recursive timer -- for example a function that sets a timer to call the same function again.

If you use `jest.runAllTimers()` to advance time, it will execute all pending tasks. If those tasks themselves schedule new tasks, those will be continually exhausted until there are no

more tasks remaining in the queue. So with a recursive timer, you could end up with an infinite loop.

To solve this, Jest ships with an alternative called `jest.runOnlyPendingTimers()`. This will run only the tasks queued by `setTimeout()` or `setInterval()` up until that point.

/examples/timer/modern/infiniteTimerTest.js

```
function infiniteTimerTest(callback) {
  // Schedule the infiniteTimer() to start in 10 seconds
  setTimeout(() => infiniteTimer(callback), 10000);
}

function infiniteTimer(callback) {
  callback('infiniteTimer: start');
  setTimeout(() => {
    callback('infiniteTimer: setTimeout');
    // Invoke itself to immediately schedule another timer in a recursive loop
    infiniteTimer(callback);
  }, 10000);
}

module.exports = infiniteTimerTest;
```

/examples/timer/modern/__tests__/infiniteTimerTest.spec.js

```
test('should not start recursive timer loop', () => {
  const infiniteTimerTest = require('../infiniteTimerTest');
  const callback = jest.fn();

  jest.useFakeTimers();

  infiniteTimerTest(callback);

  // At this point only the timer in infiniteTimerTest() is scheduled,
  // and time is frozen. So infiniteTimer() or the callback passed to it
  // should not have been invoked.
  expect(callback).not.toHaveBeenCalled();

  jest.runOnlyPendingTimers();

  // Now the callback should have been invoked only once, at the beginning of
  // infiniteTimer(). The setTimeout() within should have been ignored.
  expect(callback).toHaveBeenCalledTimes(1);
  expect(callback).toHaveBeenCalledWith('infiniteTimer: start');
});
```

Thanks to using `jest.runOnlyPendingTimers()`, the test finishes successfully.

If you replace it with `jest.runAllTimers()`, you will see that Jest throws an error:

```
Aborting after running 100000 timers, assuming an infinite loop!
```

Yikes... that's a lot of timers!

More code examples

TODO: Info about / link to more code examples here.

Frequently Asked Questions

Timer functions are no longer a mock or spy function. How can I assert against them?

In the legacy fake timers, prior to Jest 27, native timer functions were replaced by Jest mock functions. After moving to a new implementation based on an external library, this is no longer a feature.

You can still spy on the global yourself after enabling the fake timers, but it would generally be more robust to test against what you expect to happen when a task runs. The specific timer function used should be considered an implementation detail.

Looking back at the first code example on this page, you will see that we assert if the mocked callback function is called with the parameter we expect. This decouples us from the details of what happens inside `timerTest()`. At some point in the future, we can safely replace `setTimeout()` with another timer without invalidating the test.

If you need to verify that callbacks are scheduled with a particular time or interval, consider using `jest.advanceTimersByTime()` and make assertions based on what you expect at different points in time.

TODO: Polish this...

Edit this page

*Last updated on **8/31/2021** by **Sigve Hoel***