Jest cheat sheet

I recommend Mrm and jest-codemods for single-command Jest installation and easy migration from other frameworks.

Test structure

```js
describe('makePoniesPink', () => {
  beforeAll(() => {
    /* Runs before all tests */
  })
  afterAll(() => {
    /* Runs after all tests */
  })
  beforeEach(() => {
    /* Runs before each test */
  })
  afterEach(() => {
    /* Runs after each test */
  })

  test('make each pony pink', () => {
    const actual = fn(['Alice', 'Bob', 'Eve'])
    expect(actual).toEqual(['Pink Alice', 'Pink Bob', 'Pink Eve'])
  })
})
```

Matchers

Using matchers, matchers docs

Basic matchers

```js
expect(42).toBe(42) // Strict equality (===)
expect(42).not.toBe(3) // Strict equality (!==)
expect([1, 2]).toEqual([1, 2]) // Deep equality
expect({ a: undefined, b: 2 }).toEqual({ b: 2 }) // Deep equality
expect({ a: undefined, b: 2 }).not.toStrictEqual({ b: 2 }) // Strict equality (Jest 23+)
```

Truthiness

```
// Matches anything that an if statement treats as true (true, 1, 'hello', {}, [], 5.3)
expect('foo').toBeTruthy()
// Matches anything that an if statement treats as false (false, 0, '', null, undefined, NaN)
expect('').toBeFalsy()
// Matches only null
expect(null).toBeNull()
// Matches only undefined
expect(undefined).toBeUndefined()
// The opposite of toBeUndefined
expect(7).toBeDefined()
// Matches true or false
expect(true).toEqual(expect.any(Boolean))
```

Numbers

```
expect(2).toBeGreaterThan(1)
expect(1).toBeGreaterThanOrEqual(1)
expect(1).toBeLessThan(2)
expect(1).toBeLessThanOrEqual(1)
expect(0.2 + 0.1).toBeCloseTo(0.3, 5)
expect(NaN).toEqual(expect.any(Number))
```

Strings

```
expect('long string').toMatch('str')
expect('string').toEqual(expect.any(String))
expect('coffee').toMatch(/ff/)
expect('pizza').not.toMatch('coffee')
expect(['pizza', 'coffee']).toEqual([expect.stringContaining('zz'), expect.stringMatching(/ff/)]
```

Arrays

```
expect([]).toEqual(expect.any(Array))
expect(['Alice', 'Bob', 'Eve']).toHaveLength(3)
expect(['Alice', 'Bob', 'Eve']).toContain('Alice')
expect([{ a: 1 }, { a: 2 }]).toContainEqual({ a: 1 })
expect(['Alice', 'Bob', 'Eve']).toEqual(expect.arrayContaining(['Alice', 'Bob']))
```

Objects

```
expect({ a: 1 }).toHaveProperty('a')
expect({ a: 1 }).toHaveProperty('a', 1)
expect({ a: { b: 1 } }).toHaveProperty('a.b')
expect({ a: 1, b: 2 }).toMatchObject({ a: 1 })
expect({ a: 1, b: 2 }).toMatchObject({
  a: expect.any(Number),
  b: expect.any(Number),
})
expect([{ a: 1 }, { b: 2 }]).toEqual([
  expect.objectContaining({ a: expect.any(Number) }),
  expect.anything(),
])
```

Exceptions🔗

```
// const fn = () => { throw new Error('Out of cheese!') }
expect(fn).toThrow()
expect(fn).toThrow('Out of cheese')
expect(fn).toThrowErrorMatchingSnapshot()
```

▶ Aliases

Snapshots🔗

```
expect(node).toMatchSnapshot()
// Jest 23+
expect(user).toMatchSnapshot({
  date: expect.any(Date),
})
expect(user).toMatchInlineSnapshot()
```

Mock functions🔗

```
// const fn = jest.fn()
// const fn = jest.fn().mockName('Unicorn') -- named mock, Jest 22+
expect(fn).toBeCalled() // Function was called
expect(fn).not.toBeCalled() // Function was *not* called
expect(fn).toHaveBeenCalledTimes(1) // Function was called only once
expect(fn).toBeCalledWith(arg1, arg2) // Any of calls was with these arguments
expect(fn).toHaveBeenLastCalledWith(arg1, arg2) // Last call was with these arguments
expect(fn).toHaveBeenNthCalledWith(callNumber, args) // Nth call was with these arguments (Jest
expect(fn).toHaveReturnedTimes(2) // Function was returned without throwing an error (Jest 23+)
expect(fn).toHaveReturnedWith(value) // Function returned a value (Jest 23+)
expect(fn).toHaveLastReturnedWith(value) // Last function call returned a value (Jest 23+)
expect(fn).toHaveNthReturnedWith(value) // Nth function call returned a value (Jest 23+)
expect(fn.mock.calls).toEqual([
  ['first', 'call', 'args'],
  ['second', 'call', 'args'],
]) // Multiple calls
expect(fn.mock.calls[0][0]).toBe(2) // fn.mock.calls[0][0] — the first argument of the first cal
```

▶ Aliases

Misc

```
expect(new A()).toBeInstanceOf(A)
expect(() => {}).toEqual(expect.any(Function))
expect('pizza').toEqual(expect.anything())
```

Promise matchers (Jest 20+)

```
test('resolve to lemon', () => {
  expect.assertions(1)
  // Make sure to add a return statement
  return expect(Promise.resolve('lemon')).resolves.toBe('lemon')
  return expect(Promise.reject('octopus')).rejects.toBeDefined()
  return expect(Promise.reject(Error('pizza'))).rejects.toThrow()
})
```

Or with async/await:

```
test('resolve to lemon', async () => {
  expect.assertions(2)
  await expect(Promise.resolve('lemon')).resolves.toBe('lemon')
  await expect(Promise.resolve('lemon')).resolves.not.toBe('octopus')
})
```

resolves docs

Async tests

See more examples in Jest docs.

It's a good practice to specify a number of expected assertions in async tests, so the test will fail if your assertions weren't called at all.

```
test('async test', () => {
  expect.assertions(3) // Exactly three assertions are called during a test
  // OR
  expect.hasAssertions() // At least one assertion is called during a test

  // Your async tests
})
```

Note that you can also do this per file, outside any `describe` and `test` :

```
beforeEach(expect.hasAssertions)
```

This will verify the presense of at least one assertion per test case. It also plays nice with more specific `expect.assertions(3)` declarations.

In addition, you can enforce it globally, across all test files (instead of having to repeat per file) by adding the exact same line into one of the scripts referenced by the `setupFilesAfterEnv` configuration option. (For example, `setupTests.ts` and that is referenced via a `setupFilesAfterEnv: ['<rootDir>/setupTests.ts']` entry in `jest.config.ts`.)

async/await

```
test('async test', async () => {
  expect.assertions(1)
  const result = await runAsyncOperation()
  expect(result).toBe(true)
})
```

Promises

Return a Promise from your test:

```
test('async test', () => {
  expect.assertions(1)
  return runAsyncOperation().then((result) => {
    expect(result).toBe(true)
  })
})
```

done() callback

Wrap your assertions in try/catch block, otherwise Jest will ignore failures:

```
test('async test', (done) => {
  expect.assertions(1)
  runAsyncOperation()
  setTimeout(() => {
    try {
      const result = getAsyncOperationResult()
      expect(result).toBe(true)
      done()
    } catch (err) {
      done.fail(err)
    }
  })
})
```

Mocks

Mock functions

```
test('call the callback', () => {
  const callback = jest.fn()
  fn(callback)
  expect(callback).toBeCalled()
  expect(callback.mock.calls[0][1].baz).toBe('pizza') // Second argument of the first call
  // Match the first and the last arguments but ignore the second argument
  expect(callback).toHaveBeenLastCalledWith('meal', expect.anything(), 'margarita')
})
```

You can also use snapshots:

```
test('call the callback', () => {
  const callback = jest.fn().mockName('Unicorn') // mockName is available in Jest 22+
  fn(callback)
  expect(callback).toMatchSnapshot()
  // ->
  // [MockFunction Unicorn] {
  //   "calls": Array [
  // ...
})
```

And pass an implementation to `jest.fn` function:

```
const callback = jest.fn(() => true)
```

Mock functions docs

Returning, resolving and rejecting values

Your mocks can return values:

```
const callback = jest.fn().mockReturnValue(true)
const callbackOnce = jest.fn().mockReturnValueOnce(true)
```

Or resolve values:

```
const promise = jest.fn().mockResolvedValue(true)
const promiseOnce = jest.fn().mockResolvedValueOnce(true)
```

They can even reject values:

```
const failedPromise = jest.fn().mockRejectedValue('Error')
const failedPromiseOnce = jest.fn().mockRejectedValueOnce('Error')
```

You can even combine these:

```
const callback = jest.fn().mockReturnValueOnce(false).mockReturnValue(true)

// ->
//  call 1: false
//  call 2+: true
```

Mock modules using `jest.mock` method

```
jest.mock('lodash/memoize', () => (a) => a) // The original lodash/memoize should exist
jest.mock('lodash/memoize', () => (a) => a, { virtual: true }) // The original lodash/memoize is
```

jest.mock docs

> Note: When using `babel-jest`, calls to `jest.mock` will automatically be hoisted to the top of the code block. Use `jest.doMock` if you want to explicitly avoid this behavior.

Mock modules using a mock file

1. Create a file like `__mocks__/lodash/memoize.js`:

   ```
   module.exports = (a) => a
   ```

2. Add to your test:

   ```
   jest.mock('lodash/memoize')
   ```

> Note: When using `babel-jest`, calls to `jest.mock` will automatically be hoisted to the top of the code block. Use `jest.doMock` if you want to explicitly avoid this behavior.

Manual mocks docs

Mock object methods

```
const spy = jest.spyOn(console, 'log').mockImplementation(() => {})
expect(console.log.mock.calls).toEqual([['dope'], ['nope']])
spy.mockRestore()
```

```
const spy = jest.spyOn(ajax, 'request').mockImplementation(() => Promise.resolve({ success: true
expect(spy).toHaveBeenCalled()
spy.mockRestore()
```

Mock getters and setters (Jest 22.1.0+)

```
const location = {}
const getTitle = jest.spyOn(location, 'title', 'get').mockImplementation(() => 'pizza')
const setTitle = jest.spyOn(location, 'title', 'set').mockImplementation(() => {})
```

Mock getters and setters

```
const getTitle = jest.fn(() => 'pizza')
const setTitle = jest.fn()
const location = {}
Object.defineProperty(location, 'title', {
  get: getTitle,
  set: setTitle,
})
```

Clearing and restoring mocks

For one mock:

```
fn.mockClear() // Clears mock usage date (fn.mock.calls, fn.mock.instances)
fn.mockReset() // Clears and removes any mocked return values or implementations
fn.mockRestore() // Resets and restores the initial implementation
```

> Note: `mockRestore` works only with mocks created by `jest.spyOn`.

For all mocks:

```
jest.clearAllMocks()
jest.resetAllMocks()
jest.restoreAllMocks()
```

Accessing the original module when using mocks

```
jest.mock('fs')
const fs = require('fs') // Mocked module
const fs = require.requireActual('fs') // Original module
```

Timer mocks

Write synchronous test for code that uses native timer functions (`setTimeout`, `setInterval`, `clearTimeout`, `clearInterval`).

```
// Enable fake timers
jest.useFakeTimers()

test('kill the time', () => {
  const callback = jest.fn()

  // Run some code that uses setTimeout or setInterval
  const actual = someFunctionThatUseTimers(callback)

  // Fast-forward until all timers have been executed
  jest.runAllTimers()

  // Check the results synchronously
  expect(callback).toHaveBeenCalledTimes(1)
})
```

Or adjust timers by time with advanceTimersByTime():

```
// Enable fake timers
jest.useFakeTimers()

test('kill the time', () => {
  const callback = jest.fn()

  // Run some code that uses setTimeout or setInterval
  const actual = someFunctionThatUseTimers(callback)

  // Fast-forward for 250 ms
  jest.advanceTimersByTime(250)

  // Check the results synchronously
  expect(callback).toHaveBeenCalledTimes(1)
})
```

Use jest.runOnlyPendingTimers() for special cases.

Note: you should call `jest.useFakeTimers()` in your test case to use other fake timer methods.

Data-driven tests (Jest 23+)

Run the same test with different data:

```
test.each([
  [1, 1, 2],
  [1, 2, 3],
  [2, 1, 3],
])('.add(%s, %s)', (a, b, expected) => {
  expect(a + b).toBe(expected)
})
```

Or the same using template literals:

```
test.each`
  a    | b    | expected
  ${1} | ${1} | ${2}
  ${1} | ${2} | ${3}
  ${2} | ${1} | ${3}
`('returns $expected when $a is added $b', ({ a, b, expected }) => {
  expect(a + b).toBe(expected)
})
```

Or on `describe` level:

```
describe.each([['mobile'], ['tablet'], ['desktop']])('checkout flow on %s', (viewport) => {
  test('displays success page', () => {
    //
  })
})
```

describe.each() docs, test.each() docs,

Skipping tests

Don't run these tests:

```
describe.skip('makePoniesPink'...
tests.skip('make each pony pink'...
```

Run only these tests:

```
describe.only('makePoniesPink'...
tests.only('make each pony pink'...
```

Testing modules with side effects

Node.js and Jest will cache modules you `require`. To test modules with side effects you'll need to reset the module registry between tests:

```
const modulePath = '../module-to-test'

afterEach(() => {
  jest.resetModules()
})

test('first test', () => {
  // Prepare conditions for the first test
  const result = require(modulePath)
  expect(result).toMatchSnapshot()
})

test('second text', () => {
  // Prepare conditions for the second test
  const fn = () => require(modulePath)
  expect(fn).toThrow()
})
```

Usage with Babel and TypeScript

Add babel-jest or ts-jest. Check their docs for installation instructions.

Resources

- Jest site
- Modern React testing, part 1: best practices by Artem Sapegin
- Modern React testing, part 2: Jest and Enzyme by Artem Sapegin
- Modern React testing, part 3: Jest and React Testing Library by Artem Sapegin
- React Testing Examples
- Testing React Applications by Max Stoiber
- Effective Snapshot Testing by Kent C. Dodds
- Migrating to Jest by Kent C. Dodds
- Migrating AVA to Jest by Jason Brown
- How to Test React and MobX with Jest by Will Stern
- Testing React Intl components with Jest and Enzyme by Artem Sapegin
- Testing with Jest: 15 Awesome Tips and Tricks by Stian Didriksen
- Taking Advantage of Jest Matchers by Ben McCormick: Part 1, Part 2

You may also like

- Opinionated list of React components

Contributing

Improvements are welcome! Open an issue or send a pull request.

Sponsoring

This software has been developed with lots of coffee, buy me one more cup to keep it going.

Buy me a coffee

Author and license

Artem Sapegin, a frontend engineer at Omio and the creator of React Styleguidist. I also write about frontend at my blog.

CC0 1.0 Universal license, see the included License.md file.