Getting StartedInstall Jest using your favorite package manager:npmYarnpnpmnpm install --save-dev jestyarn add --dev jestpnpm add --save-dev jestLet's get started by writing a test for a hypothetical function that adds two numbers. First, create a sum.js file:function sum(a, b) { return a + b;}module.exports = sum;Then, create a file named sum.test.js. This will contain our actual test:const sum = require('./sum');test('adds 1 + 2 to equal 3', () => { expect(sum(1, 2)).toBe(3);});Add the following section to your package.json:{ "scripts": {   "test": "jest" }}Finally, run yarn test or npm test and Jest will print this message:PASS ./sum.test.js' adds 1 + 2 to equal 3 (5ms)You just successfully wrote your first test using Jest!This test used expect and toBe to test that two values were exactly identical. To learn about the other things that Jest can test, see Using Matchers.Running from command line You can run Jest directly from the CLI (if it's globally available in your PATH, e.g. by yarn global add jest or npm install jest --global) with a variety of useful options.Here's how to run Jest on files matching my-test, using config.json as a configuration file and display a native OS notification after the run:jest my-test --notify --config=config.jsonIf you'd like to learn more about running jest through the command line, take a look at the Jest CLI Options page.Additional Configuration Generate a basic configuration file Based on your project, Jest will ask you a few questions and will create a basic configuration file with a short description for each option:jest --initUsing Babel To use Babel, install required dependencies:npmYarnpnpmnpm install --save-dev babel-jest @babel/core @babel/preset-envyarn add --dev babel-jest @babel/core @babel/preset-envpnpm add --save-dev babel-jest @babel/core @babel/preset-envConfigure Babel to target your current version of Node by creating a babel.config.js file in the root of your project:babel.config.jsmodule.exports = { presets: [['@babel/preset-env', {targets: {node: 'current'}}]],};The ideal configuration for Babel will depend on your project. See Babel's docs for more details.Making your Babel config jest-awareJest will set process.env.NODE_ENV to 'test' if it's not set to something else. You can use that in your configuration to conditionally setup only the compilation needed for Jest, e.g.babel.config.jsmodule.exports = api => { const isTest = api.env('test'); // You can use isTest to determine what presets and plugins to use. return {   // ... };};notebabel-jest is automatically installed when installing Jest and will automatically transform files if a babel configuration exists in your project. To avoid this behavior, you can explicitly reset the transform configuration option:jest.config.jsmodule.exports = { transform: {},};Using webpack Jest can be used in projects that use webpack to manage assets, styles, and compilation. webpack does offer some unique challenges over other tools. Refer to the webpack guide to get started.Using Vite Jest can be used in projects that use vite to serve source code over native ESM to provide some frontend tooling, vite is an opinionated tool and does offer some out-of-the box workflows. Jest is not fully supported by vite due to how the plugin system from vite works, but there are some working examples for first-class jest integration using vite-jest, since this is not fully supported, you might as well read the limitation of the vite-jest. Refer to the vite guide to get started.Using Parcel Jest can be used in projects that use parcel-bundler to manage assets, styles, and compilation similar to webpack. Parcel requires zero configuration. Refer to the official docs to get started.Using TypeScript Via babel Jest supports TypeScript, via Babel. First, make sure you followed the instructions on using Babel above. Next, install the @babel/preset-typescript:npmYarnpnpmnpm install --save-dev

@babel/preset-typescriptyarn add --dev @babel/preset-typescriptpnpm add --save-dev @babel/preset-typescriptThen add @babel/preset-typescript to the list of presets in your babel.config.js.babel.config.jsmodule.exports = { presets: [['@babel/preset-env', {targets: {node: 'current'}}],   '@babel/preset-typescript',],};However, there are some caveats to using TypeScript with Babel. Because TypeScript support in Babel is purely transpilation, Jest will not type-check your tests as they are run. If you want that, you can use ts-jest instead, or just run the TypeScript compiler tsc separately (or as part of your build process).Via ts-jest ts-jest is a TypeScript preprocessor with source map support for Jest that lets you use Jest to test projects written in TypeScript.npmYarnpnpmnpm install --save-dev ts-jestyarn add --dev ts-jestpnpm add --save-dev ts-jestIn order for Jest to transpile TypeScript with ts-jest, you may also need to create a configuration file.Type definitions There are two ways to have Jest global APIs typed for test files written in TypeScript.You can use type definitions which ships with Jest and will update each time you update Jest. Install the @jest/globals package:npmYarnpnpmnpm install --save-dev @jest/globalsyarn add --dev @jest/globalspnpm add --save-dev @jest/globalsAnd import the APIs from it:sum.test.tsimport {describe, expect, test} from '@jest/globals';import {sum} from './sum';describe('sum module', () => { test('adds 1 + 2 to equal 3', () => {   expect(sum(1, 2)).toBe(3); });});tipSee the additional usage documentation of describe.each/test.each and mock functions.Or you may choose to install the @types/jest package. It provides types for Jest globals without a need to import them.npmYarnpnpmnpm install --save-dev @types/jestyarn add --dev @types/jestpnpm add --save-dev @types/jestinfo@types/jest is a third party library maintained at DefinitelyTyped, hence the latest Jest features or versions may not be covered yet. Try to match versions of Jest and @types/jest as closely as possible. For example, if you are using Jest 27.4.0 then installing 27.4.x of @types/jest is ideal.

Using MatchersJest uses "matchers" to let you test values in different ways. This document will introduce some commonly used matchers. For the full list, see the expect API doc.Common Matchers The simplest way to test a value is with exact equality.test('two plus two is four', () => { expect(2 + 2).toBe(4);});In this code, expect(2 + 2) returns an "expectation" object. You typically won't do much with these expectation objects except call matchers on them. In this code, .toBe(4) is the matcher. When Jest runs, it tracks all the failing matchers so that it can print out nice error messages for you.toBe uses Object.is to test exact equality. If you want to check the value of an object, use toEqual:test('object assignment', () => { const data = {one: 1}; data['two'] = 2; expect(data).toEqual({one: 1, two: 2});});toEqual recursively checks every field of an object or array.tiptoEqual ignores object keys with undefined properties, undefined array items, array sparseness, or object type mismatch. To take these into account use toStrictEqual instead.You can also test for the opposite of a matcher using not:test('adding positive numbers is not zero', () => { for (let a = 1; a < 10; a++) {   for (let b = 1; b < 10; b++) {   expect(a + b).not.toBe(0);   } }});Truthiness In tests, you sometimes need to distinguish between undefined, null, and false, but you sometimes do not want to treat these differently. Jest contains helpers that let you be explicit about what you want.toBeNull matches only nulltoBeUndefined matches only undefinedtoBeDefined is the opposite of toBeUndefinedtoBeTruthy matches anything that an if statement treats as truetoBeFalsy matches anything that an if statement treats as falseFor example:test('null', () => { const n = null; expect(n).toBeNull(); expect(n).toBeDefined(); expect(n).not.toBeUndefined(); expect(n).not.toBeTruthy(); expect(n).toBeFalsy();});test('zero', () => { const z = 0; expect(z).not.toBeNull(); expect(z).toBeDefined(); expect(z).not.toBeUndefined(); expect(z).not.toBeTruthy(); expect(z).toBeFalsy();});You should use the matcher that most precisely corresponds to what you want your code to be doing.Numbers Most ways of comparing numbers have matcher equivalents.test('two plus two', () => { const value = 2 + 2; expect(value).toBeGreaterThan(3); expect(value).toBeGreaterThanOrEqual(3.5); expect(value).toBeLessThan(5); expect(value).toBeLessThanOrEqual(4.5); // toBe and toEqual are equivalent for numbers expect(value).toBe(4); expect(value).toEqual(4);});For floating point equality, use toBeCloseTo instead of toEqual, because you don't want a test to depend on a tiny rounding error.test('adding floating point numbers', () => { const value = 0.1 + 0.2; //expect(value).toBe(0.3); This won't work because of rounding error expect(value).toBeCloseTo(0.3); // This works.});Strings You can check strings against regular expressions with toMatch:test('there is no I in team', () => { expect('team').not.toMatch(/I/);});test('but there is a "stop" in Christoph', () => { expect('Christoph').toMatch(/stop/);});Arrays and iterables You can check if an array or iterable contains a particular item using toContain:const shoppingList = ['diapers', 'kleenex', 'trash bags', 'paper towels', 'milk',];test('the shopping list has milk on it', () => { expect(shoppingList).toContain('milk'); expect(new Set(shoppingList)).toContain('milk');});Exceptions If you want to test whether a particular function throws an error when it's called, use toThrow.function compileAndroidCode() { throw new Error('you are using the wrong JDK!');}test('compiling android goes as expected', () => { expect(() => compileAndroidCode()).toThrow(); expect(() => compileAndroidCode()).toThrow(Error); // You can also use a string that must be contained in the error message or a regexp expect(() => compileAndroidCode()).toThrow('you are using the wrong JDK'); expect(() => compileAndroidCode()).toThrow(/JDK/); // Or you can match an exact error message using a regexp like below expect(() => compileAndroidCode()).toThrow(/^you are using the wrong JDK$/); // Test fails expect(() => compileAndroidCode()).toThrow(/^you are using the wrong JDK!$/); // Test pass});tipThe function that throws an exception needs to be invoked within a wrapping function otherwise the toThrow assertion will fail.And More This is just a taste. For a complete list of matchers, check out the reference docs.Once you've learned about the matchers that are available, a good next step is to check out how Jest lets you test asynchronous code.

Testing Asynchronous CodeIt's common in JavaScript for code to run asynchronously. When you have code that runs asynchronously, Jest needs to know when the code it is testing has completed, before it can move on to another test. Jest has several ways to handle this.Promises Return a promise from your test, and Jest will wait for that promise to resolve. If the promise is rejected, the test will fail.For example, let's say that fetchData returns a promise that is supposed to resolve to the string 'peanut butter'. We could test it with:test('the data is peanut butter', () => { return fetchData().then(data => {   expect(data).toBe('peanut butter'); });});Async/Await Alternatively, you can use async and await in your tests. To write an async test, use the async keyword in front of the function passed to test. For example, the same fetchData scenario can be tested with:test('the data is peanut butter', async () => { const data = await fetchData(); expect(data).toBe('peanut butter');});test('the fetch fails with an error', async () => { expect.assertions(1); try {   await fetchData(); } catch (e) {   expect(e).toMatch('error'); }});You can combine async and await with .resolves or .rejects.test('the data is peanut butter', async () => { await expect(fetchData()).resolves.toBe('peanut butter');});test('the fetch fails with an error', async () => { await expect(fetchData()).rejects.toMatch('error');});In these cases, async and await are effectively syntactic sugar for the same logic as the promises example uses.cautionBe sure to return (or await) the promise - if you omit the return/await statement, your test will complete before the promise returned from fetchData resolves or rejects.If you expect a promise to be rejected, use the .catch method. Make sure to add expect.assertions to verify that a certain number of assertions are called. Otherwise, a fulfilled promise would not fail the test.test('the fetch fails with an error', () => { expect.assertions(1); return fetchData().catch(e => expect(e).toMatch('error'));});Callbacks If you don't use promises, you can use callbacks. For example, let's say that fetchData, instead of returning a promise, expects a callback, i.e. fetches some data and calls callback(null, data) when it is complete. You want to test that this returned data is the string 'peanut butter'.By default, Jest tests complete once they reach the end of their execution. That means this test will not work as intended:// Don't do this!test('the data is peanut butter', () => { function callback(error, data) {   if (error) {   throw error;   }   expect(data).toBe('peanut butter'); } fetchData(callback);});The problem is that the test will complete as soon as fetchData completes, before ever calling the callback.There is an alternate form of test that fixes this. Instead of putting the test in a function with an empty argument, use a single argument called done. Jest will wait until the done callback is called before finishing the test.test('the data is peanut butter', done => { function callback(error, data) {   if (error) {   done(error);   return;   }   try {   expect(data).toBe('peanut butter');   done();   } catch (error) {   done(error);   } } fetchData(callback);});If done() is never called, the test will fail (with timeout error), which is what you want to happen.If the expect statement fails, it throws an error and done() is not called. If we want to see in the test log why it failed, we have to wrap expect in a try block and pass the error in the catch block to done. Otherwise, we end up with an opaque timeout error that doesn't show what value was received by expect(data).cautionJest will throw an error, if the same test function is passed a done() callback and returns a promise. This is done as a precaution to avoid memory leaks in your tests..resolves / .rejects You can also use the .resolves matcher in your expect statement, and Jest will wait for that promise to resolve. If the promise is rejected, the test will automatically fail.test('the data is peanut butter', () => { return expect(fetchData()).resolves.toBe('peanut butter');});Be sure to return the assertion—if you omit this return statement, your test will complete before the promise returned from fetchData is resolved and then() has a chance to execute the callback.If you expect a promise to be rejected, use the .rejects matcher. It works analogically to the .resolves matcher. If the promise is fulfilled, the test will automatically fail.test('the fetch fails with an error', () => { return expect(fetchData()).rejects.toMatch('error');});None of these forms is particularly superior to the others, and you can mix and match them across a codebase or even in a single file. It just depends on which style you feel makes your tests simpler.

Setup and TeardownOften while writing tests you have some setup work that needs to happen before tests run, and you have some finishing work that needs to happen after tests run. Jest provides helper functions to handle this.Repeating Setup If you have some work you need to do repeatedly for many tests, you can use beforeEach and afterEach hooks.For example, let's say that several tests interact with a database of cities. You have a method initializeCityDatabase() that must be called before each of these tests, and a method clearCityDatabase() that must be called after each of these tests. You can do this with:beforeEach(() => { initializeCityDatabase();});afterEach(() => { clearCityDatabase();});test('city database has Vienna', () => { expect(isCity('Vienna')).toBeTruthy();});test('city database has San Juan', () => { expect(isCity('San Juan')).toBeTruthy();});beforeEach and afterEach can handle asynchronous code in the same ways that tests can handle asynchronous code - they can either take a done parameter or return a promise. For example, if initializeCityDatabase() returned a promise that resolved when the database was initialized, we would want to return that promise:beforeEach(() => { return initializeCityDatabase();});One-Time Setup In some cases, you only need to do setup once, at the beginning of a file. This can be especially bothersome when the setup is asynchronous, so you can't do it inline. Jest provides beforeAll and afterAll hooks to handle this situation.For example, if both initializeCityDatabase() and clearCityDatabase() returned promises, and the city database could be reused between tests, we could change our test code to:beforeAll(() => { return initializeCityDatabase();});afterAll(() => { return clearCityDatabase();});test('city database has Vienna', () => { expect(isCity('Vienna')).toBeTruthy();});test('city database has San Juan', () => { expect(isCity('San Juan')).toBeTruthy();});Scoping The top level before* and after* hooks apply to every test in a file. The hooks declared inside a describe block apply only to the tests within that describe block.For example, let's say we had not just a city database, but also a food database. We could do different setup for different tests:// Applies to all tests in this filebeforeEach(() => { return initializeCityDatabase();});test('city database has Vienna', () => { expect(isCity('Vienna')).toBeTruthy();});test('city database has San Juan', () => { expect(isCity('San Juan')).toBeTruthy();});describe('matching cities to foods', () => { // Applies only to tests in this describe block beforeEach(() => {   return initializeFoodDatabase(); }); test('Vienna <3 veal', () => {   expect(isValidCityFoodPair('Vienna', 'Wiener Schnitzel')).toBe(true); }); test('San Juan <3 plantains', () => {   expect(isValidCityFoodPair('San Juan', 'Mofongo')).toBe(true); });});Note that the top-level beforeEach is executed before the beforeEach inside the describe block. It may help to illustrate the order of execution of all hooks.beforeAll(() => console.log('1 - beforeAll'));afterAll(() => console.log('1 - afterAll'));beforeEach(() => console.log('1 - beforeEach'));afterEach(() => console.log('1 - afterEach'));test('', () => console.log('1 - test'));describe('Scoped / Nested block', () => { beforeAll(() => console.log('2 - beforeAll')); afterAll(() => console.log('2 - afterAll')); beforeEach(() => console.log('2 - beforeEach')); afterEach(() => console.log('2 - afterEach')); test('', () => console.log('2 - test'));});// 1 - beforeAll// 1 - beforeEach// 1 - test// 1 - afterEach// 2 - beforeAll// 1 - beforeEach// 2 - beforeEach// 2 - test// 2 - afterEach// 1 - afterEach// 2 - afterAll// 1 - afterAllOrder of Execution Jest executes all describe handlers in a test file before it executes any of the actual tests. This is another reason to do setup and teardown inside before* and after* handlers rather than inside the describe blocks. Once the describe blocks are complete, by default Jest runs all the tests serially in the order they were encountered in the collection phase, waiting for each to finish and be tidied up before moving on.Consider the following illustrative test file and output:describe('describe outer', () => { console.log('describe outer-a'); describe('describe inner 1', () => {   console.log('describe inner 1');   test('test 1', () => console.log('test 1')); }); console.log('describe outer-b'); test('test 2', () => console.log('test 2')); describe('describe inner 2', () => {   console.log('describe inner 2');   test('test 3', () => console.log('test 3')); }); console.log('describe outer-c');});// describe outer-a// describe inner 1// describe outer-b// describe inner 2// describe outer-c// test 1// test 2// test 3Just like the describe and test blocks Jest calls the before* and after* hooks in the order of declaration. Note that the after* hooks of the enclosing scope are called first. For example, here is how you can set up and tear down resources which depend on each other:beforeEach(() => console.log('connection setup'));beforeEach(() => console.log('database setup'));afterEach(() => console.log('database teardown'));afterEach(() => console.log('connection teardown'));test('test 1', () => console.log('test 1'));describe('extra', () => { beforeEach(() => console.log('extra database setup')); afterEach(() => console.log('extra database teardown')); test('test 2', () => console.log('test 2'));});// connection setup// database setup// test 1// database teardown// connection teardown// connection setup// database setup// extra database setup// test 2// extra database teardown// database teardown// connection teardownnoteIf you are using jasmine2 test runner, take into account that it calls the after* hooks in the reverse order of declaration. To have identical output, the above example should be altered like this: beforeEach(() => console.log('connection setup'));+ afterEach(() => console.log('connection teardown')); beforeEach(() => console.log('database setup'));+ afterEach(() => console.log('database teardown'));- afterEach(() => console.log('database teardown'));- afterEach(() => console.log('connection teardown')); // ...General Advice If a test is failing, one of the first things to check should be whether the test is failing when it's the only test that runs. To run only one test with Jest, temporarily change that test command to a test.only:test.only('this will be the only test that runs', () => { expect(true).toBe(false);});test('this test will not run', () => { expect('A').toBe('A');});If you have a test that often fails when it's run as part of a larger suite, but doesn't fail when you run it alone, it's a good bet that something from a different test is interfering with this one. You can often fix this by clearing some shared state with beforeEach. If you're not sure whether some shared state is being modified, you can also try a beforeEach that logs data.

Mock FunctionsMock functions allow you to test the links between code by erasing the actual implementation of a function, capturing calls to the function (and the parameters passed in those calls), capturing instances of constructor functions when instantiated with new, and allowing test-time configuration of return values.There are two ways to mock functions: Either by creating a mock function to use in test code, or writing a manual mock to override a module dependency.Using a mock function Let's imagine we're testing an implementation of a function forEach, which invokes a callback for each item in a supplied array.forEach.jsexport function forEach(items, callback) { for (let index = 0; index < items.length; index++) {   callback(items[index]); }}To test this function, we can use a mock function, and inspect the mock's state to ensure the callback is invoked as expected.forEach.test.jsconst forEach = require('./forEach');const mockCallback = jest.fn(x => 42 + x);test('forEach mock function', () => { forEach([0, 1], mockCallback); // The mock function was called twice expect(mockCallback.mock.calls).toHaveLength(2); // The first argument of the first call to the function was 0 expect(mockCallback.mock.calls[0][0]).toBe(0); // The first argument of the second call to the function was 1 expect(mockCallback.mock.calls[1][0]).toBe(1); // The return value of the first call to the function was 42 expect(mockCallback.mock.results[0].value).toBe(42);});.mock property All mock functions have this special .mock property, which is where data about how the function has been called and what the function returned is kept. The .mock property also tracks the value of this for each call, so it is possible to inspect this as well:const myMock1 = jest.fn();const a = new myMock1();console.log(myMock1.mock.instances);// > [<a>]const myMock2 = jest.fn();const b = {};const bound = myMock2.bind(b);bound();console.log(myMock2.mock.contexts);// > [<b>]These mock members are very useful in tests to assert how these functions get called, instantiated, or what they returned:// The function was called exactly onceexpect(someMockFunction.mock.calls).toHaveLength(1);// The first arg of the first call to the function was 'first arg'expect(someMockFunction.mock.calls[0][0]).toBe('first arg');// The second arg of the first call to the function was 'second arg'expect(someMockFunction.mock.calls[0][1]).toBe('second arg');// The return value of the first call to the function was 'return value'expect(someMockFunction.mock.results[0].value).toBe('return value');// The function was called with a certain `this` context: the `element` object.expect(someMockFunction.mock.contexts[0]).toBe(element);// This function was instantiated exactly twiceexpect(someMockFunction.mock.instances.length).toBe(2);// The object returned by the first instantiation of this function// had a `name` property whose value was set to 'test'expect(someMockFunction.mock.instances[0].name).toBe('test');// The first argument of the last call to the function was 'test'expect(someMockFunction.mock.lastCall[0]).toBe('test');Mock Return Values Mock functions can also be used to inject test values into your code during a test:const myMock = jest.fn();console.log(myMock());// > undefinedmyMock.mockReturnValueOnce(10).mockReturnValueOnce('x').mockReturnValue(true);console.log(myMock(), myMock(), myMock(), myMock());// > 10, 'x', true, trueMock functions are also very effective in code that uses a functional continuation-passing style. Code written in this style helps avoid the need for complicated stubs that recreate the behavior of the real component they're standing in for, in favor of injecting values directly into the test right before they're used.const filterTestFn = jest.fn();// Make the mock return `true` for the first call,// and `false` for the second callfilterTestFn.mockReturnValueOnce(true).mockReturnValueOnce(false);const result = [11, 12].filter(num => filterTestFn(num));console.log(result);// > [11]console.log(filterTestFn.mock.calls[0][0]); // 11console.log(filterTestFn.mock.calls[1][0]); // 12Most real-world examples actually involve getting ahold of a mock function on a dependent component and configuring that, but the technique is the same. In these cases, try to avoid the temptation to implement logic inside of any function that's not directly being tested.Mocking Modules Suppose we have a class that fetches users from our API. The class uses axios to call the API then returns the data attribute which contains all the users:users.jsimport axios from 'axios';class Users { static all() {   return axios.get('/users.json').then(resp => resp.data); }}export default Users;Now, in order to test this method without actually hitting the API (and thus creating slow and fragile tests), we can use the jest.mock(...) function to automatically mock the axios module.Once we mock the module we can provide a mockResolvedValue for .get that returns the data we want our test to assert against. In effect, we are saying that we want axios.get('/users.json') to return a fake response.users.test.jsimport axios from 'axios';import Users from './users';jest.mock('axios');test('should fetch users', () => { const users = [{name: 'Bob'}]; const resp = {data: users}; axios.get.mockResolvedValue(resp); // or you could use the following depending on your use case: // axios.get.mockImplementation(() => Promise.resolve(resp)) return Users.all().then(data => expect(data).toEqual(users));});Mocking Partials Subsets of a module can be mocked and the rest of the module can keep their actual implementation:foo-bar-baz.jsexport const foo = 'foo';export const bar = () => 'bar';export default () => 'baz';//test.jsimport defaultExport, {bar, foo} from '../foo-bar-baz';jest.mock('../foo-bar-baz', () => { const originalModule = jest.requireActual('../foo-bar-baz'); //Mock the default export and named export 'foo' return {   __esModule: true,   ...originalModule,   default: jest.fn(() => 'mocked baz'),   foo: 'mocked foo', };});test('should do a partial mock', () => { const defaultExportResult = defaultExport(); expect(defaultExportResult).toBe('mocked baz'); expect(defaultExport).toHaveBeenCalled(); expect(foo).toBe('mocked foo'); expect(bar()).toBe('bar');});Mock Implementations Still, there are cases where it's useful to go beyond the ability to specify return values and full-on replace the implementation of a mock function. This can be done with jest.fn or the mockImplementationOnce method on mock functions.const myMockFn = jest.fn(cb => cb(null, true));myMockFn((err, val) => console.log(val));// > trueThe mockImplementation method is useful when you need to define the default implementation of a mock function that is created from another module:foo.jsmodule.exports = function () { // some implementation;};test.jsjest.mock('../foo'); // this happens automatically with automockingconst foo = require('../foo');// foo is a mock functionfoo.mockImplementation(() => 42);foo();// > 42When you need to recreate a complex behavior of a mock function such that multiple function calls produce different results, use the mockImplementationOnce method:const myMockFn = jest .fn() .mockImplementationOnce(cb => cb(null, true)) .mockImplementationOnce(cb => cb(null, false));myMockFn((err, val) => console.log(val));// > truemyMockFn((err, val) => console.log(val));// > falseWhen the mocked function runs out of implementations defined with mockImplementationOnce, it will execute the default implementation set with jest.fn (if it is defined):const myMockFn = jest .fn(() => 'default') .mockImplementationOnce(() => 'first call') .mockImplementationOnce(() => 'second call');console.log(myMockFn(), myMockFn(), myMockFn(), myMockFn());// > 'first call', 'second call', 'default', 'default'For cases where we have methods that are typically chained (and thus always need to return this), we have a sugary API to simplify this in the form of a .mockReturnThis() function that also sits on all mocks:const myObj = { myMethod:

jest.fn().mockReturnThis(),};// is the same asconst otherObj = { myMethod: jest.fn(function () {   return this; }),};Mock Names You can optionally provide a name for your mock functions, which will be displayed instead of 'jest.fn()' in the test error output. Use .mockName() if you want to be able to quickly identify the mock function reporting an error in your test output.const myMockFn = jest .fn() .mockReturnValue('default') .mockImplementation(scalar => 42 + scalar) .mockName('add42');Custom Matchers Finally, in order to make it less demanding to assert how mock functions have been called, we've added some custom matcher functions for you:// The mock function was called at least onceexpect(mockFunc).toHaveBeenCalled();// The mock function was called at least once with the specified argsexpect(mockFunc).toHaveBeenCalledWith(arg1, arg2);// The last call to the mock function was called with the specified argsexpect(mockFunc).toHaveBeenLastCalledWith(arg1, arg2);// All calls and the name of the mock is written as a snapshotexpect(mockFunc).toMatchSnapshot();These matchers are sugar for common forms of inspecting the .mock property. You can always do this manually yourself if that's more to your taste or if you need to do something more specific:// The mock function was called at least onceexpect(mockFunc.mock.calls.length).toBeGreaterThan(0);// The mock function was called at least once with the specified argsexpect(mockFunc.mock.calls).toContainEqual([arg1, arg2]);// The last call to the mock function was called with the specified argsexpect(mockFunc.mock.calls[mockFunc.mock.calls.length - 1]).toEqual([arg1, arg2,]);// The first arg of the last call to the mock function was `42`// (note that there is no sugar helper for this specific of an assertion)expect(mockFunc.mock.calls[mockFunc.mock.calls.length - 1][0]).toBe(42);// A snapshot will check that a mock was invoked the same number of times,// in the same order, with the same arguments. It will also assert on the name.expect(mockFunc.mock.calls).toEqual([[arg1, arg2]]);expect(mockFunc.getMockName()).toBe('a mock name');For a complete list of matchers, check out the reference docs.

Jest PlatformYou can cherry pick specific features of Jest and use them as standalone packages. Here's a list of the available packages:jest-changed-files Tool for identifying modified files in a git/hg repository. Exports two functions:getChangedFilesForRoots returns a promise that resolves to an object with the changed files and repos.findRepos returns a promise that resolves to a set of repositories contained in the specified path.Example const {getChangedFilesForRoots} = require('jest-changed-files');// print the set of modified files since last commit in the current repogetChangedFilesForRoots(['./'], { lastCommit: true,}).then(result => console.log(result.changedFiles));You can read more about jest-changed-files in the readme file.jest-diff Tool for visualizing changes in data. Exports a function that compares two values of any type and returns a "pretty-printed" string illustrating the difference between the two arguments.Example const {diff} = require('jest-diff');const a = {a: {b: {c: 5}}};const b = {a: {b: {c: 6}}};const result = diff(a, b);// print diffconsole.log(result);jest-docblock Tool for extracting and parsing the comments at the top of a JavaScript file. Exports various functions to manipulate the data inside the comment block.Example const {parseWithComments} = require('jest-docblock');const code = `/** * This is a sample * * @flow */ console.log('Hello World!');`;const parsed = parseWithComments(code);// prints an object with two attributes: comments and pragmas.console.log(parsed);You can read more about jest-docblock in the readme file.jest-get-type Module that identifies the primitive type of any JavaScript value. Exports a function that returns a string with the type of the value passed as argument.Example const {getType} = require('jest-get-type');const array = [1, 2, 3];const nullValue = null;const undefinedValue = undefined;// prints 'array'console.log(getType(array));// prints 'null'console.log(getType(nullValue));// prints 'undefined'console.log(getType(undefinedValue));jest-validate Tool for validating configurations submitted by users. Exports a function that takes two arguments: the user's configuration and an object containing an example configuration and other options. The return value is an object with two attributes:hasDeprecationWarnings, a boolean indicating whether the submitted configuration has deprecation warnings,isValid, a boolean indicating whether the configuration is correct or not.Example const {validate} = require('jest-validate');const configByUser = { transform: '<rootDir>/node_modules/my-custom-transform',};const result = validate(configByUser, { comment: ' Documentation: http://custom-docs.com', exampleConfig: {transform: '<rootDir>/node_modules/babel-jest'},});console.log(result);You can read more about jest-validate in the readme file.jest-worker Module used for parallelization of tasks. Exports a class JestWorker that takes the path of Node.js module and lets you call the module's exported methods as if they were class methods, returning a promise that resolves when the specified method finishes its execution in a forked process.Example heavy-task.jsmodule.exports = { myHeavyTask: args => {   // long running CPU intensive task. },};main.jsasync function main() { const worker = new Worker(require.resolve('./heavy-task.js')); // run 2 tasks in parallel with different arguments const results = await Promise.all([worker.myHeavyTask({foo: 'bar'}),   worker.myHeavyTask({bar: 'foo'}),]); console.log(results);}main();You can read more about jest-worker in the readme file.pretty-format Exports a function that converts any JavaScript value into a human-readable string. Supports all built-in JavaScript types out of the box and allows extension for application-specific types via user-defined plugins.Example const {format: prettyFormat} = require('pretty-format');const val = {object: {}};val.circularReference = val;val[Symbol('foo')] = 'foo';val.map = new Map([['prop', 'value']]);val.array = [-0, Infinity, NaN];console.log(prettyFormat(val));You can read more about pretty-format in the readme file.

Jest CommunityThe community around Jest is working hard to make the testing experience even greater.jest-community is a new GitHub organization for high quality Jest additions curated by Jest maintainers and collaborators. It already features some of our favorite projects, to name a few:vscode-jestjest-extendedeslint-plugin-jestawesome-jestCommunity projects under one organization are a great way for Jest to experiment with new ideas/techniques and approaches. Encourage contributions from the community and publish contributions independently at a faster pace.Awesome Jest The jest-community org maintains an awesome-jest list of great projects and resources related to Jest.If you have something awesome to share, feel free to reach out to us! We'd love to share your project on the awesome-jest list (send a PR here) or if you would like to transfer your project to the jest-community org reach out to one of the owners of the org.

More ResourcesBy now you should have a good idea of how Jest can help you test your applications. If you're interested in learning more, here's some related stuff you might want to check out.Browse the docs Learn about Snapshot Testing, Mock Functions, and more in our in-depth guides.Migrate your existing tests to Jest by following our migration guide.Learn how to configure Jest.Look at the full API Reference.Troubleshoot problems with Jest.Learn by example You will find a number of example test cases in the examples folder on GitHub. You can also learn from the excellent tests used by the React, Relay, and React Native projects.Join the community Ask questions and find answers from other Jest users like you. Reactiflux is a Discord chat where a lot of Jest discussion happens. Check out the #testing channel.Follow the Jest Twitter account and blog to find out what's happening in the world of Jest. Snapshot TestingSnapshot tests are a very useful tool whenever you want to make sure your UI does not change unexpectedly.A typical snapshot test case renders a UI component, takes a snapshot, then compares it to a reference snapshot file stored alongside the test. The test will fail if the two snapshots do not match: either the change is unexpected, or the reference snapshot needs to be updated to the new version of the UI component.Snapshot Testing with Jest A similar approach can be taken when it comes to testing your React components. Instead of rendering the graphical UI, which would require building the entire app, you can use a test renderer to quickly generate a serializable value for your React tree. Consider this example test for a Link component:import renderer from 'react-test-renderer';import Link from '../Link';it('renders correctly', () => { const tree = renderer   .create(<Link page="http://www.facebook.com">Facebook</Link>)   .toJSON(); expect(tree).toMatchSnapshot();});The first time this test is run, Jest creates a snapshot file that looks like this:exports[`renders correctly 1`] = `<a  className="normal"  href="http://www.facebook.com"  onMouseEnter={[Function]}  onMouseLeave={[Function]}>  Facebook</a>`;The snapshot artifact should be committed alongside code changes, and reviewed as part of your code review process. Jest uses pretty-format to make snapshots human-readable during code review. On subsequent test runs, Jest will compare the rendered output with the previous snapshot. If they match, the test will pass. If they don't match, either the test runner found a bug in your code (in the <Link> component in this case) that should be fixed, or the implementation has changed and the snapshot needs to be updated.noteThe snapshot is directly scoped to the data you render – in our example the <Link> component with page prop passed to it. This implies that even if any other file has missing props (say, App.js) in the <Link> component, it will still pass the test as the test doesn't know the usage of <Link> component and it's scoped only to the Link.js. Also, rendering the same component with different props in other snapshot tests will not affect the first one, as the tests don't know about each other.infoMore information on how snapshot testing works and why we built it can be found on the release blog post. We recommend reading this blog post to get a good sense of when you should use snapshot testing. We also recommend watching this egghead video on Snapshot Testing with Jest.Updating Snapshots It's straightforward to spot when a snapshot test fails after a bug has been introduced. When that happens, go ahead and fix the issue and make sure your snapshot tests are passing again. Now, let's talk about the case when a snapshot test is failing due to an intentional implementation change.One such situation can arise if we intentionally change the address the Link component in our example is pointing to.// Updated test case with a Link to a different addressit('renders correctly', () => { const tree = renderer   .create(<Link page="http://www.instagram.com">Instagram</Link>)   .toJSON(); expect(tree).toMatchSnapshot();});In that case, Jest will print this output:Since we just updated our component to point to a different address, it's reasonable to expect changes in the snapshot for this component. Our snapshot test case is failing because the snapshot for our updated component no longer matches the snapshot artifact for this test case.To resolve this, we will need to update our snapshot artifacts. You can run Jest with a flag that will tell it to re-generate snapshots:jest --updateSnapshotGo ahead and accept the changes by running the above command. You may also use the equivalent single-character -u flag to re-generate snapshots if you prefer. This will re-generate snapshot artifacts for all failing snapshot tests. If we had any additional failing snapshot tests due to an unintentional bug, we would need to fix the bug before re-generating snapshots to avoid recording snapshots of the buggy behavior.If you'd like to limit which snapshot test cases get re-generated, you can pass an additional --testNamePattern flag to re-record snapshots only for those tests that match the pattern.You can try out this functionality by cloning the snapshot example, modifying the Link component, and running Jest.Interactive Snapshot Mode Failed snapshots can also be updated interactively in watch mode:Once you enter Interactive Snapshot Mode, Jest will step you through the failed snapshots one test at a time and give you the opportunity to review the failed output.From here you can choose to update that snapshot or skip to the next:Once you're finished, Jest will give you a summary before returning back to watch mode:Inline Snapshots Inline snapshots behave identically to external snapshots (.snap files), except the snapshot values are written automatically back into the source code. This means you can get the benefits of automatically generated snapshots without having to switch to an external file to make sure the correct value was written.Example:First, you write a test, calling .toMatchInlineSnapshot() with no arguments:it('renders correctly', () => { const tree = renderer   .create(<Link page="https://example.com">Example Site</Link>)   .toJSON(); expect(tree).toMatchInlineSnapshot();});The next time you run Jest, tree will be evaluated, and a snapshot will be written as an argument to toMatchInlineSnapshot:it('renders correctly', () => { const tree = renderer   .create(<Link page="https://example.com">Example Site</Link>)   .toJSON(); expect(tree).toMatchInlineSnapshot(`<a  className="normal"  href="https://example.com"  onMouseEnter={[Function]}  onMouseLeave={[Function]}>  Example Site</a>`);});That's all there is to it! You can even update the snapshots with --updateSnapshot or using the u key in --watch mode.By default, Jest handles the writing of snapshots into your source code. However, if you're using prettier in your project, Jest will detect this and delegate the work to prettier instead (including honoring your configuration).Property Matchers Often there are fields in the object you want to snapshot which are generated (like IDs and Dates). If you try to snapshot these objects, they will force the snapshot to fail on every run:it('will fail every time', () => { const user = {   createdAt: new Date(),   id: Math.floor(Math.random() * 20),   name: 'LeBron James', }; expect(user).toMatchSnapshot();});// Snapshotexports[`will fail every time 1`] = `{ "createdAt": 2018-05-19T23:36:09.816Z, "id": 3, "name": "LeBron James",}

`;For these cases, Jest allows providing an asymmetric matcher for any property. These matchers are checked before the snapshot is written or tested, and then saved to the snapshot file instead of the received value:it('will check the matchers and pass', () => { const user = {   createdAt: new Date(),   id: Math.floor(Math.random() * 20),   name: 'LeBron James', }; expect(user).toMatchSnapshot({   createdAt: expect.any(Date),   id: expect.any(Number), });});// Snapshotexports[`will check the matchers and pass 1`] = `{  "createdAt": Any<Date>,  "id": Any<Number>,  "name": "LeBron James",}`;Any given value that is not a matcher will be checked exactly and saved to the snapshot:it('will check the values and pass', () => { const user = {   createdAt: new Date(),   name: 'Bond... James Bond', }; expect(user).toMatchSnapshot({   createdAt: expect.any(Date),   name: 'Bond... James Bond', });});// Snapshotexports[`will check the values and pass 1`] = `{ "createdAt": Any<Date>, "name": 'Bond... James Bond',}
`;tipIf the case concerns a string not an object then you need to replace random part of that string on your own before testing the snapshot.
You can use for that e.g. replace() and regular expressions.const randomNumber = Math.round(Math.random() * 100);const stringWithRandomData = `<div id="${randomNumber}">Lorem ipsum</div>`;const stringWithConstantData = stringWithRandomData.replace(/id="\d+"/, 123);expect(stringWithConstantData).toMatchSnapshot();Another way is to mock the library responsible for generating the random part of the code you're snapshotting.Best Practices Snapshots are a fantastic tool for identifying unexpected interface changes within your application – whether that interface is an API response, UI, logs, or error messages. As with any testing strategy, there are some best-practices you should be aware of, and guidelines you should follow, in order to use them effectively.1. Treat snapshots as code Commit snapshots and review them as part of your regular code review process. This means treating snapshots as you would any other type of test or code in your project.Ensure that your snapshots are readable by keeping them focused, short, and by using tools that enforce these stylistic conventions.As mentioned previously, Jest uses pretty-format to make snapshots human-readable, but you may find it useful to introduce additional tools, like eslint-plugin-jest with its no-large-snapshots option, or snapshot-diff with its component snapshot comparison feature, to promote committing short, focused assertions.The goal is to make it easy to review snapshots in pull requests, and fight against the habit of regenerating snapshots when test suites fail instead of examining the root causes of their failure.2. Tests should be deterministic Your tests should be deterministic. Running the same tests multiple times on a component that has not changed should produce the same results every time. You're responsible for making sure your generated snapshots do not include platform specific or other non-deterministic data.For example, if you have a Clock component that uses Date.now(), the snapshot generated from this component will be different every time the test case is run. In this case we can mock the Date.now() method to return a consistent value every time the test is run:Date.now = jest.fn(() => 1482363367071);Now, every time the snapshot test case runs, Date.now() will return 1482363367071 consistently. This will result in the same snapshot being generated for this component regardless of when the test is run.3. Use descriptive snapshot names Always strive to use descriptive test and/or snapshot names for snapshots. The best names describe the expected snapshot content. This makes it easier for reviewers to

verify the snapshots during review, and for anyone to know whether or not an outdated snapshot is the correct behavior before updating.For example, compare:exports[`<UserName /> should handle some test case`] = `null`;exports[`<UserName /> should handle some other test case`] = `<div>  Alan Turing</div>`;To:exports[`<UserName /> should render null`] = `null`;exports[`<UserName /> should render Alan Turing`] = `<div>  Alan Turing</ div>`;Since the latter describes exactly what's expected in the output, it's more clear to see when it's wrong:exports[`<UserName /> should render null`] = `<div>  Alan Turing</ div>`;exports[`<UserName /> should render Alan Turing`] = `null`;Frequently Asked Questions Are snapshots written automatically on Continuous Integration (CI) systems? No, as of Jest 20, snapshots in Jest are not automatically written when Jest is run in a CI system without explicitly passing --updateSnapshot. It is expected that all snapshots are part of the code that is run on CI and since new snapshots automatically pass, they should not pass a test run on a CI system. It is recommended to always commit all snapshots and to keep them in version control.Should snapshot files be committed? Yes, all snapshot files should be committed alongside the modules they are covering and their tests. They should be considered part of a test, similar to the value of any other assertion in Jest. In fact, snapshots represent the state of the source modules at any given point in time. In this way, when the source modules are modified, Jest can tell what changed from the previous version. It can also provide a lot of additional context during code review in which reviewers can study your changes better.Does snapshot testing only work with React components? React and React Native components are a good use case for snapshot testing. However, snapshots can capture any serializable value and should be used anytime the goal is testing whether the output is correct. The Jest repository contains many examples of testing the output of Jest itself, the output of Jest's assertion library as well as log messages from various parts of the Jest codebase. See an example of snapshotting CLI output in the Jest repo.What's the difference between snapshot testing and visual regression testing? Snapshot testing and visual regression testing are two distinct ways of testing UIs, and they serve different purposes. Visual regression testing tools take screenshots of web pages and compare the resulting images pixel by pixel. With Snapshot testing values are serialized, stored within text files, and compared using a diff algorithm. There are different trade-offs to consider and we listed the reasons why snapshot testing was built in the Jest blog.Does snapshot testing replace unit testing? Snapshot testing is only one of more than 20 assertions that ship with Jest. The aim of snapshot testing is not to replace existing unit tests, but to provide additional value and make testing painless. In some scenarios, snapshot testing can potentially remove the need for unit testing for a particular set of functionalities (e.g. React components), but they can work together as well.What is the performance of snapshot testing regarding speed and size of the generated files? Jest has been rewritten with performance in mind, and snapshot testing is not an exception. Since snapshots are stored within text files, this way of testing is fast and reliable. Jest generates a new file for each test file that invokes the toMatchSnapshot matcher. The size of the snapshots is pretty small: For reference, the size of all snapshot files in the Jest codebase itself is less than 300 KB.How do I resolve conflicts within snapshot files? Snapshot files must always represent the current state of the modules they are covering. Therefore, if you are merging two branches and

encounter a conflict in the snapshot files, you can either resolve the conflict manually or update the snapshot file by running Jest and inspecting the result.Is it possible to apply test-driven development principles with snapshot testing? Although it is possible to write snapshot files manually, that is usually not approachable. Snapshots help to figure out whether the output of the modules covered by tests is changed, rather than giving guidance to design the code in the first place.Does code coverage work with snapshot testing? Yes, as well as with any other test.
Snapshot TestingSnapshot tests are a very useful tool whenever you want to make sure your UI does not change unexpectedly.A typical snapshot test case renders a UI component, takes a snapshot, then compares it to a reference snapshot file stored alongside the test. The test will fail if the two snapshots do not match: either the change is unexpected, or the reference snapshot needs to be updated to the new version of the UI component.Snapshot Testing with Jest A similar approach can be taken when it comes to testing your React components. Instead of rendering the graphical UI, which would require building the entire app, you can use a test renderer to quickly generate a serializable value for your React tree. Consider this example test for a Link component:import renderer from 'react-test-renderer';import Link from '../Link';it('renders correctly', () => { const tree = renderer   .create(<Link page="http://www.facebook.com">Facebook</Link>)   .toJSON(); expect(tree).toMatchSnapshot();});The first time this test is run, Jest creates a snapshot file that looks like this:exports[`renders correctly 1`] = `<a  className="normal"  href="http://www.facebook.com"  onMouseEnter={[Function]}  onMouseLeave={[Function]}>  Facebook</a>`;The snapshot artifact should be committed alongside code changes, and reviewed as part of your code review process. Jest uses pretty-format to make snapshots human-readable during code review. On subsequent test runs, Jest will compare the rendered output with the previous snapshot. If they match, the test will pass. If they don't match, either the test runner found a bug in your code (in the <Link> component in this case) that should be fixed, or the implementation has changed and the snapshot needs to be updated.noteThe snapshot is directly scoped to the data you render – in our example the <Link> component with page prop passed to it. This implies that even if any other file has missing props (say, App.js) in the <Link> component, it will still pass the test as the test doesn't know the usage of <Link> component and it's scoped only to the Link.js. Also, rendering the same component with different props in other snapshot tests will not affect the first one, as the tests don't know about each other.infoMore information on how snapshot testing works and why we built it can be found on the release blog post. We recommend reading this blog post to get a good sense of when you should use snapshot testing. We also recommend watching this egghead video on Snapshot Testing with Jest.Updating Snapshots It's straightforward to spot when a snapshot test fails after a bug has been introduced. When that happens, go ahead and fix the issue and make sure your snapshot tests are passing again. Now, let's talk about the case when a snapshot test is failing due to an intentional implementation change.One such situation can arise if we intentionally change the address the Link component in our example is pointing to.// Updated test case with a Link to a different addressit('renders correctly', () => { const tree = renderer   .create(<Link page="http://www.instagram.com">Instagram</Link>)   .toJSON();

expect(tree).toMatchSnapshot();});In that case, Jest will print this output:Since we just updated our component to point to a different address, it's reasonable to expect changes in the snapshot for this component. Our snapshot test case is failing because the snapshot for our updated component no longer matches the snapshot artifact for this test case.To resolve this, we will need to update our snapshot artifacts. You can run Jest with a flag that will tell it to re-generate snapshots:jest --updateSnapshotGo ahead and accept the changes by running the above command. You may also use the equivalent single-character -u flag to re-generate snapshots if you prefer. This will re-generate snapshot artifacts for all failing snapshot tests. If we had any additional failing snapshot tests due to an unintentional bug, we would need to fix the bug before re-generating snapshots to avoid recording snapshots of the buggy behavior.If you'd like to limit which snapshot test cases get re-generated, you can pass an additional --testNamePattern flag to re-record snapshots only for those tests that match the pattern.You can try out this functionality by cloning the snapshot example, modifying the Link component, and running Jest.Interactive Snapshot Mode Failed snapshots can also be updated interactively in watch mode:Once you enter Interactive Snapshot Mode, Jest will step you through the failed snapshots one test at a time and give you the opportunity to review the failed output.From here you can choose to update that snapshot or skip to the next:Once you're finished, Jest will give you a summary before returning back to watch mode:Inline Snapshots Inline snapshots behave identically to external snapshots (.snap files), except the snapshot values are written automatically back into the source code. This means you can get the benefits of automatically generated snapshots without having to switch to an external file to make sure the correct value was written.Example:First, you write a test, calling .toMatchInlineSnapshot() with no arguments:it('renders correctly', () => { const tree = renderer   .create(<Link page="https://example.com">Example Site</Link>)   .toJSON(); expect(tree).toMatchInlineSnapshot();});The next time you run Jest, tree will be evaluated, and a snapshot will be written as an argument to toMatchInlineSnapshot:it('renders correctly', () => { const tree = renderer   .create(<Link page="https://example.com">Example Site</Link>)   .toJSON(); expect(tree).toMatchInlineSnapshot(`<a  className="normal" href="https://example.com"  onMouseEnter={[Function]}  onMouseLeave={[Function]}>  Example Site</a>`);});That's all there is to it! You can even update the snapshots with --updateSnapshot or using the u key in --watch mode.By default, Jest handles the writing of snapshots into your source code. However, if you're using prettier in your project, Jest will detect this and delegate the work to prettier instead (including honoring your configuration).Property Matchers Often there are fields in the object you want to snapshot which are generated (like IDs and Dates). If you try to snapshot these objects, they will force the snapshot to fail on every run:it('will fail every time', () => { const user = {   createdAt: new Date(),   id: Math.floor(Math.random() * 20),   name: 'LeBron James', }; expect(user).toMatchSnapshot();});// Snapshotexports[`will fail every time 1`] = `{  "createdAt": 2018-05-19T23:36:09.816Z,  "id": 3,  "name": "LeBron James",}`;For these cases, Jest allows providing an asymmetric matcher for any property. These matchers are checked before the snapshot is written or tested, and then saved to the snapshot file instead of the received value:it('will check the matchers and pass', () => { const user = {   createdAt: new Date(),   id: Math.floor(Math.random() * 20),   name:

'LeBron James', }; expect(user).toMatchSnapshot({ createdAt: expect.any(Date), id: expect.any(Number), });});// Snapshotexports[`will check the matchers and pass 1`] = `{ "createdAt": Any<Date>, "id": Any<Number>, "name": "LeBron James",}`;Any given value that is not a matcher will be checked exactly and saved to the snapshot:it('will check the values and pass', () => { const user = { createdAt: new Date(), name: 'Bond... James Bond', }; expect(user).toMatchSnapshot({ createdAt: expect.any(Date), name: 'Bond... James Bond', });});// Snapshotexports[`will check the values and pass 1`] = `{ "createdAt": Any<Date>, "name": 'Bond... James Bond',}`;tipIf the case concerns a string not an object then you need to replace random part of that string on your own before testing the snapshot.
You can use for that e.g. replace() and regular expressions.const randomNumber = Math.round(Math.random() * 100);const stringWithRandomData = `<div id="${randomNumber}">Lorem ipsum</div>`;const stringWithConstantData = stringWithRandomData.replace(/id="\d+"/, 123);expect(stringWithConstantData).toMatchSnapshot();Another way is to mock the library responsible for generating the random part of the code you're snapshotting.Best Practices Snapshots are a fantastic tool for identifying unexpected interface changes within your application – whether that interface is an API response, UI, logs, or error messages. As with any testing strategy, there are some best-practices you should be aware of, and guidelines you should follow, in order to use them effectively.1. Treat snapshots as code Commit snapshots and review them as part of your regular code review process. This means treating snapshots as you would any other type of test or code in your project.Ensure that your snapshots are readable by keeping them focused, short, and by using tools that enforce these stylistic conventions.As mentioned previously, Jest uses pretty-format to make snapshots human-readable, but you may find it useful to introduce additional tools, like eslint-plugin-jest with its no-large-snapshots option, or snapshot-diff with its component snapshot comparison feature, to promote committing short, focused assertions.The goal is to make it easy to review snapshots in pull requests, and fight against the habit of regenerating snapshots when test suites fail instead of examining the root causes of their failure.2. Tests should be deterministic Your tests should be deterministic. Running the same tests multiple times on a component that has not changed should produce the same results every time. You're responsible for making sure your generated snapshots do not include platform specific or other non-deterministic data.For example, if you have a Clock component that uses Date.now(), the snapshot generated from this component will be different every time the test case is run. In this case we can mock the Date.now() method to return a consistent value every time the test is run:Date.now = jest.fn(() => 1482363367071);Now, every time the snapshot test case runs, Date.now() will return 1482363367071 consistently. This will result in the same snapshot being generated for this component regardless of when the test is run.3. Use descriptive snapshot names Always strive to use descriptive test and/or snapshot names for snapshots. The best names describe the expected snapshot content. This makes it easier for reviewers to verify the snapshots during review, and for anyone to know whether or not an outdated snapshot is the correct behavior before updating.For example, compare:exports[`<UserName /> should handle some test case`] = `null`;exports[`<UserName /> should handle some other test case`] = `<div> Alan

Turing</div>`;To:exports[`<UserName /> should render null`] =
`null`;exports[`<UserName /> should render Alan Turing`] = `<div> Alan Turing</
div>`;Since the latter describes exactly what's expected in the output, it's more clear to
see when it's wrong:exports[`<UserName /> should render null`] = `<div> Alan Turing</
div>`;exports[`<UserName /> should render Alan Turing`] = `null`;Frequently Asked
Questions Are snapshots written automatically on Continuous Integration (CI) systems?
No, as of Jest 20, snapshots in Jest are not automatically written when Jest is run in a
CI system without explicitly passing --updateSnapshot. It is expected that all snapshots
are part of the code that is run on CI and since new snapshots automatically pass, they
should not pass a test run on a CI system. It is recommended to always commit all
snapshots and to keep them in version control.Should snapshot files be committed?
Yes, all snapshot files should be committed alongside the modules they are covering
and their tests. They should be considered part of a test, similar to the value of any
other assertion in Jest. In fact, snapshots represent the state of the source modules at
any given point in time. In this way, when the source modules are modified, Jest can tell
what changed from the previous version. It can also provide a lot of additional context
during code review in which reviewers can study your changes better.Does snapshot
testing only work with React components? React and React Native components are a
good use case for snapshot testing. However, snapshots can capture any serializable
value and should be used anytime the goal is testing whether the output is correct. The
Jest repository contains many examples of testing the output of Jest itself, the output of
Jest's assertion library as well as log messages from various parts of the Jest
codebase. See an example of snapshotting CLI output in the Jest repo.What's the
difference between snapshot testing and visual regression testing? Snapshot testing
and visual regression testing are two distinct ways of testing UIs, and they serve
different purposes. Visual regression testing tools take screenshots of web pages and
compare the resulting images pixel by pixel. With Snapshot testing values are
serialized, stored within text files, and compared using a diff algorithm. There are
different trade-offs to consider and we listed the reasons why snapshot testing was built
in the Jest blog.Does snapshot testing replace unit testing? Snapshot testing is only one
of more than 20 assertions that ship with Jest. The aim of snapshot testing is not to
replace existing unit tests, but to provide additional value and make testing painless. In
some scenarios, snapshot testing can potentially remove the need for unit testing for a
particular set of functionalities (e.g. React components), but they can work together as
well.What is the performance of snapshot testing regarding speed and size of the
generated files? Jest has been rewritten with performance in mind, and snapshot testing
is not an exception. Since snapshots are stored within text files, this way of testing is
fast and reliable. Jest generates a new file for each test file that invokes the
toMatchSnapshot matcher. The size of the snapshots is pretty small: For reference, the
size of all snapshot files in the Jest codebase itself is less than 300 KB.How do I
resolve conflicts within snapshot files? Snapshot files must always represent the current
state of the modules they are covering. Therefore, if you are merging two branches and
encounter a conflict in the snapshot files, you can either resolve the conflict manually or
update the snapshot file by running Jest and inspecting the result.Is it possible to apply
test-driven development principles with snapshot testing? Although it is possible to write
snapshot files manually, that is usually not approachable. Snapshots help to figure out

whether the output of the modules covered by tests is changed, rather than giving guidance to design the code in the first place.Does code coverage work with snapshot testing? Yes, as well as with any other test.
An Async ExampleFirst, enable Babel support in Jest as documented in the Getting Started guide.Let's implement a module that fetches user data from an API and returns the user name.user.jsimport request from './request';export function getUserName(userID) { return request(`/users/${userID}`).then(user => user.name);}In the above implementation, we expect the request.js module to return a promise. We chain a call to then to receive the user name.Now imagine an implementation of request.js that goes to the network and fetches some user data:request.jsconst http = require('http');export default function request(url) { return new Promise(resolve => {   // This is an example of an http request, for example to fetch   // user data from an API.   // This module is being mocked in __mocks__/request.js   http.get({path: url}, response => {   let data = '';   response.on('data', _data => (data += _data));   response.on('end', () => resolve(data));   }); });}Because we don't want to go to the network in our test, we are going to create a manual mock for our request.js module in the __mocks__ folder (the folder is case-sensitive, __MOCKS__ will not work). It could look something like this:__mocks__/request.jsconst users = { 4: {name: 'Mark'}, 5: {name: 'Paul'},};export default function request(url) { return new Promise((resolve, reject) => {   const userID = parseInt(url.substr('/users/'.length), 10);   process.nextTick(() =>   users[userID]   ? resolve(users[userID])   : reject({   error: `User with ${userID} not found.`,   }),   ); });}Now let's write a test for our async functionality.__tests__/user-test.jsjest.mock('../request');import * as user from '../user';// The assertion for a promise must be returned.it('works with promises', () => { expect.assertions(1); return user.getUserName(4).then(data => expect(data).toBe('Mark'));});We call jest.mock('../request') to tell Jest to use our manual mock. it expects the return value to be a Promise that is going to be resolved. You can chain as many Promises as you like and call expect at any time, as long as you return a Promise at the end..resolves There is a less verbose way using resolves to unwrap the value of a fulfilled promise together with any other matcher. If the promise is rejected, the assertion will fail.it('works with resolves', () => { expect.assertions(1); return expect(user.getUserName(5)).resolves.toBe('Paul');});async/await Writing tests using the async/await syntax is also possible. Here is how you'd write the same examples from before:// async/await can be used.it('works with async/await', async () => { expect.assertions(1); const data = await user.getUserName(4); expect(data).toBe('Mark');});// async/await can also be used with `.resolves`.it('works with async/await and resolves', async () => { expect.assertions(1); await expect(user.getUserName(5)).resolves.toBe('Paul');});To enable async/await in your project, install @babel/preset-env and enable the feature in your babel.config.js file.Error handling Errors can be handled using the .catch method. Make sure to add expect.assertions to verify that a certain number of assertions are called. Otherwise a fulfilled promise would not fail the test:// Testing for async errors using Promise.catch.it('tests error with promises', () => { expect.assertions(1); return user.getUserName(2).catch(e =>   expect(e).toEqual({   error: 'User with 2 not found.',   }),);});// Or using async/await.it('tests error with async/await', async () => { expect.assertions(1); try {   await user.getUserName(1); } catch (e)

{   expect(e).toEqual({   error: 'User with 1 not found.',   }); }});.rejects The.rejects helper works like the .resolves helper. If the promise is fulfilled, the test will automatically fail. expect.assertions(number) is not required but recommended to verify that a certain number of assertions are called during a test. It is otherwise easy to forget to return/await the .resolves assertions.// Testing for async errors using `.rejects`.it('tests error with rejects', () => { expect.assertions(1); return expect(user.getUserName(3)).rejects.toEqual({   error: 'User with 3 not found.', });});// Or using async/await with `.rejects`.it('tests error with async/await and rejects', async () => { expect.assertions(1); await expect(user.getUserName(3)).rejects.toEqual({   error: 'User with 3 not found.', });});The code for this example is available at examples/async.If you'd like to test timers, like setTimeout, take a look at the Timer mocks documentation.
Timer MocksThe 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!infoAlso see Fake Timers API documentation.Enable Fake Timers In the following example we enable fake timers by calling jest.useFakeTimers(). This is replacing the original implementation of setTimeout() and other timer functions. Timers can be restored to their normal behavior with jest.useRealTimers().timerGame.jsfunction timerGame(callback) { console.log('Ready....go!'); setTimeout(() => {   console.log("Time's up -- stop!");   callback && callback(); }, 1000);}module.exports = timerGame;__tests__/timerGame-test.jsjest.useFakeTimers();jest.spyOn(global, 'setTimeout');test('waits 1 second before ending the game', () => { const timerGame = require('../timerGame'); timerGame(); expect(setTimeout).toHaveBeenCalledTimes(1); expect(setTimeout).toHaveBeenLastCalledWith(expect.any(Function), 1000);});Run All Timers Another test we might want to write for this module is one that asserts that the callback is called after 1 second. To do this, we're going to use Jest's timer control APIs to fast-forward time right in the middle of the test:jest.useFakeTimers();test('calls the callback after 1 second', () => { const timerGame = require('../timerGame'); const callback = jest.fn(); timerGame(callback); // At this point in time, the callback should not have been called yet expect(callback).not.toBeCalled(); // Fast-forward until all timers have been executed jest.runAllTimers(); // Now our callback should have been called! expect(callback).toBeCalled(); expect(callback).toHaveBeenCalledTimes(1);});Run Pending Timers There are also scenarios where you might have a recursive timer – that is a timer that sets a new timer in its own callback. For these, running all the timers would be an endless loop, throwing the following error: "Aborting after running 100000 timers, assuming an infinite loop!"If that is your case, using jest.runOnlyPendingTimers() will solve the problem:infiniteTimerGame.jsfunction infiniteTimerGame(callback) { console.log('Ready....go!'); setTimeout(() => {   console.log("Time's up! 10 seconds before the next game starts...");   callback && callback();   // Schedule the next game in 10 seconds   setTimeout(() => {   infiniteTimerGame(callback);   }, 10000); }, 1000);}module.exports = infiniteTimerGame;__tests__/infiniteTimerGame-test.jsjest.useFakeTimers();jest.spyOn(global, 'setTimeout');describe('infiniteTimerGame', () => { test('schedules a 10-second timer

after 1 second', () => {   const infiniteTimerGame = require('../infiniteTimerGame');   const callback = jest.fn();   infiniteTimerGame(callback);   // At this point in time, there should have been a single call to   // setTimeout to schedule the end of the game in 1 second.   expect(setTimeout).toHaveBeenCalledTimes(1);   expect(setTimeout).toHaveBeenLastCalledWith(expect.any(Function), 1000);   // Fast forward and exhaust only currently pending timers   // (but not any new timers that get created during that process)   jest.runOnlyPendingTimers();   // At this point, our 1-second timer should have fired its callback   expect(callback).toBeCalled();   // And it should have created a new timer to start the game over in   // 10 seconds   expect(setTimeout).toHaveBeenCalledTimes(2);   expect(setTimeout).toHaveBeenLastCalledWith(expect.any(Function), 10000); });});noteFor debugging or any other reason you can change the limit of timers that will be run before throwing an error:jest.useFakeTimers({timerLimit: 100});Advance Timers by Time Another possibility is use jest.advanceTimersByTime(msToRun). When this API is called, all timers are advanced by msToRun milliseconds. All pending "macro-tasks" that have been queued via setTimeout() or setInterval(), and would be executed during this time frame, will be executed. Additionally, if those macro-tasks schedule new macro-tasks that would be executed within the same time frame, those will be executed until there are no more macro-tasks remaining in the queue that should be run within msToRun milliseconds.timerGame.jsfunction timerGame(callback) { console.log('Ready....go!'); setTimeout(() => {   console.log("Time's up -- stop!");   callback && callback(); }, 1000);}module.exports = timerGame;__tests__/timerGame-test.jsjest.useFakeTimers();it('calls the callback after 1 second via advanceTimersByTime', () => { const timerGame = require('../timerGame'); const callback = jest.fn(); timerGame(callback); // At this point in time, the callback should not have been called yet expect(callback).not.toBeCalled(); // Fast-forward until all timers have been executed jest.advanceTimersByTime(1000); // Now our callback should have been called! expect(callback).toBeCalled(); expect(callback).toHaveBeenCalledTimes(1);});Lastly, it may occasionally be useful in some tests to be able to clear all of the pending timers. For this, we have jest.clearAllTimers().Selective Faking Sometimes your code may require to avoid overwriting the original implementation of one or another API. If that is the case, you can use doNotFake option. For example, here is how you could provide a custom mock function for performance.mark() in jsdom environment:/** * @jest-environment jsdom */ const mockPerformanceMark = jest.fn();window.performance.mark = mockPerformanceMark;test('allows mocking `performance.mark()`', () => { jest.useFakeTimers({doNotFake: ['performance']}); expect(window.performance.mark).toBe(mockPerformanceMark);});

Manual MocksManual mocks are used to stub out functionality with mock data. For example, instead of accessing a remote resource like a website or a database, you might want to create a manual mock that allows you to use fake data. This ensures your tests will be fast and not flaky.Mocking user modules Manual mocks are defined by writing a module in a __mocks__/ subdirectory immediately adjacent to the module. For example, to mock a module called user in the models directory, create a file called user.js and put it in the models/__mocks__ directory.cautionThe __mocks__ folder is case-sensitive, so naming the directory __MOCKS__ will break on some

systems.noteWhen we require that module in our tests (meaning we want to use the manual mock instead of the real implementation), explicitly calling jest.mock('./moduleName') is required.Mocking Node modules If the module you are mocking is a Node module (e.g.: lodash), the mock should be placed in the __mocks__ directory adjacent to node_modules (unless you configured roots to point to a folder other than the project root) and will be automatically mocked. There's no need to explicitly call jest.mock('module_name').Scoped modules (also known as scoped packages) can be mocked by creating a file in a directory structure that matches the name of the scoped module. For example, to mock a scoped module called @scope/project-name, create a file at __mocks__/@scope/project-name.js, creating the @scope/ directory accordingly.cautionIf we want to mock Node's build-in modules (e.g.: fs or path), then explicitly calling e.g. jest.mock('path') is required, because build-in modules are not mocked by default.Examples .% % % config% % % __mocks__%   % % % fs.js% % % models% user.js%   % % % user.js% % % node_modules% % % viewsWhen a manual mock exists for a give
Jest's module system will use that module when explicitly calling jest.mock('moduleName'). However, when automock is set to true, the manual mock implementation will be used instead of the automatically created mock, even if jest.mock('moduleName') is not called. To opt out of this behavior you will need to explicitly call jest.unmock('moduleName') in tests that should use the actual module implementation.infoIn order to mock properly, Jest needs jest.mock('moduleName') to be in the same scope as the require/import statement.Here's a contrived example where we have a module that provides a summary of all the files in a given directory. In this case, we use the core (built in) fs module.FileSummarizer.js'use strict';const fs = require('fs');function summarizeFilesInDirectorySync(directory) { return fs.readdirSync(directory).map(fileName => ({   directory,   fileName, }));}
exports.summarizeFilesInDirectorySync = summarizeFilesInDirectorySync;Since we'd like our tests to avoid actually hitting the disk (that's pretty slow and fragile), we create a manual mock for the fs module by extending an automatic mock. Our manual mock will implement custom versions of the fs APIs that we can build on for our tests:__mocks__/fs.js'use strict';const path = require('path');const fs = jest.createMockFromModule('fs');// This is a custom function that our tests can use during setup to specify// what the files on the "mock" filesystem should look like when any of the// `fs` APIs are used.let mockFiles = Object.create(null);function __setMockFiles(newMockFiles) { mockFiles = Object.create(null); for (const file in newMockFiles) {   const dir = path.dirname(file);   if (!mockFiles[dir]) {   mockFiles[dir] = [];   }   mockFiles[dir].push(path.basename(file)); }}// A custom version of `readdirSync` that reads from the special mocked out// file list set via __setMockFilesfunction readdirSync(directoryPath) { return mockFiles[directoryPath] || [];}fs.__setMockFiles = __setMockFiles;fs.readdirSync = readdirSync;module.exports = fs;Now we write our test. In this case jest.mock('fs') must be called explicitly, because fs is Node’s build-in module:__tests__/FileSummarizer-test.js'use strict';jest.mock('fs');describe('listFilesInDirectorySync', () => { const MOCK_FILE_INFO = {   '/path/to/file1.js': 'console.log("file1 contents");',   '/path/to/file2.txt': 'file2 contents', }; beforeEach(() => {   // Set up some mocked out file info before each test   require('fs').__setMockFiles(MOCK_FILE_INFO); }); test('includes all files in the directory in the summary', () => {   const FileSummarizer = require('../

FileSummarizer');   const fileSummary =
FileSummarizer.summarizeFilesInDirectorySync('/path/to');
expect(fileSummary.length).toBe(2); });});The example mock shown here uses jest.createMockFromModule to generate an automatic mock, and overrides its default behavior. This is the recommended approach, but is completely optional. If you do not want to use the automatic mock at all, you can export your own functions from the mock file. One downside to fully manual mocks is that they're manual – meaning you have to manually update them any time the module they are mocking changes. Because of this, it's best to use or extend the automatic mock when it works for your needs.To ensure that a manual mock and its real implementation stay in sync, it might be useful to require the real module using jest.requireActual(moduleName) in your manual mock and amending it with mock functions before exporting it.The code for this example is available at examples/manual-mocks.Using with ES module imports If you're using ES module imports then you'll normally be inclined to put your import statements at the top of the test file. But often you need to instruct Jest to use a mock before modules use it. For this reason, Jest will automatically hoist jest.mock calls to the top of the module (before any imports). To learn more about this and see it in action, see this repo.cautionjest.mock calls cannot be hoisted to the top of the module if you enabled ECMAScript modules support. The ESM module loader always evaluates the static imports before executing code. See ECMAScriptModules for details.Mocking methods which are not implemented in JSDOM If some code uses a method which JSDOM (the DOM implementation used by Jest) hasn't implemented yet, testing it is not easily possible. This is e.g. the case with window.matchMedia(). Jest returns TypeError: window.matchMedia is not a function and doesn't properly execute the test.In this case, mocking matchMedia in the test file should solve the issue:Object.defineProperty(window, 'matchMedia', { writable: true, value: jest.fn().mockImplementation(query => ({   matches: false,   media: query,   onchange: null,   addListener: jest.fn(), // deprecated   removeListener: jest.fn(), // deprecated   addEventListener: jest.fn(),   removeEventListener: jest.fn(),   dispatchEvent: jest.fn(), })),});This works if window.matchMedia() is used in a function (or method) which is invoked in the test. If window.matchMedia() is executed directly in the tested file, Jest reports the same error. In this case, the solution is to move the manual mock into a separate file and include this one in the test before the tested file:import './matchMedia.mock'; // Must be imported before the tested fileimport {myMethod} from './file-to-test';describe('myMethod()', () => { // Test the method here...});
ES6 Class MocksJest can be used to mock ES6 classes that are imported into files you want to test.ES6 classes are constructor functions with some syntactic sugar. Therefore, any mock for an ES6 class must be a function or an actual ES6 class (which is, again, another function). So you can mock them using mock functions.An ES6 Class Example We'll use a contrived example of a class that plays sound files, SoundPlayer, and a consumer class which uses that class, SoundPlayerConsumer. We'll mock SoundPlayer in our tests for SoundPlayerConsumer.sound-player.jsexport default class SoundPlayer { constructor() {   this.foo = 'bar'; } playSoundFile(fileName) {   console.log('Playing sound file ' + fileName); }}sound-player-consumer.jsimport SoundPlayer from './sound-player';export default class SoundPlayerConsumer { constructor() {   this.soundPlayer = new SoundPlayer(); } playSomethingCool()

{   const coolSoundFileName = 'song.mp3';
this.soundPlayer.playSoundFile(coolSoundFileName); }}The 4 ways to create an ES6 class mock Automatic mock Calling jest.mock('./sound-player') returns a useful "automatic mock" you can use to spy on calls to the class constructor and all of its methods. It replaces the ES6 class with a mock constructor, and replaces all of its methods with mock functions that always return undefined. Method calls are saved in theAutomaticMock.mock.instances[index].methodName.mock.calls.noteIf you use arrow functions in your classes, they will not be part of the mock. The reason for that is that arrow functions are not present on the object's prototype, they are merely properties holding a reference to a function.If you don't need to replace the implementation of the class, this is the easiest option to set up. For example:import SoundPlayer from './sound-player';import SoundPlayerConsumer from './sound-player-consumer';jest.mock('./sound-player'); // SoundPlayer is now a mock constructorbeforeEach(() => { // Clear all instances and calls to constructor and all methods: SoundPlayer.mockClear();});it('We can check if the consumer called the class constructor', () => { const soundPlayerConsumer = new SoundPlayerConsumer();
expect(SoundPlayer).toHaveBeenCalledTimes(1);});it('We can check if the consumer called a method on the class instance', () => { // Show that mockClear() is working:
expect(SoundPlayer).not.toHaveBeenCalled(); const soundPlayerConsumer = new SoundPlayerConsumer(); // Constructor should have been called again:
expect(SoundPlayer).toHaveBeenCalledTimes(1); const coolSoundFileName = 'song.mp3'; soundPlayerConsumer.playSomethingCool(); // mock.instances is available with automatic mocks: const mockSoundPlayerInstance = SoundPlayer.mock.instances[0]; const mockPlaySoundFile = mockSoundPlayerInstance.playSoundFile; expect(mockPlaySoundFile.mock.calls[0][0]).toBe(coolSoundFileName); // Equivalent to above check:
expect(mockPlaySoundFile).toHaveBeenCalledWith(coolSoundFileName);
expect(mockPlaySoundFile).toHaveBeenCalledTimes(1);});Manual mock Create a manual mock by saving a mock implementation in the __mocks__ folder. This allows you to specify the implementation, and it can be used across test files.__mocks__/sound-player.js// Import this named export into your test file:export const mockPlaySoundFile = jest.fn();const mock = jest.fn().mockImplementation(() => { return {playSoundFile: mockPlaySoundFile};});export default mock;Import the mock and the mock method shared by all instances:sound-player-consumer.test.jsimport SoundPlayer, {mockPlaySoundFile} from './sound-player';import SoundPlayerConsumer from './sound-player-consumer';jest.mock('./sound-player'); // SoundPlayer is now a mock constructorbeforeEach(() => { // Clear all instances and calls to constructor and all methods: SoundPlayer.mockClear(); mockPlaySoundFile.mockClear();});it('We can check if the consumer called the class constructor', () => { const soundPlayerConsumer = new SoundPlayerConsumer();
expect(SoundPlayer).toHaveBeenCalledTimes(1);});it('We can check if the consumer called a method on the class instance', () => { const soundPlayerConsumer = new SoundPlayerConsumer(); const coolSoundFileName = 'song.mp3';
soundPlayerConsumer.playSomethingCool();
expect(mockPlaySoundFile).toHaveBeenCalledWith(coolSoundFileName);});Calling jest.mock() with the module factory parameter jest.mock(path, moduleFactory) takes a

module factory argument. A module factory is a function that returns the mock.In order to mock a constructor function, the module factory must return a constructor function. In other words, the module factory must be a function that returns a function - a higher-order function (HOF).import SoundPlayer from './sound-player';const mockPlaySoundFile = jest.fn();jest.mock('./sound-player', () => { return jest.fn().mockImplementation(() => {   return {playSoundFile: mockPlaySoundFile}; });});cautionSince calls to jest.mock() are hoisted to the top of the file, Jest prevents access to out-of-scope variables. By default, you cannot first define a variable and then use it in the factory. Jest will disable this check for variables that start with the word mock. However, it is still up to you to guarantee that they will be initialized on time. Be aware of Temporal Dead Zone.For example, the following will throw an out-of-scope error due to the use of fake instead of mock in the variable declaration.// Note: this will failimport SoundPlayer from './sound-player';const fakePlaySoundFile = jest.fn();jest.mock('./sound-player', () => { return jest.fn().mockImplementation(() => {   return {playSoundFile: fakePlaySoundFile}; });});The following will throw a ReferenceError despite using mock in the variable declaration, as the mockSoundPlayer is not wrapped in an arrow function and thus accessed before initialization after hoisting.import SoundPlayer from './sound-player';const mockSoundPlayer = jest.fn().mockImplementation(() => { return {playSoundFile: mockPlaySoundFile};});// results in a ReferenceErrorjest.mock('./sound-player', () => { return mockSoundPlayer;});Replacing the mock using mockImplementation() or mockImplementationOnce() You can replace all of the above mocks in order to change the implementation, for a single test or all tests, by calling mockImplementation() on the existing mock.Calls to jest.mock are hoisted to the top of the code. You can specify a mock later, e.g. in beforeAll(), by calling mockImplementation() (or mockImplementationOnce()) on the existing mock instead of using the factory parameter. This also allows you to change the mock between tests, if needed:import SoundPlayer from './sound-player';import SoundPlayerConsumer from './sound-player-consumer';jest.mock('./sound-player');describe('When SoundPlayer throws an error', () => { beforeAll(() => {   SoundPlayer.mockImplementation(() => {   return {   playSoundFile: () => {   throw new Error('Test error');   },   };   }); }); it('Should throw an error when calling playSomethingCool', () => {   const soundPlayerConsumer = new SoundPlayerConsumer();   expect(() => soundPlayerConsumer.playSomethingCool()).toThrow(); });});In depth: Understanding mock constructor functions Building your constructor function mock using jest.fn().mockImplementation() makes mocks appear more complicated than they really are. This section shows how you can create your own mocks to illustrate how mocking works.Manual mock that is another ES6 class If you define an ES6 class using the same filename as the mocked class in the __mocks__ folder, it will serve as the mock. This class will be used in place of the real class. This allows you to inject a test implementation for the class, but does not provide a way to spy on calls.For the contrived example, the mock might look like this:__mocks__/sound-player.jsexport default class SoundPlayer { constructor() {   console.log('Mock SoundPlayer: constructor was called'); } playSoundFile() {   console.log('Mock SoundPlayer: playSoundFile was called'); }}Mock using module factory parameter The module factory function passed to jest.mock(path, moduleFactory) can be a HOF that returns a function*. This will allow calling new on the mock. Again, this allows you to inject different behavior for testing, but does not provide a way to spy on calls.* Module factory function must return a function In order to mock a constructor function, the module factory must return a constructor function. In other words, the module factory must be a function that returns a function - a higher-order function (HOF).jest.mock('./sound-player', () => { return function () {   return {playSoundFile: () => {}}; };});noteThe mock can't be an arrow function because calling new on an arrow function is not allowed in JavaScript. So this won't work:jest.mock('./sound-player', () => { return () => {   // Does not work; arrow functions can't be called with new   return {playSoundFile: () => {}}; };});This will throw TypeError: _soundPlayer2.default is not a constructor, unless the code is transpiled to ES5, e.g. by @babel/preset-env. (ES5 doesn't have arrow functions nor classes, so both will be transpiled to plain functions.)Mocking a specific method of a class Lets say that you want to mock or spy on the method playSoundFile within the class SoundPlayer. A simple example:// your jest test file belowimport SoundPlayer from './sound-player';import SoundPlayerConsumer from './sound-player-consumer';const playSoundFileMock = jest .spyOn(SoundPlayer.prototype, 'playSoundFile') .mockImplementation(() => {   console.log('mocked function'); }); // comment this line if just want to "spy"it('player consumer plays music', () => { const player = new SoundPlayerConsumer(); player.playSomethingCool(); expect(playSoundFileMock).toHaveBeenCalled();});Static, getter and setter methods Lets imagine our class SoundPlayer has a getter method foo and a static method brandexport default class SoundPlayer { constructor() {   this.foo = 'bar'; } playSoundFile(fileName) {   console.log('Playing sound file ' + fileName); } get foo() {   return 'bar'; } static brand() {   return 'player-brand'; }}You can mock/spy on them easily, here is an example:// your jest test file belowimport SoundPlayer from './sound-player';const staticMethodMock = jest .spyOn(SoundPlayer, 'brand') .mockImplementation(() => 'some-mocked-brand');const getterMethodMock = jest .spyOn(SoundPlayer.prototype, 'foo', 'get') .mockImplementation(() => 'some-mocked-result');it('custom methods are called', () => { const player = new SoundPlayer(); const foo = player.foo; const brand = SoundPlayer.brand(); expect(staticMethodMock).toHaveBeenCalled(); expect(getterMethodMock).toHaveBeenCalled();});Keeping track of usage (spying on the mock) Injecting a test implementation is helpful, but you will probably also want to test whether the class constructor and methods are called with the correct parameters.Spying on the constructor In order to track calls to the constructor, replace the function returned by the HOF with a Jest mock function. Create it with jest.fn(), and then specify its implementation with mockImplementation().import SoundPlayer from './sound-player';jest.mock('./sound-player', () => { // Works and lets you check for constructor calls: return jest.fn().mockImplementation(() => {   return {playSoundFile: () => {}}; });});This will let us inspect usage of our mocked class, using SoundPlayer.mock.calls: expect(SoundPlayer).toHaveBeenCalled(); or near-equivalent: expect(SoundPlayer.mock.calls.length).toBeGreaterThan(0);Mocking non-default class exports If the class is not the default export from the module then you need to return an object with the key that is the same as the class export name.import {SoundPlayer} from './sound-player';jest.mock('./sound-player', () => { // Works and lets you check for constructor calls: return {   SoundPlayer: jest.fn().mockImplementation(() =>

{   return {playSoundFile: () => {}};   }), };});Spying on methods of our class Our mocked class will need to provide any member functions (playSoundFile in the example) that will be called during our tests, or else we'll get an error for calling a function that doesn't exist. But we'll probably want to also spy on calls to those methods, to ensure that they were called with the expected parameters.A new object will be created each time the mock constructor function is called during tests. To spy on method calls in all of these objects, we populate playSoundFile with another mock function, and store a reference to that same mock function in our test file, so it's available during tests.import SoundPlayer from './sound-player';const mockPlaySoundFile = jest.fn();jest.mock('./sound-player', () => { return jest.fn().mockImplementation(() => {   return {playSoundFile: mockPlaySoundFile};   // Now we can track calls to playSoundFile });});The manual mock equivalent of this would be:__mocks__/sound-player.js// Import this named export into your test fileexport const mockPlaySoundFile = jest.fn();const mock = jest.fn().mockImplementation(() => { return {playSoundFile: mockPlaySoundFile};});export default mock;Usage is similar to the module factory function, except that you can omit the second argument from jest.mock(), and you must import the mocked method into your test file, since it is no longer defined there. Use the original module path for this; don't include __mocks__.Cleaning up between tests To clear the record of calls to the mock constructor function and its methods, we call mockClear() in the beforeEach() function:beforeEach(() => { SoundPlayer.mockClear(); mockPlaySoundFile.mockClear();});Complete example Here's a complete test file which uses the module factory parameter to jest.mock:sound-player-consumer.test.jsimport SoundPlayer from './sound-player';import SoundPlayerConsumer from './sound-player-consumer';const mockPlaySoundFile = jest.fn();jest.mock('./sound-player', () => { return jest.fn().mockImplementation(() => {   return {playSoundFile: mockPlaySoundFile}; });});beforeEach(() => { SoundPlayer.mockClear(); mockPlaySoundFile.mockClear();});it('The consumer should be able to call new() on SoundPlayer', () => { const soundPlayerConsumer = new SoundPlayerConsumer(); // Ensure constructor created the object: expect(soundPlayerConsumer).toBeTruthy();});it('We can check if the consumer called the class constructor', () => { const soundPlayerConsumer = new SoundPlayerConsumer(); expect(SoundPlayer).toHaveBeenCalledTimes(1);});it('We can check if the consumer called a method on the class instance', () => { const soundPlayerConsumer = new SoundPlayerConsumer(); const coolSoundFileName = 'song.mp3'; soundPlayerConsumer.playSomethingCool(); expect(mockPlaySoundFile.mock.calls[0][0]).toBe(coolSoundFileName);});
Bypassing module mocksJest allows you to mock out whole modules in your tests, which can be useful for testing if your code is calling functions from that module correctly. However, sometimes you may want to use parts of a mocked module in your test file, in which case you want to access the original implementation, rather than a mocked version.Consider writing a test case for this createUser function:createUser.jsimport fetch from 'node-fetch';export const createUser = async () => { const response = await fetch('https://website.com/users', {method: 'POST'}); const userId = await response.text(); return userId;};Your test will want to mock the fetch function so that we can be sure that it gets called without actually making the network

request. However, you'll also need to mock the return value of fetch with a Response (wrapped in a Promise), as our function uses it to grab the created user's ID. So you might initially try writing a test like this:jest.mock('node-fetch');import fetch, {Response} from 'node-fetch';import {createUser} from './createUser';test('createUser calls fetch with the right args and returns the user id', async () => { fetch.mockReturnValue(Promise.resolve(new Response('4'))); const userId = await createUser(); expect(fetch).toHaveBeenCalledTimes(1); expect(fetch).toHaveBeenCalledWith('https://website.com/users', {   method: 'POST', }); expect(userId).toBe('4');});However, if you ran that test you would find that the createUser function would fail, throwing the error: TypeError: response.text is not a function. This is because the Response class you've imported from node-fetch has been mocked (due to the jest.mock call at the top of the test file) so it no longer behaves the way it should.To get around problems like this, Jest provides the jest.requireActual helper. To make the above test work, make the following change to the imports in the test file:// BEFOREjest.mock('node-fetch');import fetch, {Response} from 'node-fetch';// AFTERjest.mock('node-fetch');import fetch from 'node-fetch';const {Response} = jest.requireActual('node-fetch');This allows your test file to import the actual Response object from node-fetch, rather than a mocked version. This means the test will now pass correctly.
ECMAScript ModulescautionJest ships with experimental support for ECMAScript Modules (ESM).The implementation may have bugs and lack features. For the latest status check out the issue and the label on the issue tracker.Also note that the APIs Jest uses to implement ESM support are still considered experimental by Node (as of version 18.8.0).With the warnings out of the way, this is how you activate ESM support in your tests.Ensure you either disable code transforms by passing transform: {} or otherwise configure your transformer to emit ESM rather than the default CommonJS (CJS).Execute node with --experimental-vm-modules, e.g. node --experimental-vm-modules node_modules/jest/bin/jest.js or NODE_OPTIONS=--experimental-vm-modules npx jest etc.On Windows, you can use cross-env to be able to set environment variables.If you use Yarn, you can use yarn node --experimental-vm-modules $(yarn bin jest). This command will also work if you use Yarn Plug'n'Play.If your codebase includes ESM imports from *.wasm files, you do not need to pass --experimental-wasm-modules to node. Current implementation of WebAssembly imports in Jest relies on experimental VM modules, however, this may change in the future.Beyond that, we attempt to follow node's logic for activating "ESM mode" (such as looking at type in package.json or .mjs files), see their docs for details.If you want to treat other file extensions (such as .jsx or .ts) as ESM, please use the extensionsToTreatAsEsm option.Differences between ESM and CommonJS Most of the differences are explained in Node's documentation, but in addition to the things mentioned there, Jest injects a special variable into all executed files - the jest object. To access this object in ESM, you need to import it from the @jest/globals module or use import.meta.import {jest} from '@jest/globals';jest.useFakeTimers();// etc.// alternativelyimport.meta.jest.useFakeTimers();// jest === import.meta.jest => trueModule mocking in ESM Since ESM evaluates static import statements before looking at the code, the hoisting of jest.mock calls that happens in CJS won't work for ESM. To mock modules in ESM, you need to use require or dynamic import() after

jest.mock calls to load the mocked modules - the same applies to modules which load the mocked modules.ESM mocking is supported through jest.unstable_mockModule. As the name suggests, this API is still work in progress, please follow this issue for updates.The usage of jest.unstable_mockModule is essentially the same as jest.mock with two differences: the factory function is required and it can be sync or async:import {jest} from '@jest/globals';jest.unstable_mockModule('node:child_process', () => ({ execSync: jest.fn(), // etc.}));const {execSync} = await import('node:child_process');// etc.For mocking CJS modules, you should continue to use jest.mock. See the example below:main.cjsconst {BrowserWindow, app} = require('electron');// etc.module.exports = {example};main.test.cjsimport {createRequire} from 'node:module';import {jest} from '@jest/globals';const require = createRequire(import.meta.url);jest.mock('electron', () => ({ app: {   on: jest.fn(),   whenReady: jest.fn(() => Promise.resolve()), }, BrowserWindow: jest.fn().mockImplementation(() => ({   // partial mocks. })),}));const {BrowserWindow} = require('electron');const exported = require('./main.cjs');// alternativelyconst {BrowserWindow} = (await import('electron')).default;const exported = await import('./main.cjs');// etc.

Using with webpackJest can be used in projects that use webpack to manage assets, styles, and compilation. webpack does offer some unique challenges over other tools because it integrates directly with your application to allow managing stylesheets, assets like images and fonts, along with the expansive ecosystem of compile-to-JavaScript languages and tools.A webpack example Let's start with a common sort of webpack config file and translate it to a Jest setup.webpack.config.jsmodule.exports = { module: {   rules: [{   test: /\.jsx?$/,   exclude: ['node_modules'],   use: ['babel-loader'],   },   {   test: /\.css$/,   use: ['style-loader', 'css-loader'],   },   {   test: /\.gif$/,   type: 'asset/inline',   },   {   test: /\.(ttf|eot|svg)$/,   type: 'asset/resource',   },   ], }, resolve: {   alias: {   config$: './configs/app-config.js',   react: './vendor/react-master',   },   extensions: ['.js', '.jsx'],   modules: ['node_modules',   'bower_components',   'shared',   '/shared/vendor/modules',   ], },};If you have JavaScript files that are transformed by Babel, you can enable support for Babel by installing the babel-jest plugin. Non-Babel JavaScript transformations can be handled with Jest's transform config option.Handling Static Assets Next, let's configure Jest to gracefully handle asset files such as stylesheets and images. Usually, these files aren't particularly useful in tests so we can safely mock them out. However, if you are using CSS Modules then it's better to mock a proxy for your className lookups.jest.config.jsmodule.exports = { moduleNameMapper: {   '\\.(jpg|jpeg|png|gif|eot|otf|webp|svg|ttf|woff|woff2|mp4|webm|wav|mp3|m4a|aac|oga)$': '<rootDir>/__mocks__/fileMock.js',   '\\.(css|less)$': '<rootDir>/__mocks__/styleMock.js', },};And the mock files themselves:__mocks__/styleMock.jsmodule.exports = {};__mocks__/fileMock.jsmodule.exports = 'test-file-stub';Mocking CSS Modules You can use an ES6 Proxy to mock CSS Modules:npmYarnpnpmnpm install --save-dev identity-obj-proxyyarn add --dev identity-obj-proxypnpm add --save-dev identity-obj-proxyThen all your className lookups on the styles object will be returned as-is (e.g., styles.foobar === 'foobar'). This is pretty handy for React Snapshot Testing.jest.config.js (for CSS Modules)module.exports = { moduleNameMapper: {   '\\.(jpg|jpeg|png|gif|eot|otf|webp|svg|ttf|woff|woff2|mp4|webm|wav|mp3|m4a|aac|oga)$':   '<rootDir>/__mocks__/fileMock.js',   '\\.(css|less)$':

'identity-obj-proxy', },};If moduleNameMapper cannot fulfill your requirements, you can use Jest's transform config option to specify how assets are transformed. For example, a transformer that returns the basename of a file (such that require('logo.jpg'); returns 'logo') can be written as:fileTransformer.jsconst path = require('path');module.exports = { process(sourceText, sourcePath, options) {   return {   code: `module.exports = ${JSON.stringify(path.basename(sourcePath))};`,   }; },};jest.config.js (for custom transformers and CSS Modules)module.exports = { moduleNameMapper: {   '\\.(css|less)$': 'identity-obj-proxy', }, transform: {   '\\.(jpg|jpeg|png|gif|eot|otf|webp|svg|ttf|woff|woff2|mp4|webm|wav|mp3|m4a|aac|oga)$':   '<rootDir>/fileTransformer.js', },};We've told Jest to ignore files matching a stylesheet or image extension, and instead, require our mock files. You can adjust the regular expression to match the file types your webpack config handles.tipRemember to include the default babel-jest transformer explicitly, if you wish to use it alongside with additional code preprocessors:"transform": { "\\.[jt]sx?$": "babel-jest", "\\.css$": "some-css-transformer",}Configuring Jest to find our files Now that Jest knows how to process our files, we need to tell it how to find them. For webpack's modules, and extensions options there are direct analogs in Jest's moduleDirectories and moduleFileExtensions options.jest.config.jsmodule.exports = { moduleFileExtensions: ['js', 'jsx'], moduleDirectories: ['node_modules', 'bower_components', 'shared'], moduleNameMapper: {   '\\.(css|less)$': '<rootDir>/__mocks__/styleMock.js',   '\\.(gif|ttf|eot|svg)$': '<rootDir>/__mocks__/fileMock.js', },};note<rootDir> is a special token that gets replaced by Jest with the root of your project. Most of the time this will be the folder where your package.json is located unless you specify a custom rootDir option in your configuration.Similarly, Jest's counterpart for Webpack's resolve.roots (an alternative to setting NODE_PATH) is modulePaths.jest.config.jsmodule.exports = { modulePaths: ['/shared/vendor/modules'], moduleFileExtensions: ['js', 'jsx'], moduleDirectories: ['node_modules', 'bower_components', 'shared'], moduleNameMapper: {   '\\.(css|less)$': '<rootDir>/__mocks__/styleMock.js',   '\\.(gif|ttf|eot|svg)$': '<rootDir>/__mocks__/fileMock.js', },};And finally, we have to handle the webpack alias. For that, we can make use of the moduleNameMapper option again.jest.config.jsmodule.exports = { modulePaths: ['/shared/vendor/modules'], moduleFileExtensions: ['js', 'jsx'], moduleDirectories: ['node_modules', 'bower_components', 'shared'], moduleNameMapper: {   '\\.(css|less)$': '<rootDir>/__mocks__/styleMock.js',   '\\.(gif|ttf|eot|svg)$': '<rootDir>/__mocks__/fileMock.js',   '^react(.*)$': '<rootDir>/vendor/react-master$1',   '^config$': '<rootDir>/configs/app-config.js', },};That's it! webpack is a complex and flexible tool, so you may have to make some adjustments to handle your specific application's needs. Luckily for most projects, Jest should be more than flexible enough to handle your webpack config.tipFor more complex webpack configurations, you may also want to investigate projects such as: babel-plugin-webpack-loaders.Using with webpack In addition to installing babel-jest as described earlier, you'll need to add @babel/preset-env like so:npmYarnpnpmnpm install --save-dev @babel/preset-envyarn add --dev @babel/preset-envpnpm add --save-dev @babel/preset-envThen, you'll want to configure Babel as follows:.babelrc{ "presets": ["@babel/preset-env"]}tipJest caches files to speed up test execution. If you updated .babelrc and Jest is not working as expected, try clearing the cache by running jest --clearCache.tipIf you use dynamic imports (import('some-file.js').then(module => ...)), you need to enable the dynamic-

import-node plugin..babelrc{ "presets": [["env", {"modules": false}]], "plugins": ["syntax-dynamic-import"], "env": {   "test": {   "plugins": ["dynamic-import-node"]   } }}For an example of how to use Jest with webpack with React, you can view one here.
Using with puppeteerWith the Global Setup/Teardown and Async Test Environment APIs, Jest can work smoothly with puppeteer.noteGenerating code coverage for test files using Puppeteer is currently not possible if your test uses page.$eval, page.$$eval or page.evaluate as the passed function is executed outside of Jest's scope. Check out issue #7962 on GitHub for a workaround.Use jest-puppeteer Preset Jest Puppeteer provides all required configuration to run your tests using Puppeteer.First, install jest-puppeteernpmYarnpnpmnpm install --save-dev jest-puppeteeryarn add --dev jest-puppeteerpnpm add --save-dev jest-puppeteerSpecify preset in your Jest configuration: { "preset": "jest-puppeteer"}Write your testdescribe('Google', () => { beforeAll(async () => {   await page.goto('https://google.com'); }); it('should be titled "Google"', async () => {   await expect(page.title()).resolves.toMatch('Google'); });});There's no need to load any dependencies. Puppeteer's page and browser classes will automatically be exposedSee documentation.Custom example without jest-puppeteer preset You can also hook up puppeteer from scratch. The basic idea is to:launch & file the websocket endpoint of puppeteer with Global Setupconnect to puppeteer from each Test Environmentclose puppeteer with Global TeardownHere's an example of the GlobalSetup scriptsetup.jsconst {mkdir, writeFile} = require('fs').promises;const os = require('os');const path = require('path');const puppeteer = require('puppeteer');const DIR = path.join(os.tmpdir(), 'jest_puppeteer_global_setup');module.exports = async function () { const browser = await puppeteer.launch(); // store the browser instance so we can teardown it later // this global is only available in the teardown but not in TestEnvironments globalThis.__BROWSER_GLOBAL__ = browser; // use the file system to expose the wsEndpoint for TestEnvironments await mkdir(DIR, {recursive: true}); await writeFile(path.join(DIR, 'wsEndpoint'), browser.wsEndpoint());};Then we need a custom Test Environment for puppeteerpuppeteer_environment.jsconst {readFile} = require('fs').promises;const os = require('os');const path = require('path');const puppeteer = require('puppeteer');const NodeEnvironment = require('jest-environment-node').TestEnvironment;const DIR = path.join(os.tmpdir(), 'jest_puppeteer_global_setup');class PuppeteerEnvironment extends NodeEnvironment { constructor(config) {   super(config); } async setup() {   await super.setup();   // get the wsEndpoint   const wsEndpoint = await readFile(path.join(DIR, 'wsEndpoint'), 'utf8');   if (!wsEndpoint) {   throw new Error('wsEndpoint not found');   }   // connect to puppeteer   this.global.__BROWSER_GLOBAL__ = await puppeteer.connect({   browserWSEndpoint: wsEndpoint,   }); } async teardown() {   if (this.global.__BROWSER_GLOBAL__) {   this.global.__BROWSER_GLOBAL__.disconnect();   }   await super.teardown(); } getVmContext() {   return super.getVmContext(); }}module.exports = PuppeteerEnvironment;Finally, we can close the puppeteer instance and clean-up the fileteardown.jsconst fs = require('fs').promises;const os = require('os');const path = require('path');const DIR = path.join(os.tmpdir(), 'jest_puppeteer_global_setup');module.exports = async function () { // close the browser instance await globalThis.__BROWSER_GLOBAL__.close(); // clean-up the wsEndpoint file await fs.rm(DIR, {recursive: true, force: true});};With all the things set

up, we can now write our tests like this:test.jsconst timeout = 5000;describe('/ (Home Page)', () => {   let page;   beforeAll(async () => {   page = await globalThis.__BROWSER_GLOBAL__.newPage();   await page.goto('https://google.com');   }, timeout);   it('should load without error', async () => {   const text = await page.evaluate(() => document.body.textContent);   expect(text).toContain('google');   }); }, timeout,);Finally, set jest.config.js to read from these files. (The jest-puppeteer preset does something like this under the hood.)module.exports = { globalSetup: './setup.js', globalTeardown: './teardown.js', testEnvironment: './puppeteer_environment.js',};Here's the code of full working example.
Using with MongoDBWith the Global Setup/Teardown and Async Test Environment APIs, Jest can work smoothly with MongoDB.Use jest-mongodb Preset Jest MongoDB provides all required configuration to run your tests using MongoDB.First install @shelf/jest-mongodbnpmYarnpnpmnpm install --save-dev @shelf/jest-mongodbyarn add --dev @shelf/jest-mongodbpnpm add --save-dev @shelf/jest-mongodbSpecify preset in your Jest configuration:{ "preset": "@shelf/jest-mongodb"}Write your testconst {MongoClient} = require('mongodb');describe('insert', () => { let connection; let db; beforeAll(async () => {   connection = await MongoClient.connect(globalThis.__MONGO_URI__, {   useNewUrlParser: true,   useUnifiedTopology: true,   });   db = await connection.db(globalThis.__MONGO_DB_NAME__); }); afterAll(async () => {   await connection.close(); }); it('should insert a doc into collection', async () => {   const users = db.collection('users');   const mockUser = {_id: 'some-user-id', name: 'John'};   await users.insertOne(mockUser);   const insertedUser = await users.findOne({_id: 'some-user-id'});   expect(insertedUser).toEqual(mockUser); });});There's no need to load any dependencies.See documentation for details (configuring MongoDB version, etc).
Using with DynamoDBWith the Global Setup/Teardown and Async Test Environment APIs, Jest can work smoothly with DynamoDB.Use jest-dynamodb Preset Jest DynamoDB provides all required configuration to run your tests using DynamoDB.First, install @shelf/jest-dynamodbnpmYarnpnpmnpm install --save-dev @shelf/jest-dynamodbyarn add --dev @shelf/jest-dynamodbpnpm add --save-dev @shelf/jest-dynamodbSpecify preset in your Jest configuration:{ "preset": "@shelf/jest-dynamodb"}Create jest-dynamodb-config.js and define DynamoDB tablesSee Create Table APImodule.exports = { tables: [{   TableName: `files`,   KeySchema: [{AttributeName: 'id', KeyType: 'HASH'}],   AttributeDefinitions: [{AttributeName: 'id', AttributeType: 'S'}],   ProvisionedThroughput: {ReadCapacityUnits: 1, WriteCapacityUnits: 1},   },   // etc],};Configure DynamoDB clientconst {DocumentClient} = require('aws-sdk/clients/dynamodb');const isTest = process.env.JEST_WORKER_ID;const config = { convertEmptyValues: true, ...(isTest && {   endpoint: 'localhost:8000',   sslEnabled: false,   region: 'local-env', }),};const ddb = new DocumentClient(config);Write testsit('should insert item into table', async () => { await ddb   .put({TableName: 'files', Item: {id: '1', hello: 'world'}})   .promise(); const {Item} = await ddb.get({TableName: 'files', Key: {id: '1'}}).promise(); expect(Item).toEqual({   id: '1',   hello: 'world', });});There's no need to load any dependencies.See documentation for details.
DOM ManipulationAnother class of functions that is often considered difficult to test is

code that directly manipulates the DOM. Let's see how we can test the following snippet of jQuery code that listens to a click event, fetches some data asynchronously and sets the content of a span.displayUser.js'use strict';const $ = require('jquery');const fetchCurrentUser = require('./fetchCurrentUser.js');$('#button').click(() => { fetchCurrentUser(user => {   const loggedText = 'Logged ' + (user.loggedIn ? 'In' : 'Out'); $('#username').text(user.fullName + ' - ' + loggedText); });});Again, we create a test file in the __tests__/ folder:__tests__/displayUser-test.js'use strict';jest.mock('../fetchCurrentUser');test('displays a user after a click', () => { // Set up our document body document.body.innerHTML =   '<div>' +   ' <span id="username" />' +   ' <button id="button" />' +   '</div>'; // This module has a side-effect require('../displayUser'); const $ = require('jquery'); const fetchCurrentUser = require('../fetchCurrentUser'); // Tell the fetchCurrentUser mock function to automatically invoke // its callback with some data fetchCurrentUser.mockImplementation(cb => {   cb({   fullName: 'Johnny Cash',   loggedIn: true,   }); }); // Use jquery to emulate a click on our button $('#button').click(); // Assert that the fetchCurrentUser function was called, and that the // #username span's inner text was updated as we'd expect it to. expect(fetchCurrentUser).toBeCalled(); expect($('#username').text()).toBe('Johnny Cash - Logged In');});We are mocking fetchCurrentUser.js so that our test doesn't make a real network request but instead resolves to mock data locally. This ensures that our test can complete in milliseconds rather than seconds and guarantees a fast unit test iteration speed.Also, the function being tested adds an event listener on the #button DOM element, so we need to set up our DOM correctly for the test. jsdom and the jest-environment-jsdom package simulate a DOM environment as if you were in the browser. This means that every DOM API that we call can be observed in the same way it would be observed in a browser!To get started with the JSDOM test environment, the jest-environment-jsdom package must be installed if it's not already:npmYarnpnpmnpm install --save-dev jest-environment-jsdomyarn add --dev jest-environment-jsdompnpm add --save-dev jest-environment-jsdomThe code for this example is available at examples/jquery.

Watch PluginsThe Jest watch plugin system provides a way to hook into specific parts of Jest and to define watch mode menu prompts that execute code on key press. Combined, these features allow you to develop interactive experiences custom for your workflow.Watch Plugin Interface class MyWatchPlugin { // Add hooks to Jest lifecycle events apply(jestHooks) {} // Get the prompt information for interactive plugins getUsageInfo(globalConfig) {} // Executed when the key from `getUsageInfo` is input run(globalConfig, updateConfigAndRun) {}}Hooking into Jest To connect your watch plugin to Jest, add its path under watchPlugins in your Jest configuration:jest.config.jsmodule.exports = { // ... watchPlugins: ['path/to/yourWatchPlugin'],};Custom watch plugins can add hooks to Jest events. These hooks can be added either with or without having an interactive key in the watch mode menu.apply(jestHooks) Jest hooks can be attached by implementing the apply method. This method receives a jestHooks argument that allows the plugin to hook into specific parts of the lifecycle of a test run.class MyWatchPlugin { apply(jestHooks) {}}Below are the hooks available in Jest.jestHooks.shouldRunTestSuite(testSuiteInfo) Returns a boolean (or Promise<boolean> for handling asynchronous operations) to specify if a test should be run or not.For example:class MyWatchPlugin { apply(jestHooks)

{   jestHooks.shouldRunTestSuite(testSuiteInfo => {   return testSuiteInfo.testPath.includes('my-keyword');   });   // or a promise jestHooks.shouldRunTestSuite(testSuiteInfo => {   return Promise.resolve(testSuiteInfo.testPath.includes('my-keyword'));   }); }} jestHooks.onTestRunComplete(results) Gets called at the end of every test run. It has the test results as an argument.For example:class MyWatchPlugin { apply(jestHooks) {   jestHooks.onTestRunComplete(results => {   this._hasSnapshotFailure = results.snapshot.failure;   }); }}jestHooks.onFileChange({projects}) Gets called whenever there is a change in the file systemprojects: Array<config: ProjectConfig, testPaths: Array<string>: Includes all the test paths that Jest is watching.For example:class MyWatchPlugin { apply(jestHooks) {   jestHooks.onFileChange(({projects}) => {   this._projects = projects;   }); }}Watch Menu Integration Custom watch plugins can also add or override functionality to the watch menu by specifying a key/prompt pair in getUsageInfo method and a run method for the execution of the key.getUsageInfo(globalConfig) To add a key to the watch menu, implement the getUsageInfo method, returning a key and the prompt:class MyWatchPlugin { getUsageInfo(globalConfig) {   return {   key: 's',   prompt: 'do something',   }; }}This will add a line in the watch mode menu (› Press s to do something.)Watch Usage › Press p to filter by a filename regex pattern. › Press t to filter by a test name regex pattern. › Press q to quit watch mode. › Press s to do something. // <-- This is our plugin › Press Enter to trigger a test run.noteIf the key for your plugin already exists as a default key, your plugin will override that key.run(globalConfig, updateConfigAndRun) To handle key press events from the key returned by getUsageInfo, you can implement the run method. This method returns a Promise<boolean> that can be resolved when the plugin wants to return control to Jest. The boolean specifies if Jest should rerun the tests after it gets the control back.globalConfig: A representation of Jest's current global configurationupdateConfigAndRun: Allows you to trigger a test run while the interactive plugin is running.class MyWatchPlugin { run(globalConfig, updateConfigAndRun) {   // do something. }}noteIf you do call updateConfigAndRun, your run method should not resolve to a truthy value, as that would trigger a double-run.Authorized configuration keys For stability and safety reasons, only part of the global configuration keys can be updated with updateConfigAndRun. The current white list is as follows:bailchangedSincecollectCoveragecollectCoverageFromcoverageDirectorycoverageReportersnotifynotifyModeonlyFailuresreporterstestNamePatterntestPathPatternupdateSnapshotverboseCustomization Plugins can be customized via your Jest configuration.jest.config.jsmodule.exports = { // ... watchPlugins: [['path/to/yourWatchPlugin',   {   key: 'k', // <- your custom key   prompt: 'show a custom prompt',   },   ],],};Recommended config names:key: Modifies the plugin key.prompt: Allows user to customize the text in the plugin prompt.If the user provided a custom configuration, it will be passed as an argument to the plugin constructor.class MyWatchPlugin { constructor({config}) {}}Choosing a good key Jest allows third-party plugins to override some of its built-in feature keys, but not all. Specifically, the following keys are not overwritable :c (clears filter patterns)i (updates non-matching snapshots interactively)q (quits)u (updates all non-matching snapshots)w (displays watch mode usage / available actions)The following keys for built-in functionality can be

overwritten :p (test filename pattern)t (test name pattern)Any key not used by built-in functionality can be claimed, as you would expect. Try to avoid using keys that are difficult to obtain on various keyboards (e.g. é, €), or not visible by default (e.g. many Mac keyboards do not have visual hints for characters such as |, \, [, etc.)When a conflict happens Should your plugin attempt to overwrite a reserved key, Jest will error out with a descriptive message, something like:Watch plugin YourFaultyPlugin attempted to register key `q`, that is reserved internally for quitting watch mode. Please change the configuration key for this plugin.Third-party plugins are also forbidden to overwrite a key reserved already by another third-party plugin present earlier in the configured plugins list (watchPlugins array setting). When this happens, you'll also get an error message that tries to help you fix that:Watch plugins YourFaultyPlugin and TheirFaultyPlugin both attempted to register key `x`. Please change the key configuration for one of the conflicting plugins to avoid overlap.
Migrating to JestIf you'd like to try out Jest with an existing codebase, there are a number of ways to convert to Jest:If you are using Jasmine, or a Jasmine like API (for example Mocha), Jest should be mostly compatible, which makes it less complicated to migrate to.If you are using AVA, Expect.js (by Automattic), Jasmine, Mocha, proxyquire, Should.js or Tape you can automatically migrate with Jest Codemods (see below).If you like chai, you can upgrade to Jest and continue using chai. However, we recommend trying out Jest's assertions and their failure messages. Jest Codemods can migrate from chai (see below).jest-codemods If you are using AVA, Chai, Expect.js (by Automattic), Jasmine, Mocha, proxyquire, Should.js, Tape, or Sinon you can use the third-party jest-codemods to do most of the dirty migration work. It runs a code transformation on your codebase using jscodeshift.To transform your existing tests, navigate to the project containing the tests and run:npx jest-codemodsMore information can be found at https://github.com/skovhus/jest-codemods.
TroubleshootingUh oh, something went wrong? Use this guide to resolve issues with Jest.Tests are Failing and You Don't Know Why Try using the debugging support built into Node. Place a debugger; statement in any of your tests, and then, in your project's directory, run:node --inspect-brk node_modules/.bin/jest --runInBand [any other arguments here]or on Windowsnode --inspect-brk ./node_modules/jest/bin/jest.js --runInBand [any other arguments here]This will run Jest in a Node process that an external debugger can connect to. Note that the process will pause until the debugger has connected to it.To debug in Google Chrome (or any Chromium-based browser), open your browser and go to chrome://inspect and click on "Open Dedicated DevTools for Node", which will give you a list of available node instances you can connect to. Click on the address displayed in the terminal (usually something like localhost:9229) after running the above command, and you will be able to debug Jest using Chrome's DevTools.The Chrome Developer Tools will be displayed, and a breakpoint will be set at the first line of the Jest CLI script (this is done to give you time to open the developer tools and to prevent Jest from executing before you have time to do so). Click the button that looks like a "play" button in the upper right hand side of the screen to continue execution. When Jest executes the test that contains the debugger statement, execution will pause and you can examine the current scope and call stack.noteThe --runInBand cli option makes sure Jest runs the test in the same process rather than spawning processes for individual tests. Normally Jest parallelizes test runs across

processes but it is hard to debug many processes at the same time.Debugging in VS Code There are multiple ways to debug Jest tests with Visual Studio Code's built-in debugger.To attach the built-in debugger, run your tests as aforementioned:node --inspect-brk node_modules/.bin/jest --runInBand [any other arguments here]or on Windowsnode --inspect-brk ./node_modules/jest/bin/jest.js --runInBand [any other arguments here]Then attach VS Code's debugger using the following launch.json config:{ "version": "0.2.0", "configurations": [{   "type": "node",   "request": "attach",   "name": "Attach",   "port": 9229   }]}To automatically launch and attach to a process running your tests, use the following configuration:{ "version": "0.2.0", "configurations": [{   "name": "Debug Jest Tests",   "type": "node",   "request": "launch",   "runtimeArgs": ["--inspect-brk",   "${workspaceRoot}/node_modules/.bin/jest",   "--runInBand"   ],   "console": "integratedTerminal",   "internalConsoleOptions": "neverOpen"   }]}or the following for Windows:{ "version": "0.2.0", "configurations": [{   "name": "Debug Jest Tests",   "type": "node",   "request": "launch",   "runtimeArgs": ["--inspect-brk",   "${workspaceRoot}/node_modules/jest/bin/jest.js",   "--runInBand"   ],   "console": "integratedTerminal",   "internalConsoleOptions": "neverOpen"   }]}If you are using Facebook's create-react-app, you can debug your Jest tests with the following configuration:{ "version": "0.2.0", "configurations": [{   "name": "Debug CRA Tests",   "type": "node",   "request": "launch",   "runtimeExecutable": "${workspaceRoot}/node_modules/.bin/react-scripts",   "args": ["test",   "--runInBand",   "--no-cache",   "--env=jsdom",   "--watchAll=false"   ],   "cwd": "${workspaceRoot}",   "console": "integratedTerminal",   "internalConsoleOptions": "neverOpen"   }]}More information on Node debugging can be found here.Debugging in WebStorm WebStorm has built-in support for Jest. Read Testing With Jest in WebStorm to learn more.Caching Issues The transform script was changed or Babel was updated and the changes aren't being recognized by Jest?Retry with --no-cache. Jest caches transformed module files to speed up test execution. If you are using your own custom transformer, consider adding a getCacheKey function to it: getCacheKey in Relay.Unresolved Promises If a promise doesn't resolve at all, this error might be thrown:- Error: Timeout - Async callback was not invoked within timeout specified by jasmine.DEFAULT_TIMEOUT_INTERVAL.`Most commonly this is being caused by conflicting Promise implementations. Consider replacing the global promise implementation with your own, for example globalThis.Promise = jest.requireActual('promise'); and/or consolidate the used Promise libraries to a single one.If your test is long running, you may want to consider to increase the timeout by calling jest.setTimeoutjest.setTimeout(10000); // 10 second timeoutWatchman Issues Try running Jest with --no-watchman or set the watchman configuration option to false.Also see watchman troubleshooting.Tests are Extremely Slow on Docker and/or Continuous Integration (CI) server. While Jest is most of the time extremely fast on modern multi-core computers with fast SSDs, it may be slow on certain setups as our users have discovered.Based on the findings, one way to mitigate this issue and improve the speed by up to 50% is to run tests sequentially.In order to do this you can run tests in the same thread using --runInBand:npmYarnpnpm# Using Jest CLIjest --runInBand# Using your package manager's `test` script (e.g. with create-react-app)npm test -- --runInBand# Using Jest CLIjest --runInBand# Using your package manager's

`test` script (e.g. with create-react-app)yarn test --runInBand# Using Jest CLIjest --runInBand# Using your package manager's `test` script (e.g. with create-react-app)pnpm test -- --runInBandAnother alternative to expediting test execution time on Continuous Integration Servers such as Travis-CI is to set the max worker pool to ~4. Specifically on Travis-CI, this can reduce test execution time in half. Note: The Travis CI free plan available for open source projects only includes 2 CPU cores.npmYarnpnpm# Using Jest CLIjest --maxWorkers=4# Using your package manager's `test` script (e.g. with create-react-app)npm test -- --maxWorkers=4# Using Jest CLIjest --maxWorkers=4# Using your package manager's `test` script (e.g. with create-react-app)yarn test --maxWorkers=4# Using Jest CLIjest --maxWorkers=4# Using your package manager's `test` script (e.g. with create-react-app)pnpm test -- --maxWorkers=4If you use GitHub Actions, you can use github-actions-cpu-cores to detect number of CPUs, and pass that to Jest.- name: Get number of CPU cores id: cpu-cores uses: SimenB/github-actions-cpu-cores@v1- name: run tests run: yarn jest --max-workers ${{ steps.cpu-cores.outputs.count }}Another thing you can do is use the shard flag to parallelize the test run across multiple machines.coveragePathIgnorePatterns seems to not have any effect. Make sure you are not using the babel-plugin-istanbul plugin. Jest wraps Istanbul, and therefore also tells Istanbul what files to instrument with coverage collection. When using babel-plugin-istanbul, every file that is processed by Babel will have coverage collection code, hence it is not being ignored by coveragePathIgnorePatterns.Defining Tests Tests must be defined synchronously for Jest to be able to collect your tests.As an example to show why this is the case, imagine we wrote a test like so:// Don't do this it will not worksetTimeout(() => { it('passes', () => expect(1).toBe(1));}, 0);When Jest runs your test to collect the tests it will not find any because we have set the definition to happen asynchronously on the next tick of the event loop. This means when you are using test.each you cannot set the table asynchronously within a beforeEach / beforeAll.Still unresolved? See Help.
ArchitectureIf you are interested in learning more about how Jest works, understand its architecture, and how Jest is split up into individual reusable packages, check out this video:If you'd like to learn how to build a testing framework like Jest from scratch, check out this video:There is also a written guide you can follow. It teaches the fundamental concepts of Jest and explains how various parts of Jest can be used to compose a custom testing framework.
Testing React AppsAt Facebook, we use Jest to test React applications.Setup Setup with Create React App If you are new to React, we recommend using Create React App. It is ready to use and ships with Jest! You will only need to add react-test-renderer for rendering snapshots.RunnpmYarnpnpmnpm install --save-dev react-test-rendereryarn add --dev react-test-rendererpnpm add --save-dev react-test-rendererSetup without Create React App If you have an existing application you'll need to install a few packages to make everything work well together. We are using the babel-jest package and the react babel preset to transform our code inside of the test environment. Also see using babel.RunnpmYarnpnpmnpm install --save-dev jest babel-jest @babel/preset-env @babel/preset-react react-test-rendereryarn add --dev jest babel-jest @babel/preset-env @babel/preset-react react-test-rendererpnpm add --save-dev jest babel-jest @babel/preset-env @babel/preset-react react-test-rendererYour package.json should

look something like this (where <current-version> is the actual latest version number for the package). Please add the scripts and jest configuration entries:{ "dependencies": {   "react": "<current-version>",   "react-dom": "<current-version>" }, "devDependencies": {   "@babel/preset-env": "<current-version>",   "@babel/preset-react": "<current-version>",   "babel-jest": "<current-version>",   "jest": "<current-version>",   "react-test-renderer": "<current-version>" }, "scripts": {   "test": "jest" }} babel.config.jsmodule.exports = { presets: ['@babel/preset-env',   ['@babel/preset-react', {runtime: 'automatic'}],],};And you're good to go!Snapshot Testing Let's create a snapshot test for a Link component that renders hyperlinks:Link.jsimport {useState} from 'react';const STATUS = { HOVERED: 'hovered', NORMAL: 'normal',};export default function Link({page, children}) { const [status, setStatus] = useState(STATUS.NORMAL); const onMouseEnter = () => {   setStatus(STATUS.HOVERED); }; const onMouseLeave = () => {   setStatus(STATUS.NORMAL); }; return (<a   className={status} href={page || '#'}   onMouseEnter={onMouseEnter} onMouseLeave={onMouseLeave}   >   {children}   </a>);}noteExamples are using Function components, but Class components can be tested in the same way. See React: Function and Class Components. Reminders that with Class components, we expect Jest to be used to test props and not methods directly.Now let's use React's test renderer and Jest's snapshot feature to interact with the component and capture the rendered output and create a snapshot file:Link.test.jsimport renderer from 'react-test-renderer';import Link from '../Link';it('changes the class when hovered', () => { const component = renderer.create(<Link page="http://www.facebook.com">Facebook</Link>,); let tree = component.toJSON(); expect(tree).toMatchSnapshot(); // manually trigger the callback renderer.act(() => {   tree.props.onMouseEnter(); }); // re-rendering tree = component.toJSON(); expect(tree).toMatchSnapshot(); // manually trigger the callback renderer.act(() => {   tree.props.onMouseLeave(); }); // re-rendering tree = component.toJSON(); expect(tree).toMatchSnapshot();});When you run yarn test or jest, this will produce an output file like this:__tests__/__snapshots__/Link.test.js.snapexports[`changes the class when hovered 1`] = `<a  className="normal"  href="http://www.facebook.com"  onMouseEnter={[Function]} onMouseLeave={[Function]}>  Facebook</a>`;exports[`changes the class when hovered 2`] = `<a  className="hovered"  href="http://www.facebook.com" onMouseEnter={[Function]}  onMouseLeave={[Function]}>  Facebook</a>`;exports[`changes the class when hovered 3`] = `<a  className="normal" href="http://www.facebook.com"  onMouseEnter={[Function]} onMouseLeave={[Function]}>  Facebook</a>`;The next time you run the tests, the rendered output will be compared to the previously created snapshot. The snapshot should be committed along with code changes. When a snapshot test fails, you need to inspect whether it is an intended or unintended change. If the change is expected you can invoke Jest with jest -u to overwrite the existing snapshot.The code for this example is available at examples/snapshot.Snapshot Testing with Mocks, Enzyme and React 16+ There's a caveat around snapshot testing when using Enzyme and React 16+. If you mock out a module using the following style:jest.mock('../SomeDirectory/SomeComponent', () => 'SomeComponent');Then you will see warnings in the console:Warning: <SomeComponent /> is using uppercase HTML. Always use lowercase HTML tags in React.# Or:Warning: The tag <SomeComponent> is unrecognized in this browser. If you meant to render a React component, start its name with an uppercase letter.React 16 triggers these warnings due to how it checks element types, and the mocked module fails these checks. Your options are:Render as text. This way you won't see the props passed to the mock component in the snapshot, but it's straightforward:jest.mock('./SomeComponent', () => () => 'SomeComponent');Render as a custom element. DOM "custom elements" aren't checked for anything and shouldn't fire warnings. They are lowercase and have a dash in the name.jest.mock('./Widget', () => () => <mock-widget />);Use react-test-renderer. The test renderer doesn't care about element types and will happily accept e.g. SomeComponent. You could check snapshots using the test renderer, and check component behavior separately using Enzyme.Disable warnings all together (should be done in your jest setup file):jest.mock('fbjs/lib/warning', () => require('fbjs/lib/emptyFunction'));This shouldn't normally be your option of choice as useful warnings could be lost. However, in some cases, for example when testing react-native's components we are rendering react-native tags into the DOM and many warnings are irrelevant. Another option is to swizzle the console.warn and suppress specific warnings.DOM Testing If you'd like to assert, and manipulate your rendered components you can use react-testing-library, Enzyme, or React's TestUtils. The following two examples use react-testing-library and Enzyme.react-testing-library npmYarnpnpmnpm install --save-dev @testing-library/reactyarn add --dev @testing-library/reactpnpm add --save-dev @testing-library/reactLet's implement a checkbox which swaps between two labels:CheckboxWithLabel.jsimport {useState} from 'react';export default function CheckboxWithLabel({labelOn, labelOff}) { const [isChecked, setIsChecked] = useState(false); const onChange = () => {   setIsChecked(!isChecked); }; return (<label>   <input type="checkbox" checked={isChecked} onChange={onChange} />   {isChecked ? labelOn : labelOff}   </label>);}__tests__/CheckboxWithLabel-test.jsimport {cleanup, fireEvent, render} from '@testing-library/react';import CheckboxWithLabel from '../CheckboxWithLabel';// Note: running cleanup afterEach is done automatically for you in @testing-library/react@9.0.0 or higher// unmount and cleanup DOM after the test is finished.afterEach(cleanup);it('CheckboxWithLabel changes the text after click', () => { const {queryByLabelText, getByLabelText} = render(<CheckboxWithLabel labelOn="On" labelOff="Off" />,); expect(queryByLabelText(/off/i)).toBeTruthy(); fireEvent.click(getByLabelText(/off/i)); expect(queryByLabelText(/on/i)).toBeTruthy();});The code for this example is available at examples/react-testing-library.Enzyme npmYarnpnpmnpm install --save-dev enzymeyarn add --dev enzymepnpm add --save-dev enzymeIf you are using a React version below 15.5.0, you will also need to install react-addons-test-utils.Let's rewrite the test from above using Enzyme instead of react-testing-library. We use Enzyme's shallow renderer in this example.__tests__/CheckboxWithLabel-test.jsimport Enzyme, {shallow} from 'enzyme';import Adapter from 'enzyme-adapter-react-16';import CheckboxWithLabel from '../CheckboxWithLabel';Enzyme.configure({adapter: new Adapter()});it('CheckboxWithLabel changes the text after click', () => { // Render a checkbox with label in the document const checkbox = shallow(<CheckboxWithLabel labelOn="On" labelOff="Off" />); expect(checkbox.text()).toBe('Off'); checkbox.find('input').simulate('change'); expect(checkbox.text()).toBe('On');});Custom

transformers If you need more advanced functionality, you can also build your own transformer. Instead of using babel-jest, here is an example of using @babel/core:custom-transformer.js'use strict';const {transform} = require('@babel/core');const jestPreset = require('babel-preset-jest');module.exports = { process(src, filename) {   const result = transform(src, {   filename,   presets: [jestPreset],   });   return result || src; },};Don't forget to install the @babel/core and babel-preset-jest packages for this example to work.To make this work with Jest you need to update your Jest configuration with this: "transform": {"\\.js$": "path/to/custom-transformer.js"}.If you'd like to build a transformer with babel support, you can also use babel-jest to compose one and pass in your custom configuration options:const babelJest = require('babel-jest');module.exports = babelJest.createTransformer({ presets: ['my-custom-preset'],});See dedicated docs for more details.
Testing React Native AppsAt Facebook, we use Jest to test React Native applications.Get a deeper insight into testing a working React Native app example by reading the following series: Part 1: Jest – Snapshot come into play and Part 2: Jest – Redux Snapshots for your Actions and Reducers.Setup Starting from react-native version 0.38, a Jest setup is included by default when running react-native init. The following configuration should be automatically added to your package.json file: { "scripts": {   "test": "jest" }, "jest": {   "preset": "react-native" }}Run yarn test to run tests with Jest.tipIf you are upgrading your react-native application and previously used the jest-react-native preset, remove the dependency from your package.json file and change the preset to react-native instead.Snapshot Test Let's create a snapshot test for a small intro component with a few views and text components and some styles:Intro.jsimport React, {Component} from 'react';import {StyleSheet, Text, View} from 'react-native';class Intro extends Component { render() {   return (<View style={styles.container}>   <Text style={styles.welcome}>Welcome to React Native!</Text>   <Text style={styles.instructions}>   This is a React Native snapshot test.   </Text>   </View>   ); }}const styles = StyleSheet.create({ container: {   alignItems: 'center',   backgroundColor: '#F5FCFF',   flex: 1,   justifyContent: 'center', }, instructions: {   color: '#333333',   marginBottom: 5,   textAlign: 'center', }, welcome: {   fontSize: 20,   margin: 10,   textAlign: 'center', },});export default Intro;Now let's use React's test renderer and Jest's snapshot feature to interact with the component and capture the rendered output and create a snapshot file:__tests__/Intro-test.jsimport React from 'react';import renderer from 'react-test-renderer';import Intro from '../Intro';test('renders correctly', () => { const tree = renderer.create(<Intro />).toJSON(); expect(tree).toMatchSnapshot();});When you run yarn test or jest, this will produce an output file like this:__tests__/__snapshots__/Intro-test.js.snapexports[`Intro renders correctly 1`] = `<View style={   Object {   "alignItems": "center",   "backgroundColor": "#F5FCFF",   "flex": 1,   "justifyContent": "center",   } }> <Text   style={   Object {   "fontSize": 20,   "margin": 10,   "textAlign": "center",   }   } >   Welcome to React Native! </Text> <Text   style={   Object {   "color": "#333333",   "marginBottom": 5,   "textAlign": "center",   }   }>   This is a React Native snapshot test. </Text></View>`;The next time you run the tests, the rendered output will be compared to the previously created snapshot. The snapshot should be committed along with code changes. When a snapshot test fails, you need to inspect whether it is an intended or unintended change. If the change is expected you can

invoke Jest with jest -u to overwrite the existing snapshot.The code for this example is available at examples/react-native.Preset configuration The preset sets up the environment and is very opinionated and based on what we found to be useful at Facebook. All of the configuration options can be overwritten just as they can be customized when no preset is used.Environment react-native ships with a Jest preset, so the jest.preset field of your package.json should point to react-native. The preset is a node environment that mimics the environment of a React Native app. Because it doesn't load any DOM or browser APIs, it greatly improves Jest's startup time.transformIgnorePatterns customization The transformIgnorePatterns option can be used to specify which files shall be transformed by Babel. Many react-native npm modules unfortunately don't pre-compile their source code before publishing.By default the jest-react-native preset only processes the project's own source files and react-native. If you have npm dependencies that have to be transformed you can customize this configuration option by including modules other than react-native by grouping them and separating them with the | operator:{ "transformIgnorePatterns": ["node_modules/(?!(react-native|my-project|react-native-button)/)"]}You can test which paths would match (and thus be excluded from transformation) with a tool like this.transformIgnorePatterns will exclude a file from transformation if the path matches against any pattern provided. Splitting into multiple patterns could therefore have unintended results if you are not careful. In the example below, the exclusion (also known as a negative lookahead assertion) for foo and bar cancel each other out: { "transformIgnorePatterns": ["node_modules/(?!foo/)", "node_modules/(?!bar/)"] // not what you want}setupFiles If you'd like to provide additional configuration for every test file, the setupFiles configuration option can be used to specify setup scripts.moduleNameMapper The moduleNameMapper can be used to map a module path to a different module. By default the preset maps all images to an image stub module but if a module cannot be found this configuration option can help: { "moduleNameMapper": {   "my-module.js": "<rootDir>/path/to/my-module.js" }}Tips Mock native modules using jest.mock The Jest preset built into react-native comes with a few default mocks that are applied on a react-native repository. However, some react-native components or third party components rely on native code to be rendered. In such cases, Jest's manual mocking system can help to mock out the underlying implementation.For example, if your code depends on a third party native video component called react-native-video you might want to stub it out with a manual mock like this:jest.mock('react-native-video', () => 'Video');This will render the component as <Video {...props} /> with all of its props in the snapshot output. See also caveats around Enzyme and React 16.Sometimes you need to provide a more complex manual mock. For example if you'd like to forward the prop types or static fields of a native component to a mock, you can return a different React component from a mock through this helper from jest-react-native:jest.mock('path/to/MyNativeComponent', () => { const mockComponent = require('react-native/jest/mockComponent'); return mockComponent('path/to/MyNativeComponent');});Or if you'd like to create your own manual mock, you can do something like this:jest.mock('Text', () => { const RealComponent = jest.requireActual('Text'); const React = require('react'); class Text extends React.Component {   render() {   return React.createElement('Text', this.props, this.props.children);   } } Text.propTypes = RealComponent.propTypes;

return Text;});In other cases you may want to mock a native module that isn't a React component. The same technique can be applied. We recommend inspecting the native module's source code and logging the module when running a react native app on a real device and then modeling a manual mock after the real module.If you end up mocking the same modules over and over it is recommended to define these mocks in a separate file and add it to the list of setupFiles.
Testing Web FrameworksJest is a universal testing platform, with the ability to adapt to any JavaScript library or framework. In this section, we'd like to link to community posts and articles about integrating Jest into popular JS libraries.React Testing ReactJS components with Jest by Kent C. Dodds (@kentcdodds)Vue.js Testing Vue.js components with Jest by Alex Jover Morales (@alexjoverm)Jest for all: Episode 1 — Vue.js by Cristian Carlesso (@kentaromiura)AngularJS Testing an AngularJS app with Jest by Matthieu Lux (@Swiip)Running AngularJS Tests with Jest by Ben Brandt (@benjaminbrandt)AngularJS Unit Tests with Jest Actions (Traditional Chinese) by Chris Wang (@dwatow)Angular Testing Angular faster with Jest by Micha " –W'¦6† Ba (@thymikee)MobX How to Test React and MobX with Jest by Will Stern (@willsterndev)Redux Writing Tests by Redux docsExpress.js How to test Express.js with Jest and Supertest by Albert Gao (@albertgao)GatsbyJS Unit Testing by GatsbyJS docsHapi.js Testing Hapi.js With Jest by NiralarNext.js Jest and React Testing Library by Next.js docs