

Running Tests

Note: this feature is available with `react-scripts@0.3.0` and higher.

Read the migration guide to learn how to enable it in older projects!

Create React App uses Jest as its test runner. To prepare for this integration, we did a major revamp of Jest so if you heard bad things about it years ago, give it another try.

Jest is a Node-based runner. This means that the tests always run in a Node environment and not in a real browser. This lets us enable fast iteration speed and prevent flakiness.

While Jest provides browser globals such as `window` thanks to `jsdom`, they are only approximations of the real browser behavior. Jest is intended to be used for unit tests of your logic and your components rather than the DOM quirks.

We recommend that you use a separate tool for browser end-to-end tests if you need them. They are beyond the scope of Create React App.

Filename Conventions

Jest will look for test files with any of the following popular naming conventions:

- Files with `.js` suffix in `__tests__` folders.
- Files with `.test.js` suffix.
- Files with `.spec.js` suffix.

The `.test.js` / `.spec.js` files (or the `__tests__` folders) can be located at any depth under the `src` top level folder.

We recommend to put the test files (or `__tests__` folders) next to the code they are testing so that relative imports appear shorter. For example, if `App.test.js` and `App.js` are in the same folder, the test only needs to `import App from './App'` instead of a long relative path. Collocation also helps find tests more quickly in larger projects.

Command Line Interface

When you run `npm test`, Jest will launch in watch mode*. Every time you save a file, it will re-run the tests, like how `npm start` recompiles the code.

The watcher includes an interactive command-line interface with the ability to run all tests, or focus on a search pattern. It is designed this way so that you can keep it open and enjoy fast re-runs. You can learn the commands from the “Watch Usage” note that the watcher prints after every run:

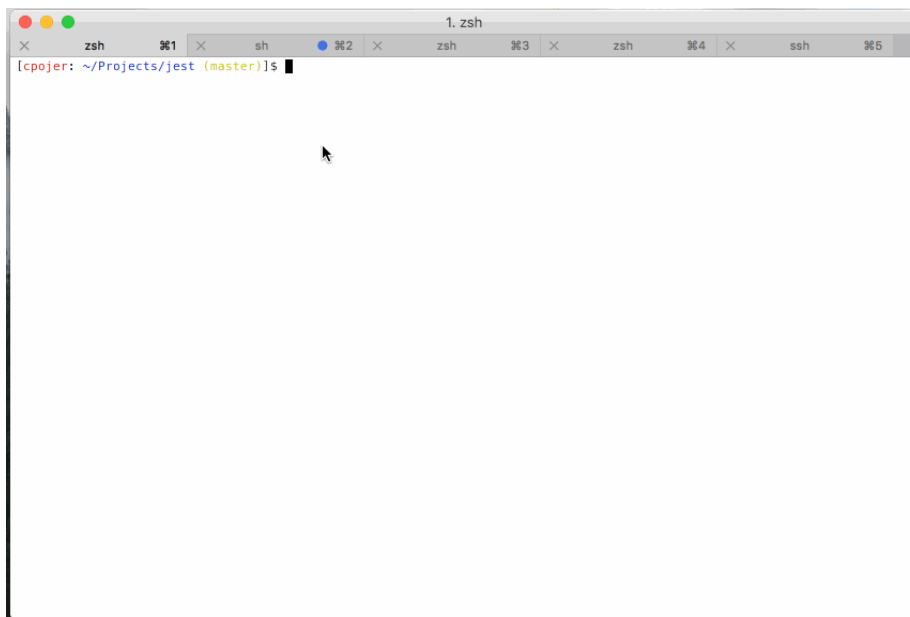


Figure 1: Jest watch mode

*Although we recommend running your tests in watch mode during development, you can disable this behavior by passing in the `--watchAll=false` flag. In most CI environments, this is handled for you (see [On CI servers](#)).

Version Control Integration

By default, when you run `npm test`, Jest will only run the tests related to files changed since the last commit. This is an optimization designed to make your tests run fast regardless of how many tests you have. However it assumes that you don’t often commit the code that doesn’t pass the tests.

Jest will always explicitly mention that it only ran tests related to the files changed since the last commit. You can also press `a` in the watch mode to force Jest to run all tests.

Jest will always run all tests on a continuous integration server or if the project is not inside a Git or Mercurial repository.

Writing Tests

To create tests, add `it()` (or `test()`) blocks with the name of the test and its code. You may optionally wrap them in `describe()` blocks for logical grouping but this is neither required nor recommended.

Jest provides a built-in `expect()` global function for making assertions. A basic test could look like this:

```
import sum from './sum';

it('sums numbers', () => {
  expect(sum(1, 2)).toEqual(3);
  expect(sum(2, 2)).toEqual(4);
});
```

All `expect()` matchers supported by Jest are extensively documented [here](#).

You can also use `jest.fn()` and `expect(fn).toBeCalled()` to create “spies” or mock functions.

Testing Components

There is a broad spectrum of component testing techniques. They range from a “smoke test” verifying that a component renders without throwing, to shallow rendering and testing some of the output, to full rendering and testing component lifecycle and state changes.

Different projects choose different testing tradeoffs based on how often components change, and how much logic they contain. If you haven’t decided on a testing strategy yet, we recommend that you start with creating basic smoke tests for your components:

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<App />, div);
});
```

This test mounts a component and makes sure that it didn’t throw during rendering. Tests like this provide a lot of value with very little effort so they are great as a starting point, and this is the test you will find in `src/App.test.js`.

When you encounter bugs caused by changing components, you will gain a deeper insight into which parts of them are worth testing in your application. This might be a good time to introduce more specific tests asserting specific expected output or behavior.

React Testing Library

If you'd like to test components in isolation from the child components they render, we recommend using `react-testing-library`. `react-testing-library` is a library for testing React components in a way that resembles the way the components are used by end users. It is well suited for unit, integration, and end-to-end testing of React components and applications. It works more directly with DOM nodes, and therefore it's recommended to use with `jest-dom` for improved assertions.

To install `react-testing-library` and `jest-dom`, you can run:

```
npm install --save @testing-library/react @testing-library/jest-dom
```

Alternatively you may use yarn:

```
yarn add @testing-library/react @testing-library/jest-dom
```

If you want to avoid boilerplate in your test files, you can create a `src/setupTests.js` file:

```
// react-testing-library renders your components to document.body,  
// this adds jest-dom's custom assertions  
import '@testing-library/jest-dom';
```

Here's an example of using `react-testing-library` and `jest-dom` for testing that the `<App />` component renders "Learn React".

```
import React from 'react';  
import { render, screen } from '@testing-library/react';  
import App from './App';  
  
it('renders welcome message', () => {  
  render(<App />);  
  expect(screen.getByText('Learn React')).toBeInTheDocument();  
});
```

Learn more about the utilities provided by `react-testing-library` to facilitate testing asynchronous interactions as well as selecting form elements from the `react-testing-library` documentation and examples.

Using Third Party Assertion Libraries

We recommend that you use `expect()` for assertions and `jest.fn()` for spies. If you are having issues with them please file those against Jest, and we'll fix

them. We intend to keep making them better for React, supporting, for example, pretty-printing React elements as JSX.

However, if you are used to other libraries, such as Chai and Sinon, or if you have existing code using them that you'd like to port over, you can import them normally like this:

```
import sinon from 'sinon';
import { expect } from 'chai';
```

and then use them in your tests like you normally do.

Initializing Test Environment

Note: this feature is available with `react-scripts@0.4.0` and higher.

If your app uses a browser API that you need to mock in your tests or if you need a global setup before running your tests, add a `src/setupTests.js` to your project. It will be automatically executed before running your tests.

For example:

src/setupTests.js

```
const localStorageMock = {
  getItem: jest.fn(),
  setItem: jest.fn(),
  removeItem: jest.fn(),
  clear: jest.fn(),
};
global.localStorage = localStorageMock;
```

Note: Keep in mind that if you decide to “eject” before creating `src/setupTests.js`, the resulting `package.json` file won't contain any reference to it, so you should manually create the property `setupFilesAfterEnv` in the configuration for Jest, something like the following:

```
"jest": {
  // ...
  "setupFilesAfterEnv": ["<rootDir>/src/setupTests.js"]
}
```

Focusing and Excluding Tests

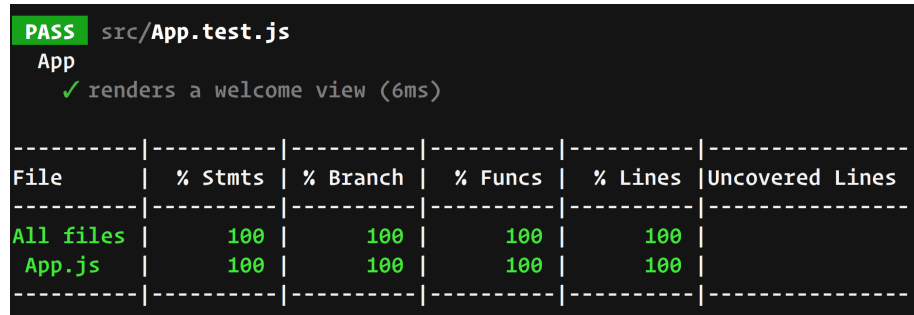
You can replace `it()` with `xit()` to temporarily exclude a test from being executed.

Similarly, `fit()` lets you focus on a specific test without running any other tests.

Coverage Reporting

Jest has an integrated coverage reporter that works well with ES6 and requires no configuration.

Run `npm test -- --coverage` (note extra `--` in the middle) to include a coverage report like this:



```
PASS src/App.test.js
App
  ✓ renders a welcome view (6ms)
```

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
All files	100	100	100	100	
App.js	100	100	100	100	

Figure 2: coverage report

Note that tests run much slower with coverage so it is recommended to run it separately from your normal workflow.

Configuration

The default configuration that Create React App uses for Jest can be overridden by adding any of the following supported keys to a Jest config in your `package.json`.

Supported overrides:

- `clearMocks`
- `collectCoverageFrom`
- `coveragePathIgnorePatterns`
- `coverageReporters`
- `coverageThreshold`
- `displayName`
- `extraGlobals`
- `globalSetup`
- `globalTeardown`
- `moduleNameMapper`
- `resetMocks`
- `resetModules`
- `restoreMocks`
- `snapshotSerializers`
- `testMatch`
- `transform`

- transformIgnorePatterns
- watchPathIgnorePatterns

Example package.json:

```
{
  "name": "your-package",
  "jest": {
    "collectCoverageFrom": [
      "src/**/*.{js,jsx,ts,tsx}",
      "!<rootDir>/node_modules/",
      "!<rootDir>/path/to/dir/"
    ],
    "coverageThreshold": {
      "global": {
        "branches": 90,
        "functions": 90,
        "lines": 90,
        "statements": 90
      }
    },
    "coverageReporters": ["text"],
    "snapshotSerializers": ["my-serializer-module"]
  }
}
```

Continuous Integration

By default `npm test` runs the watcher with interactive CLI. However, you can force it to run tests once and finish the process by setting an environment variable called `CI`.

When creating a build of your application with `npm run build` linter warnings are not checked by default. Like `npm test`, you can force the build to perform a linter warning check by setting the environment variable `CI`. If any warnings are encountered then the build fails.

Popular CI servers already set the environment variable `CI` by default but you can do this yourself too:

On CI servers

Travis CI

1. Following the Travis Getting started guide for syncing your GitHub repository with Travis. You may need to initialize some settings manually in your profile page.
2. Add a `.travis.yml` file to your git repository.

```

language: node_js
node_js:
  - 8
cache:
  directories:
    - node_modules
script:
  - npm run build
  - npm test

```

1. Trigger your first build with a git push.
2. Customize your Travis CI Build if needed.

CircleCI

Follow this article to set up CircleCI with a Create React App project.

On your own environment

Windows (cmd.exe)

```

set CI=true&&npm test
set CI=true&&npm run build

```

(Note: the lack of whitespace is intentional.)

Windows (Powershell)

```

($env:CI = "true") -and (npm test)
($env:CI = "true") -and (npm run build)

```

Linux, macOS (Bash)

```

CI=true npm test
CI=true npm run build

```

The test command will force Jest to run in CI-mode, and tests will only run once instead of launching the watcher.

For non-CI environments, you can pass the `--watchAll=false` flag to disable test-watching.

The build command will check for linter warnings and fail if any are found.

Disabling jsdom

If you know that none of your tests depend on jsdom, you can safely set `--env=node`, and your tests will run faster:


```
"scripts": {  
  "start": "react-scripts start",  
  "build": "react-scripts build",  
-  "test": "react-scripts test"  
+  "test": "react-scripts test --env=node"
```

To help you make up your mind, here is a list of APIs that **need jsdom**:

- Any browser globals like `window` and `document`
- `ReactDOM.render()`
- `TestUtils.renderIntoDocument()` (a shortcut for the above)
- `mount()` in Enzyme
- `render()` in React Testing Library

In contrast, **jsdom is not needed** for the following APIs:

- `TestUtils.createRenderer()` (shallow rendering)
- `shallow()` in Enzyme

Finally, jsdom is also not needed for snapshot testing.

Snapshot Testing

Snapshot testing is a feature of Jest that automatically generates text snapshots of your components and saves them on the disk so if the UI output changes, you get notified without manually writing any assertions on the component output. Read more about snapshot testing.

Editor Integration

If you use Visual Studio Code, there is a Jest extension which works with Create React App out of the box. This provides a lot of IDE-like features while using a text editor: showing the status of a test run with potential fail messages inline, starting and stopping the watcher automatically, and offering one-click snapshot updates.

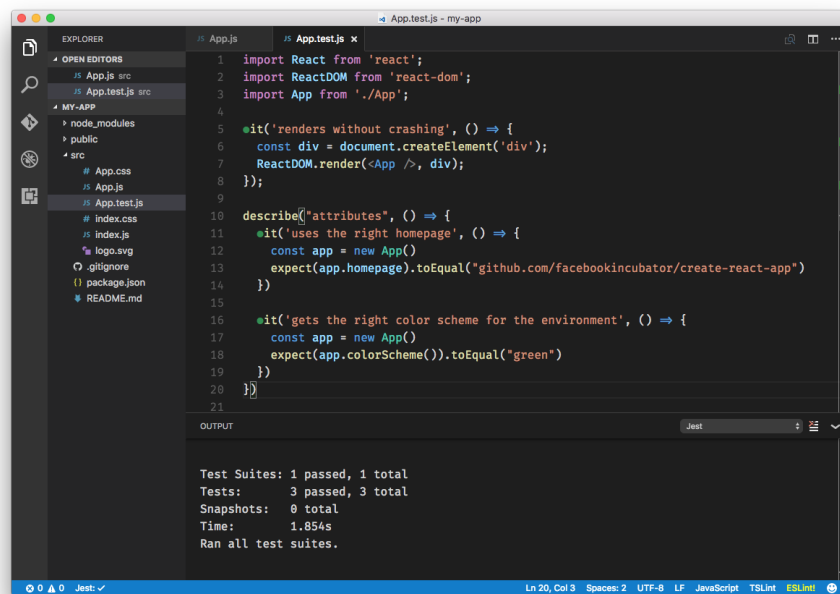


Figure 3: VS Code Jest Preview