Snapshot Testing

Snapshot 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.

> **NOTE**
>
> The 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.

> **INFO**
>
> More 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.

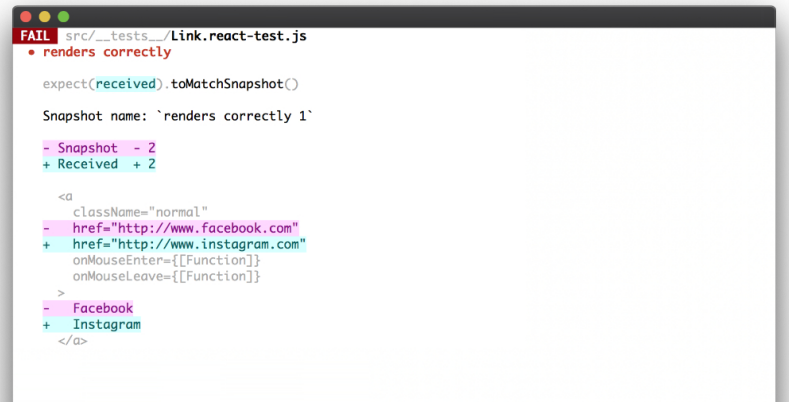
Updating Snapshots

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 address
it('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 --updateSnapshot
```

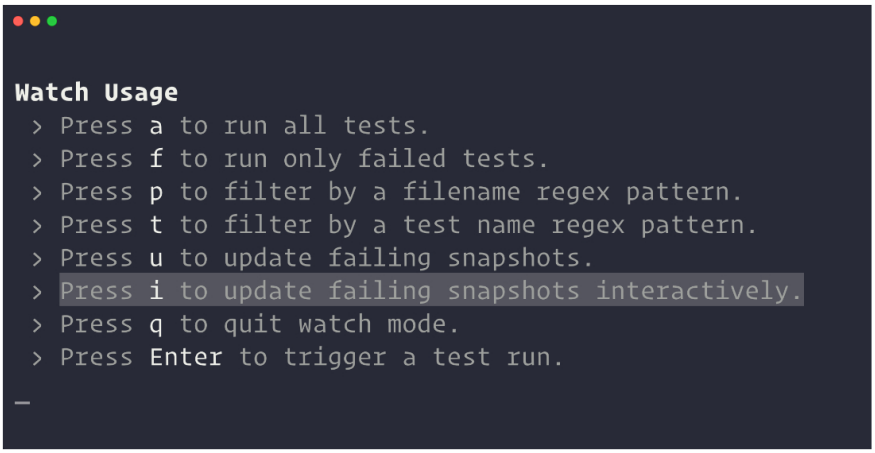
Go ahead and accept the changes by running the above command. You may also use the equivalent single-character `-u` flag to re-generate snapshots if you prefer. This will re-generate snapshot artifacts for all failing snapshot tests. If we had any additional failing snapshot tests due to an unintentional 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.

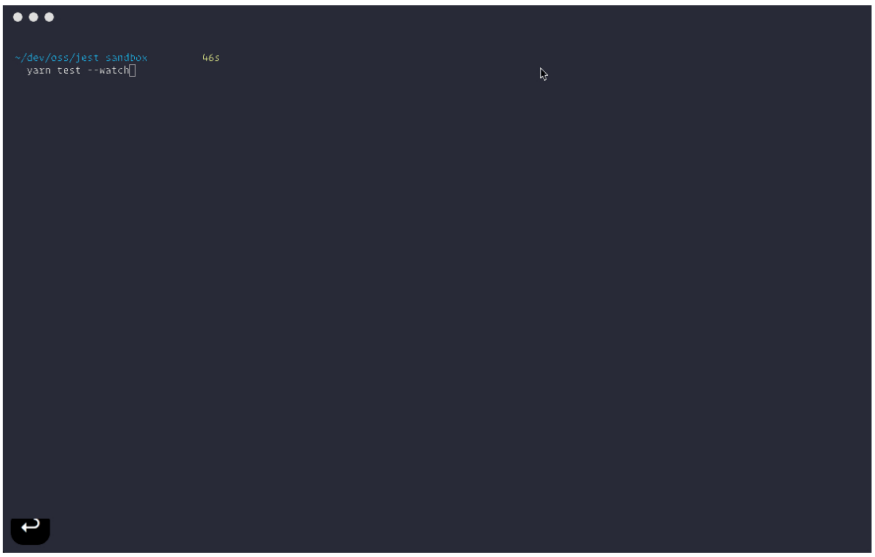
Interactive Snapshot Mode

Failed snapshots can also be updated interactively in watch mode:

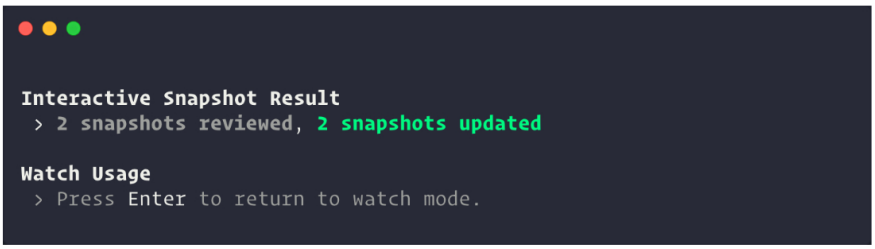


Once you enter Interactive Snapshot Mode, Jest will step you through the failed snapshots one test at a time and give you the opportunity to review the failed output.

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();
});

// Snapshot
exports[`will fail every time 1`] = `
Object {
  "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),
  });
});

// Snapshot
exports[`will check the matchers and pass 1`] = `
Object {
  "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',
  });
});

// Snapshot
exports[`will check the values and pass 1`] = `
Object {
  "createdAt": Any<Date>,
  "name": 'Bond... James Bond',
```

```
`;
```

> **TIP**
>
> If 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.