Unit testing is a great way to protect against errors in your code before you deploy it. While Gatsby does not include support for unit testing out of the box, it only takes a few steps to get up and running. However, there are a few features of the Gatsby build process that mean the standard Jest setup doesn't quite work. This guide shows you how to set it up.

Setting up your environment

The most popular testing framework for React is Jest, which was created by Facebook. While Jest is a general-purpose JavaScript unit testing framework, it has lots of features that make it work particularly well with React.

Note: For this guide, you will be starting with `gatsby-starter-default`, but the concepts should be the same or very similar for your site.

1. Installing dependencies

First, you need to install Jest and some more required packages. Install babel-jest and babel-preset-gatsby to ensure that the babel preset(s) that are used match what are used internally for your Gatsby site.

```
npm install --save-dev jest babel-jest react-test-renderer babel-preset-gatsby
identity-obj-proxy
```

2. Creating a configuration file for Jest

Because Gatsby handles its own Babel configuration, you will need to manually tell Jest to use `babel-jest`. The easiest way to do this is to add a `jest.config.js`. You can set up some useful defaults at the same time:

```
module.exports = {
  transform: {
    "^.+\\.jsx?$": `<rootDir>/jest-preprocess.js`,
  },
  moduleNameMapper: {
    ".+\\.(css|styl|less|sass|scss)$": `identity-obj-proxy`,
    ".+\\.
(jpg|jpeg|png|gif|eot|otf|webp|svg|ttf|woff|woff2|mp4|webm|wav|mp3|m4a|aac|oga)$":
`<rootDir>/__mocks__/file-mock.js`,
    "^gatsby-page-utils/(.*)$": `gatsby-page-utils/dist/$1`, // Workaround for
https://github.com/facebook/jest/issues/9771
    "^gatsby-core-utils/(.*)$": `gatsby-core-utils/dist/$1`, // Workaround for
https://github.com/facebook/jest/issues/9771
    "^gatsby-plugin-utils/(.*)$": [
      `gatsby-plugin-utils/dist/$1`,
      `gatsby-plugin-utils/$1`,
    ], // Workaround for https://github.com/facebook/jest/issues/9771
  },
  testPathIgnorePatterns: [`node_modules`, `\\.cache`, `<rootDir>.*/public`],
  transformIgnorePatterns: [`node_modules/(?!(gatsby)/)`],
  globals: {
    __PATH_PREFIX__: ``,
  },
  testURL: `http://localhost`,
```

```
  setupFiles: [`<rootDir>/loadershim.js`],
}
```

Go over the content of this configuration file:

- The `transform` section tells Jest that all `js` or `jsx` files need to be transformed using a `jest-preprocess.js` file in the project root. Go ahead and create this file now. This is where you set up your Babel config. You can start with the following minimal config:

```
const babelOptions = {
  presets: ["babel-preset-gatsby"],
}

module.exports = require("babel-jest").default.createTransformer(babelOptions)
```

> ***Note:*** *If you're using Jest 26.6.3 or below, the last line has to be changed to* `module.exports = require("babel-jest").createTransformer(babelOptions)`

- The next option is `moduleNameMapper`. This section works a bit like webpack rules and tells Jest how to handle imports. You are mainly concerned here with mocking static file imports, which Jest can't handle. A mock is a dummy module that is used instead of the real module inside tests. It is good when you have something that you can't or don't want to test. You can mock anything, and here you are mocking assets rather than code. For stylesheets you need to use the package `identity-obj-proxy`. For all other assets, you need to use a manual mock called `file-mock.js`. You need to create this yourself. The convention is to create a directory called `__mocks__` in the root directory for this. Note the pair of double underscores in the name.

```
module.exports = "test-file-stub"
```

- The next config setting is `testPathIgnorePatterns`. You are telling Jest to ignore any tests in the `node_modules` or `.cache` directories.
- The next option is very important and is different from what you'll find in other Jest guides. The reason that you need `transformIgnorePatterns` is because Gatsby includes un-transpiled ES6 code. By default Jest doesn't try to transform code inside `node_modules`, so you will get an error like this:

```
/my-app/node_modules/gatsby/cache-dir/gatsby-browser-entry.js:1
({"Object.
<anonymous>":function(module,exports,require,__dirname,__filename,global,jest)
{import React from "react"

^^^^^^
SyntaxError: Unexpected token import
```

This is because `gatsby-browser-entry.js` isn't being transpiled before running in Jest. You can fix this by changing the default `transformIgnorePatterns` to exclude the `gatsby` module.

- The `globals` section sets `__PATH_PREFIX__`, which is usually set by Gatsby, and which some components need.
- You need to set `testURL` to a valid URL, because some DOM APIs such as `localStorage` are unhappy with the default (`about:blank`).

> *Note: if you're using Jest 23.5.0 or later, `testURL` will default to `http://localhost` so you can skip this setting.*

- There's one more global that you need to set, but as it's a function you can't set it here in the JSON. The `setupFiles` array lets you list files that will be included before all tests are run, so it's perfect for this.

```
global.___loader = {
  enqueue: jest.fn(),
}
```

3. Useful mocks to complete your testing environment

Mocking `gatsby`

Finally, it's a good idea to mock the `gatsby` module itself. This may not be needed at first, but will make things a lot easier if you want to test components that use `Link` or GraphQL.

```
const React = require("react")
const gatsby = jest.requireActual("gatsby")

module.exports = {
  ...gatsby,
  graphql: jest.fn(),
  Link: jest.fn().mockImplementation(
    // these props are invalid for an `a` tag
    ({
      activeClassName,
      activeStyle,
      getProps,
      innerRef,
      partiallyActive,
      ref,
      replace,
      to,
      ...rest
    }) =>
      React.createElement("a", {
        ...rest,
        href: to,
      })
  ),
  StaticQuery: jest.fn(),
  useStaticQuery: jest.fn(),
}
```

This mocks the `graphql()` function, `Link` component, and `StaticQuery` component.

Writing tests

A full guide to unit testing is beyond the scope of this guide, but you can start with a snapshot test to check that everything is working.

First, create the test file. You can either put these in a `__tests__` directory, or put them elsewhere (usually next to the component itself), with the extension `.spec.js` or `.test.js`. The decision comes down to your own preference. In this guide, you will use the `__tests__` folder convention. To test the header component, create a `header.js` file in `src/components/__tests__/`:

```
import React from "react"
import renderer from "react-test-renderer"

import Header from "../header"

describe("Header", () => {
  it("renders correctly", () => {
    const tree = renderer
      .create(<Header siteTitle="Default Starter" />)
      .toJSON()
    expect(tree).toMatchSnapshot()
  })
})
```

This is a very brief snapshot test, which uses `react-test-renderer` to render the component, and then generates a snapshot of it on the first run. It then compares future snapshots against this, which means you can quickly check for regressions. Visit the Jest docs to learn more about other tests that you can write.

Running tests

If you look inside `package.json` you will probably find that there is already a script for `test`, which just outputs an error message. Change this to use the `jest` executable that you now have available, like so:

```
  "scripts": {
    "test": "jest"
  }
```

This means you can now run tests by typing `npm test`. If you want you could also run with a flag that triggers watch mode to watch files and run tests when they are changed: `npm test -- --watch`.

Run the tests again now and it should all work! You may get a message about the snapshot being written. This is created in a `__snapshots__` directory next to your tests. If you take a look at it, you will see that it is a JSON representation of the `<Header />` component. You should check your snapshot files into a source control system (for example, a GitHub repo) so that any changes are tracked in history. This is particularly important to remember if you are using a continuous integration system such as Travis or CircleCI to run tests, as these will fail if the snapshot is not checked into source control.

If you make changes that mean you need to update the snapshot, you can do this by running `npm test -- -u`.

Using TypeScript

If you are using TypeScript, you need to install typings packages and make two changes to your config.

```
npm install --save-dev @types/jest @types/react-test-renderer
```

Update the transform in `jest.config.js` to run `jest-preprocess` on files in your project's root directory.

Note: `<rootDir>` is replaced by Jest with the root directory of the project. Don't change it.

```
    "^.+\\.[jt]sx?$": "<rootDir>/jest-preprocess.js",
```

Also update `jest-preprocess.js` with the following Babel preset to look like this:

```
const babelOptions = {
  presets: ["babel-preset-gatsby", "@babel/preset-typescript"],
}
```

Once this is changed, you can write your tests in TypeScript using the `.ts` or `.tsx` extensions.

Using tsconfig paths

If you are using tsconfig paths there is a single change to your config.

1. Add ts-jest

```
npm install --save-dev ts-jest
```

2. Update `jest.config.js` to import and map `tsconfig.json` paths

```
const { compilerOptions } = require("./tsconfig.json")
const { pathsToModuleNameMapper } = require("ts-jest/utils")
const paths = pathsToModuleNameMapper(compilerOptions.paths, {
  prefix: "<rootDir>/",
})
```

3. Add paths to `jest.config.js` moduleNameMapper

```
  moduleNameMapper: {
    '.+\\.(css|styl|less|sass|scss)$': `identity-obj-proxy`,
    '.+\\.
(jpg|jpeg|png|gif|eot|otf|webp|svg|ttf|woff|woff2|mp4|webm|wav|mp3|m4a|aac|oga)$':
`<rootDir>/__mocks__/file-mock.js`,
    ...paths,
  },
```

Other resources

If you need to make changes to your Babel config, you can edit the config in `jest-preprocess.js`. You may need to enable some of the plugins used by Gatsby, though remember you may need to install the Babel 7 versions. See the Gatsby Babel config guide for some examples.

For more information on Jest testing, visit the Jest site.

For an example encapsulating all of these techniques--and a full unit test suite with @testing-library/react, check out the using-jest example.