Testing a Component

In this lesson, we will learn about component testing and the most common issues related to nested component testing.

WE'LL COVER THE FOLLOWING

Jest Configuration

Let's add the following Jest configuration in the `package.json`:

package.json

```
{
  "jest": {
    "moduleNameMapper": {
      "@/([^\\.]*).vue$": "<rootDir>/src/$1.vue",
      "@/([^\\.]*)$": "<rootDir>/src/$1.js",
      "^vue$": "vue/dist/vue.common.js"
    },
    "moduleFileExtensions": [
      "js",
      "vue"
    ],
    "transform": {
      "^.+\\.js$": "<rootDir>/node_modules/babel-jest",
      ".*\\.(vue)$": "<rootDir>/node_modules/jest-vue-preprocessor"
    }
}
```

`moduleFileExtensions` will tell Jest which extensions to look for, whereas transform will tell it which preprocessor to use for a file extension.

Lastly, let's add a test script to the `package.json`:

package.json

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

Single File Component

We will start our example by using single file components.

Step 1:

Let's first create a `MessageList.vue` component under `src/components`:

MessageList.vue

```
<template>
  <ul>
    <li v-for="message in messages">
      {{ message }}
    </li>
  </ul>
</template>

<script>
  export default {
    name: "list",
    props: ["messages"]
  };
</script>
```

Step 2:

Now update `App.vue` to use it, as follows:

App.vue

```
<template>
  <div id="app">
    <MessageList :messages="messages" />
```

```
  <MessageList :messages="messages" />
  </div>
</template>

<script>
  import MessageList from "./components/MessageList";

  export default {
    name: "app",
    data: () => ({ messages: ["Hey John", "Howdy Paco"] }),
    components: {
      MessageList
    }
  };
</script>
```

Step 3:

Now that we have a couple of components to test, let's create a `test` folder under the project root, and an `App.test.js:`

```
'use strict'
require('./check-versions')()

process.env.NODE_ENV = 'production'

const ora = require('ora')
const rm = require('rimraf')
const path = require('path')
const chalk = require('chalk')
const webpack = require('webpack')
const config = require('../config')
const webpackConfig = require('./webpack.prod.conf')

const spinner = ora('building for production...')
spinner.start()

rm(path.join(config.build.assetsRoot, config.build.assetsSubDirectory), err => {
  if (err) throw err
  webpack(webpackConfig, (err, stats) => {
    spinner.stop()
    if (err) throw err
    process.stdout.write(stats.toString({
      colors: true,
      modules: false,
      children: false, // If you are using ts-loader, setting this to true will make TypeScri
      chunks: false,
      chunkModules: false
    }) + '\n\n')

    if (stats.hasErrors()) {
      console.log(chalk.red('  Build failed with errors.\n'))
      process.exit(1)
    }

    console.log(chalk.cyan('  Build complete.\n'))
    console.log(chalk.yellow(
      '  Tip: built files are meant to be served over an HTTP server.\n' +
      '  Opening index.html over file:// won\'t work.\n'
```

```
    ))
  })
})
```

Step 4: #

As the last step, let's run our test. If we run `npm test` (or `npm t` as a shorthand version), the test should run and pass. Since we'll be modifying the tests, we should run them in **watch mode**.

```
npm t -- --watch
```

Watch mode runs tests related only to the files that have changed.

Note: Pressing the `Run` button will run a test script for you by default.

The Problem with Nested Components

The test we just ran was a simple one. Let's check if the output is as expected. We can use the amazing Snapshots feature of Jest which will generate a snapshot of the output. Add this after the last `it` in `App.test.js`:

```
'use strict'
require('./check-versions')()

process.env.NODE_ENV = 'production'

const ora = require('ora')
const rm = require('rimraf')
const path = require('path')
const chalk = require('chalk')
const webpack = require('webpack')
const config = require('../config')
const webpackConfig = require('./webpack.prod.conf')

const spinner = ora('building for production...')
spinner.start()

rm(path.join(config.build.assetsRoot, config.build.assetsSubDirectory), err => {
  if (err) throw err
  webpack(webpackConfig, (err, stats) => {
    spinner.stop()
    if (err) throw err
    process.stdout.write(stats.toString({
      colors: true,
      modules: false,
      children: false, // If you are using ts-loader, setting this to true will make TypeScri
      chunks: false,
      chunkModules: false
    }) + '\n\n')
```

```
    if (stats.hasErrors()) {
      console.log(chalk.red('  Build failed with errors.\n'))

      process.exit(1)
    }

    console.log(chalk.cyan('  Build complete.\n'))
    console.log(chalk.yellow(
      '  Tip: built files are meant to be served over an HTTP server.\n' +
      '  Opening index.html over file:// won\'t work.\n'
    ))
  })
})
```

This will create a `test/__snapshots__/App.test.js.snap` file. Let's open and inspect it:

```
'use strict'
require('./check-versions')()

process.env.NODE_ENV = 'production'

const ora = require('ora')
const rm = require('rimraf')
const path = require('path')
const chalk = require('chalk')
const webpack = require('webpack')
const config = require('../config')
const webpackConfig = require('./webpack.prod.conf')

const spinner = ora('building for production...')
spinner.start()

rm(path.join(config.build.assetsRoot, config.build.assetsSubDirectory), err => {
  if (err) throw err
  webpack(webpackConfig, (err, stats) => {
    spinner.stop()
    if (err) throw err
    process.stdout.write(stats.toString({
      colors: true,
      modules: false,
      children: false, // If you are using ts-loader, setting this to true will make TypeScri
      chunks: false,
      chunkModules: false
    }) + '\n\n')

    if (stats.hasErrors()) {
      console.log(chalk.red('  Build failed with errors.\n'))
      process.exit(1)
    }

    console.log(chalk.cyan('  Build complete.\n'))
    console.log(chalk.yellow(
      '  Tip: built files are meant to be served over an HTTP server.\n' +
      '  Opening index.html over file:// won\'t work.\n'
    ))
  })
})
```

If you don't know much about snapshots, don't worry. We'll cover that in Snapshot Testing.

In case you haven't noticed, there is a big problem here: the `MessageList` component has been rendered as well.

> Unit tests must be tested as independent units.

This means that, in `App.test.js` we only want to test the App component, and don't care about anything else.

This can be the cause of several problems. What if, for example, the children components (`MessageList` in this case) perform side effect operations on the `created` hook, such as calling `fetch,` a Vuex action or state changes? That's something we definitely don't want.

Luckily, **Shallow Rendering** solves this nicely.

What is Shallow Rendering?

Shallow Rendering is a technique that assures your component is rendering without children. This is useful for:

- Testing only the component you want to test (that's what Unit Test stands for)
- Avoid side effects that children components can have, such as making HTTP calls, calling store actions...

In the next lesson, we'll be testing a component with vue-test-utils.