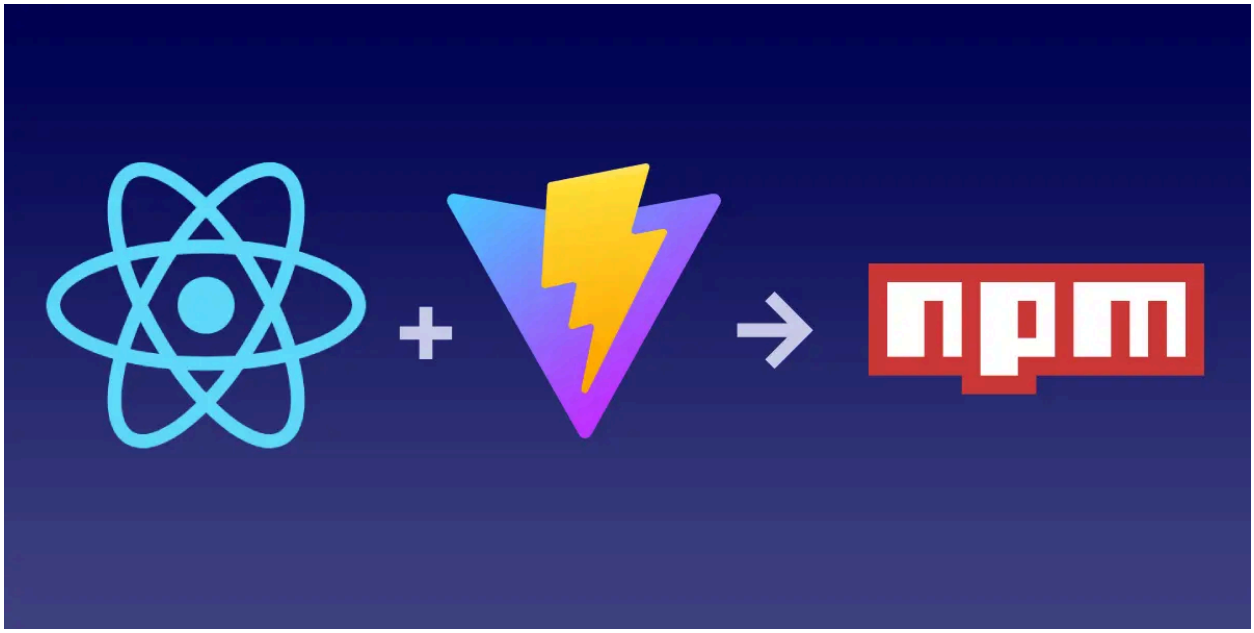
Lundi 28 février 2022

Publication d'un composant React sur npm à l'aide de Vite

Tom Southall
@tomsouthall



Aperçu

Ce guide est conçu pour vous aider à créer un composant React à l'aide de Vanilla JS, puis à le publier sur npm en tant que package pour le réutiliser par d'autres applications React.

Comme moi, vous avez peut-être découvert que publier un composant React n'est pas aussi simple que vous l'auriez imaginé au départ. Après de nombreuses

tentatives, j'ai fini par obtenir une configuration très pratique que je peux réutiliser pour n'importe quel projet.

Nous allons utiliser Vite comme outil de création et environnement de développement local. Si vous n'êtes pas familier avec Vite, c'est un outil de création ultra-rapide et ultra-léger qui exploite la possibilité d'utiliser des modules ES natifs dans les navigateurs modernes. En bref, c'est génial.

Guide

Tout le code de ce guide est disponible dans le référentiel d'accompagnement sur Github .

Pré-requis

- Vous avez installé Git
- Vous avez installé Node.js et npm
- Vous avez un compte npm et êtes connecté

Configuration initiale

1. Créez votre dossier de projet et `cd` placez-vous dedans. Pour les besoins de ce guide, nous supposerons que votre composant s'appelle `my-component`

```
$ mkdir my-component
$ cd my-component
```

2. Créer un nouveau dépôt git

```
$ git init .
```

3. Créez les dossiers et fichiers suivants. Tous les fichiers peuvent être laissés vides pour le moment :

Vos fichiers de composants qui seront regroupés et empaquetés prêts à être publiés sur npm se trouvent dans le `/src/lib` dossier. Les autres fichiers `/src` ne sont nécessaires que pour l'environnement de développement. Ils seront ignorés au moment de la construction.

Ma préférence pour les fichiers de test est de les avoir à côté du fichier qu'ils testent, dans ce cas, `/src/lib` mais ils peuvent être placés, par exemple, dans un `__tests__` dossier séparé si vous préférez.

Notez que dans Vite, les fichiers jsx doivent avoir une `.jsx` extension de fichier et **non** `.js` .

Configurer .gitignore

1. Dans votre éditeur, ouvrez `.gitignore`
2. Collez ce qui suit et ajoutez tous les autres fichiers ou dossiers que vous souhaitez ignorer.

```
# dependencies
/node_modules

# testing
/coverage
```

```
# component build
/dist

# misc
.DS_Store
```

Configurer package.json

1. Dans votre éditeur, ouvrez `/package.json` et collez ce qui suit :

```
{
  "name": "my-component",
  "version": "0.0.0",
  "description": "",
  "license": "MIT",
  "author": "",
  "keywords": [],
  "files": [
    "dist",
    "README.md"
  ],
  "main": "./dist/my-component.umd.js",
  "module": "./dist/my-component.es.js",
  "exports": {
    ".": {
      "import": "./dist/my-component.es.js",
      "require": "./dist/my-component.umd.js"
    }
  },
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "test": "vitest run",
    "watch": "vitest",
    "coverage": "vitest run --coverage"
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
```

```
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  },
  "eslintConfig": {
    "env": {
      "browser": true,
      "node": true,
      "es2020": true
    },
    "extends": [
      "eslint:recommended",
      "plugin:react/recommended"
    ],
    "parserOptions": {
      "sourceType": "module"
    }
  }
}
```

2. Remplacez toutes les instances de `my-component` par le nom de votre composant/package. Il ne doit pas s'agir d'un nom déjà utilisé par un autre package npm car c'est ce que les utilisateurs saisiront pour installer votre composant dans leurs projets, par exemple : `npm install my-component` . Vous pouvez vérifier si le nom est déjà utilisé en le recherchant sur le site Web de npm .

3. Ajoutez vos propres valeurs pour `description` , `author` et `keywords`

4. Modifiez la licence selon vos préférences

5. Installez React en tant que dépendance de développement. Nous ne l'installons pas en tant que dépendance de production, car l'application React qui utilise votre package de composants est responsable de la dépendance du produit React.

```
$ npm install react react-dom --save-dev
```

6. Installez Vite en tant que dépendance de développement.

```
$ npm install vite @vitejs/plugin-react --save-dev
```

7. Installer toutes les dépendances de développement requises pour les tests et le linting

```
$ npm install @testing-library/dom @testing-library/react c8 eslint eslint
```

Configurer Vite

1. Dans votre éditeur, ouvrez `/vite.config.js` et collez ce qui suit :

```
import path from 'path'
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  build: {
    lib: {
      entry: path.resolve(__dirname, 'src/lib/index.jsx'),
      name: 'My Component',
      fileName: (format) => `my-component.${format}.js`
    },
    rollupOptions: {
      external: ['react', 'react-dom'],
      output: {
        globals: {
          react: 'React'
        }
      }
    }
  },
  plugins: [react()]
})
```

2. Make sure you replace `name: 'My Component'` with the name of your component

3. Likewise, make sure that `my-component` in the `fileName` value is changed to the name of your component

Configure Testing and Linting

1. In your editor, open `/vitest.config.js` and paste in the following:

```
import { defineConfig } from 'vite'

export default defineConfig({
  test: {
    globals: false,
    environment: 'jsdom'
  }
})
```

2. If you prefer different ESLint rules, configure these in `eslintConfig` in `package.json`

Write your Component along with Snapshot Tests

The entrypoint for your component is `/src/lib/index.jsx`. You can of course create a whole set of child components too within `/src/lib` as you would in any React app.

For now you may want to create a dummy HelloWorld component as in the examples below, then once the development environment is set up, come back and build your component.

1. Create your component in `/src/lib/index.jsx`. Here's a very simple example:

```
import React from 'react'

export default function HelloWorld(props) {
```

```
  const {
    greetee = 'World'
  } = props

  return (
    <div>Hello, {greetee}!</div>
  )
}
```

2. Add snapshot tests to `/src/lib/index.test.jsx` . For example:

```
import React from 'react'
import renderer from 'react-test-renderer'
import { describe, expect, test } from 'vitest'
import HelloWorld from './index'

describe('HelloWorld', () => {
  test('HelloWorld component renders correctly', () => {
    const component = renderer.create(
      <HelloWorld />
    )

    const tree = component.toJSON()

    expect(tree).toMatchSnapshot()
  })

  test('The greetee prop works', () => {
    const component = renderer.create(
      <HelloWorld greetee={'Universe'} />
    )

    const tree = component.toJSON()

    expect(tree).toMatchSnapshot()
  })
})
```

You can run your tests from the command line by typing:

```
$ npm test
```

Or you can run tests and set a watch to re-run tests when anything changes.

```
$ npm run watch
```

Note: Running snapshot tests will create a `/src/lib/__snapshots__` folder. Do not gitignore this folder. Snapshot tests are intended to be committed to your Git repository.

Set Up the Development App

We need to create a development React app in order to test your component library.

1. Open `/index.html` and paste in the following. Change the `<title>` element to whatever you choose.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
    <title>My React Component</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

2. Open `/src/main.jsx` and paste in the following:

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

ReactDOM.render(
```

```
    <React.StrictMode>
      <App />
    </React.StrictMode>,
    document.getElementById('root')
  )
```

3. Open `/src/index.css` and enter any CSS you want to be applied to the local development page, for example:

```
body {
  font-family: sans-serif;
}
```

4. Open `/src/main.jsx` and paste in the following, replacing the `HelloWorld` component with your own.

```
import React from 'react'
import HelloWorld from './lib'

const App = () => {
  return (
    <HelloWorld greetee={'Universe'} />
  )
}

export default App
```

To launch the development environment from the command line enter:

```
$ npm run dev
```

You can then navigate to `http://localhost:3000` in your browser to see your component.

Validate Props

(Optional)

I recommend typechecking your component's props. This will help developers who are using your component in their projects to ensure they are passing the expected types. This only applies if you are using Vanilla JS as assumed by this guide. Typescript does away with this need.

1. Install the prop-types package as a production dependency:

```
$ npm install prop-types
```

2. Add validation rules, for example in our `/src/lib/index.jsx` file, we would make the following additions:

```
import React from 'react'
import PropTypes from 'prop-types'

export default function HelloWorld(props) {
  const {
    greetee = 'World'
  } = props

  return (
    <div>Hello, {greetee}!</div>
  )
}
```

```
HelloWorld.propTypes = {
  greetee: PropTypes.string
}
```

Integration Testing

(Optional)

If your component is at all complex and has child components, then writing integration tests is highly recommended so that you can test your component as an end-user. Snapshot and unit tests can even be replaced in many instances by good integration tests using Testing Library.

Here is an example `/src/lib/integration.test.jsx` for our dummy `HelloWorld` component:

```
import * as React from 'react'
import { render, screen, cleanup } from '@testing-library/react'
import { describe, expect, test, afterEach } from 'vitest'
import HelloWorld from './index'

describe('Integration test', () => {
  afterEach(cleanup)

  test('Minimal render display expected text', () => {
    render(<HelloWorld />)
    expect(screen.getByText('Hello, World!'))
  })

  test('Expected greetee is displayed', () => {
    const greetee = 'Universe'
    render(<HelloWorld greetee={greetee} />)
    expect(screen.getByText(`Hello, ${greetee}!`))
  })
})
```

Write a README

If you want other developers to use your component, it's really important to write a top-notch README file including detailed documentation and examples, as well a the features and benefits of using your component.

Ceci sera publié sur npm.

1. Ouvrir `/src/README.md`

2. Créez votre fichier README en utilisant Markdown.

Publication sur npm

Nous sommes enfin prêts à publier notre composant sur npm.

Voici notre liste de contrôle de publication.

1. Assurez-vous tout d'abord que le numéro de version de votre `package.json` fichier est correct et qu'il utilise des règles de contrôle de version sémantique . Chaque fois que vous publiez sur npm, vous devez le faire avec un nouveau numéro de version.

2. Assurez-vous que tous les tests réussissent :

   ```
   $ npm test
   ```

3. Créez les fichiers de build :

   ```
   $ npm run build
   ```

   Les formats de module UMD et ESM sont créés et placés dans le `/dist` dossier. Notez encore une fois que React n'est pas fourni avec votre composant. Il s'agit uniquement du code de votre composant et de ses éventuelles dépendances.

4. Assurez-vous d'être connecté à npm. Sinon, saisissez :

   ```
   $ npm login
   ```

5. Publiez votre composant

```
$ npm publish
```

Et c'est tout !

Une chose utile à faire ensuite est de créer une application de test qui récupère votre composant à partir de npm. Vous pouvez ensuite vérifier qu'il fonctionne comme prévu.

Discutez sur Twitter

MOTS CLÉS
VITE RÉAGIR NPM

ARTICLE PRÉCÉDENT
Un composant React pour trouver des correspondances dans une chaîne divisée

ARTICLE SUIVANT
Comment créer des palettes de couleurs personnalisées dans Tailwind CSS

← Retour au blog

post ténébreux lux