**Alex Eagleson**

Posted on Nov 17, 2021 • Updated on Dec 6, 2022

How to Create and Publish a React Component Library

#webdev #react #javascript #tutorial

All code from this tutorial as a complete package is available in [this repository](#).

If you are interested in a video version of this tutorial, check out the link below! You can follow along with the code in this blog.

(The video is entirely optional, every step and instruction is covered in the blog post)

How to Create and Publis...



1. [Introduction](#)
2. [Prerequisites and Setup](#)
3. [Creating Components](#)
4. [Adding Typescript](#)
5. [Adding Rollup](#)
6. [Building Your Library](#)
7. [Publishing Your Library](#)
8. [Using Your Library](#)
9. [Adding CSS](#)
10. [Optimizing](#)
11. [Adding Tests](#)
12. [Adding Storybook](#)
13. [Adding SCSS](#)
14. [Wrapping Up](#)

Introduction

This tutorial will take you through the process of creating and publishing your own custom React component library and hosting it on Github.

At the end of this tutorial you will have the ability to the following in all of your future React projects:

```
npm install @my-github-account/my-cool-component-library
```

```
import MyCustomComponent from '@my-github-account/my-cool-component-library';
```

```
const MyApp = () => {  
  return (  
    <div>  
      <MyCustomComponent />  
    </div>  
  )  
}
```

Prerequisites and Setup

This project assumes you are familiar with and have installed:

- Code editor / IDE (this tutorial uses VS Code but any IDE will work)
- NPM (NPM is installed when you install Node.js on your machine)
- Installing packages (presume you know how to add packages to a Javascript project with `npm install`)
- Bash terminal (or another terminal you are comfortable with for running commands)
- Git (we will be creating a git repository on our machine and publishing it to Github, though all instructions will be provided on how to follow along)
- React (how to create simple components using JSX)
- Typescript (how to create an object interface with simple properties)

First we will initialize our project.

```
npm init
```

You can take the defaults for all the values, we'll edit them later in the tutorial.

Next we will add the tools necessary to create our components.

```
npm install react typescript @types/react --save-dev
```

Creating Components

Now we can create our first component. Because we are creating a library, we are going to create *index* files for each tier, and export our components from each one to make it as easy as possible for the people using our library to import them.

Within the root of your project, create the following file structure:

```
.
├── src
│   ├── components
│   │   ├── Button
│   │   │   ├── Button.tsx
│   │   │   └── index.ts
│   │   └── index.ts
│   └── index.ts
├── package.json
└── package-lock.json
```

Make sure to double check your structure. You should have three `index.ts` files, and a `Button.tsx` file inside of a `Button` directory. If you have a preferred way of structuring React components within a project you are of course welcome to do it however you like, but this is the structure we will follow for this tutorial.

Begin by creating `Button.tsx`:

```
src/components/Button/Button.tsx
```

```
import React from "react";

export interface ButtonProps {
  label: string;
}

const Button = (props: ButtonProps) => {
  return <button>{props.label}</button>;
};

export default Button;
```

To keep things simple we will just export a button that takes a single prop called `label`. We can add more complexity and styles to our components once we have confirmed that

our basic template is setup correctly.

After our button, we update the index file inside our Button directory:

```
src/components/Button/index.ts
```

```
export { default } from "./Button";
```

Then we export that button from the components directory:

```
src/components/index.ts
```

```
export { default as Button } from "./Button";
```

And finally, we will export all of our components from the base *src* directory:

```
src/index.ts
```

```
export * from './components';
```

Adding Typescript

Up until now, we haven't yet initialized Typescript in our project. Although you technically don't need a configuration file to use Typescript, for the complexity of building a library we are definitely going to need one.

You can initialize a default configuration by running the following command:

```
npx tsc --init
```

That will create a `tsconfig.json` file for us in the root of our project that contains all the default configuration options for Typescript.

If you would like to learn more about the many options in a `tsconfig.json` file, modern versions of TS will automatically create descriptive comments for each value. In addition you can find full documentation on the configuration [here](#).

You may notice depending on your IDE that immediately after initializing you begin to get errors in your project. There are two reasons for that: the first is that Typescript isn't configuration to understand React by default, and the second is that we haven't defined

our method for handling modules yet: so it may not understand how to manage all of our exports.

To fix this we are going to add the following values to `tsconfig.json`:

```
{
  "compilerOptions": {
    // Default
    "target": "es5",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true,

    // Added
    "jsx": "react",
    "module": "ESNext",
    "declaration": true,
    "declarationDir": "types",
    "sourceMap": true,
    "outDir": "dist",
    "moduleResolution": "node",
    "allowSyntheticDefaultImports": true,
    "emitDeclarationOnly": true,
  }
}
```

I have separated these values into a couple different sections based on the default `tsconfig.json` created using the most recent version of Typescript as of this writing (4.4). The values commented *default* should already be set for you by default (you will want to double check and make sure however).

The values marked *added* are new values that we need for our project. We'll briefly outline why we need them:

- `"jsx": "react"` -- Transform JSX into React code
- `"module": "ESNext"` -- Generate modern JS modules for our library
- `"declaration": true` -- Output a `.d.ts` file for our library types
- `"declarationDir": "types"` -- Where to place the `.d.ts` files
- `"sourceMap": true` -- Mapping JS code back to its TS file origins for debugging
- `"outDir": "dist"` -- Directory where the project will be generated

- "moduleResolution": "node" -- Follow node.js rules for finding modules
- "allowSyntheticDefaultImports": true -- Assumes default exports if none are created manually
- "emitDeclarationOnly": true -- Don't generate JS (rollup will do that) only export type declarations

Once you add those values to your TS configuration file you should see the errors in `Button.tsx` and other files immediately disappear.

Adding Rollup

Next we will add [rollup](#) to our project. If you've never used rollup before, it's very similar to [webpack](#) in that it is a tool for bundling individual Javascript modules into a single source that a browser is better able to understand.

Though both tools can accomplish the same goal depending on configuration, typically webpack is used for bundling applications while rollup is particularly suited for bundling libraries (like ours). That's why we've chosen rollup.

Also similar to webpack, rollup uses a *plugin ecosystem*. By design rollup does not know how to do everything, it relies on plugins installed individually to add the functionality that you need.

We are going to rely on four plugins for the initial configuration of our library (more will be added later):

- [@rollup/plugin-node-resolve](#) - Uses the [node resolution algorithm](#) for modules
- [@rollup/plugin-typescript](#) - Teaches rollup how to process Typescript files
- [@rollup/plugin-commonjs](#) - Converts commonjs modules to ES6 modules
- [rollup-plugin-dts](#) - rollup your `.d.ts` files

Important Update!

As time passes, some of these tools have been updated and the exact instructions as written below no longer work. Many people in the comments have graciously taken the time to add corrections, I would highly recommend you read [this comment in particular](#) to educate yourself about the errors and changes you may encounter.

So with that said, let's go ahead and install rollup and our plugins:

```
npm install rollup @rollup/plugin-node-resolve @rollup/plugin-typescript @rollup/plugin-c
```

To configure how rollup is going to bundle our library we need to create a configuration file in the root of our project:

rollup.config.js

```
import resolve from "@rollup/plugin-node-resolve";
import commonjs from "@rollup/plugin-commonjs";
import typescript from "@rollup/plugin-typescript";
import dts from "rollup-plugin-dts";

const packageJson = require("../package.json");

export default [
  {
    input: "src/index.ts",
    output: [
      {
        file: packageJson.main,
        format: "cjs",
        sourcemap: true,
      },
      {
        file: packageJson.module,
        format: "esm",
        sourcemap: true,
      },
    ],
    plugins: [
      resolve(),
      commonjs(),
      typescript({ tsconfig: "../tsconfig.json" }),
    ],
  },
  {
    input: "dist/esm/types/index.d.ts",
    output: [{ file: "dist/index.d.ts", format: "esm" }],
    plugins: [dts()],
  },
];
```


In this file we import our four plugins that we installed. We also import our `package.json` file as a commonJS module into a variable called `packageJson`. We use this variable to refer to the *main* and *module* values that we will define in the next section.

The entrypoint for our library (input) is the `index.ts` file in the `src` directory which exports all of our components. We will be distributing both ES6 and commonJS modules so the consumers of our library can choose which type works best for them. We also invoke three of our four plugins on the first of two configuration objects on the exported array. This first configuration defines how the actual Javascript code of our library is generated.

The second configuration object defines how our libraries types are distributed and uses the `dts` plugin to do so.

The final step before we can run our first rollup is to define the values of "main" and "module" in our `package.json` file:

`package.json`

```
{
  "name": "template-react-component-library",
  "version": "0.0.1",
  "description": "A simple template for a custom React component library",
  "scripts": {
    "rollup": "rollup -c"
  },
  "author": "Alex Eagleson",
  "license": "ISC",
  "devDependencies": {
    "@rollup/plugin-commonjs": "^21.0.1",
    "@rollup/plugin-node-resolve": "^13.0.6",
    "@rollup/plugin-typescript": "^8.3.0",
    "@types/react": "^17.0.34",
    "react": "^17.0.2",
    "rollup": "^2.60.0",
    "rollup-plugin-dts": "^4.0.1",
    "typescript": "^4.4.4"
  },
  "main": "dist/cjs/index.js",
  "module": "dist/esm/index.js",
  "files": [
    "dist"
  ],
}
```

```
"types": "dist/index.d.ts"  
}
```

Here is the sample of the `package.json` file we are using for this tutorial. Obviously your author name can be different, and the specific version of each of your libraries might be different as well.

The most important changes are as follows:

- "main" -- We have defined the output path for commonjs modules
- "module" -- We have defined the output path for es6 modules
- "files" -- We have defined the output directory for our entire library
- "types" -- We have defined the location for our library's types
- "scripts" -- We have defined a new script called **rollup**. This will run the rollup package with the `-c` flag which means "use the rollup configuration file". If you're not familiar with *script* in a `package.json` file, these are simply shorthand commands you can run by name with `npm run {SCRIPTNAME}`. So to run this one will be `npm run rollup`.

Building your library

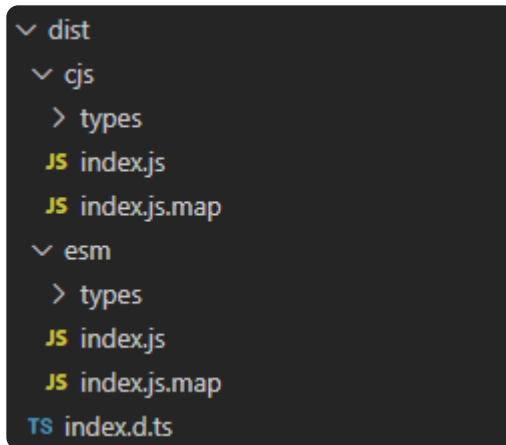
With these configurations in place you are now ready to run rollup for the first time and make sure your basic configuration is correct. Your project structure should look like this before you run:

```
.  
├── src  
│   ├── components  
│   │   ├── Button  
│   │   │   ├── Button.tsx  
│   │   │   └── index.ts  
│   │   └── index.ts  
│   └── index.ts  
├── package.json  
├── package-lock.json  
├── tsconfig.json  
└── rollup.config.js
```

The contents of each file should be as described above. Once you have confirmed this, run the following command:

```
npm run rollup
```

If everything has been configured correctly rollup will run without error and you will see a `dist` directory created in the root of your project with a structure that looks like:



(If you received an error make sure to read it closely to try and identify the issue. Double check that each of your files follows exactly the structure of the examples. Depending on the amount of time passed since the publishing of this tutorial, new major versions of libraries could potentially be published with breaking changes. All versions of libraries numbers are visible above in the `package.json` example in the event you need to specify a specific version)

Publishing your library

Now that we've created our component library, we need a way to allow ourselves (or others) to download and install it. We will be publishing our library via NPM through hosting on Github. First before anything else we need to create a repository for our library.

Create a new repository on Github. I have titled mine `template-react-component-library`. Then follow the steps to initialize your project as a git project, and push to your new repository.

Log into Github and create a new repository called whatever you like. For this example I've titled it `template-react-component-library` and it will be available for everyone to clone and use publicly. You can choose to make your library private if you like, methods described in this tutorial will work for private packages as well (in case you are making a library for your company for example).

Once the repository is created we need to initialize git within our project locally. Run the following command:

```
git init
```

Next create a `.gitignore` file in the root of the directory (make particular note of the leading period, that signifies this is a hidden file):

```
.gitignore
```

```
dist
node_modules
```

In our `.gitignore` file we are adding the `dist` and `node_modules` directories. The reason being that both of these are auto-generated directories that we create using commands, so there is no need to include them in our repository.

Now follow the instructions on Github shown in your new repository for committing your code.

This repository that you have created is the one you will clone & edit when you want to make changes and updates to your component library. This is not the package itself that your (as a user) would install and use. To configure within our project where our package needs to be published to, next we need to update `package.json` with that information:

```
package.json
```

```
{
  "name": "@YOUR_GITHUB_USERNAME/YOUR_REPOSITORY_NAME",
  "publishConfig": {
    "registry": "https://npm.pkg.github.com/YOUR_GITHUB_USERNAME"
  },
  ...
}
```

You will be updating the field "name" value and adding a new field called "publishConfig". Note the values above in caps are meant to be replaced with your own values. For example my "name" field value would be `@alexeagleson/template-react-component-library`.

Notice the "packageConfig" also has your Github account name in it as well, but that value does not lead with the @ symbol.

Now that we have configured our project, we need to configure our local install of *NPM* itself to be authorized to publish to your Github account. To do this we use a `.npmrc` file.

This file is **NOT PART OF OUR PROJECT**. This is a global file in a central location. For Mac/Linux users it goes in your home directory `~/.npmrc`.

For Windows users it goes in your home directory as well, though the syntax will be different. Something along the lines of `c:\Users\{YOUR_WINDOWS_USERNAME}`

For more information about this configuration file [read this](#).

Once you have created the file, edit it to include the following information:

```
~/.npmrc
```

```
registry=https://registry.npmjs.org/  
@YOUR_GITHUB_USERNAME:registry=https://npm.pkg.github.com/  
//npm.pkg.github.com/:_authToken=YOUR_AUTH_TOKEN
```

There are two values in caps to replace in the example above. The first is `YOUR_GITHUB_USERNAME`. Make sure to include the leading @ symbol.

The second is `YOUR_AUTH_TOKEN` which we haven't created yet. Back to Github!

Go to your Github profile: Settings -> Developer Settings -> Personal access tokens. Or just click [this link](#)

Click **Generate new token**. Give it a name that suits the project you are building. Give it an expiry date (Github recommends you don't create tokens with an infinite lifespan for security reasons, but that's up to you).

The most important thing is to click the `write:packages` access value. This will give your token permission to read & write packages to your Github account, which is what we need.

New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Note

Template React Component Library

What's this token for?

Expiration *

7 days

The token will expire on Thu, Nov 18 2021

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

<input checked="" type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations
<input checked="" type="checkbox"/> security_events	Read and write security events
<input type="checkbox"/> workflow	Update GitHub Action workflows
<input checked="" type="checkbox"/> write:packages	Upload packages to GitHub Package Registry
<input checked="" type="checkbox"/> read:packages	Download packages from GitHub Package Registry
<input type="checkbox"/> delete:packages	Delete packages from GitHub Package Registry
<input type="checkbox"/> admin:org	Full control of orgs and teams, read and write org projects
<input type="checkbox"/> write:org	Read and write org and team membership, read and write org projects
<input type="checkbox"/> read:org	Read org and team membership, read org projects

Once you are done you can click to create the token. Github will **ONLY SHOW YOU THE TOKEN ONCE**. When you close/refresh the page it will be gone, so make sure to copy it to a secure location (perhaps a password manager if you use one).

The main location you need to place this token is in the `~/.npmrc` file that you created replacing the `YOUR_AUTH_TOKEN` value from the example above.

Before you continue, do one more sanity check to be sure you didn't create the `.npmrc` file in the root directory of your actual library project. This is technically an option, however the reason you need to be careful is that you could accidentally commit it to your Github

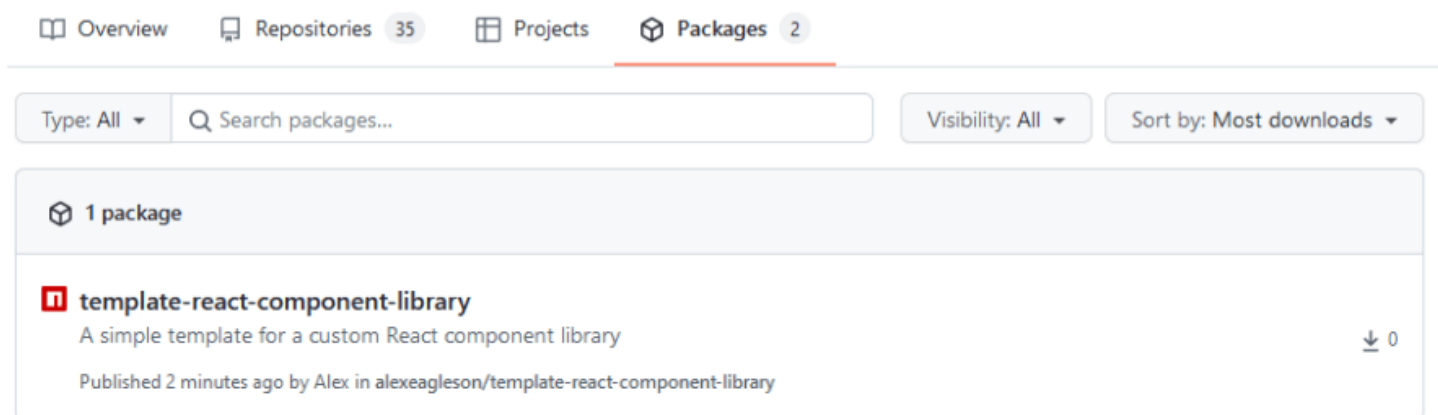
repository with the rest of your library code and expose your token to the public. If your `.npmrc` file is in your home directory the risk of this is minimized.

At this point, once you `~/ .npmrc` file has both your Github username and access token added, go back to your project directory and run the following command:

```
npm publish
```

(If you get prompted for login credentials, the username is your Github username and your password is the access token you generated)

Congratulations! You have now published version 0.0.1 of your React component library! You can view it on your Github account by going to your main account dashboard and clicking "packages" along the top to the right of "repositories":



Using Your Library

Now that your library is live, you'll want to use it!

Note that the instructions for using your library are slightly different if you published to a *private* repository. Everyone (aside from your own machine) who tries to import it is going to get a *404 Not Found* error if they are not authorized.

Those users also need to add a `~/ .npmrc` file with the same information. To be more secure however you can provide those users with an access token that has only **read privileges**, not write.

(From this point onward we will presume you have completed that step, or are working with a public repository.)

Since we have created a component library using React and Typescript, we are presuming that the consumers of our library will be using those tools as well. Technically all of our type files (`.d.ts`) are supplemental: meaning they are simply ignored if working with standard Javascript, so it's not necessary to use Typescript to use our library. The types are simply there if desired.

For our example we will use it however so that we can confirm that they are working properly. We will initialize a React app using one of the most popular and simple methods: [Create React App](#).

Run the following command in a **new directory**:

(Remember we are simulating other users downloading and installing our library, so this project should be completely separate from the library itself)

```
npx create-react-app my-app --template typescript
```

Open the new `my-app` directory that is created and run:

```
npm run start
```

Confirm that you are able to open and load the default application screen on `localhost:3000` (or whatever port it opens on).

Now comes the test for our library. From the root directory of your new `my-app` project, run the following command:

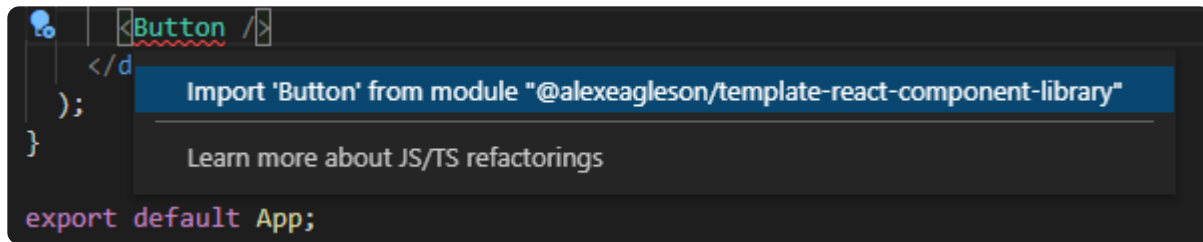
```
npm install @YOUR_GITHUB_USERNAME/YOUR_REPOSITORY_NAME
```

So for my project for example its: `npm install @alexeagleson/template-react-component-library`

Presuming your tokens and configuration are set up properly, everything will install correctly *(if there are any issues, revisit the example for the `~/.npmrc` config.)*

Now open the `my-app` project in your IDE of choice (VS Code for example) and navigate to the `src/App.tsx` file.

When you go to add a `<Button />` component, if your editor supports import auto complete (`ctrl/cmd + .` for VS Code) then you will see it automatically recognize thanks to Typescript that our library exports that button.



Lets add it! The simplest example to update `src/App.tsx` is:

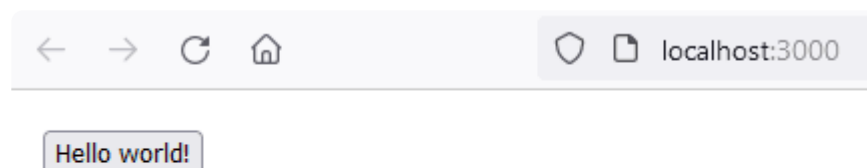
`src/App.tsx`

```
import React from "react";
import { Button } from "@alexeagleson/template-react-component-library";

function App() {
  return <Button label="Hello world!"/>;
}

export default App;
```

And when we run `npm run start` again, there tucked up in the corner is our *Hello world!* button.



And that's it! Congratulations! You now have all the tools you need to create and distribute a React component library using Typescript! At this point you end the tutorial and continue on your own if you wish.

If you choose to continue, we will look at how to expand our component library to include a number of extremely useful features such as:

- **CSS:** For exporting components *with style*

- **Storybook:** For testing our components within the library itself as we design them
- **React Testing Library & Jest:** For testing our components

Adding CSS

Before we do any additional configuration, we'll begin by creating a CSS file that will apply some styles to our Button. Inside of the `Button` directory where our component lives, we'll create a file called: `Button.css`:

`src/components/Button/Button.css`

```
button {  
  font-size: 60px;  
}
```

This will turn our regular *Hello world!* button into a REALLY BIG button.

Next we will indicate that these styles are meant to be applied on our button component. We'll be using special syntax that isn't native to Javascript, but thanks to rollup and the appropriate plugins, we are able to use it. Update our `Button.tsx` file with the following:

`src/components/Button/Button.tsx`

```
import React from "react";  
import "../Button.css";  
  
export interface ButtonProps {  
  label: string;  
}  
  
const Button = (props: ButtonProps) => {  
  return <button>{props.label}</button>;  
};  
  
export default Button;
```

Notice the `import '../Button.css'` that has been added.

Now we need to tell rollup how to process that syntax. To do that we use a plugin called `rollup-plugin-postcss`. Run the following command:

```
npm install rollup-plugin-postcss --save-dev
```

Next we need to update our rollup config:

rollup.config.js

```
import resolve from "@rollup/plugin-node-resolve";
import commonjs from "@rollup/plugin-commonjs";
import typescript from "@rollup/plugin-typescript";
import dts from "rollup-plugin-dts";

// NEW
import postcss from "rollup-plugin-postcss";

const packageJson = require("./package.json");

export default [
  {
    input: "src/index.ts",
    output: [
      {
        file: packageJson.main,
        format: "cjs",
        sourcemap: true,
      },
      {
        file: packageJson.module,
        format: "esm",
        sourcemap: true,
      },
    ],
    plugins: [
      resolve(),
      commonjs(),
      typescript({ tsconfig: "./tsconfig.json" }),

      // NEW
      postcss(),
    ],
  },
  {
    input: "dist/esm/types/index.d.ts",
    output: [{ file: "dist/index.d.ts", format: "esm" }],
    plugins: [dts()],
  },
];
```

```
// NEW
external: [/\.css$/],
},
];
```

Note the three new lines indicated with the `NEW` comments. In the `mts` config we need to specify that `.css` modules are external and should not be processed as part of our type definitions (otherwise we will get an error).

Finally we need to update the *version number* in our `package.json` file. Remember we are publishing a package so when we make changes, we need to ensure we don't impact users of previous versions of our library. Every time we publish we should increment the version number:

`package.json`

```
{
  "version": "0.0.2",
  ...
}
```

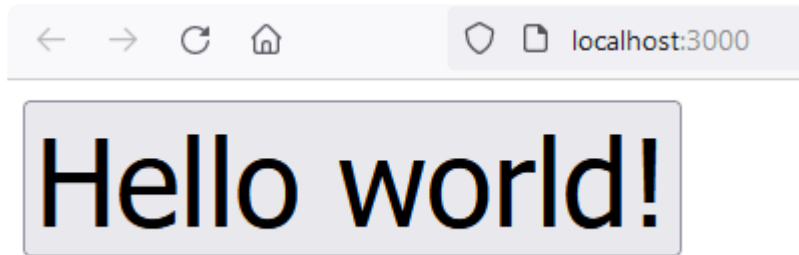
Now run these commands:

```
npm run rollup
npm publish
```

On the library consuming side (`my-app` React app from our tutorial) we also need to update to get the latest version of the package. The easiest way is to increment the version number in the `package.json` file of `my-app`. It should show `^0.0.1`. Increment that to `^0.0.2` and then you can update with the `npm install` command:

```
npm install
npm run start
```

And you'll be treated to a giant button component from our library that now supports bundling CSS!



Optimizing

There are a couple of easy optimizations we can make with this setup. The first is to add a plugin called [terser](#) that will minify our bundle and reduce the overall file size.

Update: see [this comment](#) that rollup now has its own better maintained version of terser

The other is to update some of our dependencies to `peerDependencies`. With rollup's [peer dependencies](#) plugin we can tell the projects that are using our libraries which dependencies are required (like React) but won't actually bundle a copy of React with the library itself. If the consumer already has React in their project it will use that, otherwise it will get installed when they run `npm install`.

First we will install these two plugins:

```
npm install rollup-plugin-peer-deps-external rollup-plugin-terser --save-dev
```

Then we will update our rollup config:

rollup.config.js

```
import resolve from "@rollup/plugin-node-resolve";
import commonjs from "@rollup/plugin-commonjs";
import typescript from "@rollup/plugin-typescript";
import postcss from "rollup-plugin-postcss";
import dts from "rollup-plugin-dts";

//NEW
import { terser } from "rollup-plugin-terser";
import peerDepsExternal from 'rollup-plugin-peer-deps-external';

const packageJson = require("../package.json");
```

```
export default [  
  {  
    input: "src/index.ts",  
    output: [  
      {  
        file: packageJson.main,  
        format: "cjs",  
        sourcemap: true,  
      },  
      {  
        file: packageJson.module,  
        format: "esm",  
        sourcemap: true,  
      },  
    ],  
    plugins: [  
      // NEW  
      peerDepsExternal(),  
  
      resolve(),  
      commonjs(),  
      typescript({ tsconfig: "./tsconfig.json" }),  
      postcss(),  
  
      // NEW  
      terser(),  
    ],  
  },  
  {  
    input: "dist/esm/types/index.d.ts",  
    output: [{ file: "dist/index.d.ts", format: "esm" }],  
    plugins: [dts()],  
    external: [/\.css$/],  
  },  
];
```

Then we move React from `devDependencies` to `peerDependencies` in our `package.json` file:

`package.json`

```
{  
  "devDependencies": {  
    "@rollup/plugin-commonjs": "^21.0.1",  
    "@rollup/plugin-node-resolve": "^13.0.6",
```

```
"@rollup/plugin-typescript": "^8.3.0",
"@types/react": "^17.0.34",
"rollup": "^2.60.0",
"rollup-plugin-dts": "^4.0.1",
"rollup-plugin-peer-deps-external": "^2.2.4",
"rollup-plugin-postcss": "^4.0.1",
"rollup-plugin-terser": "^7.0.2",
"typescript": "^4.4.4"
},
"peerDependencies": {
  "react": "^17.0.2"
},
...

```

Adding Tests

To add tests for our components we are going to install [React Testing Library](#), and to run those tests we will install [Jest](#).

```
npm install @testing-library/react jest @types/jest --save-dev
```

Inside of our Button directory, create a new file called `Button.test.tsx`

```
src/components/Button/Button.test.tsx
```

```
import React from "react";
import { render } from "@testing-library/react";

import Button from "../Button";

describe("Button", () => {
  test("renders the Button component", () => {
    render(<Button label="Hello world!" />);
  });
});
```

What this will do is render our button on a non-browser DOM implementation and make sure that it mounts properly. This is a very simple test, but it serves as a good example of the syntax you can use to get started. To go deeper in depth read further in the React Testing Library [documentation](#).

Before we can run the tests we need to configure jest, and create a test runner script in our `package.json`. We'll start with the configuration, create a `jest.config.js` file in the root of the project:

`jest.config.js`

```
module.exports = {  
  testEnvironment: "jsdom",  
};
```

This tells Jest to use [jsdom](#) as our DOM implementation.

Update: `jsdom` now must be added as a manual dependency in the newest version see [this comment](#) for more info

Next update your `package.json` file:

`package.json`

```
{  
  "scripts": {  
    "rollup": "rollup -c",  
    "test": "jest"  
  },  
  ...  
}
```

Now we can run our tests with:

```
npm run test
```

Unfortunately we are going to get an error! The error is when our JSX code is encountered. If you recall we used Typescript to handle JSX with our rollup config, and a Typescript plugin for rollup to teach it how to do that. We have no such setup in place for Jest unfortunately.

We are going to need to install [Babel](#) to handle our JSX transformations. We will also need to install a Jest plugin called `babel-jest` that tells Jest to use Babel! Let's install them now, along with Babel plugins to handle our Typescript and React code. The total collection of all of them looks like:


```
npm install @babel/core @babel/preset-env @babel/preset-react @babel/preset-typescript ba
```

Now we create our Babel configuration file in the root of our project, which tells Babel to use all these plugins we've just installed:

babel.config.js

```
module.exports = {
  presets: [
    "@babel/preset-env",
    "@babel/preset-react",
    "@babel/preset-typescript",
  ],
};
```

Now we should be able to run our tests with `npm run test ...` but... there is one more problem!

You'll get an error saying the `import` of the `.css` file isn't understood. That makes sense because, again, we configured a `postcss` plugin for rollup to handle that, but we did no such thing for Jest.

The final step will be to install a package called [identity-obj-proxy](#). What this does is allow you to configure Jest to treat any type of imports as just generic objects. So we'll do that with CSS files so we don't get an error.

```
npm install identity-obj-proxy --save-dev
```

We need to update our Jest config to include the `moduleNameMapper` property. We've also added `less` and `scss` in there for good measure in case you want to expand your project later to use those:

jest.config.js

```
module.exports = {
  testEnvironment: "jsdom",
  moduleNameMapper: {
    "^(css|less|scss)$": "identity-obj-proxy",
  },
};
```

```
  },  
};
```

Now finally if you've followed up step up to this point, you can run:

```
npm run test
```

And you will be treated to a successful test!

```
> jest  
  
PASS src/components/Button/Button.test.tsx  
  Button  
    ✓ renders the Button component (11 ms)  
  
Test Suites: 1 passed, 1 total  
Tests:       1 passed, 1 total  
Snapshots:   0 total  
Time:        0.601 s  
Ran all test suites.
```

Adding Storybook

Storybook is a a tool for visualizing UI components outside of your site / application. It's fantastic for prototyping and testing different visual states of components to ensure they work the way they are designed to, without the extra overhead of having other unrelated components on the screen.

It also gives you an easy way to see and use your components while working on them in your library project, without having to build an unnecessary testing page just to display them.

Initializing Storybook is very easy. To set it up and configure it automatically we just run the following command:

```
npx sb init --builder webpack5
```

(Note as of this writing Storybook still defaults to using webpack 4 which is why we have added the extra builder flag. Presumably 5 will be the default soon so it may be unnecessary in the future)

Also see [this comment](#) that explains how to properly set Storybook as a devDependency when you are using it solely for testing your component library.

Unlike some of the other tools we have added so far, Storybook much more of a "batteries included" kind of package that handles most of the initial setup for you. It will even add the `scripts` to run it into your `package.json` file automatically.

You will also notice that it creates a `stories` directory in your `src` directory. This directory is full of pre-built templates for you to use as an example of how to create your own stories. I recommend you don't delete these until you become familiar with Storybook and how to write your own stories, having them close by will be very handy.

Now let's create a simple story for our button. Create a new file in the `Button` directory called `Button.stories.tsx`:

```
src/components/Button/Button.stories.tsx
```

```
import React from "react";
import { ComponentStory, ComponentMeta } from "@storybook/react";
import Button from "../Button";

// More on default export: https://storybook.js.org/docs/react/writing-stories/introduction#default-export
export default {
  title: "ReactComponentLibrary/Button",
  component: Button,
} as ComponentMeta<typeof Button>;

// More on component templates: https://storybook.js.org/docs/react/writing-stories/introduction#using-args
const Template: ComponentStory<typeof Button> = (args) => <Button {...args} />;

export const HelloWorld = Template.bind({});
// More on args: https://storybook.js.org/docs/react/writing-stories/args
HelloWorld.args = {
  label: "Hello world!",
};

export const ClickMe = Template.bind({});
ClickMe.args = {
  label: "Click me!",
};
```

This might be a little overwhelming at first, but when you go through it piece by piece you should see it's fairly straightforward.

- The *default export* defines where the button will appear in the Storybook. I've chosen **ReactComponentLibrary** as a simple name to group our custom components together separately from the examples.
- The *Template* determines which component is actually being rendered, and which default args/props to apply to it.
- The *Template.bind* objects are instances or example states of the component. So in a real project you might have something like "LargeButton" and "SmallButton". Since our button is always big I've just used an example of testing the button with two different labels.

If you look at your `package.json` file you'll see that Storybook has already added a `storybook` and `storybook-build` script. The first will host the Storybook application locally for quick and easy testing. The second one will build a static HTML/JS bundle that can easily be hosted on a remote server, so all members of your team can try your components.

For now let's just run:

```
npm run storybook
```

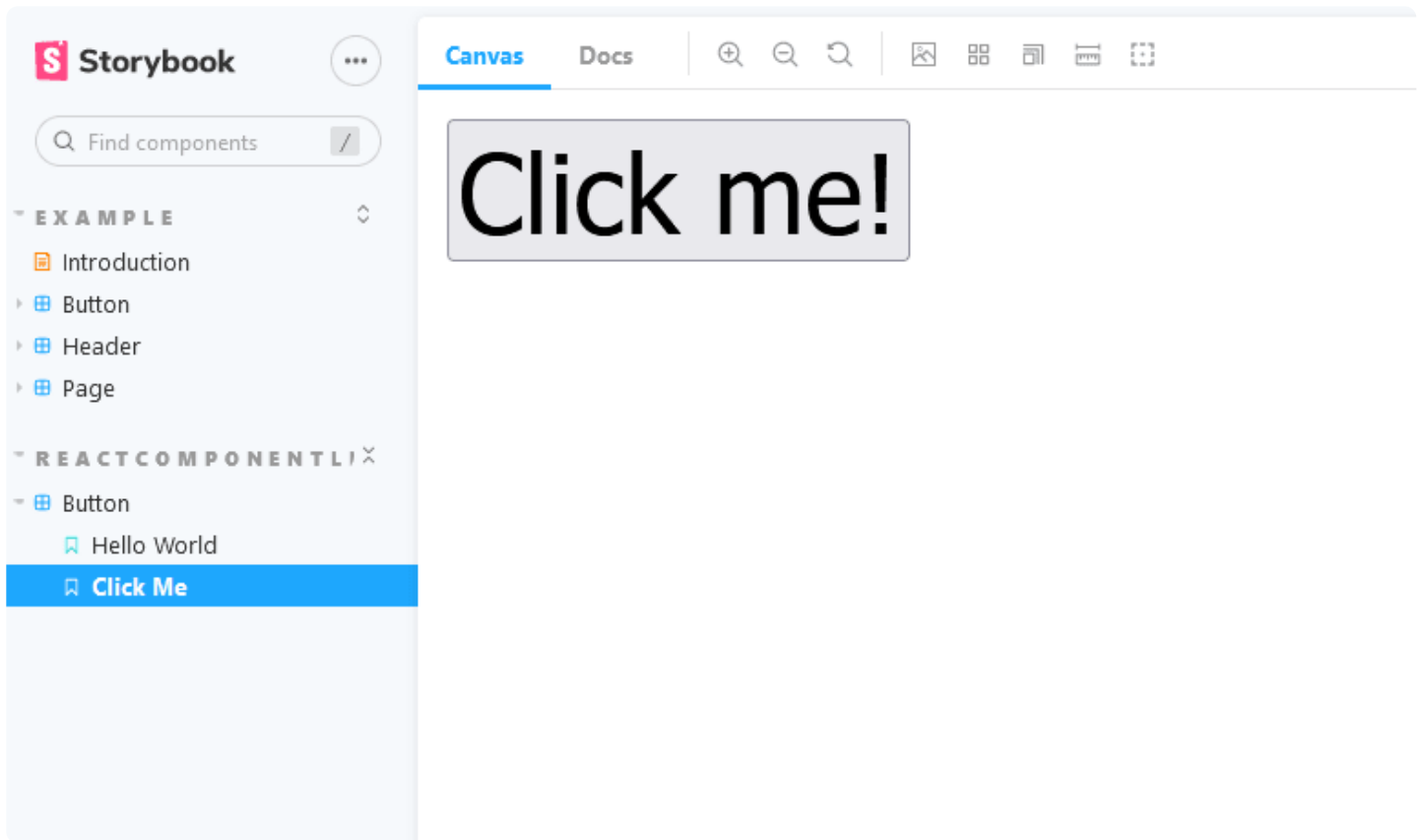
Edit: It's possible you may encounter errors due to missing dependencies. If this occurs there are a few solutions.

The first is to install those dependencies manually. For example `react-dom`. This is not ideal as your project itself should not depend on these libraries, so it should not be necessary to include them as they are included with Storybook's peer dependencies, [as example here](#).

If you simply run a fresh `npm install` command it will install all the `peerDependencies` of the libraries you are using. Before running this you may need to delete your `package-lock.json` and `node_modules` directory. They will be regenerated automatically after your fresh install.

It can be tricky to troubleshoot issues related to both overlapping and missing dependencies between libraries. Stay patient and make sure to read your error messages!)

If all goes well you will be greeted with a friendly interface that lets you navigate through the example components as well as your own custom button in real time. Click between them to check out the different states that you have created.



There is plenty more to learn about Storybook, make sure to read through the [documentation](#).

Adding SCSS

Thanks to `rollup-plugin-postcss` you should already be able to simply rename your `.css` file to `.scss` and then `import 'Button.scss'` and be on your way. Running `npm run rollup` will compile it all just fine with the current configuration.

To get it running with Storybook is a different matter however. Note that this is the main reason we used the `--builder webpack5` flag when installing in the previous section, you will likely encounter a lot of errors trying to configure Storybook to support SCSS with webpack 4. With version 5 it's fairly simple using the SCSS preset.

(If you followed an earlier version of this tutorial you may have initialized Storybook with the default webpack 4. You can remove anything related to Storybook from your

package.json file. Next delete your *package-lock.json* and */node_modules/* directory and initialize Storybook again with the *--builder webpack5* flag).

```
npm install @storybook/preset-scss css-loader sass sass-loader style-loader --save-dev
```

To read more on different kinds of CSS support and Storybook click [here](#).

(If you'd like to understand more about the difference between what these different loaders do, here is a great answer on [Stack Overflow](#))

Then all you need to do is add *@storybook/preset-scss* to your main Storybook config:

.storybook/main.js

```
module.exports = {  
  "stories": [  
    "../src/**/*.stories.mdx",  
    "../src/**/*.stories.@(js|jsx|ts|tsx)"  
  ],  
  "addons": [  
    "@storybook/addon-links",  
    "@storybook/addon-essentials",  
    "@storybook/preset-scss"  
  ],  
  "core": {  
    "builder": "webpack5"  
  }  
}
```

Now you will be able to run *npm run storybook* and see all your SCSS styles.

*(One last reminder that it's common to encounter dependency errors with Storybook. Before you begin installing the missing dependencies it asks for, always try deleting *package-lock.json* and *node_modules* first and then running *npm install* again. This will often fix your issue without requiring you to add unnecessary dependencies to your own project.)*

Wrapping Up

You should now have a good understanding about how to create your own React component library. Doing so can not only teach you a lot about how the Javascript

package management ecosystem works, but it can be a great way to make code that you use across multiple projects easily available with a simple command.

Please check some of my other learning tutorials. Feel free to leave a comment or question and share with others if you find any of them helpful:

- [Learnings from React Conf 2021](#)
- [How to Create a Dark Mode Component in React](#)
- [How to Analyze and Improve your 'Create React App' Production Build](#)
- [How to Create and Publish a React Component Library](#)
- [How to use IndexedDB to Store Local Data for your Web App](#)
- [Running a Local Web Server](#)
- [ESLint](#)
- [Prettier](#)
- [Babel](#)
- [React & JSX](#)
- [Webpack: The Basics](#)
- [Webpack: Loaders, Optimizations & Bundle Analysis](#)

For more tutorials like this, follow me [@eagleson_alex](#) on Twitter

Top comments (133) ↕



Nashe Omirro • Dec 6 '22



It's been almost a year and a half now after this post was written so there are a couple of errors that the article doesn't mention, I listed some of the ones I've encountered and the solutions:

- `rollup -c` **Failed to load Config...** this is because rollup's config cannot support both import statements (esm) and require (cjs) simultaneously. You either have to use one or the other for every import inside your config file. If you want to use esm, first you need to rename your config file to `rollup.config.mjs` to tell rollup that it's a module then theres something extra you need to do for [importing jsons](#):

```
import packageJson from "./package.json" assert { type: "json" };
```

- the `@rollup/plugin-typescript` package now needs `tslib` as a peer dependency, just download `tslib` as dev dependency and you should be good to go.
- for optimization with `terser`, `rollup-plugin-terser` seems to be unmaintained and won't work in some of the latest versions of `rollup`, since this was a pretty heavily used package `rollup` came up with an official plugin, use `@rollup/plugin-terser` instead.
- `jest` no longer has `jsdom` by default, to use the test environment, download `jest-environment-jsdom` as a dev dependency separately.
- `storybook` with `react` and `react-dom` peer dependencies we're already mentioned but I wanted to mention my fix for it, so first is to download both `react` and `react-dom` as a dev dependency, then duplicate `react` to be a peer-dependency (this is surprisingly what other libraries do):

```
{
  "devDependencies": {
    "react": <version>,
    "react-dom": <version>
    // ...
  },
  "peerDependencies": {
    "react": <version>
    // ...
  }
}
```

- don't place `react-dom` as a peer dependency because we solely need that just for `storybook` and our library doesn't use it anywhere. and just in case we accidentally include it in our bundle we need to mark the library as external on our `rollup` config file, `react` is already excluded because of the `peerDepsExternal` plugin.

```
export default [
  {
    input: "src/index.ts",
    external: ['react-dom'],
    // ...
  }
]
```

That's all of the problems I had to deal with but I did skip some steps (adding SCSS and CSS) so there's probably some stuff I missed.



Alex Eagleson 🌟 • Dec 6 '22



Much appreciated for this, and the time and effort that went into it. I've updated the article and placed links to this comment where appropriate for the things that have changed in the past year.

Thanks again, and cheers!



Aman 🚀 • Dec 31 '22



When running `rollup -c` if anyone is getting `SyntaxError: Unexpected Identifier` error than just upgrade your nodejs version to v18 LTS, I was on 16.3 so the importing json using the `assert` keyword was causing this. Thanks to [@nasheomirro](#) for providing a link to a blog where it states that nodejs 17.5+ required.



Rodrigo García • Jan 21



version 16.19.0 works just fine... just in case anyone needs to work with 16 version



Wahaj Aayani • Apr 24 '22



Well written and very concise. But on a side note, has any one came across the issue while doing `$ rollup -c`?

Error: Could not resolve entry module (dist/esm/types/index.d.ts)



Alex Eagleson 🌟 • Apr 25 '22



That command means "run rollup with the configuration found in `rollup.config.js`"

So I would start by making sure your config file looks exactly like the example and all the dependencies are installed

From this section:

[dev.to/alexeagleson/how-to-create-...](https://dev.to/alexeagleson/how-to-create-and-publish-a-react-component-library-2oe)



Ovidiu-Deliu • Aug 8 '22





I've stumbled upon the same issue. It is an issue with the rollup/plugin-typescript that has been also reported here:

github.com/rollup/plugins/issues/1230

You can follow the instructions there to get a workaround or just do the workaround from here:

github.com/alexeagleson/template-r...



Smitter hane • Jan 18



check this solution

stackoverflow.com/questions/731974...



Teymur Salimzade • Sep 25 '22



just change **dist/esm/types/index.d.ts** -> **dist/esm/index.d.ts**



Nathan Dunn • May 1 '22



Could be skipped the first step, making a component to export.



eltr • Aug 1 '22



Hi Wahaj Aayani
Did you fixed it?



Zakaria EL AISSAOUI • Aug 1 '22 • Edited on



Have you found a fix to this issue ?



eltr • Aug 2 '22



github.com/alexeagleson/template-r...



Zakaria EL AISSAOUI • Aug 3 '22



Thank you :)



Pano Papadatos • Jun 4 '22



Awesome article!

Just in case this helps anyone:

I decided to use yarn instead of npm and ran into this issue --

```
[!] (plugin typescript) Error: @rollup/plugin-typescript: Could not find module 'tslib',  
which is required by this plugin. Is it installed?
```

```
Error: @rollup/plugin-typescript: Could not find module 'tslib', which is required by this  
plugin. Is it installed?
```

which was resolved by adding `tslib` as a dev dependency



Wally Atkins • Jul 12 '22



stackoverflow.com/questions/559257...



Lamin • Aug 15 '22



fix using `npm i -D tslib`



Keanu Kent B. Gargar • Jul 3 '22



got the same error even when using just npm



Charles • Dec 7 '21



Hi ! Thank you a lot for this article and video, it helps me a lot !

I may have found something useful if you want to use external librairies (like material UI, HighchartJS, etc) and do some custom components in your npm package (storybook).

The error

Unhandled Runtime Error Error: Invalid hook call. Hooks can only be called inside of the body of a function component. This could happen for one of the following reasons:

1. You might have mismatching versions of React and the renderer (such as React DOM)
2. You might be breaking the Rules of Hooks
3. You might have more than one copy of React in the same app

The why

The React of the parent project is being used, instead of using the second, that is a **duplicate React**.

The solution

I found a solution ([here](#) and [here](#)) where you just have to add this line inside the **rollup.config.js** file :

```
external: ["react", "react-dom"],
```



Pavel Laptev • Jun 4 '22



That's really helpful. Thanks!



blueberrychopsticks • Nov 19 '22



My fix was simply moving `react` and `react-dom` to `peerDependencies` in `package.json`



Alex Eagleson 🌟 • Dec 7 '21



Good stuff, thanks for the resource!



eltr • Nov 8 '22



That's really helpful. Thanks!



Tamb • Jul 12 '22



I've been programming in JS for 7 years and this is one of the best tutorials I have ever taken. You explained the reason behind each configuration perfectly and I actually am excited to use React again.



Alex Eagleson 🌟 • Jul 12 '22



Thank you! Glad it helped



Paul-Simon Emechebe • Aug 5 '22



How do you test it before publishing



Gonzalo Geraldo • Feb 24 '22



Excellent! followed the whole thing and in a couple of hours already have a package on github that I started using in another project (just for testing). Also added jest for unit testing and storybook. All working as expected, thanks!



Jay • Nov 18 '21



Thanks for this article. I just recently created a pirate library for my job. This article looks spot on.

The one thing that helped me, I didn't want to post all my project, and when you do an npm publish from the root, it posts your whole project. So what I do is copy my package.json, and node modules into the dist folder, cd into it, and do my npm publish from there. It makes a much cleaner finished product.



Mike Barkmin • Nov 18 '21



You can add a files property to your package.json. It is an array containing all paths which should be uploaded to npm.

For example

```
"files": ["dist", "README.MD"]
```



Vytenis • Nov 19 '21



Readme, license and packge.json will be included by default. To check what will be available as you publish your package you can run `npm pack`



Alex Eagleson • Nov 18 '21



Great tips, thank you!



Laura • Sep 12 '22



Hey, how did you use a private package in another project? I created a private repo/package and I tried to use it in another project but got 404 and can't find the way around to fix it and use it!
Thanks in advance!



Luiz Azevedo • Feb 22 '22



Hi Alex!

Congratulations for the tutorial and the beautiful explanation.
After a long time of your tutorial, it is still possible to apply it in a simple way.

I'm currently getting a Warning when I run npm start of a project created with create-react-app referring to the package created from your tutorial.
I've been looking for a solution but haven't found it yet.

When running npm start from the application that is using my package I get this warning:

warning in ./node_modules/my_package/dist/esm/index.js

Module Warning (from ./node_modules/source-maploader/dist/cjs.js)

Failed to parse source map from '.../components/my_component.tsx' file: Error ENOENT: no such file or directory, open '.../components/my_component.tsx' @ ./src/index.tsx

I know it's just a warning, but I wanted to resolve it.

Thanks.



Malith Ranasinghe • Oct 10 '22 • Edited on



create new react app and install your custom npm package.
then,
Create an environment variables file named .env in the project root and add
GENERATE_SOURCEMAP=false to it.

OR

Run `npm i -D cross-env` and then in the `package.json` file update the start script to the following:

```
"scripts": {  
  "start": "cross-env GENERATE_SOURCEMAP=false react-scripts start"  
}
```



Malith Ranasinghe • Oct 10 '22



or comment out sourcemap in `rollup.config.js` file

```
{  
  file: packageJson.main,  
  format: "cjs",  
  ** // sourcemap: true,**  
},  
{  
  file: packageJson.module,  
  format: "esm",  
  ** // sourcemap: true,**  
},
```



xingming2020 • Mar 4 '22



I have the same warning. Did you fix it?



Andrew • Apr 7 '22



Hi, love the post.

One problem for me is that `@testing-library/react` is throwing loads of errors, mainly that it can't be found or that it also requires `react-dom` and most of the time also says that it can't find `react-dom` even when it's installed.

I've found the package `react-test-renderer` runs without any issues just incase anyone sees this and are having the same issues.

Also to get storybook to work 3 more packages needed to be installed. `@mdx-js/react` `assert` `@storybook/addon-docs`



abhinav anshul • Jan 9 '22



Amazing article, however I'm stuck into an issue here when I try to install the package, it says not found (it is a public repo), neither its available on the npmjs website.

I've written down about my problem here too : stackoverflow.com/questions/706391...



Alex Eagleson 🌟 • Jan 9 '22



This tutorial is specifically for publishing a package on Github, I would imagine you could probably publish a public package to npmjs.com as well but I've never personally done that before so I'm not sure the exact steps to follow.

If you figure it out and share I'd be happy to update the tutorial

Cheers



Patrick Pulwey • May 19 '22



I've wrote a sample lib and some instructions based on this tutorial, with a quick introductions how to publish to npm. The steps are almost the same. You can find it here: github.com/ppulwey/remake-lib#publish



Vikrant Bhat • Feb 5 '22



I am facing the same issue, did you stumble upon a way to publish to NPM as well?



Gavin Sykes • Mar 10 '22



It looks like you can only really do one or the other at a time, I'm yet to give this a go but this article seems to cover it - sevic.dev/npm-publish-github-actions/



Matthew Dean • Nov 26 '22



Note that this rollup.config.js file no longer works in the latest versions of rollup. It will assume it is a CommonJS with that extension at first. However, if you change it to .mjs as the command line suggests, then it will fail again, because of the use of `require` in the file.

At this point, you could try adding `--bundleConfigAsCjs` to the rollup script, but *that* will fail with "dts is not a function".

What finally solved this for me was renaming to `.mjs` and removing the `package.json` require statement and just using strings to define the `esm / cjs` outputs.



Rodrigo Tolentino de Novais • Dec 13 '22



It's quite ugly but you can use `dts.default()` instead of `dts()` as a alternative



Ben • Nov 19 '21



I really appreciate that you've summarised this process.

I wonder what would you recommend as a workflow to actually author those components?

What I mean is that I'd never develop a design system in isolation. I usually need my whole app/website to see the context and make adjustments if necessary, but - obviously - I don't want to include any of that into the final commit of the design system.

So what do you recommend to seamlessly bring together the development of a main application and the belonging design system locally?



Alex Eagleson 🌟 • Nov 19 '21 • Edited on



That's a great question, I want to make sure I'm understanding it properly.

I can only speak personally, since we work with a designer, all of our design work is done in advance in [Figma](#) so the overall picture & theming of everything can be designed together, then I turn those designs into components and use Storybook to validate that the components match the intention of the designs.

If I were working alone or without a designer I would probably create a few helper components that are meant to group components together (but not actually exported as part of the library). Something that would allow me to wrap unrelated components and compare them in proximity with one another with some padding between. Something along the lines of the [Stack](#) from Material-UI (which is basically just a convenient wrapper around a flex element). Then I could drop all my components inside of it and use Storybook to make a story where I can see them.

That said, since I am not skilled at design (and don't pretend to be) I would definitely encourage anyone else who has an opinion on this to respond as well!

Cheers.



Ben • Dec 3 '21



Thanks Alex sharing your take on this.

I also encourage everyone to read this article from the authors of [Tailwind CSS / UI](#).

They've explained their process of designing components and then turning them into code.

Their high level overview with your practical article I think is a good starting point.



Nathan Pham • Aug 29 '22



literal gold



Kristin Bradley • Feb 1 '22 • Edited on



Great article! Very clear and easy to follow.

When importing the library I created into my test application, it worked great except I got this error: TS7016: Could not find a declaration file for module ... implicitly has an 'any' type.

It looks like I need to declare the module for my components but I'm unsure where to do this? I want to add this in the component library itself so users importing it won't see this error.



Alex Eagleson 🌟 • Feb 1 '22 • Edited on



The output of the declaration files is covered in the "Adding Rollup" section. From this part of the rollup config in particular:

```
{
  input: "dist/esm/types/index.d.ts",
  output: [{ file: "dist/index.d.ts", format: "esm" }],
```

```
plugins: [dts()],  
},
```



Kristin Bradley • Feb 2 '22



Ok thanks. I double checked and I have that in my rollup.config.js file.



Samiul H. Sohan • Apr 6 '22 • Edited on



Great tutorial, [@alexeagleson](#) !

I am facing a problem. Whenever I'm using things like useState in the library and then importing in my Next.js project, it's giving me error.

TypeError: Cannot read properties of null (reading 'useState')



1 of 1 unhandled error

Server Error

TypeError: Cannot read properties of null (reading 'useState')

This error happened while generating the page. Any console logs will be displayed in the terminal window.

Source

../../packages/roshogolla/dist/cjs/index.js (10:25492) @ Object.t.useState



```
8 | * This source code is licensed under the MIT license found in the  
9 | * LICENSE file in the root directory of this source tree.  
> 10 | */!function(e,t){"production"!==process.env.NODE_ENV&&function(){"undefined"!==typeof _  
    |  
11 | /** @license React v16.13.1  
12 | * react-is.development.js  
13 | *
```

Call Stack

exports.PSwitch

../../packages/roshogolla/dist/cjs/index.js (23:19933)

Show collapsed frames

And if I move react inside the peerDependencies , it shows this error:


 1 of 4 unhandled errors **Unhandled Runtime Error**

Error: Invalid hook call. Hooks can only be called inside of the body of a function component. This could happen for one of the following reasons:

1. You might have mismatching versions of React and the renderer (such as React DOM)
2. You might be breaking the Rules of Hooks
3. You might have more than one copy of React in the same app

See <https://reactjs.org/link/invalid-hook-call> for tips about how to debug and fix this problem.

Source

../../packages/roshogolla/dist/esm/index.js (1:116) @ c 

```
> 1 | import e,{useState as t}from"react";function n(t){return e.createElement("button",null,t
    |
    2 |   //# sourceMappingURL=index.js.map
    3 | }
```

Show collapsed frames



Matthias Nannt • May 31 '22 



Hi [@samiulhsohan](#)

I faced the same problem and found a solution. This is what causes the error:
blog.maximeheckel.com/posts/duplic...

And this is my `next.config.js` that solves the issue:

```
var path = require("path");

module.exports = {
  webpack: (config, { buildId, dev, isServer, defaultLoaders, webpack }) => {
    config.resolve.alias["react"] = path.resolve("./node_modules/react");
    return config;
  },
};
```



Lucas Silva de Jesus • May 11 '22 



Hi [@samiulhsohan](#)!

I'm facing the same problem in my react application. Did u solved it??

[View full discussion \(133 comments\)](#)

Some comments have been hidden by the post's author - [find out more](#)

[Code of Conduct](#) • [Report abuse](#)

All DEV content is created by the community!

Hey, if you're landing here for the first time, you should know that this website is a global community of folks who blog about their experiences to help folks like you out.

[Sign up now](#) if you're curious. It's free!



Alex Eagleson

Fullstack developer specializing in React, Typescript, Node & Rust.

LOCATION

Barrie, Ontario

EDUCATION

University of Guelph

WORK

Rust Developer @ 1Password

JOINED

Jun 27, 2021

More from [Alex Eagleson](#)

How to Set Up a Fullstack Rust Project with Axum, React, Vite, and Shared Types

[#rust](#) [#typescript](#) [#react](#) [#vite](#)

How to Build a Rust CLI Tool to Generate Typescript Types from Rust

[#rust](#) [#typescript](#) [#tutorial](#) [#opensource](#)

The Complete Beginner's Guide to a Web Development Career

[#javascript](#) [#webdev](#) [#beginners](#) [#tutorial](#)