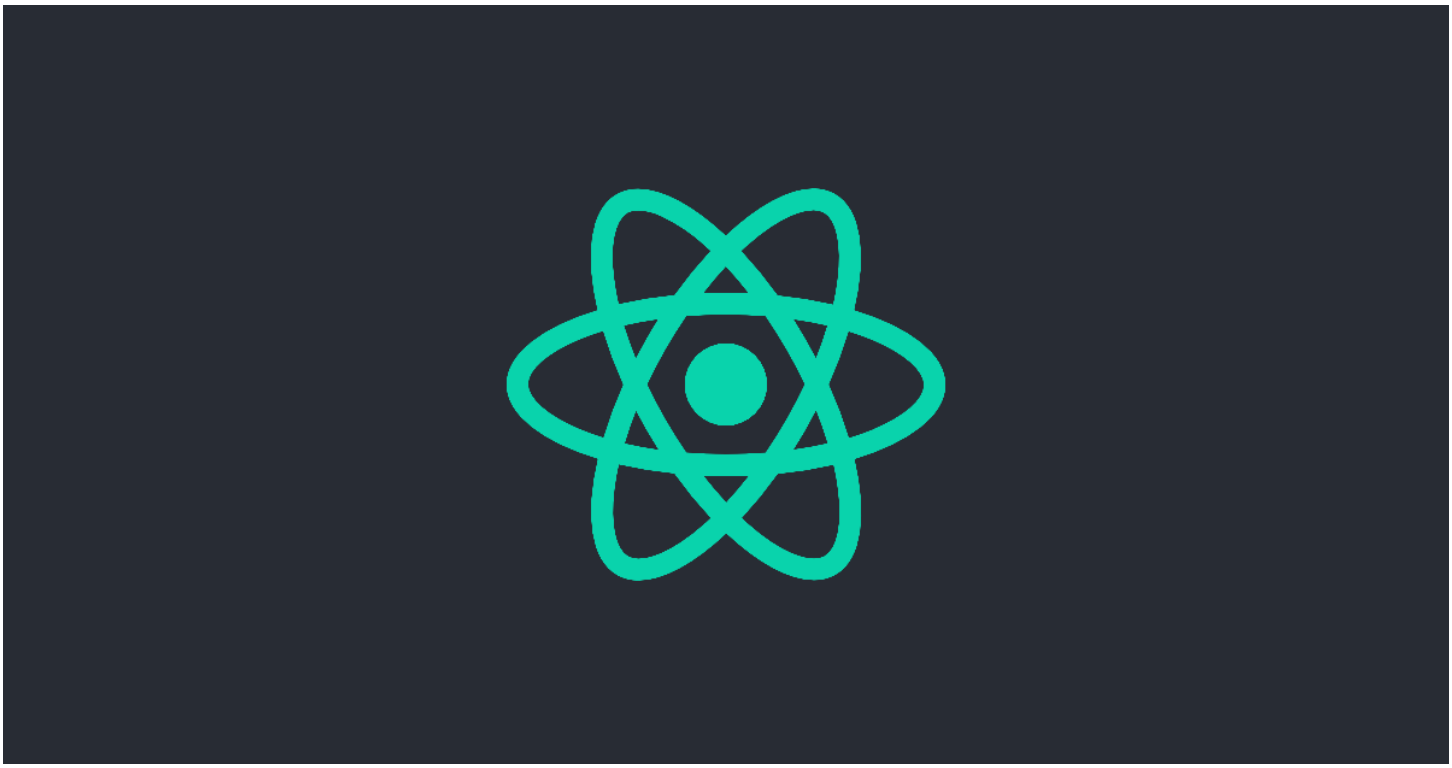
# Deployment



`npm run build` creates a `build` directory with a production build of your app. Set up your favorite HTTP server so that a visitor to your site is served `index.html`, and requests to static paths like `/static/js/main.<hash>.js` are served with the contents of the `/static/js/main.<hash>.js` file. For more information see the production build section.

## Static Server

For environments using Node, the easiest way to handle this would be to install serve and let it handle the rest:

```
npm install -g serve
serve -s build
```

The last command shown above will serve your static site on the port **3000**. Like many of serve's internal settings, the port can be adjusted using the `-l` or `--listen` flags:

```
serve -s build -l 4000
```

Run this command to get a full list of the options available:

```
serve -h
```

## Other Solutions

You don't necessarily need a static server in order to run a Create React App project in production. It also works well when integrated into an existing server side app.

Here's a programmatic example using Node and Express:

```
const express = require('express');
const path = require('path');
const app = express();

app.use(express.static(path.join(__dirname, 'build')));

app.get('/', function (req, res) {
  res.sendFile(path.join(__dirname, 'build', 'index.html'));

app.listen(9000);
```

The choice of your server software isn't important either. Since Create React App is completely platform-agnostic, there's no need to explicitly use Node.

The `build` folder with static assets is the only output produced by Create React App.

However this is not quite enough if you use client-side routing. Read the next section if you want to support URLs like `/todos/42` in your single-page app.

## Serving Apps with Client-Side Routing

If you use routers that use the HTML5 `pushState` history API under the hood (for example, React Router with `browserHistory`), many static file servers will fail. For example, if you used React Router with a route for `/todos/42`, the development server will respond to `localhost:3000/todos/42` properly, but an Express serving a production build as above will not.

This is because when there is a fresh page load for a `/todos/42`, the server looks for the file `build/todos/42` and does not find it. The server needs to be configured to respond to a request to `/todos/42` by serving `index.html`. For example, we can amend our Express example above to serve `index.html` for any unknown paths:

```
 app.use(express.static(path.join(__dirname, 'build')));

-app.get('/', function (req, res) {
+app.get('/*', function (req, res) {
   res.sendFile(path.join(__dirname, 'build', 'index.html'));
 });
```

If you're using Apache HTTP Server, you need to create a `.htaccess` file in the `public` folder that looks like this:

```
    Options -MultiViews
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^ index.html [QSA,L]
```

It will get copied to the `build` folder when you run `npm run build`.

If you're using Apache Tomcat, you need to follow this Stack Overflow answer.

Now requests to `/todos/42` will be handled correctly both in development and in production.

On a production build, and when you've opted-in, a service worker will automatically handle all navigation requests, like for `/todos/42`, by serving the cached copy of your `index.html`. This service worker navigation routing can be configured or disabled by `ejecting` and then modifying the `navigateFallback` and `navigateFallbackWhitelist` options of the `SWPrecachePlugin` configuration.

When users install your app to the homescreen of their device the default configuration will make a shortcut to `/index.html`. This may not work for client-side routers which expect the app to be served from `/`. Edit the web app manifest at `public/manifest.json` and change `start_url` to match the required URL scheme, for example:

```
  "start_url": ".",
```

## Building for Relative Paths

By default, Create React App produces a build assuming your app is hosted at the server root.

To override this, specify the `homepage` in your `package.json`, for example:

```
  "homepage": "http://mywebsite.com/relativepath",
```

This will let Create React App correctly infer the root path to use in the generated HTML file.

**Note**: If you are using `react-router@^4`, you can root `<Link>`s using the `basename` prop on any `<Router>`.

More information here.

For example:

```
<BrowserRouter basename="/calendar"/>
<Link to="/today"/> // renders <a href="/calendar/today">
```

### Serving the Same Build from Different Paths

> Note: this feature is available with `react-scripts@0.9.0` and higher.

If you are not using the HTML5 `pushState` history API or not using client-side routing at all, it is unnecessary to specify the URL from which your app will be served. Instead, you can put this in your `package.json`:

```
  "homepage": ".",
```

This will make sure that all the asset paths are relative to `index.html`. You will then be able to move your app from `http://mywebsite.com` to `http://mywebsite.com/relativepath` or even `http://mywebsite.com/relative/path` without having to rebuild it.

## Customizing Environment Variables for Arbitrary Build Environments

You can create an arbitrary build environment by creating a custom `.env` file and loading it using env-cmd.

For example, to create a build environment for a staging environment:

1. Create a file called `.env.staging`
2. Set environment variables as you would any other `.env` file (e.g. `REACT_APP_API_URL=http://api-staging.example.com`)
3. Install env-cmd

   ```
   $ npm install env-cmd --save
   $ # or
   $ yarn add env-cmd
   ```

4. Add a new script to your `package.json`, building with your new environment:

```
{
  "scripts": {
    "build:staging": "env-cmd -f .env.staging npm run build"
}
```

Now you can run `npm run build:staging` to build with the staging environment config. You can specify other environments in the same way.

Variables in `.env.production` will be used as fallback because `NODE_ENV` will always be set to `production` for a build.

## AWS Amplify

The AWS Amplify Console provides continuous deployment and hosting for modern web apps (single page apps and static site generators) with serverless backends. The Amplify Console offers globally available CDNs, custom domain setup, feature branch deployments, and password protection.

1. Login to the Amplify Console here.
2. Connect your Create React App repo and pick a branch. If you're looking for a Create React App+Amplify starter, try the create-react-app-auth-amplify starter that demonstrates setting up auth in 10 minutes with Create React App.
3. The Amplify Console automatically detects the build settings. Choose Next.
4. Choose *Save and deploy*.

If the build succeeds, the app is deployed and hosted on a global CDN with an amplifyapp.com domain. You can now continuously deploy changes to your frontend or backend. Continuous deployment allows developers to deploy updates to their frontend and backend on every code commit to their Git repository.

## Azure

Azure Static Web Apps creates an automated build and deploy pipeline for your React app powered by GitHub Actions. Applications are geo-distributed by default with multiple points of presence. PR's are built automatically for staging environment previews.

1. Create a new Static Web App here.
2. Add in the details and connect to your GitHub repo.
3. Make sure the build folder is set correctly on the "build" tab and create the resource.

Azure Static Web Apps will automatically configure a GitHub Action in your repo and begin the deployment.

See the Azure Static Web Apps documentation for more information on routing, APIs, authentication and authorization, custom domains and more.

## Firebase

Install the Firebase CLI if you haven't already by running `npm install -g firebase-tools`. Sign up for a Firebase account and create a new project. Run `firebase login` and login with your previous created Firebase account.

Then run the `firebase init` command from your project's root. You need to choose the **Hosting: Configure and deploy Firebase Hosting sites** and choose the Firebase project you created in the previous step. You will need to agree with `database.rules.json` being created, choose `build` as the public directory, and also agree to **Configure as a single-page app** by replying with `y`.

```
    === Project Setup

    First, let's associate this project directory with a Firebase project.
    You can create multiple project aliases by running firebase use --add,
    but for now we'll set up a default project.

    ? What Firebase project do you want to associate as default? Example app
(example-app-fd690)

    === Database Setup

    Firebase Realtime Database Rules allow you to define how your data should
be
    structured and when your data can be read from and written to.

    ? What file should be used for Database Rules? database.rules.json
    ✔  Database Rules for example-app-fd690 have been downloaded to
database.rules.json.
    Future modifications to database.rules.json will update Database Rules
when you run
    firebase deploy.

    === Hosting Setup

    Your public directory is the folder (relative to your project directory)
that
    will contain Hosting assets to uploaded with firebase deploy. If you
    have a build process for your assets, use your build's output directory.

    ? What do you want to use as your public directory? build
```

```
    ? Configure as a single-page app (rewrite all urls to /index.html)? Yes
    ✔  Wrote build/index.html

    i  Writing configuration info to firebase.json...
    i  Writing project information to .firebaserc...

    ✔  Firebase initialization complete!
```

IMPORTANT: you need to set proper HTTP caching headers for `service-worker.js` file in `firebase.json` file or you will not be able to see changes after first deployment (issue #2440). It should be added inside "`hosting`" key like next:

```
{
  "hosting": {
    ...
    "headers": [
      {"source": "/service-worker.js", "headers": [{"key": "Cache-Control",
"value": "no-cache"}]}

    ...
```

Now, after you create a production build with `npm run build`, you can deploy it by running `firebase deploy`.

```
    === Deploying to 'example-app-fd690'...

    i  deploying database, hosting
    ✔  database: rules ready to deploy.
    i  hosting: preparing build directory for upload...
    Uploading: [==============================          ] 75%✔  hosting:
build folder uploaded successfully
    ✔  hosting: 8 files uploaded successfully
    i  starting release process (may take several minutes)...

    ✔  Deploy complete!

    Project Console: https://console.firebase.google.com/project/example-app-
fd690/overview
    Hosting URL: https://example-app-fd690.firebaseapp.com
```

For more information see Firebase Hosting.

# GitHub Pages

> Note: this feature is available with `react-scripts@0.2.0` and higher.

### Step 1: Add `homepage` to `package.json`

**The step below is important!**

**If you skip it, your app will not deploy correctly.**

Open your `package.json` and add a `homepage` field for your project:

```
  "homepage": "https://myusername.github.io/my-app",
```

or for a GitHub user page:

```
  "homepage": "https://myusername.github.io",
```

or for a custom domain page:

```
  "homepage": "https://mywebsite.com",
```

Create React App uses the `homepage` field to determine the root URL in the built HTML file.

### Step 2: Install `gh-pages` and add `deploy` to `scripts` in `package.json`

Now, whenever you run `npm run build`, you will see a cheat sheet with instructions on how to deploy to GitHub Pages.

To publish it at https://myusername.github.io/my-app, run:

```
npm install --save gh-pages
```

Alternatively you may use `yarn`:

```
yarn add gh-pages
```

Add the following scripts in your `package.json`:

```
  "scripts": {
+   "predeploy": "npm run build",
+   "deploy": "gh-pages -d build",
    "start": "react-scripts start",
    "build": "react-scripts build",
```

The `predeploy` script will run automatically before `deploy` is run.

If you are deploying to a GitHub user page instead of a project page you'll need to make one additional modification:

1. Tweak your `package.json` scripts to push deployments to **master**:

```
  "scripts": {
    "predeploy": "npm run build",
-   "deploy": "gh-pages -d build",
+   "deploy": "gh-pages -b master -d build",
```

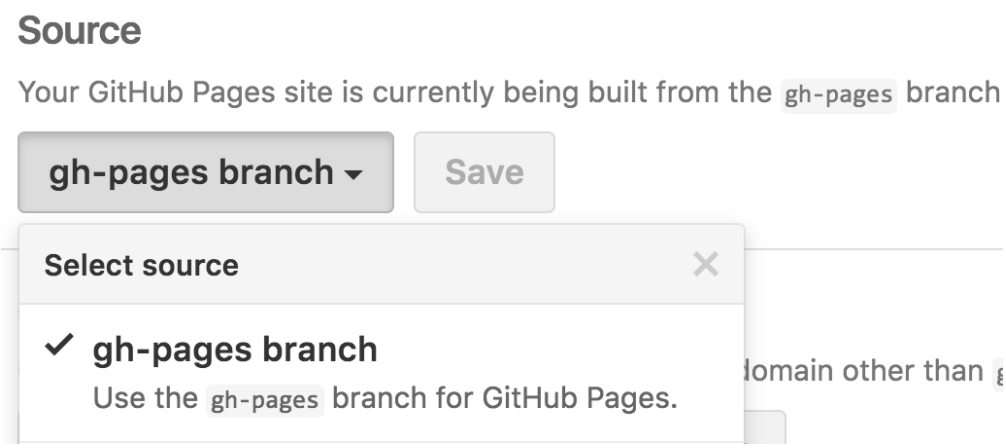
### Step 3: Deploy the site by running `npm run deploy`

Then run:

```
npm run deploy
```

### Step 4: For a project page, ensure your project's settings use `gh-pages`

Finally, make sure **GitHub Pages** option in your GitHub project settings is set to use the `gh-pages` branch:




### Step 5: Optionally, configure the domain

You can configure a custom domain with GitHub Pages by adding a `CNAME` file to the `public/` folder.

Your CNAME file should look like this:

```
mywebsite.com
```

### Notes on client-side routing

GitHub Pages doesn't support routers that use the HTML5 `pushState` history API under the hood (for example, React Router using `browserHistory`). This is because when there is a fresh page load for a url like `http://user.github.io/todomvc/todos/42`, where `/todos/42` is a frontend route, the GitHub Pages server returns 404 because it knows nothing of `/todos/42`. If you want to add a router to a project hosted on GitHub Pages, here are a couple of solutions:

- You could switch from using HTML5 history API to routing with hashes. If you use React Router, you can switch to `hashHistory` for this effect, but the URL will be longer and more verbose (for example, `http://user.github.io/todomvc/#/todos/42?_k=yknaj`). Read more about different history implementations in React Router.
- Alternatively, you can use a trick to teach GitHub Pages to handle 404s by redirecting to your `index.html` page with a custom redirect parameter. You would need to add a `404.html` file with the redirection code to the `build` folder before deploying your project, and you'll need to add code handling the redirect parameter to `index.html`. You can find a detailed explanation of this technique in this guide.

### Troubleshooting

#### "/dev/tty: No such a device or address"

If, when deploying, you get `/dev/tty: No such a device or address` or a similar error, try the following:

1. Create a new Personal Access Token
2. `git remote set-url origin https://<user>:<token>@github.com/<user>/<repo>`.
3. Try `npm run deploy` again

#### "Cannot read property 'email' of null"

If, when deploying, you get `Cannot read property 'email' of null`, try the following:

1. `git config --global user.name '<your_name>'`
2. `git config --global user.email '<your_email>'`
3. Try `npm run deploy` again

## Heroku

Use the Heroku Buildpack for Create React App.

You can find instructions in Deploying React with Zero Configuration.

### Resolving Heroku Deployment Errors

Sometimes `npm run build` works locally but fails during deploy via Heroku. Following are the most common cases.

**"Module not found: Error: Cannot resolve 'file' or 'directory'"**

If you get something like this:

```
remote: Failed to create a production build. Reason:
remote: Module not found: Error: Cannot resolve 'file' or 'directory'
MyDirectory in /tmp/build_1234/src
```

It means you need to ensure that the lettercase of the file or directory you `import` matches the one you see on your filesystem or on GitHub.

This is important because Linux (the operating system used by Heroku) is case sensitive. So `MyDirectory` and `mydirectory` are two distinct directories and thus, even though the project builds locally, the difference in case breaks the `import` statements on Heroku remotes.

**"Could not find a required file."**

If you exclude or ignore necessary files from the package you will see a error similar this one:

```
remote: Could not find a required file.
remote:   Name: `index.html`
remote:   Searched in: /tmp/build_a2875fc163b209225122d68916f1d4df/public
remote:
remote: npm ERR! Linux 3.13.0-105-generic
remote: npm ERR! argv
"/tmp/build_a2875fc163b209225122d68916f1d4df/.heroku/node/bin/node"
"/tmp/build_a2875fc163b209225122d68916f1d4df/.heroku/node/bin/npm" "run"
"build"
```

In this case, ensure that the file is there with the proper lettercase and that's not ignored on your local `.gitignore` or `~/.gitignore_global`.

## Netlify

**To do a manual deploy to Netlify's CDN:**

```
npm install netlify-cli -g
netlify deploy
```

Choose `build` as the path to deploy.

**To setup continuous delivery:**

With this setup Netlify will build and deploy when you push to git or open a pull request:

1. Start a new netlify project
2. Pick your Git hosting service and select your repository
3. Click `Build your site`

**Support for client-side routing:**

To support `pushState`, make sure to create a `public/_redirects` file with the following rewrite rules:

```
/*  /index.html  200
```

When you build the project, Create React App will place the `public` folder contents into the build output.

## Vercel

Vercel is a cloud platform that enables developers to host Jamstack websites and web services that deploy instantly, scale automatically, and requires no supervision, all with zero configuration. They provide a global edge network, SSL encryption, asset compression, cache invalidation, and more.

### Step 1: Deploying your React project to Vercel

To deploy your React project with a Vercel for Git Integration, make sure it has been pushed to a Git repository.

Import the project into Vercel using the Import Flow. During the import, you will find all relevant options preconfigured for you with the ability to change as needed.

After your project has been imported, all subsequent pushes to branches will generate Preview Deployments, and all changes made to the Production Branch (commonly "master" or "main") will result in a Production Deployment.

Once deployed, you will get a URL to see your app live, such as the following: https://create-react-app-example.vercel.app/.

### Step 2 (optional): Using a Custom Domain

If you want to use a Custom Domain with your Vercel deployment, you can **Add** or **Transfer in** your domain via your Vercel account Domain settings.

To add your domain to your project, navigate to your Project from the Vercel Dashboard. Once you have selected your project, click on the "Settings" tab, then select the **Domains** menu item. From your projects **Domain** page, enter the domain you wish to add to your project.

Once the domain has been added, you will be presented with different methods for configuring it.

### Deploying a fresh React project

You can deploy a fresh React project, with a Git repository set up for you, with the following Deploy Button:


### Vercel References:

## Render

Render offers free static site hosting with fully managed SSL, a global CDN and continuous auto deploys from GitHub.

Deploy your app in only a few minutes by following the Create React App deployment guide.

Use invite code `cra` to sign up or use this link.

## S3 and CloudFront

See this blog post on how to deploy your React app to Amazon Web Services S3 and CloudFront. If you are looking to add a custom domain, HTTPS and continuous deployment see this blog post.

## Surge

Install the Surge CLI if you haven't already by running `npm install -g surge`. Run the `surge` command and log in you or create a new account.

When asked about the project path, make sure to specify the `build` folder, for example:

```
        project path: /path/to/project/build
```

Note that in order to support routers that use HTML5 `pushState` API, you may want to rename the `index.html` in your build folder to `200.html` before deploying to Surge. This ensures that every URL falls back to that file.

# Publishing Components To npm

Create React App doesn't provide any built-in functionality to publish a component to npm. If you're ready to extract a component from your project so other people can use it, we recommend moving it to a separate directory outside of your project and then using a tool like nwb to prepare it for publishing.