

Section 24: Deploying React Apps

21 May 2025 12:03



Deploying React Apps

From Development To Production



- ▶ Deployment **Steps & Pitfalls**
- ▶ **Server-side** Routing vs **Client-side** Routing

How Do You Deploy a React Application?

So, how do you deploy a React application?

How do you push it to a real server?

There are a few steps involved, which you'll follow every time you deploy or redeploy your app.

1. Write and Test Your Code

It all starts with writing your code, of course — but also **testing** it thoroughly.

This might sound obvious, but you *really* want to test your application before deploying it.

- Play around with different features.
- Test how it handles errors.
- Make sure everything works as expected.

You want to ship an application that's **production-ready**.

2. Optimize Your Code

Before moving forward, explore optimization opportunities.

Look for parts of your code that can be improved.

One major technique is **lazy loading**, which we'll explore in this course section.

This step ensures your code is lean and efficient before it's deployed.

3. Build the App for Production

Once you're happy with the code and it's optimized, it's time to **build** the app for production.

You're not writing more code — you're running a script that's already included in your React project.

This script creates a **production-ready bundle**:

- Unified
- Minified

- Automatically optimized

This bundle is what you'll deploy to your server. It's small, efficient, and fast — which is crucial because users can only interact with your app once it's fully loaded.

4. Deploy the Build

Now that you have the optimized build, it's time to deploy it.

You'll:

- Take the generated production code.
- Upload it to a server.

There are many hosting options — and we'll go through an example in this course section. But really, you can deploy it almost anywhere.

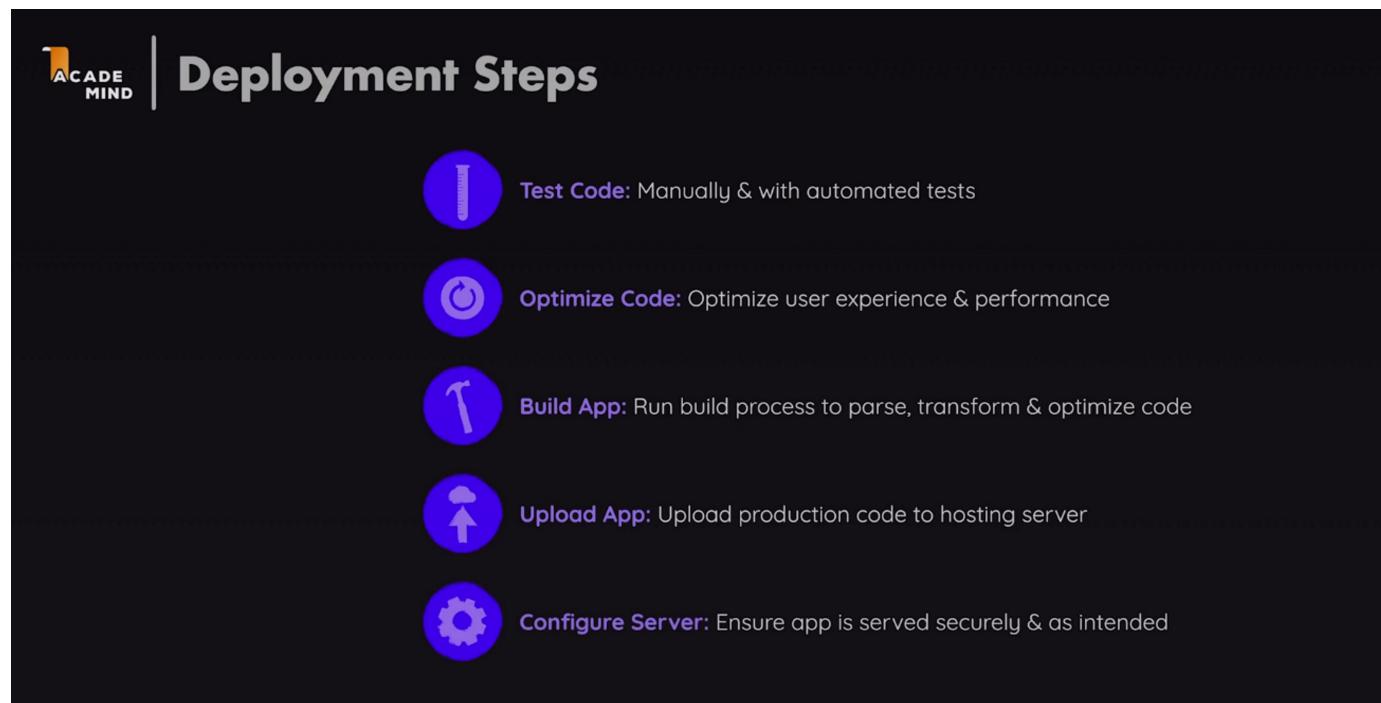
5. Configure the Server

Once uploaded, you'll need to **configure your server** (or hosting provider).

We'll dive into:

- What configs matter
- What you should pay attention to
- How to do it correctly

This way, you'll have a clear idea of how to set things up and successfully deploy your React app to the web.



Understanding Lazy Loading

Welcome to This Course Section!

For this part of the course, I've prepared a **brand-new example application** — and yep, you'll find

it attached!

In this app:

- We fetch a list of **dummy blog posts**
- You can view the **details of each post**

It's super simple by design, but it covers important concepts like:

- **Routing** using react-router
- Multiple **routes** and **page components**
- Several **reusable components**

So yeah, while the app itself isn't complex, it uses key React features that you've been learning throughout the course. But remember — the app isn't the focus here.

The Real Goal: Deployment

This section is **all about deployment**.

And before we deploy, we need to **prepare the app** properly.

Let's assume:

- We've already tested the app thoroughly.
- Everything works as expected.
- We're happy with the code and don't want to make any more changes.

So, what's next?

Step: Code Optimization

Before building the final production version, we should **optimize** our code.

That's what this lecture is all about.

Now, there are various techniques we could use — including ones we already covered in earlier course sections like:

- **React.memo**
- Other behind-the-scenes optimization techniques

But instead of repeating those here, I want to introduce you to a **new technique** you *really* should know about:

Lazy Loading

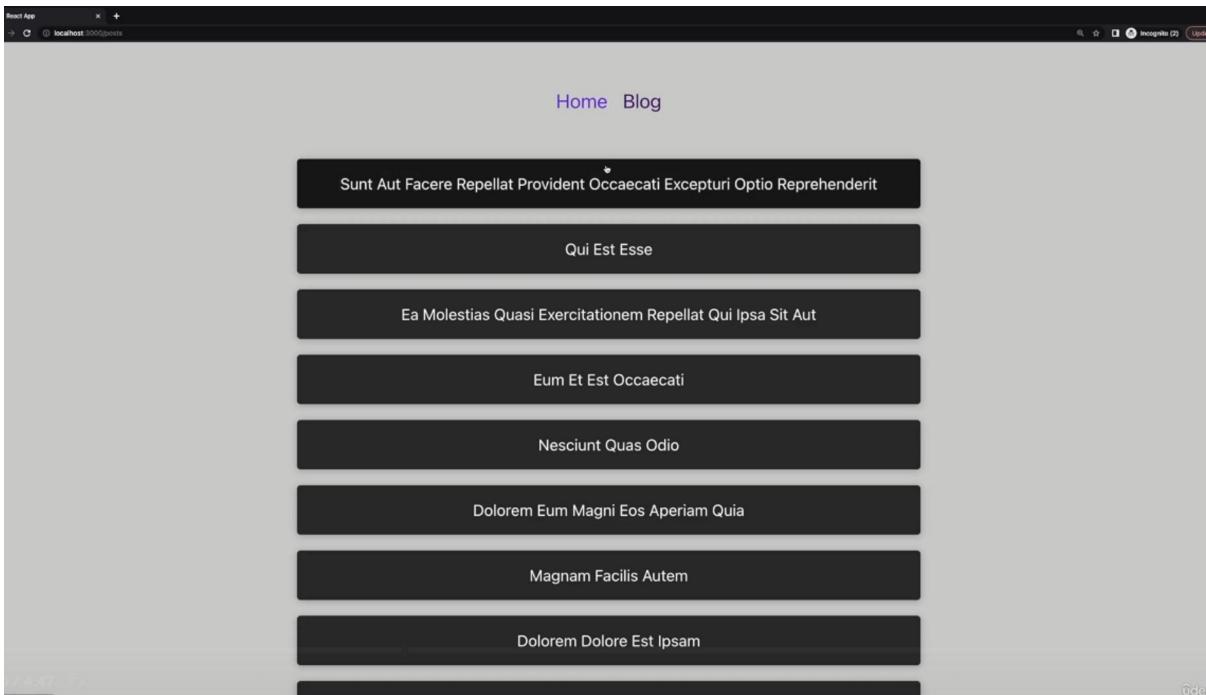
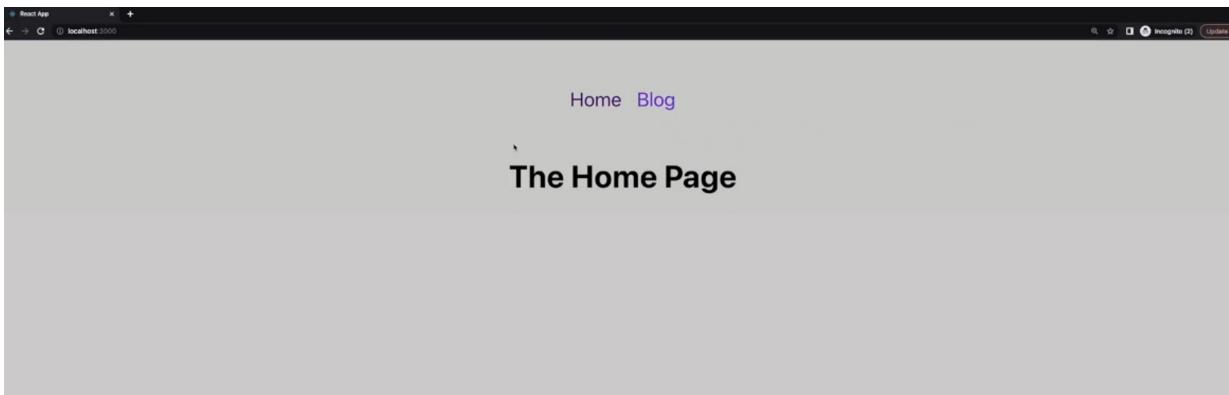
Lazy loading is all about loading parts of your app **only when they're actually needed**.

It's a game-changer for performance and user experience.

So, what exactly *is* lazy loading?

And how does it work in React?

 Stick with me, and we'll break it all down in the next part of this section.

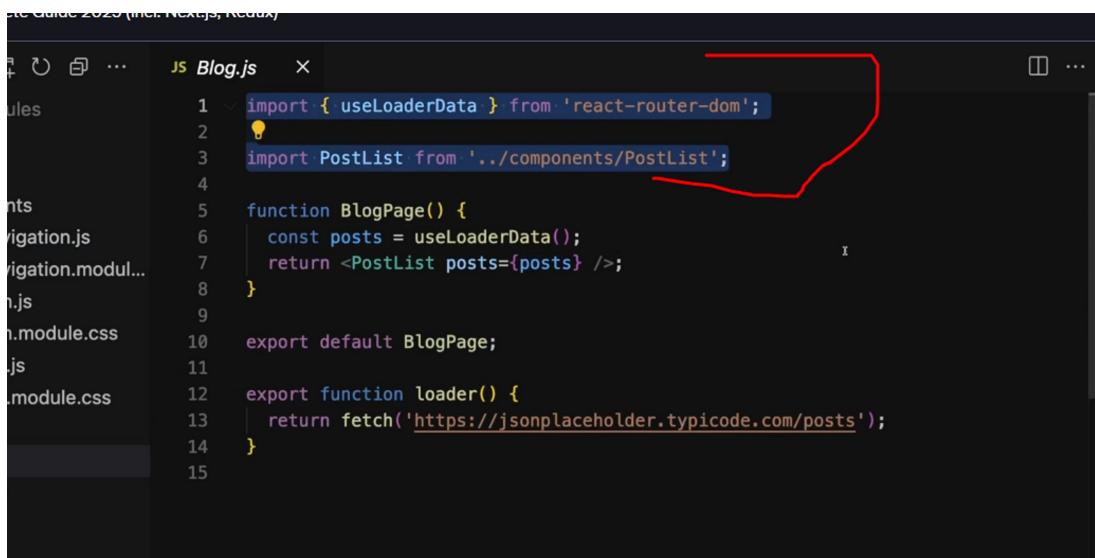


Lazy Loading

Load code only when it's needed

But what exactly is lazy loading and how does it work? To understand lazy loading, it's important to understand how this project here will be built and served to end users without lazy loading.

It's important to understand that we have all these import statement in our various files where we import code from other files into the file where the import statement was added.



```
use Guide 2023 (incl. Next.js, Redux)
JS Blog.js  X
1 import { useLoaderData } from 'react-router-dom';
2
3 import PostList from '../components/PostList';
4
5 function BlogPage() {
6   const posts = useLoaderData();
7   return <PostList posts={posts} />;
8 }
9
10 export default BlogPage;
11
12 export function loader() {
13   return fetch('https://jsonplaceholder.typicode.com/posts');
14 }
```

For example in this `Blog.js` file, I'm importing '`loader`' data from `react-router-dom`, and that simply means that when this component file here is evaluated by the browser, this code for this hook will be imported because this code is needed in order to handle this component.

```

... ⌂ ⌄ ⌁ ⌂ ⌃ ... JS Blog.js ×
node_modules
public
rc
components
MainNavigation.js
MainNavigation.module...
PostItem.js
PostItem.module.css
PostList.js
PostList.module.css
pages
Blog.js
Home.js
Post.js
Root.js
App.js

```

```

1 import { useLoaderData } from 'react-router-dom';
2
3 import PostList from '../components/PostList';
4
5 function BlogPage() {
6   const posts = useLoaderData();
7   return <PostList posts={posts} />;
8 }
9
10 export default BlogPage;
11
12 export function loader() {
13   return fetch('https://jsonplaceholder.typicode.com/posts');
14 }
15

```

All these imports in the end connect these different files, and when this application is served to end users, all these imports must be resolved before something's shown on the screen.

And when this application is served to end users, all these imports must be resolved before something is shown on the screen.

That hopefully also makes sense. If all these code pieces depend on each other all code must be resolved, before something can be shown to the end user.

When we build this application then all these imported files will actually be merged together into one big file.

That means all code files must be loaded before anything's shown on the screen.

In a complex application, having to load all the code initially will slow down that initial page load. When the user visits the website for the first time, all the code must be downloaded before anything's showing up on the screen.

And therefore that initial load can take quite long and leads to a bad user experience.

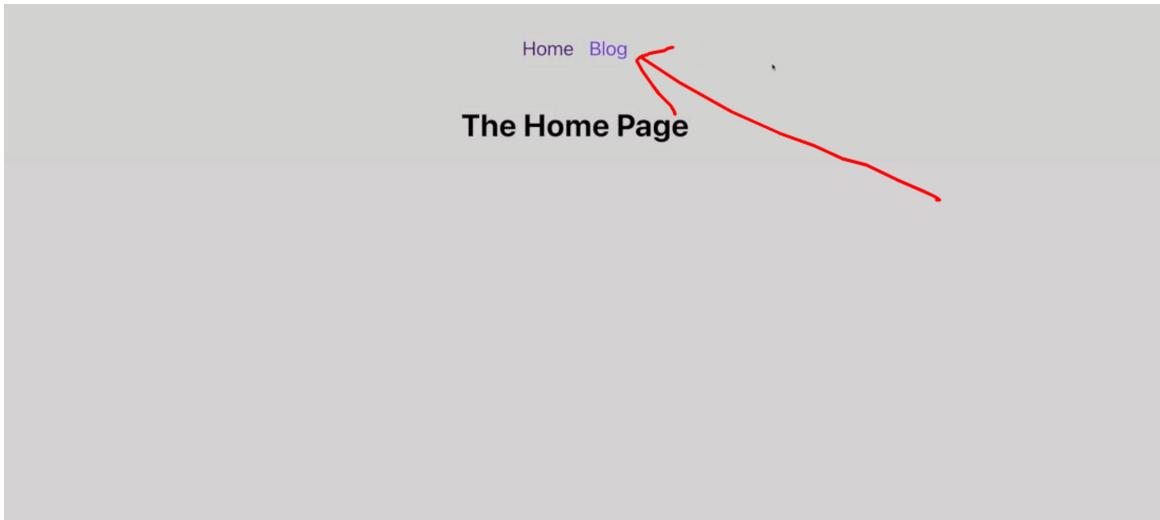
And that's exactly where lazy loading comes into play. The idea behind lazy loading is that we load certain components in the end only when they're needed instead of ahead of time.

Adding Lazy Loading

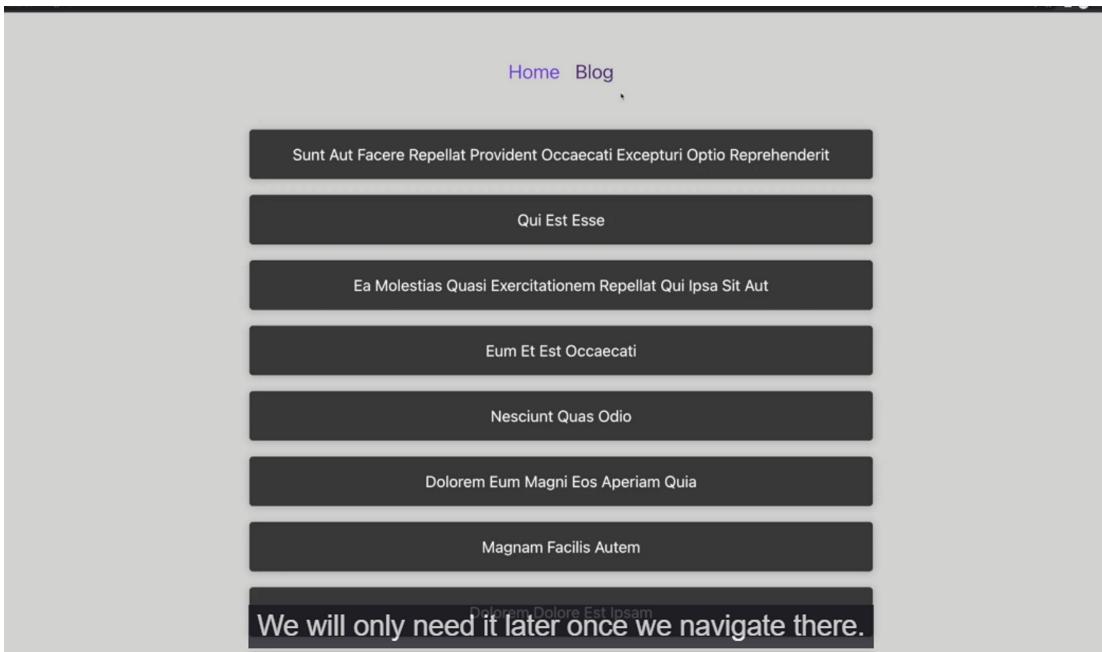
How do we add lazy loading?

Well let's say we wanna load that blow page lazily.

For example, when we visit this website and we land on the homepage. We don't need the blog page code yet.



We will only need it only once we navigate it. And at that point of time, once we navigate there it should be downloaded. And that's exactly what lazy loading will do for us.



In order to load this blog page lazily we first of all have to remove this import. Otherwise it's always loaded. We are actually importing two things which we should now load lazily.

The screenshot shows the VS Code interface with the following details:

- File Explorer:** On the left, it shows a project structure with files like node_modules, public, src (containing components, MainNavigation.js, PostItem.js, PostList.js, PostList.module.css), pages (containing Blog.js, Home.js, Post.js, Root.js), and App.js.
- Editor:** The main editor area is titled "JS App.js 1". A red arrow points to the status bar at the top right of the editor, which shows "1".
- Code:** The code in App.js is as follows:

```
1 import { createBrowserRouter, RouterProvider } from 'react-router-dom';
2
3 // import BlogPage, { loader as postsLoader } from './pages/Blog';
4 import HomePage from './pages/Home';
5 import PostPage, { loader as postLoader } from './pages/Post';
6 import RootLayout from './pages/Root';
7
8 // | const BlogPage
9
10 const router = createBrowserRouter([
11   {
12     path: '/',
13     element: <RootLayout />,
14     children: [
15       {
16         index: true,
17         element: <HomePage />,
18       },
19       {
20         path: 'posts',
21       }
22     ]
23   }
24 ])
```

At the bottom, the status bar shows "node" and other icons.

And it actually will start with the loader

```
EXP... ⌂ ⌂ ⌂ ⌂ ... JS App.js ●  
> node_modules  
> public  
└ src  
  └ components  
    JS MainNavigation.js  
    # MainNavigation.module.css  
    JS PostItem.js  
    # PostItem.module.css  
    JS PostList.js  
    # PostList.module.css  
  └ pages  
    JS Blog.js  
    JS Home.js  
    JS Post.js  
    JS Root.js  
  JS App.js  
14   children: [  
15     {  
16       index: true,  
17       element: <HomePage />,  
18     },  
19     {  
20       path: 'posts',  
21       children: [  
22         { index: true, element: <BlogPage />, loader: postLoader },  
23         { path: ':id', element: <PostPage />, loader: postLoader },  
24       ],  
25     },  
26   ],  
27 },  
28 );  
29  
30 function App() {  
31   return <RouterProvider router={router} />;  
32 }
```

Imports returns a promise.

```
JS App.js
12  / ,
13  t: <RootLayout />,
14  en: [
15
16  dex: true,
17  ement: <HomePage />,
18
19
20  th: 'posts',
21  ildren: [
22  { index: true, element: <BlogPage />, loader: () => import('./pages/Blog').then(module => module.loader()) },
23  { path: ':id', element: <PostPage />, loader: postLoader },
24
25
26
27
28
29
30  pp() {
31    RouterProvider router={router} />;
```

The next step is, to also load the blog page component lazily.

But wait, BlogPage is not a functional component. Because in react, a component returns JSX at the end. So for this problem, react gives us special {lazy}

```
JS App.js
1 | import { lazy } from 'react'; -----^
2 | import { createBrowserRouter, RouterProvider } from 'react-router-dom';
3 |
4 | // import BlogPage, { loader as postsLoader } from './pages/Blog';
5 | import HomePage from './pages/Home';
6 | import PostPage, { loader as postLoader } from './pages/Post';
7 | import RootLayout from './pages/Root';
8 |
9 | const BlogPage = lazy(() => import('./pages/Blog')); -----^
10
11 const router = createBrowserRouter([
12   {
13     path: '/',
14     element: <RootLayout />,
15     children: [
16       {
```

Here this code will work again at least almost. It will still take some time to load the code for this component because that code has to be downloaded after all, and the effort you must wrap this with another component provided by React Suspense component.

```
JS App.js ●
14   element: <RootLayout />,
15   children: [
16     {
17       index: true,
18       element: <HomePage />,
19     },
20     {
21       path: 'posts',
22       children: [
23         {
24           index: true,
25           element: <BlogPage />, ↖
26           loader: () =>
27             import('../pages/Blog').then((module) => module.loader()),
28         },
29         { path: ':id', element: <PostPage />, loader: postLoader },
30       ],
31     },
32   ],
33 ].
```

```
JS App.js ●
14   element: <RootLayout />,
15   children: [
16     {
17       index: true,
18       element: <HomePage />,
19     },
20     {
21       path: 'posts',
22       children: [
23         {
24           index: true,
25           element: [
26             <Suspense fallback=<p>Loading...</p>>
27               <BlogPage />
28             </Suspense>
29           ],
30           loader: () =>
31             import('../pages/Blog').then((module) => module.loader()),
32         },
33         { path: ':id', element: <PostPage />, loader: postLoader },
34       ],
35     },
36   ],
37 ].
```

The code has been modified to include a suspense component. A red bracket highlights the suspense block, and a red arrow points from the original 'element' assignment to the new suspense block. Another red arrow points from the 'BlogPage' component back to the original 'element' assignment.

Go to our page.

The screenshot shows a browser window with a React application titled "React App" at "localhost:3000". The page content includes a "Home" link and a "Blog" link. Below the links, the text "The Home Page" is displayed. To the right of the page content is the Chrome DevTools Network tab, which is active. The Network tab shows a single request: "Recording network activity...". Below the tab, instructions say "Perform a request or hit ⌘ R to record the reload." and a "Learn more" link.

Click on "blog".

The screenshot shows a browser window with the same React application at "localhost:3000". This time, the "Blog" link has been clicked, and the page content has changed to a list of blog posts. A red arrow points from the "Blog" link on the previous screen to the list of posts on this screen. The Network tab in DevTools shows several requests: "src_pages..." (304 script), "posts" (200 fetch), and "favicon.ico" (304 x-icon). The "posts" request is highlighted with a red box. The status bar at the bottom of the browser indicates "3 requests" and "52.6 kB resources". A tooltip "and then if you click on blog," is overlaid on the screen near the highlighted request.

This javascript file was downloaded dynamically because we add lazy loading.

We can also add individual blog post now For that, let me first of all show you how it looks like if we don't have lazy loading for them here on the blog page component.

If I click on a single post, we just have a request that downloads that post data from the backend.

A screenshot of a browser window showing a network request. The URL is `localhost:3000/posts/1`. The page content includes a header "Sunt Aut Facere Repellat Provident Occaecati Excepturi Optio Reprehenderit" and a paragraph of placeholder text. The network tab shows one request:

Name	Status	Type	Initiator	Size	Time	Waterfall
1	200	fetch	Post.js:15	3...	6...	<div style="width: 100%;">[progress bar]</div>
favicon.ico	304	x-icon	Other	3...	9...	<div style="width: 100%;">[progress bar]</div>

A red arrow points from the text "But we don't have request that would download the code for this component." to the initiator column of the first row.

But we don't have request that would download the code for this component.

A screenshot of a code editor showing the `App.js` file. The code defines a `createBrowserRouter` and sets up routes for `BlogPage` and `PostPage` using `lazy` components. Red arrows point from the text "But we don't have request that would download the code for this component." to the `PostPage` import statement and the `PostPage` component definition.

```
JS App.js
1 | import { lazy, Suspense } from 'react';
2 | import { createBrowserRouter, RouterProvider } from 'react-router-dom';
3 |
4 | // import BlogPage, { loader as postsLoader } from './pages/Blog';
5 | import HomePage from './pages/Home';
6 | // import PostPage, { loader as postLoader } from './pages/Post'; ←
7 | import RootLayout from './pages/Root';
8 |
9 | const BlogPage = lazy(() => import('./pages/Blog'));
10| const PostPage = lazy(() => import('./pages/Post')); ←
11|
12const router = createBrowserRouter([
13  {
14    path: '/',
15    element: <RootLayout />,
16    children: [
17      {
18        index: true,
19        element: <HomePage />,
20      },
21    ],
22  },
23]);
```

```
App.js
●
  element: (
    <Suspense fallback={<p>Loading...</p>}>
      <BlogPage />
    </Suspense>
  ),
  loader: () =>
    import('./pages/Blog').then((module) => module.loader()),
  },
  [
    {
      path: ':id',
      element: <PostPage />, // Red arrow pointing here
      loader: () =>
        import('./pages/Post').then((module) => module.loader()), // Red arrow pointing here
    },
  ],
),
],
].
```

```
s App.js 2 ×
25     index: true,
26     element: (
27       <Suspense fallback=<p>Loading...</p>>
28         | <BlogPage />
29       </Suspense>
30     ),
31     loader: () =>
32       import('./pages/Blog').then((module) => module.loader()),
33   },
34   {
35     path: ':id',
36     element: (
37       <Suspense fallback=<p>Loading...</p>>
38         | <PostPage />
39       </Suspense>
40     ),
41     loader: (meta) =>
42       import('./pages/Post').then((module) => module.loader(meta)),
43   },

```

Now let's see the difference.
Click on the post link.

A screenshot of a browser window showing a list of placeholder text items. The developer tools Network tab is open, displaying a single request for 'src_pages_Pos...'. A red arrow points from the placeholder text area down to the Network tab, and another red arrow points from the Network tab back up to the placeholder text area.

Sunt Aut Facere Repellat Provident Occaecati Excepturi Optio Reprehenderit

Qui Est Esse

Ea Molestias Quasi Exercitationem Repellat Qui Ipsa Sit Aut

Eum Et Est Occaecati

Nesciunt Quas Odio

Dolorem Eum Magni Eos Aperiam Quia

Magnam Facilis Autem

Dolorem Dolore Est Insam

Elements Console Network > ↗ 1 ⚙️ ⋮ ×

No throttling ⚙️ 🔍 Preset log ⌂ Disable cache

Filter Invert Hide data URLs

All Fetch/XHR JS CSS Img Media Font Doc WS Wasm Manifest Other

Has blocked cookies Blocked Requests 3rd-party requests

Recording network activity...

Perform a request or hit ⌘ R to record the reload.

[Learn more](#)

Now we can see the code for this file in downloaded dynamically.

A screenshot of a browser window showing a list of placeholder text items. The developer tools Network tab is open, displaying a list of requests. A red arrow points from the placeholder text area down to the Network tab, and another red arrow points from the Network tab back up to the placeholder text area.

Sunt Aut Facere Repellat Provident Occaecati
Excepturi Optio Reprehenderit

ia Et Suscipit Suscipit Recusandae Consequuntur Expedita Et Cum Reprehenderit
Molestiae Ut Ut Quas Totam Nostrum Rerum Est Autem Sunt Rem Eveniet
Architecto

Name St... Type Initiator S... T... Waterfall ▲

Name	St...	Type	Initiator	S...	T...	Waterfall
src_pages_Pos...	304	script	load scr...	3...	2...	<div style="width: 100px; height: 10px; background-color: #00AEEF;"></div>
1	200	fetch	Post.js:15	2...	3...	<div style="width: 100px; height: 10px; background-color: #00AEEF;"></div>
favicon.ico	304	x-icon	Other	3...	1...	<div style="width: 100px; height: 10px; background-color: #00AEEF;"></div>

Elements Console Network > ↗ 1 ⚙️ ⋮ ×

No throttling ⚙️ 🔍 Preset log ⌂ Disable cache

Filter Invert Hide data URLs

All Fetch/XHR JS CSS Img Media Font Doc WS Wasm Manifest Other

Has blocked cookies Blocked Requests 3rd-party requests

Building the Code for Production



Deployment Steps



Test Code: Manually & with automated tests



Optimize Code: Optimize user experience & performance



Build App: Run build process to parse, transform & optimize code

This is not the code we are going to upload. This is the code which we use during development.

The screenshot shows a code editor interface with the following details:

- File Structure:** The left sidebar shows a project structure with folders "components", "pages", and files like "MainNavigation.js", "PostItem.js", "PostList.js", "Blog.js", "Home.js", "Post.js", "Root.js", "App.js", "index.css", and "index.js".
- Code Editor:** The main area displays the content of the "App.js" file. The code is JSX and includes imports from "Suspense", "PostPage", and "Post". It defines a component structure with loader functions and suspense components.
- Bottom Bar:** The bottom bar includes tabs for "PROBLEMS", "DEBUG CONSOLE", "OUTPUT", "TERMINAL", and "...". It also has icons for "node", "terminal", and other development tools.

Even sometimes we use code which are not supported by the browser. Like this JSX code,

Which must be transformed before we can upload.

The screenshot shows the VS Code interface with the following details:

- File Explorer:** On the left, it shows a tree view of files and folders. The 'components' folder contains 'MainNavigation.js', 'PostItem.js', and 'PostList.js'. The 'pages' folder contains 'Blog.js', 'Home.js', 'Post.js', and 'Root.js'. Other files include 'App.js', 'index.css', 'index.js', '.gitignore', and 'package-lock.json'.
- Code Editor:** The main area displays the 'PostList.js' file. The code defines a functional component 'PostList' that takes an array of posts and returns an
 element. Each post is mapped to a - element containing a component with the 'to' prop set to the post's id.

```
1 import { Link } from 'react-router-dom';
2
3 import classes from './PostList.module.css';
4
5 function PostList({ posts }) {
6   return (
7     <ul className={classes.list}>
8       {posts.map((post) => (
9         <li key={post.id}>
10           <Link to={post.id.toString()}>{post.title}</Link>
11         </li>
12       ))}
13     </ul>
14   );
15 }
16
17 export default PostList;
```
- Bottom Bar:** The bottom bar includes tabs for PROBLEMS, DEBUG CONSOLE, OUTPUT, TERMINAL, and other developer tools.

We run

npm run build

Under the hood, this will produce a code bundle with highly optimized and transformed code
Which is ready to be uploaded.

The screenshot shows the VS Code interface with the following details:

- File Explorer:** On the left, it shows a tree view of files and folders, identical to the previous screenshot.
- Code Editor:** The main area displays the 'package.json' file. It contains a 'scripts' object with four keys: 'start', 'build', 'test', and 'eject'. It also contains an 'eslintConfig' object with an 'extends' array containing 'react-app'.

```
{
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app"
    ]
  }
}
```

- Terminal:** A terminal window at the bottom shows the command '\$ npm run build'.
- Bottom Bar:** The bottom bar includes tabs for PROBLEMS, DEBUG CONSOLE, OUTPUT, TERMINAL, and other developer tools.

A tooltip message appears in the bottom right corner: "In Vite-based projects, it's exactly the same => There is a 'build' script you must execute."

The screenshot shows a code editor interface with a sidebar on the left displaying a file tree. The tree includes 'build', 'static' (with 'css' and 'js' subfolders), and a selected 'js' folder containing files like '145.6ced6aef.chunk.js', 'main.926c437b.js', and 'manifest.json'. The main panel shows the content of '145.6ced6aef.chunk.js' with two lines of code: 'use strict';(self.webpackChunkreact_temp=self.webpackChunkreact_temp||[]).push and a comment about sourceMappingURL. Below the code, the terminal tab is active, showing deployment instructions: 'The project was built assuming it is hosted at /.', 'The build folder is ready to be deployed.', and 'You may serve it with a static server:' followed by the command 'npm install -g serve serve -s build'. A red box highlights the 'js' folder in the sidebar.

```
1 "use strict";(self.webpackChunkreact_temp=self.webpackChunkreact_temp||[]).push
2 //# sourceMappingURL=145.6ced6aef.chunk.js.map
```

PROBLEMS DEBUG CONSOLE OUTPUT TERMINAL ...

115 B build/static/css/229.ee7d7335.chunk.css

The project was built assuming it is hosted at /.
You can control this with the homepage field in your package.json.

The build folder is ready to be deployed.
You may serve it with a static server:

npm install -g serve
serve -s build

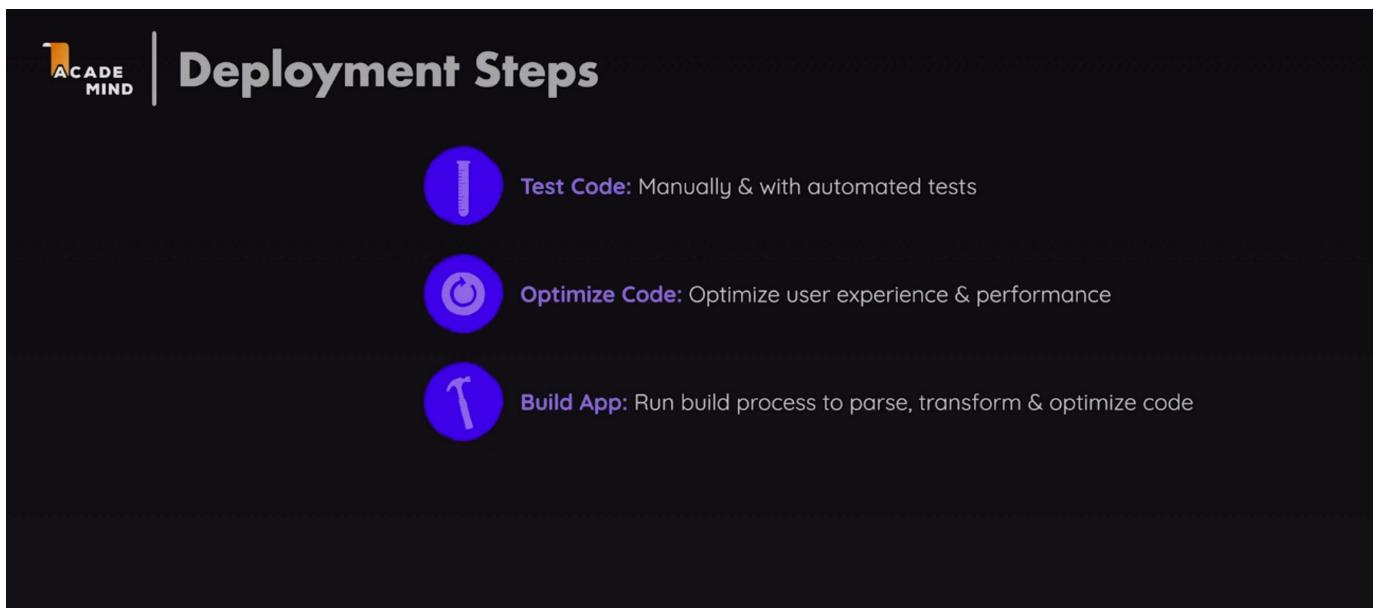
Find out more about deployment here:

Here is our optimized javascript code.

Deployment Example:

We built the app for the production and let's upload it to the server.

Let's deploy it.





A React SPA is a “Static Website”

Only HTML, CSS & JavaScript



A React SPA is a “Static Website”

Only HTML, CSS & JavaScript



A static site host is needed

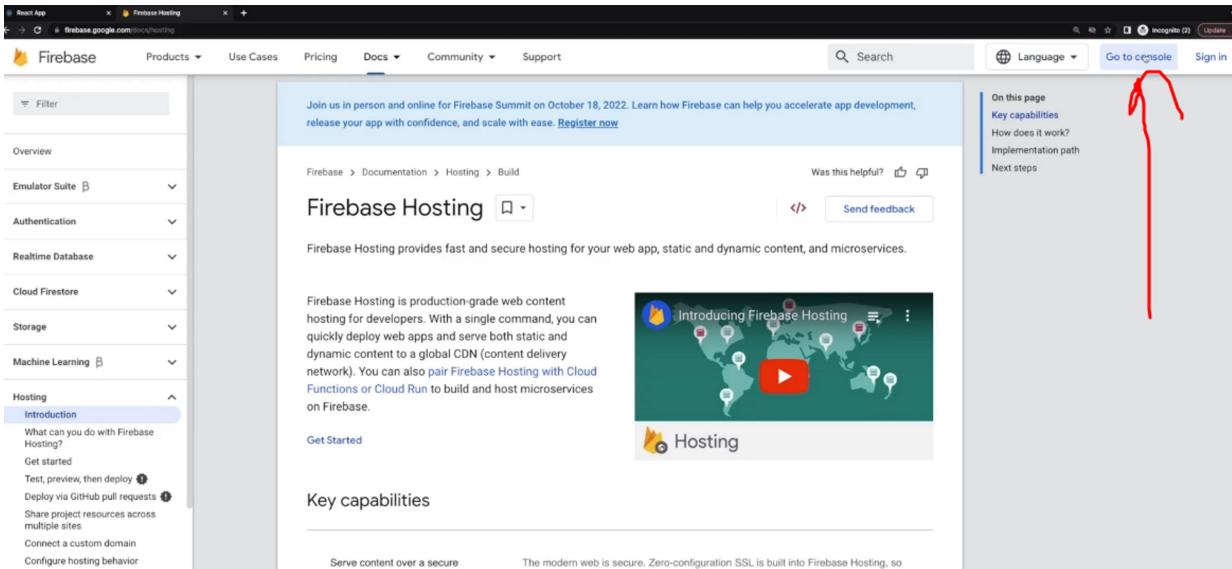
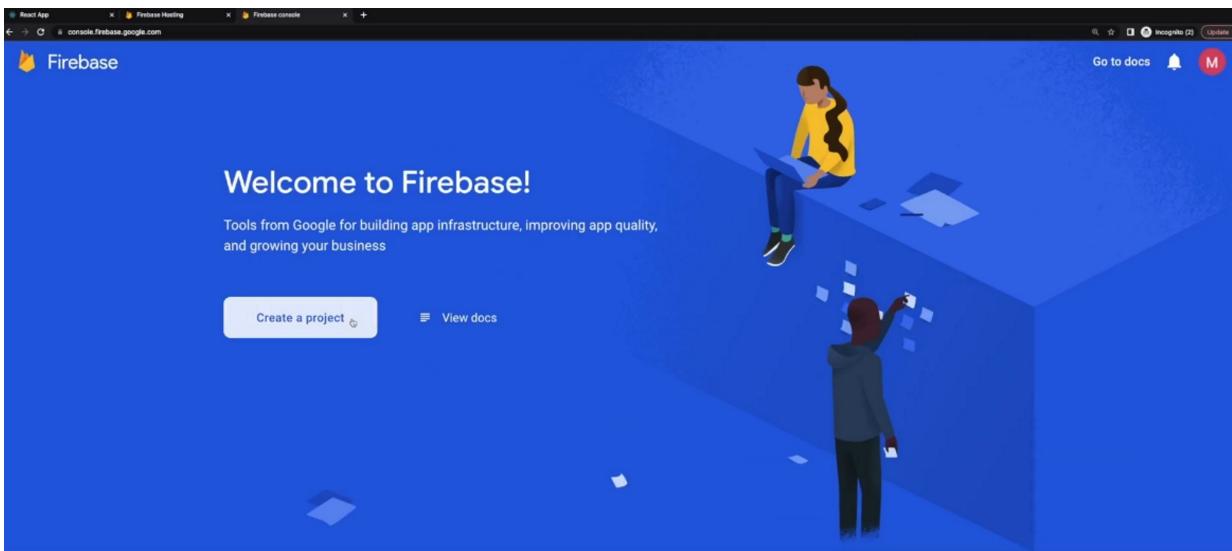
A screenshot of a Google search results page. The search query is "deploy static site". The results include:

- [A Step-by-Step Guide: Deploying A Static Site or Single-page ...](https://www.netlify.com/blog/2016/10/27/a-step-by-step-guide-to-deploying-a-static-site-or-single-page-app/)
27 Oct 2016 — Step 1: Add Your New Site · Step 2: Link to Your GitHub (or supported version-control tool of choice) · Step 3: Authorize Netlify · Step 4: Select ...
- [Free Static Site Hosting | Render](https://render.com/docs/static-sites)
Deploy your static sites on Render in just a few clicks. Includes a global CDN, automatic TLS certificates, auto-deploys from Git, and custom domains, ...
- [8 Best Static Website Hosting for Business and Personal Use](https://geekflare.com/geekflare-articles/8-best-static-website-hosting-for-business-and-personal-use)
8 Best Static Website Hosting for Business and Personal Use · Netlify · Google Cloud Storage · Surge · Render · GitHub Pages · Vercel · Cloudflare.

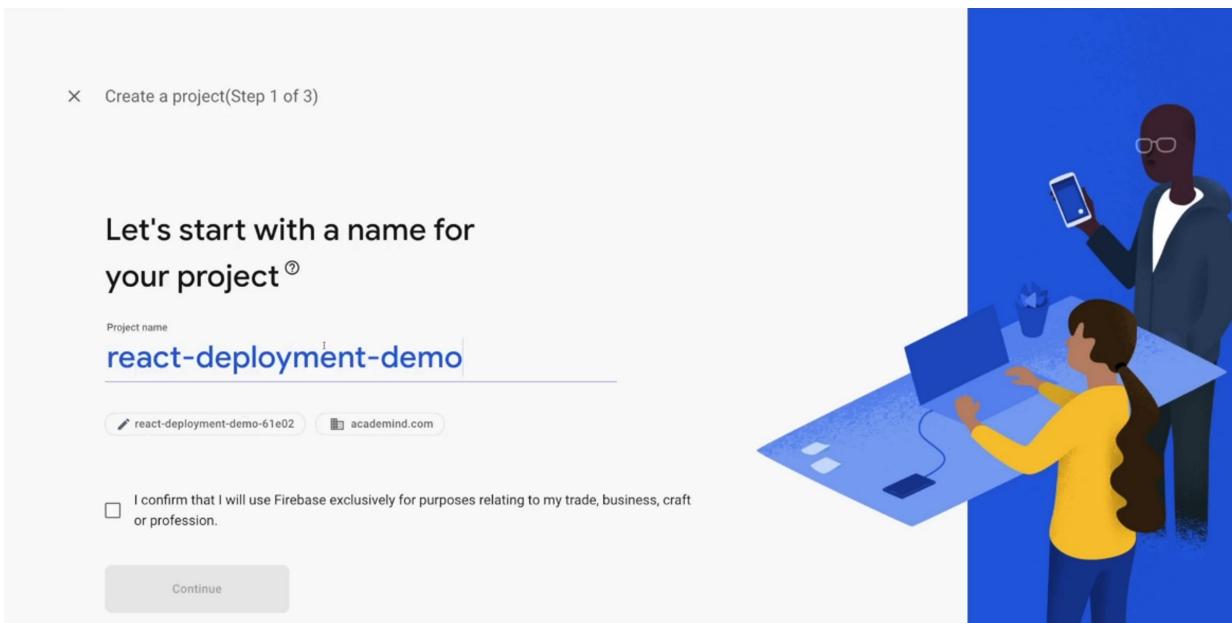
A screenshot of a Google search results page. The search query is "firebase hosting". The results include:

- [Firebase Hosting - Streamline common dev tasks](https://firebase.google.com/hosting)
Firebase provides tools to grow your app and business, for startups & global enterprises.
- [Firebase Docs](#)
Get started by Product or Platform Develop High-Quality apps, fast.
- [Build Products](#)
Accelerate & scale app development Get to market, deliver value faster
- [Firebase Crashlytics](#)
Track, prioritize, and fix crashes faster
- [Release/Monitor Products](#)
Improve app quality in less time Release apps with confidence

Go to firebase and login.



A screenshot of the Firebase Documentation page for 'Hosting'. The left sidebar shows navigation links for various services like Emulator Suite, Authentication, Realtime Database, Cloud Firestore, Storage, Machine Learning, and Hosting. The 'Hosting' link is highlighted. The main content area displays information about Firebase Hosting, including a video thumbnail titled 'Introducing Firebase Hosting' and a 'Get Started' button. A red arrow points from the top right towards the 'On this page' sidebar, which lists 'Key capabilities', 'How does it work?', 'Implementation path', and 'Next steps'.



A screenshot of the 'Create a project' wizard, Step 1 of 3. The heading is 'Let's start with a name for your project'. A text input field contains 'react-deployment-demo'. Below the input field is a checkbox for accepting terms and conditions. A 'Continue' button is at the bottom. To the right, there is an illustration of two people, one sitting at a desk and one standing, both looking at a laptop screen.

Disable 'Enable Google Analytics for this project'

The screenshot shows the second step of creating a Firebase project. The title is 'for your Firebase project'. It explains that Google Analytics is a free and unlimited analytics solution that enables targeting, reporting and more in Firebase Crashlytics, Cloud Messaging, in-app messaging, Remote Config, A/B Testing and Cloud Functions. Below this, a list of features enabled by Google Analytics includes A/B testing, crash-free users, user segmentation and targeting across products, event-based Cloud Functions triggers, and free unlimited reporting. A checkbox labeled 'Enable Google Analytics for this project' is checked and marked as 'Recommended'. At the bottom are 'Previous' and 'Create project' buttons.

The screenshot shows the 'Project Overview' page for the 'react-deployment-demo' project. The sidebar on the left lists services: Authentication, App Check, Firestore Database, Realtime Database, Extensions, Storage, Hosting (with a red arrow pointing to it), Functions, Machine Learning, Remote Config, and Cloud Messaging. The main area displays the project name 'react-deployment-demo' and a call-to-action 'Get started by adding Firebase to your app' with icons for iOS+, Android, JavaScript, and .NET. Below this is a section titled 'Store and sync app data in milliseconds'.

The screenshot shows the Firebase Hosting landing page for the project 'react-deployment-demo'. The main heading is 'Hosting' with the subtext: 'Deploy web and mobile web apps in seconds using a secure, global content-delivery network'. Below this is a 'Get started' button. A red arrow points from the top of the page down towards the 'Get started' button. On the right side of the page, there is a vertical navigation bar with numbered steps: 1, 2, and 3, each with a small circular icon. Step 1 has a red arrow pointing to it.

The screenshot shows the first step of the 'Set up Firebase Hosting' wizard. The title is 'Set up Firebase Hosting'. Step 1 is 'Install Firebase CLI'. It includes instructions: 'To host your site with Firebase Hosting, you need the Firebase CLI (a command line tool). Run the following [npm](#) command to install the CLI or update to the latest CLI version.' Below this is a code input field containing '\$ npm install -g firebase-tools'. A red arrow points from the bottom right of the input field towards the 'Next' button. Step 2 is 'Initialise your project' and Step 3 is 'Deploy to Firebase Hosting'. On the right side of the page, there is a blue smartphone and a tablet connected by a blue line, with a red arrow pointing from the phone towards the tablet.

The screenshot shows the VS Code interface with the following details:

- Left Sidebar:** Shows the project structure with files like build, static, asset-manifest.json, favicon.ico, index.html, logo192.png, logo512.png, manifest.json, robots.txt, node_modules, public, and src.
- Current File:** App.js is open in the editor.
- Editor Content:** The code for App.js includes a Suspense component for a blog post page.
- Terminal:** A terminal tab is open with the command `$ npm install -g firebase-tools`.

[Go to docs](#)

Set up Firebase Hosting

1 Install Firebase CLI

2 Initialise your project

Open a terminal window and navigate to or create a root directory for your web app

Sign in to Google

```
$ firebase login
```

Initiate your project
Run this command from your app's root directory:

```
$ firebase init
```

[Next](#)



The screenshot shows the VS Code interface with the following details:

- File Explorer (left sidebar):** Shows the project structure with files like build, static, asset-manifest.json, favicon.ico, index.html, logo192.png, logo512.png, manifest.json, robots.txt, node_modules, public, src, and components.
- Code Editor (center):** Displays the `App.js` file content, which includes code for a dynamic route and a suspense component.
- Terminal (bottom):** Shows the command `$ firebase init` entered in the terminal, with a red arrow pointing from the bottom of the terminal area towards the bottom center of the screen.

The screenshot shows the VS Code interface with the following details:

- File Explorer:** On the left, it lists project files and folders: build, static, asset-manifest.json, favicon.ico, index.html, logo192.png, logo512.png, manifest.json, robots.txt, node_modules, public, src, components, MainNavigation.js, MainNavigation.module.css, PostItem.js, PostItem.module.css, PostList.js, PostList.module.css, and pages.
- Terminal:** The terminal tab is active, showing the command `npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/` followed by a series of hash symbols (#).
- Output:** The output pane displays a message: "You're about to initialize a Firebase project in this directory: /Users/max/development/teaching/react-complete-guide".
- Terminal Content:** The terminal content continues with a question: "? Which Firebase features do you want to set up for this directory? Press Space to select features, then Enter to confirm your choices. (Press <space> to select, <a> to toggle all, <i> to invert selection, and <enter> to proceed)" followed by a list of options:
 - Firestore: Configure security rules and indexes files for Firestore
 - Functions: Configure a Cloud Functions directory and its files
 - Hosting: Configure files for Firebase Hosting and (optionally) set up GitHub Action deploys
 - Hosting: Set up GitHub Action deploys
 - Storage: Configure a security rules file for Cloud Storage

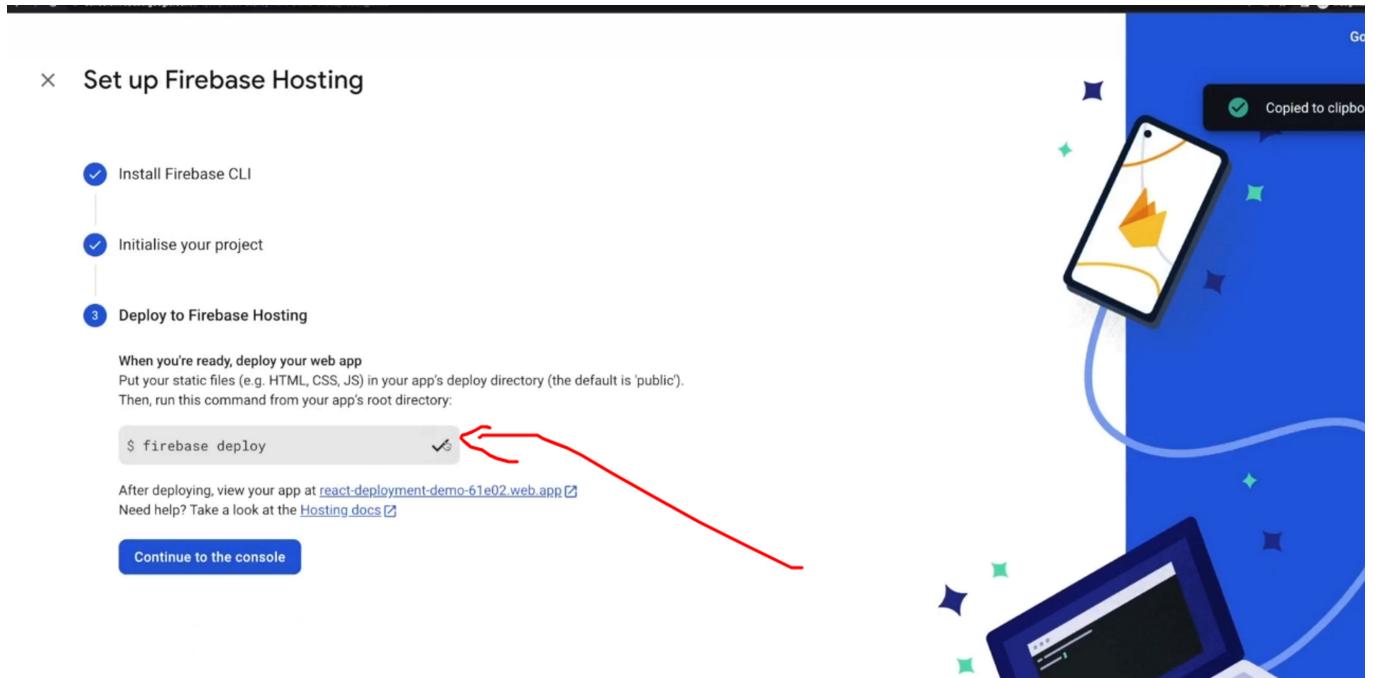
The screenshot shows a Visual Studio Code interface with a terminal window open. The terminal is displaying a series of prompts from the Firebase CLI to set up a project. The user has typed 'y' to proceed through the configuration steps.

```
29      </Suspense>
30      ),
/Users/max/development/teaching/react-complete-guide
? Which Firebase features do you want to set up for this directory? Press Space to select features, then Enter to confirm your choices. Hosting: Configure files for Firebase Hosting and (optionally) set up GitHub Action deploys
*** 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 just set up a default project.
? Please select an option: Use an existing project
? Select a default Firebase project for this directory: react-deployment-demo-61e02 (react-deployment-demo)
i Using project react-deployment-demo-61e02 (react-deployment-demo)
*** Hosting Setup
Your public directory is the folder (relative to your project directory) that
will contain Hosting assets to be 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)? (y/N) y
```

```
? Configure as a single-page app (rewrite all urls to /index.html)? Yes
? Set up automatic builds and deploys with GitHub? No
? File build/index.html already exists. Overwrite? No
i  Skipping write of build/index.html

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

✓  Firebase initialization complete!
```



The screenshot shows a code editor with several files listed on the left: manifest.json, robots.txt, node_modules, public, src, .firebaserc, .gitignore, firebase.json, package-lock.json, package.json, and README.md. The firebase.json file is currently selected. The code editor shows the following JSON configuration:

```
{
  "rewrites": [
    {
      "source": "**",
      "destination": "/index.html"
    }
  ]
}
```

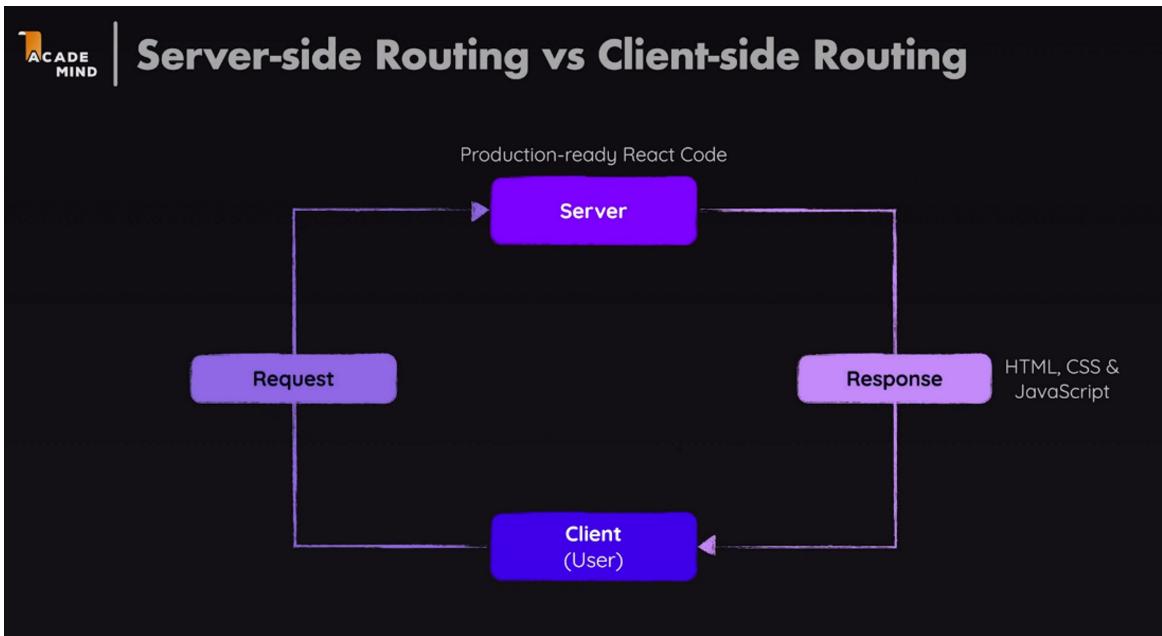
The code editor has tabs for PROBLEMS, DEBUG CONSOLE, OUTPUT, TERMINAL, and a terminal window open at the bottom showing the command '\$ firebase deploy'.

Server-side Routing & Required Configuration



Deployment Steps

-  **Test Code:** Manually & with automated tests
 -  **Optimize Code:** Optimize user experience & performance
 -  **Build App:** Run build process to parse, transform & optimize code
 -  **Upload App:** Upload production code to hosting server
 -  **Configure Server:** Ensure app is served securely & as intended



```

{
  "hosting": {
    "public": "build",
    "ignore": [
      "firebase.json",
      "**/.*",
      "**/node_modules/**"
    ],
    "rewrites": [
      {
        "source": "**",
        "destination": "/index.html"
      }
    ]
  }
}

```

✓ Final Steps: Deployment Configuration & A Gotcha You *Must* Know

So yup — we've **uploaded our code** and **deployed the website**.

But before we wrap things up, there's **one last thing** I wanna talk about:

👉 Configuring our project for deployment

...and a sneaky little **gotcha** that trips up a *lot* of devs 😬

🤔 Remember That SPA Question?

You might recall that, during deployment, we were asked:

"Do you want to configure this as a **Single Page Application (SPA)**?"
And we said **yes**.

But why does that matter?

🧠 Client-Side Routing FTW

Let's break it down:

Our app uses **React Router** — specifically react-router-dom — which means:

- All navigation between routes (like /posts, /about, etc.)
- Happens in the browser, *not* on the server

Yup, it's all client-side logic 🖥

So, when you click a link inside the app (like the blog page), React Router handles that internally. It doesn't go ask the server for a new page. It just loads the new component.

But What If Someone Types the URL Directly?

Let's say someone types example.com/posts directly into the address bar.

Here's what happens under the hood:

The browser sends a **request to the server** for /posts
The server tries to find a **file or folder** called /posts
But... it doesn't exist. 🤯
Boom — **404 error** unless we've handled it properly
This happens because servers, **by default**, try to serve files based on the path.
But in our case, **we want every path to load the same index.html**, so that React Router can take over and render the right component.

How Firebase Solves This

When we answered **yes** to the SPA question, Firebase:

- Automatically configured things so that **no matter what path is requested**
- It always returns index.html

This means:

- React Router always gets loaded
- Client-side routing just works™

 Without this setup, routing breaks when refreshing or entering URLs manually.

Other Hosting Providers? Be Careful!

Not all platforms ask you that SPA question.

So if you're hosting somewhere else (like Netlify, Vercel, or a custom server), you'll need to:

- **Manually set up a redirect or rewrite rule**
- Ensure that **all unmatched paths redirect to index.html**

Final Takeaway

Understanding the difference between:

- **Client-side routing** (React Router)
- vs. **Server-side routing** (handled by the backend)

...is *crucial* when deploying your app.

And make sure you configure your host to play nice with SPAs — or your routes will break 💣