

I'm human



How to create a react application

If you're building a React app from scratch, it's essential to consider the pros and cons of this approach. While it can be an easy way to get started with React, it also means that you'll need to tackle many framework-like problems on your own. This might lead to difficulties when implementing future features or getting support, as your implementation will likely be unique. For instance, if your app requires server-side rendering (SSR), static site generation (SSG), or React Server Components (RSC), you'll need to implement these features yourself. Similarly, future React features that require integrating at the framework level will also have to be implemented on your own. Our recommended frameworks, however, provide well-developed and supported solutions for these problems. Additionally, using a build tool like Vite, Parcel, or Rsbuild can improve app performance by reducing waterfalls from network requests. These tools offer fast refresh, JSX, Babel/SWC, and other common features out of the box. By choosing one of these tools, you'll also get access to a rich ecosystem of plugins and examples to help you get started. Vite is an opinionated build tool that aims to provide a faster and leaner development experience for modern web projects. Parcel combines a great out-of-the-box development experience with a scalable architecture. Rsbuild provides a seamless development experience for React applications, including built-in support for React features like fast refresh, JSX, TypeScript, and styling. If you're starting from scratch with React Native, you'll need to use Metro, the JavaScript bundler for React Native. However, we recommend starting with Vite, Parcel, or Rsbuild unless your project requires React Native features. To build common application patterns, start with client-only single-page apps (SPAs) using tools like React Router or Tanstack Router for routing. For data fetching, use purpose-built libraries like React Query, SWR, or RTK Query, which handle loading states, error states, and caching. These libraries should be used directly in components or integrated into router loaders for faster pre-fetching. For GraphQL APIs, consider using Code-splitting to break your app into smaller bundles that can be loaded on demand. This process involves splitting code by route to minimize client bundle sizes and improve performance. Libraries like Parcel support React.lazy for code splitting, but it's essential to consider the implications of load waterfall effects. When implementing various rendering techniques in an app, developers can reduce initial load times through bundling and data fetching. This includes optimizing Largest Contentful Paint (LCP), the time it takes for the largest visible content to render. Code-splitting instructions are available in build tool documentation: Vite's build optimizations, Parcel code splitting, and Rsbuild code splitting. Since most build tools only support single-page apps (SPAs), developers must implement other rendering patterns like server-side rendering (SSR), static site generation (SSG), or React Server Components (RSC) for complex routes. Even if these features aren't needed initially, they can be beneficial in the future. SPAs load a single HTML page and update dynamically as users interact with the app. Streaming and SSR can improve performance but are more complex to set up than SPAs. Vite's SSR guide provides further information on this technique. SSG generates static HTML files at build time and offers improved performance, although it requires more setup and maintenance than SSR. For RSC, see Parcel's examples for a deeper understanding. Integrating rendering strategies with routers allows apps to choose the right strategy per route without needing complete app rewriting. This can significantly decrease Time to First Byte (TTFB), First Contentful Paint (FCP), and Largest Contentful Paint (LCP). When building an app from scratch, developers will encounter various limitations that require deep expertise. Frameworks like Create React App make developing React apps easier by providing essential features out of the box. This package is ideal for personal projects as well as large-scale applications due to its ease of use and comprehensive feature set. Firstly, make sure you've got Node installed on your computer; you can grab it from nodejs.org. Next, ensure that your Node version is at least 10 and npm (a package manager) is version 5.2 or higher - both come bundled with Node. For a smoother workflow, I recommend using Visual Studio Code as your code editor; download it from code.visualstudio.com. To start building your React app with Create React App, open up your terminal or command line. You can then use the npx tool to create a new project without having to install Create React App separately - this ensures you're always working with the latest version. Simply run `npx create-react-app my-react-app` in your terminal; this will create a folder named "my-react-app" and automatically install all required packages, which should take around 2-3 minutes to complete. Create React App also comes with templates for different types of projects - for example, if you want to use TypeScript, run `npx create-react-app my-react-app --template typescript`. After your project is set up, your project structure should look something like this: my-react-app |— README.md |— node_modules |— package.json |— .gitignore |— public |— src Each of these files and folders has a specific role: - 'README.md' provides helpful tips and links for learning Create React App. - 'node_modules' contains all the dependency-related code installed by Create React App - you won't need to go into this folder directly. - 'package.json' manages your app's dependencies, scripts, and more. - '.gitignore' is used to exclude large folders like 'node_modules' from being tracked by Git. - 'public' is where you store static assets for your app, such as images and fonts. - 'src' contains all your React-related code. Once your project is set up, start it with `npm start`. When you create a new React app, a browser tab will automatically open on your computer's default browser to view your app. You can access your app by going to localhost:3000 in the address bar. The development server starts immediately, and you can see the starting home page for your app. The App.js file in the src folder contains the code for your app. In this file, you'll find an import statement for the logo from `"/logo.svg"` and another import statement for the CSS file `"/App.css"`. The function App() returns a JSX element that represents the main content of your app. To remove the `p` and `tags` and add an `h1` element with the name "React Posts Sharer", you can modify the App.js file as follows: `jsx import logo from "/logo.svg"; import "/App.css"; function App() { return ( React Posts Sharer ); } export default App;` When you save the file, your page will immediately update to reflect the changes. The development server automatically refreshes to show the updated content. Create React App also includes all the necessary packages for running tests using the React Testing Library (@testing-library/react). A basic test is included in the App.test.js file in src that tests whether the app component successfully displays a link with the text "learn react". You can run this test by executing the command `npm run test`. However, when you modify your code to remove the link element and add a title element, the test will initially fail. To make the test pass, you need to update the test file App.test.js as follows: `jsx import { render, screen } from "@testing-library/react"; import App from "/App"; test("renders app title element", () => { render(); const titleElement = screen.getByText(/React Posts Sharer/i); expect(titleElement).toBeInTheDocument(); });` Once you've updated the test file and saved your changes, you can run the test again by executing the command `npm run test`. This time, the test should pass. The index.js file in your project root directory contains the main code that renders your application: `jsx import React from 'react'; import ReactDOM from 'react-dom'; import '/index.css'; import App from '/App'; import reportWebVitals from '/reportWebVitals'; ReactDOM.render( , document.getElementById('root') );` This code imports the React and ReactDOM modules, along with your App component. It then uses ReactDOM to render your application in an HTML element with the id "root". React App You need to enable JavaScript to run this app. The entire React app is embedded within the HTML page through a `div` element with an id of "root". No modifications are required in the body tags, but it's beneficial to update metadata in the head section to provide specific information about the app. This includes meta tags for title, description, favicon image, theme-color, apple-touch-icon, and manifest. To enhance user experience, the metadata can be tailored as follows: React Posts Sharer You need to enable JavaScript to run this app. In the App component, an `img` element is used to import a logo file from the src folder. This feature is made possible by Create React App's webpack configuration, which allows importing static assets directly into components. Alternatively, static assets can be moved to the public folder and referenced using the filename.extension syntax in `js` or `css` files. `jsx import "/App.css"; function App() { return ( React Posts Sharer ); } export default App;` Moreover, with Create React App, it's not necessary to use an `img` element to display the `svg` file; it can be imported as a component. Importing SVG files as ReactComponents is made easy with the syntax `import { ReactComponent as Logo } from "/logo.svg";`. This allows us to use our imported SVG just like a regular component, making it effortless to incorporate into our application. `svg` files were previously challenging to use in React, but this component syntax simplifies the process and enables features such as inline styles. To display post data in our app, we can fetch it from the JSON Placeholder API using the `axios` dependency. Installing `axios` involves running `npm install axios`, which adds it to our `node_modules` folder. We can review all installed dependencies within our `package.json` file, where `axios` is listed under the "dependencies" section. However, we will not include `axios` in this project. For those interested in using TypeScript with their existing Create React App project, a simple installation of the `typescript` dependency and relevant type definitions for React development and testing is required: `npm install typescript @types/node @types/react @types/react-dom @types/jest`. After restarting the development server and renaming any React file from `.js` to `.tsx`, a working React and TypeScript project can be achieved. To fetch and display our post data, let's create a separate component called Posts. We'll place this component within a folder named `src/components/Posts.js`. The Posts component will request posts from JSON Placeholder, store them in a state variable called `posts`, and then map over them to display their title and body. To see the post data in our app, we need to import the Posts component into the App component. We'll go back to `App.js` and import the Posts component by navigating to the `components` folder and retrieving it from `src/components/Posts.js`. In the App component, we can use the imported Posts component to display the post data fetched from JSON Placeholder. This will allow us to visualize our post sharing app's functionality in action. After importing the Posts component and adding some CSS styles to improve the appearance of the app, we can see all fetched posts below the header in the home page. We should note that Create React App comes with built-in CSS support, allowing us to import `css` files globally. Within the App.css file, we added some styles to enhance the app's look: centered text, maximum width, and font size adjustments. Additionally, we modified the body element in the index.css file to have a dark background and white text. The resulting app has an improved appearance, and we can further enhance it by exploring more advanced CSS configurations like CSS modules or SASS. We can find additional resources for CSS styling in the README.md file. To publish our app, we need to run `npm run build` to create an optimized production build. This command will output the size of each generated file, helping us understand how our app's performance might be affected by large `.js` files. Each chunk includes a unique cache-busting hash to ensure changes are visible after deployment. Finally, we can run our built React project locally using the `npm package serve`, which helps detect any errors in the final version before pushing it live. To start our React app on a different development port, we'll use `npx serve`. This will deploy our app to a different port, such as localhost:5000, allowing us to publish it live to services like Netlify, Github Pages, or Heroku. React can be challenging to learn and master. However, with the right resources, you can set yourself up for success. I've created The React Bootcamp, an in-depth course covering everything from setting up your development environment to advanced topics. You can click below to try it out. React has revolutionized web development by providing a powerful way to build interactive user interfaces. In this blog post, we'll take you through the process of creating a React app from scratch. We'll cover everything from setting up our development environment to rendering components. Let's Begin! Before proceeding, ensure you have Node.js and NPM installed on your system. You can download Node.js from []. Open your terminal and verify their installation by running the following commands: `node -v` `npm -v` To start a new React project, open your terminal and run the command `npx create-react-app my-react-app`. This will create a new directory with the necessary files and folder structure for a React project. Once the project is created, we'll explore its structure. The project consists of the following folders: * `node_modules` : contains all project dependencies installed by npm. * `public` : holds static assets, including the index.html file. * `src` : where most of our work will be done, containing main application files like App.js. Next, we'll create a basic React component. Open `src/App.js` and replace its content with: `jsx import React from 'react'; const App = () => { return ( Hello, It's my first React App! ); }`

Happy Coding!!

); }; export default App;` ```` This component returns a simple 'div' element containing two headings with the provided text. Now, let's render this component in 'src/index.js'. Replace its content with: ````javascript import React from 'react'; import ReactDOM from 'react-dom'; import App from './App'; ReactDOM.render(, document.getElementById('root')); ```` Finally, start the development server by running `npx serve`. Congratulations on successfully building a React app from scratch! Starting the development server allows you to see your app in action by running `npm start` in the terminal, which will display your React app at . The page should show "Hello, It's my first React App!" "Happy Coding!!" as headings. You've now set up a new React project and learned how to create basic components and render them to the DOM. This is just the beginning; you can explore more advanced concepts like state management, routing, and API interactions.

How to create a micro frontend application using react. How to create api for react js. Ways to create a react app. How do i create a new react app. Create an app using react. How to create a react application from scratch. How to create a react native application. How to create an app in react. How to create a react application using vite. How to start a create react app. How to create a react application with vite. Create simple react application. How to create a web application using react js. How to create a react application in visual studio code. How to create a react application with webpack.