Click to verify

Share copy and redistribute the material in any medium or format for any purpose, even commercially. Adapt remix, transform, and build upon the material for any purpose, even commercially. The licensor cannot revoke these freedoms as long as you follow the license terms. Attribution You must give appropriate credit , provide a link to the license, and indicate if changes were made . You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use. ShareAlike If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. No additional restrictions You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits. You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation . No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material. Unlock the full potential of React and take your coding skills to the next level with this tutorial! We'll build a practical and functional todo list app that showcases the incredible capabilities of this popular JavaScript library. You'll learn how to create your own dynamic and interactive UI elements, making your app more user-friendly and efficient. Goals:Define the TodoList Component: We'll kick off by creating a functional component that defines the TodoList component.Manage State with useState Hook: Next, we'll leverage the power of the useState hook to manage the state in our TodoList component.Design the User Interface with JSX: To create an engaging and stylish user interface, we'll use JSX to define the UI for our TodoList component.Handle Events: We'll implement event handling to add and delete todo items, including form submissions and button clicks.Conditional Rendering: Lastly, we'll use conditional rendering to only display the delete button for a todo item if it exists, making our app more user-friendly and efficient. Ready to take your React skills to the next level and build a functional and stylish todo list app? Let's get started! Setup Let's kick off our React project by opening up your terminal or command prompt. Navigate to the directory where you want to create your new project. Once you're in the desired directory, enter the following command: npx create-react-app todo-app Once the project is created, navigate to the project directory using the following command: Next, install React and ReactDOM: npm install react react-dom They are both essential for creating a React app. React allows you to create reusable UI components, while ReactDOM is responsible for rendering those components into the browser. Now you can start the development server by running the following command: Create a Todo list componentfunction TodoList () { return (Add Todo);}export default TodoList; Result Adding the rest of react features Add import statement that brings in the React library and the useState hook from the react package at the top most part of your component.import React, { useState } from 'react'; Add useState hook to create two pieces of state: todos and inputValue. todos is an array that will hold the list of todos, and inputValue is a string that will hold the value of the input field. const [todos, setTodos] = useState([]);const [inputValue, setInputValue] = useState(''); Adding Todo Listimport React from 'react'import { useState } from 'react'function TodoList () { const [todos, setTodos] = useState([]) const [inputValue, setInputValue] = useState('')function handleChange(e){ setInputValue(e.target.value)}function handleSubmit(e){ e.preventDefault() setTodos([...todos, inputValue]) setInputValue('')} return (Todo List Add Todo {todos.map(todo) => ({todo} Delete)}} Inside the form, the value of the input field is set to the inputValue state and the onChange event is set to the handleChange function, which updates the inputValue state every time the input field changes. Add Todo Inside the ul, the map method is used to render each todo as an li element with a key set to the index of the todo. Each li element also contains a delete button. {todos.map(todo) => ({todo} Delete)}} This is a function that sets the inputValue state to the value of the input field. It's called every time the input field changes. const handleChange = (e) => { setInputValue(e.target.value);}; This is a function that adds a new todo to the todos state and clears the inputValue state when the form is submitted. It's called when the form is submitted. const handleSubmit = (e) => { e.preventDefault(); setTodos([...todos, inputValue]); setInputValue('');}; Deleteing Todo List This is a function that deletes a todo from the todos state. It creates a copy of the todos array using the spread operator, removes the todo at the specified index using the splice method, and sets the todos state to the new array.const handleDelete = (index) => { const newTodos = [...todos]; newTodos.splice(index, 1); setTodos(newTodos);};// {todos.map((todo, index) => ({todo} handleDelete(index)}>Delete)}} Result This is our final codes put togetherimport React from 'react'import { useState } from 'react'function TodoList () { const [todos, setTodos] = useState([]) const [inputValue, setInputValue] = useState('')function handleChange(e){ setInputValue(e.target.value)}function handleSubmit(e){ e.preventDefault() setTodos([...todos, inputValue]) setInputValue('')}function handleDelete(index){ const newTodos = [...todos] newTodos.splice(index, 1) setTodos(newTodos)} return (Todo List Add Todo {todos.map(todo, index) => ({todo} handleDelete(index)}>Delete)}})}export default TodoList; Embarking on your journey to create a ToDo List application can be exhilarating without the right ReactJS tutorial guiding you through. React, known for its efficiency in building dynamic user interfaces, offers tools like state management and event handling that are pivotal when dealing with tasks such as utilizing forms, making API calls with AJAX, or maintaining application state. This article promises a hands-on approach to understanding these React concepts, handling AJAX for dynamic content retrieval, manipulating state for task management, and more. Navigate through the creation of your app from setting up your React project to styling it, ensuring a rich learning experience teeming with practical code examples to bolster your development skills. Build your ToDo List Application in 5 Easy Steps:1) Setting Up Your React Project2) Building the App Component3) Creating the TodoList Component4) Implementing the TodoItem ComponentThe code for all of these steps that we've discussed above will look like this in your app.jsx file:5) Styling Your Todo AppAll your CSS styling will be in a separate file, here in the app.css file, and will look like this:The output after running our files together will look like this:Takeaways...FAQsCan I use React and Firebase together?Is Google Firebase free?What is the purpose of Firebase? Ive divided this entire project into 5 simple steps for you to follow along with leaving space for you to add your own touch and make it yours, lets get started. Download and Install Node.js: First, visit the official Node.js website to download the installer. Choose the LTS (Long Term Support) version for stability and extended support. Install npm: Node.js installation includes npm (Node Package Manager), with essential for managing JavaScript packages.Install Create React App: Open your terminal and runnpm install -g create-react-appto install this utility globally. Generate Your Project: Executenpx create-react-app my-appto create a new directory named my-app with all the necessary setups. Also Read: How to Install React.js on Windows: A Complete Guide Explore Initial Setup: Navigate to your project directory and observe the default structure provided by Create React App. Key folders include public for static assets, src for source files, and node_modules for dependencies. Key Files: The src folder contains App.js, the root component, and index.js, which renders the App component into the DOM.Launch the Server: In your project directory, runnpm startto fire up the development server. This command compiles the app and opens it in your default web browser. Automatic Refresh: Any changes you make in the source files will automatically refresh the app in the browser, thanks to the hot reloading feature. This setup process leverages the Create React App toolchain, which simplifies the configuration and initial file structure, allowing you to focus on building your React application. Before diving into the next section, ensure youre solid on full-stack development essentials like front-end frameworks, back-end technologies, and database management. If you are looking for a detailed Full Stack Development career program, you can join GUVIs Full Stack Development Course with Placement Assistance. You will be able to master the MERN stack (MongoDB, Express.js, React, Node.js) and build real-life projects. Additionally, if you want to explore ReactJS through a self-paced course, try GUVIs ReactJS certification course. Initialize the Component: Start by creating a new file in your projectssrc/componentsdirectory. Name itApp.js. This file will house your main App component, which serves as the entry point for your ToDo List application. Define the App Component: Within App.js, define your App component as a functional component. Use the following basic structure to get started.React and ReactDOM: At the top of yourApp.jsfile, ensure you import React. This import is crucial as it allows you to use JSX and other React features.Additional Components: If your application will use other custom components or third-party libraries, import them at the beginning of the file. For instance, if you are using Material-UI components, you might add.CSS Files: To style your component, import the necessary CSS files at the top of your component file.Create Root Element: In your projectssrc/index.js, set up the root for your React application by importingReactDOMand pointing it to the appropriate DOM element:Display the App Component: TheReactDOM.createRoot()method links your React application to an existing DOM element (typically a div with the id of root in yourpublic/index.htmlfile). Theroot.render()method then renders your App component within this root element, displaying your initial UI on the page. By following these steps, youve successfully set up and rendered your primary App component, forming the foundation of your ToDo List application. This setup allows for further expansion and integration of additional features and components as you develop your application. Must Read: Stopwatch Using ReactJS: 4 Easy Steps to Create Your First React Application To manage the list of tasks within your TodoList component, utilize theuseStatehook from React. Start by initializing the state with an empty array or a default set of tasks. This state will hold all the tasks you add, each represented by an object with properties such as id, text, and completed status. For adding new tasks, handle user input with a controlled component approach. Include an input field in your components return statement and bind it to thetextstate withonChangeto capture user input dynamically. To add tasks to your list, create anaddTaskfunction that constructs a new task object and updates the state. Bind this function to a buttonsonClickevent within your component to trigger task addition. Use themapfunction to iterate over the tasks array and display each task. Each task is rendered as a list item within an unordered list, with buttons to mark it as complete or to delete it. ImplementtoggleCompletedanddeleteTaskfunctions to handle these actions, updating the tasks array accordingly. By following these steps, youll have a functional TodoList component that allows users to add, display, complete, and delete tasks dynamically. Also Read: Currency Converter App Using ReactJS: A Step-by-Step Guide Initialize the Component: Begin by creating a file namedTodoItem.jsin yoursrc/componentsdirectory. This component represents each task in your ToDo list.Props Overview:TodoItemreceivestask,deleteTask, andtoggleCompletedas props. These functions are crucial for manipulating the tasks state from the parent component. Functionality:task: Contains details of a specific task. deleteTask: Function to remove a task from the list. toggleCompleted: Function to toggle the completion status of a task. import React, { useState } from 'react';import './App.css';function App() { const [tasks, setTasks] = useState([]); const [input, setInput] = useState(''); const addTask = () => { if (input.trim()) { setTasks([...tasks, { text: input, completed: false }]); setInput(''); } }; const toggleTask = (index) => { const newTasks = tasks.map((task, i) => i === index ? { ...task, completed: !task.completed } : task); setTasks(newTasks); }; const deleteTask = (index) => { const newTasks = tasks.filter((_, i) => i !== index); setTasks(newTasks); }; return (To-Do List setInput(e.target.value)} placeholder="Add a new task" /> Add {tasks.map((task, index) => (toggleTask(index)}>{task.text} deleteTask(index)}>Delete)}});}export default App; Find Out 6 Essential Prerequisites For Learning ReactJS Inline Styling: Begin by applying styles directly to your components using thestyleattribute. Remember, in React, inline styles are not strings but objects with camelCased properties as keys.External CSS Styling: For more general styling, create a CSS file and import it into your component(as mentioned before). This method is excellent for reusable styles across multiple components.CSS Modules: Utilize CSS Modules for component-specific styles without worrying about global namespace pollution.SASS/SCSS: Leverage the power of SASS for complex styling scenarios, like nested selectors and variables, by creating a.scssfile and importing it similarly to a regular CSS file. Know More About Types of CSS: A Comprehensive Guide to Styling Web Pages Styled Components: Install thestyled-componentslibrary and use it to encapsulate styles within your components. This method couples the styles directly with the component logic, enhancing modularity and reusability.Dynamic Styling: Use props to dynamically alter styles based on component state or props. Styled Components make this particularly intuitive.Global Styles: Define global styles withcreateGlobalStylefrom styled-components to maintain consistency across your application.Theming: Implement theming within your app to allow for style variations across different components using the same theme provider.Media Queries: Use media queries within styled-components to create responsive designs that adapt to various device sizes.Animations and Transitions: Add visual flair to your components with CSS animations and transitions to enhance user experience. By integrating these styling techniques, your ToDo app will function well and look modern and appealing. Must Explore: 10 Best React Project Ideas for Developers [with Source Code] .app { font-family: Arial, sans-serif; text-align: center; background-color: #f4f4f9; padding: 20px; max-width: 500px; margin: 50px auto; border-radius: 10px; box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);}.header { margin-bottom: 20px;}.input-container { display: flex; justify-content: center; margin-bottom: 20px;}.input-container input { width: 70%; padding: 10px; font-size: 16px; border: 1px solid #ddd; border-radius: 5px; outline: none;}.input-container button { padding: 10px 15px; font-size: 16px; margin-left: 10px; border: none; border-radius: 5px; background-color: #28a745; color: white; cursor: pointer;}.input-container button:hover { background-color: #218838;}.task-list { list-style: none; padding: 0;}.task { background-color: white; padding: 15px; margin-bottom: 10px; border-radius: 5px; display: flex; justify-content: space-between; align-items: center; box-shadow: 0 0 5px rgba(0, 0, 0, 0.1); cursor: pointer;}.task.completed { text-decoration: line-through; color: #888;}.task button { background-color: #dc3545; border: none; color: white; padding: 5px 10px; border-radius: 5px; cursor: pointer;}.task button:hover { background-color: #c82333;} As you can see above, you can add whatever tasks you would like to in the Add a new task box and then click on Add. You can add as many tasks as you want and also delete them using the Delete button as shown in the image below. Cool right? How we can just make stuff like this! Kickstart your Full Stack Development journey by enrolling in GUVIs Full Stack Development Course with Placement Assistance where you will master the MERN stack (MongoDB, Express.js, React, Node.js) and build interesting real-life projects. This program is crafted by our team of experts to help you upskill and assist you in placements. Alternatively, if you want to explore ReactJS through a self-paced course, try GUVIs ReactJS certification course. Through the guidance offered in this article, readers now possess the foundational knowledge required to embark on creating a dynamic ToDo List application using ReactJS. We covered essential phases from setting up the React project, crafting the App and TodoList components, to meticulously styling the application for an appealing user interface. The significance of this tutorial extends beyond just the creation of a ToDo list; it lays down the groundwork for understanding Reacts core functionalities such as component creation, state management, and the use of hooks. This not only aids in building more complex applications but also in appreciating Reacts efficient rendering and state management capabilities. As readers continue to refine their skills, exploring further into areas like context API or Redux for state management on a larger scale could be the next step. Also Explore: 6 Compelling Reasons to choose ReactJS over AngularJS Yes, React and Firebase can be used together. React provides the front-end framework, while Firebase offers backend services like authentication, real-time database, and hosting, making it easier to build full-featured web applications. Google Firebase offers a free tier called the Spark Plan, which includes limited usage of core features such as authentication, Firestore database, and hosting. For more extensive usage, you can upgrade to a paid plan. Firebase is a platform developed by Google for building mobile and web applications. It provides a variety of tools and services, including real-time databases, authentication, analytics, cloud storage, and serverless functions, to help developers build, manage, and grow their apps efficiently. If you're new to React.js and you're eager to dive into application development, then you've come to the right place!Join me in this tutorial as I walk you through building a basic TODO app from the ground up. A TODO app serves as an ideal project for beginners to grasp the fundamentals of a new programming language or framework quickly. It provides a practical context for learning essential concepts while working towards a tangible outcome.If you're embarking on your React.js journey, building a TODO app along with this tutorial could be the perfect starting point. Before we begin, ensure you have basic knowledge of React.js and have Node.js and npm installed on your computer. If you haven't already, take a moment to set up your development environment. Our goal is to create a simple TODO app with features. Here's what we'll be aiming for:Adding New TODOs: Enable users to add new tasks to the list.Editing and Deleting TODOs: Provide functionality to modify or remove existing tasks.Marking TODOs as Completed: Allow users to indicate when tasks are finished.Tracking Completed TODOs: Implement a feature to keep track of all completed tasks.Feel free to expand upon this list with additional features if you like. For the purpose of this tutorial, we'll focus on these core functionalities.This is an example of the TODO app we are going to be building:A preview of our todo appTable of Contents: In 2024, using a framework like Next.js or Remix is a recommended approach for initiating a React project. Either framework will suffice so just choose the one you're most comfortable with. For this tutorial, we'll be using Next.js.To create a React app with Next.js, navigate to your preferred directory and run the following command:npx create-next-app@latestNote: We won't be using TypeScript and TailwindCSS for this project, so you can proceed with the default settings.Once the installation is complete, navigate into your newly created app directory (I've named mine 'todo') and start the development server by running:cd todonpm run dev cd todoyarn run devWith your development server up and running, we're ready to begin crafting our TODO app! In React, we build UIs out of components. Our TODO app's UI consists of several parts. Let's break them down: The Header component serves to display the title of our app. Rather than directly embedding HTML, we'll build this functionality within a React component.Start by creating a directory for our components:mkdir src/components cd src/componentstouch Header.jsxComponents in React are essentially JavaScript functions that return HTML. In our Header.jsx file, define a function that returns the HTML content for our Header component:function Header() { return (TODO);} export default Header;We're exporting the Header function so that we can utilize it throughout our project. The TODO Hero Component plays a pivotal role in our application. It serves as the section where we provide an overview of the total number of todos and the number of completed tasks.An image showing the TODOHero componentUnlike the header component, which remains static throughout our app usage, the TODOHero component is dynamic. It continuously updates based on the number of completed todos and the total number of todos.When building components, it's important to identify dynamic parts early on. In React, we achieve this by passing arguments, called props, to our components.Let's create the TODOHero component. First, make sure you're in the src/components directory:cd src/componentsNow, create a new file for the TODOHero component:touch TODOHero.jsxIn TODOHero.jsx, define a function that takes props as arguments:function TODOHero({ todos_completed, total_todos }) { return (Task Done Keep it up {todos_completed}/{total_todos});}export default TODOHero;This function returns HTML content for our TODOHero component. We're using props to dynamically update the number of completed todos and the total number of todos. Our Form component is going to be a simple input with a submit button, so go ahead and create a new componenttouch src/components/Form.jsxLike I said, this is going to be a very simple form: just an input with a submit button. The label is for accessibility. function Form() { const handleSubmit = (event) => { event.preventDefault(); event.target.reset(); }; return (Submit);}export default Form;We've added an onSubmit event to the form with a handleSubmit event handler. The event.preventDefault() prevents the form from submitting and reloading the entire app. Lastly, we reset the form with event.target.reset(). Lastly, let's create the List component. Start by creating a new component file named TODOList.jsx:touch src/components/TODOList.jsxThe list itself is a straightforward ordered list: function TODOList() { return {/* list goes here */};}export default TODOList;The list items will be generated dynamically from the todo data. But before we proceed, let's create a separate component for the list item.In React, nearly everything is a component, so we'll create the Item component alongside the TODOList component: function Item({ item }) { return ({item?.title} Edit Delete);The list item itself is simply an element with buttons for editing and deleting tasks. We've ensured that the itself is not clickable, following the principle that "anything clickable on the web should either be a button or a link".Now, we can use the Item component within our list: function TODOList({ todos }) { return ({todos && todos.length > 0 ? (todos?.map((item, index) =>)) : (Seems lonely in here, what are you up to?) });}export default TODOList;With these components in place, our TODO app's UI is fully constructed. So far, we've created four separate components, each of which doesn't do much on its own. Now, we need to render these components in our index page.In Next.js, pages are located inside the src/app directory, and the index page is typically named page.js.First, let's empty the contents of the file as we won't need anything inside it:echo -n > src/app/page.jsNext, import all the components we've created and utilize them inside the page.js file as shown below: import React from "react";import Form from "@/components/Form";import Header from "@/components/Header";import TODOHero from "@/components/TODOHero";import TODOList from "@/components/TODOList";function Home() { return ( );}export default Home;By viewing the output in your browser, it should resemble something like this:A preview of our app without CSS For styling, we'll stick to good old CSS. Let's create a styles.css file to hold our styles:touch src/app/styles.cssAlso, delete all the CSS files that came with installing Next.js as we won't need them:rm src/app/page.module.css && src/app/globals.cssNow, you can add your CSS rules in the styles.css file. Though not perfect, the following CSS should suffice for our simple example:*,*::after,*::before { padding: 0; margin: 0; font-family: inherit; box-sizing: border-box;}html,body { font-family: sans-serif; background-color: #0d0d0d; color: #fff; display: flex; align-items: center; justify-content: center; width: 100vw;}button { cursor: pointer;}.visually-hidden { position: absolute ; clip: rect(1px, 1px, 1px, 1px); padding: 0 ; border: 0 ; height: 1px ; width: 1px ; overflow: hidden; white-space: nowrap;}.text_large { font-size: 32px;}.text_small { font-size: 24px;}.wrapper { display: flex; flex-direction: column; width: 70%;}@media (max-width: 510px) { .wrapper { width: 100%; } header { justify-content: center; }}header { display: flex; align-items: center; justify-content: flex-start; gap: 12px; padding: 42px;}.todohero_section { border: 1px solid #c2b39a; display: flex; align-items: center; justify-content: space-around; align-self: center; width: 90%; max-width: 455px; padding: 12px; border-radius: 11px;}.todohero_section div:last-child { background-color: #88ab33; width: 150px; height: 150px; border-radius: 75px; font-size: 48px; display: flex; align-items: center; justify-content: center; text-align: center;}.form { align-self: center; width: 97%; max-width: 455px; display: flex; align-items: center; gap: 12px; margin-top: 38px;}.form label { width: 90%;}.form input { background-color: #1f2937; color: #fff; width: 100%; height: 50px; outline: none; border: none; border-radius: 11px; padding: 12px;}.form button { width: 10%; height: 50px; border-radius: 11px; background-color: #88ab33; border: none;}.todo_list { align-self: center; width: 97%; max-width: 455px; display: flex; flex-direction: column; align-items: center; margin-top: 27px; margin-bottom: 27px; gap: 27px;}.todo_item,.edit-form input { display: flex; justify-content: space-between; align-items: center; height: 70px; width: 100%; max-width: 455px; border: 1px solid #c2b39a; font-size: 16px; background-color: #0d0d0d; color: #fff; padding: 12px;}.edit-form input { outline: transparent; width: calc(100% - 14px); height: calc(100% - 12px); border: transparent;}.todo_items_left,.todo_items_right { display: flex; align-items: center;}.todo_items_left { background-color: transparent; border: none; color: #fff; gap: 12px; font-size: 16px;}.todo_items_right { gap: 4px;}.todo_items_right button { background-color: transparent; color: #fff; border: none;}.todo_items_right button svg { fill: #c2b39a;}Lastly, we need to import the CSS file in our layout. Open the layout.js file located right next to the page.js and import the CSS file as demonstrated below:An image showing how to import styles.css file in our componentUpon previewing the app again, it should now reflect the applied styles:An image showing the preview of our app after adding CSS At this stage, we've crafted a visually appealing todo app, but it lacks functionality. Let's change that in this section. Firstly, we need a method to store our todo data. In React, this is accomplished using statea JavaScript object that holds information about a component's state.React provides a hook called useState(), which enables us to manage state in our React apps. But in Next.js, before utilizing useState, you need to specify that the component is a client component.Add the following code to the top of your src/app/page.js file:"use client";As depicted in the image below:An image showing how to add "use client" to the top of our page.jsNow, we can use the useState hook to create a state for our todo data: "use client";import React from "react";import Form from "@/components/Form"; function Home() { const [todos, setTodos] = React.useState([]); return ( );}export default Home;In the above code snippet, you'll notice that useState initially holds an empty array. It's important to understand that useState returns two values: todos and setTodos (you can name these anything you prefer).The first value, todos, holds the current value of the state, while setTodos (the second value) is a function used to update the state. Clear so far? Now that we have a means to store our data, let's define the type of data we intend to store. Essentially, it will be an array of objects, where each object holds the necessary information to render our list of todos:const [todos, setTodos] = React.useState([{ },{ },{ },]);Each object in the array will have the following structure:{title: "Some task", id: self.crypto.randomUUID(), is_completed: false }Here, self.crypto.randomUUID() is a method that allows the browser to generate unique IDs for each todo item. If you view the console, you'll observe that the generated IDs are indeed unique.console.log of our todo dataThis structure ensures that each todo item has a title, a unique identifier (id), and a Boolean value indicating whether the task is completed (is_completed). In React, there's a concept called state sharing, which allows children components to access the state of their parent components. This means that the todo state we created earlier can be shared among all our components.The first place we need the data from the state is in our List Component. Let's pass the state to the List component: "use client";import React from "react"; import TODOList from "@/components/TODOList"; function Home() { const [todos, setTodos] = React.useState([{ title: "Some task", id: self.crypto.randomUUID(), is_completed: false }, { title: "Some other task", id: self.crypto.randomUUID(), is_completed: true, }, { title: "last task", id: self.crypto.randomUUID(), is_completed: false },]); return (...);}export default Home;We already made provisions in our List component to receive a todos prop: function TODOList({ todos }) { return ({todos && todos.length > 0 ? (todos?.map((item, index) => ( )) : (Seems lonely in here, what are you up to?) });}Now, the todos prop will be populated by the data from our state, and without any further ado, this will work. Heres an image showing the List created from our todos data:An image showing a list of our todosThe other place we need the data is in our TODOHero component. We dont need all of the data in that component we just need to count the total number of todos and the number of completed todos: "use client";import React from "react"; import TODOHero from "@/components/TODOHero";import TODOList from "@/components/TODOList";function Home() { const [todos, setTodos] = React.useState([{ title: "Some task", id: self.crypto.randomUUID(), is_completed: false },]); const todos_completed = todos.filter((todo) => todo.is_completed === true).length; const total_todos = todos.length; return ( );}export default Home;Here, the JavaScript filter method is used to filter out all the todos with is_completed set to true, and then we get the length. The total_todos is simply the length of the entire array.Heres an image showing the TODOHero component with updated values:An image showing the updated TODOHero component Currently, our todo app displays todos from our dummy data:const [todos, setTodos] = React.useState([{ title: "Some task", id: self.crypto.randomUUID(), is_completed: false }, { title: "Some other task", id: self.crypto.randomUUID(), is_completed: true, }, { title: "last task", id: self.crypto.randomUUID(), is_completed: false },]);But the purpose of creating a Form component was to enable us to create new todos ourselves, not rely on dummy data.The good news is that just as we have access to the todo state data, we can also update the state of a parent from a children component. This means we can pass the function used to update the state, setTodos, to our Form component: "use client";import React from "react";import Form from "@/components/Form"; function Home() { const [todos, setTodos] = React.useState([{ title: "Some task", id: self.crypto.randomUUID(), is_completed: false },]); ... return (...);}export default Home;With access to the setTodos function in our Form component, we can now add new todos to our state when we submit the form: function Form({ setTodos }) { const handleSubmit = (event) => { event.preventDefault(); const value = event.target.todo.value; setTodos((prevTodos) => [...prevTodos, { title: value, id: self.crypto.randomUUID(), is_completed: false },]); event.target.reset(); }; return ( );}export default Form;The code snippet below is where the magic happens:setTodos((prevTodos) => [...prevTodos, { title: value, id: self.crypto.randomUUID(), is_completed: false },]);It is the equivalent of doing the following in plain JavaScript:let prevTodos = []; prevTodos.push({ title: value, id: self.crypto.randomUUID(), is_completed: false,});Now that we can add new todos to our state by ourselves, we can get rid of the dummy data. We no longer need it. Let's go back to using an empty array:const [todos, setTodos] = React.useState([]);Now that we're through with the first part, we can add todos as we please. Heres a video demonstrating it working: In our List component, we constructed an element with buttons. Now, we're going to attach an onClick event handler to the first button. function Item({ item }) { const completeTodo = () => { }; return ({item?.title} ... ...);}When we click on this button and the completeTodo handler is invoked, our objective is to:Filter the data to find the todo that was clicked.Modify the data and set the is_completed value to true.Before we can proceed with data modification, we need access to the setTodo function in our component. Fortunately, React allows state to be passed down to grandchildren components.This means we can pass the setTodo function from the component to our component: "use client";import React from "react"; import TODOList from "@/components/TODOList"; function Home() { const [todos, setTodos] = React.useState([]); ... return (...);}export default Home;Then, within our component, we pass the setTodo function to our component: function TODOList({ todos, setTodos }) { return ({todos && todos.length > 0 ? (todos?.map((item, index) => ( )) : (Seems lonely in here, what are you up to?) });}Now, within our component, we can use the setTodos function to update the todo's is_completed status when the button is clicked: function Item({ item, setTodos }) { const completeTodo = () => { setTodos((prevTodos) => prevTodos.map((todo) => todo.id === item.id ? { ...todo, is_completed: !todo.is_completed } : todo)); }; return (... ... ...);}Now, clicking on the first button within the todo item will toggle its completion status, effectively modifying the todo data.When a todo is marked as completed, we want to enhance its visual representation. This includes adding a fill to the SVG circle beside the todo title, creating the illusion that the todo is completed. Also, we want to add a strike-through to the text to signify completion. {item?.title} ;In the above code snippet, the button's color changes based on the completion status of the todo item. If the item is completed (is_completed is true), the SVG circle fills with a green color otherwise, it fills with a dark color. Also, the todo title text receives a line-through style if the todo is completed, indicating its completion visually.And now everything is working perfectly: When editing todos, we want to have a form in which we can edit the title of the todo. When the edit button is clicked we want to swap out the everything in the and have a form instead: function Item({ item, setTodos }) { const [editing, setEditing] = React.useState(false); const inputRef = React.useRef(null); const completeTodo = () => { }; const handleEdit = () => { setEditing(true); }; React.useEffect(() => { if (editing && inputRef.current) { inputRef.current.focus(); inputRef.current.setSelectionRange(inputRef.current.value.length, inputRef.current.value.length); } }, [editing]); const handleInpuSubmit = (event) => { event.preventDefault(); setEditing(false); }; const handleInputBlur = () => { setEditing(false); }; return ({editing ? ( ) : (... ... ...)});}I know the code above is quite a handful. Well, thats because we are doing a lot here but the first thing we did was create a state:const [editing, setEditing] = React.useState(false);When the edit button is clicked we set the value of our editing state to true, which will render our form:const handleEdit = () => { setEditing(true);};Now, when we submit the edit todo form by pressing enter, we also want to set the variable back to false so we can get back our list:const handleInpuSubmit = (event) => { event.preventDefault(); setEditing(false);};When we mouse out of the edit form, we also want to set the state back to false:const handleInputBlur = () => { setEditing(false);};Another thing we want to do is to focus the input once editing is set to true:React.useEffect(() => { if (editing && inputRef.current) { inputRef.current.focus(); inputRef.current.setSelectionRange(inputRef.current.value.length, inputRef.current.value.length); } }, [editing]);The edit todo itself has a single input field with an onChange event. As we edit the title in the input field, we want to modify the current todo with the updated title:const handleInputChange = (e) => { setTodos((prevTodos) => prevTodos.map((todo) => todo.id === item.id ? { ...todo, title: e.target.value } : todo));};The JavaScript array.map() method is perfect for this because it returns a new array with the same number of elements after modifying the title.Heres a video of it working seamlessly: Deleting todos is a straightforward process. When the delete button is clicked, we filter out the todo that triggered the delete event from the todo list. const handleDelete = () => { setTodos((prevTodos) => prevTodos.filter((todo) => todo.id !== item.id));};Dont forget to add an onClick event to the delete button: function Item({ item, setTodos }) { ... const handleDelete = () => { setTodos((prevTodos) => prevTodos.filter((todo) => todo.id !== item.id)); }; return ({editing ? (...) : (Delete)});And voil! It just works like a charm: Up to this point, our todo data has been stored solely in the application's state:const [todos, setTodos] = React.useState([]);While this approach works, it presents a challenge: when the app is reloaded, all todo data is lost.When it comes to persisting data, we typically think of databases. Storing our todo data in a database offers several advantages, such as easy access from any device. But there's an alternative: localStorage.LocalStorage is a browser-based storage system. It has some limitations, like a 5MB storage cap and data accessibility restricted to the browser where it's stored. Despite these drawbacks, we'll use localStorage in this tutorial for simplicity's sake. Currently, when we add a new todo, we're only updating the todo state in our Form component: const handleSubmit = (event) => { event.preventDefault(); const value = event.target.todo.value; setTodos((prevTodos) => [...prevTodos, { title: value, id: self.crypto.randomUUID(), is_completed: false },]); event.target.reset();};We still want to keep this, but at the same time we want to add the same data to localStorage, so well modify the code above to look like this: const handleSubmit = (event) => { event.preventDefault(); const value = event.target.todo.value; const newTodo = { title: value, id: self.crypto.randomUUID(), is_completed: false, }; setTodos((prevTodos) => [...prevTodos, newTodo]); const updatedTodoList = JSON.stringify([...todos, newTodo]); localStorage.setItem("todos", updatedTodoList); event.target.reset();};Did I mention that you can only store strings in localStorage? We cant store an array or object in localStorage. That's why we first convert our array of todo data to a string:const updatedTodoList = JSON.stringify([...prevTodos, newTodo]);And then finally we persist the data in localStorage with this code:localStorage.setItem('todos', updatedTodoList);Youll notice we used our todos state data in our component:const updatedTodoList = JSON.stringify([...todos, newTodo]);So dont forget to pass the todo state to the component:Also, since we can edit and delete todos in our app, we need to update the data in localStorage accordingly. First, pass the todos data to our component: function TODOList({ todos, setTodos }) { return ({todos && todos.length > 0 ? (todos?.map((item, index) => (// pass the todos to))) : (Seems lonely in here, what are you up to?) });}Now that we have access to the todo data in our component, we can persist data to localStorage after marking todo as completed: const completeTodo = () => { setTodos((prevTodos) => prevTodos.map((todo) => todo.id === item.id ? { ...todo, is_completed: !todo.is_completed } : todo)); const updatedTodos = JSON.stringify(todos); localStorage.setItem("todos", updatedTodos); setEditing(false);}; const handleInpuSubmit = (event) => { event.preventDefault(); const updatedTodos = JSON.stringify(todos); localStorage.setItem("todos", updatedTodos); setEditing(false);}; const handleInputBlur = () => { setEditing(false);};Lastly we want to also persist the data to localStorage after we delete a todo: const handleDelete = () => { setTodos((prevTodos) => prevTodos.filter((todo) => todo.id !== item.id)); const updatedTodos = JSON.stringify(todos.filter((todo) => todo.id !== item.id)); localStorage.setItem("todos", updatedTodos);};And thats all you need pretty easy right? Now when we create new todos, theyll be persisted in localStorage even after reloading our app. Even though we've successfully persisted our data to localStorage, our app data is still wiped when we reload our app or the browser. That's because we are not yet utilizing the data stored in localStorage.To address this, when our app is mounted (loaded), we want to retrieve the data from localStorage and then pass it to our state.In our src/app/page.js, we'll read the data from localStorage and store it in our todos state. "use client";import React from "react";import Form from "@/components/Form";import Header from "@/components/Header";import TODOHero from "@/components/TODOHero";import TODOList from "@/components/TODOList"; function Home() { const [todos, setTodos] = React.useState([]); React.useEffect(() => { const storedTodos = localStorage.getItem("todos"); if (storedTodos) { setTodos(JSON.parse(storedTodos)); } }, []); const todos_completed = todos.filter((todo) => todo.is_completed == true).length; const total_todos = todos.length; return ( );} export default Home;The code inside the useEffect() hook we run once the component is mounted.This is the part that reads the data from localStorage:const storedTodos = localStorage.getItem("todos");Since the data stored in localStorage is a string, we have to convert it back to our array of objects before we can use it:JSON.parse(storedTodos)And thats all you need to get it working. Now even when we reload the app, the data is persisted as you can see in this video: Congratulations! After a journey filled with coding and persistence, we've successfully built a simple yet functional todo app from scratch. The journey might have been long, but the result is worth it.You can explore the entire source code of the app here. Feel free to dive into the code and see how it all comes together.But wait, there's more! If you're eager to try out the app yourself, I have a hosted version available here. Go ahead and give it a try to experience the app firsthand.Thank you for joining me on this coding adventure. I hope you've gained valuable insights into building React apps and persisting data with localStorage.If you have any questions, feel free to message on Twitter at @sprucekhalifa, and don't forget to follow me for more insights and updates. Happy coding! In this article, we will create a to-do app to understand the basics of ReactJS. We will be working with class based components in this application and use the React-Bootstrap module to style the components. This to-do list can add new tasks we can also delete the tasks by clicking on them. The logic is handled by a click event handler whenever the user clicks on a task it gets deleted from the list.Lets have a quick look at how the final application will look like:ToDo App using ReactJSSteps to create the Application:NPX: It is a package runner tool that comes with npm 5.2+, npx is easy to use CLI tools. The npx is used for executing Node packages.npx create-react-app todo-reactcd todo-reactInstall the bootstrap and react-bootstrap modulenpm install bootstrapnpm install react-bootstrapAfter following the above steps the Folder structure will look like:Folder Structure: The dependencies in package.json file will look like:"dependencies": { "@testing-library/jest-dom": "^5.16.5", "@testing-library/react": "^13.4.0", "@testing-library/user-event": "^13.5.0", "react": "^18.2.0", "bootstrap": "^5.3.0", "react-bootstrap": "^2.7.4", "react-dom": "^18.2.0", "react-scripts": "5.0.1", "web-vitals": "^2.1.4"}Example: Write the below code in App.js file in the src directory javascript // App.js Fileimport React, { Component } from "react";import "bootstrap/dist/css/bootstrap.css";import Container from "react-bootstrap/Container";import Row from "react-bootstrap/Row";import Col from "react-bootstrap/Col";import Button from "react-bootstrap/Button";import InputGroup from "react-bootstrap/InputGroup";import FormControl from "react-bootstrap/FormControl";import ListGroup from "react-bootstrap/ListGroup"; class App extends Component { constructor(props) { super(props); // Setting up state this.state = { userInput: "", list: [], }; } // Set a user input value updateInput(value) { this.setState({ userInput: value, }); } // Add item if user input in not empty addItem() { if (this.state.userInput !== "") { const userInput = { // Add a random id which is used to delete id: Math.random(), // Add a user value to list value: this.state.userInput, }; // Update list const list = [...this.state.list]; list.push(userInput); // reset state this.setState({ list, userInput: "", }); } } // Function to delete item from list use id to delete deleteItem(key) { const list = [...this.state.list]; // Filter values and leave value which we need to delete const updateList = list.filter((item) => item.id !== key); // Update list in state this.setState({ list: updateList, }); } editItem = (index) => { const todos = [...this.state.list]; const editedTodo = prompt('Edit the todo:'); if (editedTodo !== null && editedTodo.trim() !== '') { let updatedTodos = [...todos] updatedTodos[index].value= editedTodo this.setState({ list: updatedTodos, }); } } render() { return (TODO LIST this.updateInput(item.target.value) } aria-label="add something" aria-describedby="basic-addon2" /> this.addItem()} > ADD {/* map over and print items */} {this.state.list.map((item, index) => { return ({item.