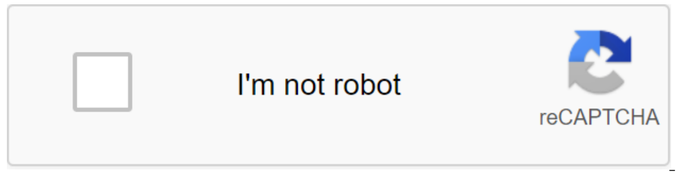

Continue

React js tutorialspoint pdf

This tutorial does not involve any existing Knowledge Of React. Before we start the tutorial we will build a small game during this tutorial. You may be tempted to miss it because you don't build games - but give it a chance. The techniques you learn in the tutorial are fundamental to building any React application, and mastering it will give you a deep understanding of React. Tip This tutorial is designed for people who prefer to learn by doing. If you prefer to learn concepts from scratch, check out our step-by-step guide. You can find this tutorial and guides complement each other. The tutorial is divided into several sections: You don't have to complete all the sections at once to get value out of this tutorial. Try to get as far as you can - even if it's one or two sections. What are we building? In this tutorial we will show how to build an interactive tick-tok-to-knock game with React. You can see what we will build here: The final result. If the code doesn't make sense to you, or if you're not familiar with code syntax, don't worry! The purpose of this tutorial is to help you understand React and its syntax. We recommend that you check out the tick-to-knock game before continuing with the tutorial. One of the features that you will notice is that there is a omemer list to the right of the game board. This list gives you a history of all the moves that occurred in the game, and it is updated as the game progresses. You can close tick-to-take games as soon as you are familiar with it. We'll start with a simpler template in this tutorial. Our next step is to set you up so that you can start building the game. Premise We assume that you have some familiarity with HTML and JavaScript, but you should be able to follow along even if you come from another programming language. We'll also assume that you're familiar with programming concepts such as features, objects, arrays, and, to a lesser extent, classes. If you need to view JavaScript, we recommend reading this guide. Please note that we also use some features from ES6, the recent version of JavaScript. In this tutorial we use the functions of arrows, classes, albeit const statements. You can use Babel REPL to check what the ES6 code compiles. Setting up for a tutorial there are two ways to complete this tutorial: you can either write the code in your browser, or you can customize the local development environment on your computer. Setting up Option 1: Write code in your browser Is the fastest way to get started! First, open this starter code in the new tab. The new tab should display an empty tic-tac-toe board and React code. We will edit the React code in this tutorial. You can now skip the second option and go to the Review section to get an React review. Option 2: Local development development This is completely optional and is not required for this tutorial! Optional: Instructions for following along locally using your preferred text editor This setup requires more work but allows you to complete the tutorial with the help of the editor of your choice. Here are the steps to follow: npx create-responsive-app my-app Delete all files in the src/new project note folder: Don't delete the entire src folder, just the original files inside it. The next step is to replace the original files with default examples for this project. CD my-app CD src rm-f and del th CD .. Add a file called index.css to the src/css folder. Add a file called index.js to the src/folder folder with this JS code. Add these three lines to the top of index.js in the src/: Import React from 'react'; ReactDOM import from reaction-home; Imports './index.css'; Now, if you run npm, start in the project folder and in the browser, you'll see an empty tick-tok-so-nou box. We recommend following these instructions to set up a selection syntax for your editor. Help, I'm stuck! If you're stuck, check out community support resources. In particular, Reactiflux Chat is a great way to get help quickly. If you have not received an answer, or if you are stuck, please submit a question and we will help you. Review Now that you're tuned in, let's review React! What is a reaction? React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It allows you to make complex UIs out of small and isolated pieces of code called components. React has several different types of components, but we'll start with The React.Component subclass: The ShoppingList class expands React.Component and render () (s)); We'll soon get to the fun XML tags. We use components to tell React what we want to see on the screen. When our data changes, React will effectively update and redraw our components. Here, ShoppingList is a React component class or a type of React component. The component accepts parameters called props (short of property) and returns the view hierarchy to display using the rendering method. The rendering method returns a description of what you want to see on the screen. React takes the description and displays the result. Specifically, visualization returns the React element, which is an easy description of what to visualize. Most React developers use a special syntax called JSX, which makes it easier to write these structures. The syntax is converted during assembly ('div'). Приведенная выше пример эквивалентна: return React.createElement ('div', «className: 'shopping-list'», React.createElement('h1',),),)); See the full extended version. If you're interested, createElement is described in more detail in the API link, but we won't use it in this tutorial. Instead, we will continue to use JSX. JSX comes with full JavaScript power. You can put any JavaScript expression in brackets inside JSX. Each React element is a JavaScript object that can be stored in a variable or transmitted in a program. The ShoppingList component above only displays the built-in components of the DOM, such as the zlt'div'gt;ltand the zlt'gt.lt. But you can compose and visualize the custom components of React too. For example, we can now refer to the entire shopping list by writing a shopping list. Each React component is encapsulated and can work independently; this allows you to create complex AI from simple components. Check the start code If you're going to be working on a tutorial in your browser, open this code in a new tab: Starter Code. If you're going to work on the tutorial locally, instead open src/index.js in the project folder (you've already touched this file while you're setting it up). This starter code is the basis of what we are building. We've provided CSS style so that you only need to focus on learning React and programming tick-to-take games. When you check the code, you'll notice that we have three React components: the Square component displays one, and the board has nine squares. The game component displays a board with placeholder values that we'll change later. There are currently no interactive components. Transfer data through props To get our feet wet, let's try to transfer some data from our board component to our component square. We strongly recommend entering the code manually as you work through the tutorial rather than using copy/paste. This will help you develop muscle memory and better understanding. In the renderSquare Council method, change the code to convey props called value to square: The Council class expands React.Component - renderSquare (i) Change Square visualization method to show this value by replacing TODO with this.props.value: class square extends React.Component (component) zlt'button classname'square'gt;lt;buttongt; Before: You should see the number in each square in the visualized output. See the full code at the moment Congratulations! You've just passed the props from the parent board component to the children's square component. Transferring props is how information flows in React apps, from parents to children. Create an interactive component Let's fill in the Square X component when we click on it. First, change the button tag that returns from the component rendering function () <button classname=square onclick={function() {= alert('click');= }}=></button> на эту: класс Square расширяет React.Component</button> и render () - return (( ); If you click on the square now, you should see the alert in your browser. Note To save the typing and avoid the confusing behavior of this, we will use the arrow syntax function for event handlers here and beyond below: square class expands React.Component (button classname) square onclick ()/lt;)/button/gt; Note, as with onClick () warning ('click') (React will only trigger this feature after clicking. Forgetting and writing on 'click'alert' is a common mistake, and will shoot alerts every time the component re-makes. As a next step, we want the Square component to remember what was clicked on and fill it with the X mark. To remember things, components use a state. React components can have a status, placing this.state in its designers. this.state should be seen as a private component of React, to which it is defined. Let's save the current value of the area in this.state, and change it when the square is clicked. First, we'll add a constructor to the class to initiate the state: the class square extends React.Component - constructor (props) - super (props); The value of the javascript, null, note JavaScript classes, you should always call super when determining a subclass designer. All React component classes that have a constructor should start with a super (props) method of rendering Square to display the current state value when clicked: Replace this.props.value at this.state.value. After these changes, the tag returned by Square's rendering is as such: Square class expands the button-gt.Component - constructor (props) - super (props); this.state - value: null, q; - render () - return q (this.setState(meaning: 'X'); , we're talking React to re-set aside this area whenever it's pressed. After the update, the area of this.state.value will be 'X', so we'll see X on the game board. If you click on any area, X should appear. When you call setState in the React component, it automatically updates the children's components inside it. Viewing the full code at the moment, the React Devtools extension for Chrome and Firefox allows you to check the tree of React components using the browser developer's tools. React DevTools will allow you to check out the props and the button </button> </button> React. Once you install React DevTools,

you can right click on any item on the page, click the Check button to open the developer's tools, and the React tab (⚛ Components and ⚛ Profiler) will appear as the latest tabs on the right. Use ⚛ components to test the component tree. However, please note there are a few additional steps to make it work with CodePen: login or sign up and confirm your email (required to prevent spam). Click the Fork button. Click change the view and then select Debugging Mode. In the new tab that opens, devtools must now have an React tab. Completion of the game Now we have the main building blocks for our tick-tok-to-nos game. To have a full game, we now need to alternate the placement of X and O on the board, and we need a way to determine the winner. Lifting the state now, each component of the square supports the state of the game. To test the winner, we will maintain the value of each of the 9 squares in one place. We may think that the council should just ask every area for the state of the square. While this approach is possible in React, we don't recommend it because the code becomes difficult to understand, error-prone and difficult to refactor. Instead, the best approach is to store the game in the parent component of the board, not in each area. The component of the board can tell each area that to display, passing props, just as we did when we passed the room on each square. To collect data from multiple children or to communicate with two components of the child with each other, you should instead declare the general condition in the parent component. The parent component can transfer the condition back to the children with the help of props; this keeps the child's components in sync with each other and with the parent component. Lifting the state into the parent component is common when the components of React refactoring - let's take this opportunity to try it out. Add the designer to the board and establish the initial state of the Council to contain an array of 9 zeros corresponding to 9 squares: Class Tip expands React.Component - designer (prop) - super (prop); this.state and squares: Array (9). When we fill the board later, the array of this.state.squares will look like this: 'O', null, 'X', 'X', 'X', 'O', 'O', null, null, - renderSquare Council method now looks like this: renderSquare (i) qlt;Square value';zgt;qth;qth;square/square/square/gt; In the beginning, we passed the prop value down from the board to show the numbers from 0 to 8 in each square. In another previous phase, we replaced the numbers with X, which is determined by our own state That's why Square is now ignoring the value of the props handed over to him by the council. Now we will use the mechanism of passing props again. We're going to change. Tip to instruct each individual square on its current value ('X', 'O', or null). We have already identified an array of squares in the Council Designer, and we will change the renderSquare Council method to read from it: renderSquare (i) qlt;Square value'this.state.squares View the full code at this point Each square will now get value props, which will be either X, O, or zero for empty squares. Next, we have to change what happens when the square is clicked. The Council component now supports which squares are filled. We need to create a way for the square to update the council's fortunes. Since the state is considered a private component, we cannot update the Council's condition directly from the area. Instead, we'll move the feature from the board to the square, and we'll have a square call to this feature when the square is clicked. We will change the renderSquare method in the Board of Directors to: renderSquare (i) - return of the zlt;Square value'this.state.squares'onclick ()(this.handleClick (i)); Please note that we've divided the returned item into a few lines for readability and added brackets to prevent JavaScript from inserting a column after returning and breaking our code. Now we go down two props from the board to the square: value and onClick. OnClick Props is a feature that square can cause when clicked. We will make the following changes in Square: Replace this.state.value with this.props.value in the Square Render method To replace this.setState () with this.props.onClick() in the area render method Remove the Designer from the square, because square no longer tracks the state of the game After these changes, the Square component looks like this: the square of the class extends React.Component (render) - return of the zlt'square onclick ()(this.props.onClick.). When you click on a square, the onClick function provided by the Council is called. Here's an overview of how this is achieved: OnClick's reliance on the built-in component of the DOM is a button to tune in to the event listener button. When the button is pressed, React will call the onClick event handler, which is defined in the area visualization method. This event handler calls this.props.onClick. The area onClick props was listed by the Council. Ever since the board switched to Click (i) - this.handleClick (i) - on the square, The Square has been calling this.handleClick (i) when clicking. We haven't defined the handleClick method yet, so our code is being cut off. If you click on the square now, you should see red screen bugs saying something like this.handleClick is not a feature. Please note that the DOM onClick attribute is of particular importance to React because it is a built-in component. For custom components such as Square, naming is yours. We are бы дать любое имя площади onClick реквизит или совета</button> </button> </Square> </Square> method, and the code will work the same way. React is commonly used on event names for props representing events and processing for methods that process events. When we try to click on the square, we have to get the error because we have not yet identified handleClick. Now we'll add handleClick to the Board of Directors class: The Board expands React.Component - Designer (prop) - super (prop); this.state - squares: Array (9).fill (null), zlt;Square value'this.state.squares'onclick()'gt; this.handleClick (i);); - Render - status const - Next player: X; return (this.renderSquare (0) (this.renderSquare (1) (this.renderSquare (2) (this.renderSquare (3) (this.renderSquare (4) (this.renderSquare (5) this.renderSquare (6) (this.renderSquare (7) (this.renderSquare (8) View the full code at this point After these changes, we can again click on the squares to fill them, Just like we had before. Square components receive values from the Board component and inform the Board component when pressed. From an React perspective, Square components are now controlled by components. The Council has full control over them. Notice how in handleClick we call .slice to create a copy of the array of squares to change instead of changing the existing array. We'll explain why we're creating a copy of the array of squares in the next section. Why immutability is important in the previous code example, we suggested using the .slice method to create a copy of the square array to change instead of changing the existing array. Now we will discuss immutability and why immutability is important to study. Typically, there are two approaches to changing the data. The first approach is to mutate the data by directly changing the values of the data. The second approach is to replace the data with a new copy that has the desired changes. Changing the data with the var player mutation (estimate: 1, name: 'Jeff';; player.score No 2; Changing data without mutation var player (estimate: 1, name: 'Jeff'; var newPlayer - Object.assign(me, player, 'score: 2'):; the end result is the same, but without mutating (or changing the baseline data) directly, we get a few benefits described below.makes complex functions much easier. Later in this tutorial, we'll implement a time-travel feature that allows us to review tick-to-the-like game history and jump back to previous moves. This functionality is not specific to games - the ability to cancel and re-hunt certain actions is a common requirement in applications. Avoiding direct data mutations allows you to keep previous versions of the game's history intact, and reuse them later. It is difficult to detect changes in variable objects because they are modified directly. This detection requires that the mutated object be compared to previous copies of itself and the entire tree of objects that must be passed. Detecting changes in immutable objects is much easier. If the immutable object it refers to is different from the previous one, the object has changed. Determining when to re-render in React the main advantage of immutability is that it helps you create clean components in React. Unchanging data can easily determine whether changes have been made, which helps determine when a component needs to be re-rendered. You can learn more about shouldComponentUpdate and how you can create clean components by reading Performance Optimization. Function components We will now change the square to be a component of the function. In React, function components are an easier way to write components that contain only a rendering method and don't have their own state. Instead of defining a class that extends React.Component, we can write a feature that takes props as input and returns what needs to be drawn. Functional components are less tedious to write than classes, and many components can be expressed in this way. Replace the Square class with this feature: Square (props) - return to the button classname'square onclick's (props.onClick); (props.value); We've changed this.props to props both times it appears. Reviewing the full code at the moment Note When we changed the square to be a feature component, we also changed onClick () - this.props.onClick () to shorter onClick'props.onClick (note the lack of brackets on both sides). By turns now we need to fix the obvious flaw in our tick-tok-so-nou game: O can't be marked on the board. We will set the first step to be the X by default. We can set this default by changing the original state in our Board of Directors: The Board class expands React.Component - constructor (props) - super (props); this.state - squares: Array (9).fill (null), xIsNext: true, Every time a player moves, xIsNext (bulean) will be flipped to determine the player goes further and the state of the game will be saved. We will update the handleClick Council function to flip the value of xIsNext: handleClick (i) - squares const - this.state.squares.slice (); squares and this.state.xIsNext? 'X': 'O'; this.setState (me) (me) квадраты, xIsNext: !this.state.xIsNext, No); С этим изменением, X и O может по очереди. Попробуй! Давайте также изменим текст статус в рендере Совета, чтобы он отображал, какой игрок имеет следующий поворот: рендер () - статус const Следующий игрок: ' (this.state.xIsNext ? 'X' : 'O'); возвращение (После применения этих изменений, вы должны иметь этот компонент совета директоров: класс Совет расширяет React.Component - конструктор (проп); супер (props); this.state - квадраты: Array(9).fill(null), xIsNext: true, 'X': 'O'; this.setState (квадраты: квадраты, xIsNext: !this.state.xIsNext, No); - renderSquare (i) - возврат (<Square value={this.state.squares[i]} onclick={() ==> this.handleClick (i)» /); - Рендер () - статус const - Следующий игрок: (this.state.xIsNext? 'X' : 'O'); возвращение (<div><div classname=status>(статус)</div><div classname=board-row> «this.renderSquare (0)» (this.renderSquare(1)» (this.renderSquare(2)» </div><div classname=board-row> (this.renderSquare(3)» (this.renderSquare(4)» (this.renderSquare(5)» </div><div classname=board-row> «this.renderSquare(6)» (this.renderSquare(7)» (this.renderSquare(8)» </div></div>); Посмотреть полный код на данный момент Объявление победителя Теперь, когда мы показываем, какой игрок очередь следующая, мы также должны показать, когда игра выиграна, и Есть не более поворотов, чтобы сделать. Скопировать эту функцию помощника и вставить его в конце файла: функция calculateWinner (квадраты) 1, 2, 3, 4, 5, 6, 7, 8, «0, 3, 6», «1, 4, 7», «2, 5, 8», «0, 4», «8» , «2, 4, 6»; < lines.length; i++) { const [a, b, c] = lines[i]; if (squares[a] && squares[a] === squares[b] && squares[a] === squares[c]) { return squares[a]; } } return null; } Given an array of 9 squares, this function will check for a winner and return 'X', 'O', or null as appropriate. We will call calculateWinner(squares) in the Board's render function to check if a player has won. If a player has won, we can display text such as “Winner: X” or “Winner: O”. We'll replace the status declaration in Board's render function with this code: render() { const winner = calculateWinner(this.state.squares); let status; if (winner) { status = 'Winner: ' + winner; } else { status = 'Next player: ' + (this.state.xIsNext ? 'X' : 'O'); } return (We can now change the Board's handleClick function to return early by ignoring a click if someone has won the game or if a Square is already filled: handleClick(i) { const squares = this.state.squares.slice(); if (calculateWinner(squares) || squares[i]) { return; } squares[i] = this.state.xIsNext ? 'X' : 'O'; this.setState({ squares: squares, xIsNext: !this.state.xIsNext, }); } View the full code at this point Congratulations! You now have a working tic-tac-toe game. And you've just ></Square> ></Square> The basics of React too. So you're probably the real winner here. Adding Time Travel as a final exercise, let's make it possible to go back in time to previous moves in the game. Keeping the history of motion If we mutated an array of squares, realizing time travel would be very difficult. However, we used the slice to create a new copy of the array of squares after each turn, and viewed it as immutable. This will allow us to store each past version of the array of squares and move between the turns that have already happened. We will store arrays of past squares in another array called history. The array of history represents all the states of the board, from the first to the last turn, and has a form like this: history - squares: zero, zero, zero, zero, q, squares: - null, null, null, null, null, null, lifting state up, again we want the component of the game to display a list of past moves. To do this, he will need access to the story, so we will place the state of history in the component of the game of the highest level. Placing the state of history in the Game component allows you to remove the state of the squares from the childboard component. Just as we've raised the fortune from Square's component to the Board component, we're going to take it from the board to the top-level game component. This gives the game component complete control over the Board's data and allows it to instruct the Board to make previous twists of history. First, we will adjust the original state for the game component within its designer: the class of the game expands React.Component - designer (props) - super (prop); this.state - History: squares: Array (9).fill (null), q, xIsNext: true, Div classname game-info.gt;); Next, we'll have a board component get squares and onClick props from the game component. Since we now have a one click handler in the board for many squares, we have to go through the location of each square in onClick handler to indicate which area has been clicked. Here are the necessary steps to transform the board component: Remove the designer in the board. Replace this.state.squares'i with this.props.squares'i in renderSquare Council. Replace this.handleClick (i) at this.props.onClick (i) in renderSquare Council. The board component now looks like this: Class Tip expands React.Component - penClick (i) - const squares - this.state.squares.slice (); if (calculateWinner (squares) 'X': 'O'; this.setState (squares: squares, xIsNext: !this.state.xIsNext, - renderSquare (i) - возврат (<Square value={this.props.squares[i]} onclick={() ==> this.props.onClick (i)) - рендер () - победитель конкурса - вычислитеWinner (this.state.squares); пусть статус; если (победитель) - статус Победитель: Победитель; 'X' : 'O'); - возврат (<div><div classname=status>(статус)</div><div classname=board-row> «this.renderSquare (0)» (this.renderSquare(1)» (this.renderSquare(2)» </div><div classname=board-row> (this.renderSquare(3)» (this.renderSquare(4)» (this.renderSquare(5)» </div><div classname=board-row> «this.renderSquare(6)» (this.renderSquare(7)» (this.renderSquare(8)» </div></div>); Мы обновим функцию рендер-рендеров компонента игры, чтобы использовать самые последние записи в истории для определения и отображения состояния игры: визуализации () const текущий - history'history.length - 1 ; победитель const - calculateWinner (current.squares); пусть статус; если (победитель) - статус Победитель: Победитель; 'X' : 'O'); - возврат (<div classname=game><div classname=game-board><Board squares={current.squares} onclick={(i) ==>this.handleClick (i)</Board></div><div classname=game-info><div>(статус)</div> {} </div></div>); Поскольку компонент Игры в настоящее время оказывает статус игры, мы можем удалить соответствующий код из метода рендеринга Совета. После рефакторинга функция рендеринга Совета выглядит так: рендер () - возврат (<div><div classname=board-row> «this.renderSquare (0)» (this.renderSquare(1)» (this.renderSquare(2)» </div><div classname=board-row> (this.renderSquare(3)» (this.renderSquare(4)» (this.renderSquare(5)» </div><div classname=board-row> «this.renderSquare(6)» (this.renderSquare(7)» (this.renderSquare(8)» </div></div>); Наконец , нам нужно переместить метод handleClick из компонента Доски в компонент игры. Мы также должны изменить handleClick, потому что состояние компонента игры структурировано по-разному. В методе handleClick игры мы сокатим новые записи истории в историю. handleClick(i) - история const - this.state.history; const ток - история.история.длина - 1; конст квадраты - current.squares.slice (); если (calculateWinner (квадраты) 'X': 'O'; this.setState (история: history.concat(квадраты: квадраты, q), xIsNext: !this.state.xIsNext, q); Заметьте, в отличие от метода нажима массива () вы можете быть более знакомы с, concat () метод не мутирует исходный массив, поэтому мы предпочитаем его. На данный момент компоненту Совета директоров нужны только методы renderSquare и визуализации. Состояние игры и метод handleClick должны быть в компоненте игры. Посмотреть полный код на данный момент Показаны прошлые движения, так как мы записываем тик-так-нок игры </Square&gtWe can now display it to the player as a list of past moves. We've learned before that React elements are first-class JavaScript objects; We can pass them around in our applications. We can use an array of React elements to visualize multiple elements in React. In JavaScript, arrays have a map method, which is typically used to display data on other data, such as const numbers No.1, 2, 3; const twice - numbers.map (x x x 2); Using the map method, we can map our history of moves to the React elements that represent the buttons on the screen, and display a list of buttons to jump on past moves. Let's draw a story in the render rendering method of the game: render () - history const - this.state.history; const current - history.history.length - 1 ; winner const - calculateWinner (current.squares); const moves - history.map (step, move) - const desc - move? 'Go to move' and move: 'Go to the game start'; return (the button onclick) Let the status; If (winner) - Winner status: - winner; - another status - Next player: (this.state.xIsNext? 'X' : 'O'); - возврат (<div classname=game><div classname=game-board><Board squares={current.squares} onclick={(i) ==>this.handleClick (i)</Board></div><div classname=game-info><div>(статус)</div> (движения) </div></div>); Посмотреть полный код на данный момент для каждого хода в истории тик-так-нок игры, мы создаем список пункта <li> которая содержит кнопку <button>. Кнопка имеет обработчик onClick, который вызывает метод под названием this.jumpTo (). Мы еще не реализовали метод jumpTo () еще. На данный момент, мы должны увидеть список ходов, которые произошли в игре и предупреждение в консоли инструментов разработчика, который говорит: Предупреждение: Каждый ребенок в массиве или итератор должен иметь уникальный ключ реквизит. Проверьте метод рендера Игра. Давайте обсудим что means the above warning. Choosing the key when visualizing the List React stores some information about each item on the list. When we update the list, React needs to determine what has changed. We could add, delete, overwork or update list items. Imagine the transition from zlt'gt)Alexa: there are 7 tasks left. To whom there are 9 tasks left: in addition to the updated calculations, the person reading this probably changed the order that we Alexa and Ben had an order and put in between Alexa and Ben. However, React is a computer program and does not know what we intended. Because React is not To know about our intentions, we need to specify a key property for each item on the list to differentiate each item of the list from its siblings. One option would be to use the lines of Alexa, Ben, Claudia. If we were to display data from the database, Alexa, Ben, and Claudia database documents could be used as keys. </button> ></button> user.taskCount tasks left when re-visualizing the list, React takes the key of each item of the list and searches for elements of the previous list for the right key. If there's a key on the current list that didn't exist before, React creates a component. If you don't have a key on the previous list on the current list, React destroys the previous component. If the two keys are the same, the corresponding component moves. Keys tells React about identifying each component, allowing React to maintain a state between redraces. If the component key changes, the component will be destroyed and re-created with a new state. The key is a special and reserved property in React (along with the referee, a more advanced function). When an item is created, React removes the key property and stores the key directly to the returned item. Even if the key may look like it belongs in props, the key may not be referenced using this.props.key. React automatically uses the key to decide which components to upgrade. The component can't tell you how it's key. It's highly recommended to assign the right keys whenever you build dynamic lists. If you don't have the appropriate key, you may want to consider restructuring your data, so what do you do. If the key is not specified, React will provide a warning and use the array index as the default key. Using the array index as a key is problematic when trying to re-enter list items or insert/delete list items. Explicit key passage will drown out the warning, but has the same problems as array indices, and in most cases is not recommended. Keys don't have to be globally unique; they should only be unique between the components and their brothers and sisters. Implementation of Time Travel In the history of the tick-tok-tack game, each last move has a unique identifier associated with it: it's a consistent number of moves. The moves are never oversused, removed or inserted in the middle, so you can safely use the travel index as a key. In the game component visualization method, we can add a key as a key and React's key warning should disappear: const moves - history.map (step, move) - const desc - move? 'Go to move' and move: 'Go to the game start'; return (the key) Viewing the full code at the moment, clicking any of the list element buttons makes a mistake because the jumpTo method is not defined. Before we implement jumpTo, we'll add stepNumber to the game component state to indicate what step we're currently reviewing. First, add stepNumber: 0 to the original state in Games: Class game expands React.Component - Designer (props) - super (props); this.state - Story: Next, we'll identify the jumpTo method in the game to update this stepNumber. We're also installing xIsNext to get it right if it's a good time. what we change stepNumber to even: handleClick (i) - jumpTo (step) - this.setState (stepNumber: step, xIsNext: (step % 2) - 0, q); The stepNumber state we've added reflects the move the user is now seeing. Once we take a new step, we need to update stepNumber by adding stepNumber: history.length as part of this.setState argument. This ensures that we don't get stuck showing the same move after the new one has been made. We will also replace reading this.state.history with this.state.history.slice (0, this.state.stepNumber No 1). This ensures that if we go back in time and then take a new step from this point, we throw away all the future stories that would now become wrong. handleClick (i) - history const - this.state.history.slice (0, this.state.stepNumber) 1); Const current - history.history.length - 1 ; squares - current.squares.slice (calculateWinner (squares) 'X': 'O'; this.setState (history: history.concat (squares: squares), stepNumber: history.length, xIsNext: !this.state.xIsNext, No); Finally, we will change the game component rendering method from always rendering the last move to rendering the chosen one currently moving according to stepNumber: render () If we click on any step in the history of the game, the tick-so-knock board should immediately update to show what the board looked like after this step happened. You created the game tic-tac-toe Which: Allows you to play tick-t-nose, indicates when a player has won a game, keeps the history of the game as the game progresses, allows players to review the history of the game and see previous versions of the game board. Good work! We hope that now you feel like you have a decent understanding of how React works. Check out the final result here: The final result. If you have extra time or want to practice your new React skills, here are some ideas for improvements that you can make to the tic-tac-toe game, which are listed in order of increasing complexity: Displaying the location for each turn in the format (col, row) in the travel history list. The bold item currently selected is on the travel list. Rewrite the board to use two cycles to make squares instead of hard coding them. Add a switch button that allows you to sort the moves in order of ascent or descent. When someone wins, highlight the three squares that sparked the victory. When no one wins, displaying the result message is a draw. Throughout this lesson, we have touched on React concepts, including elements, components, props, and status. Handicap a detailed explanation of each of these topics, check the rest of the documentation. To learn more about component definitions, check out the Link.Component API link. Links. react js tutorialspoint pdf. react js interview questions tutorialspoint. tutorialspoint react js environment setup