**Documentation for TIC-TAC-TOE Game in ReactJS**

1. Mainly four components of ReactJS are used in this game:

* Components
* Props
* State
* Time Travel

1. First of all, the Board is build using HTML and CSS. And all the things are defined in the board Component.
2. All the data is passed among various components and functions using the **PROPS** .
3. Mainly there are two components:

* **Board.js**
* **Square.js**

IMPORTANT POINTS:

* Each component state is private, in our case we cannot change Board component state directly. Instead we will pass a function down to the square component from board component.
* JavaScript supports closures which means an inner function (e.g. handleClick) has access to variables and functions defined in a outer function (e.g. Board).

1. The **USESTATE** is used to manage the state within a component as it stores the state of each square to make it responsive to user interaction.
2. Overall game state is managed by lifting state up to the Board component.
3. A function is passed from the board component to the Square component which will handle the clicks.
4. The handleClick() function can read the squares state and call the setSquares method because they are both defined inside of the Board function.
5. We cannot directly pass the value to the handleClick() function as it will call the function again and again leading to an infinite loop.
6. Instead of defining different functions according to the values, we will iterate them using an arrow function.
7. State needs to be updated in the Board component to reflect changes in the game board.

Important:

* The DOM <button> element’s onClick attribute has a special meaning to React because it is a built-in component. For custom components like Square, the naming is up to you. You could give any name to the Square’s onSquareClick prop or Board’s handleClick function, and the code would work the same. In React, it’s conventional to use onSomething names for props which represent events and handleSomething for the function definitions which handle those events.

1. The concept of immutability is used in this game by performing all the operations on the copy of the array instead of modifying the original array. Copy is done using the **.slice()** method in the **handleClick()** function.
2. Immutability makes it easier to debug the code and to implement undo and redo functionalities easily.

**Setting Up Turn-Based Mechanics**

1. **Default First Move**:
   * Initialize the game with 'X' as the first player.
   * Use a boolean state variable xIsNext to track the current player.
2. **Alternating Turns**:
   * Flip the value of xIsNext each time a player makes a move to alternate turns between 'X' and 'O'.

**Preventing Overwrites**

1. **Preventing Overwrites on Click**:
   * Before marking a square, check if it is already filled.
   * Return early from the handleClick function if the square already contains 'X' or 'O'.

**Declaring a Winner**

1. **Winner Calculation Function**:
   * Implement a helper function calculateWinner to check the board for a winner.
   * Return 'X', 'O', or null based on the game state.
2. **Check for Winner**:
   * In the handleClick function, check for a winner before allowing further moves.
3. **Display Winner or Next Player**:
   * Add a status section to the Board component to display the current game state (winner or next player's turn).

**Adding Time Travel**

1. **Immutable State for History**:
   * Use slice() to create a new copy of the squares array after every move to ensure immutability.
   * Store each version of the board state in a history array.

**Lifting State Up**

1. **Creating a Game Component**:
   * Create a new Game component to manage the game state, including the history of moves and the current player.
   * Lift state from the Board component to the Game component.

**Managing State in Game Component**

1. **Tracking Moves and Player**:
   * Use useState in the Game component to track history, currentMove, and derive xIsNext from currentMove.
2. **Handle Play Function**:
   * Implement handlePlay to update the history and currentMove state when a player makes a move.
   * Pass xIsNext, currentSquares, and handlePlay as props to the Board component.

**Rendering Based on Props**

1. **Controlled Component**:
   * Modify the Board component to receive xIsNext, squares, and onPlay as props.
   * Remove local state from Board, making it a controlled component.

**Implementing Time Travel**

1. **Displaying Past Moves**:
   * Map over the history array to create a list of buttons representing each past move.
   * Add an onClick handler to each button to call jumpTo.
2. **Unique Keys for List Items**:
   * Use the move index as a key for each list item to help React efficiently update and re-render the list.
3. **Jump To Function**:
   * Implement jumpTo to update currentMove and adjust xIsNext based on the move index.
4. **Update History Management**:
   * When making a new move, update the history to discard any future moves beyond the current step.
   * Always update currentMove to the latest move.

**Final Cleanup**

1. **Remove Redundant State**:
   * Eliminate the xIsNext state variable in favor of deriving it from currentMove.
   * Simplify state management to avoid potential bugs and improve code clarity.

**Final Features**

1. **Interactive Features**:
   * Ensure the game allows players to review past moves and see the board state at each step.
   * Highlight the winning combination when a player wins.
   * Provide messages for draw situations and current move status.

**Additional Improvements**

1. **Potential Enhancements**:
   * Show "You are at move #" for the current move.
   * Use loops to dynamically create squares in the Board component.
   * Add a toggle for sorting move history.
   * Highlight the winning squares and display draw messages.
   * Show the move locations in (row, col) format in the history list.

This breakdown covers the key steps and concepts in implementing the described tic-tac-toe game using React, emphasizing the importance of state management, immutability, and lifting state up for better component interaction and time travel feature.