Web Framework Tutorial 02

Step-by-Step React Development

Danqing Shi

Setup for the Tutorial

Visual Studio Code

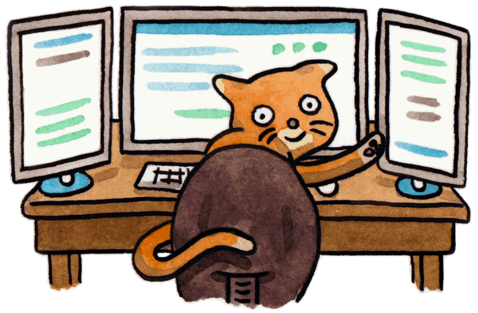
https://code.visualstudio.com/

Node: an asynchronous event driven JavaScript runtime.

https://nodejs.org

Yarn: fast, reliable, and secure dependency management.

https://yarnpkg.com



Create React App

Create React App is a comfortable environment for **learning React**, and is the best way to start building **a new single-page application** in React.

https://github.com/facebook/create-react-app

To create a new app, you may choose one of the following methods (yarn is recommended) . So you can set up a modern web app by running one command.

Npm: npm init react-app my-app

Yarn: yarn create react-app my-app

Create React App

It will create a directory called my-app inside the current folder.

Inside that directory, it will generate the initial project structure and install the transitive dependencies.

No configuration or complicated folder structures, just the files you need to build your app.

```
my-app
├── README.md
├── node_modules
├── package.json
├── .gitignore
├── public
│   ├── favicon.ico
│   ├── index.html
│   └── manifest.json
└── src
    ├── App.css
    ├── App.js
    ├── App.test.js
    ├── index.css
    ├── index.js
    ├── logo.svg
    └── serviceWorker.js
```

https://reactjs.org/docs/create-a-new-react-app.html

Create React App

Once the installation is done, you can open your project folder:

- ➢ cd my-app

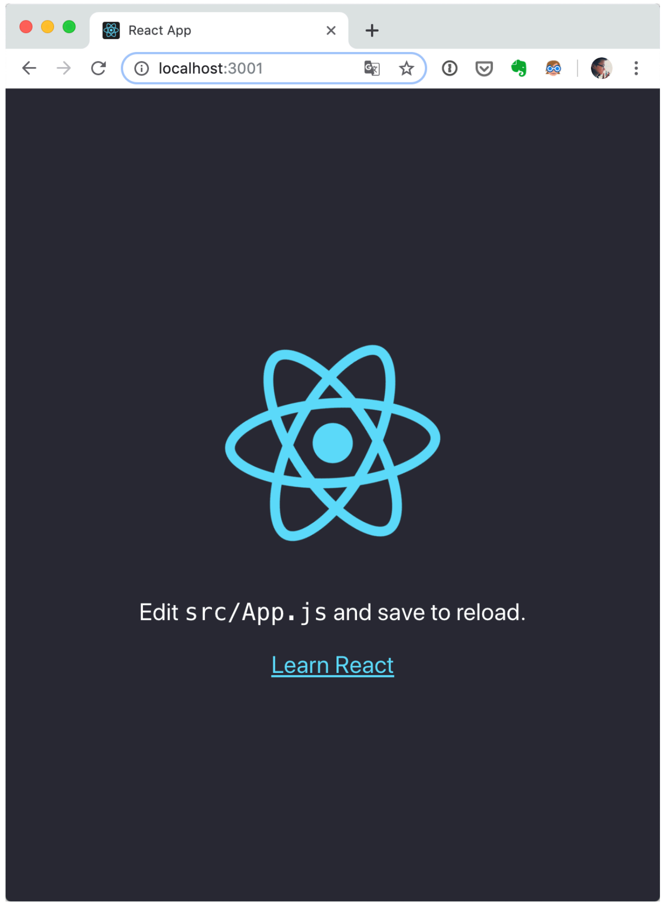
Inside the newly created project, you can run some built-in commands:

- ➢ yarn start (or npm start)

Runs the app in development mode.
Open http://localhost:3000 to view it in the browser.

Builds the app for production to the build folder.:

- ➢ yarn build (or npm run build)

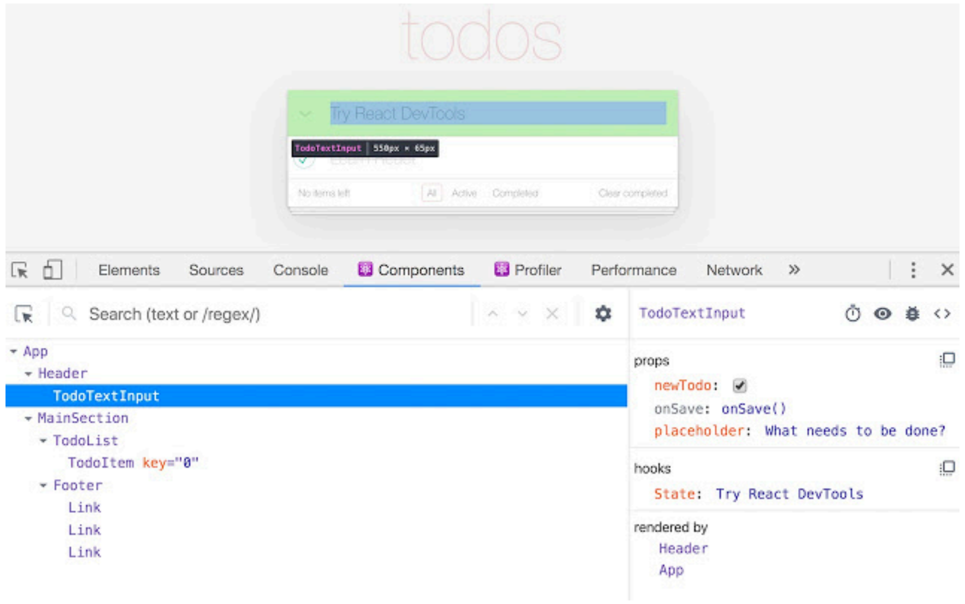


Edit `src/App.js` and save to reload.

React Developer Tools

React Developer Tools is a Chrome DevTools extension for the open-source React JavaScript library. It allows you to inspect the React component hierarchies in the Chrome Developer Tools.

https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en



```
▾ App
  ▾ Header
      TodoTextInput
  ▾ MainSection
    ▾ TodoList
        TodoItem key="0"
    ▾ Footer
        Link
        Link
        Link
```

```
TodoTextInput
props
  newTodo: ✓
  onSave: onSave()
  placeholder: What needs to be done?
hooks
  State: Try React DevTools
rendered by
  Header
  App
```

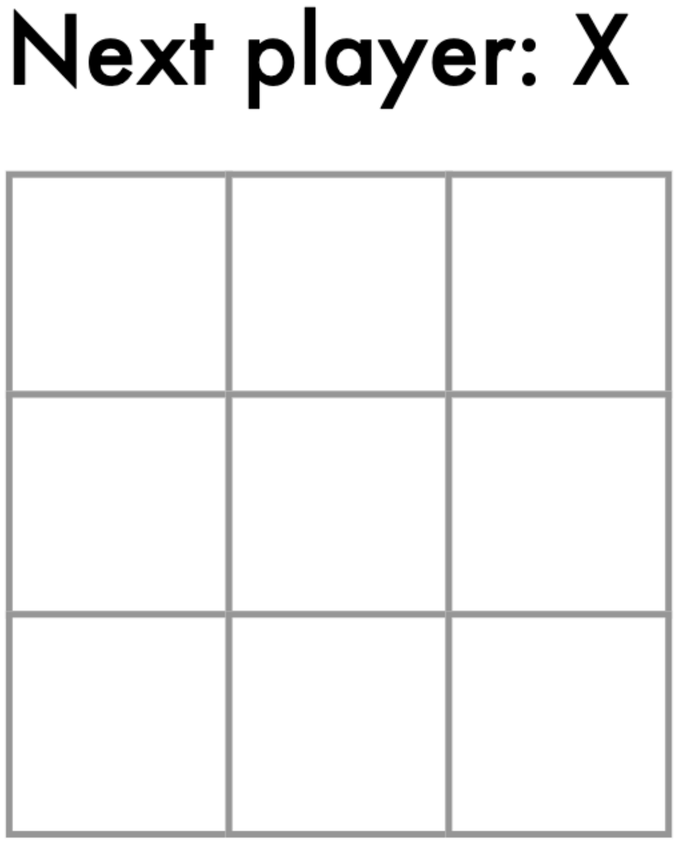
Starter Code – Tic Tac Toe Game

This Starter Code is the base of what we're building. We've provided the CSS styling so that you only need to focus on learning React and programming the tic-tac-toe game.

By inspecting the code, you'll notice that we have three React components:

- Square
- Board
- Game

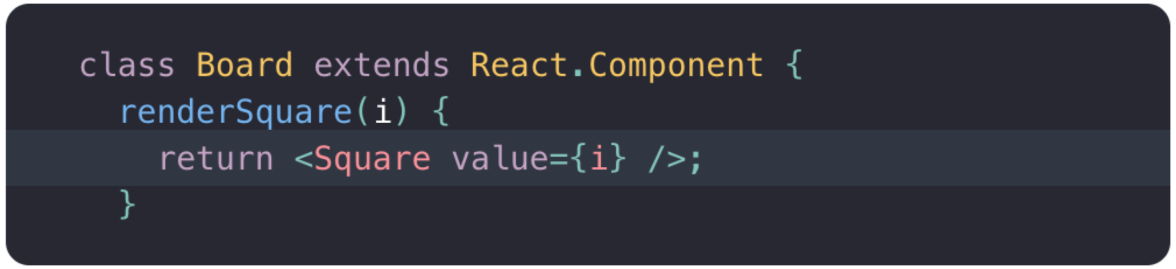
The Square component renders a single <button> and the Board renders 9 squares. The Game component renders a board with placeholder values which we'll modify later. There are currently no interactive components.



https://reactjs.org/tutorial/tutorial.html

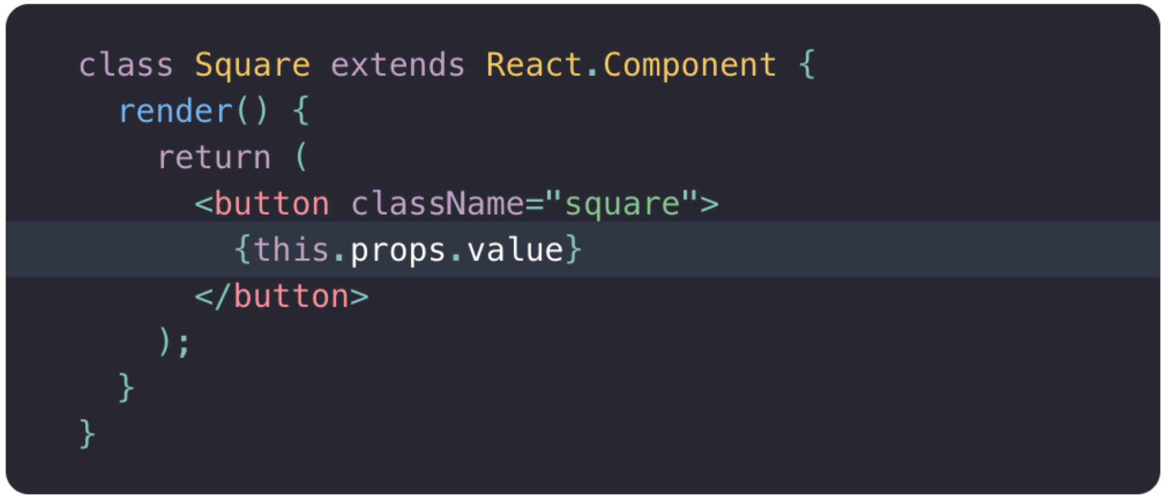
Passing Data Through Props

To get our feet wet, let’s try passing some data from our Board component to our Square component.



```
class Board extends React.Component {
  renderSquare(i) {
    return <Square value={i} />;
  }
```

In Board’s **renderSquare** method, change the code to pass a prop called value to the Square:



```
class Square extends React.Component {
  render() {
    return (
      <button className="square">
        {this.props.value}
      </button>
    );
  }
}
```

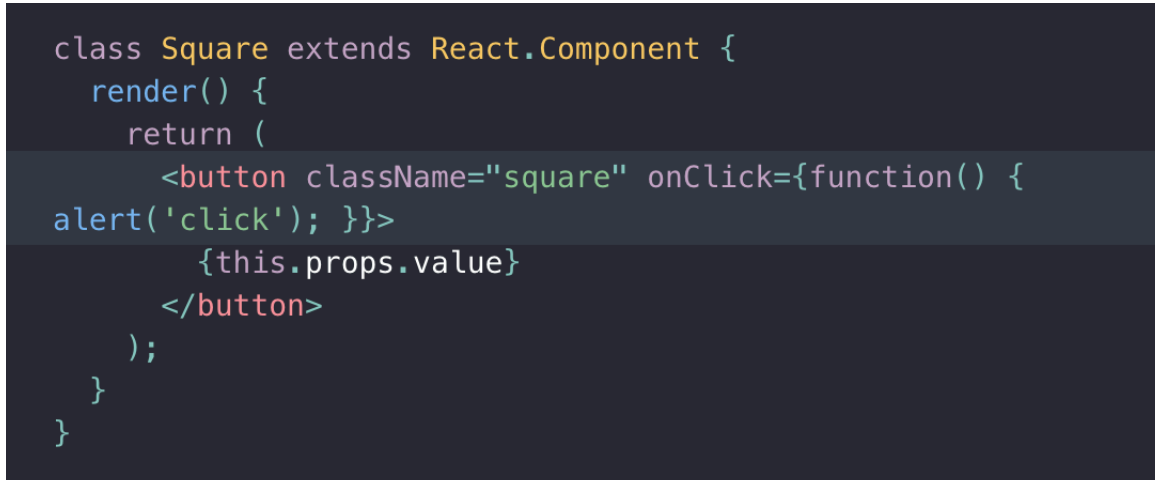
Change Square’s render method to show that value by replacing {/* TODO */} with {this.props.value}:

Making an Interactive Components

Let’s fill the Square component with an “X” when we click it.

First, change the button tag that is returned from the Square component’s **render()** function to this.

If you click on a Square now, you should see an alert in your browser.



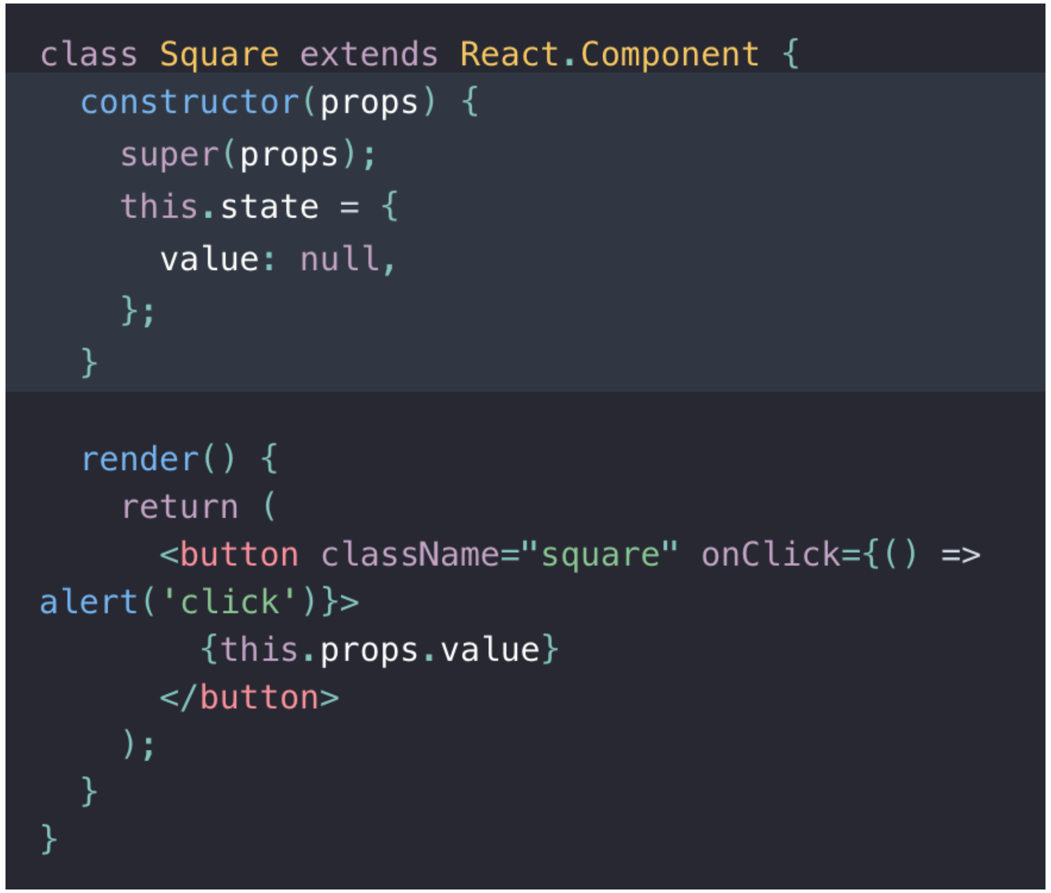
```
class Square extends React.Component {
  render() {
    return (
      <button className="square" onClick={function() {
alert('click'); }}>
        {this.props.value}
      </button>
    );
  }
}
```

Making an Interactive Components

As a next step, we want the Square component to "remember" that it got clicked, and fill it with an "X" mark. To "remember" things, components use **state**.

React components can have state by setting **this.state** in their constructors. **this.state** should be considered as private to a React component that it's defined in. Let's store the current value of the Square in **this.state**, and change it when the Square is clicked.

In JavaScript classes, you need to always call super when defining the constructor of a subclass. All React component classes that have a constructor should start it with a super(props) call.



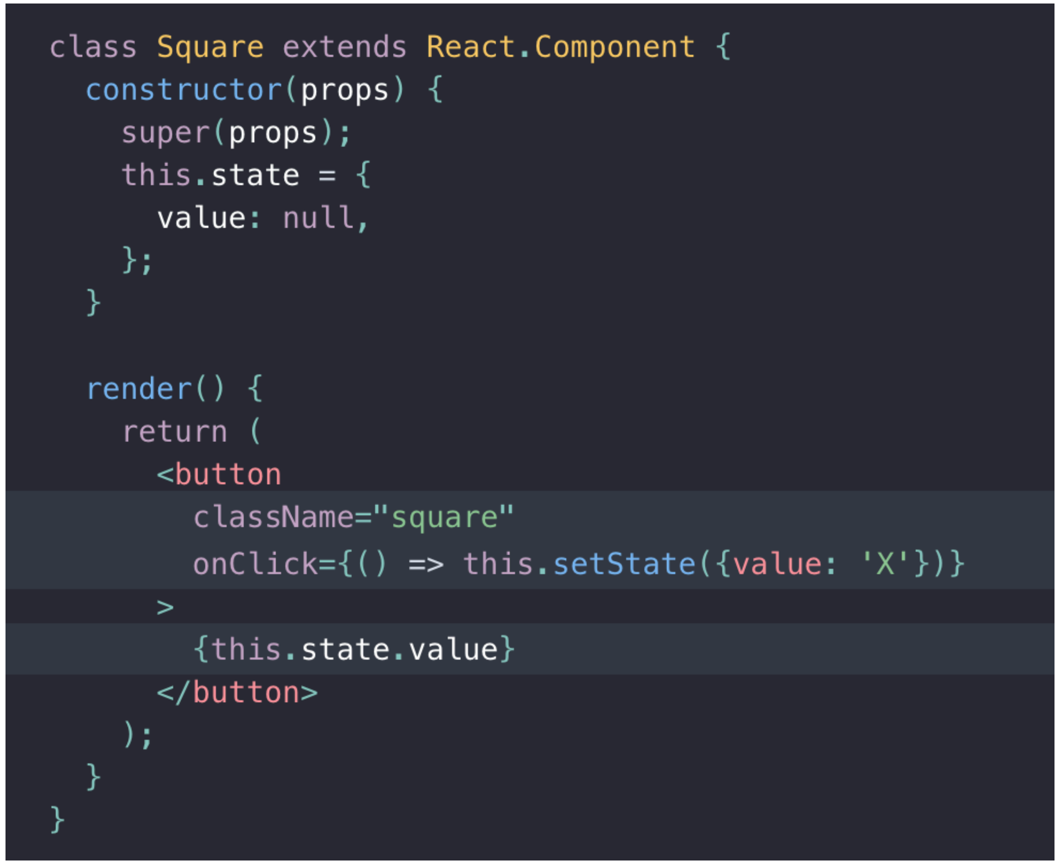
```
class Square extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: null,
    };
  }

  render() {
    return (
      <button className="square" onClick={() =>
alert('click')}>
        {this.props.value}
      </button>
    );
  }
}
```

Making an Interactive Components

Now we’ll change the Square’s render method to display the current state’s value when clicked:

- Replace this.props.value with this.state.value inside the <button> tag.
- Replace the onClick={...} event handler with onClick={() => this.setState({value: 'X'})}.
- Put the className and onClick props on separate lines for better readability.

By calling **this.setState** from an onClick handler in the Square’s render method, we tell React to re-render that Square whenever its <button> is clicked. After the update, the Square’s **this.state.value** will be 'X', so we’ll see the X on the game board. If you click on any Square, an X should show up.



```
class Square extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: null,
    };
  }

  render() {
    return (
      <button
        className="square"
        onClick={() => this.setState({value: 'X'})}
      >
        {this.state.value}
      </button>
    );
  }
}
```

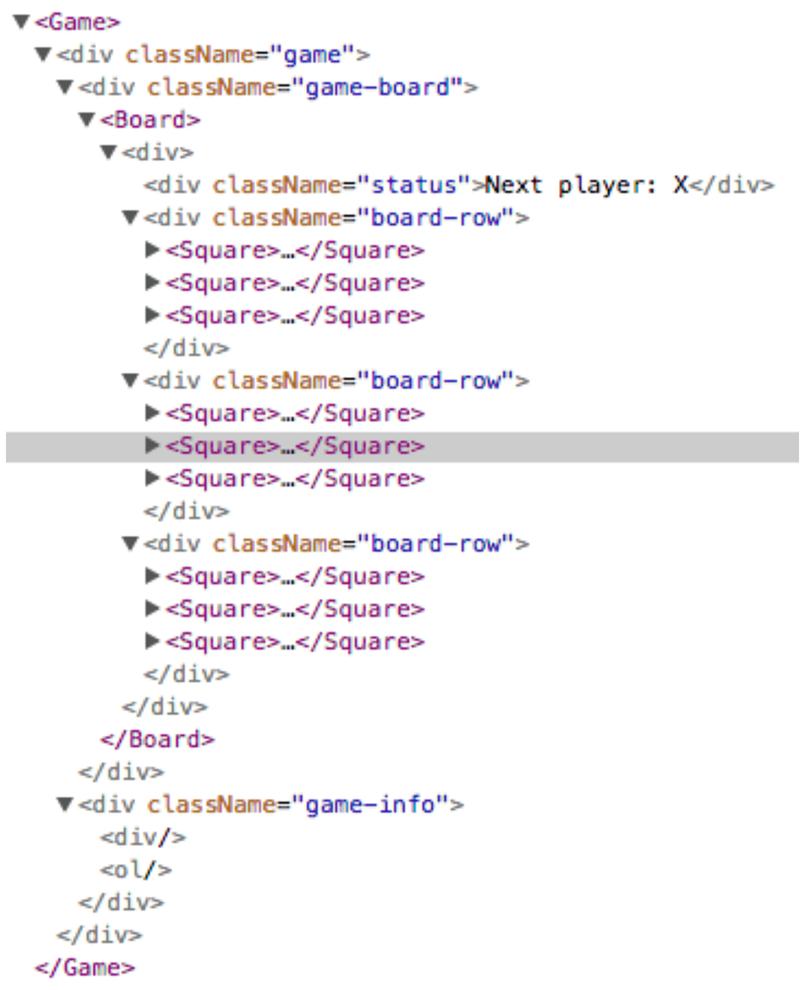
When you call setState in a component, React automatically updates the child components inside of it too.

https://reactjs.org/tutorial/tutorial.html

Debug with Develop Tools

The React Devtools extension lets you inspect a React component tree with your browser's developer tools. The React DevTools let you check the props and the state of your React components.

After installing React DevTools, you can right-click on any element on the page, click "Inspect" to open the developer tools, and the React tabs ("⚛️ Components" and "⚛️ Profiler") will appear as the last tabs to the right. Use "⚛️ Components" to inspect the component tree.



```
▼<Game>
  ▼<div className="game">
    ▼<div className="game-board">
      ▼<Board>
        ▼<div>
            <div className="status">Next player: X</div>
          ▼<div className="board-row">
            ▶<Square>…</Square>
            ▶<Square>…</Square>
            ▶<Square>…</Square>
            </div>
          ▼<div className="board-row">
            ▶<Square>…</Square>
            ▶<Square>…</Square>
            ▶<Square>…</Square>
            </div>
          ▼<div className="board-row">
            ▶<Square>…</Square>
            ▶<Square>…</Square>
            ▶<Square>…</Square>
            </div>
          </div>
        </Board>
      </div>
    ▼<div className="game-info">
        <div/>
        <ol/>
      </div>
    </div>
</Game>
```

Completing the Game

We now have the basic building blocks for our tic-tac-toe game. To have a complete game, we now need to alternate placing “X”s and “O”s on the board, and we need a way to determine a winner.

Lifting State Up

Currently, each Square component maintains the game’s state. To check for a winner, we’ll maintain the value of each of the 9 squares in one location.

The best approach is to store the game’s state in the parent Board component instead of in each Square. The Board component can tell each Square what to display by passing a prop

```
class Board extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      squares: Array(9).fill(null),
    };
  }

  renderSquare(i) {
    return <Square value={i} />;
  }
```

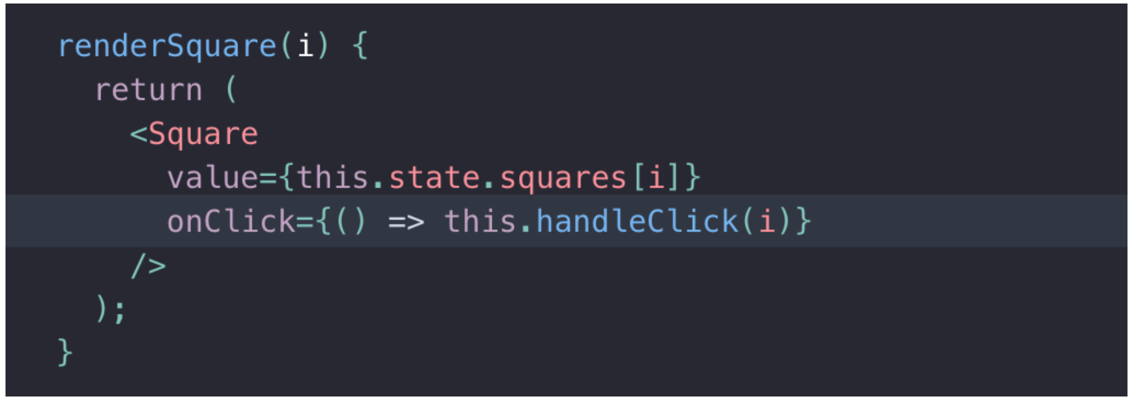
```
renderSquare(i) {
  return <Square value={this.state.squares[i]} />;
}
```

Completing the Game

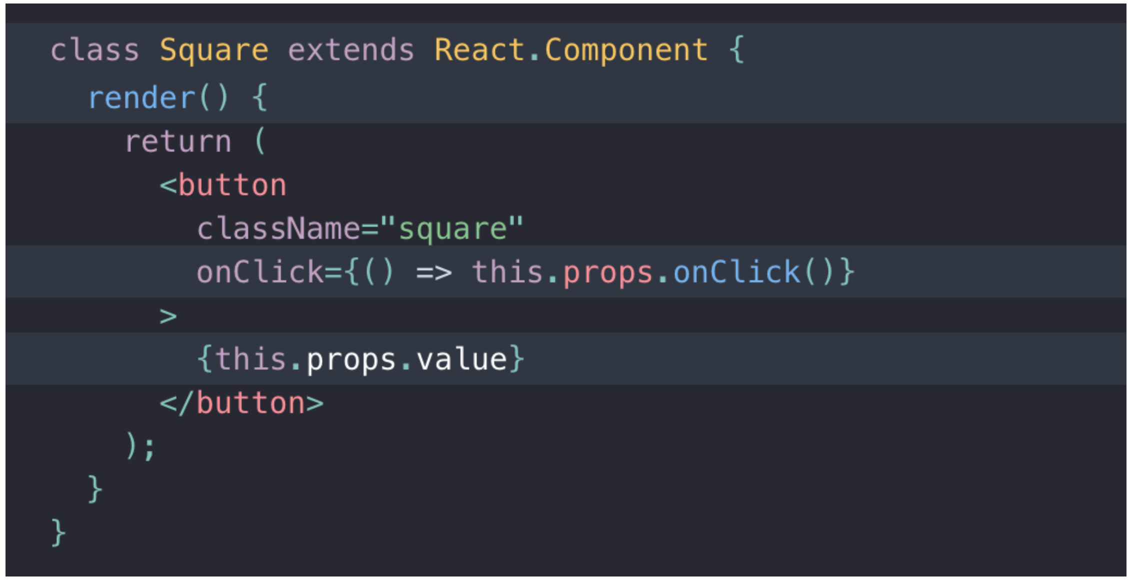
Each Square will now receive a value prop that will either be **‘X’, ‘O’, or null** for empty squares.

Next, we need to change what happens when a Square is clicked. The Board component now maintains which squares are filled. We need to create a way for the Square to update the Board’s state. Since state is considered to be private to a component that defines it, we cannot update the Board’s state directly from Square.

Instead, we’ll pass down a function from the Board to the Square, and we’ll have Square call that function when a square is clicked.
Now we’re passing down two props from Board to Square: value and onClick. The onClick prop is a function that Square can call when clicked.



```
renderSquare(i) {
  return (
    <Square
      value={this.state.squares[i]}
      onClick={() => this.handleClick(i)}
    />
  );
}
```



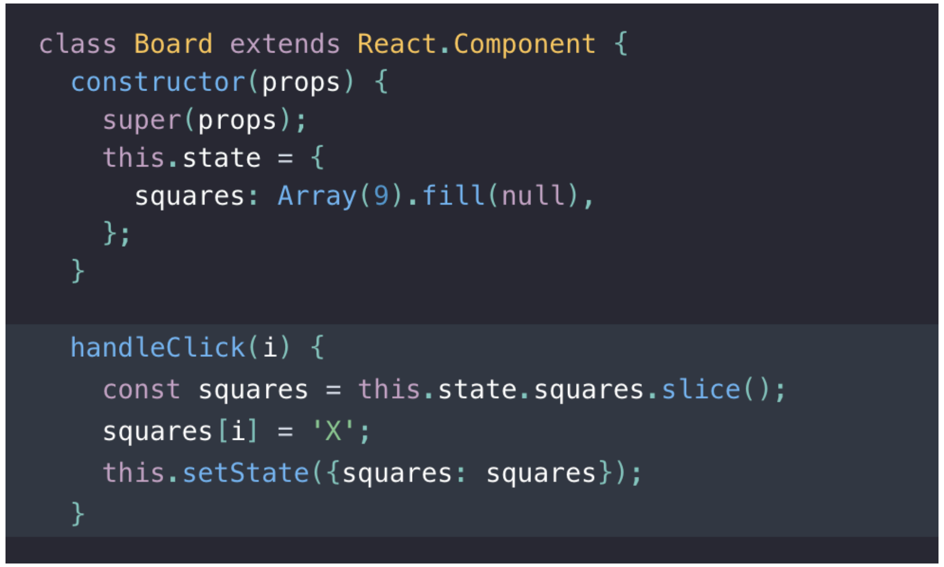
```
class Square extends React.Component {
  render() {
    return (
      <button
        className="square"
        onClick={() => this.props.onClick()}
      >
        {this.props.value}
      </button>
    );
  }
}
```

https://reactjs.org/tutorial/tutorial.html

Completing the Game

When a Square is clicked, the onClick function provided by the Board is called. Here's a review of how this is achieved:

1. The onClick prop on the built-in DOM <button> component tells React to set up a click event listener.
2. When the button is clicked, React will call the onClick event handler that is defined in Square's render() method.
3. This event handler calls this.props.onClick(). The Square's onClick prop was specified by the Board.
4. Since the Board passed onClick={() => this.handleClick(i)} to Square, the Square calls this.handleClick(i) when clicked.
5. We have not defined the handleClick() method yet, so our code crashes. If you click a square now, you should see a red error screen saying something like "this.handleClick is not a function".



```
class Board extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      squares: Array(9).fill(null),
    };
  }

  handleClick(i) {
    const squares = this.state.squares.slice();
    squares[i] = 'X';
    this.setState({squares: squares});
  }
```

Why Immutability Is Important

In the previous code example, we suggested that you use the **.slice()** method to create a copy of the **squares** array to modify instead of modifying the existing array. We'll now discuss immutability and why immutability is important to learn.

There are generally two approaches to changing data.

1. The first approach is to *mutate* the data by directly changing the data's values.
2. The second approach is to replace the data with a new copy which has the desired changes.

Data Change with Mutation

```
var player = {score: 1, name: 'Jeff'};
player.score = 2;
// Now player is {score: 2, name: 'Jeff'}
```

Data Change without Mutation

```
var player = {score: 1, name: 'Jeff'};

var newPlayer = Object.assign({}, player, {score: 2});
// Now player is unchanged, but newPlayer is {score: 2, name:
'Jeff'}
```

The end result is the same but by not mutating

Why Immutability Is Important

1. Complex Features Become Simple

Immutability makes complex features much easier to implement. Later in this tutorial, we will implement a “time travel” feature that allows us to review the tic-tac-toe game’s history and “jump back” to previous moves. This functionality isn’t specific to games — an ability to undo and redo certain actions is a common requirement in applications. Avoiding direct data mutation lets us keep previous versions of the game’s history intact, and reuse them later

Why Immutability Is Important

2. Detecting Changes

Detecting changes in mutable objects is difficult because they are modified directly. This detection requires the mutable object to be compared to previous copies of itself and the entire object tree to be traversed.

Detecting changes in immutable objects is considerably easier. If the immutable object that is being referenced is different than the previous one, then the object has changed.

Why Immutability Is Important

3. Determining When to Re-Render in React

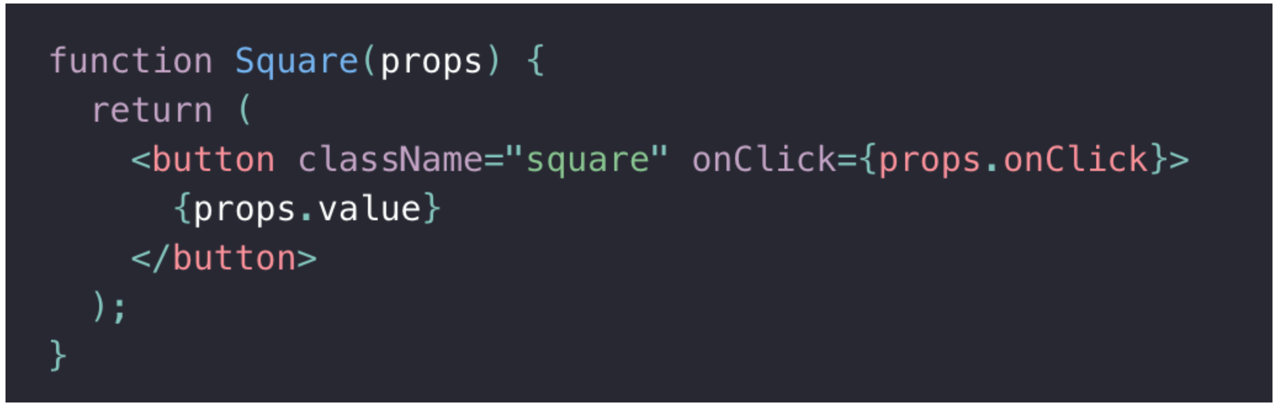
The main benefit of immutability is that it helps you build *pure components* in React. Immutable data can easily determine if changes have been made which helps to determine when a component requires re-rendering.

Function Components

We’ll now change the Square to be
a **function component**.

In React, **function components** are a simpler way to write components that only contain a render method and don’t have their own state.

Instead of defining a class which extends React.Component, we can write a function that takes props as input and returns what should be rendered. Function components are less tedious to write than classes, and many components can be expressed this way.



```
function Square(props) {
  return (
    <button className="square" onClick={props.onClick}>
      {props.value}
    </button>
  );
}
```

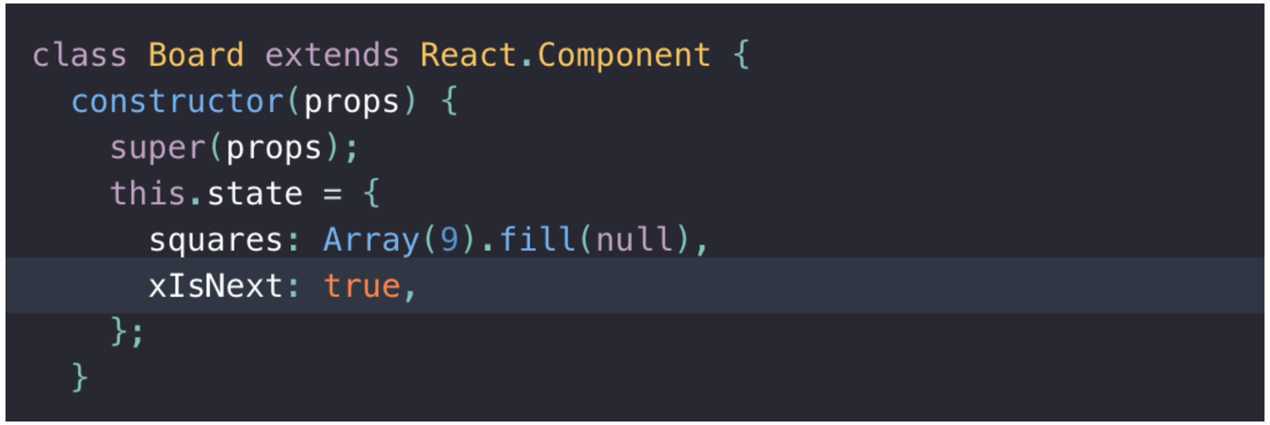
We have changed this.props to props both times it appears.

Taking Turns

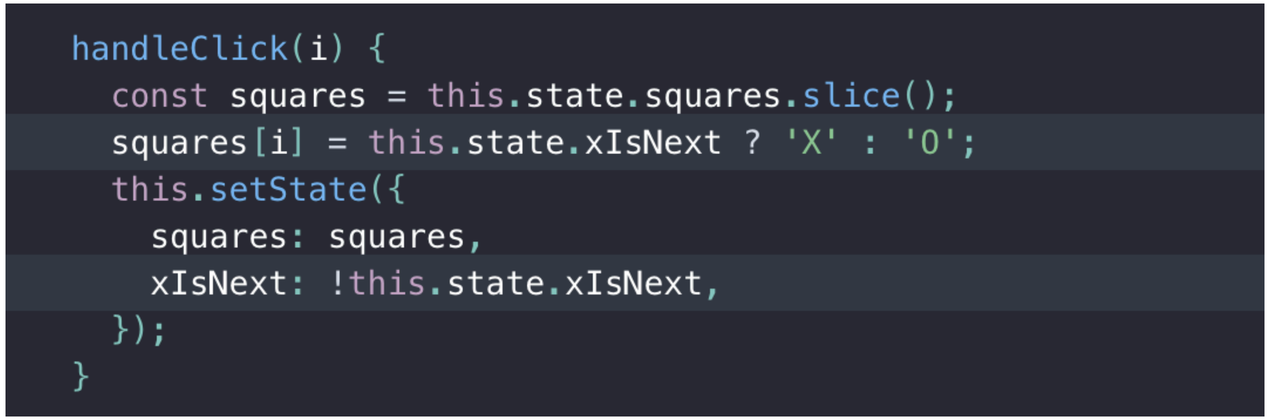
We now need to fix an obvious defect in our tic-tac-toe game: the “O”s cannot be marked on the board.

We’ll set the first move to be “X” by default. We can set this default by modifying the initial state in our Board constructor.

Each time a player moves, xIsNext (a boolean) will be flipped to determine which player goes next and the game’s state will be saved. We’ll update the Board’s handleClick function to flip the value of xIsNext.



```
class Board extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      squares: Array(9).fill(null),
      xIsNext: true,
    };
  }
```



```
  handleClick(i) {
    const squares = this.state.squares.slice();
    squares[i] = this.state.xIsNext ? 'X' : 'O';
    this.setState({
      squares: squares,
      xIsNext: !this.state.xIsNext,
    });
  }
```

https://reactjs.org/tutorial/tutorial.html

Taking Turns

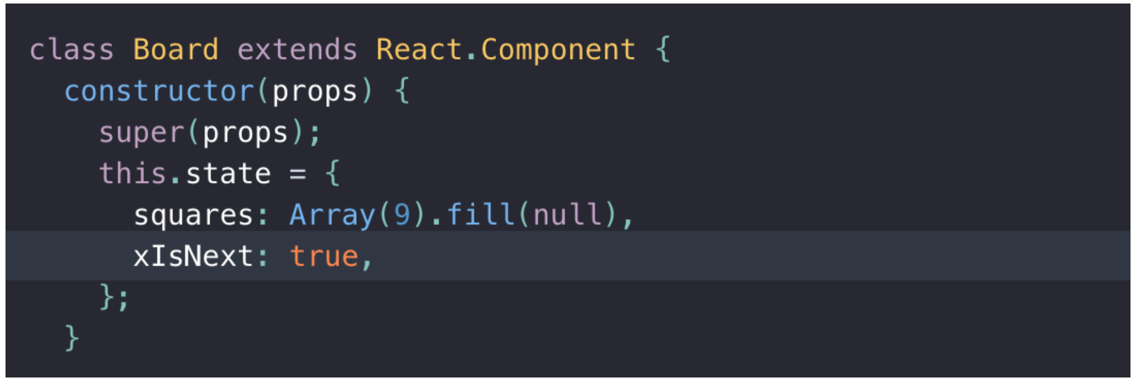
We now need to fix an obvious defect in our tic-tac-toe game: the “O”s cannot be marked on the board.

We’ll set the first move to be “X” by default. We can set this default by modifying the initial state in our Board constructor.

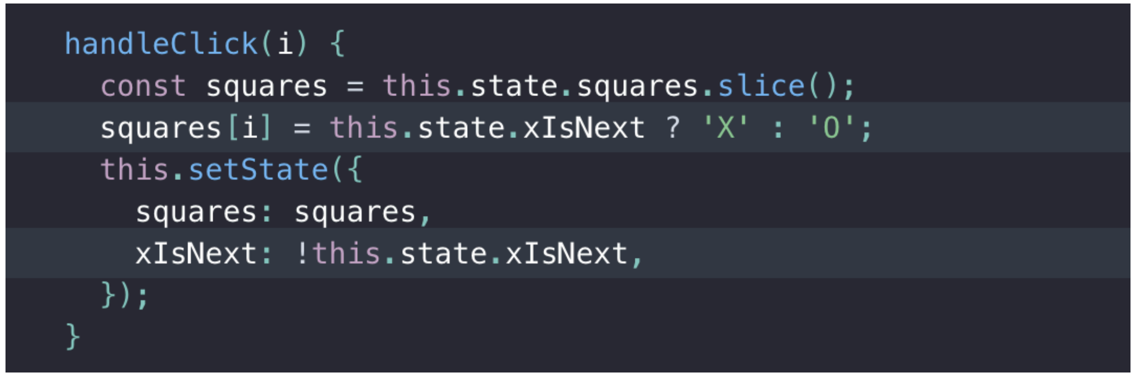
Each time a player moves, xIsNext (a boolean) will be flipped to determine which player goes next and the game’s state will be saved. We’ll update the Board’s handleClick function to flip the value of xIsNext.

With this change, “X”s and “O”s can take turns. Try it!

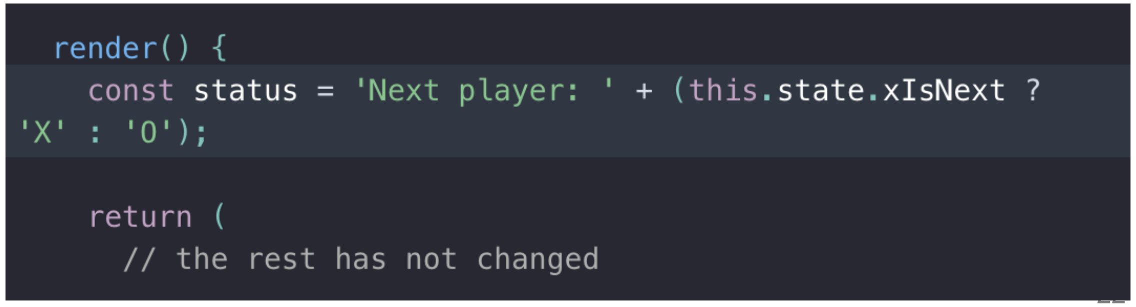
https://reactjs.org/tutorial/tutorial.html



```
class Board extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      squares: Array(9).fill(null),
      xIsNext: true,
    };
  }
```



```
  handleClick(i) {
    const squares = this.state.squares.slice();
    squares[i] = this.state.xIsNext ? 'X' : 'O';
    this.setState({
      squares: squares,
      xIsNext: !this.state.xIsNext,
    });
  }
```



```
  render() {
    const status = 'Next player: ' + (this.state.xIsNext ?
'X' : 'O');

    return (
      // the rest has not changed
```

Declaring a Winner

Now that we show which player’s turn is next, we should also show when the game is won and there are no more turns to make.

Given an array of 9 squares, this function will check for a winner and return 'X', 'O', or null as appropriate.

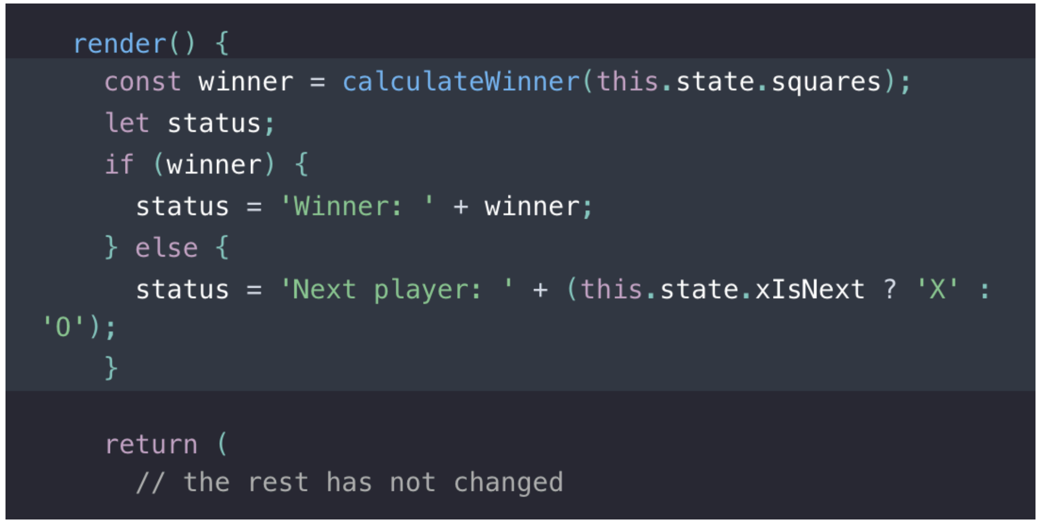
```
function calculateWinner(squares) {
  const lines = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8],
    [0, 3, 6],
    [1, 4, 7],
    [2, 5, 8],
    [0, 4, 8],
    [2, 4, 6],
  ];
  for (let i = 0; i < lines.length; i++) {
    const [a, b, c] = lines[i];
    if (squares[a] && squares[a] === squares[b] &&
squares[a] === squares[c]) {
      return squares[a];
    }
  }
  return null;
}
```

Declaring a Winner

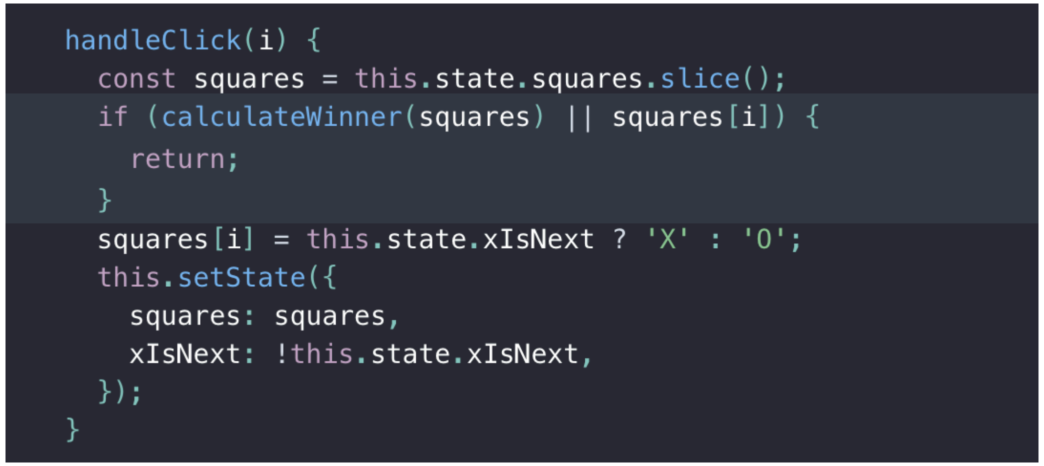
We will call **calculateWinner**(squares) in the Board’s render function to check if a player has won.

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.

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.



```
  render() {
    const winner = calculateWinner(this.state.squares);
    let status;
    if (winner) {
      status = 'Winner: ' + winner;
    } else {
      status = 'Next player: ' + (this.state.xIsNext ? 'X' :
'O');
    }

    return (
      // the rest has not changed
```



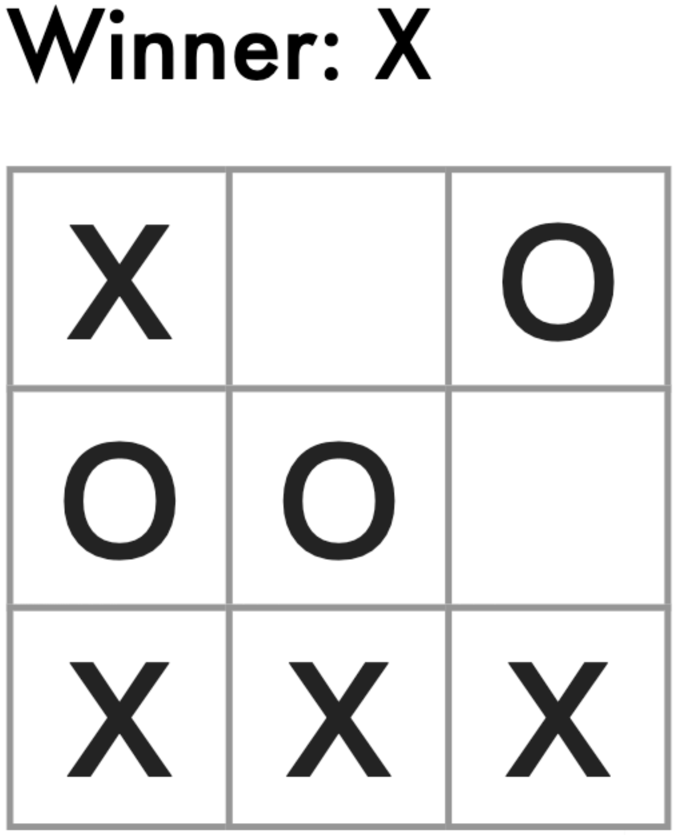
```
  handleClick(i) {
    const squares = this.state.squares.slice();
    if (calculateWinner(squares) || squares[i]) {
      return;
    }
    squares[i] = this.state.xIsNext ? 'X' : 'O';
    this.setState({
      squares: squares,
      xIsNext: !this.state.xIsNext,
    });
  }
```

https://reactjs.org/tutorial/tutorial.html

Storing a History of Moves

You now have a working tic-tac-toe game. And you've just learned the basics of React too.
So *you're* probably the real winner here.

Next, let's make it possible to "go back in time" to the previous moves in the game.



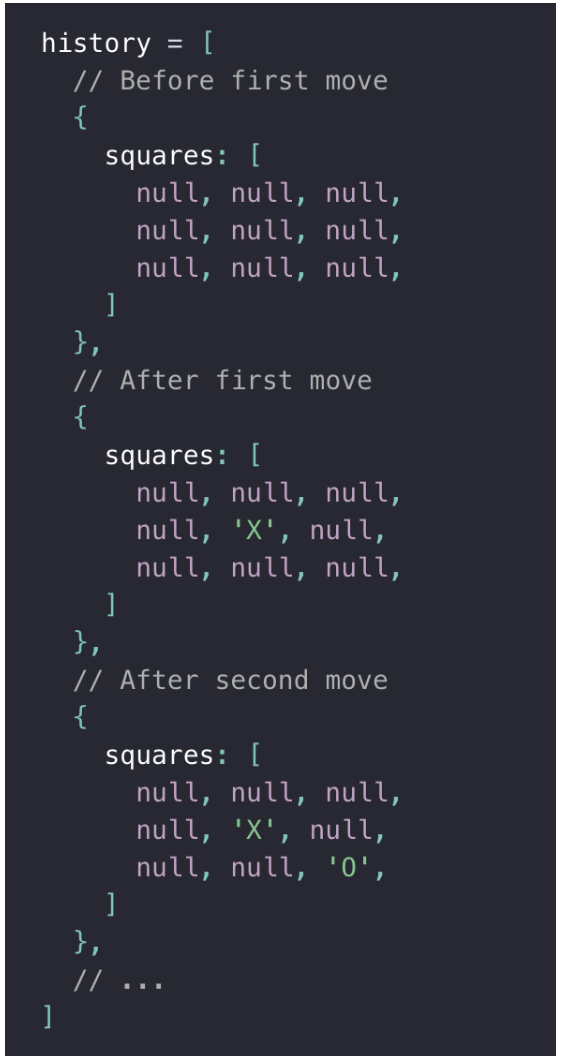
https://reactjs.org/tutorial/tutorial.html

Storing a History of Moves

If we mutated the squares array, implementing time travel would be very difficult.

However, we used slice() to create a new copy of the squares array after every move, and treated it as immutable. This will allow us to store every past version of the squares array, and navigate between the turns that have already happened.

We'll store the past squares arrays in another array called history. The history array represents all board states, from the first to the last move, and has a shape like this:



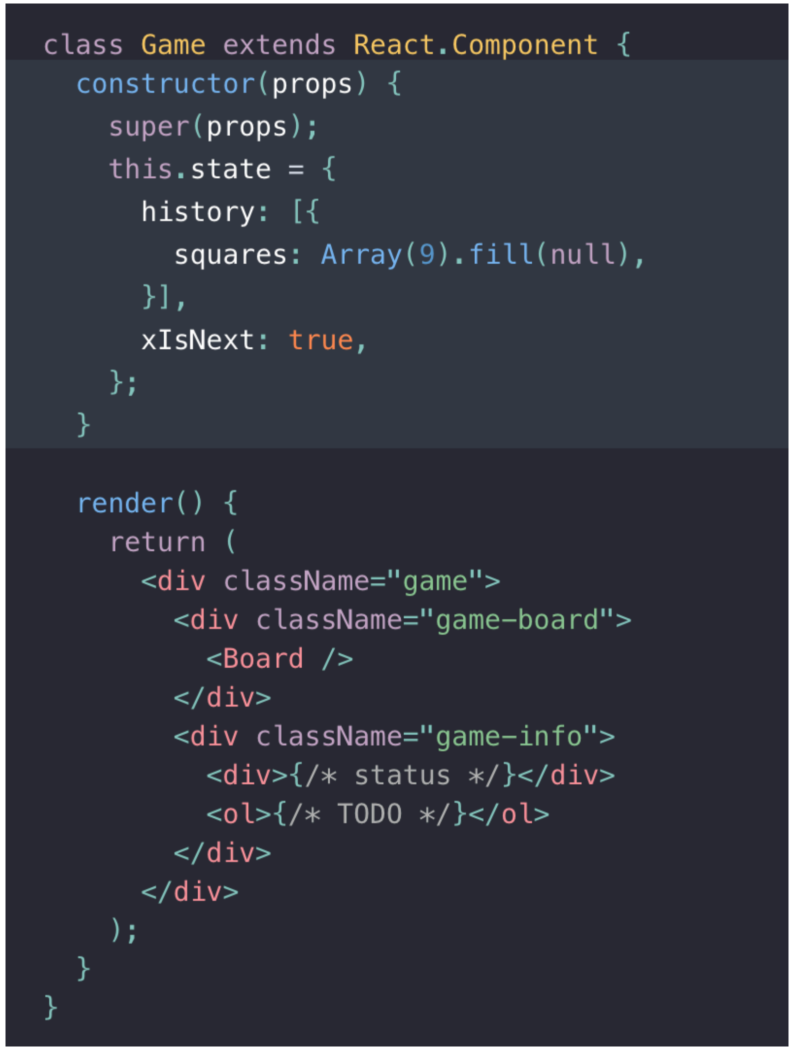
```
history = [
  // Before first move
  {
    squares: [
      null, null, null,
      null, null, null,
      null, null, null,
    ]
  },
  // After first move
  {
    squares: [
      null, null, null,
      null, 'X', null,
      null, null, null,
    ]
  },
  // After second move
  {
    squares: [
      null, null, null,
      null, 'X', null,
      null, null, 'O',
    ]
  },
  // ...
]
```

Storing a History of Moves

We'll want the top-level Game component to display a list of past moves. It will need access to the history to do that, so we will place the history state in the top-level Game component.

Placing the history state into the Game component lets us remove the squares state from its child Board component. We are now lifting it up from the Board into the top-level Game component. This gives the Game component full control over the Board's data, and lets it instruct the Board to render previous turns from the history.

First, we'll set up the initial state for the Game component within its constructor



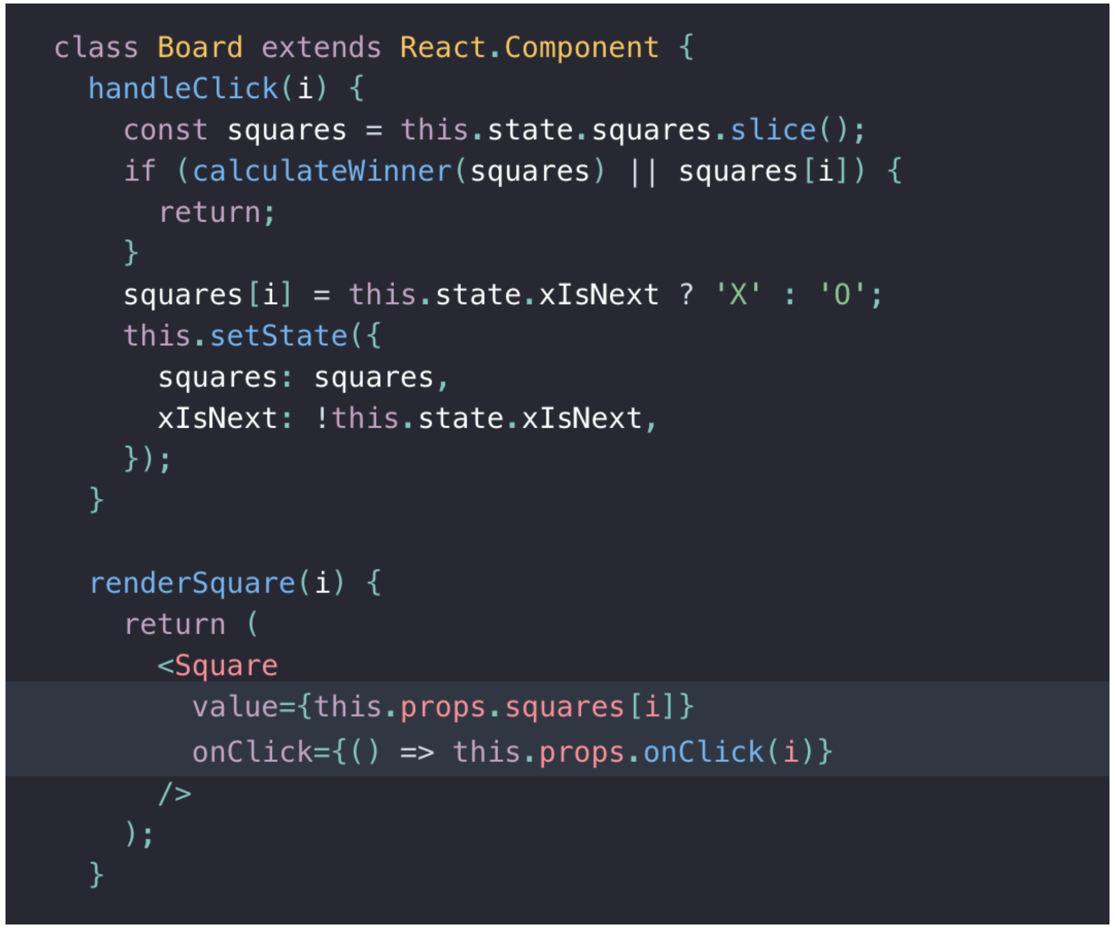
```
class Game extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      history: [{
        squares: Array(9).fill(null),
      }],
      xIsNext: true,
    };
  }

  render() {
    return (
      <div className="game">
        <div className="game-board">
          <Board />
        </div>
        <div className="game-info">
          <div>{/* status */}</div>
          <ol>{/* TODO */}</ol>
        </div>
      </div>
    );
  }
}
```

https://reactjs.org/tutorial/tutorial.html

Storing a History of Moves

Next, we'll have the Board component receive **squares** and **onClick** props from the Game component. Since we now have a single click handler in Board for many Squares, we'll need to pass the location of each Square into the **onClick** handler to indicate which Square was clicked. Here are the required steps to transform the Board component:

1. Delete the constructor in Board.
2. Replace **this.state.squares[i]** with **this.props.squares[i]** in Board's renderSquare.
3. Replace **this.handleClick(i)** with **this.props.onClick(i)** in Board's render Square.

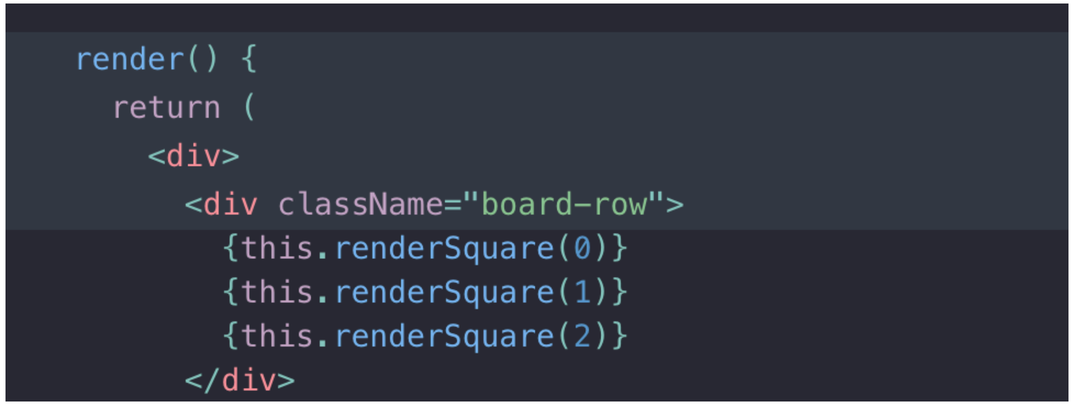


```
class Board extends React.Component {
  handleClick(i) {
    const squares = this.state.squares.slice();
    if (calculateWinner(squares) || squares[i]) {
      return;
    }
    squares[i] = this.state.xIsNext ? 'X' : 'O';
    this.setState({
      squares: squares,
      xIsNext: !this.state.xIsNext,
    });
  }

  renderSquare(i) {
    return (
      <Square
        value={this.props.squares[i]}
        onClick={() => this.props.onClick(i)}
      />
    );
  }
```

Storing a History of Moves

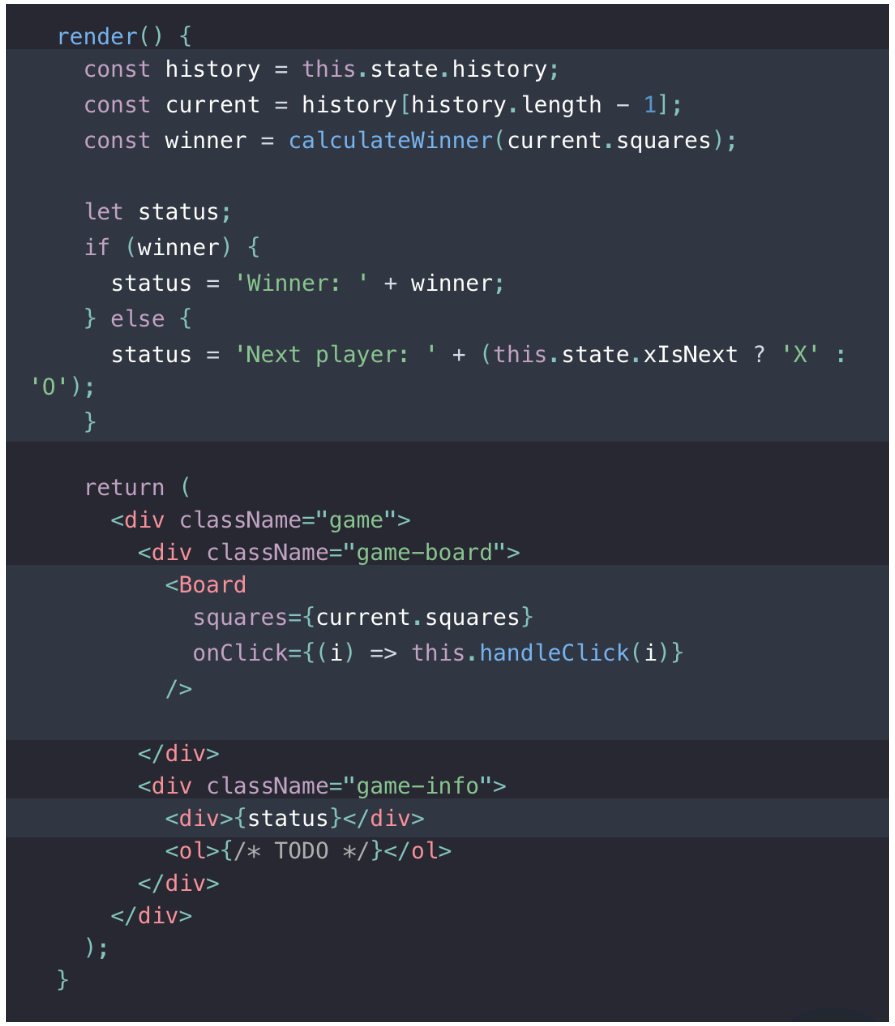
We'll update the Game component's render function to use the most recent history entry to determine and display the game's status.

Since the Game component is now rendering the game's status, we can remove the corresponding code from the Board's render method.



```
  render() {
    return (
      <div>
        <div className="board-row">
          {this.renderSquare(0)}
          {this.renderSquare(1)}
          {this.renderSquare(2)}
        </div>
```

Board



```
  render() {
    const history = this.state.history;
    const current = history[history.length - 1];
    const winner = calculateWinner(current.squares);

    let status;
    if (winner) {
      status = 'Winner: ' + winner;
    } else {
      status = 'Next player: ' + (this.state.xIsNext ? 'X' :
'O');
    }

    return (
      <div className="game">
        <div className="game-board">
          <Board
            squares={current.squares}
            onClick={(i) => this.handleClick(i)}
          />

        </div>
        <div className="game-info">
          <div>{status}</div>
          <ol>{/* TODO */}</ol>
        </div>
      </div>
    );
  }
```

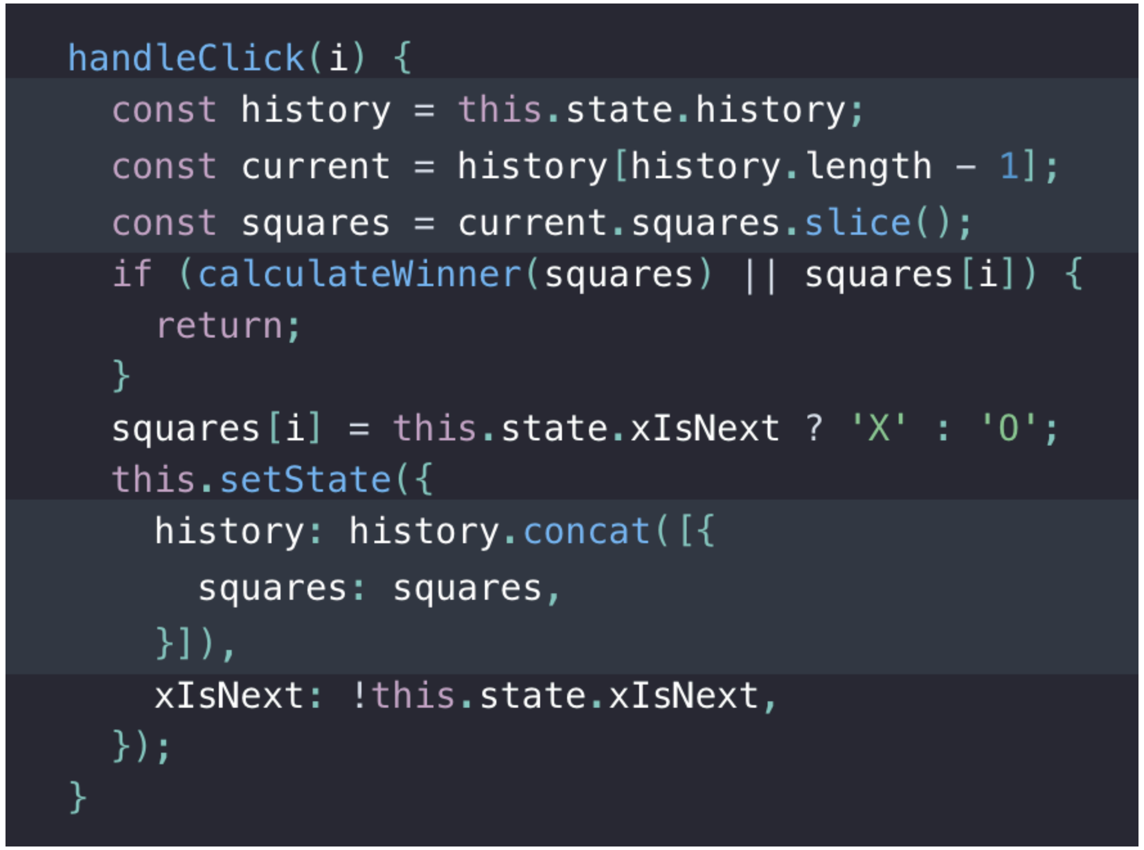
Game

https://reactjs.org/tutorial/tutorial.html

Storing a History of Moves

Finally, we need to move the handleClick method from the Board component to the Game component. We also need to modify handleClick because the Game component’s state is structured differently. Within the Game’s handleClick method, we concatenate new history entries onto history.

At this point, the Board component only needs the renderSquare and render methods. The game’s state and the handleClick method should be in the Game component.



```
handleClick(i) {
  const history = this.state.history;
  const current = history[history.length - 1];
  const squares = current.squares.slice();
  if (calculateWinner(squares) || squares[i]) {
    return;
  }
  squares[i] = this.state.xIsNext ? 'X' : 'O';
  this.setState({
    history: history.concat([{
      squares: squares,
    }]),
    xIsNext: !this.state.xIsNext,
  });
}
```

Unlike the array push() method you might be more familiar with, the concat() method doesn’t mutate the original array, so we prefer it.

https://reactjs.org/tutorial/tutorial.html

Showing the Past Moves

Since we are recording the tic-tac-toe game's history, we can now display it to the player as a list of past moves.

We learned earlier that React elements are first-class JavaScript objects; we can pass them around in our applications.

In JavaScript, arrays have a map() method that is commonly used for mapping data to other data.

```
const numbers = [1, 2, 3];
const doubled = numbers.map(x => x * 2); // [2, 4, 6]
```

Using the map method, we can map our history of moves to React elements representing buttons on the screen, and display a list of buttons to "jump" to past moves.

```
  render() {
    const history = this.state.history;
    const current = history[history.length - 1];
    const winner = calculateWinner(current.squares);

    const moves = history.map((step, move) => {
      const desc = move ?
        'Go to move #' + move :
        'Go to game start';
      return (
        <li>
          <button onClick={() => this.jumpTo(move)}>{desc}
</button>
        </li>
      );
    });
```

```
      <div className="game-info">
        <div>{status}</div>
        <ol>{moves}</ol>
      </div>
```

https://reactjs.org/tutorial/tutorial.html

Picking a Key

When we render a list, React stores some information about each rendered list item. When we update a list, React needs to determine what has changed. We could have added, removed, re-arranged, or updated the list's items.

In addition to the updated counts, a human reading this would probably say that we swapped Alexa and Ben's ordering and inserted Claudia between Alexa and Ben. However, React is a computer program and does not know what we intended. Because React cannot know our intentions, we need to specify a *key* property for each list item to differentiate each list item from its siblings.

Imagine transitioning from

```
<li>Alexa: 7 tasks left</li>
<li>Ben: 5 tasks left</li>
```

to

```
<li>Ben: 9 tasks left</li>
<li>Claudia: 8 tasks left</li>
<li>Alexa: 5 tasks left</li>
```

```
<li key={user.id}>{user.name}: {user.taskCount} tasks
left</li>
```

It's strongly recommended that you assign proper keys whenever you build dynamic lists.

https://reactjs.org/tutorial/tutorial.html

Implementing Time Travel

In the tic-tac-toe game's history, each past move has a unique ID associated with it: it's the sequential number of the move. The moves are never re-ordered, deleted, or inserted in the middle, so it's safe to use the **move** index as a key.

1. First, add stepNumber: 0 to the initial state in Game's constructor.
2. Next, we'll define the jumpTo method in Game to update that stepNumber. We also set xIsNext to true if the number that we're changing stepNumber to is even

```
  jumpTo(step) {
    this.setState({
      stepNumber: step,
      xIsNext: (step % 2) === 0,
    });
  }
```

```
    const moves = history.map((step, move) => {
      const desc = move ?
        'Go to move #' + move :
        'Go to game start';
      return (
        <li key={move}>
          <button onClick={() => this.jumpTo(move)}>{desc}
</button>
        </li>
      );
    });
```

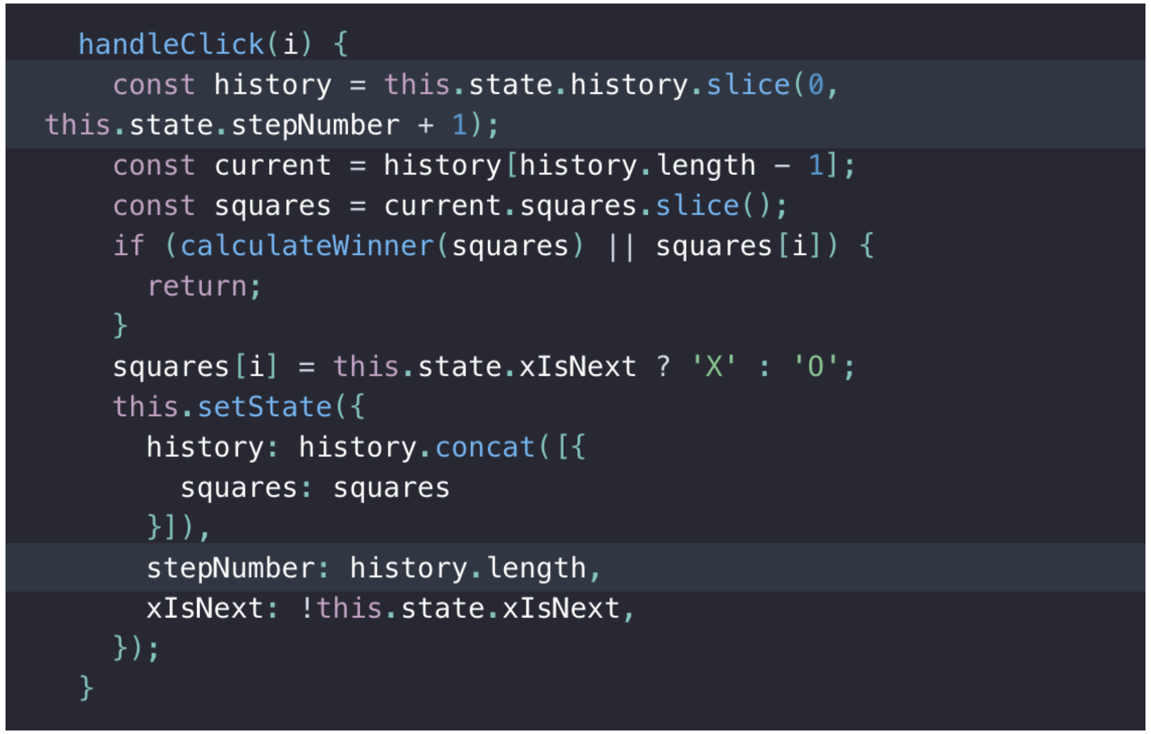
```
class Game extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      history: [{
        squares: Array(9).fill(null),
      }],
      stepNumber: 0,
      xIsNext: true,
    };
  }
```

Implementing Time Travel

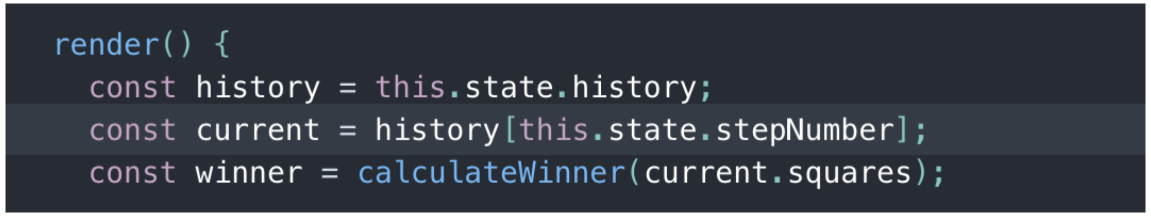
The **stepNumber** state we've added reflects the move displayed to the user now. After we make a new move, we need to update stepNumber by adding stepNumber: history.length as part of the this.setState argument.

We will also replace reading **this.state.history** with **this.state.history.slice(0, this.state.stepNumber + 1).**

Finally, we will modify the Game component's **render** method from always rendering the last move to rendering the currently selected move according to **stepNumber**.



```
  handleClick(i) {
    const history = this.state.history.slice(0,
this.state.stepNumber + 1);
    const current = history[history.length - 1];
    const squares = current.squares.slice();
    if (calculateWinner(squares) || squares[i]) {
      return;
    }
    squares[i] = this.state.xIsNext ? 'X' : 'O';
    this.setState({
      history: history.concat([{
        squares: squares
      }]),
      stepNumber: history.length,
      xIsNext: !this.state.xIsNext,
    });
  }
```



```
  render() {
    const history = this.state.history;
    const current = history[this.state.stepNumber];
    const winner = calculateWinner(current.squares);
```

Wrapping Up

Congratulations! You’ve created a tic-tac-toe game that:

- Lets you play tic-tac-toe
- Indicates when a player has won the game
- Stores a game’s history as a game progresses
- Allows players to review a game’s history and see previous versions of a game’s board.

Homework

Practice your new React skills:

- ➢ Display the location for each move in the format (col, row) in the move history list.
- ➢ Bold the currently selected item in the move list.
- ➢ Rewrite Board to use two loops to make the squares instead of hardcoding them.
- ➢ Add a toggle button that lets you sort the moves in either ascending or descending order.
- ➢ When someone wins, highlight the three squares that caused the win.
- ➢ When no one wins, display a message about the result being a draw.

https://reactjs.org/tutorial/tutorial.html

Web Framework Tutorial 02

Danqing Shi
Tongji University