TUTORIAL

Edit on GitHub

Tutorial: Intro To React

Before We Start

What We're Building

Today, we're going to build an interactive tic-tac-toe game.

If you like, you can check out the final result here: Final Result. Don't worry if the code doesn't make sense to you yet, or if it uses an unfamiliar syntax. We will be learning how to build this game step by step throughout this tutorial.

Try playing the game. You can also click on a link in the move list to go "back in time" and see what the board looked like just after that move was made.

Once you get a little familiar with the game, feel free to close that tab, as we'll start from a simpler template in the next sections.

Prerequisites

We'll assume some familiarity with HTML and JavaScript but you should be able to follow along even if you haven't used them before.

If you need a refresher on JavaScript, we recommend reading this guide. Note that we're also using some features from ES6, a recent version of JavaScript. In this tutorial, we're using arrow functions, classes, `let`, and `const` statements. You can use Babel REPL to check what ES6 code compiles to.

How to Follow Along

There are two ways to complete this tutorial: you could either write the code right in the browser, or you could set up a local development environment on your machine. You can choose either option depending on what you feel comfortable with.

If You Prefer to Write Code in the Browser

This is the quickest way to get started!

First, open this starter code in a new tab. It should display an empty tic-tac-toe field. We will be editing that code during this tutorial.

You can now skip the next section about setting up a local development environment and head straight to the overview.

If You Prefer to Write Code in Your Editor

Alternatively, you can set up a project on your computer.

Note: **this is completely optional and not required for this tutorial!**

This is more work, but lets you work from the comfort of your editor.

If you want to do it, here are the steps to follow:

1. Make sure you have a recent version of Node.js installed.
2. Follow the installation instructions to create a new project.
3. Delete all files in the `src/` folder of the new project.
4. Add a file named `index.css` in the `src/` folder with this CSS code.
5. Add a file named `index.js` in the `src/` folder with this JS code, and then add three lines to the top of it:

Code

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
```

Now if you run `npm start` in the project folder and open `http://localhost:3000` in the browser, you should see an empty tic-tac-toe field.

We recommend following these instructions to configure syntax highlighting for your editor.

Help, I'm Stuck!

If you get stuck, check out the community support resources. In particular, Reactiflux chat is a great way to get quick help. If you don't get a good answer anywhere, please file an issue,

and we'll help you out.

With this out of the way, let's get started!

Overview

What is React?

React is a declarative, efficient, and flexible JavaScript library for building user interfaces.

React has a few different kinds of components, but we'll start with `React.Component` subclasses:

Code

```
class ShoppingList extends React.Component {
  render() {
    return (
      <div className="shopping-list">
        <h1>Shopping List for {this.props.name}</h1>
        <ul>
          <li>Instagram</li>
          <li>WhatsApp</li>
          <li>Oculus</li>
        </ul>
      </div>
    );
  }
}

// Example usage: <ShoppingList name="Mark" />
```

We'll get to the funny XML-like tags in a second. Your components tell React what you want to render – then React will efficiently update and render just the right components when your data changes.

Here, ShoppingList is a **React component class**, or **React component type**. A component takes in parameters, called `props`, and returns a hierarchy of views to display via the `render` method.

The `render` method returns a *description* of what you want to render, and then React takes that description and renders it to the screen. In particular, `render` returns a **React element**, which is a lightweight description of what to render. Most React developers use a special syntax called JSX which makes it easier to write these structures. The `<div />` syntax is transformed at build time to `React.createElement('div')`. The example above is equivalent to:

Code

```
return React.createElement('div', {className: 'shopping-list'},
  React.createElement('h1', /* ... h1 children ... */),
  React.createElement('ul', /* ... ul children ... */)
```

```
  );
```

See full expanded version.

If you're curious, `createElement()` is described in more detail in the API reference, but we won't be using it directly in this tutorial. Instead, we will keep using JSX.

You can put any JavaScript expression within braces inside JSX. Each React element is a real JavaScript object that you can store in a variable or pass around your program.

The `ShoppingList` component only renders built-in DOM components, but you can compose custom React components just as easily, by writing `<ShoppingList />`. Each component is encapsulated so it can operate independently, which allows you to build complex UIs out of simple components.

Getting Started

Start with this example: Starter Code.

It contains the shell of what we're building today. We've provided the styles so you only need to worry about the JavaScript.

In particular, we have three components:

- Square
- Board
- Game

The Square component renders a single `<button>`, the Board renders 9 squares, and the Game component renders a board with some placeholders that we'll fill in later. None of the components are interactive at this point.

Passing Data Through Props

Just to get our feet wet, let's try passing some data from the Board component to the Square component.

In Board's `renderSquare` method, change the code to pass a `value` prop to the Square:

Code

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

Then change Square's `render` method to show that value by replacing `{/* TODO */}` with `{this.props.value}`:

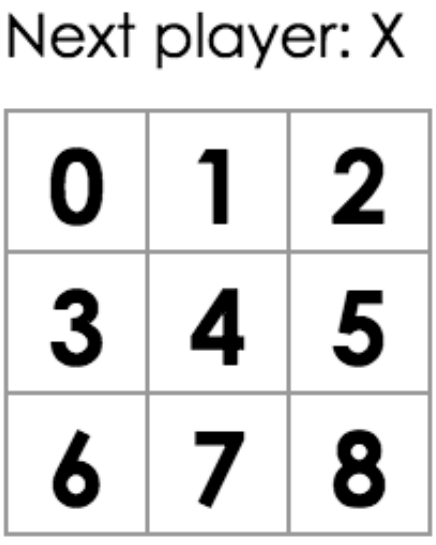
Code

```
class Square extends React.Component {
  render() {
```

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

Before:

After: You should see a number in each square in the rendered output.



View the current code.

An Interactive Component

Let's make the Square component fill in an "X" when you click it. Try changing the button tag returned in the `render()` function of the Square like this:

Code

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

If you click on a square now, you should get an alert in your browser.

This uses the new JavaScript arrow function syntax. Note that we're passing a function as the `onClick` prop. Doing `onClick={alert('click')}` would alert immediately instead of when the button is clicked.

React components can have state by setting `this.state` in the constructor, which should be considered private to the component. Let's store the current value of the square in state, and change it when the square is clicked.

First, add a constructor to the class to initialize the state:

Code

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

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

In JavaScript classes, you need to explicitly call `super();` when defining the constructor of a subclass.

Now change the Square `render` method to display the value from the current state, and to toggle it on click:

- Replace `this.props.value` with `this.state.value` inside the `<button>` tag.
- Replace the `() => alert()` event handler with `() => this.setState({value: 'X'})`.

Now the `<button>` tag looks like this:

Code

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

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

Whenever `this.setState` is called, an update to the component is scheduled, causing React to merge in the passed state update and rerender the component along with its descendants. When the component rerenders, `this.state.value` will be `'X'` so you'll see an X in the grid.

If you click on any square, an X should show up in it.

View the current code.

Developer Tools

The React Devtools extension for Chrome and Firefox lets you inspect a React component tree in your browser devtools.

It lets you inspect the props and state of any of the components in your tree.

After installing it, you can right-click any element on the page, click "Inspect" to open the developer tools, and the React tab will appear as the last tab to the right.

However, note there are a few extra steps to get it working with CodePen:

1. Log in or register and confirm your email (required to prevent spam).
2. Click the "Fork" button.
3. Click "Change View" and then choose "Debug mode".
4. In the new tab that opens, the devtools should now have a React tab.

Lifting State Up

We now have the basic building blocks for a tic-tac-toe game. But right now, the state is encapsulated in each Square component. To make a fully-working game, we now need to check if one player has won the game, and alternate placing X and O in the squares. To check if someone has won, we'll need to have the value of all 9 squares in one place, rather than split up across the Square components.

You might think that Board should just inquire what the current state of each Square is. Although it is technically possible to do this in React, it is discouraged because it tends to make code difficult to understand, more brittle, and harder to refactor.

Instead, the best solution here is to store this state in the Board component instead of in

each Square – and the Board component can tell each Square what to display, like how we made each square display its index earlier.

When you want to aggregate data from multiple children or to have two child components communicate with each other, move the state upwards so that it lives in the parent component. The parent can then pass the state back down to the children via props, so that the child components are always in sync with each other and with the parent.

Pulling state upwards like this is common when refactoring React components, so let's take this opportunity to try it out. Add a constructor to the Board and set its initial state to contain an array with 9 nulls, corresponding to the 9 squares:

Code

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

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

  render() {
    const status = 'Next player: X';

    return (
      <div>
        <div className="status">{status}</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>
    );
  }
}
```

We'll fill it in later so that a board looks something like

Code

```
[
  'O', null, 'X',
  'X', 'X', 'O',
  'O', null, null,
]
```

Board's `renderSquare` method currently looks like this:

Code

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

Modify it to pass a `value` prop to Square.

Code

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

View the current code.

Now we need to change what happens when a square is clicked. The Board component now stores which squares are filled, which means we need some way for Square to update the state of Board. Since component state is considered private, we can't update Board's state directly from Square.

The usual pattern here is pass down a function from Board to Square that gets called when the square is clicked. Change `renderSquare` in Board again so that it reads:

Code

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

We split the returned element into multiple lines for readability, and added parens around it so that JavaScript doesn't insert a semicolon after `return` and break our code.

Now we're passing down two props from Board to Square: `value` and `onClick`. The latter is a function that Square can call. Let's make the following changes to Square:

- Replace `this.state.value` with `this.props.value` in Square's `render`.

- Replace `this.setState()` with `this.props.onClick()` in Square's `render`.
- Delete `constructor` definition from Square because it doesn't have state anymore.

After these changes, the whole Square component looks like this:

Code

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

Now when the square is clicked, it calls the `onClick` function that was passed by Board. Let's recap what happens here:

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 defined in Square's `render()` method.
3. This event handler calls `this.props.onClick()`. Square's props were specified by the Board.
4. Board passed `onClick={() => this.handleClick(i)}` to Square, so, when called, it runs `this.handleClick(i)` on the Board.
5. We have not defined the `handleClick()` method on the Board yet, so the code crashes.

Note that `onClick` on the DOM `<button>` component has a special meaning to React, but we could have called `onClick` prop in Square and `handleClick` in Board something else. It is, however, a common convention in React apps to use `on*` names for the handler prop names and `handle*` for their implementations.

Try clicking a square – you should get an error because we haven't defined `handleClick` yet. Add it to the Board class.

Code

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

  handleClick(i) {
    const squares = this.state.squares.slice();
```

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

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

    render() {
      const status = 'Next player: X';

      return (
        <div>
          <div className="status">{status}</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>
      );
    }
  }
```

View the current code.

We call `.slice()` to copy the `squares` array instead of mutating the existing array. Jump ahead a section to learn why immutability is important.

Now you should be able to click in squares to fill them again, but the state is stored in the Board component instead of in each Square, which lets us continue building the game. Note how whenever Board's state changes, the Square components rerender automatically.

Square no longer keeps its own state; it receives its value from its parent Board and informs its parent when it's clicked. We call components like this **controlled components**.

Why Immutability Is Important

In the previous code example, we suggest using the `.slice()` operator to copy the `squares` array prior to making changes and to prevent mutating the existing array. Let's talk about what this means and why it is an important concept to learn.

There are generally two ways for changing data. The first method is to *mutate* the data by directly changing the values of a variable. The second method is to replace the data with a new copy of the object that also includes desired changes.

Data change with mutation

Code

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

Data change without mutation

Code

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

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

// Or if you are using object spread syntax proposal, you can write:
// var newPlayer = {...player, score: 2};
```

The end result is the same but by not mutating (or changing the underlying data) directly we now have an added benefit that can help us increase component and overall application performance.

Easier Undo/Redo and Time Travel

Immutability also makes some complex features much easier to implement. For example, further in this tutorial we will implement time travel between different stages of the game. Avoiding data mutations lets us keep a reference to older versions of the data, and switch between them if we need to.

Tracking Changes

Determining if a mutated object has changed is complex because changes are made directly to the object. This then requires comparing the current object to a previous copy, traversing the entire object tree, and comparing each variable and value. This process can become increasingly complex.

Determining how an immutable object has changed is considerably easier. If the object being referenced is different from before, then the object has changed. That's it.

Determining When to Re-render in React

The biggest benefit of immutability in React comes when you build simple *pure components*.

Since immutable data can more easily determine if changes have been made it also helps to determine when a component requires being re-rendered.

To learn more about `shouldComponentUpdate()` and how you can build *pure components* take a look at Optimizing Performance.

Functional Components

We've removed the constructor, and in fact, React supports a simpler syntax called **functional components** for component types like Square that only consist of a `render` method. Rather than define a class extending `React.Component`, simply write a function that takes props and returns what should be rendered.

Replace the whole Square class with this function:

Code

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

You'll need to change `this.props` to `props` both times it appears. Many components in your apps will be able to be written as functional components: these components tend to be easier to write and React will optimize them more in the future.

While we're cleaning up the code, we also changed `onClick={() => props.onClick()}` to just `onClick={props.onClick}`, as passing the function down is enough for our example. Note that `onClick={props.onClick()}` would not work because it would call `props.onClick` immediately instead of passing it down.

View the current code.

Taking Turns

An obvious defect in our game is that only X can play. Let's fix that.

Let's default the first move to be by 'X'. Modify our starting state in our Board constructor.

Code

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

Each time we move we shall toggle `xIsNext` by flipping the boolean value and saving the state. Now update Board's `handleClick` function to flip the value of `xIsNext`.

Code

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

Now X and O take turns. Next, change the "status" text in Board's `render` so that it also displays who is next.

Code

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

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

After these changes you should have this Board component:

Code

```
class Board extends React.Component {
  constructor() {
    super();
    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,
    });
  }

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

```
      );
    }

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

      return (
        <div>
          <div className="status">{status}</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>
      );
    }
  }
```

View the current code.

Declaring a Winner

Let's show when a game is won. Add this helper function to the end of the file:

Code

```
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;
  }
```

You can call it in Board's `render` function to check if anyone has won the game and make the status text show "Winner: [X/O]" when someone wins.

Replace the `status` declaration in Board's `render` with this code:

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 (
      // the rest has not changed
```

You can now change `handleClick` in Board to return early and ignore the click if someone has already won the game or if a square is already filled:

Code

```
  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,
    });
  }
```

Congratulations! You now have a working tic-tac-toe game. And now you know the basics of React. So *you're* probably the real winner here.

View the current code.

Storing a History

Let's make it possible to revisit old states of the board so we can see what it looked like after any of the previous moves. We're already creating a new `squares` array each time a move is

made, which means we can easily store the past board states simultaneously.

Let's plan to store an object like this in state:

Code

```
history = [
  {
    squares: [
      null, null, null,
      null, null, null,
      null, null, null,
    ]
  },
  {
    squares: [
      null, null, null,
      null, 'X', null,
      null, null, null,
    ]
  },
  // ...
]
```

We'll want the top-level Game component to be responsible for displaying the list of moves. So just as we pulled the state up before from Square into Board, let's now pull it up again from Board into Game – so that we have all the information we need at the top level.

First, set up the initial state for Game by adding a constructor to it:

Code

```
class Game extends React.Component {
  constructor() {
    super();
    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>
      );
    }
  }
```

Then change Board so that it takes `squares` via props and has its own `onClick` prop specified by Game, like the transformation we made for Square earlier. You can pass the location of each square into the click handler so that we still know which square was clicked. Here is a list of steps you need to do:

- Delete the `constructor` in Board.
- Replace `this.state.squares[i]` with `this.props.squares[i]` in Board's `renderSquare`.
- Replace `this.handleClick(i)` with `this.props.onClick(i)` in Board's `renderSquare`.

Now the whole Board component looks like this:

Code

```
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)}
      />
    );
  }

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

    return (
```

```
        <div>
          <div classname="status">{status}</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>
      );
    }
  }
```

Game's `render` should look at the most recent history entry and can take over calculating the game status:

Code

```
  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>
      );
    }
```

Since Game is now rendering the status, we can delete `<div className="status">{status}</div>` and the code calculating the status from the Board's `render` function:

Code

```
  render() {
    return (
      <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>
    );
  }
```

Next, we need to move the `handleClick` method implementation from Board to Game. You can cut it from the Board class, and paste it into the Game class.

We also need to change it a little, since Game state is structured differently. Game's `handleClick` can push a new entry onto the stack by concatenating the new history entry to make a new history array.

Code

```
  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,
      });
    }
```

At this point, Board only needs `renderSquare` and `render`; the state initialization and click handler should both live in Game.

View the current code.

Showing the Moves

Let's show the previous moves made in the game so far. We learned earlier that React elements are first-class JS objects and we can store them or pass them around. To render multiple items in React, we pass an array of React elements. The most common way to build that array is to map over your array of data. Let's do that in the `render` method of Game:

Code

```
  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 ?
        'Move #' + move :
        'Game start';
      return (
        <li>
          <a href="#" onClick={() => this.jumpTo(move)}>{desc}</a>
        </li>
      );
    });

    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>{moves}</ol>
        </div>
      </div>
    );
  }
```

View the current code.

For each step in the history, we create a list item `<li>` with a link `<a>` inside it that goes nowhere (`href="#"`) but has a click handler which we'll implement shortly. With this code, you should see a list of the moves that have been made in the game, along with a warning that says:

> **Warning: Each child in an array or iterator should have a unique "key" prop. Check the render method of "Game".**

Let's talk about what that warning means.

Keys

When you render a list of items, React always stores some info about each item in the list. If you render a component that has state, that state needs to be stored – and regardless of how you implement your components, React stores a reference to the backing native views.

When you update that list, React needs to determine what has changed. You could've added, removed, rearranged, or updated items in the list.

Imagine transitioning from

Code

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

to

Code

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

To a human eye, it looks likely that Alexa and Ben swapped places and Claudia was added – but React is just a computer program and doesn't know what you intended it to do. As a result, React asks you to specify a *key* property on each element in a list, a string to differentiate each component from its siblings. In this case, `alexa`, `ben`, `claudia` might be sensible keys; if the items correspond to objects in a database, the database ID is usually a good choice:

Code

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

`key` is a special property that's reserved by React (along with `ref`, a more advanced feature). When an element is created, React pulls off the `key` property and stores the key directly on the returned element. Even though it may look like it is part of props, it cannot be referenced with `this.props.key`. React uses the key automatically while deciding which children to update; there is no way for a component to inquire about its own key.

When a list is rerendered, React takes each element in the new version and looks for one with a matching key in the previous list. When a key is added to the set, a component is created; when a key is removed, a component is destroyed. Keys tell React about the identity of each component, so that it can maintain the state across rerenders. If you change the key of a component, it will be completely destroyed and recreated with a new state.

It's strongly recommended that you assign proper keys whenever you build dynamic lists. If you don't have an appropriate key handy, you may want to consider restructuring your data so that you do.

If you don't specify any key, React will warn you and fall back to using the array index as a key – which is not the correct choice if you ever reorder elements in the list or add/remove items anywhere but the bottom of the list. Explicitly passing `key={i}` silences the warning but has the same problem so isn't recommended in most cases.

Component keys don't need to be globally unique, only unique relative to the immediate siblings.

Implementing Time Travel

For our move list, we already have a unique ID for each step: the number of the move when it happened. In the Game's `render` method, add the key as `<li key={move}>` and the key warning should disappear:

Code

```
const moves = history.map((step, move) => {
  const desc = move ?
    'Move #' + move :
    'Game start';
  return (
    <li key={move}>
      <a href="#" onClick={() => this.jumpTo(move)}>{desc}</a>
    </li>
  );
});
```

View the current code.

Clicking any of the move links throws an error because `jumpTo` is undefined. Let's add a new key to Game's state to indicate which step we're currently viewing.

First, add `stepNumber: 0` to the initial state in Game's `constructor`:

Code

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

Next, we'll define the `jumpTo` method in Game to update that state. We also want to update `xIsNext`. We set `xIsNext` to true if the index of the move number is an even number.

Add a method called `jumpTo` to the Game class:

Code

```
  handleClick(i) {
    // this method has not changed
  }

  jumpTo(step) {
    this.setState({
      stepNumber: step,
      xIsNext: (step % 2) ? false : true,
    });
  }

  render() {
    // this method has not changed
  }
```

Then update `stepNumber` when a new move is made by adding `stepNumber: history.length` to the state update in Game's `handleClick`:

Code

```
  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,
      });
    }
```

Now you can modify Game's `render` to read from that step in the history:

Code

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

    // the rest has not changed
```

View the current code.

If you click any move link now, the board should immediately update to show what the game looked like at that time.

You may also want to update `handleClick` to be aware of `stepNumber` when reading the current board state so that you can go back in time then click in the board to create a new entry. (Hint: It's easiest to `.slice()` off the extra elements from `history` at the very top of `handleClick`.)

Wrapping Up

Now, you've made a tic-tac-toe game that:

- lets you play tic-tac-toe,
- indicates when one player has won the game,
- stores the history of moves during the game,
- allows players to jump back in time to see older versions of the game board.

Nice work! We hope you now feel like you have a decent grasp on how React works.

Check out the final result here: Final Result.

If you have extra time or want to practice your new skills, here are some ideas for improvements you could make, listed in order of increasing difficulty:

1. Display the move locations in the format "(1, 3)" instead of "6".
2. Bold the currently-selected item in the move list.
3. Rewrite Board to use two loops to make the squares instead of hardcoding them.
4. Add a toggle button that lets you sort the moves in either ascending or descending order.
5. When someone wins, highlight the three squares that caused the win.

Throughout this tutorial, we have touched on a number of React concepts including elements, components, props, and state. For a more in-depth explanation for each of these

topics, check out the rest of the documentation. To learn more about defining components, check out the `React.Component` API reference.
