РЕАКТ. НАЧАЛО.

Прежде чем мы начнем учебное пособие

Во время этого урока мы создадим небольшую игру. У вас может возникнуть соблазн пропустить это, потому что вы не создаете игры, но дайте ему шанс. Методы, которые вы изучите в этом руководстве, являются основополагающими для создания любых приложений React, и их освоение даст вам глубокое понимание React.

Совет

Этот учебник предназначен для людей, которые предпочитают учиться на практике. Если вы предпочитаете изучать концепции с нуля, ознакомьтесь с нашим пошаговым руководством. Возможно, вам покажется, что этот учебник и руководство дополняют друг друга.

Учебное пособие разделено на несколько разделов:

Настройка для руководства даст вам отправную точку для следования руководству.

Обзор научит вас основам React: компонентам, реквизитам и состоянию.

Прохождение игры научит вас наиболее распространенным техникам разработки React.

Добавление путешествия во времени даст вам более глубокое представление об уникальных сильных сторонах React.

Вам не обязательно заполнять все разделы сразу, чтобы извлечь пользу из этого руководства. Постарайтесь пройти как можно дальше — даже если это один или два участка.

Вы можете копировать и вставлять код, следуя инструкциям в руководстве, но мы рекомендуем вводить его вручную. Это поможет вам развить мышечную память и более глубокое понимание.

Что Мы строим?

В этом уроке мы покажем, как создать интерактивную игру в крестики-нолики с помощью React.

Вы можете увидеть, что мы будем создавать здесь: Конечный результат. Если код не имеет для вас смысла или если вы не знакомы с синтаксисом кода, не волнуйтесь! Цель этого руководства - помочь вам понять React и его синтаксис.

Мы рекомендуем вам ознакомиться с игрой в крестики-нолики, прежде чем продолжить обучение. Одна из особенностей, которую вы заметите, заключается в том, что справа от игрового поля есть нумерованный список. Этот список дает вам историю всех ходов, которые произошли в игре, и обновляется по ходу игры.

Вы можете закрыть игру в крестики-нолики, как только ознакомитесь с ней. В этом руководстве мы начнем с более простого шаблона. Наш следующий шаг - настроить вас так, чтобы вы могли начать создавать игру.

Предпосылки

Мы предположим, что вы немного знакомы с HTML и JavaScript, но вы должны быть в состоянии следовать им, даже если вы используете другой язык программирования. Мы также предположим, что вы знакомы с такими концепциями программирования, как функции, объекты, массивы и, в меньшей степени, классы.

Если вам нужно ознакомиться с JavaScript, мы рекомендуем прочитать это руководство. Обратите внимание, что мы также используем некоторые функции из ES6 — последней версии JavaScript. В этом руководстве мы используем функции со стрелками, классы, операторы let и const. Вы можете использовать Babel REPL, чтобы проверить, к чему компилируется код ES6.

Настройка для учебного пособия

Есть два способа завершить это руководство: вы можете либо написать код в своем браузере, либо настроить локальную среду разработки на своем компьютере.

Вариант настройки 1: Напишите код в браузере

Это самый быстрый способ начать работу!

Сначала откройте этот стартовый код на новой вкладке. На новой вкладке должно отображаться пустое игровое поле в крестики-нолики и код React. В этом руководстве мы будем редактировать код React.

Теперь вы можете пропустить второй вариант настройки и перейти в раздел Обзор, чтобы получить общее представление о React.

Вариант настройки 2: Локальная среда разработки

Это совершенно необязательно и не требуется для данного руководства!

Необязательно: Инструкции для выполнения локально с помощью предпочитаемого вами текстового редактора

Эта настройка требует больше работы, но позволяет вам завершить учебное пособие, используя редактор по вашему выбору. Вот шаги, которым нужно следовать:

Убедитесь, что у вас установлена последняя версия Node.js установлен.

Следуйте инструкциям по установке Create React App, чтобы создать новый проект. npx create-react-app my-app

Удалите все файлы в папке src/ нового проекта

Примечание: не удаляйте всю папку src, только исходные файлы внутри нее .. На следующем шаге мы заменим исходные файлы по умолчанию примерами для этого проекта.

cd my-app

cd src

# If you're using a Mac or Linux:

rm -f \*

# Or, if you're on Windows:

del \*

# Then, switch back to the project folder

cd ..

Примечание: не удаляйте всю папку src, только исходные файлы внутри нее .. На следующем шаге мы заменим исходные файлы по умолчанию примерами для этого проекта. Добавьте файл с именем index.css в папку src/ с помощью этого CSS-кода.

Добавьте файл с именем index.js в папке src/ с этим JS-кодом.

Добавьте эти три строки в начало index.js в папке src/:

import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

Теперь, если вы запустите npm, запустите в папке проекта и откройте http://localhost:3000 в браузере вы должны увидеть пустое поле для игры в крестики-нолики.

Мы рекомендуем следовать этим инструкциям, чтобы настроить подсветку синтаксиса для вашего редактора.

Помогите, я застрял!

Если вы застряли, обратитесь к ресурсам поддержки сообщества. В частности, чат Reactiflux - отличный способ быстро получить помощь. Если вы не получили ответа или по-прежнему застряли, пожалуйста, сообщите о проблеме, и мы поможем вам.

Обзор

Теперь, когда вы настроены, давайте рассмотрим React!

Что такое React?

React - это декларативная, эффективная и гибкая библиотека JavaScript для создания пользовательских интерфейсов. Это позволяет вам создавать сложные пользовательские интерфейсы из небольших и изолированных фрагментов кода, называемых “компонентами”.

В React есть несколько различных типов компонентов, но мы начнем с React.Подклассы компонентов:

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" />

В React есть несколько различных типов компонентов, но мы начнем с React.Подклассы компонентов:

Скоро мы перейдем к забавным XML-подобным тегам. Мы используем компоненты, чтобы сообщить React, что мы хотим видеть на экране. Когда наши данные изменятся, React эффективно обновит и повторно отобразит наши компоненты.

Здесь ShoppingList - это класс компонента React или тип компонента React. Компонент принимает параметры, называемые props (сокращение от “свойства”), и возвращает иерархию представлений для отображения с помощью метода render.

Метод render возвращает описание того, что вы хотите видеть на экране. React принимает описание и отображает результат. В частности, render возвращает элемент React, который представляет собой упрощенное описание того, что нужно отобразить. Большинство разработчиков React используют специальный синтаксис под названием “JSX”, который упрощает написание этих структур. Синтаксис <div /> преобразуется во время сборки в React.createElement('div'). Приведенный выше пример эквивалентен:

return React.createElement('div', {className: 'shopping-list'},

React.createElement('h1', /\* ... h1 children ... \*/),

React.createElement('ul', /\* ... ul children ... \*/)

);

Смотрите полную расширенную версию.

Если вам интересно, функция createElement() более подробно описана в справочнике по API, но мы не будем использовать ее в этом руководстве. Вместо этого мы продолжим использовать JSX.

JSX поставляется со всей мощью JavaScript. Вы можете поместить любые выражения JavaScript в фигурные скобки внутри JSX. Каждый элемент React - это объект JavaScript, который вы можете сохранить в переменной или передать в своей программе.

Приведенный выше компонент ShoppingList отображает только встроенные компоненты DOM, такие как <div /> и <li />. Но вы также можете создавать и визуализировать пользовательские компоненты React. Например, теперь мы можем обратиться ко всему списку покупок, написав <ShoppingList />. Каждый компонент React инкапсулирован и может работать независимо; это позволяет создавать сложные пользовательские интерфейсы из простых компонентов.

Проверка начального кода

Если вы собираетесь работать над руководством в своем браузере, откройте этот код на новой вкладке: Начальный код. Если вы собираетесь работать над учебником локально, вместо этого откройте src/index.js в папке вашего проекта (вы уже касались этого файла во время установки).

Этот стартовый код является основой того, что мы создаем. Мы предусмотрели CSS-стиль, так что вам нужно сосредоточиться только на изучении React и программировании игры в крестики-нолики.

Изучив код, вы заметите, что у нас есть три компонента React:

Площадь

Доска

Игра

Квадратный компонент отображает одну <кнопку>, а доска отображает 9 квадратов. Игровой компонент отображает доску со значениями-заполнителями, которые мы изменим позже. В настоящее время интерактивных компонентов нет.

Передача Данных через Реквизиты

Просто чтобы промочить наши ноги, давайте попробуем передать некоторые данные из нашего компонента Board в наш квадратный компонент.

В методе renderSquare Board измените код, чтобы передать значение prop, называемое значением, квадрату:

class Board extends React.Component {

renderSquare(i) {

return <Square value={i} />;

}

Измените метод рендеринга Square, чтобы показать это значение, заменив {/\* TODO \*/} на {this.props.value}:

class Square extends React.Component {

render() {

return (

<button className="square">

{this.props.value}

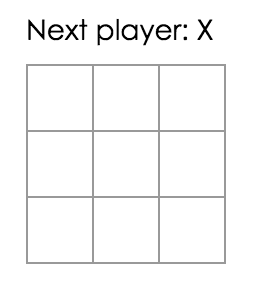
</button>

);

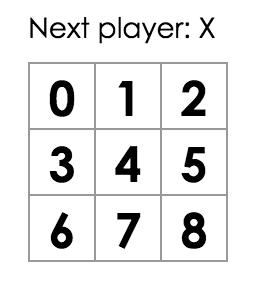
}

}

До



После: Вы должны увидеть число в каждом квадрате в отображаемом выводе.



Просмотрите полный код на этом этапе

Поздравляю! Вы только что “передали реквизит” из родительского компонента платы дочернему компоненту Square. Передача реквизитов - это способ передачи информации в приложениях React от родителей к детям.

Создание интерактивного компонента

Давайте заполним квадратный компонент символом “X”, когда мы щелкнем по нему. Сначала измените тег кнопки, который возвращается из функции render() компонента Square, на этот:

class Square extends React.Component {

render() {

return (

<button className="square" onClick={function() { alert('click'); }}>

{this.props.value}

</button>

);

}

}

Если мы сейчас нажмем на квадрат, мы должны получить предупреждение в нашем браузере.

Примечание

Чтобы сэкономить на вводе текста и избежать такого запутанного поведения, мы будем использовать синтаксис функции arrow для обработчиков событий здесь и далее ниже:

class Square extends React.Component {

render() {

return (

<button className="square" onClick={() => alert('click')}>

{this.props.value}

</button>

);

}

}

Обратите внимание, что с помощью onClick={() => alert('click')} мы передаем функцию в качестве реквизита onClick. Он срабатывает только после щелчка. Забывание () => и запись onClick={alert('click')} является распространенной ошибкой и приводит к срабатыванию предупреждения каждый раз при повторном отображении компонента.

В качестве следующего шага мы хотим, чтобы квадратный компонент “запомнил”, что на него нажали, и заполнил его знаком “X”. Чтобы “запоминать” вещи, компоненты используют состояние.

Компоненты React могут иметь состояние, установив this.state в своих конструкторах. this.state следует рассматривать как частное для компонента React, в котором оно определено. Давайте сохраним текущее значение квадрата в this.state и изменим его при нажатии на квадрат.

Сначала мы добавим в класс конструктор для инициализации состояния:

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>

);

}

}

Примечание

В классах JavaScript вам всегда нужно вызывать super при определении конструктора подкласса. Все классы компонентов React, у которых есть конструктор, должны запускать его с помощью вызова super(props).

Теперь мы изменим метод рендеринга квадрата, чтобы при нажатии отображалось значение текущего состояния:

Замените this.props.value на this.state.value внутри тега <button>.

Замените обработчик события () => alert() на () => this.setState({значение: 'X'}).

Поместите реквизиты className и onClick в отдельные строки для лучшей читаемости.

После этих изменений тег <button>, возвращаемый методом рендеринга Square, выглядит следующим образом:

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>

);

}

}

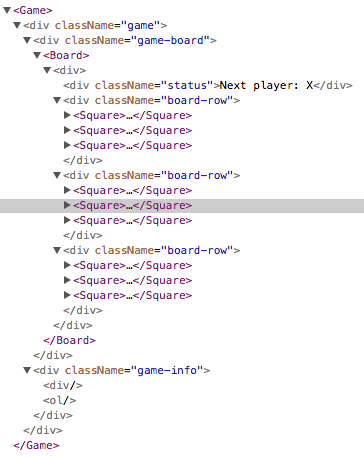
Вызывая this.setState из обработчика onClick в методе рендеринга Square, мы говорим React повторно отображать этот квадрат всякий раз, когда нажимается его <кнопка>. После обновления значение this.state.value квадрата будет равно 'X', поэтому мы увидим X на игровом поле. Если вы нажмете на любой квадрат, должен появиться крестик.

Когда вы вызываете setState в компоненте, React автоматически обновляет дочерние компоненты внутри него.

Просмотрите полный код на этом этапе

Инструменты разработчика

Расширение React Devtools для Chrome и Firefox позволяет просматривать дерево компонентов React с помощью инструментов разработчика вашего браузера.



React DevTools позволяют вам проверять реквизиты и состояние ваших компонентов React.

После установки React DevTools вы можете щелкнуть правой кнопкой мыши на любом элементе страницы, нажать “Проверить”, чтобы открыть инструменты разработчика, и вкладка React появится как последняя вкладка справа.

Однако обратите внимание, что есть несколько дополнительных шагов, чтобы заставить его работать с CodePen:

Войдите в систему или зарегистрируйтесь и подтвердите свой адрес электронной почты (требуется для предотвращения спама).

Нажмите кнопку “Вилка”.

Нажмите “Изменить вид”, а затем выберите “Режим отладки”.

На открывшейся новой вкладке у devtools теперь должна быть вкладка React.

Завершение игры

Теперь у нас есть основные строительные блоки для нашей игры в крестики-нолики. Чтобы провести полноценную игру, нам теперь нужно поочередно расставлять “X” и “O” на доске, и нам нужен способ определить победителя.

Состояние подъема Вверх

В настоящее время каждый квадратный компонент поддерживает состояние игры. Чтобы определить победителя, мы сохраним значение каждого из 9 квадратов в одном месте.

Мы можем подумать, что Правление должно просто запрашивать у каждого квадрата его состояние. Хотя такой подход возможен в React, мы не рекомендуем его использовать, потому что код становится трудным для понимания, подвержен ошибкам и трудно поддается рефакторингу. Вместо этого наилучшим подходом является сохранение состояния игры в компоненте родительской платы, а не в каждом квадрате. Компонент Board может указывать каждому квадрату, что отображать, передавая реквизит, точно так же, как мы делали, когда передавали число каждому квадрату.

Чтобы собирать данные от нескольких дочерних компонентов или чтобы два дочерних компонента взаимодействовали друг с другом, вам необходимо вместо этого объявить общее состояние в их родительском компоненте. Родительский компонент может передавать состояние обратно дочерним элементам с помощью реквизитов; это позволяет синхронизировать дочерние компоненты друг с другом и с родительским компонентом.

Перенос состояния в родительский компонент является обычным явлением при рефакторинге компонентов React — давайте воспользуемся этой возможностью, чтобы попробовать это. Мы добавим конструктор к плате и установим начальное состояние платы так, чтобы оно содержало массив с 9 нулями. Эти 9 нулей соответствуют 9 квадратам:

class Board extends React.Component {

constructor(props) {

super(props);

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>

);

}

}

Когда мы позже заполним доску, она будет выглядеть примерно так:

[

'O', null, 'X',

'X', 'X', 'O',

'O', null, null,

]

Метод renderSquare платы в настоящее время выглядит следующим образом:

renderSquare(i) {

return <Square value={i} />;

}

Вначале мы передали значение реквизита вниз с доски, чтобы показать числа от 0 до 8 в каждом квадрате. На другом предыдущем шаге мы заменили цифры знаком “X”, определяемым собственным состоянием Square. Вот почему Square в настоящее время игнорирует значение prop, переданное ему правлением.

Теперь мы снова будем использовать механизм передачи реквизита. Мы изменим доску, чтобы указать каждому отдельному квадрату его текущее значение ('X', 'O' или null). Мы уже определили массив квадратов в конструкторе платы, и мы изменим метод renderSquare платы для чтения из него:

renderSquare(i) {

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

}

Просмотрите полный код на этом этапе

Каждый квадрат теперь получит значение prop, которое будет либо 'X', 'O', либо null для пустых квадратов.

Далее нам нужно изменить то, что происходит при нажатии на квадрат. Компонент Board теперь поддерживает, какие квадраты заполнены. Нам нужно создать способ для квадрата обновлять состояние доски. Поскольку состояние считается закрытым для компонента, который его определяет, мы не можем обновить состояние платы непосредственно из Square.

Чтобы сохранить конфиденциальность состояния доски, мы передадим функцию с доски на Квадрат. Эта функция будет вызвана при нажатии на квадрат. Мы изменим метод renderSquare в Board на:

renderSquare(i) {

return (

<Square

value={this.state.squares[i]}

onClick={() => this.handleClick(i)}

/>

);

}

Примечание

Мы разделили возвращаемый элемент на несколько строк для удобства чтения и добавили круглые скобки, чтобы JavaScript не вставлял точку с запятой после возврата и не прерывал наш код.

Теперь мы передаем два реквизита с доски на квадрат: значение и onClick. Реквизит onClick - это функция, которую Square может вызывать при нажатии. Мы внесем следующие изменения в Square:

Замените this.state.value на this.props.value в методе визуализации Square

Замените this.setState() на this.props.onClick() в методе рендеринга Square

Удалите конструктор из Square, потому что Square больше не отслеживает состояние игры

После этих изменений квадратный компонент выглядит следующим образом:

class Square extends React.Component {

render() {

return (

<button

className="square"

onClick={() => this.props.onClick()}

>

{this.props.value}

</button>

);

}

}

При щелчке по квадрату вызывается функция onClick, предоставляемая доской. Вот обзор того, как это достигается:

Реквизит onClick во встроенном компоненте DOM <кнопка> сообщает React настроить прослушиватель событий щелчка.

Когда кнопка нажата, React вызовет обработчик события onClick, который определен в методе Square render().

Этот обработчик события вызывает this.props.onClick(). Реквизит onClick для квадрата был указан Правлением.

Поскольку доска передала onClick={() => this.handleClick(i)} квадрату, квадрат вызывает this.handleClick(i) при нажатии.

Мы еще не определили метод handleClick(), поэтому наш код выходит из строя.

Примечание

Атрибут onClick элемента DOM <button> имеет особое значение для React, поскольку он является встроенным компонентом. Для пользовательских компонентов, таких как Square, именование зависит от вас. Мы могли бы назвать метод onClick prop квадрата или метод handleClick доски по-другому. В React, однако, принято использовать имена on[Event] для реквизитов, которые представляют события, и handle[Event] для методов, которые обрабатывают события.

Когда мы пытаемся щелкнуть по квадрату, мы должны получить сообщение об ошибке, потому что мы еще не определили handleClick. Теперь мы добавим handleClick в класс Board:

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});

}

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>

);

}

}

Просмотрите полный код на этом этапе

После этих изменений мы снова можем нажимать на квадраты, чтобы заполнить их. Однако теперь состояние сохраняется в компоненте платы вместо отдельных квадратных компонентов. При изменении состояния платы квадратные компоненты автоматически перерисовываются. Сохранение состояния всех квадратов в компоненте доски позволит ему определить победителя в будущем.

Поскольку квадратные компоненты больше не поддерживают состояние, квадратные компоненты получают значения от компонента платы и сообщают компоненту платы при нажатии на них. В терминах React квадратные компоненты теперь являются управляемыми компонентами. Правление имеет полный контроль над ними.

Обратите внимание, как в handleClick мы вызываем .slice() для создания копии массива squares для изменения вместо изменения существующего массива. В следующем разделе мы объясним, почему мы создаем копию массива squares.

Почему важна Неизменяемость

В предыдущем примере кода мы предложили вам использовать оператор .slice() для создания копии массива squares для изменения вместо изменения существующего массива. Теперь мы обсудим неизменяемость и почему неизменяемость важна для изучения.

Как правило, существует два подхода к изменению данных. Первый подход заключается в изменении данных путем непосредственного изменения значений данных. Второй подход заключается в замене данных новой копией, которая содержит желаемые изменения.

Изменение данных с помощью мутации

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

player.score = 2;

// Now player is {score: 2, name: 'Jeff'}

Изменение данных без мутации

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};

Конечный результат тот же, но, не мутируя (или не изменяя базовые данные) напрямую, мы получаем несколько преимуществ, описанных ниже.

Сложные функции становятся простыми

Неизменяемость значительно упрощает реализацию сложных функций. Позже в этом руководстве мы внедрим функцию “путешествия во времени”, которая позволит нам просмотреть историю игры в крестики-нолики и “вернуться” к предыдущим ходам. Эта функциональность не специфична для игр — возможность отмены и повтора определенных действий является общим требованием в приложениях. Избегая прямой мутации данных, мы сохраняем предыдущие версии истории игры нетронутыми и повторно используем их позже.

Обнаружение изменений

Обнаружение изменений в изменяемых объектах затруднено, поскольку они модифицируются напрямую. Это обнаружение требует сравнения изменяемого объекта с предыдущими копиями самого себя и обхода всего дерева объектов.

Обнаруживать изменения в неизменяемых объектах значительно проще. Если неизменяемый объект, на который ссылаются, отличается от предыдущего, значит, объект изменился.

Определение момента повторного рендеринга в React

Главное преимущество неизменяемости заключается в том, что она помогает вам создавать чистые компоненты в React. Неизменяемые данные позволяют легко определить, были ли внесены изменения, что помогает определить, когда компоненту требуется повторная визуализация.

Вы можете узнать больше о shouldComponentUpdate() и о том, как вы можете создавать чистые компоненты, прочитав "Оптимизация производительности".

Функциональные компоненты

Теперь мы изменим квадрат на функциональный компонент.

В React функциональные компоненты - это более простой способ написания компонентов, которые содержат только метод визуализации и не имеют собственного состояния. Вместо определения класса, который расширяет React.Компонент, мы можем написать функцию, которая принимает props в качестве входных данных и возвращает то, что должно быть отрисовано. Функциональные компоненты менее утомительны в написании, чем классы, и многие компоненты могут быть выражены таким образом.

Замените класс Square этой функцией:

function Square(props) {

return (

<button className="square" onClick={props.onClick}>

{props.value}

</button>

);

}

Мы изменили this.props на props оба раза, когда он появляется.

Просмотрите полный код на этом этапе

Примечание

Когда мы модифицировали квадрат, чтобы он был функциональным компонентом, мы также изменили onClick={() => this.props.onClick()} на более короткий onClick ={props.onClick} (обратите внимание на отсутствие круглых скобок с обеих сторон). В классе мы использовали функцию со стрелкой для доступа к правильному значению this, но в функциональном компоненте нам не нужно беспокоиться об этом.

По очереди

Теперь нам нужно исправить очевидный дефект в нашей игре в крестики-нолики: буквы “О” не могут быть отмечены на доске.

По умолчанию мы установим для первого хода значение “X”. Мы можем установить это значение по умолчанию, изменив начальное состояние в нашем конструкторе платы:

class Board extends React.Component {

constructor(props) {

super(props);

this.state = {

squares: Array(9).fill(null),

xIsNext: true,

};

}

Каждый раз, когда игрок делает ход, xIsNext (логическое значение) будет переворачиваться, чтобы определить, какой игрок идет следующим, и состояние игры будет сохранено. Мы обновим функцию handleClick на плате, чтобы изменить значение xIsNext:

handleClick(i) {

const squares = this.state.squares.slice();

squares[i] = this.state.xIsNext ? 'X' : 'O';

this.setState({

squares: squares,

xIsNext: !this.state.xIsNext,

});

}

С этим изменением буквы “X” и “O” могут меняться по очереди. Давайте также изменим текст “статус” в рендере доски, чтобы он отображал, у какого игрока следующий ход:

render() {

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

return (

// the rest has not changed

После применения этих изменений у вас должен быть этот компонент платы:

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,

});

}

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>

);

}

}

Просмотрите полный код на этом этапе

Объявление победителя

Теперь, когда мы показываем, у какого игрока следующий ход, мы также должны показать, когда игра выиграна и больше не нужно делать ходов. Мы можем определить победителя, добавив эту вспомогательную функцию в конец файла:

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;

}

Мы вызовем calculateWinner(квадраты) в функции визуализации доски, чтобы проверить, выиграл ли игрок. Если игрок выиграл, мы можем отобразить текст, такой как “Победитель: X” или “Победитель: O”. Мы заменим объявление статуса в функции отображения доски этим кодом:

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

Теперь мы можем изменить функцию handleClick на доске, чтобы она возвращалась раньше, игнорируя щелчок, если кто-то выиграл игру или если квадрат уже заполнен:

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,

});

}

Просмотрите полный код на этом этапе

Поздравляю! Теперь у вас есть рабочая игра в крестики-нолики. И вы тоже только что изучили основы React. Так что вы, вероятно, здесь настоящий победитель.

Добавление путешествия во времени

В качестве заключительного упражнения давайте сделаем возможным “вернуться во времени” к предыдущим ходам в игре.

Хранение истории перемещений

Если бы мы изменили массив квадратов, реализовать путешествие во времени было бы очень сложно.

Однако мы использовали slice() для создания новой копии массива squares после каждого перемещения и рассматривали его как неизменяемый. Это позволит нам сохранить каждую прошлую версию массива квадратов и перемещаться между поворотами, которые уже произошли.

Мы сохраним массивы прошлых квадратов в другом массиве, называемом историей. Массив истории представляет все состояния доски, от первого до последнего хода, и имеет форму, подобную этой:

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',

]

},

// ...

]

Теперь нам нужно решить, какому компоненту должно принадлежать состояние истории.

Поднимая состояние вверх, Снова

Мы хотим, чтобы игровой компонент верхнего уровня отображал список прошлых ходов. Для этого потребуется доступ к истории, поэтому мы поместим состояние истории в игровой компонент верхнего уровня.

Размещение состояния истории в игровом компоненте позволяет нам удалить состояние квадратов из его дочернего компонента Board. Точно так же, как мы “подняли состояние” из квадратного компонента в компонент доски, теперь мы поднимаем его с доски в игровой компонент верхнего уровня. Это дает игровому компоненту полный контроль над данными доски и позволяет ему указывать доске отображать предыдущие ходы из истории.

Сначала мы настроим начальное состояние для игрового компонента в его конструкторе:

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>

);

}

}

Далее мы заставим компонент Board получать квадраты и нажимать на реквизит из игрового компонента. Поскольку теперь у нас есть обработчик одного клика в Board для многих квадратов, нам нужно будет передать местоположение каждого квадрата в обработчик onClick, чтобы указать, на какой квадрат был нажат. Вот необходимые шаги для преобразования компонента платы:

Удалите конструктор в Board.

Замените this.state.squares[i] на this.props.squares[i] в renderSquare доски.

Замените this.handleClick(i) на this.props.onClick(i) в renderSquare доски.

Компонент платы теперь выглядит следующим образом:

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>

);

}

}

Мы обновим функцию рендеринга игрового компонента, чтобы использовать самую последнюю запись истории для определения и отображения статуса игры:

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>

);

}

Поскольку игровой компонент теперь отображает статус игры, мы можем удалить соответствующий код из метода рендеринга платы. После рефакторинга функция рендеринга платы выглядит следующим образом:

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>

);

}

Наконец, нам нужно переместить метод handleClick из компонента Board в компонент Game. Нам также необходимо изменить handleClick, потому что состояние игрового компонента структурировано по-другому. В игровом методе handleClick мы объединяем новые записи истории в историю.

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,

});

}

Примечание

В отличие от метода array push(), с которым вы, возможно, более знакомы, метод concat() не изменяет исходный массив, поэтому мы предпочитаем его.

На данный момент компоненту Board нужны только методы render Square и render render. Состояние игры и метод handleClick должны быть в игровом компоненте.

Просмотрите полный код на этом этапе

Показ прошлых ходов

Поскольку мы записываем историю игры в крестики-нолики, теперь мы можем отобразить ее игроку в виде списка прошлых ходов.

Ранее мы узнали, что элементы React - это первоклассные объекты JavaScript; мы можем передавать их в наших приложениях. Чтобы отобразить несколько элементов в React, мы можем использовать массив элементов React.

В JavaScript массивы имеют метод map(), который обычно используется для сопоставления данных с другими данными, например:

const numbers = [1, 2, 3];

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

Используя метод map, мы можем сопоставить нашу историю ходов с элементами React, представляющими кнопки на экране, и отобразить список кнопок для “перехода” к прошлым ходам.

Давайте сопоставим историю с помощью метода рендеринга игры:

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>

);

});

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>

);

}

Просмотрите полный код на этом этапе

Для каждого хода в истории игры в крестики-нолики мы создаем элемент списка <li>, который содержит кнопку <button>. Кнопка имеет обработчик onClick, который вызывает метод с именем this.jump To(). Мы еще не внедрили метод jumpTo(). На данный момент мы должны увидеть список ходов, которые произошли в игре, и предупреждение в консоли инструментов разработчика, в котором говорится:

Предупреждение: Каждый дочерний элемент в массиве или итераторе должен иметь уникальный “ключевой” реквизит. Проверьте метод рендеринга “Игры”.

Давайте обсудим, что означает приведенное выше предупреждение.

Подбираем ключ

Когда мы визуализируем список, React сохраняет некоторую информацию о каждом отображаемом элементе списка. Когда мы обновляем список, React должен определить, что изменилось. Мы могли бы добавить, удалить, переставить или обновить элементы списка.

Представьте себе переход от

<li>Alexa: 7 tasks left</li>

<li>Ben: 5 tasks left</li>

К

<li>Ben: 9 tasks left</li>

<li>Claudia: 8 tasks left</li>

<li>Alexa: 5 tasks left</li>

В дополнение к обновленным подсчетам, человек, читающий это, вероятно, сказал бы, что мы поменяли местами заказы Алексы и Бена и вставили Клаудию между Алексой и Беном. Однако React - это компьютерная программа, и она не знает, что мы задумали. Поскольку React не может знать наших намерений, нам нужно указать ключевое свойство для каждого элемента списка, чтобы отличать каждый элемент списка от его братьев и сестер. Одним из вариантов было бы использовать строки alexa, ben, claudia. Если бы мы отображали данные из базы данных, идентификаторы базы данных Алексы, Бена и Клаудии можно было бы использовать в качестве ключей.

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

При повторном отображении списка React берет ключ каждого элемента списка и ищет соответствующий ключ в элементах предыдущего списка. Если в текущем списке есть ключ, которого ранее не существовало, React создает компонент. Если в текущем списке отсутствует ключ, который существовал в предыдущем списке, React уничтожает предыдущий компонент. Если две клавиши совпадают, перемещается соответствующий компонент. Ключи сообщают React об идентичности каждого компонента, что позволяет React поддерживать состояние между повторными рендерингами. Если ключ компонента изменится, компонент будет уничтожен и повторно создан с новым состоянием.

ключ - это специальное и зарезервированное свойство в React (наряду с ref, более продвинутой функцией). Когда элемент создается, React извлекает свойство key и сохраняет ключ непосредственно в возвращаемом элементе. Несмотря на то, что ключ может выглядеть так, как будто он принадлежит props, на ключ нельзя ссылаться с помощью this.props.key. React автоматически использует ключ, чтобы решить, какие компоненты следует обновить. Компонент не может запросить свой ключ.

Настоятельно рекомендуется назначать правильные ключи всякий раз, когда вы создаете динамические списки. Если у вас нет соответствующего ключа, вы можете рассмотреть возможность реструктуризации ваших данных таким образом, чтобы у вас это было.

Если ключ не указан, React выдаст предупреждение и по умолчанию будет использовать индекс массива в качестве ключа. Использование индекса массива в качестве ключа проблематично при попытке изменить порядок элементов списка или вставке / удалении элементов списка. Явная передача key={i} отключает предупреждение, но имеет те же проблемы, что и индексы массива, и в большинстве случаев не рекомендуется.

Ключи не обязательно должны быть глобально уникальными; они должны быть уникальными только между компонентами и их братьями и сестрами.

Реализация путешествия во времени

В истории игры в крестики-нолики с каждым прошлым ходом связан уникальный идентификатор: это порядковый номер хода. Ходы никогда не переупорядочиваются, не удаляются и не вставляются посередине, поэтому безопасно использовать индекс хода в качестве ключа.

В методе рендеринга игрового компонента мы можем добавить ключ как <li key={move}>, и предупреждение React о ключах должно исчезнуть:

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>

);

});

Просмотрите полный код на этом этапе

Нажатие любой из кнопок элемента списка выдает ошибку, поскольку метод jumpTo не определен. Прежде чем мы внедрим jumpTo, мы добавим stepNumber в состояние игрового компонента, чтобы указать, какой шаг мы просматриваем в данный момент.

Сначала добавьте stepNumber: 0 в начальное состояние в конструкторе игры:

class Game extends React.Component {

constructor(props) {

super(props);

this.state = {

history: [{

squares: Array(9).fill(null),

}],

stepNumber: 0,

xIsNext: true,

};

}

Далее мы определим метод jumpTo в игре, чтобы обновить этот номер шага. Мы также устанавливаем xIsNext в значение true, если число, на которое мы меняем stepNumber, четное:

handleClick(i) {

// this method has not changed

}

jumpTo(step) {

this.setState({

stepNumber: step,

xIsNext: (step % 2) === 0,

});

}

render() {

// this method has not changed

}

Теперь мы внесем несколько изменений в игровой метод handleClick, который срабатывает, когда вы нажимаете на квадрат.

Добавленное нами состояние stepNumber отражает перемещение, отображаемое пользователю сейчас. После того, как мы сделаем новый ход, нам нужно обновить номер шага, добавив номер шага: history.length как часть аргумента this.setState. Это гарантирует, что мы не застрянем, показывая один и тот же ход после того, как был сделан новый.

Мы также заменим чтение this.state.history на this.state.history.slice(0, this.state.stepNumber + 1). Это гарантирует, что если мы “вернемся назад во времени”, а затем сделаем новый ход с этой точки, мы отбросим всю “будущую” историю, которая теперь стала бы неверной.

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,

});

}

Наконец, мы изменим метод рендеринга игрового компонента с всегда рендеринга последнего хода на рендеринг текущего выбранного хода в соответствии с номером шага:

render() {

const history = this.state.history;

const current = history[this.state.stepNumber];

const winner = calculateWinner(current.squares);

// the rest has not changed

Если мы нажмем на любой шаг в истории игры, доска для игры в крестики-нолики должна немедленно обновиться, чтобы показать, как выглядела доска после того, как произошел этот шаг.

Просмотрите полный код на этом этапе

Подведение итогов

Поздравляю! Вы создали игру в крестики-нолики, которая:

Позволяет играть в крестики-нолики,

указывает, когда игрок выиграл игру,

Сохраняет историю игры по ходу игры,

позволяет игрокам просматривать историю игры и просматривать предыдущие версии игрового поля.

Отличная работа! Мы надеемся, что теперь вы чувствуете, что имеете приличное представление о том, как работает React.

Ознакомьтесь с окончательным результатом здесь: Окончательный результат.

Если у вас есть дополнительное время или вы хотите попрактиковаться в своих новых навыках реагирования, вот несколько идей по улучшению игры в крестики-нолики, которые перечислены в порядке возрастания сложности:

Отобразите местоположение для каждого перемещения в формате (col, row) в списке истории перемещений.

Выделите жирным шрифтом выбранный в данный момент элемент в списке перемещения.

Перепишите доску, чтобы использовать две петли для создания квадратов вместо их жесткого кодирования.

Добавьте кнопку переключения, которая позволяет сортировать ходы в порядке возрастания или убывания.

Когда кто-то выигрывает, выделите три квадрата, которые привели к выигрышу.

Когда никто не выиграет, отобразится сообщение о том, что результат - ничья.

На протяжении всего этого урока мы касались концепций React, включая элементы, компоненты, реквизиты и состояние. Для более подробного объяснения каждой из этих тем ознакомьтесь с остальной документацией. Чтобы узнать больше об определении компонентов, ознакомьтесь с React.Ссылка на компонентный API.