

# React Oyun Geliştirme

Bu öğreticide küçük bir oyun geliştireceğiz. Oyun geliştiricisi olmadığınızdan dolayı bu öğreticiyi atlamak istiyor olabilirsiniz — ama bir şans vermeniz iyi olacaktır. Zira bu öğreticide edineceğiniz teknikler herhangi bir React uygulaması geliştirmek için temel niteliğindedir, ve bu temeller üzerinde uzmanlaşmak React'i daha derinlemesine öğrenmenizi sağlayacaktır.

Not:

Bu öğretici, kodlayarak öğrenmek isteyen kişiler için tasarlanmıştır. Eğer bu konseptleri her yönüyle edinmek isterseniz adım adım öğrenme rehberini inceleyebilirsiniz. İncelediğinizde, öğreticinin ve adım adım öğrenme rehberinin birbirini tamamlayıcı nitelikte olduğunu görebilirsiniz.

Bu öğretici birkaç bölüme ayrılmıştır:

Öğretici İçin Kurulum Rehberi: bu öğreticiyi takip etmek için size bir başlangıç noktası sunar.

Genel bakış: React'in temellerini öğretecektir: component'lar, prop'lar, ve uygulama state'i.

Oyunun Tamamlanması: React geliştirimindeki en yaygın teknikleri aktaracaktır.

Zamanda Yolculuğun Eklenmesi: React'in benzersiz özellikleri hakkında daha derinlemesine bilgiler edinmenizi sağlayacaktır.

Bu öğreticiden yararlanmanız için tüm bölümleri tamamen bitirmek zorunda değilsiniz. Bir-iki bölüm tamamlamanız bile sizin için yararlı olacaktır. Fakat yine de tüm bölümleri tamamlamaya çalışınız.

Bu öğreticiyi takip ederken kodları kopyala-yapıştır yaparak denemenizde bizce hiçbir sorun yoktur. Fakat elle kodlayarak ilerlemenizi tavsiye ederiz. Bu sayede kas hafızanız gelişecek ve React'i daha güçlü bir şekilde öğrenmiş hale geleceksiniz.

## Ne kodlayacağız?

Bu öğreticide, React ile bir tic-tac-toe (XOX oyunu) nasıl geliştirilir onu göstereceğiz.

Buradan oyunun son halini görebilirsiniz. Eğer bu kodlara aşina değilseniz ve size karışık geliyorsa endişelenmeyin. Çünkü bu öğreticinin amacı, React'in ve React'teki kod yapısının anlaşılmasında size yardımcı olmaktır.

Bu öğreticiye başlamadan önce, yukarıda belirttiğimiz linke giderek oyunu oynamanızı ve incelemenizi tavsiye ediyoruz. Oyunu oynadığınızda farkedeyeceğiniz gibi; oyun tahtasının sağında, numaralandırılmış bir liste bulunmaktadır. Bu liste size, oyunda oluşan hamlelerin bir geçmişini sunar ve oyunda ilerledikçe liste de otomatik olarak güncellenir.

Oyunu inceledikten sonra ilgili sayfayı kapatabilirsiniz. Çünkü bu öğreticiye sıfırdan bir şablonla başlayacağız.

Gelin şimdi oyunu kodlamak için gereken geliştirme ortamının kurulumuna değinelim.

## Ön gereksinimler

Bu öğreticide, HTML ve JavaScript'i az-çok bildiğinizi varsayacağız. Fakat herhangi bir programlama dilinden gelseniz bile aşamaları takip edebilirsiniz. Ayrıca temel programlama konseptleri olan fonksiyonlara, nesnelere, dizilere ve az da olsa sınıflara aşina olduğunuzu varsayıyoruz.

Eğer JavaScript hakkında bilgi edinmeniz gerekiyorsa, bu rehberi okumanızı tavsiye ederiz. JavaScript'in en güncel versiyonu olan ES6 (EcmaScript 6)'dan bazı özellikleri kullanacağız. Bu öğreticide de hepsi birer ES6 özelliği olan arrow function, class, let, ve const ifadelerini kullanıyor olacağız. ES6, henüz her tarayıcı tarafından tam olarak desteklenmediği için Babel REPL'i kullanarak ES6 kodunun derlenmiş halini görebilirsiniz.

## Öğretici için Kurulum

Bu öğreticiyi tamamlamanın iki yolu bulunmaktadır: kodu tarayıcınızda yazabilir veya bilgisayarınıza yerel geliştirme ortamını kurabilirsiniz.

### Kurulum Seçeneği 1: Kodu Tarayıcıda Yazma

Başlamanız için en kolay olan yöntemdir.

Öncelikle, buradaki başlangıç kodunu yeni sekmede açınız. Yeni sekme, React kodu ile birlikte boş bir tic-tac-toe oyununu görüntülüyor olacaktır. Bu öğreticide React kodunu düzenliyor olacağız.

Bu seçeneği tercih ediyorsanız, ikinci seçeneği es geçebilir, ve Genel bakış bölümüne giderek genel bilgi edinebilirsiniz.

### Kurulum Seçeneği 2: Yerel Geliştirme Ortamı

Bu seçenek tamamen isteğe bağlıdır ve bu öğreticiyi takip etmek için zorunlu değildir.

Takıldım, Yardım Edin!

Eğer bu öğreticiyi takip ederken herhangi bir yerde takıldıysanız, topluluk destek kaynaklarına bakınız. Özellikle Discord'da yer alan Reactiflux Chat kanalı, hızlıca yardım almak için oldukça elverişlidir. Eğer bir cevap alamadıysanız veya hala takıldığınız için devam edemiyorsanız lütfen bize GitHub üzerinden issue açınız. Devamında size yardımcı olacağız.

## Genel Bakış

Kurulumu tamamladığınıza göre artık, React'e giriş yapabiliriz.

## React Nedir?

React, kullanıcı arayüzleri oluşturmak için açık, verimli ve esnek bir JavaScript kütüphanesidir.

Component (bileşen) denilen küçük ve izole parçalar sayesinde karmaşık arayüz birimlerini oluşturmanıza olanak tanır.

React'te birkaç tipte bileşen bulunmaktadır. Fakat şimdilik React.Component'e değinelim:

```
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>  
  
    );  
  
  }  
  
}
```

// Örnek kullanım: <ShoppingList name="Mark" />

Birazdan yukarıda kullandığımız XML-tarzı etiketlere değineceğiz. React bileşenleri sayesinde, ekranda görmek istediğimiz arayüz birimlerini React'e belirtmiş oluyoruz. Verilerimiz değiştiği zaman React, etkili bir şekilde bileşenlerimizi güncelleyecek ve tekrar render edecektir (arayüze işleyecektir).

Buradaki ShoppingList, React bileşen sınıfıdır, veya React bileşen tipidir diyebiliriz. Bir React bileşeni, özellikler anlamına gelen “properties”’in kısaltması olan props isimli parametreleri alır, ve arayüzü görüntülemek amacıyla render metodundan geriye bir görünüm hiyerarşisi (XML kodu) döndürür.

render metodu, ekranda neyi görüntülemek istiyorsanız onunla ilgili bir tanımlama geri döndürür. React de bu tanımlamayı alır ve görüntüler. Bilhassa render metodu, neyi render edeceği ile ilgili bir tanımlama olan React elemanı (React element) geri döndürür. Birçok React geliştiricisi, bu tanımlamaları kolayca kodlamak için “JSX” denilen özel bir kod dili kullanır. <div /> içeriği bir JSX kodu teşkil eder. Bu kod derlendiğinde, React.createElement('div') şeklinde bir JavaScript metod çağrısına dönüştürülür. Üstteki

örneğin derlenmiş hali aşağıdaki gibidir:

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

Eğer createElement() fonksiyonu hakkında daha fazla bilgi almak istiyorsanız API dokümanını inceleyebilirsiniz. Fakat bu öğreticide createElement() fonksiyonunu kullanmayacağız. Bunun yerine daha basit ve okunaklı olan JSX gösterimini ele alacağız.

JSX, JavaScript'in bütün gücünü kullanacak şekilde tasarlanmıştır. Bu sayede, JSX kodu içerisinde süslü parantezler kullanarak herhangi bir JavaScript kodunu çalıştırabilirsiniz. Her React elemanı bir JavaScript nesnesi olduğu için, herhangi bir değişkene atayabilir veya uygulama içerisinde herhangi bir yere koyabilirsiniz.

Yukarıdaki ShoppingList bileşeni, yalnızca <div /> ve <li /> gibi HTML bileşenlerini render eder. Fakat uygulamanıza özel React bileşenleri oluşturarak, toplu halde render edilmesini sağlayabilirsiniz. Örneğin sadece <ShoppingList /> yazarak bütün alışveriş listesinin görüntülenmesini sağlayabilirsiniz. Her React bileşeni birbirinden izole edilmiştir ve birbirinden bağımsız olarak çalışabilir. Bu sayede basit bileşenleri bir araya getirerek karmaşık kullanıcı arayüzleri oluşturabilirsiniz.

## Başlangıç Kodunun İncelenmesi

Eğer bu öğreticiyi tarayıcınızda takip ediyorsanız, başlangıç kodunu yeni sekmede açınız. Eğer öğreticiyi yerel makinenizde takip ediyorsanız, bunun yerine proje dizininde yer alan src/index.js dosyasını açınız kurulum aşamasında bu dosyaya değinmiştik).

Bu başlangıç kodu, yapacağımız proje ile ilgili bir temel niteliğindedir. tic-tac-toe oyununu programlayarak React öğreniminize yoğunlaşabilmeniz için size CSS kodlarını hazır olarak sunduk. Bu nedenle öğretici boyunca CSS kodu yazmanız gerekli değildir.

Kodu incelediğinizde aşağıdaki 3 React bileşenini fark edeceksiniz:

- Square (Kare)
- Board (Tahta)
- Game (Oyun)

Şu an Square bileşeni yalnızca bir adet <button> elemanını, Board ise 9 adet Square bileşenini render ediyor. Game bileşeni ise Board bileşenini ve daha sonra değiştireceğimiz kısa bir metni render ediyor. Henüz uygulama içerisinde etkileşimli bir bileşen yer almıyor.

## Prop'lar Aracılığıyla Veri Aktarımı

Şimdi işe koyulalım ve Board bileşeninden Square bileşenimize bazı verileri göndermeyi deneyelim.

Öğretici üzerinde çalışırken ve kopyala / yapıştır yerine, kodu elle yazmanızı şiddetle öneriyoruz. Bu, kas hafızanızı ve daha güçlü bir kavrayış geliştirmenize yardımcı olacaktır.

Board bileşeninin renderSquare metodunda, value prop'unu Square bileşenine gönderecek şekilde kodu değiştirelim:

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

Square bileşeninin render metodunu, ilgili değeri göstermesi için `/* TODO */` kısmını `{this.props.value}` şeklinde değiştirelim:

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

Öncesi:

Next player: X


Sonrası: Eğer değişiklikleri doğru bir şekilde uyguladıysanız render işlemi bitiminde her kare içerisinde bir sayı görüyor olmalısınız.

Next player: X

<b>0</b>	<b>1</b>	<b>2</b>
<b>3</b>	<b>4</b>	<b>5</b>
<b>6</b>	<b>7</b>	<b>8</b>

kodun tamamı :

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

```
class Board extends React.Component {  
  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>
    );
  }
}
```

```

class Game extends React.Component {
  render() {
    return (
      <div className="game">
        <div className="game-board">
          <Board />
        </div>
        <div className="game-info">
          <div>{/* status */</div>
          <ol>{/* TODO */</ol>
        </div>
      </div>
    );
  }
}

// =====

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<Game />);

```

### Etkileşimli bir Bileşen Yapımı

Haydi şimdi Square bileşenine tıkladığımızda içeriğini “X” ile dolduralım. Öncelikle, Square bileşeninin render() fonksiyonundan dönen button etiketini bu şekilde değiştirelim:

```

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

```



```
}  
  
}
```

Şimdi herhangi bir kareye tıkladığımızda, tarayıcınızın devtools konsolunda bir alert mesajı görüntülenecektir.

Not:

Daha az kod yazmak ve this'in kafa karıştırıcı kullanımından kaçınmak için, butona tıklanması gibi olay fonksiyonlarında, arrow function kullanacağız:

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

Farkedeceğiniz üzere, `onClick={() => console.log('click')}` kısmında butonun `onClick` prop'una bir fonksiyon ataması gerçekleştiriyoruz. Bu fonksiyon sadece butona tıkladığımızda çalışıyor. Genellikle bir yazılımcı hatası olarak parantezli ok `() =>` ifadesinin unutulması yerine direkt olarak `onClick={console.log('click')}` ifadesinin yazılması gerçekleşebiliyor. Bu durumda tıklama anında gerçekleşmesi istenen olay yanlış bir şekilde çalışarak, bileşen tekrar render edildiğinde gerçekleşmiş oluyor.

Sonraki adımda Square bileşeninin, tıkladığı durumu “hatırlamasını” ve “X” işareti ile doldurulmasını sağlayacağız. Bir şeyleri “hatırlamak” için bileşenler state (durum)’u kullanırlar.

React bileşenleri, constructor (yapıcı) fonksiyonlarında `this.state`’e atama yaparak bir state’e sahip olurlar. React bileşeni içerisinde tanımlanan `this.state` özelliğinin erişim belirleyicisi `private` olarak düşünülmelidir. Çünkü sadece o bileşene özeldir ve diğer bileşenler tarafından direkt olarak erişilemezler.

Şimdi Square’in mevcut değerini `this.state` içerisinde saklayalım ve Square’e tıkladığında değiştirelim. Bunun için öncelikle Square sınıfına bir constructor ekleyeceğiz ve içerisinde state’in başlangıç değerlerini oluşturacağız:

```
class Square extends React.Component {  
  
  constructor(props) {  
  
    super(props);
```

```
this.state = {  
  value: null,  
};  
}  
  
render() {  
  return (  
    <button className="square" onClick={() => console.log('click')}>  
      {this.props.value}  
    </button>  
  );  
}  
}
```

Not:

JavaScript class'larında, alt sınıfın constructor'ını oluştururken her zaman super fonksiyonunu çağırmanız gerekmektedir. constructor metodu olan her bir React sınıf bileşeninde, constructor metodu super(props) çağırısı ile başlamalıdır.

Şimdi, Square'e tıklanıldığında, state'indeki value değerinin render metodunda görüntülenebilmesi için aşağıdaki adımları izleyelim:

<button> etiketi içerisinde yer alan this.props.value yerine this.state.value yazalım.

() => alert() event handler'ını () => this.setState({value: 'X'}) ile değiştirelim.

Okunabilirliği arttırmak için className ve onClick prop'larını ayrı satırlara alalım.

Bu değişikliklerden sonra Square'in render metodundan dönen <button> etiketi aşağıdaki gibi görüntülenecektir:

```
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>  
  );  
}  
}
```

Square'ın render metodundaki onClick metodundan, this.setState'in çağrılmasını sağladık. Bu sayede Square'deki <button> elemanına her tıklandığında React, Square bileşenini tekrar render edecektir. Güncelleme sonrasında Square'ın this.state.value değerine 'X' ataması gerçekleşecektir, ve bu sayede oyun tahtasında 'X'i göreceğiz. Herhangi bir Square bileşenine tıklandığı anda içerisinde 'X' görüntülenecektir.

Bir bileşenteki setState fonksiyonunu çağırdığınızda, React otomatik olarak içerisindeki alt bileşenleri de güncellemiş oluyor.

kodun tamamı:

```
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>  
  );  
}
```

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

```
class Game extends React.Component {  
  render() {  
    return (  
      <div className="game">  
        <div className="game-board">  
          <Board />  
        </div>  
        <div className="game-info">
```

```

    <div>{/* status */}</div>

    <ol>{/* TODO */}</ol>

  </div>

</div>

);

}

}

// =====

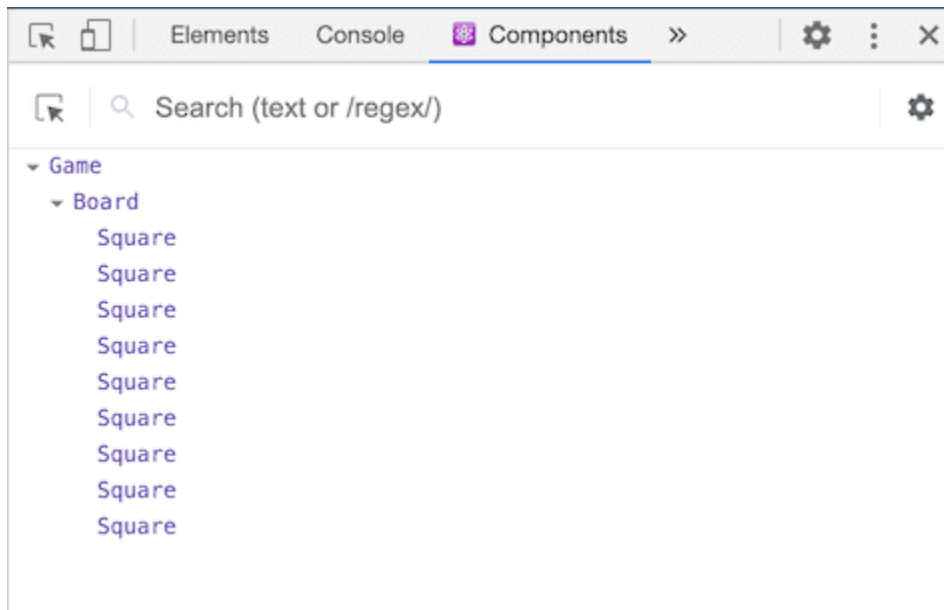
const root = ReactDOM.createRoot(document.getElementById("root"));

root.render(<Game />);

```


## Geliştirici Araçları

Chrome ve Firefox için React Devtools eklentisi sayesinde herhangi bir React bileşen ağacını, tarayıcınızın geliştirici araçları kısmından görüntüleyebilirsiniz.



React DevTools, React bileşenlerinizin state'ini ve prop'larını kontrol etmenize olanak tanır.

React DevTools kurulumundan sonra, sayfa içerisindeki herhangi bir elemana sağ tıklayarak çıkan menüde "İncele"'yi seçerseniz, geliştirici araçlarını açabilir, ve devamında en sağda yer alan React sekmesinde ("🔧 Components" and "🔧 Profiler") incelemelerinizi yürütebilirsiniz. Bileşen ağacını

incelemek için “ Components” tabına bakın.

Eklentinin CodePen ile çalışabilmesi için harici olarak birkaç adım daha vardır:

CodePen’e e-posta adresiniz ile giriş yapınız veya kayıt olunuz (spam’lerin engellenmesi için gereklidir).

“Fork” butonuna basınız.

“Change View”’a tıklayarak devamında “Debug mode”’u seçiniz.

Açılan yeni sekmede, Devtools içerisinde React sekmesi yer alacaktır.

## Oyunun Tamamlanması

Artık tic-tac-toe oyunumuz için temel kod bloklarına sahibiz. Oyunun tamamlanması için tahta üzerinde “X” ve “O”’ların birbiri ardına yerleştirilmesi gerekiyor. Sonrasında oyunda bir kazananın belirlenmesi için değişiklikler yapılmasına ihtiyaç var.

## State’in Yukarı Taşınması

Şu an her bir Square bileşeni oyunun state’ini değiştirebiliyor. Kazananı belirleyebilmemiz için, 9 square’in de değerine ihtiyacımız var.

Bunu gerçekleştirmek için Board’un, her bir Square’e, kendi state’inin ne olduğunu sorması gerektiğini düşünebiliriz. Bu yöntem her ne kadar React’te uygulanabilir olsa da, yapmanızı tavsiye etmiyoruz. Çünkü bu şekilde yazılan kod; anlaşılabilirlikten uzak olacak, hataların oluşmasına daha müsait hale gelecek ve kodu refactor etmek istediğimizde bize çok daha büyük zorluklar çıkaracaktır. Bu nedenle, her bir Square sınıfında, kendi state’inin tutulmasının yerine, üst bileşen olan Board bileşeninde oyunun tüm state’ini tutmak en iyi çözümdür. Bunun sonucunda Board bileşeni, her bir Square’e neyi göstermesi gerektiğini prop’lar aracılığıyla aktarır (daha önce de prop’lar aracılığıyla her bir Square’e bir sayı atamıştık).

Bu örnekteki gibi, birçok alt bileşenden verilerin toplanması veya iki çocuğun birbirleri arasında iletişim kurabilmesi için, üst bileşende paylaşımlı bir state oluşturmanız gerekmektedir. Üst bileşen, prop’lar aracılığıyla state’ini alt bileşenlere aktarabilir. Bu sayede alt bileşenler hem birbirleri arasında hem de üst bileşen ile senkronize hale gelirler.

React bileşenleri refactor edilirken, state’in yukarı taşınması çok yaygın bir durumdur. Şimdi bu fırsatı değerlendirelim ve işe koyulalım.

Board’a bir constructor ekleyelim ve Board’un başlangıç state’ine bir dizi atayarak içerisinde 9 adet null değerinin bulunmasını sağlayalım. 9 kareye, 9 adet null karşılık gelecektir:

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

```
};  
}
```

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

Daha sonra Board’u doldurdukça, this.state.squares dizisinin içeriği aşağıdaki gibi görünmeye başlayacaktır:

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

Board’un renderSquare metodu aşağıdaki gibi görünüyor:

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

Projeye başladığımızda, 0’dan 8’e kadar olan sayıları her bir karede göstermek için, Board’daki value prop’unu alt bileşenlere aktarmıştık. Bir diğer önceki aşamada ise sayıların yerine mevcut Square bileşeninin kendi state’i tarafından belirlenen “X” işaretinin almasını sağlamıştık. İşte bu nedenle Square bileşeni, Board tarafından kendisine gönderilen value prop’unu göz ardı ediyor.

Şimdi prop aktarma mekanizmasını tekrar kullanacağız. Bunun için her bir Square’e kendi mevcut değerini ('X', 'O', or null) atamak için Board bileşeninde değişiklik yapalım. Board’un constructor’ında halihazırda tanımladığımız bir squares dizisi bulunuyor. Board’un renderSquare metodunu, bu diziden verileri alacak şekilde değiştirelim:

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

Kodun bu kısma kadar olan son halini görüntülemek için tıklayınız

Artık her bir Square bileşeni value prop’unu alacak ve ‘X’, ‘O’ veya boş square’ler için null değerini edinecektir.



Şimdi Square'e tıklandığında ne olacağına karar vermemiz gerekiyor. Board bileşeni artık hangi Square'in doldurulacağına karar verebildiğine göre, Square'e tıklandığında Board bileşeninin state'inin güncellenmesini sağlamalıyız. State her bir bileşene private olduğundan dolayı Square üzerinden direkt olarak Board'un state'ini değiştiremeyiz.

Board'un state'inin gizliliğini korumak için, Board'dan Square'e bir fonksiyon aktarmamız gerekiyor. Square'e her tıklama anında bu fonksiyonun otomatik olarak çağrısı gerçekleşecektir. Şimdi Board'un renderSquare metodunu aşağıdaki şekilde değiştirelim:

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

Not:

Kodun okunabilirliği için geri dönüş elemanını birçok satıra böldük ve parantezler ekledik. Bu sayede JavaScript, return'den sonra otomatik olarak bir noktalı virgül eklemeyecek ve bundan dolayı kodun bozulması engellenmiş hale gelecektir.

Artık Board'dan Square'e, value ve onClick olmak üzere iki tane prop gönderiyoruz. Square'e tıklandığında ise prop olarak gelen onClick fonksiyonu çağrılmasına ihtiyacımız var. Bunun için Square'e aşağıdaki değişiklikleri uygulamamız gerekiyor:

Square'in render metodu içerisinde yer alan this.state.value yerine this.props.value yazınız.

Yine Square'in render metodundaki this.setState() yerine this.props.onClick() yazınız.

Square artık oyunun state'ini değiştirmeyeceği için, Square'in constructor metodunu siliniz.

Bu değişikliklerin ardından, Square bileşeni aşağıdaki gibi görüntülenecektir:

```
class Square extends React.Component {  
  
  render() {  
  
    return (  
  
      <button
```

```
    className="square"

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

  >

    {this.props.value}

  </button>

);

}

}
```

Artık Square'e tıklandığında, Board tarafından aktarılan onClick fonksiyonu çağrılacaktır. Bunun nasıl gerçekleştiğini açıklayalım:

HTML'de varsayılan olan <button> bileşenin onClick prop'u React'e, tıklama olayını oluşturmasını söyler.

Butona tıklandığında React, Square'in render() metodunda tanımlanan onClick fonksiyonunu çalıştırır.

Bu fonksiyon ise, this.props.onClick() çağrısını gerçekleştirir. Square'in onClick prop'u, Board tarafından kendisine aktarılmıştır.

Board, Square'e onClick={() => this.handleClick(i)} kodunu aktardığı için, Square'e tıklandığında Board'un this.handleClick(i) metodu çağrılır.

Şu an handleClick() metodunu oluşturmadığımız için kodumuz hata verecektir. Şimdi bir kareye tıklarsanız, "this.handleClick is not a function" gibi bir şey söyleyen kırmızı bir hata ekranı görmelisiniz.

Not:

HTML'deki <button> elemanı varsayılan bileşen olarak geldiği için, onClick fonksiyonu, React için özel bir anlam ihtiva eder. Fakat Square gibi özel olarak yazılan bileşenlerde, prop isimlendirmesi size kalmıştır. Bu nedenle Square'in onClick prop'unu veya Board'un handleClick metodunu daha farklı şekillerde isimlendirebilirsiniz. Ancak React'teki isimlendirme kuralına uymak gereklidir. Bu kural şu şekildedir: olayları temsil eden prop'lar için on[Olay], olayları handle eden metodlar için ise handle[Olay] ifadeleri kullanılır.

Square'e tıkladığımızda, handleClick'i tanımlamadığımız için hata aldığımızdan bahsetmiştik. Gelin şimdi Board sınıfına handleClick'i ekleyelim:

```
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>  
  );  
}
```

```
    </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>

);

}

}
```

Bu değişikliklerden sonra, oyundaki karelere tıkladığımızda içeriğinin “X” ile doluyor olduğunu tekrar görebiliyoruz. Fakat her bir Square’ın ayrı ayrı state’e sahip olması yerine, Board bileşeninde tek bir state barındırılmış hale geldi. Bu sayede Board’daki state değiştiğinde tüm Square bileşenleri otomatik olarak tekrar render edilecektir. Bunun yanında, bütün Square’lerin state’inin Board bileşeninde tutulması, gelecekte kazananı belirlememiz için önemli bir yöntem teşkil edecektir.

Square bileşenleri artık state’i direkt olarak değiştirmedeği için, değerleri Board bileşeninden alıyorlar ve tıkladıklarında Board’u haberdar ediyorlar. React terminolojisinde Square bileşenleri için controlled components (kontrol edilen bileşenler) adı verilir. Çünkü tüm kontrol Board bileşeninin elindedir.

Fark edeceğiniz gibi, handleClick fonksiyonu içerisinde, halihazırda var olan squares dizisini direkt olarak değiştirmek yerine, .slice()’ı kullanarak bir kopyasını oluşturduk ve bu kopyayı değiştirdik. Şimdi squares dizisinin neden bir kopyasını oluşturduğumuza değineceğiz.

## Neden Immutability Önemlidir

Immutability, anlam olarak mutate (değişmek) kelimesinin zıttı olan değişmezlik kavramını oluşturmaktadır. Önceki kod örneğinde, mevcut squares dizisini değiştirmek yerine, dizinin .slice() metodu ile bir kopyasının oluşturulması gerektiğini önermiştik. Şimdi ise immutability kavramına ve immutability’i öğrenmenin neden önemli olduğuna değineceğiz.

Genellikle verinin değiştirilmesi için iki farklı yaklaşım vardır. İlk yaklaşımda, verinin değerleri direkt olarak değiştirilerek ilgili verinin değişmesi (mutate) sağlanır. İkinci yaklaşımda ise, ilgili veri kopyalanarak, kopya

veri üzerinde istenen değişiklikler gerçekleştirildikten sonra, kopya verinin ana veriye atanması işlemidir.

### Mutasyon Kullanılarak Verinin Değiştirilmesi

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

player.score = 2;

// Oyuncu nesnesinin son hali: {score: 2, name: 'Jeff'}
```

### Mutasyon Kullanılmadan Verinin Değiştirilmesi

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

var newPlayer = Object.assign({}, player, {score: 2});

// Şu an oyuncu nesnesi değişmedi, fakat oyuncu nesnesinden yeniOyuncu nesnesi oluşturuldu: {score: 2, name: 'Jeff'}

// Object spread sözdizimini kullanarak aşağıdaki gibi de yazabilirsiniz:

// var yeniOyuncu = {...player, score: 2};
```

Sonuç iki durumda da aynı oldu ama direkt olarak veriyi değiştirmeden kopya üzerinde değişiklikler yapmanın aşağıdaki gibi birçok yararı vardır.

### Karmaşık Özellikleri Basit Hale Getirir

Immutability sayesinde karmaşık özellikler kodlamak çok daha kolaydır. Bu öğreticinin sonunda, tic-tac-toe oyunundaki hamlelerin geçmişini incelemeyi ve önceki hamlelere geri dönmeyi sağlayan “zaman yolculuğu” özelliğini kodlayacağız. Bu özellik sadece oyunlarda değil, birçok uygulamada ileri ve geri alma işlemlerinin kurgulanması için bir gereksinim teşkil edebilir. Direkt olarak veri mutasyonundan kaçınarak, oyunun önceki versiyonlarını oyun geçmişinde bozmadan tutabilir ve daha sonra, önceki bir versiyona geri dönmeyi sağlayabilirsiniz.

### Değişikliklerin Tespit Edilmesini Kolaylaştırır

Mutable nesneler, direkt olarak değiştirilebildikleri için, değişip/değişmediklerinin tespit edilmesi güçtür. Değişikliğin tespit edilebilmesi için, nesnenin kendisi ile önceki kopyalarının karşılaştırılması ve bütün nesne ağacı üzerinde gezilmesi gereklidir.

Immutable nesnelerdeki değişikliklerin tespit edilmesi daha kolaydır. Immutable nesne kopyalanarak ataması yapıldığı için, ilgili değişken, öncekinden farklı bir değişkene referans edilmişse o halde nesne değişmiştir diyebiliriz.

### Tekrar Render Etme Zamanını Belirlemek

React'te Immutability'nin ana faydası ise, pure component'ler (saf/katkısız bileşenler) yapmayı kolaylaştırmasıdır. Immutable veriler, değişiklik yapıldığını kolayca tespit edebilirler. Bu sayede değişiklik olduğunda ilgili bileşenin tekrar render edilmesine yardımcı olurlar.

Performansın iyileştirmesi yazısında `shouldComponentUpdate()` fonksiyonunun ne olduğuna ve nasıl pure component'leri oluşturabileceğiniz hakkında bilgi edinebilirsiniz.

## Fonksiyon bileşenleri

Square bileşenini nasıl fonksiyon bileşeni haline getireceğimize değinelim.

React'te fonksiyon bileşenleri, sadece render metodunu içerirler. İçerisinde herhangi bir state bulundurmadıkları için daha kolay bir şekilde bileşen oluşturmayı sağlarlar. `React.Component`'tan türetilen bir sınıf bileşeni oluşturmak yerine, sadece propları girdi olarak alan ve render edilecek kısımları döndüren bir fonksiyon bileşeni yazabiliriz. Fonksiyon bileşenleri kısa bir şekilde yazıldığı için, sizi sınıf bileşenlerine göre daha az yorar.

Square sınıfını aşağıdaki fonksiyon ile değiştirelim:

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

Dikkat edecek olursanız sınıf bileşeninde kullandığımız `this.props` ifadesi yerine sadece `props`'u kullandık.

Not:

Square'i, fonksiyon bileşeni olarak değiştirdiğimiz için, uzun olan `onClick={() => this.props.onClick()}` kod parçasını, `onClick={props.onClick}` şeklinde yazarak daha kısa hale getirmiş olduk (her iki taraftaki parantezlerin de gittiğine dikkat ediniz). Sınıf bileşeninde gerçek `this` değerine ulaşmak için arrow (ok) fonksiyonu kullanmıştık. Bunun aksine fonksiyon bileşenlerinde `this` ile uğraşmanıza gerek yoktur.

## Hamle Sırası Değişikliği

Şimdi tic-tac-toe oyunumuzdaki hatayı çözmemiz gerekiyor. Oyunun son hali ile sadece "X" eklenebiliyor ama "O" eklenemiyor.

Oyuna varsayılan olarak "X" başlıyor. X'in ilk başlayıp/başlamayacağını `Board`'un constructor'ındaki başlangıç state'inde belirleyebiliriz:

```
class Board extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {
```

```
squares: Array(9).fill(null),  
  
  xIsNext: true,  
  
};  
  
}
```

Herhangi bir oyuncu hamlesini yaptığında `xIsNext` (`xSonrakiElemanMı`) boolean değişkeninin tersini alarak hangi oyuncunun sonraki hamleyi yapacağını belirleyebiliriz. Ayrıca oyunun state'inde bu değişkeni kaydedebiliriz. `Board`'un `handleClick` fonksiyonunu, `xIsNext` değişkeninin tersini alacak şekilde ilgili değişikliği yapalım:

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

Bu değişiklik ile sayesinde, “X”’ler ve “O”’lar sırasıyla hamle yapabiliyor olacaklar.

Ayrıca oyunda, sıradaki hamlenin kimde olduğunu gösteren metni değiştirmek için, `Board`'un `render` metodunda “status” değişkenini oluşturabiliriz:

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

Bu değişikliklerden sonra `Board` bileşeninin son hali aşağıdaki gibi olacaktır:

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

```
    xlsNext: true,  
  };  
}
```

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

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

```
render() {  
  const status = 'Next player: ' + (this.state.xlsNext ? '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>

);
}
}
```

## Kazananın Belirlenmesi

Artık sonraki oyuncuyu görüntüleyebiliyoruz. Bundan sonraki amacımız olarak, oyunun bitmesi durumunu belirtmek için, oyunun kazanıldığını ve artık başka bir hamle kalmadığını göstermemiz gerekiyor. Bunun için, kazanan oyuncuyu belirtmek amacıyla, dosyanın sonuna yardımcı bir fonksiyon ekleyebiliriz:

```
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;
}

```

9 kareden oluşan bir dizi göz önüne alındığında, bu fonksiyon kazananı kontrol edecek ve uygun şekilde 'X', 'O' veya null döndürecektir.

Board'un render fonksiyonunda, calculateWinner(squares) fonksiyonunu çağırarak, ilgili oyuncunun kazanma durumunun kontrol edilmesini sağlayabiliriz. Hamleyi yapan oyuncu kazandıysa, "Winner: X" veya "Winner: O" gibi kazananı belirten bir metin görüntüleyebiliriz. Şimdi, Board'un render fonksiyonunda yer alan status değişkenini aşağıdaki şekilde değiştirelim:

```

render() {

  const winner = calculateWinner(this.state.squares);

  let status;

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

```

```
return (  
  
  // geriye kalan kısımlar değiştirilmedi
```

Oyunda farketmiyseniz bir oyuncu, diğeri oyuncunun işaretlediği karenin üstüne tekrar işaretleme yapabiliyor. Buna ek olarak oyun kazanıldığı durumda da tekrar işaretleme yapmayı engellemeliyiz. Bunun için Board'un handleClick fonksiyonunu, belirli koşullarda return edecek şekilde değiştirelim:

```
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,  
  
  });  
  
}
```

Tebrikler. Artık çalışan bir tic-tac-toe oyununuz var. Ayrıca bu kısma kadar React'in temel özelliklerini de öğrenmiş durumdasınız. Bu nedenle aslında gerçek kazanan sizsiniz.

## Zamanda Yolculuğun Eklenmesi

Son çalışma olarak, oyunda önceki hamlelere gitmeyi sağlayacak olan “zamanda geriye gitme” özelliğini ekleyelim.

## Hamlelerin Geçmişinin Saklanması

Eğer squares dizisine direkt olarak elle müdahale ederek değiştirseydik, zaman yolculuğu özelliğini geliştirmemiz daha zor olurdu.

Ancak, slice() fonksiyonu yardımıyla her hamleden sonra squares dizisinin kopyasını alarak immutable olarak değiştirilmesini sağladık. Bu durum bize, squares dizisinin geçmişteki her halinin kaydedebilmemize, ve halihazırda oluşan hamleler arasında gezinebilmemize imkan sağlamış oldu.

squares dizisinin geçmiş hallerini tutabilmek için history adında başka bir dizi oluşturabiliriz. history dizisi, oyunda ilk hamleden son hamleye kadar tahtanın tüm durumlarını barındırıyor olacaktır:

```
history = [
```

```
// ilk hamleden öncesi
{
  squares: [
    null, null, null,
    null, null, null,
    null, null, null,
  ]
},
// ilk hamleden sonrası
{
  squares: [
    null, null, null,
    null, 'X', null,
    null, null, null,
  ]
},
// ikinci hamleden sonrası
{
  squares: [
    null, null, null,
    null, 'X', null,
    null, null, 'O',
  ]
},
// ...
]
```

Şimdi history dizisinin, hangi bileşenin state'inde yer alması gerektiğine karar vereceğiz.

### **State'in Yukarı Taşınması (Tekrar)**

En üst seviyedeki Game bileşeninin, geçmiş hamlelerin listesini görüntülemesini istiyoruz. Bunun için, Game bileşeninin history'e erişebilmesi gerekiyor. Bunu sağlamanın yolu, history'i Game bileşenine taşımaktan geçiyor.

history state'ini, Game bileşenine yerleştireceğimiz için, bir alt bileşen olan Board'dan squares state'ini çıkarmamız gerekiyor. State'in Yukarı Taşınması bölümünde Square bileşeninden Board bileşenine taşıma yaptığımız gibi, şimdi de Board bileşeninden Game bileşenine taşıma işlemini gerçekleştirmemiz gerekiyor. Bu sayede Game bileşeni, Board'un verisi üzerinde tamamen kontrolü ele almış olacak ve history'deki önceki hamlelerin Board'a işlemlerini bildirebilecektir.

Öncelikle, Game bileşeninin constructor'ında, state'in ilk halini oluşturmamız gerekiyor:

```
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>

    );
  }
}
```

Şimdi squares dizisini ve onClick event'ini, prop'lar aracılığıyla Game bileşeninden, Board bileşenine aktarmamız gerekiyor. Birden fazla Square için Board'da sadece bir tane click handler'ı bulunduğundan dolayı, tıklanan square'in hangisi olduğunun belirlenebilmesi için, onClick handler'ına her bir Square'in konumunu iletmemiz gerekiyor. Bu gereksinimler için Board bileşenini aşağıdaki gibi değiştirebilirsiniz:

Board'daki constructor'ı siliniz.

Board'un renderSquare metodunda this.state.squares[i] yerine this.props.squares[i] yazınız.

Board'un renderSquare metodunda this.handleClick(i) yerine this.props.onClick(i) yazınız.

Board'un son hali aşağıdaki gibi olmalıdır:

```
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>
  );
}
```

```
        </div>

        <div className="board-row">

            {this.renderSquare(6)}

            {this.renderSquare(7)}

            {this.renderSquare(8)}

        </div>

    </div>

    );
}
}
```

Şimdi de oyun geçmişindeki son girdiyi kullanarak, oyunun son durumunun belirlenmesi ve görüntülenmesi için, Game bileşenindeki render fonksiyonunu değiştirelim:

```
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={() => this.handleClick(i)}
      />
    </div>

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

      <ol>{/* TODO */}</ol>

    </div>
  </div>
);
}
```

Oyunun durumunu Game bileşeni render ettiği için, Board'daki render metodundan oyunun durumunu ilgilendiren kısımları çıkarabiliriz:

```
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)}
      </div>
    </div>
  );
}
```

```
    {this.renderSquare(8)}  
  </div>  
</div>  
);  
}
```

Son olarak, Board bileşenindeki handleClick metodunu Game bileşenine taşıyacağız. Ayrıca, Game bileşeni Board'a göre daha farklı oluşturulduğu için, handleClick metodunu da uygun şekilde değiştirmemiz gerekiyor. Bunun için Game'in handleClick metodu içerisinde, oyundaki hamleleri history dizisine ekleyeceğiz:

```
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,  
  });  
}
```

Not

Bir diziye eleman eklemek için, genellikle dizinin push() metodu kullanılır. Fakat push()'un aksine concat() metodu orijinal diziyi değiştirmez. Bu nedenle immutability'nin sağlanması için concat() fonksiyonunun kullanılması önem teşkil etmektedir.

Geldiğimiz noktada, Board bileşeni sadece renderSquare ve render metodlarına ihtiyaç duyuyor. Oyunun durumu ve handleClick metodu ise artık Game bileşeninde yer alıyor.

## Geçmiş Hamlelerin Görüntülenmesi

tic-tac-toe oyununun geçmişini kaydedebildiğimize göre, artık oyuncuya geçmiş hamlelerin görüntülenmesini sağlayabiliriz.

Daha önce React elemanlarının, birinci kalite JavaScript nesneleri olduğunu öğrenmiştik. Bu sayede React elemanlarını, uygulama içerisinde istediğimiz yere aktarabiliyoruz. Bu nedenle JavaScript mantığıyla düşündüğümüzde, React'te birden fazla elemanı render edebilmek için, React elemanlarından oluşan bir diziye kullanabiliriz.

JavaScript'te diziler bir map() (harita) metodu içerirler. Bu metod sayesinde verileri istenilen şekilde haritalayabilirler. Örneğin 1, 2, 3 sayılarının, iki katını alan bir dizinin oluşturulmasını sağlayabilirler:

```
const numbers = [1, 2, 3];
```

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

map metodunu kullanarak oyunun hamle geçmişini, ekranda butonlar halinde görüntülemek için React elemanlarına map edebiliriz. Ve bu butonlara tıklayarak geçmiş hamlelere atlanmasını sağlayabiliriz.

Game'in render metodunda yer alan history diziyi üzerinde map fonksiyonunun çalıştırılmasını sağlayalım:

```
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>
);
}

```

history dizisinin içinde yineleme yaptığımız için, step değişkeni mevcut history elemanının değerini, move ise geçerli history elemanının dizinini (index) ifade eder. Burada sadece move ile ilgilendiğimiz için step değişkeni hiçbir şeye atanmıyor.

tic-tac-toe oyununun geçmişindeki her bir hamle için, <button> içeren bir <li> elemanı oluşturuyoruz. Butondaki onClick metodu, üzerine tıklandığında this.jumpTo() fonksiyonunu çağırıyor fakat, henüz

jumpTo() metodunu oluşturmamak. Şu an, oyun içerisinde oluşan hamlelerin bir listesini görüyor olmanız lazım. Ayrıca geliştirici araçları konsolunda da aşağıdaki şekilde bir uyarı vermiş olmalıdır:

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

Üstteki uyarının ne anlama geldiğine bakalım.

## Key seçimi

Bir liste görüntülediğimizde React, render edilen her bir liste elemanı için bazı bilgileri saklar. Listeyi güncellediğimizde ise listede neyin değiştiğine karar vermesi gerekir. Çünkü listenin elemanlarını eklemiş, silmiş, tekrar düzenlemiş veya güncellemiş olabiliriz.

Örnek olarak bir listenin kodlarının bu şekilde olup:

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

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

bu koda değiştiğini düşünelim:

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

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

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

Bu iki kodu okuyan bir kişi, sayıların değişmesine ek olarak Alexa ile Ben'in sıralamasının değiştiğini, ve araya Claudia'nın eklendiğini farkedecektir. Ancak React bir bilgisayar programıdır, ve amacımızın ne olduğunu kestiremez. React uygulamada listeyi değiştirmemizdeki maksadımızın ne olduğunu bilemeyeceğinden dolayı, her liste elemanını birbirinden ayırt etmek için, liste elemanlarına bir key (anahtar değer) vermemiz gerekir. Bu örnekte, alexa, ben, claudia isimlerini key olarak kullanabiliriz. Fakat bu verileri veritabanından getirseydik key olarak; Alexa, Ben, ve Claudia'nın ID'lerini kullanabilirdik:

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

Bir liste tekrar render edileceği zaman React, her liste elemanının key'ini alır ve önceki listenin elemanlarıyla karşılaştırır. Eğer yeni listede, önceki listede bulunmayan bir key varsa, React bir <li> bileşeni oluşturur. Eğer önceki listede bulunan bir key, yeni listede bulunmuyorsa React, ilgili <li>'yi yok eder. Eğer iki key eşleşiyorsa, eski liste elemanı yeni listeye taşınır. Render etme aşamaları arasında, state'in korunması amacıyla key'ler, her bir bileşenin kimliği hakkında React'e bilgi sunar. Bu sayede eğer bir bileşenin key'i değiştiyse, ilgili bileşen React tarafından yok edilir ve yeni bir state ile tekrar oluşturulur.

React'teki key kelimesi özeldir ve React içerisinde rezerve edilmiş kelimeler arasındadır (ref de rezerve edilmiştir, fakat daha gelişmiş bir özelliktir). Bir eleman oluşturulduğunda React, elemanın key özelliğini alır ve direkt olarak return edilen elemanın üzerinde saklar. key, props'a ait gibi görünse de, this.props.key kullanılarak erişilemez. Çünkü key özelliği, React'in otomatik olarak hangi bileşeni güncelleyeceğine karar vermesi için tasarlanmıştır. Bu nedenle props bir bileşenin, key'i hakkında bilgi edinemez.

Dinamik listeler oluştururken, benzersiz key değerleri atamanız kesinlikle tavsiye edilir. Eğer uygun key değerine sahip değilseniz, verinizi gözden geçirerek uygun bir id değerini bulmak mantıklı olacaktır.

Eğer bir key ataması yapmazsanız React, ekranda bir uyarı görüntüler ve varsayılan olarak ilgili liste elemanının index'ini key olarak kullanır. Dizinin indeksini key olarak kullanmak, liste elemanlarına ekleme/çıkarma veya tekrar sıralama yapılırken problem oluşturabilir. key={i} ataması yapmak uyarının susturulmasını sağlar fakat dizi indeksleri üzerindeki problemi gidermiş olmaz. Bu nedenle birçok durum için bu kullanım önerilmez.

Key'lerin uygulama içerisinde global olarak benzersiz olmasına gerek yoktur. Sadece bulunduğu bileşenin içerisinde yer alan diğer list elemanları arasında benzersiz olması yeterlidir.

### Zaman Yolculuğunun Kodlanması

tic-tac-toe oyununun geçmişinde, her bir geçmiş hamlenin benzersiz bir ID'si bulunmaktadır. Bu ID'ler, ardışık hamle sayılarından oluşurlar. Hamleler asla silinmezler, ortadan eklenmezler ve tekrar sıralanmazlar. Bu nedenle key olarak hamle index'inin kullanılması bu durum için uygundur.

Game bileşenindeki render metoduna <li key={move}> olacak şekilde key'imizi ekleyelim ve bu sayede React'in key hakkındaki uyarısını kaldıralım:

```
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>  
  
  );  
  
});
```

Liste elemanlarındaki butonlara tıklamak, jumpTo metodu bulunmadığı için bir hata oluşturur. jumpTo'yu kodlamadan önce, mevcut durumda hangi adımın görüntülendiğini belirtmek için Game bileşeninin state'ine stepNumber değişkenini eklememiz gerekiyor.

Game'in constructor'ındaki başlangıç state'ine stepNumber: 0'ı ekleyelim:

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

```
this.state = {  
  history: [{  
    squares: Array(9).fill(null),  
  }],  
  stepNumber: 0,  
  xIsNext: true,  
};  
}
```

Sonra, Game'in içerisinde stepNumber değişkenini güncelleyecek olan jumpTo metodunu oluşturalım. Ayrıca değiştirdiğimiz stepNumber değişkeni çift ise xIsNext değişkenine true'yu atayalım:

```
handleClick(i) {  
  // Bu metot değişmedi  
}  
  
jumpTo(step) {  
  this.setState({  
    stepNumber: step,  
    xIsNext: (step % 2) === 0,  
  });  
}  
  
render() {  
  // Bu metot değişmedi  
}
```

jumpTo metodunda dikkat edilmesi gereken, state'in history özelliğini güncellemedik. Bunun nedeni, state güncellemelerinin birleştirilmesi veya daha basit bir deyişle React'ın yalnızca setState yönteminde belirtilen özellikleri güncellemesi ve kalan durumu olduğu gibi bırakmasıdır. Daha fazla bilgi için dokümanın geri kalanına bakınız.

Şimdi, oyundaki bir kareye tıklandığında çağrılan handleClick metodunda birkaç değişiklik yapalım.

Artık eklediğimiz `stepNumber` değişkeni, kullanıcının mevcut hamlesini gösteriyor. Yeni bir hamle yaptıktan sonra, `stepNumber` değerini güncellememiz için `this.setState()` çağrısına `stepNumber: history.length`'i eklememiz gerekiyor. Bu sayede, yeni bir hamle yapıldıktan sonra, sürekli aynı hamleyi görüntülemekten dolayı oluşan takılmayı engellemiş oluyoruz.

Ayrıca oyun geçmişine atama yapmak için `this.state.history` yerine `this.state.history.slice(0, this.state.stepNumber + 1)` yazacağız. Bu sayede, “zamanda geriye döndüğümüzde” o noktadan devam edebileceğiz, ve gelecekte yaptığımız hamleler işe yaramaz hale geleceğinden dolayı bu hamlelerin de `slice()` ile oyun tahtasından atılmasını sağlamış olacağız:

```
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,  
  
  });  
  
}
```

Son olarak, `Game` bileşeninin `render` metodunda, her zaman yapılan son hamlenin render edilmesi yerine, `stepNumber`'a göre mevcut seçilen hamlenin render edilmesini sağlayacağız:

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



// Kalan kısımlar değişmedi

Oyun geçmişinde herhangi bir adıma tıkladığımızda, tic-tac-toe tahtası o adım bittikten sonraki halini alacak şekilde anında güncellenecektir.

Sonuç Olarak

Tebrikler, bir tic-tac-toe oyunu kodladınız. Bu oyun:

Kendisinden de bekleyeceğiniz gibi bir tic-tac-toe oynamanızı sağlar,

Bir oyuncu kazandığında bunu gösterir,

Oyun ilerledikçe oyun geçmişini saklar,

Oyunculara oyun geçmişini görüntüleyebilmelerini ve oyun tahtasının önceki versiyonlarına gidebilmelerini sağlar.

İyi iş çıkardınız. Umarız artık React'in nasıl çalıştığını öğrenmişsinizdir.

Eğer biraz daha boş vaktiniz varsa ve yeni edindiğiniz React yetenekleriniz ile ilgili pratik yapmak istiyorsanız, aşağıda zorluk derecesine göre sıralanmış işler sayesinde, tic-tac-toe oyununuzu geliştirerek daha ileriye götürebilirsiniz:

Oyun geçmişinde, her hamlenin konumunun “(sıra,sütun)” formatına göre görüntülenmesi.

Oyun geçmişi listesinde tıklanan liste elemanının, seçili olarak işaretlenmesi.

Board'daki karelerin, elle hardcoded olarak kodlanmasının yerine iki for döngüsü kullanılarak Board bileşeninin düzenlenmesi.

Bir buton eklenerek, tıklandığında oyun geçmişinin artan veya azalan şekilde sıralanmasının sağlanması.

Bir kişi kazandığında, kazanmasına vesile olan 3 karenin renklendirilerek vurgulanması.

Eğer hiç kazanan yoksa, berabere mesajının görüntülenmesi.

Bu öğreticide, React konseptleri olan elemanlar, bileşenler, prop'lar, ve state'e değindik. Bu konular hakkında daha detaylı bir açıklama için dokümanın geri kalanını inceleyebilirsiniz. Bileşen tanımlama hakkında daha fazla bilgi almak için React.Component API dokümanını inceleyebilirsiniz.