State ve Yaşam Döngüsü

Bu sayfada, state kavramı ve React bileşenlerinin yaşam döngüsü tanıtılacaktır. Bileşen API'si hakkında ayrıntılı bilgi için, bu dokümana bakabilirsiniz.

Önceki bölümlerde bahsettiğimiz, analog saat örneğini ele alacağız. Hatırlayacağınız gibi, Elementlerin Render Edilmesi bölümünde, kullanıcı arayüzünün yalnızca tek yönlü güncellenmesine yer vermiştik. Bunu root.render() metodu ile gerçekleştirebiliyorduk:

tick component

```
const root = ReactDOM.createRoot(document.getElementById('root'));

function tick() {
 const element = (
  <div>
   <h1>Hello, world!</h1>
   <h2>It is {new Date().toLocaleTimeString()}.</h2>
  </div>
 );
 root.render(element);
}setInterval(tick, 1000);
```

Bu bölümde ise, Clock bileşenini nasıl sarmalayacağımıza ve tekrar kullanılabilir hale getireceğimize değineceğiz. Bu bileşen, kendi zamanlayıcısını başlatacak ve her saniye kendisini güncelleyecek.

Öncelikle Clock'u, ayrı bir bileşen halinde sarmalayarak görüntüleyelim:

Clock component

```
const root = ReactDOM.createRoot(document.getElementById('root'));

function Clock(props) {
 return (
  <div>
   <h1>Hello, world!</h1>
```

```
    <h2>It is {props.date.toLocaleTimeString()}.</h2>
  </div>
 );
}
```

tick component

```
function tick() {
 root.render(<Clock date={new Date()} />);
}setInterval(tick, 1000);
```

Güzel görünüyor ancak bu aşamada kritik bir gereksinimi atladık: Clock'un kendi zamanlayıcısını ayarlaması ve her saniye kullanıcı arayüzünü güncellemesi işini kendi bünyesinde gerçekleştirmesi gerekiyordu.

Aşağıdaki kodu bir kere yazdığımızda, Clock'un artık kendi kendisini güncellemesini istiyoruz:

```
root.render(<Clock />);
```

Bunu yapmak için, Clock bileşenine state eklememiz gerekiyor.

State'ler, prop'larla benzerlik gösterir. Fakat sadece ilgili bileşene özeldir ve yalnızca o bileşen tarafından kontrol edilirler.

Sınıf olarak oluşturulan bileşenlerin, fonksiyon bileşenlerine göre bazı ek özelliklerinin bulunduğundan bahsetmiştik. Bahsettiğimiz ek özellik yerel state değişkenidir ve sadece sınıf bileşenlerine özgüdür.

* Bir Fonksiyonun Sınıfa Dönüştürülmesi

Clock gibi bir fonksiyon bileşenini 5 adımda sınıf bileşenine dönüştürebilirsiniz:

1-Öncelikle, fonksiyon ismiyle aynı isimde bir ES6 sınıfı oluşturun. Ve bu sınıfı **React.Component**'tan türetin.

2-Sınıfın içerisine, **render()** adında boş bir fonksiyon ekleyin.

3-Fonksiyon bileşeni içerisindeki kodları **render()** metoduna taşıyın.

4-**render()** metodu içerisindeki **props** yazan yerleri, **this.props** ile değiştirin.

5-Son olarak, içi boşaltılmış fonksiyonu tamamen silin.

```
class Clock extends React.Component {
```

```
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.props.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Önceden fonksiyon bileşeni olan Clock, artık bir sınıf bileşeni haline gelmiş oldu.

Bu kodda render metodumuz, her güncelleme olduğunda yeniden çağrılacaktır. Fakat **<Clock/>** bileşenini aynı DOM düğümünde render ettiğimizden, Clock sınıfının yalnızca bir örneği kullanılacaktır.

* Bir Sınıfa Yerel State'in Eklenmesi

date değişkenini, props'tan state'e taşımamız gerekiyor. Bunu 3 adımda gerçekleştirebiliriz:

1-render() metodundaki this.props.date'i this.state.date ile değiştirelim:

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

2-state'in ilk kez oluşturulacağı yer olan sınıf constructor'ını ekleyelim:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

props'ı, constructor içerisinde nasıl oluşturduğumuza yakından bakalım:

```
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
```

Sınıf bileşenleri React.Component sınıfından türetildikleri için, daima super(props)'u çağırmaları gerekir.

3-<Clock /> elementinden date prop'unu çıkaralım:

```
root.render(<Clock />);
```

Zamanlayıcı kodunu, daha sonra Clock bileşenin içerisine ekleyeceğiz. Fakat şimdilik Clock bileşeninin son hali aşağıdaki gibi olacaktır:

```
class Clock extends React.Component {
```

```
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

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

root.render(<Clock />);
```

Şimdi Clock bileşenini, kendi zamanlayıcısını kuracak ve her saniye kendisini güncelleyecek şekilde ayarlayalım.

* Bir Sınıfın Yaşam Döngüsü Kodlarının Eklenmesi

Birçok bileşene sahip uygulamalarda, bileşenler yok edildiğinde ilgili kaynakların bırakılması çok önemlidir.

Clock bileşeni ilk kez DOM'a render edildiğinde bir zamanlayıcı kurmak istiyoruz. React'te bu olaya **"mounting" (değişkenin takılması)** adı verilir.

Ayrıca, Clock bileşeni DOM'dan çıkarıldığında, zamanlayıcının da temizlenmesini istiyoruz. React'te bu olaya **"unmounting" (değişkenin çıkarılması)** adı verilir.

Clock bileşeni takılıp çıkarıldığında bazı işleri gerçekleştirebilmek için özel metotlar tanımlayabiliriz:

```
class Clock extends React.Component {
  constructor(props) {
```

```
    super(props);
    this.state = {date: new Date()};
  }
  componentDidMount() {
  }
  componentWillUnmount() {
  }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Bu metotlara **“lifecycle methods” (yaşam döngüsü metotları)** adı verilir.

Bileşenin çıktısı, DOM’a render edildikten sonra **componentDidMount()** metodu çalıştırılır. Burası aynı zamanda bir zamanlayıcı oluşturmak için en elverişli yerdir:

```
  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }
```

this‘e (this.timerID) zamanlayıcı ID’sini nasıl atadığımızı inceleyebilirsiniz.

Daha önce de belirttiğimiz gibi, this.props React tarafından yönetiliyor ve this.state'in de özel bir yaşam döngüsü var. Eğer timerID gibi veri akışına dahil olmayan değişkenleri saklamanız gerekiyorsa, bu örnekte yaptığımız gibi sınıf içerisinde değişkenler tanımlayabilirsiniz.

Oluşturduğumuz zamanlayıcıyı **componentWillUnmount()** yaşam döngüsü metodu içerisinde Clock bileşeninden söküp çıkaralım:

```
 componentWillUnmount() {
   clearInterval(this.timerID);
 }
```

Son olarak, Clock bileşeninin saniyede bir çalıştıracağı tick() fonksiyonunu kodlayalım.

tick() fonksiyonu, this.setState()'i çağırarak Clock bileşeninin yerel state'ini güncelleyecektir:

```
class Clock extends React.Component {
 constructor(props) {
   super(props);
   this.state = {date: new Date()};
 }

 componentDidMount() {
   this.timerID = setInterval(
     () => this.tick(),
     1000
   );
 }

 componentWillUnmount() {
   clearInterval(this.timerID);
 }
```

```
  tick() {
    this.setState({
      date: new Date()
    });
  }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Clock />);
```

Artık saat, her saniye başı tikleyerek mevcut zamanı görüntüleyecektir.

Şimdi kısa bir özet geçerek neler yaptığımızı ve sırasıyla hangi metotların çağrıldığını kontrol edelim:

1-root.render() metoduna <Clock /> aktarıldığı zaman React, Clock bileşeninin constructor'ını çağırır. Clock bileşeni, mevcut saati görüntülemesi gerektiğinden, this.state'e o anki zamanı atar. Daha sonra bu state güncellenecektir.

2-Devamında React, Clock bileşeninin render() metodunu çağırır. Bu sayede React, ekranda nelerin gösterilmesi gerektiğini bilir. Sonrasında ise Clock'un render edilmiş çıktısı ile eşleşmek için ilgili DOM güncellemelerini gerçekleştirir.

3-Clock bileşeninin çıktısı DOM'a eklendiğinde, yaşam döngüsündeki componentDidMount() metodu çağrılır. Bu metotta Clock bileşeni, her saniyede bir tick() metodunun çalıştırılması gerektiğini tarayıcıya bildirir.

4-Tarayıcı her saniyede bir tick() metodunu çağırır. tick() metodunda Clock bileşeni, kullanıcı arayüzünü güncellemek için setState() metodunu çağırır ve bu metoda mevcut tarih/saat değerini aktarır. setState()'in çağrılması sayesinde React, state'in değiştiğini anlar ve ekranda neyin görüntüleneceğini anlamak için tekrar render() metodunu çağırır. Artık render() metodundaki this.state.date'in değeri eski halinden farklı olduğundan, render çıktısı güncellenmiş zamanı içerecek demektir. Buna göre React, DOM'ı ilgili şekilde günceller.

5-Eğer Clock bileşeni, DOM'dan çıkarılırsa, yaşam döngüsündeki componentWillUnmount() metodu çağrılır ve tarayıcı tarafından zamanlayıcı durdurulmuş olur.

* State'in Doğru Kullanımı

setState() hakkında bilmeniz gereken 3 şey bulunmaktadır.

1-State'i Doğrudan Değiştirmeyiniz

Örneğin, aşağıdaki kod bileşeni yeniden render etmeyecektir:

// Yanlış kullanım

```
this.state.comment = 'Hello';
```

Bunun yerine setState() kullanınız:

// Doğru kullanım

```
this.setState({comment: 'Hello'});
```

this.state'e atama yapmanız gereken tek yer, ilgili bileşenin constructor'ıdır.

2-State Güncellemeleri Asenkron Olabilir

React, çoklu setState() çağrılarını, performans için tekil bir güncellemeye dönüştürebilir.

this.props ve this.state, asenkron olarak güncellenebildiklerinden, sonraki state'i hesaplarken bu nesnelerin mevcut değerlerine güvenmemelisiniz.

Örneğin, aşağıdaki kod counter'ı güncellemeyebilir:

// Yanlış kullanım

```
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

Bunu düzeltmek için, setState()'in ikinci formunu kullanmamız gerekir. Bu formda setState() fonksiyonu, parametre olarak nesne yerine fonksiyon alır. Bu fonksiyon, ilk parametre olarak önceki state'i, ikinci

parametre olarak da o anda güncellenen props değerini alır:

// Doğru kullanım

```
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

Yukarıda bir ok fonksiyonu kullandık. Fakat normal fonksiyonlarla da gayet çalışabilir:

// Doğru kullanım

```
this.setState(function(state, props) {
  return {
    counter: state.counter + props.increment
  };
});
```

3-State Güncellemeleri Birleştirilir

React, setState()'i çağırdığınızda, parametre olarak verdiğiniz nesneyi alıp mevcut state'e aktarır.

Örneğin, state'iniz aşağıdaki gibi birçok bağımsız değişkeni içerebilir:

```
  constructor(props) {
    super(props);
    this.state = {
      posts: [],
      comments: []
    };
  }
```

Ve siz de bu değişkenleri, ayrı birer setState() çağrıları ile güncellemek isteyebilirsiniz:

```
  componentDidMount() {
    fetchPosts().then(response => {
```

```
    this.setState({
      posts: response.posts
    });
  });

  fetchComments().then(response => {
    this.setState({
      comments: response.comments
    });
  });
 }
```

Birleşme işlemi yüzeysel olduğundan, this.setState({comments}) çağrısı this.state.posts değişkenini değişmeden bırakırken, this.state.comments'i tamamıyla değiştirecektir.

* Verinin Alt Bileşenlere Aktarılması

Ne üst ne de alt bileşenler, belirli bir bileşenin state'li veya state'siz olduğunu bilebilir. Ayrıca o bileşenin fonksiyon veya sınıf olarak tanımlanmasını da önemsemezler.

Bu nedenle state'e, yerel state denir. State, kendisine sahip olan ve kendisini ayarlayan bileşen haricinde hiçbir bileşen için erişilebilir değildir.

Bir bileşen kendi state'ini, prop'lar aracılığıyla alt bileşenlere aktarabilir:

```
<FormattedDate date={this.state.date} />
```

FormattedDate bileşeni, date değişkenini props'tan alabilir ve bunu alırken Clock'un state'inden mi yoksa prop'undan mı geldiğini bilemez. Hatta date değişkeni, Clock bileşeni içerisinde state'ten harici olarak tanımlanmış bir değer de olabilir ve bunu bilmesi mümkün değildir:

```
function FormattedDate(props) {
  return <h2>It is {props.date.toLocaleTimeString()}.</h2>;
}
```

Bu olaya genellikle yukarıdan-aşağıya veya tek yönlü veri akışı denir. Her state, belirli bir bileşen tarafından tutulur. Bu bileşenden türetilen herhangi bir veri veya kullanıcı arayüzü, yalnızca bu bileşenin altındaki bileşen ağacına etki edebilir.

Bileşen ağacını, prop'lardan oluşan bir şelale olarak düşünebilirsiniz. Her bileşenin state'i, prop'ları istenilen bir noktada birleştirebilen ve aynı zamanda alt bileşenlere de akıtan ek bir su kaynağı gibidir.

Tüm bileşenlerin tamamen izole olduğunu göstermek için, 3 adet <Clock> render eden bir App bileşeni oluşturabiliriz:

```
function App() {
  return (
    <div>
      <Clock />
      <Clock />
      <Clock />
    </div>
  );
}
```

Bu örnekte yer alan her bir Clock bileşeni, kendi zamanlayıcısını oluşturup birbirinden bağımsız bir şekilde güncellemektedir.

React uygulamalarında, bir bileşenin state'li veya state'siz olması, bir kodlama detayıdır ve zaman içerisinde değişkenlik gösterebilir. State'li bileşenler içerisinde, state'siz bileşenleri kullanabilirsiniz veya bu durumun tam tersi de geçerlidir.