React.Component

Giriş

React, bileşenleri sınıf ve fonksiyon olarak tanımlamanızı sağlar. Sınıf olarak tanımlanan bileşenler, fonksiyon olanlara göre daha fazla özellik sunar. Bu özellikler sayfanın ilerleyen bölümlerinde daha detaylı şekilde ele alınacaktır.

React bileşen sınıfı oluşturmak için, sınıfınızı React.Component'ten türetmeniz gerekir:

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

React.Component'ten türetilen sınıflarda, zorunlu olarak tanımlamanız gereken metot sadece render()'dır. Bu sayfada tanıtılacak olan diğer metotlar ise opsiyoneldir.

React.Component yerine kendi ürettiğiniz temel sınıfları oluşturmanızı kesinlikle tavsiye etmiyoruz. Çünkü React bileşenlerinde, kodun tekrar kullanılabilirliği kalıtım yoluyla değil, kompozisyon ile sağlanır.

Not:

React'te sınıf bileşeni tanımlarken, ES6 kod yapısını kullanmak zorunda değilsiniz. Eğer ES6 kullanmak istemiyorsanız, npm paketi olarak yer alan create-react-class modülünü veya benzer bir özel soyutlama yöntemini kullanabilirsiniz.

Bir Bileşenin Yaşam Döngüsü

Her bileşen, belirli durumlarda çalıştırabileceğiniz birkaç "yaşam döngüsü metodu" (lifecycle methods) sunar. Bu metodları hatırlamak için, yaşam döngüsü diyagramını kullanabilirsiniz. Aşağıdaki listede, yaygın olarak kullanılan yaşam döngüsü metodları kalın harfler ile belirtilmiştir. Geri kalan metotlar, daha nadir kullanımlar için uygundur.

Eklenmesi

Bir bileşenin oluşumundan ve DOM'a eklenmesine kadar geçen süreç içerisinde çağrılan metotlar, sırasıyla aşağıdaki gibi belirlenmiştir:

```
constructor()
```

static getDerivedStateFromProps()

render()

componentDidMount()

Not:

Aşağıdaki metot eski React projelerinde kullanılmaktaydı. Fakat asenkron render etme süreçlerinde problemli olduğundan dolayı artık yeni projelerde kullanmamanız gerekmektedir:

UNSAFE_componentWillMount()

Güncellenmesi

Bileşenin güncellemesi, kendi props'u veya state'i üzerindeki değişikliklerden oluşabilir. Bir bileşen tekrar render edildiğinde çağrılan metotlar sırasıyla aşağıdaki gibidir:

static getDerivedStateFromProps()

shouldComponentUpdate()

render()

getSnapshotBeforeUpdate()

componentDidUpdate()

Not:

Aşağıdaki metotları, problemli olduklarından dolayı, artık yeni projelerde kullanmamanız gerekir:

UNSAFE_componentWillUpdate()

UNSAFE_componentWillReceiveProps()

Çıkarılması

Bir bileşen, DOM'dan çıkarıldığında bu metot çalışır:

componentWillUnmount()

Hatanın Yakalanması

Aşağıdaki metotlar; render esnasında, yaşam döngüsü metodunda veya herhangi bir alt bileşenin constructor'ında bir hata oluştuğunda çağrılmaktadır:

static getDerivedStateFromError()

componentDidCatch()

Diğer API'lar

Her bileşen, bu metotların haricinde aşağıdaki gibi bazı API'ları sunmaktadır:

setState()

forceUpdate()

Sınıf Özellikleri

defaultProps

displayName

Nesne Özellikleri

props

state

Yaygın Olarak Kullanılan Yaşam Döngüsü Metotları

Bu bölümde anlatılacak metotlar, React bileşenleri oluştururken yaygın olarak karşılaşacağınız kullanım senaryolarını içerir. Görsel bir anlatım için yaşam döngüsü diyagramını inceleyebilirsiniz.

render()

render()

render() metodu, bir sınıf bileşeni oluşturmak için gereken tek metottur.

Çağrıldığında, this.props ile this.state'i denetler ve aşağıdaki veri tiplerinden birini geri döndürür:

React elementleri. Genellikle JSX kodu kullanılarak oluşturulurlar. Örneğin, ve <MyComponent /> birer React elementidir. , React'e bir HTML DOM düğümünü render etmesini gerektiğini bildirir. <MyComponent/> ise kullanıcının tanımladığı bir React bileşendir.

Diziler and fragment'lar. Render edilecek olan birden fazla elemanı geri döndürürler. Daha fazla bilgi için fragments dokümanını inceleyebilirsiniz.

Portal'lar. Alt bileşenleri, farklı bir DOM alt ağacı olarak render etmeyi sağlarlar. Daha fazla bilgi için portals dokümanını inceleyebilirsiniz.

String'ler ve sayılar. DOM içerisinde metin düğümü olarak render edilirler.

Boolean'lar ve null. Hiçbir şey render etmezler. (Genellikle return test && <Child /> tarzındaki kod yapısını desteklemek için vardırlar. Buradaki test, boolean tipinde bir değişkendir.)

render() metodu saf halde olmalıdır. Yani bileşenin state'ini değiştirmemeli, aynı parametrelerle çağrıldığında hep aynı çıktıyı vermeli, ve internet tarayıcısı ile direkt olarak etkileşimde bulunmamalıdır. Eğer tarayıcı ile etkileşimde bulunmanız gerekirse, componentDidMount()'ta veya diğer yaşam döngüsü metotlarında bu işlemi gerçekleştiriniz.

render()'ın saf halde tutulması, bileşen üzerinde daha kolay düşünmenizi sağlar.

Not

Eğer shouldComponentUpdate() metodu false dönerse, render() metodu çağrılmaz.

constructor()

constructor(props)

state'i kullanmadığınız durumlarda veya bind() fonksiyonu ile herhangi bir metot bağlamadığınız sürece, React bileşeni için bir constructor metodu oluşturmanız gerekli değildir.

Bir React bileşeninin constructor'ı, ilgili bileşen uygulamaya eklenmeden önce çağrılır. React.Component'tan türetilen sınıf için bir constructor oluştururken, fonksiyon içerisinde ilk satırda super(props) çağırmanız zorunludur. Aksi halde this.props özelliği, constructor içerisinde undefined olarak değer alacaktır. Bu durum, uygulamanızda birtakım hatalara neden olabilir.

Constructor, genellikle iki temel amaçla kullanılır:

1-this.state'e bir nesne atanarak yerel state'in oluşturulması.

2-bind() metodu kullanılarak, bileşene olay metotlarının bağlanması.

constructor() içerisinde, setState() metodunu çağırmamalısınız. Eğer bileşeninizin yerel state'i kullanması gerekiyorsa, ilgili değişkenleri constructor içerisinde direkt olarak this.state'e atayınız:

```
constructor(props) {
  super(props);
  // Burada this.setState()'i çağırmayınız!
  this.state = { counter: 0 };
  this.handleClick = this.handleClick.bind(this);
}
```

this.state'e direkt olarak atama yapmanız gereken tek yer constructor'dır. Diğer tüm metotlarda, this.setState()'i kullanmanız gereklidir.

Constructor'da, yan etki eden metotlardan veya setInterval() gibi abonelik metotlarını oluşturmaktan

kaçınınız. Bu tür işlemleri componentDidMount() metodunda gerçekleştiriniz.

Not

state'e prop nesnelerinin içeriklerini atamayınız! Bu çok yaygın olarak yapılan bir hatadır:

```
constructor(props) {
 super(props);
 // İşte bunu yapmayınız!
 this.state = { color: props.color };
}
```

Buradaki problemlerden birincisi, state'e props değerinin atanması gereksizdir. Çünkü this.props.color değeri direkt olarak kullanılabilir. İkinci problem ise, color prop'unda yapılan değişiklikler, state'e henüz yansıtılmadığı için hatalara neden olur.

Bu tarz bir kodlamayı, yalnızca prop güncellemelerini göz ardı etmek istediğinizde yapınız. Böyle bir durumda color değişkenini, initialColor (başlangıç rengi) veya defaultColor (varsayılan renk) olarak isimlendirmek daha uygun hale gelecektir. Bileşenin iç state'ini güncellemek için zorlamanız gerektiğinde bunu key özelliğini değiştirerek yapabilirsiniz.

State'in türetilmesinden kaçınmak adlı makalemizi okuyarak, prop'lara bağlı bir state'e ihtiyacınız olduğunda ne yapmanız gerektiği ile ilgili detaylı bilgi edinebilirsiniz.

componentDidMount()

componentDidMount()

Bir bileşen, DOM ağacına eklendikten hemen sonra componentDidMount() çalıştırılır. DOM düğümleri ile ilişkili atama işlemleri bu fonksiyon içerisinde yapılmalıdır. Bu nedenle eğer verilerinizi uzak bir API'den yüklemeniz gerekiyorsa, ağ isteğini bu fonksiyonda başlatabilirsiniz.

Ayrıca bu metot, setInterval() gibi abonelik gerektiren metotları çağırmak için de uygundur. Eğer böyle bir abonelik metodu çağırdıysanız, componentWillUnmount() metodu içerisinde abonelikten çıkmayı unutmayınız.

componentDidMount()'ta setState()'i çağırabilirsiniz. Bunun sonucunda, bileşenin fazladan render edilmesi tetiklenecektir. Fakat bu işlem, tarayıcının arayüzü güncellemesinden önce gerçekleşecektir. Bu durum, render() metodu iki kez çalışsa bile, kullanıcının bu olayı farketmemesini garanti eder. Fakat bu kodlama mantığını kullanırken dikkatli olunuz. Çünkü bu kullanım, genellikle performans sorunlarına yol açmaktadır. Bu nedenle birçok durumda, state atamalarını constructor() metodu içerisinde gerçekleştiriniz. Ancak tooltip veya modal bileşenlerinin gösterildiği durumlarda, render işlemi öncesinde ilgili DOM düğümünün boyutu veya pozisyonu gibi bir özelliği ölçümlemek istiyorsanız, bu kod mantığını

kullanmanız gereklidir.

componentDidUpdate()

componentDidUpdate(prevProps, prevState, snapshot)

Adından da anlaşılacağı gibi componentDidUpdate() metodu, sadece DOM güncellemelerinde gerçekleştirilir. Bu nedenle başlangıçtaki render işleminde çağrılmaz.

Bileşen güncellendiğinde, DOM üzerinde yapmak istediğiniz işleri gerçekleştirmek için bu metodu kullanınız. Ayrıca bu metot, önceki prop ile sonraki prop değerlerini karşılaştırıp, buna bağlı olarak ağ isteklerini gerçekleştirmek için de uygun bir yerdir. Örneğin aşağıdaki örnekte olduğu gibi, prop nesnesi değişmediyse ağ isteğinin yapılmasına da gerek yoktur:

```
componentDidUpdate(prevProps) {
  // Genel kullanım (prop değerlerini karşılaştırmayı unutmayınız!):
  if (this.props.userID !== prevProps.userID) {
    this.fetchData(this.props.userID);
  }
}
```

componentDidUpdate() metodunun içerisinde setState() çağrısı yapabilirsiniz. Fakat unutmayınız ki bu çağrı, üstteki gibi bir koşul ifadesi içerisinde yer almalıdır. Aksi halde uygulamanız sonsuz döngüye girebilir. Ayrıca kullanıcı tarafından farkedilmeyen fakat bileşen performansına etki edebilecek seviyede ekstra bir render işlemi gerçekleşmesine neden olur. Eğer üst bileşenden gelen prop'u mevcut bileşenin state'ine kopyalamaya çalışıyorsanız, bunun yerine direkt olarak prop kullanabilirsiniz. Neden prop'un state'e kopyalanmasının hatalara yol açabileceği hakkında daha fazla bilgi edinmek için blog yazısını inceleyebilirsiniz..

Eğer bileşeninizde getSnapshotBeforeUpdate() yaşam döngüsü metodunu kodlamışsanız (ki bu nadir bir durumdur), geri döndürdüğü değer componentDidUpdate() metoduna "snapshot" ismiyle üçüncü parametre olarak aktarılır. Aksi halde bu parametre, undefined olacaktır.

Not:

Eğer shouldComponentUpdate() metodu false döndürüyorsa, componentDidUpdate() metodu çağrılmayacak demektir.

componentWillUnmount()

componentWillUnmount()

componentWillUnmount() metodu, bir bileşen DOM'dan çıkarıldığında veya tamamen yok edildiğinde

çalıştırılır. componentDidMount()'ta yapılan; zamanlayıcı fonksiyonların geçersiz kılınması, ağ isteklerinin iptal edilmesi, veya herhangi bir abonelik metodunun temizlenmesi gibi işlemleri bu metotta gerçekleştiriniz.

componentWillUnmount()'ta setState() metodunu çağırmamalısınız. Çünkü, bileşen artık DOM'dan ayrıldığı için, tekrar render edilme işlemi asla gerçekleştirilmeyecektir. Bir bileşen eğer DOM'dan ayrıldıysa, artık tekrar DOM'a geri takılma süreci gerçekleştirmeyecektir.

Nadiren Kullanılan Yaşam Döngüsü Metotları

Bu bölümdeki metotlar, nispeten daha az kullanılan durumlar içindir. Nadiren işinizi görseler de, büyük ihtimalle bileşenlerinizde hiçbirini kullanmayacaksınız. Bu yaşam döngüsü şemasının üst kısımında yer alan "Daha az kullanılan yaşam döngülerini göster" kutucuğunu işaretlediğinizde bu metotların çoğunu görebileceksiniz.

shouldComponentUpdate()

shouldComponentUpdate(nextProps, nextState)

Mevcut state veya prop'lar değiştiğinde, bileşenin çıktısının bu durumdan etkilenmemesini belirtmek için shouldComponentUpdate() metodunu kullanınız. Normalde bileşenin varsayılan davranışı, her state değişikliğinde tekrar render edilmesine yöneliktir. Birçok kullanımda bu varsayılan davranışa uymanız gerekmektedir.

Prop veya state değerleri değiştirildiğinde, render işleminden hemen önce shouldComponentUpdate() metodu çalıştırılır ve varsayılan olarak true döndürür. Bileşenin başlangıçtaki ilk render zamanında veya forceUpdate() metodu kullanıldığında, bu metot çalıştırılmaz.

Bu metot yalnızca performans iyileştirme işlemleri için yapılmıştır. Render işlemini engellemek için bu metodu kullanmayınız. Zira bazı hataların oluşmasına yol açabilir. Bu nedenle, shouldComponentUpdate() metodunu yazmak yerine, React içerisinde varsayılan olarak gelen PureComponent kullanınız. PureComponent, prop ve state'leri yüzeysel olarak karşılaştırır. Bu sayede büyük DOM ağaçlarına sahip bileşenlerde, küçük değişiklikler gerçekleştiğinde oluşacak güncellemelerin oluşma şansını azaltır. Böylece gereksiz güncellemeler göz ardı edilerek performans artışı sağlanmış olur.

Eğer bu metodu kullanmak için eminseniz, güncellemenin göz ardı edilmesi için nextProps ile this.props'u, nextState ile this.state karşılaştırabilir ve bunun sonucunda false değerini döndürebilirsiniz. false'un geri döndürülmesi işlemi, alt bileşenlerin state'i değiştiğinde tekrar render edilmelerini engellemeyeceğini unutmayınız.

shouldComponentUpdate() metodu içerisinde, eşitlik kontrollerinin derinlemesine gerçekleştirilmesi veya JSON.stringify()'ın kullanımı önerilmez. Bu tür kullanımlar verimsizdir ve performansı olumsuz yönde etkiler.

React'in mevcut sürümünde, shouldComponentUpdate() metodu false döndürdüğünde;

UNSAFE_componentWillUpdate(), render(), ve componentDidUpdate() metodları çağrılmaz. Gelecek sürümlerde React, shouldComponentUpdate() metodunu sıkı bir şekilde uygulamak yerine bir ipucu şeklinde ele alabilir. Bu nedenle false döndürülmesine rağmen, bileşenin tekrar render edilmesi ile sonuçlanabilir.

static getDerivedStateFromProps()

static getDerivedStateFromProps(props, state)

Bileşenin başlangıçta DOM'a eklenmesinde ve devamında süregelen güncellemelerde, render metodundan hemen önce getDerivedStateFromProps çalıştırılır. Bu metot, state'in güncellenmesi için bir nesne geri döndürür. null döndürdüğünde ise güncelleme gerçekleştirilmemiş olur.

Bu metot, state'in props değişikliklerine bağlı olduğu nadiren kullanılan durumlar için vardır. Örneğin <Transition> bileşeninin, önceki ve sonraki alt bileşenlerini karşılaştırması sayesinde animasyona girme/çıkma süreçlerinin yönetimi için kullanışlı olabilir.

State üretmek, daha fazla kod yazmaya neden olur. Ve bir süre sonra bileşen kodunu takip edemez hale gelirsiniz. Bunun yerine alternatif yollar deneyebilirsiniz:

Eğer props'ta oluşan değişikliklere cevap olarak, web isteği veya animasyon işlemi gibi yan etki içeren bir işlem gerçekleştirmeniz gerekiyorsa, componentDidUpdate yaşam döngüsü metodunu kullanınız.

Eğer bir prop değiştiğinde, bazı verileri yeniden işlemeniz gerekiyorsa, bunun için memoization yöntemini kullanınız.

Eğer bir prop değiştiğinde, bazı state değerlerini sıfırlamanız gerekiyorsa, bunun için tamamen kontrollü bir bileşen veya bir key'e sahip tamamen kontrolsüz bir bileşen kullanınız.

Bu metodun, bileşen nesnesine erişimi yoktur. Eğer dilerseniz sınıf tanımının dışarısında, bileşen prop'larından ve state'inden saf fonksiyonlar çıkararak, getDerivedStateFromProps() ve diğer sınıf metotları arasında bazı kod içeriklerini tekrar kullanabilirsiniz.

Unutmayınız ki bu metot, sebebi ne olursa olsun her render işlemi esnasında çağrılır. Bu durum, yalnızca üst bileşenin tekrar render işlemine sebebiyet verdiği ve yerel setState'in render etmediğinde çalıştırılan UNSAFE_componentWillReceiveProps'un tam zıttıdır.

getSnapshotBeforeUpdate()

getSnapshotBeforeUpdate(prevProps, prevState)

Bileşenin render edilmiş çıktısı DOM'a yerleştirilmeden hemen önce getSnapshotBeforeUpdate() çağrılır. Bu sayede DOM değişmeden önce, kaydırma çubuğu (scrollbar) pozisyonu gibi bazı bilgilerin DOM'dan alınması sağlanır. Bu yaşam döngüsü metodundan döndürülen her değer, componentDidUpdate()'e parametre olarak geçilir.

getSnapshotBeforeUpdate()'in kullanımı yaygın değildir. Fakat bir sohbet uygulamasında yeni mesaj

geldiğinde, kaydırma çubuğunun aşağı kaydırılması gibi özel işlemlerde gerekli olabilir.

Bir anlık görüntü değeri (snapshot) veya null geri döndürülür.

Örneğin:

```
class ScrollingList extends React.Component {
 constructor(props) {
  super(props);
  this.listRef = React.createRef();
 }

 getSnapshotBeforeUpdate(prevProps, prevState) {
  // Are we adding new items to the list?
  // Capture the scroll position so we can adjust scroll later.
  if (prevProps.list.length < this.props.list.length) {
   const list = this.listRef.current;
   return list.scrollHeight - list.scrollTop;
  }
  return null;
 }

 componentDidUpdate(prevProps, prevState, snapshot) {
  // If we have a snapshot value, we've just added new items.
  // Adjust scroll so these new items don't push the old ones out of view.
  // (snapshot here is the value returned from getSnapshotBeforeUpdate)
  if (snapshot !== null) {
   const list = this.listRef.current;
```

```
      list.scrollTop = list.scrollHeight - snapshot;
    }
  }
  render() {
    return (
      <div ref={this.listRef}>{/* ...contents... */}</div>
    );
  }
}
```

Üstteki örnekte, scrollHeight özelliğinin getSnapshotBeforeUpdate'ten okunması çok önemlidir. Çünkü "render" adımındaki yaşam döngüsü (render()) ile "commit" adımındaki yaşam döngüleri (getSnapshotBeforeUpdate ve componentDidUpdate) arasında gecikmeler yaşanabilir.

Hata sınırları

Hata sınırları (error boundaries), alt bileşen ağacında gerçekleşen bir JavaScript hatasını yakalayan React bileşenleridir. Yakaladıkları hatayı kaydeder ve bu hatadan dolayı çöken bileşen ağacının gösterilmesi yerine, yedek olarak oluşturulan bir arayüz öğesinin görüntülenmesini sağlarlar. Hata sınırları, kendi alt ağacında gerçekleşen render işlemlerinde, yaşam döngüsü metotlarında ve constructor'larda oluşan herhangi bir hatayı yakalarlar.

Bir sınıf bileşeni, static getDerivedStateFromError() veya componentDidCatch() yaşam döngüsü metotlarını içerirse, o bileşen artık bir hata sınırı haline gelir. State'in bu yaşam döngüsü metotları ile güncellenmesi, alt ağaçta oluşabilecek beklenmedik JavaScript hatalarının yakalanmasını ve bunun için bir arayüz görüntülenmesini sağlar.

Hata sınırlarını yalnızca beklenmedik exception'ların giderilmesi için kullanınız: kontrol akışı için kullanmayınız.

Daha fazla bilgi için React 16'da hata yönetimini inceleyiniz.

Not

Hata sınırları sadece altındaki ağaçta bulunan bileşenlerde oluşan hataları yakalarlar. Bu nedenle hata sınırları, kendi içerisinde oluşan bir hatayı yakalayamazlar.

static getDerivedStateFromError()

static getDerivedStateFromError(error)

getDerivedStateFromError(error) metodu, bir alt bileşende hata oluştuktan hemen sonra çalıştırılır. Parametre olarak oluşan hata nesnesini alır ve state'in güncellenmesi için geriye bir değer döndürür:

```
class ErrorBoundary extends React.Component {
 constructor(props) {
  super(props);
  this.state = { hasError: false };
 }

 static getDerivedStateFromError(error) {
  // state'in güncellenmesi ile sonraki aşamada hata mesajının render edilmesi sağlanacaktır
  return { hasError: true };
 }
 render() {
  if (this.state.hasError) {
   // Hatanın görüntülenmesi için herhangi bir içerik sunabilirsiniz
   return <h1>Something went wrong.</h1>;
  }
  return this.props.children;
 }
}
```

Not

getDerivedStateFromError() metodu, "render" aşamasında çağrılır. Bu nedenle herhangi bir yan etkiye izin verilmez. Bu tür kullanımlar için, componentDidCatch()'i kullanınız.

componentDidCatch()

componentDidCatch(error, info)

componentDidCatch(error, info) metodu, bir alt bileşende hata oluştuktan sonra hemen çalıştırılır. İki parametre alır:

error - Oluşan hata nesnesi.

info - Hatayı hangi bileşenin verdiği ile ilgili bilgileri tutan componentStack'i içeren hata bilgisi nesnesidir.

componentDidCatch() metodu, güncellemenin "commit" adımında çalıştırılır. Bu nedenle yan etkiye izin verir. Hataların log'lanması tarzındaki işler için kullanılmalıdır:

```
class ErrorBoundary extends React.Component {
 constructor(props) {
  super(props);
  this.state = { hasError: false };
 }

 static getDerivedStateFromError(error) {
  // Sonraki render aşamasında hata mesajının görüntülenmesi için state güncelleniyor.
  return { hasError: true };
 }

 componentDidCatch(error, info) {
  // Örnek bir `componentStack` metninin içeriği aşağıdaki gibidir:
  //   in ComponentThatThrows (created by App)
  //   in ErrorBoundary (created by App)
  //   in div (created by App)
  //   in App
  logComponentStackToMyService(info.componentStack);
 }
 render() {
  if (this.state.hasError) {
   // Herhangi bir hata bileşeni görüntüleyebilirsiniz.
```

```
    return <h1>Something went wrong.</h1>;
  }
  return this.props.children;
 }
}
```

Not

Bir hata durumunda, setState'i çağırarak componentDidCatch() ile bir hata arayüzü görüntüleyebilirsiniz. Fakat bu yaklaşım, gelecekteki React sürümlerinde kullanımdan kaldırılacaktır. Bunun yerine hata arayüzünün render edilmesi için static getDerivedStateFromError() metodunu kullanınız.

Eski Yaşam Döngüsü Methotları

Aşağıdaki yaşam döngüsü metotları eski (legacy) olarak işaretlenmişlerdir. Bu metotlar hala çalışıyor olmalarına rağmen, yeni yazacağınız kodlarda bu metotları kullanmanızı tavsiye etmiyoruz. Eski yaşam döngüsü metotlarından kurtulmak için bu blog yazısını inceleyebilirsiniz.

UNSAFE_componentWillMount()

UNSAFE_componentWillMount()

Not:

Bu yaşam döngüsü metodunun adı önceden componentWillMount şeklindeydi. Bu isim, React'in 17 sürümüne kadar çalışmaya devam edecektir. Bileşenlerinizi otomatik olarak güncellemek için, rename-unsafe-lifecycles komutunu kullanabilirsiniz.

UNSAFE_componentWillMount() metodu, bileşenin DOM'a eklenmesinden önce çağrılır. render() metodundan önce çalıştırıldığından dolayı, bu metot içerisinde setState() metodunun senkron olarak çağrılması ekstra bir render işlemini tetiklememektedir. Bunun yerine, state'in başlatılması için constructor() metodunu kullanmanızı tavsiye ederiz.

Bu metot içerisinde herhangi bir yan etki içeren işlem veya abonelik işlemleri yapmaktan kaçınınız. Bu tür yaklaşımlar için componentDidMount() metodunu kullanınız.

Sunucu taraflı render etme işleminde yalnızca bu yaşam döngüsü metodu çağrılır.

UNSAFE_componentWillReceiveProps()

UNSAFE_componentWillReceiveProps(nextProps)

Not:

Bu yaşam döngüsü metodunun adı önceden componentWillReceiveProps şeklindeydi. Bu isim, React'in 17 sürümüne kadar çalışmaya devam edecektir. Bileşenlerinizi otomatik olarak güncellemek için, rename-unsafe-lifecycles komutunu kullanabilirsiniz.

Not:

Bu yaşam döngüsü metodunun kullanımı, uygulamanın hatalı ve tutarsız çalışmasına sebep olacaktır.

Eğer verilerin getirilmesi veya bir animasyonun uygulanması gibi bir yan etki gerçekleştirmek istiyorsanız, props'ta oluşan değişikliklere yanıt vermek için componentDidUpdate metodunu kullanınız.

Sadece bir prop değiştiğinde bazı verilerin tekrar hesaplanması için componentWillReceiveProps metodunu kullandıysanız, bunun yerine memoization helper kullanınız.

Eğer bir prop değiştiğinde bazı state değişkenlerini componentWillReceiveProps metodu ile yeniden başlattıysanız, bunun yerine tamamen kontrollü bir bileşen veya bir key'e sahip tamamen kontrolsüz bir bileşen oluşturabilrsiniz.

Diğer kullanım durumları için, state'in türetilmesi ile ilgili bu blog yazısını inceleyebilirsiniz.

DOM'a eklenmiş bir bileşen, yeni prop değerlerini almasından hemen önce UNSAFE_componentWillReceiveProps() metodu çağrılır. Prop değişikliklerine göre state'i güncellemeniz (örneğin state'i yeniden başlatmanız) gerekiyorsa, this.props ve nextProps değerlerini karşılaştırabilir ve bu metot içerisinde this.setState()'i kullanarak state geçişlerini gerçekleştirebilirsiniz.

Unutmayınız ki, eğer bir üst bileşen, altındaki bileşenin tekrar render edilmesine sebep oluyorsa, props değerleri değişmemiş olsa bile bu metot çağrılır. Bu nedenle oluşan değişiklikleri yönetmek için, mevcut değer ile ve sonraki değerleri karşılaştırınız.

React, bileşenin DOM'a eklenmesi sırasında, başlangıç prop değerleri ile UNSAFE_componentWillReceiveProps() metodunu çağırmaz. Yalnızca bileşenin prop değerleri değiştiğinde bu metodu çalıştırır. Bunun haricinde this.setState()'in çağrılması da UNSAFE_componentWillReceiveProps() metodunun çağrılmasını tetiklemez.

UNSAFE_componentWillUpdate()

UNSAFE_componentWillUpdate(nextProps, nextState)

Not:

Bu yaşam döngüsü metodunun adı önceden componentWillUpdate şeklindeydi. Bu isim, React'in 17 sürümüne kadar çalışmaya devam edecektir. Bileşenlerinizi otomatik olarak güncellemek için, rename-unsafe-lifecycles komutunu kullanabilirsiniz.

UNSAFE_componentWillUpdate() metodu, yeni bir prop veya state değeri alındığında, render işleminin hemen öncesinde çağrılır. Bu fırsatı değerlendirerek, güncelleme oluşmadan önce ilgili hazırlıkları yapabilirsiniz. Bu metot, ilk render işleminde çalıştırılmaz.

Unutmayınız ki, bu fonksiyon içerisinde, UNSAFE_componentWillUpdate() metodunun geriye değer döndürmesinden önce, React bileşeninin güncellenmesini tetikleyecek; this.setState()'i veya herhangi bir metodu (örneğin Redux action'ının dispatch edilmesini) çağıramazsınız.

Genellikle bu metot, componentDidUpdate() metodu ile değiştirilebilir. Eğer bu metot içerisinde DOM'dan bir değer okuması yapıyorsanız (örneğin kaydırma çubuğu pozisyonunu kaydediyorsanız), bu kodları getSnapshotBeforeUpdate()'e taşıyabilirsiniz.

Not:

Eğer shouldComponentUpdate() metodu false döndürüyorsa, UNSAFE_componentWillUpdate() metodu çağrılmaz.

Diğer API'lar

React'in sizin için otomatik olarak çağırdığı yukarıdaki yaşam döngüsü metotlarının aksine, aşağıda yer alan metotları sadece siz çağırabilirsiniz.

Yalnızca iki adet metot vardır. Bunlar setState() ve forceUpdate() metotlarıdır.

setState()

setState(updater[, callback])

setState() metodu, bileşenin state'inde olan değişiklikleri bir kuyruğa atar ve React'e, bu bileşenin ve alt bileşenlerinin güncellenen state ile birlikte tekrar render edilmesi gerektiğini bildirir. Sunucu cevapları ve onClick gibi olay gidericilerinden dönen değişikliklerin arayüze yansıtılması için başlıca metottur.

setState(), bir bileşeni direkt olarak güncelleyen bir metot değildir. Bu nedenle setState()'i, React'e yapılan bir istek olarak düşünmelisiniz. React, daha iyi bir performans için bu metodun çalışmasını geciktirebilir, ve daha sonra tüm güncellemeler ile birlikte tek seferde gerçekleştirebilir. Bu nedenle React, state güncellemelerinin anında gerçekleştirileceğini garanti etmemektedir. Nadiren, DOM güncellemelerinin anında ve senkronize bir şekilde olmasına ihtiyacınız olabilir. Bu durumda state güncellemelerini flushSync ile çevreleyebilirsiniz. Unutmayın bu performası olumsuz yönde etkileyebilir.

setState() metodu, her zaman bileşeni anında güncellemez. Güncellemeleri yığın haline getirebilir veya daha sonra gerçekleşmesi için geciktirebilir. Bu nedenle, setState() çağrımından sonra this.state değerinin okunması yaygın olarak yapılan bir yanlıştır. Bunun yerine, componentDidUpdate metodunu veya setState callback'ini (setState(updater, callback)) kullanınız. Her iki kullanım da, güncellemeler uygulandıktan sonra kodun çalıştırılacağını garanti eder. Eğer mevcut state'i, önceki state'e göre güncellemeye ihtiyacınız varsa, aşağıda yer alan updater parametresini inceleyebilirsiniz.

setState(), shouldComponentUpdate() metodu false dönmediği sürece, ilgili bileşenin tekrar render edilmesini sağlar. Eğer değiştirilebilir (mutable) nesneler kullanılırsa ve buna bağlı olarak shouldComponentUpdate() içerisinde koşullu render'lama (conditional rendering) mantığı kurulamazsa, önceki state'ten yeni state'in farklı olduğu durumda yalnızca setState() çağrımı gereksiz render işlemini

gerçekleştirmeyecektir.

setState()'in ilk parametresi bir updater fonksiyonudur ve aşağıdaki gibi yer almaktadır:

```
(state, props) => stateChange
```

state, değişikliğin uygulandığı andaki bileşenin state'ini tutmaktadır. Değişiklikler, state ve props girdilerini baz alan yeni bir nesne oluşturularak temsil edilmelidir. Örneğin, state'teki bir değeri, props.step değeri ile arttırdığımızı varsayalım:

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

Bu kodda, updater fonksiyonuna aktarılan state ve props değerlerinin güncel olduğu garanti edilir. Updater'ın çıktısı, state ile yüzeysel olarak birleştirilir.

setState()'in ikinci parametresi, setState çağrımı tamamlandığında ve bileşen tekrar render edildiğinde bir defa çağrılacak olan ve isteğe bağlı olarak tanımlanan callback fonksiyonudur. Bunun yerine genellikle componentDidUpdate() metodunun kullanılmasını öneririz.

İsteğe bağlı olarak, setState()'in ilk parametresi için, bir fonksiyon yerine aşağıdaki gibi bir nesne geçebilirsiniz:

```
setState(stateChange[, callback])
```

Bu, stateChange'in yüzeysel olarak state ile birleştirilmesini sağlar. Örneğin bir alışveriş sepetindeki ürünün adedini aşağıdaki gibi güncelleyebilirsiniz:

```
this.setState({quantity: 2})
```

Ayrıca setState()'in bu şekilde kullanımı asenkron olarak çalışır. Bu nedenle aynı render döngüsünde birkaç defa yapılan setState() çağrıları, tekil hale getirilerek işlenebilir. Örneğin, aynı render döngüsünde ürün adedini birden fazla kez arttırmaya çalışırsanız, işlemin sonucu Object.assign() ile tekil bir metot çağrısı haline getirilecek ve aşağıdaki gibi olacaktır:

```
Object.assign(
  previousState,
  {quantity: state.quantity + 1},
  {quantity: state.quantity + 1},
  ...
```

```
)
```

Aynı render döngüsünde art arda yapılan çağrımlar, önceki çağrımdan gelen değişiklikleri ezerek üstüne yazacaktır. Böylece quantity değeri yalnızca bir kez arttırılacaktır. Bu nedenle eğer sonraki state, mevcut state'e bağımlı ise, aşağıdaki gibi bir updater fonksiyonu kullanmanızı öneririz:

```
this.setState((state) => {
  return {quantity: state.quantity + 1};
});
```

State ve Yaşam Döngüsü Rehberi

Derinlemesine inceleme: setState() çağrıları neden ve ne zaman tekil hale getirilirler?

Derinlemesine inceleme: this.state neden anında güncellenmez?

```
forceUpdate()
```

```
component.forceUpdate(callback)
```

Bileşeninizin state'i veya prop'ları değiştiğinde, varsayılan olarak bileşeniniz tekrar render edilecektir. Eğer render() metodunuz, bunların haricinde başka verilere bağımlı ise, forceUpdate()'i çağırarak bileşeninizin tekrar render edilmesi gerektiğini React'e söyleyebilirsiniz.

Bileşen üzerinde forceUpdate() çağrımı, shouldComponentUpdate()'in es geçilerek render() metodunun çalışmasına neden olacaktır. forceUpdate() çağrımı, alt bileşenler için normal yaşam döngüsü metotlarını tetikleyecektir. Bu metotlara, her bir alt bileşen için çağrılacak shouldComponentUpdate() metodu da dahildir. Buna rağmen HTML tarafında oluşan değişikliklerde, React sadece DOM'ı güncellemeye devam edecektir.

Normalde forceUpdate()'in kullanımından kaçınmalı ve yalnızca render() metodu içerisinde this.props ve this.state'ten okuma işlemi yapmalısınız.

Sınıf Bileşeninin Değişkenleri

defaultProps

defaultProps, bileşen sınıfının varsayılan prop değerlerini atamak için sınıf içerisinde özellik olarak tanımlanabilir. Bu değişken, tanımlı olmayan prop değerleri (undefined) için kullanılır. null değeri içeren prop'lar için kullanılmaz. Örneğin:

```
class CustomButton extends React.Component {
  // ...
}
```

```
CustomButton.defaultProps = {
  color: 'blue'
};
```

Eğer bileşene props.color değeri aktarılmazsa, varsayılan olarak 'blue' atanacaktır:

```
 render() {
   return <CustomButton /> ; // props.color değeri blue olarak atanacaktır
 }
```

props.color değeri null olarak atanmışsa, değişmeden null olarak kalacaktır:

```
 render() {
   return <CustomButton color={null} /> ; // props.color değeri null olarak kalacaktır
 }
```

displayName

displayName değişkeni, hata ayıklama mesajlarında kullanılır. Genellikle açık olarak bu değişkene atama yapmanıza gerek yoktur. Çünkü tanımlandığı fonksiyon veya sınıf bileşeninin isminden oluşturulmaktadır. Hata ayıklama işleminde farklı bir isim kullanmak istiyorsanız, elbette bu değişkeni açık bir şekilde tanımlayıp atama yapabilirsiniz. Ayrıca high-order bir bileşen oluştururken de kullanabilirsiniz. Kolay bir şekilde hata ayıklama için görünen ismin değiştirilmesi yazısından daha detaylı bilgi edinebilirsiniz.

Bileşen Nesnesinin Değişkenleri

props

this.props, bu değişkeni çağıran eleman tarafından tanımlanan prop değerlerini içerir. Prop'lara giriş olması açısından fazla bilgi için Bileşenler ve Prop'lar yazısını inceleyebilirsiniz.

Bilhassa, this.props.children özel bir prop'tur. Genellikle etiketin kendisi yerine JSX ifadesindeki alt etiketler tarafından tanımlanır.

state

State, zaman içerisinde değişim gösterebilen ve ilgili değişkene özgü olan verileri tutar. State değişkeni kullanıcı tarafından tanımlanır, ve düz bir JavaScript nesnesi olmalıdır.

Eğer state'te tanımlanan bazı değerler, render işleminde veya zamanlayıcı ID'sinin tutulması gibi veri akışına yönelik işlemlerde kullanılmıyorsa, bu değişkenler state içerisine konulmamalıdır. Bu değerler,

bileşen nesnesinde değişken olarak tanımlanabilirler.

State hakkında daha fazla bilgi için State ve Lifecycle sayfasına göz atabilirsiniz.

this.state'i direkt olarak değiştirmeyiniz. Çünkü daha sonra yapılan setState() çağrıları, yaptığınız değişikliklerin üzerine yazabilir. Bu nedenle this.state'e immutable'mış gibi davranmalısınız.