Hook'lara Giriş

Hook’lar React 16.8’deki yeni bir eklentidir. Bir sınıf yazmadan state ve diğer React özelliklerini kullanmanıza olanak sağlarlar.

```
import React, { useState } from 'react';

function Example() {
 // "count" diyeceğimiz yeni bir state değişkeni tanımlayın
 const [count, setCount] = useState(0);
 return (
  <div>
   <p>You clicked {count} times</p>
   <button onClick={() => setCount(count + 1)}>
    Click me
   </button>
  </div>
 );
}
```

Bu yeni useState fonksiyonu öğreneceğimiz ilk “Hook”, ancak bu örnek sadece kısa bir tanıtım. Henüz bir anlam ifade etmiyorsa endişelenmeyin!

Bir sonraki sayfada Hook’ları öğrenmeye başlayabilirsiniz. Bu sayfada, React’e Hook’ları neden eklediğimizi ve harika uygulamalar yazmanıza nasıl yardımcı olabileceklerini açıklayarak devam edeceğiz.

Not:

React 16.8.0, Hook’ları destekleyen ilk sürümdür. Sürüm yükseltme yaparken, React DOM dahil olmak üzere tüm paketleri güncellemeyi unutmayın. React Native, Hook’ları 0.59 sürümünden itibaren desteklemektedir.

Mevcut Kodu Bozan Değişiklikler Yok

Devam etmeden önce, unutmayın ki Hook’lar:

Tamamen opsiyoneldir. Hook'ları mevcut herhangi bir kodu tekrar yazmadan birkaç bileşende deneyebilirsiniz. Fakat istemiyorsanız şu anda Hook'ları öğrenmek veya kullanmak zorunda değilsiniz.

100% geriye uyumludur. Hook'lar mevcut kodu bozan herhangi bir değişiklik içermiyor.

Şu an kullanılabilir. Hook'lar v16.8.0 sürümünün yayımlanması ile şu an kullanıma uygundur.

React'ten sınıfları kaldırmak gibi bir planımız yok. Hook'lar için kademeli kabul stratejisi hakkında daha fazla bilgiyi bu sayfanın alt kısmında bulabilirsiniz.

Hook'lar, React kavramları hakkındaki bilgilerinizin yerini almaz. Bunun yerine, Hook'lar zaten bildiğiniz React kavramlarına (props, state, context, refs ve lifecycle) daha doğrudan bir API sağlar. Daha sonra göstereceğimiz gibi, Hook'lar bunları bir araya getirmek için yeni bir güçlü yol sunar.

Sadece Hook'ları öğrenmeye başlamak istiyorsanız, doğrudan bir sonraki sayfaya atlamaktan çekinmeyin! Ayrıca neden Hook'ları eklediğimizi ve uygulamalarımızı yeniden kodlamadan nasıl kullanmaya başlayacağımızı öğrenmek için bu sayfayı okumaya devam edebilirsiniz

Motivasyon

Hook'lar, React'te beş yıldan fazla bir süredir yazdığımız ve on binlerce bileşenin bakımını yaptığımız çok çeşitli görünüşte birbirinden bağımsız sorunları çözüyor. React'i öğreniyor, günlük hayatınızda kullanıyor veya hatta benzer bir bileşen modeline sahip farklı bir kütüphaneyi tercih ediyor da olsanız, bu sorunların bazılarını fark edebilirsiniz.

Bileşenler arasındaki stateful logic'i yeniden kullanmak zor

React, bir bileşene yeniden kullanılabilir davranışları "iletmenin" bir yolunu sunmaz (örneğin, bir store'a bağlamak). Bir süre React ile çalıştıysanız, bu sorunu çözmeye çalışan prop'ları render etme ve higher-order bileşenler gibi kalıplara aşina olabilirsiniz. Ancak bu modeller, bileşenlerinizi kullandıkça yeniden yapılandırmanızı gerektirir ki bu da kullanışsızdır ve kodun takip edilmesini zorlaştırır. React DevTools'ta tipik bir React uygulamasına bakarsanız, büyük olasılıkla; sağlayıcılar, tüketiciler, higher-order bileşenler, prop'ları render etme ve diğer soyutlamala katmanları ile çevrili bileşenlerin "wrapper hell" problemini bulacaksınız. Bunları DevTools'ta filtreleyebiliyor olsak da, bu daha derin bir soruna işaret ediyor: React'e, stateful logic'i paylaşmak için daha iyi bir genel çözüm gerekli.

Hook'lar ile, bir bileşenden stateful logic çıkarabilir, böylece bileşen bağımsız olarak test edilebilir ve yeniden kullanılabilir. Hook'lar, bileşen hiyerarşinizi değiştirmeden stateful logic'i yeniden kullanmanıza olanak sağlar. Bu, Hook'ları birçok bileşen arasında veya toplulukla paylaşmayı kolaylaştırır.

Buna daha çok Kendi Hook'larınızı Oluşturma bölümünde değineceğiz.

Karmaşık bileşenlerin anlaşılması zorlaşıyor

Basit bir şekilde başlayan, ancak yönetilemez bir stateful logic ve yan etki karmaşasına dönüşen bileşenlerin bakımını sıkça yapmak zorunda kaldık. Her yaşam döngüsü metodu çoğu zaman alakasız bir

mantık karışımı içerir. Örneğin, bileşenler componentDidMount ve componentDidUpdate içerisinde bazı verileri getirebilir. Bununla birlikte, aynı componentDidMount yöntemi de, componentWillUnmount içerisinde gerçekleştirilen temizleme işlemiyle olay dinleyicilerini ayarlayan alakasız bir mantık içerebilir. Birlikte değişen karşılıklı ilişkili kod parçalanır, ancak tamamen ilişkisiz kod tek bir metotta bir araya gelmiş olur. Bu, hataları ve tutarsızlıkları ortaya çıkarmayı çok kolaylaştırır.

Çoğu durumda, bu bileşenleri daha küçük parçalara bölmek mümkün değil çünkü stateful logic darmadağın durumdadır. Test etmek de zordur. Bu, birçok insanın React'i ayrı bir state yönetimi kütüphanesiyle kullanmayı tercih etmelerinin nedenlerinden biridir. Ancak, bu genellikle çok fazla soyutlama getirir, farklı dosyalar arasında atlamanızı gerektirir ve bileşenleri yeniden kullanmayı daha da zorlaştırır.

Bunu çözmek için, yaşam döngüsü metotlarını baz alan bir ayrımı zorlamak yerine Hook'lar, bir bileşeni hangi parçalarla ilgili olduğunu (bir abonelik ayarlamak veya veri almak gibi) baz alarak daha küçük fonksiyonlara ayırmanıza olanak tanır. Ayrıca, daha öngörülebilir hale getirmek için bileşenin state durumunu bir reducer ile yönetmeyi de seçebilirsiniz.

Buna daha çok Efekt Hook'unu Kullanma bölümünde değineceğiz.

Sınıflar hem insanların hem de makinelerin kafasını karıştırıyor

Sınıfların, kodun yeniden kullanılmasını ve kod organizasyonunu zorlaştırmasının yanı sıra, React'i öğrenme konusunda büyük bir engel olabileceğini gördük. Bunun JavaScript'te nasıl çalıştığını anlamalısınız, bu birçok dilde nasıl çalıştığından çok farklı. Olay yöneticilerini bağlamayı için hatırlamanız gereklidir. ES2022 public class fields olmadan, kod çok fazla ayrıntılıdır. İnsanlar prop'ları, state'i ve yukarıdan aşağıya veri akışını mükemmel bir şekilde anlayabilir, ancak yine de sınıfları anlamak için çaba sarfedebilir. React'teki fonksiyon ve sınıf bileşenleri arasındaki ayrım ve her birinin ne zaman kullanılacağı, deneyimli React geliştiricileri arasında bile anlaşmazlıklara yol açar.

Ayrıca, React 5 yıldır mevcut ve önümüzdeki 5 yıl için de kalıcı olacağından emin olmak istiyoruz. Svelte, Angular, Glimmer ve diğerlerinin gösterdiği gibi, bileşenlerin önceden yapılmış derlemeler çok fazla gelecek potansiyeli var. Özellikle de şablonlarla sınırlı değilse. Son zamanlarda, Prepack kullanarak bileşen katlama ile deneme yaptık ve ilk sonuçların umut verici olduğunu gözlemledik. Bununla birlikte, sınıf bileşenlerinin bu optimizasyonları daha yavaş bir yola geri çekmesini sağlayan istemsiz kalıpları teşvik edebileceğini gördük. Sınıflar da bugünün araçları için sorunlar sunmaktadır. Örneğin, sınıflar çok küçültmezler ve hot reloading'i tuhaf ve güvenilmez yaparlar. Kodun optimize edilebilir yolda kalmasını daha kolay kılan bir API sunmak istiyoruz.

Bu sorunları çözmek için, Hook'lar, sınıfsız şekilde React'in özelliklerini daha fazla kullanmanızı sağlar. Kavramsal olarak, React bileşenleri her zaman fonksiyonlara daha yakın olmuştur. Hook'lar React'in pratik ruhundan ödün vermeden. fonksiyonları kucaklar. Hook'lar, mecburi kaçış kapaklarına erişim sağlar ve karmaşık fonksiyonel veya reaktif programlama tekniklerini öğrenmenizi gerektirmez.

Örnekler

Bir Bakışta Hook'lar, Hook'ları öğrenmeye başlamak için iyi bir yer.

Kademeli Kabul Stratejisi

Kısaca: Sınıfları React'ten kaldırmak gibi bir planımız yok.

React geliştiricilerinin ürünlerini teslim etmeye odaklandığını ve yayımlanan her yeni API'yı incelemeye zamanlarının olmadığını biliyoruz. Hook'lar çok yeni ve öğrenmeyi veya benimsemeyi düşünmeden önce daha fazla örnek ve öğretici beklemek daha iyi olabilir.

Ayrıca, React'e yeni bir ilkel (primitive) eklemek için çıtanın son derece yukarıda olduğunun farkındayız. Meraklı okuyucular için, daha detaylıca motivasyonu işleyen ve belirli tasarım kararları ve önceki ilgili teknikler hakkında ekstra perspektif sağlayan detaylı RFC'i hazırladık.

Önemli olan Hook'lar, mevcut kodla yan yana çalışır, böylece bunları aşamalı olarak kullanabilirsiniz. Hook'lara geçmeniz için acele etmenize gerek yok. Özellikle mevcut, karmaşık sınıf bileşenlerinizi "yeniden yazmak"tan kaçınmanızı öneririz. "Hook'ları anlamak" biraz zaman alabilir. Tecrübelerimize göre, Hook'ları öncelikle yeni ve kritik olmayan bileşenlerde kullanarak pratik yapmak ve ekibinizdeki herkesin kendisini bunu konuda rahat hissetmesini sağlamak en iyisi. Hook'ları denedikten sonra, lütfen bize olumlu ya da olumsuz geri bildirim gönderin.

Hook'ların sınıflar için mevcut tüm kullanım durumlarını kapsamasını istiyoruz, ancak öngörülebilir gelecek için sınıf bileşenlerini desteklemeye devam edeceğiz. Facebook'ta, sınıf olarak yazılmış on binlerce bileşene sahibiz ve bunları yeniden yazmak için kesinlikle hiçbir planımız yok. Bunun yerine, yeni kodda Hook'ları yan yana sınıflarla kullanmaya başlıyoruz.

İlk Bakışta Hook'lar

Hook'lar React 16.8'deki yeni bir eklentidir. Bir sınıf yazmadan state ve diğer React özelliklerini kullanmanıza olanak sağlar.

Hook'larda mevcut kodu bozan değişiklikler yok. Bu sayfa, tecrübeli React kullanıcılarına Hook'lar hakkında genel bir fikir sağlar. Bu hızlı bir gözden geçirme demektir. Eğer kafanız karışırsa bu tarz bir sarı kutu arayın:

Detaylı açıklama

Neden Hook'ları çıkardığımızı anlamak için Motivasyon bölümünü okuyun.

↑↑↑ Her bölüm bunun gibi bir sarı kutuyla biter Bunlar detaylı açıklamaların nerede bulunacağını gösterir.

📌 State Hook'u

Bu örnek bir sayaç render ediyor. Tuşa basıldığında değeri bir arttırıyor:

```
import React, { useState } from 'react';
```

```
function Example() {
  // Yeni bir state değişkeni belirlenir, biz buna "count" diyeceğiz.
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Burada, useState bir Hook (birazdan bunun ne demek olduğuyla alakalı konuşacağız). Bu fonksiyonu; fonksiyonel bir bileşene, yerel bir state eklemek amacıyla, bu bileşenin içerisinde çağırıyoruz. React bu state'i yenilenen render'lar arasında muhafaza edecek. useState bir çift döndürür: anlık state değeri ve bunu değiştirmenize yarayan bir fonksiyon. Bu fonksiyonu bir olay yöneticisinde veya başka bir yerde çağırabilirsiniz. Bu, class'lardaki this.setState fonksiyonuna benzer, fakat eski ve yeni state'i birleştirmez. (useState ve this.state farklarını State Hook'unu Kullanmak bölümünde göstereceğiz.)

useState'in aldığı tek argüman başlangıçtaki state'dir. Yukarıdaki örnekte bu argüman 0, çünkü sayacımız sıfırdan başlıyor. this.state'ten farklı olarak state'in bir obje olması gerekmediğine dikkat edin — tabi isterseniz obje de kullanabilirsiniz. Başlangıç state argümanı sadece ilk render'da kullanılır.

Birden fazla state değişkeni tanımlamak

State Hook'unu tek bir bileşende, birden fazla kullanabilirsiniz:

```
function ExampleWithManyStates() {
  // Declare multiple state variables!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
```

```
  // ...
}
```

Dizi parçalama (array destructuring) syntax'i useState kullanarak tanımladığımız state değişkenlerine farklı isimler vermemize olanak tanır. Bu isimler useState API'nin bir parçası değildir. Bunun yerine; eğer useState'i çok fazla çağırırsanız, React her render'da aynı sırayla çağırdığınızı var sayar. Bunun niye çalıştığına ve nasıl kullanışlı olacağına ileride değineceğiz.

Peki bir hook nedir?

Hook'lar React state ve yaşam döngüsü özelliklerine fonksiyonel bileşenleri kullanarak "bağlamanıza" yarayan fonksiyonlardır. Hook'lar class'ların içerisinde çalışmazlar — React'ı class'lar olmadan kullanmanıza yararlar. (Var olan bileşenlerinizi bir gecede tekrar yazmanızı önermiyoruz fakat yeni bileşenleriniz için Hook'ları kullanmaya başlayabilirsiniz.)

React üzerinde useState gibi bir kaç Hook bulunmaktadır. Ayrıca siz de state'le alakalı davranışlarınızın, farklı bileşenler tarafından yeniden kullanılması için özel Hook'larınızı yazabilirsiniz. Öncelikle React üzerinde var olan Hook'ları inceleyeceğiz.

Detaylı açıklama

State Hook'u hakkında daha fazla bilgiye bu sayfadan ulaşabilirsiniz: State Hook'unu Kullanmak.

⚡ Effect Hook'u

Yüksek ihtimalle daha öncesinde data çekme, dışarıya bağlanma ya da DOM'u elle değiştirme gibi işlemleri React bileşenleri kullanarak yapmışsınızdır. Bu tarz işlemleri "yan etkiler(side effects)" (veya kısaca "etkiler") olarak adlandırıyoruz çünkü başka bileşenleri etkileyebiliyorlar ve render sırasında yapılamayan işlemler oluyorlar.

Effect Hook'u; useEffect, fonksiyonel bir bileşene yan etkileri kullanabilme yetkisini ekler. React class'larındaki componentDidMount, componentDidUpdate, ve componentWillUnmount ile aynı işleve sahiptir fakat tek bir API içerisinde birleştirilmiştir. (useEffect ve bu metodların farklarını örneklerle Effect Hook'unu kullanmak bölümünde göstereceğiz.)

Örneğin, bu bileşen html dosyasının başlığını React DOM'u güncelledikten sonra değiştirir:

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // componentdidMount ve componentDidUpdate'e benzer bir şekilde:
  useEffect(() => {
```

```
  // Browser API kullanılarak document title güncellenir
  document.title = `You clicked ${count} times`;
 });
 return (
  <div>
   <p>You clicked {count} times</p>
   <button onClick={() => setCount(count + 1)}>
    Click me
   </button>
  </div>
 );
}
```

useEffect'i çağırdığınız zaman, React değişiklikleri DOM'a ilettikten sonra "effect" fonksiyonunu çağırmasını söylüyorsunuz. Effect'ler bileşenin içerisinde tanımlandığından state ve prop'lara erişebiliyor. Varsayılan şekliyle React effect'leri her render sonrasında çalıştırır — ilk render da bunların içerisinde. (Class yaşam döngüleriyle farkını detaylı olarak Effect Hook'unu kullanmak bölümünde işleyeceğiz.)

İsteğe bağlı olarak Efect'lerin nasıl kendi "arkalarını toplayacakları", bir fonksiyon döndürülerek belirtilebilir. Örneğin, bu bileşen bir effect kullanarak, bir arkadaşın online bilgisine bağlanıyor ve kendi arkasını bu bağlantıyı kapatarak topluyor:

```
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
 const [isOnline, setIsOnline] = useState(null);
 function handleStatusChange(status) {
  setIsOnline(status.isOnline);
 }
 useEffect(() => {
  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
```

```
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

Bu örnekte, bileşen hem unmount anında hem de sonraki render yüzünden effect'i tekrar çalıştırmadan önce, React ChatAPI'ımızla bağlantıyı kesiyor. (eğer ChatAPI'a verilen props.firend.id değişmediyse, React'a tekrar bağlantı kurmamasını söyleyebilirsiniz.)

useState'de olduğu gibi tek bir bileşende birden fazla effect kullanabilirsiniz:

```
function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
  function handleStatusChange(status) {
```

```
  setIsOnline(status.isOnline);
}
// ...
```

Hook'lar bir bileşen içerisindeki yan etkileri yaşam döngüsü metodlarına ayırmaktansa, hangi parçaların etkilendğine bağlı olarak bu yan etkileri organize etmenize yarar.

Detaylı Açıklama

useEffect hakkında daha fazla bilgiye bu sayfadan ulaşabilirsiniz: Effect Hook'unu kullanmak.

🤞 Hook'ların Kuralları

Hook'lar JavaScript fonksiyonlarıdır ama ek olarak iki kural koymaktadırlar:

Hook'ları sadece en üst seviyede çağırın. Hook'ları döngülerin, koşulların veya iç içe fonksiyonların içerisinde çağırmayın.

Hook'ları sadece fonksiyonel React bileşenlerinde çağırın. Normal JavaScript fonksiyonları içerisinde Hook'ları çağırmayın. (Hook'ları başka çağırabileceğiniz tek bir uygun yer var — kendi yarattığınız Hook'lar. Birazdan bunlar hakkında daha fazla şey öğreneceğiz.)

Bu kuralları otomatik bir şekilde yürütmek için bir linter eklentisi sağlıyoruz. Bu kuralların ilk bakışta sınırlayıcı ve kafa karıştırıcı görünebileceğini anlıyoruz fakat Hook'ların düzgün çalışması için hepsi çok önemlidir.

Detaylı Açıklama

Bu kurallar hakkında daha fazla bilgiye bu sayfadan ulaşabilirsiniz: Hook Kuralları.

💡 Özel Hook'larınızı Yapmak

Bazen state'le alakalı bazı davranışların bileşenler arasında yeniden kullanılabilir olmasını isteriz. Geleneksel olarak bunun için iki tane popüler çözüm vardır: üst-seviye bileşenler ve render prop'ları. Özel Hook'larınız da bunu yapmanıza izin verir ve bunu yaparken bileşen ağacınıza daha fazla bileşen eklemek zorunda kalmazsınız.

Daha öncesinde useState ve useEffect kullanarak bir arkadaşın online durumuna bağlanan FriendStatus adlı bir bileşen tanıtmıştık. Bu bağlantı davranışını başka bir bileşende tekrar kullanmak istediğimizi varsayalım.

Önce, bu davranışı kendi yarattığımız useFriendStatus adlı bir Hook'a aktaracağız:

```
import React, { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
```

```
  const [isOnline, setIsOnline] = useState(null);
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }
  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });
  return isOnline;
}
```

Argüman olarak friendID alıyor, ve arkadaşımızın online olup olmadığını döndürüyor.

Artık iki bileşenden de bu davranışı kullanabiliriz:

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);
  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);
  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
```

```
    {props.friend.name}
   </li>
  );
}
```

Her bileşenin state'i birbirinden tamamen bağımsızdır. Hooklar state'le alakalı davranışların tekrar kullanılmasının bir yoludur, state'in yeniden kullanılmasıyla alakalı değildir. Hatta bir Hook her çağırıldığında tamamen ayrı bir state'e sahiptir — bu sayede özel Hook'unuzu bir bileşen içerisinde iki kere çağırabilirsiniz.

Özel Hook'larınız bir özellikten daha çok bir kural gibidir. Eğer bir fonksiyonun adı "use" ile başlıyor ve başka Hook'ları çağırıyorsa, bu fonksiyon bir özel Hook'tur diyoruz. useSomething adlandırması linter eklentimizin, Hook kullanılarak yazılan kodda bugları bulmasını sağlıyor.

Özel Hook'larınızı bizim bahsetmediğimiz bir çok durum için kullanabilirsiniz, örneğin: form yönetimi, animasyon, tanımsal bağlantılar, zamanlayıcılar ve aklımıza gelmeyen bir çok farklı durum. React topluluğunun ne tür özel Hook'lar üreteceğini sabırsızlıkla bekliyoruz.

Detaylı Açıklama

Özel Hook'lar hakkında daha fazla bilgiye bu sayfadan ulaşabilirsiniz: Kendi Hook'larınızı Oluşturmak.

🔌 Diğer Hook'lar

React içerisinde bulunan ve daha az kullanılan ama yararlı olabilecek bir kaç Hook daha bulunuyor. Örneğin, useContext iç içe geçmiş bileşenler kullanmadan, React context'e bağlanmanızı sağlar:

```
function Example() {
  const locale = useContext(LocaleContext);
  const theme = useContext(ThemeContext);
  // ...
}
```

Ve useReducer karmaşık bileşenlerinizin yerel state'ini bir reducer olmadan yönetmenizi sağlar:

```
function Todos() {
  const [todos, dispatch] = useReducer(todosReducer);
  // ...
```

State Hook Kullanımı

Giriş Sayfasında, Hook'lara aşina olmak için, aşağıdaki örnek kullanılmıştık.

```
import React, { useState } from 'react';

function Example() {
  // "count" adında yeni bir state değişkeni tanımlayın.
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Bu kodu, eşdeğeri olan sınıf koduyla karşılaştırarak Hook'ları öğrenmeye başlayacağız.

Eşdeğer Sınıf Örneği

Eğer React'te sınıfları kullandıysanız, bu kod size tanıdık gelecektir.

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
```

```
  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

State { count: 0 } olarak başlar ve kullanıcı her tıkladığında this.setState() çağırılarak state.count arttırılır. Sayfa boyunca bu sınıftan parçalar kullanacağız.

Not

Neden daha gerçekçi bir örnek yerine sayaç kullandığımızı merak ediyor olabilirsiniz. Bu, Hook'lar ile ilk adımlarınızı atarken API'a odaklanmanıza yardımcı olacaktır.

Hook'lar ve Fonksiyon Bileşenleri

React'teki fonksiyon bileşenlerinin böyle gözüktüğünü unutmayınız:

```
const Example = (props) => {
  // Hook'ları burada kullanabilirsiniz!
  return <div />;
}
```

ya da bunu:

```
function Example(props) {
  // Hook'ları burada kullanabilirsiniz!
  return <div />;
```

```
}
```

Yukarıdakileri “durumsuz (stateless) bileşenler” olarak biliyor olabilirsiniz. Şimdi bu bileşenlere React state’ini kullanma özelliğini getiriyoruz; bu yüzden bunlara “fonksiyon bileşenleri” demeyi tercih ediyoruz.

Hook’lar sınıfların içinde çalışmazlar fakat onları sınıf yazmadan da kullanabilirsiniz.

Hook Nedir?

Yeni örneğimize React’ten useState hook’u import etmekle başlıyoruz

```
import React, { useState } from 'react';

function Example() {
  // ...
}
```

Hook Nedir? Hook, React özelliklerini “bağlamanıza” izin veren özel bir fonksiyondur. Örneğin useState, React state’ini fonksiyon bileşenlerine eklemenize izin veren bir Hook’tur. Yakında diğer Hook’ları da öğreneceğiz.

Ne zaman bir Hook kullanmalıyım? Eğer bir fonksiyon bileşeni yazarsanız ve ona biraz state eklemeniz gerektiğini farkederseniz, bundan önce o fonksiyonu bir sınıfa (class) dönüştürmeniz gerekiyordu. Fakat şimdi, varolan fonksiyon bileşenlerinin içinde Hook kullanabilirsiniz. Şimdi tam olarak bunu yapacağız!

Not:

Hook’ları bileşenlerin içinde kullanıp kullanamayacağımız hakkında birkaç özel kural vardır. Bunları Hook Kuralları sayfasında öğreneceğiz.

Bir State Değişkeni Tanımlamak

Bir sınıfın içinde, count state’ini constructor (yapıcı fonksiyon) içinde this.state‘i { count: 0 }‘a eşitleyerek 0 olarak başlatıyoruz.

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
```

```
  };
}
```

Fonksiyon bileşenlerinde this yoktur; bu yüzden this.state'e değer ataması veya this.stateten okuma yapamıyoruz. Bunun yerine bileşenimizin içinde direkt olarak useState hook'unu çağırıyoruz.

```
import React, { useState } from 'react';

function Example() {
  // "count" adında yeni bir state değişkeni tanımlayın.
  const [count, setCount] = useState(0);
```

useState'i çağırmak ne işe yarar? Bu, yeni bir "state değişkeni" tanımlar. Değişkenimizin adı count; fakat farklı bir şekilde de (örneğin banana) çağırabilirdik. Bu yöntemle fonksiyon çağrıları arasında verilerinizi koruyabilirsiniz. — useState ise this.state'in sınıfta sağladığı özellikleri kullanmanın yeni bir yoludur. Normalde değişkenler fonksiyon bitiminde "kaybolur"; fakat state değişkenleri React tarafından korunur.

useState'e argüman olarak ne atarız? useState() Hook'u için tek argüman initial (başlangıç) state argümanıdır. Sınıfların aksine, state bir nesne olmak zorunda değildir. Sayı ya da string tutabiliriz. Örneğimizde kullanıcının tıklama sayısı için bir sayı istedik bu yüzden değişkenimizin başlangıç değeri 0 olarak atandı. (eğer state'in içinde iki farklı değer tutmak isteseydik, useState()'i iki kere çağırmamız gerekecekti.)

useState geriye ne döndürür? useState geriye iki tane değer döndürür: Şimdiki state ve o state'i güncelleyen fonksiyon. Bu, const [count, setCount] = useState() yazmamızın sebebidir. Bu, bir sınıftaki this.state.count ve this.setState'e benzer. Eğer kullandığımız söz dizimi size tanıdık gelmediyse, sayfanın alt kısmında bu konuya tekrar döneceğiz.

Şimdi useState Hook'unun ne yaptığını bildiğimize göre, örneğimiz daha mantıklı bir hale gelecektir.

```
import React, { useState } from 'react';

function Example() {
  // "count" adında yeni bir state değişkeni tanımlayın.
  const [count, setCount] = useState(0);
```

count adında bir state değişkeni tanımladık ve 0'a eşitledik. React, tekrar eden render işlemleri arasında değişkenin mevcut değerini hatırlayacak ve

fonksiyonumuza en yeni değeri verecektir. Eğer şu anki count değerini değiştirmek isterseniz setCount çağırabilirsiniz.

Not:

useState ‘in isminin neden createState olmadığını merak ediyor olabilirsiniz.

“Create” tam olarak doğru olmazdı. Çünkü state; sadece bileşenimizi ilk kez render ettiğimizde oluşturulur. Sonraki renderlarda useState bize o anki state’i verir. Aksi takdirde bu “state” olmazdı. Aynı zamanda hook’ların isimlerinin neden hep use ile başlamasının nedeni de var. Bunu daha sonra Hook Kuralları bölümünde öğreneceğiz.

State Okuma

Sınıf temelli bir component’de geçerli count’u göstermek istediğimiz zaman this.state.count‘i okuruz:

```
<p>You clicked {this.state.count} times</p>
```

Foksiyon temelli bir component’de ise count değişkenini direk kullanabiliriz.:

```
<p>You clicked {count} times</p>
```

State Güncelleme

count state’i güncellemek için sınıfın içinde this.setState()‘i çağırmamız gerekiyor.

```
<button onClick={() => this.setState({ count: this.state.count + 1 })}>
  Click me
</button>
```

Fonksiyonda zaten değişken olarak setCount ve count‘a sahibiz, bu yüzden this‘e ihtiyaç duymuyoruz :

```
<button onClick={() => setCount(count + 1)}>
  Click me
</button>
```

Özet

Hadi şimdi Şimdi de baştan sona öğrendiklerimizin üzerinden geçelim ve anlayıp anlamadığımızı kontrol edelim.

```
import React, { useState } from 'react';
```

```
function Example() {

const [count, setCount] = useState(0);

return (

<div>

<p>You clicked {count} times</p>

<button onClick={() => setCount(count + 1)}>

Click me

</button>

</div>

);

}
```

Satır 1: React'ten useState Hook'unu ekliyoruz. Bu, yerel state'i fonksiyon bileşeninde tutmamıza izin verecek.

Satır 4: Example bileşeninin içinde useState Hook'unu çağırarak yeni bir state değişkeni tanımlıyoruz. Bu, isimlerini bizim verdiğimiz bir çift değer döndürür. Değişkenimizin adı count çünkü tıklama sayısını tutuyor.useStatee 0 yollayarak count'u 0sıfırdan başlatıyoruz. İkinci döndürülen değerin kendisi bir fonksiyondur. Bu, bize count'u güncellemek için izin verir. Bu yüzden onu setCount diye isimlendirdik.

Satır 9: Kullanıcı her tıkladığında, setCount'u yeni bir değerle çağırırız. React, Example bileşenini tekrar render edip, ona yeni count değerini atar.

İlk bakışta öğrenecek çok fazla şey varmış gibi görünebilir. Acele etmeyin! Açıklamalar arasında kaybolduysanız, yukarıdaki koda tekrar bakın ve baştan sona tekrar okumaya çalışın. State'in sınıflarda nasıl çalıştığını unutup bu koda meraklı gözlerle tekrar baktığınızda, söz veriyoruz, daha anlamlı gelecek.

İpucu: Köşeli Parantez Ne Anlama gelir?

State değişkenlerini tanımlarken köşeli parantezleri fark etmiş olabilirsiniz:

```
  const [count, setCount] = useState(0);
```

Soldaki isimler React API'ının bir parçası değildir.Kendi state değişkenlerinizi isimlendirebilirsiniz:

```
const [fruit, setFruit] = useState('banana');
```

Bu JavaScript sözdizimi “dizi parçalama (array destructuring)” olarak adlandırılır.fruit ve setFruit diye iki yeni değişken oluşturduğumuz anlamına gelir.Burada fruit, useState tarafından dönen ilk değere; setFruit ise ikinci değere atanır bu kod ile eşdeğerdir.

```
var fruitStateVariable = useState('banana');

var fruit = fruitStateVariable[0];

var setFruit = fruitStateVariable[1];
```

useState ile state değişkeni tanımladığımız zaman, bu bir çift (iki elemanlı bir dizi) döndürür. İlk eleman o anki değer, ikincisi ise onu güncelleyen fonksiyondur. [0] ve [1] kullanarak erişmek biraz kafa karıştırıcı çünkü onların kendine özgü anlamları var. Bu yüzden onun yerine dizi parçalama (array destructuring) yöntemini kullanıyoruz.

Not:

this gibi bir şey iletmediğimiz için, React’in useState’in hangi bileşene denk geldiğini nasıl bildiğini merak ediyor olabilirsiniz. Bu soruyu ve bunun gibi daha bir coğunu S.S.S. kısmında cevaplıyoruz.

İpucu: Çoklu State Değişkeni Kullanımı

Ayrıca; birden fazla state kullanmak istediğimiz durumlarda, farklı state değişkenlerine farklı adlar vermemize izin verdiği için, state değişkenlerini bir çift olarak ([something, setSomething]) tanımlamak kullanışlıdır.

```
function ExampleWithManyStates() {

  // Birden fazla state değişkeni bildir!

  const [age, setAge] = useState(42);

  const [fruit, setFruit] = useState('banana');

  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
```

Yukarıdaki bileşende, age, fruit ve todos adında yerel değişkenlerimiz var ve bunları ayrı ayrı güncelleyebiliriz.

```
  function handleOrangeClick() {

    // this.setState({ fruit: 'orange' }) ile benzerdir.

    setFruit('orange');
```

```
  }
```

birden fazla state değişkeni kullanmak zorunda değilsiniz. State değişkenleri, nesneleri ve dizileri gayet güzel bir şekilde tutabilir; böylece ilgili verileri birlikte tutabilirsiniz fakat sınıftaki this.setState'ın aksine state değişkenini güncellemek, birleştirmek (merge) yerine var olan ile değiştirir.

Effect Hook'unu Kullanmak

Effect Hook'u fonksiyon bileşenlerinde yan etkiler oluşturmanıza olanak sağlar:

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // componentDidMount ve componentDidUpdate kullanımına benzer bir kullanım sunar:
  useEffect(() => {
    // tarayıcının başlık bölümünü değiştirmemizi sağlar
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Bu kod parçacığı bir önceki sayfadaki sayaç uygulamasına dayanmaktadır, fakat bir yeni özellik eklenmiştir: sayacın tıklanma sayısına göre tarayıcının başlığı tıklanma sayısını göstermektedir.

React bileşenlerinde veri getirme, bir abonelik oluşturma ve DOM'u manuel olarak değiştirme yan etkilere örnek olarak verilebilir. Bu işlemleri "yan etkiler" (veya sadece "etkiler") olarak adlandırsanızda adlandırmasanızda, bunları muhtemelen daha önce bileşenlerinizde kullanmışsınızdır.

İpucu

Eğer React sınıf yaşam döngülerini (lifecycle) biliyorsanız, useEffect Hook'unu componentDidMount, componentDidUpdate, ve componentWillUnmount yaşam döngüsü methodlarının birleşimi olarak düşünebilirsiniz.

React bileşenlerinde iki tür yan etki vardır: temizlik gerektirmeyenler ve ihtiyaç duyanlar. Gelin bu aradaki farka detaylı olarak bakalım.

Temizlik Gerektirmeyen Etkiler

Bazen React DOM'u güncelledikten sonra bazı ek kodları çalıştırmak isteriz. Ağ istekleri, manuel DOM değişiklikleri ve günlük uygulama kayıtları, temizleme gerektirmeyen yaygın etkilere örnektir. Bu şekilde örneklendirebiliriz, çünkü onları çalıştırabilir ve ardından tamamen unutabiliriz. Sınıfların ve Hook'ların bu tür yan etkileri nasıl ifade etmemize izin verdiğini karşılaştıralım.

Örnek: Sınıflar Kullanılarak Gerçekleştirilmesi

React ta oluşturulan sınıf bileşenlerinde, render methodunun kendisi yan etkilere neden olmamalıdır. Aksi takdirde bu çok erken bir şekilde gerçekleşecektir — genellikle React'ın DOM'u güncellemesinden sonra bu yan etkilerin gerçekleşmesini isteriz.

Bu nedenle React sınıflarında, componentDidMount ve componentDidUpdate yan etkileri tanımlanmıştır. Örneğe geri dönecek olursak, React DOM'da değişiklik yaptıktan hemen sonra belge başlığını güncelleyen bir React sayaç sınıfı bileşeni aşağıdaki gibidir:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
```

```
  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }
  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }
  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

Sınıftaki bu iki yaşam döngüsü yöntemi arasındaki kodu nasıl tekrarlamamız gerektiğine dikkat edin.

Bunun nedeni çoğu durumda, bileşenin yeni oluşturulduğuna veya güncellenmiş olup olmadığına bakılmaksızın aynı yan etkiyi gerçekleştirmek istememizdir. Kavramsal olarak, her işlemeden sonra gerçekleşmesini isteriz - ancak React sınıf bileşenlerinin böyle bir methodu yoktur. Ayrı bir method olarak oluşturulabilirdi ama yine de ilgili kodu iki yerde çağırmamız gerekir.

Şimdi aynı işlemlerin useEffect Hook'u ile nasıl yapılabileceğine bakalım.

Örnek: Hook Kullanılarak Gerçekleştirilmesi

Bu örneği daha önce bu sayfanın en üstünde görmüştük, ama hadi bu örneğe daha yakından bakalım:

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

useEffect Hook'u ne yapar? Bu Hook'u kullanarak, React'e bileşenininiz oluştuktan sonra bir şeyler yapması gerektiğini söylersiniz. React, geçtiğiniz fonksiyonu hatırlayacak (buna "effect (etki)" olarak değineceğiz) ve DOM güncellemelerini yaptıktan sonra onu çağıracaktır. Bu etki de, tarayıcı başlığını atadık fakat aynı şekilde veri getirebilir veya bazı API'ları çağırabilirdik.

Neden useEffect Hook'u bir bileşen içinde çağırılıyor? Bileşenin içine "useEffect" yerleştirmek, "count" durum değişkenine (veya herhangi bir props'a) bu efektten (etkiden) erişmemizi sağlar. "Count" durum değişkenini okumak için özel bir API'a ihtiyacımız yok — fonksiyon kapanışlarında bu değişkene ulaşılabilmektedir. Hooklar, JavaScript kapanışlarını benimser ve JavaScript'in zaten bir çözüm sağladığı yerlerde React'e özgü API'leri tanımlamaktan kaçınır.

useEffect Hook'u her render (işlem) den sonra çağırılır mı? Evet! Varsayılan olarak, hem ilk oluşturmadan sonra hem de her güncellemeden sonra çalışır. Bunun nasıl özelleştirilebileceğinden. daha sonra bahsedeceğiz.) "Bileşenin oluşması" ve "güncelleme" terimleriyle düşünmek yerine, etkilerin "oluşturulduktan sonra" oluştuğunu düşünmeyi daha kolay bulabilirsiniz. React,

DOM'un etkileri çalıştırdığında güncellendiğini garanti eder.

Detaylı Açıklama

Artık etkiler hakkında daha fazla şey bildiğimize göre, bu satırlar bir anlam ifade etmeli:

```
function Example() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
}
```

Burada "Count" durum değişkenini tanımlıyoruz ve ardından React'e bir etki (effect) kullanmamız gerektiğini söylüyoruz. Daha sonra ise useEffect Hook'una bir fonksiyonu geçiyoruz. Geçmiş olduğumuz fonksiyon bizim etkimizdir. Etkimizin içinde, belge başlığını "document.title" tarayıcı API'ını kullanarak belirliyoruz. Etkinin içindeki en son "count" değişkenini okuyabiliriz çünkü bu, fonksiyonumuzun kapsamındadır. React bileşenimizi oluşturduğunda, kullandığımız efekti hatırlayacak ve ardından DOM'u güncelledikten sonra etkimizi çalıştıracak. Bu işlem her render işleminde gerçekleşecektir.

Deneyimli JavaScript geliştiricileri, useEffect e geçilen fonksiyonun her işlemde farklı olacağını düşünebilir. Bu kasıtlı olarak yapılmıştır. Aslında bu, verinin güncel olmaması endişesi olmadan etkinin içinden "count" değerini okumamıza izin veren şeydir. Her yeniden oluşturduğumuzda, bir öncekinin yerine bir farklı etki planlarız. Bir bakıma, bu, etkilerin daha çok oluşturma sonucunun bir parçası gibi davranmasını sağlar - her etki belirli bir render a "aittir". Bu durumunu daha net bir şekilde aşağıda göreceğiz.

İpucu

componentDidMount veya componentDidUpdate ten farklı olarak, useEffect ile planlanan etkiler tarayıcının ekranı güncellemesini engellemez. Bu, uygulamanızı daha duyarlı hale getirir. Etkilerin çoğunun eşzamanlı olarak gerçekleşmesi gerekmez. Yaptıkları nadir durumlarda (düzeni belirlemek gibi), useLayoutEffect adında useEffect ile aynı yapıda bir Hook vardır.

Temizlenen(CleanUp) Etkiler

Daha önce, herhangi bir temizlik gerektirmeyen yan etkilerin nasıl ifade

edileceğine bakmıştır. Bununla birlikte, bazı etkiler bunu yapar. Örneğin, bazı harici veri kaynaklarına bir abonelik ayarlamak isteyebiliriz. Bunun gibi durumlarda, bellek sızıntısına neden olmamak için temizlemek önemlidir! Bunu nasıl yapabileceğimizi sınıflarla ve Hook'larla karşılaştıralım.

Örnek: Sınıflar Kullanılarak Gerçekleştirilmesi

Bir React sınıfında, genellikle componentDidMount da bir abonelik oluşturulur ve componentWillUnmount ta temizlenir. Örneğin, ChatAPI adında arkadaşlarımızın çevrimiçi durumunu görmemize olanak sağlayan bir modülümüz olduğunu varsayalım. Bu durumu bir sınıf kullanarak şu şekilde abone olabilir ve gösterebiliriz:

```
class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }
  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
  handleStatusChange(status) {
```

```
    this.setState({
      isOnline: status.isOnline
    });
  }
  render() {
    if (this.state.isOnline === null) {
      return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
  }
}
```

componentDidMount ve componentWillUnmount un birbirlerini nasıl yansıtması gerektiğine dikkat edin. Yaşam döngüsü methodları, kavramsal olarak birbiriyle ilişkili kodlar ise mantıksal olarak bizi bölmeye zorlar.

Not:

Kartal gözlü okuyucular, bu örneğin tamamen doğru olması için bir componentDidUpdate methoduna da ihtiyaç duyduğunu fark edebilir. Şimdilik bunu görmezden geleceğiz fakat bu sayfanın bir sonraki bölümünde bundan bahsedeceğiz.

Örnek: Hook Kullanarak Gerçekleştirilmesi

Bu bileşeni Hook'lar kullanarak nasıl yazabileceğimize bakalım.

Temizlemeyi gerçekleştirmek için ayrı bir etkiye ihtiyacımız olduğunu düşünüyor olabilirsiniz. Ancak bir abonelik eklemek ve kaldırmak için olan kod o kadar yakından ilişkilidir ki, useEffect onu bir arada tutmak için tasarlanmıştır. Etkiniz bir işlev döndürürse, React temizleme zamanı geldiğinde onu çalıştırır:

```
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null)
  useEffect(() => {
```

```
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // Specify how to clean up after this effect:
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

Neden etkimizden bir işlevi döndürdük? Bu, etkiler için isteğe bağlı temizleme mekanizmasıdır. Her etki, arkasından temizleyen bir işlev döndürebilir. Bu, abonelik ekleme ve kaldırma mantığını birbirine yakın tutmamızı sağlar. Aynı etkinin parçalarıdırlar!

React bir efekti tam olarak ne zaman temizler? React temizleme işlemini bileşen ayrıldığında gerçekleştirir. Ancak, daha önce öğrendiğimiz gibi, etkiler yalnızca bir kez değil, her render da çalışır. Bu nedenle React ayrıca, etkileri bir sonraki sefer çalıştırmadan önce önceki işlemdeki etkileri temizler. Bunun neden hatalardan kaçınmaya yardımcı olduğunu ve performans sorunları yaratması durumunda bu davranışın nasıl devre dışı bırakılacağından aşağıda daha sonra bahsedeceğiz.

Not:

Etkilerden adlandırılmış bir fonksiyon dönmek zorunda değiliz. Buraada amacını belli etmesi açısından temizleme(cleanup) olarak adlandırdık fakat arrow fonksiyon döndürülebilir veya başka bir fonksiyon şeklinde çağırabilir.

Tekrar

useEffect in bir bileşen oluşturulduktan sonra farklı yan etkileri ifade etmemize izin

verdiğini öğrendik. Bazı efektler temizleme gerektirebilmektedir, bu nedenle bir fonksiyon döndürürler:

```
useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }
  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
});
```

Diğer etkilerin temizleme aşaması olmayabilir ve hiçbir şey döndürmeyebilirler.

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
});
```

Etki Hook'u, her iki kullanım durumunu da tek bir API altında birleştirir.

Effect Hook'unun nasıl çalıştığını iyi bir şekilde anladığınızı düşünüyorsanız veya bunalmış hissediyorsanız, Hook Kuralları hakkındaki bir sonraki sayfaya geçebilirsiniz.

Etkileri Kullanmak İçin İpuçları

Bu sayfaya, deneyimli React kullanıcılarının muhtemelen merak edeceği useEffect in bazı yönlerine derinlemesine bir bakışla devam edeceğiz. Kendinizi onları daha derinden incelemek zorunda hissetmeyin. Efekt Hook'u hakkında daha fazla ayrıntı öğrenmek için her zaman bu sayfaya geri dönebilirsiniz.

İpucu: Kavramları Daha İyi Ayırmak İçin Birden Çok Efekt Kullanın

Hooks için Motivasyon'da ana hatlarıyla belirttiğimiz sorunlardan biri, sınıf yaşam döngüsü yöntemlerinin genellikle ilgisiz mantık içermesi, fakat ilgili mantığın birden fazla methodta bozulmasıdır. Önceki örneklerden sayaç ve arkadaş durumu göstergesi mantığını birleştiren bir bileşen:

```
class FriendStatusWithCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0, isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }
  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }
  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
```

```
  }

  // ...
```

document.title öğesini ayarlayan mantığın componentDidMount ve componentDidUpdate arasında nasıl bölündüğüne dikkat edin. Abonelik mantığı ayrıca componentDidMount ve componentWillUnmount arasında da yayılır. Ve componentDidMount, her iki görev için kod içerir.

Tıpkı State Hook'unu birden fazla kullanabildiğiniz gibi, birkaç efekt de kullanabilirsiniz. Bu, alakası uygulama mantığını farklı etkilere ayırmamızı sağlar:

```
function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  // ...
}
```

Yaşam döngüsü methodları yerine Hook'lar, kodu yaptığı işe göre bölmemize izin verir. React, bileşen tarafından kullanılan her etkiyi, belirtilen sırayla uygulayacaktır.

Açıklama: Neden Her Güncellemede Etkiler Çalışıyor?

Sınıf kavramına alışkınsanız, efekt temizleme aşamasının neden her yeniden oluşturmadan sonra olduğunu, bileşenin işleminin bitmesi sırasında bir kez olmayıp neden gerçekleştiğini merak ediyor olabilirsiniz. Bu tasarımın neden daha az hata içeren bileşenler oluşturmamıza yardımcı olduğunu görmek için aşağıdaki örneğe bakalım.

Bu sayfanın önceki kısımlarında, bir arkadaşın çevrimiçi olup olmadığını gösteren bir örnek FriendStatus bileşenini tanıttık. Sınıfımız this.props dan friend.id yi okur ve daha sonra bileşen bağlandıktan sonra arkadaş durumunu öğrenir (abone olur) ve bağlantıyı kesme sırasında aboneliği iptal eder:

```
componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}
componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}
```

Peki bileşen ekranda iken friend değeri değişirse ne olur? Bileşenimiz, farklı bir arkadaşın çevrimiçi durumunu göstermeye devam edecektir. Bu bir hatadır (bug). Ayrıca abonelikten çıkma çağrısı yanlış arkadaş kimliğini (friend ID) kullanacağından, bağlantıyı keserken bellek sızıntısına veya çökmeye neden olabiliriz.

Bir sınıf bileşeninde, bu durumu ele almak için componentDidUpdate Hook'unu eklememiz gerekir:

```
componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
```

```
      this.props.friend.id,
      this.handleStatusChange
    );
  }
  componentDidUpdate(prevProps) {
    // Unsubscribe from the previous friend.id
    ChatAPI.unsubscribeFromFriendStatus(
      prevProps.friend.id,
      this.handleStatusChange
    );
    // Subscribe to the next friend.id
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
```

React uygulamalarında componentDidUpdate Hook'unu doğru bir şekilde yönetmeyi unutmak yaygın bir hata kaynağıdır.

Şimdi bu bileşenin Hook'ları kullanan sürümünü düşünelim:

```
function FriendStatus(props) {
  // ...
  useEffect(() => {
    // ...
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
```

Bu hatadan dolayı kaynaklanmıyor. (Fakat biz de herhangi bir değişiklik yapmadık.)

Güncellemeleri işlemek için özel bir kod yoktur çünkü useEffect bunları varsayılan olarak yönetir. Sonraki efektleri uygulamadan önce önceki efektleri temizler. Bunu somutlaştırmak için, bir grup abone olma ve abonelikten çıkma çağırısının olduğu bir bileşen kullanılabilir:

```
// Mount with { friend: { id: 100 } } props
ChatAPI.subscribeToFriendStatus(100, handleStatusChange);     // Run first effect

// Update with { friend: { id: 200 } } props
ChatAPI.unsubscribeFromFriendStatus(100, handleStatusChange); // Clean up previous effect
ChatAPI.subscribeToFriendStatus(200, handleStatusChange);     // Run next effect

// Update with { friend: { id: 300 } } props
ChatAPI.unsubscribeFromFriendStatus(200, handleStatusChange); // Clean up previous effect
ChatAPI.subscribeToFriendStatus(300, handleStatusChange);     // Run next effect

// Unmount
ChatAPI.unsubscribeFromFriendStatus(300, handleStatusChange); // Clean up last effect
```

Bu davranış, varsayılan olarak tutarlılığı sağlar ve eksik güncelleme mantığı nedeniyle sınıf bileşenlerinde yaygın olan hataları önler.

İpucu: Efektleri Atlayarak Performansı Optimize Etme

Bazı durumlarda, her işlemeden sonra efekti temizlemek veya uygulamak bir performans sorunu yaratabilir. Sınıf bileşenlerinde, componentDidUpdate içinde prevProps veya prevState ile ekstra bir karşılaştırma yazarak bunu çözebiliriz:

```
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    document.title = `You clicked ${this.state.count} times`;
  }
}
```

Bu gereksinim, useEffect Hook API'ında yerleşik olması için yeterince yaygındır. Yeniden render olması arasında belirli değerler değişmediyse React'e bir efekti uygulamayı atlamasını söyleyebilirsiniz. Bunu yapmak için, bir diziyi useEffect e isteğe bağlı ikinci bir parametre olarak iletin:

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // Only re-run the effect if count changes
```

Yukarıdaki örnekte, ikinci argüman olarak [count] u iletiyoruz. Peki bu ne anlama geliyor? count eğer 5 ise ve daha sonra bileşenimiz count hala 5 e eşit olacak şekilde yeniden oluşturulursa, React önceki render daki [5] ile sonraki render daki [5] 'i karşılaştırır. Dizideki tüm öğeler aynı olduğundan (5 === 5), React efekti atlar. Gerçekleştirdiğimiz iyileştirme bu şekildedir.

6 olarak güncellenen count ile oluşturduğumuzda, React, önceki işlemedeki [5] dizisindeki öğeleri bir sonraki işlemedeki [6] dizisindeki öğelerle karşılaştıracaktır. Bu sefer React efekti yeniden uygulayacak çünkü 5 !== 6. Dizide birden fazla öğe varsa, React, yalnızca biri farklı olsa bile efekti yeniden çalıştıracaktır.

Bu, temizleme aşaması olan efektler için de aynı şekilde işe yarar:

```
useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
```

```
  }
  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
}, [props.friend.id]); // Only re-subscribe if props.friend.id changes
```

Gelecekte, ikinci argüman bir derleme zamanı dönüşümü ile otomatik olarak eklenebilir.

Not:

Bu optimizasyonu kullanırsanız, dizinin bileşen kapsamındaki zamanla değişen ve efekt tarafından kullanılan tüm değerleri (props ve state gibi) içerdiğinden emin olun. Aksi takdirde, kodunuz önceki render'lardan eski değerlere başvurur.

Bir efekt çalıştırmak ve yalnızca bir kez temizlemek istiyorsanız (mount ve unmount sırasında), boş bir diziyi ([]) ikinci argüman olarak iletebilirsiniz. Bu, React'e efektinizin props veya state'deki hiçbir değere bağlı olmadığını, bu yüzden asla yeniden çalıştırılması gerekmediğini söyler. Bu özel bir durum olarak ele alınmaz - doğrudan bağımlılıklar dizisinin her zaman çalıştığı gibi çalışmasını sağlar.

Boş bir dizi geçirirseniz ([]), efektin içindeki props ve state her zaman başlangıç değerlerine sahip olacaktır. İkinci argüman olarak [] geçerken tanıdık componentDidMount ve componentWillUnmount modelleri, çok sık yeniden çalışan efektleri önlemek için genellikle daha iyi çözümler kullanılır.

"Eslint-plugin-react-hooks" paketimizin bir parçası olan exhaustive-deps kuralını kullanmanızı öneririz. Bağımlılıklar yanlış belirlendiğinde uyarı verir ve bir düzeltme önerir.

Hook Kuralları

Hook'ları Sadece En Üst Seviyede Çağırın

Döngülerde, koşullarda veya iç içe geçmiş fonksiyonlarda Hook çağrısı yapmayın. Bunun yerine, Hook'ları her zaman React fonksiyonunuzun en üst seviyesinde, herhangi bir return yapmadan önce kullanın. Bu kuralı uygulayarak, bir bileşenin her render edildiğinde Hook'ların aynı sırada çağrıldığından emin olursunuz. React'in çoklu useState ve useEffect çağrıları arasındaki Hook'ların durumunu doğru şekilde korumasını sağlayan şey budur. (Merak ediyorsanız, bunu aşağıda

detaylıca açıklayacağız.)

Hook'ları Sadece React Fonksiyonlarından Çağırın

Sıradan JavaScript fonksiyonlarında Hook'ları çağırmayın. Bunun yerine:

✓ React fonksiyon bileşenlerinden Hook'ları çağırabilirsiniz.

✓ Özel Hook'lardan Hook'ları çağırabilirsiniz. (bir sonraki sayfada bunları öğreneceğiz.)

Bu kuralı uygulayarak, bir bileşendeki tüm durum bilgisi mantığının kaynak kodundan açıkça görülebildiğinden emin olursunuz.

ESLint Eklentisi

Bu iki kuralı uygulayan eslint-plugin-react-hooks adında bir ESLint eklentisi yayınladık. Denemek isterseniz, bu eklentiyi projenize ekleyebilirsiniz:

```
npm install eslint-plugin-react-hooks --save-dev
```

```
// Sizin ESLint yapılandırmanız
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    // ...
    "react-hooks/rules-of-hooks": "error", // Hook kurallarını kontrol eder
    "react-hooks/exhaustive-deps": "warn" // Efekt bağımlılıklarını kontrol eder
  }
}
```

Bu eklenti varsayılan olarak Create React App aracına dahil edilmiştir.

Kendi Hook'larınızı nasıl yazacağınızı açıklayan bir sonraki sayfaya şimdi atlayabilirsiniz. Bu sayfaya, bu kuralların ardındaki mantığı açıklayarak devam edeceğiz.

Açıklama

Daha önce öğrendiğimiz gibi, tek bir bileşende birden fazla State veya Efekt Hook'larını kullanabiliriz:

```
function Form() {
  // 1. name state değişkenini kullan
  const [name, setName] = useState('Onur');

  // 2. Formun devamlılığını sağlamak için bir efekt kullan
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });

  // 3. surname state değişkenini kullan
  const [surname, setSurname] = useState('Şuyalçınkaya');

  // 4. Başlığı güncellemek için bir efekt kullan
  useEffect(function updateTitle() {
    document.title = name + ' ' + surname;
  });
  // ...
}
```

Peki React, hangi state'in hangi useState çağrısına karşılık geldiğini nasıl biliyor? Cevap, React'in Hook'ların çağrılma sırasına dayalı olmasıdır. Örneğimiz çalışıyor çünkü Hook çağrılarının sırası her render etmede aynı:

```
// ------------
// İlk render etme
// ------------
```

```
useState('Onur')          // 1. name state değişkenini 'Onur' ile başlat
useEffect(persistForm)    // 2. Formun devamlılığını sağlamak için bir efekt ekle
useState('Şuyalçınkaya')  // 3. surname state değişkenini 'Şuyalçınkaya' ile başlat
useEffect(updateTitle)    // 4. Başlığı güncellemek için bir efekt ekle
// ------------
// İkinci render etme
// ------------
useState('Onur')          // 1. name state değişkenini oku (argüman yoksayılmıştır)
useEffect(persistForm)    // 2. Formun devamlılığını sağlamak efekti değiştir
useState('Şuyalçınkaya')  // 3. surname state değişkenini oku (argüman yoksayılmıştır)
useEffect(updateTitle)    // 4. Başlığı güncellemek için efekti değiştir
// ...
```

Hook çağrılarının sırası render etmeler arasında aynı olduğu sürece, React bazı yerel state'leri bu çağrıların her biriyle ilişkilendirebilir. Ancak bir koşulun içine bir Hook çağrısı (örneğin, persistForm efekti) koyarsak ne olur?

```
// ● Bir koşul içerisinde Hook kullanarak ilk kuralı çiğniyoruz
if (name !== '') {
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });
}
```

İlk render etmede name !== '' koşulu true, bu yüzden bu Hook'u çalıştırıyoruz. Bununla birlikte, bir sonraki render etmede kullanıcı formu temizleyerek koşulu false hale getirebilir. Artık render etme sırasında bu Hook'u atladığımız için, Hook çağrılarının sırası değişiyor:

```
useState('Onur')          // 1. name state değişkenini oku (argüman yoksayılmıştır)
// useEffect(persistForm)  // ● Bu Hook atlandı!
```

useState('Şuyalçınkaya') // ● 2 (ama 3'tü). surname state değişkeni okunamadı

useEffect(updateTitle) // ● 3 (ama 4'tü). Efekt değiştirilemedi

React, ikinci useState Hook çağrısı için ne döneceğini bilemezdi. React, bu bileşendeki ikinci Hook çağrısının, bir önceki render etme sırasında olduğu gibi, persistForm efektine karşılık gelmesini bekliyordu, ancak artık gelmiyor. Bu noktadan itibaren, atladığımız çağrıdan sonraki her bir Hook çağrısı da birer birer kayıp, hatalara yol açacaktır.

Bu yüzden Hook'lar bileşenlerimizin en üst seviyesinde çağrılmalıdır. Eğer bir efekti koşullu olarak çalıştırmak istiyorsak, bu koşulu Hook'umuzun içerisine koyabiliriz:

```
useEffect(function persistForm() {
  // 👍 Artık ilk kuralı çiğnemiyoruz
  if (name !== '') {
    localStorage.setItem('formData', name);
  }
});
```