Fetching Data

React'te fetch ile bir sunucudan veri çekmek ve göndermek -- GET, POST, PUT, DELETE, loading ve error handling, basit örneklerle.

Orta 12 dk

Son güncelleme: 18 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/fetching-data

---

Fetching Data

Şimdiye kadarki tüm örneklerde veri, component'in İÇİNDE (bir dizi ya da nesne olarak) tanımlıydı. Gerçek uygulamalarda ise veri genellikle bir **sunucudan** gelir. Bu ders, React'te bir sunucudan veri çekmenin (ve oraya veri göndermenin) temel yolunu -- `fetch` fonksiyonunu -- anlatıyor.

fetch ile Veri Çekmek: GET

`fetch`, tarayıcının yerleşik bir fonksiyonudur -- bir URL'e HTTP isteği atar ve bir **Promise** döner. Hooks dersinde gördüğümüz `useEffect`, component ekrana geldiğinde bu isteği atmak için kullanılır:

```
import { useEffect, useState } from "react";

function BasicFetchGetExample() {
  const [courses, setCourses] = useState([]);

  // useEffect runs ONCE when the component first appears on screen (empty
  // dependency array []) -- this is the most common use case for fetching
  // data. fetch() returns a Promise; with .then() we first convert the
  // response to JSON, then write it to state.
  useEffect(() => {
    fetch("/api/courses")
      .then((response) => response.json())
      .then((data) => setCourses(data));
  }, []);

  return (
    <ul>
      {courses.map((course) => (
        <li key={course.id}>{course.name}</li>
      ))}
    </ul>
  );
}
```

`useEffect`'in ikinci argümanı boş bir dizi (`[]`) olduğu için, bu istek yalnızca component İLK RENDER edildiğinde bir kez atılır. `fetch(url).then(...)`, önce yanıtı JSON'a çevirir, sonra gelen veriyi `setCourses` ile state'e yazar -- state değişince React yeniden render eder ve liste ekranda görünür.

Loading Durumu Göstermek

İstek atıldıktan sonra yanıt gelene kadar bir süre geçer -- bu sürede kullanıcıya boş bir ekran yerine bir şeyler olduğunu göstermek gerekir:

```
import { useEffect, useState } from "react";

function LoadingStateExample() {
  const [courses, setCourses] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch("/api/courses")
      .then((response) => response.json())
      .then((data) => {
        setCourses(data);
        setLoading(false);
      });
  }, []);

  // Showing a "Loading..." message on screen until the data arrives lets
  // the user know something is happening -- instead of an empty list.
  if (loading) {
    return <p>Loading...</p>;
  }

  return (
    <ul>
      {courses.map((course) => (
        <li key={course.id}>{course.name}</li>
      ))}
    </ul>
  );
}
```

`loading` state'i başlangıçta `true`; veri geldiğinde `false`'a çekiyoruz. `loading` `true` iken component `<p>Loading...</p>` render edip ERKEN ÇIKIYOR (`return`) -- geri kalan JSX hiç render edilmiyor.

Hataları Yönetmek

Bir istek her zaman başarılı olmayabilir -- ağ kopabilir, sunucu hata dönebilir. Bunu yönetmezsek, kullanıcı sessizce boş bir ekranla kalır:

```
import { useEffect, useState } from "react";

function ErrorHandlingExample() {
  const [courses, setCourses] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetch("/api/courses")
      .then((response) => {
        if (!response.ok) {
          // fetch() does NOT automatically reject on HTTP errors like
          // 404/500 -- it's up to us to CHECK response.ok.
          throw new Error("Request failed: " + response.status);
        }
        return response.json();
      })
      .then((data) => setCourses(data))
      .catch((err) => setError(err.message))
      .finally(() => setLoading(false));
  }, []);

  if (loading) {
    return <p>Loading...</p>;
  }

  // If the error state is set, we show an error message without ever
  // rendering the list -- the same if pattern from the Conditional
  // Rendering lesson.
  if (error) {
    return <p>Something went wrong: {error}</p>;
  }

  return (
    <ul>
      {courses.map((course) => (
        <li key={course.id}>{course.name}</li>
      ))}
    </ul>
  );
}
```

`response.ok`, HTTP durum kodunun 200-299 aralığında olup olmadığını söyler -- `fetch`, 404 ya da 500 gibi durumlarda KENDİLİĞİNDEN reddetmediği (reject) için bunu elle kontrol edip `throw` etmemiz gerekir. `.catch()`, bu hatayı yakalayıp `error` state'ine yazar; `.finally()` ise başarılı ya da başarısız fark etmeksizin `loading`'i kapatır.

Veri Göndermek: POST

Veri okumanın (GET) yanında, yeni bir kayıt oluşturmak için POST isteği atılır:

```
import { useState } from "react";

function PostRequestExample() {
  const [name, setName] = useState("");

  function handleSubmit(event) {
    event.preventDefault();

    // For POST requests, we write the data to send into `body` as a JSON
    // string -- and tell the server it's JSON via the `Content-Type` header.
    fetch("/api/courses", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ name }),
    });
  }

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={name}
        onChange={(event) => setName(event.target.value)}
        placeholder="Course name"
      />
      <button type="submit">Add</button>
    </form>
  );
}
```

POST isteğinde `fetch`'e ikinci bir argüman -- bir seçenekler nesnesi -- veririz: `method: "POST"`, `headers` ile sunucuya gönderdiğimiz verinin JSON olduğunu bildiriyoruz, `body`'ye de göndereceğimiz veriyi `JSON.stringify(...)` ile bir metne çevirip koyuyoruz.

Güncellemek ve Silmek: PUT ve DELETE

Var olan bir kaydı güncellemek için PUT, silmek için DELETE kullanılır:

```
function PutAndDeleteExample({ courseId }) {
  function handleRename(newName) {
    // PUT COMPLETELY updates an existing record -- we send the new value,
    // and the server replaces the record with it.
    fetch(`/api/courses/${courseId}`, {
      method: "PUT",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ name: newName }),
    });
  }

  function handleDelete() {
    // DELETE usually specifies which record to delete via the id in the
    // URL, without sending a body.
    fetch(`/api/courses/${courseId}`, { method: "DELETE" });
  }

  return (
    <div>
      <button onClick={() => handleRename("Updated Name")}>Rename</button>
      <button onClick={handleDelete}>Delete</button>
    </div>
  );
}
```

PUT, POST'a benzer şekilde bir `body` gönderir -- ama URL'de HANGİ kaydın güncelleneceğini belirtir (`/courses/${courseId}`). DELETE ise genellikle hiç `body` göndermeden, yalnızca URL'deki id ile hangi kaydın silineceğini belirtir.

Özet ve Terimler Sözlüğü

`fetch` , bir URL'e HTTP isteği atıp bir Promise döner; `useEffect` ile component ekrana geldiğinde çağrılır. GET veri okur, POST yeni kayıt oluşturur, PUT var olan bir kaydı günceller, DELETE siler -- POST ve PUT'ta `body` 'ye JSON gönderilir. Bir istek her zaman `loading` ve `error` state'leriyle birlikte yönetilmelidir -- kullanıcı, veri gelene kadar bir bekleme mesajı, bir hata olduğunda da anlaşılır bir mesaj görmelidir.

Terimler Sözlüğü

fetch — Tarayıcının, bir URL'e HTTP isteği atmak için kullanılan yerleşik fonksiyonu; bir Promise döner.

HTTP Metodu — Bir isteğin AMACINI belirten kelime: GET (okuma), POST (oluşturma), PUT (güncelleme), DELETE (silme).

Promise — Henüz tamamlanmamış, ileride bir sonuç (ya da hata) üretecek bir işlemi temsil eden JavaScript nesnesi.