React + REST API

React'in bir REST API'ye bağlanması -- API katmanını bir api.js modülünde ayırmak, listeleme, oluşturma ve silme işlemlerini birlikte yönetmek, basit örneklerle.

Orta 15 dk

Son güncelleme: 18 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/react-rest-api

---

React + REST API

Fetching Data dersinde `fetch`'in GET/POST/PUT/DELETE'ini tek tek gördük. Bu ders, bunları bir araya getirip GERÇEK bir uygulamada kurslar listesi gibi bir kaynağı uçtan uca (listeleme, ekleme, silme) yönetmeyi anlatıyor -- ayrıca React'in bir backend'le nasıl konuştuğunu büyük resmiyle gösteriyor.

React → HTTP → Backend → Veritabanı

Bir React uygulaması, verisini genellikle kendi başına saklamaz -- `fetch` ile bir **backend'e** (bu projenin kendisi gibi bir Spring Boot uygulaması olabilir) HTTP isteği atar. Örneğin bir kurs listesi istendiğinde: React `fetch("/courses")` çağırır → istek HTTP üzerinden backend'e ulaşır → backend'in bir `@RestController`'ı (bu terimi Java kursundaki REST API Tasarımı dersinden hatırlayabilirsin) isteği karşılar → veriyi veritabanından (PostgreSQL) okur → JSON olarak geri döner → React bu JSON'ı state'e yazıp ekranı günceller. Bu derste gerçek bir Spring Boot backend'i kurmuyoruz -- onun yerine aynı REST kurallarına (GET/POST/DELETE, JSON, HTTP durum kodları) uyan basit bir sahte sunucu (**json-server**) kullanıyoruz; React tarafındaki kod, gerçek bir Spring Boot API'sine bağlanırken de birebir aynı olurdu.

API Katmanını Ayırmak: Bir api.js Modülü

Her component'in içine `fetch` çağrıları dağıtmak yerine, bunları TEK bir yerde toplamak yaygın bir pratiktir:

```
// In real apps, instead of scattering fetch calls across every component,
// it's common to collect them in a separate file (e.g. api.js) -- this
// keeps all requests in ONE place and lets you change `BASE_URL` in a
// single spot. This example shows the CONTENTS of that file (normally it
// would be a separate file, and components would import these functions).

const BASE_URL = "http://localhost:3000";

async function getCourses() {
  const response = await fetch(`${BASE_URL}/courses`);
  if (!response.ok) {
    throw new Error("Failed to load courses");
  }
  return response.json();
}

async function createCourse(course) {
  const response = await fetch(`${BASE_URL}/courses`, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(course),
  });
  return response.json();
}

async function deleteCourse(courseId) {
  await fetch(`${BASE_URL}/courses/${courseId}`, { method: "DELETE" });
}

function ApiModuleExample() {
  return (
    <p>
      This represents an api.js module: getCourses(), createCourse() and
      deleteCourse() wrap the raw fetch calls in one place.
    </p>
  );
}
```

`getCourses()`, `createCourse()`, `deleteCourse()` gibi fonksiyonlar, `fetch`'in ayrıntılarını (URL, `method`, `headers`) SAKLAR -- component'ler yalnızca bu fonksiyonları çağırır, `fetch`'in kendisiyle hiç uğraşmaz. `BASE_URL`'i tek bir yerde tanımlamak, backend adresi değiştiğinde (örneğin geliştirmeden production'a geçerken) tek bir satırı değiştirmeyi yeterli kılar.

Backend'den Kurs Listesini Çekmek

`useEffect` içinde `async`/`await` ile veri çekmek, `.then()` zincirine göre daha okunaklı bir alternatif:

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

const BASE_URL = "http://localhost:3000";

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

  useEffect(() => {
    // The callback of useEffect CANNOT be async directly -- so we define a
    // separate async function inside and call it right away. In a real
    // project this corresponds to GET /api/courses (with Spring Boot) --
    // React sends a request to the backend over HTTP, the backend reads
    // from the database (PostgreSQL) and returns JSON.
    async function loadCourses() {
      const response = await fetch(`${BASE_URL}/courses`);
      const data = await response.json();
      setCourses(data);
    }

    loadCourses();
  }, []);

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

`useEffect`'in callback'i doğrudan `async` OLAMAZ (React bunu desteklemiyor) -- bu yüzden içeride ayrı bir `async` fonksiyon tanımlayıp hemen çağırıyoruz. `await`, `fetch`'in ve `response.json()`'ın Promise'lerinin tamamlanmasını BEKLER; sonuç geldiğinde `setCourses` ile state güncellenir.

Yeni Bir Kurs Oluşturmak

Forms dersindeki controlled input + `onSubmit` deseni, bir POST isteğiyle birleşiyor:

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

const BASE_URL = "http://localhost:3000";

function CreateCourseFormExample({ onCreated }) {
  const [name, setName] = useState("");

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

    // The controlled input + onSubmit pattern from the Forms lesson is
    // combined here with a POST request. On the Spring Boot side, this
    // corresponds to a `@PostMapping` method that uses `@RequestBody` to
    // convert the JSON into a Java object (see the REST API Design lesson
    // in the Java course).
    const response = await fetch(`${BASE_URL}/courses`, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ name }),
    });
    const newCourse = await response.json();

    onCreated(newCourse);
    setName("");
  }

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

Form gönderildiğinde, girilen değer POST isteğiyle backend'e gönderilir; backend yeni kaydı oluşturup (genellikle kendi ürettiği bir `id` ile) geri döner. `onCreated(newCourse)`, bu yeni kaydı ÜST component'e bildirmek için kullanılan bir callback prop -- Props dersindeki "veriyi yukarı iletmek" deseni.

Bir Kursu Silmek

Bir kaydı sunucudan sildikten sonra, ekrandaki listeyi de güncellememiz gerekir:

```
const BASE_URL = "http://localhost:3000";

function DeleteCourseExample({ courses, setCourses }) {
  async function handleDelete(courseId) {
    await fetch(`${BASE_URL}/courses/${courseId}`, { method: "DELETE" });

    // AFTER deleting on the server, we also need to update the list on
    // screen -- following the immutability rule from the State lesson, we
    // create a new array by REMOVING the deleted record with filter().
    setCourses(courses.filter((course) => course.id !== courseId));
  }

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

DELETE isteği başarılı olduktan SONRA, State dersindeki immutability kuralına uyarak `filter()` ile silinen kaydı çıkarıp YENİ bir dizi oluşturuyoruz -- diziyi doğrudan değiştirmiyoruz (`splice` gibi).

Oluşturduktan Sonra Ekranı Güncellemek

Yeni bir kayıt oluşturduktan sonra, listeyi baştan tekrar çekmek yerine (bir istek daha atmak yerine) genellikle yeni kaydı doğrudan mevcut listeye eklemek tercih edilir:

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

const BASE_URL = "http://localhost:3000";

function RefreshAfterMutationExample() {
  const [courses, setCourses] = useState([]);
  const [name, setName] = useState("");

  useEffect(() => {
    fetch(`${BASE_URL}/courses`)
      .then((response) => response.json())
      .then((data) => setCourses(data));
  }, []);

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

    const response = await fetch(`${BASE_URL}/courses`, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ name }),
    });
    const newCourse = await response.json();

    // After the server creates the new record, instead of re-fetching the
    // whole list (instead of making another request), we follow the
    // immutability rule from the State lesson and add the new record
    // directly to the existing list -- the screen updates instantly.
    setCourses([...courses, newCourse]);
    setName("");
  }

  return (
    <div>
      <form onSubmit={handleSubmit}>
        <input
          type="text"
          value={name}
          onChange={(event) => setName(event.target.value)}
          placeholder="Course name"
        />
        <button type="submit">Add</button>
      </form>
      <ul>
        {courses.map((course) => (
          <li key={course.id}>{course.name}</li>
        ))}
      </ul>
    </div>
  );
}
```

`setCourses([...courses, newCourse])`, State dersindeki spread deseniyle eski listeyi kopyalayıp sonuna yeni kaydı ekliyor -- ekran, sunucuya ikinci bir istek atmadan ANINDA güncelleniyor.

Özet ve Terimler Sözlüğü

Bir React uygulaması, `fetch` ile bir backend'e (Spring Boot gibi) HTTP istekleri atarak veri okur ve yazar; `fetch` çağrılarını ayrı bir `api.js` modülünde toplamak yaygın bir pratiktir. Veri çekme genellikle `useEffect` içinde `async` / `await` ile yapılır. Bir kayıt oluşturulduğunda ya da silindiğinde, ekrandaki listeyi State dersindeki immutability kurallarına (`filter()` , spread) uyarak güncellemek gerekir -- bazen sunucuya ikinci bir istek atmak yerine, elimizdeki sonucu doğrudan state'e yansıtmak yeterlidir.

Terimler Sözlüğü

REST API — Kaynakları (örneğin kurslar) HTTP metodlarıyla (GET/POST/PUT/DELETE) yöneten bir backend arayüzü tasarım biçimi.

api.js Modülü — Bir uygulamadaki tüm `fetch` çağrılarını tek bir yerde toplayan, component'lerin doğrudan kullandığı yardımcı fonksiyon dosyası.

Pratik Proje

Bu kategoride (Fetching Data, React + REST API) öğrendiğimiz kavramları bir arada kullanan, gerçek ve çalıştırılabilir bir örnek proje var: **API & Data Fetching Demo** -- json-server ile çalışan sahte bir REST API'ye bağlanan bir kurs listesi uygulaması.

Proje; `useEffect` + `fetch` ile listeleme, `loading` / `error` state yönetimi, controlled form ile yeni kayıt oluşturma (POST), bir kaydı silme (DELETE), ve her mutasyondan sonra ekranı immutability kurallarına uyarak güncellemeyi bir arada gösteriyor. Bilgisayarına indirip çalıştırabilir, kodunu satır satır inceleyebilirsin:

```
git clone https://github.com/cdurgun/react-course-projects.git
cd react-course-projects
npm install
cd projects/api-data-fetching
npm run server
```

Sahte API'yi (json-server) ayrı bir terminalde çalıştırdıktan sonra, başka bir terminalde React uygulamasını başlat:

```
cd react-course-projects/projects/api-data-fetching
npm run dev
```

`react-course-projects` deposu **npm workspaces** kullanır -- `npm install` yalnızca bir kez, depo kökünde çalıştırılır ve tüm proje klasörleri ortak bağımlılıkları paylaşır (her klasörde ayrı bir `node_modules` oluşmaz). Kök dizinde `npm install` 'ı daha önce çalıştırdıysan, doğrudan `cd react-course-projects/projects/api-data-fetching` yapıp yukarıdaki iki komutu (`npm run server` , `npm run dev`) iki ayrı terminalde çalıştırman yeterli.