Form Handling

onSubmit ile formu göndermek, birden fazla alanı tek bir state nesnesiyle yönetmek, validation ve hata mesajları göstermek -- basit örneklerle.

Orta 13 dk

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

---

Form Handling

Controlled Components dersinde tek bir input'u state'e bağlamayı gördük. Bu ders, gerçek bir formu -- gönderimi, birden fazla alanı, doğrulamayı ve hata mesajlarını -- baştan sona ele alıyor.

Formu Göndermek: onSubmit ile Değerleri Toplamak

Events dersinde gördüğümüz `onSubmit`, formlarda genellikle state'teki değeri kullanmak için kullanılır:

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

function FormSubmitExample() {
  const [name, setName] = useState("");
  const [submittedName, setSubmittedName] = useState(null);

  function handleSubmit(event) {
    event.preventDefault();
    // When the form is submitted, we use the value currently in state --
    // since it's a controlled input, we always know what the value is.
    setSubmittedName(name);
  }

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={name}
        onChange={(event) => setName(event.target.value)}
        placeholder="Enter your name"
      />
      <button type="submit">Submit</button>

      {submittedName && <p>Submitted: {submittedName}</p>}
    </form>
  );
}
```

Input controlled olduğu için, form gönderildiğinde `name` state'i zaten güncel -- ayrıca DOM'dan değeri okumaya gerek yok, doğrudan state'i kullanıyoruz.

Birden Fazla Alanı Yönetmek

Bir formda birden fazla input olduğunda, her biri için ayrı bir `useState` açmak yerine, hepsini TEK bir state nesnesinde toplamak daha yönetilebilir:

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

function MultiFieldFormExample() {
  // Instead of opening a separate useState for each field, we keep ALL
  // fields in a single state object -- more manageable as the form grows.
  const [formData, setFormData] = useState({ name: "", email: "" });

  function handleChange(event) {
    const { name, value } = event.target;

    // Whichever input changed (based on the input's `name` attribute),
    // we only update that field, copying the rest as they are.
    setFormData({ ...formData, [name]: value });
  }

  return (
    <form>
      <input
        type="text"
        name="name"
        value={formData.name}
        onChange={handleChange}
        placeholder="Name"
      />
      <input
        type="email"
        name="email"
        value={formData.email}
        onChange={handleChange}
        placeholder="Email"
      />
      <p>
        {formData.name} / {formData.email}
      </p>
    </form>
  );
}
```

`event.target.name`, değişen input'un `name` attribute'unu verir -- `[name]: value` (computed property name) ile, TEK bir `handleChange` fonksiyonuyla hangi alanın değiştiğini anlayıp yalnızca o alanı güncelleyebiliyoruz. Bu, State dersinde gördüğümüz immutability kuralına uygun: `{ ...formData, [name]: value }`, eski nesneyi kopyalayıp yalnızca bir alanı değiştiren YENİ bir nesne oluşturuyor.

Basit Validation (Doğrulama)

Formu göndermeden önce, girilen değerlerin geçerli olup olmadığını kontrol etmek isteriz -- buna **validation** denir:

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

function RequiredFieldValidationExample() {
  const [name, setName] = useState("");
  const [error, setError] = useState("");

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

    if (name.trim() === "") {
      setError("Name cannot be empty.");
      return;
    }

    setError("");
    console.log("Form submitted:", name);
  }

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

      {error && <p className="error">{error}</p>}
    </form>
  );
}
```

Gönderim anında (`handleSubmit` içinde) değeri kontrol ediyoruz; boşsa `error` state'ine bir mesaj yazıp fonksiyondan erken çıkıyoruz (`return`), formu asıl "göndermiyoruz".

Hata Mesajları Göstermek

Bir hata bulunduğunda, kullanıcıya bunu göstermek gerekir -- Conditional Rendering dersinde gördüğümüz `&&` deseni burada tam yerine oturuyor:

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

function EmailFormatValidationExample() {
  const [email, setEmail] = useState("");
  const [error, setError] = useState("");

  // A simple email format check -- real projects may have more
  // comprehensive rules; here we only check for "@" and a dot.
  const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

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

    if (!emailPattern.test(email)) {
      setError("Enter a valid email address.");
      return;
    }

    setError("");
    console.log("Valid email:", email);
  }

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="email"
        value={email}
        onChange={(event) => setEmail(event.target.value)}
        placeholder="Email"
      />
      <button type="submit">Submit</button>

      {error && <p className="error">{error}</p>}
    </form>
  );
}
```

`error` state'i doluysa (boş string değilse), `{error && <p>...}` ifadesi hata mesajını render eder; `error` boşsa hiçbir şey render edilmez. Bu örnekte ayrıca basit bir e-posta biçimi kontrolü de var -- bir regex (`emailPattern.test(email)`) ile.

Gönderim Öncesi Tüm Formu Doğrulamak

Birden fazla alanlı bir formda, her alanın kendi hatasını göstermesi gerekebilir:

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

function FullFormValidationExample() {
  const [formData, setFormData] = useState({ name: "", email: "" });
  const [errors, setErrors] = useState({});

  function handleChange(event) {
    const { name, value } = event.target;
    setFormData({ ...formData, [name]: value });
```

```
  }

  function validate() {
    // We build a separate error message for each field and collect ALL
    // errors in a single object -- this lets each field show its own error.
    const newErrors = {};

    if (formData.name.trim() === "") {
      newErrors.name = "Name cannot be empty.";
    }

    if (formData.email.trim() === "") {
      newErrors.email = "Email cannot be empty.";
    }

    return newErrors;
  }

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

    const newErrors = validate();
    setErrors(newErrors);

    // If the object has no keys at all, there are no errors.
    if (Object.keys(newErrors).length === 0) {
      console.log("Form is valid, submitting:", formData);
    }
  }

  return (
    <form onSubmit={handleSubmit}>
      <div>
        <input
          type="text"
          name="name"
          value={formData.name}
          onChange={handleChange}
          placeholder="Name"
        />
        {errors.name && <p className="error">{errors.name}</p>}
      </div>

      <div>
        <input
          type="email"
          name="email"
          value={formData.email}
          onChange={handleChange}
          placeholder="Email"
        />
        {errors.email && <p className="error">{errors.email}</p>}
      </div>

      <button type="submit">Submit</button>
    </form>
  );
}
```

`validate()` fonksiyonu, her alan için ayrı bir hata mesajı üretip hepsini TEK bir nesnede (`newErrors`) topluyor. Bu nesnenin hiç anahtarı yoksa (`Object.keys(newErrors).length === 0`), form geçerli demektir. Her input'un altında yalnızca KENDİ hatası (`errors.name` , `errors.email`) gösteriliyor -- bu da yine `&&` ile conditional rendering.

Özet ve Terimler Sözlüğü

Bir form, `onSubmit` ile gönderilir; controlled input'lar sayesinde gönderim anında değerler zaten state'te hazır bulunur. Birden fazla alan tek bir state nesnesinde tutulup `event.target.name` ile hangi alanın değiştiği anlaşılır. Validation, genellikle gönderim anında yapılır; hatalar bir state'te (tek bir mesaj ya da alan başına bir nesne olarak) tutulup `&&` ile conditional rendering'le gösterilir.

Terimler Sözlüğü

Validation (Doğrulama) — Girilen değerlerin beklenen kurallara uyup uymadığını kontrol etme süreci.

Computed Property Name — `{ [name]: value }` gibi, bir nesnenin anahtarını bir değişkenden dinamik olarak belirleme söz dizimi.

Pratik Proje

Bu kategoride (Controlled Components, Form Handling) öğrendiğimiz kavramları bir arada kullanan, gerçek ve çalıştırılabilir bir örnek proje var: **Forms Demo** -- basit bir kayıt (sign up) formu.

Proje; `value` + `onChange` ile controlled input'ları, birden fazla alanı tek bir state nesnesinde yönetmeyi, `onSubmit` + `preventDefault` 'u, ve gönderim öncesi tüm formu doğrulayıp her alanın kendi hatasını göstermeyi 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/forms
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/forms` yapıp `npm run dev` demen yeterli.