Form Handling

Submitting a form with onSubmit, managing multiple fields with a single state object, validation, and displaying error messages -- with simple examples.

Intermediate 13 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/form-handling

---

Form Handling

In Controlled Components, we bound a single input to state. This lesson covers handling a real form end to end -- submission, multiple fields, validation, and error messages.

Submitting a Form: Collecting Values with onSubmit

The `onSubmit` we saw in Events is usually used in forms to work with the value that's already in state:

```
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>
  );
}
```

Since the input is controlled, the `name` state is already up to date by the time the form is submitted -- there's no need to read the value from the DOM, we just use state directly.

Managing Multiple Fields

When a form has more than one input, instead of opening a separate `useState` for each one, it's more manageable to keep them all in ONE state object:

```
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` gives you the `name` attribute of whichever input changed -- with `[name]: value` (a computed property name), a SINGLE `handleChange` function can figure out which field changed and update only that one. This follows the immutability rule from State: `{ ...formData, [name]: value }` copies the old object and creates a NEW object with just one field changed.

Simple Validation

Before submitting a form, you usually want to check whether the entered values are valid -- this is called **validation**:

```
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>
  );
}
```

We check the value at submit time (inside `handleSubmit`); if it's empty, we write a message into the `error` state and exit the function early (`return`) without actually "submitting" the form.

Displaying Error Messages

Once you find an error, you need to show it to the user -- the `&&` pattern from Conditional Rendering fits perfectly here:

```
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>
  );
}
```

If `error` is non-empty, the expression `{error && <p>...}` renders the error message; if `error` is empty, nothing renders. This example also includes a simple email format check -- using a regex (`emailPattern.test(email)`).

Validating the Whole Form Before Submit

In a form with multiple fields, each field may need to show its own error:

```
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>
  );
}
```

The `validate()` function produces a separate error message for each field and collects them all into ONE object (`newErrors`). If that object has no keys (`Object.keys(newErrors).length === 0`), the form is valid. Under each input, only its OWN error (`errors.name` , `errors.email`) is shown -- again with `&&` for conditional rendering.

Summary and Glossary

A form is submitted with `onSubmit` ; because the inputs are controlled, their values are already ready in state by the time of submission. Multiple fields are kept in a single state object, with `event.target.name` used to figure out which field changed. Validation is usually done at submit time; errors are kept in state (as a single message, or an object with one entry per field) and shown with `&&` for conditional rendering.

Glossary

Validation — The process of checking whether entered values follow the expected rules.

Computed Property Name — Syntax like `{ [name]: value }` that lets you set an object's key dynamically from a variable.

Practical Project

There's a real, runnable example project that brings together the concepts from this category (Controlled Components, Form Handling): **Forms Demo** -- a simple sign-up form.

It shows controlled inputs with `value` + `onChange` , managing multiple fields in a single state object, `onSubmit` + `preventDefault` , and validating the whole form before submit with each field showing its own error, all working together. You can download it and run it yourself, and read through the code line by line:

```
git clone https://github.com/cdurgun/react-course-projects.git
cd react-course-projects
npm install
cd projects/forms
npm run dev
```

The `react-course-projects` repo uses **npm workspaces** -- `npm install` only needs to run once, at the repo root, and every project folder shares the same dependencies (no separate `node_modules` per folder). If you've already run `npm install` at the root, you can just `cd react-course-projects/projects/forms` and run `npm run dev` .