Fetching Data

Fetching and sending data to a server in React with fetch -- GET, POST, PUT, DELETE, loading and error handling, with simple examples.

Intermediate 12 min

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

---

Fetching Data

In every example so far, data was defined INSIDE the component (as an array or object). In real applications, data usually comes from a **server**. This lesson covers the basic way to fetch data from a server in React (and send data to it) -- the `fetch` function.

Fetching Data with fetch: GET

`fetch` is a built-in browser function -- it sends an HTTP request to a URL and returns a **Promise**. The `useEffect` we saw in the Hooks lesson is used to fire this request when the component appears on screen:

```
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>
  );
}
```

Since `useEffect`'s second argument is an empty array (`[]`), this request fires only ONCE, when the component first renders. `fetch(url).then(...)` first converts the response to JSON, then writes the incoming data to state with `setCourses` -- when state changes, React re-renders and the list appears on screen.

Showing a Loading State

Some time passes between sending the request and receiving the response -- during that time, we need to show the user that something is happening instead of a blank screen:

```
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>
  );
}
```

The `loading` state starts as `true`; we set it to `false` once the data arrives. While `loading` is `true`, the component renders `<p>Loading...</p>` and returns EARLY -- the rest of the JSX never renders.

Handling Errors

A request doesn't always succeed -- the network can drop, the server can return an error. If we don't handle this, the user is left with a silently blank screen:

```
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` tells us whether the HTTP status code is in the 200-299 range -- `fetch` does NOT automatically reject on statuses like 404 or 500, so we have to check this ourselves and `throw`. `.catch()` catches that error and writes it to the `error` state; `.finally()` turns off `loading` regardless of success or failure.

Sending Data: POST

Besides reading data (GET), a POST request is used to create a new record:

```
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>
  );
}
```

In a POST request, we pass `fetch` a second argument -- an options object: `method: "POST"`, `headers` to tell the server the data we're sending is JSON, and `body` where we put the data we're sending, converted to a string with `JSON.stringify(...)`.

Updating and Deleting: PUT and DELETE

PUT is used to update an existing record, DELETE to remove one:

```
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>
  );
}
```

Like POST, PUT sends a `body` -- but the URL specifies WHICH record to update (`/courses/${courseId}`). DELETE usually sends no `body` at all, specifying only via the URL's id which record to remove.

Summary and Glossary

`fetch` sends an HTTP request to a URL and returns a Promise; it's called inside `useEffect` when the component appears on screen. GET reads data, POST creates a new record, PUT updates an existing record, DELETE removes one -- POST and PUT send JSON in the `body`. A request should always be managed together with `loading` and `error` state -- the user should see a waiting message until the data arrives, and a clear message if something goes wrong.

Glossary

fetch — The browser's built-in function for sending an HTTP request to a URL; returns a Promise.

HTTP Method — A word describing the PURPOSE of a request: GET (read), POST (create), PUT (update), DELETE (remove).

Promise — A JavaScript object representing an operation that hasn't completed yet, but will eventually produce a result (or an error).