React + REST API

Connecting React to a REST API -- separating the API layer into an api.js module, managing listing, creating, and deleting together, with simple examples.

Intermediate 15 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/react-rest-api

---

React + REST API

In Fetching Data, we saw `fetch`'s GET/POST/PUT/DELETE one at a time. This lesson brings them together to manage a resource -- like a course list -- end to end (listing, adding, removing) in a REAL application, and shows the bigger picture of how React talks to a backend.

React → HTTP → Backend → Database

A React application usually doesn't store its own data -- it sends an HTTP request to a **backend** (which could be a Spring Boot application like this very project) with `fetch`. For example, when a course list is requested: React calls `fetch("/courses")` → the request travels over HTTP to the backend → a `@RestController` on the backend (a term you may remember from the REST API Design lesson in the Java course) handles the request → it reads data from the database (PostgreSQL) → returns it as JSON → React writes that JSON to state and updates the screen. This lesson doesn't set up a real Spring Boot backend -- instead it uses a simple fake server (**json-server**) that follows the same REST rules (GET /POST/DELETE, JSON, HTTP status codes); the React-side code would be exactly the same when connecting to a real Spring Boot API.

Separating the API Layer: An api.js Module

Instead of scattering `fetch` calls across every component, it's common practice to gather them in ONE place:

```
// 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>
  );
}
```

Functions like `getCourses()`, `createCourse()`, `deleteCourse()` HIDE the details of `fetch` (the URL, `method`, `headers`) -- components just call these functions, never dealing with `fetch` itself. Defining `BASE_URL` in one place means that when the backend address changes (say, moving from development to production), changing a single line is enough.

Fetching the Course List from the Backend

Fetching data with `async` / `await` inside `useEffect` is a more readable alternative to a `.then()` chain:

```
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`'s callback CANNOT be `async` directly (React doesn't support that) -- so we define a separate `async` function inside it and call it immediately. `await` WAITS for the Promises from `fetch` and `response.json()` to resolve; once the result arrives, state is updated with `setCourses`.

Creating a New Course

The controlled input + `onSubmit` pattern from Forms is combined here with a POST request:

```
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>
  );
}
```

When the form is submitted, the entered value is sent to the backend with a POST request; the backend creates the new record (usually with an `id` it generates itself) and returns it. `onCreated(newCourse)` is a callback prop used to notify the PARENT component of this new record -- the "lifting data up" pattern from Props.

Deleting a Course

After deleting a record from the server, we also need to update the list on screen:

```
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>
  );
}
```

AFTER the DELETE request succeeds, following the immutability rule from State, we use `filter()` to remove the deleted record and create a NEW array -- we don't mutate the array directly (like with `splice`).

Updating the Screen After Creating

After creating a new record, instead of re-fetching the whole list (sending another request), it's often preferred to add the new record directly to the existing list:

```
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])` uses the spread pattern from State to copy the old list and append the new record -- the screen updates IMMEDIATELY, without a second request to the server.

Summary and Glossary

A React application reads and writes data by sending HTTP requests to a backend (like Spring Boot) with `fetch`; gathering `fetch` calls in a separate `api.js` module is common practice. Fetching data is usually done with `async` / `await` inside `useEffect`. When a record is created or deleted, the list on screen must be updated following State's immutability rules (`filter()`, spread) -- sometimes, instead of sending a second request to the server, it's enough to reflect the result we already have directly into state.

Glossary

REST API — A backend interface design style that manages resources (like courses) through HTTP methods (GET/POST/PUT/DELETE).

api.js Module — A helper file that gathers all of an application's `fetch` calls in one place, used directly by components.

Practical Project

There's a real, runnable example project that brings together the concepts from this category (Fetching Data, React + REST API): **API & Data Fetching Demo** -- a course list application connected to a fake REST API running on json-server.

It shows listing with `useEffect` + `fetch`, managing `loading` / `error` state, creating a new record with a controlled form (POST), deleting a record (DELETE), and updating the screen after every mutation following the immutability rules, 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/api-data-fetching
npm run server
```

After starting the fake API (json-server) in one terminal, start the React app in another:

```
cd react-course-projects/projects/api-data-fetching
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/api-data-fetching` and run the two commands above (`npm run server`, `npm run dev`) in two separate terminals.