React + Spring Boot Deployment

Deploying React to Vercel, a real Spring Boot REST API to Render (Docker), managing the backend address with environment variables, and production CORS configuration -- with simple examples.

Intermediate 14 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/react-spring-boot-deployment

---

React + Spring Boot Deployment

In Build & Deployment, we deployed a React app to Vercel -- but that app had no backend at all. This lesson covers a React app connected to a REAL Spring Boot backend, deployed to two SEPARATE platforms.

Why Two Separate Platforms?

Vercel is designed for static sites and short-lived serverless functions -- it supports runtimes like Node.js/Python, but it can't host the CONTINUOUSLY RUNNING Java server process (embedded Tomcat) that Spring Boot needs. So a common real-world pattern is: deploy React to Vercel, and deploy the Spring Boot backend SEPARATELY to a platform built for long-running servers (**Render**, in this lesson).

Reading the Backend Address from an Environment Variable

The same pattern from Build & Deployment, this time for the backend's address:

```
// The same pattern as in the Build & Deployment lesson -- except this time
// the value is the REAL address of a deployed Spring Boot API. Locally
// (against the backend running on localhost:8080) and in production
// (against the address Render provides), the SAME code works with a
// different .env / Vercel environment variable value.
const API_BASE_URL = import.meta.env.VITE_API_BASE_URL ?? "http://localhost:8080";

function ApiAddress() {
  return (
    <p>
      Data from <code>{API_BASE_URL}</code>
    </p>
  );
}

export default ApiAddress;
```

Locally, `VITE_API_BASE_URL` points at Spring Boot running on `localhost:8080`; in production (Vercel's environment variables) it points at the REAL address Render gives you. The code itself never changes.

Fetching from a Deployed Backend

The `useEffect` + `fetch` +loading/error pattern from API & Data Fetching, now running against a real deployment:

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

// The useEffect+fetch+loading/error pattern from the API & Data Fetching
// lesson -- except on the other end there's no longer json-server, but a
// REAL Spring Boot application deployed to Render.
const API_BASE_URL = import.meta.env.VITE_API_BASE_URL ?? "http://localhost:8080";

function CourseList() {
  const [courses, setCourses] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    async function loadCourses() {
      try {
        const response = await fetch(`${API_BASE_URL}/api/courses`);
        if (!response.ok) {
          throw new Error(`Request failed: ${response.status}`);
        }
        setCourses(await response.json());
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    }

    loadCourses();
  }, []);

  if (loading) return <p>Loading...</p>;
  if (error) return <p>Could not reach the backend: {error}</p>;

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

export default CourseList;
```

This component's behavior is completely unaffected by the switch from `json-server` to a real Spring Boot app on the other end -- all `fetch` sees is a URL and a JSON response.

CORS: Allowing a Deployed Frontend

In Advanced Spring MVC, we wrote a GLOBAL CORS configuration with `addCorsMappings`. In a real deployment, this reads from an environment variable instead of a hardcoded domain:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

// This is what GlobalCorsConfigExample (addCorsMappings) from the Advanced
// Spring MVC lesson looks like when used in a real deployment. allowedOrigin
// is read from a CORS_ALLOWED_ORIGIN environment variable via
// application.properties -- without changing the code at all, it's enough
// to fill in this variable in the Render Dashboard with the real address
// Vercel provides.
@Configuration
class DeploymentCorsConfigExample implements WebMvcConfigurer {

    @Value("${app.cors.allowed-origin}")
    private String allowedOrigin;

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
                .allowedOrigins(allowedOrigin, "http://localhost:5173")
                .allowedMethods("GET");
    }
}
```

`allowedOrigin` comes from the `CORS_ALLOWED_ORIGIN` environment variable via `application.properties` -- filling in that variable on the Render Dashboard with the REAL address Vercel gave you is enough, the code never changes. If the origin doesn't match EXACTLY (including `https://`, with NO trailing `/`), the browser blocks the request with a CORS error.

Deploy Order: The Chicken-and-Egg Problem

Since React needs to know the backend's address, and the backend needs to know React's address, the order matters:

1. **Deploy the backend to Render first** -- it gives you a URL.
2. **Then deploy React to Vercel**, filling `VITE_API_BASE_URL` with the URL from step 1 -- it gives you ANOTHER URL.
3. **Go back to Render**, and fill `CORS_ALLOWED_ORIGIN` with the URL from step 2. This automatically restarts the backend.

Skip this order (e.g. never fill in the CORS variable) and the frontend reaches the backend, but the browser REFUSES to hand the response to JavaScript -- you'll see a successful (200) request in the Network tab, but a CORS error in the console.

Summary and Glossary

React and Spring Boot are deployed SEPARATELY, to different platforms (Vercel + Render) -- each knows the other's address through an environment variable. The backend's CORS

configuration reads from an environment variable instead of a hardcoded domain, so which frontend is allowed can be changed without touching the code. Deploy order matters: backend first, then frontend, then update the backend's CORS setting.

Glossary

CORS (Cross-Origin Resource Sharing) — The mechanism that lets a browser hand a response from a server on a different origin (domain) to JavaScript.

Origin — The scheme + host + port triple (e.g. `https://example.vercel.app`).

Practical Project

There's a project using the concepts from this lesson that's ACTUALLY been deployed:

- Backend (Render): **fullstack-deployment-backend-zst1.onrender.com**
- Frontend (Vercel): **React + Spring Boot Deployment Demo**
- Source code: **fullstack-deployment (React)** and **backend/fullstack-deployment (Spring Boot)**

Open the live page and you'll see the "Java", "React", "Spring Boot" list coming from `GET /api/courses` -- since CORS is configured correctly, the browser lets the response from a different domain (`onrender.com`) reach the React code.

```
git clone https://github.com/cdurgun/react-course-projects.git
cd react-course-projects

# Backend (requires Java 21 + Maven)
cd backend/fullstack-deployment
mvn spring-boot:run

# In a separate terminal: the frontend
cd ../../
npm install
cp projects/fullstack-deployment/.env.example projects/fullstack-deployment/.env
cd projects/fullstack-deployment
npm run dev
```