🚀 Crash Course: Develop & Deploy a Full-Stack Web App with React, Express, AWS Amplify & Netlify

In this guide, you'll quickly set up a full-stack web application using **React** for the frontend, **Bootstrap 5** for styling, and **Express.js** for the backend. We'll deploy the frontend on **AWS Amplify** and the backend on **Netlify**, with a **custom domain** managed via **Route 53**. Let's get started! 🎯

📌 Prerequisites

Before you begin, make sure you have:

- Node.js & npm installed → Download here
- Git installed → Download here
- AWS & Netlify accounts ready

🏗️ Step 1: Set Up the Frontend (React + Bootstrap 5)

1️⃣ Create a New React App

```
npx create-react-app my-app
cd my-app
```

2️⃣ Use Bootstrap (CDN or Install via npm)

Option 1: Use Bootstrap via CDN

Add this line to the `<head>` section of `public/index.html`:

```
<link href="https://cdn.jsdelivr.net/npm/[email protected]/dist/css/bootstrap.min.css"
rel="stylesheet">
```

Option 2: Install Bootstrap via npm

```
npm install bootstrap
```

Then import it in `src/index.js`:

```
import 'bootstrap/dist/css/bootstrap.min.css';
```

3️⃣ Create a Simple Homepage Component

Edit `src/App.js`:

```
import React from "react";

function App() {
  return (
    <div className="container text-center">
      <h1>Welcome to My Full-Stack App 🚀</h1>
      <p>This is a React app styled with Bootstrap 5.</p>
    </div>
  );
}

export default App;
```

🚀 Step 2: Deploy Frontend on AWS Amplify

1. Push the code to GitHub:

```
git init
git add .
git commit -m "Initial commit"
```

```
git branch -M main
git remote add origin <your-repo-url>
git push -u origin main
```

2. Go to **AWS Amplify** → Connect your GitHub repo
3. Deploy the frontend by following AWS Amplify's setup

   After you click through everything, find your app in the console and click. Now you are at your app's management console. On the left side menu, go to **Hosting** -> **Build settings**:

   Edit `amplify.yml`:

```
version: 1
frontend:
  phases:
    preBuild:
      commands:
        - npm i
    build:
      commands:
        - npm run build
  artifacts:
    # IMPORTANT - Please verify your build output directory
    baseDirectory: dist
    files:
      - '**/*'
  cache:
    paths:
      - node_modules/**/*
```

4. (Optional) Set up a custom domain via **Route 53**

The default domain Amplify provides doesn't have meanings. If you want to have a custom domain, you can buy it from a domain provider. There are a lot of providors out there, I choose to use AWS **Route 53**.

Route 53 charges **$15 CAD** for the registration fee, and the monthly cost is based on the usage. You can visit

AWS Pricing Calculator for more details.

Once you have your own domain, you can set up **Rewrites and redirects** in your app's management console in **Amplify**. In your app's management console, simply go to **Hosting** -> **Rewrites and redirects** -> **Manage redirects**. Then add your custom domain to **Source address** and **Target address**, and set **Type** to `302 (Redirect - Temporary)`.

Now people can visit your website with a meaningful URL. 🎉

5. Trigger Deploy

Either push your code back to the `main` branch or manually trigger a deployment in the Amplify console. They both works. You will be able to see the build and deploy details in the console. You can see if the deployment is successful in the console.

🔥 Step 3: Build the Backend with Express.js

1️⃣ Create an Express Backend

First, we need to create and init the project folder, then install all the dependencies. The `@latest` just make sure we install the latest versions. However, for `node-fetch` you need **version 2**. The latest version is ver3, but you will face `ModuleNotFoundError: Module not found: Error: Can't resolve 'node:https'`, and install version 2 is a work around.

```
mkdir backend && cd backend
npm init -y
npm install express@latest cors@latest serverless-http@latest axios@latest dotenv@lates
t node-fetch@2.6.1
```

Here's the installation command along with a brief introduction to each package:

📌 Package Overview

- **express** – A lightweight web framework for handling HTTP requests and building APIs in Node.js.
- **cors** – Middleware for enabling Cross-Origin Resource Sharing (CORS), allowing your API to be accessed from different domains.
- **serverless-http** – Helps deploy Express apps to serverless platforms like AWS Lambda by wrapping them into a compatible handler.
- **axios** – A promise-based HTTP client for making API requests, handling responses, and managing errors efficiently.
- **dotenv** – Loads environment variables from a `.env` file, keeping sensitive configuration data separate from your code.
- **node-fetch** – A Fetch API implementation for Node.js, used for making HTTP requests in a more modern, `fetch()`-like style.

This setup ensures your backend is ready for API development, deployment, and secure configuration handling. 🚀

📌 Necessary files

To make sure our backend can be successfully deoloyed on Netlify, we need a `netlify.toml` file and a `dist` folder. Inside the `dist` folder, create a `index.html` file. The `index.html` file can be empty. Every time you need to deploy something on Netlify, you need to give it a `dist` folder. It is the **publish directory** Netlify's looking for.

Now, create the `netlify.toml` file directly in your project folder. `netlify.toml` is the configuration file for Netlify, it tells how Netlify should build your app.

```
[build]
  functions = "functions"
  node_bundler = "esbuild"

[functions]
  node_bundler = "esbuild"
```

The idea behind this is Netlify treat your backend as serverless functions. It will build and output to the

`/.netlify/functions` folder. We will see this later in the code.

📌 Create `api.js`:

In the project folder, create a `src` folder to store the source codes. You can name this file whatever you want.

1. First we import all the necessary libraries.

```
const express = require("express");
const serverless = require("serverless-http");
const dotenv = require("dotenv");
const cors = require("cors");
const axios = require("axios");
const fetch = require("node-fetch");
```

2. Basics of Express.js

   To use Express.js, we simply import it by using `require("express")` and create an instance of it. To make the lambda run, we also need it be able to export to a handler function and wrap the handler with `serverless`. Lastly, we need `Router` to create our `route`. A `route` responds with a message when a URL is accessed. For example, a single slash `/` responds to the root URL. `dotenv` will be used later in the tutorial.

```
// ...

dotenv.config();
const app = express();
const router = express.Router();

module.exports.handler = serverless(app);
```

This is an example of when visitors access the root URL of your api, a message in JSON format will be sent down.

```
//...
```

```
const router = express.Router();
router.get('/', (req, res) => {
   res.json({
       message: "Hello World!"
   });
});
//...
```

Then we need to bind the router into the app. What the Netlify does is put all of your functions into the `/.netlify` directory, and whatever you passed in for the `function` parameter in the `netlify.toml` file. Then we tell it the name of our function.

```
// defined route...
app.use('/.netlify/functions/api', router);
module.exports = app;
// export handler...
```

📌 run and build commands

We need to tell Netlify how to build and start in production. Edit `package.json`:

Find `"scripts"` and add these in.

```
  "scripts": {
    "start": "netlify-lambda serve src",
    "build": "netlify-lambda build src",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
```

2️⃣ Test the Backend

Now we should be able to run `npm start` and test out backend locally.

In terminal, run:

```
npm start
```

Based on the console output, visit the URL in your browser. My app is published to `port 9000`, and remember, the functions are in `/.netlify/functions/` directory. So visit `http://localhost:9000/.netlify/functions/api` in your browser. In your case, replace the port number accordingly.
You should be able to see this in your browser:

```
message "Hello World!"
```

🌍 Step 4: Deploy Backend on Netlify

1️⃣ Install Netlify CLI & Set Up Functions

```
npm install -g netlify-cli
netlify login
netlify init
```

Simply enter your credentials when you are asked to.

2️⃣ Deploy to Netlify

```
netlify deploy --prod
```

After you run this command, Netlify will ask you to enter your `publish directory`. We have `dist`, so enter `dist`. Wait for a while and then Netlify will output the link to the terminal, and you can follow the URL to access your backend.

```
Website URL:  https://[app-name].netlify.app
```

Don't forget to add `/.netlify/functions/api` after the URL. Access `https://[app-`

`name].netlify.app/.netlify/functions/api` and you should be able to see the same thing as you just saw in the local test.

🔗 Connect to Amplify

Cross-Origin Resource Sharing

Now we need to use our API in our React frontend. However, recall that our frontend is deployed on AWS Amplify, and our backend is deployed on Netlify. So we'll run into `Cross-Origin Resource Sharing (CORS)` issues if we use our API URL directly. Check out this link for more details on CORS if you are interested.

- https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

To solve this, we need to enable `CORS` in our backend. After we created the instance of `express`, use `app.use(cors());`
So our backend code should look like this:

```
const express = require("express");
const serverless = require("serverless-http");
const dotenv = require("dotenv");
const cors = require("cors");

dotenv.config();
const app = express();
const router = express.Router();
app.use(cors()); // enables cors
router.get('/', (req, res) => {
    res.json({
        message: "Hello World!"
    });
});

app.use('/.netlify/functions/api', router);
```

```
module.exports = app;
module.exports.handler = serverless(app);
```

React `useEffect` and `useState`

Let's first import them `import { useState, useEffect } from "react";`

1. What is `useEffect` in React?

   `useEffect` is a React Hook that allows you to perform **side effects** in functional components. Side effects include **fetching data from an API, subscribing to events, manipulating the DOM**, etc.

2. What is `useState` in React?

   `useState` is a React Hook that **allows functional components to have state**. It enables components to store and update values dynamically, triggering re-renders when the state changes.

   2.1. How to use `useState`

   `const [state, setState] = useState(null);`

   Breakdown of `const [state, setState] = useState(null);`

   2.1.1 `useState(initialValue)` - Parameters

   - `useState(null)` → The argument (`null`) is the **initial state value**.
   - The initial value can be **any type**:
     - `useState(0)` → Number
     - `useState("")` → String
     - `useState([])` → Array
     - `useState({})` → Object
     - `useState(false)` → Boolean

   2.1.2. `const [state, setState] = useState(null);` - Naming Conventions

   - **First variable (`state`)** → Stores the current state value.

- **Second variable (`setState`)** → A function that updates the state.
- The naming convention follows `[value, setValue]`:

```
const [count, setCount] = useState(0);     // For numbers
const [name, setName] = useState("");      // For strings
const [user, setUser] = useState({});      // For objects
const [isOpen, setIsOpen] = useState(false); // For booleans
```

2.1.3. `useState(null);` - Initial Value Considerations

- `null` is often used when the initial state is unknown or will be fetched later (e.g., API data).
- If the state is a **complex object**, it's best to initialize it with a meaningful default, like an empty object `{}` or array `[]`.

TL;DR:

- `useState(initialValue)` sets an initial state.
- `const [state, setState] = useState(null);` follows `[state, setState]` naming.
- Use meaningful initial values (`0`, `""`, `false`, `{}`, `[]`) based on the use case. 🚀

2.2. Key Features of `useState`

- **Manages Component State** – It allows variables to persist across renders instead of being reset.
- **Triggers Re-renders** – When the state changes, React re-renders the component automatically.
- **Returns an Array** – It provides the **state variable** and a **function to update it**.

2.3. When to Use `useState`

✅ When you need to store and update data inside a component (e.g., form inputs, toggles, API responses).

✅ When a UI change depends on user interaction (e.g., button clicks, user selections).

2.4. Common Pitfalls

❌ **Directly modifying state:** Always use the updater function instead of modifying the variable directly.

❌ **Using state incorrectly in async functions:** State updates are **asynchronous**, so always

update based on the previous state if needed.

TL;DR: `useState` is the go-to Hook for managing local component state in React functional components. 🚀

3. How to Use `useEffect` to Fetch Data from an API

You typically use `useEffect` to fetch data when a component mounts. Here's a simple example using `fetch` and `axios`:

Let's see some examples

Using `fetch`

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

function MyComponent() {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/posts/1")
      .then((response) => response.json())
      .then((data) => setData(data))
      .catch((error) => console.error("Error fetching data:", error));
  }, []); // The empty dependency array means this runs **only once** when the componen
t mounts.

  return <div>{data ? <p>{data.title}</p> : <p>Loading...</p>}</div>;
}

export default MyComponent;
```

Using `axios`

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

```
import axios from "axios";

function MyComponent() {
  const [data, setData] = useState(null);

  useEffect(() => {
    axios.get("https://jsonplaceholder.typicode.com/posts/1")
      .then((response) => setData(response.data))
      .catch((error) => console.error("Error fetching data:", error));
  }, []);

  return <div>{data ? <p>{data.title}</p> : <p>Loading...</p>}</div>;
}

export default MyComponent;
```

Explanation

- `useEffect(() => {...}, [])` : Runs **only once** when the component mounts (empty dependency array `[]`).
- `fetch` or `axios` is used to get data from an API.
- `useState` stores the fetched data.
- The component re-renders when `setData(data)` updates the state.

This pattern is commonly used for **fetching data, subscribing to WebSockets, or interacting with APIs** in React. 🚀

Differences between `fetch` and `axios`

Both `fetch` and `axios` are used for making HTTP requests in JavaScript, but they have some key differences:

1. Simplicity & Syntax

- `fetch` requires more manual handling of responses.
- `axios` automatically transforms responses and provides simpler syntax.

Fetch Example

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then((response) => response.json()) // Must manually convert response to JSON
  .then((data) => console.log(data))
  .catch((error) => console.error("Fetch error:", error));
```

Axios Example

```
import axios from "axios";

axios.get("https://jsonplaceholder.typicode.com/posts/1")
  .then((response) => console.log(response.data)) // No need to convert JSON manually
  .catch((error) => console.error("Axios error:", error));
```

---

2. Error Handling

- `fetch` **does not reject** on HTTP errors (e.g., 404, 500). You must handle errors manually.
- `axios` **automatically rejects** on non-2xx responses.

Fetch Handling HTTP Errors

```
fetch("https://jsonplaceholder.typicode.com/posts/123456") // Invalid ID
  .then((response) => {
    if (!response.ok) throw new Error(`HTTP error! Status: ${response.status}`);
    return response.json();
  })
  .catch((error) => console.error("Fetch error:", error));
```

Axios Handles Errors Automatically

```
axios.get("https://jsonplaceholder.typicode.com/posts/123456")
  .then((response) => console.log(response.data))
```

```
  .catch((error) => console.error("Axios error:", error)); // Automatically catches HTTP errors
```

---

3. Request & Response Interception

- `axios` allows intercepting requests and responses (e.g., adding headers, logging).
- `fetch` does not have built-in interception.

Axios Interceptors

```
axios.interceptors.request.use((config) => {
  console.log("Request sent:", config);
  return config;
});
```

---

4. Automatic JSON Handling

- `fetch` requires `response.json()` to parse JSON manually.
- `axios` automatically parses JSON.

---

5. Browser & Server-Side Support

- `fetch` is **native** to modern browsers and does not require installation.
- `axios` **works in Node.js** without extra polyfills, making it great for server-side applications.

---

6. Features Comparison

Feature	Fetch	Axios
Auto JSON parsing	❌ No	✅ Yes
Error handling	❌ Manual	✅ Automatic
Request cancellation	❌ No	✅ Yes (via `CancelToken`)
Interceptors	❌ No	✅ Yes
Timeout Handling	❌ No (requires `AbortController`)	✅ Yes (built-in)
Works in Node.js	❌ No (needs polyfill)	✅ Yes

When to Use What?

✅ **Use `fetch` if:**

- You need a lightweight, native solution in the browser.
- You're okay with manually handling errors and JSON parsing.

✅ **Use `axios` if:**

- You want easier error handling and automatic JSON conversion.
- You need advanced features like request cancellation, interceptors, and timeout handling.
- You're working in a Node.js environment.

TL;DR: `axios` is easier to use and has more features, but `fetch` is native and does not require installation. 🚀

Apply on our frontend

You can create a new component, but for demo purpose, we keep editing our `App.js`.
As introduced before, we use `useEffect` to fetch the data from the api we built, then set the `state` by passing the data into `setState`. Finally, we can use the data by wrapping it with a curly braket: `{state}`.
Edit `src/App.js`:

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

function App() {
    const [state, setState] = useState(null);
    useEffect(() => {
        fetch("your-api")
            .then((response) => response.json())
            .then((data) => setState(data))
            .catch((error) => console.error("Error fetching data:", error));
    }, []);

  return (
```

```
    <div className="container text-center">
      <h1>Welcome to My Full-Stack App 🚀</h1>
      <p>This is a React app styled with Bootstrap 5.</p>
      <p>{state}</p>
    </div>
  );
}

export default App;
```

Now you can push the code back to the main branch to trigger the deployment on AWS Amplify.

🌐 Step 5: Use a third-party api in backend

Sometimes we will need to use a third-party in our backend. In this demo, let's use OpenWeather API. Most of the API providors need registration and subscription, a lot of them have free tiers. After you go through the registration process, log into your account and the **API** page. The api we are going to use is `Current Weather Data`, you need to subscribe it.

Then go to **My API keys** at the top right corner, click your name. You can find your api key here, copy it for later use.

Set environment variables

API key is considered a secret that only yourself should know. Thus we can not show it in the code or send it in plaintext. To protect secrets like this, we need `environment variables`. And this is also why we installed the `dotenv` library before.

First, in the backend project folder create a `.env` file. Put your secret in it.

```
NODE_ENV=development
API_KEY="Your api key"
```

You can name the secret whatever you like.

Note: Some frameworks or platform may require you to follow a sepecific naming convention, otherwise can't read the value. For example, React asks to name all the environment variables starts with `REACT_APP_`. So a secret api key would look like `REACT_APP_SEC_API` in React.
Then, add this `.env` into the `.gitignore` file to make sure it is not sent to the remote repo. (Unless you want to reveal your secret to the public)
Now you can access the `environment variable` by using `process.env.YOUR_VARIABLES`. In our case, `process.env.API_KEY`.

Send requests

Always check the API documentations to find out how to use the API.

Take `OpenWeather API` as an example, `https://api.openweathermap.org/data/2.5/weather?` is the base endpoint, and add requests after the question mark. And at the end of the URL, add your API key to it.

According to the documentation, if I would like to access the weather info of **Halifax,CA** and show results in **metric** units, we need to construct this URL: `https://api.openweathermap.org/data/2.5/weather?q=halifax,ca&&units=metric&appid=${process.env.WEATHER_API_KEY}`. Note: here I name my API key as `WEATHER_API_KEY` in the `.env` file.

Now we can send a request to the endpoint using `get` method and `fetch`.

```
router.get('/weather', async (req, res) => {
    fetch(weatherAPI)
        .then(res => res.json())
        .then(json => console.log(json))
        .catch(err => console.error('error:' + err));
    try {
        let response = await fetch(weatherAPI);
        response = await response.json();
        res.status(200).json(response);
    } catch (err) {
        console.log(err);
        res.status(500).json({msg: `Internal Server Error.`});
```

```
    }
});
```

The return value is a JSON object that contains a lot of informaion. You will need to extract what's useful to you. I recommend you copy the response to a place so you can have a reference.

Here are the information I want:

```
// Extract required fields
const weatherData = {
            city: response.name,
            country: response.sys.country,
            temperature: {
                current: response.main.temp,
                feels_like: response.main.feels_like,
                min: response.main.temp_min,
                max: response.main.temp_max,
            },
            wind: {
                speed: response.wind.speed,
                direction: response.wind.deg
            },
            humidity: response.main.humidity
        };
```

And now our backend looks like this:

```
const express = require("express");
const serverless = require("serverless-http");
const dotenv = require("dotenv");
const cors = require("cors");

const fetch = (...args) =>
    import('node-fetch').then(({default: fetch}) => fetch(...args));
```

```
dotenv.config();
const app = express();
const router = express.Router();

app.use(cors());
const weatherAPI = `https://api.openweathermap.org/data/2.5/weather?q=halifax,ca&&units
=metric&appid=${process.env.WEATHER_API_KEY}`;

router.get('/weather', async (req, res) => {
    fetch(weatherAPI)
        .then(res => res.json())
        .then(json => console.log(json))
        .catch(err => console.error('error:' + err));
    try {
        let response = await fetch(weatherAPI);
        response = await response.json();
        // Extract required fields
        const weatherData = {
            city: response.name,
            country: response.sys.country,
            temperature: {
                current: response.main.temp,
                feels_like: response.main.feels_like,
                min: response.main.temp_min,
                max: response.main.temp_max,
            },
            wind: {
                speed: response.wind.speed,
                direction: response.wind.deg
            },
            humidity: response.main.humidity
        };
        res.status(200).json(weatherData);
```

```
    } catch (err) {
        console.log(err);
        res.status(500).json({msg: `Internal Server Error.`});
    }
});

// Home route
router.get('/', (req, res) => {
    res.json({
        message: "Hello World!"
    });
});

app.use('/.netlify/functions/api', router);

module.exports = app;
module.exports.handler = serverless(app);
```

The directory structure looks like this:

📦 EXPRESS-NETLIFY
├── 📂 .netlify # Generate by Netlify
├── 📂 dist # Static files (e.g., built frontend)
│ ├── index.html # Entry point for frontend (if applicable)
├── 📂 functions # Serverless functions for Netlify
├── 📂 node_modules # Installed dependencies (auto-generated)
├── 📂 src # Source code
│ ├── api.js # Express API logic
├── .env # Environment variables (ignored in Git)
├── .gitignore # Git ignore file
├── netlify.toml # Netlify configuration file
├── package-lock.json # Lockfile for npm dependencies
├── package.json # Project dependencies & scripts

🎉 Conclusion

You now have a **React frontend deployed on AWS Amplify** and an **Express backend running on Netlify**! 🎯 You can further expand this by adding authentication, a database, or API routes.
