Building a Full-Stack CRUD Application with Express.js and React.js

1. Introduction: The Full-Stack Foundation

A full-stack application encompasses both the client-facing side, known as the frontend, and the server-side, called the backend. For a beginner, a helpful way to conceptualize this relationship is to use the analogy of a car. The frontend is the vehicle's exterior—the visual elements, buttons, and dashboard that a user sees and interacts with. It defines the structure using HTML, applies styling with CSS, and adds interactivity with JavaScript. In contrast, the backend is the car's intricate machinery: the engine, transmission, and all the mechanical systems that make it function. It works behind the scenes to process data, manage application logic, and ensure the entire system runs efficiently and securely. While a beautifully designed car (frontend) might look impressive, it cannot run without the internal machinery (backend) working correctly.
This handbook aims to demystify the backend by focusing on a core set of operations known as CRUD. The acronym CRUD stands for Create, Read, Update, and Delete. These four functions are considered the essential building blocks for any application that manages persistent data, whether it's stored in a traditional database or a temporary in-memory structure. For example, in a simple to-do list application, creating a new task, viewing a list of all tasks, editing an existing task's title, and deleting a completed task all correspond directly to the four CRUD operations. The pedagogical approach of this guide is to demonstrate these fundamental patterns. By implementing a simple data structure, a reader can master the logical flow of a full-stack application without being overwhelmed by the complexities of a database.
The technologies chosen for this project are Express.js and React.js. Express.js is a minimal and flexible web application framework built on Node.js that provides a robust set of features for creating web servers and APIs. Its unopinionated nature gives developers significant freedom in how they structure their projects, which is both a strength and a challenge for beginners. React.js, on the other hand, is a popular JavaScript library for building user interfaces with a component-based architecture. The objective of this handbook is to guide the reader through the creation of a "Simple User Manager" application, which will perform all four CRUD operations. The project will start with a simple in-memory array to illustrate the core concepts and provide a clear, actionable path for transitioning to a real database in the future.

2. Project Setup: Laying the Groundwork

2.1 Installing Node.js and the Development Environment

The first step in building a full-stack application is to install Node.js, the runtime environment for executing JavaScript on the server. Node.js can be downloaded directly from its official website, and the installation also includes npm (Node Package Manager). npm is a crucial command-line utility for managing project dependencies and packages.
A modern code editor, such as Visual Studio Code (VS Code), is highly recommended due to its integrated terminal, which streamlines the development workflow.

2.2 Backend Setup: Initializing the Express.js Project

To begin the backend portion of the project, a new directory must be created. Once inside the directory, the command npm init -y is used to initialize a new Node.js project. This command creates a package.json file with default settings, which serves as a manifest for the project, tracking metadata and dependencies.
Next, the Express.js framework is installed by running npm install express. This command downloads the Express package and saves it as a dependency in the package.json file, making it available for use in the project.
To enhance the development experience, a tool called nodemon is used. Nodemon is a utility that automatically restarts the server whenever file changes are detected, eliminating the need for a developer to manually stop and start the server after every code modification. This automation is a small but critical detail that prevents a major point of frustration for beginners, allowing them to focus on writing code and seeing changes reflected immediately. To install nodemon, the command npm install nodemon --save-dev is used. The --save-dev flag specifies that this package is a development-only dependency and will not be included in the final production build. A script can be added to the package.json file to make running the server even easier:

```
"scripts": {
  "dev": "nodemon server.js"
}
```

With this script, the server can be started by simply running npm run dev in the terminal.

2.3 Frontend Setup: Initializing the React.js Project

The frontend project can be set up in a separate directory to maintain a clean project structure. For this guide, a modern build tool called Vite is recommended over older alternatives like Create React App due to its speed and leaner development experience. To create a new React project with Vite, the following command is executed in the terminal: npm create vite@latest frontend-app -- --template react.
After the project is scaffolded, navigating into the new directory (cd frontend-app) and running npm install will install all the required dependencies. Finally, npm run dev starts the React development server, which typically runs on a separate port from the backend, such as port 5173. This separation of the frontend and backend development environments is a core concept in full-stack development that will be explored in more detail later.

3. Express.js Basics: Your First Backend Server

3.1 The "Hello, World!" of Express.js

A minimal Express.js application can be created with just a few lines of code. This simple "Hello, World!" server demonstrates the core components of the Express framework.

```
const express = require('express');
const app = express();
const port = 3000;
```

```
app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`);
});
```

This code snippet can be saved in a file named server.js within the backend project directory. The first line loads the Express module. The second line creates an instance of an Express application, which is the object that will be used to define routes and middleware. The app.get() method is a route handler that listens for a GET request on the root URL (/). The callback function inside app.get() is executed when a request is received, and it uses res.send() to send the string "Hello World!" back to the client. Finally, app.listen() starts the server and tells it to listen for incoming requests on a specific port, in this case, port 3000.

3.2 The Request (req) and Response (res) Objects Explained

In every Express route handler, two objects are provided: the request object (req) and the response object (res). The req object encapsulates all the information about the incoming HTTP request from the client, including its parameters, headers, and body. The res object is used to build and send the HTTP response back to the client.
For a beginner, understanding how to access data from the req object is crucial. Data from the client can be sent in several ways, each corresponding to a different property on the req object. The req.body property contains data submitted in the request body, which is typically used for POST or PUT requests. The req.params property holds the route named parameters from the URL, such as the :id in a route like /users/:id. For data sent via query strings (e.g., /search?query=express), the req.query property contains all the query string parameters. The req.method property indicates the HTTP method used (e.g., GET, POST), and req.url provides the full URL path. A clear understanding of these properties is essential to avoid common errors where the code fails to access the correct data.

3.3 Understanding Routes and HTTP Methods

A route in Express is a section of code that associates an HTTP verb (GET, POST, PUT, etc.), a URL path, and a handler function that processes requests to that path. This approach aligns with the principles of RESTful API design.
The analysis of a CRUD application's core functions reveals a direct and standardized mapping to HTTP methods. This conceptual link is fundamental for building any modern web service.

CRUD Operation	HTTP Method	Purpose
Create	POST	Used to submit data to a specified resource, often resulting in a new entry.
Read	GET	Used to retrieve data from a specified resource.
Update	PUT	Used to update or replace an existing resource with new

CRUD Operation	HTTP Method	Purpose
		data.
Delete	DELETE	Used to remove a specified resource.

This table provides a standardized mental model that will serve a developer well beyond this specific project. While PATCH is another method often used for partial updates, for a beginner, focusing on PUT for full updates is a good starting point.

3.4 Introduction to Express Middleware

Middleware functions are a powerful feature of Express. They are functions that have access to the req and res objects and can execute any code, modify the request or response, and even end the request-response cycle. A middleware function must call the next() function to pass control to the next middleware or route handler in the application's stack.
A simple example is a logging middleware that prints information about every incoming request.

```
app.use((req, res, next) => {
  console.log(`${req.method} request for ${req.url}`);
  next();
});
```

This middleware is loaded using app.use(). The order in which middleware is loaded is critically important; functions that are loaded first are also executed first. A common and confusing error for a new developer occurs when the middleware for parsing JSON data, express.json(), is placed after a route handler that needs access to the request body. In that scenario, req.body will be undefined because the middleware responsible for populating it has not yet run. This specific ordering requirement highlights how the sequential "chain of command" of middleware directly impacts the availability of data, turning a potential bug into a valuable lesson on application flow.

4. CRUD Operations in Express.js: The Backend API

4.1 The Data Model: A Simple In-Memory Array

For this beginner-friendly project, a simple in-memory array will serve as a temporary "database". This approach simplifies the project by removing the overhead of setting up and managing a real database, allowing the reader to focus on the core logic of the Express routes. The data will not be persistent and will be reset every time the server restarts, but this is a perfectly acceptable trade-off for a learning exercise.

```
let users =;
```

4.2 Create (POST): Adding a New User

To add a new user, a POST request is used, typically to an endpoint like /users. The server needs to handle this request by extracting data from the request body.

```
app.post('/users', (req, res) => {
  const newUser = {
    id: users.length + 1,
    name: req.body.name,
    email: req.body.email
  };
  users.push(newUser);
  res.status(201).json(newUser);
});
```

This handler takes the incoming data from req.body, assigns a new unique ID, and adds the new user object to the users array. It then sends a JSON response with the new user object and a 201 Created status code, a standard practice for successful creation operations.

4.3 Read (GET): Fetching All Users and a Single User

Reading data is handled with GET requests. There are two common use cases: fetching all resources and fetching a single resource.
To fetch all users, a GET request is made to the /users endpoint.

```
app.get('/users', (req, res) => {
  res.json(users);
});
```

To fetch a single user, a dynamic route parameter is used to identify the specific user, such as /users/:id. The :id is a placeholder that can be accessed from req.params.

```
app.get('/users/:id', (req, res) => {
  const userId = parseInt(req.params.id);
  const user = users.find(u => u.id === userId);

  if (user) {
    res.json(user);
  } else {
    res.status(404).json({ message: 'User not found' });
  }
});
```

This handler finds the user with the matching ID in the array. If the user is found, their data is returned in a JSON response; otherwise, a 404 Not Found error is sent.

4.4 Update (PUT): Modifying an Existing User

Updating an existing user requires a PUT request to a specific endpoint, such as /users/:id.

```
app.put('/users/:id', (req, res) => {
  const userId = parseInt(req.params.id);
  const userIndex = users.findIndex(u => u.id === userId);

  if (userIndex!== -1) {
    users[userIndex] = {...users[userIndex],...req.body };
    res.json(users[userIndex]);
  } else {
    res.status(404).json({ message: 'User not found' });
  }
});
```

The handler extracts the ID from the URL and the new data from the request body. It then uses findIndex() to locate the user's position in the array. If the user is found, their data is updated; otherwise, a 404 Not Found error is returned.

4.5 Delete (DELETE): Removing a User

The DELETE request is used to remove a user from the collection. The endpoint is similar to GET and PUT, using the user's ID to identify which record to remove.

```
app.delete('/users/:id', (req, res) => {
  const userId = parseInt(req.params.id);
  const userIndex = users.findIndex(u => u.id === userId);

  if (userIndex!== -1) {
    users.splice(userIndex, 1);
    res.status(204).send();
  } else {
    res.status(404).json({ message: 'User not found' });
  }
});
```

This code finds the user's index and uses Array.prototype.splice() to remove the single element at that index. A successful deletion typically returns a 204 No Content status code, indicating that the request was fulfilled but there is no new data to send back to the client.

4.6 Sending JSON Responses: res.json() vs. res.send()

Both res.send() and res.json() are used to send responses to the client, but they have a subtle difference that is important for building APIs. The res.send() method is a versatile, general-purpose function that automatically infers the Content-Type header based on the data being sent. For example, if a JavaScript object is passed to res.send(), it automatically sets the Content-Type to application/json.
In contrast, res.json() is specifically designed for sending JSON data. It ensures that the Content-Type header is explicitly set to application/json, and it automatically converts the JavaScript object into a JSON-formatted string. For all API responses, it is recommended to use res.json() to ensure predictable and correct behavior for the frontend client.

5. Connecting the React Frontend: Bridging the Gap

5.1 Frontend-to-Backend Communication: An Overview

The frontend, running in a web browser, cannot directly access the data stored in the backend server's in-memory array. The two are separate applications, and communication must occur via HTTP requests. The frontend sends a request (e.g., a GET request to fetch all users) to the

backend API endpoint, and the backend processes the request and sends a formatted response back, typically in JSON format.
There are two primary methods for making these requests from a React application: the native fetch API and a third-party library called axios. While fetch is a global function and requires no installation, it requires an extra step to manually parse the JSON response. Axios, a promise-based HTTP client, simplifies this process by automatically transforming JSON data, providing a cleaner API, and offering robust features like request cancellation and better error handling. For this project, axios is the recommended choice due to its ease of use for beginners. It can be installed by running npm install axios in the frontend project directory.

5.2 The CORS Issue and How to Fix It

A common hurdle for beginners in full-stack development is the CORS (Cross-Origin Resource Sharing) error. This is not a bug but a crucial security feature of modern web browsers known as the Same-Origin Policy. The policy prevents a web page from making requests to a different domain from the one that served the original page. Since the React development server runs on one port (e.g., localhost:5173) and the Express backend runs on another (e.g., localhost:3000), the browser views them as two different "origins" and blocks the request.
There are two primary solutions to this problem during development.
Solution 1 (The Backend Fix): The most robust and proper solution is to configure the backend server to explicitly allow requests from the frontend's origin. This is done using the cors middleware package for Express. It can be installed with npm install cors and then used in the Express application with app.use(cors()). This is the standard way to handle CORS in a production environment.
Solution 2 (The Frontend Development Fix): A simpler, development-specific workaround involves using React's built-in proxy feature. By adding a proxy key to the frontend's package.json file, the React development server is configured to forward API requests to the backend.

```
"proxy": "http://localhost:3000"
```

With this configuration, any request made to /api/users from the frontend will be automatically proxied to http://localhost:3000/api/users, bypassing the browser's CORS check entirely. This approach simplifies the local development environment by avoiding the security policy check and is a valuable pattern for a full-stack developer to understand, as it teaches the importance of thinking about both client-side and server-side solutions to a problem.

6. End-to-End Example Project: The Full User Manager

6.1 Backend Code: Consolidating the CRUD API

The following code represents the complete server.js file for the Express backend. It includes all the necessary imports, middleware, the in-memory user data, and the four CRUD routes for the user manager application.

```
// server.js
const express = require('express');
const cors = require('cors');
const app = express();
const port = 3000;

// Middleware to enable CORS and parse JSON data
app.use(cors());
app.use(express.json());

// In-memory data store
let users =;

// --- CRUD Routes ---

// GET: Read all users
app.get('/api/users', (req, res) => {
  res.json(users);
});

// GET: Read a single user by ID
app.get('/api/users/:id', (req, res) => {
  const userId = parseInt(req.params.id);
  const user = users.find(u => u.id === userId);
  if (user) {
    res.json(user);
  } else {
    res.status(404).json({ message: 'User not found' });
  }
});

// POST: Create a new user
app.post('/api/users', (req, res) => {
  const newUser = {
    id: users.length? Math.max(...users.map(u => u.id)) + 1 : 1,
    name: req.body.name,
    email: req.body.email
  };
  users.push(newUser);
  res.status(201).json(newUser);
});

// PUT: Update an existing user
app.put('/api/users/:id', (req, res) => {
  const userId = parseInt(req.params.id);
  const userIndex = users.findIndex(u => u.id === userId);
  if (userIndex!== -1) {
    users[userIndex] = {...users[userIndex],...req.body };
    res.json(users[userIndex]);
  } else {
    res.status(404).json({ message: 'User not found' });
  }
});

// DELETE: Remove a user
app.delete('/api/users/:id', (req, res) => {
  const userId = parseInt(req.params.id);
  const userIndex = users.findIndex(u => u.id === userId);
  if (userIndex!== -1) {
    users.splice(userIndex, 1);
    res.status(204).send();
  } else {
    res.status(404).json({ message: 'User not found' });
  }
});

// Start the server
app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

6.2 Frontend Code: Building the User Interface

The React frontend will be built with a simple component structure to manage the user list. The main App.js component will handle the state for the user data and all the axios calls to the backend. It will use React's useState and useEffect hooks to manage the component's state and trigger data fetching when the component mounts.

```
// src/App.js
import React, { useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [users, setUsers] = useState();
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [editingUserId, setEditingUserId] = useState(null);

  // Read: Fetch all users from the backend
  useEffect(() => {
    fetchUsers();
  },);

  const fetchUsers = async () => {
    const response = await
axios.get('http://localhost:3000/api/users');
    setUsers(response.data);
  };
  // Create: Add a new user
  const handleCreateUser = async (event) => {
    event.preventDefault();
    await axios.post('http://localhost:3000/api/users', { name, email
});
    setName('');
    setEmail('');
    fetchUsers(); // Refresh the list
  };

  // Update: Handle edit and save logic
  const handleUpdateUser = async (id) => {
    await axios.put(`http://localhost:3000/api/users/${id}`, { name,
email });
    setEditingUserId(null);
    setName('');
    setEmail('');
    fetchUsers(); // Refresh the list
  };

  // Delete: Remove a user
  const handleDeleteUser = async (id) => {
    await axios.delete(`http://localhost:3000/api/users/${id}`);
    fetchUsers(); // Refresh the list
  };

  return (
    <div style={{ padding: '20px' }}>
      <h1>Simple User Manager</h1>
      {/* User Form for Create/Update */}
      <form onSubmit={editingUserId? () =>
handleUpdateUser(editingUserId) : handleCreateUser}>
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
          placeholder="Name"
          required
        />
        <input
          type="email"
          value={email}
          onChange={(e) => setEmail(e.target.value)}
          placeholder="Email"
          required
        />
        <button type="submit">{editingUserId? 'Update User' : 'Add
User'}</button>
```

```
      </form>

      {/* User List */}
      <ul style={{ listStyle: 'none', padding: 0 }}>
        {users.map(user => (
          <li key={user.id} style={{ margin: '10px 0', border: '1px
solid #ccc', padding: '10px' }}>
            {user.name} - {user.email}
            <button
              onClick={() => {
                setEditingUserId(user.id);
                setName(user.name);
                setEmail(user.email);
              }}
              style={{ marginLeft: '10px' }}
            >
              Edit
            </button>
            <button
              onClick={() => handleDeleteUser(user.id)}
              style={{ marginLeft: '10px' }}
            >
              Delete
            </button>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

This code uses a single component to manage the entire application logic, which is a common pattern for small projects. A more scalable approach would be to separate the form and list into their own components, but this single-file example is perfect for a beginner to grasp the core functionality.

7. Common Errors and Debugging: Solving the Hurdles

7.1 CORS Errors

As discussed earlier, CORS errors are a frequent point of frustration for beginners. The browser console will typically display an error message like "Access-Control-Allow-Origin" is missing. This is a clear indicator that the browser's Same-Origin Policy is blocking the request from the

React app (on one port) to the Express server (on another). The solution is to use either the backend cors middleware or the frontend proxy configuration as detailed in Section 5.2.

7.2 Port Conflicts

Another common error is a port conflict, where the terminal displays an error message such as "Address already in use". This means that the port the Express server is trying to use (e.g., 3000) is already being used by another process, such as a previously running instance of the same server. The solution is to either kill the process currently using the port or change the port number in the Express application's app.listen() method.

7.3 Unhandled Promise Rejections and Async/Await Errors

When dealing with asynchronous code, such as API calls, it is critical to handle potential errors gracefully. A beginner's first instinct might be to ignore errors, but this can lead to the entire application crashing if the backend server is down or a request fails. The standard practice for handling errors in async/await functions is to wrap the code in a try...catch block.

```
try {
  const response = await axios.get('/api/users');
  setUsers(response.data);
} catch (error) {
  console.error('Error fetching data:', error);
}
```

This simple pattern teaches the importance of writing defensively and ensures that the application doesn't crash, a hallmark of robust software development. It demonstrates that the software should be able to handle unexpected failures without disrupting the user experience.

7.4 Incorrect JSON Data

A frequent issue encountered during POST or PUT requests is an undefined req.body. This typically occurs because the middleware app.use(express.json()) has not been added to the Express application. Without this middleware, the server does not know how to parse the incoming JSON data from the request, and the req.body property remains empty. The fix is to ensure the middleware is properly loaded at the beginning of the server file.

8. Best Practices for Your First Full-Stack App

8.1 Project Structure and Modularity

While a single server.js file is acceptable for a small project, a clean folder structure is crucial for readability and scalability. The most common approach is to separate the frontend and backend into their own root-level folders. Within the backend folder, the logic can be further modularized by creating separate directories for routes, controllers, and utility functions. This separation of concerns—keeping routing, business logic, and data handling in distinct files—makes the code easier to read, debug, and maintain as the application grows.

8.2 Using Environment Variables for Configuration

It is a core best practice to avoid hardcoding sensitive information, such as API keys, database credentials, or secret keys, directly into the code. A common solution is to use environment variables, which are stored in a .env file and are not checked into version control. Libraries like dotenv are used to load these variables into the application's process. This approach keeps sensitive data separate from the source code, making the application more secure and portable across different environments (development, staging, production).

8.3 The Importance of HTTP Status Codes

HTTP status codes are not just numbers; they are a standardized way for the server to communicate the outcome of a request to the client. A 200 OK code signifies a successful request, while a 201 Created is used specifically for a successful POST request. A 404 Not Found indicates that the requested resource could not be found, and a 500 Internal Server Error points to a problem on the server side. By using the correct status codes, the server provides clear and meaningful feedback to the frontend, which can then handle different outcomes gracefully.

9. Conclusion: What's Next?

This handbook has guided the reader through the creation of a complete, foundational full-stack application using Express.js and React.js. The core concepts of full-stack architecture, CRUD operations, backend API design, and frontend-backend communication have been introduced and implemented through a practical example. By successfully building this project, the reader has gained the fundamental skills needed to develop robust web applications.
The most logical and powerful next step is to replace the in-memory array with a real database. This will introduce data persistence, meaning the user data will no longer be erased every time the server restarts. A recommended choice for a beginner is MongoDB, a document-oriented NoSQL database that works well with JavaScript and the JSON data format. To simplify the interaction with MongoDB, the Node.js library Mongoose is an excellent tool.
By transitioning to a real database, the reader can see how the core CRUD logic they have already mastered with Express.js remains fundamentally the same, regardless of the data storage technology. This demonstrates that the skills and patterns learned in this guide are directly transferable and scalable, providing a solid foundation for a career in full-stack development.

Works cited

1. What's the Difference Between Frontend and Backend in Application Development? - AWS, https://aws.amazon.com/compare/the-difference-between-frontend-and-backend/ 2. Frontend vs Backend Development - GeeksforGeeks, https://www.geeksforgeeks.org/blogs/frontend-vs-backend/ 3. www.sumologic.com, https://www.sumologic.com/glossary/crud#:~:text=CRUD%20is%20an%20acronym%20from,%2C%20read%2C%20update%20and%20delete. 4. What Is CRUD? Create, Read, Update, and Delete | CrowdStrike, https://www.crowdstrike.com/en-us/cybersecurity-101/observability/crud/ 5. Todo List Application using MERN - GeeksforGeeks,

https://www.geeksforgeeks.org/mern/todo-list-application-using-mern/ 6. mern-todo-app · GitHub Topics, https://github.com/topics/mern-todo-app 7. The Express + Node.js Handbook – Learn the Express JavaScript Framework for Beginners, https://www.freecodecamp.org/news/the-express-handbook/ 8. Express - Node.js web application framework, https://expressjs.com/ 9. Express.js Tutorial - GeeksforGeeks, https://www.geeksforgeeks.org/node-js/express-js/ 10. Project structure for an Express REST API when there is no "standard way" - Corey Cleary, https://www.coreycleary.me/project-structure-for-an-express-rest-api-when-there-is-no-standard-way 11. Set up your Node.js and Express development environment | Twilio, https://www.twilio.com/docs/usage/tutorials/how-to-set-up-your-node-js-and-express-development-environment 12. express - npm, https://www.npmjs.com/package/express 13. Express/Node introduction - Learn web development - MDN, https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/Express_Nodejs/Introduction 14. REST-API CRUD Operations Using Express.js and In-Memory Database | by Ravi Patel, https://medium.com/@ravipatel.it/rest-api-crud-operations-using-express-js-and-in-memory-database-0fb0989a708f 15. Setup Node.js Express Project: A Beginner's Guide - Daily.dev, https://daily.dev/blog/setup-nodejs-express-project-a-beginners-guide 16. nodemon - npm, https://www.npmjs.com/package/nodemon 17. How to setup ReactJs with Vite ? - GeeksforGeeks, https://www.geeksforgeeks.org/reactjs/how-to-setup-reactjs-with-vite/ 18. Express.js Folder Structure Best Practices for Clean Code - DhiWise, https://www.dhiwise.com/post/express-js-folder-structure-best-practices-for-clean-code 19. Build a React app from Scratch, https://react.dev/learn/build-a-react-app-from-scratch 20. Routing - Express.js, https://expressjs.com/en/guide/routing.html 21. Express Tutorial Part 4: Routes and controllers - Learn web development - MDN, https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/Express_Nodejs/routes 22. How to use (GET/POST/PUT/DELETE) http request in express js | by Deepak Tailor, https://medium.com/@deepaktailor2305/how-to-use-get-post-put-delete-http-request-in-express-js-2c6743c23e89 23. Introduction to Building a CRUD API with Node.js and Express - Harness, https://www.harness.io/blog/introduction-to-building-a-crud-api-with-node-js-and-express 24. How to Create a CRUD API – NodeJS and Express Project for Beginners - freeCodeCamp, https://www.freecodecamp.org/news/create-crud-api-project/ 25. Writing middleware for use in Express apps - Express.js, https://expressjs.com/en/guide/writing-middleware.html 26. Understanding Middleware in Express.js: A Comprehensive Guide | by Aryan kumar, https://medium.com/@finnkumar6/understanding-middleware-in-express-js-a-comprehensive-guide-5b13d72427fa 27. Build a CRUD App With Only JSON Files Using a Node.js API - Adevait, https://adevait.com/nodejs/build-a-crud-app-with-only-json-files 28. Express: send() vs json() vs end() - GitHub, https://gist.github.com/acidtone/df91c6276e69ae3726e3f8b39223ceec 29. Difference between res.send() and res.json() in Express.js? - GeeksforGeeks, https://www.geeksforgeeks.org/node-js/difference-between-res-send-and-res-json-in-express-js/ 30. React project with Express back-end : r/reactjs - Reddit, https://www.reddit.com/r/reactjs/comments/x2i3x5/react_project_with_express_backend/ 31. Using the Fetch API - Web APIs | MDN - Mozilla, https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch 32. Getting Started | Axios Docs, https://axios-http.com/docs/intro 33. Axios in React: A Guide for Beginners - GeeksforGeeks, https://www.geeksforgeeks.org/reactjs/axios-in-react-a-guide-for-beginners/ 34. Connecting React Frontend to Express Backend | by Akshay Bendadi | Medium,

https://medium.com/@akshaybendadi/connecting-react-frontend-to-express-backend-ee38a339
3f83 35. Demystifying CORS for React Developers - StackHawk,
https://www.stackhawk.com/blog/react-cors-guide-what-it-is-and-how-to-enable-it/ 36. CORS
Error in testing my React app : r/reactjs - Reddit,
https://www.reddit.com/r/reactjs/comments/1jbgo4q/cors_error_in_testing_my_react_app/ 37.
React CRUD Operation with ExpressJS, MySQL , Cors, Nodemon and Axios - GitHub,
https://github.com/VidhuraNeethika/React-CRUD-Operation 38. Error handling - Express.js,
https://expressjs.com/en/guide/error-handling.html 39. Async/Await Error Handling - Wes Bos,
https://wesbos.com/javascript/12-advanced-flow-control/71-async-await-error-handling