Hyperiondev

Task

React – Back-End Integration

Visit our website

Introduction

Welcome to the React – Back-End Integration task! Having covered the fundamentals of back-end development – creating web applications with Express, handling HTTP requests and responses, and interacting with MongoDB using Mongoose – we now focus on a key aspect of modern web development: integrating the front end with the back end. In this task, you will learn how to connect a **React.js** front end with an **Express.js** back end.

REACT AND BACK-END INTEGRATION

Connecting the React front end with the back end requires communication between the two using HTTP requests. You will need to set up the front and back end using their required dependencies.

CREATE AN EXPRESS.JS BACK END

Open up your command-line terminal/interface and create a project directory using the `mkdir` command:

```
mkdir <project name>
```

Navigate into the project directory using the `cd` command and create the **backend** directory:

```
mkdir backend
```

Change directory to the **backend** directory:

```
cd backend
```

Run the following command to initialise the **package.json** file so that dependencies can be installed:

```
npm init -y
```

You will need `express` for creating the server and `cors` for cross-origin resource sharing. It's a process that allows the user to access restricted resources on the web. You can install both `express` and `cors` using the command given below:

```
npm install express cors
```

Lastly, create a file called **index.js** inside your **backend** directory.

Take note:

CORS is an abbreviation for **cross-origin resource sharing**. It's a security measure that restricts unwanted domain access by setting rules for handling cross-domain requests. When a request is made, the server receives an origin header containing the request's origin.

You can now write the following code inside your **index.js** file in your **backend** folder. This will create a simple server that you can adjust according to your requirements. The port number used in this server is **5000**, which can be changed according to your browser. However, it is not advisable to use a port number on which your React app will be running, such as port **3000** or **5173**, as it can cause your back end to crash.

```
// Importing the required modules
const express = require("express");
const cors = require("cors");

// Create an instance of the express server
const app = express();

// Define the port number for the server
const PORT = process.env.PORT || 5000;

// Enable Cross-Origin Resource Sharing
app.use(cors());

// Define the route to retrieve the message
app.get("/api/data", (req, res) => {
  const data = { message: "Hello from the back end!" };
  res.json(data); // Send data as a response
});

// Displaying the requested URL for each request
app.use((req, res, next) => {
  console.log(`Request URL: ${req.url}`);
  next();
});

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

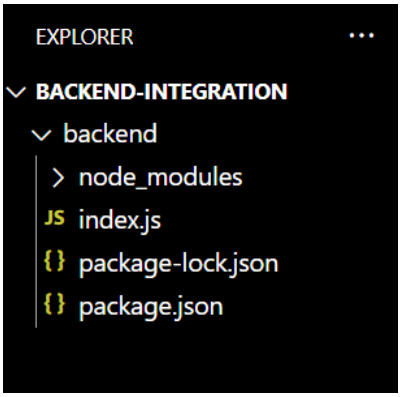
- **Lines 2 & 3:** import the `express` and `cors` modules.
- **Line 6:** initialise an instance of the `express` server.
- **Line 9:** define the port number of the server. If the port number is not stored inside of an **.env** file, by default the port number will be 5000.
- **Line 12:** enable the server to use `Cross-Origin Resource Sharing`.
- **Line 15:** define the route to retrieve the message.
- **Line 21**: display the requested URL for each request.
- **Line 27**: start the server and display a message on the console.

To check if your server is working, you can type in the following command in your terminal:

```
node index.js
```

If it shows a message similar to "`Server is running on port 5000`", then that means your server's connection was established successfully.

Now that the server side has been set up, you can move on to creating the front end of your project. Your project directory in VS Code will look something like this:



CREATE A REACT.JS FRONT END

Open up a new terminal from your project's root directory, then create a new React application named `frontend` using the **React + Vite Starter Kit** by following the instructions found **here**.

Alternatively, you may refer to the **Additional Reading – Installation, Sharing, and Collaboration** guide to complete your React installation. You'll want to reference the "Setting up a React app with Vite" section in the Additional Reading.

Once your React app installation is complete, we are going to create a proxy server. In simple terms, the `server.proxy` option in Vite is a way to redirect or forward certain network requests made by your app during development. This is especially

helpful when you're working with APIs that are hosted on different servers and you want to avoid problems like CORS issues.

What does a proxy do?

A **proxy** acts as a "middleman" between a user and the Internet, forwarding your requests from one server to another. So, instead of your app directly making a request to another server (which might cause security issues), the proxy forwards it for you.

How to use a proxy server

In your React project's root directory (where **package.json** is located), find a file called **vite.config.js**. Inside of **vite.config.js**, add the proxy configuration like this:

```
import { defineConfig } from "vite";
import react from "@vitejs/plugin-react";

// Export the configuration using the defineConfig helper provided by Vite
export default defineConfig({
  // Specify the Vite plugins to use, in this case, the React plugin
  plugins: [react()],

  // Configure the development server settings
  server: {
    // Set up a proxy to handle API requests during development
    proxy: {
      // All requests starting with /api will be forwarded to the target URL
      "/api": {
        // The target server that will handle the requests
        target: "http://localhost:5000",

        // This option changes the origin of the host header to the target URL
        changeOrigin: true
      },
    },
  },
});
```

Breakdown of key parts:

- `plugins: [react()]`: This tells Vite to use the React plugin, which allows you to use React in your project.
- `server.proxy`: The proxy option is used to configure URL redirection (proxying) during development to avoid issues like CORS.

- `/api`: This is the URL prefix that Vite looks for. Any requests starting with /api (e.g., `/api/posts`) will be proxied.
- `target: “http://localhost:5000”`: The target is the back-end server that was previously created.
- `changeOrigin: true`: This option changes the origin of the request to match the target’s origin. It’s necessary when making cross-origin requests.

Now install the `Axios` library for making HTTP requests from your React app. Axios helps with API fetching and facilitates communication with the back end.

Open your terminal in the **frontend** directory and run the following command:

```
npm install axios
```

Now open your **src/App.jsx** file in your project directory and update the following code as starter code, which you can alter according to your project requirements. Notice that the following code is written by removing the pre-written code in the **App.jsx** file.

```
import { useState, useEffect } from 'react';
import axios from 'axios';
import './App.css';

function App() {
  const [data, setData] = useState({}); // State to store fetched data

  useEffect(() => {
    fetchData(); // Fetch data each time the component loads
  }, []);

  // Function to fetch data from the server
  const fetchData = async () => {
    try {
      /* Sends a GET request to
      'http://localhost:5000/api/data' (backend server) */
      const response = await axios.get('/api/data');
      setData(response.data); // Update state with fetched data
    } catch (error) {
      console.error('Error fetching data:', error);
    }
  };

  return (
    <div className="App">
      <header className="App-header">
        {/* Display the message, or 'Loading...' if data is not yet fetched*/}
        <h1>{data.message || 'Loading...'}</h1>
      </header>
```

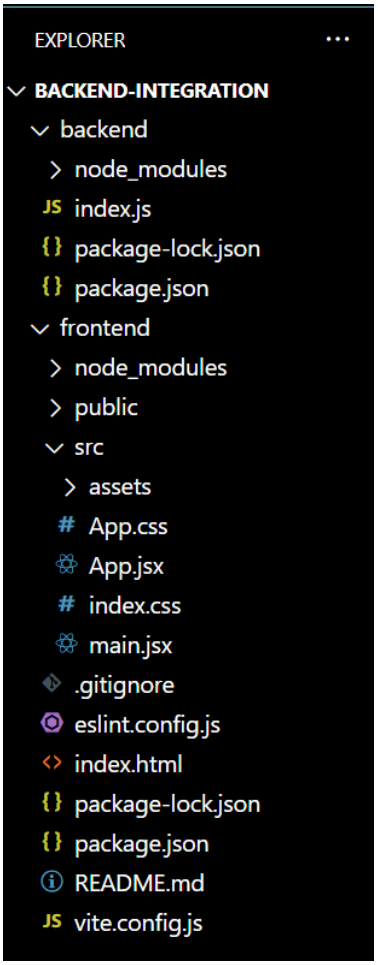
```
    </div>
  );
}
export default App;
```

You can start working on your front end by running the command in the **frontend** directory of your terminal:

```
npm run dev
```

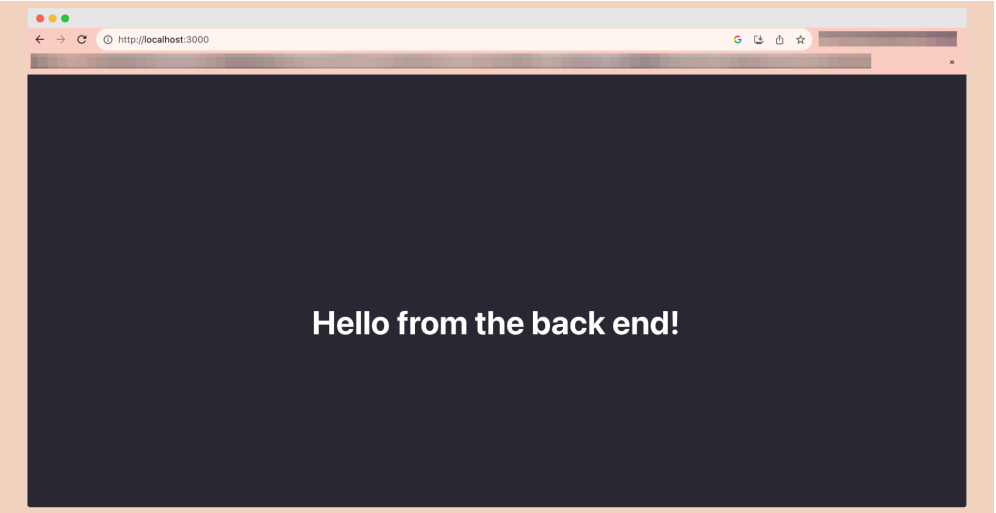
Your React app will be accessible at a local port site like `http://localhost:5173/`.

Your directory will look something like this at the end:

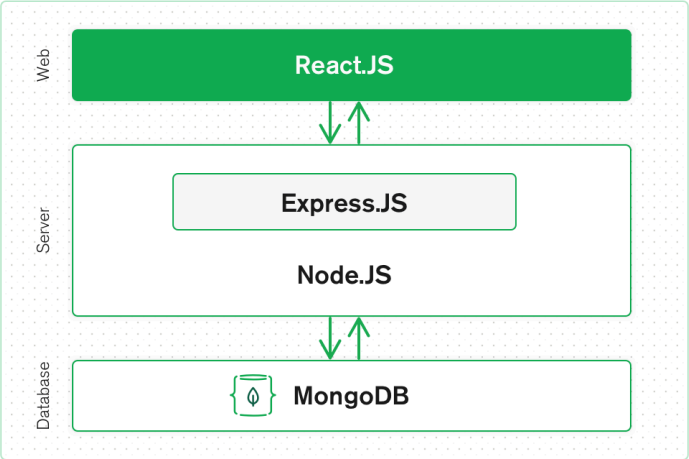


CONNECTING THE FRONT END WITH THE BACK END

After setting up your front and back ends, run both servers in separate terminals. When you run your React app and access it through your browser, it will request the **express.js** back end's `/api/data` (this is a generic example) endpoint and display the response. You will see the following output:



The diagram below illustrates how a React front end communicates with a **Node.js** Express back end and a MongoDB database. React components handle the UI, routing, and make HTTP requests to the back end. Express routes these requests to controllers, which process them and interact with the MongoDB database. The back end then sends a response back to React, completing the communication.



Interaction between Express, React, and MongoDB (MongoDB, n.d.)

In conclusion, full-stack web development requires seamless integration of the front end in **React.js** and the back end in **Express.js**. This task illustrates the process of integrating a React front end with a **Node.js** Express back end, providing step-by-step instructions for creating both components and establishing communication between them.

Practical task

In the same **index.js** file that is part of the back end for the example app that you've been working on, create a new API endpoint that will be responsible for displaying a custom message.

Step 1: Add a back-end endpoint.

- Go to the back-end folder.
- Edit `index.js`.
- Add a new endpoint: `app.get('/api/message', ...)`.
- Define a custom message in the endpoint's response.
- Save and start the back-end server.

Step 2: Fetch the message from the front end.

- Go to the front-end folder.
- Open `src/App.jsx`.
- Import the necessary libraries.
- Add a new state: `const [customMessage, setCustomMessage] = useState('');`.
- Use `useEffect()` to fetch a message from "`/api/message`".
- Update `customMessage` with the fetched message.
- Display `customMessage` in your app.

Step 3: Run servers and check.

- Run both back-end and front-end servers.
- Open your app in a web browser.
- You should see the original and custom messages displayed.

To submit the task for review:

- Include a screenshot of your app showing both messages.
- Compress your application folder and add it to the relevant task folder. Remember to delete the **node_modules** folder before submitting.

Be sure to place files for submission inside your task folder and click “Request review” on your dashboard.

Rate us

Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we’ve done a good job?

Click here to share your thoughts anonymously.

Reference list

MongoDB. (n.d.). *MERN stack explained*.
https://www.mongodb.com/resources/languages/mern-stack