**Vue.js Introduction**

**Introduction**

Code space provides an environment for developing codes with full customization options. The development environment for the project can be accessed at [bug-free-lamp-grrv744prvrh776.github.dev](https://bug-free-lamp-grrv744prvrh776.github.dev/).

**Vue.js for Efficient Web Development**

**Why Vue.js?**

Vue.js makes the task of creating performant websites easier and faster. It facilitates the creation of modular and reusable components that can be rearranged to form a fully functional website.

**Reusability Advantage**

When applying a sign-in feature to all pages, Vue.js stands out. While other platforms may require applying it individually to 100 pages, Vue.js allows the creation of a reusable page applied to all, requiring changes only once.

**Vue vs. React**

Vue is designed for incremental addition to existing sites, offering flexibility. To create a new Vue project, use the command: npm install -g @vue/cli; install vue command line for you

vue create "front-end" create the front-end of your application

**Project Setup**

1. Choose Vue version and package manager.
2. Run the command 'npm run serve' to launch your application.

**Understanding the Structure**

* The application runs within the div in index.html.
* Configuration files in the root directory include settings for Babel and Vue.
* package.json and package.lock.json provide project information.
* src contains all view components.
* The top-level component is App.vue, rendering other components directly or indirectly.
* main.js is the entry point of the application.

**Vue Router Setup**

1. Install Vue Router using: npm install vue-router@4
2. Navigate to the desired directory and run the command.
3. Implement routing logic for displaying random pages.

Certainly! Here's a revised version of your note with some clarifications:

**Setting Up Vue Router and Best Practices:**

1. **Installation:**
   * Use **npm install vue-router@4** to install Vue Router for your application.
2. **Styling Considerations:**
   * Ensure you add **export default** to each component. This is crucial for proper style recognition and prevents potential errors.
3. **Router Configuration:**
   * Create a router with **createWebHistory** to manage the browser history and define the base URL for returning to the main page.
4. **Defining Routes:**
   * Define routes within the **routes** array, specifying each route with curly braces **{}**. Include **path** and **component** for each route.
5. **Linking Components:**
   * Utilize the **<router-link>** tag for linking components. This tag simplifies navigation between different views.
6. **App.vue Structure:**
   * Always include the following structure in your **App.vue**:
7. <div id="app">
8. <router-view></router-view>
9. </div>
10. **Figma for Design:**
    * Consider using Figma as a design tool to visualize and plan the structure of your website.

By following these steps, you have successfully set up the Vue Router in your application, configured history mode, and added the necessary template tag to display the routed components.

### Adding Files to Workspace:

To include files in your workspace, follow these steps:

1. Right-click on the file.
2. Select the "Upload" option.

### Writing a For Loop in Vue.js:

When utilizing a for loop in Vue.js, consider the following steps:

#### **Template Iteration:**

Use the **v-for** directive in your template to iterate over an array. For instance:

htmlCopy code

<div v-for="product in products" :key="product.id"> <!-- Your content here --> </div>

#### **Binding Data:**

Ensure that the data you want to iterate over is declared in the **data** section of your component:

javascriptCopy code

data() { return { products: [] // Initialize products array with your data }; }

#### **Referencing Values:**

When referencing values from an object, use the colon (**:**) before the attribute, such as:

htmlCopy code

<img :src="product.imageName">

#### **Displaying Object Values:**

To display object values in the template, enclose them within double curly braces **{{ }}**, like so:

htmlCopy code

<h3>{{ product.name }}</h3>

#### **Key Attribute for Better Rendering:**

To optimize component re-rendering, include a unique **key** attribute after the **v-for** directive, like this:

htmlCopy code

<div v-for="product in products" :key="product.id"> <!-- Your content here --> </div>

### Styling Considerations:

Ensure that the styles in **App.vue** are removed to prevent unintended application to your Vue page.

By organizing your Vue component in this manner, you establish a clear structure for iterating over data and presenting it in your application.

### Vue.js: Accessing Single Component Using URL Parameters

#### **Header:**

In Vue.js, you can access and display a single component based on URL parameters. This is particularly useful when you want to show a specific item, such as a single article from a list of articles, using the value provided in the URL parameter. Here's a step-by-step guide on how to achieve this:

#### **Step 1: Define Route with Parameter**

Set up a route in your Vue router configuration with a parameter placeholder, such as **// Example route configuration**

**{**

**path: '/products/:productId',**

**component: ProductDetail, // Component to display single product**

**name: 'productDetail',**

**}Step 2: Access URL Parameter in Component**

Within the component (**ProductDetail** in this example), access the parameter using **this.$route.params.productId**. This allows you to retrieve the specific value from the URL:

// ProductDetail.vue

export default {

data() {

return {

product: null,

};

},

created() {

// Fetch data for the specific product using the productId from the URL

this.product = this.fetchProduct(this.$route.params.productId);

},

methods: {

fetchProduct(productId) {

// Use productId to retrieve the specific product from your data source

// Example: Replace this with your actual data fetching logic

return products.find(product => product.id === parseInt(productId));

},

},

};In the template of your **ProductDetail** component, use the **product** data property to render the details of the single product:

<!-- ProductDetail.vue template -->

<template>

<div v-if="product">

<h1>{{ product.name }}</h1>

<!-- Display other product details as needed -->

</div>

<div v-else>

<p>Loading...</p>

</div>

</template>

By following these steps, you create a dynamic route that allows you to display a single component based on the value provided in the URL parameter. The **fetchProduct** method can be customized to suit your data-fetching needs, such as fetching data from an API or database.

### ****Create NavBar Component:****

Create a new file named **NavBar.vue**.

### 2. ****Structure the Component:****

Inside **NavBar.vue**, structure the component with template and script tags.

### <!-- NavBar.vue -->

### <template>

### <!-- Your template structure goes here -->

### </template>

### <script>

### export default {

### // Your component logic goes here

### }

### </script>

### 3. ****Import Images:****

To use images in your application, import them inside the script. Use **@/** to refer to the source directory.

### import logo from '@/assets/logo.png'; // Adjust the path accordingly

### 4. ****Data Handling:****

Return all your data inside the **data()** function within the **export default** block.

### // Inside the script tag of NavBar.vue

### export default {

### data() {

### return {

### // Your data properties go here

### };

### },

### // Other component methods and lifecycle hooks

### }

### 5. ****Conditional Rendering:****

Use **v-if** for conditional rendering, for example, to handle scenarios like an empty array.

### <!-- Inside your template in NavBar.vue -->

### <div v-if="yourArray && yourArray.length > 0">

### <!-- Render content when the array is not empty -->

### </div>

### <div v-else>

### <!-- Handle the case when the array is empty -->

### </div>

### 6. ****Router Link:****

To enable navigation to other pages, use the **router-link** tag with the **to** attribute.

<!-- Inside your template in NavBar.vue -->

<router-link to="/your-page">Click me</router-link>

Make sure to set up your routes in the main application file or the router file.

### 7. ****Click Event Handling:****

Handle the click event to perform actions when the link is clicked.

<!-- Inside your template in NavBar.vue -->

<router-link @click="handleClick" to="/your-page">Click me</router-link>

// Inside the script tag of NavBar.vue

export default {

methods: {

handleClick() {

// Your click event handling logic goes here

},

},

}

This structure should help you organize and implement the necessary features for your navigation bar in a Vue.js application. Adjust the details according to your specific use case and project structure.

**404 Page:**

To create a 404 page in Vue.js, you can follow these steps:

1. Create a new component for the 404 page. Let's call it **NotFoundPage**.
2. In your router configuration (**main.js**), add a new path that matches any route and point it to the **NotFoundPage** component:

{

path: '/:pathMatch(.\*)\*',

component: NotFoundPage

}

This configuration ensures that if no other route matches, the **NotFoundPage** component will be displayed.

**Modular Components:**

Modular components in Vue.js allow you to create reusable components. If you have a component that is used multiple times, you can create a modular component.

1. Create a modular component, e.g., **ProductDetailPage.vue**, that represents the structure and functionality of a product detail page.
2. Use this component wherever you need a product detail page in your application.

**Props in Vue.js:**

Props in Vue.js are a way to pass data from a parent component to a child component. Here's how you can use props:

1. In your child component (**ProductList.vue**, for example), define the props you want to receive:
2. export default {
3. props: ['products'],
4. // Other component options...
5. }
6. In the parent component where you use **ProductList**, pass the **products** property:

<ProductList :products="products" />

Make sure that the property names match exactly. In this case, you're passing the **products** property from the parent component to the **ProductList** component.

In Vue.js, the **router-link** is used for navigating between pages in a Vue.js application. It is a component provided by the Vue Router, which is the official router for Vue.js. Here's an example of how you can use **router-link** in a Vue component:

Assuming you have a route named **'product'** defined in your router configuration:

<template>

<div>

<h1>Product Page</h1>

<router-link :to="{ name: 'product', params: { id: 1 }}">Product 1</router-link>

<router-link :to="{ name: 'product', params: { id: 2 }}">Product 2</router-link>

<!-- You can also use a string path directly -->

<router-link to="/products">All Products</router-link>

<!-- This will render the component matched by the route -->

<router-view></router-view>

</div>

</template>

<script>

export default {

name: 'ProductPage'

}

</script>

**Backend**

## Setting up a Node.js Backend with Express for a Survey Application

### 1. Initialize Node.js Project

Use the following commands to set up your Node.js project:

npm init -y

npm install express

This creates a **package.json** file and installs the Express library.

### 2. Create Project Structure

Organize your project by creating a **backend** directory and a **src** directory inside it. Inside the **src** directory, create a file named **server.js** where your backend logic will reside.

### mkdir backend

### cd backend

### mkdir src

### touch src/server.js

### 3. Install Babel for Modern JavaScript

To have access to the latest JavaScript syntax, use Babel. Install the necessary Babel packages:

### npm install --save-dev @babel/core @babel/node @babel/preset-env @babel/cli @babel/plugin-transform-runtime

### npm install @babel/runtime

### 4. Configure Babel

Create a **.babelrc** file in your backend directory with the following configuration:

npm install --save-dev @babel/core @babel/node @babel/preset-env @babel/cli @babel/plugin-transform-runtime

npm install @babel/runtime

This configuration enables Babel to transpile your code using the preset for the latest ECMAScript features and the runtime plugin for efficient code transformation.

### 5. Build Your Backend Logic

Write your backend logic in the **server.js** file inside the **src** directory. Utilize Express for building the API endpoints.

This script uses Babel to run your server code, ensuring compatibility with the latest JavaScript features.

### 6. create babelrc in src file

1. Create a **.babelrc** file in your project's root directory.
2. Inside the **.babelrc** file, add the following configuration:

{

"presets": ["@babel/preset-env"],

"plugins": [

"@babel/plugin-transform-runtime"

// Add other plugins if needed

]

}

Simple express code for test backend

// File: server.js

// Import the Express framework

import express from 'express';

// Create an instance of the Express application

const app = express();

// Set up the server to listen on port 8000

app.listen(8000, () => {

console.log("Server is listening on port 8000");

});

// Define a simple example for the GET method in Express

app.get("/hello", (req, res) => {

// Send the response "hello!" when the "/hello" route is accessed

res.send("Hello!");

});

### Express Routes and Testing with Postman

#### **1. Making Endpoint Public in Postman:**

* To test the Codespace endpoint in Postman, make sure the endpoint is publicly accessible. Go to the specified port and set the visibility to public in Postman.

#### **2. Creating Routes in Express:**

* Import data into your **server.js** file.

const express = require('express');

const app = express();

const products = require('./data/products'); // Import your data your data

* Define routes using **app.get**:

#### // Example route to get all products

#### app.get('/api/products', (req, res) => {

#### res.json(products); // Send JSON data

#### });

#### **3. Handling Parameters:**

* To get the id from params, use **req.params.productId**:

// Example route to get a specific product by id

app.get('/api/products/:productId', (req, res) => {

const productId = req.params.productId;

const product = products.find(item => item.id === productId);

if (product) {

res.json(product);

} else {

res.status(404).json({ error: 'Product not found' });

}

});

* Use the **find()** method to locate the exact item based on the product id.

#### **4. Response Format:**

* It's a good practice to send JSON data using **res.json()** instead of the data itself:

japp.get('/api/products', (req, res) => {

res.json(products);

});

app.get('/api/products/:productId', (req, res) => {

// Find the product by id

const product = products.find(item => item.id === req.params.productId);

// Check if the product exists

if (product) {

res.json(product);

} else {

res.status(404).json({ error: 'Product not found' });

}

});

**Summary:**

* Import data into **server.js**.
* Define routes using **app.get**.
* Use **req.params** to get parameters from the URL.
* Utilize the **find()** method to locate specific items.
* Respond with JSON using **res.json()**.

### Setting Up Automatic Server Restarts with Nodemon

#### **Step 1: Install Nodemon**

In your backend folder, run the following command to install Nodemon as a development dependency:

#### npm install --save-dev nodemon

#### **Step 2: Add "dev" Script to package.json**

Open your **package.json** file in the backend folder and add a "dev" script inside the "scripts" section. This script uses Nodemon to watch for code changes and restart the server automatically.

#### {

#### "scripts": {

#### "dev": "nodemon --exec babel-node src/server.js"

#### },

#### "dependencies": {

#### // Your regular dependencies go here

#### },

#### "devDependencies": {

#### "nodemon": "^2.0.15"

#### }

#### }

#### **Step 3: Run Server in Development Mode**

Now, whenever you want to run your server in development mode, simply use the following command:

npm run dev

Nodemon will monitor your codebase, and whenever changes are detected, it will automatically restart your server, making the development process smoother.

Top of Form

### Creating User Management Endpoint with Express

#### **Step 1: Install Required Packages**

Before creating the endpoint, ensure you have Express and any other necessary packages installed:

npm install express

**Step 2: Setting Up Express Server**

import express from 'express';

import { cartItems as cartItemsRaw, products as productsRaw } from './temp-data';

let cartItems = cartItemsRaw;

let products = productsRaw;

// Endpoint to add a product to the user's cart

app.post('/cart', (req, res) => {

// Extract product ID from the request body

const productId = req.body.id;

// Find the product with the given ID

const product = products.find(product => product.id === productId);

// Add the product to the user's cart

cartItems.push(product);

// Respond with the updated cart items

res.json(cartItems);

});

#### **Explanation:**

1. **Setting Up Express Server:** The code initializes an Express server, sets up middleware to parse JSON in the request body, and defines various endpoints for managing users and products.
2. **Hello Message Endpoint:** The **/hello** endpoint returns a simple "Hello!" message.
3. **Products Endpoints:**
   * **/products**: Returns the list of products.
   * **/products/:productId**: Returns details of a specific product based on the provided product ID.
4. **User Cart Endpoints:**
   * **/cart**: Returns the current state of the user's cart.
   * **/cart** (POST): Adds a product to the user's cart based on the product ID provided in the request body.
5. **Server Listening:** The server is set to listen on port 8000. Adjust the port as needed.

### Providing Request Data in Postman

When using Postman to send requests, ensure that the request body is formatted as JSON. Here's a quick guide on how to include an "id" parameter:

1. Open Postman and navigate to the request you want to send.
2. In the request builder, go to the "Body" section.
3. Select the "raw" option.
4. Choose "JSON" from the drop-down menu.
5. Enter your JSON data, for example:

jsonCopy code

{ "id": 123 }

1. Send the request.

This ensures that your request contains a JSON body with the specified "id" parameter. The server, as configured, will then extract this "id" from the request body and process it accordingly.

Top of Form

**Important JavaScript Array Functions: filter, map, and More**

JavaScript provides powerful array functions that simplify common tasks like filtering, mapping, and traversing arrays. Two of these essential functions are filter and map.

1. filter Function:

Usage: filter creates a new array with elements that pass a specified test.

Example:

javascript

Copy code

const numbers = [1, 2, 3, 4, 5];

const evenNumbers = numbers.filter(num => num % 2 === 0);

// Result: evenNumbers = [2, 4]

2. map Function:

Usage: map creates a new array by applying a function to each element of the original array.

Example:

javascript

Copy code

const numbers = [1, 2, 3];

const squaredNumbers = numbers.map(num => num \* num);

// Result: squaredNumbers = [1, 4, 9]

3. forEach Function:

Usage: forEach executes a provided function once for each array element.

Example:

javascript

Copy code

const fruits = ['apple', 'banana', 'orange'];

fruits.forEach(fruit => console.log(fruit));

// Output: apple, banana, orange

4. reduce Function:

Usage: reduce reduces the array to a single value, applying a function for each element.

Example:

javascript

Copy code

const numbers = [1, 2, 3, 4];

const sum = numbers.reduce((acc, num) => acc + num, 0);

// Result: sum = 10

5. find Function:

Usage: find returns the first element in an array that satisfies a provided testing function.

Example:

javascript

Copy code

const users = [{ id: 1, name: 'Alice' }, { id: 2, name: 'Bob' }];

const user = users.find(u => u.id === 2);

// Result: user = { id: 2, name: 'Bob' }

6. some and every Functions:

Usage: some checks if at least one element satisfies a condition, while every checks if all elements satisfy a condition.

Example:

javascript

Copy code

const numbers = [1, 2, 3, 4, 5];

const hasEven = numbers.some(num => num % 2 === 0);

const allEven = numbers.every(num => num % 2 === 0);

// Result: hasEven = true, allEven = false

**Monogdb Atlas connection to your server.js file**

For connecting to database using mongsh you should first create your user and connection in your mongo atlas then you should add the path variable to your environemnt variable then you should install mongoose where you server.js file is npm install mongoose then you should write the following code for your connection

const mongoose = require("mongoose");

const dbUrl =

"mongodb+srv://<user>:<Password>@cluster0.tzzkwds.mongodb.net/?retryWrites=true&w=majority"; replace this with the username and password of your user in monogo atlas

then you should connect to your database using mongoose

mongoose

.connect(dbUrl, connectionParams)

.then(() => {

console.log("Connected to database");

})

.catch((error) => { // Corrected the syntax here: 'error;' to '(error) => { console.log(error); }'

console.log(error);

});

now run your server you should have this message "Connected to database"

Top of Form

# Connecting to MongoDB Atlas and Inserting Data

## Step 1 : Go to your database section press on shell copy the url and paste in your shell

## Step 2 : Insert your password that you created for user

## Step 3: Inserting Data into the Database

1. Switch to the desired database using the command:
2. use fsv-db
3. Copy and insert your data into the database. To insert a single document, use **insertOne**:
4. db.products.insertOne( /\* Paste your data here \*/ )
5. To insert multiple documents, use **insertMany**:

db.products.insertMany( /\* Paste all your data here \*/ )

all your data here \*/ )

1. Ensure that you close the parentheses and hit enter to execute the command.

## Step 4: Checking Inserted Data

To view information about the inserted products, use the following command:

db.products.find()

This command will display the details of the products in the specified collection.

### Connecting Node.js to MongoDB Atlas

**1. Install MongoDB Node.js Driver**

First, you need to install the MongoDB Node.js driver. You can do this using npm:

npm install mongodb

**2. Import MongoClient**

In your **server.js** file, import the **MongoClient** at the top of the file:

const { MongoClient } = require('mongodb');

**3. Create MongoClient Instance**

Before defining anything else in your server file, create an instance of **MongoClient**:

const client = new MongoClient();

**4. Connect to MongoDB Atlas**

* Go to MongoDB Atlas and press on "Connect."
* Choose "Connect to your application."
* Download the connection string (URL).

**5. Update MongoClient with Connection String**

Replace the empty parentheses in **MongoClient()** with the connection string you obtained from MongoDB Atlas:

const client = new MongoClient('your\_connection\_string\_here');

**6. Use .env for Sensitive Information**

For security, it's better to store sensitive information like passwords in a **.env** file. Create a **.env** file in your project's root and add your MongoDB Atlas password:

MONGODB\_PASSWORD=your\_password\_here

In your **server.js** file, use the **dotenv** package to access environment variables:

require('dotenv').config(); const client = new MongoClient({ connectionString: `mongodb+srv://<username>:${process.env.MONGODB\_PASSWORD}@<your-cluster-url>/test?retryWrites=true&w=majority`, });

Ensure to replace **<username>** and **<your-cluster-url>** with your actual MongoDB Atlas username and cluster URL.

**Server .js code to connect to database :**

const client = new MongoClient(url, { useNewUrlParser: true, useUnifiedTopology: true });

app.get('/products', async (req, res) => {

try {

await client.connect();

const db = client.db('fsv-db');

const products = await db.collection('products').find({}).toArray();

res.json(products);

} catch (error) {

console.error(error);

res.status(500).json({ error: 'Internal Server Error' });

} finally {

await client.close();

}

});

**Find By Id for req res in vue.js**

app.get('/user/:id', async (req, res) => {

const userId = req.params.id;

try {

// Connect to the MongoDB server

await client.connect();

// Access the database

const db = client.db('your\_database\_name');

// Access the 'users' collection

const usersCollection = db.collection('users');

// Use findOne to retrieve a user by a specific criteria (assuming 'id' field here)

const user = await usersCollection.findOne({ id: userId });

if (!user) {

// If user is not found, return a 404 status

res.status(404).json({ error: 'User not found' });

} else {

// If user is found, return the user data as JSON

res.json(user);

}

} catch (error) {

// Handle any errors that occurred during the database operations

console.error(error);

res.status(500).json({ error: 'Internal Server Error' });

} finally {

// Close the MongoDB connection

await client.close();

}

});

**Using Promise.all in JavaScript**

**Promise.all** is a powerful method in JavaScript that allows you to handle multiple promises concurrently. It takes an iterable (e.g., an array) of promises as input and returns a new promise. This new promise fulfills with an array of the resolved values of the input promises, in the same order as the input iterable. However, if any of the input promises is rejected, the new promise is rejected with the reason of the first rejected promise.

Here's an example of using **Promise.all**:

const promise1 = new Promise((resolve) => setTimeout(() => resolve('One'), 1000));

const promise2 = new Promise((resolve) => setTimeout(() => resolve('Two'), 2000));

const promise3 = new Promise((resolve, reject) => setTimeout(() => reject(new Error('Three')), 1500));

Promise.all([promise1, promise2, promise3])

.then((results) => {

console.log('All promises resolved:', results);

})

.catch((error) => {

console.error('At least one promise rejected:', error.message);

});

**Simplifying Database Connection in Node.js**

To make the database connection in Node.js less repetitive, you can organize your code as follows:

1. **Surround code with an async function:**

async function start() {

try {

// Database connection code here

// Your other functions or routes here

} catch (error) {

console.error('Error during initialization:', error);

} finally {

// Close database connection here if needed

}

}

// Call the async function to start your application

start();

**Move the database connection code to the beginning of the file:**

const database = require('./database'); // Import your database module

async function start() {

try {

// Database connection code

await database.connect();

// Your other functions or routes here

} catch (error) {

console.error('Error during initialization:', error);

} finally {

// Close database connection here if needed

await database.disconnect();

}

}

// Call the async function to start your application

start();

This structure allows you to have a cleaner and more organized codebase. It ensures that the database connection is established at the beginning and closed at the end, providing a better separation of concerns.

**Connecting Backend to Frontend with Axios in Vue.js**

1. **Install Axios in Vue Frontend:**
   * Open two terminals for backend and frontend.
   * In the frontend directory, install Axios:

npm install axios

1. **Import Axios in Vue Files:**
   * In your Vue files, import Axios at the top of the script section.
2. **Create Axios Request in created Lifecycle Hook:**
   * Inside the **export default** block, create a method named **created** using the **async** keyword:

export default {

data() {

return {

products: [] // Set default value to an empty array

};

},

created: async function () {

try {

const response = await axios.get('/api/products/');

this.products = response.data;

} catch (error) {

console.error('Error fetching data:', error);

}

}

}

:ror); } } }

1. **Handling CORS Issues with Proxy:**
   * To avoid CORS issues, set up a proxy in **vue.config.js**:

module.exports = {

configureWebpack: {

// ... your other configurations

},

devServer: {

proxy: 'http://localhost:8000'

}

};

1. **Update Backend Requests with /api/:**
   * Modify backend requests to include **/api/** at the beginning for easier communication:

// Before

app.get('/products', (req, res) => {

// ... your logic

});

// After

app.get('/api/products', (req, res) => {

// ... your logic

});

1. **Inspecting Requests:**
   * Open the browser's inspector window and navigate to the Network tab.
   * Observe the headers for each request to ensure proper communication.
2. **Note:**
   * Keep both backend and frontend terminals open, especially during installations.
   * Make sure the **products** variable in **data()** has a default value (empty array).

By following these steps, you establish a connection between the Vue.js frontend and the backend using Axios, handle CORS issues with a proxy, and inspect network requests for debugging purposes.

Top of Form

**Setting Up Image that works for backend too and Making API Requests with Express and Axios in Vue.js**

1. **Backend Setup:**
   * Ensure that your backend server is configured to serve static files. You can use Express middleware for this.

const express = require('express');

const path = require('path');

const app = express();

// Serve images from the 'assets' folder

app.use('/images', express.static(path.join(\_\_dirname, '../assets')));

1. **Frontend Setup:**
   * In your frontend code, make sure to import Axios and use it to make asynchronous requests.

// Import Axios in your frontend file

import axios from 'axios';

export default {

async created() {

try {

// Make a GET request to fetch data

const response = await axios.get(`/api/products/${this.$route.params.productid}`);

// Access data from the response

const data = response.data;

// Process the data as needed

// ...

} catch (error) {

console.error('Error fetching data:', error);

}

},

};

* + Ensure that your backend API route (**/api/products/:productid**) is correctly defined to handle the request and respond with the necessary data.
  + Make sure that your frontend is set up to handle the retrieved data appropriately. You can update the state or perform any necessary actions inside the **created** lifecycle hook.
  + If there are headers required for your requests, you can include them in the Axios request like this:

const response = await axios.get(`/api/products/${this.$route.params.productid}`, {

headers: {

// Your headers here

'Content-Type': 'application/json',

// Add any other headers as needed

},

});

* + These steps assume that you have a basic understanding of setting up an Express server, handling routes, and making asynchronous requests with Axios in a frontend framework like Vue.js. Make sure to adapt the code based on your specific project structure and requirements.

In Vue.js, **this.$route** is a reference to the current route object. Vue Router, the official router for Vue.js, provides a way to navigate between different pages or views in a Vue application. The **$route** object contains information about the current route, including the URL, route parameters, query parameters, and other details.

Here's a brief overview of some of the properties you can access through **this.$route**:

* **this.$route.path**: A string representing the current route's path.
* **this.$route.params**: An object containing key/value pairs of dynamic segments and their values.
* **this.$route.query**: An object containing key/value pairs of the query parameters.
* **this.$route.hash**: The hash portion of the URL.

**this.$route.params**: is helping you to add the value that it has in the parameter of you request to add to your shopping cart as it is working with the dynamic value of the url.

***MongoDB Array Update Operators Overview***

<template>

<div>

<!-- Your button to add an item to the shopping cart -->

<button @click="addItemToCart">Add to Cart</button>

</div>

</template>

<script>

export default {

methods: {

async addItemToCart() {

try {

// Make an asynchronous request to your API

const response = await this.$axios.post('/api/add-to-cart', {

// Data to be sent to the API, you can customize this based on your API requirements

itemId: 'your\_item\_id',

quantity: 1, // Assuming you want to add one item

});

// Handle the API response as needed

console.log('Item added to cart:', response.data);

// You can update your local state or perform other actions based on the API response

// For example, update a shopping cart in your Vue data

this.$store.commit('updateCart', response.data);

} catch (error) {

// Handle errors, show an alert, or perform other error handling

console.error('Error adding item to cart:', error);

}

},

},

};

</script>

How to add something to your shopping cart using @click and adding method

<template>

<div>

<!-- Your button to add an item to the shopping cart -->

<button @click="addItemToCart">Add to Cart</button>

</div>

</template>

<script>

export default {

methods: {

async addItemToCart() {

try {

// Make an asynchronous request to your API

const response = await this.$axios.post('/api/add-to-cart', {

// Data to be sent to the API, you can customize this based on your API requirements

itemId: 'your\_item\_id',

quantity: 1, // Assuming you want to add one item

});

// Handle the API response as needed

console.log('Item added to cart:', response.data);

// You can update your local state or perform other actions based on the API response

// For example, update a shopping cart in your Vue data

this.$store.commit('updateCart', response.data);

} catch (error) {

// Handle errors, show an alert, or perform other error handling

console.error('Error adding item to cart:', error);

}

},

},

};

</script>

* We use the **@click** directive to bind the **addItemToCart** method to the button click event.
* The **addItemToCart** method is marked as **async** to allow asynchronous operations, such as making an API request using **await**.
* We use the **$axios** instance (assuming you have Axios installed and set up in your project) to make an asynchronous POST request to the **/api/add-to-cart** endpoint.
* The request payload includes data like the **itemId** and **quantity**, which you can customize based on your API requirements.
* We handle the success response in the **then** block, where you can update your local state or perform other actions based on the API response.
* Errors are caught in the **catch** block, where you can handle errors, show alerts, or perform other error-related actions.

Make sure to adjust the code according to your specific API and application requirements. If you haven't set up Axios, you can use the native **fetch** API or any other HTTP library of your choice.

Computed Property :

In Vue.js, a computed property is a property that is derived from other properties. It's like a function that returns a value based on the values of other properties. The special thing about computed properties is that they are cached and only re-evaluated when one of the dependent properties changes. This makes them efficient for calculations that depend on reactive data.

 computed: {

    itemIsInCart() {

      return this.cartItems.some(item => item.id === this.$route.params.productId);

    }

If you want to load data from server into a Vue component when the component is first displayed, you can put that logic into your component's created lifecycle method.

**Firebase Authentication Setup for Vue.js Frontend:**

1. **Create a Firebase Project:**
   * Visit [console.firebase.google.com](https://console.firebase.google.com/).
   * Log in with your Gmail account and click on the "Add Project" button.
   * Name your project and optionally enable Google Analytics.
   * Click on the "Create Project" button.
2. **Setup Firebase for Web:**
   * In the Firebase Console, click on the "</>" (Web) button to add a web app.
   * Provide a name for your app (e.g., Vue Front-end) and register the app.
   * You'll be provided with a Firebase configuration snippet.
3. **Install Firebase in your Vue.js project:**
   * Open your terminal and run:

bashCopy code

npm install firebase@9.15.0

1. **Integrate Firebase in your Vue.js project:**
   * In your **main.js** file, add the Firebase configuration:

javascriptCopy code

import { initializeApp } from 'firebase/app'; const firebaseConfig = { // Your Firebase configuration here }; const app = initializeApp(firebaseConfig);

1. **Enable Authentication in Firebase:**
   * In the Firebase Console, navigate to "Authentication."
   * Click on "Get Started" and choose your first sign-in method (e.g., Email/Password).
   * Configure the sign-in method, enable email link (if needed), and click "Save."
2. **Configure Authorized Domains:**
   * To ensure your frontend and backend domains are trusted:
     + In the Firebase Console, go to "Authentication."
     + Click on "Settings" and then "Authorized Domains."
     + Add your domain to the list of authorized domains.
3. **Final Steps:**
   * Continue back to the Firebase Console Overview.
   * Make sure to reorganize and adjust your code as needed.
   * Remove the **const app** line if not used in your code.
4. **Note:**
   * Ensure that you don't miss any points mentioned in this guide.
   * If using email authentication, verify that email link functionality is enabled.

**Adding Email Link Authentication to Vue.js App:**

1. **Setup Firebase Email Link Authentication:**
   * Ensure you've followed the previous steps to set up Firebase in your Vue.js app.
2. **Import Necessary Functions:**
   * In the component where you want to handle the sign-in process, import the required functions from **firebase/auth**:

javascriptCopy code

import { getAuth, sendSignInLinkToEmail, isSignInWithEmailLink } from 'firebase/auth';

1. **Create Sign-In Function:**
   * Implement a method in your Vue component to handle the sign-in process:

methods: {

async signIn() {

const email = prompt('Please enter your email to sign in:');

const auth = getAuth();

const actionCodeSettings = {

url: `https://shaunwa-cautious-space-happiness-5v76p774rw63j6w-8080.preview.app.github.dev/products/${this.$route.params.productId}`,

handleCodeInApp: true,

}

await sendSignInLinkToEmail(auth, email, actionCodeSettings);

alert('A login link was sent to the email you provided');

window.localStorage.setItem('emailForSignIn', email);

}

**Handle Email Link Verification:**

* + Implement logic to handle the email link verification. This could be in the component where the user is redirected after clicking the link:

created() {

const auth = getAuth();

if (isSignInWithEmailLink(auth, window.location.href)) {

let email = window.localStorage.getItem('emailForSignIn');

if (!email) {

email = prompt('Please enter your email for confirmation');

}

signInWithEmailLink(auth, email, window.location.href)

.then((result) => {

// User successfully signed in

// You can now proceed with actions like adding products to the shopping cart

console.log(result);

})

.catch((error) => {

console.error(error);

});

}

},

}); } },

1. **Note:**
   * Adjust the URL in **actionCodeSettings** to match the URL where users will be redirected after clicking the email link.
   * Ensure the URL in the **created** hook of your component matches the configured Firebase Dynamic Link.
   * Organize and modify the code based on your project structure.

**Vue.js Watcher: Understanding the watch Option**

The **watch** option in Vue.js allows you to watch for changes on a specific property or expression and execute custom logic when that property or expression changes. This is useful for performing actions in response to changes in data.

**Example:**

Let's say you have a Vue component managing a user object, and you want to perform some actions whenever the user's name changes. You can use the **watch** option to achieve this. <template>

<div>

<h2>User Information</h2>

<p>Name: {{ user.name }}</p>

<input v-model="user.name" placeholder="Enter new name">

</div>

</template>

<script>

export default {

data() {

return {

user: {

name: 'John Doe',

// Other user properties...

},

};

},

watch: {

'user.name': function(newName, oldName) {

console.log(`User's name changed from ${oldName} to ${newName}`);

// Perform additional actions here, such as making an API call or updating other data.

},

},

};

</script>

In this example:

* The **user** object has a **name** property.
* The **watch** option is used to watch changes to **user.name**.
* When the user changes the name in the input field, the watcher function is triggered.

**Deploying Vue.js App with Render and Backend Integration:**

### Storing User Email in Local Storage:

To store the user's email after they sign in, you can use **localStorage** in your Vue.js component:

javascriptCopy code

// Storing the user's email in localStorage window.localStorage.setItem('emailForSignIn', email);

### Handling Email Link Click:

When the user clicks the link, you can retrieve the email from **localStorage** and use it for further authentication:

javascriptCopy code

// In the component where you handle email link verification const auth = getAuth(); if (isSignInWithEmailLink(auth, window.location.href)) { let email = window.localStorage.getItem('emailForSignIn'); if (!email) { email = prompt('Please enter your email for confirmation'); } signInWithEmailLink(auth, email, window.location.href) .then((result) => { // User successfully signed in // Additional actions... }) .catch((error) => { console.error(error); }); }

### Deploying the App:

1. **Build and Serve Frontend:**
   * Run the following commands in your Vue.js project:

bashCopy code

npm install npm run build

1. **Prepare Backend:**
   * Navigate to your backend directory and run:

bashCopy code

npm install npm run build

1. **Copy Frontend dist to Backend:**
   * Copy the contents of the **dist** directory in your frontend to the corresponding directory in your backend.
2. **Configure Backend to Serve Frontend:**
   * In your backend's server file, add the following code to serve the frontend statically:

javascriptCopy code

app.use(express.static( path.resolve(\_\_dirname, '../dist'), { maxAge: '1y', etag: false }, )); // Add this before app.listen app.get('\*', (req, res) => { res.sendFile(path.join(\_\_dirname, '../dist/index.html')); });

1. **Update Server Configuration:**
   * Modify your server configuration to use the correct port and start command:

javascriptCopy code

const port = process.env.PORT || 8000; app.listen(port, () => { console.log('Server is listening on port ' + port) });

1. **Deploy on Render:**
   * Go to [Render](https://render.com/) and create a new web service.
   * Link it to your GitHub repo and choose the backend directory.
   * Set up the environment as "Node" and use the following build and start commands:
     + Build Command: **npm install && npx babel ./src --out-dir ./build**
     + Start Command: **node ./build/server.js**
   * Create the web service.
2. **Update Frontend URL:**
   * Once the deployment is complete, replace the previous URL in your application with the new Render URL.
3. **Push Changes to Git:**
   * Push all the changes, including frontend and backend modifications, to your GitHub repository.
4. **Monitor Deployment:**
   * Check the Render dashboard for deployment progress and ensure the deployment is successful.

**Pinia for State Management**

**Overview:**

Pinia is a state management library designed to be more modular and lightweight than Vuex. It is built with TypeScript in mind and follows Vue 3's Composition API.

**Why Pinia:**

* **Simplicity:**
  + Pinia simplifies the process of managing your application's state with fewer concepts to learn.
* **Type Safety:**
  + All data types are inferred, providing type safety.
  + Enables autocompletion even in JavaScript, enhancing the development experience.
* **Dev Tools Support:**
  + Pinia comes with dev tools support.
  + View and alter your state directly from the dev tools.
  + Utilize the timeline to observe mutations to the store over time.
* **Modularity:**
  + Pinia is modular, allowing for a more flexible and scalable state management solution.
* **Lightweight:**
  + Extremely lightweight, with a size of only 1 kb.
  + Effortlessly integrates into your project, enhancing overall efficiency.

**Installation:**

To integrate Pinia into your project, use the following npm command:

npm install pinia

**Setting up a Store:**

Create a store to manage your application's state. For example, a ticket store:

// store.ts

import { defineStore } from "pinia";

import tempData from "../temp-data";

const useTicketStore = defineStore({

id: "ticketStore",

state: () => ({

tickets: tempData.tickets,

}),

});

export default useTicketStore; // Export the store instance

**Usage in Vue Component:**

<!-- YourVueComponent.vue -->

<template>

<div class="wrapper">

<header class="header-page">

<h1>Ticket Master</h1>

</header>

<nav>

<div>

<router-link to="/">Home</router-link>

<router-link to="/admin">Admin</router-link>

<router-link to="/cart">Cart</router-link>

</div>

</nav>

<!-- Display the content based on the current route -->

<router-view></router-view>

</div>

</template>

<script>

import "../style.css";

import useTicketStore from "../store/ticketStore.ts";

export default {

mounted() {

const ticketStore = useTicketStore();

console.log("Contents of useTicketStore:", ticketStore.tickets);

},

};

</script>

**Organizing Stores:**

* Create separate files for each store, such as **ticketStore.js**.
* Use **defineStore** in your TypeScript file inside the **stores** folder.
* Each **defineStore** instance should have a unique name (id) for easy identification in dev tools.
* Export the store instance with a variable name preceded by the "use" keyword (e.g., **useTicketStore**).

This structure ensures a clear and organized approach to managing state with Pinia in your Vue 3 application.

1. **Install Pinia:**

npm install pinia

1. **Create a Store:** Create a folder named **store** in your **src** directory. Inside this folder, create a file named **ticketStore.ts**.

// store.ts

import { defineStore } from "pinia";

import tempData from "../temp-data";

const useTicketStore = defineStore({

id: "ticketStore",

state: () => {

return {

tickets: tempData.tickets,

cartItems: tempData.cartItems,

};

},

actions: {

decrement(ticketId: number) {

const ticket = this.tickets.find((t) => t.ticketId === ticketId);

if (ticket) {

ticket.count--;

}

},

// You can add more actions related to cart management here

},

});

1. export default useTicketStore;**Use the Store in Vue Components:** In your Vue components, you can now import and use the **useTicketStore** to manage the state.

<template>

<div class="wrapper">

<header class="header-page">

<h1>Ticket Master</h1>

</header>

<nav>

<div>

<router-link to="/">Home</router-link>

<router-link to="/admin">Admin</router-link>

<router-link to="/cart">Cart</router-link>

</div>

</nav>

<div class="ticket">

<h2 class="ticketDesc">All Tickets :</h2>

<Ticket

v-for="ticket in localTickets"

:key="ticket.ticketId"

:ticketName="ticket.ticketName"

:description="ticket.description"

:count="ticket.count"

:price="ticket.price"

:isVip="ticket.isVip"

/>

</div>

<router-view></router-view>

</div>

</template>

<script>

import "../style.css";

import useTicketStore from "../store/ticketStore.ts";

import Ticket from "./Ticket.vue";

export default {

data() {

return {

localTickets: [], // Initialize localTickets as an empty array

};

},

mounted() {

const ticketStore = useTicketStore();

// Assign the value of ticketStore.tickets to localTickets

this.localTickets = ticketStore.tickets;

},

components: {

Ticket,

},

};

</script>

<style>

/\* Add your styles here \*/

</style>

logic or organization of data can be done here }, }; </script>

Make sure to replace **tempdata.tickets** and any other placeholders with your actual data and logic.

**Performing Unit Testing in Vue.js**

* **Setup Pinia Instance:**
  + Create a **setuptest.ts** file to initialize the Pinia instance before running your component.
  + Import Pinia from **setuptest.ts** to resolve initialization issues.
* **Writing Unit Tests:**
  + Use the **describe** function to group related test cases together.

javascriptCopy code

describe("ComponentName", () => { // Test cases go here });

* + The first argument of **describe** is a name describing the unit being tested.
  + The second argument is a function containing the actual test cases.
* **Running Tests:**
  + Execute **npm run test:unit** to run the unit tests.
* **Individual Test Cases:**
  + Use the **it** function to define individual test cases.

javascriptCopy code

it("should describe specific behavior", () => { // Test logic here });

* + The first argument of **it** is a string describing the behavior being tested.
  + The second argument is a function containing the test logic.
  + You can pass the function as an arrow function.
* **Using the Wrapper:**
  + **wrapper** is a testing utility that helps test your components.
  + Utilize the **mount** function, which takes two arguments:
    - The component to be tested.
    - Options:
      * **data**: An object merged with the component's data.
      * **props**: An object containing props to pass to the component.
      * **slots**: An object containing named slots to pass to the component.
      * **attrs**: An object containing attributes to add to the root of the component.
      * **global**: An object containing properties to add to the global Vue instance.
      * **attachTo**: A DOM element or selector where the component should be attached.

**Example:**

const wrapper = mount(ComponentName, {

data() {

return {

// Data properties

};

},

props: {

// Props to pass

},

slots: {

// Named slots

},

attrs: {

// Attributes

},

global: {

// Global properties

},

attachTo: "#app", // Attach to a DOM element

});This organized note provides a clear structure and explanations for key concepts in Vue.js unit testing.