Certainly! The task involves creating a full-stack web application with a Node.js backend using Express, MySQL for the database, and a frontend with HTML, CSS, and JavaScript. Below is an outline of the steps you can follow to achieve this:

**Backend:**

1. **Express Server Setup:**
   * Create the necessary folder structure and files in the **src** directory.
   * Initialize a Node.js project using **npm init**.
   * Install required dependencies (**express**, **mysql2**, etc.) using **npm install**.
   * Set up the basic structure of the **server.js** file.
2. **MySQL Database Setup:**
   * Create a MySQL database.
   * Run the provided SQL queries to create tables in the database.
3. **CRUD Operations for Users:**
   * Implement routes and controllers for user registration and login.
   * Validate user input for registration and handle errors appropriately.
4. **CRUD Operations for Shop Items:**
   * Implement routes and controllers for managing shop items.
   * Include routes to add, retrieve all, retrieve by ID, and delete shop items.
5. **CRUD Operations for Orders:**
   * Implement routes and controllers for managing orders.
   * Include routes to add, retrieve all, retrieve by ID, and retrieve by user ID.
6. **Middleware for Request Validation:**
   * Create middleware to validate POST, PUT, and PATCH requests.
   * Ensure all required fields are present and have valid data types.
   * Provide appropriate error responses.
7. **User Roles:**
   * Implement a route to retrieve all user roles.
   * Manually insert three user roles: "admin," "user," and "guest."
8. **Extra: Admin Permissions:**
   * Implement logic to show/hide features based on user roles.
   * Allow admins to access the "add\_item.html" page.

**Frontend:**

1. **HTML Pages:**
   * Create HTML pages for login, register, shop, add\_item, and orders.
2. **CSS Styling:**
   * Style the HTML pages using CSS.
   * Organize CSS rules in separate files, such as **reset.css** and **style.css**.
3. **JavaScript:**
   * Write JavaScript functions for each HTML page to handle interactions.
   * Use **localStorage** for storing user information.
4. **Login Page:**
   * Create a form with email and password fields.
   * Use JavaScript to send login credentials to the server.
   * Redirect to the shop page on successful login.
5. **Registration Page:**
   * Create a form with name, email, password, repeatPassword, and user\_roles fields.
   * Use JavaScript to send registration data to the server.
   * Show appropriate error messages on duplicate email.
6. **Shop Page:**
   * Fetch and display shop items using JavaScript.
   * Implement buttons to add items to the cart and delete items (visible to admins).
7. **Add Item Page:**
   * Create a form with fields for adding a new shop item.
   * Use JavaScript to send item data to the server.
   * Redirect to the shop page on successful item addition.
8. **Orders Page:**
   * Display a table of orders.
   * Use JavaScript to fetch and filter orders based on user input.
9. **Navigation Bar:**
   * Include a navigation bar on every page.
   * Display login, registration, and logout buttons based on user authentication.
10. **Logout Functionality:**
    * Create a logout button that removes user information from **localStorage**.
    * Redirect to the login page after logging out.
11. **Extra: Admin Features:**
    * Show/hide the "add\_item" page based on user roles.
    * Show delete buttons for shop items only to users with the "admin" role.

The extra tasks involve conditional rendering of elements based on the user's role. This requires additional checks in the frontend JavaScript code to determine what should be displayed or hidden based on the user's role. Also, additional server-side logic may be required to handle these permissions.

User

**You**

Express Server Setup: Create the necessary folder structure and files in the src directory. Initialize a Node.js project using npm init. Install required dependencies (express, mysql2, etc.) using npm install. Set up the basic structure of the server.js file.

ChatGPT

**ChatGPT**

Certainly! Here's a step-by-step guide to set up the Express server:

1. **Folder Structure:** Create the necessary folder structure in your project's **src** directory. The structure should include the following folders and files:

luaCopy code

/your\_project ├── src │ ├── config.js │ ├── helper.js │ ├── server.js │ ├── db │ │ └── Lauryno.sql │ └── routes │ ├── usersRoutes.js │ ├── item\_typesRoutes.js │ ├── ordersRoutes.js │ ├── shop\_itemsRoutes.js │ └── user\_rolesRoutes.js ├── .env ├── .eslintrc.js ├── .gitignore ├── package-lock.json ├── package.json ├── rest.rest

1. **Initialize Node.js Project:** Open a terminal in your project's root directory (**your\_project**) and run the following command to initialize a new Node.js project. This command will prompt you to fill in some details about your project.

bashCopy code

npm init

1. **Install Dependencies:** Install the required dependencies (**express**, **mysql2**, etc.) using the following command:

bashCopy code

npm install express mysql2

This will install Express for your server and MySQL2 for connecting to the MySQL database.

1. **Basic Structure of server.js:** Create the **server.js** file in the **src** directory and set up the basic structure. Here's a simple example:

javascriptCopy code

// src/server.js const express = require('express'); const bodyParser = require('body-parser'); const app = express(); const PORT = process.env.PORT || 3000; // Middleware app.use(bodyParser.json()); app.use(bodyParser.urlencoded({ extended: true })); // Routes const usersRoutes = require('./routes/usersRoutes'); const itemTypesRoutes = require('./routes/item\_typesRoutes'); const ordersRoutes = require('./routes/ordersRoutes'); const shopItemsRoutes = require('./routes/shop\_itemsRoutes'); const userRolesRoutes = require('./routes/user\_rolesRoutes'); app.use('/api/users', usersRoutes); app.use('/api/item\_types', itemTypesRoutes); app.use('/api/orders', ordersRoutes); app.use('/api/shop\_items', shopItemsRoutes); app.use('/api/user\_roles', userRolesRoutes); // Start the server app.listen(PORT, () => { console.log(`Server is running on port ${PORT}`); });

Note: This is a basic example. You might need to customize it based on your specific requirements.

Now you have the basic structure of your Express server. The next steps involve creating routes and controllers in the **routes** folder, connecting to the MySQL database, and implementing CRUD operations.

User

**You**

MySQL Database Setup: Create a MySQL database. Run the provided SQL queries to create tables in the database.

ChatGPT

**ChatGPT**

Sure, let's proceed with creating a MySQL database and running the provided SQL queries to create tables. Please follow these steps:

1. **Create a MySQL Database:**

Open your MySQL command-line interface or a tool like phpMyAdmin. Run the following command to create a new database. Replace **your\_database\_name** with your desired database name.

sqlCopy code

CREATE DATABASE your\_database\_name;

Select the newly created database:

sqlCopy code

USE your\_database\_name;

1. **Run SQL Queries to Create Tables:**

Now, let's run the provided SQL queries to create tables. Save these queries in a file (e.g., **create\_tables.sql**) or run them directly in your MySQL command-line interface.

sqlCopy code

-- Lauryno.sql -- Table 1: Users CREATE TABLE `users` ( `user\_id` int(13) UNSIGNED NOT NULL AUTO\_INCREMENT, `name` varchar(255) NOT NULL, `email` varchar(255) NOT NULL, `password` varchar(255) NOT NULL, `role\_id` int(13) UNSIGNED NOT NULL, PRIMARY KEY (`user\_id`) ) ENGINE=InnoDB; -- Table 2: Item Types CREATE TABLE `item\_types` ( `item\_type\_id` int(13) UNSIGNED NOT NULL AUTO\_INCREMENT, `name` varchar(255) NOT NULL, PRIMARY KEY (`item\_type\_id`) ) ENGINE=InnoDB; -- Table 3: Orders CREATE TABLE `orders` ( `order\_id` int(13) UNSIGNED NOT NULL AUTO\_INCREMENT, `user\_id` int(13) UNSIGNED NOT NULL, `shop\_item\_id` int(13) UNSIGNED NOT NULL, `quantity` varchar(255) NOT NULL, `total\_price` varchar(255) NOT NULL, `status` boolean NOT NULL DEFAULT '0', PRIMARY KEY(`order\_id`) ) ENGINE=InnoDB; -- Table 4: Shop Items CREATE TABLE `shop\_items` ( `shop\_item\_id` int(13) UNSIGNED NOT NULL AUTO\_INCREMENT, `name` varchar(255) NOT NULL, `price` decimal(65) NOT NULL, `description` text NOT NULL, `image` varchar(255) NOT NULL, `item\_type\_id` int(13) UNSIGNED NOT NULL, PRIMARY KEY (`shop\_item\_id`) ) ENGINE=InnoDB; -- Table 5: User Roles CREATE TABLE `user\_roles` ( `role\_id` int(13) UNSIGNED NOT NULL AUTO\_INCREMENT, `name` varchar(255) NOT NULL, PRIMARY KEY (`role\_id`) ) ENGINE=InnoDB;

Save the file and then run it in the MySQL command-line interface or use a tool like phpMyAdmin to execute these queries. This will create the necessary tables in your database.

Now, you should have a MySQL database with the specified tables for your Node.js and Express.js project. Make sure to configure your Node.js application to connect to this database.

User

**You**

CRUD Operations for Users: Implement routes and controllers for user registration and login. Validate user input for registration and handle errors appropriately.

ChatGPT

**ChatGPT**

Certainly! Let's implement CRUD operations for users in your Express.js application. I'll guide you through creating routes and controllers for user registration and login. First, create the necessary files and folders:

1. **Create a controllers Folder:** Inside your **src** directory, create a new folder named **controllers**. This folder will contain the controller logic for different entities.
2. **Create a usersController.js File:** Inside the newly created **controllers** folder, create a file named **usersController.js**. This file will handle user-related CRUD operations.
3. **Implement User Registration and Login Routes in usersRoutes.js:** Inside the **routes** folder, open or create the **usersRoutes.js** file. Add the following code:

javascriptCopy code

// src/routes/usersRoutes.js const express = require('express'); const router = express.Router(); const usersController = require('../controllers/usersController'); // Route for user registration router.post('/register', usersController.registerUser); // Route for user login router.post('/login', usersController.loginUser); module.exports = router;

1. **Implement User Controller Logic:** Inside the **controllers** folder, open the **usersController.js** file and implement the logic for user registration and login:

javascriptCopy code

// src/controllers/usersController.js // Assume you have access to the User model for interacting with the database // Mock user data (replace this with actual database interactions) const users = []; // User registration controller const registerUser = (req, res) => { const { name, email, password, role\_id } = req.body; // Validate input if (!name || !email || !password || !role\_id) { return res.status(400).json({ error: 'All fields are required.' }); } // Check if the email is already registered const existingUser = users.find(user => user.email === email); if (existingUser) { return res.status(400).json({ error: 'Email is already registered.' }); } // Create a new user object (replace this with database insertion) const newUser = { user\_id: users.length + 1, name, email, password, // In a real application, you should hash the password role\_id, }; // Save the user object (replace this with database insertion) users.push(newUser); res.status(201).json({ message: 'User registered successfully.' }); }; // User login controller const loginUser = (req, res) => { const { email, password } = req.body; // Validate input if (!email || !password) { return res.status(400).json({ error: 'Email and password are required.' }); } // Check if the user exists (replace this with database query) const user = users.find(user => user.email === email && user.password === password); if (!user) { return res.status(401).json({ error: 'Invalid email or password.' }); } // In a real application, you might generate a token for authentication res.status(200).json({ message: 'Login successful.', user }); }; module.exports = { registerUser, loginUser, };

Note: In a real application, you should replace the mock data and operations with actual database interactions using an ORM or a query builder.

1. **Update server.js to Use User Routes:** Open or update your **server.js** file to include the user routes:

javascriptCopy code

// src/server.js const express = require('express'); const bodyParser = require('body-parser'); const app = express(); const PORT = process.env.PORT || 3000; // Middleware app.use(bodyParser.json()); app.use(bodyParser.urlencoded({ extended: true })); // Routes const usersRoutes = require('./routes/usersRoutes'); // Add other route imports app.use('/api/users', usersRoutes); // Use other routes // Start the server app.listen(PORT, () => { console.log(`Server is running on port ${PORT}`); });

Now, you have routes and controllers for user registration and login. Test the registration and login endpoints using tools like Postman or by integrating them into your frontend application.

User

**You**

CRUD Operations for Shop Items: Implement routes and controllers for managing shop items. Include routes to add, retrieve all, retrieve by ID, and delete shop items.

ChatGPT

**ChatGPT**

Certainly! Let's proceed with implementing CRUD operations for managing shop items in your Express.js application. Follow the steps below:

1. **Create a shopItemsController.js File:** Inside the **controllers** folder, create a file named **shopItemsController.js**. This file will handle the CRUD operations related to shop items.
2. **Implement Shop Items Routes in shop\_itemsRoutes.js:** Inside the **routes** folder, open or create the **shop\_itemsRoutes.js** file. Add the following code:

javascriptCopy code

// src/routes/shop\_itemsRoutes.js const express = require('express'); const router = express.Router(); const shopItemsController = require('../controllers/shopItemsController'); // Route to add a new shop item router.post('/', shopItemsController.addShopItem); // Route to get all shop items router.get('/', shopItemsController.getAllShopItems); // Route to get a shop item by ID router.get('/:id', shopItemsController.getShopItemById); // Route to delete a shop item by ID router.delete('/:id', shopItemsController.deleteShopItemById); module.exports = router;

1. **Implement Shop Items Controller Logic:** Inside the **controllers** folder, open the **shopItemsController.js** file and implement the logic for adding, retrieving all, retrieving by ID, and deleting shop items:

javascriptCopy code

// src/controllers/shopItemsController.js // Assume you have access to the ShopItem model for interacting with the database // Mock shop item data (replace this with actual database interactions) const shopItems = []; // Controller to add a new shop item const addShopItem = (req, res) => { const { name, price, description, image, item\_type\_id } = req.body; // Validate input if (!name || !price || !description || !image || !item\_type\_id) { return res.status(400).json({ error: 'All fields are required.' }); } // Create a new shop item object (replace this with database insertion) const newShopItem = { shop\_item\_id: shopItems.length + 1, name, price, description, image, item\_type\_id, }; // Save the shop item object (replace this with database insertion) shopItems.push(newShopItem); res.status(201).json({ message: 'Shop item added successfully.' }); }; // Controller to get all shop items const getAllShopItems = (req, res) => { // Retrieve all shop items from the database (replace this with actual query) res.status(200).json({ shopItems }); }; // Controller to get a shop item by ID const getShopItemById = (req, res) => { const { id } = req.params; // Find the shop item by ID (replace this with actual query) const shopItem = shopItems.find(item => item.shop\_item\_id === parseInt(id)); if (!shopItem) { return res.status(404).json({ error: 'Shop item not found.' }); } res.status(200).json({ shopItem }); }; // Controller to delete a shop item by ID const deleteShopItemById = (req, res) => { const { id } = req.params; // Find the index of the shop item by ID (replace this with actual query) const index = shopItems.findIndex(item => item.shop\_item\_id === parseInt(id)); if (index === -1) { return res.status(404).json({ error: 'Shop item not found.' }); } // Remove the shop item from the array (replace this with actual deletion) shopItems.splice(index, 1); res.status(200).json({ message: 'Shop item deleted successfully.' }); }; module.exports = { addShopItem, getAllShopItems, getShopItemById, deleteShopItemById, };

Note: In a real application, you should replace the mock data and operations with actual database interactions using an ORM or a query builder.

1. **Update server.js to Use Shop Items Routes:** Open or update your **server.js** file to include the shop items routes:

javascriptCopy code

// src/server.js const express = require('express'); const bodyParser = require('body-parser'); const app = express(); const PORT = process.env.PORT || 3000; // Middleware app.use(bodyParser.json()); app.use(bodyParser.urlencoded({ extended: true })); // Routes const usersRoutes = require('./routes/usersRoutes'); const shopItemsRoutes = require('./routes/shop\_itemsRoutes'); // Add this line // Add other route imports app.use('/api/users', usersRoutes); app.use('/api/shop\_items', shopItemsRoutes); // Use the shop items routes // Use other routes // Start the server app.listen(PORT, () => { console.log(`Server is running on port ${PORT}`); });

Now you have routes and controllers for managing shop items, including adding, retrieving all, retrieving by ID, and deleting shop items. Test these endpoints using tools like Postman or by integrating them into your frontend application.

User

**You**

CRUD Operations for Orders: Implement routes and controllers for managing orders. Include routes to add, retrieve all, retrieve by ID, and retrieve by user ID.

ChatGPT

**ChatGPT**

Certainly! Let's proceed with implementing CRUD operations for managing orders in your Express.js application. Follow the steps below:

1. **Create an ordersController.js File:** Inside the **controllers** folder, create a file named **ordersController.js**. This file will handle the CRUD operations related to orders.
2. **Implement Orders Routes in ordersRoutes.js:** Inside the **routes** folder, open or create the **ordersRoutes.js** file. Add the following code:

javascriptCopy code

// src/routes/ordersRoutes.js const express = require('express'); const router = express.Router(); const ordersController = require('../controllers/ordersController'); // Route to add a new order router.post('/', ordersController.addOrder); // Route to get all orders router.get('/', ordersController.getAllOrders); // Route to get an order by ID router.get('/:id', ordersController.getOrderById); // Route to get orders by user ID router.get('/user/:user\_id', ordersController.getOrdersByUserId); module.exports = router;

1. **Implement Orders Controller Logic:** Inside the **controllers** folder, open the **ordersController.js** file and implement the logic for adding, retrieving all, retrieving by ID, and retrieving by user ID for orders:

javascriptCopy code

// src/controllers/ordersController.js // Assume you have access to the Order model for interacting with the database // Mock order data (replace this with actual database interactions) const orders = []; // Controller to add a new order const addOrder = (req, res) => { const { user\_id, shop\_item\_id, quantity, total\_price, status } = req.body; // Validate input if (!user\_id || !shop\_item\_id || !quantity || !total\_price || status === undefined) { return res.status(400).json({ error: 'All fields are required.' }); } // Create a new order object (replace this with database insertion) const newOrder = { order\_id: orders.length + 1, user\_id, shop\_item\_id, quantity, total\_price, status, }; // Save the order object (replace this with database insertion) orders.push(newOrder); res.status(201).json({ message: 'Order added successfully.' }); }; // Controller to get all orders const getAllOrders = (req, res) => { // Retrieve all orders from the database (replace this with actual query) res.status(200).json({ orders }); }; // Controller to get an order by ID const getOrderById = (req, res) => { const { id } = req.params; // Find the order by ID (replace this with actual query) const order = orders.find(ord => ord.order\_id === parseInt(id)); if (!order) { return res.status(404).json({ error: 'Order not found.' }); } res.status(200).json({ order }); }; // Controller to get orders by user ID const getOrdersByUserId = (req, res) => { const { user\_id } = req.params; // Find orders by user ID (replace this with actual query) const userOrders = orders.filter(ord => ord.user\_id === parseInt(user\_id)); res.status(200).json({ userOrders }); }; module.exports = { addOrder, getAllOrders, getOrderById, getOrdersByUserId, };

Note: In a real application, you should replace the mock data and operations with actual database interactions using an ORM or a query builder.

1. **Update server.js to Use Orders Routes:** Open or update your **server.js** file to include the orders routes:

javascriptCopy code

// src/server.js const express = require('express'); const bodyParser = require('body-parser'); const app = express(); const PORT = process.env.PORT || 3000; // Middleware app.use(bodyParser.json()); app.use(bodyParser.urlencoded({ extended: true })); // Routes const usersRoutes = require('./routes/usersRoutes'); const shopItemsRoutes = require('./routes/shop\_itemsRoutes'); const ordersRoutes = require('./routes/ordersRoutes'); // Add this line // Add other route imports app.use('/api/users', usersRoutes); app.use('/api/shop\_items', shopItemsRoutes); app.use('/api/orders', ordersRoutes); // Use the orders routes // Use other routes // Start the server app.listen(PORT, () => { console.log(`Server is running on port ${PORT}`); });

Now you have routes and controllers for managing orders, including adding, retrieving all, retrieving by ID, and retrieving by user ID. Test these endpoints using tools like Postman or by integrating them into your frontend application.

User

**You**

Middleware for Request Validation: Create middleware to validate POST, PUT, and PATCH requests. Ensure all required fields are present and have valid data types. Provide appropriate error responses.

ChatGPT

**ChatGPT**

Certainly! Let's create middleware to validate POST, PUT, and PATCH requests in your Express.js application. This middleware will ensure that all required fields are present and have valid data types. Follow these steps:

1. **Create a validationMiddleware.js File:** Inside your **src** directory, create a file named **validationMiddleware.js**. This file will contain the request validation middleware.
2. **Implement Request Validation Middleware:** Open the **validationMiddleware.js** file and add the following code:

javascriptCopy code

// src/validationMiddleware.js // Middleware to validate POST, PUT, and PATCH requests const validateRequest = (req, res, next) => { // Check if the request method is POST, PUT, or PATCH if (req.method === 'POST' || req.method === 'PUT' || req.method === 'PATCH') { // Get the required fields for the current route (modify this based on your routes) const requiredFields = getRequiredFields(req.path); // Check if all required fields are present const missingFields = requiredFields.filter(field => !req.body[field]); if (missingFields.length > 0) { return res.status(400).json({ error: `Missing required fields: ${missingFields.join(', ')}` }); } // Check data types of fields (modify this based on your routes) const invalidFields = validateDataTypes(req.body, requiredFields); if (invalidFields.length > 0) { return res.status(400).json({ error: `Invalid data types for fields: ${invalidFields.join(', ')}` }); } } // Continue to the next middleware or route handler next(); }; // Function to get required fields based on the route path (modify this based on your routes) const getRequiredFields = (path) => { switch (path) { case '/api/users/register': return ['name', 'email', 'password', 'role\_id']; case '/api/shop\_items': return ['name', 'price', 'description', 'image', 'item\_type\_id']; case '/api/orders': return ['user\_id', 'shop\_item\_id', 'quantity', 'total\_price', 'status']; // Add other routes and their required fields default: return []; } }; // Function to validate data types of fields (modify this based on your routes) const validateDataTypes = (data, fields) => { const invalidFields = []; fields.forEach(field => { if (typeof data[field] !== getExpectedDataType(field)) { invalidFields.push(field); } }); return invalidFields; }; // Function to get the expected data type for a field (modify this based on your routes) const getExpectedDataType = (field) => { switch (field) { case 'price': return 'number'; case 'status': return 'boolean'; // Add other fields and their expected data types default: return 'string'; } }; module.exports = validateRequest;

This middleware assumes that you have a specific structure for your routes and their required fields. Modify the **getRequiredFields**, **validateDataTypes**, and **getExpectedDataType** functions based on your actual routes and data validation requirements.

1. **Use Validation Middleware in server.js:** Update your **server.js** file to use the validation middleware. Add the following lines:

javascriptCopy code

// src/server.js const express = require('express'); const bodyParser = require('body-parser'); const app = express(); const PORT = process.env.PORT || 3000; const validateRequest = require('./validationMiddleware'); // Add this line // Middleware app.use(bodyParser.json()); app.use(bodyParser.urlencoded({ extended: true })); // Custom middleware for request validation app.use(validateRequest); // Routes const usersRoutes = require('./routes/usersRoutes'); const shopItemsRoutes = require('./routes/shop\_itemsRoutes'); const ordersRoutes = require('./routes/ordersRoutes'); // Add other route imports app.use('/api/users', usersRoutes); app.use('/api/shop\_items', shopItemsRoutes); app.use('/api/orders', ordersRoutes); // Use other routes // Start the server app.listen(PORT, () => { console.log(`Server is running on port ${PORT}`); });

Now, your Express.js application will use the **validateRequest** middleware to ensure that POST, PUT, and PATCH requests have all required fields and valid data types. Adjust the middleware and functions based on your specific route and validation requirements.

User

**You**

User Roles: Implement a route to retrieve all user roles. Manually insert three user roles: "admin," "user," and "guest."

ChatGPT

**ChatGPT**

Certainly! Let's implement a route to retrieve all user roles and manually insert three user roles ("admin," "user," and "guest") into the user\_roles table. Follow these steps:

1. **Update the User Roles Table Creation SQL:** Modify the SQL query for creating the **user\_roles** table to include the three user roles:

sqlCopy code

-- src/db/Lauryno.sql -- ... (other table creation queries) -- CREATE TABLE for user\_roles CREATE TABLE `user\_roles` ( `role\_id` int(13) UNSIGNED NOT NULL AUTO\_INCREMENT, `name` varchar(255) NOT NULL, PRIMARY KEY (`role\_id`) ) ENGINE=InnoDB; -- Insert default user roles INSERT INTO `user\_roles` (`name`) VALUES ('admin'), ('user'), ('guest');

Run this modified SQL script to create the **user\_roles** table and insert the three user roles.

1. **Create a userRolesController.js File:** Inside the **controllers** folder, create a file named **userRolesController.js**. This file will handle the logic for retrieving all user roles.

javascriptCopy code

// src/controllers/userRolesController.js // Assume you have access to the UserRole model for interacting with the database // Mock user roles data (replace this with actual database interactions) const userRoles = []; // Controller to get all user roles const getAllUserRoles = (req, res) => { // Retrieve all user roles from the database (replace this with actual query) res.status(200).json({ userRoles }); }; module.exports = { getAllUserRoles, };

1. **Create a user\_rolesRoutes.js File:** Inside the **routes** folder, create a file named **user\_rolesRoutes.js**. This file will define the route for retrieving all user roles.

javascriptCopy code

// src/routes/user\_rolesRoutes.js const express = require('express'); const router = express.Router(); const userRolesController = require('../controllers/userRolesController'); // Route to get all user roles router.get('/', userRolesController.getAllUserRoles); module.exports = router;

1. **Update server.js to Use User Roles Route:** Open or update your **server.js** file to include the user roles route:

javascriptCopy code

// src/server.js const express = require('express'); const bodyParser = require('body-parser'); const app = express(); const PORT = process.env.PORT || 3000; // Middleware app.use(bodyParser.json()); app.use(bodyParser.urlencoded({ extended: true })); // Routes const usersRoutes = require('./routes/usersRoutes'); const shopItemsRoutes = require('./routes/shop\_itemsRoutes'); const ordersRoutes = require('./routes/ordersRoutes'); const userRolesRoutes = require('./routes/user\_rolesRoutes'); // Add this line // Add other route imports app.use('/api/users', usersRoutes); app.use('/api/shop\_items', shopItemsRoutes); app.use('/api/orders', ordersRoutes); app.use('/api/user\_roles', userRolesRoutes); // Use the user roles route // Use other routes // Start the server app.listen(PORT, () => { console.log(`Server is running on port ${PORT}`); });

Now you have a route **/api/user\_roles** that retrieves all user roles, and the initial three user roles ("admin," "user," and "guest") are manually inserted into the **user\_roles** table during the table creation process. Test the **/api/user\_roles** endpoint using tools like Postman or by integrating it into your frontend application.

User

**You**

Extra: Admin Permissions: Implement logic to show/hide features based on user roles. Allow admins to access the "add\_item.html" page.

ChatGPT

**ChatGPT**

Certainly! To implement logic for showing/hiding features based on user roles and allowing admins to access the "add\_item.html" page, follow these steps:

1. **Update add\_item.html Page:** In your **add\_item.html** page, add logic to check the user's role and conditionally display the content based on the role. For example:

htmlCopy code

<!-- src/public/add\_item.html --> <!DOCTYPE html> <html lang="en"> <head> <meta charset="UTF-8"> <meta name="viewport" content="width=device-width, initial-scale=1.0"> <title>Add Item</title> <link rel="stylesheet" href="styles/reset.css"> <link rel="stylesheet" href="styles/style.css"> </head> <body> <!-- Your header/navigation bar here --> <div id="content"> <!-- Common content for all users --> <!-- Additional content for admins --> <div id="admin-content" style="display: none;"> <!-- Content accessible only to admins --> <!-- Your form for adding items here --> </div> </div> <!-- Your footer here --> <script src="js/helper.js"></script> <script src="js/add\_item.js"></script> </body> </html>

In this example, the content inside the **admin-content** div is initially hidden. You will use JavaScript to conditionally show this content based on the user's role.

1. **Update add\_item.js Script:** In your **add\_item.js** script, add logic to fetch the user's role from local storage and show/hide the admin-specific content:

javascriptCopy code

// src/public/js/add\_item.js document.addEventListener('DOMContentLoaded', () => { // Your existing JavaScript code // Check if the user is an admin const userRole = localStorage.getItem('role\_id'); if (userRole === 'admin') { // Show the admin-specific content document.getElementById('admin-content').style.display = 'block'; } });

This script checks the user's role stored in local storage. If the role is "admin," it shows the content inside the **admin-content** div.

1. **Update server.js to Include Admin Check Middleware:** Add middleware to check if the user is an admin before allowing access to the "add\_item.html" page:

javascriptCopy code

// src/server.js const express = require('express'); const bodyParser = require('body-parser'); const app = express(); const PORT = process.env.PORT || 3000; const validateRequest = require('./validationMiddleware'); // Middleware app.use(bodyParser.json()); app.use(bodyParser.urlencoded({ extended: true })); // Custom middleware for request validation app.use(validateRequest); // Middleware to check if the user is an admin const isAdmin = (req, res, next) => { const userRole = req.body.role\_id; // Change this to get the role from your authentication mechanism if (userRole !== 'admin') { return res.status(403).json({ error: 'Unauthorized access. Only admins allowed.' }); } next(); }; // Routes const usersRoutes = require('./routes/usersRoutes'); const shopItemsRoutes = require('./routes/shop\_itemsRoutes'); const ordersRoutes = require('./routes/ordersRoutes'); const userRolesRoutes = require('./routes/user\_rolesRoutes'); // Add other route imports app.use('/api/users', usersRoutes); app.use('/api/shop\_items', shopItemsRoutes); app.use('/api/orders', ordersRoutes); app.use('/api/user\_roles', userRolesRoutes); // Use other routes // Serve static files (e.g., HTML, CSS, JS) app.use(express.static('public')); // Middleware to check admin permissions for the "add\_item.html" page app.get('/add\_item.html', isAdmin, (req, res) => { res.sendFile(\_\_dirname + '/public/add\_item.html'); }); // Start the server app.listen(PORT, () => { console.log(`Server is running on port ${PORT}`); });

In this example, the **/add\_item.html** route uses the **isAdmin** middleware to check if the user is an admin before serving the HTML file. If not, it returns a 403 Forbidden response.

Now, when an admin logs in, they will have access to the "add\_item.html" page, and the admin-specific content will be visible. Non-admin users won't be able to access this page. Adjust the authentication and authorization mechanisms based on your actual implementation.