**Creating the Server**  
**Step 1:** Initialize the package.json file using npm init -y for the default configuration, or npm init for a custom setup

**Step 2:** Install the required packages like bcrypt.js, or mongoose etc

**Step 3:** Create a .gitignore file, to tell the git which files I do not want to push in my repo like env file and node\_modules file

**Step 4:** Create a server.js file as a main entry point in the folder

**Step 5:** Connect with database, create a project in Mongodb

KylGPOyqHd3zN9kx

mongodb+srv://dubeyanshul2204:KylGPOyqHd3zN9kx@cluster0.ng0yraz.mongodb.net/

**Step 6:** Create a app in server.js and connect the database with mongoose.connect()

**Step 7:** Configure the CORS (cross origin resource sharing) with app.use()

**Step 8:** Configure the cookie parser and express.json() with app.use()

===========Done with basic server configuration=========

**Step 9:** Create models folder (model defines what properties we save in database), create controllers folder (controller holds the logic for the each route we define), create routes folder (routes holds the api routes used on the website)

**Step 10:** Create a User Model

Create a User.js file inside the models folder.  
Import mongoose and define a UserSchema with userName, email, password, and role. Set proper types and required values.  
Export the model using mongoose.model("User", UserSchema)

**Step 11:** Create Auth Controller

Make a folder named auth/controllers. inside it, create auth-controllers.js and define the registerUser function. This function receives user data from req.body, checks if the user exists, hashes the password, saves the user using the model, and returns a success or error response

**Step 12:** Create Auth Route

Create a folder auth/routes and add a file auth-route.js.  
Import registerUser, create a router using express.Router(), and define a POST route at /register. use registerUser as the handler and export the router instance

**Step 13:** Configure Auth Route in server.js

In your backend server.js, import the route as authRoute from auth/routes/auth-route.js. use app.use("/api/auth", authRoute) to connect it. now the register route is available at /api/auth/register

Go and configure redux in frontend

===========Done with Basic Routing for register==========

**Step 14:** User is still not authenticated after registration, so we need to log them in. Based on the user's role, navigate them to either the admin view or the shopping view. For now, manually update a registered user’s role to admin

**Step-15:** Create a loginUser function in the userController.js and configure it with login logic and cookie setup

**Step 16:** Get credentials and check user  
Extract email and password from req.body. Then, search the database for a user with the matching email. If the user is not found, return a response with an error message like "User not found"

**Step 17:** Validate password and set token  
Use bcrypt.compare() to check if the entered password matches the stored hashed password. If the password is correct, generate a JWT token using jsonwebtoken, store it in an HTTP-only cookie using res.cookie(), and return a success response

**Step 18:** Export the loginUser function from userController.js  
After defining the loginUser function, export it using module.exports so it can be used in other files like route handlers

**Step 19:** Import and use the loginUser function in authRoute.js  
Go to the routes/authRoute.js file. Import the loginUser function from userController.js. Then, create a POST route like router.post("/login", loginUser) to handle login requests from the client

**Step 20:** Create a logout function in authController.js  
Create an async function named logoutUser that clears the cookie using res.clearCookie("token"), and then sends a response with a success message like "User logged out successfully"

Go and configure redux in frontend

============Done with Basic Routing for Login============

**Step 21:** Create the middleware in authController.js  
Create an async function named authMiddleware which will run when the user refreshes the page. Middleware helps detect if a valid cookie (token) exists, so the user can stay logged in and be auto navigated to the correct page without logging in again

**Step 22:** Configure the authMiddleware function  
Inside authMiddleware, extract the token using req.cookies.token. If the token doesn’t exist, return a response with success: false and a message like "Unauthorized user" to prevent access without authentication

**Step 23:** Decode the token  
If the token exists, use jwt.verify() to decode it. If successful, attach the decoded user to req.user and call next() to pass control to the next middleware. If verification fails, send a response with success: false and include the error message

**Step 24:** Configure authMiddleware and logoutUser in authRoute.js

* Create a **POST** route with path "/logout" for logoutUser because logging out involves clearing a cookie, which is a change in the server state — and by REST convention, state-changing operations should use POST.
* Create a **GET** route with path "/check-auth" for authMiddleware because it’s only used to *retrieve* and *verify* the current user’s authentication status, without changing anything — which makes GET the appropriate method

Go and configure redux in frontend

============Done with Middleware and Logout===========

**Step 25:** Set up Cloudinary for Image Upload

* Create an account on [Cloudinary](https://cloudinary.com/) to handle image uploads.
* Go to the **Getting Started** section and select SDK: Node.js.
* Install the Cloudinary using : npm install cloudinary
* Retrieve your cloudName, apiKey, and apiSecret from your Cloudinary dashboard.
* Inside the server directory, create a new folder named helpers.
* Inside the helpers folder, create a file called cloudinary.js to configure and use Cloudinary

**Step 26:** Configure Cloudinary and Multer for File Uploads

* Inside cloudinary.js, create a cloudinary instance, create a multer instance
* Set up the Cloudinary config using cloudName, apiKey, and apiSecret.
* Create a storage instance using multer.memoryStorage() for in-memory file handling.
* Write an async function to upload the received file to Cloudinary using the configured storage, create a upload instance with multer
* Export both upload and async function(handelImageUpload)

**Step 27:** Create Controller and Route for Uploading File to Cloudinary, To handle file uploads to Cloudinary, create an admin folder inside the controller directory. Then, create a products-controller.js file inside it. In this file, add a controller function that calls the async Cloudinary upload logic when a route is hit. This connects your upload route to the storage function

**Step 32:** Create controller function to upload image to Cloudinary

* Inside controllers/admin/products-controller.js, define an async function called handelImageUpload.
* Convert req.file.buffer to base64 and form a base64 URL using the file's mimetype.
* Call imageUploadUtils(url) with the base64 URL and await the result.
* Send a JSON response with success: true and result on success, otherwise return success: false with an error message.
* Export the handelImageUpload function using module.exports

**Step 33:** Create image upload route for admin

* Inside routes/admin/product-routes.js, import express, handleImageUpload from the controller, and upload middleware from Cloudinary helper.
* Initialize express.Router() to set up route handling.
* Create a POST route /upload-image using upload.single("my\_file") as middleware and handleImageUpload as the controller.
* Export the configured router using module.exports

**Now configure the front end**

=======Done with Configuring the cloudinary and Backed=====