**Express Application Setup**

This document provides an overview of the setup for an Express.js application that incorporates user authentication, session management, and MongoDB integration.

**Dependencies:**

1. **express**: Web framework for Node.js, used for handling routing and requests.
2. **path**: Provides utilities for working with file and directory paths.
3. **bcrypt**: Used for hashing passwords (although not shown in use directly here, it's likely used in other parts of the code).
4. **jsonwebtoken (JWT)**: Used for generating and verifying JSON Web Tokens for authentication.
5. **cookie-parser**: Middleware to parse cookies in incoming requests.
6. **express-session**: Middleware to manage user sessions, including storing session data.
7. **connect-flash**: Middleware to store and display flash messages in requests (often used for showing success or error messages).
8. **connect-mongo**: A session store for Express, which stores session data in MongoDB.
9. **mongoose**: MongoDB object modeling tool, used for interacting with the MongoDB database.

**Environment Variables:**

* **JWT\_KEY**: Secret key used for signing JWTs and sessions.
* **MONGODB\_URI**: MongoDB connection string used to connect the app to a MongoDB database.
* **PORT**: Port the server will listen on. Defaults to 8080 if not specified in the environment.

**Middleware Configuration:**

1. **Body Parsing:**
   * express.json(): Parses incoming JSON payloads.
   * express.urlencoded({ extended: true }): Parses incoming form data (supports rich data types).
2. **Static File Serving:**
   * express.static(path.join(\_\_dirname, 'public')): Serves static files (like CSS, JavaScript, images) from the public directory.
3. **Cookie Parser:**
   * cookieParser(): Parses cookies from incoming requests and makes them available in req.cookies.
4. **Session Management:**
   * **express-session**: Manages user sessions.
     + Uses **MongoStore** to store sessions in MongoDB.
     + **cookie settings**: The session cookie expires after 2 hours.
     + **secret**: A secret key (stored in process.env.JWT\_KEY) is used for signing the session cookie. This is crucial for security.
5. **Flash Messages:**
   * **connect-flash**: Used for displaying messages that are only available for the next request, such as success or error messages.

**Routes:**

* **HomeRouter**: Routes for the homepage (/).
* **UserRouter**: Routes related to user authentication and data management (/User).
  + **Note**: The routes are not fully defined here, but the files for these routes are expected in the routes folder as HomeRouter.js and UserRouter.js.

**Starting the Server:**

* The server listens on a port defined in the environment variables (process.env.PORT) or defaults to port 8080.
* Logs a message to the console when the server starts.

**App Structure:**

bash

Copy code

.

├── config

│ └── mongoose-connection.js # Database connection configuration

├── routes

│ ├── HomeRouter.js # Routes for the homepage

│ └── UserRouter.js # Routes related to user actions (login, signup, etc.)

├── public # Directory for static assets (CSS, JS, images)

├── views # Directory for EJS views (UI templates)

├── .env # Environment variables (e.g., MONGODB\_URI, JWT\_KEY)

├── app.js # Main application file (this one)

└── package.json # Project dependencies and scripts

**Sample Route Handling:**

1. **HomeRouter**:
   * Manages requests to the homepage (/).
2. **UserRouter**:
   * Handles routes like login, signup, user profile, and data management.
   * Example:
     + POST /User/register: Registers a new user.
     + POST /User/login: Logs a user in.
     + GET /User/logout: Logs the user out and clears the session.

**Security Notes:**

1. **JWT Secret**: The JWT\_KEY should be kept secure and never exposed in the public code repository. It is critical for ensuring the integrity and security of the JWT tokens.
2. **Session Management**: The session store is configured to use MongoDB, ensuring that session data persists across application restarts.
3. **Environment Variables**: Ensure sensitive information like MONGODB\_URI and JWT\_KEY are stored in a .env file and never pushed to version control.

**Authentication Controller Documentation**

This module handles user authentication operations, including user registration, login, and logout. It uses **bcrypt** for password hashing, **jsonwebtoken (JWT)** for token generation, and **userModel** to interact with the database. The controller contains three key functions: registerUser, loginUser, and logout.

**1. registerUser**

Registers a new user if they do not already have an account.

**Route**: POST /register

**Request Body:**

* email (string): The email of the user (unique).
* password (string): The password of the user.
* fullname (string): The full name of the user.

**Process:**

* Checks if the user already exists in the database.
  + If the user exists, redirects to the home page with an error message.
* If the user does not exist:
  + Hashes the password using bcrypt.
  + Creates a new user in the database with the provided details (email, password, fullname).
  + Generates a JWT token using generateToken and stores it in a cookie named token.
  + Redirects the user to the login page.

**Response:**

* If the user already exists: Redirects to the home page with an error.
* If registration is successful: Redirects to the login page.

**Example:**

json

Copy code

{

"email": "user@example.com",

"password": "securePassword",

"fullname": "John Doe"

}

**2. loginUser**

Authenticates an existing user by checking the credentials.

**Route**: POST /login

**Request Body:**

* email (string): The email of the user.
* password (string): The password of the user.

**Process:**

* Searches for the user by email in the database.
  + If no user is found, redirects to the login page with an error message.
* If the user is found:
  + Compares the entered password with the stored hashed password using bcrypt.
  + If the passwords match, generates a JWT token and stores it in a cookie named token.
  + Redirects the user to their profile page.
  + If the passwords do not match, redirects to the login page with an error message.

**Response:**

* If the login is successful: Redirects to the user's profile page.
* If login fails: Redirects to the login page with an error message.

**Example:**

json

Copy code

{

"email": "user@example.com",

"password": "securePassword"

}

**3. logout**

Logs the user out by clearing the authentication cookie.

**Route**: GET /logout

**Process:**

* Clears the token cookie.
* Redirects the user to the home page.

**Response:**

* Redirects to the home page with the user logged out.

**Utility Functions**

**generateToken(user)**

Generates a JWT token for the authenticated user.

**Parameters:**

* user (object): The user object to generate the token for.

**Returns:**

* A JWT token that can be used for authentication in future requests.

**Dependencies:**

* **bcrypt**: For hashing and comparing passwords securely.
* **jsonwebtoken**: For generating JSON Web Tokens for user authentication.
* **userModel**: The model used to interact with the user collection in the database.
* **generateToken**: A utility function to generate a JWT token.

**Cookie Handling**

* The token cookie is used to store the JWT token for user authentication.
* The cookie is set in the browser for each successful login and registration.
* The cookie is cleared on logout.

**Error Handling**

* If there is an error during any of the operations (registration, login, or logout), the error message is sent in the response.
* For validation errors (e.g., invalid login credentials), users are redirected with a flash message indicating the error.

**Authentication Middleware Documentation**

This middleware function is used to verify if the user is authenticated by checking the validity of the JWT token stored in the browser’s cookies. It is used to protect routes that require authentication.

**Purpose:**

The middleware ensures that the user is logged in by checking the token cookie. If the token is present and valid, it decodes the token and attaches the user object to the request, allowing access to protected routes. If the token is absent or invalid, it redirects the user to the home page with an error message.

**How It Works:**

1. **Token Check**:
   * The middleware first checks if a token cookie exists in the incoming request.
   * If the token cookie is missing, the user is considered not logged in, and the middleware redirects them to the home page with an error message: "You need to login."
2. **Token Verification**:
   * If the token cookie is present, the middleware attempts to decode the token using jwt.verify(). It uses the secret key stored in process.env.JWT\_KEY for verification.
   * If the token is invalid or expired, an error is caught, and the user is redirected to the home page with a "something went wrong" message.
3. **Fetching User Data**:
   * If the token is valid, the email decoded from the token is used to find the corresponding user in the database (excluding the password field).
   * The user data is attached to the req.user object so that the route handler can access the authenticated user’s details.
4. **Proceed to Next Middleware/Handler**:
   * Once the user is successfully authenticated and the user object is attached to the request, the next() function is called to proceed with the route handler.

**Usage:**

This middleware should be added to any route where you want to enforce authentication. It ensures that only authenticated users can access the protected route.

**Example:**

javascript

Copy code

const authMiddleware = require("./middlewares/authMiddleware");

// Protected route that requires authentication

app.get('/user/dashboard', authMiddleware, (req, res) => {

res.render("dashboard", { user: req.user });

});

**Error Handling:**

* **No Token Found**: If the token cookie is not found, the user is redirected with a flash message indicating that login is required.
* **Token Verification Failure**: If there is an error while verifying the token (e.g., the token is invalid or expired), the user is redirected with a flash message indicating "something went wrong."

**Dependencies:**

* **jsonwebtoken (jwt)**: Used to decode and verify the JWT token.
* **userModel**: Used to find the user in the database based on the email decoded from the token.

**Flash Messages:**

* req.flash("error", "You need to login"): If the user is not logged in.
* req.flash("error", "something went wrong"): If there is an issue verifying the token.

**Home Routes Documentation**

This module handles the routes related to user authentication and profile management. It includes routes for the home page, user dashboard, and related functionalities like registering, logging in, logging out, and displaying the user's encrypted data.

**Dependencies:**

* **express**: The web framework for building the server.
* **authController**: Contains the controller functions (registerUser, loginUser, logout) for user registration, login, and logout.
* **isLoggedIn**: Middleware to verify if a user is authenticated before granting access to specific routes.
* **userModel**: The model used to interact with the database for user data.
* **CryptoJS**: Used to decrypt the stored passwords in the user data.
* **dotenv**: For loading environment variables like SECRET\_KEY.

**Routes:**

**1. Home Route (GET /)**

* **Purpose**: Renders the home page and shows an authentication status message.
* **Route**: GET /
* **Process**:
  + Checks if the user is authenticated by looking for the token cookie or authorization header.
  + Passes the authentication status (isAuthenticated) and any error messages from req.flash("error") to the view.
* **Response**:
  + Renders the home page.
  + Displays any errors set via req.flash("error").

**2. User Dashboard Route (GET /user)**

* **Purpose**: Displays the user's dashboard with their decrypted data (e.g., passwords and usernames).
* **Route**: GET /user
* **Middleware**: isLoggedIn
  + Ensures the user is logged in before proceeding.
* **Process**:
  + Fetches the user's data from the database using userModel.findOne({ email: req.user.email }).
  + Populates the datas field (assuming this references the user’s password data).
  + Decrypts the passwords stored in datas using **CryptoJS** with a SECRET\_KEY.
  + Passes the decrypted data along with the user object to the index view.
* **Response**:
  + Renders the index page with the user’s profile and decrypted data.

**Example Decryption Process:**

For each data object in user.datas, the password is decrypted using CryptoJS.AES.decrypt() and converted to a UTF-8 string:

javascript

const decryptedPassword = CryptoJS.AES.decrypt(data.password, process.env.SECRET\_KEY).toString(CryptoJS.enc.Utf8);

**Controller Functions:**

* **registerUser**: Registers a new user.
* **loginUser**: Logs in a user.
* **logout**: Logs out the user by clearing the authentication cookie.

**Middleware:**

* **isLoggedIn**:
  + This middleware checks if the user is logged in by verifying the presence of a valid token.
  + If the token is missing or invalid, the user is redirected to the home page with an error message.
  + If the token is valid, it fetches the user from the database and attaches the user data to the req.user object for use in the route handler.

**Environment Variables:**

* **SECRET\_KEY**: The secret key used for encrypting and decrypting sensitive data (like passwords).

**Error Handling:**

* The route for fetching the user’s data (GET /user) contains error handling for cases where fetching or decrypting the user’s data fails. It catches errors, logs them, and responds with a 500 Internal Server Error message.

**Example Usage:**

1. **Home Page (GET /)**:
   * If a user is not logged in, the home page will show an error message like "You need to login."
   * If a user is logged in (authenticated via token), it will show their status as logged in.
2. **User Dashboard (GET /user)**:
   * If the user is logged in, their profile page will display decrypted information (e.g., URLs, usernames, passwords) from the database.
   * If an error occurs during data fetching or decryption, the page will show an error message.

**User Authentication and Data Management Routes Documentation**

This module defines routes related to user registration, login, logout, and managing their password data. It uses **Express.js** for routing, **bcrypt** for password hashing, **CryptoJS** for encrypting passwords, and **Mongoose** models for interacting with the database.

**Dependencies:**

* **express**: Web framework for building the server and handling routing.
* **bcrypt**: Used for password hashing (though it's not used in the code here).
* **authController**: Contains logic for user registration, login, and logout.
* **userModel**: MongoDB model for interacting with user data.
* **dataModel**: MongoDB model for storing user data (e.g., URLs, usernames, encrypted passwords).
* **isLoggedIn**: Middleware to ensure the user is authenticated before allowing access to specific routes.
* **CryptoJS**: Library for encrypting passwords before storing them in the database.
* **dotenv**: For environment variables like the SECRET\_KEY.

**Routes:**

**1. Login Page (GET /login)**

* **Purpose**: Renders the login page.
* **Route**: GET /login
* **Process**:
  + Checks if there are any error messages in req.flash('error').
  + Passes the error messages to the view.
* **Response**: Renders the login view with any error messages.

**2. Signup Page (GET /signup)**

* **Purpose**: Renders the signup page.
* **Route**: GET /signup
* **Process**:
  + Checks if the user is authenticated by looking for the token cookie or authorization header.
  + Passes the authentication status (isAuthenticated) to the view.
* **Response**: Renders the signup view with authentication status.

**3. Edit Data Page (GET /edit/:id)**

* **Purpose**: Renders the edit page for a specific data entry.
* **Route**: GET /edit/:id
* **Middleware**: isLoggedIn
  + Ensures that the user is authenticated before accessing this route.
* **Process**:
  + Fetches the data by id and populates the associated user information.
* **Response**: Renders the EditPage view with the data to be edited.

**4. Delete Data Confirmation Page (GET /delete/:id)**

* **Purpose**: Renders the delete confirmation page for a specific data entry.
* **Route**: GET /delete/:id
* **Middleware**: isLoggedIn
  + Ensures that the user is authenticated before accessing this route.
* **Process**:
  + Fetches the data by id.
* **Response**: Renders the delete view with the data to be deleted.

**5. Logout (GET /logout)**

* **Purpose**: Logs the user out by clearing the authentication cookie.
* **Route**: GET /logout
* **Process**:
  + Calls the logout function from the authController, which clears the token cookie.
* **Response**: Redirects to the home page or login page.

**6. Register User (POST /register)**

* **Purpose**: Registers a new user.
* **Route**: POST /register
* **Process**:
  + Uses the registerUser function from the authController to handle user registration.
* **Response**: Redirects to the login page or returns an error if registration fails.

**7. Login User (POST /login)**

* **Purpose**: Logs in a user.
* **Route**: POST /login
* **Process**:
  + Uses the loginUser function from the authController to handle user authentication.
* **Response**: Redirects to the user's dashboard or the login page with an error if login fails.

**8. Create Data (POST /data)**

* **Purpose**: Allows the user to save a new data entry (e.g., username, password) after encrypting the password.
* **Route**: POST /data
* **Middleware**: isLoggedIn
  + Ensures the user is authenticated before allowing them to create new data.
* **Process**:
  + Encrypts the password using **CryptoJS** with a SECRET\_KEY.
  + Creates a new dataModel entry with the encrypted password, URL, and username.
  + Saves the data to the database and associates it with the logged-in user.
* **Response**: Redirects the user to their dashboard.

**9. Update Data (POST /update/:id)**

* **Purpose**: Allows the user to update an existing data entry.
* **Route**: POST /update/:id
* **Middleware**: isLoggedIn
  + Ensures the user is authenticated before allowing them to update data.
* **Process**:
  + Finds and updates the data entry by id with the new URL, username, and password.
* **Response**: Redirects the user to their dashboard.

**10. Delete Data (POST /delete/:id)**

* **Purpose**: Allows the user to delete a data entry.
* **Route**: POST /delete/:id
* **Middleware**: isLoggedIn
  + Ensures the user is authenticated before allowing them to delete data.
* **Process**:
  + Finds and deletes the data entry by id.
* **Response**: Redirects the user to their dashboard.

**Error Handling:**

* If an error occurs during data creation, update, or deletion (e.g., database errors), the server responds with a 500 Internal Server Error message.

**Environment Variables:**

* **SECRET\_KEY**: A secret key used for encrypting passwords using **CryptoJS**.

**Authentication Middleware:**

* **isLoggedIn**: Ensures that the user is authenticated before accessing protected routes, such as those for editing, deleting, or creating data.

**Example Use Case:**

1. **Login**: Users can log in via the /login route. If they provide valid credentials, they are redirected to their dashboard (/user).
2. **Data Management**: Users can create, update, and delete their stored password data via routes like /data, /update/:id, and /delete/:id.

**Token Generation Utility**

**Purpose:**

This utility function generates a JSON Web Token (JWT) for a user. The token is used for authenticating users in the application, allowing them to access protected routes after logging in.

**Dependencies:**

* **jsonwebtoken**: A package used for signing and verifying JSON Web Tokens (JWTs).

**Function: generateToken**

**Parameters:**

* **user** (Object): The user object containing the user's information.
  + Expected properties in the user object:
    - **email** (String): The user's email address.
    - **\_id** (String): The user's unique identifier (usually from the database).

**Process:**

1. **Payload Creation**: The function creates a payload using the user's email and ID.
2. **Token Generation**: The jwt.sign() method is used to create a JWT. The payload is signed with the secret key (process.env.JWT\_KEY).
3. **Return**: The generated JWT is returned.

**Example:**

const token = generateToken(user);

**Response:**

* The function returns a JWT (string) that is valid for a period defined by the secret key's configuration

**Environment Variables:**

* **JWT\_KEY**: The secret key used to sign the JWT. It should be stored securely in environment variables to prevent unauthorized access.

**Usage in the Application:**

* The generated token can be set as a cookie or included in HTTP headers for authenticating requests.
* The token is used by the authentication middleware to verify the user's identity and grant access to protected routes