Complete Authentication System Documentation

Project Overview

A full-stack Node.js authentication system with separate user and admin panels. Uses Express.js, MongoDB, bcrypt for password hashing, and express-session for session management.

Architecture

MVC Pattern

- **Model -** Data structure (MongoDB schemas)
- **View -** Frontend templates (Handlebars/HBS)
- **Controller -** Business logic and request handling
- **Routes -** URL mapping to controllers
- **Middleware -** Authentication gatekeepers

Two Systems

1. **User System -** Regular users can register, login, logout
2. **Admin System -** Admins manage users (CRUD operations)

Technology Stack

- **Backend Framework -** Express.js
- **Database -** MongoDB
- **Password Hashing -** Bcrypt
- **Session Management -** express-session
- **View Engine -** Handlebars (HBS)
- **Additional -** nocache, mongoose

Project Structure

```
userauth/
```

```
├── db/
│   └── connectDB.js          # MongoDB connection
├── model/
│   ├── userModel.js          # User schema
│   └── adminModel.js         # Admin schema
├── controller/
│   ├── userController.js     # User business logic
│   └── adminController.js    # Admin business logic
├── middleware/
│   └── auth.js               # Authentication middleware
├── routes/
│   ├── user.js               # User routes
│   └── admin.js              # Admin routes
├── views/
│   ├── user/
│   │   ├── login.hbs
│   │   ├── register.hbs
│   │   ├── userHome.hbs
│   │   └── layout.hbs
│   └── admin/
│       ├── login.hbs
│       ├── register.hbs
│       ├── dashboard.hbs
│       └── editUser.hbs
├── public/
│   └── style/                # CSS files
├── server.js                 # Main server file
├── createFirstAdmin.js       # Admin creation script
├── package.json
└── node_modules/
```

Installation & Setup

1. Initialize Project

```bash
mkdir userauth
cd userauth
npm init -y
```

2. Install Dependencies

```bash
npm install express mongoose bcrypt express-session hbs nocache
```

Packages:

- `express` - Web server framework
- `mongoose` - MongoDB ODM
- `bcrypt` - Password hashing (salt rounds: 10)
- `express-session` - Session management
- `hbs` - Handlebars view engine
- `nocache` - Prevent browser caching

3. Create Folder Structure

```bash
mkdir db model controller middleware routes views public
mkdir views/user views/admin public/style
```

4. MongoDB Setup

Ensure MongoDB is running:

```bash
mongod
```

Database Connection

File: `db/connectDB.js`

Purpose: Establish connection to MongoDB

```javascript
```

```
const mongoose = require('mongoose');

const connectDB = async () => {
  try {
    const conn = await mongoose.connect("mongodb://127.0.0.1:27017/usermanagementDb");
    console.log(`MongoDB Connected: ${conn.connection.host}`);
  } catch (error) {
    console.error('Database connection error:', error.message);
    process.exit(1);
  }
};

module.exports = connectDB;
```

Key Concepts:

- `mongoose.connect()` - Connects to MongoDB
- Connection string format: `mongodb://host:port/databaseName`
- `async/await` - Waits for connection
- `process.exit(1)` - Stops app if connection fails

Data Models

User Model

File: `model/userModel.js`

Purpose: Define user data structure

```
javascript
```

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  email: {
    type: String,
    required: true
  },
  password: {
    type: String,
    required: true
  }
});

module.exports = mongoose.model('user', userSchema);
```

Key Concepts:

- `required: true` - Field must be provided
- Schema defines how data looks in database
- Model created with name 'user' → MongoDB creates 'users' collection

Admin Model

File: `model/adminModel.js`

Purpose: Define admin data structure

```
javascript
```

```
const mongoose = require('mongoose');

const adminSchema = new mongoose.Schema({
  email: {
    type: String,
    required: true
  },
  password: {
    type: String,
    required: true
  }
});

module.exports = mongoose.model('admin', adminSchema);
```

Same structure as user but separate collection for admins

Authentication Middleware

File: `middleware/auth.js`

Purpose: Protect routes and check user sessions

```
javascript
```

```
const checkSession = (req, res, next) => {
  if (req.session.user) {
    next();
  } else {
    res.redirect('/user/login');
  }
};

const isLogin = (req, res, next) => {
  if (req.session.user) {
    res.redirect('/user/home');
  } else {
    next();
  }
};

const checkAdminSession = (req, res, next) => {
  if (req.session.admin) {
    next();
  } else {
    res.redirect('/admin/login');
  }
};

module.exports = {
  checkSession,
  isLogin,
  checkAdminSession
};
```

Three Middleware Functions

1. **checkSession -** Allows only logged-in users
   - Checks `req.session.user`
   - Redirects to login if not authenticated
2. **isLogin -** Prevents logged-in users from seeing login page
   - If already logged in → redirects to home
   - If not logged in → allows access to login page
3. **checkAdminSession -** Allows only logged-in admins
   - Checks `req.session.admin`

- Redirects to admin login if not authenticated

User Controller

File: `controller/userController.js`

Purpose: Handle all user-related logic

Function 1: loadRegister

Shows registration page

Function 2: registerUser

Handles user registration

Flow:

1. Extract email and password from form
2. Check if user already exists with `findOne({ email })`
3. If exists → show error
4. Hash password with `bcrypt.hash(password, 10)`
5. Create new user and save to database
6. Redirect to login with success message

```javascript
```

```
const registerUser = async (req, res) => {
  try {
    const { email, password } = req.body;

    const user = await userSchema.findOne({ email });
    if (user) {
      return res.render('user/register', { message: 'User already exists' });
    }

    const hashedPassword = await bcrypt.hash(password, 10);
    const newUser = new userSchema({ email, password: hashedPassword });
    await newUser.save();

    res.redirect("/user/login?success=registered");
  } catch (error) {
    console.error(error);
    res.status(500).send("Server error");
  }
};
```

Key Keywords:

- `req.body` - Form data
- `findOne()` - Search for one document
- `bcrypt.hash()` - Hashes password
- `new userSchema()` - Creates instance
- `.save()` - Saves to database
- `res.redirect()` - Sends to another page

Function 3: loadLogin

Shows login page with optional success message

Flow:

1. Check for `req.query.success === 'registered'`
2. If true, set message: "User created successfully! Please login."
3. Render login page with message

Function 4: login

Handles user login

Flow:

1. Extract email and password from form
2. Find user by email with `findOne({ email })`
3. If not found → show error
4. Compare passwords with `bcrypt.compare(password, user.password)`
5. If password doesn't match → show error
6. Set session: `req.session.user = true`
7. Redirect to home page

```javascript
const login = async (req, res) => {
  try {
    const { email, password } = req.body;

    const user = await userSchema.findOne({ email });
    if (!user) {
      return res.render('user/login', { message: "User does not exist" });
    }

    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) {
      return res.render("user/login", { message: "Incorrect credentials" });
    }

    req.session.user = true;
    res.redirect('/user/home');
  } catch (error) {
    console.error(error);
    res.status(500).send("Server error");
  }
};
```

Key Keywords:

- `bcrypt.compare()` - Compares plain password with hashed
- Returns true if match, false if not

- `req.session.user = true` - Marks user as logged in

Function 5: loadHome

Shows home page (simple - middleware checks session)

Function 6: logout

Logs user out

```javascript
const logout = (req, res) => {
  req.session.user = null;
  res.redirect('/user/login');
};
```

Admin Controller

File: `controller/adminController.js`

Purpose: Handle all admin-related logic

Function 1: loadLogin

Shows admin login page

Function 2: login

Handles admin login (same flow as user but uses `adminModel` and `req.session.admin`)

Function 3: loadRegister

Shows admin registration page

Function 4: registerAdmin

Creates new admin account (similar to `registerUser`)

Function 5: loadDashboard

Shows all users

Flow:

1. Check admin session

2. Fetch all users with `userModel.find({})`
3. Check for success messages (deleted/updated)
4. Render dashboard with users array and message

```javascript
const loadDashboard = async (req, res) => {
  try {
    const admin = req.session.admin;
    if (!admin) return res.redirect("/admin/login");

    const users = await userModel.find({});

    let message = null;
    if (req.query.success === 'deleted') {
      message = 'User deleted successfully';
    } else if (req.query.success === 'updated') {
      message = 'User updated successfully';
    }

    res.render("admin/dashboard", { users, message });
  } catch (error) {
    console.error("Error loading dashboard:", error);
    res.status(500).send("Internal Server Error");
  }
};
```

Key Keywords:

- `find({})` - Gets all documents
- Pass multiple variables to view: `{ users, message }`

Function 6: deleteUser

Deletes a user from database

Flow:

1. Get user ID from URL parameter: `req.params.id`
2. Delete with `findByIdAndDelete(userId)`
3. Redirect to dashboard with success message

```javascript
const deleteUser = async (req, res) => {
  try {
    const userId = req.params.id;
    await userModel.findByIdAndDelete(userId);
    res.redirect("/admin/dashboard?success=deleted");
  } catch (error) {
    console.error("Error deleting user:", error);
    res.status(500).send("Internal Server Error");
  }
};
```

Function 7: loadEditUser

Shows edit form for a specific user

Flow:

1. Get user ID from `req.params.id`
2. Find user with `findById(userId)`
3. Render edit page with user data

Function 8: updateUser

Updates user email/password

Flow:

1. Get user ID and new data from form
2. Create update object
3. Only hash password if provided (optional update)
4. Update with `findByIdAndUpdate(userId, updateData)`
5. Redirect with success message

```javascript
```

```
const updateUser = async (req, res) => {
  try {
    const userId = req.params.id;
    const { email, password } = req.body;

    const updateData = { email };

    if (password && password.trim() !== '') {
      const hashedPassword = await bcrypt.hash(password, 10);
      updateData.password = hashedPassword;
    }

    await userModel.findByIdAndUpdate(userId, updateData);
    res.redirect("/admin/dashboard?success=updated");
  } catch (error) {
    console.error("Error updating user:", error);
    res.status(500).send("Internal Server Error");
  }
};
```

Function 9: logout

Destroys admin session

javascript

```
const logout = async (req, res) => {
  try {
    req.session.destroy((err) => {
      if (err) {
        console.error("Error destroying session:", err);
        return res.status(500).send("Internal Server Error");
      }
      res.redirect("/admin/login");
    });
  } catch (error) {
    console.error("Error during logout:", error);
    res.status(500).send("Internal Server Error");
  }
};
```

Key Concept: `req.session.destroy()` completely removes session from server

User Routes

File: `routes/user.js`

Purpose: Map URLs to user controller functions

```javascript
const express = require('express');
const router = express.Router();
const userController = require('../controller/userController');
const auth = require('../middleware/auth');

router.get('/login', userController.loadLogin);
router.post('/login', userController.login);

router.get('/register', userController.loadRegister);
router.post('/register', userController.registerUser);

router.get('/home', auth.checkSession, userController.loadHome);
router.get('/logout', auth.checkSession, userController.logout);

module.exports = router;
```

Route Breakdown

Method	Path	Middleware	Controller	Purpose
GET	/login	-	loadLogin	Show login page
POST	/login	-	login	Process login
GET	/register	-	loadRegister	Show register page
POST	/register	-	registerUser	Process registration
GET	/home	checkSession	loadHome	Show home (protected)
GET	/logout	checkSession	logout	Log out (protected)

Key Concepts:

- `router.get()` - Handle GET requests
- `router.post()` - Handle POST requests
- Middleware placed between route and controller
- Protected routes have middleware as 2nd parameter

Admin Routes

File: routes/admin.js

Purpose: Map URLs to admin controller functions

```javascript
const express = require('express');
const router = express.Router();
const adminController = require('../controller/adminController');
const auth = require('../middleware/auth');

router.get('/login', adminController.loadLogin);
router.post('/login', adminController.login);

router.get('/register', auth.checkAdminSession, adminController.loadRegister);
router.post('/register', auth.checkAdminSession, adminController.registerAdmin);

router.get('/dashboard', auth.checkAdminSession, adminController.loadDashboard);

router.get('/user/edit/:id', auth.checkAdminSession, adminController.loadEditUser);
router.post('/user/edit/:id', auth.checkAdminSession, adminController.updateUser);
router.get('/user/delete/:id', auth.checkAdminSession, adminController.deleteUser);

router.get('/logout', auth.checkAdminSession, adminController.logout);

module.exports = router;
```

Route Breakdown

Method	Path	Middleware	Controller	Purpose
GET	/login	-	loadLogin	Show admin login
POST	/login	-	login	Process admin login
GET	/register	checkAdminSession	loadRegister	Show admin register
POST	/register	checkAdminSession	registerAdmin	Create admin
GET	/dashboard	checkAdminSession	loadDashboard	Show all users
GET	/user/edit/:id	checkAdminSession	loadEditUser	Show edit form
POST	/user/edit/:id	checkAdminSession	updateUser	Save changes
GET	/user/delete/:id	checkAdminSession	deleteUser	Delete user

Method	Path	Middleware	Controller	Purpose
GET	/logout	checkAdminSession	logout	Log out admin

Key Concepts:

- `:id` is a route parameter (dynamic)
- Accessed with `req.params.id`
- All admin routes protected with `checkAdminSession`

Server Setup

File: `server.js`

Purpose: Configure Express app and start server

```javascript
```

```
const express = require('express');
const app = express();
const userRoutes = require('./routes/user');
const adminRoutes = require('./routes/admin');
const path = require('path');
const session = require('express-session');
const connectDB = require('./db/connectDB');
const nocache = require('nocache');

// Middleware setup (ORDER MATTERS!)
app.use(nocache());

app.use(session({
  secret: 'mysecretkey',
  resave: false,
  saveUninitialized: true,
  cookie: { maxAge: 1000 * 60 * 60 * 24 }
}));

// View engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'hbs');

// Static files and body parsers
app.use(express.static('public'));
app.use(express.urlencoded({ extended: true }));
app.use(express.json());

// Routes
app.use('/user', userRoutes);
app.use('/admin', adminRoutes);

// Database and server
connectDB();

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

Middleware Configuration Explained

1. **nocache() -** Prevents browser caching (important for logout)

2. **session()** - Enables session management
   - `secret` - Used to sign session cookies (security)
   - `resave: false` - Don't save if nothing changed
   - `saveUninitialized: true` - Create session even if empty
   - `cookie.maxAge` - Session lifetime in milliseconds (24 hours)
3. **View engine** - Configure Handlebars
   - `views` path - Where template files are
   - `view engine` - Use HBS for rendering
4. **Body parsers** - Parse incoming data
   - `express.static()` - Serve static files
   - `express.urlencoded()` - Parse form data
   - `express.json()` - Parse JSON
5. **Routes** - Mount routers with prefixes
   - `/user` prefix → routes become `/user/login`, `/user/register`, etc.
   - `/admin` prefix → routes become `/admin/login`, `/admin/dashboard`, etc.

Important: Middleware Order Matters!

- nocache should be first
- Session before routes
- Routes should be last (almost)

Create First Admin

File: `createFirstAdmin.js`

Purpose: One-time script to create initial admin account

```javascript
```

```
const mongoose = require('mongoose');
const bcrypt = require('bcrypt');
const adminModel = require('./model/adminModel');
const connectDB = require('./db/connectDB');

const createFirstAdmin = async () => {
  try {
    await connectDB();

    const email = 'admin@example.com';
    const password = 'admin123';

    const existingAdmin = await adminModel.findOne({ email });
    if (existingAdmin) {
      console.log('Admin already exists!');
      process.exit(0);
    }

    const hashedPassword = await bcrypt.hash(password, 10);

    const admin = new adminModel({
      email,
      password: hashedPassword
    });

    await admin.save();
    console.log('✅ First admin created successfully!');
    console.log('Email:', email);
    console.log('Password:', password);

    process.exit(0);
  } catch (error) {
    console.error('Error creating admin:', error);
    process.exit(1);
  }
};

createFirstAdmin();
```

How to use:

bash

```
node createFirstAdmin.js
```

Key Concepts:

- `process.exit(0)` - Successful exit
- `process.exit(1)` - Error exit
- Runs once, then connection closes
- Creates admin with credentials (change them in file)

Important Concepts

1. Bcrypt & Password Hashing

Why hash passwords?

- Never store plain text passwords in database
- If database is leaked, passwords are still secure
- Hashed passwords cannot be reversed

How bcrypt works:

- `saltround = 10` means hash is computed 1024 times
- Each password gets unique salt
- Same password produces different hash each time

Comparison:

```javascript
const hashedPassword = await bcrypt.hash('password123', 10);
// Result: $2b$10$abcdef123...

const isMatch = await bcrypt.compare('password123', hashedPassword);
// Returns true if password matches
```

2. Sessions

What is a session?

- Server-side storage for user data
- Browser gets a session ID cookie
- Session ID is used to retrieve user info

Flow:

1. User logs in
2. Server creates session: `req.session.user = true`
3. Browser gets cookie with session ID
4. On next request, browser sends cookie
5. Server retrieves session using ID

Why sessions?

- User doesn't need to login on every request
- Server remembers who is logged in
- More secure than storing data in browser

3. Middleware

What is middleware?

- Function that runs before controller
- Can inspect/modify requests
- Can allow or block access

Flow:

```
Request → Middleware 1 → Middleware 2 → Controller → Response
```

Example:

```javascript
router.get('/home', auth.checkSession, userController.loadHome);
// Request → checkSession (check if logged in) → loadHome → Response
```

4. Request Objects

req.body - Form data (POST)

```javascript
// When form is submitted:
const { email, password } = req.body;
```

req.params - URL parameters

```javascript
// URL: /admin/user/delete/12345
const userId = req.params.id; // '12345'
```

req.query - Query strings

```javascript
// URL: /user/login?success=registered
const success = req.query.success; // 'registered'
```

req.session - Session data

```javascript
req.session.user = true;  // Set
const user = req.session.user;  // Get
```

5. Database Operations

Create:

```javascript
const user = new userModel({ email, password });
await user.save();
```

Read (find all):

```javascript
```

```
const users = await userModel.find({});
```

Read (find one):

```
javascript

const user = await userModel.findOne({ email });
```

Read (find by ID):

```
javascript

const user = await userModel.findById(userId);
```

Update:

```
javascript

await userModel.findByIdAndUpdate(userId, { email, password });
```

Delete:

```
javascript

await userModel.findByIdAndDelete(userId);
```

Authentication Flow

User Registration Flow

```
1. User visits /user/register
   ↓
2. Browser loads registration form
   ↓
3. User fills email & password
   ↓
4. User clicks submit
   ↓
5. Form POSTs to /user/register
   ↓
6. registerUser controller runs:
```

- Extract email & password from req.body
- Check if user exists (findOne)
- Hash password with bcrypt
- Create new user
- Save to database
- Redirect to /user/login?success=registered

↓

7. Browser redirects to login page

↓

8. Success message displays

User Login Flow

1. User visits /user/login

↓

2. Browser loads login form

↓

3. User enters credentials

↓

4. User clicks submit

↓

5. Form POSTs to /user/login

↓

6. login controller runs:
- Extract email & password
- Find user (findOne)
- Compare password with bcrypt.compare
- If match: req.session.user = true
- Redirect to /user/home

↓

7. Browser redirects to home

↓

8. Session cookie is stored in browser

Protected Route Access

1. User tries to access /user/home

↓

2. checkSession middleware runs:
- Check if req.session.user exists
- If yes: call next() → proceed to controller
- If no: redirect to /user/login

↓
3. If allowed, loadHome controller renders page

Logout Flow

1. User clicks logout
↓
2. GET request to /user/logout
↓
3. logout controller runs:
- req.session.user = null
- Redirect to /user/login
↓
4. Session is cleared
↓
5. Browser redirected to login
↓
6. If user tries /user/home again:
- checkSession middleware blocks
- Redirects to login

Admin User Management Flow

1. Admin logs in → /admin/dashboard
↓
2. Dashboard displays all users
↓
3. Admin can:
a) Edit user: /admin/user/edit/:id
- loadEditUser shows form
- updateUser saves changes

b) Delete user: /admin/user/delete/:id
- deleteUser removes from database

c) Logout: /admin/logout
- req.session.destroy() clears session

Testing Checklist

User Registration

- [ ] Visit `/user/register`
- [ ] Fill form with email & password
- [ ] Submit form
- [ ] Redirects to `/user/login?success=registered`
- [ ] Success message displays
- [ ] Check database: user exists with hashed password

User Login

- [ ] Visit `/user/login`
- [ ] Enter credentials
- [ ] Submit form
- [ ] Redirects to `/user/home`
- [ ] Home page loads

Protected Routes

- [ ] Try accessing `/user/home` without login
- [ ] Should redirect to `/user/login`

User Logout

- [ ] Click logout button
- [ ] Redirects to `/user/login`
- [ ] Try accessing `/user/home`
- [ ] Should redirect to login (session cleared)

Admin Creation

- [ ] Run `node createFirstAdmin.js`
- [ ] Check database for admin with hashed password

Admin Login

- [ ] Visit `/admin/login`
- [ ] Enter admin credentials
- [ ] Submit form
- [ ] Redirects to `/admin/dashboard`
- [ ] Dashboard shows all users

Admin Edit User

- [ ] Click edit on a user
- [ ] Change email/password
- [ ] Click save
- [ ] Redirects with "User updated successfully" message
- [ ] Check database for changes

Admin Delete User

- [ ] Click delete on a user
- [ ] Redirects with "User deleted successfully" message
- [ ] User removed from dashboard and database

Admin Logout

- [ ] Click logout
- [ ] Redirects to `/admin/login`
- [ ] Try accessing `/admin/dashboard`
- [ ] Should redirect to login

Key Takeaways

1. **MVC separates concerns -** Models, Views, Controllers have distinct responsibilities
2. **Middleware controls access -** Authentication happens before requests reach controllers
3. **Sessions track users -** Server stores login state, browser gets session cookie
4. **Bcrypt secures passwords -** Never store plain text, always hash
5. **Routes map URLs to logic -** URL determines which controller function runs
6. **Database operations are async -** Always use await and handle errors
7. **Middleware order matters -** Session must be before routes
8. **Try-catch handles errors -** Always wrap async operations in error handlers

---

Common Issues & Solutions

Issue	Solution
"MongoDB connection failed"	Make sure `mongod` is running

Issue	Solution
"Cannot find module"	Check file paths in require() statements
"Login not working"	Check if session middleware is configured
"Password not hashing"	Make sure using `await bcrypt.hash()`
"Redirect not working"	Check route paths are correct
"Views not rendering"	Check views folder path and file names
"Middleware not running"	Check middleware is placed before routes
"Session lost after refresh"	Check session secret is configured

---

Commands Reference

```bash
# Initialize project
npm init -y

# Install dependencies
npm install express mongoose bcrypt express-session hbs nocache

# Start MongoDB
mongod

# Start application
node server.js

# Create first admin
node createFirstAdmin.js

# Connect to MongoDB shell
mongosh
use usermanagementDb
db.users.find()
db.admins.find()
```

Summary

You've built a complete authentication system with:

- ✅ User registration & login
- ✅ Password hashing with bcrypt
- ✅ Session-based authentication
- ✅ Protected routes with middleware
- ✅ Admin panel for user management (CRUD)
- ✅ Database persistence with MongoDB
- ✅ Clean MVC architecture

This is production-ready code that can be enhanced with validation, JWT, roles, and more!