GharSetu

A project submitted in partial fulfilment of requirements for the degree of

**MASTER OF COMPUTER APPLICATION**

BY

**Siddhartha Raghuvanshi**

**Roll No: 2366003038**

**[Session: 2024-2025]**

Under the Supervision of

**Rakesh Joshi**

School of Computer Science & Applications



IIMT UNIVERSITY

‘O’ Pocket, Ganga Nagar Colony, Mawana Road, Meerut (U.P.), India

**DECLARATION BY THE CANDIDATE**

The Research work embodied in this project entitled **“GharSetu”** has been carried out at the **“School of Computer Science & Applications, IIMT University, Meerut, U.P.”** The extent of information derived from the existing literature has been indicated in the body of the project at appropriate places along with the source of information. The work is original and has not been submitted in part or for nay degree or diploma of this or any other University.

**Date: 06-06-2025 Siddhartha Raghuvanshi**

**Place: Meerut Roll No: 2366003038**

**CERTIFICATE BY THE SUPERVISOR**

This is to certify that the project report entitled **“GharSetu”** submitted by **Siddhartha Raghuvanshi** in partial fulfilment of requirements for degree of **Master of Computer Application** at the **“School of Computer Science & Applications, IIMT University, Meerut, U.P.”** is a record of the candidate own work carried out by him under my supervision. The matter embodied in this project is original and has not submitted for the award of any other degree.

**Suraj Malik Rachna Chaudhary**

**Dean HOD**

**SOCSA SOCSA**

**ACKNOWLEDGEMENT**

I am greatly thankful to **Dr. Suraj Malik, [**Dean] School of Computer Science & Applications IIMT University. Meerut, U.P. India for providing necessary infrastructure to carry out my project work at the university. I am express my sincere thanks and gratitude to **Ms. Rachna Chaudhary**, [Head of Department] School of Computer Science & Applications IIMT University. Meerut, U.P. India for her moral support. valuable guidance and encouragement during the various stages of my work.

I am feeling oblige in taking the opportunity to sincerely thanks to **Rakesh Joshi**, School of Computer Science & Applications IIMT University. Meerut, U.P. India for his valuable guidance, valuable advice and whole heartedly co-operation. His sincerity, thoroughness, timely help, and constructive criticism has been a constant source of inspiration for me.

I would also like to acknowledge my parents and friends for the whole hearted moral support and unending encouragement they provided me during my project work.

**Date: 06-06-2025 Siddhartha Raghuvanshi**

**Place: Meerut Roll No: 2366003038**

# What Inspired Me to Create This Project

My frequent use of Airbnb left a lasting impression on me. The platform’s intuitive design, seamless user experience, and its ability to connect millions of people to unique accommodations around the globe fascinated me. It’s not just a service—it’s a well-thought-out ecosystem that combines design, technology, and user psychology in a way that feels effortless. Every interaction—from searching for properties to booking a stay—is smooth and engaging, and that piqued my curiosity.

As a developer, I found myself increasingly interested in the underlying architecture and technical decisions that make a platform like Airbnb function so efficiently. I began wondering: how does Airbnb manage such a vast amount of data in real-time? How does it maintain responsiveness across different devices? How do its systems handle user authentication, listings, payments, and recommendations with such reliability?

This curiosity turned into motivation. I decided to embark on a journey to build a similar application from scratch—not to replicate Airbnb exactly, but to use it as a benchmark for learning. I saw it as an opportunity to challenge myself, deepen my understanding of full-stack development, and explore new tools and frameworks. Through this project, I aimed to implement modern development practices, improve my problem-solving skills, and gain hands-on experience with technologies like React, Node.js, databases, and user authentication systems.

Ultimately, this project is both a tribute to a product that inspired me and a personal stepping stone toward becoming a more skilled and versatile developer.

# Can you give an overview of your project?

My project is an Airbnb-inspired web application, built using **MongoDB**, **Express.js**, and **Node.js**. MongoDB handles the database, Express.js is used for backend routing, Node.js as the runtime environment, and EJS (Embedded JavaScript) is used for the frontend.

Passport.js manages user authentication, while Mapbox provides the mapping features for location-based searches.

It allows users to view, search, and filter property listings, as well as manage user accounts and reviews. Users can create listings, upload images, and manage reviews. The project is deployed using **Render** with **MongoDB Atlas** handling the database, ensuring scalability and cloud-based performance.

The application follows a **Model-View-Controller (MVC)** pattern:

* **Model**: Mongoose is used for defining data models like users, listings, and reviews.
* **View**: EJS templates render dynamic content.
* **Controller**: Controllers handle logic for various routes, such as listings and reviews.

# How Does My Platform Differ from Other Similar Services like Airbnb?

While my platform shares some foundational features with well-known services like Airbnb—such as listing properties, user accounts, and location-based search—it sets itself apart through a more focused approach to user experience, security, and customizability. One of my primary goals was to go beyond replication and instead introduce meaningful enhancements that contribute to a smoother, safer, and more engaging user experience.

To begin with, I placed strong emphasis on designing a clean and intuitive user interface. Navigation is streamlined to ensure users can effortlessly browse, filter, and discover properties that meet their preferences. Whether users are looking for a quick weekend getaway or a long-term rental, the layout and design make it easy to access relevant information without unnecessary complexity or clutter.

From a technical perspective, one of the standout features is the implementation of Google OAuth for user authentication. This addition simplifies the login process and provides a familiar, secure way for users to sign in without the need to remember another set of credentials. It adds an extra layer of convenience and trust, aligning with modern web standards for user-friendly authentication.

In terms of enhancing location-based features, I opted to integrate Mapbox for an interactive map experience. Unlike traditional static map views, Mapbox offers dynamic, real-time map rendering that allows users to explore listings geographically with ease.

This feature makes the process of searching for properties in specific neighborhoods or near landmarks far more engaging and effective.

Security has also been a core focus in my application. I implemented encryption and secure data handling practices to ensure that sensitive user data is protected throughout the platform. By prioritizing these measures, I aimed to make users feel safe while using

the service—especially when dealing with personal information, bookings, and communication.

Overall, while the inspiration for this platform came from established giants like Airbnb, my version is a reflection of thoughtful improvements tailored toward enhancing usability, maintaining robust security, and delivering a refined and trustworthy user experience.

# Who Is the Target Audience for This Platform?

The platform is designed to serve two primary groups of users: property owners and travelers seeking short-term accommodations. Each group benefits from specific features tailored to their unique needs, making the platform a versatile and practical solution in the online rental space.

For property owners, the platform provides a straightforward and effective way to list their available spaces for rent. These may include private homes, apartments, guesthouses, or even individual rooms. Many homeowners or landlords are looking for a reliable method to generate passive income by renting out unused or spare spaces. This platform empowers them to do just that by offering an easy-to-use interface where they can create detailed listings, upload high-quality photos, set pricing and availability, and manage bookings—all from a centralized dashboard. Additionally, built-in security and verification features provide peace of mind by ensuring that only legitimate users can interact with listings.

On the other side, the platform caters to travelers—ranging from tourists and backpackers to business professionals and digital nomads—who are looking for temporary places to stay. These users typically seek accommodations that are affordable, comfortable, and conveniently located. The platform supports their needs by offering robust search and filtering options, allowing them to find suitable properties based on location, price, amenities, and more. A key feature that enhances this experience is the integration of an interactive map powered by Mapbox, which lets users visually explore listings in specific neighborhoods or proximity to landmarks, transit hubs, and attractions.

Moreover, the platform places a strong emphasis on simplicity and security. Many users, whether hosts or guests, prefer a streamlined experience that doesn’t overwhelm them with unnecessary complexity. The intuitive interface, combined with secure login options such as Google authentication, ensures that both browsing and booking are smooth and

trustworthy processes. This makes the platform especially appealing to individuals who value ease of use and data privacy.

In essence, this platform is ideal for anyone interested in either renting out a space or finding one for short-term use. It brings together hosts and guests in a secure,

user-friendly, and visually engaging environment, designed to meet the practical demands of both sides.

# Why did you choose MongoDB as your database?

I chose MongoDB over SQL because it offers more flexibility when handling different types of data. For example, in a rental platform, each property listing can have various attributes, such as price, location, amenities, and even unique features like

pet-friendliness or swimming pool availability.

With MongoDB, I can easily store these properties in a more dynamic way. If I want to add a new feature to a property listing, like a "smart home" option, I can simply add it to the document without changing the entire database structure.

In contrast, with a traditional SQL database, I would need to alter the table schema, which can be time-consuming and complicated. This flexibility makes MongoDB a better fit for the dynamic nature of my application.

# How did you ensure data consistency and reliability in your MongoDB database?

I made sure the data is consistent and reliable by designing the MongoDB schema with Mongoose, which helps keep the data organized and maintains the relationships between documents. I also used Joi to check that the data sent to the database follows specific rules, which reduces the chances of invalid or incomplete entries.

# Schema Design with Mongoose

I utilized Mongoose to define clear schemas for my collections, which helps enforce a consistent structure across documents. Each schema defines the types of data expected, and this reduces the likelihood of invalid data being stored.

In this example, the **price** field is marked as **required** and has a **min** value of 0, ensuring that negative prices cannot be stored, thus maintaining data integrity.

**Relationships Between Documents**: MongoDB doesn't have traditional foreign key constraints like relational databases, but Mongoose allows you to establish relationships between documents using references. For instance, a property listing might reference a user document (the host), ensuring that each property is associated with an existing user. This enforces a level of consistency and relational integrity.

# Input Validation with Joi

I used **Joi**, a schema description language and validator for JavaScript, to validate incoming data for listings, reviews, and user information. This ensures that all form submissions meet the required criteria before they are processed and saved to the database.

**Joi** is used to define validation schemas for incoming request data.

Middleware functions validate the data against these schemas and handle errors accordingly.

Successful validations allow the request to proceed, while failures provide feedback to the user through error messages.

This structure promotes cleaner code and better error handling in your Express.js applications.

# By combining a well-defined Mongoose schema with Joi validation, you effectively reduce the risk of:

* **Missing Fields**: Mongoose enforces required fields and correct data types, throwing errors for any discrepancies, ensuring that all necessary data is present.
* **Improper Data**: Joi validates input data before it reaches the database, catching issues like negative prices, thereby preventing invalid entries.
* **Incomplete Entries**: Together, Joi and Mongoose help ensure that only complete and consistent data is stored, minimizing the potential for bugs and reducing the need for manual data cleanup in production environments.

# Can you explain how Express.js handles requests and responses in your project?

**Express.js** is used as the main backend framework to handle HTTP requests, route them to the appropriate controllers, process user input, and send back the appropriate responses.

# Example: Handling the Creation of a New Listing

In this example, we will walk through the flow of a **POST request** to create a new listing on your platform. This covers routing, middleware, controller logic, and response handling.

# Routing the Request

When a user submits a form to create a new listing, the **POST request** is sent to the

/listings route. The router file listing.js defines this route and the actions that should happen when the request is received:

router.route('/')

.post(isLoggedIn, upload.single('listing[image]'), validateListing, wrapAsync(listingController.createListing));

* + **POST /listings**: This is the route that handles listing creation.

# Middleware functions:

* + - isLoggedIn: Ensures that only authenticated users can create a listing.
    - upload.single('listing[image]'): Handles the image file upload using Multer and Cloudinary.
    - validateListing: Validates the listing data (title, price, etc.) before it is saved.
  + **Controller**: The createListing method from the listingController processes the request.

# Middleware Flow

Before the request reaches the controller, middleware functions are executed. Each middleware performs specific tasks in the request lifecycle.

* + **isLoggedIn** Middleware:

module.exports.isLoggedIn = (req, res, next) => {

if (!req.isAuthenticated()) {

req.flash('error', 'You must be signed in first!');

return res.redirect('/login');

}

next(); // Move to the next middleware if the user is authenticated

};

o This middleware checks if the user is logged in using req.isAuthenticated(). If not, the user is redirected to the login page. If authenticated, the request moves to the next middleware.

* + **upload.single('listing[image]')** Middleware:

const multer = require('multer');

const { storage } = require('../cloudConfig'); const upload = multer({ storage });

// Middleware in the route handles the file upload

* + - This middleware processes the image file uploaded by the user, storing it in Cloudinary. It makes the image URL available in req.file.
  + **validateListing** Middleware:

const { listingSchema } = require('../schemas');

const ExpressError = require('../utils/ExpressError');

module.exports.validateListing = (req, res, next) => { const { error } = listingSchema.validate(req.body);

if (error) {

const msg = error.details.map(el => el.message).join(',');

throw new ExpressError(msg, 400);

} else {

next();

}

};

This middleware validates the submitted listing data (title, description, price) using **Joi**. If the data is invalid, an error is thrown; otherwise, the request proceeds to the controller.

1. **Controller Logic**

Once the request passes through all the middleware, it reaches the controller where the listing is actually created and saved to the database.

# Controller Method (createListing in listingController.js):

module.exports.createListing = async (req, res) => {

const listing = new Listing(req.body.listing); // Create a new listing from the form data

listing.owner = req.user.\_id; // Set the current user as the owner of the listing

listing.image = { url: req.file.path, filename: req.file.filename }; // Save the image from Cloudinary

await listing.save(); // Save the listing to the database req.flash('success', 'Successfully created a new listing!');

res.redirect(`/listings/${listing.\_id}`); // Redirect to the newly created listing

};

* **Creating a Listing**: The listing data (req.body.listing) submitted by the user is used to create a new **Listing** document in MongoDB.
* **Setting the Owner**: The current logged-in user (stored in req.user.\_id) is assigned as the owner of the listing.
* **Handling Image**: The uploaded image (processed by Multer and Cloudinary) is saved to the listing.image field with its URL and filename.
* **Saving to the Database**: The await listing.save() command saves the new listing to MongoDB.
* **Response**: After successfully saving the listing, a success message is flashed, and the user is redirected to the newly created listing's detail page using res.redirect().

1. **Response Handling**

After the listing is created, the server sends an appropriate response back to the client. In this case, the response is an HTTP redirect to the new listing's detail page.

res.redirect(`/listings/${listing.\_id}`);

* + **Redirection**: The res.redirect() function sends a response that tells the browser to navigate to a new page (in this case, the URL of the newly created listing).
  + **Flash Messages**: Before redirecting, a flash message is sent to inform the user that the listing was successfully created.

In summary, Express.js handles the request for creating a new listing by:

* + Routing the request to the correct controller.
  + Using middleware to ensure the user is authenticated, validate the data, and process file uploads.
  + Passing control to the controller to create the listing and save it in MongoDB.
  + Sending a response back to the client, redirecting them to the new listing's detail page.

# Why Did I Use Node.js for Backend Development?

Choosing Node.js for backend development was a strategic decision based on its powerful features, efficiency, and ability to streamline the development process. Below are the key reasons why Node.js was the ideal choice for this project:

# JavaScript Everywhere – A Unified Language Across the Stack

One of the most compelling advantages of Node.js is that it allows developers to use JavaScript on both the client side (frontend) and the server side (backend). This unified language environment significantly simplifies the development workflow. By writing both client and server logic in the same language, there is less context-switching for developers, which leads to faster development cycles and easier debugging. It also makes it more feasible for a single developer or a small team to manage the entire stack efficiently, which was particularly useful for me while working independently on this project.

# Asynchronous and Event-Driven – Efficient Non-Blocking I/O

Node.js is built on an event-driven architecture and supports non-blocking I/O operations by default. This means it can handle thousands of concurrent requests without getting bogged down. Unlike traditional server models that create a new thread for every request, Node.js uses a single-threaded event loop to handle multiple requests asynchronously.

This makes it extremely lightweight and efficient, especially for web applications that require real-time interactions—such as live data updates, chat systems, or user interactions happening simultaneously. In my case, this approach helped ensure the platform could scale effectively and remain responsive under load.

# Rich Ecosystem – NPM (Node Package Manager)

Node.js comes with a massive ecosystem of open-source libraries and modules available through NPM (Node Package Manager). This ecosystem provides ready-to-use packages for virtually any functionality—ranging from database integration and authentication to form validation, payment gateways, and beyond. This allowed me to rapidly prototype and implement features without having to build everything from scratch. For example, libraries like Passport.js for authentication and Mongoose for MongoDB interaction were seamlessly integrated into the project, accelerating development while maintaining code quality.

# Strong Community and Extensive Support

Another major reason for choosing Node.js is its highly active and supportive community. The technology has widespread adoption and is backed by continuous contributions from developers around the world. Whether I needed documentation, troubleshooting advice, or tutorials, the community resources available for Node.js were incredibly helpful. This strong community support reduced development friction and helped me implement best practices with confidence.

How did you implement user authentication, and why did you choose Passport.js for this purpose?

I used **Passport.js** for authentication because it provides flexible, modular authentication strategies. I implemented both **local strategy** (email and password) and **OAuth2** strategies (Google and Facebook logins) using Passport.js, making it easy for users to log in with their preferred method.

user authentication was implemented using **Passport.js**, a middleware for Node.js that simplifies the authentication process.

# Why Choose Passport.js for User Authentication

1. **Versatility**:
   * Passport.js supports a wide range of authentication strategies (local, social, JWT, etc.), making it easy to implement various authentication methods as needed in your application.

# Simplicity:

* + Its middleware approach simplifies the integration of authentication into your existing Express.js application, allowing you to focus on building your features without dealing with complex authentication logic.

# Session Management:

* + Passport.js seamlessly integrates with Express session management, making it easier to maintain user sessions across requests.

# Community Support:

* + Being widely used in the Node.js community, Passport.js has extensive documentation and a large number of tutorials and resources, facilitating easier troubleshooting and implementation.

# Security:

* + Passport.js handles secure password hashing and user session management, allowing you to focus on implementing business logic without worrying about the security complexities of authentication.

# How do you ensure that only authorized users can modify or delete listings and reviews?

**Answer:** I implemented role-based authorization using middleware. For listings, the isOwner middleware ensures that only the owner of a listing can modify or delete it. For reviews, the isReviewAuthor middleware checks if the current user is the one who created the review before allowing them to delete it.

module.exports.isOwner = async (req, res, next) => {

const { id } = req.params;

const listing = await Listing.findById(id);

if (!listing.owner.equals(req.user.\_id)) {

return res.redirect(`/listings/${id}`);

}

next();

};

# How do you handle environment variables securely using dotenv? Explanation of Handling Environment Variables Securely:

1. **Using dotenv:** I use the **dotenv** package to manage environment variables, which are

stored in a .env file and never committed to version control. This ensures that sensitive information, such as database connection strings and API keys, remains secure.

* include dotenv at the top of your application to load these variables:

require('dotenv').config();

All environment variables from your .env file (like CLOUD\_API\_KEY, MAP\_TOKEN, ATLASDB\_URL, etc.) can then be accessed securely in your code using process.env:

By keeping these keys in the .env file, they are not hard-coded in your application, making the project easier to manage and more secure.

**Don’t Share Your .env File**: Ensure the .env file is included in your .gitignore file so it’s not pushed to your version control system (like GitHub).

This keeps your sensitive data private and prevents accidental leaks.

# How Does the Login System Work and How Is User Data Secured During Authentication?

The login system in my application is designed to be both user-friendly and highly secure, accommodating multiple authentication methods while ensuring that sensitive user data is protected at every step. To achieve this, I’ve implemented Passport.js—a flexible and widely used authentication middleware for Node.js—which supports both local login and Google OAuth 2.0 authentication.

# Local Authentication with Secure Password Handling

For users who prefer to sign up and log in using their email and password, the application utilizes Passport’s local strategy. During the registration process, the user’s password is never stored in plain text. Instead, it is hashed using bcrypt, a popular and secure hashing algorithm that adds a salt to the password and generates a cryptographically strong hash. This ensures that even if someone were to gain access to the database, they would not be able to retrieve the original passwords.

When users attempt to log in, the entered password is hashed and compared with the stored hash in the database using bcrypt’s comparison function. This method guarantees that password verification is secure and resistant to brute-force attacks.

# Google OAuth 2.0 Authentication for Seamless Sign-In

To provide users with a more convenient and modern authentication option, the platform also supports Google login using OAuth 2.0. This allows users to log in with their existing Google accounts without needing to create a separate username and password for the platform. When users choose to log in with Google, they are redirected to Google’s authentication page. Upon successful verification, Google returns a secure OAuth token to the application.

This token contains essential information, such as the user’s Google ID and email, which the app uses to either log in an existing user or register a new user automatically. Since all token handling occurs over HTTPS and is handled by a trusted third-party provider (Google), this approach adds an extra layer of trust and convenience.

# Secure Session Management with MongoDB

After successful authentication—whether through local login or Google—the application uses session-based authentication to maintain the user’s login state. Sessions are managed using connect-mongo, a middleware that stores session data directly in MongoDB rather than in memory. This approach ensures scalability and persistence, especially

important in production environments where memory-based session storage can be unreliable.

By storing sessions in MongoDB, the platform ensures that even if the server restarts, the user’s session remains valid until it naturally expires or is manually destroyed.

Additionally, cookies are used to store session identifiers on the client-side, and these cookies are protected using secure flags and expiration settings.

# Overall Security Considerations

The entire authentication flow is protected using best practices such as HTTPS communication, encrypted session data, and secure cookie handling. User input is sanitized and validated to protect against common vulnerabilities such as injection attacks and session hijacking.

In summary, the login system is designed to be robust, secure, and flexible—allowing users to choose their preferred authentication method while ensuring that their credentials and personal data are always handled with the highest level of security.

# What strategies did you use to secure user data, especially passwords?

Step-by-Step Process for Local Authentication:

# User Registration:

* + - When a new user signs up, their password is hashed using **bcrypt** before being stored in the database.
    - . **passportLocalMongoose**: This plugin automatically hashes the password using bcrypt before saving the user record in MongoDB. It also adds methods to the user schema for easy user authentication.**Step 3: Session Management**
    - During registration, passwords are never stored in plaintext. The **bcrypt hashing** ensures that even if someone accesses the database, they cannot retrieve the original password.

# Local Login Process:

* + - When a user logs in, Passport.js compares the hashed version of the entered password with the stored hashed password in the database.
    - **passport.authenticate('local')**: This middleware handles the local authentication logic using the passport-local strategy. It automatically compares the user’s entered password (hashed) with the stored password hash in the database.
    - If authentication is successful, the user is logged in, and req.user is populated with the user’s information. If it fails, the user is redirected to the login page with an error message (flash message).

# Handling Sessions:

* + - After successful login, the user’s session is established, and the session data is stored in MongoDB using **connect-mongo**.
    - This ensures that session data is securely stored in the database, and cookies are used to maintain the session between requests. The httpOnly flag on cookies ensures they are not accessible via client-side JavaScript.

# Google OAuth 2.0 Authentication (Using Passport.js)

The project also allows users to log in using their Google accounts via **Google OAuth2.0**. This method allows users to authenticate without needing to create a separate account or password for this platform.

# Step 1: Redirect to Google

* When the user chooses to log in with Google:
  + Passport.js (with Google OAuth 2.0 strategy) redirects the user to Google’s login page.
  + The user logs in to their Google account.

# Step 2: Google Sends OAuth Token

* After the user successfully logs into their Google account, Google generates an **OAuth token** and sends it back to your application using a callback URL (configured in the .env file).

# Step 3: Verification and User Creation

* Passport.js uses the token to authenticate the user:
  + It checks if the user already exists in the database (by checking the Google profile ID).
  + If the user exists, the login is successful.
  + If the user does not exist, a new user is created in the database with the Google profile information (e.g., name, email, profile picture).

# Step 4: Session Management

* After successful authentication, Passport.js creates a session for the user, similar to the local authentication process.

# How is User Data Secured?

Passwords are hashed using bcrypt (local strategy), OAuth2 tokens are handled by Google, and sessions are stored securely with cookies set to be httpOnly.

**How did you implement Google OAuth2.0, and why did you choose this method of authentication?**

**How I Implemented Google OAuth2.0**

1. **Setting Up Passport.js**:
   * I used the **Passport.js** library to handle authentication in the application. Passport is a popular middleware for Node.js that provides different strategies for authentication, including OAuth.

# Installing the Necessary Packages:

* + I installed the required packages:

npm install passport passport-google-oauth20

passport-google-oauth20 is the strategy that specifically handles Google OAuth2.0 authentication.

# Creating a Google OAuth App:

* I went to the **Google Developer Console** and created a project.
* From there, I obtained the **Google Client ID** and **Google Client Secret**, which are necessary for OAuth authentication.
* I also set up the **redirect URL** in the console, which is the URL where Google will send the user after they've authenticated.

# Configuring the Passport Google OAuth20 Strategy:

* In the application, I configured the Google OAuth strategy like this

# Handling User Sessions:

* Passport handles user sessions using serializeUser and deserializeUser methods:

# Creating Routes for Authentication:

* Routes were set up to handle login and callback from Google

**Why I Choose Google OAuth2.0**

1. **Improved User Experience**:
   * **Convenience**:
   * **Reduced Friction**: The process of signing in is quicker and easier, encouraging more users to engage with the application.

# Enhanced Security:

* + **Passwordless Login**: There’s no need to store or manage passwords, decreasing the risk of breaches associated with weak or reused passwords.
  + **Token-based Authentication**: The use of short-lived access tokens limits exposure; even if a token is compromised, it expires quickly.

# Reputation and Trust:

* + Users generally trust Google, making them more comfortable logging in through a familiar service rather than creating new credentials for each application.

# Social and Data Integration:

Overall, Google OAuth2.0 provides a secure, user-friendly, and efficient way for users to log in, improving engagement and trust in the application.

The user clicks "Login with Google" → Redirect to Google for authentication → User grants permissions → Google redirects back to your app → Passport handles the response and checks/create user in the database → User data is serialized into the session → User is redirected to the home page.

# How do you manage session data? Why did you use Connect Mongo for session storage?

I manage session data using **express-session** and store it in MongoDB using

**connect-mongo**. This ensures that session data is persistent and scalable, allowing the app to handle multiple users.**Summary of the Process**

# User Logs In:

* + The user provides their login info (like a username and password).
  + If they log in successfully, the app stores just their unique ID in the session (thanks to serializeUser).

# Session Storage:

* + The session (which holds the ID) is saved somewhere (like a database or in memory).

# On Subsequent Requests:

* + Whenever the user does something (like going to another page), the app uses deserializeUser to look up the full user details using the ID stored in the session.
  + It then attaches this information to the request so that it can be accessed easily.

# Why This Matters

* **Security**: By only storing the user ID, sensitive information (like passwords) isn’t kept in the session, making it safer.
* **Efficiency**: The app can quickly fetch user details from the database as needed, allowing for a personalized experience without storing a lot of data in the session itself.

# Why did you use Connect Mongo for session storage:

1. Persistence:Connect Mongo stores session data in a MongoDB database, ensuring that session information persists even if the application restarts.
2. **Optimized for Read/Write:** MongoDB is designed for fast data retrieval and can efficiently handle session data read/write operations. Connect Mongo leverages MongoDB's performance characteristics for session management.

# Session Expiration

* + Automatic Management: Connect Mongo allows you to easily manage session expiration settings. You can specify how long sessions should last, and the library will automatically handle session expiration in the database.

# Compatibility with Express.js

* + Middleware Integration: Connect Mongo integrates seamlessly with Express.js, making it easy to use in applications that already utilize Express session middleware.

# Easy to Implement

Minimal Configuration: Setting up Connect Mongo is straightforward. You only need to configure it with your MongoDB connection details, and it will handle session storage for you.

**What measures did you take to ensure secure handling of cookies and sessions?**

1. **Using Secure Cookie Settings**
   * **HTTP-Only Cookies**:

Setting the httpOnly flag on cookies means that they cannot be accessed via JavaScript in the browser. This helps protect against cross-site scripting (XSS) attacks, where an attacker tries to steal cookies through malicious scripts.

# Using a Secure Session Store

using connect-mongo to store sessions in MongoDB, which adds an extra layer of security. The crypto.secret option is used to encrypt the session data, making it more secure against interception. This means that even if someone accesses the session store, they would find it difficult to decrypt and misuse the stored data..

# Secure Cookie Expiration Settings

Setting a reasonable expiration for cookies helps ensure that even if a cookie is compromised, it won't remain valid indefinitely. The cookie will expire after 7 days, reducing the window of opportunity for misuse.

# Why Did I Choose EJS for Dynamic Content Rendering?

When developing the server-side rendering portion of the application, I chose EJS (Embedded JavaScript) as the templating engine due to its simplicity, flexibility, and strong integration with Express.js. EJS is a lightweight and powerful tool that allows developers to embed JavaScript code directly into HTML, making it ideal for creating dynamic web pages based on real-time data.

# Seamless Integration with Express.js

One of the key reasons for choosing EJS was its tight compatibility with Express.js, the web framework I used for building the backend. EJS fits naturally into Express’s rendering pipeline, allowing for a smooth and straightforward setup. With minimal configuration, I was able to define EJS as the view engine and begin rendering templates immediately. This seamless integration significantly sped up development and reduced boilerplate code.

# Server-Side Rendering of Dynamic Data

EJS enables server-side rendering (SSR) of dynamic content, which is particularly valuable for applications where data is frequently updated or fetched from a database. For instance, in my application, property listings, user reviews, and account details are all retrieved from MongoDB and dynamically injected into EJS templates. This approach ensures that users always see up-to-date content without the need for complex frontend frameworks or client-side rendering.

# Simplicity and Readability

EJS templates closely resemble standard HTML files, with the added ability to embed JavaScript using <% %> tags. This makes EJS intuitive to use, especially for developers who are already familiar with HTML and JavaScript. The learning curve is minimal, and the code remains clean and maintainable. I found this especially helpful when creating templates for listing pages, forms, user dashboards, and review sections, where embedding loops and conditional logic directly in the HTML structure was both convenient and readable.

# Flexibility in Template Design

Another major advantage of EJS is its flexibility. It allows for the use of partials and layouts, which helps in maintaining DRY (Don’t Repeat Yourself) principles. For example, I created reusable partials for headers, footers, and navigation bars, which made it easy to apply consistent design and functionality across all pages. Layouts helped

maintain a uniform structure while allowing individual pages to insert unique content where needed.

# Performance and Server Efficiency

Because EJS renders templates on the server before sending them to the client, it reduces the amount of processing required on the client side. This can improve load times for users, especially on slower devices or networks. The server handles logic and data binding, delivering fully rendered HTML pages to the browser.

In summary, EJS provided a practical, elegant, and developer-friendly way to render dynamic content on the server side. Its simplicity, compatibility with Express.js, and ability to inject live data into views made it the ideal choice for this project. It allowed me to focus on building features and refining the user experience without the overhead of a heavy frontend framework.

# How Do You Ensure the User Interface Is Intuitive and Easy to Use?

Designing a user interface (UI) that is both intuitive and easy to use was one of my primary objectives while developing the platform. A seamless and accessible user experience not only enhances user satisfaction but also encourages engagement and repeat visits. To achieve this, I employed a combination of modern design principles, responsive frameworks, and interactive tools.

# Clean Layout Using HTML5 and CSS3

I began by structuring the application using semantic HTML5, which improves both accessibility and maintainability. Semantic tags like <section>, <article>, and

<nav> make the layout more understandable for screen readers and assistive technologies. Paired with CSS3, this allowed me to implement visual consistency and responsive design across all pages. I paid close attention to spacing, alignment, font hierarchy, and color contrast to ensure the interface is visually appealing and easy to navigate.

# Integration of Bootstrap for Consistency and Responsiveness

To streamline development and maintain a consistent design language, I incorporated Bootstrap—a widely-used frontend framework. Bootstrap provided a grid system that made it easier to structure layouts and ensure they adapt gracefully across different screen sizes and devices. Its prebuilt components, such as navigation bars, modals, buttons, and forms, helped maintain a unified look and feel throughout the platform while speeding up development time. This also ensured that users could enjoy a polished and professional interface, whether they were accessing the application from a desktop, tablet, or smartphone.

# Interactive Location-Based Search with Mapbox

To enhance usability and provide a modern feature that aligns with real-world user needs, I integrated Mapbox for interactive map functionality. Users can view and explore available property listings directly on a map, helping them to understand the geographical context and proximity to points of interest like transit stations, parks, or tourist attractions. This visual, location-based search is far more intuitive than text-based location filters alone, especially for users unfamiliar with a city or region. The dynamic pins and real-time interaction offered by Mapbox significantly contribute to a more engaging and user-centric experience.

# How Did You Integrate Mapbox into the Frontend for Location-Based Searches?

To provide a modern and user-friendly way for users to search for properties based on location, I integrated **Mapbox** into the frontend of my application. Mapbox offers powerful tools for interactive mapping, which allowed me to implement an engaging visual experience for users exploring listings geographically.

# Interactive Mapping with Mapbox GL JS

At the core of the integration is **Mapbox GL JS**, a JavaScript library specifically designed for building highly customizable and interactive maps in web applications. This library enabled me to create maps that respond to user interactions like zooming, panning, and dragging. I used it to visually display the exact location of each property listing, helping users get a real sense of where a property is situated in relation to the surrounding area.

Using Mapbox’s styling options, I was able to design maps that are both visually appealing and functionally effective. For instance, I used custom map styles that match the aesthetic of the platform, and added interactive markers so users can click on a location to view listing details.

# Location Search with Mapbox Geocoder Plugin

To make the map more functional, I integrated the **Mapbox Geocoder plugin**. This tool adds a search box to the map interface, allowing users to search for specific cities, addresses, or places. When a user types in a location, the plugin uses Mapbox’s geocoding service to suggest accurate and relevant results in real time.

Once the user selects a location from the search suggestions, the map automatically updates to center on that area. This makes it extremely convenient for users to explore listings in specific neighborhoods or regions without needing to manually scroll or pan the map.

# Enhanced User Experience

Combining these tools, the result is a much more intuitive and engaging way to find accommodations. Instead of relying solely on dropdowns or text-based filters, users can visually interact with the map to find properties near desired landmarks or in specific geographic zones. This is particularly useful for travelers unfamiliar with a location, as they can visually understand proximity to transit, attractions, or business districts.

# APIs Used and Performance Considerations

To support this functionality, I utilized two key Mapbox services:

* + The **Mapbox GL JS API**, which powers the rendering of maps in the browser.
  + The **Mapbox Geocoding API**, which processes location searches and returns geographic coordinates.

I also ensured optimal performance and security by properly configuring API access, keeping tokens secure, and loading map features efficiently to maintain a smooth user experience.

# What tools or techniques did you use to ensure responsiveness across different devices?

I used **Bootstrap** for responsive design, ensuring that the website adapts well to different screen sizes.

* + Media Queries: Different styles are applied based on the screen width.
  + Flexbox Layout:The use of Flexbox allows elements to wrap and adjust their alignment and spacing automatically, making the layout responsive.
  + Flexible Widths and Heights:Setting widths to 100% allows elements to scale based on their parent container, ensuring they fit on various devices.
  + Responsive Font Sizes:Using rem units for font sizes ensures that text scales proportionally with the user's default font size settings.
  + Hide/Show Elements with Media Queries:Elements are hidden on smaller screens to improve usability and streamline the interface.

How did you implement CRUD operations for property listings?

CRUD operations (Create, Read, Update, Delete) for property listings are handled via **Mongoose** in the backend. For example, when a user submits a form to create a listing, the data is validated using **Joi**, saved to MongoDB, and then displayed on the website

# Create Listing

* + **Method:** createListing
  + **Description:** This function creates a new property listing.

# Process:

* + - It uses Mapbox's geocoding API to get the geographical coordinates for the listing's location.
    - It constructs a new Listing object using the data received from the request body and assigns the current user as the owner.
    - The listing image and geolocation data are saved to the database.
    - Finally, a success message is flashed to the user, and they are redirected to the listings page.

# Read Listings

* + **Method:** index and showListing
  + **Description:** These methods retrieve and display property listings.

# Process:

* + - index fetches all listings from the database, sorts them, and renders the listings page.
    - showListing retrieves a single listing based on its ID, populates the reviews and owner details, and renders the specific listing page.

# Update Listing

* + **Method:** updateListing
  + **Description:** This function updates an existing property listing.

# Process:

* + - It retrieves the listing using its ID, updates its details, and re-fetches the geolocation data.
    - If a new image file is provided, it updates the image field as well.
    - After updating, a success message is flash

# Delete Listing

* + **Method:** destroyListing
  + **Description:** This function deletes a property listing.

# Process:

* + - It finds the listing by ID and deletes it from the database.
    - A deletion confirmation message is flashed, and the user is redirected back to the listings page.

.How does the review system work? Can users edit or delete their reviews?

The review system allows users to create, manage, and interact with reviews associated with listings

# Creating a Review:

* + When a user wants to submit a review, they make a POST request to the relevant route (e.g., /listings/:id/reviews).
  + The request must include the review data, and the user must be logged in.
  + Middleware functions validate the review and ensure the user is logged in.
  + If all checks pass, the createReview function is called, which:
    - Fetches the listing by ID.
    - Creates a new review and associates it with the listing and the user.
    - Saves the review to the database and updates the listing to include the new review.
    - Redirects the user back to the listing page with a success message.

# Deleting a Review:

* + Users can delete their reviews by making a DELETE request to the route (e.g.,

/listings/:id/reviews/:reviewId).

* + The user must be logged in and must be the author of the review they wish to delete.
  + Middleware functions check the user's authentication and ownership of the review.
  + If all checks pass, the destroyReview function is called, which:
    - Removes the review from the listing and deletes the review from the database.
    - Redirects the user back to the listing page with an error message indicating the review has been deleted.

**Users can create and delete reviews** in your application.**There is no option for users to edit their reviews** at this time.

What was the most challenging feature to implement, and how did you overcome that challenge?

# What is OAuth 2.0?

OAuth 2.0 is a protocol that allows third-party applications (like your app) to access user data from services like Google without the user having to share their password directly with your app. Instead, the user authorizes Google to give specific data to your app.

# What is Passport.js?

Passport.js is an authentication middleware for Node.js that makes it easy to implement different login strategies, including Google OAuth 2.0.

# "The most challenging feature I implemented in my application was Google authentication. Initially, I didn't have a clear understanding of how it worked, but I

**I wanted to ,**

**Improved User Experience:** The feature would simplify the onboarding process for users.

* 1. **Faster Registration:** It allows users to register quickly without filling out lengthy forms.
  2. **Easy Login:** Users can log in seamlessly using their Google accounts.
  3. **Higher Conversion Rates:** By reducing the barriers to sign-up, I anticipated a decrease in user drop-off rates.
  4. **Accurate User Data:** Authenticating through Google would help ensure that the data collected is valid and reliable.

I decided to learn everything from scratch. I thoroughly studied the official Google Developers documentation for OAuth 2.0, which provided detailed guidelines on implementing the authentication.As I worked through the implementation, I encountered several challenges, particularly with token management and user session handling. At this stage, I reached out to my project guide. By combining self-study, documentation review, and mentorship, I successfully integrated Google authentication into my application.

Ultimately, this feature not only improved the overall user experience but also enhanced the app's reliability and security."

# User clicks "Login with Google"

* + The user initiates login by clicking the "Login with Google" button on your app.

# App sends request to Google

* + Your app sends a request to Google’s OAuth 2.0 server with:
    - Client ID (your app’s unique ID)
    - Redirect URI (where Google should send the user after login)
    - Scope (what data your app is requesting, like email and profile)

# Google prompts user for consent

* + Google asks the user to log in (if not already logged in) and confirm that they want to share their data with your app.

# Google sends an authorization code to your app

* + After the user consents, Google sends a temporary **authorization code** to your app’s **Redirect URI**.

# App exchanges authorization code for an access token

* + Your app sends the **authorization code** back to Google, along with your **Client ID**

and **Client Secret**.

* + Google responds with an **access token**.

# App requests user data from Google

* + Your app sends the **access token** to Google’s API to request user data (like email and profile information).

# Google sends user data

* + Google responds with the requested user data (like Google ID, name, email, etc.).

# App processes user data

* + Your app checks if the user already exists in your database:
    - If yes, the user is logged in.
    - If no, a new account is created, and the user is logged in.

**Summary:**

1. **User initiates login** →
2. **App requests Google’s authorization** →
3. **Google asks for user consent** →
4. **Google sends authorization code** →
5. **App exchanges code for access token** →
6. **App requests user data using the token** →
7. **Google responds with user data** →
8. **App logs in or creates the user**.

How does your interactive map system improve the user experience?

**User-Friendly Interface**: Interactive maps provide intuitive navigation tools, allowing users to zoom in/out, pan, and click on specific areas for detailed information.

**Search Functionality**: Users can easily search for specific locations, addresses, or points of interest, making it simpler to find what they need without extensive browsing.

**Interactive Features**: Features like route planning, directions, and the ability to explore nearby points of interest keep users engaged and encourage longer sessions within the application.

# How did you ensure the platform can scale with an increasing number of users and properties?

Scalability was a paramount consideration in the architecture of the platform. To accommodate a growing user base and an expanding catalog of property listings, I implemented several strategic measures:

* **Database Scalability with MongoDB Atlas:** MongoDB Atlas offers robust horizontal scaling capabilities through sharding. This allows the database to distribute data across multiple servers, ensuring efficient handling of large volumes of data and high-throughput operations.
* **Application Deployment with Render:** Render provides automatic scaling features that adjust computing resources based on real-time traffic and resource demands. This ensures that the application can handle increased load without manual intervention.
* **Stateless Application Design:** By designing the application to be stateless, each instance can handle any request, facilitating easier scaling and load balancing.
* **Monitoring and Alerts:** Implementing monitoring tools allows for real-time tracking of system performance, enabling proactive scaling and resource allocation.

These measures collectively ensure that the platform remains responsive and reliable, even as user engagement and data volume grow.

# How does your app handle a large number of concurrent users?

Handling numerous concurrent users efficiently is critical for maintaining a seamless user experience. The application employs several techniques to manage high concurrency:

* **Non-Blocking I/O with Node.js:** Node.js utilizes an event-driven, non-blocking I/O model, allowing it to handle multiple simultaneous connections without creating new threads for each request. This architecture is particularly effective for I/O-bound operations.
* **Efficient Database Operations:** MongoDB Atlas supports concurrent connections and provides features like connection pooling and optimized query performance, ensuring that database operations do not become a bottleneck.
* **Load Balancing:** Deploying the application behind a load balancer distributes incoming traffic across multiple server instances, preventing any single server from becoming overwhelmed.
* **Caching Strategies:** Implementing caching mechanisms for frequently accessed data reduces the load on the database and speeds up response times.

These strategies work in concert to ensure that the application can serve a large user base effectively.

# Why did you choose Cloudinary for image and video storage? How does it improve the user experience?

Selecting Cloudinary for media management was driven by its comprehensive feature set and the benefits it offers to both developers and end-users:

* **Automatic Image Optimization:** Cloudinary automatically compresses and serves images in optimal formats and sizes based on the user's device and browser, enhancing load times and reducing bandwidth usage.
* **Responsive Delivery:** By delivering appropriately sized images, Cloudinary ensures that users experience fast-loading pages, which is crucial for engagement and retention.
* **Simplified Media Management:** Cloudinary's APIs and SDKs streamline the process of uploading, transforming, and managing media assets, reducing development overhead.
* **Scalability and Reliability:** As a cloud-based service, Cloudinary scales seamlessly with the application's needs and offers high availability, ensuring that media assets are always accessible.

# How did you handle errors or exceptions when working with third-party APIs like Mapbox or Cloudinary?

Robust error handling is essential when integrating third-party APIs to ensure application stability and provide meaningful feedback to users:

* **Try-Catch Blocks:** Asynchronous operations involving external APIs are wrapped in try-catch blocks to catch and handle exceptions gracefully.[LinkedIn](https://www.linkedin.com/pulse/mastering-error-handling-nodejs-applications-juan-soares-s1f2f?utm_source=chatgpt.com)
* **Centralized Error Handling Middleware:** Express.js middleware is used to centralize error handling, allowing for consistent responses and easier maintenance.
* **Logging:** Errors are logged with detailed information to facilitate debugging and monitoring.
* **User Feedback:** Utilizing flash messages, users are informed of any issues in a user-friendly manner, enhancing transparency and trust.
* **Fallback Mechanisms:** In case of API failures, the application provides fallback options or default data to maintain functionality.

# Can you explain how Mongoose is used in this project?

Mongoose serves as the Object Data Modeling (ODM) library, facilitating structured interaction with the MongoDB database:

* **Schema Definitions:** Mongoose allows for the definition of schemas that enforce data structure and validation rules, ensuring data integrity.
* **Model Relationships:** By defining references between schemas (e.g., a Review referencing a User), Mongoose enables the modeling of complex relationships and supports population of related data.
* **Middleware (Hooks):** Mongoose provides middleware functions that execute at specific stages of the data lifecycle, such as pre-save or post-delete, allowing for operations like cascading deletions.
* **Integration with Joi:** For additional validation, Joi is used alongside Mongoose to validate data before it reaches the database layer, providing a robust validation mechanism.

Mongoose's features streamline database operations and enforce consistency across the application's data models.

# If you had more time, what features or improvements would you add to this project?

Given additional time, several enhancements could be implemented to enrich the platform's functionality and user experience:

* **Review Management:** Allow users to edit and reply to reviews, fostering community engagement and dialogue.
* **Favorites System:** Enable users to bookmark or favorite listings for easy access and personalized experiences.
* **Enhanced Media Support:** Support multiple images and videos per listing to provide comprehensive visual information.
* **Calendar Integration:** Implement a calendar system for booking management and availability tracking.
* **Messaging Feature:** Introduce real-time messaging between users to streamline communication.
* **Social Logins:** Expand authentication options to include platforms like Facebook, GitHub, and Twitter for user convenience.
* **Admin Dashboard:** Develop an administrative interface for managing users, listings, and content moderation.

# Can you describe a situation where you had to make a key technical decision during development?

A pivotal technical decision involved choosing between local file storage and a cloud-based solution for handling media assets:

* **Local Storage Considerations:** While storing files locally simplifies initial development, it poses challenges in scalability, backup, and serving optimized content.
* **Cloudinary Selection:** Opting for Cloudinary addressed these concerns by offering automatic optimization, CDN delivery, and scalable storage, leading to improved performance and user experience.

This decision underscored the importance of aligning technical choices with long-term scalability and maintenance goals.

# Have you deployed this application? If so, where did you host it?

Yes, the application is deployed on Render, a cloud platform known for its simplicity and scalability:

* **Automatic Deployments:** Render integrates with GitHub, enabling automatic deployments upon code commits, streamlining the development workflow.
* **Custom Domains and SSL:** The platform supports custom domain configurations and provides SSL certificates, enhancing security and branding.
* **Scalability:** Render's autoscaling features adjust resources based on traffic, ensuring consistent performance.
* **Environment Management:** Environment variables are managed securely through Render's dashboard, keeping sensitive information out of the codebase.

# How did you handle environment variables and security configurations for deployment?

Managing environment variables securely is crucial for protecting sensitive information:

* **.env Files:** During development, environment variables are stored in a .env file, which is excluded from version control via .gitignore.
* **dotenv Package:** The dotenv package loads these variables into the application's process environment, allowing for easy access within the code.
* **Render Environment Settings:** In production, environment variables are configured directly in Render's dashboard, ensuring they remain secure and separate from the codebase.
* **Security Best Practices:** Regular audits and adherence to the principle of least privilege are employed to minimize exposure risks.

# How did you ensure the code quality of the project?

Maintaining high code quality was achieved through several best practices:

* **Modular Architecture:** The application follows the Model-View-Controller (MVC) pattern, promoting separation of concerns and easier maintenance.
* **Code Reviews:** Regular code reviews were conducted to catch issues early and encourage knowledge sharing.
* **Linting and Formatting:** Tools like ESLint and Prettier were used to enforce coding standards and consistent formatting.
* **Testing:** Unit and integration tests were written to verify functionality and prevent regressions.
* **Documentation:** Comprehensive documentation was maintained to facilitate onboarding and future development.

**Source Code**

* App.js

if (process.env.NODE\_ENV != "production") {

require("dotenv").config();

}

const express = require("express");

const app = express();

const mongoose = require("mongoose");

const path = require("path");

const methodOverride = require("method-override");

const ejsMate = require("ejs-mate");

const ExpressError = require("./utils/ExpressError");

const session = require("express-session");

const MongoStore = require("connect-mongo");

const flash = require("connect-flash");

const passport = require("passport");

const LocalStrategy = require("passport-local");

const User = require("./models/user.js");

const listingRouter = require("./routes/listing.js");

const reviewRouter = require("./routes/review.js");

const userRouter = require("./routes/user.js");

app.set("view engine", "ejs");

app.set("views", path.join(\_\_dirname, "views"));

app.use(express.urlencoded({ extended: true }));

app.use(methodOverride("\_method"));

app.use(express.static(path.join(\_\_dirname, "/public/")));

app.engine("ejs", ejsMate);

//Database Connection

// let MONGO\_URL = "mongodb://127.0.0.1:27017/GharSetu";

let MONGO\_URL = process.env.ATLASDB\_URL;

main()

.then(() => {

console.log("DB connected Sucessfully");

})

.catch((err) => console.log(err));

async function main() {

await mongoose.connect(MONGO\_URL);

}

//Mongo Session

const store = MongoStore.create({

mongoUrl: process.env.ATLASDB\_URL,

crypto: {

secret: process.env.SECRET,

},

touchAfter: 24 \* 60 \* 60,

});

store.on("error", () => {

console.log("Error in Mongo Session Store", err);

});

//Creating Sessions

const sessionOptions = {

store,

secret: process.env.SECRET,

resave: false,

saveUninitialized: true,

cookie: {

expires: 7 \* 24 \* 60 \* 60 \* 1000,

maxAge: 7 \* 24 \* 60 \* 60 \* 1000,

httpOnly: true,

},

};

//Home Route

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

res.redirect("/listings");

});

//Session and Flash

app.use(session(sessionOptions));

app.use(flash());

//Login (Authentication - Passport npm)

app.use(passport.initialize());

app.use(passport.session());

passport.use(new LocalStrategy(User.authenticate()));

passport.serializeUser(User.serializeUser()); // Login

passport.deserializeUser(User.deserializeUser()); // Logout

//Store Info for Ejs- Middleware

app.use((req, res, next) => {

res.locals.success = req.flash("success");

res.locals.error = req.flash("error");

res.locals.currUser = req.user;

next();

});

//Routes

app.use("/listings", listingRouter);

app.use("/listings/:id/reviews", reviewRouter);

app.use("/", userRouter);

//Page not found

app.all("\*", (req, res, next) => {

next(new ExpressError(404, "Page not found"));

});

//------------------Error Handler Middleware

app.use((err, req, res, next) => {

let { status = 500, message = "Something went wrong" } = err;

res.status(status).render("listings/error", { message, err });

});

//Server listen on port 8080

app.listen(8080, () => console.log("Server is listening on port 8080"));

* Listing Schema

if (process.env.NODE\_ENV != "production") {

require("dotenv").config();

}

const express = require("express");

const app = express();

const mongoose = require("mongoose");

const path = require("path");

const methodOverride = require("method-override");

const ejsMate = require("ejs-mate");

const ExpressError = require("./utils/ExpressError");

const session = require("express-session");

const MongoStore = require("connect-mongo");

const flash = require("connect-flash");

const passport = require("passport");

const LocalStrategy = require("passport-local");

const User = require("./models/user.js");

const listingRouter = require("./routes/listing.js");

const reviewRouter = require("./routes/review.js");

const userRouter = require("./routes/user.js");

app.set("view engine", "ejs");

app.set("views", path.join(\_\_dirname, "views"));

app.use(express.urlencoded({ extended: true }));

app.use(methodOverride("\_method"));

app.use(express.static(path.join(\_\_dirname, "/public/")));

app.engine("ejs", ejsMate);

//Database Connection

// let MONGO\_URL = "mongodb://127.0.0.1:27017/GharSetu";

let MONGO\_URL = process.env.ATLASDB\_URL;

main()

.then(() => {

console.log("DB connected Sucessfully");

})

.catch((err) => console.log(err));

async function main() {

await mongoose.connect(MONGO\_URL);

}

//Mongo Session

const store = MongoStore.create({

mongoUrl: process.env.ATLASDB\_URL,

crypto: {

secret: process.env.SECRET,

},

touchAfter: 24 \* 60 \* 60,

});

store.on("error", () => {

console.log("Error in Mongo Session Store", err);

});

//Creating Sessions

const sessionOptions = {

store,

secret: process.env.SECRET,

resave: false,

saveUninitialized: true,

cookie: {

expires: 7 \* 24 \* 60 \* 60 \* 1000,

maxAge: 7 \* 24 \* 60 \* 60 \* 1000,

httpOnly: true,

},

};

//Home Route

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

res.redirect("/listings");

});

//Session and Flash

app.use(session(sessionOptions));

app.use(flash());

//Login (Authentication - Passport npm)

app.use(passport.initialize());

app.use(passport.session());

passport.use(new LocalStrategy(User.authenticate()));

passport.serializeUser(User.serializeUser()); // Login

passport.deserializeUser(User.deserializeUser()); // Logout

//Store Info for Ejs- Middleware

app.use((req, res, next) => {

res.locals.success = req.flash("success");

res.locals.error = req.flash("error");

res.locals.currUser = req.user;

next();

});

//Routes

app.use("/listings", listingRouter);

app.use("/listings/:id/reviews", reviewRouter);

app.use("/", userRouter);

//Page not found

app.all("\*", (req, res, next) => {

next(new ExpressError(404, "Page not found"));

});

//------------------Error Handler Middleware

app.use((err, req, res, next) => {

let { status = 500, message = "Something went wrong" } = err;

res.status(status).render("listings/error", { message, err });

});

//Server listen on port 8080

app.listen(8080, () => console.log("Server is listening on port 8080"));

* Review Schema

const mongoose = require("mongoose");

const { Schema } = mongoose;

const reviewSchema = new mongoose.Schema(

{

comment: String,

rating: {

type: Number,

min: 1,

max: 5,

},

author: {

type: Schema.Types.ObjectId,

ref: "User",

},

},

{ timestamps: true }

);

const Review = mongoose.model("Review", reviewSchema);

module.exports = Review;

* User Schema

const mongoose = require("mongoose");

const passportLocalMongoose = require("passport-local-mongoose");

const userSchema = new mongoose.Schema({

email: {

type: String,

required: true,

unique: true,

},

});

userSchema.plugin(passportLocalMongoose);

const User = mongoose.model("User", userSchema);

module.exports = User;

* Middleware

const Listing = require("./models/listing");

const ExpressError = require("./utils/ExpressError.js");

const { listingSchema, reviewSchema } = require("./schema.js");

const Review = require("./models/review.js");

module.exports.isLoggedIn = (req, res, next) => {

if (!req.isAuthenticated()) {

req.session.redirectUrl = req.originalUrl;

req.flash("error", "User must be logged in for creating new listing!");

return res.redirect("/login");

}

next();

};

module.exports.saveRedirectUrl = (req, res, next) => {

if (req.session.redirectUrl) {

res.locals.redirectUrl = req.session.redirectUrl;

}

next();

};

module.exports.isOwner = async (req, res, next) => {

let { id } = req.params;

let listing = await Listing.findById(id);

if (

!res.locals.currUser &&

listing.owner.\_id.equals(res.locals.currUser.\_id)

) {

req.flash("error", "You are not the owner of this listing ");

return res.redirect(`/listings/${id}`);

}

next();

};

module.exports.isReviewAuthor = async (req, res, next) => {

let {id, reviewId } = req.params;

let review = await Review.findById(reviewId);

if (!review.author.equals(res.locals.currUser.\_id)) {

req.flash("error", "You are not the author of this review ");

return res.redirect(`/listings/${id}`);

}

next();

};

//Validate listing from server side(hopscotch)

module.exports.validateListing = (req, res, next) => {

let { error } = listingSchema.validate(req.body);

if (error) {

let errMsg = error.details.map((el) => el.message).join(",");

throw new ExpressError(400, errMsg);

} else {

next();

}

};

//Validate Review from server side(hopscotch)

module.exports.validateReview = (req, res, next) => {

let { error } = reviewSchema.validate(req.body);

if (error) {

let errMsg = error.details.map((el) => el.message).join(",");

throw new ExpressError(400, errMsg);

} else {

next();

}

};

* Cloud – Cloudinary

const cloudinary = require("cloudinary").v2;

const { CloudinaryStorage } = require("multer-storage-cloudinary");

cloudinary.config({

cloud\_name: process.env.CLOUD\_NAME,

api\_key: process.env.CLOUD\_API\_KEY,

api\_secret: process.env.CLOUD\_API\_SECRET,

});

const storage = new CloudinaryStorage({

cloudinary: cloudinary,

params: {

folder: "GharSetu",

allowerdFormats: ["png", "jpg", "jpeg"],

},

});

module.exports = { cloudinary, storage };

* Controllers Listing

const Listing = require("../models/listing");

const mbxGeocoding = require("@mapbox/mapbox-sdk/services/geocoding");

const mapToken = process.env.MAP\_TOKEN;

const geocodingClient = mbxGeocoding({ accessToken: mapToken });

module.exports.index = async (req, res) => {

const allListings = await Listing.find({}).sort({ updatedAt: -1 });

res.render("listings/index", { allListings });

};

module.exports.renderNewForm = (req, res) => {

res.render("listings/new.ejs");

};

module.exports.createListing = async (req, res, next) => {

let response = await geocodingClient

.forwardGeocode({

query: req.body.listing.location,

limit: 1,

})

.send();

let url = req.file.path;

let filename = req.file.filename;

let geometry = response.body.features[0].geometry;

let { title, description, image, price, location, country } =

req.body.listing;

let owner = req.user.\_id;

const newListing = await Listing.insertOne({

title,

description,

image: { url: url, filename: filename },

price,

location,

country,

geometry,

owner,

});

console.log(newListing);

req.flash("success", "New Listing Created");

res.redirect("/listings");

};

module.exports.showListing = async (req, res) => {

let { id } = req.params;

const curListing = await Listing.findById(id)

.populate({

path: "reviews",

options: { sort: { updatedAt: -1 } },

populate: { path: "author" },

})

.populate("owner");

if (!curListing) {

req.flash("error", "Listing you requested for does not exist");

res.redirect("/listings");

}

res.render("listings/show", { curListing });

};

module.exports.renderEditForm = async (req, res) => {

let { id } = req.params;

let editListing = await Listing.findById(id);

if (!editListing) {

req.flash("error", "Listing you requested for does not exist");

res.redirect("/listings");

}

let originalImageUrl = editListing.image.url;

originalImageUrl.replace("/upload", "/upload/w\_250");

res.render("listings/edit", { editListing, originalImageUrl });

};

module.exports.updateListing = async (req, res) => {

let { id } = req.params;

const listingData = Object.fromEntries(Object.entries(req.body.listing));

let updateListing = await Listing.findByIdAndUpdate(id, listingData, {

new: true,

runValidators: true,

});

if (typeof req.file !== "undefined") {

updateListing.image.url = req.file.path;

updateListing.image.filename = req.file.filename;

await updateListing.save();

}

req.flash("success", "Listing Updated Successfully");

res.redirect(`/listings/${id}`);

};

module.exports.destroyListing = async (req, res) => {

let { id } = req.params;

await Listing.findByIdAndDelete(id);

req.flash("success", "Listing Deleted Successfully");

res.redirect("/listings");

};

* Controllers Review

const Review = require("../models/review.js");

const Listing = require("../models/listing.js");

module.exports.createReview = async (req, res) => {

let listing = await Listing.findById(req.params.id);

let newReview = new Review(req.body.review);

newReview.author = req.user.\_id;

listing.reviews.push(newReview);

await newReview.save();

await listing.save();

let id = req.params.id;

req.flash("success", "New Review Created!");

res.redirect(`/listings/${id}`);

}

module.exports.destroyReview= async(req, res) => {

let { id, reviewId } = req.params;

await Listing.findByIdAndUpdate(id, { $pull: { reviews: reviewId } });

await Review.findByIdAndDelete(reviewId);

req.flash("success", "Review Deleted!");

res.redirect(`/listings/${id}`);

}

* Controllers User

const User = require("../models/user.js");

module.exports.signupForm = (req, res) => {

res.render("users/signup");

};

module.exports.signupUser = async (req, res) => {

try {

let { username, email, password } = req.body;

let newUser = new User({

username,

email,

});

let registeredUser = await User.register(newUser, password);

console.log(registeredUser);

req.login(registeredUser, (err) => {

if (err) {

return next(err);

}

req.flash("success", `${username} : Welcome to GharSetu!`);

res.redirect("/listings");

});

} catch (err) {

req.flash("error", err.message);

res.redirect("/signup");

}

};

module.exports.loginForm = (req, res) => {

res.render("users/login");

};

module.exports.loginUser = async (req, res, next) => {

req.flash("success", "Welcome back to GharSetu! ");

let redirectUrl = res.locals.redirectUrl || "/listings";

res.redirect(redirectUrl);

};

module.exports.logout =

(req, res) => {

req.logout((err) => {

if (err) {

return next(err);

}

req.flash("success", "you are logged out!");

res.redirect("/listings");

});

}