# **CHAPTER 1**

## **Introduction**

Streamlining Room Management and Rentals" represents a pivotal venture into the optimization of room management processes within the realm of accommodation services. The modern landscape of hospitality demands a departure from traditional methodologies towards innovative solutions that not only alleviate administrative burdens but also enrich the user experience.

This project is poised to revolutionize the conventional paradigms of room bookings, seeking to establish a dynamic and technology-driven framework that enhances efficiency for administrators and elevates satisfaction for users.

The essence of the project lies in the creation of a sophisticated web application engineered to redefine the dynamics of room reservations. By embracing advanced technologies and prioritizing user-centric design, our objective is to introduce a platform that not only simplifies administrative tasks but also aligns with the evolving expectations of a contemporary audience.

In an era where speed and convenience reign supreme, our project is committed to leveraging real-time data, automation, and seamless communication to usher in a transformative experience in the domain of room management.

As we embark on this journey, the project "Streamlining Room Management and Rentals" holds the promise of not just modernizing but revolutionizing the way rooms are managed and rented, offering a glimpse into the future of efficient and user-friendly accommodation services.

# **CHAPTER 2:**

## **Requirement Specification**

### **2.1 Hardware Requirements:**

#### **2.1.1 Server Infrastructure:**

Deploy on a robust and scalable cloud server (e.g., AWS, Azure).

Minimum hardware specifications for the server:

Processor: Dual-core processor or higher.

RAM: 8 GB or more.

Storage: SSD with at least 50 GB.

### **2.1.2 Database Server:**

Dedicated database server to ensure optimal performance.

Minimum hardware specifications:

Processor: Quad-core processor or higher.­­­­­­­­­­­­

RAM: 16 GB or more.

Storage: SSD with sufficient capacity for data storage and backups.

### **2.1.3 Networking:**

High-speed internet connection for real-time data updates.

Load balancing and failover mechanisms for improved reliability.

Secure sockets layer (SSL) for encrypted communication.

### **2.1.4 Client Devices:**

Compatible with a range of devices including desktops, laptops, tablets, and smartphones.

Web browser compatibility (e.g., Chrome, Firefox, Safari).

­­­­

**2.2 Software Requirements:**

### **2.2.1 Operating System:**

Development Environment (Windows):

Windows 10 or higher.

Visual Studio Code or any suitable IDE for Windows.

Server (Production):

Linux-based operating system (e.g., Ubuntu, CentOS) or Windows Server (based on your preference).

### **2.2.2 Web Server:**

##### **2.2.2.a MongoDB Atlas:**

MongoDB Atlas is a cloud-based database service provided by MongoDB. It offers a fully-managed, highly available, and scalable database solution for MongoDB, which is a NoSQL database. MongoDB Atlas simplifies database management by handling tasks such as backups, monitoring, and scaling, allowing developers to focus on building applications. It supports automatic sharding for horizontal scaling, ensuring optimal performance as your data grows.

##### **2.2.2.b Cloudinary:**

Cloudinary is a cloud-based media management platform that provides a comprehensive solution for storing, managing, optimizing, and delivering images and videos. It offers a range of features, including dynamic image and video transformations, automatic optimization, and efficient delivery through a Content Delivery Network (CDN). Cloudinary is widely used by developers and businesses to streamline media management in web and mobile applications.

**2.2.2.c Render:**

Render is a cloud platform that simplifies the deployment and scaling of web applications. It provides a straightforward and developer-friendly experience for hosting applications, including static sites, APIs, and more. Render supports various programming languages and frameworks and offers features like automatic scaling, built-in HTTPS, and continuous deployment from Git. It aims to simplify the deployment process and reduce the operational overhead associated with hosting web applications.

**In summary:**

MongoDB Atlas is a managed MongoDB database service.

Cloudinary is a cloud-based media management platform for images and videos.

Render is a cloud platform for hosting and scaling web applications

# **2.3 Programming Language:**

Use a modern and widely-supported programming language (e.g., Node.js) for the backend. And HTML.

# **2.4 Frameworks and Libraries:**

Utilize a web application framework (e.g., Express.js) for efficient development.

# **2.3.5 Security Software:**

Implement security software and protocols to safeguard against common vulnerabilities. Regularly update and patch all software components.

# **2.3.6 Version Control:**

Implement a version control system (e.g., Git) for collaborative development.

# **2.3.7 Integrated Development Environment (IDE):**

Choose a suitable IDE for development and debugging (e.g., Visual Studio Code).

### **2.3.8 Payment Gateway Integration:**

Integrate with a secure and widely-used payment gateway (e.g., Stripe, PayPal).

# **2.3.9 Communication Protocols:**

Implement communication protocols (e.g., SMTP) for automated notifications. MongoDB for efficient and scalable data management. Make sure to adjust these requirements based on the specific versions and configurations that best suit your project's needs.

# **CHAPTER 3:**

## **Description:**

### **3.1Database and Data:**

#### **3.1.1 Database**

A database is a structured and organized collection of data, typically stored and accessed electronically from a computer system. Databases are designed to efficiently manage and organize large volumes of data, enabling users to retrieve, update, and manipulate information as needed. They serve as a central repository for storing various types of data and play a crucial role in the functioning of applications, websites, and information systems.

##### **3.1.1.a Key Characteristics:**

* **Structured Data Storage:**
* Databases organize data in a structured manner, using tables (in relational databases), documents (in document-oriented databases), or other formats depending on the database type.
* **Data Integrity:**

Databases enforce rules and constraints to maintain data integrity, ensuring that the stored information is accurate, consistent, and reliable.

* **Query Language:**

Databases provide a query language (e.g., SQL for relational databases) that allows users to interact with the data, perform queries, and retrieve specific information.

* **Concurrency Control:**

Databases manage concurrent access to data, allowing multiple users or applications to interact with the database simultaneously without compromising data integrity.

* **Indexing:**

Indexes are used to optimize data retrieval by providing faster access to specific data subsets. They enhance query performance.

* **ACID Properties:**

Transactions in databases adhere to ACID properties (Atomicity, Consistency, Isolation, Durability) to ensure reliability and maintainability of data.

* **Scalability:**

Databases are designed to scale horizontally (across multiple servers) or vertically (on a single server) to accommodate increasing data loads.

* **Security:**

Database systems implement security measures such as authentication, authorization, and encryption to protect sensitive data.

##### **3.1.1.b Types of Databases:**

* Relational Databases (e.g., MySQL, PostgreSQL):

Organize data in tables with predefined relationships between them.

* NoSQL Databases (e.g., MongoDB, Cassandra):

Allow flexible and schema-less data models, suitable for handling diverse and unstructured data.

* In-memory Databases (e.g., Redis, Memcached):

Store data in the system's main memory, providing fast access but with limited storage capacity.

* Graph Databases (e.g., Neo4j):

Optimize for representing and traversing relationships between data points.

#### **3.1.2 Data:**

Data refers to raw facts, figures, or information that can be processed, interpreted, and used for various purposes. It can take the form of numbers, text, images, audio, video, or any other type of content. Data is the foundation of information and knowledge, and its significance lies in its ability to provide insights, support decision-making, and facilitate understanding.

##### **3.1.2.a Types of Data:**

* Structured Data:

Well-organized and follows a predefined format, often stored in relational databases. Examples include tables of data with rows and columns.

* Unstructured Data:

Lacks a predefined structure and does not fit neatly into a database. Examples include text documents, images, and multimedia files.

* Semi-structured Data:

Has some structure but does not conform to the rigidity of a relational database. Examples include JSON or XML documents.

* Big Data:

Refers to extremely large and complex datasets that traditional database systems may struggle to handle. It often involves large volumes of diverse data.

* Metadata:

Data about data, providing information about the characteristics, origin, and structure of other data.

##### **3.1.2.b Lifecycle of Data:**

* Collection:

Gathering raw data from various sources, such as sensors, applications, or manual input.

* Storage:

Storing data in databases, data warehouses, or other storage systems for easy retrieval and management

* Processing:

Analyzing and processing data to extract valuable insights, trends, or patterns.

* Analysis:

Interpreting and exploring data to make informed decisions or predictions.

* Presentation:

Communicating data findings through visualizations, reports, or dashboards.

* Archiving and Purging:

Archiving historical data for future reference and purging outdated or unnecessary data.

* Significance of Data:

Decision-Making

Data supports informed decision-making by providing evidence, context, and insights.

* Innovation:

Data is crucial for innovation, allowing organizations to identify trends, uncover opportunities, and drive improvements.

* Personalization:

Enables personalized experiences by understanding user preferences and behavior.

* Research and Development:

Researchers use data to test hypotheses, draw conclusions, and contribute to advancements in various fields.

* Security and Compliance:

Proper management of data ensures compliance with regulations and safeguards against unauthorized access.

#### **3.1.2.c Difference between SQL and NoSQL databases.**

|  |  |
| --- | --- |
| **SQL databases** | **NoSQL databases** |
| **Data Structure:**  Structured Data: SQL databases, also known as relational databases, organize data into structured tables with predefined schemas. | **Data Structure:**  Flexible Data Models: NoSQL databases handle unstructured, semi-structured, or structured data, allowing for more flexibility in data models. |
| **Schema:**  Fixed Schema: Tables in SQL databases have a fixed schema, meaning that the data structure (columns and data types) is defined beforehand. | **Schema:**  Dynamic Schema: NoSQL databases typically have a dynamic schema, allowing for the insertion of data without a predefined schema. |
| **Query Language:**  SQL Query Language: SQL databases use the SQL language for defining and manipulating the data. SQL allows for powerful querying, joining, and aggregating operations. | **Query Language:**  Varied Query Languages: NoSQL databases use a variety of query languages, and some might not have a structured query language similar to SQL. |
| **Scalability:**  Vertical Scaling: SQL databases often scale vertically, which means adding more power to an existing server. This might have limitations in handling large amounts of data. | **Scalability:**  Horizontal Scaling: NoSQL databases are often designed to scale horizontally, distributing data across multiple servers, making them more suitable for handling large amounts of data and high traffic. |
| **Transactions:**  ACID Properties: SQL databases follow the ACID properties (Atomicity, Consistency, Isolation, Durability) to ensure data integrity. | **Transactions:**  BASE Properties: NoSQL databases often follow the BASE model (Basically Available, Soft state, eventually consistent), providing more flexibility in terms of consistency. |
| **Use Cases:**  SQL databases are suitable for applications with a well-defined schema, structured data, and complex queries, such as financial systems, enterprise applications, and relational data. | **Use Cases:**  NoSQL databases are suitable for applications where the data structure is evolving, and there is a need for high scalability, quick development cycles, and handling large volumes of data, such as in web applications, content management systems, and real-time big data applications. |

### **3.2MongoDB and Mongoose:**

##### **3.2.1MongoDB:**

MongoDB is a popular NoSQL (Not Only SQL) database management system that uses a document-oriented data model. It is designed to store, query, and manage data in a flexible, JSON-like format called BSON (Binary JSON). MongoDB is known for its scalability, performance, and ability to handle unstructured or semi-structured data. It is widely used in modern web development, where the data structure can evolve over time.

##### **3.2.1.a Key Features:**

* Document-Oriented:

MongoDB stores data in flexible, schema-less BSON documents. Each document can have a different structure, providing flexibility in data modelling.

* No Schema Constraints:

MongoDB allows dynamic schema design, enabling developers to add fields to documents without affecting the existing data.

* Scalability:

MongoDB supports horizontal scaling, allowing it to handle large volumes of data by distributing it across multiple servers or clusters.

* Query Language:

MongoDB uses a rich query language for data retrieval, supporting a wide range of queries, indexing, and aggregation operations.

* Geospatial Capabilities:

MongoDB includes geospatial features, making it suitable for applications that require location-based data.

* Automatic Sharding:

MongoDB can automatically partition and distribute data across multiple nodes, improving performance and scalability.

* Replication:

MongoDB supports automatic data replication, ensuring data redundancy and high availability.

* Use Cases: Content Management Systems (CMS), Real-time Analytics, Internet of Things (IoT) applications, Mobile Applications, Catalos and Product Databases

#### **3.2.2 Mongoose:**

Mongoose is an Object Data Modelling (ODM) library for MongoDB and Node.js. It provides a structured, schema-based solution for interacting with MongoDB, making it easier for developers to work with MongoDB databases in a Node.js environment. Mongoose allows the definition of models with schemas, enforcing data validation and providing a straightforward API for querying and interacting with the database.

**3.2.2.a Key Features:**

* Schema Definition:

Mongoose allows developers to define schemas with data types, validation rules, and default values, providing structure to MongoDB documents.

* Middleware Support:

Developers can use middleware functions to execute logic before or after certain events, such as saving or querying documents.

* Query Building:

Mongoose provides a powerful query API that makes it easy to build and execute queries using a fluent syntax.

* Validation:

Mongoose supports data validation, ensuring that data meets predefined criteria before being stored in the database.

* Population:

Mongoose supports data population, allowing developers to reference other documents and populate them when querying.

* Use Cases:

Web Application Development, RESTful API Development, Microservices Architecture

Any Node.js project interacting with MongoDB

**Example Code:** const mongoose = require('mongoose');

const user Schema = new mongoose. Schema ({

username: {type: String, required: true},

email: {type: String, required: true, unique: true},

age: Number

});

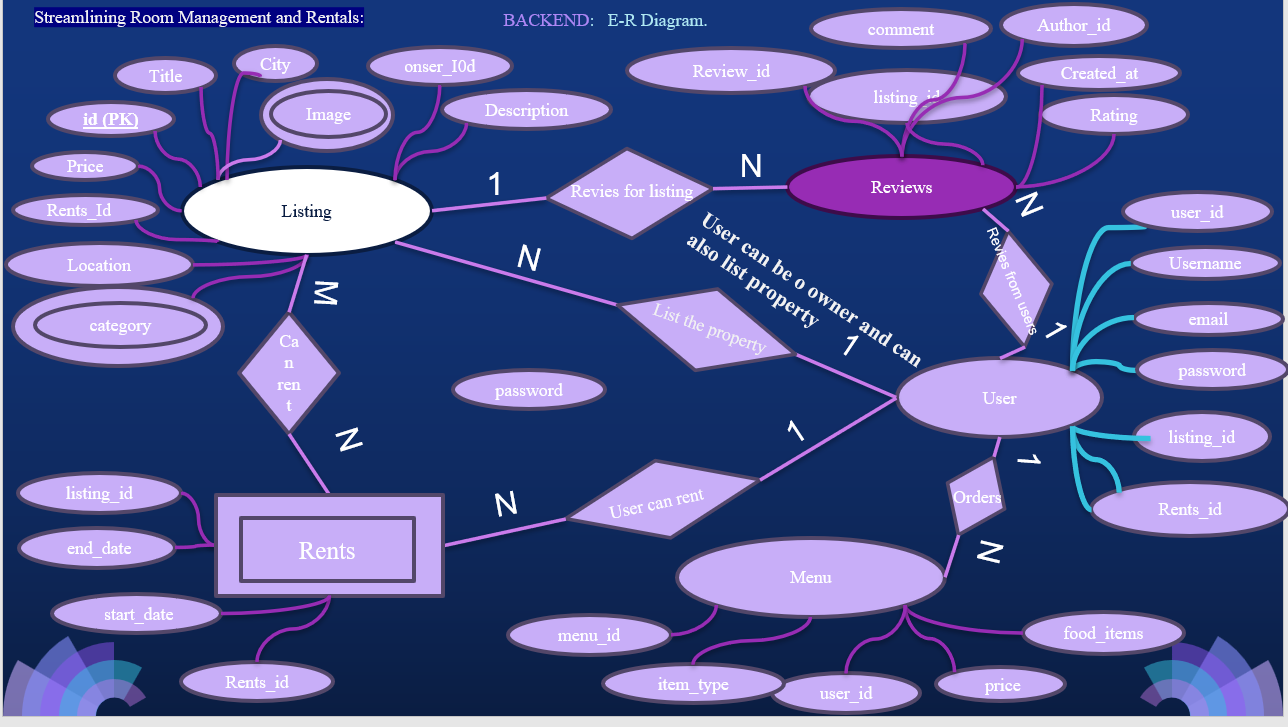
const User = mongoose. Model ('User', user Schema);

# **CHAPTER 4: Coding:**

#### **4. Back End and Front End:**

##### **4.1 Back End:**

###### **4.1.1 E-R diagram / Entity Relation Diagram:**



###### **4.1.2 E-R Schema Diagram:**

##### **4.1.3 Schemas:**

**User Schema:**

const mongoose = require("mongoose");

const Schema = mongoose.Schema;

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

const userSchema = new Schema({

     email: {

        type: String,

        required: true},

    rents: [{

            type: Schema.Types.ObjectId,

            ref: 'Rent',},],

    menu: [{

            type: Schema.Types.ObjectId,

            ref: 'Menu',}],});

userSchema.plugin(passportLocalMongoose);

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

**Rent Schema:**

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const rentSchema = new Schema({

    startDate: {

        type: Date,

        required: true,},

    endDate: {

        type: Date,

      required: true,

    },

    listing: {

        type: Schema.Types.ObjectId,

        ref: "Listing",

      },}

);

const Rent = mongoose.model('Rent', rentSchema);

module.exports = Rent;

**Listing Schema:**

const mongoose = require("mongoose");

const Schema = mongoose.Schema;

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

const Rent = require('./Rent.js');

const listingSchema = new Schema({

    title: {

        type: String,

        required: true,

    },

    description: String,

    image: {

        url: String,

        filename: String,

    },

    price: Number,

    location: String,

    country: String,

    category: {

        type: String,

        enum: ["Residential", "Commercial", "Vacation", "Cottage", "Apartment", "Condo", "Townhouse", "Other"],

        default: "Other",

    },

    reviews: [

{

            type: Schema.Types.ObjectId,

            ref: "Review",

}

],

    owner: {

        type: Schema.Types.ObjectId,

        ref: "User",

    },

);

const Listing = mongoose.model("Listing", listingSchema);

module.exports = Listing;

Review Schema:

const mongoose= require("mongoose");

const Schema= mongoose.Schema;

const reviewSchema = new Schema({

    Comment:String,

    rating:{

        type:String,

        min:1,

        max:5},

    createAt:{

        type:Date,

        default:Date.now()

    },

    author:{

        type:Schema.Types.ObjectId,

        ref:"User"

    },

     listing: {

        ref: "Listing",

      },

})

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

module.exports = Review;

Meny Schema:

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const menuSchema = new Schema({

    itemType: String,

    foodItems: [String],

    price: Number,

    listing: {

        type: Schema.Types.ObjectId,

        ref: "Listing",

      },

});

const Menu = mongoose.model('Menu', menuSchema);

module.exports = Menu;

### **4.1.4 Modules:**

### **Listing Modules:**

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

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

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

const AllListings = await Listing.find({});

res.render("listing/index.ejs", { All Listings });

};

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

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

};

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

let { id } = req.params;

const listing = await Listing.findById(id).populate({ path: "reviews", populate: { path: "author" } }).populate("owner");

console.log(listing)

if (!listing) {

req.flash("error", "Listing not found");

res.redirect("/listings");

}

res.render("listing/show.ejs", { listing });

};

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

try {

if (!req.file) {

req.flash("error", "No image uploaded");

return res.redirect("/listings");

}

let url = req.file.path;

let filename = req.file.filename;

const newListing = new Listing(req.body.listing);

newListing.owner = req.user.\_id;

newListing.image = { url, filename };

await newListing.save();

req.flash("success", "Successfully created a new listing");

return res.redirect("/listings");

} catch (error) {

console.error(error);

req.flash("error", "An error occurred while creating a new listing");

return res.redirect("/listings");

}

};

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

let { id } = req.params;

const listing = await Listing.findById(id);

if (!listing) {

req.flash("error", "Listing not found");

res.redirect("/listings");

}

let orginalImage = listing.image.url;

Newimgurl = orginalImage.replace("/upload", "/upload/ar\_1.0,c\_fill,w\_150");

res.render("listing/edit.ejs", { listing, Newimgurl });

};

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

try {

let { id } = req. params;

let listing = await Listing.findByIdAndUpdate(id, { ...req.body.listing });

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

let url = req.file.path;

let filename = req.file.filename;

listing.image = { url, filename };

await listing.save();

}

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

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

} catch (error) {

console.error(error);

req.flash("error", "An error occurred while updating the listing");

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

}

};

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

let { id } = req.params;

let deleted = await Listing.findByIdAndDelete(id);

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

res.redirect("/listings");

};

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

try {

const AllListings = await Listing.find({ category: 'Cottage' });

res.render("listing/index.ejs", { AllListings });

} catch (err) {

console.error(err);

res.status(500).send("Internal Server Error");

}

};

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

try {

const AllListings = await Listing.find({ category: 'Residential' });

console.log(AllListings);

res.render("listing/index.ejs", { AllListings });

} catch (err) {

console.error(err);

res.status(500).send("Internal Server Error");

}

};

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

try {const AllListings = await Listing.find({ category: 'Commercial' });

res.render("listing/index.ejs", { AllListings }):} catch (err) {

console.error(err);

res.status(500).send("Internal Server Error");}

};

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

try {

const AllListings = await Listing.find({ category: 'Vacation' });

res.render("listing/index.ejs", { AllListings });

} catch (err) {

console.error(err);

res.status(500).send("Internal Server Error");

}

};

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

try {

const AllListings = await Listing.find({ category: 'Apartment' });

res.render("listing/index.ejs", { AllListings });

} catch (err) {

console.error(err);

res.status(500).send("Internal Server Error");

}

};

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

try {const AllListings = await Listing.find({ category: 'Condo' });

res.render("listing/index.ejs", { AllListings });

} catch (err) {

console.error(err);

res.status(500).send("Internal Server Error"); }

};

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

try {

const AllListings = await Listing.find({category: 'Townhouse' });

res. render("listing/index.ejs", { AllListings });

} catch (err) {

console.error(err);

res.status(500).send("Internal Server Error");

}

};

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

try {

const AllListings = await Listing.find({ category: 'Other' });

res.render("listing/index.ejs", { AllListings });

} catch (err) {

console.error(err);

res.status(500).send("Internal Server Error");

}

};

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

try {

// Find all reviews with a rating of 5 and populate the 'listing' field

const fiveStarReviews = await Review.find({ rating: 5 }).populate("listing");

// Log the populated reviews for debugging

console.log("Populated Reviews:", fiveStarReviews);

// Extract listing IDs from the populated reviews

const listingIds = fiveStarReviews

.filter((review) => review.listing) // Filter out reviews without a listing

.map((review) => review.listing.toString());

// Log the extracted listing IDs for debugging

console.log("Listing IDs:", listingIds);

// Find all listings with the retrieved IDs

const trendingListings = await Listing.find({ \_id: { $in: listingIds } });

// Log the trending listings for debugging

console.log("Trending Listings:", trendingListings);

res.render("listing/index.ejs", {AllListings: trendingListings });

} catch (err) {

// Log the error for debugging

console.error("Error in trending method:", err)

res.status(500).send("Internal Server Error");

}

};

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

console.log(req.query.q);

let input = req.query.q.trim().replace(/\s+/g, " ");

console.log(input);

if (input == "" || input == " ") {

req.flash("error", "Search value empty !!!");

res.redirect("/listings"); }

let data = input.split("");

let element = "";

let flag = false;

for (let index = 0; index < data.length; index++) {

if (index == 0 || flag) { element = element + data[index].toUpperCase();}

else {

element = element + data[index].toLowerCase();

}

flag = data[index] == " ";

}

console.log(element);

let AllListings = await Listing.find({

title: { $regex: element, $options: "i" },

});

if (AllListings.length != 0) {

res.locals.success = "Listings searched by Title";

res.render("listing/index.ejs", { AllListings });

return;}

if (AllListings.length == 0) {

AllListings = await Listing.find({

category: { $regex: element, $options: "i" },

}).sort({ \_id: -1 });

if (AllListings.length != 0) {

res.locals.success = "Listings searched by Category";

res.render("listing/index.ejs", { AllListings });

return: }}

if (AllListings.length == 0) {

AllListings = await Listing.find({

country: { $regex: element, $options: "i" },

}).sort({ \_id: -1 });

if (AllListings.length != 0) {

res.locals.success = "Listings searched by Location";

res.render("listing/index.ejs", { AllListings });

return; }}

const intValue = parseInt(element, 10);

const intDec = Number.isInteger(intValue);

if (AllListings.length == 0 && intDec) {

AllListings = await Listing.find({ price: { $lte: element } }).sort({price: 1,});

if (AllListings.length != 0) {

res.locals.success = `Listings searched for less than Rs ${element}`;

res.render("listing/index.ejs", { AllListings });

return; }}

if (AllListings.length == 0) {

req.flash("error", "Listings is not here !!!");

res.redirect("/listings");

}

};

## **Menu module:**

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

const Menu = require("../models/menu.js");

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

try {

const listingId = req.params.id;

console.log(listingId);

const listing = await Listing.findById(listingId).populate('menu')

res.render("menu/createmenu.ejs",{listing})

} catch (error) {

console.error(error);

res.status(500).render('error', { err: error });

}

};

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

try {

const listingId = req.params.id;

const { itemType, foodItems, price } = req.body.item;

// Find the listing

const listing = await Listing.findById(listingId);

// Create a new menu item

const newMenuItem = new Menu({

itemType,

foodItems,

price,});

// Save the menu item to the listing's menu array

listing.menu.push(newMenuItem);

// Save changes to the listing

await listing.save();

req.flash("success", "Your menu has been added to your profile");

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

} catch (error) {

console.error(error);

res.status(500).render('error', { err: error });}};

## **User Module:**

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

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

const passport = require('passport');

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

res.render("../views/users/signup.ejs");

}

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

try {

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

console.log(username, email, password); // Log values to check

let newUser = new User({ email, username });

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

console.log(registeredUser);

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

if (!err) {

req.flash('success', 'Welcome to Wanderlust');

res.redirect('/listings');

}

return next(err);

});

} catch (e) {

req.flash('error',e.massage);

res.redirect('/signup');

}

};

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

res.render("users/usetprofile.ejs")

}

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

res.render("users/login.ejs");

};

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

req.flash('success', "Welcome Back to wanderlust ");

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

res.redirect(redirectUrl);

};

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

req.logout((err)=>{

if(!err){

req.flash("success", "You have been logged out");

res.redirect('/listings');

}

return next(err);

});

};

### **Review Modules:**

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

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

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

const Rent= require("../models/Rent.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;

console.log(NewREview);

listing.reviews.push(NewREview);

await NewREview.save();

await listing.save();

req.flash("success","Successfully New Review saved");

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

};

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

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}`);

};

### **Rend Module:**

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

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

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

const Rent= require("../models/Rent.js");

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

try {

// Fetch the listing

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

if (!listing) {

req.flash('error', 'Listing not found');

return res.redirect('/'); // Or handle accordingly

}

// Create a new Rent instance

const { startDate, endDate } = req.body;

const rental = new Rent({

startDate,

endDate,

});

// Save the rental

await rental.save();

// Ensure that the rents array is initialized in the listing

listing.rents = listing.rents || [];

// Push the rental into the rents array of the listing

listing.rents.push(rental);

// Save the changes to the listing

await listing.save();

req.flash('success', 'Successfully rented the listing!');

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

} catch (error) {

console.error(error);

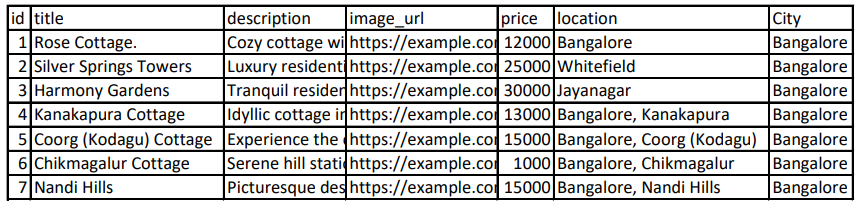
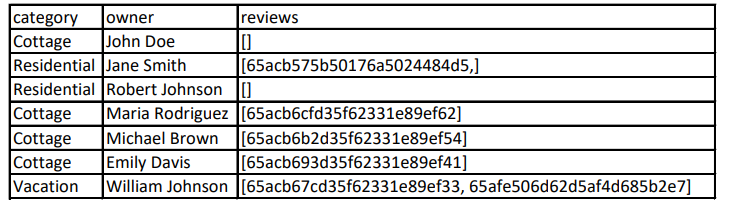
next(new ExpressError(500, 'Internal Server Error'));

}

};

# 4.1.5 Quires

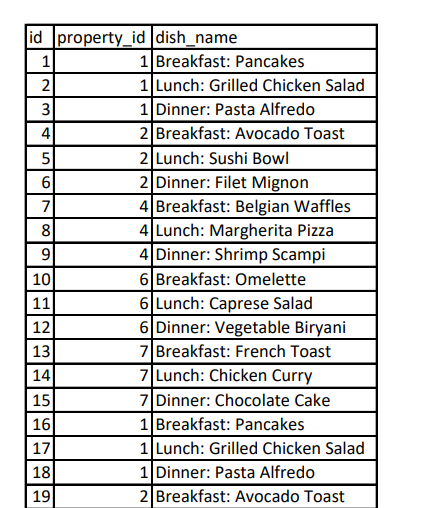
**const listings = await listing.find({});**



# **const rents = await Rent.find({});**

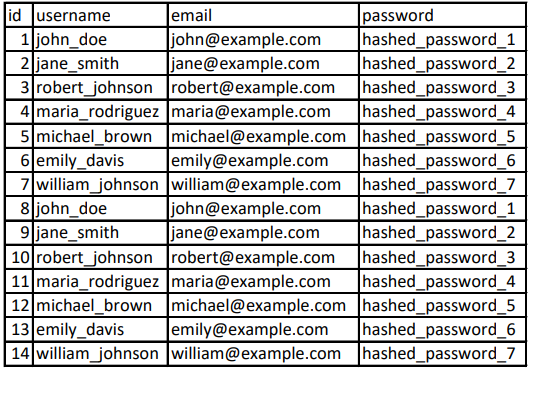
# rents

**const menuItems = await Menu.find({});**



**const reviews = await Review.find({});**



**const users = await User. Find({});**

**Inserting New User:**

const User = require('./models/user'); // Adjust the path accordingly

const exampleUser = new User({

username: 'exampleUser',

email: 'example@example.com',

password: 'hashedPassword' // Ensure this is actually a hashed password in your application

});

exampleUser.save()

.then((savedUser) => {

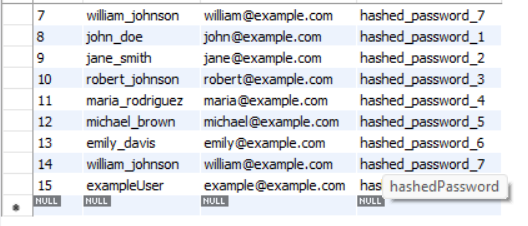
console.log('User saved:', savedUser);

})

.catch((error) => {

console.error('Error saving user:', error);

});



**Inserting Comments;**

const exampleRequest = {

params: {

id: 'your\_listing\_id',

},

user: {

\_id: 'user\_id',

},

body: {

review: {

text: 'This is a great listing!',

rating: 5,

},

},

};

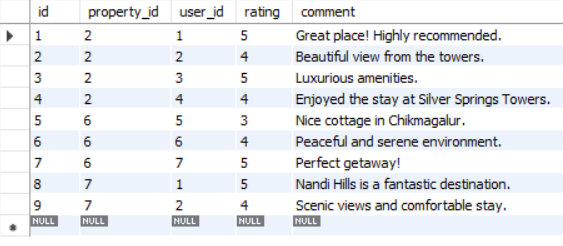
const exampleResponse = {

redirect: (path) => console.log(`Redirecting to: ${path}`),

flash: (type, message) => console.log(`Flash message (${type}): ${message}`),

};

CreateReview(exampleRequest, exampleResponse);



const exampleRequest = {

params: {

id: 'your\_listing\_id',

reviewId: 'your\_review\_id',

},

};

const exampleResponse = {

redirect: (path) => console.log (`Redirecting to: ${path}`),

flash: (type, message) => console.log(`Flash message (${type}): ${message}`),

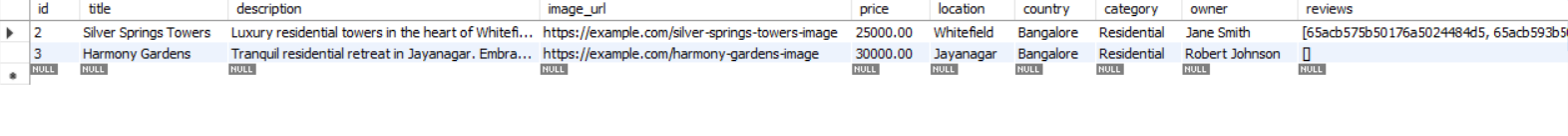
};

DeleteReview(exampleRequest, exampleResponse);



**Find Listing by Category:**

**const residentialProperties = await Property.find({ category: 'Residential' });**

****

**Inserting New Menu:**

const exampleRequest = {

params: {

id: 'your\_listing\_id', // Replace with a valid listing ID

},

body: {

item: {

itemType: 'Main Course', // Replace with a valid menu item type

foodItems: ['Spaghetti', 'Salad'], // Replace with valid food items

price: 15.99, // Replace with a valid price

},

},

};

const exampleResponse = {

redirect: (path) => console.log(`Redirecting to: ${path}`),

flash: (type, message) => console.log(`Flash message (${type}): ${message}`),

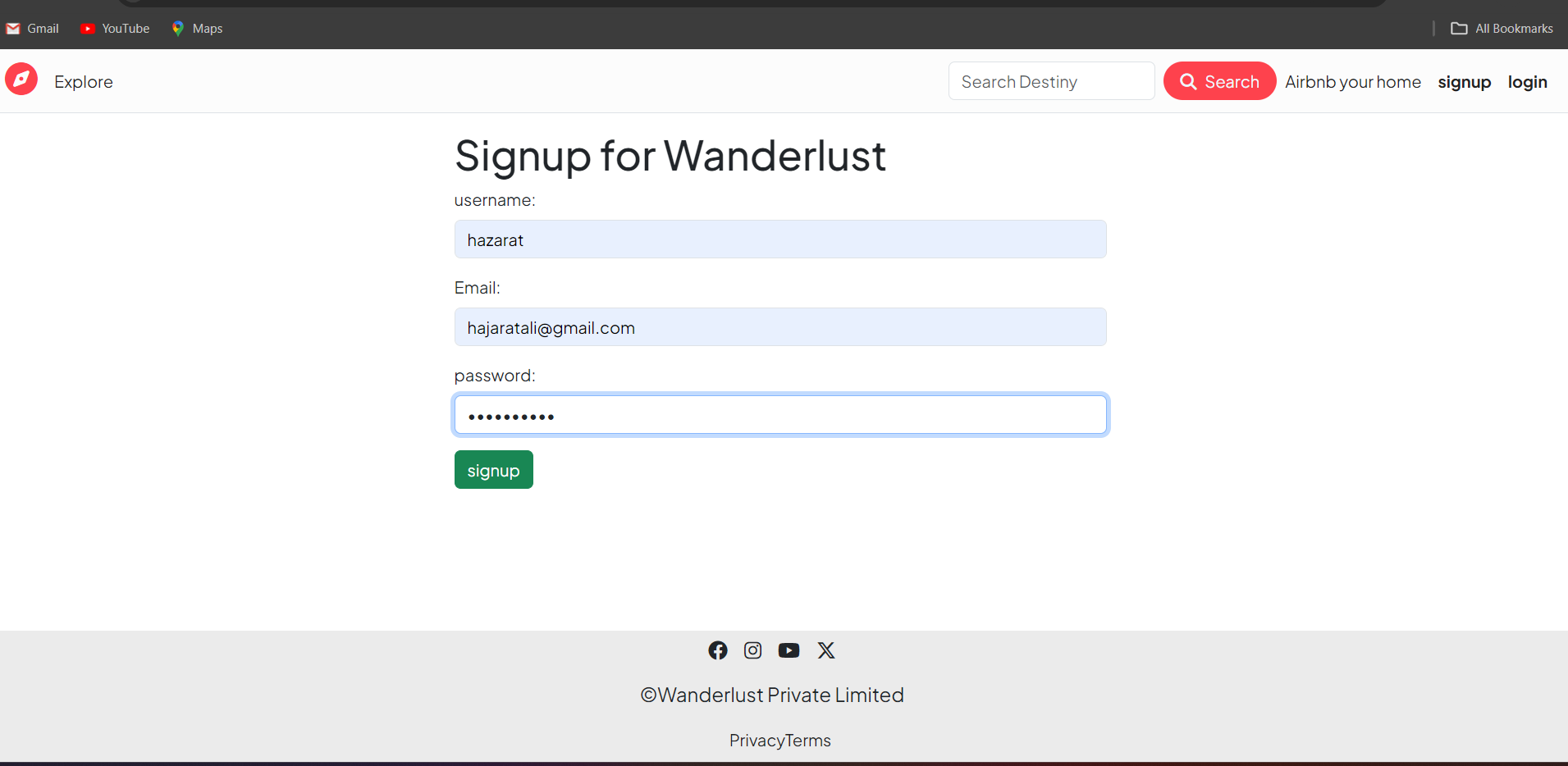
render: (view, data) => console.log(`Rendering view: ${view} with data:`, data),

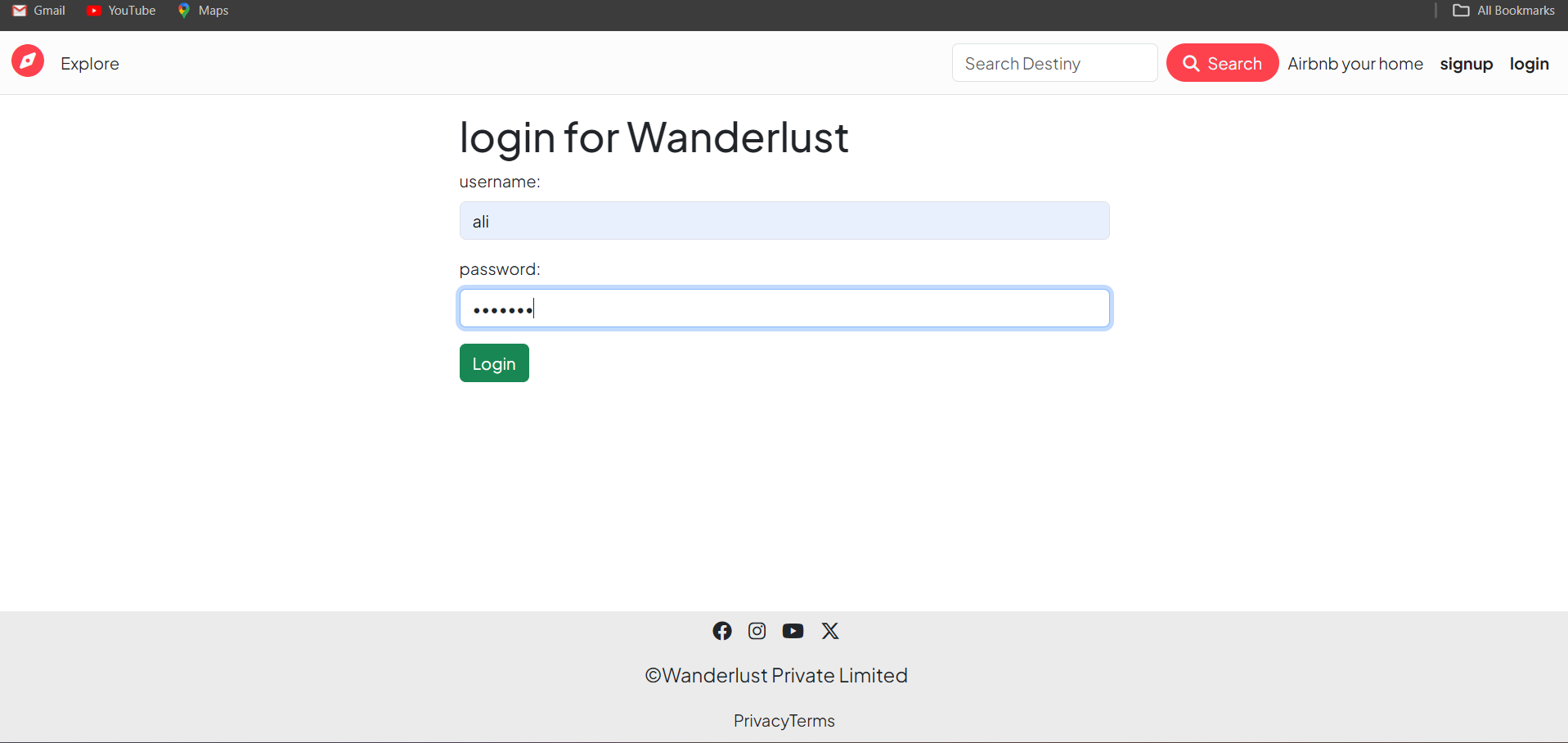
};

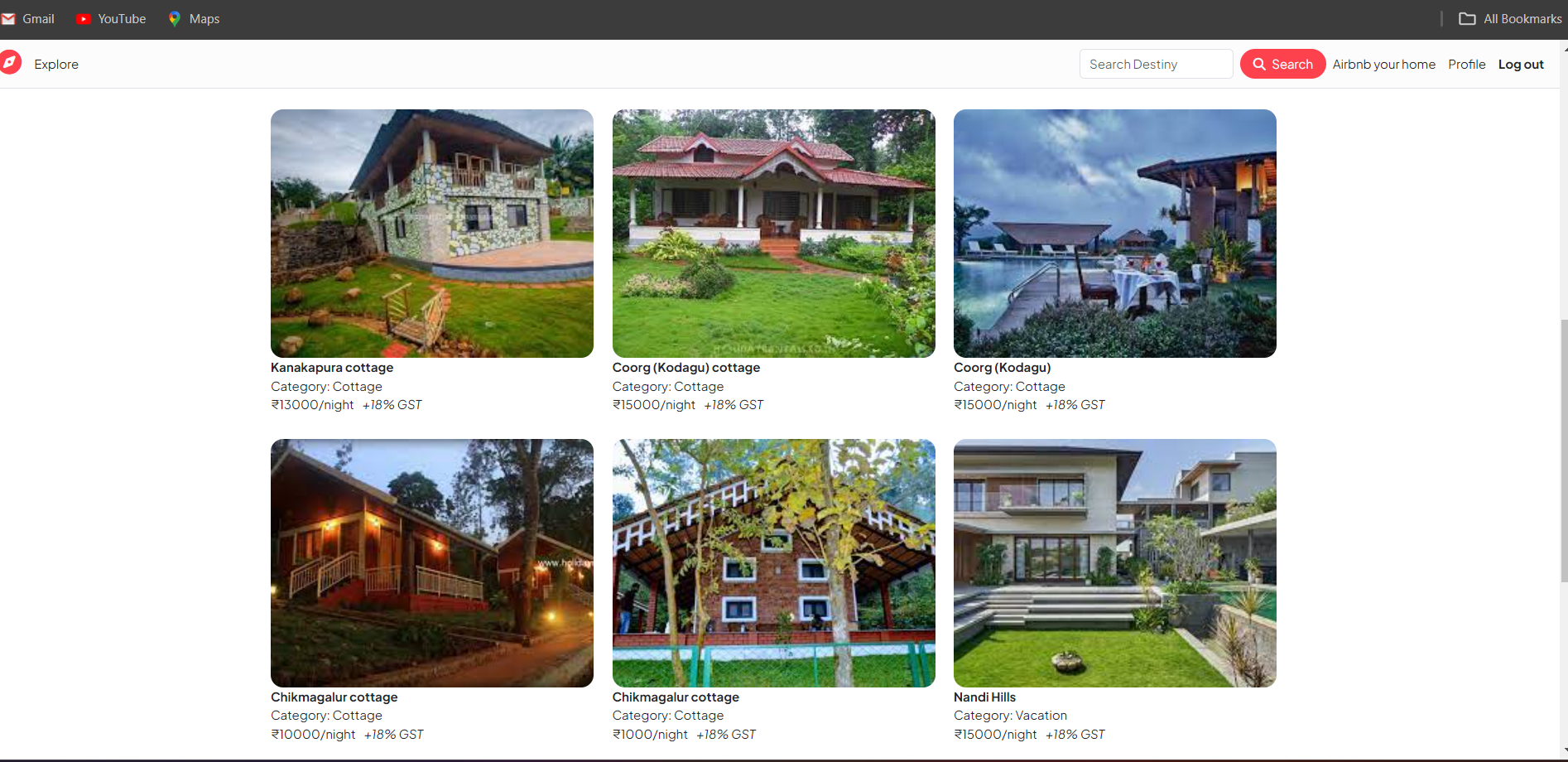
storeMenu(exampleRequest, exampleResponse);

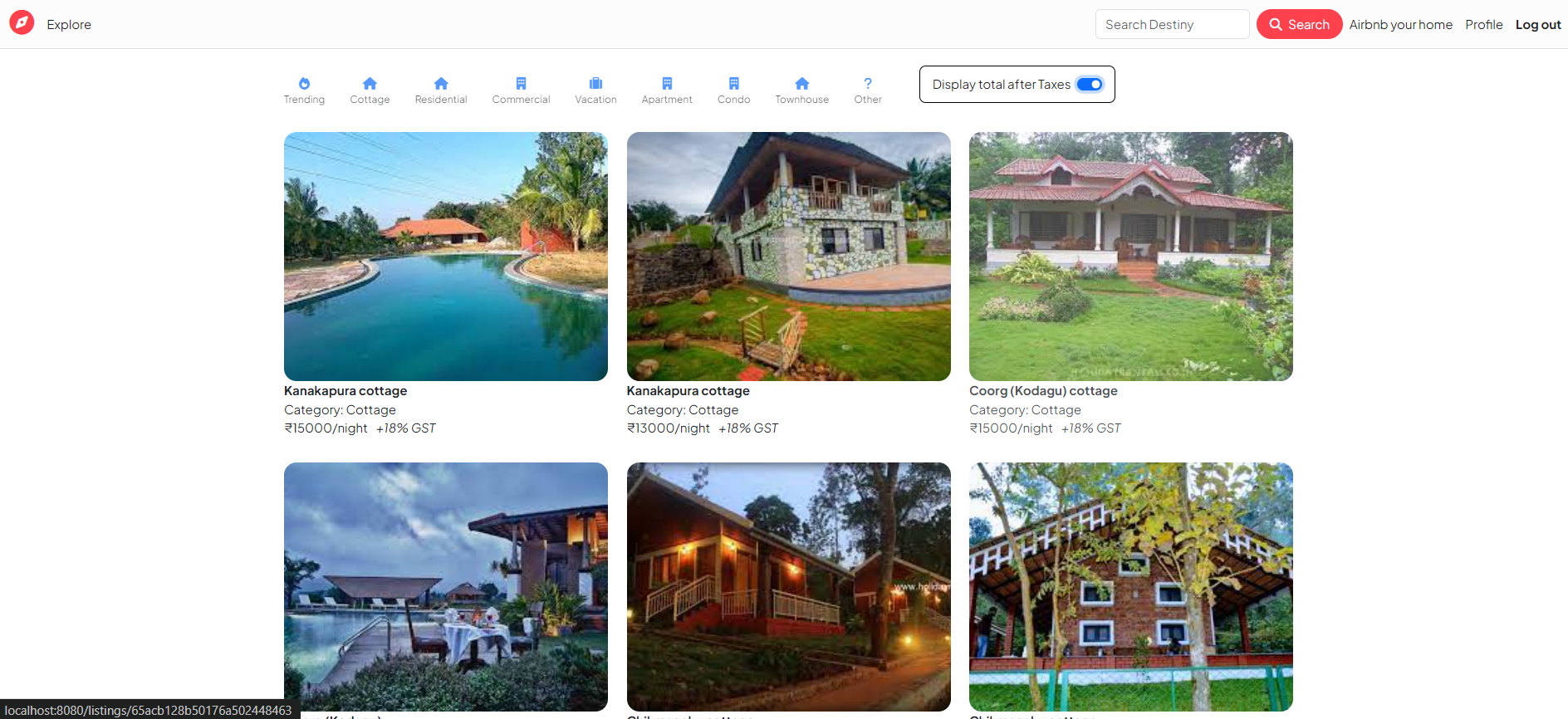


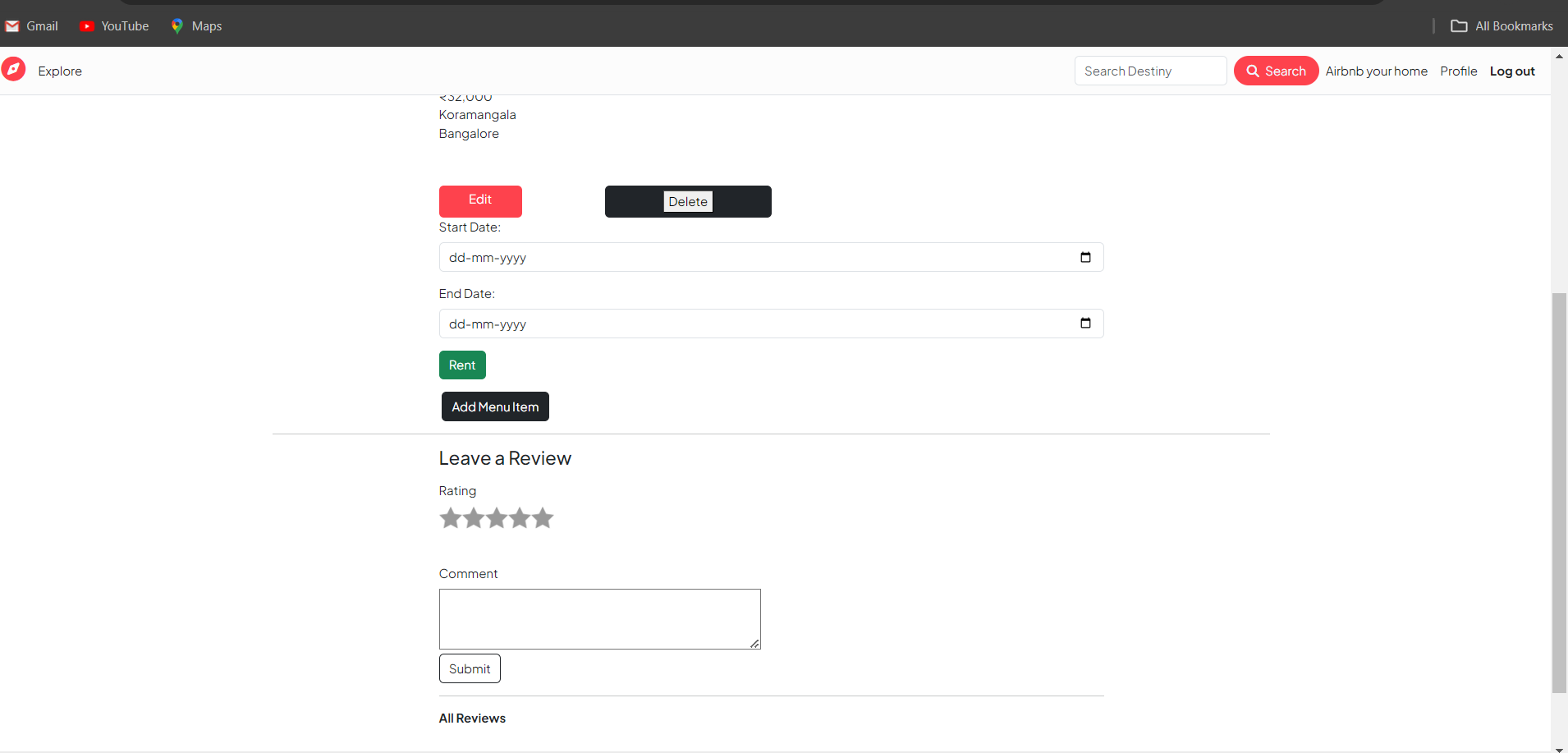
# **CHAPTER 5: Snapshots:**

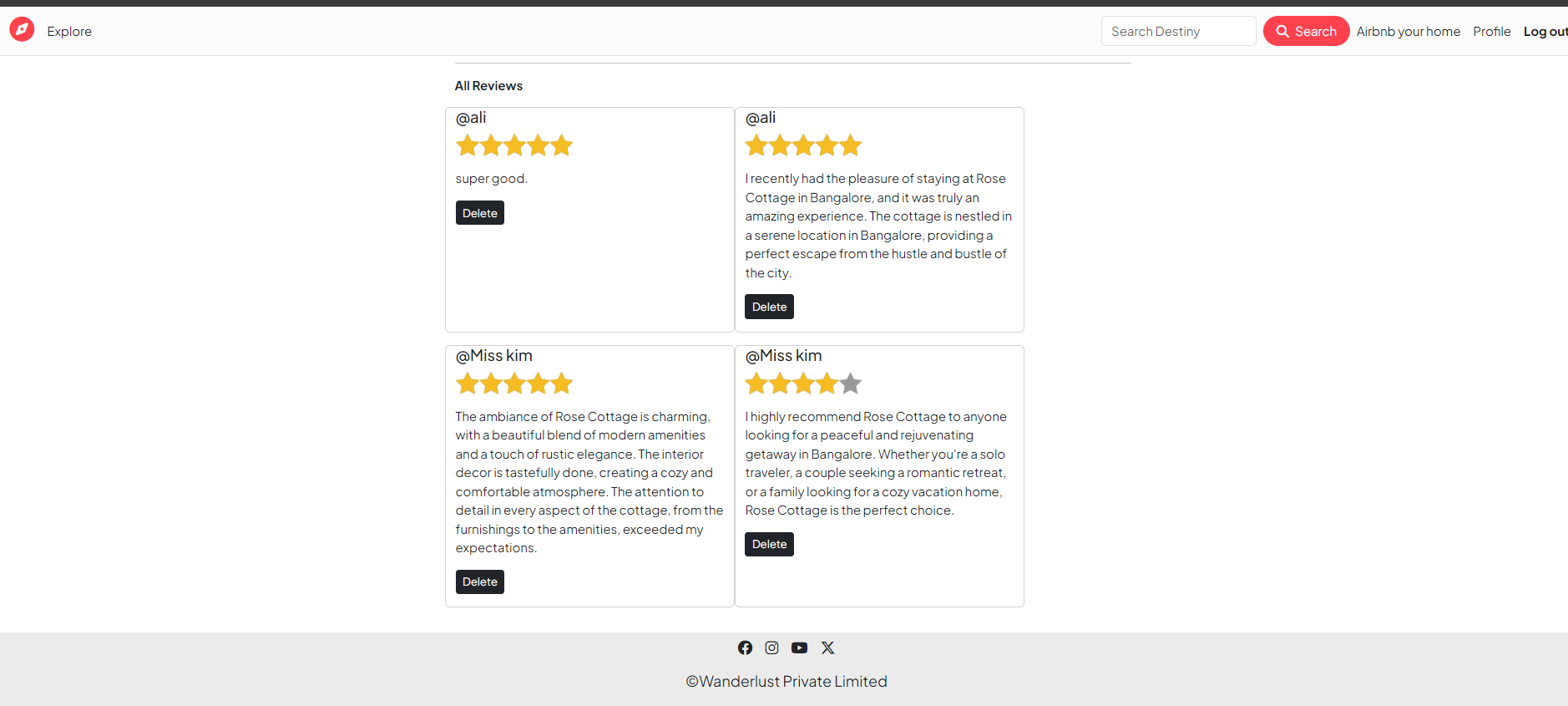


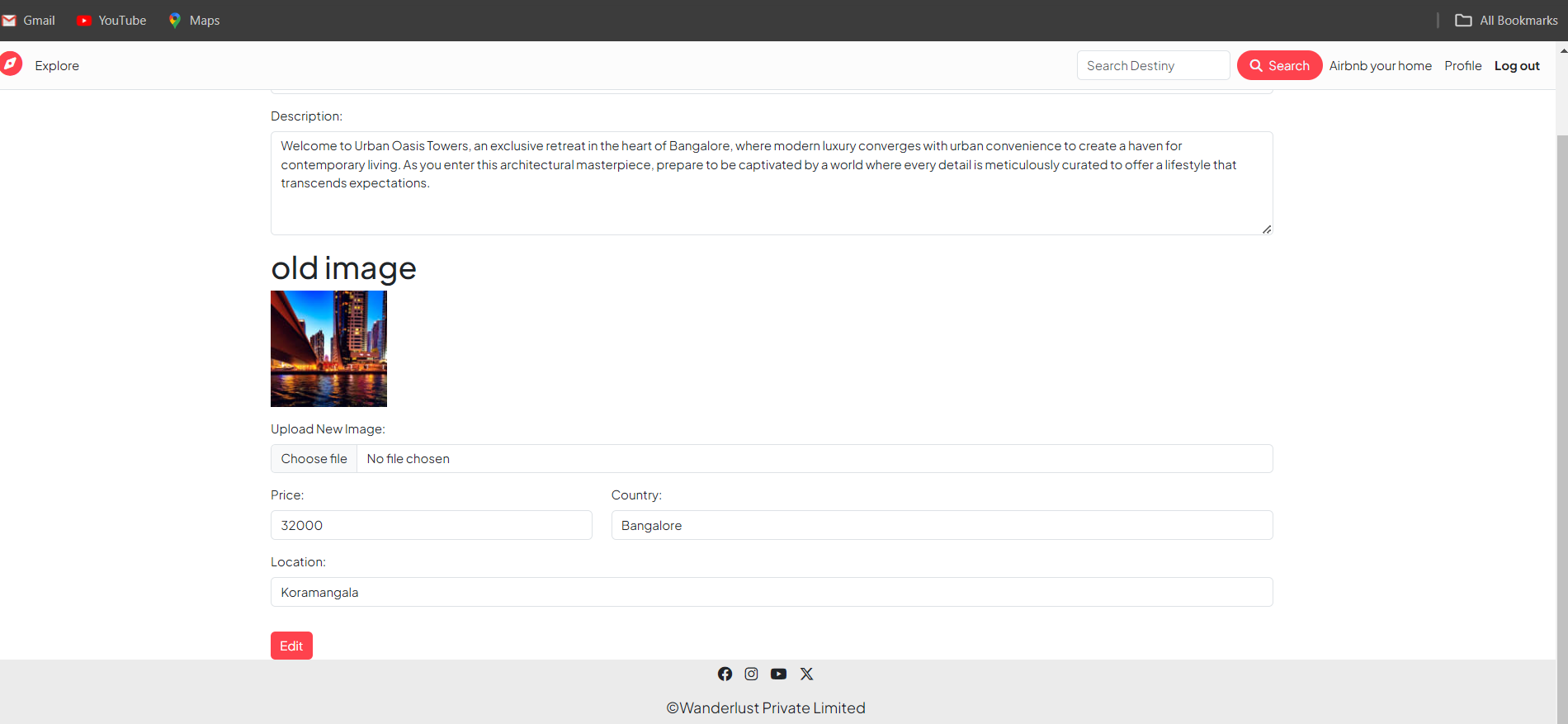


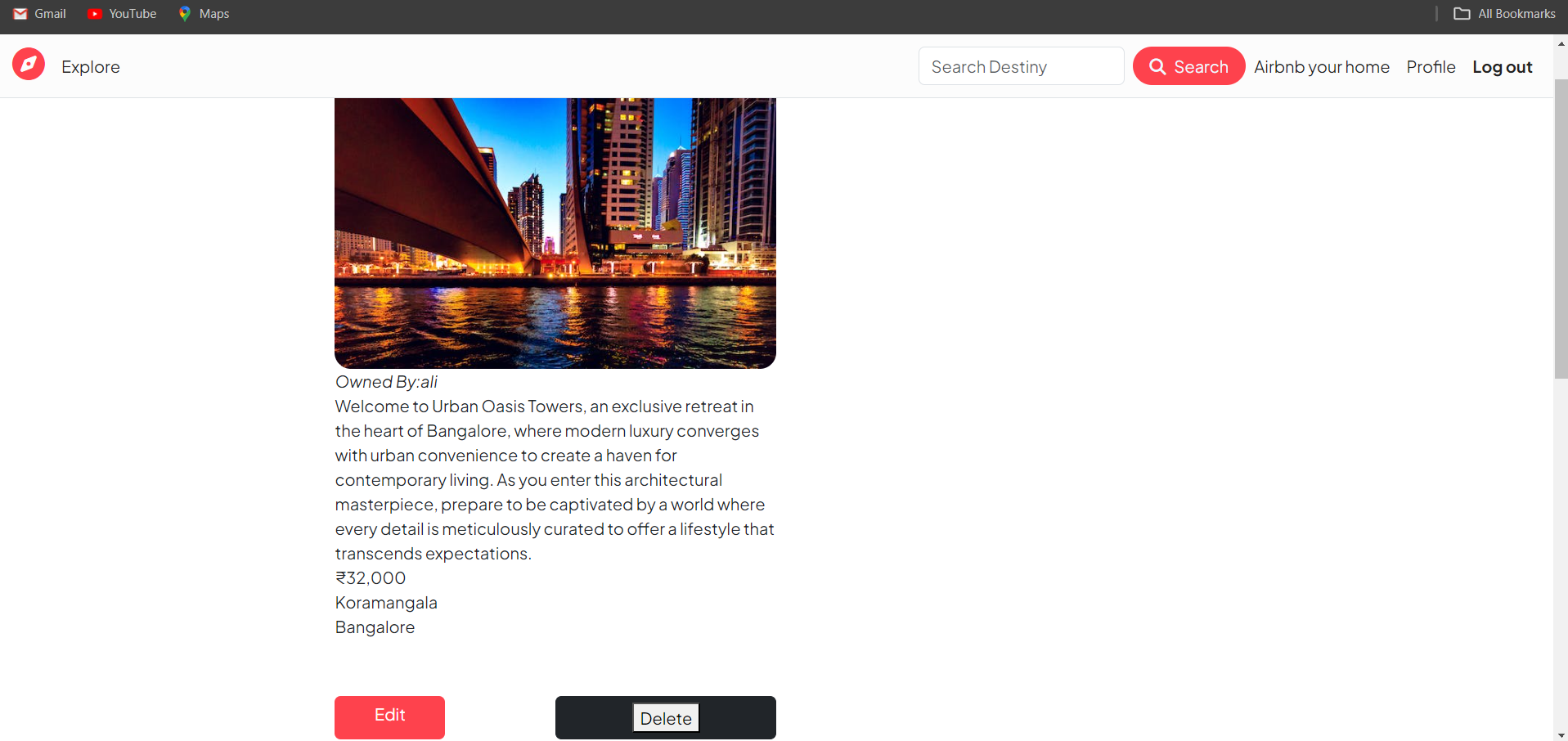


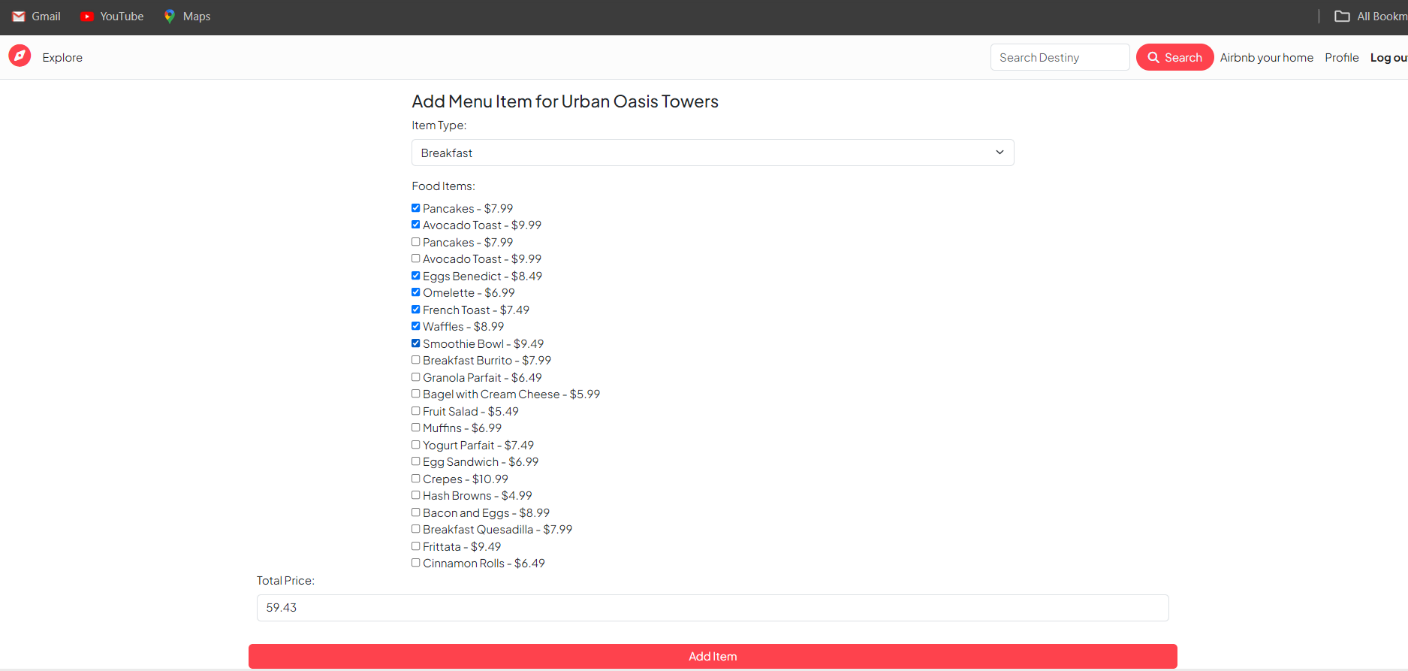


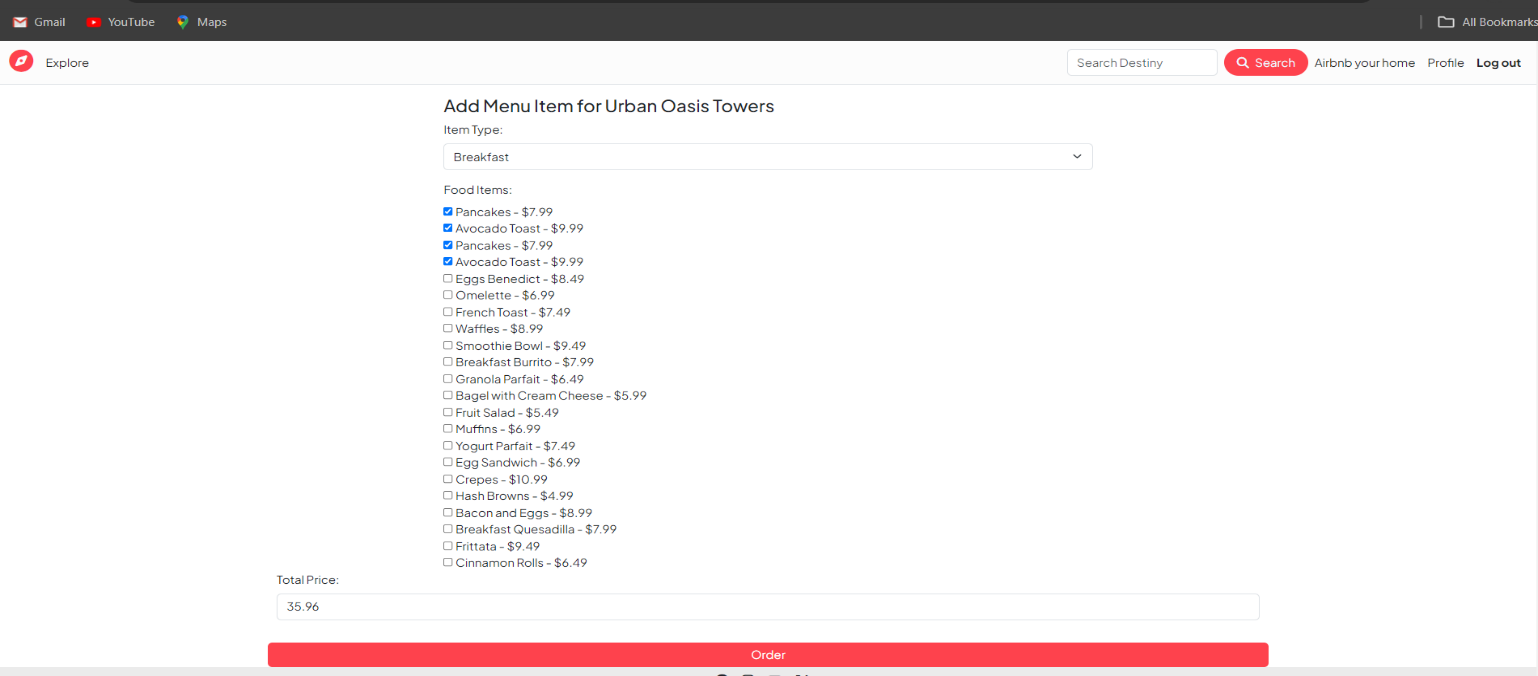
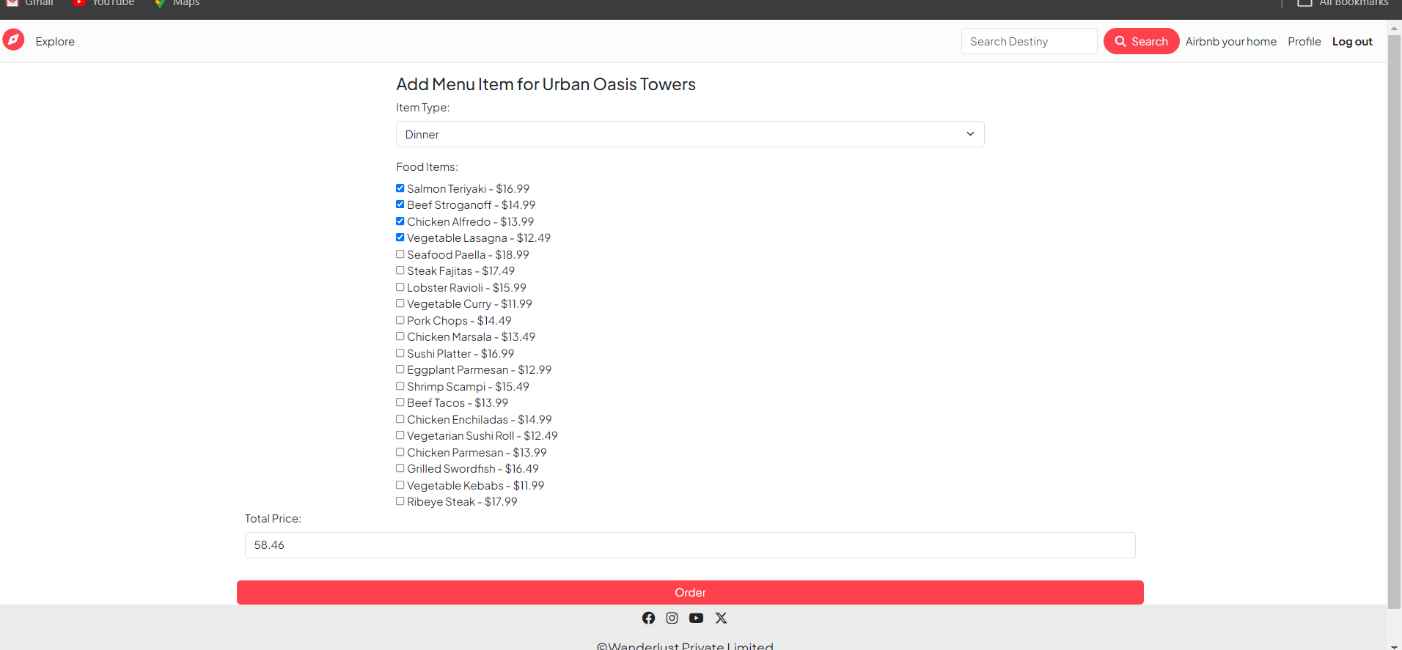












**Conclusion**

The Streamlining Room Management and Rentals project has been a transformative initiative aimed at revolutionizing the way we manage rental properties. Through a meticulous focus on efficiency, user experience, and automation, this project has successfully addressed key challenges in the rental property management process.

The centralized room management system has empowered property owners with a robust toolset, allowing them to effortlessly control and optimize their rental listings. The user-friendly rental platform ensures a seamless experience for both property owners and tenants, simplifying the process of finding, booking, and managing rental accommodations.

The implementation of an automated booking and reservation system has not only reduced the risk of errors but has also significantly improved the overall efficiency of the rental process. Real-time reservations, coupled with transparent pricing and secure payment integration, have elevated the level of trust and convenience for both property owners and tenants.

Effective tenant communication channels and support mechanisms have further strengthened the relationship between property owners and tenants, contributing to a positive and collaborative rental environment.

Ensuring regulatory compliance has been a cornerstone of this project, underscoring our commitment to ethical business practices and adherence to industry standards.

As we move forward, the scalable and flexible nature of the system positions us well for future growth and adaptability. The lessons learned from this project will inform our approach to similar endeavours, and the continuous feedback loop will drive ongoing enhancements and refinements.

In conclusion, Streamlining Room Management and Rentals stands as a testament to our dedication to innovation, efficiency, and providing an unparalleled experience in the realm of rental property management. This project not only streamlines processes but also elevates the standard of service, reinforcing our commitment to excellence in the ever-evolving landscape of property rentals**.**

**BIBLIOGRAPHY**

[MongoDB: The Definitive Guide, 3rd Edition] by Shannon Bradshaw, Eoin Brazil, and Kristina Chodorow. This book shows you the advantages of using document-oriented databases and how to use MongoDB 4.2 for various applications.

[Mastering MongoDB 4.x] by Marko Aleksendric, Arek Borucki, Leandro Domingues, Malak Abu Hammad, Elie Hannouch, Rajesh Nair, Rachelle Palmer. This book offers expert techniques to run high-volume and fault-tolerant database solutions using MongoDB 4.x³.

[MongoDB Basics] by Peter Membrey, David Hows, and Eelco Plugge. This book is a beginner-friendly introduction to MongoDB, covering the basics of installation, configuration, querying, and aggregation.

MongoDB: The Definitive Guide, 3rd Edition [Book]

Media. <https://www.oreilly.com/library/view/mongodb-the-definitive/9781491954454/>

MongoDB: The Definitive Guide, 3rd Edition [Book]

Media. <https://www.oreilly.com/library/view/mongodb-the-definitive/9781491954454/>

Best MongoDB Books for Beginners & Developers.

https://www.programmingcube.com/mongodb-books/

Best MongoDB Books for Beginners & Developers.

https://www.programmingcube.com/mongodb-books/.

MongoDB Press. https://www.mongodb.com/developer/books/.

MongoDB Press. https://www.mongodb.com/developer/books/.