Information Technology Project – IT2080

Assignment 02 - Progressive Review

ITP_Metro_06

Y2 S2

SLIIT Metropolitan Campus- Weekday Batch

Details of the Group Members: *(Provide the details of the group leader in the first row)*

	Name with Initials (Surname first)	**Registration Number**	**Contact Phone Number**	**Email**
1.	Appuhamy W.A.D.A.K.	IT23479746	0771429193	it23479746@my.sliit.lk
2.	Dilshan.N	IT23250574	0764645900	it23250574@my.sliit.lk
3.	########	########	#########	#########
4.	Rajapaksha R.H.L	IT23369160	0766034924	it23369160@my.sliit.lk
5.	DE MEL L.M.V.S.M.D.	IT23410572	0762790082	it23410572@my.sliit.lk

Table of Contents

2. Normalization Schema 2

3. Test Case Diagram 3

4. High-Level System design Diagram 4

5. Innovation Parts of the Project 5

I. User Account Management System 5

6. Commercialization 7

1. High Level Functions of the Project – Individual Contribution 10

1. Unified Vehicle Management System 10

MongoDB (No SQL) 11

2. User Account Management System 20

3. Safari Package Management 29

The Level of Completeness: 29

Work incomplete to be completed at the final: 30

MongoDB (No SQL) 30

4. Feedback System 41

The level of completeness for features: 41

Work incomplete to be completed at the final 41

MongoDB (No SQL) 42

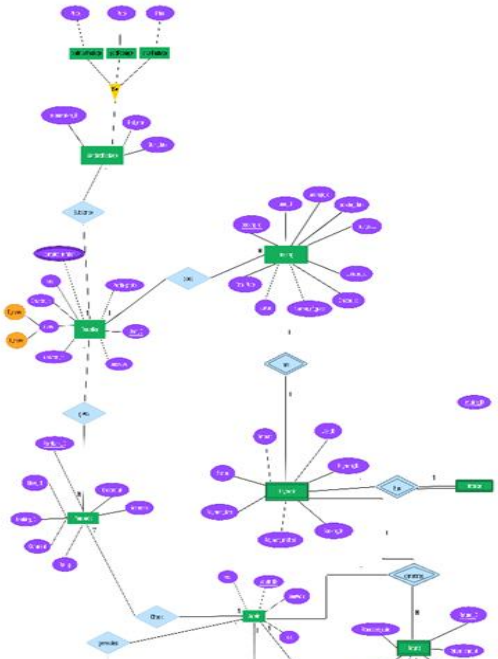
5. Booking & Payment System 51

The completion level of the features: 51

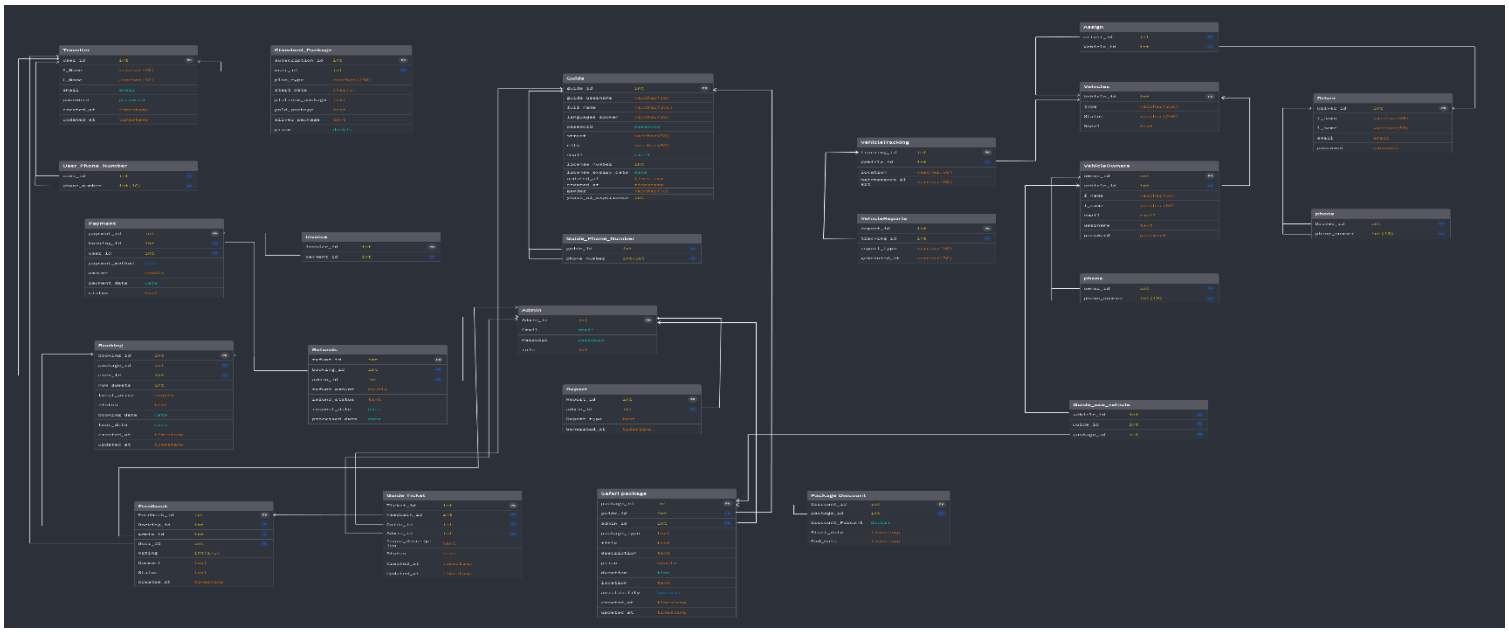
Work not completed and to be completed by final: 52

Git Hub Repository and Branches 65

1. ER Diagram

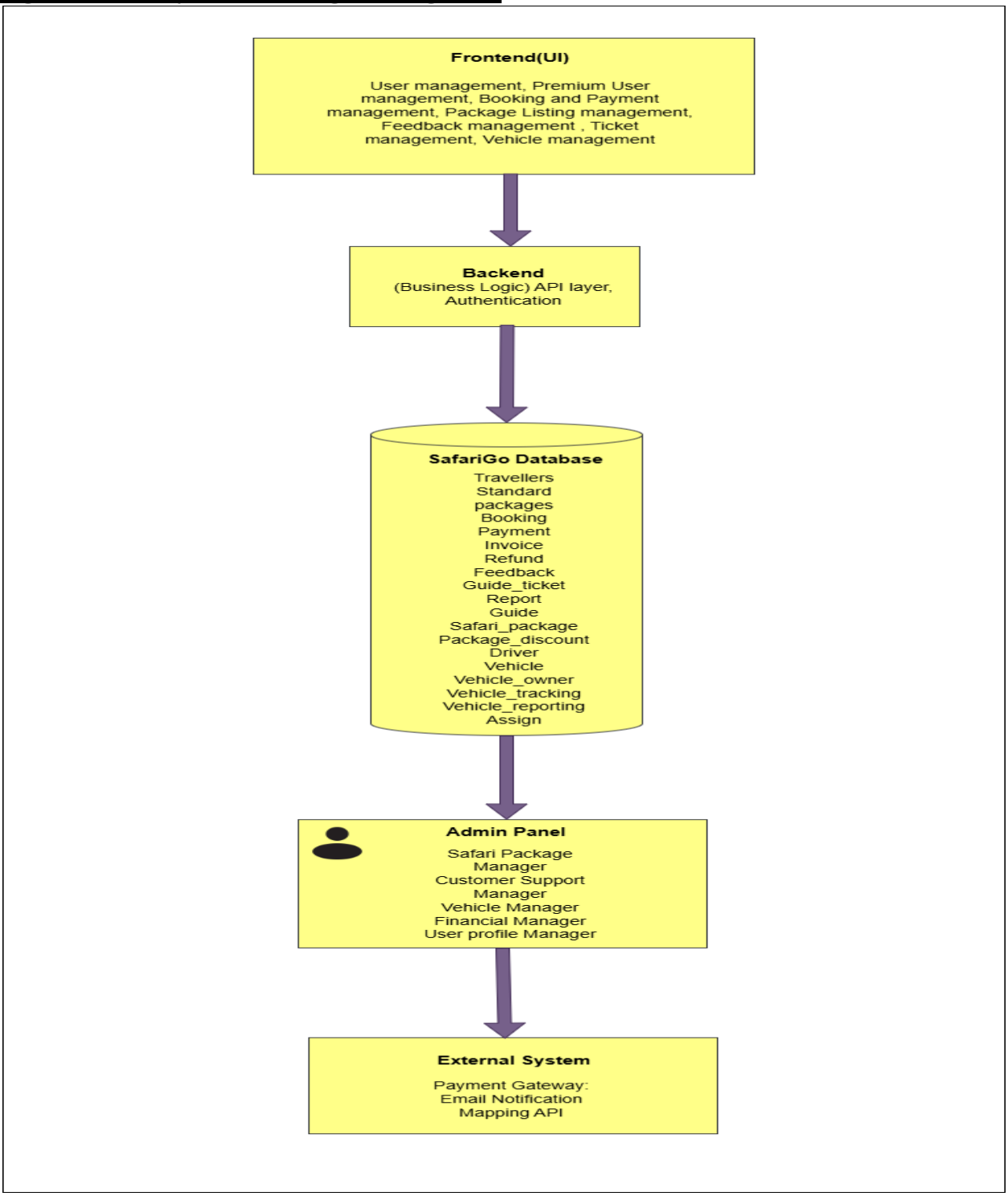


2. Normalization Schema



3. Test Case Diagram

4. High-Level System design Diagram



5. Innovation Parts of the Project

The SfariGo project is a web-based platform designed to connect multiple safari operators under one system, allowing users to compare options, bookings before arrival, and access transparent pricing. By digitizing previously manual processes, SafariGo aims to enhance the experience of everyone who wishes to enjoy a safari tour in Sri Lanka by providing the most curated and innovative platform for the users to select and explore the options available. The platform includes a wide range of modules, each targeting specific aspects of package browsing, personalized package booking, premium packages with added benefits to a convenient feedback system that allows customers to connect with the administration of the business. By integrating various administrative tasks into a single system, SfariGo enhances transparency, accountability, and service delivery for travelers, tour guides and vehicle owners.

I. User Account Management System

The User Account Management System permit users to register and log in with social media, Apple, or Google, with device synchronization happening automatically. The system also makes it simpler for new users by enabling guest reservations and automatically creating an account upon purchase. For the attraction of travelers the system provides loyalty points that can be exchanged for savings or other benefits in exchange for recurring reservations, reviews, and recommendations. As today's world moves towards convenience at every step of the way this system has enabled the use email, SMS, and WhatsApp to inform users of impending travel, due dates, and special deals.

II. Package Management System for Safari

The Package Management System allows the travelers to make a reservations and witness a virtual 360° safari preview of each package before booking the package. To make the package browsing and selection easier this system has an advanced in built filtering system that will allow the browser to choose the most appropriate the package to them according to their expected travelling dates, location they want to visit, number of participants and their budget. After each safari, customers can rate guides. To maximize income and availability, the system will showcase and give early reservations or off-peak hours discount.

III. Payment and Booking System

The Payment and Reservation System allows easy and swift payment and also sends instant confirmation of the booking status to the traveler. This system is also built in way that it saves user payment information safely for quicker reservations in the future. Since majority of the customers of SafariGo are multinationals, package costs will be displayed in a variety of currencies and convert them in real time. To make the booking process convenient and precise the availability of seats and vehicles in real time will also be shown to avoid overbooking. This system will accept local payment methods, Apple Pay, Google Pay, PayPal.

IV. Unified Vehicle Management System

The Vehicle Management System allocates vehicles efficiently based on group size, package selection, and user preferences, ensuring optimal resource use. After each safari, customers can rate drivers, promoting accountability and high service standards. This system handles the maintenance schedules which are automatically tracked, and with alerts they are sent to administrators when servicing is due, keeping the fleet in top condition. Additionally, users can opt for eco-friendly safari experiences by choosing hybrid or electric vehicles, supporting sustainable tourism initiatives.

V. Customer Feedback System

The Customer Feedback system lets users share feedback easily through voice recordings or short video submissions, offering a more personal and expressive way to communicate their experiences. To encourage participation, detailed reviews are rewarded with discounts or loyalty points, adding value for both users and the business. Ready-made response templates help staff respond quickly and appropriately to user feedback. Furthermore, a dedicated community section highlights top-rated reviews, travel tips, and user-generated safari experiences, encouraging engagement and fostering a sense of community among travelers

VI. Premium Account Management

SafariGo offers Premium accounts as well for customers who are regulars to tours. Premium members can enjoy a suite of exclusive benefits, including early booking access, special discounts, and first access to new safari packages. Premium safari packages include discounts, offers and early access to new packages and offers. Based on travel history and preferences, the system suggests the best available deals, making planning easier. Additionally, premium

accounts support group and family sharing, simplifying bookings for multiple travelers under one account.

VII. Ticket Raising System

The Ticket Raising system efficiently organizes tickets by type and urgency, ensuring prompt attention to issues such as booking errors or safari delays. For faster issue reporting, users can submit voice notes or video clips. Real-time updates keep users informed about the progress of their tickets, including estimated response times. As users type complaints, suggested solutions from FAQs appear instantly, helping to resolve issues quicker. Multi-language support ensures smooth communication, while callback requests allow users to receive help at their convenience. A public Q&A forum lets users find answers from both staff and fellow travelers. The system prioritizes urgent tickets based on service-level agreements (SLAs) and escalates unresolved issues automatically. After support is provided, users can rate their experience and share feedback, helping improve service quality.

VIII. Payment Refund System

The refund process is streamlined, offering instant partial refunds for eligible cancellations without the need for manual approval. Users receive a clear, detailed breakdown of their refund amount based on the applicable policy. To speed up the process, refunds can also be issued as wallet credits for future bookings. Instead of a cash refund, users can opt for flexible alternatives like rescheduling, credit vouchers, or even donating the refund to wildlife conservation efforts.

6. Commercialization

The commercial roll out of the Safari Booking System seeks to transform safari tourism by harnessing digital innovation to improve accessibility and streamline operations. With the global travel industry increasingly shifting toward online reservation platforms. This system addresses pressing issues such as the lack of dependable booking options, inefficient reservation processes, and delayed availability updates. By offering a unified and user-friendly digital interface, it aims to provide a smooth and enjoyable experience for travelers.

1. Market Opportunities

The primary users of the Safari Booking System include safari guides, wildlife parks, and travel companies focused on nature-based adventures. These groups need an organized and efficient tool to handle bookings, lighten administrative tasks, and improve customer happiness

A secondary audience consists of solo travelers who want a direct, trustworthy way to arrange safari trips without relying on middlemen. The growing worldwide enthusiasm for eco-friendly tourism and adventure experiences broadens the market's possibilities. As digital tools become more common in travel, there's also potential to reach new safari regions on the rise.

2. Business Framework

The Safari Booking System will function as a Software as a Service (SaaS) solution, where businesses can subscribe monthly or yearly. This setup offers affordability and flexibility, meeting the needs of both small operators and larger safari firms. Pricing will vary depending on chosen features, booking numbers, and support levels, giving users tailored options. Alongside this, a commission model will take a portion of each completed booking, aligning with common industry practices and offering operators a pay-as-you-go alternative to fixed costs.

3. Income Sources

To maintain financial health, the Safari Booking System will draw from several revenue channels:

- Subscription Payments: Regular fees from safari businesses using the SaaS platform.
- Booking Shares: A percentage earned from each finalized reservation.
- Custom Setup Fees: One-time costs for operators needing special features or links to their current tools.
- Highlighted Promotions: Revenue from subscriptions paid by businesses to showcase their offerings on the platform.
- Additional Services: Arrangements with insurance providers to provide optional travel insurance packages for customers at checkout.
- Data Insights: Fees for in-depth reports around consumer behavior, market trends, and business outcome

Overall, a combination of revenue sources promotes sustainability by diversifying financial dependencies.

4. Promotional Efforts

A well-rounded marketing plan will drive market entry and brand visibility:

- Digital Campaigns: Using social media, search engine tools, and focused ads to connect with safari operators and travelers.
- Key Alliances: Partnering with tourism councils, parks, and travel firms to embed the system in existing networks.
- Industry Showcases: Displaying the platform at travel fairs and tech events to win over business users.

- Reward Programs: Encouraging influencers and agencies to spread the word with performance-based incentives.

- Trial Phases: Providing free demos to select operators to prove the system's value before a full rollout.

These efforts aim to establish the platform as a vital resource in the safari tourism world.

5. Growth Plans

demand safari areas, with later expansion into growing eco-tourism zones. As the system builds a following, new features—

The launch will first target high-d like support for multiple languages, smart trip suggestions, and ties to lodging providers—could be added to enrich its offerings. Implementation as a phased rollout will improve the platform leveraging real user feedback prior to entering new market segments.

6. Risk and Challenges

Some issues could arise while bringing the Safari Booking System to market; each of which can be addressed with careful planning:

- System Compatibility: Ensuring that all will work with an operator's current booking and payment systems through flexibility.
- Data Protection: Using strong security to secure user and payment information.
- User Acceptance: Offering in-depth training and help to ease operators into adopting the system.

To tackle these issues, the approach will focus on ongoing research, user input, and working with industry pros.

1. High Level Functions of the Project – Individual Contribution

1. Unified Vehicle Management System

Appuhamy W.A.D.A.K- IT23479746

The **Unified Vehicle Management System (VMS)** streamlines the management of vehicles, vehicle owners, and drivers, ensuring seamless coordination and efficient resource allocation. This module allows vehicle owners to register their vehicles, drivers to manage their accounts, and administrators to oversee the entire fleet while maintaining transparency and security.

The level of completeness for features:

1. Backend Fully Implemented:
   All CRUD (Create, Read, Update, Delete) operations regarding Vehicle management have been implemented.

2. User authentication and login features are implemented to securely authenticate the Vehicle Owners.

3. Vehicle models and schema are present for Vehicle data management.

4. Connecting Backend to Frontend:
   The backend, with respect to the ticketing system, has to be integrated with the frontend to enable interaction between the Vehicle Owner and the Admin.

5. Vehicle Owner Panel Interface - In Progress:
   Front-end interface, which allows Vehicle Owner to view , update, delete Vehicles, is being worked on.

Work incomplete to be completed at the final:

1. Complete Integration of Backend and Frontend:
   The integration of backend to the frontend should be totally complete from Admin monitoring
2. Admin Monitoring Logic:
   Finalize the logic to be used by Admin in updating the vehicle status, such as to add vehicle to safari package.

MongoDB (No SQL)

```
const VehicleSchema = new

mongoose.Schema({

vehicle_id: { type: Number,

unique: true, required: true },

type: { type: String, required:

true },

state: { type: String, required:

true },

model: { type: String, required:

true }

});
```

```
const Vehicle =
mongoose.model("Vehicle",
VehicleSchema);
module.exports = Vehicle;


const VehicleOwnerSchema =
new mongoose.Schema({
owner_id: { type: Number,
unique: true, required: true },
vehicle_id: { type: Number, ref:
"Vehicle", required: true },
f_name: { type: String,
required: true },
l_name: { type: String,
required: true },
email: { type: String, required:
true, unique: true },
username: { type: String,
required: true, unique: true },
password: { type: String,
required: true } // Should be
hashed
});

const VehicleOwner =
mongoose.model("VehicleOwn
er", VehicleOwnerSchema);
```

```
module.exports =
VehicleOwner;


const VehicleReportsSchema =
new mongoose.Schema({
report_id: { type: Number,
unique: true, required: true },
tracking_id: { type: Number,
ref: "VehicleTracking",
required: true },
report_type: { type: String,
required: true },
generated_at: { type: String,
required: true }
});

const VehicleReports =
mongoose.model("VehicleRepo
rts", VehicleReportsSchema);
module.exports =
VehicleReports;
```

```
// BACKEND/controllers/vehicleController.js
const Vehicle = require('../models/Vehicle');
const User = require('../models/User');
```

```
const Booking = require('../models/Booking');

exports.getAllVehicles = async (req, res) => {
try {
const vehicles = await Vehicle.find().populate('owner', 'email name');
res.status(200).json(vehicles);
} catch (error) {
res.status(500).json({ error: 'Error fetching vehicles' });
}
};

exports.getVehicleById = async (req, res) => {
try {
const vehicle = await Vehicle.findById(req.params.id).populate('owner', 'email name');
if (!vehicle) return res.status(404).json({ message: 'Vehicle not found' });
res.status(200).json(vehicle);
} catch (error) {
res.status(500).json({ error: 'Error fetching vehicle' });
}
};

exports.createVehicle = async (req, res) => {
const { type, licensePlate } = req.body;
try {
const owner = req.user.id; // Use authenticated user's ID
const user = await User.findById(owner);
if (!user || (user.role !== 'vehicle_owner' && user.role !== 'admin')) {
return res.status(400).json({ message: 'Invalid or unauthorized user' });
}
const serviceReports = req.files['serviceReports'] ? req.files['serviceReports'].map(file =>
file.path) : [];
```

```
const license = req.files['license'] ? req.files['license'][0].path : null;
const insurance = req.files['insurance'] ? req.files['insurance'][0].path : null;
const vehicle = new Vehicle({ type, licensePlate, owner, serviceReports, license, insurance
});
await vehicle.save();
res.status(201).json({ message: 'Vehicle added', vehicle });
} catch (error) {
console.error('Error adding vehicle:', error); // Log detailed error
res.status(400).json({ error: 'Error adding vehicle', details: error.message });
}
};

exports.updateVehicle = async (req, res) => {
try {
const vehicle = await Vehicle.findById(req.params.id);
if (!vehicle) return res.status(404).json({ message: 'Vehicle not found' });
if (req.user.role !== 'admin' && vehicle.owner.toString() !== req.user.id) {
return res.status(403).json({ message: 'Not authorized' });
}
// NEW: Handle file uploads for updates
const updateData = { ...req.body };
if (req.files['serviceReports']) updateData.serviceReports =
req.files['serviceReports'].map(file => file.path);
if (req.files['license']) updateData.license = req.files['license'][0].path;
if (req.files['insurance']) updateData.insurance = req.files['insurance'][0].path;
const updatedVehicle = await Vehicle.findByIdAndUpdate(req.params.id, updateData, {
new: true });
res.status(200).json({ message: 'Vehicle updated', vehicle: updatedVehicle });
} catch (error) {
res.status(400).json({ error: 'Error updating vehicle' });
}
```

```
};

exports.deleteVehicle = async (req, res) => {
try {
const vehicle = await Vehicle.findById(req.params.id);
if (!vehicle) return res.status(404).json({ message: 'Vehicle not found' });
await Vehicle.findByIdAndDelete(req.params.id);
res.status(200).json({ message: 'Vehicle deleted' });
} catch (error) {
res.status(500).json({ error: 'Error deleting vehicle' });
}
};

exports.getVehiclesByOwner = async (req, res) => {
try {
const vehicles = await Vehicle.find({ owner: req.user.id });
res.status(200).json(vehicles);
} catch (error) {
res.status(500).json({ error: 'Error fetching vehicles' });
}
};

exports.toggleVehicleAvailability = async (req, res) => {
try {
const vehicle = await Vehicle.findById(req.params.id);
if (!vehicle) return res.status(404).json({ message: 'Vehicle not found' });
if (vehicle.owner.toString() !== req.user.id) {
return res.status(403).json({ message: 'Not authorized' });
}
vehicle.isAvailable = !vehicle.isAvailable;
await vehicle.save();
```

```
res.status(200).json({ message: 'Availability updated', vehicle });
} catch (error) {
res.status(500).json({ error: 'Error updating availability' });
}
};

exports.getVehicleUsageStats = async (req, res) => {
try {
const stats = await Booking.aggregate([
{ $match: { vehicle: { $exists: true } } },
{ $group: { _id: '$vehicle', totalBookings: { $sum: 1 } } },
{ $lookup: { from: 'vehicles', localField: '_id', foreignField: '_id', as: 'vehicle' } },
{ $unwind: '$vehicle' },
{ $project: { _id: 0, vehicleType: '$vehicle.type', totalBookings: 1 } },
]);
res.status(200).json(stats);
} catch (error) {
res.status(500).json({ error: 'Error fetching usage stats' });
}
};

exports.getAvailabilityReport = async (req, res) => {
try {
const available = await Vehicle.countDocuments({ isAvailable: true });
const unavailable = await Vehicle.countDocuments({ isAvailable: false });
res.status(200).json({ available, unavailable });
} catch (error) {
res.status(500).json({ error: 'Error generating report' });
}
};
```

```
exports.getMaintenanceAlerts = async (req, res) => {
try {
const today = new Date();
const alerts = await Vehicle.find({
nextMaintenanceDate: { $lte: today },
}).populate('owner', 'email name');
res.status(200).json(alerts);
} catch (error) {
res.status(500).json({ error: 'Error fetching maintenance alerts' });
}
};
```

```
// BACKEND/models/Vehicle.js
const mongoose = require('mongoose');

const VehicleSchema = new mongoose.Schema({
type: { type: String, required: true },
licensePlate: { type: String, required: true, unique: true },
owner: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
isAvailable: { type: Boolean, default: true },
// NEW: Added fields for PDFs (stored as file paths or URLs)
serviceReports: [{ type: String }], // Array of file paths for service reports
license: { type: String }, // File path for license PDF
insurance: { type: String }, // File path for insurance PDF
maintenanceRecords: [{
date: { type: Date, default: Date.now },
description: String,
}],
```

```
nextMaintenanceDate: { type: Date },
});

module.exports = mongoose.model('Vehicle', VehicleSchema);
```

```
// BACKEND/models/VehicleOwnerProfile.js
const mongoose = require('mongoose');

const VehicleOwnerProfileSchema = new mongoose.Schema({
userId: {
type: mongoose.Schema.Types.ObjectId,
ref: 'User',
required: true,
unique: true,
},
name: { type: String, required: true },
companyName: { type: String },
// REMOVED: 'vehicles' array, as vehicle data is now in the Vehicle model
});

const VehicleOwnerProfile = mongoose.model('VehicleOwnerProfile',
VehicleOwnerProfileSchema);
module.exports = VehicleOwnerProfile;
```

2. User Account Management System

Dilshan.N – IT23250574

This feature is designed to manage the registration, authentication, and account control of users within the system, including the ability to upgrade to premium accounts. The system allows users to register, login, update their profile, and manage their account status. Additionally, premium account functionalities are provided, enabling users to access exclusive services upon successful subscription. The administrative interface facilitates the management of all user accounts and their premium status.

Premium account management allows users to view subscription packages, make payments, and enjoy additional features tied to their premium status. Admins are enabled to oversee and update user subscription statuses when necessary.

Level of Completeness:

1) **Backend Operations are fully implemented** – Full Create, Read, Update, and Delete (CRUD) operations for user accounts and premium subscriptions have been completed.
2) **Backend Validations are implemented** – All necessary validations for user input, account updates, and premium subscription handling are in place.
3) **Frontend Developed** – User interface for standard user operations (registration, login, profile management) and premium subscription interactions is developed according to backend functionalities.

4) **User Models and Schema present** – Comprehensive schema to manage both regular and premium user data is developed and in use.
5) **Working on Premium Subscription Interface** – Developing enhanced frontend components for managing subscriptions, handling payments, and displaying premium features. Backend functions such as subscription status updates and automated notification/email systems are under development.

Work Incomplete to be Completed at Final:

1) **Integration of Frontend and Backend** – Connect frontend interface with backend APIs for seamless user and subscription management.
2) **Develop Features** – Implement file upload for profile pictures and generate reports on user and subscription statistics.
3) **Improve UI/UX Design** – Refine interface for ease of use, accessibility, and enhanced user engagement, especially for premium services.

MongoDB (No SQL)

Controllers / UserContraller.js

```
//BACKEND/controllers/userContraller.js
const User = require('../models/User');
const CustomerProfile = require('../models/CustomerProfile');
const GuideProfile = require('../models/GuideProfile');
const VehicleOwnerProfile = require('../models/VehicleOwnerProfile');
const Notification = require('../models/Notification');
const jwt = require('jsonwebtoken');

exports.registerCustomer = async (req, res) => {
  try {
    const { email, password, name, Lname, Gender, Phonenumber1, Phonenumber2 } =
req.body;
    const profilePicture = req.file ? req.file.path : '';

    let user = await User.findOne({ email });
    if (user) return res.status(400).json({ message: 'Email already registered'
});

    user = new User({ email, password, role: 'user' });
    await user.save();
```

```
    const customerProfile = new CustomerProfile({
      userId: user._id,
      name,
      Lname,
      Gender,
      Phonenumber1,
      Phonenumber2,
      profilePicture,
    });
    await customerProfile.save();

    res.status(201).json({ message: 'Customer registered successfully' });
  } catch (error) {
    res.status(500).json({ error: 'Error registering customer' });
  }
};

exports.registerGuide = async (req, res) => {
  try {
    const { email, password, name, experienceYears, specialties } = req.body;

    let user = await User.findOne({ email });
    if (user) return res.status(400).json({ message: 'Email already registered'
});

    user = new User({ email, password, role: 'guide' });
    await user.save();

    const guideProfile = new GuideProfile({
      userId: user._id,
      name,
      experienceYears,
      specialties: specialties.split(','), // Assuming specialties come as a
comma-separated string
    });
    await guideProfile.save();

    res.status(201).json({ message: 'Guide registered successfully' });
  } catch (error) {
    res.status(500).json({ error: 'Error registering guide' });
  }
};
```

```
exports.registerVehicleOwner = async (req, res) => {
  try {
    const { email, password, name, companyName, vehicles } = req.body;

    let user = await User.findOne({ email });
    if (user) return res.status(400).json({ message: 'Email already registered'
});

    user = new User({ email, password, role: 'vehicle_owner' });
    await user.save();

    const vehicleOwnerProfile = new VehicleOwnerProfile({
      userId: user._id,
      name,
      companyName,
      vehicles: JSON.parse(vehicles), // Assuming vehicles come as a JSON string
    });
    await vehicleOwnerProfile.save();

    res.status(201).json({ message: 'Vehicle Owner registered successfully' });
  } catch (error) {
    res.status(500).json({ error: 'Error registering vehicle owner' });
  }
};

exports.login = async (req, res) => {
  try {
    const { email, password } = req.body;
    const user = await User.findOne({ email });

    if (!user || !(await user.comparePassword(password))) {
      return res.status(400).json({ message: 'Invalid email or password' });
    }

    const token = jwt.sign({ id: user._id, role: user.role },
process.env.JWT_SECRET, { expiresIn: '1h' });

    res.status(200).json({ message: 'Login successful', token, role: user.role
});
  } catch (error) {
    res.status(500).json({ error: 'Error logging in' });
  }
};
```

```
exports.getProfile = async (req, res) => {
  try {
    let profile;
    switch (req.user.role) {
      case 'user':
        profile = await CustomerProfile.findOne({ userId: req.user.id });
        break;
      case 'guide':
        profile = await GuideProfile.findOne({ userId: req.user.id });
        break;
      case 'vehicle_owner':
        profile = await VehicleOwnerProfile.findOne({ userId: req.user.id });
        break;
      case 'admin':
        profile = { role: 'admin' }; // Minimal data for admin
        break;
      default:
        return res.status(400).json({ message: 'Invalid role' });
    }
    if (!profile) return res.status(404).json({ message: 'Profile not found' });
    res.status(200).json(profile);
  } catch (error) {
    res.status(500).json({ error: 'Error fetching profile' });
  }
};

// ... (existing imports and exports)

exports.updateProfile = async (req, res) => {
  try {
    const { name, Lname, Gender, Phonenumber1, Phonenumber2 } = req.body;
    const profilePicture = req.file ? req.file.path : undefined;

    let profile = await CustomerProfile.findOne({ userId: req.user.id });
    if (!profile) return res.status(404).json({ message: "Profile not found" });

    profile.name = name || profile.name;
    profile.Lname = Lname || profile.Lname;
    profile.Gender = Gender || profile.Gender;
    profile.Phonenumber1 = Phonenumber1 || profile.Phonenumber1;
    profile.Phonenumber2 = Phonenumber2 || profile.Phonenumber2;
    if (profilePicture) profile.profilePicture = profilePicture;

    await profile.save();
    res.status(200).json(profile);
```

```
  } catch (error) {
    res.status(500).json({ error: "Error updating profile" });
  }
};

exports.deleteProfile = async (req, res) => {
  try {
    await User.findByIdAndDelete(req.user.id);
    await CustomerProfile.findOneAndDelete({ userId: req.user.id });
    res.status(200).json({ message: "Account deleted successfully" });
  } catch (error) {
    res.status(500).json({ error: "Error deleting account" });
  }
};

exports.subscribeToPlan = async (req, res) => {
  try {
    const { plan } = req.body;
    if (!["platinum", "gold", "silver"].includes(plan)) {
      return res.status(400).json({ message: "Invalid plan" });
    }

    const profile = await CustomerProfile.findOne({ userId: req.user.id });
    if (!profile) return res.status(404).json({ message: "Profile not found" });

    profile.plan = plan;
    await profile.save();

    // Create notification for admin
    const user = await User.findById(req.user.id);
    const notification = new Notification({
      message: `User ${user.email} subscribed to ${plan} plan`,
    });
    await notification.save();

    res.status(200).json({ message: "Subscribed to plan successfully", plan });
  } catch (error) {
    res.status(500).json({ error: "Error subscribing to plan" });
  }
};
```

Models / Customer.js

```
//BACKEND/models/Customer.js
```

```
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs'); // Import bcrypt for password hashing

//Correct capitalization of Schema
const Schema = mongoose.Schema;

// Define the schema with email and password
const CustomerSchema = new Schema({
    name: {
        type: String,
        required: true,
    },

    Lname: {
        type: String,
        required:  false,
    },
    Gender: {
        type: String,
        required:  false,
    },
    Phonenumber1: {
        type: Number,
        required:  false,
    },
    Phonenumber2: {
        type: Number,
        required:  false,
    },
    email: {
        type: String,
        required: true,
        unique: true, // Ensure that email is unique
        match: [/\S+@\S+\.\S+/, 'Please use a valid email address'], // Basic
email validation
    },
    password: {
        type: String,
        required: true,
        minlength: 6, // Minimum password length
    },
    profilePicture: {
        type: String, // URL to the profile picture
        required: false,
    },
```

```
});

// Create a method to compare passwords during login
CustomerSchema.methods.comparePassword = async function (candidatePassword) {
    try {
        return await bcrypt.compare(candidatePassword, this.password); // Compare
provided password with hashed password
    } catch (err) {
        throw new Error('Password comparison failed');
    }
};

// Create the model
const Customer = mongoose.model('Customer', CustomerSchema);

module.exports = Customer;
```

Models / CustomerProfile.js

```
//BACKEND/models/CustomerProfile.js
const mongoose = require("mongoose");

const CustomerProfileSchema = new mongoose.Schema({
  userId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "User",
    required: true,
    unique: true,
  },
  name: { type: String, required: true },
  Lname: { type: String },
  Gender: { type: String },
  Phonenumber1: { type: String },
  Phonenumber2: { type: String },
  profilePicture: { type: String },
  plan: {
    type: String,
    enum: ["platinum", "gold", "silver"],
    default: "silver",
  },
});

const CustomerProfile = mongoose.model("CustomerProfile", CustomerProfileSchema);
module.exports = CustomerProfile;
```

Routes / userRoutes.js

```
const express = require("express");
const multer = require("multer");
const path = require("path");
const { auth, authorize } = require("../middleware/auth");
const {
  registerCustomer,
  registerGuide,
  registerVehicleOwner,
  login,
  getProfile,
  updateProfile,
  deleteProfile,
  subscribeToPlan,
} = require("../controllers/userController");

const router = express.Router();

const storage = multer.diskStorage({
  destination: "./uploads/",
  filename: (req, file, cb) => {
    cb(null, `${file.fieldname}-
${Date.now()}${path.extname(file.originalname)}`);
  },
});
const upload = multer({ storage });

router.post("/register/customer", upload.single("profilePicture"),
registerCustomer);
router.post("/register/guide", registerGuide);
router.post("/register/vehicle_owner", registerVehicleOwner);
router.post("/login", login);
router.get("/profile", auth, getProfile);
router.put("/profile", auth, upload.single("profilePicture"), updateProfile);
router.delete("/profile", auth, deleteProfile);
router.post("/subscribe", auth, subscribeToPlan);

// New endpoints
router.get('/notifications', auth, authorize(['admin']), async (req, res) => {
  const Notification = require('../models/Notification');
  try {
    const notifications = await Notification.find().sort({ createdAt: -1 });
```

```
    res.status(200).json(notifications);
  } catch (error) {
    res.status(500).json({ error: 'Error fetching notifications' });
  }
});

router.get('/all', auth, authorize(['admin']), async (req, res) => {
  try {
    const users = await User.find().select('-password');
    res.status(200).json(users);
  } catch (error) {
    res.status(500).json({ error: 'Error fetching users' });
  }
});

module.exports = router;
```

3. Safari Package Management

T.R Fernando – IT23400122

This feature is designed to allow safari guides to manage and update safari packages. The system enables guides to create, update, and remove safari packages, making sure that customers can view up-to-date information. Additionally, safari package managers can also approve or reject packages before they are published.

The Level of Completeness:

1. Backend Operations – CRUD operations are fully implemented.

2. Backend Validations are implemented.

3. Basic UI is developed according to backend operations.

4. A structured database model for safari packages and guides is implemented.

5. Working on Package Approval interface – Developing the interface for Safari Package Managers to view and approve packages.

Work incomplete to be completed at the final:

1. Integration of Frontend and Backend.

2. Developing features that will enable image or file uploads for safari package details.

3. Improve UI/UX design to make it a user-friendly system.

MongoDB (No SQL)

packageModel.js

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

const SafariPackageSchema = new mongoose.Schema({
  package_id: { type: Number, unique: true, required: true },
  name: { type: String, required: true },
  description: { type: String },
  price: { type: Number, required: true },
  duration: { type: Number, required: true },
  location: { type: String, required: true },
  availability: { type: Boolean, default: true },
```

```
  guide_id: { type: mongoose.Schema.Types.ObjectId, ref: 'Guide', required: true
},
  package_type: { type: String, required: true },
  image_url: { type: String },
  status: { type: String, enum: ['pending', 'approved', 'declined'], default:
'pending' },
  createdAt: { type: Date, default: Date.now },
  updatedAt: { type: Date, default: Date.now }
});

const SafariPackage = mongoose.model('SafariPackage', SafariPackageSchema);
module.exports = SafariPackage;
```

PackageController.js

```
const SafariPackage = require('../models/SafariPackage');

// Create a new package
exports.createPackage = async (req, res) => {
  try {
    const maxPackage = await SafariPackage.findOne().sort({ package_id: -1 });
    const newPackageId = maxPackage ? maxPackage.package_id + 1 : 1;

    const packageData = {
      package_id: newPackageId,
      ...req.body,
      guide_id: req.user.guideId,
      status: 'pending'
    };
```

```
    const newPackage = new SafariPackage(packageData);
    await newPackage.save();
    res.status(201).json(newPackage);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
};

// Update a package
exports.updatePackage = async (req, res) => {
  try {
    const package = await SafariPackage.findById(req.params.id);
    if (!package) return res.status(404).json({ message: 'Package not found' });
    if (package.guide_id.toString() !== req.user.guideId.toString()) {
      return res.status(403).json({ message: 'Not authorized to update this
package' });
    }
    Object.assign(package, req.body);
    package.status = 'pending';
    package.updatedAt = Date.now();
    await package.save();
    res.json(package);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
};

//  Delete a package
exports.deletePackage = async (req, res) => {
  try {
    const package = await SafariPackage.findById(req.params.id);
    if (!package) return res.status(404).json({ message: 'Package not found' });
    if (package.guide_id.toString() !== req.user.guideId.toString()) {
      return res.status(403).json({ message: 'Not authorized to delete this
package' });
    }
    await package.remove();
    res.json({ message: 'Package deleted' });
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
};

// View all packages
exports.getAllPackages = async (req, res) => {
```

```
  try {
    const packages = await SafariPackage.find().populate('guide_id', 'name');
    res.json(packages);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
};

// Approve a package
exports.approvePackage = async (req, res) => {
  try {
    const package = await SafariPackage.findById(req.params.id);
    if (!package) return res.status(404).json({ message: 'Package not found' });
    package.status = 'approved';
    package.updatedAt = Date.now();
    await package.save();
    res.json(package);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
};

// Decline a package
exports.declinePackage = async (req, res) => {
  try {
    const package = await SafariPackage.findById(req.params.id);
    if (!package) return res.status(404).json({ message: 'Package not found' });
    package.status = 'declined';
    package.updatedAt = Date.now();
    await package.save();
    res.json(package);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
};

//View approved packages
exports.getApprovedPackages = async (req, res) => {
  try {
    const packages = await SafariPackage.find({ status: 'approved'
}).populate('guide_id', 'name');
    res.json(packages);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
```

```
};

// View their own packages
exports.getGuidePackages = async (req, res) => {
  try {
    const packages = await SafariPackage.find({ guide_id: req.user.guideId });
    res.json(packages);
  } catch (error) {
    res.status(500).json({ messag


e: error.message });
  }
};
```

Routes

PackageRoutes.js

```
const express = require('express');
const multer = require('multer');
const path = require('path');
const {
    createPackage,
    updatePackage,
    deletePackage,
    getAllPackages,
```

```
    approvePackage,
    declinePackage,
    getApprovedPackages,
    getGuidePackages
} = require('../controllers/packageController');

const router = express.Router();

const storage = multer.diskStorage({
    destination: './uploads/',
    filename: (req, file, cb) => {
        cb(null, file.fieldname + '-' + Date.now() +
path.extname(file.originalname));
    }
});
const upload = multer({ storage });

// Routes
router.post('/', upload.single('image'), createPackage);

router.put('/:id', upload.single('image'), updatePackage);

router.delete('/:id', deletePackage);

router.get('/all', getAllPackages);

router.put('/:id/approve', approvePackage);

router.put('/:id/decline', declinePackage);

router.get('/', getApprovedPackages);

router.get('/guide/:guideId', getGuidePackages);

module.exports = router;
```

Pseudocodes

```
FUNCTION SafariPackageManagementSystem():

WHILE user is authenticated:

 IF user is a Tourist:
```

```
            CALL TouristInterface()

ELSE IF user is a TourOperator:

           CALL TourOperatorInterface()

END IF

END WHILE



FUNCTION TouristInterface():

DISPLAY language selection options

GET user language preference

SET interface language to user preference

DISPLAY main menu options
GET user choice

SWITCH user choice:
    CASE "View Available Packages":
        CALL ViewPackages()
    CASE "Exit":
        EXIT function
END SWITCH


FUNCTION ViewPackages():

RETRIEVE list of available safari packages from database

FOR EACH package in packages:

          DISPLAY package name

          DISPLAY description

          DISPLAY price

          DISPLAY duration
```

```
END FOR


FUNCTION TourOperatorInterface():

DISPLAY main menu options

 GET user choice

SWITCH user choice:
    CASE "Manage Packages":
        CALL ManagePackages()
    CASE "Update Package Status":
        CALL UpdatePackageStatus()
    CASE "View Booking History":
        CALL ViewBookingHistory()
    CASE "Exit":
        EXIT function
END SWITCH


FUNCTION ManagePackages():

DISPLAY package management options

GET user choice

SWITCH user choice:
    CASE "Add Package":
        CALL AddPackage()
    CASE "Modify Package":
        CALL ModifyPackage()
    CASE "Remove Package":
        CALL RemovePackage()
    CASE "Back":
        RETURN to TourOperatorInterface()
END SWITCH




FUNCTION AddPackage():
```

```
INPUT new package details (name, description, price, duration,
availability)

VALIDATE input

IF input is valid:

      ADD new package to database

      DISPLAY "Package added successfully"

ELSE:

      DISPLAY "Invalid input. Please try again"

END IF


FUNCTION ModifyPackage():

DISPLAY list of existing packages

GET package to modify

DISPLAY current package details

INPUT modified package details

VALIDATE input

IF input is valid:

             UPDATE package in database

            DISPLAY "Package updated successfully"

ELSE:

          DISPLAY "Invalid input. Please try again"

END IF


FUNCTION RemovePackage():

DISPLAY list of existing packages
```

```
GET package to remove
PROMPT for confirmation
 IF confirmed:
          REMOVE package from database
          DISPLAY "Package removed successfully"
ELSE:
          DISPLAY "Removal cancelled"
END IF



FUNCTION UpdatePackageStatus():
INPUT package identifier
 RETRIEVE package details from database
DISPLAY current status
INPUT new status
UPDATE package status in database
 LOG status change with timestamp
            DISPLAY "Package status updated successfully"

FUNCTION ViewBookingHistory():
INPUT search criteria (date range, customer, package type, etc.)
 RETRIEVE booking history from database based on criteria
 DISPLAY booking history summary
```

```
OFFER option to export data or view detailed records


FUNCTION LogPackageInteraction(package_id, interaction_details):
RETRIEVE package from database
ADD interaction_details to package log
UPDATE package in database
DISPLAY "Interaction logged successfully"
```

4. Feedback System

R.H.L. Rajapaksha - IT23369160

This feature allows customers to submit feedback once they have finished the safari. which is then reviewed by an admin. The admin assigns the feedback to a guide as a ticket, who is responsible for addressing the customer's concerns. This system ensures a structured and efficient approach to managing customer feedback, ultimately improving customer satisfaction and service quality.

The level of completeness for features:

1. CRUD (Create, Read, Update, Delete) operations regarding feedback management have been implemented.
2. Basic UI is developed according to backend operations.
3. A structured database model for feedback and ticket is implemented.
4. Working on implementing a ticket management system where feedback tickets are assigned to guides.

Work incomplete to be completed at the final

1. Complete Integration of Backend and Frontend.
2. Improvements in UI/UX Design.
3. Integration of Frontend and Backend.
4. Implementation of Guide response to customer.

MongoDB (No SQL)

routers

Feedback.js

```
const express = require('express');
const router = express.Router();
const FeedbackController = require('../controllers/feedbackController');

// Route to submit feedback
router.post('/submit', FeedbackController.submitFeedback);

// Route to get all feedbacks
router.get('/', FeedbackController.getAllFeedbacks);

// Route to update feedback status
router.put('/:id/status', FeedbackController.updateFeedbackStatus);

module.exports = router;
```

Ticket.js

```
const express = require('express');
const router = express.Router();
const TicketController = require('../controllers/ticketController');
```

```
// Route to submit a ticket
router.post('/submit', TicketController.submitTicket);

// Route to get all tickets
router.get('/', TicketController.getAllTickets);

// Route to update ticket status
router.put('/:id/status', TicketController.updateTicketStatus);

module.exports = router;
```

Models

Feedback.js

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

const feedbackSchema = new mongoose.Schema({
    customerEmail: {type: String, required: true},
    rating: {type: Number, required: true},// Rating from 1 to 5
    comment: {type: String, required: true},
    safariPackageId: {type: mongoose.Schema.Types.ObjectId, ref: 'SafariPackage',
required: true},
    status: {type: String, default: 'Pending'}, // Default is 'Pending'
    assignedGuideId: {type: mongoose.Schema.Types.ObjectId, ref: 'Guide'}, //
Guide assigned to feedback
    createdAt: {type: Date, default: Date.now }
});

module.exports = mongoose.model('Feedback', feedbackSchema);
```

Ticket.js

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

const ticketSchema = new mongoose.Schema({
    feedbackId: { type: mongoose.Schema.Types.ObjectId, ref: 'Feedback',
required: true },
```

```
    guideId: { type: mongoose.Schema.Types.ObjectId, ref: 'Guide', required: true
},  // Guide to whom the ticket is assigned
    issueDescription: { type: String, required: true },
    status: { type: String, default: 'Assigned' },  // Ticket status (e.g.,
Pending, Assigned, Closed)
    createdAt: { type: Date, default: Date.now }
});

module.exports = mongoose.model('Ticket', ticketSchema);
```

Controllers

Feedback.js

```
const Feedback = require('../models/Feedback');

// Submit feedback
exports.submitFeedback = async (req, res) => {
    try {
        const { customerEmail, rating, comment, safariPackageId, assignedGuideId
} = req.body;

        // Validate required fields
        if (!customerEmail || !rating || !comment || !safariPackageId) {
            return res.status(400).json({ message: 'Customer email, rating,
comment, and safari package are required'});
        }

        // Rating should be between 1 and 5
        if (rating < 1 || rating > 5) {
            return res.status(400).json({message: 'Rating must be between 1 and
5'});
        }

        const newFeedback = new Feedback({
            customerEmail,
            rating,
            comment,
```

```
            safariPackageId,
            assignedGuideId,          });

        await newFeedback.save();
        res.status(201).json({message: 'Feedback submitted successfully'});
    } catch (error) {
        res.status(400).json({error: error.message });
    }
};

// Get all feedback
exports.getAllFeedbacks = async (req, res) => {
    try {
        const {page = 1, limit = 10} = req.query;
        const feedbacks = await Feedback.find()
            .populate('safariPackageId assignedGuideId')
            .skip((page - 1) * limit)
.limit(Number(limit)); // Limit results per page

        res.json(feedbacks);
    } catch (error) {
        res.status(500).json({error: error.message });
    }
};

// Get feedback by ID
exports.getFeedbackById = async (req, res) => {
    try {
        const feedback = await
Feedback.findById(req.params.id).populate('safariPackageId assignedGuideId');
        if (!feedback) {
            return res.status(404).json({message: 'Feedback not found'});
        }
        res.json(feedback);
    } catch (error) {
        res.status(500).json({error: error.message });
    }
};

// Update feedback status
exports.updateFeedbackStatus = async (req, res) => {
    try {
        const {status} = req.body;

        if (!status) {
```

```
            return res.status(400).json({message: 'Status is required'});
        }

        const updatedFeedback = await Feedback.findByIdAndUpdate(
            req.params.id,
            {status},
            {new: true}
        );

        if (!updatedFeedback) {
            return res.status(404).json({message: 'Feedback not found'});
        }

        res.json({message: 'Feedback status updated successfully',
updatedFeedback });
    } catch (error) {
        res.status(400).json({error: error.message });
    }
};

// Delete feedback
exports.deleteFeedback = async (req, res) => {
    try {
        const deletedFeedback = await Feedback.findByIdAndDelete(req.params.id);

        if (!deletedFeedback) {
            return res.status(404).json({message: 'Feedback not found'});
        }

        res.json({message: 'Feedback deleted successfully'});
    } catch (error) {
        res.status(500).json({error: error.message });
    }
};
```

TicketController.

is

```
//BACKEND/controllers/ticketController.js

const Ticket = require('../models/Ticket');
```

```
// Submit a ticket
exports.submitTicket = async (req, res) => {
    try {
        const { feedbackId, guideId, issueDescription, status } = req.body;

        // Validate required fields
        if (!feedbackId || !guideId || !issueDescription) {
            return res.status(400).json({message: 'Feedback ID, guide ID, and
issue description are required'});
        }

        const newTicket = new Ticket({
            feedbackId,
            guideId,
            issueDescription,
            status: status || 'Assigned', // Default status is 'Assigned' if not
provided
        });

        await newTicket.save();
        res.status(201).json({message: 'Ticket submitted successfully'});
    } catch (error) {
        res.status(400).json({error: error.message });
    }
};

// Get all tickets
exports.getAllTickets = async (req, res) => {
    try {
        const tickets = await Ticket.find()
            .populate('feedbackId guideId')
            .exec();
        res.json(tickets);
    } catch (error) {
        res.status(500).json({error: error.message });
    }
};

// Get a specific ticket by ID
exports.getTicketById = async (req, res) => {
    try {
        const ticket = await Ticket.findById(req.params.id)
            .populate('feedbackId guideId');

        if (!ticket) {
```

```
            return res.status(404).json({ message: 'Ticket not found'});
        }
        res.json(ticket);
    } catch (error) {
        res.status(500).json({error: error.message });
    }
};

// Update ticket status
exports.updateTicketStatus = async (req, res) => {
    try {
        const {status } = req.body;

        if (!status) {
            return res.status(400).json({message: 'Status is required'});
        }

        const updatedTicket = await Ticket.findByIdAndUpdate(
            req.params.id,
            {status},
            {new: true}
        );

        if (!updatedTicket) {
            return res.status(404).json({message: 'Ticket not found'});
        }

        res.json({message: 'Ticket status updated successfully', updatedTicket
});
    } catch (error) {
        res.status(400).json({ error: error.message });
    }
};

// Delete ticket
exports.deleteTicket = async (req, res) => {
    try {
        const deletedTicket = await Ticket.findByIdAndDelete(req.params.id);

        if (!deletedTicket) {
            return res.status(404).json({ message: 'Ticket not found' });
        }

        res.json({ message: 'Ticket deleted successfully' });
```

```
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
};
```

Pseudocodes

```
FUNCTION FeedbackManagementSystem():
    WHILE user is authenticated:
        IF user is a Customer:
            CALL CustomerInterface()
        ELSE IF user is an Admin:
            CALL AdminInterface()
        ELSE IF user is a Guide:
            CALL GuideInterface()
        END IF
    END WHILE


FUNCTION CustomerInterface():
    DISPLAY "Enter your feedback"
    INPUT feedback_text
    SUBMIT feedback_text to database
    DISPLAY "Feedback submitted successfully"


FUNCTION AdminInterface():
    DISPLAY "1. View Feedback"
    DISPLAY "2. Assign Feedback to Guide"
    DISPLAY "3. Exit"
    INPUT choice
```

```
    SWITCH choice:
        CASE 1:
            CALL ViewFeedback()
        CASE 2:
            CALL AssignFeedback()
        CASE 3:
            EXIT function
    END SWITCH


FUNCTION ViewFeedback():
    RETRIEVE all feedback entries from database
    FOR EACH feedback in feedback_list:
        DISPLAY feedback ID, feedback text, status
    END FOR


FUNCTION AssignFeedback():
    CALL ViewFeedback()
    DISPLAY "Enter Feedback ID to assign"
    INPUT feedback_id
    DISPLAY "Enter Guide ID"
    INPUT guide_id
    UPDATE feedback entry in database with assigned guide_id
    DISPLAY "Feedback assigned successfully"


FUNCTION GuideInterface():
    RETRIEVE assigned feedback from database
    DISPLAY assigned feedback list

    DISPLAY "Enter Feedback ID to respond"
    INPUT feedback_id
    DISPLAY "Enter Response"
```

INPUT response_text

UPDATE feedback entry in database with response_text

DISPLAY "Response submitted successfully"

5. Booking & Payment System

DE MEL L. M. V. S. M. D. – IT23410572

This function covers the entire process of the booking completion. From the instance where the customer browses the safari packages, filters out the unnecessary options, customizes the package to the point where the customer makes the payment.

The completion level of the features:

1. Backend Fully Implemented: All CRUD (Create, Read, Update, Delete) operations for the Booking & Payment system.

2. In the Process of Connecting Backend to Frontend: Currently, the backend is being connected to the frontend interface.
3. Customer Home Page in Progress: The front-end part for the customer to land on the Home Page

Work not completed and to be completed by final:

1. Full Integration of Backend and Frontend: The connection between the backend and the frontend needs to be fully completed and tested.
2. Resident Submission Validation: The validation logic for resident document submissions needs to be finalized.
3. UI/UX Design Enhancements: Final polishing of the UI is still required for better user experience.

MongoDB Queries (No SQL)

```
//BACKEND/config/db.js

const mongoose = require("mongoose");


const connectDB = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGODB_URL, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log(`MongoDB Connected: ${conn.connection.host}`);
  } catch (err) {
    console.error(`Error: ${err.message}`);
    process.exit(1);
  }
};
```

module.exports = connectDB;

Controllers

Booking

```
//BACKEND/controllers/bookingController.js
const Booking = require('../models/Booking');
const SafariPackage = require('../models/SafariPackage');
const User = require('../models/User');

// Create a new booking
exports.createBooking = async (req, res) => {
  try {
    const { package_id, num_guests, tour_date } = req.body;

    // Validate package
    const safariPackage = await SafariPackage.findOne({ package_id });
    if (!safariPackage) {
      return res.status(404).json({ message: 'Safari package not found' });
    }

    // Validate user
    const user = await User.findById(req.user.id);
    if (!user) {
      return res.status(404).json({ message: 'User not found' });
    }

    // Check if the tour date is available
    const isDateAvailable = safariPackage.available_dates.some(
      (date) => new Date(date).toISOString().split('T')[0] === new
Date(tour_date).toISOString().split('T')[0]
    );
    if (!isDateAvailable) {
      return res.status(400).json({ message: 'Selected date is not available' });
    }

    // Check if the number of guests is within the package limit
    if (num_guests > safariPackage.max_participants) {
      return res.status(400).json({ message: 'Number of guests exceeds package limit'
});
    }

    const total_price = safariPackage.price * num_guests;
```

```
    const booking = new Booking({
      package_id: safariPackage.package_id,
      user_id: user._id,
      num_guests,
      total_price,
      booking_date: new Date(),
      tour_date,
      status: 'pending',
    });

    await booking.save();
    res.status(201).json(booking);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
};

// Get booking details by ID
exports.getBookingById = async (req, res) => {
  try {
    const booking = await Booking.findOne({ booking_id: req.params.booking_id })
      .populate('package_id', 'name location price')
      .populate('user_id', 'email');
    if (!booking) {
      return res.status(404).json({ message: 'Booking not found' });
    }
    res.json(booking);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
};
```

Payment

```
//BACKEND/controllers/paymentController.js
const Payment = require('../models/Payment');
const Booking = require('../models/Booking');
const multer = require('multer');
const path = require('path');

// Multer setup for payment slip upload
const storage = multer.diskStorage({
  destination: './uploads/payment_slips/',
  filename: (req, file, cb) => {
    cb(null, `slip-${Date.now()}${path.extname(file.originalname)}`);
  },
});
```

```
const upload = multer({ storage });

// Create a new payment
exports.createPayment = async (req, res) => {
  try {
    const { booking_id, payment_method, amount, card_details } = req.body;

    // Validate booking
    const booking = await Booking.findOne({ booking_id });
    if (!booking) {
      return res.status(404).json({ message: 'Booking not found' });
    }

    // Validate amount
    if (amount !== booking.total_price) {
      return res.status(400).json({ message: 'Payment amount does not match booking
total' });
    }

    const paymentData = {
      booking_id: booking.booking_id,
      user_id: req.user.id,
      payment_method,
      amount,
      status: payment_method === 'bank_transfer' ? 'pending' : 'completed',
    };

    if (payment_method === 'card' && card_details) {
      paymentData.card_details = card_details;
    }

    if (payment_method === 'bank_transfer' && req.file) {
      paymentData.payment_slip = req.file.path;
    }

    const payment = new Payment(paymentData);
    await payment.save();

    // Update booking status
    booking.status = 'confirmed';
    await booking.save();

    res.status(201).json(payment);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
```

```
};

// Get payment history for a user
exports.getPaymentHistory = async (req, res) => {
  try {
    const payments = await Payment.find({ user_id: req.user.id });
    res.json(payments);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
};
```

Models

Booking.js

```
 //BACKEND/models/Booking.js
const mongoose = require('mongoose');


const BookingSchema = new mongoose.Schema({
  userId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true,
  },
  safariId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Safari', // Assuming a Safari model exists
    required: true,
  },
  Fname: { type: String, required: true },
  Lname: { type: String, required: true },
  Phonenumber1: { type: String },
  email: { type: String, required: true },
});


const Booking = mongoose.model('Booking', BookingSchema);
module.exports = Booking;
```

Payment.js

```
//BACKEND/models/Payment.js
const mongoose = require('mongoose');

const PaymentSchema = new mongoose.Schema({
  payment_id: {
    type: Number,
    unique: true,
    required: true,
  },
  booking_id: {
    type: Number,
    required: true,
    ref: 'Booking', // Reference to Booking model
  },
  user_id: {
    type: Number,
    required: true,
    ref: 'User', // Reference to User model
  },
  payment_method: {
    type: String,
    enum: ['card', 'paypal', 'bank_transfer'],
    required: true,
  },
  amount: {
    type: Number,
    required: true,
  },
  payment_date: {
    type: Date,
    default: Date.now,
  },
  status: {
    type: String,
    enum: ['pending', 'completed', 'failed'],
    default: 'pending',
  },
  payment_slip: {
    type: String, // Path to the uploaded payment slip (for bank transfers)
    required: false,
  },
  card_details: {
    last_four: String,
```

```
    expiry: String,
  },
});

// Pre-save middleware to auto-increment payment_id
PaymentSchema.pre('save', async function (next) {
  if (this.isNew) {
    const lastPayment = await this.constructor.findOne().sort({ payment_id: -1 });
    this.payment_id = lastPayment ? lastPayment.payment_id + 1 : 1;
  }
  next();
});

const Payment = mongoose.model('Payment', PaymentSchema);
module.exports = Payment;
```

Routes

Booking.js

```
 //BACKEND/models/Booking.js
const mongoose = require('mongoose');


const BookingSchema = new mongoose.Schema({
  userId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true,
  },
  safariId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Safari', // Assuming a Safari model existas
    required: true,
  },
  Fname: { type: String, required: true },
  Lname: { type: String, required: true },
  Phonenumber1: { type: String },
  email: { type: String, required: true },
});


const Booking = mongoose.model('Booking', BookingSchema);
module.exports = Booking;
```

payment.js

```
//BACKEND/routes/payment.js
const router = require('express').Router();
const { createBooking, getBookingById } =
require('../controllers/bookingController');
const { auth, authorize } = require('../middleware/auth');

router.post('/add', auth, authorize(['user']), createBooking);
router.get('/:booking_id', auth, authorize(['user', 'admin']), getBookingById);

// Existing routes (update, delete, etc.) remain unchanged
router.get('/', auth, authorize(['admin']), async (req, res) => {
  try {
    const bookings = await Booking.find();
    res.json(bookings);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});

router.put('/update/:bookingid', auth, authorize(['user', 'admin']), async (req, res)
=> {
  try {
    const booking = await Booking.findById(req.params.bookingid);
    if (!booking || (booking.user_id.toString() !== req.user.id && req.user.role !==
'admin')) {
      return res.status(403).json({ message: 'Access denied' });
    }
    const updatedBooking = await Booking.findByIdAndUpdate(req.params.bookingid,
req.body, { new: true });
    res.status(200).json({ status: 'Booking Updated', booking: updatedBooking });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});

router.delete('/delete/:bookingid', auth, authorize(['user', 'admin']), async (req,
res) => {
  try {
    const booking = await Booking.findById(req.params.bookingid);
```

```
    if (!booking || (booking.user_id.toString() !== req.user.id && req.user.role !==
'admin')) {
      return res.status(403).json({ message: 'Access denied' });
    }
    await Booking.findByIdAndDelete(req.params.bookingid);
    res.status(200).json({ status: 'Booking Deleted' });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});

module.exports = router;
```

Pseudo Code

Booking a package

FUNCTION bookSafariPackage()

// Initialize state variables

SET safari = package data from navigation state

SET bookingDetails = { num_guests: 1, tour_date: "" }

SET booking = null

SET error = ""

// Function to handle booking submission

FUNCTION handleBooking(event)

PREVENT default form submission

TRY:

POST booking data TO "http://localhost:8070/bookings/add":

package_id = safari.package_id

num_guests = bookingDetails.num_guests

tour_date = bookingDetails.tour_date

headers = { Authorization: Bearer token from localStorage }

SET booking = response data

SET error = ""

```
    CATCH error:
      SET error = error message from response or "Error creating booking"

  // Render the booking form or summary
  RENDER:
    IF safari is not available:
      DISPLAY "No safari package selected"
    ELSE IF booking is null:
      DISPLAY form:
        INPUT number for num_guests (min: 1, max: safari.max_participants)
        SELECT dropdown for tour_date (options: safari.available_dates)
        IF error exists:
          DISPLAY error message
        BUTTON to submit form (calls handleBooking)
    ELSE:
      // Booking Summary
      DISPLAY "Booking Summary" heading
      DISPLAY card with:
        Package name
        Location
        Number of guests
        Tour date
        Total price
        Status
      DISPLAY "Make Payment" button:
        ON_CLICK: navigate to "/payment/{booking.booking_id}"

END FUNCTION
```

Making payment

```
FUNCTION paymentDashboard()
    // Initialize state variables
    SET booking_id = URL parameter
    SET booking = null
    SET paymentMethod = "card"
    SET cardDetails = { number: "", expiry: "", cvv: "" }
    SET paymentSlip = null
    SET savedPayments = empty list
    SET error = ""
    SET successMessage = ""

    // Fetch booking and payment history on component mount
    ON_COMPONENT_MOUNT:
        TRY:
            FETCH booking FROM "http://localhost:8070/bookings/{booking_id}"
            SET booking = fetched data
            FETCH payment history FROM "http://localhost:8070/payments/history"
            SET savedPayments = fetched data
        CATCH error:
            SET error = "Error fetching booking or payment history"

    // Function to handle payment submission
    FUNCTION handlePayment(event)
        PREVENT default form submission
        SET formData = new FormData
        ADD to formData:
```

```
        booking_id
        payment_method
        amount = booking.total_price
    IF paymentMethod is "card":
        ADD card_details to formData:
            last_four = last 4 digits of cardDetails.number
            expiry = cardDetails.expiry
    IF paymentMethod is "bank_transfer" AND paymentSlip exists:
        ADD payment_slip to formData
    TRY:
        POST formData TO "http://localhost:8070/payments/add":
            headers = { Authorization: Bearer token, Content-Type: multipart/form-data }
        SET successMessage = "Payment processed successfully!"
        SET_TIMEOUT: navigate to "/UserHomepage" after 2 seconds
    CATCH error:
        SET error = error message from response or "Error processing payment"

// Render the payment dashboard
RENDER:
    IF booking is null:
        DISPLAY "Loading..."
    ELSE:
        DISPLAY "Payment for Booking #{booking.booking_id}" heading
        DISPLAY booking summary:
            Package name
            Location
            Total amount
        DISPLAY payment method buttons:
            BUTTON "Card" (sets paymentMethod to "card")
            BUTTON "PayPal" (sets paymentMethod to "paypal")
```

```
    BUTTON "Bank Transfer" (sets paymentMethod to "bank_transfer")
  DISPLAY form:
    IF paymentMethod is "card":
      INPUT for card number
      INPUT for expiry (MM/YY)
      INPUT for CVV
    IF paymentMethod is "bank_transfer":
      INPUT file for payment slip
    IF savedPayments exists:
      SELECT dropdown for saved payment methods
    IF error exists:
      DISPLAY error message
    IF successMessage exists:
      DISPLAY success message
    BUTTON to submit form (calls handlePayment)

END FUNCTION
```

Git Hub Repository and Branches

Main : https://github.com/Dilshan-Nadeeranga/SAFARI_GO.git

IT Number & Name	Member Branch
Appuhamy W. A. D. A. K. IT23479746	https://github.com/Dilshan-Nadeeranga/SAFARI_GO/tree/AlokaNew/BACKEND https://github.com/Dilshan-Nadeeranga/SAFARI_GO/tree/AlokaNew/frontend
Dilshan.N IT23250574	https://github.com/Dilshan-Nadeeranga/SAFARI_GO/tree/Dilshan/BACKEND https://github.com/Dilshan-Nadeeranga/SAFARI_GO/tree/Dilshan/frontend
Kaveesha K.D.K IT22075994	https://github.com/Dilshan-Nadeeranga/SAFARI_GO/tree/Riana/BACKEND https://github.com/Dilshan-Nadeeranga/SAFARI_GO/tree/Riana/frontend
R.H.L. Rajapaksha IT23369160	https://github.com/Dilshan-Nadeeranga/SAFARI_GO/tree/Lakshika/BACKEND https://github.com/Dilshan-Nadeeranga/SAFARI_GO/tree/Lakshika/frontend
DE MEL L.M.V.S.M.D. IT23410572	https://github.com/Dilshan-Nadeeranga/SAFARI_GO/tree/Shivani/BACKEND https://github.com/Dilshan-Nadeeranga/SAFARI_GO/tree/Shivani/frontend

Appendix

1. User Management System (IT23250574)

User Story	Completed
As a customer, I want to Implement user registration and login (secure authentication)	Yes
As a customer, I want to subscribe to a premium package.	Yes
As a user profile manager, I want to create a report about premium users so that I can get an idea about earnings for premium users.	No

2. Guide Package Listing System (IT23400122)

High level user story	Completed
As a Guide, I want to sign up or log into my account so that I can add safari packages	Yes
As a Guide, I want to update my availability and I want to contact the customers before the Safari.	Yes
As a safari Package manager, I want to view packages so that I can approve guide packages.	No

3. Booking and Payment for Customer (IT23410572)

User Story	completed
As a customer, I want to view the availability of the safari packages so then I can choose a package.	Yes
As a Financial manager, I want to verify payments.	No
As a financial manager I want to generate invoices after the payment is done by the customer	No
As a customer, I want to receive a confirmation of my booking and be able to cancel or make any adjustments to the booking if needed	Yes
As a Financial Manager, I want to refund the necessary amount if a customer cancels their booking.	No

4. Unified Vehicle Management System (IT23479746)

User Story	Completed
As a Vehicle Owner, I want to register and manage my profile so that I can list my vehicles	Yes
As a Vehicle Owner I want to edit my listed vehicles so that I can keep my information up to date.	Yes
As a Guide I want to search for available vehicles so that safari tours have designated vehicles.	No
As a vehicle owner, I want to view a report of my earnings so that I can track my revenue from safari bookings.	No
As a vehicle admin, I want to generate reports on vehicle availability and so that I can optimize fleet management	Yes

5. Reviews and Feedback System (IT23369160)

User Story	Completed
As a Customer, I want to add reviews/feedback.	Yes
As a Customer Care Manager, I want to respond to customer feedback to improve service quality.	Yes
As a Customer Care Manager, I can raise a ticket for guides based on customer feedback so that issues can be tracked and resolved efficiently.	No

6.Reports and Analytics (IT23250574, IT23479746)

User Story	Completed
As a User Profile Manager, I want to create a report about premium users so that I can get insights on earnings from premium users.	No
As a Vehicle Owner, I want to generate a report on my earnings from safari bookings.	No