LocalGuide Proposal 2

-AI-Powered C2C Tour Guide Platform

Zhi Kang|Aarti Pandya|Sehjalpreet|Dong Zhang

Software Engineering Course Project | February 2026 | Group 3

---

1. Project Status Update

1.1 Progress Since Proposal 1

After finishing requirements analysis in Proposal 1, our team has moved into the system design phase. We followed the recommended software process from the course, going through the modeling activity (analysis and design) before starting construction.

Progress Since Proposal 1

Phase	Duration	Status
Requirements Analysis (Proposal 1)	Week 1-2	Completed
System Design (Proposal 2)	Week 3-4	In progress
Core Development — Sprint 1	Week 3-4	Planned

Core Development — Sprint 2-3	Week 5-8	Planned
Feature Enhancement & Testing	Week 9-12	Completed

We are using **Agile Scrum**. Scrum uses short sprints (we chose 2-week sprints) so the team can deliver working increments and get feedback early.

---

2. Design Goals

Design goals define what the system must achieve to run smoothly and meet user expectations. Each goal below addresses a core quality requirement for the LocalGuide Connect platform.

2.1 Design Goal Mapping

Design Goal	Quality Requirement	How It Will Be Achieved
DG-1: Easy Search & Booking	Performance Usability	Composite index on city + category in MySQL; paginated search results; simple 3-step booking flow in Vue.js
DG-2: Safe & Secure	Security	Passwords hashed with bcrypt; all traffic over HTTPS; JWT tokens for session management; input validation on all forms
DG-3: Clear & Compliant Payment	Security Reliability	Stripe handles all card data; no sensitive payment info stored

		locally; clear booking confirmation and email receipts
DG-4: Clean & Maintainable Code Structure	Maintainability Testability	Backend organized into Controller → Service → Repository layers; unit tests for core business logic; modular Vue.js components
DG-5: Fast & Reliable System	Performance Availability	Vue.js SPA served as static files via Nginx; MySQL indexes on frequently queried fields; system designed to handle expected concurrent users without crashes

2.2 Design Principles

Our design follows key principles from the course:

1. **Separation of Concerns:** Frontend (Vue.js) communicates with backend (Spring Boot) only through REST APIs, allowing independent development and deployment. (Supports DG-4, NFR-M-01)
2. **Open-Closed Principle (OCP):** Service components can be extended (e.g., new tour categories, new payment methods) without modifying existing code. (Supports DG-4)
3. **Dependency Inversion Principle:** Service layer depends on Repository interfaces (abstractions), not concrete database implementations. Spring Data JPA provides implementations at runtime. (Supports DG-4, NFR-M-02)
4. **High Cohesion, Low Coupling:** Each service class focuses on one area (e.g., BookingService only handles booking logic). Components communicate through well-defined interfaces. (Supports DG-4)
5. **Security by Design:** Authentication and authorization are enforced at the filter layer (AuthFilter) before any business logic executes — no endpoint is reachable without a valid JWT. (Supports DG-2, NFR-S-01 ~ NFR-S-07)
6. **Progressive Enhancement:** MVP delivers core features first; advanced features are designed as plug-in modules for future sprints, following the incremental development approach from the course. (Supports DG-4, NFR-M-01)

---

3. System Architecture

3.1 Architecture Overview

LocalGuide Connect uses a **three-tier client-server architecture** (Architectural Design):

- **Presentation Layer (Client Tier):** Vue.js SPA — handles user interface
- **Application Layer (Business Logic Tier):** Spring Boot backend — handles business rules
- **Data Layer:** MySQL database — stores persistent data

As discussed in the course, in a 3-tier architecture the presentation layer never communicates directly with the data layer. All requests pass through the middleware layer.

The backend follows the **MVC pattern** (Spring MVC lecture): the Controller handles HTTP requests, the Model represents data, and the View is replaced by JSON responses since we build a REST API serving a separate Vue.js frontend.

The layered architecture (Controller → Service → Repository) follows the **call-return architectural style** and supports **layer cohesion** — higher layers access lower layers, but not the reverse.

3.2 Technology Stack

Layer	Technology	Justification
Frontend	Vue.js 3 + Vite + Pinia + Element Plus	Reactive UI; team has self-study experience
HTTP Client	Axios	JWT interceptor support
Backend	Spring Boot 3.2 (Java 17)	Auto-configures Spring MVC, DB, security (as covered in lecture)
Security	Spring Security 6.x + JWT	Standard auth framework

ORM	Spring Data JPA + Hibernate	Repository interfaces; Spring generates SQL (as taught in JPA lecture)
Database	MySQL 8.0+	Team has experience; free tier on AWS RDS
Build Tool	Maven	Manages dependencies via pom.xml (POM, as discussed in lecture)
Payment	Stripe Java SDK	PCI-compliant; test mode available
Email	Spring Mail + SendGrid	Free tier (100 emails/day)
Testing	JUnit 5 + Mockito / Vitest	Standard Java & Vue testing
CI/CD	GitHub Actions	Automated build and test
Deployment	AWS EC2 + Docker + Nginx	Containerized; free tier eligible

3.3 API Design

The **DispatcherServlet** (Spring MVC lecture) routes all HTTP requests to the appropriate Controller.

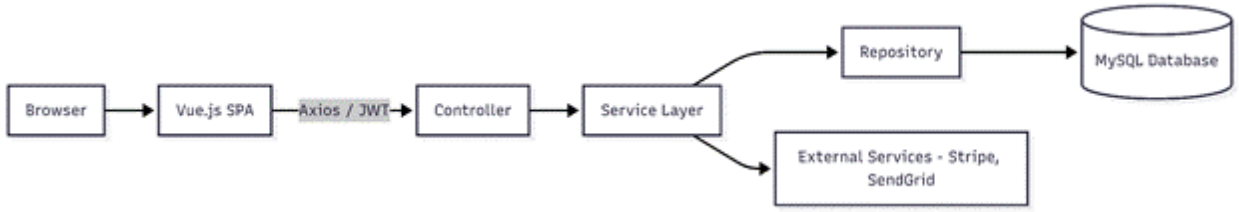
Base URL: https://api.localguide.com/api/v1

Module/Controller	Key Endpoints	Methods
Auth	/auth/register, /auth/login, /auth/refresh	POST
Tourist	/tourists/me, /tourists/me/favorites	GET, PUT, POST, DELETE
Guide	/guides, /guides/{id}, /guides/me, /guides/me/verification	GET, POST, PUT
Tours	/guides/{id}/tours, /tours/{id}	GET, POST, PUT, DELETE
Search	/search/guides, /search/tours	GET
Bookings	/bookings, /bookings/{id}, /bookings/{id}/cancel	GET, POST, PUT
Payments	/payments/create-intent, /payments/webhook	POST
Reviews	/tours/{id}/reviews, /reviews/{id}/reply	GET, POST
Availability	/guides/me/availability	GET, POST, PUT, DELETE
Admin	/admin/verifications, /admin/users, /admin/analytics	GET, PUT, DELETE

All protected endpoints require Authorization: Bearer <JWT> header. Total: **28 endpoints**.

3.4 Component Interaction

Each Controller delegates to its Service, and each Service accesses data through its Repository — following the **layered architectural style**:



---

4. Hardware and Software Configuration

4.1 Development Environment

Component	Specification
OS	Windows 11
IDE	Visual Studio Code (Frontend & Backend)
Java	JDK 17
Node.js	v20 LTS
Database (Local)	MySQL 8.0
API Testing	Postman / Swagger UI
Version Control	Git + GitHub

4.2 Production Environment (AWS)

Component	What We Use	Cost (CAD/month)

Server	AWS EC2(free tier)	$0
Database	AWS RDS MySQL(free tier)	$0 (free tier)
File Storage	AWS S3	~$1
Total	—	**~$1/month** (free tier)

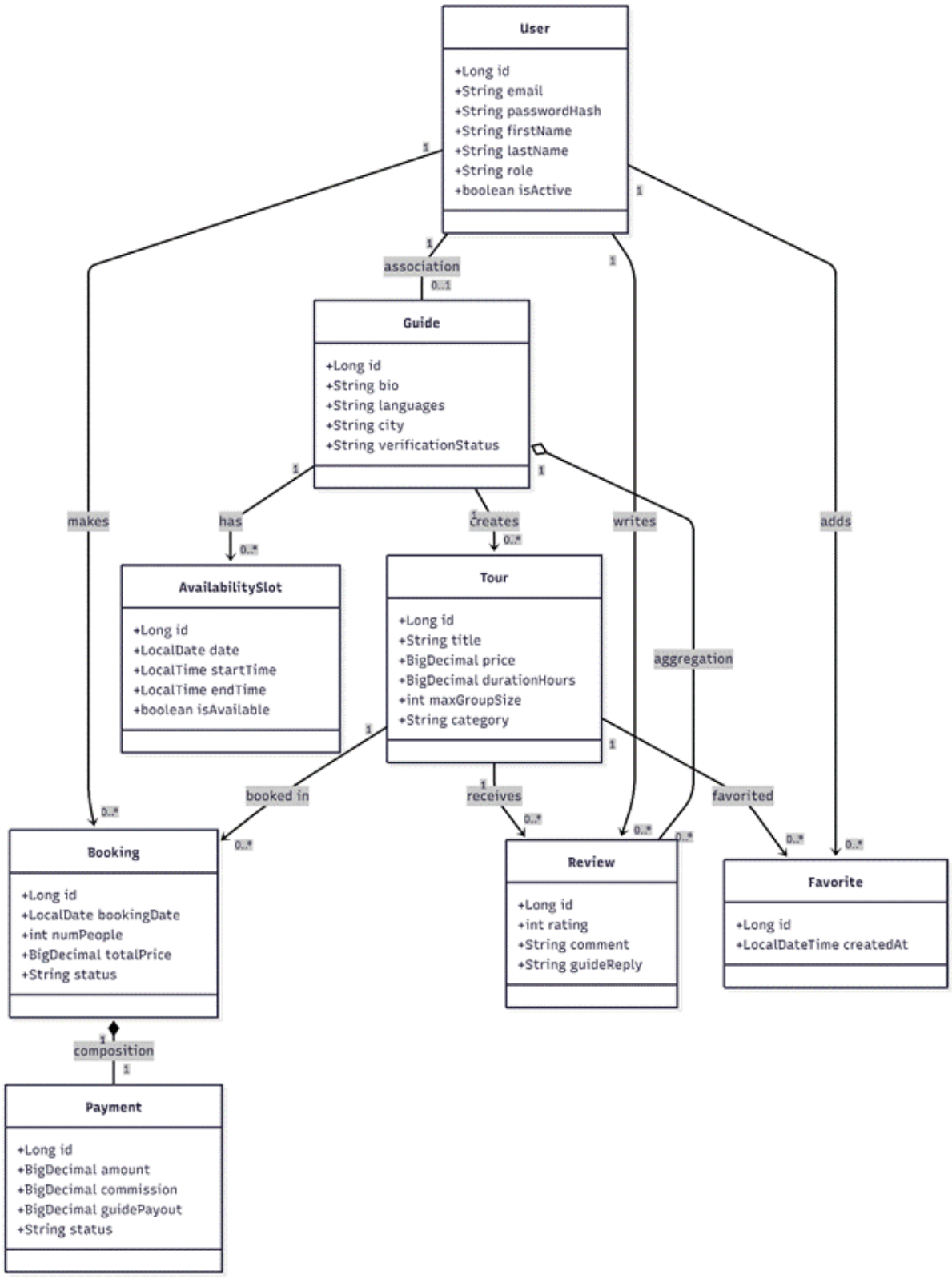
Deployment: Docker containers on EC2; Nginx as reverse proxy (routes /api/* to Spring Boot, serves Vue.js static files); HTTPS via Let's Encrypt; DB credentials stored as environment variables.

---

5. Database Design

5.1 Entity-Relationship Diagram

Our entity classes map to MySQL tables using JPA (Java Persistence API), which bridges Java objects and relational tables through ORM.


User
+Long id
+String email
+String passwordHash
+String firstName
+String lastName
+String role
+boolean isActive
association
0..1
Guide
+Long id
+String bio
+String languages
+String city
+String verificationStatus
makes
has
creates
writes
adds
AvailabilitySlot
+Long id
+LocalDate date
+LocalTime startTime
+LocalTime endTime
+boolean isAvailable
Tour
+Long id
+String title
+BigDecimal price
+BigDecimal durationHours
+int maxGroupSize
+String category
aggregation
booked in
receives
favorited
Booking
+Long id
+LocalDate bookingDate
+int numPeople
+BigDecimal totalPrice
+String status
Review
+Long id
+int rating
+String comment
+String guideReply
Favorite
+Long id
+LocalDateTime createdAt
composition
Payment
+Long id
+BigDecimal amount
+BigDecimal commission
+BigDecimal guidePayout
+String status

Relationships use UML notation (UML Class Diagrams lecture): - **1:1** — User ↔ Guide - **1:N** — Guide → Tours, Tour → Bookings - **Composition** — Booking → Payment (strong ownership) - **Aggregation** — Guide → Reviews (weak ownership)

8 Tables: users, guides, tours, bookings, payments, reviews, availability_slots, favorites

5.2 Key Table Definitions

users

Column	Type	Constraints
id	BIGINT AUTO_INCREMENT	PK
email	VARCHAR(255)	UNIQUE, NOT NULL
password_hash	VARCHAR(255)	NOT NULL
first_name, last_name	VARCHAR(100)	NOT NULL
phone	VARCHAR(20)	NULLABLE
avatar_url	VARCHAR(500)	NULLABLE
role	VARCHAR(20)	NOT NULL (TOURIST/GUIDE/ADMIN)
is_active	BOOLEAN	DEFAULT TRUE
created_at, updated_at	TIMESTAMP	NOT NULL

guides

Column	Type	Constraints
id	BIGINT AUTO_INCREMENT	PK
user_id	BIGINT	FK → users.id, UNIQUE (1:1)
bio	TEXT	NULLABLE
languages	VARCHAR(255)	NOT NULL
city, neighborhood	VARCHAR(100)	city NOT NULL
verification_status	VARCHAR(20)	NOT NULL (PENDING/APPROVED/REJECTED)
avg_rating	DECIMAL(3,2)	DEFAULT 0.00
total_reviews	INT	DEFAULT 0

bookings

Column	Type	Constraints
id	BIGINT AUTO_INCREMENT	PK
tourist_id	BIGINT	FK → users.id
tour_id	BIGINT	FK → tours.id
booking_date	DATE	NOT NULL

start_time	TIME	NOT NULL
num_people	INT	NOT NULL, DEFAULT 1
total_price	DECIMAL(10,2)	NOT NULL
status	VARCHAR(30)	NOT NULL (CREATED/PENDING_PAYMENT/CONFIRMED/COMPLETED/CANCELLED/DISPUTED)
stripe_payment_intent_id	VARCHAR(255)	NULLABLE

Other tables (tours, payments, reviews, availability_slots, favorites) follow similar patterns with appropriate foreign keys and constraints.

5.3 Indexing Strategy

Key indexes for meeting performance requirements (search < 500ms):

Table	Index	Purpose
users	idx_users_email (UNIQUE)	Fast login lookup
guides	idx_guides_city	Search by city
tours	idx_tours_city_category (COMPOSITE)	Search filtering
tours	idx_tours_price	Price range filtering
bookings	idx_bookings_tourist_id	Booking history

bookings	idx_bookings_tour_date	Date-based queries
availability_slots	idx_avail_guide_date (COMPOSITE)	Availability lookup

---

6. Interface Designs

6.1 Page Structure

Pages are organized by user roles (actors) matching our use case analysis:

Public Pages (no auth): Homepage, Search Results, Guide Profile, Tour Detail, Login/Register, About/FAQ

Tourist Pages: My Bookings, Booking Detail, My Favorites, My Profile, Write Review

Guide Pages: Dashboard, My Tours (CRUD), Availability Calendar, Booking Requests, Earnings, My Reviews, Verification Status

Admin Pages: Dashboard, Guide Verifications, User Management, Booking Management, Reports

Total: 18 pages

6.2 Key Page Descriptions

- **Homepage**: Hero image with search bar (City, Date, Category), “How It Works” section, featured guide cards, statistics bar
- **Search Results**: Left sidebar filters (city, language, price range, category, rating), guide card grid with pagination (12/page)
- **Guide Profile**: Cover photo, bio, tour listing cards, reviews section, quick booking sidebar widget
- **Booking Flow (3 steps)**: Select (tour, date, people, price breakdown) → Confirm (review details, cancellation policy) → Pay (Stripe Elements form) → Confirmation page
- **Guide Dashboard**: Stats cards (bookings, earnings, rating), upcoming tours table, recent reviews

The following wireframes illustrate the key pages of the LocalGuide Connect interface:

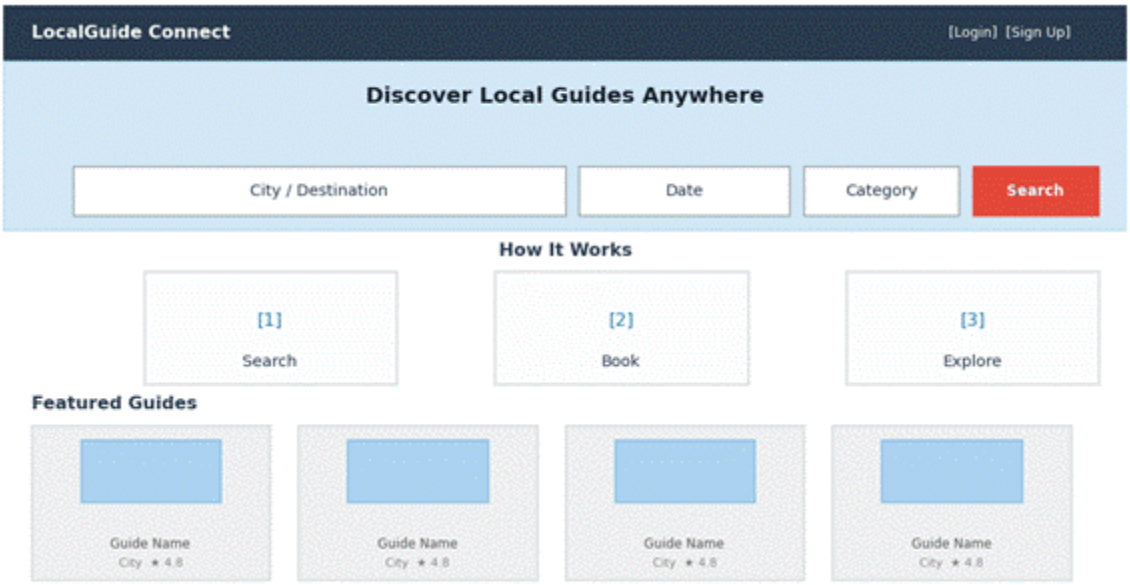


Figure 2a: Homepage Wireframe

Figure 2a: Homepage — Hero search bar, How It Works, Featured Guides

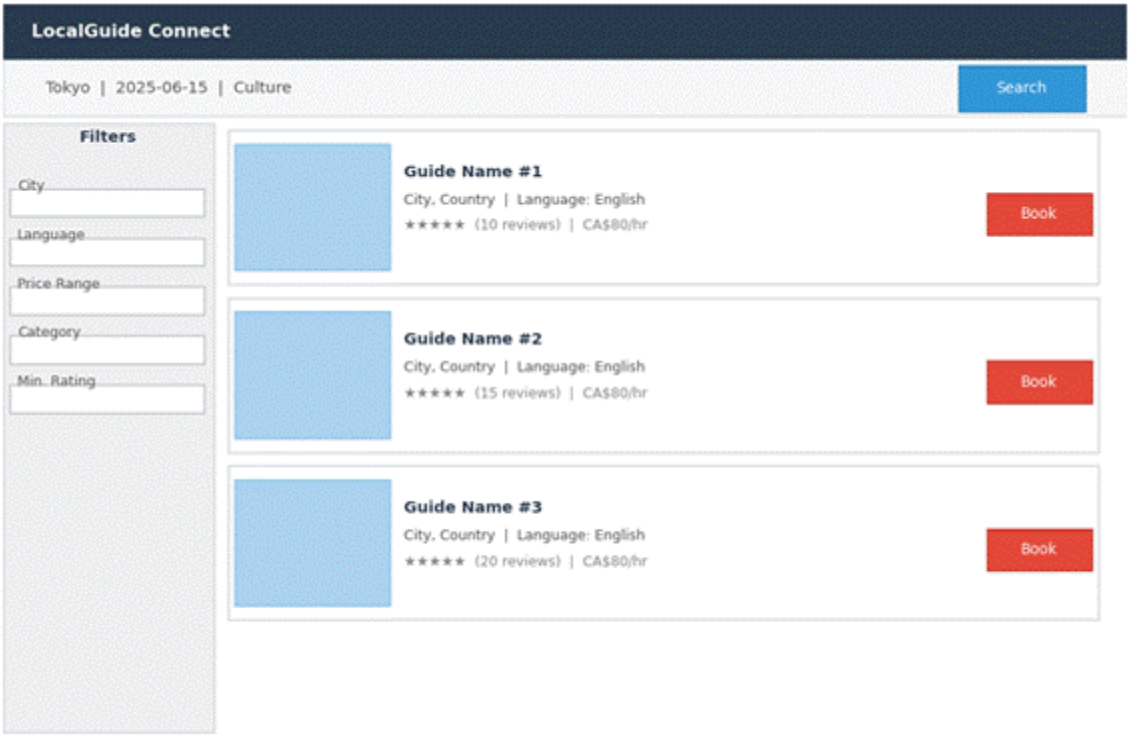


Figure 2b: Search Results Wireframe

Figure 2b: Search Results — Left filter sidebar, guide cards with ratings

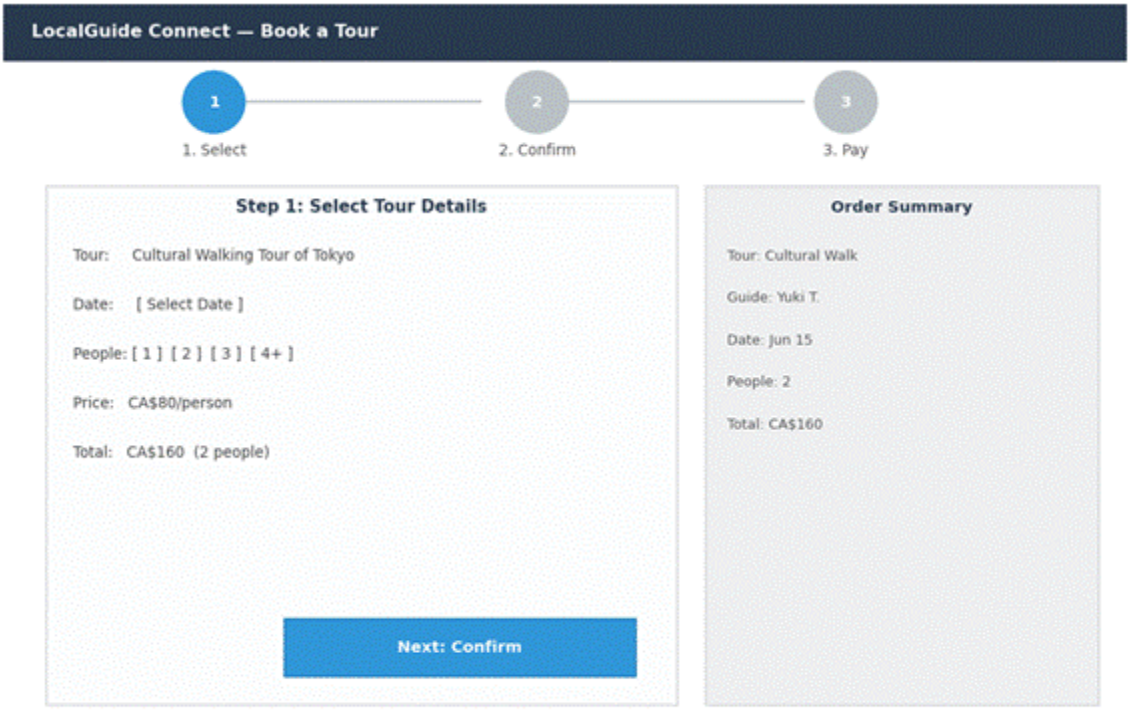


Figure 2c: Booking Flow (Step 1 shown)

Figure 2c: Booking Flow (Step 1: Select) — Tour, date, people count, price

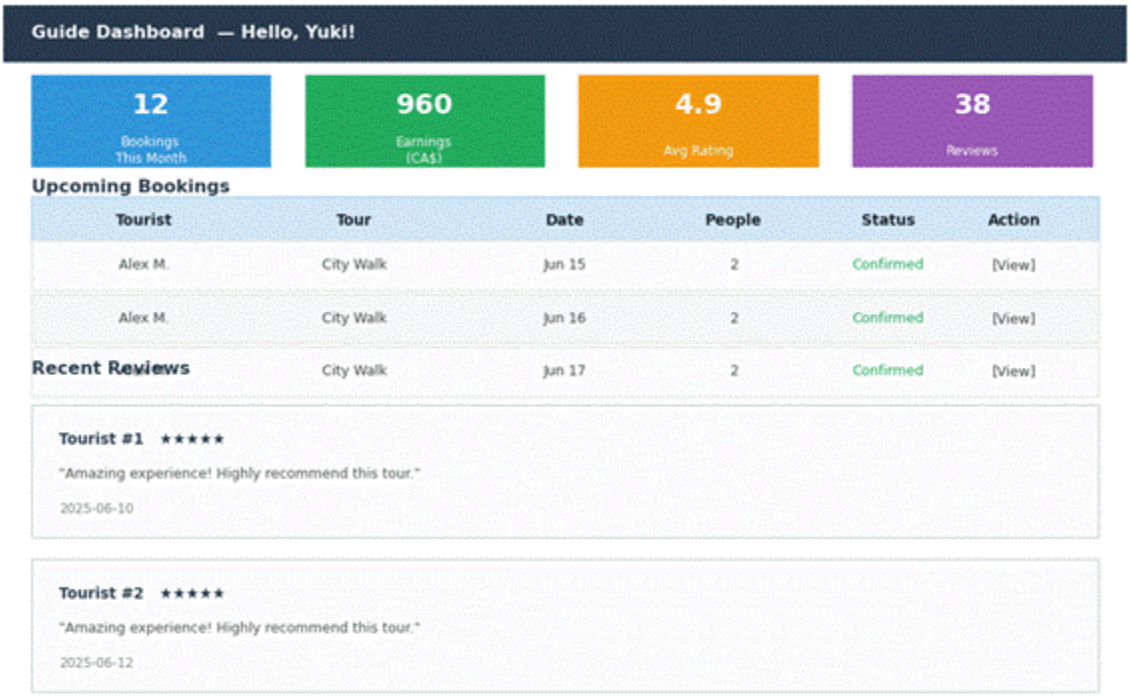


Figure 2d: Guide Dashboard Wireframe

Figure 2d: Guide Dashboard — Stats cards, upcoming bookings, recent reviews

---

7. Revised UML Diagrams — Solution Domain

In Proposal 1 we focused on the problem domain (what the system should do). Now we move to the solution domain (how the system will do it), as covered in Chapter 7.

7.1 Sequence Diagram: Booking Flow

The booking flow passes through the layered architecture:

1. Tourist clicks “Book” → Vue.js sends POST /api/v1/bookings with JWT
2. AuthFilter verifies JWT → routes to BookingController
3. BookingController → BookingService.createBooking()
4. BookingService → BookingRepository.checkAvailability() → if available, save booking

5. BookingService → StripeService.createPaymentIntent() → Stripe API returns client secret
6. Response sent to frontend with bookingId and clientSecret
7. Tourist enters card → Stripe.js confirms payment directly with Stripe
8. Stripe sends webhook (payment_intent.succeeded) → BookingController → BookingService
9. BookingService updates status to CONFIRMED → EmailService sends confirmation email
10. Tourist sees “Booking Confirmed!”

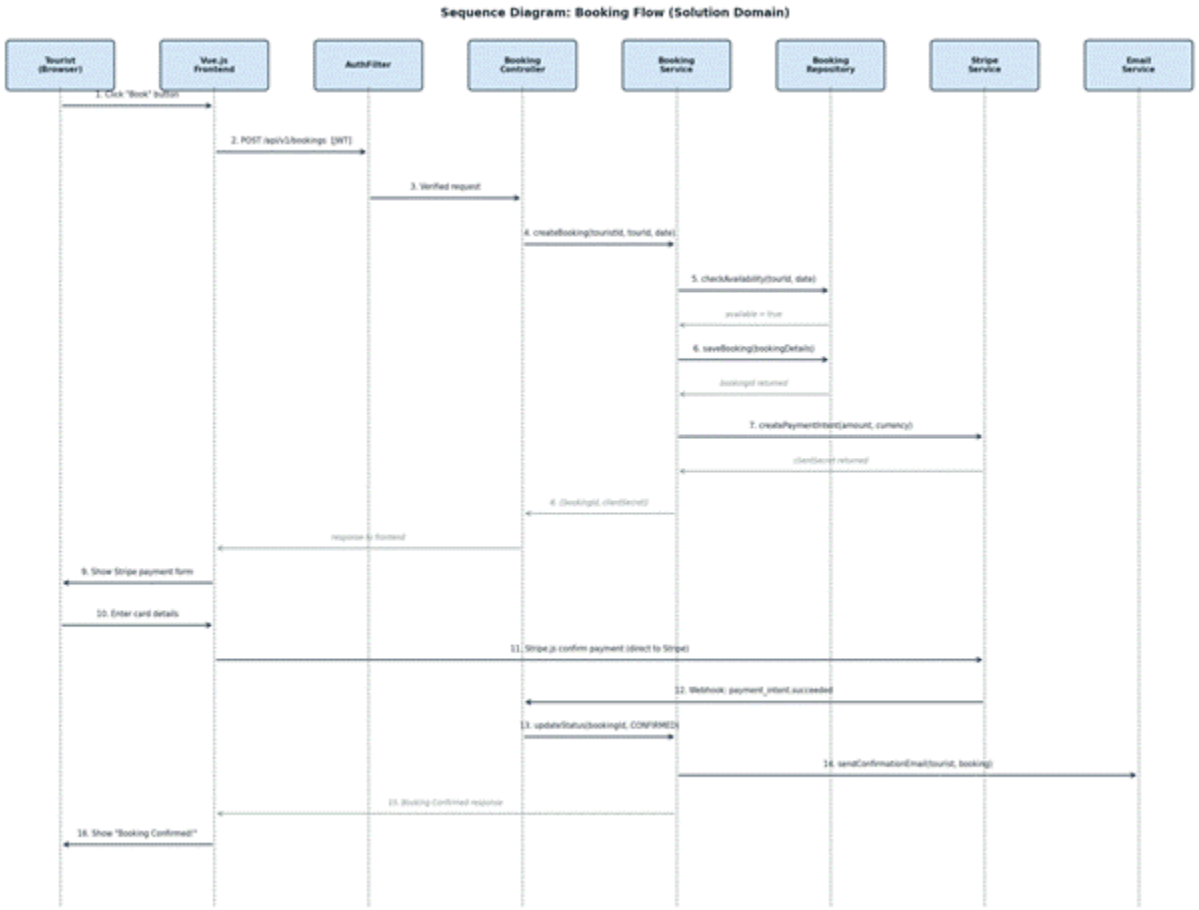
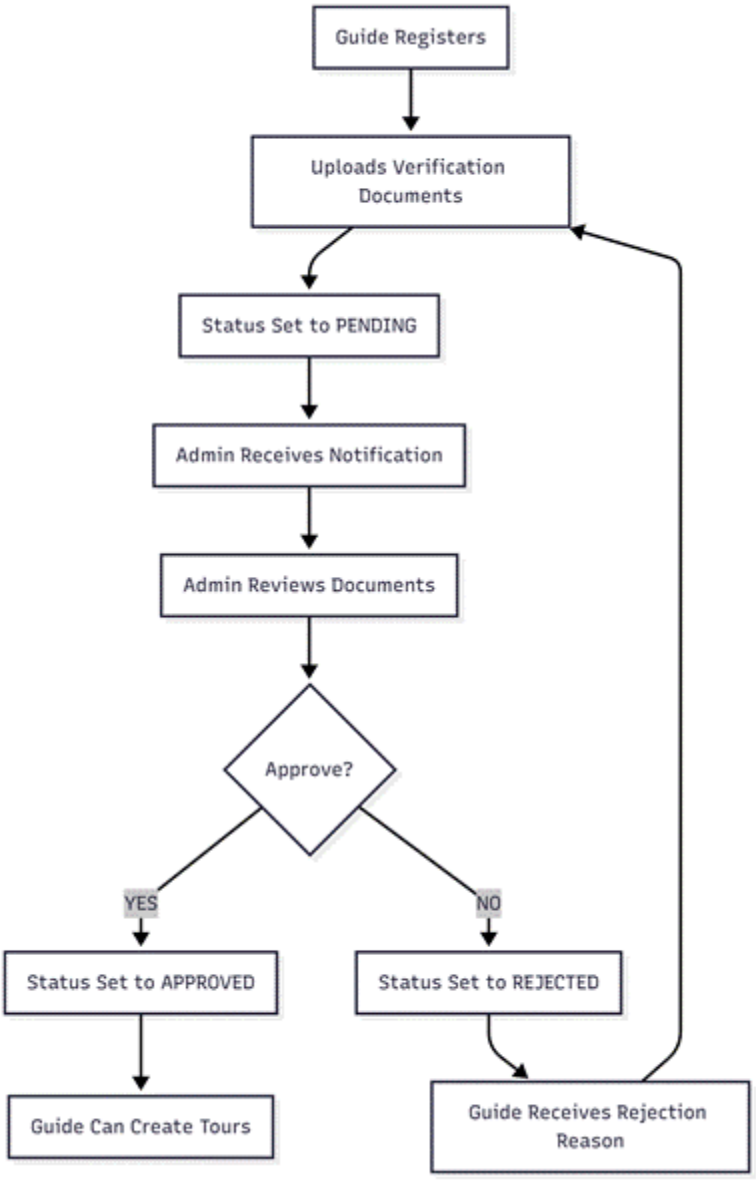


Figure 1: Sequence Diagram – Booking Flow (Solution Domain)

7.2 Activity Diagram: Guide Verification

Guide registers → uploads verification documents → status set to PENDING → Admin receives notification → Admin reviews documents → **Decision**: Approve? → YES: status = APPROVED, guide can create tours → NO: status = REJECTED, guide receives reason and can re-submit.



7.3 State Diagram: Booking Lifecycle

CREATED → [availability check] → PENDING_PAYMENT → [payment success] → CONFIRMED

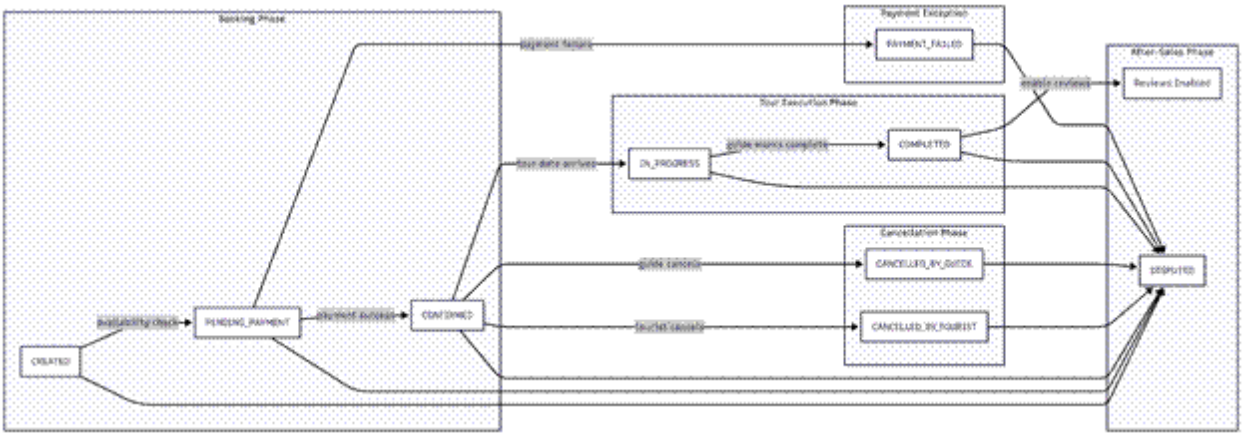
→ [payment failure] → PAYMENT_FAILED

CONFIRMED → [tourist cancels] → CANCELLED_BY_TOURIST

→ [guide cancels] → CANCELLED_BY_GUIDE

→ [tour date] → IN_PROGRESS → [guide marks complete] → COMPLETED → Reviews enabled

Any state → DISPUTED (either party files dispute)

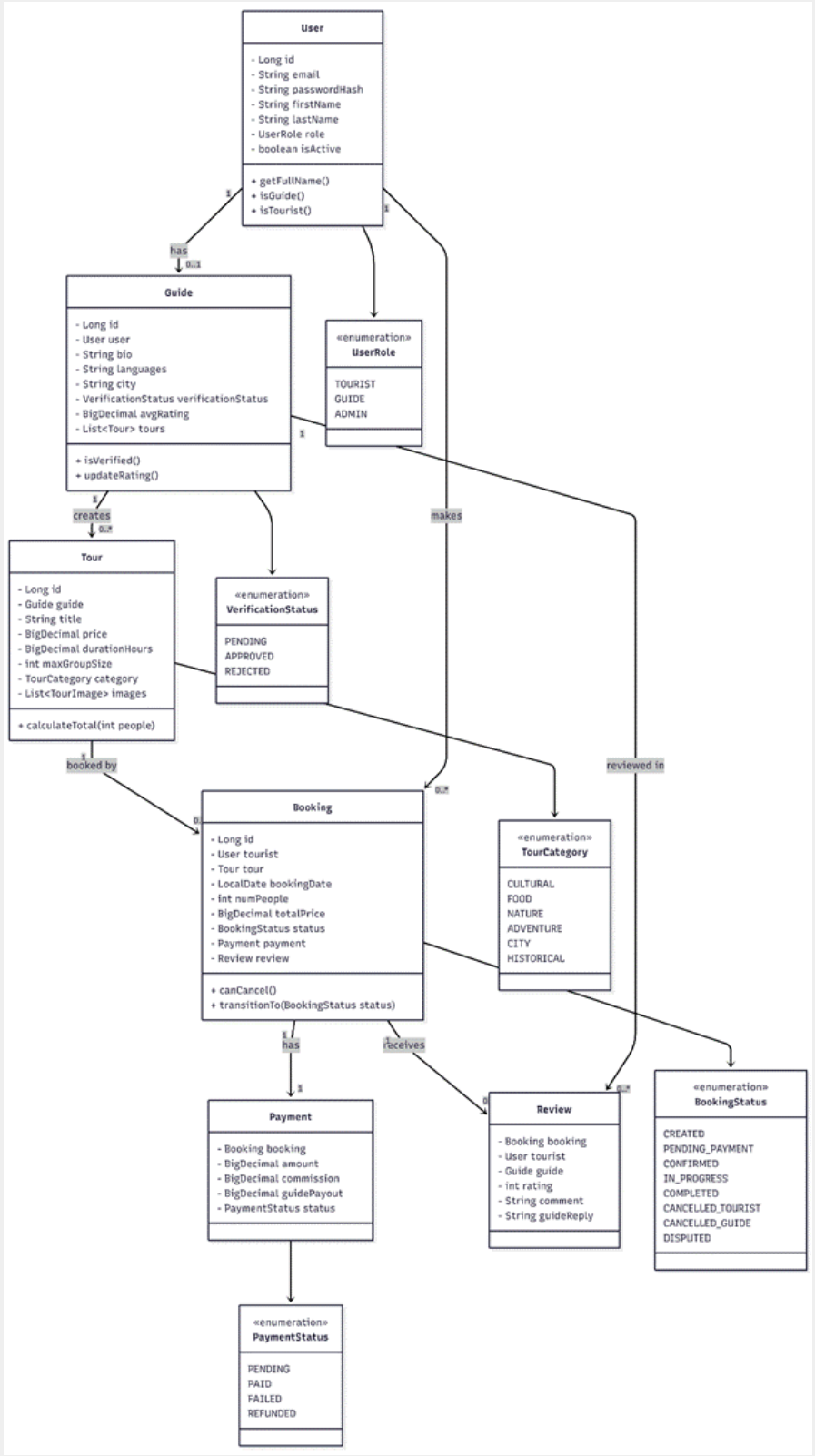


---

8. Final UML Class Diagrams

Following UML notation from the course: - private, + public, # protected. Associations with multiplicity, composition (strong ownership), aggregation (weak ownership).

8.1 Entity Classes



User (- id: Long, - email: String, - passwordHash: String, - firstName/lastName: String, - role: UserRole, - isActive: boolean, + getFullName(), + isGuide(), + isTourist())

↓ 1:1 association

Guide (- id: Long, - user: User, - bio: String, - languages: String, - city: String, - verificationStatus: VerifStat, - avgRating: BigDecimal, - tours: List<Tour>, + isVerified(), + updateRating())

↓ 1:N association

Tour (- id: Long, - guide: Guide, - title: String, - price: BigDecimal, - durationHours: BigDecimal, - maxGroupSize: int, - category: TourCategory, - images: List<TourImage>, + calculateTotal(people))

↓ 1:N association

Booking (- id: Long, - tourist: User, - tour: Tour, - bookingDate: LocalDate, - numPeople: int, - totalPrice: BigDecimal, - status: BookingStatus, - payment: Payment, - review: Review, + canCancel(), + transitionTo(status))

Payment (- booking: Booking, - amount/commission/guidePayout: BigDecimal, - status: PaymentStatus)

Review (- booking: Booking, - tourist: User, - guide: Guide, - rating: int, - comment: String, - guideReply: String)

Enumerations: UserRole (TOURIST, GUIDE, ADMIN), BookingStatus (CREATED, PENDING_PAYMENT, CONFIRMED, IN_PROGRESS, COMPLETED, CANCELLED_TOURIST, CANCELLED_GUIDE, DISPUTED), TourCategory (CULTURAL, FOOD, NATURE, ADVENTURE, CITY, HISTORICAL), VerificationStatus (PENDING, APPROVED, REJECTED)

8.2 Service Layer

The Service layer follows **ISP** (Interface Segregation Principle) — multiple focused services instead of one large one. This provides **functional cohesion**. Each Service depends on Repository interfaces (not concrete classes), following **DIP**.

Service	Key Methods	Dependencies
AuthService	register(), login(), refreshToken(), resetPassword()	UserRepository, PasswordEncoder, JwtProvider

GuideService	getProfile(), updateProfile(), createTour(), deleteTour()	GuideRepo, TourRepo
BookingService	createBooking(), confirmBooking(), cancelBooking()	BookingRepo, PaymentService, EmailService
PaymentService	createIntent(), handleWebhook(), processRefund()	StripeClient, PaymentRepo
SearchService	searchGuides(), searchTours(), filterByCity()	GuideRepo, TourRepo
ReviewService	createReview(), replyToReview()	ReviewRepo, GuideRepo
AdminService	reviewVerification(), suspendUser(), getAnalytics()	UserRepo, GuideRepo
AvailabilityService	setSlots(), checkAvailability()	AvailabilityRepo

---

9. Revised Use Cases

9.1 Changes from Proposal 1

Use cases remain mostly the same as Proposal 1. Minor updates: added availability check to the booking flow, and added password reset as a new use case.

---

10. Team Task Division and Development Plan

10.1 Module Division

We split the 15 MVP features into 4 modules. Each team member owns one module end-to-end (backend API + frontend page), so everyone knows exactly what they are responsible for.

Module	Owner	Features Covered
A: Auth & User	Dev 1	Registration, Login/Logout, Password Reset, Tourist Profile
B: Guide & Tour	Dev 2	Guide Profile, Tour CRUD, Search, Guide Availability
C: Booking & Payment	Dev 3	Create Booking, Stripe Payment, Booking Management, Email Notification
D: Reviews, Favorites & Admin	Dev 4	Reviews & Ratings, Favorites, Admin Panel, Guide Verification + DevOps

10.2 Dependencies (What Must Be Done First)

Most tasks can run in parallel, but a few items block other members. Complete these in order:

Priority	Task	Who Does It	Who Is Waiting
1st (Day 1)	Create Spring Boot project + MySQL connection (Dev 1 task 1)	Dev 1	Dev 2, 3, 4 — cannot write backend code until project exists
1st (Day 1)	Create Vue.js project + Axios setup (Dev 1 task 2)	Dev 1	Dev 2, 3, 4 — cannot write frontend code until project exists

2nd (Day 2-3)	Create users table + User entity (Dev 1 task 3)	Dev 1	Dev 2 — Guide entity has FK to users; Dev 3 — Booking has FK to users
2nd (Day 2-3)	Build login API + JWT filter (Dev 1 tasks 5, 7)	Dev 1	Dev 2, 3, 4 — all protected APIs need JWT to work
3rd (Week 1)	Create guides + tours tables + entities (Dev 2 task 1)	Dev 2	Dev 3 — Booking has FK to tours; Dev 4 — Review has FK to guides
4th (Week 2)	Create bookings table + Booking entity (Dev 3 task 1)	Dev 3	Dev 4 — Review requires a completed booking

In practice: Dev 1 spends the first 2-3 days on project setup and auth, then everyone can start building their own modules. Dev 2 and Dev 3 should create their tables/entities early so downstream members are not blocked.

Developer	Key References
Dev 1	5.2 Users Table → 3.3 Auth Module → 8.1 User Entity → 8.2 AuthService
Dev 2	5.2 Guides/Tours Table → 3.3 Guide/Tours/Search Module → 8.1 Guide/Tour Entity → 8.2 SearchService
Dev 3	5.2 Bookings/Payments Table → 3.3 Bookings/Payments Module → **7.1 Booking Flowchart** → **7.3 State Diagram** → 8.2 BookingService
Dev 4	5.2 Reviews Table → 3.3 Reviews/Admin Module → **7.2 Guide Verification Flowchart** → 8.2 ReviewService/AdminService

10.3 Dev 1 — Auth & User Module (4 features)

Dev 1 also handles project scaffold (Spring Boot + Vue.js initial setup, MySQL schema creation).

#	Task	Type	Done When
1	Create Spring Boot project with Maven, configure MySQL connection in application.yml	Backend setup	Project runs locally, connects to MySQL
2	Create Vue.js project with Vite + Pinia + Element Plus, set up Axios with JWT interceptor	Frontend setup	Frontend runs locally, can send API requests
3	Create users table in MySQL, write User entity class + UserRepository	Backend	User data can be saved and queried
4	Build registration API: POST /auth/register — validate input, hash password with bcrypt, save to DB	Backend	New user can register via Postman
5	Build login API: POST /auth/login — verify password, return JWT token	Backend	User can login and get a token

6	Build token refresh API: POST /auth/refresh — issue new JWT before old one expires	Backend	Token can be refreshed
7	Add Spring Security filter: check JWT on every request, block unauthorized access	Backend	Protected APIs return 401 without valid token
8	Build password reset API: POST /auth/reset-password — send reset email, verify token, update password	Backend	User can reset password via email link
9	Build tourist profile APIs: GET /tourists/me, PUT /tourists/me — view and edit profile	Backend	Tourist can view and update their own profile
10	Build Login page and Register page in Vue.js	Frontend	Users can register and login through the browser
11	Build Password Reset page	Frontend	Users can request and complete password reset
12	Build Tourist Profile page — view and edit personal info	Frontend	Tourist can edit name, phone, avatar

13	Store login state in Pinia, auto-redirect to login if token expires	Frontend	Login state persists across pages
14	Build shared layout: Navbar (with login/logout button, role-based menu) + Footer	Frontend	All pages share the same header and footer
15	Write unit tests for AuthService and UserService	Testing	Key auth logic has test coverage

10.4 Dev 2 — Guide & Tour Module (4 features)

#	Task	Type	Done When
1	Create guides table and tours table in MySQL, write Guide and Tour entity classes + Repositories	Backend	Guide and Tour data can be saved and queried
2	Build guide profile APIs: GET /guides/{id}, PUT /guides/me — view any guide, edit own profile	Backend	Guide profile can be viewed and updated
3	Build guide photo upload: accept image file, store to S3 or local disk, return URL	Backend	Guide can upload profile photo

4	Build tour CRUD APIs: POST /guides/me/tours, PUT /tours/{id}, DELETE /tours/{id}, GET /tours/{id}	Backend	Guide can create, edit, delete, and view tours
5	Build search API: GET /search/guides and GET /search/tours — filter by city, language, category, price range; return paginated results	Backend	Tourists can search and filter guides/tours
6	Create availability_slots table, write AvailabilitySlot entity + Repository	Backend	Availability data can be stored
7	Build availability APIs: GET/POST/PUT/DELETE /guides/me/availability — guide sets which dates and times they are available	Backend	Guide can manage their available time slots
8	Build Homepage — hero image, search bar (city + date + category), featured guides section	Frontend	Visitors see the landing page with search function
9	Build Search Results page — guide/tour cards, left sidebar filters, pagination (12 per page)	Frontend	Search results display correctly with filtering
10	Build Guide Profile page — bio, language, city, list of tours, reviews section	Frontend	Tourists can browse a guide's full profile

11	Build Tour Detail page — description, price, duration, group size, book button	Frontend	Tourists can view tour details before booking
12	Build Guide Dashboard page — stats cards (bookings, earnings, rating), upcoming tours	Frontend	Guide has a home page after login
13	Build Tour Management page — list my tours, create/edit/delete with forms	Frontend	Guide can manage all their tours in one place
14	Build Availability Calendar page — guide picks available dates/times on a calendar UI	Frontend	Guide can set availability visually
15	Write unit tests for SearchService and TourService	Testing	Key search and tour logic has test coverage

10.5 Dev 3 — Booking & Payment Module (4 features)

#	Task	Type	Done When
1	Create bookings table and payments table in MySQL, write Booking and Payment entity classes + Repositories	Backend	Booking and payment data can be saved and queried

2	Build create booking API: POST /bookings — check availability, prevent double-booking, save booking with status CREATED	Backend	Tourist can create a booking; no duplicate bookings allowed
3	Build Stripe payment integration: POST /payments/create-intent — call Stripe API to create a PaymentIntent, return client secret to frontend	Backend	Frontend receives a client secret to complete payment
4	Build Stripe webhook handler: POST /payments/webhook — Stripe notifies us when payment succeeds or fails, update booking status accordingly	Backend	Booking auto-confirms after successful payment
5	Build commission calculation: when payment succeeds, split amount into guide payout (88%) and platform fee (12%), save to payments table	Backend	Payment records show correct split
6	Build booking list and detail APIs: GET /bookings (my bookings), GET /bookings/{id}	Backend	Tourist and guide can view their bookings

7	Build cancel booking API: PUT /bookings/{id}/cancel — update status, trigger Stripe refund if already paid	Backend	Cancelled bookings get refunded automatically
8	Build email notification service: send confirmation email after booking is confirmed, cancellation email after cancel	Backend	Users receive email at key moments
9	Build 3-step Booking Flow UI: Step 1 (select date, people, see price) → Step 2 (review details, confirm) → Step 3 (Stripe card form, pay) → Success page	Frontend	Tourist can complete entire booking in the browser
10	Integrate Stripe Elements: embed Stripe's card input form on the payment step, call confirmCardPayment()	Frontend	Credit card form works with Stripe test cards
11	Build Tourist Booking History page — list all my bookings with status, click to see detail	Frontend	Tourist can track all past and current bookings
12	Build Guide Booking Requests page — see incoming bookings, accept or decline	Frontend	Guide can manage booking requests

13	Build Booking Detail page — show full info (tour, date, people, price, status, payment status)	Frontend	Both tourist and guide can see booking details
14	Test full booking flow end-to-end: search → book → pay (test card) → confirm → cancel → refund	Testing	Entire flow works without errors

10.6 Dev 4 — Reviews, Favorites, Admin & DevOps (3 features + deployment)

#	Task	Type	Done When
1	Create reviews table and favorites table in MySQL, write entity classes + Repositories	Backend	Review and favorite data can be stored
2	Build review APIs: POST /tours/{id}/reviews (submit review with 1-5 rating + comment), GET /tours/{id}/reviews (list reviews), POST /reviews/{id}/reply (guide replies)	Backend	Reviews can be created, listed, and replied to
3	Build rating aggregation: after a new review, recalculate guide's average rating and total review count	Backend	Guide profile always shows up-to-date rating

4	Build favorites APIs: POST /tourists/me/favorites (add), DELETE /tourists/me/favorites/{id} (remove), GET /tourists/me/favorites (list)	Backend	Tourist can save and manage favorite guides
5	Build guide verification APIs: POST /guides/me/verification (guide submits documents), GET /admin/verifications (admin lists pending), PUT /admin/verifications/{id} (approve or reject)	Backend	Guide verification workflow works end-to-end
6	Build admin APIs: GET /admin/users (user list), PUT /admin/users/{id} (suspend/unsuspend), GET /admin/analytics (basic stats: total users, bookings, revenue)	Backend	Admin can manage users and see platform stats
7	Build Review section on Guide Profile page — show review list with ratings, reply button for guide	Frontend	Reviews display on guide's profile
8	Build Write Review page — star rating selector, comment text box, submit button (only after completed booking)	Frontend	Tourist can write a review after their tour

9	Build Favorites page — list saved guides, remove button	Frontend	Tourist can view and manage favorites
10	Build Admin Dashboard — overview stats (total users, bookings, revenue)	Frontend	Admin has a home page with platform overview
11	Build Admin Verification page — list pending guide applications, view documents, approve/reject buttons	Frontend	Admin can review and approve guides
12	Build Admin User Management page — user list with search, suspend/unsuspend toggle	Frontend	Admin can manage users
13	Build Guide Earnings page — total earned, monthly breakdown, payout history	Frontend	Guide can track their income
14	Set up GitHub repository, branch protection rules, GitHub Actions (auto-run tests on push)	DevOps	CI pipeline runs on every push
15	Set up Docker (Dockerfile + docker-compose), deploy to AWS EC2, configure Nginx + SSL	DevOps	App is live on a public URL

16	Write unit tests for ReviewService and AdminService	Testing	Key review and admin logic has test coverage

10.7 Timeline

All four members work in parallel on their own modules. We follow 2-week sprints (Scrum).

Phase	Weeks	What Happens
Phase 1: Setup & Core	3-6	Dev 1 finishes Auth (tasks 1-14); Dev 2 finishes Guide & Tour & Search (tasks 1-14); Dev 3 starts Booking backend (tasks 1-7); Dev 4 starts Reviews & Admin backend (tasks 1-6)
Phase 2: Integration	7-10	Dev 3 finishes Booking & Payment frontend (tasks 8-13); Dev 4 finishes all frontend pages (tasks 7-13); everyone connects their modules together
Phase 3: Test & Deploy	11-12	Everyone writes tests for their own module; Dev 4 handles deployment; all members fix bugs and prepare final presentation

10.8 Git & Communication

- **Branches:** main (production) ← develop (integration) ← feature/* (e.g., feature/auth, feature/booking)
- **Rule:** Pull request + 1 code review before merging to develop
- **Communication:** Slack (daily), GitHub Issues + Projects (task tracking), Google Meet (sprint planning every 2 weeks)

---

11. Design Phase KPIs

KPI	Target	How We Measure
Design Document	All required sections completed	This document covers architecture, DB, API, UML, and task plan
MVP Feature Coverage	15 core features mapped to tasks	Each feature assigned to a team member (Section 10)
Database Design	All tables defined with correct relationships	8 tables with FKs, no data duplication
API Endpoints	Every feature has at least one endpoint	28 endpoints across 10 modules
Task Assignment	Every team member has a clear task list	4 modules, ~15 tasks each

---

12. Conclusion

This Progress Report 2 presents the system design for LocalGuide Connect, moving from requirements analysis to a technical blueprint.

Key Design Decisions: - Three-tier architecture (Vue.js + Spring Boot + MySQL) with separation of concerns (Chapter 10) - Layered backend following MVC pattern with JPA Repository interfaces (Spring MVC lecture) - RESTful API with JWT authentication - Stripe integration for PCI-compliant payments - Component design following SOLID principles and high cohesion/low coupling (Chapter 11) - Docker-based deployment on AWS

Team Readiness: - 4 members with clear task assignments across 5 sprints (400 total hours) - Git branching strategy and code review process established - Scrum ceremonies defined (sprint planning, standup, review)

Next Steps: 1. Begin Sprint 1 (project scaffold + authentication) 2. Set up GitHub repository with branch protection 3. Configure dev environments (Java 17, MySQL 8.0, Node.js 20) 4. Hold first sprint planning session

---

LocalGuide Connect — Group 3 — CSIS 3275 Software Engineering — February 2026