👩‍🎓 Member 1 — Database & Reports Lead (Beginner-friendly, from zero ➜ working DB)

I'll hold your hand step-by-step. You'll create the SQL database that exactly matches your SRS (bookings, services, partial payments, reports) and satisfies security/ACID notes. I'll keep the language simple and add tiny bits of extra context only when it helps.
(The SRS sections that define these needs are in product functions + system features + non-functional/security; see citations.)

🧭 Your mission (in plain words)

1. Design tables for **branches, rooms, room types, guests, bookings, services, service usage, payments, users/roles, audit log**.
2. Enforce rules: **no double-booking**, room status flips on **check-in/out**, **no checkout with dues**.
3. Provide **5 reports**: occupancy, billing (with unpaid balances), service breakdown, monthly revenue, top services.
4. Seed sample data as required in the SRS.
5. Share **one .sql file** that teammates can run to get everything.
   (All of this is explicitly required in your SRS features and business rules.)

🛠️ Setup (from nothing)

0) Install tools

- **MySQL 8 Community** (Windows/Mac/Linux).
- Optional GUI: **MySQL Workbench** (easier to run/inspect).
- Terminal basics you'll use:
  - `mysql -u root -p` (login)

- `SOURCE path/to/your_file.sql;` (run a whole SQL file)

1) Start a new SQL file

Create a file named `skynest_schema.sql`. You will paste everything you build in the steps below into this one file so others can run it.

🧱 Step A — Create the database + core tables (copy/paste friendly)

> This structure comes straight from SRS "Product Functions" and "System Features" (booking, check-in/out, services, payments, reporting) and the environment specifies a **relational DB** with **ACID**.

1. **Database + reference tables**
   - `branch` (Colombo/Kandy/Galle)
   - `room_type` (Single/Double/Suite: capacity, daily_rate, amenities)
   - `room` (room_number, belongs to branch + room_type, has status)
2. **Business tables**
   - `guest` (name, email, phone)
   - `service_catalog` (code, name, category, unit_price, active)
   - `booking` (room_id, guest_id, check_in/check_out, status, **booked_rate**, tax %, discount, late_fee, payment_method)
     - Why **booked_rate**? So rate changes later don't affect old bookings.
   - `service_usage` (booking_id, service_id, date, qty, **unit_price_at_use**)
   - `payment` (booking_id, amount, method, timestamp)
   - `user_account` (username, **password_hash**, role) for RBAC (Admin/Manager/Receptionist/Accountant)
   - `audit_log` (who did what & when) — for security/auditing in SRS.

💡 Tip: Add **indexes** on (room_id, dates) and on booking_id in usage/payments so the SRS performance goals (e.g., quick search, fast reports) are realistic.

🧮 Step B — Add billing helper FUNCTIONS

Write small SQL functions:

- `fn_room_charges(booking_id)` → nights × `booked_rate`
- `fn_service_charges(booking_id)` → sum of `quantity × unit_price_at_use`
- `fn_total_paid(booking_id)` → sum of payments
- `fn_bill_total(booking_id)` → (room + services + late_fee − discount) × (1 + tax%)
- `fn_balance_due(booking_id)` → bill_total − total_paid

This exactly matches the SRS billing logic (room + services, allow partial payments, show unpaid balance).

🧯 Step C — Enforce business rules with TRIGGERS

1. **Prevent double booking** on the **same room** for overlapping dates (on INSERT/UPDATE of `booking`).
2. On **check-in** (status changes to `Checked-In`) → set `room.status='Occupied'`.
3. On **check-out**:
    - **BEFORE update** to `Checked-Out`: **block** if `fn_balance_due > 0` (SRS: cannot mark Checked-Out unless dues cleared).
    - **AFTER update**: set `room.status='Available'`.

These triggers implement the SRS “Business Rules” and “Safety Requirements” that call out validation and blocking unsafe operations.

🧪 Step D — Seed data (what to insert)

Per your assignment brief: **3 branches**, **≥10 rooms** across types, **6 services**, **5 guests**, **8 bookings**, service usage rows, and **≥3 partial payments**.

> Seed close to your SRS examples and technology list (React/Node/MySQL; RBAC).

Checklist for your inserts:

- Branches: *Colombo, Kandy, Galle*
- Room types: Single/Double/Suite with realistic rates
- Rooms: at least 10, spread across branches
- Services: Room Service, Spa, Laundry, Minibar, Transport, Breakfast
- Guests: 5 names
- Bookings: 8 total; some `Booked`, some `Checked-In`, some `Checked-Out`
- Service usage: a few rows linked to different bookings
- Payments: at least 3 payments across different bookings
- Users: 3–4 staff accounts with **hashed** passwords (store a dummy bcrypt hash string)
- Audit: you'll start empty; app will insert later

📈 Step E — Build the 5 required reports (exactly as SRS asks)

1. **Room occupancy** for a date/period
2. **Guest billing summary** (incl. unpaid balances)
3. **Service usage breakdown** (per room and service type)
4. **Monthly revenue per branch** (room + services)

5. **Top-used services & customer preference trends**

   These are the same reports named in SRS §4.7.3 (room occupancy, billing with unpaid, monthly revenue; plus services usage).

Write each as a **standalone SELECT** in your `.sql` so anyone can run them.

🔐 Step F — Security & ACID basics (DB side you control)

- Use appropriate **data types** and **CHECKs** (e.g., positive amounts, quantity > 0).
- Triggers above enforce **integrity** and "**prevent check-out if billing issues**" (SRS safety/security).
- Add **foreign keys** with **ON UPDATE/ON DELETE RESTRICT** to protect references.
- Make sure monetary columns are **DECIMAL(10,2)**.
- Add **basic RBAC tables** (`user_account.role`) to support the SRS roles.
- Your teammates' API will wrap writes in **transactions** to meet ACID (SRS environment).

✅ Step G — Self-test (you can do this even as a beginner)

Run each test in MySQL Workbench/terminal after loading your `.sql`.

1. **Double booking test**
   - Try to insert a new `booking` overlapping dates for the same `room_id` where an existing booking is `Booked`/`Checked-In`.
   - ✅ Expect: **ERROR** from trigger.
2. **Check-in flow**

- `UPDATE booking SET status='Checked-In' WHERE booking_id=?;`
- `SELECT status FROM room WHERE room_id=(SELECT room_id FROM booking WHERE booking_id=?);`
- ✅ Expect: room becomes **Occupied**.

3. **Partial payments + balance**
   - Insert multiple rows in `payment`.
   - `SELECT fn_balance_due(?);`
   - ✅ Expect: correct remaining amount.
4. **Checkout guard**
   - Try `UPDATE booking SET status='Checked-Out'` when `fn_balance_due > 0`.
   - ✅ Expect: **ERROR** (blocked).
5. **Reports**
   - Run all 5 SELECTs.
   - ✅ Expect: results without errors, matching SRS expectations (e.g., show unpaid bookings).

📦 Step H — Deliverables for the team

- **`skynest_schema.sql`** containing:
  1. `CREATE DATABASE` + tables (with keys & indexes)
  2. FUNCTIONS & TRIGGERS
  3. SEED INSERTS
  4. REPORT QUERIES (comment each with a header)

- A tiny **README** (how to run: `mysql < skynest_schema.sql`)
- Screenshot(s) of report queries running (optional but helps)

🧭 What to focus on (so you don't get lost)

- Keep your `.sql` runnable **top-to-bottom** (no manual steps in between).
- Whenever you feel stuck, re-read SRS **System Features** and **Non-Functional/Security**: they literally tell us what must be enforced (double booking, check-out dues, RBAC, audit, encryption/HTTPS from app).

🧩 How your work connects to others

- **Member 2 (Backend)** will call your tables, rely on your **functions** for totals, and your **triggers** for safety.
- **Member 3 (Frontend)** will display your **reports** and room/booking data.
- **Member 4 (Security/QA)** will test your triggers and integrity rules align with SRS security.
- **Member 5 (DevOps/Docs)** will run your `.sql` in Docker/MySQL in CI.

📝 Recap in 10 bullets (TL;DR)

1. Install MySQL 8 + Workbench.
2. Create `skynest_schema.sql`.
3. Tables: branch, room_type, room, guest, service_catalog, booking, service_usage, payment, user_account, audit_log.
4. Keys + indexes (dates/booking_id).
5. Functions: room/services/paid/bill_total/balance_due.

6. Triggers: block double-booking; flip room status on check-in/out; block checkout with dues.

7. Seed: 3 branches, 10+ rooms, 6 services, 5 guests, 8 bookings, usage rows, 3+ payments, few users.

8. Queries: 5 reports exactly as SRS.

9. Test the 4 flows (overlap, check-in, balance, checkout guard).

10. Hand over the `.sql` + README.

If you're ready, say "Member 2" and I'll guide the **Backend/API Lead** from zero, mapping to the same SRS+security slides (sessions, SQL injection defense, RBAC, HTTPS).

👩‍💻 Member 2 — Backend/API Lead

(Beginner-friendly: from zero → a secure working API that matches your SRS. I'll keep it simple, step-by-step, with tiny bits of extra info only when it helps.)

🎯 Your mission (plain words)

You will build a **Node.js + Express** backend that talks to the **SQL database** (from Member 1), exposes **REST APIs** for staff and customers, enforces **business rules & security**, and produces the **reports** the SRS asks for. Public endpoints for customers (availability + create booking). Protected endpoints for staff (check-in/out, services, payments, reports).

🧩 What the SRS requires from you (why you're doing this)

- **Endpoints** for bookings, services, reports; public availability APIs.
- **Prevent check-out if dues remain; allow partial payments; link services to bookings.**
- **Security**: TLS/HTTPS, RBAC (roles), strong passwords, protect against SQLi/XSS; keep audit logs.
- **Performance targets** (fast search/billing/reports).
- **Architecture**: Web (browser) ↔ HTTPS ↔ Backend (Express) ↔ SQL DB.

🛠️ Setup (from nothing)

1. **Install:**
    - Node.js LTS
    - A code editor (VS Code)
    - Postman or Thunder Client (to test APIs)

Create project:

```
mkdir skynest-api && cd skynest-api
```

```
npm init -y
npm i express mysql2 dotenv zod jsonwebtoken bcryptjs cors helmet
morgan express-rate-limit
npm i -D nodemon
```

2.
   - `mysql2` → DB driver (prepared statements prevent SQL injection).
   - `helmet`/`cors`/rate-limit → security middleware. *(Your SRS asks for TLS + attack protection.)*
   - `zod` → validate request bodies.

Basic files/folders (copy this structure):

```
skynest-api/
├─ src/
│  ├─ app.js           (Express app + middleware)
│  ├─ server.js        (start HTTP server)
│  ├─ db.js            (MySQL pool)
│  ├─ auth/
│  │   ├─ auth.middleware.js  (JWT + role check)
│  │   └─ auth.controller.js  (login)
│  ├─ modules/
│  │   ├─ rooms/
│  │   │   ├─ rooms.routes.js
│  │   │   └─ rooms.service.js
│  │   ├─ bookings/
│  │   │   ├─ bookings.routes.js
│  │   │   └─ bookings.service.js
│  │   ├─ services/
│  │   ├─ payments/
│  │   └─ reports/
│  └─ utils/
│      ├─ validation.js
│      └─ errors.js
├─ .env.example
└─ package.json
```

3.

.env.example (fill real values in your local `.env`):

```
DB_HOST=localhost
DB_PORT=3306
DB_USER=root
DB_PASS=yourpass
DB_NAME=skynest
JWT_SECRET=change_me
NODE_ENV=development
```

4. Use **HTTPS/TLS** in deployment as required.

🧠 Architecture you'll follow

- **Three-layer:** controller (routes) → service (business logic) → data access (SQL queries). This matches MVC/tiers from your lecture slides.
- **Sessions & cookies (for staff UI)** are a thing on the Web; we'll use **JWT** for stateless auth and cookies if needed. *(Slides explain HTTP is stateless; sessions/cookies store identity.)*

🔐 Security (do this from day 1)

- **Always** use **prepared statements** (parameterized queries) to block SQL injection.
- **Validate** all inputs with `zod` (dates, numbers, enums).
- **JWT** for customers; **RBAC** for staff (Admin/Receptionist/Accountant/Manager).
- **Helmet + CORS + rate-limit** (DDoS/basic misuse).
- **Never** return full card numbers (masking).
- **Serve over HTTPS/TLS 1.3** in prod.
- Sanitize/escape any HTML in text fields to reduce **XSS/CSRF** risk (slides mention XSS/CSRF).

🚦 Routes you must build (exactly what, in friendly terms)

1) Public (no login)

- `GET /public/rooms/availability?branchId&from&to&roomType`
  Returns available rooms for dates. *(SRS public API)*

- `POST /public/bookings`
  Create a booking (customer or staff on behalf). Validate overlap before calling DB (extra safety). *(DB also enforces via triggers from Member 1.)*

2) Staff (token with role)

- `GET /bookings/:id` — booking with charges summary

- `POST /bookings` — create (for walk-ins/phone)

- `PATCH /bookings/:id/check-in` — sets status to **Checked-In**; flips room status (DB trigger also does this)

- `PATCH /bookings/:id/check-out` — **block** if dues remain; otherwise set **Checked-Out** (DB trigger blocks too)

- `POST /bookings/:id/services` — add service usage (name/date/qty/price at time)

- `POST /bookings/:id/payments` — partial payments (amount, method, time)

- `GET /reports/occupancy?from&to&branchId`

- `GET /reports/billing-summary?from&to&unpaidOnly=true`

- `GET /reports/revenue-monthly?year&branchId`

- `GET /reports/service-usage?from&to&serviceType` (top services/trends)

> 🧲 Why both app checks **and** DB triggers? Your SRS says prevent errors safely (double-booking, dues). App-level checks give fast feedback; DB triggers guarantee integrity even if someone bypasses the app.

🧪 How to implement each route (step-by-step recipe)

A) Database connection (once)

- Create `db.js` with a **MySQL pool**.
- Export a helper `query(sql, params)` that always uses **? placeholders** (prepared).

B) Validation (for beginners)

- In each route, validate body/query using `zod`:
  - Dates: ISO strings
  - Enums: status ∈ {Booked, Checked-In, Checked-Out, Cancelled}
  - Numbers: positive amounts/qty

C) Transactions where needed

Wrap multi-step actions in one **transaction**:

- **Create booking** = verify availability → insert booking → (optional) initial payment → commit.
- **Check-out** = compute bill using DB functions (Member 1), verify balance=0 → set status → commit.
  (*SRS: ACID integrity for financial/booking transactions.*)

D) Availability query (logic idea)

- SQL: "find rooms in branch/type NOT having any booking that overlaps [from,to) with status in (Booked, Checked-In)".
- Use `LEFT JOIN`/`NOT EXISTS` pattern.
- Return list to UI.

E) Double-booking guard (in app)

- Before inserting booking, run an overlap check query.
- If any found → return 409 Conflict.
  (DB trigger from Member 1 is the final safety net.)

F) Check-in/out

- **Check-in:** update booking status; DB trigger flips room to **Occupied**.
- **Check-out:** first fetch `balance_due` from DB function; if > 0, **block** (400). Otherwise update status; trigger flips room to **Available**.

G) Services & payments

- Insert to `service_usage` (store price at time of use) and `payment` (partial allowed).

H) Reports

- Run **read-only SQL** (Member 1 will give you the queries). You just expose them as `GET` endpoints:
  - Occupancy, Billing with unpaid, Revenue per month/branch, Service breakdown/top services/trends.

🔑 Authentication & RBAC (super simple)

1. **Login (staff):**
   - POST `/auth/login` → email + password → verify bcrypt hash → issue **JWT** with `{id, role}`.
   - Store minimal info in token; expire in e.g., 30–60 minutes; refresh by re-login.
     (SRS: JWT mentioned for customers; RBAC for staff.)
2. **Auth middleware:**
   - Read `Authorization: Bearer <token>`
   - Verify JWT.

    - Attach `req.user` with role.

3. **Role middleware:**
    - `requireRole('admin' | 'receptionist' | 'accountant' | 'manager')`
    - Example: only **accountant** can POST payments; only **admin/manager** can run all reports; **receptionist** handles check-in/out & services — exactly like SRS business rules.

🧠 Lecture slides reminder: **HTTP is stateless**; sessions/cookies/JWT identify the user between requests.

🛡️ More security basics (just do these)

- `app.use(helmet())`, set CORS to your frontend origin.
- Rate-limit: e.g., 100 requests / 15 min per IP for public routes.
- Log requests with `morgan` (don't log secrets).
- Never echo raw user HTML (XSS). Slides call out **XSS/CSRF** as common web risks.
- Mask card numbers; use **Stripe/PayHere tokenization** through HTTPS only (backend never stores raw card).

🚀 Running locally

1. Start DB (after Member 1's `.sql` is loaded).
2. `npm run dev` with script: `"dev": "nodemon src/server.js"`
3. Test with Postman:
    - **Public**: availability → create booking
    - **Staff**: login → add service → add partial payment → try to check-out with dues (**should fail** by rule) → pay remaining → check-out (**should pass**).

(SRS safety rule.)

📏 Meet performance targets (simple wins)

- Use **DB indexes** (Member 1) + **prepared statements**. *(SRS: optimize SQL; provide reports quickly.)*
- Add **connection pooling** (mysql2 pool). Slides recommend pooling for performance.
- Cache **read-heavy** reference data (room types, services) in memory for a few minutes if needed.

🧪 What to test (beginner checklist)

- ✅ Cannot create overlapping booking for same room (409 + DB trigger).
- ✅ Check-in updates room to **Occupied** (trigger).
- ✅ Check-out blocked until `balance_due = 0`.
- ✅ Partial payments accumulate.
- ✅ Reports return correct rows.
- ✅ Only allowed roles can access/modify (RBAC).
- ✅ Input validation errors return 400 with clear messages.

📦 Your deliverables to the team

1. **Source** under `skynest-api/` with routes, services, middleware.
2. **`.env.example`** and short **README** (how to run; Postman collection).
3. **Postman collection** for all endpoints (so Member 3 can integrate UI fast).

4. **Small “security notes”** doc: what you did for SQLi/XSS/HTTPS/RBAC and how to rotate JWT secret. *(SRS security & lecture slides.)*

🧭 How your work connects

- **Member 1** gives you the schema, functions, triggers, report SQL. You **call them** in endpoints.
- **Member 3** (Frontend) hits your endpoints and shows data/charts.
- **Member 4** (Security/QA) will pen-test your routes (SQLi/XSS), role checks, and the check-out rule.
- **Member 5** (DevOps/Docs) deploys your app over **HTTPS/TLS**, sets env vars, and sets up logs/backups.

📝 Tiny glossary (so you don’t feel lost)

- **Prepared statement**: SQL with ? placeholders to safely pass variables (prevents SQLi).
- **JWT**: a signed token proving who the user is.
- **RBAC**: let/deny access by role (admin/receptionist/accountant/manager).
- **Transaction**: all-or-nothing group of DB actions (keeps money/booking consistent).

If you want, say **“Member 3”** and I’ll guide the **Frontend/UI Lead** in the same beginner-style — wiring to your routes, building the screens in the SRS (login, dashboard, booking, services, billing, reports) and following the security notes.

🎨 Member 3 — Frontend / UI Lead

(Beginner-friendly: from zero → a working React UI that exactly matches your SRS screens + security notes. I'll tell you what to do, step by step.)

🧭 Your mission (in plain words)

Build a **React** web app (staff + public) that shows the exact screens in your SRS: **Login, Dashboard, New Booking, Service Log, Billing, Report Dashboard**, plus a simple **Public Booking** flow. Keep it fast (≤3s load), accessible, and role-based.

🛠️ Step 0 — Install tools (from nothing)

1. Install **Node.js LTS**.

In a new folder:

```
npm create vite@latest skynest-web -- --template react
cd skynest-web
npm i react-router-dom axios zod
npm i chart.js react-chartjs-2
npm i tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

2.
3. Configure Tailwind (`tailwind.config.js`) and include Tailwind in `index.css`.

> SRS says **React** for frontend, charts for reports, and consistent UI style. We'll use Tailwind + Chart.js to keep it simple.

🗂️ Step 1 — Project structure (copy this)

```
skynest-web/
├─ src/
│  ├─ api/axios.js              (base URL, auth header)
│  ├─ auth/authContext.jsx      (store token+role here)
│  ├─ components/
```

```
│   │    ├─ Navbar.jsx   ├─ Sidebar.jsx   ├─ DataTable.jsx
│   │    ├─ DateRangePicker.jsx     ├─ StatCard.jsx   ├─
ConfirmModal.jsx
│   ├─ pages/
│   │    ├─ Login.jsx              // 1️⃣
│   │    ├─ Dashboard.jsx          // 2️⃣
│   │    ├─ NewBooking.jsx         // 3️⃣
│   │    ├─ ServiceLog.jsx         // 4️⃣
│   │    ├─ Billing.jsx            // 5️⃣
│   │    ├─ Reports.jsx            // 6️⃣
│   │    └─ PublicBooking.jsx      // (public portal)
│   ├─ routes/ProtectedRoute.jsx  (checks role)
│   ├─ App.jsx
│   └─ main.jsx
└─ .env.example
```

- **Why this?** It maps 1:1 to SRS sample screens + public portal.

🔐 Step 2 — Auth (super simple & safe)

1. **Login page** posts email/password to backend; store the **JWT** and **role** in `authContext` (memory + `localStorage`).
2. Show/hide pages by **role**:
   - **Receptionist**: Dashboard, New Booking, Service Log, Check-in/out
   - **Accountant**: Billing + Payments
   - **Manager/Admin**: Reports + everything read-only
     (Exactly as SRS RBAC.)
3. Add **auto-logout after 10 minutes** of inactivity (timer reset on user actions).

🌐 Step 3 — API client (you'll reuse this everywhere)

`src/api/axios.js`

```
import axios from "axios";
const api = axios.create({ baseURL: import.meta.env.VITE_API_URL,
withCredentials: true });
api.interceptors.request.use(cfg => {
  const t = localStorage.getItem("token");
  if (t) cfg.headers.Authorization = `Bearer ${t}`;
  return cfg;
});
export default api;
```

- Uses **Authorization: Bearer** like your backend plan. Public routes don't need token.

🧑‍💼 Step 4 — Build each page (exactly what to put)

1) 🔐 Login.jsx

- Fields: **Branch (dropdown)**, **Username**, **Password**, **Log In**; **Forgot password?** link (can be placeholder).
- On submit → `POST /auth/login` → save `{token, role, name}` → `navigate('/dashboard')`.
- UX per SRS: **inline errors (red), toast success/fail**; keyboard shortcut **Ctrl+S** submits.

2) 📊 Dashboard.jsx (staff home)

- Left: **Today's Summary** (check-ins count, unpaid bookings).
- Right: **Room Status** list with colored dots: 🟢 Available / 🔴 Occupied / 🟡 Cleaning.
- Big buttons: **New Booking**, **Check-In**.
- Pull data from:
  - `/reports/occupancy?from=today&to=tomorrow`
  - `/reports/billing-summary?unpaidOnly=true`

- Matches SRS sample widgets & indicators.

3) 🗓️ NewBooking.jsx

- Controls: **Guest search/add**, **Date pickers (check-in/out)**, **RoomType**, **Available rooms** list, **Confirm**.
- Call: `GET /public/rooms/availability?branchId&from&to&type` then `POST /bookings`.
- Validate: dates required, `checkOut > checkIn`. Inline error highlights. Shortcuts: **Alt+N** for new booking dialog.

4) 🧾 ServiceLog.jsx

- Fields: **Guest/Booking**, **Service** (select), **Qty**, **Price at use** (prefill from catalog but editable), **Add**.
- Show a **today's services** table under the form.
- Call: `POST /bookings/:id/services`.
- SRS requires storing **service name/date/qty/price** linked to booking & catalog.

5) 💳 Billing.jsx

- Top: Summary card — **Room Charges + Service Charges ± Discounts/Fees + Tax = Total**; **Paid**; **Balance (big font)**.
- Middle: **Add Payment** form (amount, method). **Partial payments** allowed.
- Bottom: Buttons: **Confirm Check-Out** (disabled if balance > 0), **Print/Download Invoice**.
- Calls:
  - `GET /bookings/:id` (server computes totals)
  - `POST /bookings/:id/payments`
  - `PATCH /bookings/:id/check-out` (will fail if dues remain — expected)
- Exactly the SRS rule: **can't check out unless dues are cleared**; show unpaid in UI.

6) 📈 Reports.jsx

- Filters: **Branch, Date/Month**.
- Cards + Charts:
  - **Occupancy %** (bar/line) → `GET /reports/occupancy?from&to&branchId`
  - **Revenue per month per branch** (bar) → `GET /reports/revenue-monthly?year&branchId`
  - **Service usage breakdown & Top services** (pie/table) → `GET /reports/service-usage?...`
- SRS requires these 3 key reports and emphasizes speed (≤5s).

(Optional) 🌍 PublicBooking.jsx

- Simple flow: **Select branch + dates + type** → see **available rooms** → **guest details** → **Confirm**.
- End with **Reference ID**; show **email sent** message.

🧪 Step 5 — Validation & accessibility (beginner checklist)

- Use **zod** schemas per form (dates, quantity > 0, amounts > 0).
- **Accessible labels**, keyboard nav, **screen-reader** text on icons; match SRS accessibility line.
- **Error toast** + **red borders** on invalid fields; **confirm modals** before destructive actions (check-out/delete).

🔗 Step 6 — Security on the UI (simple rules)

- Never inject raw HTML; escape user inputs (prevents XSS).
- Send JWT only via **Authorization header**; for cookies, ensure backend sets **HttpOnly + Secure + SameSite**.
- Always call backend over **HTTPS/TLS 1.3** in prod.
- **Auto-logout after 10 min** idle (SRS).

⚡ Step 7 — Performance targets (hit these)

- Code-split routes with `React.lazy`.
- Cache drop-downs (branches, room types, service catalog) in memory for the session.
- Keep page weight light so **UI loads ≤ 3s** (SRS).

🧭 Exactly what to code next (micro-tasks you can tick off)

1. **Scaffold routing** (`/login`, `/dashboard`, `/booking/new`, `/services`, `/billing/:id`, `/reports`, `/public`).
2. **AuthContext** with `{user, role, token, login(), logout()}` + **ProtectedRoute** wrapper.
3. **Login.jsx** form + POST `/auth/login`.
4. **Navbar/Sidebar** with role-aware links.
5. **Dashboard.jsx** cards calling occupancy + unpaid summary endpoints.
6. **NewBooking.jsx** form → availability search → POST booking.
7. **ServiceLog.jsx** form + table; POST to add usage.
8. **Billing.jsx** reads booking totals; “Add Payment”; **disable checkout** until paid.

9. **Reports.jsx** with three charts (occupancy, revenue, top services).
10. **PublicBooking.jsx** minimal flow.

All line up with SRS screens & features.

📦 Your deliverables to the team

- `skynest-web/` (React app) + **README** (how to run; `VITE_API_URL`).
- Screens matching SRS mockups (include screenshots).
- **Postman/Thunder collection** not needed here (Member 2 gives it), but ensure your env file points to the API.
- Short **accessibility & security** note: labels, shortcuts, HTTPS, idle logout.

🤝 How your work plugs into others

- **Member 2 (Backend):** you call their endpoints; they enforce business rules & totals.
- **Member 1 (DB):** their triggers/functions guarantee no overlap + checkout rules.
- **Member 4 (Security/QA):** they will test XSS, role-hiding, form validation, and "blocked checkout".
- **Member 5 (DevOps/Docs):** they'll deploy your app (CDN/static hosting) and set HTTPS.

🧾 SRS bits we followed (for your report write-up)

- **Screens & UI behavior** (login, dashboard, booking, services, billing, reports; inline errors, toasts, confirmations; keyboard shortcuts).
- **Public portal** (availability, booking, email confirmation).

- **Reports & speed** (occupancy, unpaid billing, monthly revenue; UI ≤3s).
- **Security & RBAC** (TLS 1.3, strong auth, role-based access, audit in backend).

---

want me to continue with **Member 4 (Security & QA)** in the same beginner style?

🛡️ Member 4 — Security & QA Lead

(Beginner-friendly. I'll tell you exactly what to do, step by step. You'll harden the app and prove it's safe + correct with tests.)

🎯 Your mission (in simple words)

1. **Build security into the system** (TLS, roles, strong passwords, audit logs, safe inputs).
2. **Test everything** the SRS promises (no double bookings, checkout blocked until paid, reports, performance).
3. **Document proofs** (checklists, test reports) for the team & demo.

> The SRS requires: **TLS 1.3**, **RBAC**, **strong passwords**, **audit trails**, **monthly security scans**, and protection against **SQL injection/XSS**; plus **auto-logout (10 min)** and **history of logins**. You must verify these.

🧩 What you must enforce/verify (map to SRS)

- **Transport security**: All web/API over **HTTPS/TLS 1.3**.
- **Access control (RBAC)**: admin / receptionist / accountant only see & do what they should.
- **Strong passwords** (≥8 chars, symbols, caps, numbers).
- **Session rules**: auto-logout after **10 minutes**; keep login/logout history; re-auth on role switch.
- **Audit trails**: critical actions kept **~2 years**; finance logs **5 years**.
- **Common attacks**: block **SQLi / XSS**; do **monthly scans**.
- **Business rules** (security-relevant):
  - Prevent **double-booking**.
  - **No check-out if dues remain**; allow **partial payments**; flag **unpaid**.

- **Performance SLAs**: search ≤2s, bill ≤1s, reports ≤5s, UI ≤3s. You'll test these.

  Slides you'll lean on for web security basics: **HTTP is stateless → sessions/cookies**, **prepared statements vs SQLi**, **XSS/CSRF basics**, **connection pooling**.

🛠️ Set up your toolbox (super simple)

- **Browser** + **Postman/Thunder** (manual API checks).
- **OWASP ZAP** (free DAST scanner) — monthly scan target in SRS.
- **npm** built-ins: `npm audit` (frontend), and review dependencies.
- **K6** or **Apache JMeter** (performance/load tests) to verify SLAs.
- **Lighthouse** (Chrome DevTools) for UI performance & accessibility (WCAG 2.1 AA).

🧭 Your step-by-step plan

1) 🔐 Security hardening checklist (do + tick)

Transport

- Confirm **TLS 1.3** is enabled in dev/prod (reverse proxy or hosting). Capture a screenshot of response headers.

Authentication & Passwords

- Verify password policy (registration/reset rejects weak). Record test results.

Sessions / JWT

- Enforce **auto-logout after 10 minutes** idle in UI; verify token expiry or inactivity timer. Log a test.
- **Log all login/logout** to audit store.
- Slides reminder: why sessions/cookies exist & how they work.

RBAC

- Create test users for **admin / receptionist / accountant**.
- Check each protected endpoint & page denies unauthorized roles (403).

Input validation

- Confirm **prepared statements** everywhere (ask backend for proof or code snippet), then try SQLi payloads (see below). Slides stress prepared statements.
- Ensure UI **never injects raw HTML** from users; encode output to reduce XSS.

Audit trail

- Verify records kept for **critical actions ~2y**; finance **5y**. Check schema/retention job.

Compliance & masking

- Ensure card details **tokenized via Stripe/PayHere**; only **last 4 digits** shown.

---

2) 🧪 Functional test cases (copy/paste and execute)

A. Double booking guard

- Steps: book Room 101 (Colombo) from 10–12 May; try second booking same room 11–13 May.
- Expect: **reject** at API; UI shows error; DB trigger/app rule prevents overlap. ✔️

B. Partial payments + checkout block

- Steps: create booking → add services → pay part of bill → attempt check-out.
- Expect: **check-out blocked** until balance = 0; after full payment, **status updates** and room becomes **Available**. ✔️

C. Service usage capture

- Steps: add Spa (qty 1, price snapshot) & Laundry (qty 2) to a booking.

- Expect: usage rows store **name, date, qty, price at time**, linked to booking & catalog. ✔️

D. Reports

- Steps: run **occupancy**, **billing summary (unpaid)**, **monthly revenue**.
- Expect: correct counts & sums; delivered **≤5s**. ✔️

---

3) 🧷 Security tests you can run today

SQL Injection (API)

- Try fields with payloads like `X' OR 'Y'='Y` on search/login/booking inputs.
- Expect: requests **fail safely**; no data leakage; server logs warn. Slides show why prepared statements matter. ✔️

XSS / CSRF

- Try posting `<img src=x onerror=alert(1)>` or `<script>…</script>` into name/notes fields.
- Expect: server/UI **sanitize or reject**; nothing pops; no DOM injection. Slides: XSS/CSRF guidelines. ✔️

Role bypass

- As **receptionist** try calling `/reports/revenue-monthly` (manager/admin only).
- Expect: **403 Forbidden**. ✔️

Password policy

- Try weak password `"abc123"` → **reject**; `"Passw0rd!"` → **accept**. ✔️

Session controls

- Stay idle 10+ minutes → user is logged out; login/logout entries appear in audit. ✔️

Monthly scan

- Run **OWASP ZAP** crawl+active scan against staging; export report **(Baseline + any Medium/High)**. ✔️

---

4) ⚡ Performance tests (SRS SLAs)

Use **K6/JMeter** to hit API with small realistic loads:

- **Search room** API: p95 ≤ **2s**. ✔️
- **Bill calculation** (booking summary): p95 ≤ **1s**. ✔️
- **Reports** endpoints: p95 ≤ **5s**. ✔️
- **UI** (Lighthouse): Performance score ~90+, **TTI ≤ 3s** on average laptop. ✔️

Slides tip: enable **connection pooling** on backend DB client for speed.

---

5) 📜 What to write down (your deliverables)

1. **Security Checklist (filled)**
    - TLS 1.3 evidence, password policy test, session timeout, RBAC matrix, masking, audit trail retention.
2. **Test Plan & Results**
    - Test cases above with pass/fail + screenshots; include negative cases (SQLi/XSS). Slides cover what those are.
3. **Performance Report**
    - JMeter/K6 results with p95 latencies vs SLA; Lighthouse screenshot.
4. **OWASP ZAP Report**
    - Monthly scan record + remediation notes.
5. **Audit & Logs Note**
    - Where logs go, retention (2y/5y), and how to export for compliance.

🗺️ Role matrix you can copy

- **Admin/Manager**: manage services, room types, pricing; run all reports.
- **Receptionist**: bookings, **check-in/out**, record services.
- **Accountant**: generate bills, **process payments**.

Create tests to ensure each role can only see/do its own screens & APIs.

✍️ Example “pass” criteria (use these in your report)

- 🔒 **TLS**: All API calls succeed only over `https://` (TLS 1.3); `http://` requests redirect/deny.
- 🧾 **Audit**: Every login/logout and payment has an audit entry with user, time, action; export works.
- 🚪 **Checkout rule**: API blocks checkout with balance>0; after final payment, checkout flips room to **Available**.
- 🧨 **SQLi/XSS**: No payload executes; inputs are validated; prepared statements used everywhere (verified).
- 👥 **RBAC**: Unauthorized role gets **403** on protected endpoints; UI hides disallowed menus.
- ⚡ **Performance**: Meets SLAs (2s/1s/5s/3s).

🧠 Tiny learning bites (so you’re not lost)

- **Why TLS 1.3?** Encrypts traffic; SRS mandates it.
- **Sessions & cookies**: web is stateless; we track users via cookies/JWT + timeouts.
- **Prepared statements**: stop SQL injection by not concatenating SQL.

- **XSS/CSRF**: never trust user HTML; block scripts; verify origins.
- **Audit trails**: needed at app & DB level; help detect and trace breaches.

🤝 How you sync with others

- **Member 2 (Backend)**: you request small changes if any security holes found (e.g., missing role checks, audit logs).
- **Member 3 (Frontend)**: you verify idle logout, input sanitization, error messages, and that disallowed menus are hidden.
- **Member 5 (DevOps)**: you confirm TLS, log storage, backups, and monthly scan pipeline. **99.9% uptime / backups / recovery** are tracked here.

If you're ready, say **"Member 5"** and I'll guide the **DevOps/Deployment & Docs Lead** (how to deploy with TLS 1.3, environment secrets, backups, monitoring, and monthly scan automation) 😊

⚙️ Member 5 — DevOps & Documentation Lead

(Beginner-friendly. I'll show you step by step how to deploy, secure, and document the system, even if you've never done it before.)

🎯 Your mission (in plain words)

1. **Deploy** the whole app (DB + API + Frontend) in a safe, reliable way.
2. **Apply operations requirements** from the SRS: TLS 1.3, secrets, backups, monitoring, 99.9% uptime.
3. **Write docs**: how to run it locally, how to deploy, and how to use it (user manual).
4. **Automate**: CI/CD, backups, security scans (monthly).

> The SRS non-functional requirements clearly call out: TLS 1.3, encryption, secure login/logout, uptime ≥99.9%, disaster recovery (backups, restore ≤1 hour), monitoring, audit retention (2y/5y). You're in charge of these.

🛠️ Step 0 — Install your tools

- **Docker Desktop** (runs containers on your machine).
- **VS Code**.
- A free **GitHub account** (CI/CD & repo hosting).
- **Netlify/Vercel** (for frontend) or Docker-based hosting (Heroku, Render, Railway).
- **MySQL Workbench** (if you want to peek at DB).

🧱 Step 1 — Containerize everything

Create a `docker-compose.yml` at the root of the project:

```yaml
version: '3.8'
services:
  db:
    image: mysql:8
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: ${DB_PASS}
      MYSQL_DATABASE: skynest
    volumes:
      - db_data:/var/lib/mysql
      - ./db/schema.sql:/docker-entrypoint-initdb.d/schema.sql
    ports:
      - "3306:3306"

  api:
    build: ./skynest-api
    restart: always
    depends_on:
      - db
    environment:
      DB_HOST: db
      DB_USER: root
      DB_PASS: ${DB_PASS}
      DB_NAME: skynest
      JWT_SECRET: ${JWT_SECRET}
    ports:
      - "4000:4000"

  web:
    build: ./skynest-web
    restart: always
    depends_on:
      - api
    ports:
      - "3000:3000"

volumes:
  db_data:
```

- **db**: loads schema + seed data from Member 1.

- **api**: backend built by Member 2.
- **web**: frontend built by Member 3.
- Use `.env` file to store **secrets** (DB_PASS, JWT_SECRET). Never hardcode secrets.

🔐 Step 2 — Secure the system

- **TLS 1.3**: use a reverse proxy like **NGINX** or **Caddy** in front of API & frontend. Issue free certs via **Let's Encrypt**. Verify with `curl -vk https://...` that TLS 1.3 is negotiated.
- **Secrets**: keep `.env` out of Git; use `.env.example` with dummy values.

Backups: add a cron job to dump DB daily:

```
mysqldump -u root -p$DB_PASS skynest > backup-$(date +%F).sql
```

- Store in **S3/Google Drive**. Test restore monthly.
- **Monitoring**: enable **health checks** (`/healthz` endpoint in API) and use **UptimeRobot** or GitHub Actions to ping every 5 min.
- **Audit logs**: keep in DB; set up DB retention policies (2y normal, 5y financial).
- **Disaster recovery**: document procedure: *"Restore DB dump into new MySQL; re-run docker-compose; re-point DNS; target recovery ≤1h"*.

📈 Step 3 — Automate (CI/CD)

1. Push code to GitHub.
2. Add GitHub Actions workflow (`.github/workflows/ci.yml`):
    - Install Node.js.
    - Run backend unit tests.

    - Run frontend build (`npm run build`).
    - Run `docker-compose up -d` and confirm API `/healthz`.
3. On `main` branch push, deploy automatically:
    - API → Render/Heroku (Docker).
    - Frontend → Netlify/Vercel.
    - DB → Managed MySQL (AWS RDS, Azure, etc.).

🧪 Step 4 — QA in deployment

- Verify **API + Web only run on HTTPS**.
- Test **load**: simulate 100 bookings/minute → system still responds <2s (SRS SLA).
- Test **failover**: kill API container → restart → still connects to DB (restart policy in compose).
- Test **backups**: restore yesterday's dump to staging DB → run reports.
- Test **audit logs**: login/logout, payments all visible.

📘 Step 5 — Documentation you must deliver

1) Deployment Guide

- Prerequisites (Docker, Node).
- How to run locally: `docker-compose up -d`.
- How to deploy on chosen cloud provider (steps).
- TLS 1.3 setup + verify.
- Backup/restore commands.

2) User Manual (non-technical)

- **Login** → enter branch, user, password.
- **Dashboard** → see today's check-ins/unpaid.
- **New Booking** → select guest, dates, room.
- **Service Log** → add services during stay.
- **Billing** → view charges, add payments, confirm checkout.
- **Reports** → run occupancy, revenue, top services.
  (Screenshots from Member 3's UI are required.)

3) Ops Notes

- **Security checklist** (TLS, secrets, RBAC, audit retention).
- **Monitoring setup** (UptimeRobot link).
- **Disaster recovery plan** (DB restore ≤1h, fallback DNS).
- **Monthly OWASP ZAP scan** process.

✅ Beginner-friendly task breakdown (so you don't get lost)

1. Install Docker Desktop.
2. Create `docker-compose.yml`.
3. Add `.env` file (DB_PASS, JWT_SECRET).
4. Run `docker-compose up -d`.
5. Test API at `http://localhost:4000/healthz`.
6. Test Web at `http://localhost:3000/`.
7. Set up **TLS** with Let's Encrypt on deployment host.

8. Add daily `mysqldump` cronjob for backups.
9. Add **UptimeRobot** monitor for API.
10. Write Deployment Guide + User Manual in Word/PDF.

🤝 How you connect with others

- **Member 1 (DB):** load their `.sql` into container init.
- **Member 2 (API):** expose port 4000; you wrap with TLS.
- **Member 3 (UI):** deploy on Netlify/Vercel; set `VITE_API_URL=https://api.skynest.com`.
- **Member 4 (Security/QA):** you schedule **monthly ZAP scans**, store reports, fix issues they find.

✅ That's it — you're the "glue" person: making sure everything runs, is safe, backed up, monitored, and documented.