Hotello — Updated Architecture Overview

Goal: Align the technical architecture to the Final Project Specification (deadline: 12 Oct 2025). This document replaces the previous architecture overview and maps implementation details directly to the five required tasks (Core booking & payment, My Account & Booking History, UI/UX & brand, Hotel listing & advanced filtering, AI search clear/reset).

---

1. High-level system summary

- **Frontend (SPA):** React + Vite + Tailwind CSS. Clerk for authentication, Stripe Embedded Checkout for payments, RTK Query for data fetching, small custom design system (tokens for color/typography). Mobile-first, accessible components and skeleton loaders.
- **Backend (API):** Node.js + Express (TypeScript recommended) + Mongoose (MongoDB). Clerk JWT validation middleware, Stripe SDK for creating checkout sessions & webhook handling, Gemini API integration for AI search/chat.
- **Database:** MongoDB Atlas. Collections: `users`, `hotels`, `bookings`, `payments`, `chats` (AI sessions). Indexes on query-heavy fields (hotels.location, hotels.pricePerNight, bookings.userId, bookings.checkIn/checkOut).
- **Deployments:** Frontend → Netlify/Vercel; Backend → Render (region: Singapore); MongoDB Atlas for DB. CI/CD triggers on GitHub pushes.

---

2. Core components & responsibilities

Frontend responsibilities

- Clerk UI + token handling for auth flows and protected endpoints.
- Hotel browsing (list / grid), hotel details, booking flow (date picker), payment page that renders Stripe Embedded Checkout with `clientSecret` fetched from backend.
- My Account page with booking history and booking detail cards.
- Hotel listing page with FilterSidebar (locations, price slider, amenities, star rating) and SortDropdown.
- Search bar integrating Gemini-powered AI search; supports clear/reset behaviour to restore filters and original hotel list.
- Uses RTK Query to fetch: `/api/hotels`, `/api/locations`, `/api/bookings/user/:userId`, `/api/payments/create-checkout-session`, `/api/payments/session-status`.

Backend responsibilities

- Authenticate/authorize requests by validating Clerk JWTs (middleware using `getAuth(req)` or token verification).
- Booking lifecycle: create booking with `paymentStatus: PENDING`, assign unique room numbers, validate dates and availability.
- Stripe integration: when creating hotels create Stripe product+default price (or store stripePriceId from seed). Create Checkout Session (embedded) with booking metadata and return `clientSecret`.
- Webhook endpoint `/api/stripe/webhook` receives raw body, verifies signature, and marks booking `PAID` using idempotent handlers.
- AI endpoints for search: `/api/search/ai` that proxies to Gemini and returns normalized hotel-like result objects with the same structure as `/api/hotels` so UI can render uniformly.

---

3. Data model (concise)

users

```
{
  _id,
  clerkId,        // string - Clerk user identifier
  name,
  email,
  role,           // 'user' | 'hotel_owner' | 'admin'
  preferences
}
```

hotels

```
{
  _id,
  name,
  description,
  location,        // string / { city, country }
  pricePerNight,   // number (decimal)
  amenities: [],
  images: [],
  stripePriceId,   // string - Stripe default price id
  rating,          // optional
  availability: [] // optional calendar / blocks
}
```

bookings

```
{
  _id,
  userId,
  hotelId,
  checkIn,
  checkOut,
  nights,
  roomNumber,
  totalAmount,
  paymentStatus,   // 'PENDING' | 'PAID' | 'FAILED' | 'EXPIRED'
  stripeSessionId?,
  createdAt
}
```

payments

```
{
  _id,
  bookingId,
  stripeSessionId,
  amount,
  status,
  createdAt,
  rawEvent?        // for auditing
}
```

chats

```
{
  _id,
  userId,
  messages: [{ role, content, timestamp }]
}
```

---

4. API contract (examples / critical endpoints)

Authentication

- All protected endpoints require Clerk token in `Authorization: Bearer <token>` (frontend uses `getToken()` / prepareHeaders).

Bookings

- `POST /api/bookings` → Body: `{ hotelId, checkIn, checkOut }`. Creates booking with `PENDING` and returns booking id and booking details.
- `GET /api/bookings/user/:userId` → Returns array of bookings (populated with hotel info) sorted most recent first. Supports pagination and filter by paymentStatus.

Payments

- `POST /api/payments/create-checkout-session` → Body: `{ bookingId }`. Returns `{ clientSecret }` for Embedded Checkout. Server must check booking is `PENDING` before creating session.

- `GET /api/payments/session-status?session_id=...` → Returns session status plus booking/hotel details; idempotently sets booking `PAID` if session indicates payment success.
- `POST /api/stripe/webhook` → Raw body; validates signature; handles `checkout.session.completed` and `checkout.session.async_payment_succeeded` and calls `fulfillCheckout(sessionId).`

Hotels & Filters

- `GET /api/hotels?location=&minPrice=&maxPrice=&sortBy=&page=` → Returns paginated hotels & supports server-side filtering.
- `GET /api/locations` → Returns unique locations for front-end filter dropdown.

AI Search

- `POST /api/search/ai` → Body: `{ query, filters? }` → returns results shaped like hotel list items. When AI search is active, front-end shows clear button to restore browse mode.

---

5. Booking & Payment flow (sequence summary)

1. User selects hotel + dates → Frontend calls `POST /api/bookings` → Booking created with `PENDING`, unique `roomNumber` assigned.
2. User clicks Pay → Frontend calls `POST /api/payments/create-checkout-session` with bookingId and Clerk token → Backend validates booking state and hotel has `stripePriceId`, creates Checkout Session with `metadata.bookingId` and returns `clientSecret`.
3. Frontend renders Stripe EmbeddedCheckout using clientSecret; user completes payment.
4. Stripe sends webhook event to `/api/stripe/webhook`. Backend verifies signature and `fulfillCheckout(sessionId)` which retrieves session, checks metadata.bookingId, ensures idempotency and sets booking `PAID`. Also create a `payments` record.
5. Frontend may call `GET /api/payments/session-status?session_id=...` after redirect to show confirmation.

Important implementation rules

- Webhook must use `bodyParser.raw({ type: 'application/json' })` and verify `STRIPE_WEBHOOK_SECRET`.
- Handlers must be idempotent (check booking.paymentStatus before updating).
- Protect `create-checkout-session` so duplicate sessions do not create double payments (check booking.paymentStatus).

---

6. AI Search & Clear behavior (Task 5)

- **State model:** `searchMode: boolean`, `searchQuery`, `searchResults`, `originalList`.
- **Clear options:** prominent "Clear Search" button in search UI, navigation option "Show All Hotels", and `Escape` keyboard shortcut.
- **Backend:** `/api/search/ai` returns results shaped identically to `/api/hotels` to avoid extra UI transformations.
- **UI:** When entering AI search mode hide or grey-out filter sidebar but keep a visible Clear control to restore filters and original dataset.

---

7. Deployment & environment variables

Frontend (.env.local)

```
VITE_BACKEND_URL=https://<your-backend>.onrender.com
VITE_CLERK_PUBLISHABLE_KEY=
VITE_STRIPE_PUBLISHABLE_KEY=
```

Backend (.env)

```
MONGODB_URL=
CLERK_SECRET_KEY=
CLERK_PUBLISHABLE_KEY=
STRIPE_SECRET_KEY=
STRIPE_WEBHOOK_SECRET=
FRONTEND_URL=https://<your-frontend>.netlify.app
OPENAI_API_KEY=
```

Notes: Use platform secret managers (Netlify/Render). Use Singapore region for Render as recommended by course guide.

---

8. Operational concerns & best practices

- **Indexing:** Add indexes on `hotels.location`, `hotels.pricePerNight`, `bookings.userId`, `bookings.createdAt`.
- **Idempotency & retries:** Webhook handler must be idempotent; use Stripe event IDs to avoid double-processing.
- **Rate limits:** When seeding Stripe products, add delay (~300ms) or batch seeding to avoid 429s.
- **Security:** Never expose secret keys; validate Clerk tokens on server; verify webhook signatures.

- **Testing:** Use Stripe test cards & webhook test events in staging; verify Embedded Checkout clientSecret flow.
- **Monitoring:** Log webhook failures and 4xx responses; alert on repeated webhook failures.

---

9. Mapping to Final Project Specification tasks

- **Task 1:** Booking & Stripe flows are implemented as described (PENDING → PAID, webhook, product/price creation). See API endpoints above.
- **Task 2:** My Account + Booking History maps to `GET /api/bookings/user/:userId` and frontend `/my-account` route with `BookingHistory` and `BookingCard` components.
- **Task 3:** Design tokens, custom Tailwind config, component library and logos to be built in Phase 3.
- **Task 4:** Server-side filtering endpoint `GET /api/hotels?...` + `GET /api/locations` support advanced filtering and URL state.
- **Task 5:** AI search clear/reset implemented by `POST /api/search/ai` + frontend state model and Clear Search UI.

---

10. Acceptance / success criteria (quick checklist)

- Live backend & frontend URLs deployed and working with HTTPS.
- Booking can be created in `PENDING` state and paid via embedded Stripe checkout; webhook updates booking to `PAID`.
- My Account page lists bookings with hotel details and supports filtering by status/date.
- Hotel listing supports location and price filtering, sorting, pagination, and URL state.
- AI search returns results and user can clear search to restore original filters and view.

Appendix: Implementation priorities (most to least)

1. Booking + Stripe + Deployment (Task 1) — mandatory.
2. My Account & booking history (Task 2).
3. Hotel listing + server-side filters + locations endpoint (Task 4).
4. AI search integration + Clear Search UX (Task 5).
5. UI/UX polish, custom design system, accessibility (Task 3).

Prepared to be integrated directly into the project README or architecture docs.