

Final Project Specification

Deadline: 12th October 2025

This document outlines the complete requirements for the final hotel booking platform project. Students must complete all 5 tasks to demonstrate full-stack development skills with modern web technologies.

Supporting Documents

[Booking Implementation and Stripe Integration](#)

[Deploying a MERN stack app on Netlify and Render](#)

Project Overview

Build a comprehensive hotel booking platform with payment processing, user account management, advanced filtering, AI search functionality, and a polished UI/UX. The platform should demonstrate proficiency in React, Node.js, MongoDB, Stripe payments, Clerk authentication, and modern design principles.

Task 1: Core Booking & Payment System + Deployment


Important: Follow the comprehensive guides provided for this task:

- **Booking Implementation:** Refer to the codebase and booking + stripe document
- **Stripe Integration:** Refer to the codebase and the booking + stripe document
- **Deployment:** Follow your deployment guide for production setup
- **Reference Repositories:**
 - Backend: <https://github.com/ManupaDev/aidf-5-back-end>
 - Frontend: <https://github.com/ManupaDev/aidf-5-front-end>

1.1 Booking Feature Implementation

Objective: Implement a complete hotel booking system with user authentication

Requirements:

-  **User Authentication:** Integrate Clerk for secure user login/signup

- ✓ **Hotel Selection:** Users can browse and select hotels
- ✓ **Date Selection:** Interactive date picker for check-in/check-out dates
- ✓ **Booking Creation:** Create booking records with PENDING status
- ✓ **Room Assignment:** Automatic unique room number generation
- ✓ **Validation:** Proper input validation and error handling

Implementation: Follow existing codebase patterns and the Stripe integration guide.

1.2 Stripe Payment Integration

Objective: Implement secure payment processing using Stripe

Requirements:

- ✓ **Product Catalog:** Stripe Products with default Prices for each hotel
- ✓ **Embedded Checkout:** Stripe Embedded Checkout integration
- ✓ **Payment Sessions:** Checkout Session creation with metadata linking
- ✓ **Webhook Handling:** Process payment confirmations asynchronously
- ✓ **Status Management:** Update booking status from PENDING to PAID
- ✓ **Error Handling:** Handle payment failures and edge cases

Implementation: Follow the detailed Stripe Integration documentation which includes:

- Step-by-step Stripe dashboard configuration
- Code examples from the actual codebase
- Webhook setup and handling
- Environment variable configuration
- Security best practices

1.3 Deployment

Objective: Deploy both frontend and backend to production

Requirements:

- **Backend Deployment:** Deploy to cloud platform (Render, Railway, Heroku, etc.)
- **Frontend Deployment:** Deploy to Vercel, Netlify, or similar
- **Database:** MongoDB Atlas production cluster
- **Environment Variables:** Secure configuration management
- **HTTPS:** SSL certificates for secure communication
- **Webhook Configuration:** Production webhook endpoints
- **Testing:** Verify end-to-end functionality in production

Implementation: Follow your deployment guide for:

- Platform-specific deployment steps
- Environment variable configuration
- Database setup and connection
- Domain and SSL configuration
- Production testing procedures

Deliverables:

- Live backend URL
 - Live frontend URL
 - Functional payment flow in production
 - Proper error handling and logging
-

Task 2: My Account Page & Booking History

2.1 Account Dashboard

Objective: Create a comprehensive user account management page

Requirements:

- **User Profile:** Display user information from Clerk
- **Account Settings:** Basic profile management
- **Navigation:** Clear navigation between account sections
- **Responsive Design:** Mobile-friendly layout
- **Loading States:** Proper loading indicators

2.2 Booking History

Objective: Display user's booking history with detailed information

Requirements:

- **Chronological Order:** Bookings sorted by date (most recent first)
- **Booking Details:**
 - Hotel name, image, and location
 - Check-in and check-out dates
 - Room number
 - Payment status (PENDING/PAID)
 - Total amount paid
 - Booking date/time
- **Status Indicators:** Visual indicators for payment status
- **Empty State:** Meaningful message when no bookings exist
- **Filtering:** Filter by payment status or date range

2.3 Technical Implementation

Backend Requirements:

```
// New API endpoints needed:  
GET /api/bookings/user/:userId - Get user's bookings  
// Enhanced booking response with hotel details populated
```

Frontend Requirements:

- New route: /my-account
- Components: AccountDashboard, BookingHistory, BookingCard
- State management: RTK Query for booking data
- UI: Cards, tables, or list view for bookings

Database Considerations:

- Efficient queries with proper indexing
 - Population of hotel details in booking queries
 - Pagination support for large datasets
-

Task 3: UI/UX Customization & Brand Identity

3.1 Design System

Objective: Create a unique, cohesive design system that differentiates your project

Requirements:

- **Custom Color Palette:** Move beyond default Tailwind colors
- **Typography:** Custom font selections and hierarchy
- **Component Library:** Consistent button, card, and form styles
- **Spacing & Layout:** Consistent spacing and grid systems
- **Animation & Transitions:** Subtle animations for better UX

3.2 Visual Identity

Objective: Establish a strong brand presence

Requirements:

- **Logo Design:** Custom logo and branding elements
- **Imagery Strategy:** Consistent photo treatment and placeholders

- **Iconography:** Custom or consistent icon set
- **Layout Patterns:** Unique page layouts and component arrangements
- **Micro-interactions:** Hover effects, loading states, success animations

3.3 User Experience Improvements

Objective: Enhance usability and user satisfaction

Requirements:

- **Navigation:** Intuitive navigation structure
- **Accessibility:** WCAG compliance, keyboard navigation, screen reader support
- **Performance:** Optimized images, lazy loading, fast interactions
- **Mobile-First:** Excellent mobile experience
- **Feedback Systems:** Clear success/error messages, loading states

3.4 Inspiration Areas

Consider these elements to make your project unique:

- **Dark/Light Mode:** Theme switching capability
 - **Glassmorphism/Neumorphism:** Modern design trends
 - **Custom Illustrations:** Hand-drawn or custom graphics
 - **Unique Layouts:** Masonry grids, asymmetrical designs
 - **Advanced Animations:** Framer Motion, GSAP, or CSS animations
 - **3D Elements:** Three.js for interactive elements
-

Task 4: Hotel Listing Page with Advanced Filtering

4.1 Hotel Listing Page

Objective: Create a dedicated page to browse all available hotels

Requirements:

- **Grid/List View:** Toggle between card grid and detailed list view
- **Hotel Cards:**
 - High-quality images with image carousel
 - Hotel name, location, rating
 - Price per night
 - Key amenities
 - "View Details" and "Book Now" actions
- **Pagination:** Handle large hotel datasets efficiently

- **Loading States:** Skeleton screens during data fetching
- **Empty States:** Handle no results gracefully

4.2 Location Filtering

Objective: Allow users to filter hotels by location

Requirements:

- **Location Dropdown:** Searchable dropdown with all available locations
- **Multi-select:** Allow selection of multiple locations
- **Location Chips:** Show selected locations as removable chips
- **Clear Filters:** Easy way to reset location filters
- **URL State:** Preserve filters in URL for bookmarking/sharing

4.3 Price Sorting & Filtering

Objective: Enable price-based filtering and sorting

Requirements:

- **Price Range Slider:** Interactive slider for min/max price selection
- **Sort Options:**
 - Price: Low to High
 - Price: High to Low
 - Rating: High to Low
 - Alphabetical: A-Z
 - Featured/Recommended
- **Dynamic Updates:** Real-time filtering without page refresh
- **Price Display:** Clear price formatting with currency

4.4 Advanced Filtering (Bonus)

Additional filtering options:

- **Star Rating:** Filter by hotel star rating
- **Amenities:** Filter by specific amenities (WiFi, Pool, Gym, etc.)
- **Guest Rating:** Filter by user review scores
- **Availability:** Filter by check-in/check-out date availability

4.5 Technical Implementation

Backend Requirements:

```
// Enhanced API endpoints:
GET /api/hotels?location=...&minPrice=...&maxPrice=...&sortBy=...&page=...
```

GET /api/locations - Get all unique locations for filter dropdown

Frontend Requirements:

- New route: /hotels
 - Components: HotelGrid, HotelCard, FilterSidebar, SortDropdown
 - State management: URL parameters, query state management
 - Performance: Debounced search, optimistic updates
-

Task 5: AI Search Clear & Reset Functionality

5.1 Current Search Issue

Problem: In the current system, when a user enters a search query and clicks "AI Search", the location filters and hotel list are replaced with search results. However, there's no way for users to clear the search and return to viewing all hotels.

Current Flow:

1. User sees location filters and full hotel list
2. User enters search query and clicks "AI Search"
3. UI shows only search results (filters disappear)
4. **Issue:** No way to clear search and return to original view

5.2 Required Solution

Objective: Implement clear search functionality to improve user experience

Requirements:

- **Clear Search Button:** Add a prominent "Clear Search" or "Show All Hotels" button
- **Search State Management:** Properly manage search state in the application
- **UI Restoration:** When cleared, restore the original location filters and full hotel list
- **Visual Feedback:** Clear indication of current search state (searching vs. showing all)
- **Multiple Clear Options:**
 - Clear button in search results area
 - Option to clear by clicking "Show All Hotels" in navigation
 - Keyboard shortcut (Escape key) to clear search

5.3 Technical Implementation

Frontend Requirements:

- **State Management:** Track whether user is in "search mode" or "browse mode"
- **Conditional Rendering:** Show appropriate UI based on current state
- **Search Reset:** Clear search query and restore original hotel list
- **Filter Restoration:** Re-enable location filters when search is cleared

Backend Considerations:

- **API Endpoints:** Ensure `/api/hotels` endpoint works for both search and browse modes
- **Data Consistency:** Maintain consistent data structure between search and browse views

5.4 Implementation Details

Components to Modify/Create:

- **SearchBar Component:** Add clear functionality
- **HotelList Component:** Handle search vs. browse state
- **FilterSidebar Component:** Show/hide based on current mode
- **SearchResults Component:** Display search results with clear option

State Management:

```
// Example state structure
const [searchMode, setSearchMode] = useState(false);
const [searchQuery, setSearchQuery] = useState('');
const [searchResults, setSearchResults] = useState([]);

// Clear search function
const clearSearch = () => {
  setSearchMode(false);
  setSearchQuery('');
  setSearchResults([]);
  // Restore original hotel list and filters
};
```

API Integration:

- Maintain existing search endpoint functionality
 - Ensure browse endpoint returns full hotel list with filters
 - Handle state transitions smoothly
-

Booking Implementation and Stripe Integration

This guide explains in depth how Stripe is integrated in this project, step-by-step, with API examples and dashboard instructions. It targets developers familiar with REST APIs and React.

- Backend: `aidf-5-back-end` (Node/Express/Mongoose/Clerk)
 - Frontend: `aidf-5-front-end` (React, RTK Query, Clerk, Stripe Embedded Checkout)
-

Understanding Stripe Payment Architecture

High-Level Payment Flow

Stripe payment systems typically follow this pattern:

1. **Product Catalog:** Define what you're selling (products/prices)
 1. **Payable Entity Creation:** Create a record representing what the user is purchasing
2. **Payment Intent:** Create a payment session when user wants to pay
3. **Payment Collection:** Use Stripe's UI components to collect payment
4. **Payment Processing:** Stripe handles the actual payment processing
5. **State Transition:** Update the payable entity's payment status on successful completion

Core Payment Concept: Payable Entities

What is a Payable Entity? A payable entity is a database record that represents something a user is purchasing. It serves as the bridge between your business logic and the payment system.

Key Characteristics:

- **Linked to User:** Every payable entity belongs to a specific user
- **Linked to Product/Service:** Represents what the user is buying
- **Payment State:** Tracks the payment status (PENDING, PAID, FAILED, etc.)
- **Unique Identifier:** Used to reconcile payments with business records

Examples of Payable Entities:

- **Booking:** Hotel room reservation (our case)
- **Order:** E-commerce purchase

- **Subscription:** Recurring service
- **Invoice:** Service billing

Payment State Management Flow

The payment integration follows this state-driven approach:

1. **CREATE PAYABLE ENTITY**
User initiates purchase → Create payable entity with `PENDING` status
2. **INITIATE PAYMENT**
User clicks "Pay" → Create Stripe payment session with entity metadata
3. **PAYMENT PROCESSING**
User completes payment → Stripe processes the transaction
4. **STATE TRANSITION**
Payment succeeds → Update payable entity to `PAID` status
Payment fails → Update payable entity to `FAILED` status

Why This Pattern?

- **Reliability:** Payment processing is asynchronous - users might close their browser
- **Audit Trail:** Every transaction is tracked in your database
- **Business Logic:** You can enforce business rules based on payment status
- **Reconciliation:** Easy to match Stripe payments with your records

Our Implementation: Booking as Payable Entity

In our hotel booking system:

Booking Entity Structure:

```
{
  _id: "booking_123",
  userId: "user_456",           // Links to user
  hotelId: "hotel_789",         // Links to product/service
  checkIn: "2024-01-15",
  checkOut: "2024-01-17",
  roomNumber: "101",
  paymentStatus: "PENDING",     // Payment state
  paymentMethod: "CARD",
}
```

Payment Flow:

1. **Create Booking:** User selects dates → Create booking with `paymentStatus: "PENDING"`

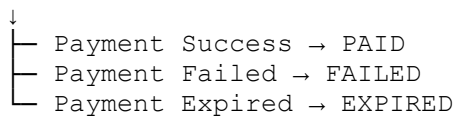
2. **Initiate Payment:** User clicks "Pay" → Create Stripe Checkout Session with `bookingId` in metadata
3. **Process Payment:** Stripe handles payment collection and processing
4. **Update Status:** Webhook receives `checkout.session.completed` → Update booking to `paymentStatus: "PAID"`

State Transitions:

- `PENDING` → `PAID`: Payment successful
- `PENDING` → `FAILED`: Payment failed or expired
- `PAID` → `REFUNDED`: Refund processed (future enhancement)

Payment Status Lifecycle

`PENDING` (Initial State)



Business Rules Based on Status:

- `PENDING`: Booking exists but not confirmed, can be cancelled
- `PAID`: Booking confirmed, room reserved, user can check in
- `FAILED`: Booking invalid, user must retry payment
- `EXPIRED`: Booking expired, user must create new booking

Metadata Linking Strategy

Why Metadata? Stripe webhooks don't know about your business entities. Metadata creates the link:

```
// When creating Checkout Session (from payment.ts)
const session = await stripe.checkout.sessions.create({
  ui_mode: "embedded",
  line_items: [lineItem],
  mode: "payment",
  return_url:
`${FRONTEND_URL}/booking/complete?session_id={CHECKOUT_SESSION_ID}`,
  metadata: {
    bookingId: booking._id.toString(), // Links Stripe session to our
    booking
  },
});

// In webhook handler (from payment.ts)
const booking = await Booking.findById(checkoutSession.metadata?.bookingId);
if (checkoutSession.payment_status !== "unpaid") {
  await Booking.findByIdAndUpdate(booking._id, { paymentStatus: "PAID" });
}
```

}

Error Handling & Edge Cases

Duplicate Payments:

- User clicks "Pay" multiple times → Create multiple sessions for same booking
- Solution: Check if booking already has `PAID` status before creating session

Webhook Failures:

- Stripe webhook fails → Booking stays `PENDING` but payment succeeded
- Solution: Implement status polling as backup, retry webhook processing

Race Conditions:

- User completes payment while webhook is processing
- Solution: Use database transactions, idempotent webhook handlers

Payment Expiration:

- User abandons payment → Session expires
- Solution: Implement cleanup job to mark expired bookings as `EXPIRED`

Key Concepts

Products & Prices

- **Product:** Represents a sellable item (e.g., "Hotel Room")
- **Price:** Defines how much to charge (e.g., "\$100/night")
- Products can have multiple prices (different currencies, billing intervals)

Checkout Sessions: The Payment Bridge

What is a Checkout Session? A Checkout Session is Stripe's way of creating a secure payment experience. Think of it as a "payment container" that holds all the information needed to process a payment for a specific payable entity.

Key Characteristics:

- **Temporary:** Sessions expire after 24 hours (configurable)
- **Secure:** Handles PCI compliance and sensitive payment data
- **Stateful:** Tracks payment progress (open, complete, expired)
- **Linked:** Connected to your payable entity via metadata

How Checkout Sessions Relate to Payable Entities:

Payable Entity (Booking)

```
{
  _id: "booking_123"
  userId: "user_789"
  hotelId: "hotel_101"
  paymentStatus: "PENDING"
  stripeSessionId: null
}
```

Checkout Session (Stripe)

```
{
  id: "cs_stripe_456"
  client_secret: "cs_..."
  status: "open"
  metadata: {
    bookingId: "123"
    userId: "789"
  }
}
```

Session Lifecycle:

1. **Created:** When user clicks "Pay" → Session created with status: "open"
2. **Active:** User fills payment form → Session remains status: "open"
3. **Completed:** Payment succeeds → Session becomes status: "complete"
4. **Expired:** User abandons → Session becomes status: "expired"

Why Use Checkout Sessions?

- **PCI Compliance:** Stripe handles sensitive payment data
- **User Experience:** Pre-built, tested payment forms
- **Security:** Built-in fraud protection and 3D Secure
- **Reliability:** Handles edge cases (network issues, browser crashes)

Session Types:

- **Hosted Checkout:** Redirects user to Stripe's payment page
- **Embedded Checkout:** Renders payment form within your app (our choice)

Our Implementation Flow:

1. User creates booking (PENDING status)
↓
2. User clicks "Pay" → Create Checkout Session
↓
3. Frontend renders embedded checkout form
↓
4. User completes payment → Session status: "complete"
↓
5. Webhook fires → Update booking status: "PAID"

Code Example - Creating a Session:

```
// Backend: Create Checkout Session (from payment.ts)
const session = await stripe.checkout.sessions.create({
```

```

    ui_mode: "embedded",
    line_items: [lineItem],
    mode: "payment",
    return_url:
`${FRONTEND_URL}/booking/complete?session_id={CHECKOUT_SESSION_ID}`,
    metadata: {
      bookingId: booking._id.toString(),
    },
  });

res.send({ clientSecret: session.client_secret });

```

Code Example - Using Session in Frontend:

```

// Frontend: Render embedded checkout (from CheckoutForm.jsx)
const fetchClientSecret = useCallback(async () => {
  const token = await getToken();
  const res = await fetch(
    `${BACKEND_URL}/api/payments/create-checkout-session`,
    {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
        Authorization: `Bearer ${token}`,
      },
      body: JSON.stringify({ bookingId }),
    }
  );
  const data = await res.json();
  return data.clientSecret;
}, [bookingId, getToken]);

return (
  <EmbeddedCheckoutProvider stripe={stripePromise} options={{
    fetchClientSecret
  }}>
    <EmbeddedCheckout />
  </EmbeddedCheckoutProvider>
);

```

Session vs Payment Intent:

- **Checkout Session:** Higher-level, includes UI, handles complex flows (what we use)
- **Payment Intent:** Lower-level, custom UI, more control but more complexity (we don't use this)

Why We Use Checkout Sessions (Not Payment Intents):

- **Simpler Integration:** Less code, fewer edge cases to handle
- **Built-in UI:** Stripe provides the payment form, we just embed it
- **PCI Compliance:** Stripe handles all sensitive payment data
- **SCA Support:** Automatic Strong Customer Authentication handling

- **Error Handling:** Stripe manages payment failures and retries

Session Metadata Strategy:

```
// Essential metadata for webhook processing (from payment.ts)
metadata: {
  bookingId: booking._id.toString(), // Primary link to payable entity
}
```

Session Status Handling:

```
// Webhook handler (from payment.ts)
if (
  event.type === "checkout.session.completed" ||
  event.type === "checkout.session.async_payment_succeeded"
) {
  await fulfillCheckout((event.data.object as any).id);
  res.status(200).send();
  return;
}

// fulfillCheckout function
async function fulfillCheckout(sessionId: string) {
  const checkoutSession = await stripe.checkout.sessions.retrieve(sessionId,
  {
    expand: ["line_items"],
  });

  const booking = await
  Booking.findById(checkoutSession.metadata?.bookingId);
  if (!booking) {
    throw new Error("Booking not found");
  }

  if (booking.paymentStatus !== "PENDING") {
    return; // already handled
  }

  if (checkoutSession.payment_status !== "unpaid") {
    await Booking.findByIdAndUpdate(booking._id, { paymentStatus: "PAID" });
  }
}
```

Payment Sessions

- **Checkout Session:** Stripe's hosted payment page (redirect or embedded) - **This is what we use**
- **Payment Intent:** For custom payment flows with your own UI - **We don't use this**
- **Client Secret:** A token that allows frontend to complete payment securely

Webhooks

What are Webhooks? Webhooks are HTTP callbacks that Stripe sends to your server when events occur. Think of them as "push notifications" from Stripe to your backend.

Why Webhooks?

- **Reliability:** Payment processing is asynchronous - users might close their browser before payment completes
- **Security:** Server-to-server communication ensures payment events can't be spoofed
- **Real-time Updates:** Your database stays synchronized with Stripe's payment status

Common Webhook Events:

- `checkout.session.completed` - Payment successful
- `payment_intent.succeeded` - Payment processed
- `invoice.payment_failed` - Payment failed
- `customer.subscription.created` - Subscription started

Security Model

- **Publishable Key:** Safe for frontend (identifies your Stripe account)
- **Secret Key:** Server-only (can create charges, access sensitive data)
- **Webhook Secret:** Validates webhook authenticity (prevents fake events)

Payment Architecture Patterns

Pattern 1: Hosted Checkout (What we use)

Frontend → Backend → Stripe Checkout → User pays → Webhook → Backend updates DB

- **Pros:** PCI compliant, handles complex flows (SCA, 3D Secure)
- **Cons:** Less customization, redirects user away

Pattern 2: Embedded Checkout (What we use)

Frontend (with Stripe.js) → Backend → Stripe API → Embedded UI → Webhook → Backend

- **Pros:** Stays in your app, still PCI compliant
- **Cons:** More complex integration

Pattern 3: Custom Payment Flow

Frontend (custom UI) → Backend → Payment Intent → Frontend confirms → Webhook

- **Pros:** Full control over UI
- **Cons:** Must handle PCI compliance, SCA, error states

Webhook Handling Best Practices

1. **Idempotency:** Handle duplicate webhook events gracefully
2. **Signature Verification:** Always verify webhook signatures
3. **Async Processing:** Don't block webhook response with heavy operations
4. **Retry Logic:** Stripe retries failed webhooks, so make your endpoint idempotent
5. **Event Ordering:** Events may arrive out of order - design for eventual consistency

Our Implementation Strategy

We use **Embedded Checkout** because:

- Users stay within our React app
- Stripe handles PCI compliance and SCA
- We get real-time payment status updates
- Simpler than custom payment flows

Our Flow:

1. User books hotel → Create booking with `PENDING` status
2. User clicks "Pay" → Create Checkout Session with booking metadata
3. Stripe renders embedded payment form
4. Payment completes → Webhook updates booking to `PAID`
5. User sees confirmation page

Before you begin

- Follow your deployment guide to deploy the backend and frontend first.
- You will need the deployed backend base URL (referred to as `BACKEND_URL`) to configure the Stripe webhook destination as
`${BACKEND_URL}/api/stripe/webhook.`

Architecture overview

1. A user books a hotel via `POST /api/bookings`. We capture `userId` from Clerk using `getAuth(req)`.
2. The app creates a Stripe Product with a default Price for each hotel. This happens in two places:

- Automatically when creating a hotel via API.
 - During seeding, sequentially with a small delay to avoid rate limits.
 - 3. To pay for a booking, the frontend requests a Checkout Session from `POST /api/payments/create-checkout-session`.
 - 4. The backend returns a `client_secret`. The frontend embeds Stripe's Checkout using this secret.
 - 5. When payment completes, Stripe redirects back to `/booking/complete?session_id=...`
 - 6. The frontend calls `GET /api/payments/session-status` to show a confirmation view. The backend also handles a webhook at `/api/stripe/webhook` to mark bookings as PAID.
-

Core Stripe concepts used (with rationale)

- **Product:** Represents a sellable item. We create a product per hotel so pricing can be reused across bookings.
 - **Price:** Represents how much is charged. We attach a default price to each product (USD; nightly rate in cents). This lets us pass a simple Price ID when creating Checkout Sessions.
 - **Checkout Session (Embedded):** Stripe's hosted payment UI integrated within our page. We use `ui_mode: "embedded"` to keep the user within our SPA while Stripe handles PCI compliance and SCA.
 - **Metadata:** We attach `bookingId` to the Checkout Session so webhooks/status can link payment events back to the booking.
 - **Webhooks:** Server-to-server notifications from Stripe. We validate the signature (`STRIPE_WEBHOOK_SECRET`) and update the booking's `paymentStatus = PAID`.
 - **Keys:** Backend uses `STRIPE_SECRET_KEY` (server-only). Frontend uses `VITE_STRIPE_PUBLISHABLE_KEY` to initialize Stripe.js.
-

Prerequisites

Important: Follow the deployment guide first to set up your backend and frontend applications. The deployment guide has already configured all necessary environment variables except for the Stripe-related ones listed below.

New Environment Variables

Backend (`aidf-5-back-end/.env`):

```
STRIPE_SECRET_KEY=  
STRIPE_WEBHOOK_SECRET=
```

Frontend (aidf-5-front-end/.env.local):

VITE_STRIPE_PUBLISHABLE_KEY=

Notes:

- Never expose STRIPE_SECRET_KEY or STRIPE_WEBHOOK_SECRET to the client.

Stripe Dashboard: step-by-step

1. Get keys
 - Dashboard → Developers → API keys
 - Copy Publishable key (frontend) and Secret key (backend).
2. Create a webhook endpoint
 - Dashboard → Developers → Webhooks → Add endpoint
 - Endpoint URL: \${BACKEND_URL}/api/stripe/webhook (your deployed backend URL + /api/stripe/webhook)
 - Select events:
 - checkout.session.completed
 - checkout.session.async_payment_succeeded
 - Save and copy the "Signing secret" and set STRIPE_WEBHOOK_SECRET in .env.

Data model changes

- Hotel:
 - stripePriceId: string (default Price ID for the hotel's Product). Mandatory for checkout in our implementation.
- Booking:
 - paymentStatus: "PENDING" | "PAID" (created as PENDING, becomes PAID after payment)

Backend implementation details

1) Creating a hotel: Product + default Price

- File: aidf-5-back-end/src/application/hotel.ts (function createHotel)

- After validating input and generating an embedding, we call:

```
const product = await stripe.products.create({
  name: result.data.name,
  description: result.data.description,
  default_price_data: {
    unit_amount: Math.round(result.data.price * 100),
    currency: "usd",
  },
});
```

- We extract `default_price` from the product and store it as `stripePriceId` on the Hotel record.

Admin re-setup endpoint:

- `POST /api/hotels/:_id/stripe/price` → Recreates a product with `default_price_data` for a hotel and updates `stripePriceId`.

2) Seeding hotels and creating Stripe products sequentially

- File: `aidf-5-back-end/src/seed.ts`
- After inserting hotels, we loop over each hotel:
 - Create Stripe product with `default_price_data`
 - Store `stripePriceId` on the hotel
 - Wait ~300ms between requests to avoid 429 rate limits.

3) Creating a booking (user attribution)

- File: `aidf-5-back-end/src/application/booking.ts` → `createBooking`
- Extract user using Clerk:

```
const { userId } = getAuth(req);
if (!userId) {
  throw new UnauthorizedError("Unauthorized");
}
```

- Create the booking with `PENDING` payment status.

4) Starting a Checkout Session (Embedded)

- File: `aidf-5-back-end/src/application/payment.ts` → `createCheckoutSession`
- For a booking with check-in/out dates, we compute number of nights and create:

```
if (!hotel.stripePriceId) {
  return res.status(400).json({ message: "Stripe price ID is missing for this hotel" });
}
```

```

}
const session = await stripe.checkout.sessions.create({
  ui_mode: "embedded",
  line_items: [{ price: hotel.stripePriceId, quantity: numberOfNights }],
  mode: "payment",
  return_url:
`${FRONTEND_URL}/booking/complete?session_id={CHECKOUT_SESSION_ID}`,
  metadata: { bookingId: booking._id.toString() },
});

```

- We intentionally do not use `price_data` here; a price ID and quantity are sufficient.

Example request:

```

POST /api/payments/create-checkout-session
Authorization: Bearer <clerk-jwt>
Content-Type: application/json

{ "bookingId": "<mongo_id>" }

```

Example response:

```

{ "clientSecret": "cs_test_..." }

```

5) Retrieving session status (and idempotent mark as PAID)

- **File:** `aidf-5-back-end/src/application/payment.ts` → `retrieveSessionStatus`
- `GET /api/payments/session-status?session_id=...`
- **Returns** `{ booking, hotel, status, customer_email, paymentStatus }` **and** updates booking to PAID if the session is paid.

6) Webhook verification and fulfillment

- **Files:**
 - `aidf-5-back-end/src/index.ts` (register raw-body route BEFORE JSON parser)
 - `aidf-5-back-end/src/application/payment.ts` (`handleWebhook`, `fulfillCheckout`)
- **Route registration:**

```

app.post(
  "/api/stripe/webhook",
  bodyParser.raw({ type: "application/json" }),
  handleWebhook
);

```

- The webhook destination in Stripe Dashboard must point to your deployed backend:
`${BACKEND_URL}/api/stripe/webhook`.
- Handler (simplified):

```
const event = stripe.webhooks.constructEvent(payload, sig, endpointSecret);
if (event.type === "checkout.session.completed" || event.type ===
"checkout.session.async_payment_succeeded") {
  await fulfillCheckout((event.data.object as any).id);
}
```

- `fulfillCheckout` retrieves the session, reads `metadata.bookingId`, and sets the booking's `paymentStatus` to `PAID`.

Security notes:

- Webhook route must NOT use `express.json()`; it must receive the raw body.
- Always verify the signature (`STRIPE_WEBHOOK_SECRET`).

Frontend implementation details

1) API layer

- File: `aidf-5-front-end/src/lib/api.js`
- Exposes RTK Query endpoints:
 - `createBooking` (POST bookings)
 - `getBookingById` (GET bookings/:id)
 - `createCheckoutSession` (POST payments/create-checkout-session)
 - `getCheckoutSessionStatus` (GET payments/session-status?session_id=...)
- Uses Clerk token authentication in `prepareHeaders`

2) Embedded Checkout component

- File: `aidf-5-front-end/src/components/CheckoutForm.jsx`
- Loads Stripe with `VITE_STRIPE_PUBLISHABLE_KEY` and fetches the `clientSecret` from the backend using the Clerk token.
- Renders `<EmbeddedCheckoutProvider stripe={stripePromise} options={{ fetchClientSecret }}>` and `<EmbeddedCheckout />`.

3) Booking flow pages

- `hotel-details.page.jsx`: launches `BookingDialog` and then navigates to `/booking/payment?bookingId=....`

- `payment.page.jsx`: renders `CheckoutForm`.
 - `complete.page.jsx`: reads `session_id` and calls `getCheckoutSessionStatus`, then shows a success view.
-

Local development steps

1. Configure env files for both backend and frontend (see above).
2. Install dependencies in both apps (already in repo, but run `npm i` if needed).
3. Start backend:

```
cd aidf-5-back-end
npm run dev
```

1. Start frontend:

```
cd aidf-5-front-end
npm run dev
```

1. Seed data (optional):

```
cd aidf-5-back-end
npm run seed
```

1. In the browser:
 - Go to a hotel details page → Book Now → pick dates → submit.
 - Redirected to `/booking/payment` → Embedded Checkout loads.
 - Complete payment with test card (e.g., 4242 4242 4242 4242, future expiry, any CVC).
 - Redirect to `/booking/complete?session_id=...` → see confirmation.
-

Production checklist

- Set `FRONTEND_URL` to the deployed frontend.
 - Configure a production webhook endpoint to your deployed backend.
 - Store secrets in your platform's secret manager (e.g., Vercel/Netlify/Render/AWS/GCP).
 - Use HTTPS in production for webhook and app URLs.
 - Rotate keys regularly in Stripe Dashboard.
 - Monitor failed webhooks and 4xx from your endpoint.
-

Troubleshooting

- Embedded Checkout not rendering: ensure `VITE_STRIPE_PUBLISHABLE_KEY` is valid and the `create-checkout-session` call succeeds with a `clientSecret`.
 - 401/403 from backend: ensure Clerk token is present (frontend `prepareHeaders` waits for Clerk) and role checks are satisfied for admin routes.
 - Webhook 400 “No signatures found” or “Invalid payload”: confirm the route uses raw body, the endpoint URL matches Stripe Dashboard, and `STRIPE_WEBHOOK_SECRET` is correct.
 - Seeding 429 rate limits: increase the delay between Stripe product creations (currently ~300ms) or run in batches.
 - Missing price: use admin route `POST /api/hotels/:_id/stripe/price` or recreate the hotel; checkout requires `stripePriceId`.
-

References (current)

- Embedded Checkout: <https://docs.stripe.com/checkout/embedded>
- Webhooks: <https://docs.stripe.com/webhooks>
- Checkout Sessions API: <https://docs.stripe.com/api/checkout/sessions>
- Prices: <https://docs.stripe.com/api/prices>
- Products: <https://docs.stripe.com/api/products>
- Testing: <https://docs.stripe.com/testing>

Deploying a MERN stack app on Netlify and Render

Due date: 8th August 2025

Before using the guide make sure you complete the project until the level we have done till 13th September (Session 16) and have all the environment variables required to run our app with you.

Front-end

```
VITE_CLERK_PUBLISHABLE_KEY=  
VITE_BACKEND_URL=  
  
// We will add these after the initial deployment when we do the stripe  
integration.  
VITE_STRIPE_PUBLISHABLE_KEY=
```

Back-end

```
MONGODB_URL=  
  
CLERK_PUBLISHABLE_KEY=  
CLERK_SECRET_KEY=  
  
OPENAI_API_KEY=  
  
// We will add these after the initial deployment when we do the stripe  
integration.  
STRIPE_SECRET_KEY=  
STRIPE_WEBHOOK_SECRET=
```

Deploying the Front-end

Step 1: Preparing the front-end for deployment

1. We are going to connect to the hosted back-end, therefore we need to modify the API baseUrl to match the that of the back-end API url. (We will be setting this url for backend in the deployment steps). Therefore update it as follows.

```
VITE_BACKEND_URL=https://aidf-5-back-end-manupa.onrender.com
```

Note that the your name has been included as part of the url to identify it as yours. Use your own name.

1. We will be deploying the front-end on Netlify as a Single Page Application. But when we visit different routes in the app Netlify will try to find separate HTML files for each

route. Therefore we need to tell Netlify to redirect all the requests to the `index.html` file. In order to do that we should add a file named `_redirects` into the Vite public folder. You can refer to the Session 18 codebase to see how we have done that.

```
/* /index.html 200
```

1. Make sure the build command has been included correctly in the `package.json` file as follows.

```
"scripts": {  
  "dev": "vite",  
  "build": "vite build",  
  "lint": "eslint .",  
  "preview": "vite preview"  
},
```

1. Also make sure you create 2 separate repositories for your front-end and the back-end and in each repository is pushed to Github and the final code is available on the main branch of the repository.

Step 2: Deploying the FE to Netlify

1. Go to [Netlify's website](#) and **sign up** or **log in** using GitHub.
1. Once logged in, navigate to **"Sites"** and click **"Add new site" → "Import an existing project"**.
1. Then select Github, you will be asked to authorize Netlify via Github.
1. Once that's done, Netlify will show your repositories like a list, select the Repository with the front-end code.
1. Then on the next step setup the build settings as follows.

1. Provide the site name as follows.

Make sure you replace the last part with your own name. Here I have added my own name at the end.

aidf-5-front-end-manupa

1. Then comes the time to setup the front-end environment variables.

Click on `Add environment variables` button and put the env variables in our local `.env` file into it.

1. Click on Deploy. and then you will get the following view

Now you will be able to see your front-end live on the url shown in green. But data won't be there as we didn't deploy the back-end yet.

Check if the FE URL is correctly generated in the format <https://fed-2-front-end-manupa.netlify.app>

Deploying the Back-end

Step 1: Preparing the back-end for deployment

1. Make sure the `package.json` file contain the correct CLI commads to build and start the back-end.

Here the commands have been setup for `npm run build` and `npm run start` commands needed to deploy the back-end.

```
"scripts": {  
  "dev": "nodemon src/index.ts",  
  "seed": "ts-node src/seed.ts",  
  "build": "npm install && tsc",  
  "start": "node ./dist/index.js"  
},
```

1. The `tsconfig` should be as follows

```
{
  "compilerOptions": {
    "sourceMap": true,
    "outDir": "./dist",
    "strict": true,
    "lib": ["esnext"],
    "esModuleInterop": true,
    "skipLibCheck": true
  },
  "ts-node": {
    "files": true
  }
}
```

1. When we deploy the back-end on Render they will inject a port internally to the environment variables to listen. So adjust the `index.ts` file as follows.

1. Since we are using Clerk as our Auth provider and Typescript in our back-end, we need to add Clerk related type information to the back-end. We do that by creating a `globals.d.ts` file inside the `src` folder as follows.

```
/// <reference types="@clerk/express/env" />

export {};

// Create a type for the roles
export type Role = "admin";

declare global {
  interface CustomJwtSessionClaims {
    metadata: {
      role?: Role;
    };
  }
}
```

Refer to the main branch codebase to and see how to set this up.

1. Since requests to this back-end will be coming from our new front-end URL we need to allow that URL in the CORS middleware. So set the relevant ENV variable as follows

```
FRONTEND_URL=https://aidf-5-front-end-manupa.netlify.app
```

Step 2: Deploying the Back-end on Render

1. Go to [Render](#) and **sign up** or **log in** using GitHub.

1. Click "+ **Add New**" and select "**Web Service**".

1. Then select the Github repository where you have included your back-end. and Click
Connect

1. Then on the deploy settings page provide the name for the back-end that we discussed earlier.

fed-2-back-end-manupa

1. Then configure the delpoy settings as follows

- Important: Make sure the region you have selected is `Singapore(Southeast, Asia)` because this is the closest region to Sri Lanka. Otherwise our requests would take a significant amount of time resolve.

1. Select the instance type as `Free` this is the details of the machine where your node app would be running.

1. Then add the back-end environment variables as we have them on our local env,

1. Then click on deploy to deploy the back-end.
2. Then you will be redirected to the Render Dashboard for your project.

1. Now the back-end should also be deployed at the URL we specified.

Finally, you will be able to observe both the URLs are available on the web and you can refer to mine as follows.

<https://aidf-5-front-end-manupa.netlify.app/>

<https://aidf-5-back-end-manupa.onrender.com>

Note:

- While in the process of setting up the front-end and the back-end deployment there's another thing that Render and Netlify has done for us. That's CI/CD(Continuous Integration and Continuous Deployment).
- You will notice that every time we push a commit to the back-end or the front-end Render and Netlify will redeploy our project with the latest updates. This is CI/CD.