Okay, let's document the database architecture, the recent updates, and considerations for deploying ("uploading") the application.

**Database Architecture and Schema Documentation**

This document outlines the PostgreSQL database structure designed for the Booking Application MVP.

**1. Overview**

The database schema is designed to manage services offered, staff members providing those services (including their working hours and external calendar integration), customers making bookings, and the bookings themselves. It employs relational principles to ensure data integrity and consistency.

**2. Entity-Relationship Diagram (Conceptual)**

+-----------+ +-----------------+ +----------+

| Customers |-------| Bookings |-------| Services |

| (1) | (0..\*)| (1) (1)| | (1) |

+-----------+ +-------+---------+ +----------+

|

| (1)

|

+-------+---------+ +---------------------+

| Staff |-------| Staff Working Hours |

| (1) | (0..\*)| (1) |

+-------+---------+ +---------------------+

|

| (0..\*)

|

+-------+---------+

| Staff Services | (Many-to-Many Join Table)

| (Linking Table) |

+-----------------+

* A Customer can have zero or many Bookings.
* A Booking belongs to exactly one Customer, one Service, and one Staff.
* A Service can be part of many Bookings and assigned to many Staff members.
* A Staff member can be assigned many Services, have many Bookings, and have specific Staff Working Hours.
* Staff Services is a join table handling the many-to-many relationship between Staff and Services.
* Staff Working Hours define the regular weekly availability for each Staff member.

**3. Table Definitions and Constraints**

*(Based on schema details provided earlier)*

* **customers Table**
  + **Purpose:** Stores information about clients making bookings. Allows for CRM features.
  + **Columns:**
    - customer\_id: SERIAL PRIMARY KEY - Unique identifier for the customer.
    - name: VARCHAR(255) NOT NULL - Customer's full name.
    - email: VARCHAR(255) UNIQUE - Customer's unique email address. (Constraint: customers\_email\_key)
    - phone: VARCHAR(50) NULL - Customer's phone number (optional).
    - created\_at: TIMESTAMPTZ NOT NULL DEFAULT CURRENT\_TIMESTAMP - Auto-set creation timestamp.
    - updated\_at: TIMESTAMPTZ NOT NULL DEFAULT CURRENT\_TIMESTAMP - Auto-updated timestamp (via trigger).
    - notes: TEXT NULL - Optional notes.
  + **Indexes:** idx\_customers\_email (on email), idx\_customers\_name (optional).
  + **Triggers:** set\_timestamp\_customers automatically updates updated\_at on row changes.
* **services Table**
  + **Purpose:** Defines the services offered by the business.
  + **Columns:**
    - service\_id: SERIAL PRIMARY KEY - Unique identifier. (Constraint: services\_pkey)
    - name: VARCHAR(255) NOT NULL - Name of the service.
    - duration\_minutes: INTEGER NOT NULL - Duration of the service in minutes. (Constraint: services\_duration\_minutes\_check ensures > 0).
    - price: NUMERIC(10, 2) NULL - Price of the service (optional).
    - description: TEXT NULL - Optional description.
    - is\_active: BOOLEAN NOT NULL DEFAULT TRUE - Whether the service is available for booking.
    - created\_at, updated\_at: TIMESTAMPTZ (with defaults/triggers - assumed).
  + **Constraints:** CHECK (duration\_minutes > 0).
* **staff Table**
  + **Purpose:** Stores information about staff members, including credentials and integration details.
  + **Columns:**
    - staff\_id: SERIAL PRIMARY KEY - Unique identifier. (Constraint: staff\_pkey)
    - name: VARCHAR(255) NOT NULL - Staff member's name.
    - email: VARCHAR(255) UNIQUE NULL - Login email (optional for 'staff' role, unique if provided). (Constraint: staff\_email\_key)
    - phone\_number: VARCHAR(50) NULL - Optional phone number.
    - role: VARCHAR(50) NOT NULL DEFAULT 'staff' - Role ('admin', 'manager', 'staff'). (Constraint: staff\_role\_check)
    - password\_hash: VARCHAR(255) NULL - Hashed password (only for 'admin'/'manager').
    - password\_reset\_token: VARCHAR(255) NULL - Hashed password reset token (HMAC).
    - password\_reset\_expires: TIMESTAMPTZ NULL - Expiry time for the reset token.
    - is\_active: BOOLEAN NOT NULL DEFAULT TRUE - Whether the staff member can be booked or log in.
    - google\_refresh\_token: TEXT NULL - Encrypted Google OAuth refresh token.
    - google\_calendar\_id: VARCHAR(255) NULL - Primary Google Calendar ID used for availability/booking sync.
    - created\_at, updated\_at: TIMESTAMPTZ (with defaults/triggers - assumed).
  + **Constraints:** CHECK (role IN ('admin', 'manager', 'staff')), UNIQUE (email).
* **bookings Table**
  + **Purpose:** Records individual appointment bookings.
  + **Columns:**
    - booking\_id: SERIAL PRIMARY KEY - Unique identifier. (Constraint: bookings\_pkey)
    - customer\_id: INTEGER NULL REFERENCES customers(customer\_id) ON DELETE RESTRICT - Foreign key to the customers table. (Constraint: fk\_booking\_customer) - **(UPDATED)**
    - service\_id: INTEGER NOT NULL REFERENCES services(service\_id) ON DELETE RESTRICT - Foreign key to services. (Constraint: fk\_booking\_service)
    - staff\_id: INTEGER NOT NULL REFERENCES staff(staff\_id) ON DELETE RESTRICT - Foreign key to staff. (Constraint: fk\_booking\_staff)
    - client\_name: VARCHAR(255) NOT NULL - Client's name at time of booking (denormalized).
    - client\_email: VARCHAR(255) NOT NULL - Client's email at time of booking (denormalized).
    - client\_phone: VARCHAR(50) NULL - Client's phone at time of booking (denormalized).
    - notes: TEXT NULL - Optional booking notes from client or admin.
    - booking\_start\_time: TIMESTAMPTZ NOT NULL - Start time/date of the booking in UTC.
    - booking\_end\_time: TIMESTAMPTZ NOT NULL - End time/date of the booking in UTC.
    - status: VARCHAR(50) NOT NULL DEFAULT 'confirmed' - Booking status ('scheduled', 'confirmed', 'completed', 'cancelled', 'no-show').
    - booking\_timezone: VARCHAR(100) NULL - IANA timezone identifier of the *client* when the booking was made (for reference/confirmation emails).
    - google\_event\_id: VARCHAR(255) NULL - ID of the corresponding event in the staff member's Google Calendar (if synced).
    - created\_at, updated\_at: TIMESTAMPTZ (with defaults/triggers - assumed).
  + **Indexes:** idx\_bookings\_customer\_id, (Consider indexes on staff\_id, service\_id, booking\_start\_time, status).
  + **Constraints:** Foreign Keys with ON DELETE RESTRICT.
* **staff\_services Table**
  + **Purpose:** Join table for the many-to-many relationship between staff and services.
  + **Columns:**
    - staff\_id: INTEGER NOT NULL REFERENCES staff(staff\_id) ON DELETE CASCADE - Foreign key to staff. Cascade delete makes sense here - if staff is deleted, their service assignments are irrelevant.
    - service\_id: INTEGER NOT NULL REFERENCES services(service\_id) ON DELETE CASCADE - Foreign key to services. Cascade delete here is debatable. If a service is deleted, should staff assignments just disappear? RESTRICT might be safer, forcing manual reassignment, matching the bookings table constraint. Currently assumed CASCADE based on common practice for join tables, but consider changing to RESTRICT if services are critical reference points.
    - PRIMARY KEY (staff\_id, service\_id) - Composite primary key ensures unique assignments.
  + **Constraints:** Foreign Keys, Primary Key.
* **staff\_working\_hours Table**
  + **Purpose:** Defines default weekly availability for staff members.
  + **Columns:**
    - working\_hours\_id: SERIAL PRIMARY KEY - Unique identifier. (Constraint: staff\_working\_hours\_pkey)
    - staff\_id: INTEGER NOT NULL REFERENCES staff(staff\_id) ON DELETE CASCADE - Foreign key to staff. (Constraint: fk\_staff\_working\_hours\_staff)
    - day\_of\_week: INTEGER NOT NULL - ISO day number (1=Mon, 7=Sun). (Constraint: check\_day\_of\_week)
    - start\_time: TIME NOT NULL - Start time (HH:MM:SS).
    - end\_time: TIME NOT NULL - End time (HH:MM:SS). (Constraint: check\_end\_time\_after\_start\_time)
  + **Constraints:** CHECK (1 <= day\_of\_week <= 7), CHECK (end\_time > start\_time), UNIQUE (staff\_id, day\_of\_week) (Constraint: unique\_staff\_day), Foreign Key.

**4. Key Updates and Design Decisions**

* **Customer Table (customers):**
  + **Added:** A dedicated customers table was introduced (Step 10).
  + **Purpose:** To store unique customer information (name, email, phone, notes) separate from individual bookings. This normalizes customer data and facilitates future CRM features (tracking booking history, customer preferences, etc.).
  + **Link:** The bookings table was updated (Step 11) with a customer\_id foreign key column (ON DELETE RESTRICT) to link each booking to a customer record.
  + **Booking Process:** The booking creation API (POST /api/bookings) was updated (Step 12) to implement a "find or create" pattern: it searches for an existing customer by email; if found, it uses their ID; if not found, it creates a new customer record and uses the new ID for the booking.
* **Denormalized Client Info in bookings:** The client\_name, client\_email, and client\_phone columns were kept in the bookings table even after adding customer\_id.
  + **Rationale:** This preserves the contact information *as provided at the time of booking*. Customer details in the customers table might be updated later, but the booking record retains the original snapshot. This can be useful for historical accuracy or if a booking was made before a customer profile was fully established/updated. It adds redundancy but can simplify displaying booking details without always needing a join.
* **Foreign Key Constraints (ON DELETE RESTRICT):** The choice of RESTRICT for bookings referencing services and staff provides strong data integrity. It prevents accidental deletion of staff/services that are still linked to bookings. The application handles this by using soft deletes (is\_active = false) for staff and providing user feedback if a service deletion fails due to existing bookings.
* **Timestamps:** Consistent use of TIMESTAMPTZ for storing date/time information is recommended for handling timezones correctly. Automatic created\_at and updated\_at (via triggers) help track data changes.

**5. Deployment ("Uploading") Considerations**

Deploying this Next.js application involves several steps:

1. **Database Setup:**
   * You need a **production PostgreSQL database** server (e.g., AWS RDS, Google Cloud SQL, Supabase, Neon, Railway, Aiven, or self-hosted).
   * Create the booking\_app\_db database and the application user (postgres or a dedicated user) with the chosen password on the production server.
   * **Run all the CREATE TABLE, ALTER TABLE, CREATE FUNCTION, CREATE TRIGGER SQL scripts** (from our steps and potentially from your original setup for tables like staff\_services) against the production database to set up the schema. Tools like psql or database migration libraries (e.g., node-pg-migrate, Prisma Migrate) can automate this. **Migration tools are highly recommended for managing schema changes over time.**
2. **Environment Variables:**
   * You **MUST NOT** commit your .env.local file to Git.
   * Your hosting provider (Vercel, Netlify, AWS, Google Cloud, Railway, etc.) will have a mechanism to securely set environment variables for the deployed application.
   * Set **ALL** the variables from your .env.local (DB connection, JWT secret, API keys, encryption keys, BUSINESS\_TIMEZONE, APP\_BASE\_URL pointing to your production domain, etc.) in the production environment using your provider's interface. **Use strong, unique secrets for production.**
   * Crucially, update GOOGLE\_REDIRECT\_URI to match the production callback URL (e.g., https://yourdomain.com/api/auth/google/callback) and ensure this URL is added as an authorized redirect URI in your Google Cloud Console credentials.
3. **Build Step:**
   * Before deploying, run the Next.js production build command:
   * npm run build
   * # or
   * yarn build

IGNORE\_WHEN\_COPYING\_START

content\_copy download

Use code [with caution](https://support.google.com/legal/answer/13505487). Bash

IGNORE\_WHEN\_COPYING\_END

* + This creates an optimized production version of your application in the .next folder.

1. **Hosting Provider:**
   * Choose a hosting provider suitable for Next.js applications (Vercel is the creator and often the simplest, but many others work well).
   * Connect your Git repository (GitHub, GitLab, Bitbucket) to the provider.
   * Configure the build command (npm run build or yarn build) and the output directory (.next).
   * Configure the environment variables as mentioned above.
   * Deploy the application. The provider will typically pull your code, run the build, and host the static assets and serverless functions (API routes, etc.).
2. **Domain Name:** Configure your custom domain name to point to the hosting provider's service.
3. **Security:** Ensure HTTPS is enabled (most providers handle this automatically). Review security headers, CORS settings (restrict origins in production), and other security best practices.

This document provides a snapshot of the database architecture and key updates. Remember to keep documentation updated as the application evolves.