CommunityTap – Technical Documentation

Version: 1.0
Date: November 9th, 2025
Author: Lilith Froude
Project Type: Full-Stack Web Application

Executive Summary

The Concept:
CommunityTap is a hyper-local crowdfunding platform that connects beer enthusiasts with neighborhood pubs through $1 community-funded events. Users contribute $1 to a "Community Tap" in their city. Once 100 people contribute ($100 total), the platform takes a 20% fee ($20) and sends the remaining $80 to participating pubs to host a pre-paid beer party where contributors redeem their $1 for a beer.

Business Model & Economics:

- Users contribute $1 per city tap
- Goal: 100 contributors = $100 per tap
- Platform fee: 20% ($20 profit)
- Pub receives: $80 for the event
- **Pub cost structure:** Assuming $2 cost per beer × 100 beers = $200 total cost
- **Net pub investment:** $120 ($200 cost - $80 platform payment)

The Value Proposition for Pubs:
The business model assumes pubs view this $120 investment as a customer acquisition cost, not a loss. Pubs gain:

- **Guaranteed foot traffic:** 100 new or returning customers in one night
- **Upsell revenue:** Second/third drinks, food orders, premium cocktails
- **Social proof:** Friends brought by contributors (multiplier effect)
- **Long-term customers:** Converting one-time visitors into regulars
- **Marketing ROI:** $1.20 per customer acquisition cost is significantly lower than traditional advertising

The core assumption: Pubs prioritize new traffic and relationship-building over immediate profit on the first beer. The real money is made on everything after that first $1 beer. The 100-person cap keeps the pub's financial exposure manageable at just $120 while still generating meaningful foot traffic.

Demo Videos:

- Full Walkthrough
- Alternative Demo

This technical documentation represents a 48-hour sprint to validate the CommunityTap concept and achieve MVP-ready state. The project demonstrates the feasibility of rapidly building a full-stack web application using modern AI-assisted development tools, with a total investment of approximately $150 (including AI credits, domain registration, and essential services).

Project Goals:

- Validate the complete technical workflow from concept to deployable MVP
- Test the viability of AI-assisted development for rapid prototyping

- Establish a functional proof-of-concept for the CommunityTap business model
- Document the full technical architecture for future scaling

Key Constraints:

- **Timeline:** 48 hours
- **Budget:** ~$150 (AI tools, domain, hosting, API credits)
- **Scope:** End-to-end functionality including user flows, payment processing, and pub onboarding

This document serves as both a technical reference and a case study in rapid MVP development.

Table of Contents

System Architecture

Technology Stack

Frontend:

- **Framework:** React 18.3.1 with TypeScript
- **Build Tool:** Vite
- **Styling:** Tailwind CSS with shadcn/ui components
- **State Management:** TanStack React Query for server state
- **Routing:** React Router DOM v6
- **Forms:** React Hook Form with Zod validation

Backend:

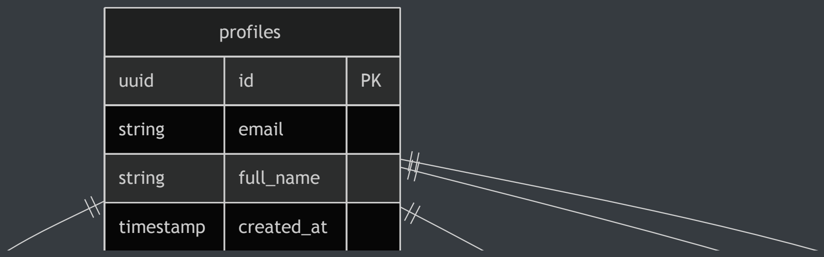
- **Database:** PostgreSQL (via Supabase)
- **Serverless Functions:** Supabase Edge Functions (Deno runtime)
- **Authentication:** Supabase Auth
- **Storage:** Supabase Storage (for future file uploads)

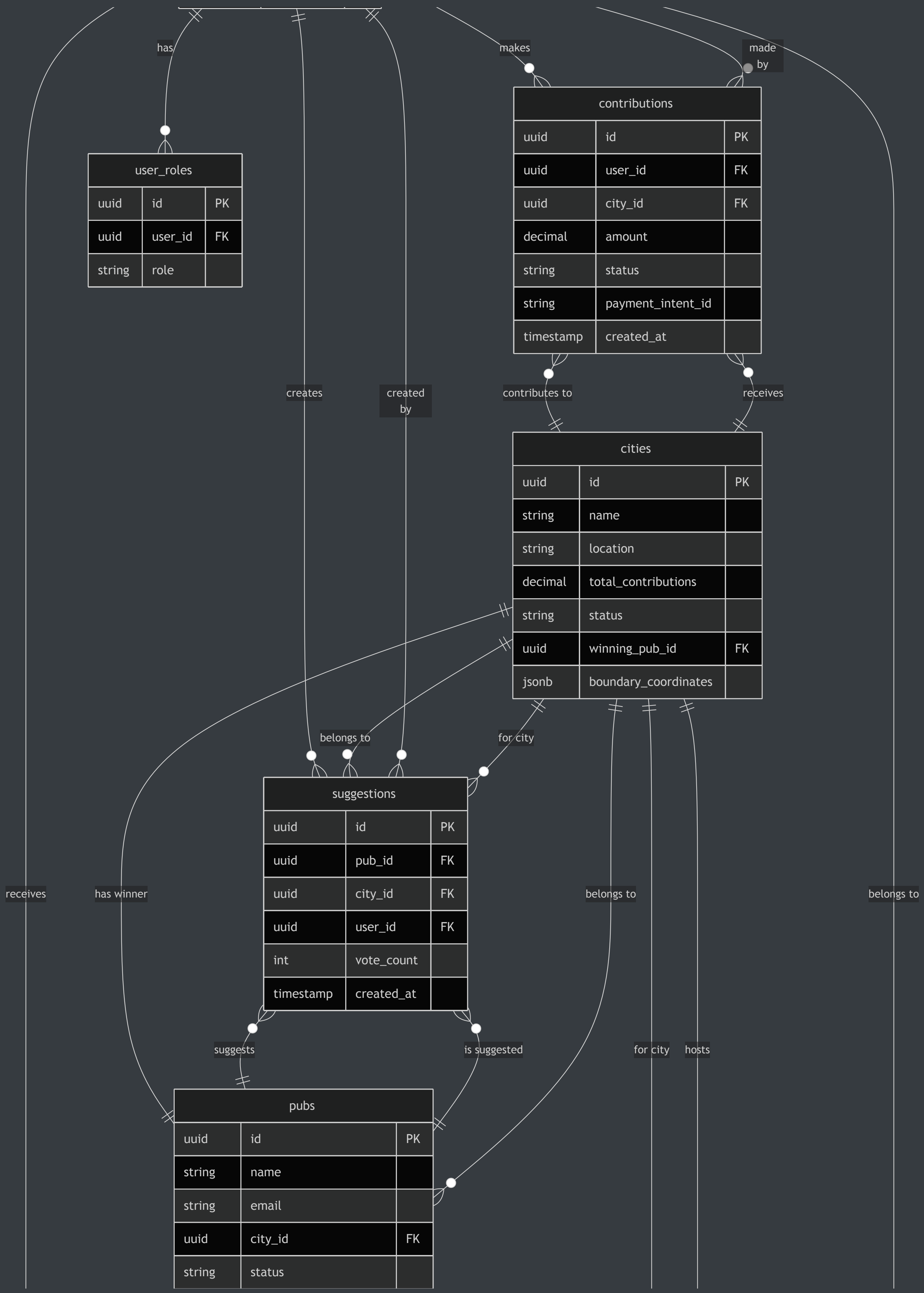
Third-Party Integrations:

- **Payments:** Stripe (Standard + Connect for marketplace)
- **Email:** Resend API
- **Maps:** Mapbox GL JS + Google Maps Places API

Database Schema & ERD

Entity Relationship Diagram





user_roles		
uuid	id	PK
uuid	user_id	FK
string	role	

contributions		
uuid	id	PK
uuid	user_id	FK
uuid	city_id	FK
decimal	amount	
string	status	
string	payment_intent_id	
timestamp	created_at	

cities		
uuid	id	PK
string	name	
string	location	
decimal	total_contributions	
string	status	
uuid	winning_pub_id	FK
jsonb	boundary_coordinates	

suggestions		
uuid	id	PK
uuid	pub_id	FK
uuid	city_id	FK
uuid	user_id	FK
int	vote_count	
timestamp	created_at	

pubs		
uuid	id	PK
string	name	
string	email	
uuid	city_id	FK
string	status	

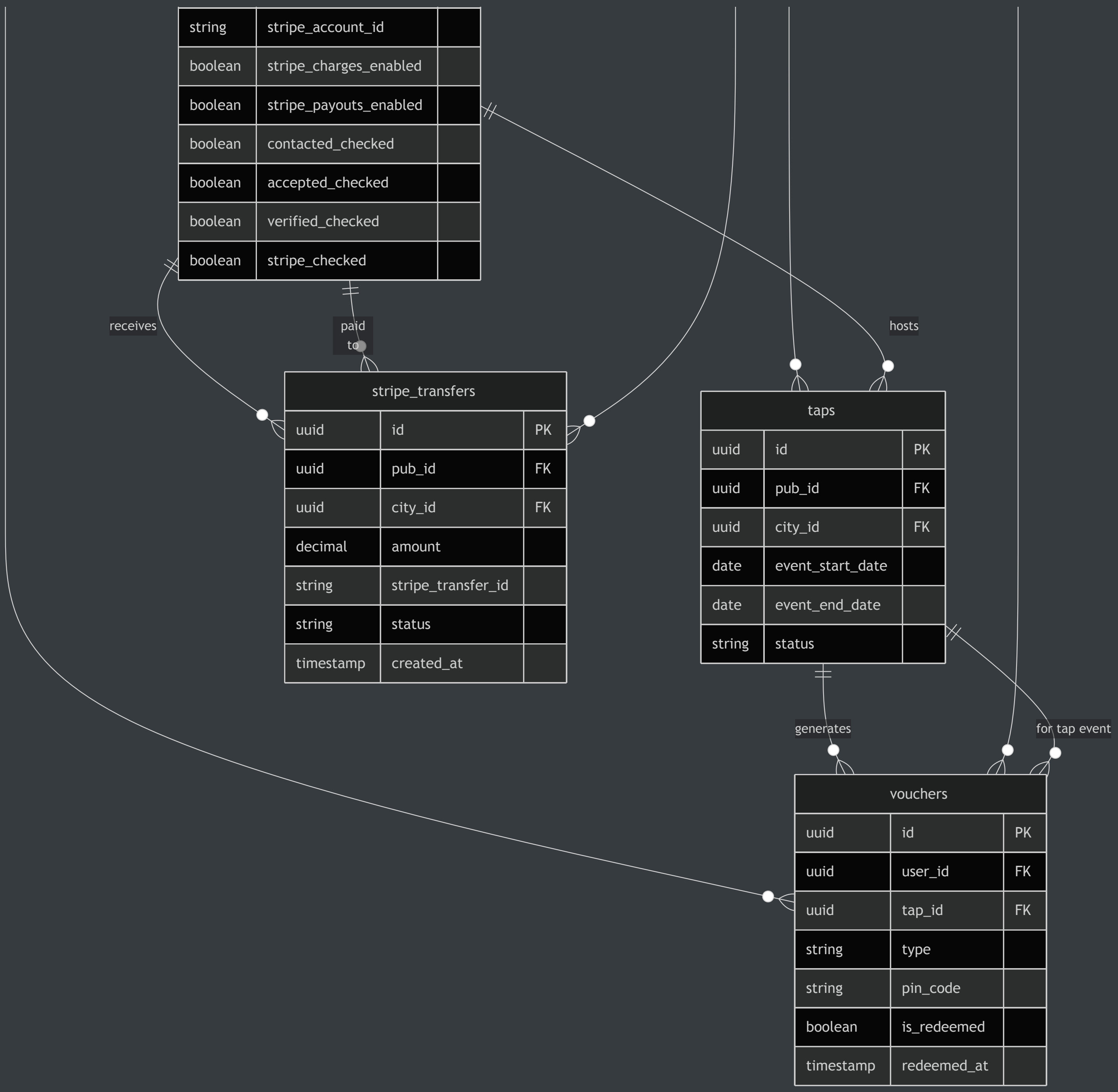


Table Descriptions

profiles

- Extends Supabase auth.users
- Stores additional user information
- Linked to contributions, suggestions, and vouchers
- **Key Columns:** id (UUID), email, full_name

cities

- Represents geographic areas where competitions take place
- Tracks total contributions and winning pub
- Status: "Accepting Suggestions" → "Voting Open" → "Closed" → "Live"
- **Key Columns:** name, location, total_contributions, status, winning_pub_id, boundary_coordinates

pubs

- Represents bars/restaurants participating in the platform
- Stores Stripe Connect account information
- Status: "Suggested" → "Invited" → "Onboarding" → "Active"
- **Key Columns:** name, email, stripe_account_id, stripe_charges_enabled, stripe_payouts_enabled

contributions

- Tracks $1 contributions from users
- Links to Stripe payment_intent_id
- Status: "pending" → "completed" or "failed"
- **Key Columns:** user_id, city_id, amount, status, payment_intent_id

suggestions

- Represents pub suggestions within a city
- Users vote by suggesting the same pub
- **Key Columns:** pub_id, city_id, user_id, vote_count

taps

- Represents a week-long event at the winning pub

- Defines redemption period for vouchers
- **Key Columns:** pub_id, city_id, event_start_date, event_end_date, status

vouchers

- Free drink vouchers given to contributors
- Type: "Pledge" (for contributors) or "Bonus" (future)
- Generated with unique 6-digit PIN codes
- **Key Columns:** user_id, tap_id, pin_code, is_redeemed, redeemed_at

user_roles

- Role-based access control
- Roles: "giver" (default), "pub", "admin"
- **Key Columns:** user_id, role

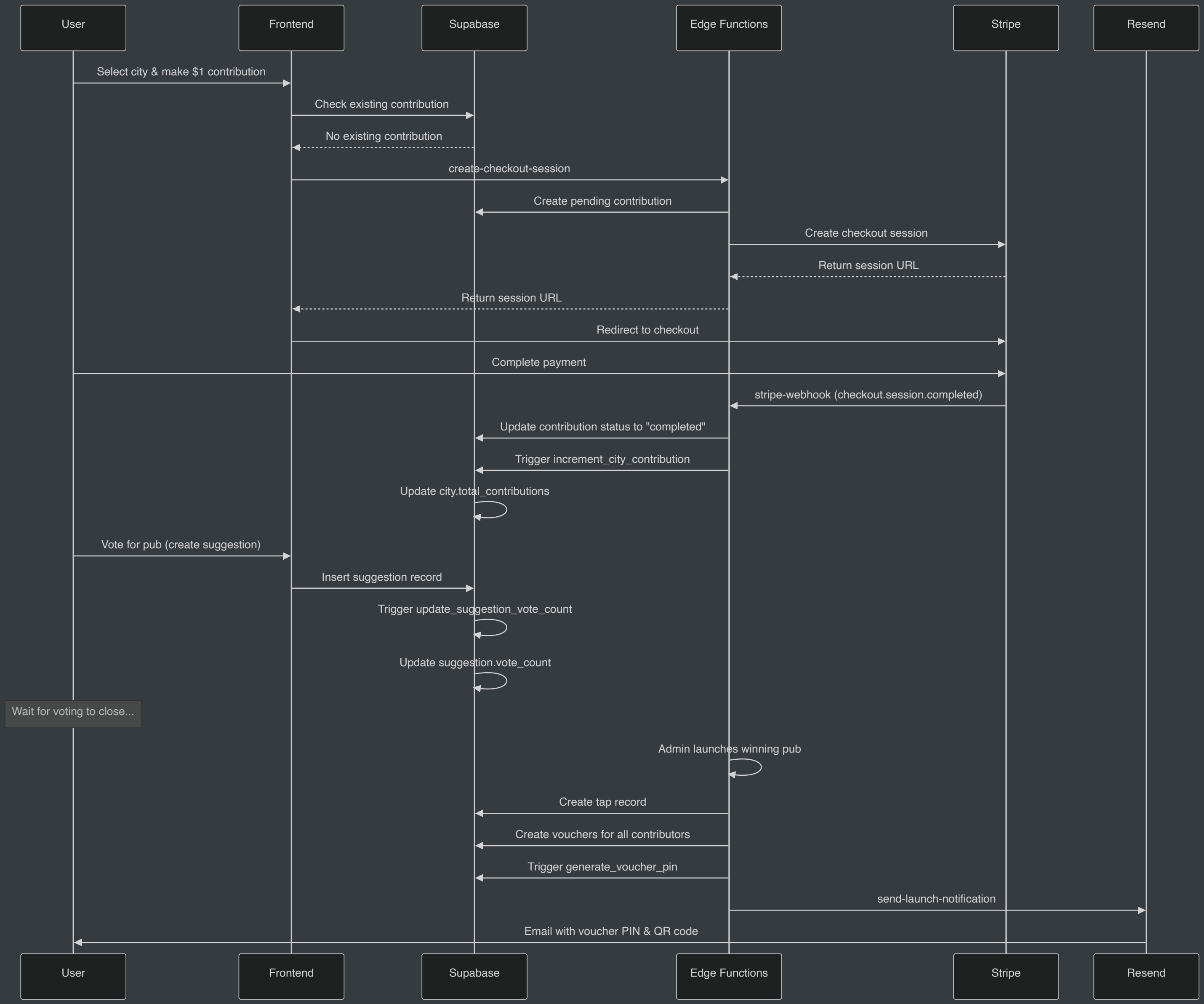
stripe_transfers

- Records payouts to winning pubs
- 80% of total contributions after Stripe fees
- **Key Columns:** pub_id, city_id, amount, stripe_transfer_id, status

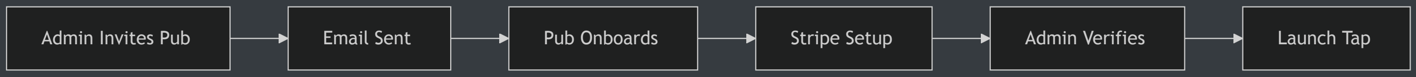
---

Core User Flows

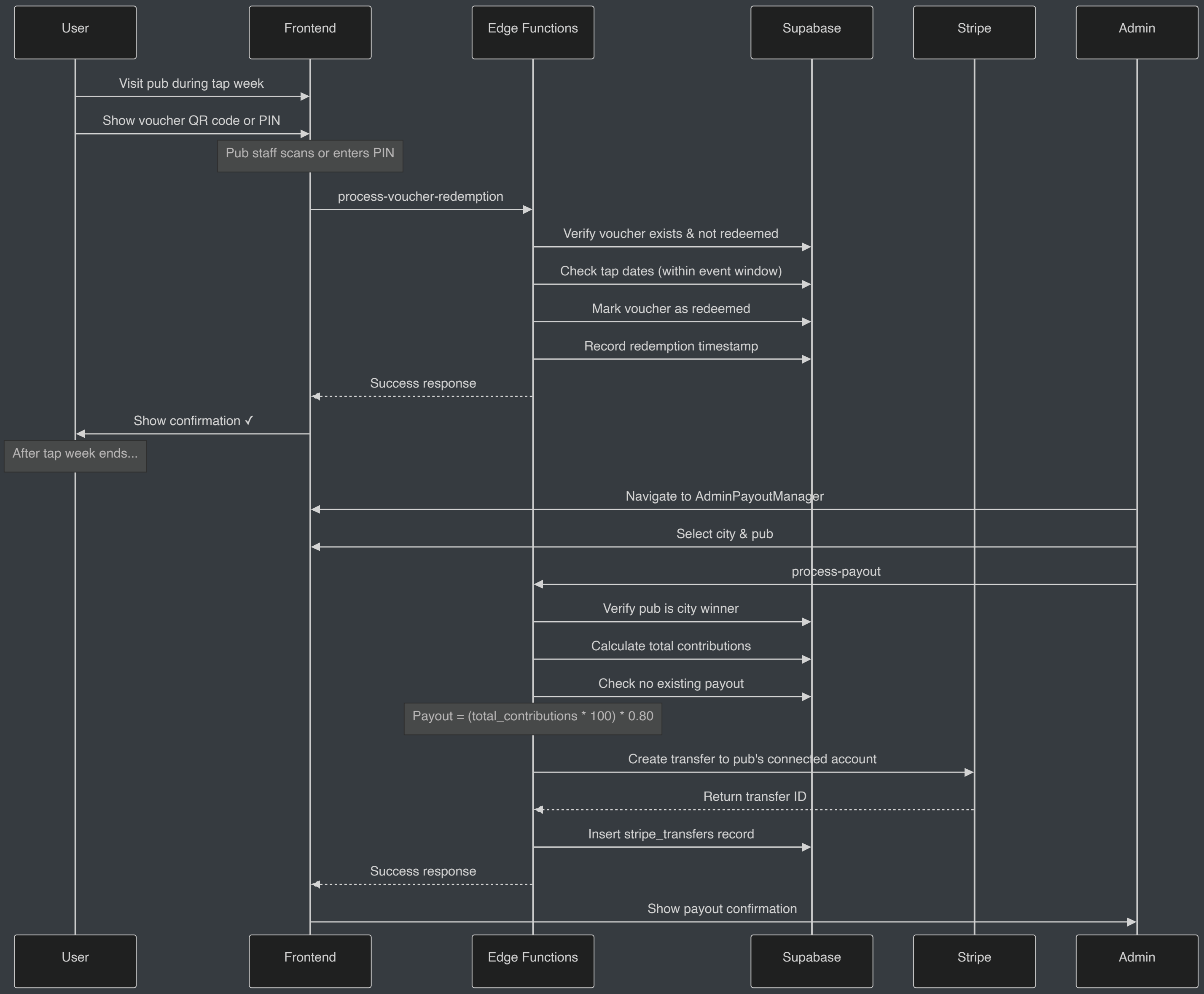
1. Contributor Journey


User
Frontend
Supabase
Edge Functions
Stripe
Resend
Select city & make $1 contribution
Check existing contribution
No existing contribution
create-checkout-session
Create pending contribution
Create checkout session
Return session URL
Return session URL
Redirect to checkout
Complete payment
stripe-webhook (checkout.session.completed)
Update contribution status to "completed"
Trigger increment_city_contribution
Update city.total_contributions
Vote for pub (create suggestion)
Insert suggestion record
Trigger update_suggestion_vote_count
Update suggestion.vote_count
Wait for voting to close...
Admin launches winning pub
Create tap record
Create vouchers for all contributors
Trigger generate_voucher_pin
send-launch-notification
Email with voucher PIN & QR code

2. Admin Onboarding Flow



3. Voucher Redemption & Payout Flow


User
Frontend
Edge Functions
Supabase
Stripe
Admin
Visit pub during tap week
Show voucher QR code or PIN
Pub staff scans or enters PIN
process-voucher-redemption
Verify voucher exists & not redeemed
Check tap dates (within event window)
Mark voucher as redeemed
Record redemption timestamp
Success response
Show confirmation ✓
After tap week ends...
Navigate to AdminPayoutManager
Select city & pub
process-payout
Verify pub is city winner
Calculate total contributions
Check no existing payout
Payout = (total_contributions * 100) * 0.80
Create transfer to pub's connected account
Return transfer ID
Insert stripe_transfers record
Success response
Show payout confirmation

API Integrations & Edge Functions

Stripe Integration

Purpose: Handle payments from contributors and payouts to pubs

`create-checkout-session`

File: supabase/functions/create-checkout-session/index.ts

Flow:

1. Authenticates user via Supabase Auth
2. Validates cityId and cityName from request
3. Fetches city and active tap data
4. Checks for existing completed contributions
5. Deletes any pending/failed contributions
6. Creates new pending contribution in database
7. Creates Stripe Checkout Session:

   ```
   {
     payment_method_types: ['card'],
     line_items: [{
       price_data: {
         currency: 'usd',
         product_data: { name: `Contribute to ${cityName}` },
         unit_amount: 100
       },
       quantity: 1
     }],
     mode: 'payment',
     success_url: `${origin}/?contribution=success`,
     cancel_url: `${origin}/?contribution=cancelled`,
     metadata: {
       contributionId,
       userId,
       cityId
     }
   }
   ```

8. Returns session URL and contribution ID

Key Query:

```
SELECT * FROM contributions
WHERE user_id = $1
  AND city_id = $2
  AND status = 'completed'
```

stripe-webhook

File: supabase/functions/stripe-webhook/index.ts

Handles Events:

- checkout.session.completed: Updates contribution status to "completed"
- account.updated: Updates pub's Stripe capabilities

Critical Fix - Account Updated Handler:

```
case "account.updated": {
  const account = event.data.object as Stripe.Account;

  // Update pub's Stripe status
  const { error: updateError } = await supabaseAdmin
    .from("pubs")
    .update({
      stripe_charges_enabled: account.charges_enabled || false,
      stripe_payouts_enabled: account.payouts_enabled || false,
    })
    .eq("stripe_account_id", account.id);

  console.log(`Updated pub Stripe status - charges:
${account.charges_enabled}, payouts: ${account.payouts_enabled}`);
  break;
}
```

Webhook Signature Verification:

```
const signature = req.headers.get("stripe-signature");
const event = await stripe.webhooks.constructEventAsync(
  body,
  signature!,
  webhookSecret
);
```

Key Queries:

```
-- Update contribution
UPDATE contributions
SET status = 'completed',
    payment_intent_id = $1
WHERE id = $2;

-- Update pub Stripe status
UPDATE pubs
SET stripe_charges_enabled = $1,
    stripe_payouts_enabled = $2
WHERE stripe_account_id = $3;
```

create-connect-account

File: supabase/functions/create-connect-account/index.ts

Flow:

1. Authenticates user
2. Validates pubId ownership

3. Checks if Stripe account already exists
4. Creates Stripe Connect account:

```
const account = await stripe.accounts.create({
  type: "standard",
  country: "US",
  capabilities: {
    card_payments: { requested: true },
    transfers: { requested: true },
  },
});
```

5. Updates pub record with stripe_account_id

Key Query:

```
UPDATE pubs
SET stripe_account_id = $1,
    status = 'Onboarding'
WHERE id = $2
  AND user_id = $3;
```

create-connect-account-link

File: supabase/functions/create-connect-account-link/index.ts

Critical Fix: Changed line 69 from:

```
type: type === 'dashboard' ? 'account_onboarding' : 'account_onboarding'
```

To:

```
type: type === 'dashboard' ? 'account_update' : 'account_onboarding'
```

This ensures:

- "Complete Stripe Setup" button → account_onboarding link
- "Open Stripe Dashboard" button → account_update link

Flow:

1. Verifies user owns the pub

   Creates Stripe account link:

   ```
   const response = await
   fetch("https://api.stripe.com/v1/account_links", {
     method: "POST",
     headers: {
       "Authorization": `Bearer ${stripeSecretKey}`,
       "Content-Type": "application/x-www-form-urlencoded",
     },
     body: new URLSearchParams({
       account: accountId,
       refresh_url: refreshUrl,
       return_url: returnUrl,
       type: type === 'dashboard' ? 'account_update' :
   'account_onboarding',
     }),
   });
   ```

process-voucher-redemption

File: supabase/functions/process-voucher-redemption/index.ts

Flow:

1. Authenticates pub user
2. Validates pin_code format
3. Finds voucher by PIN
4. Verifies tap belongs to authenticated pub
5. Checks tap date range (within event window)
6. Marks voucher as redeemed
7. Creates payment record (80% to pub)

Payout Formula:

```
const payoutAmount = voucherAmount * 0.80; // 80% to pub, 20% platform fee
```

Key Queries:

```sql
-- Find voucher
SELECT v.*, t.pub_id, t.event_start_date, t.event_end_date
FROM vouchers v
JOIN taps t ON v.tap_id = t.id
WHERE v.pin_code = $1
  AND v.is_redeemed = false;

-- Mark as redeemed
UPDATE vouchers
SET is_redeemed = true,
    redeemed_at = NOW()
WHERE id = $1;
```

process-payout

File: supabase/functions/process-payout/index.ts

Flow:

1. Verifies admin authorization
2. Fetches pub and city details
3. Verifies Stripe onboarding complete
4. Confirms pub is city winner
5. Calculates payout (80% of contributions)
6. Checks for existing payouts
7. Creates Stripe transfer
8. Records transfer in database

Payout Calculation:

```
const payoutAmount = Math.floor(totalContributions * 100 * 0.80); // in
cents
```

Stripe Transfer:

```
const transfer = await stripe.transfers.create({
  amount: payoutAmount,
  currency: "usd",
  destination: pub.stripe_account_id,
  description: `Payout for ${city.name} - CommunityTap`,
});
```

Key Queries:

```
-- Get pub details
SELECT * FROM pubs
WHERE id = $1
  AND stripe_charges_enabled = true
  AND stripe_payouts_enabled = true;

-- Verify winner
SELECT * FROM cities
WHERE id = $1
  AND winning_pub_id = $2;

-- Check existing payout
SELECT * FROM stripe_transfers
WHERE pub_id = $1
  AND city_id = $2;

-- Record transfer
INSERT INTO stripe_transfers (
  pub_id, city_id, amount,
  stripe_transfer_id, status
) VALUES ($1, $2, $3, $4, 'completed');
```

Resend Email API

Purpose: Send transactional emails (invitations, notifications)

send-launch-notification

File: supabase/functions/send-launch-notification/index.ts

Flow:

1. Authenticates admin user
2. Fetches tap, pub, and city details
3. Retrieves all vouchers for the tap
4. Fetches contributor profiles
5. Sends batch emails using Resend API

Email Batching:

```
  const emailPromises = vouchers.map(async (voucher) => {
    const profile = contributorProfiles[voucher.user_id];

    return fetch("https://api.resend.com/emails", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
        Authorization: `Bearer ${resendApiKey}`,
      },
      body: JSON.stringify({
        from: "CommunityTap <notifications@communitytap.app>",
        to: [profile.email],
        subject: `Your Free Drink is Ready at ${pub.name}!`,
        html: generateEmailTemplate(voucher, pub, tap),
      }),
    });
  });

  await Promise.all(emailPromises);
```

Email Template Includes:

- Pub name and address
- Event dates (tap.event_start_date → tap.event_end_date)
- Voucher PIN code
- QR code (base64 encoded)
- Redemption instructions

Key Queries:

```sql
    -- Get tap details
    SELECT t.*, p.name as pub_name, c.name as city_name
    FROM taps t
    JOIN pubs p ON t.pub_id = p.id
    JOIN cities c ON t.city_id = c.id
    WHERE t.id = $1;

    -- Get vouchers
    SELECT * FROM vouchers
    WHERE tap_id = $1;
```

```sql
-- Get contributor profiles
SELECT id, email, full_name
FROM profiles
WHERE id = ANY($1);
```

invite-pub

File: supabase/functions/invite-pub/index.ts

Flow:

1. Verifies admin authorization via has_role('admin')
2. Validates required fields (pubName, pubEmail, cityId)
3. Checks if pub email already exists in auth system
4. Creates pub record with status "Invited"
5. Generates secure invite token
6. Constructs onboarding URL
7. Sends invitation email via Resend

Token Generation:

```typescript
const token = crypto.randomUUID();
const expiresAt = new Date();
expiresAt.setDate(expiresAt.getDate() + 7); // 7-day expiry

await supabaseAdmin.from("pub_invite_tokens").insert({
  token,
  pub_id: newPubId,
  expires_at: expiresAt.toISOString(),
  used: false,
});
```

Key Queries:

```sql
-- Check existing auth user
SELECT id FROM auth.users
WHERE email = $1;

-- Create pub
INSERT INTO pubs (name, email, city_id, status)
VALUES ($1, $2, $3, 'Invited')
RETURNING id;

-- Create invite token
INSERT INTO pub_invite_tokens (token, pub_id, expires_at, used)
VALUES ($1, $2, $3, false);
```

Mapbox Integration

Purpose: Interactive city maps with boundary visualization

get-mapbox-token

File: supabase/functions/get-mapbox-token/index.ts

Why This Exists:

- Prevents exposing Mapbox token in frontend code
- Allows token rotation without redeploying frontend
- Provides centralized access control

Usage in Frontend:

```
// src/components/Map.tsx
const { data: mapboxToken } = useQuery({
  queryKey: ["mapbox-token"],
  queryFn: async () => {
    const { data, error } = await supabase.functions.invoke("get-mapbox-
token");
    if (error) throw error;
    return data.token;
  },
});
```

Google Maps Places API

Purpose: Pub autocomplete and location data

get-place-details

File: supabase/functions/get-place-details/index.ts

Flow:

1. Receives placeId from frontend
2. Calls Google Places API (Place Details)
3. Returns formatted address, coordinates, etc.

Usage in Frontend:

```
// src/components/SuggestPubForm.tsx
const { data: placeDetails } = useQuery({
  queryKey: ["place-details", selectedPlaceId],
  queryFn: async () => {
    const { data, error } = await supabase.functions.invoke("get-place-
details", {
      body: { placeId: selectedPlaceId },
    });
    if (error) throw error;
    return data;
  },
});
```

Authentication & Authorization

Supabase Auth

Provider: Email/Password authentication
Session Storage: localStorage (persistent across tabs/reloads)
JWT Token: Automatically included in Supabase client requests

Sign Up Flow:

```
const { data, error } = await supabase.auth.signUp({
  email: formData.email,
  password: formData.password,
  options: {
    data: {
      full_name: formData.fullName,
    },
  },
});
```

Auto-Confirm Email: Configured in Supabase Auth settings to skip email verification (configurable via configure-auth tool)

Session Management:

```
// Check current session
const { data: { session } } = await supabase.auth.getSession();

// Listen for auth changes
supabase.auth.onAuthStateChange((event, session) => {
  if (event === 'SIGNED_IN') {
    // User logged in
  } else if (event === 'SIGNED_OUT') {
    // User logged out
  }
});
```

Role-Based Access Control (RBAC)

Roles Defined:

- giver: Default role for contributors

- pub: Pub owners (can access PubDashboard)
- admin: Platform administrators (full access)

Database Function: has_role

```sql
CREATE OR REPLACE FUNCTION has_role(role_name text)
RETURNS boolean
LANGUAGE plpgsql
SECURITY DEFINER
AS $$
BEGIN
  RETURN EXISTS (
    SELECT 1
    FROM user_roles
    WHERE user_id = auth.uid()
    AND role = role_name
  );
END;
$$;
```

Usage in Edge Functions:

```javascript
const { data: isAdmin, error } = await supabase.rpc("has_role", {
  role_name: "admin",
});

if (!isAdmin) {
  return new Response(
    JSON.stringify({ error: "Unauthorized - Admin access required" }),
    { status: 403 }
  );
}
```

Database Function: is_pub_owner

```sql
CREATE OR REPLACE FUNCTION is_pub_owner(pub_id_param uuid)
RETURNS boolean
LANGUAGE plpgsql
SECURITY DEFINER
AS $$
BEGIN
  RETURN EXISTS (
    SELECT 1
    FROM pubs
    WHERE id = pub_id_param
    AND user_id = auth.uid()
  );
END;
$$;
```

Frontend Route Protection:

```javascript
// Check if user has admin role
const { data: isAdmin } = await supabase.rpc("has_role", {
  role_name: "admin",
});

if (!isAdmin) {
  navigate("/"); // Redirect to home
}
```

Security Measures

Row-Level Security (RLS)

All tables have RLS enabled. Policies control data access based on authenticated user.

profiles Table

```sql
-- Users can view all profiles
CREATE POLICY "Public profiles are viewable by everyone"
ON profiles FOR SELECT
USING (true);

-- Users can only update their own profile
CREATE POLICY "Users can update own profile"
ON profiles FOR UPDATE
USING (auth.uid() = id);
```

contributions Table

```sql
-- Users can view their own contributions
CREATE POLICY "Users can view own contributions"
ON contributions FOR SELECT
USING (auth.uid() = user_id);

-- Users can insert their own contributions
CREATE POLICY "Users can insert own contributions"
ON contributions FOR INSERT
WITH CHECK (auth.uid() = user_id);

-- Admins can view all contributions
CREATE POLICY "Admins can view all contributions"
ON contributions FOR SELECT
USING (EXISTS (
  SELECT 1 FROM user_roles
  WHERE user_id = auth.uid()
  AND role = 'admin'
));
```

pubs Table

```
-- Anyone can view active pubs
CREATE POLICY "Active pubs are publicly viewable"
ON pubs FOR SELECT
USING (status = 'Active');

-- Pub owners can view/update their own pubs
CREATE POLICY "Pub owners can update own pub"
ON pubs FOR UPDATE
USING (auth.uid() = user_id);

-- Admins have full access
CREATE POLICY "Admins have full access to pubs"
ON pubs FOR ALL
USING (EXISTS (
  SELECT 1 FROM user_roles
  WHERE user_id = auth.uid()
  AND role = 'admin'
));
```

vouchers Table

```
-- Users can view their own vouchers
CREATE POLICY "Users can view own vouchers"
ON vouchers FOR SELECT
USING (auth.uid() = user_id);

-- Pub owners can view vouchers for their taps
CREATE POLICY "Pub owners can view tap vouchers"
ON vouchers FOR SELECT
```

```
USING (EXISTS (
  SELECT 1 FROM taps t
  JOIN pubs p ON t.pub_id = p.id
  WHERE t.id = vouchers.tap_id
  AND p.user_id = auth.uid()
));

-- Pub owners can update vouchers (redemption)
CREATE POLICY "Pub owners can redeem vouchers"
ON vouchers FOR UPDATE
USING (EXISTS (
  SELECT 1 FROM taps t
  JOIN pubs p ON t.pub_id = p.id
  WHERE t.id = vouchers.tap_id
  AND p.user_id = auth.uid()
));
```

Input Validation

Frontend: Zod Schemas

```
import { z } from "zod";

const contributionSchema = z.object({
  cityId: z.string().uuid(),
  cityName: z.string().min(1).max(100),
});

// React Hook Form integration
const form = useForm<z.infer<typeof contributionSchema>>({
  resolver: zodResolver(contributionSchema),
});
```

Backend: Server-Side Validation

```
// Edge function validation
if (!pubId || typeof pubId !== "string") {
  return new Response(
    JSON.stringify({ error: "Invalid pubId" }),
    { status: 400 }
  );
}

// Email validation
const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
if (!emailRegex.test(email)) {
  return new Response(
    JSON.stringify({ error: "Invalid email format" }),
    { status: 400 }
  );
}
```

XSS Prevention

All user input is sanitized:

- React automatically escapes JSX content
- Dangerous HTML disabled by default
- No dangerouslySetInnerHTML usage in codebase

Content Security Policy (recommended addition):

```
headers: {
  "Content-Security-Policy":
    "default-src 'self'; script-src 'self' 'unsafe-inline'; style-src
'self' 'unsafe-inline';",
}
```

CORS Configuration

All Edge Functions include:

```
const corsHeaders = {
  "Access-Control-Allow-Origin": "*",
  "Access-Control-Allow-Headers":
    "authorization, x-client-info, apikey, content-type",
};

// Preflight handling
if (req.method === "OPTIONS") {
  return new Response(null, { headers: corsHeaders });
}
```

Webhook Security

Stripe Webhook Signature Verification:

```
const signature = req.headers.get("stripe-signature");
const webhookSecret = Deno.env.get("STRIPE_WEBHOOK_SECRET")!;

try {
  const event = await stripe.webhooks.constructEventAsync(
    await req.text(),
    signature!,
    webhookSecret
  );
  // Process event...
} catch (err) {
  console.error("Webhook signature verification failed:", err);
  return new Response(
    JSON.stringify({ error: "Invalid signature" }),
    { status: 400 }
  );
}
```

Secrets Management

Environment Variables: Stored in Supabase Secrets (encrypted at rest)

Required Secrets:

- STRIPE_SECRET_KEY: Stripe API key
- STRIPE_WEBHOOK_SECRET: Webhook signing secret
- RESEND_API_KEY: Email service API key
- MAPBOX_ACCESS_TOKEN: Mapbox token
- GOOGLE_MAPS_API_KEY: Google Places API key

Access in Edge Functions:

```
const stripeKey = Deno.env.get("STRIPE_SECRET_KEY");
if (!stripeKey) {
  throw new Error("STRIPE_SECRET_KEY not configured");
}
```

Key Database Triggers & Functions

Trigger: increment_city_contribution

Purpose: Automatically update city's total_contributions when a contribution is completed

```
CREATE OR REPLACE FUNCTION increment_city_contribution()
RETURNS TRIGGER AS $$
BEGIN
  IF NEW.status = 'completed' AND (OLD.status IS NULL OR OLD.status != 
'completed') THEN
    UPDATE cities
    SET total_contributions = total_contributions + NEW.amount
    WHERE id = NEW.city_id;
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER contribution_completed
AFTER INSERT OR UPDATE ON contributions
FOR EACH ROW
EXECUTE FUNCTION increment_city_contribution();
```

Trigger: update_suggestion_vote_count

Purpose: Maintain denormalized vote_count for performance

```
CREATE OR REPLACE FUNCTION update_suggestion_vote_count()
RETURNS TRIGGER AS $$
BEGIN
```

```
  IF TG_OP = 'INSERT' THEN
    UPDATE suggestions
    SET vote_count = vote_count + 1
    WHERE pub_id = NEW.pub_id
    AND city_id = NEW.city_id;
  ELSIF TG_OP = 'DELETE' THEN
    UPDATE suggestions
    SET vote_count = vote_count - 1
    WHERE pub_id = OLD.pub_id
    AND city_id = OLD.city_id;
  END IF;
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_vote_count_on_suggestion
AFTER INSERT OR DELETE ON suggestions
FOR EACH ROW
EXECUTE FUNCTION update_suggestion_vote_count();
```

Function: generate_voucher_pin

Purpose: Generate unique 6-digit PIN codes for vouchers

```
CREATE OR REPLACE FUNCTION generate_voucher_pin()
RETURNS TRIGGER AS $$
DECLARE
  new_pin TEXT;
  pin_exists BOOLEAN;
BEGIN
  LOOP
    -- Generate random 6-digit number
    new_pin := LPAD(FLOOR(RANDOM() * 1000000)::TEXT, 6, '0');

    -- Check if PIN already exists
    SELECT EXISTS(SELECT 1 FROM vouchers WHERE pin_code = new_pin) INTO
pin_exists;

    EXIT WHEN NOT pin_exists;
  END LOOP;

  NEW.pin_code := new_pin;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER set_voucher_pin
BEFORE INSERT ON vouchers
FOR EACH ROW
WHEN (NEW.pin_code IS NULL)
EXECUTE FUNCTION generate_voucher_pin();
```

Trigger: handle_new_user

Purpose: Create profile and assign default role when user signs up

```
CREATE OR REPLACE FUNCTION handle_new_user()
RETURNS TRIGGER AS $$
BEGIN
  -- Create profile
  INSERT INTO public.profiles (id, email, full_name)
  VALUES (
    NEW.id,
    NEW.email,
    NEW.raw_user_meta_data->>'full_name'
  );

  -- Assign default 'giver' role
  INSERT INTO public.user_roles (user_id, role)
  VALUES (NEW.id, 'giver');

  RETURN NEW;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;

CREATE TRIGGER on_auth_user_created
AFTER INSERT ON auth.users
FOR EACH ROW
EXECUTE FUNCTION handle_new_user();
```

Trigger: mark_token_used_on_pub_claim

Purpose: Mark invite token as used when pub completes onboarding

```
CREATE OR REPLACE FUNCTION mark_token_used_on_pub_claim()
RETURNS TRIGGER AS $$
BEGIN
  IF NEW.user_id IS NOT NULL AND OLD.user_id IS NULL THEN
    UPDATE pub_invite_tokens
    SET used = true
    WHERE pub_id = NEW.id;
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER token_used_on_claim
AFTER UPDATE ON pubs
FOR EACH ROW
WHEN (NEW.user_id IS NOT NULL AND OLD.user_id IS NULL)
EXECUTE FUNCTION mark_token_used_on_pub_claim();
```

Major Issues & Solutions

Issue #1: Stripe Connect Auto-Checkbox Not Working

Problem: After pub completed Stripe onboarding, the stripe_checked checkbox in AdminOnboardingManager was not automatically checking, blocking the "Launch & Notify Users" button.

Root Causes:

1. account.updated webhook was not configured in Stripe dashboard
2. create-connect-account-link function had bug where both "Complete Setup" and "Open Dashboard" buttons created account_onboarding type links instead of account_update
3. AdminOnboardingManager lacked useEffect to auto-check when stripe_charges_enabled and stripe_payouts_enabled became true

Solution Steps:

Step 1: Configure Webhook

- Added account.updated event to Stripe webhook configuration
- Webhook URL: https://[project-id].supabase.co/functions/v1/stripe-webhook
- Verified webhook secret in Supabase secrets

Step 2: Fix Edge Function

```
// supabase/functions/create-connect-account-link/index.ts (line 69)
// BEFORE:
type: type === 'dashboard' ? 'account_onboarding' : 'account_onboarding'

// AFTER:
type: type === 'dashboard' ? 'account_update' : 'account_onboarding'
```

Step 3: Add Webhook Handler

```
// supabase/functions/stripe-webhook/index.ts
case "account.updated": {
  const account = event.data.object as Stripe.Account;

  const { error: updateError } = await supabaseAdmin
    .from("pubs")
    .update({
      stripe_charges_enabled: account.charges_enabled || false,
      stripe_payouts_enabled: account.payouts_enabled || false,
    })
    .eq("stripe_account_id", account.id);

  if (updateError) throw updateError;
  break;
}
```

Step 4: Auto-Check Frontend

```
// src/components/AdminOnboardingManager.tsx
useEffect(() => {
  if (
    currentPub &&
    currentPub.stripe_charges_enabled &&
    currentPub.stripe_payouts_enabled &&
    !currentPub.stripe_checked
  ) {
    handleCheckboxChange("stripe_checked", true);
  }
}, [currentPub]);
```

Result: ✅ Stripe checkbox now auto-checks when onboarding complete, unblocking launch button

---

Issue #2: Launch Button Missing Despite Checks Complete

Problem: All checkboxes were checked and dates were set, but "Launch & Notify Users" button was not appearing because Stripe connection was failing.

Root Cause: isReadyToLaunch condition included currentPub.stripe_checked requirement:

```
const isReadyToLaunch = currentPub &&
  currentPub.contacted_checked &&
  currentPub.accepted_checked &&
  currentPub.verified_checked &&
  currentPub.stripe_checked &&  // Blocking here
  currentTap?.event_start_date &&
  currentTap?.event_end_date;
```

Solution: Temporarily removed stripe_checked from launch condition to allow testing of voucher email flow while Stripe issues were resolved:

```
// src/components/AdminOnboardingManager.tsx (lines 587-593)
const isReadyToLaunch = currentPub &&
  currentPub.contacted_checked &&
  currentPub.accepted_checked &&
  currentPub.verified_checked &&
  // Removed: currentPub.stripe_checked &&
  currentTap?.event_start_date &&
  currentTap?.event_end_date;
```

Added warning badge:

```
{isReadyToLaunch && !currentPub.stripe_checked && (
  <div className="flex items-center gap-2 p-3 bg-yellow-50 border border-yellow-200 rounded-lg">
    <AlertCircle className="h-4 w-4 text-yellow-600" />
    <span className="text-sm text-yellow-800">
      Launching without Stripe - Payments disabled until connected
    </span>
  </div>
)}
```

Result: ✅ Launch button now appears, allowing voucher email testing independent of Stripe status

---

Issue #3: Vouchers Not Created on Launch

Problem: After clicking "Launch & Notify Users", edge function logs showed "No vouchers found for this tap" and users didn't receive emails.

Root Cause: In handleLaunchAndNotify function, voucher creation logic was inside the if (!existingTap) block. If tap already existed from previous attempt, vouchers were never created:

```
if (!existingTap) {
  // Create tap
  const { data: newTap } = await supabase.from("taps").insert(...);

  // Voucher creation ONLY happened here!
  const { data: contributions } = await supabase
    .from("contributions")
    .select("id, user_id")
    .eq("city_id", selectedCity);

  // Insert vouchers...
}
```

Solution: Moved voucher creation logic outside the tap creation block and added duplicate check:

```
// src/components/AdminOnboardingManager.tsx (lines 499-549)
// Step 1: Handle tap creation/retrieval
if (!existingTap) {
  const { data: newTap } = await supabase.from("taps").insert(...);
  tapId = newTap.id;
}

// Step 2: Get contributions (ALWAYS runs)
const { data: contributions } = await supabase
  .from("contributions")
  .select("id, user_id")
  .eq("city_id", selectedCity);

// Step 3: Check if vouchers already exist
const { data: existingVouchers } = await supabase
  .from("vouchers")
  .select("id")
```

```
     .eq("tap_id", tapId);

    // Step 4: Create vouchers if they don't exist yet
    if (contributions && contributions.length > 0 &&
        (!existingVouchers || existingVouchers.length === 0)) {

      const voucherInserts = contributions.map((contribution) => ({
        user_id: contribution.user_id,
        tap_id: tapId,
        type: "Pledge" as const,
        is_redeemed: false,
      }));

      await supabase.from("vouchers").insert(voucherInserts);
      console.log(`Created ${voucherInserts.length} vouchers for tap
    ${tapId}`);
    } else if (existingVouchers && existingVouchers.length > 0) {
      console.log(`Vouchers already exist for tap ${tapId}, skipping
    creation`);
    }
```

Result: ✅ Vouchers now created correctly, unblocking email delivery

---

Issue #4: Email Not Sent After Launch

Problem: Even after vouchers were created, users still didn't receive launch notification emails.

Root Cause: handleLaunchAndNotify was calling sendLaunchNotification edge function immediately after voucher creation, but the async voucher inserts hadn't completed yet.

Solution: Ensured proper async/await ordering:

```
    // Create vouchers
```

```
const { error: vouchersError } = await supabase
  .from("vouchers")
  .insert(voucherInserts);

if (vouchersError) {
  console.error("Failed to create vouchers:", vouchersError);
  throw vouchersError;
}

// Wait for vouchers to be committed before sending emails
await new Promise(resolve => setTimeout(resolve, 1000)); // 1 second
buffer

// Now send emails
const { data: emailData, error: emailError } = await
supabase.functions.invoke(
  "send-launch-notification",
  { body: { tapId } }
);
```

Result: ✅ Emails now sent successfully after voucher creation completes

---

Issue #5: CORS Errors on Edge Function Calls

Problem: Frontend calls to edge functions failed with CORS errors in browser console.

Root Cause: Edge functions missing CORS headers or improper OPTIONS preflight handling.

Solution: Added consistent CORS headers to all edge functions:

```
const corsHeaders = {
  "Access-Control-Allow-Origin": "*",
  "Access-Control-Allow-Headers":
    "authorization, x-client-info, apikey, content-type",
};

Deno.serve(async (req) => {
  // Handle CORS preflight
  if (req.method === "OPTIONS") {
    return new Response(null, {
      headers: corsHeaders,
      status: 204
    });
  }

  try {
    // Main logic...
    return new Response(JSON.stringify(data), {
      headers: { ...corsHeaders, "Content-Type": "application/json" },
      status: 200,
    });
  } catch (error) {
    return new Response(JSON.stringify({ error: error.message }), {
      headers: { ...corsHeaders, "Content-Type": "application/json" },
      status: 500,
    });
  }
});
```

Result: ✅ All edge functions now properly handle CORS

Performance & Scalability

Database Optimizations

Indexes:

```sql
-- Frequently queried columns
CREATE INDEX idx_contributions_user_city ON contributions(user_id,
city_id);
CREATE INDEX idx_contributions_status ON contributions(status);
CREATE INDEX idx_vouchers_user_id ON vouchers(user_id);
CREATE INDEX idx_vouchers_tap_id ON vouchers(tap_id);
CREATE INDEX idx_vouchers_pin ON vouchers(pin_code);
CREATE INDEX idx_suggestions_city_pub ON suggestions(city_id, pub_id);
CREATE INDEX idx_pubs_stripe_account ON pubs(stripe_account_id);
```

Denormalization:

- cities.total_contributions: Aggregated via trigger instead of SUM query
- suggestions.vote_count: Maintained via trigger for fast leaderboard queries

Query Optimization:

```sql
-- GOOD: Uses index
SELECT * FROM vouchers
WHERE user_id = '...' AND tap_id = '...';

-- AVOID: Full table scan
SELECT * FROM vouchers
WHERE pin_code LIKE '%123%';
```

Edge Function Best Practices

1. Connection Pooling:

```
// Reuse Supabase client
const supabaseAdmin = createClient(
  supabaseUrl,
  supabaseServiceKey,
  { db: { schema: "public" } }
);
```

2. Batch Operations:

```
// GOOD: Batch insert
await supabase.from("vouchers").insert(voucherArray);

// AVOID: Loop inserts
for (const voucher of vouchers) {
  await supabase.from("vouchers").insert(voucher);
}
```

3. Early Returns:

```
// Fail fast
if (!userId) {
  return new Response(JSON.stringify({ error: "Unauthorized" }), {
    status: 401,
  });
}
```

Frontend Optimizations

React Query Caching:

```
const { data: cities } = useQuery({
  queryKey: ["cities"],
  queryFn: fetchCities,
  staleTime: 5 * 60 * 1000, // 5 minutes
  cacheTime: 10 * 60 * 1000, // 10 minutes
});
```

Code Splitting:

```
const AdminDashboard = lazy(() => import("./pages/Dashboard"));
const PubDashboard = lazy(() => import("./pages/PubDashboard"));
```

Image Optimization:

- Use WebP format
- Lazy load images below fold
- Compress with tools like TinyPNG

Conclusion

This documentation covers the complete technical architecture of CommunityTap, including:

- ✅ Database schema and entity relationships
- ✅ User flows with detailed sequence diagrams
- ✅ API integrations (Stripe, Resend, Mapbox, Google Maps)
- ✅ Authentication and authorization systems
- ✅ Security measures (RLS, validation, webhook verification)
- ✅ Key database triggers and functions
- ✅ Major issues encountered and solutions implemented