Multi-Auth Strategy for Your NextJS App

Your authentication requirements are absolutely feasible with Supabase, and your approach sounds well-thought-out. Let me address each point and provide implementation guidance.

Different Auth Methods for Mobile vs Desktop

There will be **two different authentication methods in the same app**, for mobile and desktop users. Implementation details are below.

1. Mobile Workers

Passcode Auth

- Create a custom passcode authentication system using Supabase functions
- Each worker gets a unique 4-6 digit code
- Store these codes in a secure table with user IDs

```
// Mobile login component
const handlePasscodeLogin = async (passcode) => {
  // Query your passcodes table
  const { data, error } = await supabase
    .from("worker_passcodes")
    .select("user_id")
    .eq("passcode", passcode)
    .single();

  if (error || !data) {
    return { error: "Invalid passcode" };
  }

  // Sign in as this user using their ID
  // This requires a server-side function with service_role key
  const { data: authData, error: authError } = await fetch(
    "/api/auth/passcode-login",
    {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ userId: data.user_id }),
    }
  ).then((res) => res.json());

  return { data: authData, error: authError };
};
```

Server-side API route

> app/api/auth/passcode-login.js

```
import { createClient } from "@supabase/supabase-js";

// Initialize with service_role key to bypass RLS
const supabase = createClient(
  process.env.NEXT_PUBLIC_SUPABASE_URL,
  process.env.SUPABASE_SERVICE_ROLE_KEY
);

export default async function handler(req, res) {
  const { userId } = req.body;

  // Create a custom session for this user
  const { data, error } = await supabase.auth.admin.createSession({
    user_id: userId,
  });

  if (error) {
    return res.status(401).json({ error: error.message });
  }

  // Set session cookie
  res.setHeader("Set-Cookie", [
    `sb-access-token=${data.session.access_token}; Path=/; HttpOnly`,
    `sb-refresh-token=${data.session.refresh_token}; Path=/; HttpOnly`,
  ]);

  return res.status(200).json({ user: data.user });
}
```

2. Desktop Users

- Email/Password Auth
- 7-day JWT refresh tokens
- Use Supabase's default email/password authentication
- Implement RLS policies to restrict access to only the relevant data
- User details are stored in the `auth.users` table

Session Durations

There will be different session durations **based on device type**.

1. Mobile sessions expire after 10 minutes

```
// When creating mobile session
const { data, error } = await supabase.auth.admin.createSession({
  user_id: userId,
  expires_in: 600, // 10 minutes in seconds
});
```

2. Desktop sessions expire after 7 days

```
// When signing in on desktop
const { data, error } = await supabase.auth.signInWithPassword({
  email,
  password,
  options: {
    expiresIn: 60 * 60 * 24 * 7, // 7 days in seconds
  },
});
```

3. Auto-logout for inactive mobile sessions

```
// On mobile view, add this activity tracker
useEffect(() => {
  let inactivityTimer;

  const resetTimer = () => {
    clearTimeout(inactivityTimer);
    inactivityTimer = setTimeout(
      () => {
        // Log out after 10 minutes of inactivity
        supabase.auth.signOut();
        router.push("/mobile-login");
      },
      10 * 60 * 1000
    ); // 10 minutes
  };

  // Reset timer on user activity
  window.addEventListener("click", resetTimer);
  window.addEventListener("keypress", resetTimer);
  window.addEventListener("touchstart", resetTimer);

  // Initialize timer
  resetTimer();

  return () => {
    clearTimeout(inactivityTimer);
    window.removeEventListener("click", resetTimer);
    window.removeEventListener("keypress", resetTimer);
    window.removeEventListener("touchstart", resetTimer);
  };
}, []);
```

Device Recognition

Supabase can recognize the device type from the request headers (via refresh tokens).

The session persistence works through cookies/local storage, so:

- If a desktop user closes their browser tab but doesn't clear cookies, they will remain logged in when they return (within the 7-day period)
- The session is tied to the browser, not the specific physical device

- If they use a different browser or clear cookies, they'll need to login again

For corporate settings, this is ideal since users typically use the same browser on their assigned PC.

Corporate Setting Advice

1. **User Management System:**

- Create a simple admin interface to manage worker passcodes
- Allow administrators to reset passcodes when needed
- Track which worker is using which mobile device at what time

2. **Passcode Security:**

- Implement passcode rotation (e.g., weekly or monthly)
- Add a "shift change" button that forces logout when workers change shifts
- Consider requiring supervisor approval for certain high-value transactions

3. **Setup for Shared Mobile Devices:**

- Add a prominent "Current User" indicator on all mobile screens
- Implement a quick-logout button that's always visible
- Consider adding a feature to "pause" session (requiring passcode re-entry) for short breaks

4. **Implementation Example for Passcode Management:**

See the passcode-login.md file for component implementation
See the passcode-auth.md file for the API route implementation
See the user-management.md file for the user management implementation

Required Database Setup

To implement this dual-authentication system, you'll need these Supabase tables:

1. `worker_passcodes` table (for passcode authentication)

- passcodes will be generated by the admin interface and stored in this table.
- admin and manager roles will be able to view all passcodes and their associated users.
- worker roles will only be able to view their own passcode.
- passcodes will be 4-6 digits long and will be generated by the admin interface.
- table will be in the `public` schema.

```sql
CREATE TABLE worker_passcodes (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  user_id UUID NOT NULL REFERENCES auth.users(id),
  worker_name TEXT NOT NULL,
  role TEXT NOT NULL,
  passcode TEXT NOT NULL,
  is_active BOOLEAN DEFAULT true,
  last_login_at TIMESTAMP WITH TIME ZONE,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT now(),
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT now(),
  created_by UUID REFERENCES auth.users(id),
  updated_by UUID REFERENCES auth.users(id)
);

-- Index for faster passcode lookups
CREATE INDEX idx_worker_passcodes_passcode ON worker_passcodes(passcode);

-- Ensure passcodes are unique
CREATE UNIQUE INDEX idx_worker_passcodes_unique_passcode ON worker_passcodes(passcode)
WHERE is_active = true;
```

2. `user_roles` (for role-based access)

```sql
CREATE TABLE user_roles (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  user_id UUID NOT NULL REFERENCES auth.users(id),
  role TEXT NOT NULL,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT now(),
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT now()
);

-- Index for faster user lookups
CREATE INDEX idx_user_roles_user_id ON user_roles(user_id);

-- Ensure one user has only one role record
CREATE UNIQUE INDEX idx_user_roles_unique_user ON user_roles(user_id);
```

3. `auth_activity_log` (for auditing)

```sql
CREATE TABLE auth_activity_log (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  user_id UUID NOT NULL REFERENCES auth.users(id),
  action TEXT NOT NULL, -- 'login', 'logout', 'mobile_login', etc.
  ip_address TEXT,
  metadata JSONB,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT now()
);

-- Index for faster user activity lookups
CREATE INDEX idx_auth_activity_log_user_id ON auth_activity_log(user_id);
-- Index for time-based queries
CREATE INDEX idx_auth_activity_log_created_at ON auth_activity_log(created_at);
```

4. `session_settings` (for device-specific duration settings)

```sql
CREATE TABLE session_settings (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  device_type TEXT NOT NULL, -- 'mobile', 'desktop'
  session_duration_seconds INTEGER NOT NULL, -- in seconds
  inactivity_timeout_seconds INTEGER, -- in seconds
  created_at TIMESTAMP WITH TIME ZONE DEFAULT now(),
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT now()
);

-- Default settings
INSERT INTO session_settings (device_type, session_duration_seconds, inactivity_timeout
VALUES
  ('mobile', 600, 600), -- 10 minutes for mobile
  ('desktop', 604800, NULL); -- 7 days for desktop, no inactivity timeout
```

Row-Level Security (RLS) Policies

`worker_passcodes`

```sql
-- Enable RLS
ALTER TABLE worker_passcodes ENABLE ROW LEVEL SECURITY;

-- Admins can view all passcodes
CREATE POLICY "Admins can view all passcodes"
ON worker_passcodes FOR SELECT
USING (
  auth.uid() IN (
    SELECT user_id FROM user_roles WHERE role = 'admin'
  )
);

-- Admins can insert passcodes
CREATE POLICY "Admins can insert passcodes"
ON worker_passcodes FOR INSERT
WITH CHECK (
  auth.uid() IN (
    SELECT user_id FROM user_roles WHERE role = 'admin'
  )
);

-- Admins can update passcodes
CREATE POLICY "Admins can update passcodes"
ON worker_passcodes FOR UPDATE
USING (
  auth.uid() IN (
    SELECT user_id FROM user_roles WHERE role = 'admin'
  )
);
```

`user_roles`

```sql
-- Enable RLS
ALTER TABLE user_roles ENABLE ROW LEVEL SECURITY;

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

-- Admins can view all roles
CREATE POLICY "Admins can view all roles"
ON user_roles FOR SELECT
USING (
  auth.uid() IN (
    SELECT user_id FROM user_roles WHERE role = 'admin'
  )
);

-- Admins can insert roles
CREATE POLICY "Admins can insert roles"
ON user_roles FOR INSERT
WITH CHECK (
  auth.uid() IN (
    SELECT user_id FROM user_roles WHERE role = 'admin'
  )
);

-- Admins can update roles
CREATE POLICY "Admins can update roles"
ON user_roles FOR UPDATE
USING (
  auth.uid() IN (
    SELECT user_id FROM user_roles WHERE role = 'admin'
  )
);
```

Additional API Endpoints Required

For the dual-authentication system, implement these additional API endpoints:

1. **Create User API (`/api/auth/create-user`)**

```
// For admin to create email+password users
import { createClient } from "@supabase/supabase-js";

const supabase = createClient(
  process.env.NEXT_PUBLIC_SUPABASE_URL,
  process.env.SUPABASE_SERVICE_ROLE_KEY
);

export default async function handler(req, res) {
  if (req.method !== "POST") {
    return res.status(405).json({ error: "Method not allowed" });
  }

  const { email, password, user_metadata } = req.body;

  try {
    const { data, error } = await supabase.auth.admin.createUser({
      email,
      password,
      email_confirm: true, // Auto-confirm for internal accounts
      user_metadata,
    });

    if (error) throw error;

    // Add user role if specified
    if (user_metadata?.role) {
      await supabase.from("user_roles").insert({
        user_id: data.user.id,
        role: user_metadata.role,
      });
    }

    return res.status(200).json({ user: data.user });
  } catch (error) {
    console.error("Error creating user:", error);
    return res.status(500).json({ error: error.message });
  }
}
```

2. **Create Placeholder User API (`/api/auth/create-placeholder-user.js`)**

```
// For workers without email addresses
import { createClient } from "@supabase/supabase-js";
import { v4 as uuidv4 } from "uuid";

const supabase = createClient(
  process.env.NEXT_PUBLIC_SUPABASE_URL,
  process.env.SUPABASE_SERVICE_ROLE_KEY
);

export default async function handler(req, res) {
  if (req.method !== "POST") {
    return res.status(405).json({ error: "Method not allowed" });
  }

  const { name, role } = req.body;

  try {
    // Create a placeholder email
    const placeholderEmail = `worker-${uuidv4()}@placeholder.internal`;

    const { data, error } = await supabase.auth.admin.createUser({
      email: placeholderEmail,
      password: uuidv4(), // Random password that won't be used
      email_confirm: true,
      user_metadata: {
        full_name: name,
        role: role,
        is_placeholder: true,
      },
    });

    if (error) throw error;

    // Add user role
    await supabase.from("user_roles").insert({
      user_id: data.user.id,
      role: role,
    });

    return res.status(200).json({ user: data.user });
  } catch (error) {
    console.error("Error creating placeholder user:", error);
    return res.status(500).json({ error: error.message });
    }
  }
```

Middleware Configuration

Finally, to handle different session durations based on device type, create a middleware file:

```
// middleware.js
import { createMiddlewareClient } from "@supabase/auth-helpers-nextjs";
import { NextResponse } from "next/server";

export async function middleware(req) {
  const res = NextResponse.next();
  const supabase = createMiddlewareClient({ req, res });

  // Check if there's a session
  const {
    data: { session },
  } = await supabase.auth.getSession();

  // Detect if this is a mobile device
  const isMobile = req.headers
    .get("user-agent")
    ?.match(/Android|webOS|iPhone|iPad|iPod|BlackBerry|IEMobile|Opera Mini/i);

  const path = req.nextUrl.pathname;

  // Handle mobile routes
  if (path.startsWith("/mobile")) {
    // Redirect to mobile login if no session
    if (!session && path !== "/mobile/login") {
      return NextResponse.redirect(new URL("/mobile/login", req.url));
    }
  }

  // Handle desktop/admin routes
  if (path.startsWith("/admin")) {
    if (!session) {
      return NextResponse.redirect(new URL("/login", req.url));
    }

    // Check if user has admin role
    const { data: userData } = await supabase.auth.getUser();
    const { data: roleData } = await supabase
      .from("user_roles")
      .select("role")
      .eq("user_id", userData.user?.id)
      .single();

    if (!roleData || roleData.role !== "admin") {
      return NextResponse.redirect(new URL("/dashboard", req.url));
    }
  }

  return res;
}

// Define which routes to run middleware on
export const config = {
  matcher: ["/mobile/:path*", "/admin/:path*", "/dashboard/:path*"],
};
```

Security Considerations

- Storing unencrypted passcodes in the public schema is not recommended for several reasons:
  - Exposure risk: Even with RLS, storing plaintext passcodes creates unnecessary risk if there's ever a security breach or misconfiguration.
  - Best practice violation: Authentication credentials should always be hashed, not stored in plaintext, regardless of where they're stored.
  - RLS limitations: While RLS can restrict access, it's a second line of defense. If there's ever an RLS policy bug or if someone gains elevated database access, all passcodes would be exposed.
- Better approaches:
  - Hash the passcodes (using bcrypt or similar)
  - Move this table to a more restricted schema like auth
  - Implement proper RLS policies that restrict workers to only see their own records

Even for an internal app, following security best practices protects against insider threats and accidental exposures.

Feature Overview

This comprehensive setup provides:

- Different auth methods for mobile (passcode) and desktop (email/password)
- Different session durations by device type
- Proper security with RLS policies
- Admin tools to manage worker passcodes
- Activity logging for security audits