Integrating Prisma with Supabase Auth: Best Practices

Overview

We've implemented a streamlined integration between Prisma ORM and Supabase Auth that uses `auth.users` as the single source of truth. Here's how we achieved this and how to work with it effectively.

Key Architecture Decisions

1. **Direct `auth.users` Reference** - Our Prisma schema directly references the Supabase Auth table instead of duplicating user data
2. **Multi-Schema Support** - Using Prisma's `multiSchema` to access both `public` and `auth` schemas
3. **Unified Foreign Keys** - All user references point to `auth.users` with proper UUID types
4. **Helper Functions** - Abstraction layer to simplify working with the integrated system

Implementation Details

1. Prisma Configuration

```
// prisma/schema.prisma
generator client {
  provider        = "prisma-client-js"
  previewFeatures = ["multiSchema"]
  engineType      = "binary"
}

datasource db {
  provider  = "postgresql"
  url       = env("DATABASE_URL")
  directUrl = env("DIRECT_URL")
  schemas   = ["public", "auth"]
}
```

2. User Model Definition

```
// prisma/schema.prisma
model User {
  id              String            @id @db.Uuid
  email           String            @unique
  raw_user_meta_data Json?          @map("raw_user_meta_data")
  raw_app_meta_data Json?           @map("raw_app_meta_data")
  created_at      DateTime          @db.Timestamptz(6)
  last_sign_in_at DateTime?         @db.Timestamptz(6)

  // Relationships
  organizations   UserOrganization[]
  userRoles       UserRoleMapping[]
  // Other relationships...

  @@map("users")
  @@schema("auth")
}
```

3. Foreign Key References

```
// prisma/schema.prisma
model UserOrganization {
  id             String       @id @default(cuid())
  userId         String       @map("user_id") @db.Uuid
  user           User         @relation(fields: [userId], references:
[id], onDelete: Cascade)
  organizationId String       @map("organization_id")
  organization   Organization @relation(fields: [organizationId],
references: [id], onDelete: Cascade)

  @@map("user_organizations")
  @@schema("public")
}
```

Working with the Integration

Retrieving User Data with Relations

```
// Example of getting a user with their organizations
```

```
import { PrismaClient } from '@prisma/client'
const prisma = new PrismaClient()

async function getUserWithOrgs(userId: string) {
  return prisma.user.findUnique({
    where: { id: userId },
    include: {
      organizations: {
        include: {
          organization: true
        }
      }
    }
  })
}

// Usage
const user = await getUserWithOrgs('auth0|123456')
console.log(user.email)
console.log(user.organizations[0].organization.name)
```

Creating User Relationships

```
// Example: Assigning a user to an organization
async function addUserToOrg(userId: string, orgId: string) {
  return prisma.userOrganization.create({
    data: {
      userId,  // UUID from auth.users
      organizationId: orgId,
      isDefault: false
    }
  })
}
```

Updating User Metadata in Supabase

```
import { createClient } from '@supabase/supabase-js'

const supabase = createClient(
  process.env.NEXT_PUBLIC_SUPABASE_URL!,
  process.env.SUPABASE_SERVICE_KEY!
)

async function updateUserMetadata(userId: string, metadata:
Record<string, any>) {
```

```
  const { data, error } = await supabase.auth.admin.updateUserById(
    userId,
    { user_metadata: metadata }
  )

  if (error) throw new Error(`Failed to update user metadata:
${error.message}`)
  return data.user
}
```

Getting The Current Authenticated User

```
async function getCurrentUser(req: any) {
  // Get the user from the session
  const { data: { session } } = await supabaseClient.auth.getSession()
  if (!session) return null

  // Use Prisma to get the user with related data
  return prisma.user.findUnique({
    where: { id: session.user.id },
    include: {
      organizations: {
        include: { organization: true }
      },
      userRoles: {
        include: { role: true }
      }
    }
  })
}
```

Common Errors and Pitfalls to Avoid

1. Duplicate User Tables

❌ **Bad Practice**:

```
// AVOID THIS!
model User {
  id        String   @id @default(cuid())
  email     String   @unique
  // Duplicating what's already in auth.users

  @@schema("public")
```

```
}
```

✅ **Correct Approach**:

```
model User {
  id        String   @id @db.Uuid
  email     String   @unique

  @@map("users")
  @@schema("auth")
}
```

2. Inconsistent ID Types

❌ **Common Error**:

```
// AVOID THIS!
model UserOrganization {
  userId    String   @map("user_id")  // Missing @db.Uuid
  user      User     @relation(fields: [userId], references: [id])
  // This will fail with type mismatch
}
```

✅ **Correct Approach**:

```
model UserOrganization {
  userId    String   @map("user_id") @db.Uuid
  user      User     @relation(fields: [userId], references: [id])
}
```

3. Forgetting Multi-Schema Configuration

❌ **Error**:

```
// AVOID THIS!
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
  // Missing schemas definition and directUrl
}
```

4. Using Text IDs with UUID Foreign Keys

❌ **Runtime Error**:

```
// AVOID THIS!
await prisma.userOrganization.create({
  data: {
    userId: "simple-text-id", // Not a UUID
    organizationId: orgId
  }
})
// Will fail with foreign key constraint error
```

✅ **Correct Approach**:

```
await prisma.userOrganization.create({
  data: {
    userId: "123e4567-e89b-12d3-a456-426614174000", // Valid UUID
    organizationId: orgId
  }
})
```

5. Manually Syncing User Data

❌ **Anti-Pattern**:

```
// AVOID THIS!
async function createUser(data) {
  // Create in Supabase Auth
  const { data: authData } = await supabase.auth.signUp({
    email: data.email,
    password: data.password
  })

  // Then duplicate in public.users table
  await prisma.customUser.create({
    data: {
      id: authData.user.id,
      email: data.email
      // Duplicating data
    }
  })
}
```

✅ **Better Approach**:
Let Supabase handle the auth, and use Prisma to work with the existing user record:

```
async function setupNewUser(userId) {
  // The user already exists in auth.users
  // Just create related records
  await prisma.userProfile.create({
    data: {
      userId: userId,
      displayName: "New User",
      // other profile data
    }
  })
}
```

Tips for Robust Implementation

1. **Always use `@db.Uuid` annotation** for fields referencing `auth.users`
2. **Keep user management in Supabase Auth** (signup, login, password reset)
3. **Store extended profile data** in a separate `profiles` table with foreign key to `auth.users`
4. **Use TypeScript to enforce UUID type** with user IDs:

```
function getUserOrganizations(userId: string) {
  // Validate UUID format first
  if (!/^[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]
{12}$/i.test(userId)) {
    throw new Error('Invalid UUID format for userId')
  }

  return prisma.userOrganization.findMany({
    where: { userId }
  })
}
```

5. **Check the Prisma query log** during development to ensure you're not making redundant queries:

```
const prisma = new PrismaClient({
  log: ['query', 'info', 'warn', 'error'],
})
```

By following these practices, you'll have a clean, efficient integration between Prisma

and Supabase that avoids duplicated data and maintains proper referential integrity.