Integrating Prisma, Supabase, and Keycloak for User Management

Overview

This guide outlines a clean integration between Prisma ORM, Supabase PostgreSQL, and Keycloak for authentication. This architecture uses Keycloak as the identity provider while leveraging Supabase for database capabilities and Prisma for type-safe data access.

Architecture Principles

1. **Single Source of Truth**: Keycloak manages identity and authentication
2. **Foreign Key References**: Application data references Keycloak user IDs
3. **Type-Safe Access**: Prisma provides structured data access with TypeScript
4. **Rich Database Features**: Supabase PostgreSQL delivers advanced capabilities

Implementation Strategy

1. Database Setup

```
### Supabase Schema Design

- Don't create redundant user tables in Supabase
- Store user IDs from Keycloak as foreign keys
- Use UUID type for Keycloak user references
```

2. Prisma Schema Configuration

```
// Key Prisma schema elements
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}
```

```
// External identity reference rather than full user model
model UserProfile {
  id              String    @id @default(cuid())
  keycloakId      String    @unique @map("keycloak_id") @db.Uuid
  displayName     String?   @map("display_name")
  avatarUrl       String?   @map("avatar_url")

  // Relationships to other tables
  organizations   UserOrganization[]
  // ...other relationships
}

model UserOrganization {
  id              String     @id @default(cuid())
  userProfileId   String     @map("user_profile_id")
  userProfile     UserProfile @relation(fields: [userProfileId],
references: [id], onDelete: Cascade)
  organizationId  String     @map("organization_id")
  organization    Organization @relation(fields: [organizationId],
references: [id], onDelete: Cascade)

  @@unique([userProfileId, organizationId])
}
```

3. Keycloak Integration Layer

```
### Auth Service Structure

Create an abstraction layer that:
- Manages Keycloak session & token handling
- Provides user identity to your application
- Syncs essential user data with your database
- Handles JWT validation and role mapping
```

Working with the Integration

User Authentication Flow

1. User authenticates via Keycloak (OIDC/OAuth2)
2. Application receives tokens and user info
3. Application uses Keycloak ID to find or create user profile
4. Prisma queries join user profile with application data

Code Patterns

1. Authentication Integration

```
// src/lib/auth/keycloak.ts

// Initialize Keycloak client
// Manage authentication state
// Handle token refresh
// Provide user identity to application
```

2. User Profile Management

```
// Find or create user profile based on Keycloak identity
async function ensureUserProfile(keycloakUser) {
  const { sub, email, name } = keycloakUser;

  // Use Prisma to find or create user profile
  // Return the user profile with related data
}
```

3. Authorization Checks

```
// Check user permissions combining Keycloak roles and application data
function canUserAccessResource(keycloakId, resourceId) {
  // Check Keycloak roles AND application permissions
  // Return boolean indicating access
}
```

Best Practices

1. User Identity Management

- **Do**: Use Keycloak ID (`sub` claim) as the unique identifier
- **Don't**: Create duplicate user tables with redundant identity information
- **Do**: Consider storing a cache of frequently needed user attributes

2. Data Access Patterns

- **Do**: Create a `UserProfile` model that references Keycloak ID
- **Don't**: Store sensitive auth data in your application database
- **Do**: Use Prisma transactions for multi-step operations

3. JWT Handling

- **Do**: Validate Keycloak JWTs on sensitive operations
- **Don't**: Trust client-provided user IDs without verification
- **Do**: Implement proper token refresh and session management

Common Pitfalls and Solutions

Pitfall 1: Inconsistent User References

❌ **Problematic Code**:

```
// Different ID formats across the application
const userId = "user123";  // String ID in one place
const keycloakId = "f47ac10b-58cc-4372-a567-0e02b2c3d479"; // UUID in
another
```

✅ **Solution**:
Consistently use Keycloak's UUID format and validate format when accepting user IDs.

Pitfall 2: Missing User Profiles

❌ **Error Scenario**:
User authenticates in Keycloak but has no profile in application database.

✅ **Solution**:
Implement a "just-in-time" provisioning pattern that creates user profiles on first login.

Pitfall 3: Overreliance on Keycloak API

❌ **Performance Issue**:
Frequent calls to Keycloak Admin API for basic user data.

✅ **Solution**:
Cache essential user data in your UserProfile model, updated on login/session refresh.

Pitfall 4: Mixing Auth Concerns

❌ **Anti-Pattern**:

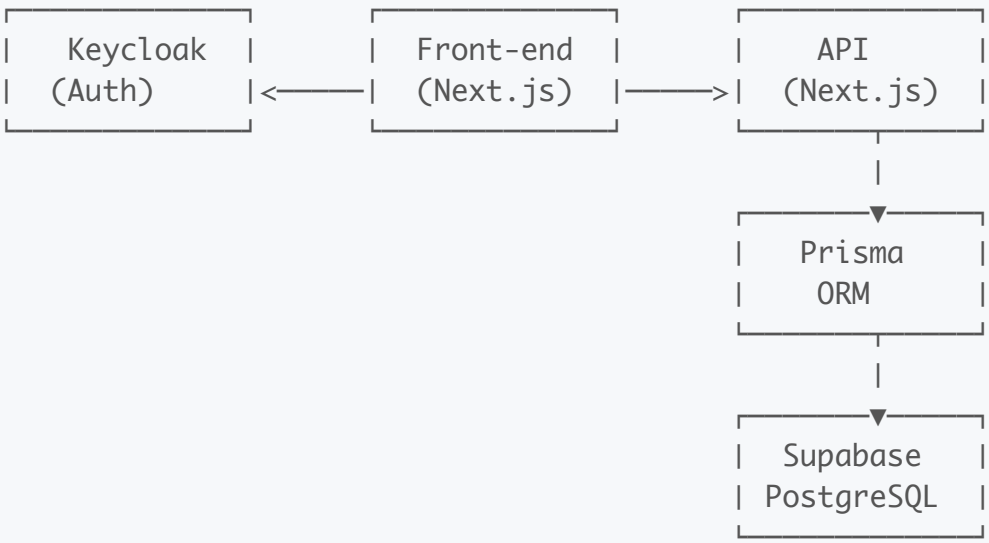
```
// Directly checking Keycloak roles deep in business logic
if (keycloak.hasRealmRole('admin')) {
  // Allow operation
}
```

✅ **Solution**:
Abstract authorization logic into a dedicated service that combines Keycloak roles with application permissions.

Implementation Checklist

- [ ] Configure Keycloak realm and client settings
- [ ] Design Prisma schema with proper Keycloak ID references
- [ ] Create authentication service with token handling
- [ ] Implement user profile synchronization
- [ ] Set up authorization middleware/hooks
- [ ] Add role mapping between Keycloak and application
- [ ] Implement JWT validation for API routes

Example Architecture Diagram



Integration Code Examples

1. Initialize Auth in Frontend

```
// pages/_app.tsx
import { KeycloakProvider } from 'your-keycloak-library';

function MyApp({ Component, pageProps }) {
  return (
    <KeycloakProvider
      realm={process.env.NEXT_PUBLIC_KEYCLOAK_REALM}
      url={process.env.NEXT_PUBLIC_KEYCLOAK_URL}
      clientId={process.env.NEXT_PUBLIC_KEYCLOAK_CLIENT_ID}
    >
      <Component {...pageProps} />
    </KeycloakProvider>
  );
}
```

2. User Profile Synchronization

```
// lib/auth/sync-user.ts
async function syncUserProfile(keycloakUser) {
  const { sub, email, name, preferred_username } = keycloakUser;

  try {
    // Find or create the user profile
    const profile = await prisma.userProfile.upsert({
      where: { keycloakId: sub },
      create: {
        keycloakId: sub,
        displayName: name || preferred_username,
        email
      },
      update: {
        displayName: name || preferred_username,
        email
      }
    });

    return profile;
  } catch (error) {
    console.error('Failed to sync user profile:', error);
    throw new Error('User profile synchronization failed');
  }
```

```
}
```

3. Protected API Route

```
// pages/api/protected.ts
import { verifyKeycloakToken } from 'lib/auth/verify-token';

export default async function handler(req, res) {
  try {
    // Verify the token and extract user info
    const user = await verifyKeycloakToken(req);
    if (!user) {
      return res.status(401).json({ error: 'Unauthorized' });
    }

    // Get the user profile with Prisma
    const profile = await prisma.userProfile.findUnique({
      where: { keycloakId: user.sub },
      include: { /* related data */ }
    });

    // Process the request with user context
    // ...

    return res.status(200).json({ data: 'Protected data' });
  } catch (error) {
    return res.status(500).json({ error: 'Internal server error' });
  }
}
```

Database Schema Considerations

Key Tables

1. **UserProfile**: Links Keycloak identity to application data
2. **Permissions**: Application-specific permissions, complementing Keycloak roles
3. **Organizations/Teams**: Group structures for users
4. **UserSettings**: User preferences specific to your application

Example Supabase Migration

```sql
-- Create UserProfile table
CREATE TABLE user_profiles (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  keycloak_id UUID UNIQUE NOT NULL,
  display_name TEXT,
  email TEXT UNIQUE,
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
  updated_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

-- Create organizations table
CREATE TABLE organizations (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name TEXT NOT NULL,
  -- other fields
);

-- Create user_organizations junction table
CREATE TABLE user_organizations (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_profile_id UUID REFERENCES user_profiles(id) ON DELETE CASCADE,
  organization_id UUID REFERENCES organizations(id) ON DELETE CASCADE,
  role TEXT NOT NULL,
  UNIQUE(user_profile_id, organization_id)
);
```

Security Considerations

1. **JWT Validation**: Always validate Keycloak tokens server-side
2. **Role Mapping**: Map Keycloak roles to application permissions
3. **Token Storage**: Store tokens securely using HTTP-only cookies
4. **CORS Configuration**: Set appropriate CORS policies for Keycloak
5. **API Security**: Implement rate limiting and monitoring

Conclusion

This integration approach gives you:

- Professional identity management with Keycloak
- Powerful database capabilities with Supabase
- Type-safe data access with Prisma

- Clean separation of concerns

By following this guide, you'll create a robust user management system that leverages the strengths of all three platforms without unnecessary duplication.