Technical Report: Google OAuth Implementation in React Native with Appwrite

1. Executive Summary

Overview

This report documents the comprehensive implementation of Google OAuth authentication in a React Native application that already had functional email/password authentication through Appwrite. The project successfully integrated Google Sign-In capabilities alongside the existing authentication system, providing users with multiple authentication options while maintaining a unified user experience.

Technology Stack Additions

- **OAuth Provider**: Google Cloud Console OAuth 2.0
- **Deep Linking**: Expo AuthSession and WebBrowser APIs
- **URL Schemes**: Custom app URL schemes for OAuth callbacks
- **Identity Management**: Appwrite Identities API for OAuth token management
- **Profile Data**: Google OAuth 2.0 User Info API

Timeline and Major Milestones

1. **Initial Setup (Day 1)**: Google Cloud Console configuration and Appwrite OAuth provider setup
2. **Basic Integration (Day 2)**: Google Sign-In button creation and initial OAuth flow implementation
3. **Deep Linking Setup (Day 3)**: Custom URL scheme configuration and OAuth callback handling
4. **Authentication Integration (Day 4)**: Merging OAuth with existing AuthContext
5. **Profile Enhancement (Day 5)**: Google profile picture and user data integration
6. **Token Management (Day 6)**: OAuth token refresh implementation

Key Challenges Overcome

- **OAuth Callback Complexity**: Implementing proper deep linking and redirect URI handling
- **Appwrite OAuth Integration**: Understanding Appwrite's OAuth token vs session distinction
- **Profile Data Access**: Discovering that OAuth access tokens are stored in Identities API, not sessions
- **Platform Configuration**: Setting up proper URL schemes and OAuth client configurations

- **State Management**: Unifying OAuth and email/password authentication flows

2. Initial Architecture Design

Original OAuth Flow Design

The initial design planned a straightforward integration where Google OAuth would complement the existing email/password authentication:

```
User Interaction → Google Sign-In Button → OAuth Flow → Appwrite Session → App Authenticatio
```

Planned Integration Strategy

```
// Expected AuthContext extension
interface AuthContextType {
  signin: (email: string, password: string) => Promise<void>;
  signup: (email: string, password: string, name: string) => Promise<void>;
  signInWithGoogle: () => Promise<void>; // New OAuth method
  signout: () => Promise<void>;
  user: any;
}
```

Initial Assumptions

1. **Appwrite OAuth Integration**: Assumed Appwrite would handle Google OAuth similarly to email/password authentication
2. **Token Management**: Expected OAuth access tokens to be directly available in session objects
3. **Profile Data**: Assumed user profile data would be automatically populated from Google
4. **Platform Uniformity**: Expected identical behavior across iOS and Android platforms

Expected Google Sign-In SDK Behavior

- Direct integration with Appwrite's OAuth 2.0 flow
- Automatic session creation after successful Google authentication
- Built-in profile data synchronization
- Seamless deep linking without custom URL scheme configuration

3. Implementation Journey - Chronological Deep Dive

Phase A: Initial Setup and Configuration

Google Cloud Console Project Setup **Step 1: OAuth Consent Screen Configuration**

```
// OAuth Consent Screen Settings
{
  "applicationName": "Tutorial Movie App",
  "userSupportEmail": "developer@example.com",
  "authorizedDomains": ["appwrite.io"],
  "scopes": [
    "https://www.googleapis.com/auth/userinfo.email",
    "https://www.googleapis.com/auth/userinfo.profile"
  ]
}
```

Why These Scopes Were Chosen: - `userinfo.email`: Required for user identification and account linking - `userinfo.profile`: Needed for profile picture and display name access

Step 2: OAuth Client ID Generation Initially attempted to create a single web application client:

```
// Initial (Incorrect) Configuration
{
  "clientType": "web",
  "authorizedRedirectURIs": ["http://localhost:3000/oauth"],
  "authorizedJavaScriptOrigins": ["http://localhost:3000"]
}
```

Why This Failed: React Native apps don't run in web browsers, so web client credentials don't work with native OAuth flows.

Corrected Configuration:

```
// Final Working Configuration
{
  "clientType": "web", // Still web type, but configured for Appwrite
  "authorizedRedirectURIs": [
    "appwrite-callback-68691394001a2a85ecc5://oauth",
    "exp://localhost:8081" // For Expo development
  ],
  "authorizedDomains": ["appwrite.io"]
}
```

Appwrite OAuth Provider Configuration Step 1: Enable Google OAuth Provider

```
# Appwrite Console Configuration
OAuth Provider: Google
Client ID: [Google Client ID from Cloud Console]
Client Secret: [Google Client Secret]
```

Step 2: Environment Variables Setup

```
# .env configuration
EXPO_PUBLIC_APPWRITE_PROJECT_ID=68691394001a2a85ecc5
EXPO_PUBLIC_APPWRITE_ENDPOINT=https://fra.cloud.appwrite.io/v1
EXPO_PUBLIC_GOOGLE_OAUTH_URL=https://fra.cloud.appwrite.io/v1/account/sessions/oauth2/callba
```

Platform-Specific Configuration app.json Configuration:

```
{
  "expo": {
    "name": "tutorial_movie_app",
    "scheme": "appwrite-callback-68691394001a2a85ecc5",
    // Other configuration...
  }
}
```

Why Custom URL Scheme Was Necessary: OAuth requires a way to redirect back to the app after authentication. Custom URL schemes allow the operating system to recognize and open the app when the OAuth provider redirects to the specified URI.

Phase B: OAuth Flow Implementation

Initial Google Sign-In Button Implementation First Attempt (Failed Approach):

```
// components/GoogleSignInButton.tsx - Initial Version
import { TouchableOpacity, Text } from 'react-native';

const GoogleSignInButton = ({ onPress }: { onPress: () => void }) => {
  return (
    <TouchableOpacity onPress={onPress} style={styles.button}>
      <Text>Sign in with Google</Text>
    </TouchableOpacity>
  );
};
```

Why This Failed: - No visual Google branding - No proper OAuth integration - Missing proper loading states

Final Working Implementation:

```
// components/GoogleSignInButton.tsx - Final Version
import { TouchableOpacity, Text, Image, ActivityIndicator } from 'react-native';

interface GoogleSignInButtonProps {
  onPress: () => void;
  disabled?: boolean;
}
```

```
export default function GoogleSignInButton({ onPress, disabled = false }: GoogleSignInButto
  const [isLoading, setIsLoading] = useState(false);

  const handlePress = async () => {
    try {
      setIsLoading(true);
      await onPress();
    } catch (error) {
      console.log('GoogleSignInButton: Error:', error);
    } finally {
      setIsLoading(false);
    }
  };

  return (
    <TouchableOpacity
      onPress={handlePress}
      disabled={disabled || isLoading}
      className="flex-row items-center justify-center bg-white border border-gray-400 rounde
    >
      {isLoading ? (
        <ActivityIndicator size="small" color="#666" className="mr-3" />
      ) : (
        <Image source={require('@/assets/images/google2.png')} className="w-6 h-6 mr-3" />
      )}
      <Text className="text-gray-800 font-medium text-base">
        {isLoading ? 'Signing in...' : 'Sign in with Google'}
      </Text>
    </TouchableOpacity>
  );
}
```

Initial OAuth Integration with Appwrite First Attempt (Using createOAuth2Session):

```
// services/appwrite.ts - Initial Failed Approach
export const signInWithGoogle = async () => {
  try {
    console.log("Attempting Google OAuth with createOAuth2Session...");

    // This approach failed - createOAuth2Session doesn't work with React Native
    const session = await account.createOAuth2Session(
      OAuthProvider.Google,
      'http://localhost:3000/success',
      'http://localhost:3000/failure'
    );
    return session;
  } catch (error) {
    console.log("OAuth error:", error);
    throw error;
  }
}
```

Why This Failed: - `createOAuth2Session` is designed for web environments - React Native apps can't handle HTTP redirects to localhost - No proper deep linking integration

Second Attempt (Using Custom Schemes):

```
// services/appwrite.ts - Second Attempt
export const signInWithGoogle = async () => {
  try {
    const redirectUri = 'movies://oauth/success'; // Custom scheme

    const session = await account.createOAuth2Session(
      OAuthProvider.Google,
      redirectUri,
      'movies://oauth/failure'
    );

    return session;
  } catch (error) {
    console.log("OAuth error:", error);
    throw error;
  }
}
```

Why This Also Failed: - Appwrite rejected custom schemes as invalid redirect URIs - `createOAuth2Session` still not appropriate for React Native

Final Working Implementation (Using createOAuth2Token):

```
// services/appwrite.ts - Final Working Version
export const signInWithGoogle = async () => {
  try {
    console.log("Initiating Google OAuth following Appwrite documentation...");

    // Create deep link that works across Expo environments
    const deepLink = new URL(makeRedirectUri({ preferLocalhost: true }));
    const scheme = `${deepLink.protocol}//`;

    console.log("Deep link:", deepLink.toString());
    console.log("Scheme:", scheme);
```

```
// Use createOAuth2Token for React Native
const loginUrl = await account.createOAuth2Token(
  OAuthProvider.Google,
  `${deepLink}`, // success URL
  `${deepLink}`  // failure URL
);

console.log("Opening login URL:", loginUrl.toString());

// Open login URL in browser
const result = await WebBrowser.openAuthSessionAsync(
  `${loginUrl}`,
  scheme,
  {
    showTitle: false,
    toolbarColor: '#1a1a1a',
    controlsColor: '#ffffff',
  }
);

console.log("WebBrowser result:", result);

if (result.type === 'cancel') {
  throw new Error('OAuth cancelled by user');
}

if (result.type === 'success' && result.url) {
  console.log("Processing OAuth success URL:", result.url);

  // Extract credentials from OAuth redirect URL
  const url = new URL(result.url);
  const secret = url.searchParams.get('secret');
  const userId = url.searchParams.get('userId');

  console.log("OAuth parameters:", { userId: !!userId, secret: !!secret });

  if (userId && secret) {
    // Create session with OAuth credentials
    console.log("Creating session with OAuth credentials...");
    const session = await account.createSession(userId, secret);
    console.log("Session created successfully:", !!session);

    // Get the current user to confirm authentication
    const currentUser = await getCurrentUser();
    if (currentUser) {
      console.log("OAuth successful, user session found");
```

```
          return { type: 'success', user: currentUser };
        } else {
          throw new Error('Session created but user not found');
        }
      } else {
        throw new Error('OAuth parameters not found in redirect URL');
      }
    } else {
      throw new Error('OAuth flow did not complete successfully');
    }

  } catch (error) {
    console.log("Google OAuth error:", error);
    throw error;
  }
}
```

Why This Finally Worked: 1. **Proper Method**: `createOAuth2Token` is designed for React Native/mobile apps 2. **Deep Linking**: `makeRedirectUri()` generates proper redirect URLs for Expo 3. **Manual Session Creation**: Manually creates session with extracted OAuth tokens 4. **Error Handling**: Comprehensive error handling for each step of the process

Phase C: State Management Integration

AuthContext Integration Challenge   Initial Integration Attempt:

```
// services/AuthContext.tsx - Initial Failed Integration
const AuthProvider = ({ children }: { children: React.ReactNode }) => {
  const [user, setUser] = useState(null);

  const googleSignIn = async () => {
    try {
      console.log("AuthContext: Attempting Google sign in...");
      const result = await signInWithGoogle();

      // This didn't work - result structure was inconsistent
      setUser(result.user);
    } catch (error) {
      console.log("AuthContext: Google sign in error:", error);
      throw error;
    }
  };

  // Rest of context...
}
```

Problems with Initial Approach: - Inconsistent user state management between OAuth and email/password - No proper session verification after OAuth - Missing profile data integration

Final Working Integration:

```
// services/AuthContext.tsx - Final Working Version
const AuthProvider = ({ children }: { children: React.ReactNode }) => {
  const [isLoading, setIsLoading] = useState(true);
  const [user, setUser] = useState(null);
  const [userProfile, setUserProfile] = useState(null);

  const checkCurrentUser = async () => {
    try {
      console.log("AuthContext: Checking for existing user...");
      setIsLoading(true);

      // First check and refresh OAuth tokens if needed
      const tokenWasRefreshed = await checkAndRefreshOAuthTokens();
      if (tokenWasRefreshed) {
        console.log("AuthContext: OAuth tokens were refreshed");
      }

      const currentUser = await getCurrentUser();
      console.log("AuthContext: getCurrentUser result:", {
        user: !!currentUser,
        email: currentUser?.email
      });

      if (currentUser) {
        setUser(currentUser);
        console.log("AuthContext: User found, fetching profile info...");

        // Fetch enhanced profile info (includes OAuth data)
        await fetchUserProfile(currentUser);

        console.log("AuthContext: User found, setting as logged in");
      } else {
        setUser(null);
        setUserProfile(null);
        console.log("AuthContext: No user found, setting as logged out");
      }
    } catch (error) {
      console.log("AuthContext: Error checking user:", error);
      setUser(null);
      setUserProfile(null);
    } finally {
      console.log("AuthContext: Setting isLoading to false");
      setIsLoading(false);
    }
  };

  const googleSignIn = async () => {
    try {
      console.log("AuthContext: Attempting Google sign in...");
      const result = await signInWithGoogle();
      console.log("AuthContext: Google OAuth result:", result);

      // Refresh the auth state to detect the new session and profile
      console.log("AuthContext: OAuth completed, refreshing auth state");
      await checkCurrentUser();
      console.log("AuthContext: Auth state refreshed after Google OAuth");
    } catch (error) {
      console.log("AuthContext: Google sign in error:", error);
      setUser(null);
      throw error;
    }
  };

  // ... rest of context
}
```

Profile Data Management Challenge **The Profile Data Problem:** After successful OAuth authentication, we discovered that while user authentication worked, the Google profile picture wasn't loading. This led to a deep investigation into how Appwrite handles OAuth data.

Initial Profile Fetching Attempt:

```
// services/AuthContext.tsx - Initial Failed Profile Fetching
const fetchUserProfile = async (currentUser: any) => {
  try {
    const profile = {
      name: currentUser.name || 'User',
      email: currentUser.email || '',
      avatar: undefined
    };

    // Try to get session info to check for OAuth provider
    const session = await getCurrentSession();

    if (session && session.provider === 'google' && session.providerAccessToken) {
      console.log("AuthContext: Fetching Google profile info...");
      const googleProfile = await getGoogleProfile(session.providerAccessToken);

      if (googleProfile) {
        profile.name = googleProfile.name || profile.name;
        profile.avatar = googleProfile.picture;
      }
    }

    setUserProfile(profile);
  } catch (error) {
    console.log("AuthContext: Error fetching user profile:", error);
    setUserProfile({
      name: currentUser.name || 'User',
      email: currentUser.email || '',
      avatar: undefined
    });
  }
};
```

Why This Failed: The session object showed:

```
{
  "provider": "oauth2",
  "providerUid": "",
  "providerAccessToken": "",
  "providerAccessTokenExpiry": ""
}
```

All OAuth token fields were empty strings, making it impossible to fetch Google profile data.

Discovery: Identities API Solution Through debugging, we discovered that OAuth access tokens are stored in the Identities API, not in session objects:

```
// services/appwrite.ts - Identity-based Profile Fetching
export const getUserIdentities = async () => {
  try {
    console.log("appwrite.ts: Fetching user identities...");
    const identities = await account.listIdentities();
    console.log("appwrite.ts: User identities:", JSON.stringify(identities, null, 2));
    return identities;
  } catch (error) {
    console.log("appwrite.ts: Error fetching identities:", error);
    return null;
  }
}
```

Final Working Profile Solution:

```
// services/AuthContext.tsx - Final Working Profile Fetching
const fetchUserProfile = async (currentUser: any) => {
  try {
    const profile = {
      name: currentUser.name || 'User',
      email: currentUser.email || '',
      avatar: undefined as string | undefined
    };

    const session = await getCurrentSession();

    if (session && session.provider === 'oauth2') {
      console.log("AuthContext: OAuth session detected, checking user identities...");
      const identities = await getUserIdentities();

      if (identities && identities.identities) {
        const googleIdentity = identities.identities.find((identity: any) =>
          identity.provider?.toLowerCase().includes('google')
        );

        if (googleIdentity) {
          console.log("AuthContext: Found Google identity:", {
            provider: googleIdentity.provider,
            hasAccessToken: !!googleIdentity.providerAccessToken,
            tokenExpiry: googleIdentity.providerAccessTokenExpiry
          });

          // Use the access token from identity to fetch Google profile
          if (googleIdentity.providerAccessToken) {
            console.log("AuthContext: Using Google identity access token to fetch profile...
            const googleProfile = await getGoogleProfile(googleIdentity.providerAccessToken)

            if (googleProfile) {
              profile.name = googleProfile.name || profile.name;
              profile.avatar = googleProfile.picture;
              console.log("AuthContext: Google profile fetched successfully from identity t
                hasName: !!googleProfile.name,
                hasAvatar: !!googleProfile.picture
              });
            }
          }
        }
      }
    }

    console.log("AuthContext: Setting user profile:", {
      hasName: !!profile.name,
      hasEmail: !!profile.email,
      hasAvatar: !!profile.avatar
    });

    setUserProfile(profile);
  } catch (error) {
    console.log("AuthContext: Error fetching user profile:", error);
    setUserProfile({
      name: currentUser.name || 'User',
      email: currentUser.email || '',
      avatar: undefined
    });
  }
};
```

4. Error Analysis and Troubleshooting

Major OAuth-Specific Errors Encountered

Error 1: Invalid Redirect URI Configuration Complete Error Message:

```
Google OAuth Error: Invalid redirect URI for OAuth success.
Error Code: OAUTH_INVALID_REDIRECT_URI
Details: The redirect URI 'movies://oauth/success' is not in the list of authorized redirect
```

Code That Caused This Error:

```
// Failed attempt using custom scheme
const redirectUri = 'movies://oauth/success';
const session = await account.createOAuth2Session(
  OAuthProvider.Google,
  redirectUri,
  'movies://oauth/failure'
);
```

Why This Occurred: - Custom URI schemes must be properly configured in Google Cloud Console - Appwrite has specific requirements for OAuth redirect URIs - The URI format didn't match Google's expected patterns

Failed Solution Attempts: 1. **Added custom scheme to Google Console**: Still rejected by Appwrite 2. **Used different URI formats**: `exp://`, `appwrite://`, all failed 3. **Tried localhost URLs**: Caused "can't connect to server" errors

Final Working Solution:

```
// Use Expo's makeRedirectUri with proper deep link configuration
const deepLink = new URL(makeRedirectUri({ preferLocalhost: true }));
const scheme = `${deepLink.protocol}//`;

const loginUrl = await account.createOAuth2Token(
  OAuthProvider.Google,
  `${deepLink}`,
  `${deepLink}`
);
```

Error 2: OAuth Method Incompatibility **Complete Error Message:**

```
TypeError: Cannot read property 'createOAuth2Session' of undefined
Platform: React Native
Context: Attempting to use web-based OAuth methods in React Native environment
```

Code That Caused This Error:

```
// Trying to use web-based OAuth method in React Native
const session = await account.createOAuth2Session(
  OAuthProvider.Google,
  successUrl,
  failureUrl
);
```

Why This Occurred: - `createOAuth2Session` is designed for web environments where redirects work differently - React Native apps can't handle server-side redirects the same way browsers can - The method expects a browser environment with automatic redirect handling

Debugging Steps Taken: 1. **Checked Appwrite Documentation**: Found React Native-specific OAuth instructions 2. **Tested Different Methods**: Tried various Appwrite OAuth methods 3. **Examined Network Requests**: OAuth flow was starting but not completing properly

Final Solution:

```
// Use React Native-appropriate OAuth method
const loginUrl = await account.createOAuth2Token(
  OAuthProvider.Google,
  `${deepLink}`,
  `${deepLink}`
);

// Handle the OAuth flow manually
const result = await WebBrowser.openAuthSessionAsync(`${loginUrl}`, scheme);
const url = new URL(result.url);
const secret = url.searchParams.get('secret');
const userId = url.searchParams.get('userId');
await account.createSession(userId, secret);
```

Error 3: Access Token Unavailability **Complete Error Message:**

```
Profile fetch failed: Access token not available in session
Session data: {
  "provider": "oauth2",
  "providerAccessToken": "",
  "providerAccessTokenExpiry": ""
}
```

Code That Caused This Error:

```
// Attempting to get OAuth access token from session
const session = await getCurrentSession();
if (session.providerAccessToken) {
  const profile = await getGoogleProfile(session.providerAccessToken);
}
```

Why This Occurred: - Appwrite's `createOAuth2Token` method doesn't populate session OAuth fields - Access tokens are stored separately in the Identities API - Session objects contain different data than expected

Failed Solution Attempts: 1. **Different session methods**: Tried various ways to access session data 2. **Token refresh attempts**: Tried refreshing sessions to populate tokens 3. **Direct API calls**: Attempted to access Appwrite's internal token storage

Discovery Process:

```
// Debugging revealed identities contain the tokens
const identities = await account.listIdentities();
console.log("Identities:", JSON.stringify(identities, null, 2));

// Output showed:
{
  "identities": [{
    "provider": "google",
    "providerAccessToken": "ya29.A0AS3H6Ny...", // Token was here!
    "providerAccessTokenExpiry": "2025-08-22T15:57:35.719+00:00"
  }]
}
```

Final Working Solution:

```
// Get OAuth tokens from identities instead of session
const identities = await getUserIdentities();
const googleIdentity = identities.identities.find(
  identity => identity.provider?.toLowerCase().includes('google')
);

if (googleIdentity && googleIdentity.providerAccessToken) {
  const googleProfile = await getGoogleProfile(googleIdentity.providerAccessToken);
  profile.avatar = googleProfile.picture;
}
```

5. Solution Evolution

Final OAuth Architecture

The final working OAuth implementation follows this flow:

```
User Action → Google Sign-In Button → OAuth URL Generation →
Browser OAuth Flow → Deep Link Callback → Token Extraction →
Session Creation → Profile Enhancement → Authentication Complete
```

Complete Working OAuth Integration

Google Sign-In Button (Final Version)

```
// components/GoogleSignInButton.tsx
import { TouchableOpacity, Text, Image, ActivityIndicator } from 'react-native';
import { useState } from 'react';

interface GoogleSignInButtonProps {
  onPress: () => void;
  disabled?: boolean;
}

export default function GoogleSignInButton({ onPress, disabled = false }: GoogleSignInButtor
  const [isLoading, setIsLoading] = useState(false);

  const handlePress = async () => {
    if (disabled || isLoading) return;

    try {
      console.log("GoogleSignInButton: Button pressed, disabled:", disabled);
      setIsLoading(true);
      await onPress();
    } catch (error) {
      console.log('GoogleSignInButton: Error during sign-in:', error);
    } finally {
      setIsLoading(false);
    }
  };

  return (
    <TouchableOpacity
      onPress={handlePress}
      disabled={disabled || isLoading}
      className="flex-row items-center justify-center bg-white border border-gray-400 rounde
    >
      {isLoading ? (
        <ActivityIndicator size="small" color="#666" className="mr-3" />
      ) : (
        <Image source={require('@/assets/images/google2.png')} className="w-6 h-6 mr-3" />
      )}
      <Text className="text-gray-800 font-medium text-base">
        {isLoading ? 'Signing in...' : 'Sign in with Google'}
      </Text>
    </TouchableOpacity>
  );
}
```

OAuth Service Functions (Final Version)

```
// services/appwrite.ts
import { Account, Client, Databases, ID, Query, OAuthProvider } from "react-native-appwrite"
import { Platform, Linking } from "react-native";
import * as WebBrowser from 'expo-web-browser';
import { makeRedirectUri } from 'expo-auth-session';

// OAuth Implementation
export const signInWithGoogle = async () => {
  try {
    console.log("appwrite.ts: Initiating Google OAuth following Appwrite documentation...")

    // Create deep link that works across Expo environments
    const deepLink = new URL(makeRedirectUri({ preferLocalhost: true }));
    const scheme = `${deepLink.protocol}//`;

    console.log("appwrite.ts: Deep link:", deepLink.toString());
    console.log("appwrite.ts: Scheme:", scheme);

    // Start OAuth flow
    const loginUrl = await account.createOAuth2Token(
      OAuthProvider.Google,
      `${deepLink}`,
      `${deepLink}`
    );

    console.log("appwrite.ts: Opening login URL:", loginUrl.toString());

    // Open login URL in browser
    const result = await WebBrowser.openAuthSessionAsync(
      `${loginUrl}`,
      scheme,
      {
        showTitle: false,
        toolbarColor: '#1a1a1a',
        controlsColor: '#ffffff',
      }
    );

    console.log("appwrite.ts: WebBrowser result:", result);

    if (result.type === 'cancel') {
      throw new Error('OAuth cancelled by user');
    }

    if (result.type === 'success' && result.url) {
      console.log("appwrite.ts: Processing OAuth success URL:", result.url);

      // Extract credentials from OAuth redirect URL
      const url = new URL(result.url);
      const secret = url.searchParams.get('secret');
      const userId = url.searchParams.get('userId');

      console.log("appwrite.ts: OAuth parameters:", { userId: !!userId, secret: !!secret })

      if (userId && secret) {
        // Create session with OAuth credentials
        console.log("appwrite.ts: Creating session with OAuth credentials...");
        const session = await account.createSession(userId, secret);
        console.log("appwrite.ts: Session created successfully:", !!session);

        // Get the current user to confirm authentication
        const currentUser = await getCurrentUser();
        if (currentUser) {
          console.log("appwrite.ts: OAuth successful, user session found");
          return { type: 'success', user: currentUser };
        } else {
          throw new Error('Session created but user not found');
        }
      } else {
        throw new Error('OAuth parameters not found in redirect URL');
      }
    } else {
      throw new Error('OAuth flow did not complete successfully');
    }
```

```
  } catch (error) {
    console.log("appwrite.ts: Google OAuth error:", error);
    throw error;
  }
}

// Profile Data Retrieval
export const getUserIdentities = async () => {
  try {
    console.log("appwrite.ts: Fetching user identities...");
    const identities = await account.listIdentities();
    return identities;
  } catch (error) {
    console.log("appwrite.ts: Error fetching identities:", error);
    return null;
  }
}

export const getGoogleProfile = async (accessToken: string) => {
  try {
    console.log("appwrite.ts: Fetching Google profile...");
    const response = await fetch(`https://www.googleapis.com/oauth2/v2/userinfo?access_toke
    const profile = await response.json();
    console.log("appwrite.ts: Google profile:", {
      hasName: !!profile?.name,
      hasAvatar: !!profile?.picture,
      email: profile?.email
    });
    return profile;
  } catch (error) {
    console.log("appwrite.ts: Error fetching Google profile:", error);
    return null;
  }
}

// Token Management
export const isTokenExpired = (expiryDateString: string): boolean => {
  if (!expiryDateString) return true;

  const expiryDate = new Date(expiryDateString);
  const now = new Date();
  const fiveMinutesFromNow = new Date(now.getTime() + 5 * 60 * 1000);

  return expiryDate <= fiveMinutesFromNow;
}
```

```
export const refreshOAuthSession = async (sessionId: string = 'current') => {
  try {
    console.log("appwrite.ts: Refreshing OAuth session...");
    const updatedSession = await account.updateSession(sessionId);
    console.log("appwrite.ts: OAuth session refreshed successfully");
    return updatedSession;
  } catch (error) {
    console.log("appwrite.ts: Error refreshing OAuth session:", error);
    throw error;
  }
}

export const checkAndRefreshOAuthTokens = async () => {
  try {
    console.log("appwrite.ts: Checking OAuth token expiry...");
    const identities = await getUserIdentities();

    if (identities && identities.identities) {
      const googleIdentity = identities.identities.find((identity: any) =>
        identity.provider?.toLowerCase().includes('google')
      );

      if (googleIdentity && googleIdentity.providerAccessTokenExpiry) {
        const isExpired = isTokenExpired(googleIdentity.providerAccessTokenExpiry);
        console.log("appwrite.ts: Token status:", {
          expiry: googleIdentity.providerAccessTokenExpiry,
          isExpired,
          timeUntilExpiry: new Date(googleIdentity.providerAccessTokenExpiry).getTime() - Da
        });

        if (isExpired) {
          console.log("appwrite.ts: Token expired or expiring soon, refreshing...");
          await refreshOAuthSession();
          console.log("appwrite.ts: Token refresh completed");
          return true;
        }
      }
    }

    return false;
  } catch (error) {
    console.log("appwrite.ts: Error checking/refreshing OAuth tokens:", error);
    return false;
  }
}
```

Unified AuthContext Integration (Final Version)

```
// services/AuthContext.tsx
import { createContext, useContext, useState, useEffect } from "react";
import {
  createAccount,
  signIn,
  getCurrentUser,
  signOut,
  signInWithGoogle,
  getUserIdentities,
  getGoogleProfile,
  checkAndRefreshOAuthTokens
} from "./appwrite";

export interface AuthContextType {
  signin: (email: string, password: string) => Promise<void>;
  signup: (email: string, password: string, name: string) => Promise<void>;
  signInWithGoogle: () => Promise<void>;
  signout: () => Promise<void>;
  refreshAuthState: () => Promise<void>;
  isLoading: boolean;
  user: any;
  userProfile: {
    name: string;
    email: string;
    avatar?: string;
  } | null;
}

const AuthContext = createContext<AuthContextType | undefined>(undefined);

const AuthProvider = ({ children }: { children: React.ReactNode }) => {
  const [isLoading, setIsLoading] = useState(true);
  const [user, setUser] = useState(null);
  const [userProfile, setUserProfile] = useState(null);

  useEffect(() => {
    checkCurrentUser();
  }, []);

  const checkCurrentUser = async () => {
    try {
      console.log("AuthContext: Checking for existing user...");
      setIsLoading(true);
      // First check and refresh OAuth tokens if needed
      const tokenWasRefreshed = await checkAndRefreshOAuthTokens();
      if (tokenWasRefreshed) {
        console.log("AuthContext: OAuth tokens were refreshed");
      }

      const currentUser = await getCurrentUser();
      console.log("AuthContext: getCurrentUser result:", {
        user: !!currentUser,
        email: currentUser?.email
      });

      if (currentUser) {
        setUser(currentUser);
        console.log("AuthContext: User found, fetching profile info...");

        await fetchUserProfile(currentUser);

        console.log("AuthContext: User found, setting as logged in");
      } else {
        setUser(null);
        setUserProfile(null);
        console.log("AuthContext: No user found, setting as logged out");
      }
    } catch (error) {
      console.log("AuthContext: Error checking user:", error);
      setUser(null);
      setUserProfile(null);
    } finally {
      console.log("AuthContext: Setting isLoading to false");
      setIsLoading(false);
    }
  };

  const fetchUserProfile = async (currentUser: any) => {
    try {
      const profile = {
        name: currentUser.name || 'User',
        email: currentUser.email || '',
        avatar: undefined as string | undefined
      };

      // Check for OAuth session and get enhanced profile
      const identities = await getUserIdentities();

      if (identities && identities.identities) {
      const googleIdentity = identities.identities.find((identity: any) =>
        identity.provider?.toLowerCase().includes('google')
      );

      if (googleIdentity && googleIdentity.providerAccessToken) {
        console.log("AuthContext: Using Google identity access token to fetch profile...")
        const googleProfile = await getGoogleProfile(googleIdentity.providerAccessToken);

        if (googleProfile) {
          profile.name = googleProfile.name || profile.name;
          profile.avatar = googleProfile.picture;
          console.log("AuthContext: Google profile fetched successfully from identity toke
            hasName: !!googleProfile.name,
            hasAvatar: !!googleProfile.picture
          });
        }
      }
    }

    console.log("AuthContext: Setting user profile:", {
      hasName: !!profile.name,
      hasEmail: !!profile.email,
      hasAvatar: !!profile.avatar
    });

    setUserProfile(profile);
  } catch (error) {
    console.log("AuthContext: Error fetching user profile:", error);
    setUserProfile({
      name: currentUser.name || 'User',
      email: currentUser.email || '',
      avatar: undefined
    });
  }
};

const refreshAuthState = checkCurrentUser;

const signin = async (email: string, password: string) => {
  try {
    console.log("AuthContext: Attempting signin...");
    const session = await signIn(email, password);
    console.log("AuthContext: signIn successful, session created:", !!session);

    console.log("AuthContext: Refreshing auth state after successful signin");
    await checkCurrentUser();
    console.log("AuthContext: Signin complete, auth state refreshed");
  } catch (error) {
    console.log("AuthContext: Sign in error:", error);
    setUser(null);
    throw error;
  }
};

const signup = async (email: string, password: string, name: string) => {
  try {
    console.log("AuthContext: Attempting signup...");
    await createAccount(email, password, name);
    console.log("AuthContext: Account created, signing in...");
    await signIn(email, password);
    console.log("AuthContext: Signed in after signup, refreshing auth state");
    await checkCurrentUser();
    console.log("AuthContext: Signup complete, auth state refreshed");
  } catch (error) {
    console.log("AuthContext: Sign up error:", error);
    setUser(null);
    throw error;
  }
};

const googleSignIn = async () => {
  try {
    console.log("AuthContext: Attempting Google sign in...");
    const result = await signInWithGoogle();
    console.log("AuthContext: Google OAuth result:", result);

    console.log("AuthContext: OAuth completed, refreshing auth state");
    await checkCurrentUser();
    console.log("AuthContext: Auth state refreshed after Google OAuth");
  } catch (error) {
    console.log("AuthContext: Google sign in error:", error);
    setUser(null);
    throw error;
  }
};

const logout = async () => {
  setIsLoading(true);
  try {
    console.log("AuthContext: Attempting signout...");
    await signOut();
```

```
      console.log("AuthContext: Signout successful");
      setUser(null);
      setUserProfile(null);
    } catch (error) {
      console.log("AuthContext: Sign out error:", error);
      setUser(null);
      setUserProfile(null);
    } finally {
      setIsLoading(false);
    }
  };

  const contextData = {
    signin,
    signup,
    signInWithGoogle: googleSignIn,
    signout: logout,
    refreshAuthState,
    isLoading,
    user,
    userProfile
  };

  return (
    <AuthContext.Provider value={contextData}>
      {children}
    </AuthContext.Provider>
  );
};

const useAuth = () => {
  const context = useContext(AuthContext);
  if (context === undefined) {
    throw new Error('useAuth must be used within an AuthProvider');
  }
  return context;
};

export { AuthContext, AuthProvider, useAuth };
```

Profile UI Integration (Final Version)

```
// app/(tabs)/profile.tsx
import { View, Text, TouchableOpacity, Alert, Image } from 'react-native';
import React from 'react';
import { useAuth } from '../../services/AuthContext';
```

```
const Profile = () => {
  const { user, userProfile, signout } = useAuth();

  const handleLogout = async () => {
    try {
      console.log("Profile: Attempting logout...");
      await signout();
      console.log("Profile: Logout successful");
    } catch (error: any) {
      console.log("Profile: Logout error:", error);
      Alert.alert("Logout Failed", error.message);
    }
  };

  return (
    <View className="bg-primary flex-1 items-center justify-center px-4">
      {/* Profile Avatar */}
      {userProfile?.avatar ? (
        <Image
          source={{ uri: userProfile.avatar }}
          className="w-24 h-24 rounded-full mb-6"
          style={{ borderWidth: 3, borderColor: '#6B7280' }}
          onError={(error) => console.log("Profile: Image load error:", error.nativeEvent.er
          onLoad={() => console.log("Profile: Image loaded successfully")}
        />
      ) : (
        <View className="w-24 h-24 rounded-full bg-secondary mb-6 items-center justify-cente
          <Text className="text-primary text-2xl font-psemibold">
            {userProfile?.name?.charAt(0)?.toUpperCase() || 'U'}
          </Text>
        </View>
      )}

      {/* Greeting */}
      <Text className="text-white text-2xl font-psemibold mb-2">
        Hi {userProfile?.name || 'there'}!
      </Text>

      <Text className="text-gray-100 text-lg mb-8 text-center">
        {userProfile?.email || user?.email || 'No email available'}
      </Text>

      <Text className="text-gray-400 text-sm mb-8">
        Auth Status: {user ? 'Logged In' : 'Logged Out'}
      </Text>
      <TouchableOpacity
        onPress={handleLogout}
        className="w-full bg-red-500 rounded-xl min-h-[62px] justify-center items-center"
      >
        <Text className="text-white font-psemibold text-lg">
          Logout
        </Text>
      </TouchableOpacity>
    </View>
  );
};

export default Profile;
```

6. Appwrite-Specific OAuth Considerations

How Appwrite Handles OAuth Providers

Appwrite's OAuth implementation has several unique characteristics that differ from direct OAuth SDK integrations:

OAuth Provider Configuration

```
// Appwrite Console Settings
{
  "provider": "Google",
  "clientId": "[Google OAuth Client ID]",
  "clientSecret": "[Google OAuth Client Secret]",
  "enabled": true
}
```

Key Insights: - Appwrite acts as an OAuth intermediary, not just a backend - All OAuth flows go through Appwrite's OAuth endpoint - Client credentials are stored securely on Appwrite servers - No need to include Google client secrets in the React Native app

OAuth Session vs Regular Session Differences **Regular Email/Password Session:**

```
{
  "userId": "user123",
  "provider": "email",
  "factors": ["email"],
  "providerUid": "",
  "providerAccessToken": "",
  "providerRefreshToken": ""
}
```

OAuth Session:

```
{
  "userId": "user123",
  "provider": "oauth2",
  "factors": ["email"],
  "providerUid": "",
  "providerAccessToken": "",
  "providerRefreshToken": ""
}
```

Critical Discovery: OAuth sessions in Appwrite don't contain provider-specific tokens in the session object. These tokens are stored in the Identities API instead.

OAuth Token Management Architecture **Session Object (Limited OAuth Data):** - Contains basic session information - `provider` field shows "oauth2" for all OAuth providers - OAuth-specific tokens are NOT stored here

Identities API (Complete OAuth Data):

```
{
  "identities": [{
    "provider": "google",
    "providerUid": "107601867217453092770",
    "providerEmail": "user@gmail.com",
    "providerAccessToken": "ya29.A0AS3H6Ny...",
    "providerAccessTokenExpiry": "2025-08-22T15:57:35.719+00:00",
    "providerRefreshToken": ""
  }]
}
```

Undocumented Appwrite OAuth Behaviors

1. Token Storage Separation **Discovery:** OAuth access tokens are stored in Identities, not Sessions. **Impact:** Required completely different approach to accessing OAuth tokens. **Solution:** Always use `account.listIdentities()` for OAuth token access.

2. createOAuth2Token vs createOAuth2Session **Discovery:** `createOAuth2Session` doesn't work properly in React Native. **Impact:** Initial implementations failed completely. **Solution:** Use `createOAuth2Token` with manual session creation.

3. Provider Name Inconsistency **Discovery:** Session shows `provider: "oauth2"` instead of `provider: "google"`. **Impact:** Can't identify OAuth provider from session data. **Solution:** Check Identities API for specific provider information.

4. Token Refresh Behavior **Discovery:** `account.updateSession()` refreshes OAuth tokens in Identities. **Impact:** Token refresh affects Identities, not Session objects. **Solution:** Check token expiry in Identities and refresh via session update.

7. OAuth Authentication Logic Deep Dive

Complete OAuth Flow Walkthrough

Step 1: OAuth URL Generation

```
// Deep link creation for OAuth callback
const deepLink = new URL(makeRedirectUri({ preferLocalhost: true }));
// Result: "exp://localhost:8081" (in development)

const scheme = `${deepLink.protocol}//`;
// Result: "exp://"

// OAuth URL generation through Appwrite
const loginUrl = await account.createOAuth2Token(
  OAuthProvider.Google,
  `${deepLink}`, // success URL
  `${deepLink}`  // failure URL
);
// Result: "https://fra.cloud.appwrite.io/v1/account/tokens/oauth2/google?success=exp%3A%2F
```

Why This Approach: - `makeRedirectUri()` generates platform-appropriate URLs - Appwrite's `createOAuth2Token` creates mobile-compatible OAuth URLs - Single URL for both success and failure keeps configuration simple

Step 2: Browser-Based OAuth Authentication

```
// Open OAuth URL in system browser
const result = await WebBrowser.openAuthSessionAsync(
  `${loginUrl}`,
  scheme, // App will capture URLs starting with this scheme
  {
    showTitle: false,
    toolbarColor: '#1a1a1a',
    controlsColor: '#ffffff',
  }
);
```

```
// Expected result on success:
// {
//   type: 'success',
//   url: 'exp://localhost:8081/?secret=abc123&userId=user456#'
// }
```

Security Considerations: - Browser isolation prevents app from accessing Google credentials - User sees actual Google sign-in interface - OAuth tokens generated server-side by Google - App only receives final authorization tokens

Step 3: OAuth Callback Processing

```
// Extract OAuth credentials from callback URL
const url = new URL(result.url);
const secret = url.searchParams.get('secret');
const userId = url.searchParams.get('userId');

// Validate OAuth parameters
if (!userId || !secret) {
  throw new Error('OAuth parameters not found in redirect URL');
}

console.log("OAuth parameters:", {
  userId: !!userId,
  secret: !!secret,
  fullUrl: result.url
});
```

Why Manual Parameter Extraction: - Appwrite passes OAuth tokens via URL parameters - `secret` is used to create authenticated session - `userId` identifies the user account in Appwrite - Manual extraction provides better error handling

Step 4: Session Creation and Validation

```
// Create Appwrite session using OAuth tokens
const session = await account.createSession(userId, secret);
console.log("Session created successfully:", !!session);

// Verify user authentication
const currentUser = await getCurrentUser();
if (!currentUser) {
  throw new Error('Session created but user not found');
}

return { type: 'success', user: currentUser };
```

Session Creation Logic: - `createSession()` exchanges OAuth tokens for Appwrite session - Session validation ensures authentication actually worked - User object retrieval confirms account access

Step 5: Profile Enhancement

```
// Get OAuth provider information
const identities = await getUserIdentities();
const googleIdentity = identities.identities.find(
  identity => identity.provider?.toLowerCase().includes('google')
);

// Use OAuth access token to fetch Google profile
if (googleIdentity && googleIdentity.providerAccessToken) {
  const googleProfile = await getGoogleProfile(googleIdentity.providerAccessToken);

  // Enhance user profile with Google data
  profile.name = googleProfile.name || profile.name;
  profile.avatar = googleProfile.picture;
}
```

Profile Enhancement Rationale: - Appwrite user object has basic information - Google profile API provides additional data (avatar, full name) - OAuth access token required for Google API calls - Profile data cached in app state for performance

Error Handling Strategy

Network Failure Handling

```
const signInWithGoogle = async () => {
  try {
    // OAuth implementation
  } catch (error) {
    if (error.message.includes('network')) {
      throw new Error('Network error during sign-in. Please check your connection.');
    } else if (error.message.includes('cancelled')) {
      throw new Error('Sign-in was cancelled by user.');
    } else {
      console.log("Unexpected OAuth error:", error);
      throw new Error('Sign-in failed. Please try again.');
    }
  }
}
```

User Cancellation Handling

```
if (result.type === 'cancel') {
  throw new Error('OAuth cancelled by user');
}
```

Invalid Token Handling

```
// Check for OAuth token validity
if (!googleIdentity.providerAccessToken) {
  console.log("No OAuth access token available, using basic profile");
  return basicProfile;
}

// Validate token expiry
if (isTokenExpired(googleIdentity.providerAccessTokenExpiry)) {
  console.log("OAuth token expired, refreshing...");
  await refreshOAuthSession();
}
```

Why OAuth Security Decisions Were Made

1. Browser-Based Authentication **Decision:** Use system browser instead of WebView **Reasoning:** - Better security isolation - User sees real Google interface - Prevents phishing attacks - Follows OAuth 2.0 best practices

2. Server-Side Token Exchange **Decision:** Let Appwrite handle token exchange **Reasoning:** - Client secrets stay secure on server - Reduces app attack surface - Simplifies token management - Follows OAuth security guidelines

3. Automatic Token Refresh **Decision:** Proactively refresh tokens before expiry **Reasoning:** - Prevents sudden authentication failures - Better user experience - Reduces API call failures - Follows Appwrite recommendations

8. Performance Considerations in OAuth Implementation

OAuth Loading States and UX

Loading State Management

```
// GoogleSignInButton.tsx - Loading state implementation
const [isLoading, setIsLoading] = useState(false);

const handlePress = async () => {
  try {
    setIsLoading(true);
    await onPress();
  } catch (error) {
    console.log('GoogleSignInButton: Error during sign-in:', error);
  } finally {
    setIsLoading(false);
  }
};

// Visual loading indicator
{isLoading ? (
  <ActivityIndicator size="small" color="#666" className="mr-3" />
) : (
  <Image source={require('@/assets/images/google2.png')} className="w-6 h-6 mr-3" />
)}
```

Performance Impact: - Prevents multiple concurrent OAuth requests - Provides visual feedback during authentication - Reduces user confusion during OAuth flow

AuthContext Loading Management

```
// Global authentication loading state
const [isLoading, setIsLoading] = useState(true);

const checkCurrentUser = async () => {
  try {
    setIsLoading(true);

    // Token refresh check (minimal performance impact)
    const tokenWasRefreshed = await checkAndRefreshOAuthTokens();

    // User authentication check
    const currentUser = await getCurrentUser();

    if (currentUser) {
      // Profile enhancement (cached after first load)
      await fetchUserProfile(currentUser);
    }
  } finally {
    setIsLoading(false);
  }
};
```

OAuth Token Caching Strategy

Identity Data Caching

```
// Avoid repeated identity API calls
const fetchUserProfile = async (currentUser: any) => {
  try {
    // Check if we already have profile data
    if (userProfile && userProfile.avatar) {
      console.log("Profile already loaded, skipping API call");
      return;
    }

    // Only fetch identities when needed
    const identities = await getUserIdentities();

    // Cache OAuth access token for multiple profile API calls
    if (googleIdentity.providerAccessToken) {
      const googleProfile = await getGoogleProfile(googleIdentity.providerAccessToken);
      // Store profile data in state for subsequent renders
    }
  } catch (error) {
    // Fallback to cached data on error
  }
};
```

Network Request Optimization Parallel API Calls:

```
// Batch independent API calls
const [currentUser, identities] = await Promise.all([
  getCurrentUser(),
  getUserIdentities()
]);
```

Request Deduplication:

```
// Prevent duplicate profile fetches during rapid state changes
let profileFetchPromise = null;

const fetchUserProfile = async (currentUser: any) => {
  if (profileFetchPromise) {
    return await profileFetchPromise;
  }

  profileFetchPromise = performProfileFetch(currentUser);

  try {
    return await profileFetchPromise;
  } finally {
    profileFetchPromise = null;
  }
};
```

OAuth Redirect Handling Efficiency

Deep Link Processing Optimization

```
// Efficient URL parameter extraction
const processOAuthCallback = (callbackUrl: string) => {
  try {
    const url = new URL(callbackUrl);

    // Direct parameter access (faster than iteration)
    const secret = url.searchParams.get('secret');
    const userId = url.searchParams.get('userId');

    // Early validation to prevent unnecessary processing
    if (!userId || !secret) {
      throw new Error('Invalid OAuth callback parameters');
    }

    return { userId, secret };
  } catch (error) {
    // Fast failure path
    throw new Error('Invalid OAuth callback URL format');
  }
};
```

Memory Management

```
// Clean up WebBrowser resources
const result = await WebBrowser.openAuthSessionAsync(loginUrl, scheme);

// Process result immediately and release references
const oauthParams = processOAuthCallback(result.url);

// Clear result object to free memory
result = null;

return oauthParams;
```

Performance Monitoring

OAuth Flow Timing

```
const signInWithGoogle = async () => {
  const startTime = Date.now();

  try {
    console.log("OAuth flow started");
    // OAuth implementation...

    const endTime = Date.now();
    console.log(`OAuth flow completed in ${endTime - startTime}ms`);

  } catch (error) {
    const endTime = Date.now();
    console.log(`OAuth flow failed after ${endTime - startTime}ms:`, error);
    throw error;
  }
};
```

Profile Loading Performance

```
const fetchUserProfile = async (currentUser: any) => {
  const profileStart = Date.now();

  // Profile fetching logic...

  console.log(`Profile loaded in ${Date.now() - profileStart}ms`);
};
```

9. Testing and Validation Process

Platform-Specific Testing Strategy

iOS Testing Approach

```
// iOS-specific OAuth testing considerations
const testIOSOAuth = async () => {
  // Test 1: URL scheme handling
  console.log("Testing iOS URL scheme:", 'appwrite-callback-68691394001a2a85ecc5://');

  // Test 2: Deep link processing
  const testUrl = 'appwrite-callback-68691394001a2a85ecc5://oauth?secret=test&userId=test';
  const params = processOAuthCallback(testUrl);

  // Test 3: Browser integration
  // Verify Safari in-app browser opens correctly

  // Test 4: Profile picture loading
  // Check image caching and rendering
};
```

Android Testing Approach

```
// Android-specific OAuth considerations
const testAndroidOAuth = async () => {
  // Test 1: Custom scheme registration
  // Verify Android manifest includes scheme

  // Test 2: Chrome Custom Tabs integration
  // Test WebBrowser behavior on Android

  // Test 3: Back button handling
  // Ensure OAuth flow handles Android back button correctly

  // Test 4: Memory management
  // Monitor OAuth flow memory usage
};
```

OAuth Flow Testing Methodology

Happy Path Testing

```
const testSuccessfulOAuthFlow = async () => {
  console.log("Testing successful OAuth flow...");

  // Step 1: Verify OAuth URL generation
  const deepLink = new URL(makeRedirectUri({ preferLocalhost: true }));
  const loginUrl = await account.createOAuth2Token(OAuthProvider.Google, deepLink, deepLink

  assert(loginUrl.toString().includes('google'), 'OAuth URL should include Google provider'
  assert(loginUrl.toString().includes(deepLink.toString()), 'OAuth URL should include redire

  // Step 2: Mock successful browser callback
  const mockCallback = `${deepLink}?secret=test-secret&userId=test-user-id`;
  const params = processOAuthCallback(mockCallback);

  assert(params.userId === 'test-user-id', 'Should extract userId correctly');
  assert(params.secret === 'test-secret', 'Should extract secret correctly');

  // Step 3: Test session creation (mocked)
  // Step 4: Test profile enhancement

  console.log("Successful OAuth flow test passed");
};
```

Error Path Testing

```
const testOAuthErrorScenarios = async () => {
  console.log("Testing OAuth error scenarios...");
  // Test 1: User cancellation
  const cancelResult = { type: 'cancel' };
  try {
    await processOAuthResult(cancelResult);
    assert(false, 'Should throw error on cancellation');
  } catch (error) {
    assert(error.message.includes('cancelled'), 'Should indicate cancellation');
  }

  // Test 2: Invalid callback URL
  const invalidUrl = 'invalid-url-format';
  try {
    processOAuthCallback(invalidUrl);
    assert(false, 'Should throw error on invalid URL');
  } catch (error) {
    assert(error.message.includes('Invalid'), 'Should indicate invalid URL');
  }

  // Test 3: Missing OAuth parameters
  const missingParamsUrl = `${deepLink}?secret=test-secret`;
  try {
    processOAuthCallback(missingParamsUrl);
    assert(false, 'Should throw error on missing parameters');
  } catch (error) {
    assert(error.message.includes('parameters'), 'Should indicate missing parameters');
  }

  console.log("OAuth error scenario tests passed");
};
```

Multi-Account Testing

Different Google Accounts

```
const testMultipleGoogleAccounts = async () => {
  console.log("Testing multiple Google account scenarios...");

  // Test 1: Account A sign-in
  const accountAProfile = await testAccountSignIn('user.a@gmail.com');
  assert(accountAProfile.email === 'user.a@gmail.com', 'Account A should be correct');

  // Test 2: Sign out
  await signOut();

  // Test 3: Account B sign-in
  const accountBProfile = await testAccountSignIn('user.b@gmail.com');
  assert(accountBProfile.email === 'user.b@gmail.com', 'Account B should be correct');

  // Test 4: Profile data isolation
  assert(accountBProfile.avatar !== accountAProfile.avatar, 'Profiles should be different')

  console.log("Multiple account testing completed");
};
```

Edge Case Testing

Network Interruption Testing

```
const testNetworkInterruption = async () => {
  console.log("Testing network interruption scenarios...");

  // Test 1: OAuth URL generation during poor network
  const startTime = Date.now();
  try {
    const loginUrl = await account.createOAuth2Token(OAuthProvider.Google, redirectUri, redi
    const responseTime = Date.now() - startTime;
    console.log(`OAuth URL generated in ${responseTime}ms`);

    if (responseTime > 5000) {
      console.warn("Slow OAuth URL generation detected");
    }
  } catch (error) {
    console.log("Network error during OAuth URL generation:", error);
    // Test error handling
  }

  // Test 2: Profile picture loading during poor network
  const profileStart = Date.now();
  try {
    const googleProfile = await getGoogleProfile(mockAccessToken);
    console.log(`Profile loaded in ${Date.now() - profileStart}ms`);
  } catch (error) {
    console.log("Network error during profile loading:", error);
    // Verify graceful degradation
  }
};
```

Token Expiry Testing

```
const testTokenExpiry = async () => {
  console.log("Testing OAuth token expiry scenarios...");

  // Test 1: Expired token detection
  const expiredTokenDate = new Date(Date.now() - 60000).toISOString(); // 1 minute ago
  const isExpired = isTokenExpired(expiredTokenDate);
  assert(isExpired === true, 'Should detect expired token');

  // Test 2: Soon-to-expire token detection
  const soonToExpireDate = new Date(Date.now() + 240000).toISOString(); // 4 minutes from no
  const willExpireSoon = isTokenExpired(soonToExpireDate);
  assert(willExpireSoon === true, 'Should detect soon-to-expire token (5 min buffer)');

  // Test 3: Fresh token detection
  const freshTokenDate = new Date(Date.now() + 3600000).toISOString(); // 1 hour from now
  const isFresh = isTokenExpired(freshTokenDate);
  assert(isFresh === false, 'Should detect fresh token');

  console.log("Token expiry testing completed");
};
```

Debugging OAuth Issues

OAuth Flow Debugging Commands

```
# Clear Expo cache
npx expo start --clear

# Reset Metro bundler
npx expo start --reset-cache

# View detailed logs
npx expo start --dev-client --clear

# Check deep link registration
npx expo install --check

# Verify OAuth configuration
cat app.json | grep -A 5 -B 5 "scheme"
```

Common Debug Scenarios

```
const debugOAuthIssues = () => {
  console.log("=== OAuth Debug Information ===");

  // Debug 1: Environment verification
  console.log("Project ID:", process.env.EXPO_PUBLIC_APPWRITE_PROJECT_ID);
  console.log("Endpoint:", process.env.EXPO_PUBLIC_APPWRITE_ENDPOINT);

  // Debug 2: Deep link verification
  const redirectUri = makeRedirectUri({ preferLocalhost: true });
  console.log("Generated redirect URI:", redirectUri);

  // Debug 3: OAuth URL verification
  console.log("Expected OAuth URL pattern:",
    `https://[region].cloud.appwrite.io/v1/account/tokens/oauth2/google`);

  // Debug 4: Session state verification
  getCurrentSession().then(session => {
    console.log("Current session provider:", session?.provider);
    console.log("Session has access token:", !!session?.providerAccessToken);
  });

  // Debug 5: Identity verification
  getUserIdentities().then(identities => {
    console.log("Number of identities:", identities?.total || 0);
    identities?.identities?.forEach((identity, index) => {
      console.log(`Identity ${index}:`, {
        provider: identity.provider,
        email: identity.providerEmail,
        hasToken: !!identity.providerAccessToken
      });
    });
  });
};
```

10. Lessons Learned

Key Insights About Google OAuth with Expo/React Native

1. OAuth Complexity in Mobile Apps **Insight:** OAuth in mobile apps is significantly more complex than web applications.

Why This Matters: - Web OAuth uses simple redirects and cookies - Mobile OAuth requires deep linking and custom URL schemes - Platform-specific configurations are essential - Browser integration adds another layer of complexity

Technical Learning:

```
// Web OAuth (simple)
window.location.href = oauthUrl; // Redirect happens automatically

// Mobile OAuth (complex)
const result = await WebBrowser.openAuthSessionAsync(oauthUrl, scheme);
const params = new URL(result.url).searchParams;
const session = await account.createSession(params.get('userId'), params.get('secret'));
```

2. Appwrite's OAuth Architecture **Insight:** Appwrite's OAuth implementation separates sessions from identity data.

Discovery Process: - Initial assumption: OAuth tokens would be in session objects - Reality: OAuth tokens are stored in the Identities API - Solution: Always use `account.listIdentities()` for OAuth token access

Code Pattern Learned:

```
// Wrong approach - checking session for OAuth tokens
const session = await getCurrentSession();
if (session.providerAccessToken) { // This is always empty!
  // This never executes
}

// Correct approach - checking identities for OAuth tokens
const identities = await getUserIdentities();
const googleIdentity = identities.identities.find(i => i.provider === 'google');
if (googleIdentity.providerAccessToken) {
  // This works!
}
```

3. Platform-Specific OAuth Requirements **Insight:** iOS and Android have different OAuth configuration requirements.

iOS Requirements: - Custom URL schemes must be registered in Info.plist - Safari in-app browser integration - Specific bundle ID configuration in Google Console

Android Requirements: - Manifest file configuration for custom schemes - Chrome Custom Tabs support - SHA-1 certificate fingerprints for production

Configuration Learning:

```
// app.json - Critical for both platforms
{
  "expo": {
    "scheme": "appwrite-callback-68691394001a2a85ecc5"
  }
}
```

Appwrite OAuth Integration Discoveries

1. OAuth Method Selection **Discovery:** `createOAuth2Session` doesn't work in React Native, but `createOAuth2Token` does.

Why This Difference Exists: - `createOAuth2Session` is designed for web browsers with automatic redirects - `createOAuth2Token` provides manual control needed for mobile apps - React Native requires explicit session creation after OAuth completion

Implementation Pattern:

```
// Use createOAuth2Token for React Native
const loginUrl = await account.createOAuth2Token(provider, successUrl, failureUrl);

// Handle OAuth manually
const result = await WebBrowser.openAuthSessionAsync(loginUrl, scheme);

// Create session manually with OAuth tokens
const session = await account.createSession(userId, secret);
```

2. Token Refresh Strategy **Discovery:** OAuth tokens expire and must be refreshed proactively.

Appwrite's Refresh Mechanism:

```
// Check token expiry before API calls
if (isTokenExpired(identity.providerAccessTokenExpiry)) {
  await account.updateSession('current'); // Refreshes OAuth tokens in identities
}
```

Why This Pattern Works: - `updateSession()` refreshes OAuth tokens stored in identities - Provides seamless token refresh without user intervention - Prevents API failures due to expired tokens

Technical Concepts About OAuth vs Traditional Authentication

1. Authentication Flow Differences **Traditional Email/Password:**

```
User Input → Credentials Validation → Session Creation → App Access
```

OAuth Flow:

```
User Action → OAuth Provider Redirect → User Authorization →
OAuth Token Exchange → Provider API Access → Session Creation → App Access
```

Complexity Comparison: - Traditional: 2 steps, 1 API call - OAuth: 6 steps, multiple API calls, external browser interaction

2. Token Management Differences **Traditional Authentication:** - Single session token - Server-side session validation - Simple expiry handling

OAuth Authentication: - Multiple tokens (access, refresh) - Provider-specific token formats - Complex expiry and refresh logic - Provider API rate limiting concerns

3. User Data Access Patterns **Traditional:**

```
// Direct access to user data
const user = await getCurrentUser();
console.log(user.name); // Data stored in Appwrite
```

OAuth:

```
// Indirect access via provider APIs
const identity = await getUserIdentities();
const profile = await getGoogleProfile(identity.providerAccessToken);
console.log(profile.name); // Data from Google API
```

Configuration Insights

1. Redirect URI Configuration **Critical Learning:** Redirect URIs must be exact matches between OAuth provider and app configuration.

Common Mistakes:

```
// Wrong - trailing slash
"redirectUri": "appwrite-callback-123://oauth/"

// Wrong - different scheme
"redirectUri": "movies://oauth"

// Correct - exact match
"redirectUri": "appwrite-callback-68691394001a2a85ecc5://oauth"
```

2. Environment-Specific Configuration **Development vs Production:**

```
// Development
const redirectUri = makeRedirectUri({ preferLocalhost: true });
// Result: "exp://localhost:8081"

// Production
const redirectUri = makeRedirectUri({ preferLocalhost: false });
// Result: "appwrite-callback-68691394001a2a85ecc5://"
```

Error Handling Patterns

1. OAuth-Specific Error Categories **Network Errors:** - OAuth provider unavailable - Slow network during OAuth flow - Connection timeout during token exchange

Configuration Errors: - Invalid redirect URI - Incorrect client credentials - Missing platform configuration

User Experience Errors: - User cancellation - Account selection timeout - Browser back button during OAuth

2. Graceful Degradation Strategy

```
const fetchUserProfile = async (user) => {
  try {
    // Try to get OAuth profile data
    const oauthProfile = await getOAuthProfile();
    return { ...user, ...oauthProfile };
  } catch (error) {
    console.log("OAuth profile unavailable, using basic profile");
    // Fall back to basic user data
    return {
      name: user.name,
      email: user.email,
      avatar: null // No avatar available
    };
  }
};
```

What Knowledge Was Crucial

1. Understanding OAuth 2.0 Flow **Essential Concepts:** - Authorization vs Authentication - Access tokens vs Refresh tokens - Redirect-based flow principles - Provider-specific API requirements

2. React Native Deep Linking **Critical Knowledge:** - Custom URL scheme registration - Platform-specific configuration requirements - WebBrowser API capabilities and limitations - Deep link parameter extraction

3. Appwrite's OAuth Architecture **Must-Know Details:** - Session vs Identity data separation - OAuth method selection criteria - Token storage and access patterns - Refresh token management

11. Appendices

Appendix A: Updated package.json Dependencies

```
{
  "name": "tutorial_movie_app",
  "main": "expo-router/entry",
  "version": "1.0.0",
  "scripts": {
    "start": "expo start",
    "android": "expo start --android",
    "ios": "expo start --ios",
    "web": "expo start --web",
    "test": "jest --watchAll"
  },
  "jest": {
    "preset": "jest-expo"
  },
  "dependencies": {
    "@expo/vector-icons": "^14.0.4",
    "@react-navigation/native": "^6.0.2",
    "expo": "~52.0.11",
    "expo-auth-session": "~6.0.2",
    "expo-constants": "~17.0.3",
    "expo-font": "~13.0.1",
    "expo-linking": "~7.0.3",
    "expo-router": "~4.0.9",
    "expo-splash-screen": "~0.29.13",
    "expo-status-bar": "~2.0.0",
    "expo-system-ui": "~4.0.4",
    "expo-web-browser": "~14.0.1",
    "nativewind": "^2.0.11",
    "react": "18.3.1",
    "react-dom": "18.3.1",
    "react-native": "0.76.3",
    "react-native-appwrite": "^0.4.0",
    "react-native-reanimated": "~3.16.1",
    "react-native-safe-area-context": "4.12.0",
    "react-native-screens": "~4.1.0",
    "react-native-web": "~0.19.13"
  },
  "devDependencies": {
    "@babel/core": "^7.25.2",
    "@types/react": "~18.3.12",
    "@types/react-native": "^0.73.0",
    "jest": "^29.2.1",
    "jest-expo": "~52.0.2",
    "react-test-renderer": "18.3.1",
    "tailwindcss": "3.3.2",
    "typescript": "~5.3.3"
  },
  "private": true
}
```

Key OAuth-Related Dependencies: - `expo-auth-session`: Provides `makeRedirectUri()` for OAuth redirect handling - `expo-web-browser`: Enables `WebBrowser.openAuthSessionAsync()` for OAuth flows - `expo-linking`: Handles deep link processing (though not directly used in final implementation) - `react-native-appwrite`: Provides OAuth provider constants and methods

Appendix B: Complete Environment Configuration

.env File

```
# TMDB API Configuration
EXPO_PUBLIC_MOVIE_API_KEY=eyJhbGciOiJIUzI1NiJ9.eyJhdWQiOiJiNDFkM2FiNjI0MzAwOGI2MzZlNWFmMGY4N

# Appwrite Configuration
EXPO_PUBLIC_APPWRITE_PROJECT_ID=68691394001a2a85ecc5
EXPO_PUBLIC_APPWRITE_ENDPOINT=https://fra.cloud.appwrite.io/v1
EXPO_PUBLIC_APPWRITE_DATABASE_ID=687558070007955a1389
EXPO_PUBLIC_APPWRITE_COLLECTION_ID=6875583c002e38c8e09c

# Platform Configuration
EXPO_PUBLIC_APPWRITE_BUNDLE_ID=com.ayda.tutorial_movie_app
EXPO_PUBLIC_APPWRITE_PACKAGE_NAME=

# OAuth Configuration
EXPO_PUBLIC_GOOGLE_OAUTH_URL=https://fra.cloud.appwrite.io/v1/account/sessions/oauth2/callba
```

app.json Configuration

```
{
  "expo": {
    "name": "tutorial_movie_app",
    "slug": "tutorial_movie_app",
    "version": "1.0.0",
    "orientation": "portrait",
    "icon": "./assets/images/logo.png",
    "scheme": "appwrite-callback-68691394001a2a85ecc5",
    "userInterfaceStyle": "automatic",
    "newArchEnabled": true,
    "ios": {
      "supportsTablet": true,
      "bundleIdentifier": "com.ayda.tutorial_movie_app"
    },
    "android": {
      "adaptiveIcon": {
        "foregroundImage": "./assets/images/logo.png",
        "backgroundColor": "#ffffff"
      },
      "edgeToEdgeEnabled": true,
      "package": "com.ayda.tutorial_movie_app"
    },
    "web": {
      "bundler": "metro",
      "output": "static",
      "favicon": "./assets/images/logo.png"
    },
    "plugins": [
      "expo-router",
      [
        "expo-splash-screen",
        {
          "image": "./assets/images/logo.png",
          "imageWidth": 200,
          "resizeMode": "contain",
          "backgroundColor": "#ffffff"
        }
      ]
    ],
    "experiments": {
      "typedRoutes": true
    }
  }
}
```

Critical OAuth Configuration: - `scheme`: Must match the Appwrite project ID format - `bundleIdentifier` (iOS): Must match Google Cloud Console configuration - `package` (Android): Must match Google Cloud Console configuration

Appendix C: Google Cloud Console Configuration

OAuth Consent Screen Configuration

```
{
  "applicationType": "Public",
  "applicationName": "Tutorial Movie App",
  "userSupportEmail": "developer@example.com",
  "developerContactInformation": ["developer@example.com"],
  "authorizedDomains": ["appwrite.io"],
  "scopes": [
    "https://www.googleapis.com/auth/userinfo.email",
    "https://www.googleapis.com/auth/userinfo.profile"
  ],
  "testUsers": [
    "test.user@gmail.com"
  ]
}
```

OAuth Client Configuration

```
{
  "clientType": "Web application",
  "clientName": "Tutorial Movie App OAuth Client",
  "authorizedJavaScriptOrigins": [
    "https://fra.cloud.appwrite.io"
  ],
  "authorizedRedirectURIs": [
    "https://fra.cloud.appwrite.io/v1/account/sessions/oauth2/callback/google/68691394001a2a
    "appwrite-callback-68691394001a2a85ecc5://oauth",
    "exp://localhost:8081"
  ]
}
```

Configuration Rationale: - **Web application type**: Required for Appwrite's OAuth integration - **Multiple redirect URIs**: Supports development and production environments - **Appwrite domain origin**: Enables OAuth requests from Appwrite servers

Appendix D: Platform-Specific Configuration Files

iOS Info.plist Additions (Auto-generated by Expo)

```
<dict>
  <!-- Existing configuration -->

  <!-- OAuth URL Scheme Configuration -->
  <key>CFBundleURLTypes</key>
  <array>
    <dict>
      <key>CFBundleURLName</key>
      <string>$(PRODUCT_BUNDLE_IDENTIFIER)</string>
      <key>CFBundleURLSchemes</key>
      <array>
        <string>appwrite-callback-68691394001a2a85ecc5</string>
      </array>
    </dict>
  </array>

  <!-- App Transport Security -->
  <key>NSAppTransportSecurity</key>
  <dict>
    <key>NSAllowsArbitraryLoads</key>
    <true/>
  </dict>
</dict>
```

Android Manifest Additions (Auto-generated by Expo)

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
  <!-- Existing configuration -->

  <!-- OAuth Intent Filter -->
  <activity
    android:name=".MainActivity"
    android:exported="true"
    android:launchMode="singleTask">

    <!-- Existing intent filters -->

    <!-- OAuth Deep Link Intent Filter -->
    <intent-filter android:autoVerify="true">
      <action android:name="android.intent.action.VIEW" />
      <category android:name="android.intent.category.DEFAULT" />
      <category android:name="android.intent.category.BROWSABLE" />
      <data android:scheme="appwrite-callback-68691394001a2a85ecc5" />
    </intent-filter>
  </activity>
</manifest>
```

Appendix E: OAuth Debugging Commands and Resources

Development Debugging Commands

```
# Clear all caches and restart development server
npx expo start --clear --reset-cache

# Check deep link registration
npx uri-scheme list

# Test deep link handling (iOS Simulator)
xcrun simctl openurl booted "appwrite-callback-68691394001a2a85ecc5://oauth?secret=test&use

# Test deep link handling (Android Emulator)
adb shell am start -W -a android.intent.action.VIEW -d "appwrite-callback-68691394001a2a85e

# Monitor OAuth flow logs
npx expo start --dev-client

# Check OAuth URL generation
node -e "
const { makeRedirectUri } = require('expo-auth-session');
console.log('Redirect URI:', makeRedirectUri({ preferLocalhost: true }));
"
```

Production Debugging

```
# Build production app with OAuth support
eas build --platform ios --profile production
eas build --platform android --profile production
```

```
# Test OAuth in production environment
# (Requires physical device and production OAuth client credentials)
```

Useful OAuth Testing URLs

```
# Test OAuth URL format
https://fra.cloud.appwrite.io/v1/account/tokens/oauth2/google?project=68691394001a2a85ecc5&s

# Google OAuth endpoint (for reference)
https://accounts.google.com/oauth/authorize

# Google token exchange endpoint (used by Appwrite)
https://oauth2.googleapis.com/token

# Google user info endpoint (used for profile data)
https://www.googleapis.com/oauth2/v2/userinfo
```

Appendix F: OAuth-Specific Technical Glossary

OAuth 2.0 Terms

- **Authorization Server**: Google's OAuth service that issues tokens
- **Resource Server**: Google's API servers that provide user data
- **Client**: The mobile app requesting OAuth access
- **Authorization Code**: Temporary code exchanged for access tokens
- **Access Token**: Token used to access protected resources (user profile)
- **Refresh Token**: Long-lived token used to obtain new access tokens
- **Redirect URI**: URL where OAuth provider sends authorization results
- **Scope**: Permissions requested from the OAuth provider

Appwrite-Specific Terms

- **OAuth Provider**: Google, configured in Appwrite console
- **Identity**: Appwrite's representation of OAuth account linkage
- **Session**: Appwrite's authentication session (separate from OAuth tokens)
- **Project ID**: Unique identifier for Appwrite project
- **createOAuth2Token**: Appwrite method for React Native OAuth flows
- **providerAccessToken**: OAuth access token stored in Identity object

React Native OAuth Terms

- **Deep Link**: URL scheme that opens the mobile app
- **URL Scheme**: Custom protocol for app-specific URLs
- **WebBrowser.openAuthSessionAsync**: Expo method for OAuth flows
- **makeRedirectUri**: Expo method for generating OAuth redirect URLs
- **Custom Tabs** (Android): Browser component integrated into apps

- **SFSafariViewController** (iOS): Safari browser component for apps

Technical Implementation Terms

- **Token Expiry**: Time when OAuth access tokens become invalid
- **Token Refresh**: Process of obtaining new tokens before expiry
- **Profile Enhancement**: Adding OAuth provider data to user profiles
- **Session Creation**: Converting OAuth tokens into Appwrite sessions
- **Graceful Degradation**: Fallback behavior when OAuth fails

---

This technical report documents the complete journey of implementing Google OAuth authentication in a React Native application with Appwrite backend. The implementation successfully integrates OAuth alongside existing email/password authentication, providing users with multiple sign-in options while maintaining a unified authentication experience.