Identity Design

This document provides detailed implementation guidance for the Identity system, following the architectural decisions made in ADR 0008.

Table of Contents

Architecture Overview

High-Level Flow

```
sequenceDiagram
    participant User
    participant Nuxt as Nuxt Frontend
    participant Auth as @sidebase/nuxt-auth
    participant API as ASP.NET Core API
    participant Identity as ASP.NET Core Identity

    User->>Nuxt: Enter credentials
    Nuxt->>Auth: signIn(credentials)
    Auth->>API: POST /auth/login
    API->>Identity: ValidateUser(credentials)
    Identity-->>API: User validated
    API-->>Auth: JWT token + user info
    Auth-->>Nuxt: Session established
    Nuxt-->>User: Logged in
```

Key Integration Points

1. **Authentication**: `@sidebase/nuxt-auth` calls your ASP.NET Core `/api/auth/login` endpoint
2. **Token Management**: Frontend stores JWT token, includes it in API calls
3. **Session Persistence**: Nuxt auth handles token refresh and session management
4. **Authorization**: ASP.NET Core Identity validates JWT tokens on protected endpoints
5. **User Data**: Both systems share the same user data through the API

Benefits of This Approach

✅ **Stateless**: JWT tokens mean no server-side session storage needed
✅ **Secure**: ASP.NET Core Identity handles password hashing, validation
✅ **Familiar**: You use standard ASP.NET Core auth patterns you already know

✅ **Flexible**: Easy to add external providers later (Google, Microsoft, etc.)
✅ **Frontend Friendly**: Nuxt auth handles token storage, refresh, route protection

Backend Implementation

1. ASP.NET Core Identity Configuration

Program.cs Setup

```
// filepath: c:\Source\jcoliz\YoFi.V3\src\BackEnd\Program.cs
// Add Identity services
builder.Services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();

// Configure JWT Authentication
builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = builder.Configuration["Jwt:Issuer"],
        ValidAudience = builder.Configuration["Jwt:Audience"],
        IssuerSigningKey = new SymmetricSecurityKey(
            Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]))
    };
});

// Add Authorization policies
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("AccountAccess", policy =>
        policy.RequireAssertion(context =>
        {
            var accountId = context.Resource as string;
            var userId = context.User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
            return CheckUserAccountAccess(userId, accountId);
        }));
});
```

2. Authorization Strategy

For your specific requirement of users having access to specific accounts:

```
// filepath: c:\Source\jcoliz\YoFi.V3\src\BackEnd\Program.cs
// Custom authorization policy
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("AccountAccess", policy =>
        policy.RequireAssertion(context =>
        {
            var accountId = context.Resource as string; // Account ID from route
            var userId = context.User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
            // Check database for user-account relationship
            return CheckUserAccountAccess(userId, accountId);
        }));
});
```

Usage in controllers:

```
// In your controllers
[Authorize(Policy = "AccountAccess")]
[HttpGet("accounts/{accountId}/transactions")]
public async Task<IActionResult> GetTransactions(string accountId) { ... }
```

3. Authentication Controller

Create authentication endpoints that `@sidebase/nuxt-auth` can call:

```
// filepath: c:\Source\jcoliz\YoFi.V3\src\BackEnd\Controllers\AuthController.cs
[ApiController]
[Route("api/[controller]")]
public class AuthController : ControllerBase
{
    private readonly SignInManager<ApplicationUser> _signInManager;
    private readonly UserManager<ApplicationUser> _userManager;
    private readonly IConfiguration _configuration;

    public AuthController(
        SignInManager<ApplicationUser> signInManager,
        UserManager<ApplicationUser> userManager,
        IConfiguration configuration)
    {
        _signInManager = signInManager;
        _userManager = userManager;
        _configuration = configuration;
    }

    [HttpPost("login")]
    public async Task<IActionResult> Login([FromBody] LoginRequest request)
    {
        var result = await _signInManager.PasswordSignInAsync(
            request.Email,
            request.Password,
            isPersistent: false,
            lockoutOnFailure: false);

        if (result.Succeeded)
        {
            var user = await _userManager.FindByEmailAsync(request.Email);
            var token = GenerateJwtToken(user);

            return Ok(new
            {
                token = token,
                user = new
                {
                    id = user.Id,
                    email = user.Email,
                    name = user.UserName
                }
            });
        }

        return Unauthorized();
    }

    [HttpGet("user")]
    [Authorize]
    public async Task<IActionResult> GetCurrentUser()
    {
        var userId = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
        var user = await _userManager.FindByIdAsync(userId);

        return Ok(new
        {
            id = user.Id,
            email = user.Email,
            name = user.UserName
        });
    }

    [HttpPost("refresh-token")]
    [Authorize]
    public async Task<IActionResult> RefreshToken()
    {
        var userId = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
        var user = await _userManager.FindByIdAsync(userId);

        if (user == null)
        {
            return Unauthorized();
        }

        var token = GenerateJwtToken(user);
        return Ok(new { token });
```

```
    }

    [HttpPost("logout")]
    [Authorize]
    public async Task<IActionResult> Logout()
    {
        await _signInManager.SignOutAsync();
        return Ok();
    }

    private string GenerateJwtToken(ApplicationUser user)
    {
        var tokenHandler = new JwtSecurityTokenHandler();
        var key = Encoding.ASCII.GetBytes(_configuration["Jwt:Key"]);
        var tokenDescriptor = new SecurityTokenDescriptor
        {
            Subject = new ClaimsIdentity(new[]
            {
                new Claim(ClaimTypes.NameIdentifier, user.Id),
                new Claim(ClaimTypes.Email, user.Email)
            }),
            Expires = DateTime.UtcNow.AddHours(24),
            SigningCredentials = new SigningCredentials(
                new SymmetricSecurityKey(key),
                SecurityAlgorithms.HmacSha256Signature)
        };
        var token = tokenHandler.CreateToken(tokenDescriptor);
        return tokenHandler.WriteToken(token);
    }
}
```

4. JWT Token Generation with Claims

Enhanced JWT token generation to include account access claims:

```
// filepath: c:\Source\jcoliz\YoFi.V3\src\BackEnd\Controllers\AuthController.cs
private async Task<string> GenerateJwtToken(ApplicationUser user)
{
    var tokenHandler = new JwtSecurityTokenHandler();
    var key = Encoding.ASCII.GetBytes(_configuration["Jwt:Key"]);

    // Get user's accessible accounts from database
    var userAccounts = await GetUserAccountAccess(user.Id);

    var claims = new List<Claim>
    {
        new Claim(ClaimTypes.NameIdentifier, user.Id),
        new Claim(ClaimTypes.Email, user.Email),
        new Claim(ClaimTypes.Name, user.UserName ?? user.Email),
        new Claim("sub", user.Id), // Standard JWT claim
        new Claim("email", user.Email),
        new Claim("name", user.UserName ?? user.Email)
    };

    // Add account access claims
    foreach (var account in userAccounts)
    {
        claims.Add(new Claim("account_access", account.AccountId));
        claims.Add(new Claim($"account_role_{account.AccountId}", account.Role));
// e.g., "owner", "viewer", "editor"
    }

    // Add any other custom claims
    claims.Add(new Claim("user_preferences", user.Preferences ?? "{}"));

    var tokenDescriptor = new SecurityTokenDescriptor
    {
        Subject = new ClaimsIdentity(claims),
        Expires = DateTime.UtcNow.AddHours(24),
        Issuer = _configuration["Jwt:Issuer"],
        Audience = _configuration["Jwt:Audience"],
        SigningCredentials = new SigningCredentials(
            new SymmetricSecurityKey(key),
            SecurityAlgorithms.HmacSha256Signature)
    };

    var token = tokenHandler.CreateToken(tokenDescriptor);
    return tokenHandler.WriteToken(token);
}

private async Task<List<UserAccountAccess>> GetUserAccountAccess(string userId)
{
    // This would query your database for user-to-account relationships
    return await _context.UserAccountAccess
        .Where(uaa => uaa.UserId == userId)
        .Select(uaa => new UserAccountAccess
        {
            AccountId = uaa.AccountId,
            Role = uaa.Role
        })
        .ToListAsync();
}
```

5. Data Models

```
// filepath: c:\Source\jcoliz\YoFi.V3\src\BackEnd\Entities\Models\Identity.cs
public class LoginRequest
{
    public string Email { get; set; }
    public string Password { get; set; }
}
```

```
public class UserAccountAccess
{
    public string AccountId { get; set; }
    public string Role { get; set; } // "owner", "editor", "viewer"
}

public class ApplicationUser : IdentityUser
{
    public string? Preferences { get; set; }
    public virtual ICollection<UserAccountAccess> AccountAccess { get; set; }
}
```

Frontend Implementation

1. Nuxt Configuration

Configure `@sidebase/nuxt-auth` to use your custom ASP.NET Core endpoints:

```
// filepath: c:\Source\jcoliz\YoFi.V3\src\FrontEnd.Nuxt\nuxt.config.ts
export default defineNuxtConfig({
  modules: ['@sidebase/nuxt-auth'],
  auth: {
    baseURL: process.env.NUXT_PUBLIC_API_BASE_URL,
    provider: {
      type: 'authjs',
      trustHost: true,
      defaultProvider: 'credentials'
    },
    session: {
      enableRefreshPeriodically: true,
      enableRefreshOnWindowFocus: true,
    }
  },
  runtimeConfig: {
    authSecret: process.env.NUXT_AUTH_SECRET,
    public: {
      authJs: {
        baseUrl: process.env.NUXT_PUBLIC_API_BASE_URL
      }
    }
  }
})
```

2. Authentication Handler

Configure Auth.js to work with ASP.NET Core backend and expose JWT claims:

```
// filepath: c:\Source\jcoliz\YoFi.V3\src\FrontEnd.Nuxt\server\api\auth\[...].ts
import CredentialsProvider from '@auth/core/providers/credentials'
import { NuxtAuthHandler } from '#auth'

export default NuxtAuthHandler({
  secret: useRuntimeConfig().authSecret,
  providers: [
    CredentialsProvider({
      name: 'credentials',
      credentials: {
        email: { label: 'Email', type: 'email' },
        password: { label: 'Password', type: 'password' }
      },
      async authorize(credentials) {
        try {
          const response = await
$fetch(`${process.env.NUXT_PUBLIC_API_BASE_URL}/api/auth/login`, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({
              email: credentials.email,
              password: credentials.password
            })
          })

          if (response.token) {
            return {
              id: response.user.id,
              email: response.user.email,
              name: response.user.name,
              accessToken: response.token
            }
          }
          return null
        } catch (error) {
          console.error('Auth error:', error)
          return null
        }
      }
    })
  ],
  callbacks: {
    async jwt({ token, user }) {
      if (user) {
        token.accessToken = user.accessToken
      }
      return token
    },
    async session({ session, token }) {
      session.accessToken = token.accessToken
      return session
    }
  }
})
```

3. Authenticated API Calls

Composable to automatically include JWT token in API requests:

```
// filepath:
c:\Source\jcoliz\YoFi.V3\src\FrontEnd.Nuxt\composables\useAuthenticatedFetch.ts
export const useAuthenticatedFetch = () => {
  const { data: session } = useAuth()

  return $fetch.create({
    onRequest({ request, options }) {
      if (session.value?.accessToken) {
        options.headers = {
          ...options.headers,
          Authorization: `Bearer ${session.value.accessToken}`
        }
      }
    }
  })
}

// Usage in components
// const authFetch = useAuthenticatedFetch()
// const transactions = await authFetch('/api/transactions')
```

4. Enhanced Authentication Handler with JWT Claims

For the claims-based authorization features, here's the enhanced authentication handler:

```
// filepath: c:\Source\jcoliz\YoFi.V3\src\FrontEnd.Nuxt\server\api\auth\[...].ts
import CredentialsProvider from '@auth/core/providers/credentials'
import { NuxtAuthHandler } from '#auth'
import jwt from 'jsonwebtoken'

export default NuxtAuthHandler({
  secret: useRuntimeConfig().authSecret,
  providers: [
    CredentialsProvider({
      name: 'credentials',
      credentials: {
        email: { label: 'Email', type: 'email' },
        password: { label: 'Password', type: 'password' }
      },
      async authorize(credentials) {
        try {
          const response = await
$fetch(`${process.env.NUXT_PUBLIC_API_BASE_URL}/api/auth/login`, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({
              email: credentials.email,
              password: credentials.password
            })
          })

          if (response.token) {
            // Decode the JWT to access claims on the frontend
            const decodedToken = jwt.decode(response.token) as any

            return {
              id: decodedToken.sub || decodedToken.nameid,
              email: decodedToken.email,
              name: decodedToken.name,
              accessToken: response.token,
              // Include custom claims in the session
              accountAccess: decodedToken.account_access || [],
              accountRoles: Object.keys(decodedToken)
                .filter(key => key.startsWith('account_role_'))
                .reduce((acc, key) => {
                  const accountId = key.replace('account_role_', '')
                  acc[accountId] = decodedToken[key]
                  return acc
                }, {} as Record<string, string>),
              userPreferences: decodedToken.user_preferences ?
                JSON.parse(decodedToken.user_preferences) : {}
            }
          }
          return null
        } catch (error) {
          console.error('Auth error:', error)
          return null
        }
      }
    })
  ],
  callbacks: {
    async jwt({ token, user }) {
      if (user) {
        // Preserve all user data including claims
        return {
          ...token,
          ...user
        }
      }
      return token
    },
    async session({ session, token }) {
      // Make claims available in the session
      session.user.id = token.id
      session.user.accountAccess = token.accountAccess
      session.user.accountRoles = token.accountRoles
      session.user.userPreferences = token.userPreferences
      session.accessToken = token.accessToken
      return session
    }
  }
})
```

5. Accessing Claims in Components

Now you can easily access the claims throughout your Nuxt app:

```
// filepath:
c:\Source\jcoliz\YoFi.V3\src\FrontEnd.Nuxt\composables\useAccountAccess.ts
export const useAccountAccess = () => {
  const { data: session } = useAuth()

  const hasAccountAccess = (accountId: string): boolean => {
    return session.value?.user?.accountAccess?.includes(accountId) || false
  }

  const getAccountRole = (accountId: string): string | null => {
    return session.value?.user?.accountRoles?.[accountId] || null
  }

  const canEditAccount = (accountId: string): boolean => {
    const role = getAccountRole(accountId)
    return role === 'owner' || role === 'editor'
  }

  const canViewAccount = (accountId: string): boolean => {
    return hasAccountAccess(accountId)
  }

  const isAccountOwner = (accountId: string): boolean => {
    return getAccountRole(accountId) === 'owner'
  }

  const accessibleAccounts = computed(() => {
    return session.value?.user?.accountAccess || []
  })

  const userPreferences = computed(() => {
    return session.value?.user?.userPreferences || {}
  })

  return {
    hasAccountAccess,
    getAccountRole,
    canEditAccount,
    canViewAccount,
    isAccountOwner,
    accessibleAccounts,
    userPreferences
  }
}
```

6. Middleware for Route Protection

Create middleware to automatically protect routes based on claims:

```
// filepath: c:\Source\jcoliz\YoFi.V3\src\FrontEnd.Nuxt\middleware\account-
access.ts
export default defineNuxtRouteMiddleware((to) => {
  const { hasAccountAccess } = useAccountAccess()
  const accountId = to.params.id as string

  if (accountId && !hasAccountAccess(accountId)) {
    throw createError({
      statusCode: 403,
      statusMessage: `Access denied to account ${accountId}`
    })
  }
})
```

7. Real-time Claims Updates

If account access can change while the user is logged in, you can create a method to refresh the token:

```
// filepath:
c:\Source\jcoliz\YoFi.V3\src\FrontEnd.Nuxt\composables\useAuthRefresh.ts
export const useAuthRefresh = () => {
  const { data: session, update } = useAuth()

  const refreshUserClaims = async () => {
    try {
      const authFetch = useAuthenticatedFetch()
      const response = await authFetch('/api/auth/refresh-token', {
        method: 'POST'
      })

      if (response.token) {
        // Decode new token and update session
        const decodedToken = jwt.decode(response.token) as any

        await update({
          ...session.value,
          user: {
            ...session.value?.user,
            accountAccess: decodedToken.account_access || [],
            accountRoles: Object.keys(decodedToken)
              .filter(key => key.startsWith('account_role_'))
              .reduce((acc, key) => {
                const accountId = key.replace('account_role_', '')
                acc[accountId] = decodedToken[key]
                return acc
              }, {} as Record<string, string>),
          },
          accessToken: response.token
        })
      }
    } catch (error) {
      console.error('Failed to refresh claims:', error)
    }
  }

  return { refreshUserClaims }
}
```

Claims-Based Authorization

1. Claim Types

- `account_access`: List of account IDs the user has access to
- `account_role_{accountId}`: Role of the user for the specific account (e.g., "owner", "viewer", "editor")
- `user_preferences`: JSON string of user-specific settings/preferences

2. Using Claims in API

ASP.NET Core Identity automatically validates the claims in the JWT. You can also create custom authorization attributes:

```
// filepath:
c:\Source\jcoliz\YoFi.V3\src\BackEnd\Authorization\AccountAccessAttribute.cs
public class AccountAccessAttribute : AuthorizeAttribute
{
    protected override bool IsAuthorized(HttpActionContext actionContext)
    {
        var user = actionContext.User;
        var accountId = actionContext.ActionArguments["accountId"]?.ToString();

        return user != null && CheckUserAccountAccess(user.Identity.GetUserId(),
accountId);
    }
}

// Usage in controllers
[Authorize(Policy = "AccountAccess")]
[HttpGet("accounts/{accountId}/transactions")]
public async Task<IActionResult> GetTransactions(string accountId) { ... }
```

3. Using Claims in Frontend

Claims are available in the Nuxt app through the `useAuth` composable:

```
const { data: session } = useAuth()

const accountAccess = session.value?.user?.accountAccess
const canEdit = session.value?.user?.accountRoles[accountId] === 'editor'
```

4. Key Benefits of Claims-Based Authorization

☑ **Client-side Authorization**: No need to call the API to check permissions
☑ **Performance**: Claims are cached in the JWT, reducing database calls
☑ **Security**: Claims are cryptographically signed and tamper-proof
☑ **Flexibility**: Easy to add new claim types without frontend changes
☑ **Real-time**: Can refresh claims when permissions change
☑ **Offline-friendly**: Works even when temporarily disconnected

Advanced Patterns

1. Token Refresh Strategy

Handle token refresh for long-lived sessions:

```
// filepath:
c:\Source\jcoliz\YoFi.V3\src\FrontEnd.Nuxt\composables\useAuthRefresh.ts
export const useAuthRefresh = () => {
  const { data: session, update } = useAuth()

  const refreshUserClaims = async () => {
    try {
      const authFetch = useAuthenticatedFetch()
      const response = await authFetch('/api/auth/refresh-token', {
        method: 'POST'
      })

      if (response.token) {
        // Decode new token and update session
        const decodedToken = jwt.decode(response.token) as any

        await update({
          ...session.value,
          user: {
            ...session.value?.user,
            accountAccess: decodedToken.account_access || [],
            accountRoles: Object.keys(decodedToken)
              .filter(key => key.startsWith('account_role_'))
              .reduce((acc, key) => {
                const accountId = key.replace('account_role_', '')
                acc[accountId] = decodedToken[key]
                return acc
              }, {} as Record<string, string>),
          },
          accessToken: response.token
        })
      }
    } catch (error) {
      console.error('Failed to refresh claims:', error)
    }
  }

  return { refreshUserClaims }
}
```

2. Dynamic Permission Updates

Handle real-time permission changes:

```
// filepath:
c:\Source\jcoliz\YoFi.V3\src\FrontEnd.Nuxt\composables\usePermissionUpdates.ts
export const usePermissionUpdates = () => {
  const { refreshUserClaims } = useAuthRefresh()

  // Call this when user permissions might have changed
  const handlePermissionChange = async () => {
    await refreshUserClaims()

    // Optionally redirect if user lost access to current page
    const route = useRoute()
    const { hasAccountAccess } = useAccountAccess()

    if (route.params.id && !hasAccountAccess(route.params.id as string)) {
      await navigateTo('/accounts')
    }
  }

  // Listen for permission change events (e.g., from WebSocket)
  onMounted(() => {
    // Example: WebSocket listener for permission changes
    // websocket.on('permissions-updated', handlePermissionChange)
  })

  return {
    handlePermissionChange
  }
}
```

3. Endpoint Configuration Flexibility

`@sidebase/nuxt-auth` provides complete flexibility in endpoint configuration:

Custom Provider Configuration

```
// filepath: c:\Source\jcoliz\YoFi.V3\src\FrontEnd.Nuxt\server\api\auth\[...].ts
// Custom endpoints - fully configurable
const API_BASE = process.env.NUXT_PUBLIC_API_BASE_URL // e.g.,
"https://api.yourdomain.com"

// Your custom endpoints
const authEndpoints = {
  login: `${API_BASE}/api/auth/login`,
  user: `${API_BASE}/api/auth/user`,
  logout: `${API_BASE}/api/auth/logout`,
  refresh: `${API_BASE}/api/auth/refresh-token`
}

export default NuxtAuthHandler({
  providers: [
    CredentialsProvider({
      async authorize(credentials) {
        // Use your custom endpoints
        const response = await $fetch(authEndpoints.login, {
          method: 'POST',
          body: JSON.stringify(credentials)
        })
        return response
      }
    })
  ]
})
```

Key Benefits

☑ **Environment-specific**: Different URLs for dev/staging/prod
☑ **Runtime configurable**: Can change without rebuilding
☑ **Custom endpoints**: Use any URL structure you want
☑ **Multiple backends**: Can authenticate against different services

So you have complete control over where `@sidebase/nuxt-auth` makes its authentication calls - nothing is hard-coded!

4. Error Handling

Comprehensive error handling for authentication flows:

```
// filepath:
c:\Source\jcoliz\YoFi.V3\src\FrontEnd.Nuxt\composables\useAuthErrors.ts
export const useAuthErrors = () => {
  const handleAuthError = (error: any) => {
    if (error.status === 401) {
      // Token expired or invalid
      navigateTo('/login')
    } else if (error.status === 403) {
      // Insufficient permissions
      throw createError({
        statusCode: 403,
        statusMessage: 'Access denied'
      })
    } else if (error.status === 422) {
      // Validation error
      console.error('Authentication validation failed:', error.data)
    }
  }

  return { handleAuthError }
}
```

Security Considerations

1. JWT Best Practices

- **Token Expiration**: Use reasonable expiration times (1-24 hours)
- **Token Size**: Limit claims to keep JWT lightweight (< 8KB recommended)
- **Sensitive Data**: Never include sensitive information in claims (they're base64 encoded, not encrypted)
- **Refresh Strategy**: Implement token refresh for long-lived sessions
- **Secure Storage**: Frontend stores tokens in secure httpOnly cookies when possible

2. Claims Validation

- **Backend Validation**: Always validate claims on the backend for sensitive operations
- **Frontend Guards**: Use claims for UI/UX decisions but not security enforcement
- **Role Hierarchy**: Implement proper role hierarchy checking
- **Claim Integrity**: Claims are cryptographically signed and tamper-proof

3. Configuration Security

```
// filepath: c:\Source\jcoliz\YoFi.V3\src\FrontEnd.Nuxt\.env.example
# Auth Configuration - Keep these secret!
NUXT_AUTH_SECRET=your-super-secret-auth-secret-here-minimum-32-chars
NUXT_PUBLIC_API_BASE_URL=http://localhost:5000

# JWT Configuration (Backend) - Keep these secret!
JWT_KEY=your-jwt-signing-key-here-minimum-256-bits
JWT_ISSUER=YoFi.V3
JWT_AUDIENCE=YoFi.V3.Client
```

4. Key Benefits of Claims-Based Authorization

✅ **Client-side Authorization**: No need to call the API to check permissions
✅ **Performance**: Claims are cached in the JWT, reducing database calls
✅ **Security**: Claims are cryptographically signed and tamper-proof
✅ **Flexibility**: Easy to add new claim types without frontend changes
✅ **Real-time**: Can refresh claims when permissions change
✅ **Offline-friendly**: Works even when temporarily disconnected

This comprehensive implementation provides a secure, scalable identity system using ASP.NET Core Identity with @sidebase/nuxt-auth, including advanced features like claims-based authorization and real-time permission updates.