# IdentityUser — Authentication Starter Kit

# Documentation

**Author: Sadegh Shojayefard**

**Package Creator: Sadegh Shojayefard**

**Generated: November 22, 2025**

## Contents

## Introduction

IdentityUser is a lightweight authentication starter kit inspired by Microsoft Identity for ASP.NET Core.

It is developed to provide Next.js applications with a modular, secure and ready-to-use user authentication system including roles, claims, models, validation schemas, services, utilities, and optional NextAuth integration.

This documentation explains what IdentityUser is, what it contains, how to install and initialize it, and how to use its main components. The examples and instructions are based on the repository named `identityusers_sample` which will be linked in this document once the repository is published to GitHub.

## What this document contains

• A high-level overview of IdentityUser and its design goals.

• Step-by-step installation and initialization instructions.

• Description of folder structure and important files.

• Code snippets demonstrating common tasks (hashing passwords, adding claims, fetching roles).

• Compatibility and upgrade notes.

• Contribution, license and author/contact information.

## Intended audience

This document targets frontend and full-stack developers using Next.js (v15+) who want a quick and reliable authentication solution with minimal setup. Basic knowledge of Node.js, Next.js, MongoDB (Mongoose), and TypeScript/JavaScript is assumed.

## 1. Overview (README)

IdentityUser — A lightweight and ready-to-use User Authentication Starter Kit for Next.js applications.

IdentityUser helps you quickly scaffold a fully functional authentication system into your project — including models, validation schemas, services, utilities, and optional NextAuth integration. Perfect for developers who want a clean, modular, and production-ready user system with minimal setup.

### Features

• Auto-copy authentication boilerplate into your project.

• Built-in Zod validation.

• Mongoose user model.

• Password hashing with bcrypt.

• Resend-ready to send email to user

• NextAuth-ready structure.

• Zero configuration — just install & run one command.

• Fully TypeScript-compatible.

• Clean and maintainable folder structure.

### Installation

Run the following command inside your Next.js project:

```
npm install identityuser
```

### Initialize the Authentication Module

IdentityUser includes a CLI tool that copies the entire src/identityUser folder into your project. Run:

```
npx identityuser
```

After running this command, a folder like `src/identityUser/` will appear inside your project.
If a folder named `src/identityUser` already exists the CLI will avoid overwriting by creating a versioned folder (identityUser-2, identityUser-3, ...). You may need to adjust imports or merge files manually after running the CLI.

### Required Dependencies

IdentityUser relies on several peer dependencies that must be installed manually (npm does not auto-install peerDependencies).

Install required packages with:

```
npm install next-auth bcrypt mongoose zod @conform-to/zod @conform-to/react resend
@upstash/ratelimit @upstash/redis otplib qrcode
```

If using TypeScript add the typing for bcrypt:

```
npm install -D @types/bcrypt @types/qrcode
```

## 2. Zod Validation Note

If you are using Zod v4, note that the `required_error` field has been removed. Use `error` or direct validators such as `.min()`, `.email()`, `.max()`.

Example (Zod v4 compatible):

```
z.string({ error: "Please fill the Username field first" })
```

IdentityUser's internal schemas follow Zod v4 syntax.

## 3. Folder Structure (Generated After Init)

A full authentication starter pack will be added to: `src/identityUser/`

Included folders and files (summary):

```
app
 └api
  └auth
   └ [...nextauth]/route.ts
   └ session
     └ update/route.ts

identityUser/
 ├ api/
 |  ├ auth/
 |  |  └ [...nextauth]/
 |  |    ├ authHelpers.ts
 |  |    └ options.ts
 ├ components/
 |  └ sessionWatcher/SessionWatcher.tsx
 ├ helper/
 |  ├ claimsAction.ts
 |  ├ roleAction.ts
 |  ├ sharedFunction.ts
 |  ├ signInAction.ts
 |  ├ signUpformAction.ts
 |  └ userAction.ts
 ├ lib/
 |  ├ models/
 |  |  ├ identityUser_claims.ts
 |  |  ├ identityUser_roleClaims.ts
 |  |  ├ identityUser_roles.ts
 |  |  ├ identityUser_Tokens.ts
 |  |  ├ identityUser_userClaims.ts
 |  |  ├ identityUser_userRoles.ts
 |  |  └ identityUser_users.ts
 |  ├ utils/rateLimit.ts
 |  ├ authGuard.ts
 |  ├ db.ts
 |  └ session.ts
 ├ providers/SessionProvider.tsx
 ├ Type/next-auth.d.ts
 └ validation/*.ts
```

## 4. IdentityUser Database Structure (Tables & Fields)

This section provides a detailed overview of all database tables used in the IdentityUser package.
These tables are inspired by the original ASP.NET Core Identity schema, but adapted for Next.js + MongoDB (Mongoose) environments.

Each table includes:

- Field name
- Data type (as used in MongoDB/Mongoose)
- Description and purpose

Note:
The classic ASP.NET Identity tables used for storing external logins (e.g., Google, GitHub, Microsoft OAuth providers) are not implemented here, because NextAuth already handles external authentication flows internally and does not require database tables for them.

## 4.1. IdentityUser_Users Table

The main user table. Stores core authentication and profile data.

| | Field | Type | Description |
|---|---|---|---|
| 1 | username | string | The unique username chosen by the user. |
| 2 | normalizedUserName | string | Uppercase username used for security and case-insensitive queries. |
| 3 | email | string | User’s email address. |
| 4 | normalizedEmail | string | Uppercase email used for security and searching. |
| 5 | emailConfirmed | boolean | Whether the user’s email has been verified. |
| 6 | passwordHash | string | The bcrypt-hashed password. |
| 7 | securityStamp | uuid | A random UUID that invalidates all user sessions when changed. |
| 8 | concurrencyStamp | uuid | Used to prevent update conflicts (similar to row versioning). |
| 9 | phoneNumber | string | User’s phone number. |
| 10 | phoneNumberConfirmed | boolean | True if the phone has been verified. |
| 11 | twoFactorEnabled | boolean | Enables 2FA for the account. |
| 12 | twoFactorSecret | String | Save secret code for TOTP |
| 13 | recoveryCodes | String[] | Hold recovery codes for emergency login |
| 14 | lockoutEnd | Date / null | Until what date the user is locked out (if any). |
| 15 | lockoutEnabled | boolean | Whether lockout functionality is active. |
| 16 | accessFailedCount | number | Consecutive invalid login attempts. |
| 17 | avatar | string | URL of the profile image. |
| 18 | name | string | Full name displayed in the UI. |

## 4.2. IdentityUser_UserRoles Table

Mapping table between Users and Roles (many-to-many).

| | Field | Type | Description |
|---|---|---|---|
| 1 | role | ObjectId (Role) | The assigned role. |
| 2 | user | ObjectId (User) | The user receiving this role. |

## 4.3. IdentityUser_UserClaims Table

Store user-specific claims independent of roles.

| | Field | Type | Description |
|---|---|---|---|
| 1 | user | ObjectId (User) | Owner of the claim. |
| 2 | claim | ObjectId (Claim) | Claim assigned directly to the user. |

## 4.4. IdentityUser_Roles Table

Stores system roles (e.g., Admin, User, Manager).

| | Field | Type | Description |
|---|---|---|---|
| 1 | name | string | Role name. |
| 2 | normalizedName | string | Uppercase role name for database lookups. |
| 3 | concurrencyStamp | string | Used for safe updates. |
| 4 | claimStamp | string | Updated whenever role claims change (helps session refresh). |
| 5 | description | string | Human-readable role description. |

## 4.5. IdentityUser_RoleClaims Table

Mapping table between Roles and Claims.

| | Field | Type | Description |
|---|---|---|---|
| 1 | role | ObjectId (Role) | The role receiving the claim. |
| 2 | claim | ObjectId (Claim) | The claim assigned to this role. |

## 4.6. IdentityUser_Claims Table

Stores all available claims in the system.

| | Field | Type | Description |
|---|---|---|---|
| 1 | claimType | string | Category/type of claim (e.g., "access", "permissions"). |
| 2 | claimValue | string | The specific value (e.g., "edit-users", "delete-posts"). |
| 3 | description | string | Human-readable explanation of the claim. |

## 4.7. IdentityUser_Tokens  Table

Stores user tokens for recovery password and verify email/phone

| | Field | Type | Description |
|---|---|---|---|
| 1 | user | UUID | User ID |
| 2 | identifier | String | Hold the user input like phoneNumber or Email address |
| 3 | type | Enum String | Type of the token. Only accept email,phone,email-verify,phone-verify |
| 4 | hashedToken | string | Save the user token in hash mode |
| 5 | expireAt | Date | Date of token expire and delete automatically after it. |
| 6 | attempts | Number | User failed attempts to use token. Mostly used for phone OTP |

# 5. NextAuth Integration & Authentication Flow

## 5.1. Overview

IdentityUser integrates NextAuth as its main authentication layer.

While NextAuth supports multiple OAuth providers (Google, GitHub, etc.), this project uses a custom Credentials Provider because:

- The system requires custom user fields (roles, claims, securityStamp).
- The app uses a fully custom user model stored in MongoDB.
- IdentityUser handles all user/role/claim logic internally.
- OAuth login tables used in Microsoft ASP.NET Identity are not required because NextAuth manages those internally.

## 5.2. Default NextAuth Session Structure

By default, a NextAuth session contains:

```
session = {
 user: {
  name: string | null,
  email: string | null,
  image: string | null
 },
 expires: string
}
```

This structure is too limited for IdentityUser because the app needs:

- user.id
- username
- name
- email
- avatar
- roles[]
- claims[]
- securityStamp
- Therefore, the session and JWT were extended.

### 5.3. Custom NextAuth Session Structure

IdentityUser overrides the default NextAuth types through:

**File: /types/next-auth.d.ts**

It extends:

- Session.user
- JWT
- User

This enables storing identity-specific information inside:

- The JWT (server-side)
- The session (client-side)
- The User object returned during login

This makes role-based pages, permissions, guards, and CMS admin panel possible.

## 5.4. Authentication Flow Architecture

```
Credentials Login → authorize() → JWT Callback → Session Callback → Client Session
```

Breakdown:

1. User submits username/password
2. authorize()
    - connect to MongoDB
    - load user with getUserByUsernameAction
    - validate password
    - return user object with roles & claims
3. jwt() callback
    - save user data into JWT
    - auto-sync every 30 minutes with DB
    - detect changes in roles/claims/securityStamp
4. session() callback
    - validate securityStamp
    - validate roles/claims
    - finalize session.user

## 5.5. File-by-File Explanation

### 5.5.1. src/app/api/auth/[...nextauth]/route.ts

```
import { options } from "@/identityUser/api/auth/[...nextauth]/options";
import NextAuth from "next-auth";

const handler = NextAuth(options);
export { handler as GET, handler as POST };
```

**Purpose**

- Bridges your global project structure with IdentityUser's authentication configuration.
- Enables both GET and POST requests for authentication-related operations.

### 5.5.2. identityUser /api/auth/[...nextauth]/authHelpers.ts

Utility functions for server authentication.

Functions

**getSession()**

- Returns the current NextAuth session using your custom options.

**signIn(provider, data)**

- Wrapper for next-auth/react signIn
- Enables custom provider login from the frontend.

**auth()**

- Returns NextAuth instance with your options (used in server components or route handlers).

### 5.5.3. identityUser /api/auth/[...nextauth]/options.ts

This is the core authentication engine.

**Providers**

Only CredentialsProvider is used:

- Reads username + password
- Loads user from MongoDB
- Compares hashed password
- Returns custom user object with roles & claims

**JWT Callback**

This part:

- Saves user data into JWT on login
- Applies updates when trigger === 'update'
- Auto-syncs database values every 30 minutes
- Detects:
    - role changes
    - claim changes
    - securityStamp changes
    - and instantly forces logout

**Session Callback**

This ensures session integrity:

- Loads securityStamp from DB
- Logs the user out if the stamp changed
- Validates updated roles
- Validates updated claims
- Returns full session.user object to client

**Why this matters**

- This design makes IdentityUser behave like:
- ASP.NET Identity (securityStamp invalidation)
- Enterprise Role-Based Access Control
- Dynamic permissions without requiring logout

### 5.5.4. app/api/session/update/route.ts
A custom session refresh API.

**Purpose**

- Refresh session data manually from the client
- Returns updated roles, claims, avatar, securityStamp, etc.

**Used for:**

- CMS dashboards
- Role-change refresh
- Updating avatar or profile info without logout

## 5.6. Type Extensions (types/next-auth.d.ts)

These are essential to safely work with extended User & Session objects.

**Session.user**

Contains full identity data:

- id
- username
- name
- email
- avatar
- roles
- claims
- securityStamp

**JWT**

Stores all user identity info internally, plus:

- lastSync (used for auto-sync timing)

## 5.7. Summary: Why This Authentication System Is Better

Compared to default NextAuth:

| | Feature | Default | IdentityUser |
|---|---|---|---|
| 1 | Role support | None | Built-in |
| 2 | Claims support | None | Full RBAC |
| 3 | securityStamp | No | Automatic logout on change |
| 4 | Auto-sync with DB | No | Every 30 minutes |
| 5 | Admin-ready session | Simple user | Full user profile |
| 6 | TypeScript safety | Limited | Full type extension |

IdentityUser basically transforms NextAuth into:

**A fully custom identity management system similar to .NET Identity — but for Next.js.**

## 6. Session Management & Access Control Helpers

This section includes the utilities responsible for session validation, automatic logout handling, and access control throughout the application. These tools work alongside NextAuth to enhance security and ensure consistent user state across all browser tabs and server-side logic.

### 6.1. SessionWatcher Component

**Purpose**

A client-side component that:

1. Automatically logs out the user if the session becomes invalid.
2. Synchronizes logout events across all open browser tabs using localStorage.

**How It Works**

- When status === "authenticated" but session.user is missing, the component triggers a logout and writes a timestamp to localStorage.logout.
- All other tabs listen for this change and immediately execute signOut().
- A callbackUrl is used to redirect the user to the homepage of the current locale.

## 6.2. AuthGuard (Server-Side Page Protection)

**Purpose**

Server-side access control for protected and guest-only pages.

**Functions**

1. requireGuest()

- Redirects the user if they are already logged in.
- Useful for Login, Register, and Forgot Password pages.

2. requireAuth()

- Ensures the user is authenticated.
- Redirects instantly if the session is missing or corrupted.
- Used for Dashboard, CMS, Profile, and all other private pages.

**Notes**

- Works inside Server Components, providing true backend-level access protection.
- Prevents any possibility of bypassing authentication from the client side.
- Essential for building a secure CMS panel.

## 6.3. Session Utilities (Roles & Claims Checking)

This Session file location is in:

identityUser/lib/session.ts

**Purpose**

Utility functions for checking roles, claims, and session validity.

Used when building authorization logic for:

- CMS dashboards
- Server Actions
- Page-level and component-level permissions
- Conditional UI rendering

**Features**

- getSession() → Returns full server session
- hasClaim(claim) → Checks for a specific claim
- hasRole(role) → Checks for a specific role
- hasAnyClaim() → Returns true if user has at least one claim
- hasAnyRole() → Returns true if user has at least one role

**Notes**

These utilities enforce server-side authorization and guarantee that user permissions cannot be faked or manipulated from the client.

## 7. Rate Limiting Overview

This package provides several rate limiters powered by Upstash Redis, designed to protect the authentication system from abuse, brute-force attacks, and API overuse.

IdentityUser uses **Upstash Redis** and **Upstash Rate** Limit to protect your authentication endpoints from abuse.

All limiters are powered by:

- @upstash/redis → Serverless Redis database
- @upstash/ratelimit → Production-grade rate limiting with sliding window algorithm

The entire system works serverless, meaning it supports:

- Vercel
- Netlify
- Cloudflare
- Node servers
- Any serverless environment

No manual server configuration required.

### 7.1. Setup Requirements

To enable rate limiting, you must complete two simple steps:

#### 7.1.1. UserUpdateAction

1. Go to [https://upstash.com](https://upstash.com)
2. Create an account (GitHub, Google, or email)
3. Click Redis → Create Database
4. Region: choose nearest to your deployment (e.g. eu-central-1)
5. After creation, open the database and copy:
6. UPSTASH_REDIS_REST_URL
7. UPSTASH_REDIS_REST_TOKEN
8. These are required environment variables.

### 7.1.2. Add Required Environment Variables

Add these to your .env.local or deployment environment:

```
UPSTASH_REDIS_REST_URL=your_upstash_rest_url
UPSTASH_REDIS_REST_TOKEN=your_upstash_access_token
```

If these values are missing, the system automatically throws:

```
⍰ Upstash ENV variables are missing!
```

So the developer knows exactly what to fix.

## 7.2. Why Upstash is Used

Upstash is a perfect match for Next.js App Router because:

- Native REST API access
- No persistent TCP connections required
- Zero maintenance
- Free tier available
- Works in serverless environments
- Minimal latency (globally distributed)
- You do not need to run or manage Redis manually.

## 7.3. Limiters Used by IdentityUser

Each limiter targets a different type of request behavior:

### 7.3.1. IP Limiter

The IP-based limiter restricts requests based on the client's IP address.

- Every IP receives its own independent limit window.
- Multiple users from different IPs will not affect each other.
- Useful for preventing brute-force attempts and repeated attacks from a single source.

For example, if the limit is 2 requests / 2 minutes, each IP can send 2 requests within that period, regardless of how many other users are active.

**Note**:

The IP-based rate limiter may not behave correctly in local development because all requests originate from the same IP address (127.0.0.1).

This causes every request—no matter who sends it—to count toward the same rate limit bucket.

For accurate results, test the IP limiter on a deployed environment where clients have real IP addresses.

### 7.3.2. Global Limiter

The global limiter applies a shared request quota across the entire application.

- All users and all IPs share the same bucket.
- When the global quota is exhausted, no one can make further requests until the window resets.
- Useful for protecting the system from high-traffic spikes or distributed attacks.

For example, with a global limit of 10 requests / 5 minutes, once 10 requests are made (by one user or many), the API stops accepting requests globally.

**Note:**

The global limiter behaves normally in local development, but since traffic volume is extremely low, it may appear as if nothing is being throttled.

Global limits become meaningful only under real-world load conditions on a live server.

### 7.3.3. Email & Phone Limiters

These limiters specifically control actions involving email or phone verification.

- Prevents users from repeatedly requesting verification codes.
- Helps minimize costs and blocks potential abuse.

Each user (identified by email or phone number) has a separate limiter window.

**Note:**

Email and phone verification limiters work in local development, but since local systems typically do not send real email or SMS messages, throttling might not feel as restrictive.

Their real effectiveness becomes clear when integrated with live email/SMS services.

### 7.3.4. Reset Password Limiter

Prevents excessive password reset attempts for the same account identifier (email/username).

It protects users from targeted reset-link spam and slows malicious attempts to take over accounts.

**Note:**

The reset-password limiter functions normally in development, but it still shares the same local constraints: all simulated traffic likely originates from one user/IP, which may distort behavior compared to real usage.

## 7.4. Summary Table

| | Limiter | Scope | Use Case |
|---|---|---|---|
| 1 | emailLimiter | Per email | Verification code abuse prevention |
| 2 | PhoneLimiter | Per phone number | OTP spam prevention |
| 3 | ipLimiter | Per IP | Brute-force & single-source attacks |
| 4 | globalLimiter | Entire system | DDoS-like protection & resource control |
| 5 | resetPasswordLimiter | Per account identifier | Prevent reset-link spam |

# 8. Action Handlers (Helper Layer)

This chapter documents all server actions inside the helper folder.

These actions are the core functional layer of the IdentityUser system, responsible for user creation, login flow, profile management, role/claim administration, concurrency control, and security stamping.

Each section below describes:

- The purpose of the action
- A clear step-by-step explanation of what the action does
- Important security and architectural notes

## 8.1. signUpFormAction (Signup)

**Purpose**

Creates a new user after validating the submitted data.

**Flow**

1. Validate the incoming form data.
2. Confirm that password and confirmPassword match.
3. Check whether the chosen username already exists.
4. Check whether the chosen email already exists.
5. Hash the password.
6. Create a new user.
7. Assign a default user type (if needed).
8. Return the username + password so NextAuth can auto-login the user after signup.

## 8.2. signInAction File

This file contains sign in, reset password, verify email/phone and 2FA management.

### 8.2.1. signInFormAction

**Flow**

1. Validate input data.
2. Check if username exist or not
3. Check if Password correct or not
4. Check If 2FA enable or not

**Note:**

Because NextAuth's built-in signIn() function only works on the client side, you cannot use it inside a server action to determine the authentication flow.

For this reason, the sign-in logic is handled manually on the server:

- We validate the user's credentials directly.
- We check whether the account exists and the password is correct.
- We detect whether Two-Factor Authentication (2FA) is enabled for the user.
- Based on these conditions, we return a structured response indicating the next required step.

This approach gives full server-side control over the login process and allows the application to decide whether it should proceed with normal login, request a 2FA code, or return an error — without depending on any client-side NextAuth behaviors.

### 8.2.2. canUserSignInAction

**Purpose**

Verifies whether the user is allowed to attempt signing in.

**Flow**

1. Check whether the user exists.
2. If found, check whether the account is currently locked out or not.

### 8.2.3. signInFailedAction

**Purpose**

Increases the failed login counter when the user enters invalid credentials.

**Details**

- The maximum allowed failure count is configurable.
- In the sample project the default value is 5 failures.

### 8.2.4. signInSuccessAction

**Purpose**

Resets login-related security fields when the user successfully signs in.

**Resets**

- accessFailed
- lockoutEnabled
- lockoutEnd

### 8.2.5. Forget password flow

#### *8.2.5.1. forgotPasswordRequestAction*

**Purpose**

Detect resets password base on email or phone otp

**Flow**

1. Validate input data.
2. Detect input type (phone or email).
3. Get client IP
4. Create Limiter keys
5. Check if the input is Email
    a. Check if user exists (returns always success for protection)
    b. User exists → generate token
    c. Apply rate limiters (always)
    d. Return data
6. Check if the input is Phone
    a. Check if phone exists (always hides true/false)
    b. User exists → generate OTP
    c. Check if an active OTP already exists (limit user spam)
    d. If user already has an OTP → calculate remaining time
    e. Apply rate limiters for phone + IP
    f. Return data

**Note:**

We always return a generic success message, regardless of whether the email or phone number actually exists in our system.

This is done for security reasons — to prevent attackers from using the “Forgot Password” or “OTP Login” endpoints to discover which accounts are registered.

By not revealing whether a user exists or not, we effectively block account enumeration attacks and keep user information private.

### *8.2.5.2. Reset password flow with email*

**Purpose**

Detect resets password base on email.

**Flow**

1. Run: **createEmailPasswordResetTokenAction**
2. GET HEADERS and IP.
3. Apply all rate limits.
4. Create Token.
5. Hash Token.
6. Delete any previous tokens.
7. Save token.
8. Create reset link.
9. Send email with **sendPasswordResetEmail**.

**Note:**

This flow uses Resend as the email delivery provider.
Resend offers 100 free emails per month, and additional usage requires upgrading to a paid plan.
When running locally, Resend can only send emails to the address associated with your API key, meaning you cannot send emails to arbitrary recipients during local development.
However, in a real production environment, once you verify your domain inside the Resend dashboard, you are allowed to send emails to any recipient without restrictions.

### *8.2.5.3. Reset password flow with phone*

**Purpose**

Detect resets password base on phone.

**Flow**

1. Run: **createPhonePasswordResetTokenAction**
    a. Get IP for rate-limiting (server-side)
    b. Apply rate-limits (global, ip, phone)
    c. Generate OTP
    d. Hash OTP.
    e. IMPORTANT: delete previous tokens using the same field name 'identifier' Save token.
    f. store new OTP record and return it.
    g. SMS raw OTP code to user smart phone (the logic of sending SMS should write here).

2. Run **verifyOtpAction** when user enter the otp code.
    a. Validate input.
    b. Find the OTP record for current Phone Number.
    c. If no token record => generic error (avoid enumeration)
    d. Check expiration
    e. expired -> delete record to allow new request later.
    f. Check attempts limit
    g. Compare OTP with hashedToken in DB
    h. use your helper comparePassword (makes bcrypt.compare). fallback to bcrypt if needed
    i. if invalid Increment attempts.
    j. OTP is valid -> create a RESET token (raw + hashed) and delete/mark OTP as used.
    k. remove all phone OTPs for this identifier (prevent reuse)
    l. Return raw reset token (used in redirect URL)

#### *8.2.5.4. resetForgetPasswordAction*

**Purpose**

After create and sending token in email or phone flow, we send user to reset password page and use this action to change password.

**Flow**

1. Validate input
2. Find ALL tokens that are not expired
3. Compare raw token with hashed tokens
4. Update user password
5. Delete the token so it cannot be reused

### 8.2.6. Email Verify Flow

Verify Email with this actions:

1. createEmailVerificationToken
2. sendVerifyTokenForEmail
3. verifyEmailToken

#### *8.2.6.1. createEmailVerificationToken*

**Flow**

1. Validate input data.
2. Check if user email already verify or not.
3. Check if the token exist or not.
4. Check the token expire or not if exist.
    a. If token expire then delete it.
5. Create new token.
6. Token storage with 24-hour expiration.

#### *8.2.6.2. sendVerifyTokenForEmail*

**Flow**

1. Sending token to user email

#### *8.2.6.3. verifyEmailToken*

**Flow**

1. Validate input data.
2. Find ALL tokens that are not expired.
3. Compare raw token with hashed tokens.
4. Update user to verify email.
5. Delete the token so it cannot be reused

### 8.2.7. Phone Verify Flow

Verify Phone with this actions:

1. creatPhoneVerificationOTP
2. verifyPhoneAction verifyEmailToken

#### *8.2.7.1. creatPhoneVerificationOTP*

**Flow**

1. Validate input data.
2. check if user phone already verify or not.
3. Check if the token exist or not.
4. Check the token expire of not if exist.
5. If token expire then delete it.
6. Create and Hash new otp.
7. store new OTP record
8. Write SMS API logic in this step
9. Return data

#### *8.2.7.2. verifyPhoneAction*

**Flow**

1. Validate input data.
2. Find the OTP record for current phone number.
3. If no token record => generic error (avoid enumeration).
4. Check expiration.
5. expired -> delete record to allow new request later.
6. Check attempts limit.
7. Compare OTP with hashedToken in DB.
8. Remove all phone OTPs for this identifier (prevent reuse)

### 8.2.8. Two-Factor Authentication (2FA) Overview

Two-Factor Authentication (2FA) adds an extra security layer on top of the standard username & password login.

Even if an attacker obtains the user's credentials, they cannot access the account without the second factor.

#### *8.2.8.1. Why TOTP?*

This package uses Time-based One-Time Passwords (TOTP) as the second authentication factor.

TOTP codes are:

- Generated on the user's device
- Valid for only 30 seconds
- Impossible to guess
- Do not require SMS or email delivery
- Work completely offline

Because TOTPs are generated locally using a shared secret key, they are significantly more secure and more reliable than email/SMS codes, which can be intercepted or delayed.

#### *8.2.8.2. How TOTP Works*

1. The server generates a unique secret key for the user.
2. The user scans a QR code using an authenticator app.
3. The app stores the secret and generates a new 6-digit code every 30 seconds.
4. During login, the user must enter the code displayed in the authenticator app.
5. The server verifies the code using the same secret.
6. This ensures that only the device holding the secret key can generate valid codes.

#### *8.2.8.3. What Is an Authenticator App?*
An authenticator app is a mobile or desktop application that supports TOTP generation.

Popular options include:

- Google Authenticator
- Microsoft Authenticator
- Authy
- 1Password
- LastPass Authenticator

The user simply scans a QR code once, and the app automatically produces valid codes forever (until 2FA is disabled).

#### *8.2.8.4. Recovery Codes*
Recovery codes are one-time passwords that allow users to log in when:

- They lose their phone
- They delete their authenticator app
- Their device is unavailable

Each recovery code can be used only once, and the user is responsible for saving them securely.

#### *8.2.8.5. generate2FASecretAction*
This action is first step to active to disable the 2FA.

**Flow:**

1. Validate input data.
2. Check if the user exists.
3. Case1: Disable 2FA if enabled
    a. Create new securityStamp and log out user.
4. Case 2: Enable 2FA → generate secret if not exist
    a. Return Redirect URL to confirm page for scan QR code.

### *8.2.8.6. generate2FASecretAction*

Generate a QR code for the user to scan using an authenticator app (Google Authenticator, Authy, etc.) after a TOTP secret has already been generated.

**Flow:**

1. Check if the user exists.
2. Check if the 2FA secret key exists or not.
3. Create URL with user email, your service name (here is IdentityUser Authenticator) and user TOTP secret key
4. Convert the URL to QR Code.
5. Return the QR Code to front end.

### *8.2.8.6. verify2FAAction*

User Scan the QR Code with the Authenticator app and enter the code to verify 2FA Enable.

**Flow:**

1. Validate input data.
2. Check if the user exists.
3. Compare the user input with secret key.
4. If valid Enable 2FA.
5. Create 10 recoveries Code and hash it.
6. Reset session with update securityStamp.
7. Return raw of recoveries code.

### *8.2.8.7. verifyLogin2FAAction*

Use this action for verify user in 2FA login

**Flow:**

1. Validate input data.
2. Check if the user exists.
3. Check if the TFA enable for the user.
4. Compare the user input with secret key to verify it.

#### *8.2.8.8. verifyRecoveryCodeAction*

If user  lose their phone, delete their authenticator app or device is unavailable use the recovery form.

**Flow:**

1. Validate input data.
2. Check if the user exists.
3. Check and compare user input with recoveries code.
4. Delete the recovery code matched with user input.
5. Return success to redirect to account page and create session.

## 8.3. UserAction File

This file contains multiple user-related management actions.

### 8.3.1. AddUserAction

**Flow**

1. Verify that the requester has the required claim/role to perform this action.
2. Validate input data.
3. Ensure the username is not already taken.
4. Ensure the email is not already taken.
5. Hash the password.
6. Create the user and retrieve the generated userId.
7. Assign claims to the user (if any were selected).
8. Assign a role to the user. (if any were selected).

**Note**

The sample implementation allows only one role per user, but the system structure makes multi-role support possible if needed.

### 8.3.2. UserUpdateAction

**Flow**

1. Verify that the requester has the required claim/role.
2. Validate input data.
3. Check whether the user exists.
4. Compare the submitted concurrencyStamp with the current one (to avoid conflicting updates).
5. Ensure the new username is not already taken.
6. Ensure the new email is not already taken.
7. Assign any updated roles or claims.
8. Build the updated user object.
9. If sensitive fields changed (password, role, claim, username, email), generate a new securityStamp, forcing logout on next request.
10. Generate a new concurrencyStamp.
11. Save the updated user.

### 8.3.3. deleteUserAction

**Flow**

1. Verify claim/role permission.
2. Validate input.
3. Check whether the user exists.
4. Remove user roles.
5. Remove direct user claims.
6. Delete the user account.

### 8.3.4. resetPasswordAction (Admin Panel Only)

**Flow**

1. Verify claim/role permission.
2. Validate input.
3. Check whether the user exists.
4. Hash the new password.
5. Update the password and assign a new securityStamp.

**Note**

This action does not check the old password — it is an admin-level override.

### 8.3.5. getAllUsersAction

Fetches all users along with their roles and claims.

### 8.3.6. getUserByIdAction

**Flow**

1. Retrieve user by ID.
2. Retrieve role list.
3. Retrieve direct user claims.
4. Combine role claims + direct claims and remove duplicates.
5. Return full user data.

### 8.3.7. getUserByUsernameAction

Same as **getUserByIdAction**, but queries by username.

### 8.3.8. getUserByPhoneNumberAction

Same as **getUserByIdAction**, but queries by PhoneNumber.

**Note:**

This Action should only be used when your project requires phone numbers to be unique.

### 8.3.9. getUserByEmailAction

Same as **getUserByIdAction**, but queries by Email.

### 8.3.10. changeNameAction (Profile)

**Flow**

1. Ensure the user has at least one claim/permission.
2. Validate input.
3. Check whether the user exists.
4. Compare the provided concurrencyStamp.
5. Update the name and assign a new concurrencyStamp.

### 8.3.11. changePasswordAction

**Flow**

1. Ensure the user has at least one valid claim.
2. Validate the submitted data.
3. Verify user existence.
4. Check the old password.
5. Verify new password matches confirmation.
6. Hash new password.
7. Update user password and assign a new securityStamp.

### 8.3.12. checkUserExistByUserNameAction

Checks whether the submitted username already exists.

### 8.3.13. checkUserExistByEmailAction

Checks whether the submitted email already exists.

### 8.3.14. checkUserExistByIdAction

Checks whether the submitted Id already exists.

### 8.3.15. checkUserExistByPhoneNumberAction

Checks whether the submitted PhoneNumber already exists.

**Note:**

Phone-number checks should only be used when your project requires phone numbers to be unique.

### 8.3.16. changeUserNameAction

**Flow**

1. Ensure the user has at least one valid claim.
2. Validate the submitted data.
3. Verify user existence.
4. Check new username not submitted already.
5. Update username and assign a new securityStamp.

### 8.3.17. changeEmailAction

**Flow**

1. Ensure the user has at least one valid claim.
2. Validate the submitted data.
3. Verify user existence.
4. Check new email  not submitted already.
5. Update email and assign a new securityStamp.

### 8.3.18. LockUnlockUserAction
Manually lock or unlock a user and assign a new securityStamp.

### 8.3.19. resetSecurityStampAction
Manually reset the security stamp of selected user.

### 8.3.20. getCurrentCCSAction

Returns the user's **concurrencyStamp**.

### 8.3.21. getUserByUsernameForSessionAction

Almost same as **getUserByIdAction**, but queries by username and get the necessary data of user for session.

## 8.4. RoleAction File

This file contains multiple role-related management actions.

### 8.4.1. roleAddAction

**Flow**

1. Verify claim/role permission.
2. Validate input.
3. Create the new role.
4. Retrieve the generated role ID.
5. Retrieve the selected claims.
6. Assign the selected claims to the role in RoleClaims.

### 8.4.2. roleUpdateAction

**Flow**

1. Verify claim/role permission.
2. Validate input.
3. Check role existence.
4. Compare the submitted concurrencyStamp.
5. Generate a new concurrencyStamp.
6. Assign a new claimStamp.
7. Update the role information.
8. Update RoleClaims according to the new claim selection.
9. Find all users who have this role.
10. Update each affected user with a new securityStamp.

### 8.4.3. deleteRoleAction

**Flow**

1. Verify claim/role permission.
2. Validate input.
3. Check role existence.
4. Retrieve the default fallback role (USER).
5. Find all users assigned to the role being deleted.
6. Reassign those users to USER, and update their securityStamp.
7. Remove RoleClaims for the deleted role.
8. Delete the role itself.

**Note**

The fallback logic is optional — developers may choose another strategy (e.g., assign no role).

### 8.4.4. getRolesAction

Returns all roles along with their claims.

### 8.4.5. getRoleByIDAction

Returns all roles Retrieves a role by ID along with all associated claims.with their claims.

## 8.4. ClaimAction File

This file contains multiple Claims-related management actions.

### 8.4.1. addClaimAction

**Flow**

1. Verify claim/role permission.
2. Validate input.
3. Create the new claim.

### 8.4.2. getClaimsAction

Returns all available claims.

### 8.4.3. deleteClaimsAction

**Flow**

1. Verify claim/role permission.
2. Validate input.
3. Check whether the claim exists.
4. Find roles that have this claim.
5. Find users who have this claim directly.
6. Remove claim entries from RoleClaims.
7. Remove claim entries from UserClaims.
8. Update claimStamp for affected roles.
9. Update securityStamp for all users who were impacted.

### 8.4.4. updateClaimsAction

**Flow**

1. Verify claim/role permission.
2. Validate input.
3. Check whether the claim exists.
4. Update the claim.
5. Find roles that contain this claim.
6. Update their claimStamp.
7. Find users who have those roles.
8. Find users who have the claim directly.
9. Update the securityStamp for all affected users.

## 8.5. SharedFunction File

**hashPassword**

Hashes a submitted password.

**comparePassword**

Compares a submitted password with the stored one.

## 8.6. Additional Notes

1. All permission checks inside actions are commented by default.

Developers can customize the required claim/role names to match their own system.

2. The sample project uses a claim-based approach, but developers can choose:
    - claim-based
    - role-based
    - hybrid model
    - or no authorization at all
    - depending on the needs of their application.
3. Many more actions can be added, such as:
    - forgotPassword
    - Two-factor authentication
    - Manual unlock/lock for users
    - Device tracking

These are not included in the base version but are easy to extend.

## 9. Code Samples & Usage Tips

### 9.1. Sign-In Form Component Logic (useEffect Example)

The following example is taken from the sign-in form component.

It demonstrates how the system handles automatic login, lockout protection, session sync between tabs, and fallback logic:

```
useEffect(() => {
  if (lastResult?.status === 'success') {
    if (hasPayload(lastResult)) {
      const { username, password } = lastResult.payload;
      (async () => {
        try {
          const logInAllow = await canUserSignInAction(username);
          if (logInAllow?.status === "success") {
            const res = await signIn("credentials", {
              username,
              password,
              redirect: false,
              callbackUrl: `/en/account/profile/${username}`,
            });
            if (res?.ok) {
              setSignInError(false);
              signInSuccessAction(username);
              try {
                const bc = new BroadcastChannel("auth");
                bc.postMessage({ type: "login", username, avatar: "/Avatar/Default Avatar.png" });
                bc.close();
              } catch (_) { }
              // fallback
              localStorage.setItem("auth-login", JSON.stringify({
                type: "login",
                username,
                avatar: "/Avatar/Default Avatar.png",
                time: Date.now()
              }));
              router.push(res.url || `/en/account/profile/${username}`);
            } else {
              signInFailedAction(username);
              setSignInError(true);
            }
          } else {
            setRemainingLockoutMinutes(Number(logInAllow?.message));
            setIsLockedOut(true);
          }
        } catch (error) {
          setSignInError(true);
        }
      })();
    }
  }
}, [lastResult]);
```

**Explanation**

- After form submission, if validation succeeds, the effect receives the username/password returned by signInFormAction.
- The component calls canUserSignInAction to check if the account is locked.
- If not locked:
    - We call NextAuth's built-in signIn() with the provided credentials.
    - On a successful login:
        - We call signInSuccessAction to reset lockout counters.
        - We broadcast a "login" message using BroadcastChannel and also store a fallback entry in localStorage.
        - This ensures all open browser tabs log in simultaneously without waiting for the session to refresh (NextAuth session may take a second to sync).

If login fails:

- signInFailedAction increments failed attempts.
- After a configurable limit (default: 5), the account is locked for one hour.

- The same architecture is used for the sign-up form.

## 9.2. Using requireAuth, requireGuest, and SessionWatcher

### 9.2.1. Example: Account Layout (Protected Pages)

```
export default async function AccountLayout({ children, params }) {
  const { locale } = await params;

  const session = await requireAuth(`/${locale}`);

  return (
    <>
      <AuthProvider>
        <SessionWatcher locale={locale} />
        <div className="flex flex-row justify-start items-center my-20 rounded-2xl">
          {children}
        </div>
      </AuthProvider>
    </>
  );
}
```

**Explanation**

- Pages under the account section should only be visible to authenticated users.
- requireAuth() checks the session on the server:
  - If no session exists → user is redirected to the home page.
- SessionWatcher must be placed inside AuthProvider and as the highest element in the tree.
- SessionWatcher continuously verifies session validity:
  - If the session becomes invalid, it signs the user out.
  - It triggers logout across all open tabs using both BroadcastChannel and localStorage

### 9.2.2. Example: SignIn / SignUp Page Layout (Guests Only)

```
export default async function SignInPage({ params }) {
  const { locale } = await params;

  await requireGuest(`/${locale}`);

  return (
    <>
      <AuthProvider>
        <SessionWatcher locale={"en"} />
      </AuthProvider>
    </>
  );
}
```

**Explanation**

- requireGuest() ensures this page is only visible when no session exists.
- If a logged-in user tries to access it, they are redirected away.
- Useful for login, registration, forgot-password, etc.

## 9.3. Tips for Permissions (hasClaim, hasAnyClaim, hasRole, hasAnyRole)

When checking permissions, always validate both on the frontend and backend.

**Why?**

- Frontend check controls UI visibility. (e.g., hide the "Edit Role" button if user lacks the required claim/role).
- Backend check protects security. (because UI can be bypassed and requests can be sent manually).

**Example scenario**

If editing a role requires the role.edit claim:

- The frontend hides the edit button if:

```
hasClaim(session, "role.edit")
```

- The backend action roleUpdateAction must also check:

```
if (!hasClaim(userClaims, "role.edit")) throw new Error("Access denied");
```

Never rely on frontend-only permission checks.

### 9.4. Additional Notes

- Many sample files in the project include comments to help developers understand the logic more easily.
- All permission checks in actions are commented by default.
- Developers can customize claim/role names and enable them as needed.
- The system can be designed:
    - claim-based
    - role-based
    - hybrid (recommended in most real-world apps)
    - or even no authorization layer, if not needed
- Additional actions can be added easily such as:
    - forgot password
    - two-factor authentication
    - manual lock/unlock
    - password reset via email
    - session invalidation
    - API rate limiting

    These are not included in the default version but can be implemented with the same architecture.

## 10. Compatibility

IdentityUser supports:
- Next.js 15+
- Node 18+
- React 18+
- TypeScript or JavaScript

Tested with Next.js 15 and 16.

### 10.1. Upgrade Note (Next.js 15 → 16)

If you want to upgrade an older Next 15 project, run:

```
npm install next@latest react@latest react-dom@latest
```

Then update your tsconfig.json or next.config.js if needed. Contact the author for step-by-step guidance when ready.

## 11. Contributing & Support

Contributions, issues, and feature requests are welcome. Please open an issue or a pull request on the repository once published.

## 12. Author & Contact

**Sadegh Shojayefard**
**GitHub:** https://github.com/SadeghShojayefard
**Website**: https://sadegh-shojayee-fard.vercel.app/
**Telegram:** https://t.me/link_lover1
**Email:** sadeghshojayefard@gmail.com

## 13. License

## 14. Support & Star

If you like this package, don't forget to star the GitHub repo once uploaded.