Authentication Feature Design Specification

Contents

General Authentication Design

Overview

This document describes the authentication architecture used in the system, focusing on token-based access control, session lifecycle, and frontend-backend separation of concerns. It is intended for developers and junior engineers who need to understand the session handling model and contribute to its implementation or maintenance.

The system uses **NextAuth.js** on the frontend solely to handle session visibility and session-triggered hooks. The backend, built on **Express.js**, fully owns the authentication logic, token issuance, renewal, and revocation. This separation allows tight security control in the backend while minimizing frontend complexity.

Token Types and Lifetimes

The system relies on three main token types:

Token	Purpose	TTL	Renewed By	Stored In
access_token	Authenticates API requests	15 mins	/renew	Database
refresh_token	Renews access token without login	30 days	/extend	Database
NextAuth JWT	Frontend session trigger	1 minute	Auto-regenerated	Browser/session

Access Token

- Issued after a successful login or token exchange.
- Used by the frontend to authenticate requests to the backend.
- Stored in memory or secure browser storage.
- Valid only if:
    1. Its **JWT signature is valid**
    2. It **has not expired**

3. It is **still present in the backend database**

Refresh Token

- Allows renewal of a new access_token without re-authentication.
- Stored and validated on the backend.
- Not automatically renewable — expires after 30 days.
- Can be renewed only through the /extend endpoint.

NextAuth JWT

- A short-lived session token (1 min) that triggers a jwt() callback on expiry.
- Used to check or refresh backend-issued access tokens automatically.
- Not used to authenticate backend API requests directly.

Roles of Frontend and Backend

Frontend Responsibilities

- Display login interface and trigger signIn().
- Store and use access tokens returned from the backend.
- Renew tokens automatically via NextAuth jwt() callback.
- Manually call /extend to prolong the session, if needed.
- Automatically redirect user to login on token failure or expiry.

Backend Responsibilities

- Handle login, token issuance, renewal, and revocation.
- Maintain a secure database of issued tokens (for revocation and validation).
- Provide a consistent session model for both credentials-based and SSO logins.
- Issue short-lived access tokens and long-lived refresh tokens.
- Verify all incoming tokens for validity and integrity.

Supported Authentication Endpoints

HTTP Method	Path	Description
POST	/api/auth/login	Login with email and password (credentials flow)
POST	/api/auth/:provider/login	Exchange SSO JWT (e.g., Google) for backend-issued access/refresh tokens

POST	/api/auth/renew	Renews access token using a valid refresh token
POST	/api/auth/extend	Renews both access and refresh tokens without reauthentication
POST	/api/auth/revoke	Revokes the session by removing tokens from the database
POST	/api/auth/logout	Alias for /api/auth/revoke
GET	/api/auth/verify	Verifies if an access token is valid (signature + presence + expiration)

Single Sign-On Flow (SSO)

When a user logs in using an SSO provider (e.g., Google), NextAuth receives the SSO credentials. If the provider is not credentials, the jwt() callback on the frontend automatically posts the received token to the backend's /:provider/login endpoint. The backend verifies the provider token and issues its own access and refresh tokens. This ensures that **all sessions, regardless of login method, are unified under the backend's session model**.

Security Considerations

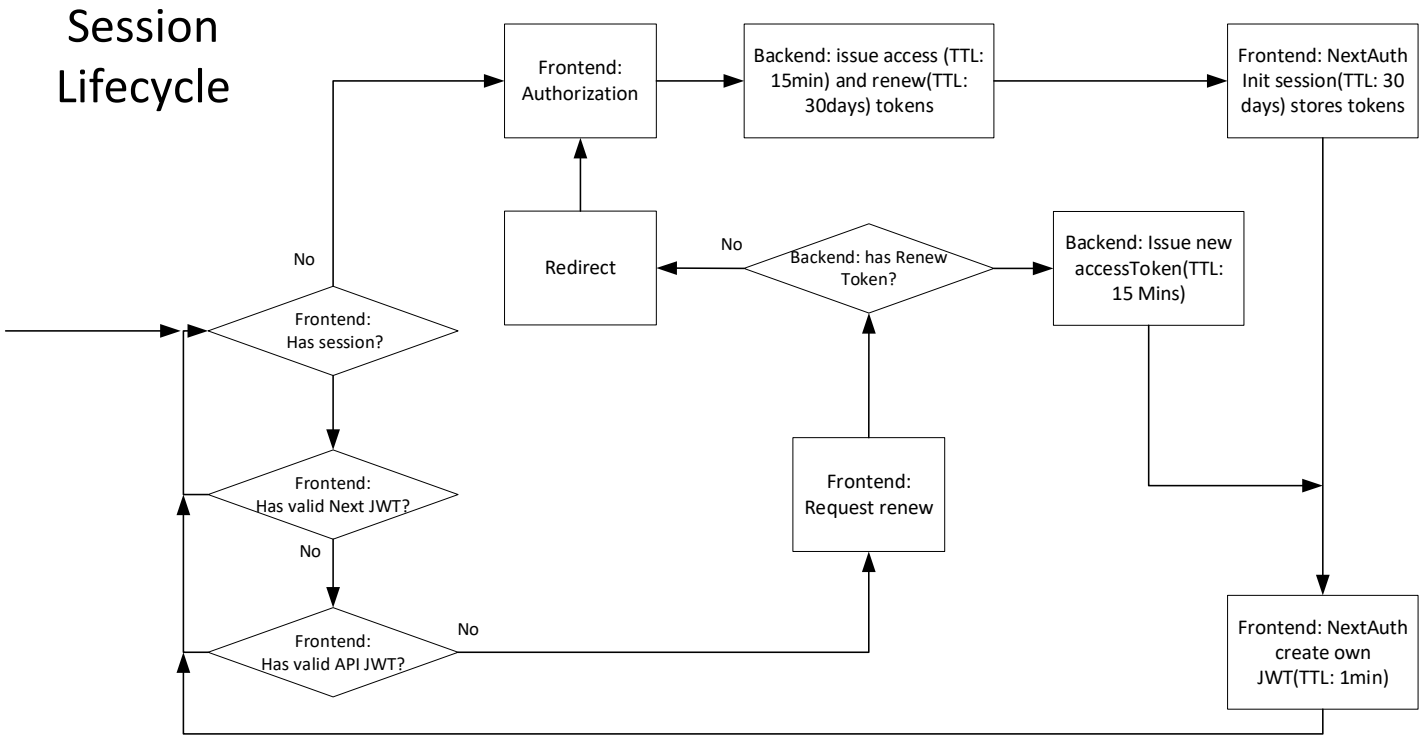
- All access and refresh tokens are stored in the database to allow revocation and introspection.
- Token validation checks include:
    - JWT **signature verification**
    - JWT **expiration timestamp**
    - Token **existence in database**
- The access token TTL is deliberately short (15 minutes) to reduce exposure if compromised.
- Sessions expire automatically after 30 days unless extended manually via /extend.
- Tokens are rotated securely, and old refresh tokens are invalidated on renewal.

Session Lifecycle

Overview

This chapter explains the full lifecycle of an authenticated session in the system, from login through expiration, renewal, and revocation. It covers how sessions are managed across both the frontend and backend, how tokens are validated and renewed, and what happens at each stage of the session timeline.

Diagram



Login and Session Initialization

When a user logs in, the frontend (typically via a signIn() call) triggers the authentication process.

- For **credentials login**, the frontend directly calls POST /api/auth/login.
- For **SSO login** (e.g., Google), the frontend authenticates via NextAuth, then forwards the provider's token to POST /api/auth/:provider/login.

If authentication succeeds, the backend responds with:

- access_token (15 min TTL)
- refresh_token (30 day TTL)

These tokens are securely stored (in memory, cookie, or session), and the session is now active.

Session Tokens and Storage

Upon successful login, tokens are stored as follows:

- The **frontend** holds:
    - access_token for API requests
    - refresh_token for token renewal
- The **backend**:
    - Stores both tokens in the database

- Associates them with the user session for validation and revocation

Automatic Access Token Renewal

To avoid user disruption, short-lived access tokens are automatically renewed.

- The **NextAuth JWT** has a TTL of 1 minute.
- Each time it expires, the jwt() callback is triggered.
- Inside this callback:
  - The frontend checks whether the current access_token is expired.
  - If expired, it sends POST /api/auth/renew with the refresh_token.
  - If the refresh token is valid, a new access_token is issued.

This automatic renewal continues until the refresh token expires (30 days).

Manual Session Extension

To prolong the session beyond the refresh token's lifetime without requiring the user to reauthenticate:

- The frontend can call POST /api/auth/extend before the refresh token expires.
- This endpoint:
  - Verifies the existing refresh token
  - Issues a **new pair** of access and refresh tokens
- The session's validity is now reset for another 30 days.

This should be triggered based on app-specific logic (e.g., user activity, "Remember me" setting, or periodic background extension).

Token Validation on API Requests

Every authenticated API request is validated on the backend using the access_token.

The backend performs the following checks:

1. **Signature** — the token must be correctly signed with the backend secret.
2. **Expiration** — the exp claim must not be in the past.
3. **Database Presence** — the token must still exist in the DB.

If any of these checks fail, the request is rejected with an appropriate error (e.g., 401 Unauthorized).

Session Revocation

Sessions can be forcefully terminated using:

- POST /api/auth/revoke
- or its alias: POST /api/auth/logout

This endpoint removes the access and refresh tokens from the database, effectively invalidating the session. Even if the JWTs have not yet expired, they become unusable because they no longer exist in the backend store.

This can be triggered:

- Explicitly by the user (logout button)
- Automatically (e.g., device ban, suspicious activity)
- Internally by admin action or policy

Session Expiration

A session expires automatically under two conditions:

1. The **access token** is expired and cannot be renewed.
2. The **refresh token** is expired or revoked.

Once both tokens are invalid, the frontend redirects the user to the login page to restart the authentication process.

Summary: Lifecycle States

State	Triggered By	Result
Login	POST /login or /:provider/login	Session tokens issued
Active Session	Valid tokens	User can access APIs
Auto Renewal	NextAuth jwt() callback	New access token via POST /renew
Manual Extension	App logic	New access + refresh tokens via POST /extend
Expiration	Refresh token expires or is revoked	Session invalid, reauthentication required

Revocation	User logout or admin action	Tokens deleted, session ends immediately

Login Flows

Overview

This chapter describes how users authenticate using either credentials (email/password) or an external provider (e.g., Google) and how both flows converge into a single, unified session model managed entirely by the backend. This ensures consistent token issuance, session logic, and user role handling regardless of the authentication method.

Credentials Login Flow

The credentials flow uses a custom form rendered on the frontend.

Steps:

1. **User enters login form** (email and password).
2. The frontend calls: signIn('credentials', { email, password })
3. NextAuth forwards these credentials to: POST /api/auth/login
4. The backend:
    - Validates the credentials.
    - If successful, responds with: { "accessToken": "...", "refreshToken": "..." }
5. The frontend, via the NextAuth jwt() callback:
    - Stores these tokens in the session context.
6. The user is redirected to the target page (defined in NextAuth.pages config).

Provider (SSO) Login Flow

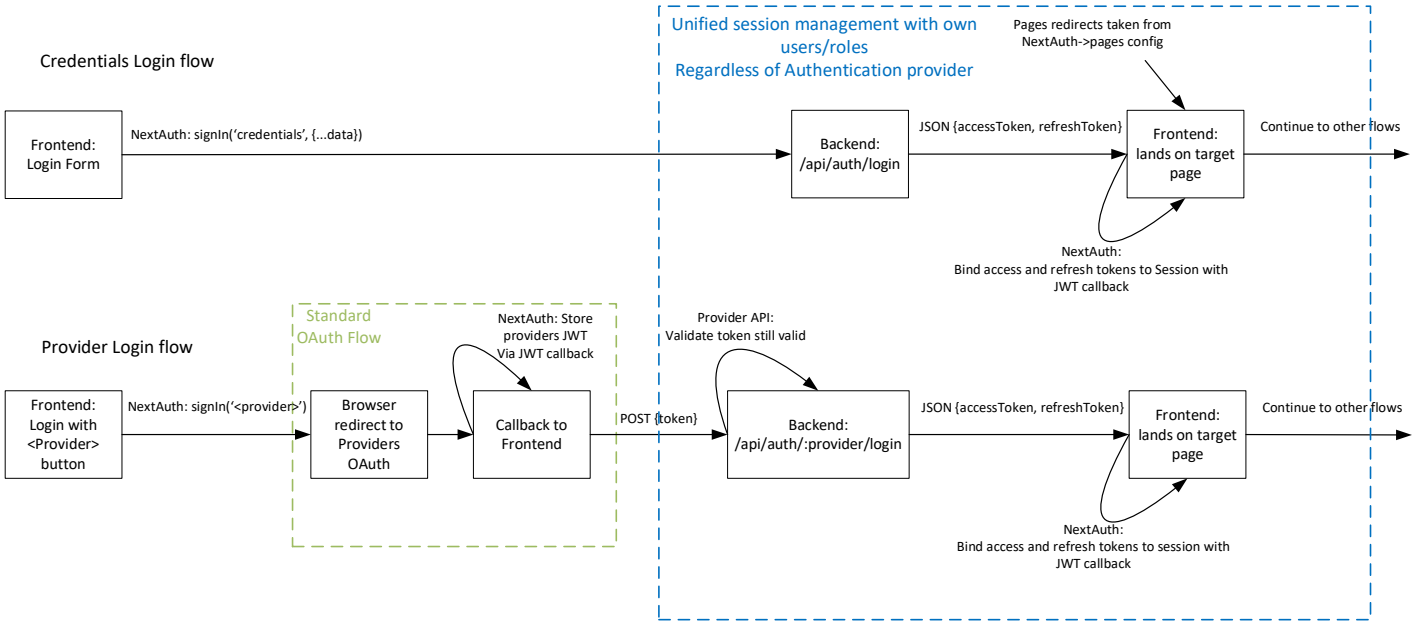
The provider flow uses the standard OAuth 2.0 redirect flow with token exchange after login.

Steps:

1. **User clicks "Login with Google"** (or another provider).
2. The frontend calls: signIn('google')
3. The browser is redirected to the provider's OAuth consent screen.
4. After successful login, the provider redirects back to the frontend.
5. In the NextAuth jwt() callback, the provider's token is extracted.
6. The frontend calls: POST /api/auth/google/login with {"token": "…"} payload
7. The backend:
    - Verifies the token with the provider.
    - If valid, returns: { "accessToken": "...", "refreshToken": "..." }
8. NextAuth stores these backend tokens into the session context
9. The user is redirected to the target page.

Unified Login Flow Diagram

Refer to the diagram for a visual breakdown of both login flows and the convergence point.



Key Notes from the Diagram:

- Both flows land at the **same backend interface** for issuing accessToken and refreshToken.
- The frontend handles storing those tokens in session via the **NextAuth jwt() callback**.
- Redirection is handled by **NextAuth pages config**, so both flows land on the same app page post-login.

Summary: Flow Comparison

Step	Credentials Login	Provider Login (e.g., Google)
Trigger	signIn('credentials')	signIn('google')
Backend entrypoint	POST /api/auth/login	POST /api/auth/google/login
Token validation	Against local DB	Against provider API
Token issuance	accessToken, refreshToken from backend	accessToken, refreshToken from backend
Session binding	In jwt() callback	In jwt() callback
Target redirection	NextAuth pages config	NextAuth pages config

Signup Flows

Overview

This chapter describes the two supported signup flows: standard form-based signup and third-party provider (SSO) signup. Regardless of the method, both flows converge into a unified session model managed by the backend. The backend issues all tokens, handles session ownership, and ensures consistent user management across login sources.

Standard Signup Flow (Email + Password)

This flow is initiated when a user completes the signup form on the frontend.

Steps

1. User completes and submits the registration form.

2. The frontend calls: POST /api/register/signup

3. The backend:

- Stores the user in a temporary users-pre table with a TTL of 1 day.
- Sends a confirmation email with a secure token link.
- Optionally sends a rejection link if needed for UX purposes.

4. The frontend redirects to a pending page (/register/pending) where the user is informed to check their email.

- This page also tracks state using NextAuth to prevent multiple tabs from completing registration simultaneously.

Confirmation or Rejection

Confirmation

1. User clicks the confirmation link (e.g. /api/register/confirm?token=...).
2. The backend:
    - Moves the record from users-pre to the real users table.
    - Issues accessToken and refreshToken.
3. Redirects to frontend: /register/confirm?accessToken=...&renewToken=...
4. The frontend calls: signIn('token', { accessToken, refreshToken })
5. Session is initialized and the user is redirected to the homepage.

Rejection

If the user clicks the rejection link, the backend removes the temporary record via: POST /api/register/reject?token

SSO Signup Flow (Third-Party Providers)

This flow uses OAuth 2.0 with identity providers (e.g., Google) to register users.

Steps:

1. User clicks “Sign up with Google” or similar.

2. Frontend calls: signIn('google')

3. Standard OAuth 2.0 redirect flow:

    - Redirects to Google’s login screen.
    - Returns to frontend with tokens.

4. The frontend calls: POST /api/register/google/confirm

5. Backend:

    - Validates the provider JWT and retrieves identity info (via /userinfo endpoint).
    - Creates new user in the users table.
    - Issues accessToken and refreshToken.
    - Redirects to: /register/confirm?accessToken=...&renewToken=...

6. Frontend: signIn('token', { accessToken, refreshToken })
7. Session is created and the user is logged in.

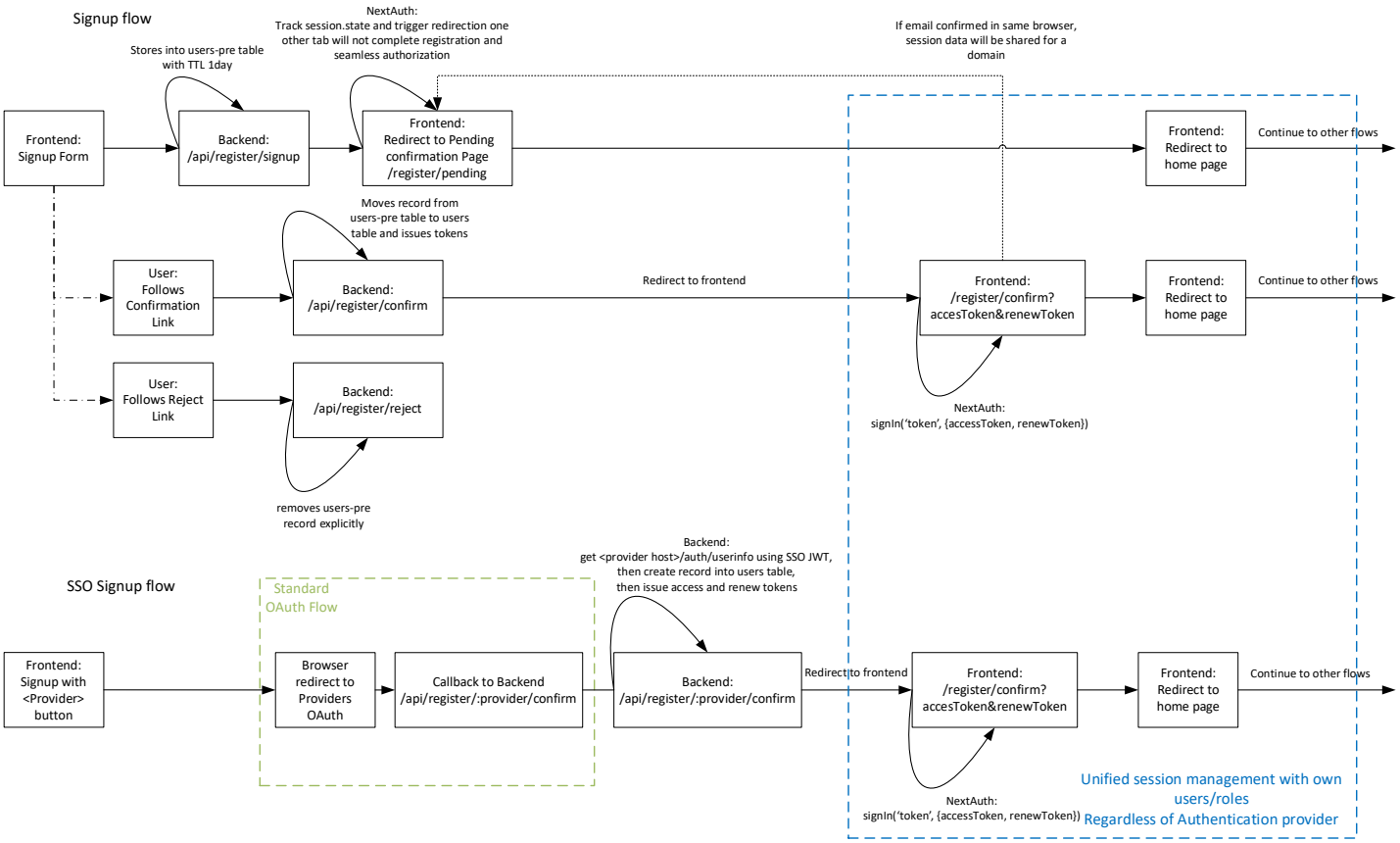
Email Verification for SSO

Unlike email/password signup, SSO signup skips email confirmation. This is intentional and secure:

- It is assumed that trusted third-party providers (e.g., Google, GitHub) have already verified the user’s email.
- The backend may still choose to verify the email_verified flag from the /userinfo response for extra assurance.

Unified Signup Flow Diagram

Refer to the diagram above to see how both the standard and SSO flows converge at:



- signIn('token', { accessToken, refreshToken })
- Backend-issued session tokens
- Unified session storage and redirection to the app homepage

NextAuth Configuration Note

To enable both standard login and final session binding via backend-issued tokens, two CredentialsProviders must be configured in NextAuth:

1. credentials — for standard login (/api/auth/login)

2. token — used by the signup flow to complete the session after token-based redirect

Example:

```
CredentialsProvider({

  id: 'credentials',
```

```
  name: 'Email/Password',
  authorize: async (credentials) => { /* call /api/auth/login */ }
}),
CredentialsProvider({
  id: 'token',
  name: 'Token Bootstrap',
  authorize: async (credentials) => {
    // Accept backend-issued tokens and inject into session
    return { accessToken: credentials.accessToken, refreshToken: credentials.refreshToken }
  }
}),
```

This second provider allows short-circuiting a login from a secure redirect without repeating user input.

Summary

Flow Type	Email Confirmed?	User Record Created In	Tokens Issued By	Flow Ends With
Form Signup	✅ via email	users table	Backend	signIn('token', { accessToken... })
SSO Signup	✅ by provider	users table	Backend	signIn('token', { accessToken... })

Both flows unify under the same session management system, ensuring consistent access control, user roles, and token renewal behavior.

Password Reset Flow

Overview

This chapter outlines the process of resetting a user's password through the web application. The flow supports both standard users and users who originally signed up via SSO. Resetting a password results in a new backend-managed session and does not break the unified session management model. If an SSO user resets their password, they can subsequently log in using either SSO or the new password via credentials.

Password Reset Request

Steps:

1. The user clicks the "Forgot Password" button on the frontend.

2. They are redirected to a reset form at /auth/forgot.

3. After entering their email, the frontend calls: POST /api/auth/reset/init

Backend logic:

1. Verifies that the user exists (regardless of whether they registered via credentials or SSO).
2. Generates a one-time resetToken.
3. Sends an email to the user with:
    - A reset link (redirects to reset form)
    - A reject link (optional - used to cancel the reset process)

Reset Link Flow

1. The user follows the reset link.
2. They are redirected to the reset password form at: /auth/reset
3. user entering a new password(twice)
4. The frontend submits the following request: POST /api/auth/reset/finalize with payload { id, resetToken, password }
5. Backend verifies the reset token and:
    - Stores the new password securely
    - Issues a new accessToken and refreshToken
    - Redirects to the frontend with: /register/confirm?accessToken=...&renewToken=...
6. The frontend

    - Calls signIn('token', { accessToken, refreshToken })
    - The session is created and the user is redirected to the home page

Reject Link Flow

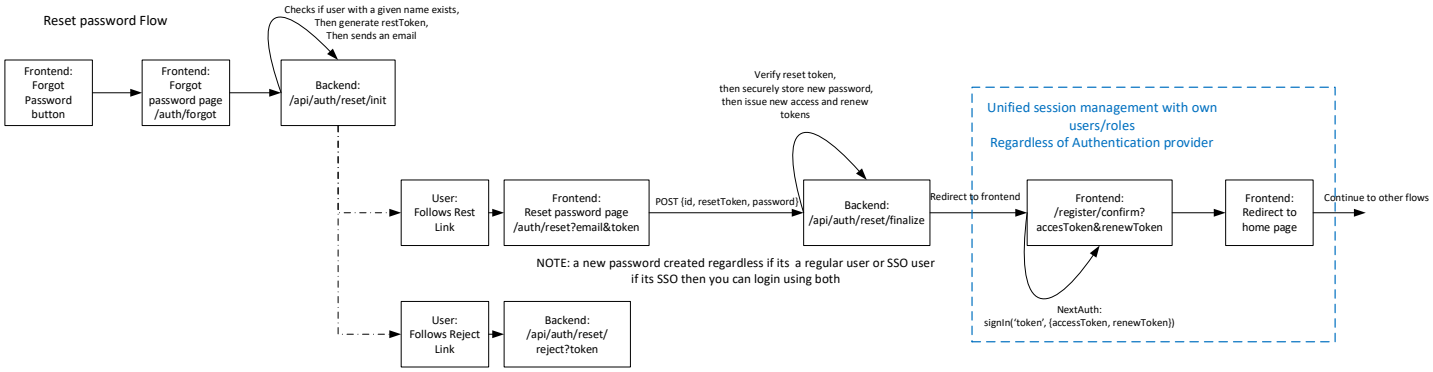
If the user follows the reject link (e.g. for security or cancellation), the frontend or email opens:

GET /api/auth/reset/reject

The backend verifies and invalidates the reset token, ensuring that the reset link can no longer be used.

Flow Diagram

This flow concludes in the same way as a successful signup:



- A valid backend-managed accessToken and refreshToken are issued
- signIn('token', ...) finalizes the session
- The user is redirected to the homepage

The session is now active regardless of whether the user originally used SSO or credentials.

Summary

Scenario	Input Method	Verification	Outcome
Password Reset Request	Email	resetToken (email)	Tokens issued and session active
Reset Rejected	Email	resetToken (email)	resetToken invalidated
Applies To	All users	Any login origin	SSO and standard users supported

Note: If an SSO user resets their password, they can subsequently log in using either method.

Database & Models Implementation

Overview

This chapter describes the MongoDB schema layout and Mongoose model configuration used to support the authentication and session lifecycle system. The models are designed to support secure access, expiration, and lifecycle transitions such as sign-up confirmation, login, reset, and session renewal.

The database stack consists of MongoDB with Mongoose ODM in the backend.

Required Collections

The following collections are required:

1. users - Stores registered users (via credentials or SSO)

2. users-pre - Temporary collection for unconfirmed signup records; TTL: 1 day
3. accessTokens - Stores active access tokens for validation; TTL: 15 minutes
4. renewTokens - Stores valid refresh tokens; TTL: 30 days
5. resetTokens - Stores one-time reset password tokens; TTL: 30 minutes
6. emailVerificationTokens - Stores email confirmation tokens TTL: 1 day

Each of these collections uses TTL indexing to ensure that expired tokens or pending records are automatically removed by MongoDB.

Minimal Required Fields (per Collection)

users

- _id
- email (unique)
- passwordHash (nullable if SSO)
- provider (credentials | google | github | etc.)
- createdAt, updatedAt

users-pre

- _id
- email
- passwordHash
- createdAt

accessTokens

- _id
- userId (ref: users)
- token (signed JWT string or opaque)
- createdAt

renewTokens

1. _id
2. userId (ref: users)
3. token
4. createdAt

resetTokens

- _id
- userId (ref: users)
- token

- createdAt

emailVerificationTokens

- _id
- userId or email
- token
- createdAt

Implementation Examples

On save hooks

```
import mongoose from 'mongoose';
import bcrypt from 'bcrypt';

const UserSchema = new mongoose.Schema({
  email: { type: String, unique: true, required: true },
  passwordHash: { type: String },
  provider: { type: String, required: true },
  createdAt: { type: Date, default: Date.now },
}, { timestamps: true });

UserSchema.pre('save', async function (next) {
  if (this.isModified('passwordHash')) {
    const salt = await bcrypt.genSalt(10);
    this.passwordHash = await bcrypt.hash(this.passwordHash, salt);
  }
  next();
});

export default mongoose.model('User', UserSchema);
```

TTL definitions

```
const ResetTokenSchema = new mongoose.Schema({
  userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
  token: { type: String, required: true },
  createdAt: { type: Date, default: Date.now, expires: 60 * 30 }, // 30 mins
});


export default mongoose.model('ResetToken', ResetTokenSchema);
// The expires field value is in seconds
// MongoDB will automatically remove the document 30 minutes after createdAt
```

Summary

Collection	TTL Duration	Notes
users	N/A	Permanent
users-pre	1 day	Deleted if not confirmed
accessTokens	15 minutes	Short-lived auth
renewTokens	30 days	Long-term session extension
resetTokens	30 minutes	For password recovery
emailVerificationTokens	1 day	Used for signup email confirm

These models enforce secure session and user data lifecycle handling without relying on cron jobs or manual cleanup.

Extension Possibilities

Overview

While the current authentication system covers core functionality — including login, signup, session lifecycle, password reset, and token management — the architecture is intentionally designed to support further extensions without disruption.

This chapter outlines three possible extensions that reuse or extend existing infrastructure with minimal changes.

Change Email Flow

Conceptually, changing a user's email address mirrors the signup email confirmation process:

1. The user requests an email change via a new endpoint: POST /api/auth/change-email/init
2. The backend:
    - Sends a confirmation link to the new email
    - Stores the request in a new emailChangeToken collection (TTL: 1 day)
3. The user follows the confirmation link: GET /api/auth/change-email/confirm?token=...

The backend:

- Verifies the token
- Updates the email field in the users collection
- Deletes the token record

This reuses the same security model and flow as emailVerificationTokens and can be implemented with minimal additional logic.

Two-Factor Authentication (2FA)

2FA can be added to both credential and SSO-based logins, leveraging NextAuth's redirect support and middleware logic.

Credential-based flow:

- User logs in via POST /api/auth/login
- If 2FA is enabled, backend responds with a 2FA required redirect:
- Redirect to /api/auth/2fa/init
- Frontend handles this like a login continuation:
- User completes TOTP or code-based auth
- On success, POST /api/auth/2fa/login resumes the session
- Backend returns accessToken and refreshToken
- Frontend finalizes with signIn('token')

SSO-based flow:

- If SSO provider configured to use 2FA it will invoke it automatically during OAuth flow

This adds no disruption to current token or session models.

Service-to-Service Authentication (Machine Clients)

For internal service or third-party client integration, a standards-based protocol can be layered onto the system.

Options:

1. OAuth2 Token Endpoint
    - Expose a client credentials grant at: POST /api/oauth/token
    - Clients authenticate using client_id and client_secret to get access tokens with specific scopes or role bindings.
2. OIDC Provider Integration

Deploy a dedicated OIDC provider (such as node-oidc-provider) under a separate route: e.g. /api/oidc

This allows external systems to authenticate using OIDC-compatible flows while still issuing backend-managed tokens. Tokens are linked to machine users or client records stored in the users collection with special flags.

Both options integrate cleanly into the current token validation and revocation model.