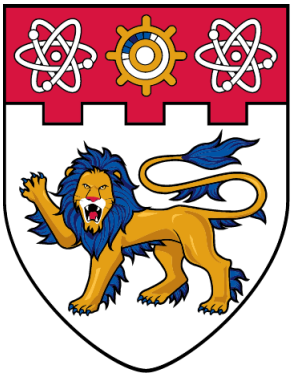
NANYANG TECHNOLOGICAL UNIVERSITY
COLLEGE OF COMPUTING AND DATA SCIENCE



NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Full Stack Development and Deployment of Web-based Speech Recognition System

By
NGUYEN PHUONG LINH

Project Supervisor: Prof Chng Eng Siong

Singapore
Academic Year 2025/2026

Abstract

This project focuses on the design and implementation of the **Gateway Dashboard**, a comprehensive management interface for the Speech Gateway API, alongside critical backend enhancements to support secure authentication and monetization.

On the **frontend**, a responsive Single Page Application (SPA) was developed using **Vue.js** and **Vuetify**, featuring a modular architecture that ensures scalability and maintainability. Key implemented features include a secure authentication system supporting **Email/Password**, **Google OAuth 2.0**, and **Apple Sign-In**, enhancing user accessibility and security.

The **backend**, built with **NestJS**, was significantly upgraded to include a **Token Management Module** ensuring session security through token blacklisting and automatic expiration handling using Redis and MongoDB TTL indexes.

Furthermore, a **Subscription and Payment Module** was integrated using **Stripe**, enabling automated recurring billing, plan management, and real-time subscription status tracking. Storage solutions utilizing **MongoDB** for persistent data and **Redis** for caching were implemented to optimize performance.

The result is a **production-ready dashboard** that facilitates seamless user onboarding, secure identity management, and a flexible subscription model, thereby establishing a solid foundation for the commercialization of the Speech Gateway services.

Acknowledgement

I would like to express my deepest gratitude to my supervisor, **Chng Eng Siong**, for his invaluable guidance, continuous encouragement, and expert advice throughout the course of this Final Year Project. His insights were instrumental in shaping the direction and reliability of this system.

I am also deeply thankful to my mentors **Kyaw Zin Tun**, **Vu Thi Ly**, and **Surana Tanmay** for their dedicated support and technical mentorship. Their practical knowledge and hands-on guidance greatly contributed to the success of this project.

I extend my sincere thanks to the **Hardware and Embedded Systems Lab** team for providing the necessary resources and technical environment to develop this project. The opportunity to work with real-world technologies like NestJS, Vue.js, and cloud infrastructure has been an immensely rewarding learning experience.

I am also grateful to my friends and family for their unwavering support and patience during the intense periods of development and debugging. Their belief in my abilities kept me motivated to overcome the technical challenges encountered along the way.

Finally, I would like to thank **Nanyang Technological University** for providing the academic foundation that made this work possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In the rapidly evolving landscape of speech processing technologies, the **Speech Gateway API** serves as a critical infrastructure for delivering advanced speech services to various applications. **Speech Gateway** is a backend service that provides Speech-to-Text (STT) capabilities to end users through a secured API. It acts as a **middle layer** between client applications and an underlying Speech Engine. Instead of allowing clients to call the speech engine directly, Speech Gateway centralizes authentication, access control, token issuance, and (in this project) subscription/payment management.

As the demand for these services grows, the need for a robust, centralized management system becomes paramount. Traditionally, interacting with such APIs required direct integration and manual management, which can be inefficient for end-users and administrators alike. The **Gateway Dashboard** project was initiated to bridge this gap, providing a user-friendly graphical interface that empowers users to manage their speech processing tasks while giving administrators the tools needed to oversee system operations, manage users, and monetize services effectively.

1.2 Problem Statement

Prior to this project, the Speech Gateway ecosystem faced several challenges:

1. **Limited Access Control:** The existing system lacked a flexible authentication mechanism, supporting only basic credentials without modern Single Sign-On (SSO) capabilities.

2. **Absence of Monetization Infrastructure:** There was no automated system to handle subscriptions, billing, or tiered access plans, hindering the commercial viability of the services.

1.3 Project Objectives

The primary objective of this project is to design and implement a comprehensive **Gateway Dashboard** and enhance the backend infrastructure to support commercial-grade features. Specific objectives include:

- **SSO Integration:** Implement secure Single Sign-On (Google, Apple) to streamline user authentication, reduce credential fatigue, and centralize identity management across services.
- **Payment Portal:** Design a subscription and payment module integrating Stripe to handle recurring billing, automated invoicing, and subscription lifecycle management.
- **Frontend Development:** Develop a web-based dashboard using Vue.js and Vuetify to provide an intuitive interface for users to access speech services and manage their data.

1.4 Project Scope

The project scope encompasses three major components: Single Sign-On (SSO) Authentication, Payment & Subscription Management with Stripe Integration, and Frontend Dashboard Development.

1.4.1 Single Sign-On (SSO) Authentication

This component enables users to authenticate using their existing social accounts or traditional credentials.

Key Features:

- **Google SSO:** OAuth 2.0 integration with state management.
- **Apple SSO:** Apple Sign In integration with token verification and privacy-preserving email relay.
- **JWT Token Management:** Generation and validation of access tokens (15-minute expiry) and refresh tokens with secure cookie storage (HttpOnly, Secure, SameSite).
- **Token Blacklisting:** Redis-based blacklist for immediate token revocation on logout or security events.
- **Traditional Authentication:** Email/password authentication with bcrypt hashing.
- **Session Security:** XSS prevention and secure token lifecycle management.

1.4.2 Payment Portal with Stripe Integration

This component handles monetization through Stripe, enabling tiered subscription plans with usage-based quotas.

Key Features:

- **Stripe Checkout:** Hosted checkout session creation and payment processing.
- **Webhook Handling:** Secure webhook signature verification and idempotent event processing for payment lifecycle events.

- **Plan Management:** JSON-based plan configuration with versioning support and admin API.
- **Redis Caching:** Fast plan retrieval and Stripe price data caching with automatic cache invalidation.
- **Subscription Synchronization:** Real-time sync between Stripe and internal database for subscription status.
- **Quota Management:** Usage tracking and access control middleware enforcing plan limits.
- **Payment UI:** Plan comparison cards, checkout flow, and subscription management interface.

1.4.3 Frontend Dashboard

This component provides a complete Single Page Application (SPA) using Vue.js 2 and Vuetify with Material Design.

Key Features:

- **Authentication UI:** Sign In, Sign Up pages with SSO integration and OAuth callback handling.
- **Protected Routing:** Vue Router with role-based navigation guards and lazy loading for performance.
- **State Management:** Vuex store for centralized state management (auth, user, subscription, plans).
- **HTTP Interceptors:** Automatic JWT token injection, token refresh logic, and retry mechanisms.
- **User Dashboard:** Subscription status display, usage quota tracking with progress indicators, profile management.
- **Admin Interface:** Plans management with JSON editor, validation, and version history.
- **Responsive Design:** Mobile and desktop layouts with Vuetify components and intuitive UX.

1.5 Proposed System Overview

This section presents the three main components of the proposed system, comparing the workflow before and after implementation.

1.5.1 Single Sign-On (SSO) Authentication

Before: Users needed to create separate credentials, leading to credential fatigue and security risks from weak or reused passwords.

After: The implemented flow works as follows:

1. User clicks on Google or Apple SSO button on the sign-in page.
2. System redirects to the OAuth provider for authentication.
3. Upon successful authentication, the provider returns user identity.
4. System creates or links the user account and issues JWT tokens.
5. User is redirected to the dashboard with active session.

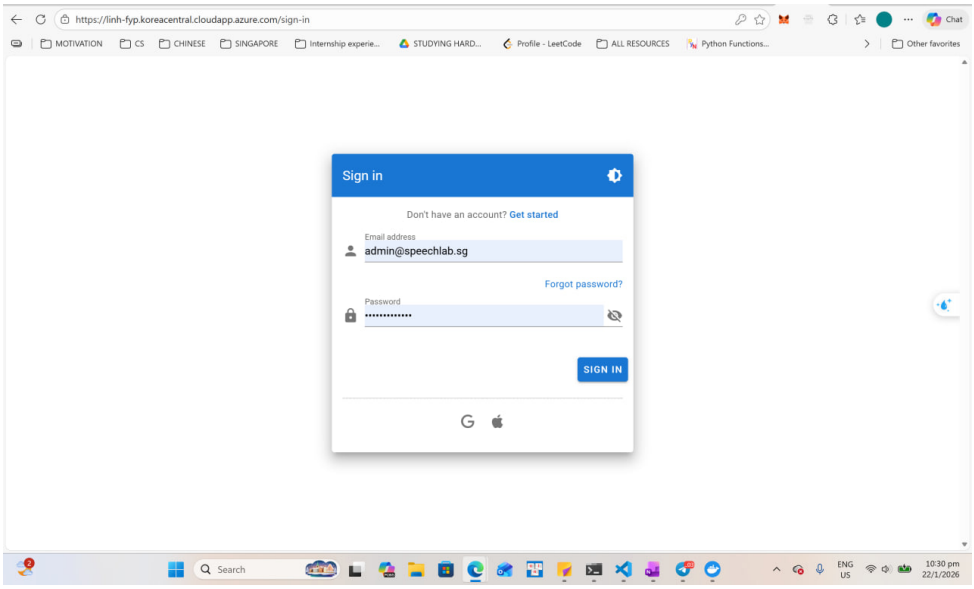


Figure 1.1: Sign-In Interface with SSO Options

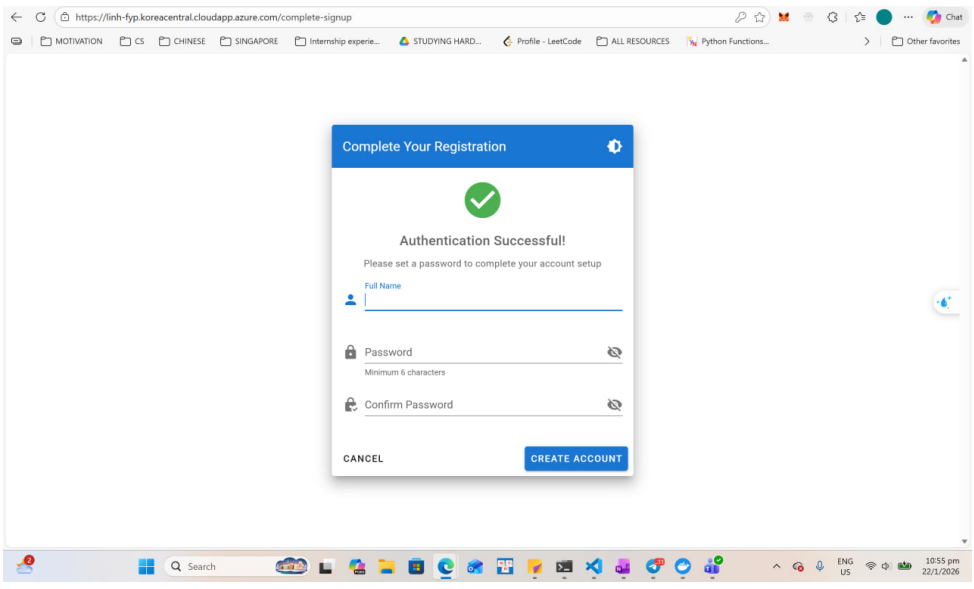


Figure 1.2: Registration Interface with SSO Integration

1.5.2 Stripe Payment Integration

Before: No automated billing, requiring manual invoice generation and payment verification before granting access.

After: The implemented payment flow works as follows:

1. Admin configures subscription plans in Stripe product catalog.
2. User selects a plan and clicks subscribe.
3. System creates a Stripe checkout session and redirects user.
4. User completes payment on Stripe's hosted checkout page.
5. Stripe sends webhook notification upon successful payment.
6. System updates subscription status and grants access to features.

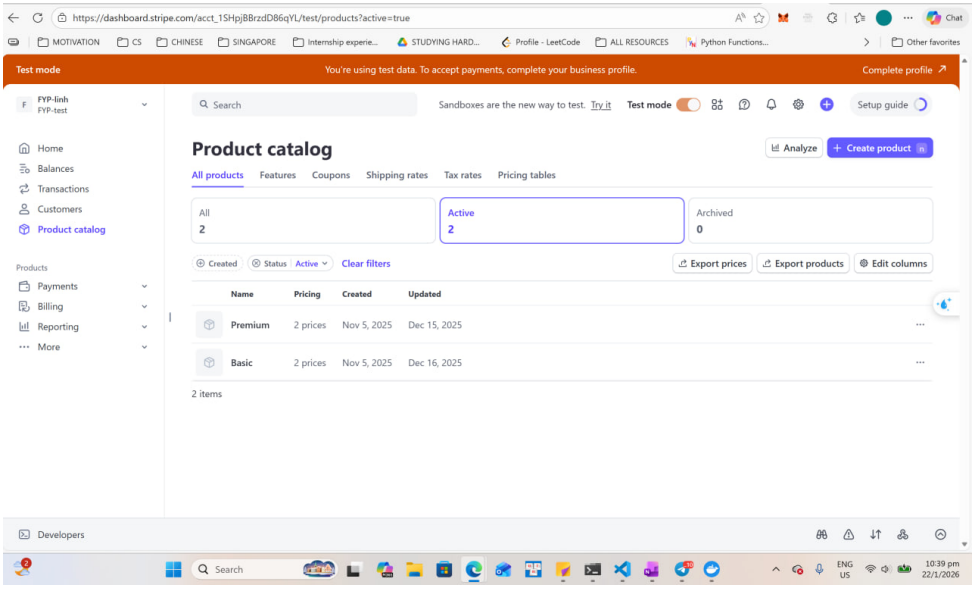


Figure 1.3: Stripe Product Catalog Configuration

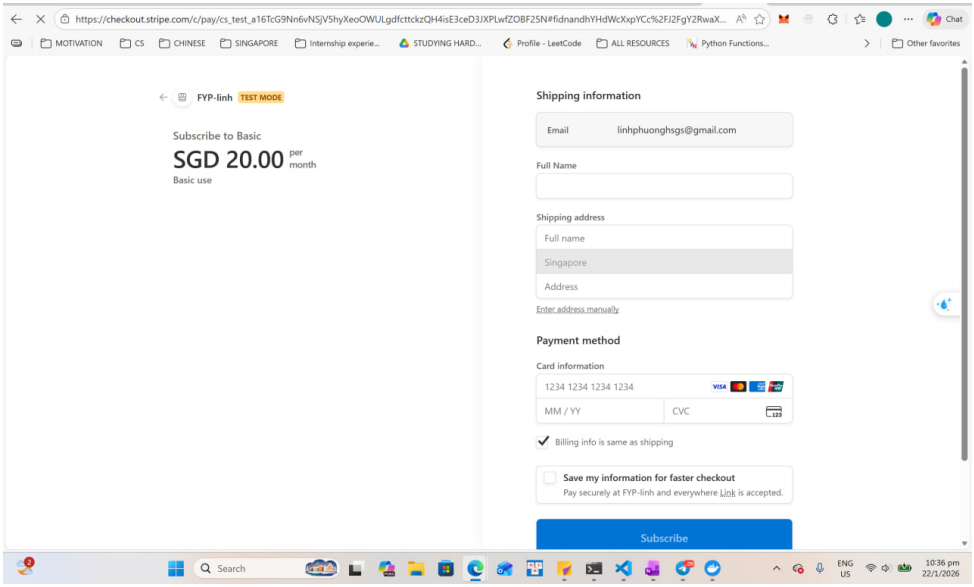


Figure 1.4: Stripe Checkout Interface for Basic Plan

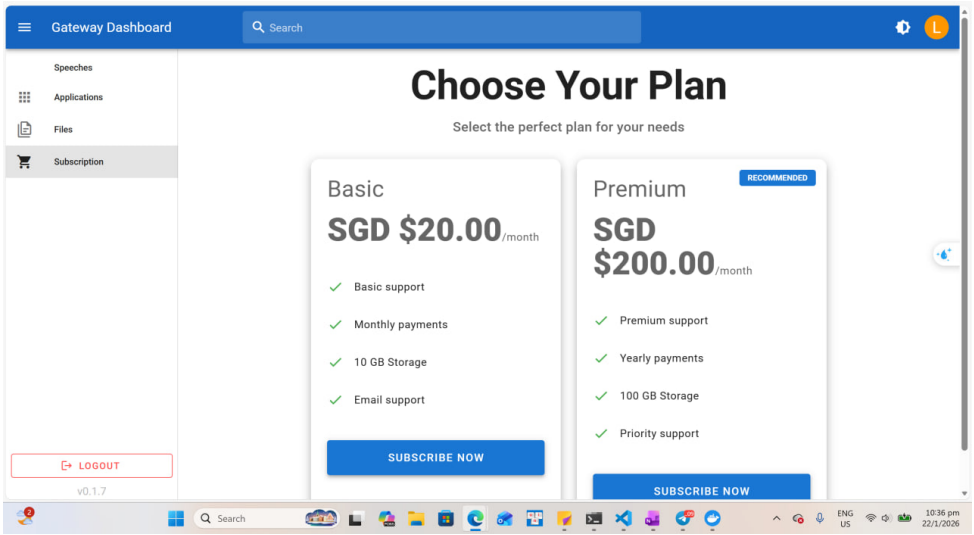


Figure 1.5: Subscription Management Interface for Users

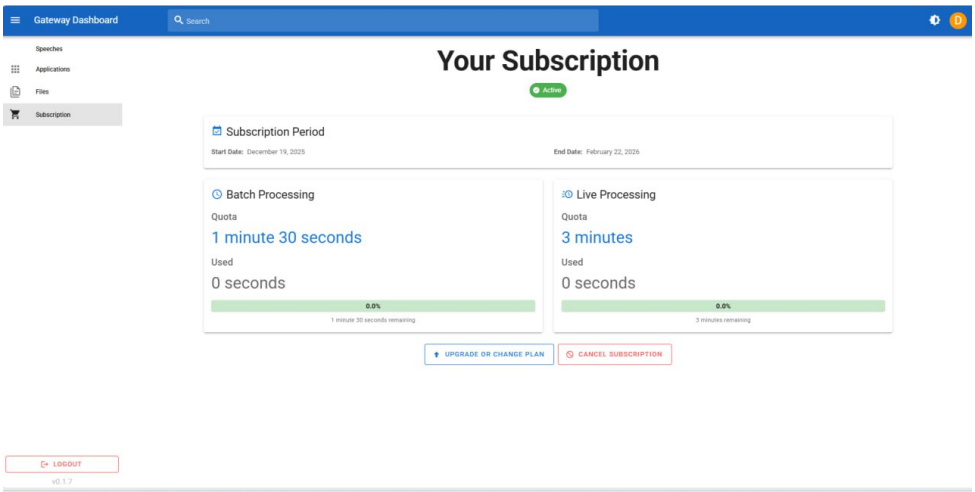


Figure 1.6: Successful Subscription Confirmation

1.5.3 Frontend Dashboard

Before: No visual interface, requiring direct API calls and making it difficult to monitor subscriptions or manage users.

After: The implemented dashboard provides a complete user flow:

1. User authenticates via SSO or traditional login.
2. Dashboard displays subscription status and usage quotas.
3. User can manage profile, view transactions, and upgrade plans.
4. Admin can manage users, configure plans, and monitor subscriptions.

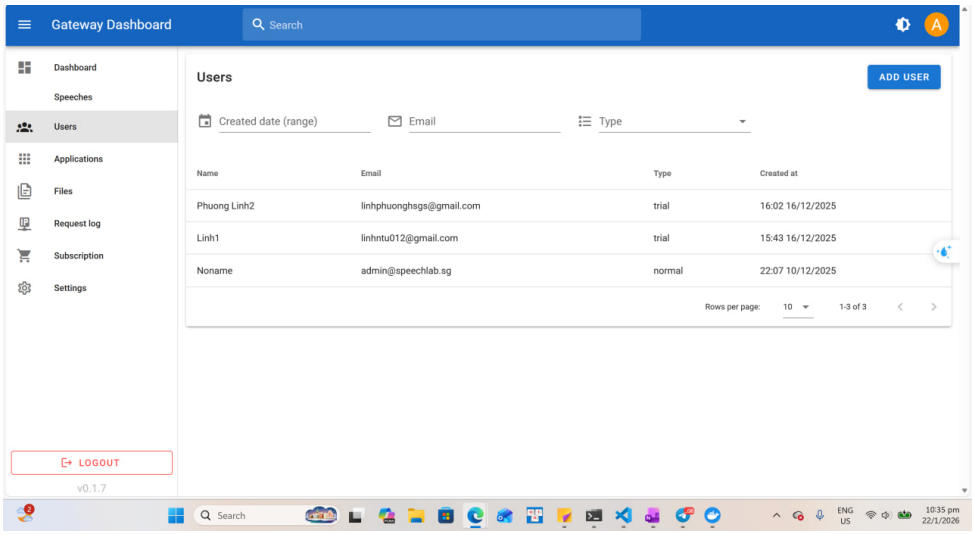


Figure 1.7: User Management Dashboard

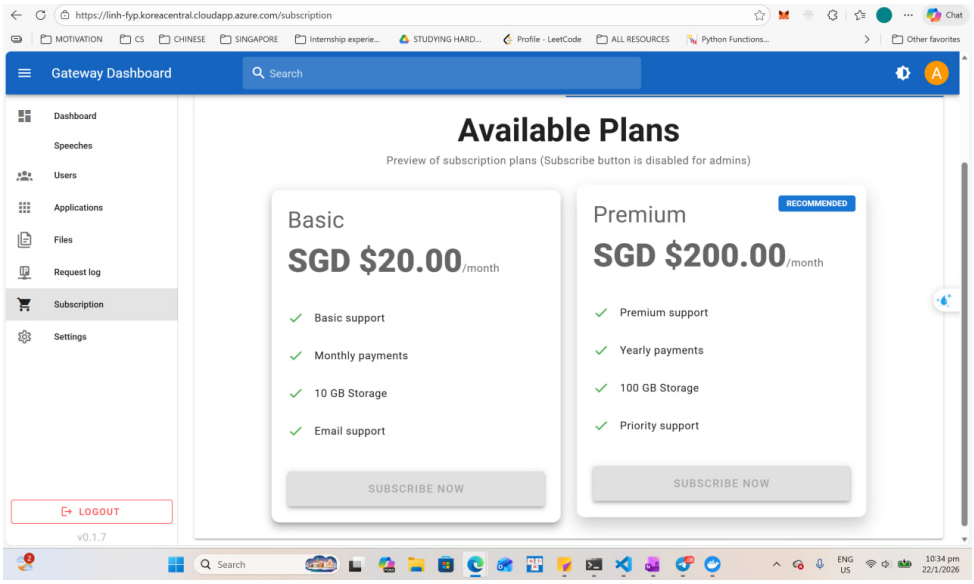


Figure 1.8: Admin Subscription Management Interface

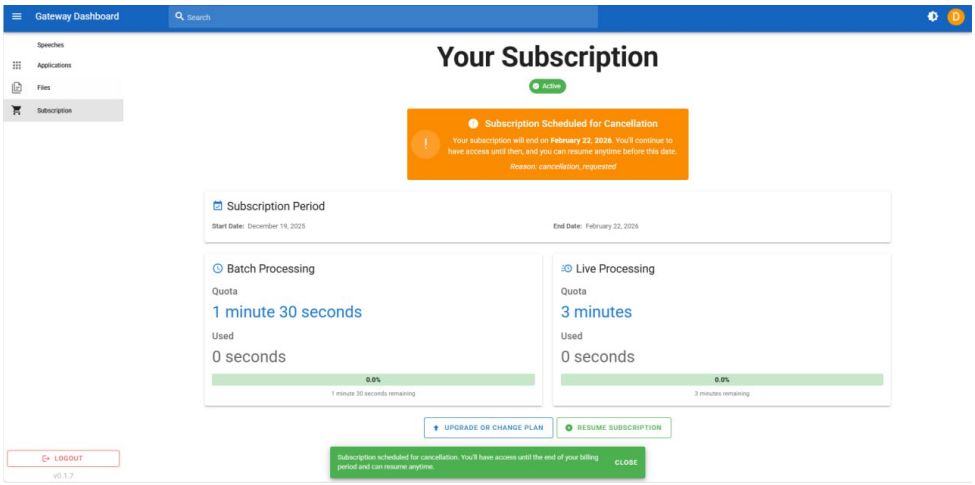


Figure 1.9: Plan Upgrade Interface

Chapter 2

Literature Review

This chapter examines the key technology decisions for the Gateway Dashboard, focusing on SSO provider selection and payment platform comparison.

2.1 SSO Provider Selection

Single Sign-On (SSO) is an authentication method that allows users to log in to multiple applications using one set of credentials from a trusted identity provider (such as Google or Apple), eliminating the need to create separate accounts for each service.

The Gateway Dashboard implements Google and Apple SSO as primary authentication methods.

2.1.1 SSO Provider Comparison

Table 2.1 compares major SSO providers:

Table 2.1: SSO Provider Comparison

Provider	**Market Share**	**Key Advantage**
Google	71% (Android)	2B+ users, mature API
Apple	28% (iOS)	Privacy-focused, email relay
Microsoft	Enterprise strong	Azure AD integration
Facebook	2.9B users	Large user base
GitHub	Developer focus	Dev community

2.1.2 Why Google and Apple

Selection Rationale:

1. **Market Coverage:** 95%+ of internet users have Google or Apple accounts [50, 49]
2. **User Convenience:** Eliminates credential fatigue; most developers already have these accounts
3. **Technical Maturity:** Robust OAuth 2.0 implementations with 99.9%+ uptime [20, 2]
4. **Team Division:** Microsoft SSO handled by another developer, avoiding duplicate work

5. **Security:** Both provide 2FA, fraud detection, and privacy features

2.2 Payment Platform Selection: Stripe vs PayPal

A **payment system** is a software infrastructure that enables businesses to accept and process financial transactions, including one-time payments and recurring subscriptions.

Stripe is a technology company that provides payment processing APIs for internet businesses. It offers a developer-friendly platform for handling online payments, subscription billing, and financial operations through a unified API.

2.2.1 Comparison Overview

Table 2.2 compares Stripe and PayPal:

Table 2.2: Stripe vs PayPal Comparison

Criteria	**Stripe**	**PayPal**
Transaction Fee	2.9% + $0.30	2.9% + $0.30
API Quality	RESTful, excellent docs [51]	Complex multi-generation APIs
Checkout Flow	Embedded (no redirect)	Redirect-based (-10-15% conversion) [6]
Subscriptions	Native billing, prorated upgrades [52]	Limited flexibility
Webhooks	Auto-retry, signature verification [48]	Basic webhook support
Integration Time	40% faster [24]	Slower, complex documentation
Developer Focus	✓Strong	Limited
Metered Billing	✓Supported	Not supported

2.2.2 Why Stripe

Decision: Stripe chosen for:

- **Subscription Requirements:** Native support for tiered plans, quotas, prorated changes, automated renewals
- **Developer Experience:** Superior API design, documentation, official SDK for Node.js
- **Embedded Checkout:** Maintains user context, better conversion rates
- **Webhook Reliability:** Ensures payment-database consistency with idempotency
- **Target Audience:** Developers value functionality over PayPal's consumer brand

2.3 Summary

Table 2.3 summarizes key technology decisions:

Table 2.3: Technology Decision Summary

Component	**Choice**	**Key Reason**
SSO Providers	Google + Apple	95%+ user coverage, Microsoft by other dev
Payment	Stripe	Better API, subscriptions, webhooks vs PayPal

Core Principles: All choices prioritize developer experience, security, performance, and scalability for the API management domain.

Chapter 3

System Architecture

This chapter provides a comprehensive overview of the Gateway Dashboard system architecture, detailing the technology stack, component interactions, and database design decisions that underpin the platform.

3.1 Overview

The Gateway Dashboard system follows a modern three-tier architecture consisting of a **Vue.js** frontend, a **NestJS** backend API, and integration with external services. This separation of concerns ensures scalability, maintainability, and clear boundaries between presentation, business logic, and data persistence layers.

3.1.1 High-Level Architecture

Figure 3.1 illustrates the overall system architecture. The frontend communicates with the backend through RESTful APIs, while the backend interacts with external services such as Stripe for payment processing and OAuth providers for authentication.

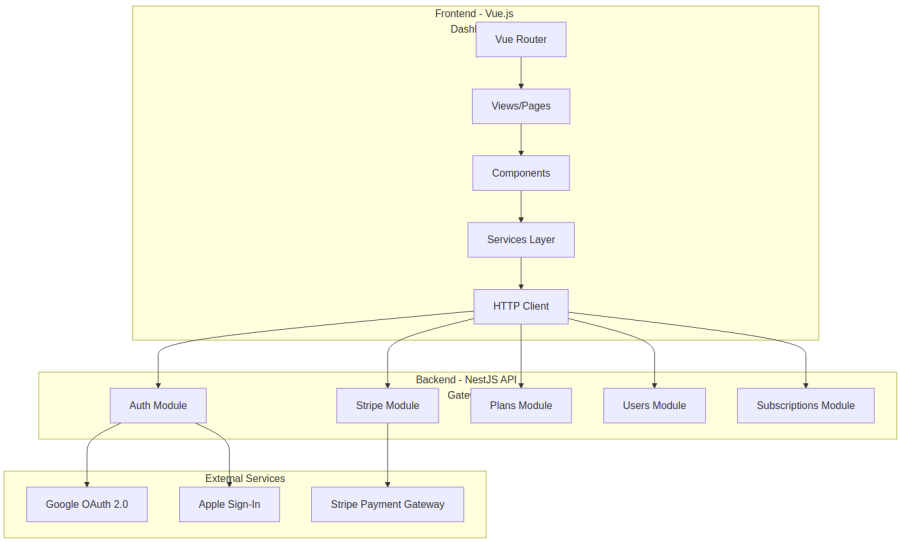


Figure 3.1: High-Level System Architecture

The architecture diagram details the data flow within the system. The **Frontend** layer consists of Vue Router directing requests to Views/Pages, which utilize Components and

a Services Layer that communicates via an HTTP Client. The **Backend** layer comprises modular NestJS components: Auth, Stripe, Plans, Users, and Subscriptions modules. The **External Services** layer includes Google OAuth 2.0, Apple Sign-In, and Stripe Payment Gateway for third-party integrations.

3.1.2 Component Overview

The system is composed of the following major components:

- **Frontend Dashboard**: A Single Page Application (SPA) built with Vue.js 3 and Vuetify, providing user interfaces for authentication, subscription management, and administrative functions.
- **Backend API Gateway**: A NestJS-based REST API that handles authentication, authorization, business logic, and orchestration of external services.
- **Database Layer**: MongoDB serves as the primary data store for users, subscriptions, plans, and tokens, with Redis providing caching for frequently accessed data.
- **External Services**: Integration with Google OAuth 2.0, Apple Sign-In, and Stripe for third-party authentication and payment processing.

3.1.3 Communication Flow

The typical request flow proceeds as follows:

1. The user interacts with the Vue.js frontend through the browser.
2. The frontend dispatches HTTP requests to the backend API via Axios interceptors.
3. The backend validates the JWT token, checks authorization, and processes the request.
4. Business logic is executed, interacting with MongoDB, Redis, or external APIs as needed.
5. The response is returned to the frontend, which updates the user interface accordingly.

3.2 Technology Stack

The Gateway Dashboard leverages modern, industry-standard technologies to ensure robustness, scalability, and developer productivity.

3.2.1 Frontend Technologies

Technology	Version	Purpose
Vue.js	3.x	Progressive JavaScript framework for building user interfaces
Vuetify	3.x	Material Design component framework for Vue.js
Vue Router	4.x	Official router for single-page application navigation
Axios	Latest	Promise-based HTTP client for API communication

Table 3.1: Frontend Technology Stack

Vue.js 3 was selected for its Composition API, improved performance, and excellent TypeScript support. **Vuetify 3** provides a comprehensive set of Material Design components, enabling rapid UI development with a professional appearance.

3.2.2 Backend Technologies

Technology	Version	Purpose
NestJS	10.x	Progressive Node.js framework for server-side applications
TypeScript	5.x	Typed superset of JavaScript for enhanced code quality
Passport	0.7.x	Authentication middleware for Node.js
JWT	Latest	JSON Web Token for stateless authentication

Table 3.2: Backend Technology Stack

NestJS provides a modular architecture with built-in dependency injection support, ideal for enterprise-grade applications. TypeScript ensures type safety and improved developer experience. **Passport** offers a flexible authentication framework with extensive strategy support for OAuth providers.

3.2.3 Data Storage and Caching

Technology	Version	Purpose
MongoDB	6.x	NoSQL document database for flexible schema design
Redis	7.x	In-memory data store for caching

Table 3.3: Storage Technologies

MongoDB was chosen for its schema flexibility, horizontal scalability, and native JSON support. **Redis** is primarily used for caching frequently accessed data (plans, prices) to reduce database load and API latency.

3.2.3.1 Caching Architecture

Figure 3.2 illustrates the caching architecture for the Plan module. The service layer first checks Redis for cached data, falling back to MongoDB on cache misses.

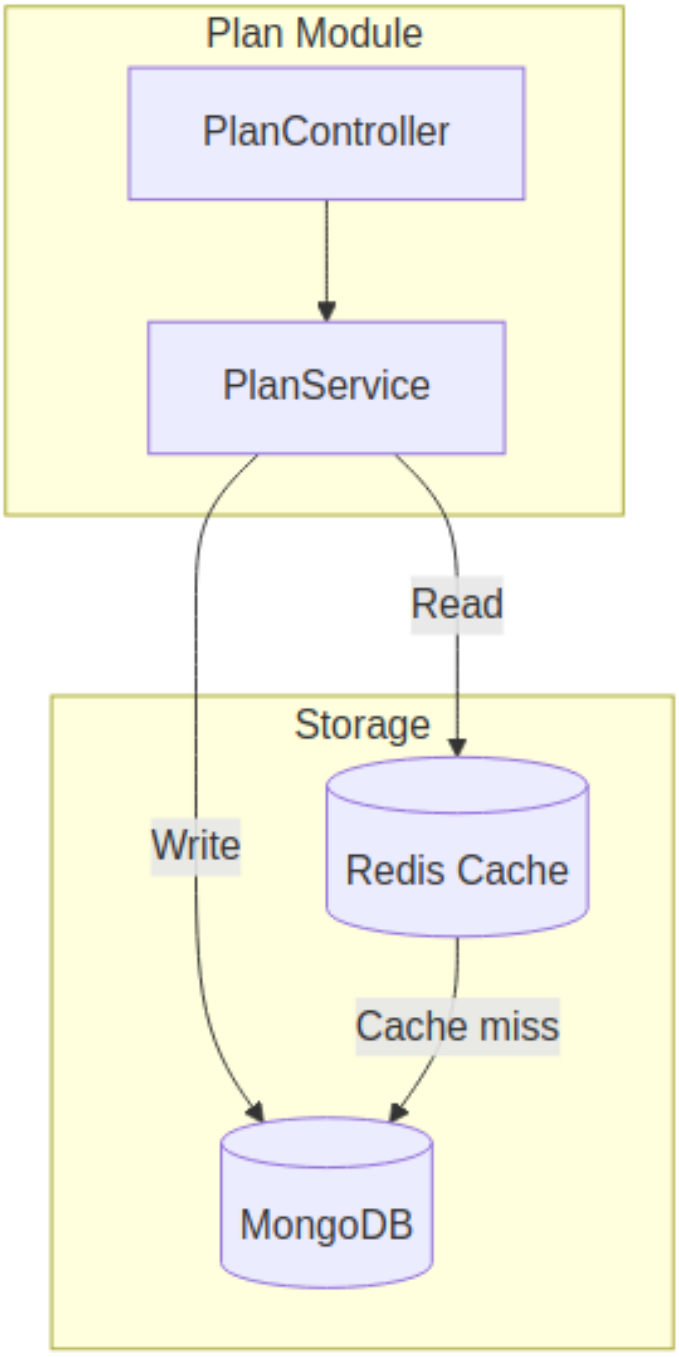


Figure 3.2: Plan Module Caching Architecture

As depicted in Figure 3.2, the Plan Module implements a read-through caching strategy. When a request for plans is received, the `PlanService` first queries the Redis cache. If the data is found (cache hit), it is returned immediately. If not (cache miss), the service retrieves the data from MongoDB, stores it in Redis for future requests, and then returns it to the controller.

Figure 3.3 shows the detailed cache flow for retrieving the latest plans. This strategy includes cache penetration protection by caching NULL values for a short duration when no plans exist.

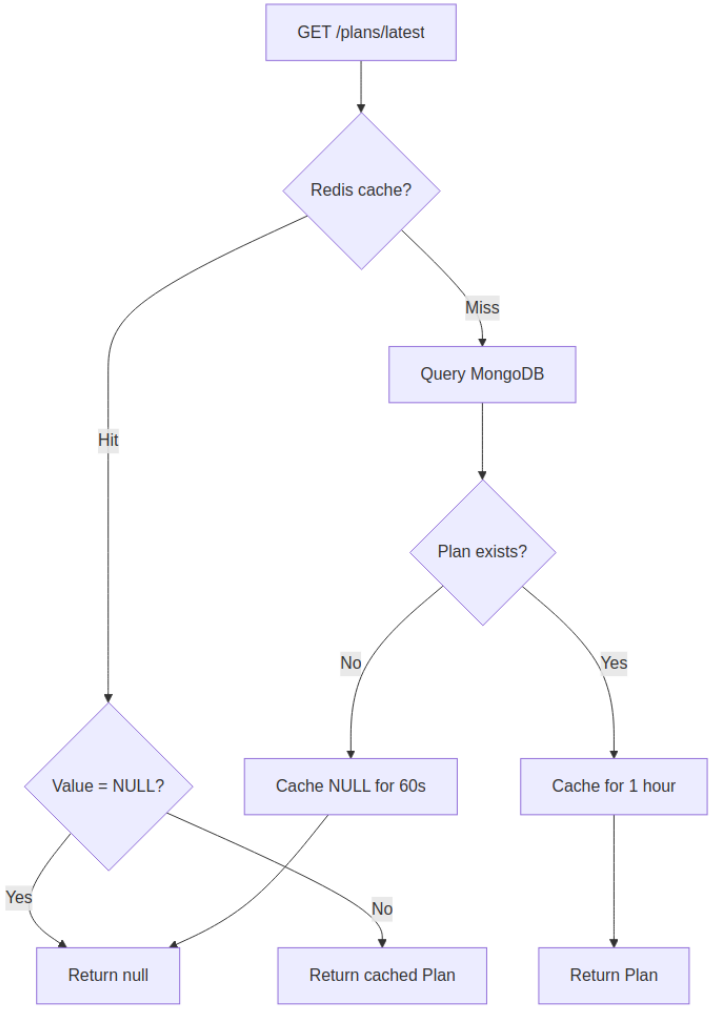


Figure 3.3: Cache Flow for Latest Plans Retrieval

Figure 3.3 provides a granular view of the "get latest plans" logic. It highlights the cache penetration protection mechanism: if a database query returns no plans, a special `NULL` value is cached for a short duration (e.g., 60 seconds). This prevents repeated queries for non-existent data from overwhelming the database essentially acting as a shield during high-traffic periods.

3.2.4 External Services

- **Stripe**: Payment processing platform for subscription billing, checkout sessions, and webhook handling.
- **Google OAuth 2.0**: Identity provider for social login functionality.
- **Apple Sign-In**: Authentication service for iOS and web users.

These third-party services were selected for their reliability, comprehensive documentation, and industry adoption, reducing development time while ensuring security and compliance.

3.3 Database Design

The database schema is designed to support user management, authentication, subscription tracking, and service plan versioning. MongoDB's document-oriented approach allows for flexible schema evolution.

3.3.1 Core Collections

The system utilizes the following primary collections:

3.3.1.1 Users Collection

The `users` collection stores user account information and authentication credentials.

Field	Type	Description
_id	ObjectId	Primary key
email	String	User email (unique)
password	String	Hashed password
name	String	Full name
role	String	User role (user/admin)
type	String	Account type (trial/paid)
isVerified	Boolean	Email verification status

Table 3.4: Users Collection Schema

3.3.1.2 Subscriptions Collection

The `subscriptions` collection tracks user subscriptions and quota usage.

Field	Type	Description
_id	ObjectId	Primary key
user	ObjectId	Reference to users collection
stripeSubscriptionId	String	Stripe subscription ID
stripeCustomerId	String	Stripe customer ID
planName	String	Current plan name
priceId	String	Stripe price ID
planId	String	Internal plan identifier
scheduledPriceId	String	Next scheduled price ID
scheduledPlanId	String	Next scheduled plan ID
scheduledPlanName	String	Next scheduled plan name
stripeScheduleId	String	Stripe schedule ID
status	String	Subscription status (active, canceled, etc.)
quota	Object	Service quotas (batch, live duration)
usage	Object	Current usage tracking
startDate	Date	Subscription start date
endDate	Date	Subscription end date

Table 3.5: Subscriptions Collection Schema

The `quota` and `usage` fields store nested objects tracking batch and live processing durations in seconds. A value of -1 indicates unlimited access.

3.3.1.3 Plans Collection

The `plans` collection maintains versioned service plans.

Field	Type	Description
_id	ObjectId	Primary key
id	String	Unique plan identifier
plans	String	JSON string of plan details
version	Number	Plan version (auto-increment)
createdAt	Date	Creation timestamp

Table 3.6: Plans Collection Schema

Storing plans as JSON strings enables dynamic schema evolution without database migrations.

3.3.2 Authentication Collections

To ensure secure session management, two additional collections were introduced:

3.3.2.1 UserTokens Collection

The `usertokens` collection tracks active JWT tokens for each user.

Field	Type	Description
_id	ObjectId	Primary key
userId	String	User identifier
token	String	JWT token string
expiresAt	Date	Token expiration (TTL indexed)
userAgent	String	Client user agent
ipAddress	String	Client IP address

Table 3.7: UserTokens Collection Schema

MongoDB's TTL (Time-To-Live) index on `expiresAt` automatically removes expired tokens.

3.3.2.2 TokenBlacklist Collection

The `tokenblacklists` collection stores revoked tokens to prevent reuse.

Field	Type	Description
_id	ObjectId	Primary key
token	String	Revoked JWT token
userId	String	User identifier
expiresAt	Date	Original token expiration
reason	String	Revocation reason

Table 3.8: TokenBlacklist Collection Schema

3.3.3 Entity Relationships

Figure 3.4 illustrates the relationships between core collections:

- Each `user` has zero or one `subscription`.
- Each `subscription` references a `plan` by Stripe priceId.
- Each `user` may have multiple active `tokens`.
- Revoked tokens are stored in `tokenblacklist`.

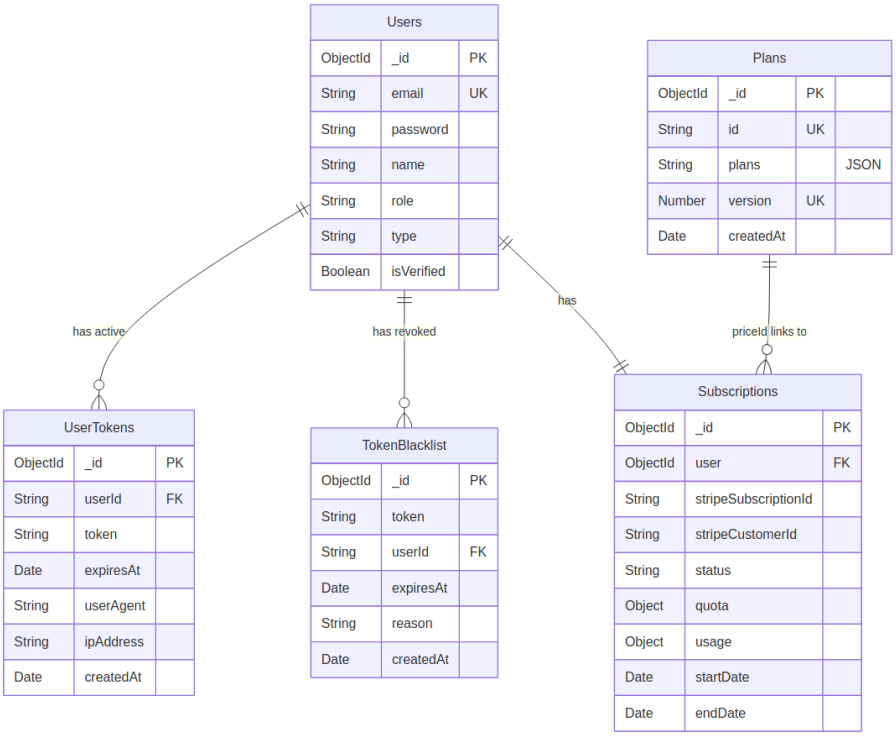


Figure 3.4: Entity Relationship Diagram

The Entity Relationship Diagram in Figure 3.4 showcases the database schema. The `Users` collection is central, linking to `Subscriptions` (1:1 relationship) and multiple `UserTokens`. The `Subscriptions` collection contains embedded objects for `quota` and `usage`, allowing efficient tracking of service limits. The `Plans` collection stands independently but is logically referenced by subscriptions via price IDs. The `TokenBlacklist` stores revoked tokens, ensuring security compliance.

Chapter 4

Detailed Implementation

This chapter provides a comprehensive examination of the key implementation details of the Gateway Dashboard system. It is organized into two main parts: Backend Implementation covering authentication and payment systems, and Frontend Development covering the Vue.js dashboard application.

4.1 Backend Implementation

This section covers the server-side implementation including Single Sign-On (SSO) authentication and payment/subscription management.

4.1.1 Authentication and SSO Implementation

This section provides a comprehensive examination of the authentication system implemented in the Gateway Dashboard. The system is organized into three distinct modules matching the backend architecture:

- **Google SSO Module**: OAuth 2.0 flow with Passport.js strategy
- **Apple SSO Module**: POST callback with JWT token verification
- **Auth Module**: Core authentication with traditional email/password, token management, and shared services

Figure 4.1 illustrates the overall architecture showing how all authentication modules integrate.

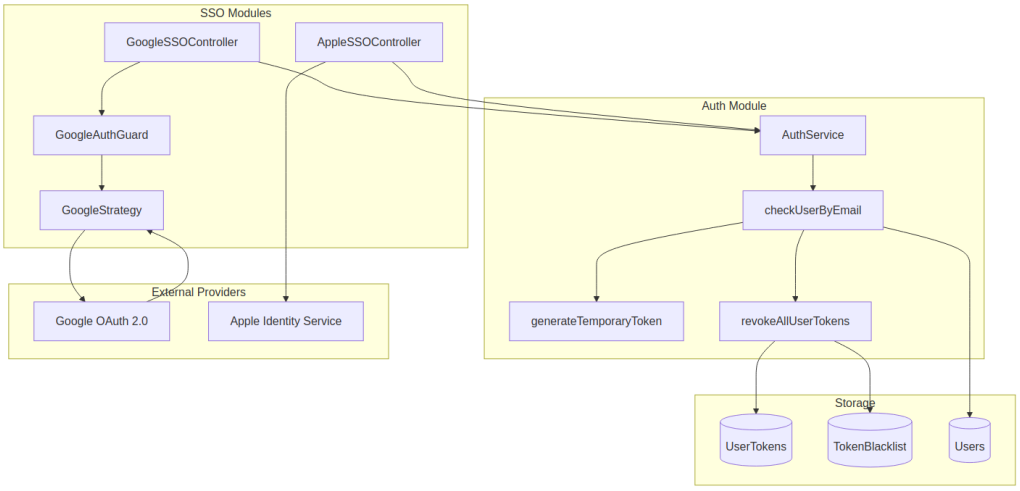


Figure 4.1: Authentication System Architecture Overview

4.1.1.1 Google SSO Module

The Google SSO Module (`google-sso/`) implements OAuth 2.0 authentication using Google's identity platform with Passport.js.

4.1.1.1.1 Module Structure Figure 4.2 shows the Google SSO module organization.

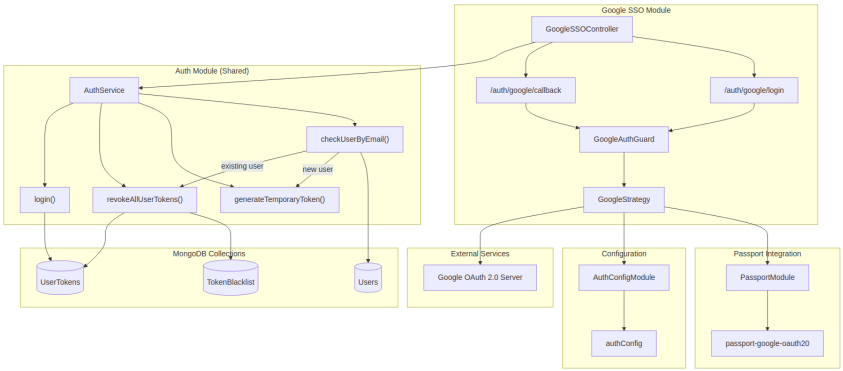


Figure 4.2: Google SSO Module Structure

The module consists of:

- `GoogleSSOController`: Handles `/auth/google/*` endpoints
- `GoogleAuthGuard`: Custom Passport guard with state parameter encoding
- `GoogleStrategy`: Passport strategy configured with OAuth credentials
- `GoogleSSOModule`: NestJS module importing `PassportModule` and `AuthModule`

4.1.1.1.2 API Endpoints Table 4.1 lists the Google SSO endpoints.

Table 4.1: Google SSO API Endpoints

Method	Endpoint	Description
GET	/auth/google/login	Initiate Google OAuth login (redirects to Google)
GET	/auth/google/callback	Handle OAuth callback from Google

4.1.1.1.3 Login Endpoint Initiates the Google OAuth flow by redirecting to Google's consent screen.

Request:

```
GET /auth/google/login?redirect=https://app.example.com&appId=123&prompt=
    select_account
```

Listing 4.1: Google Login Request

Query Parameters:

- `redirect` (optional): Frontend URL to redirect after authentication
- `appId` (optional): Application ID for multi-tenant scenarios
- `appSecret` (optional): Application secret
- `prompt` (optional): Google OAuth prompt parameter (`select_account`, `consent`, `none`)

4.1.1.1.4 Callback Endpoint Handles the OAuth callback from Google after user authentication.

For New Users: Redirects to `/complete-signup` with temporary token stored in `localStorage`:

```
<html>
  <head>
    <script>
      localStorage.setItem('temporaryToken', '<temp-jwt>');
      window.location.href = 'https://app.example.com/complete-signup';
    </script>
  </head>
</html>
```

Listing 4.2: New User Redirect Response

For Existing Users: Redirects to `/sign-in` with access token stored in `localStorage`:

```
<html>
  <head>
    <script>
      localStorage.setItem('accessToken', '<access-token>');
      localStorage.setItem('jwt', '<access-token>');
      localStorage.setItem('subscriptionEnd', '1735689600000');
      localStorage.setItem('isVerified', 'true');
      window.location.href = 'https://app.example.com/sign-in';
    </script>
  </head>
```

```
</html>
```

Listing 4.3: Existing User Redirect Response

4.1.1.1.5 OAuth Flow Figure 4.3 shows the complete OAuth flow including state parameter handling.

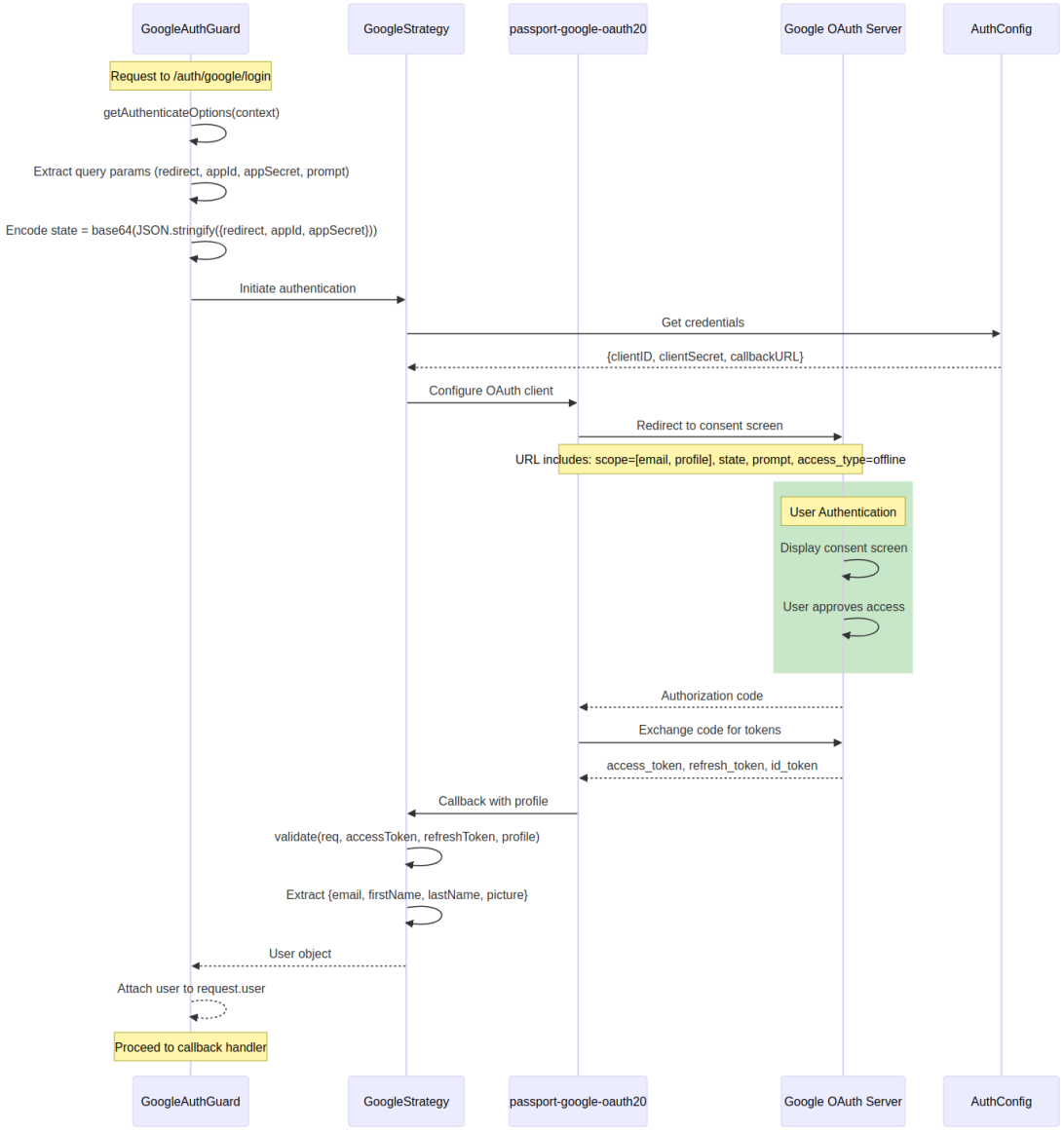


Figure 4.3: Google OAuth 2.0 Authentication Flow

4.1.1.1.6 Google Strategy Implementation The strategy is configured with OAuth credentials and requests email and profile scopes:

```
@Injectable()
export class GoogleStrategy extends PassportStrategy(Strategy, 'google') {
  constructor(
    @Inject(authConfig.KEY)
    private auth: ConfigType<typeof authConfig>,
  ) {
    super({
```

```
      clientID: auth.googleClientId,
      clientSecret: auth.googleClientSecret,
      callbackURL: auth.googleCallbackURL,
      scope: ['email', 'profile'],
      passReqToCallback: true,
    })
  }

  async validate(
    req: any,
    accessToken: string,
    _refreshToken: string,
    profile: any,
    done: VerifyCallback,
  ): Promise<any> {
    const { name, emails, photos } = profile
    const user = {
      email: emails[0].value,
      firstName: name.givenName,
      lastName: name.familyName,
      picture: photos[0].value,
      accessToken,
    }
    done(null, user)
  }
}
```

Listing 4.4: Google Strategy Configuration

4.1.1.1.7 State Parameter for Context Preservation Figure 4.4 illustrates how state parameters are preserved through the OAuth redirect to maintain user session context.

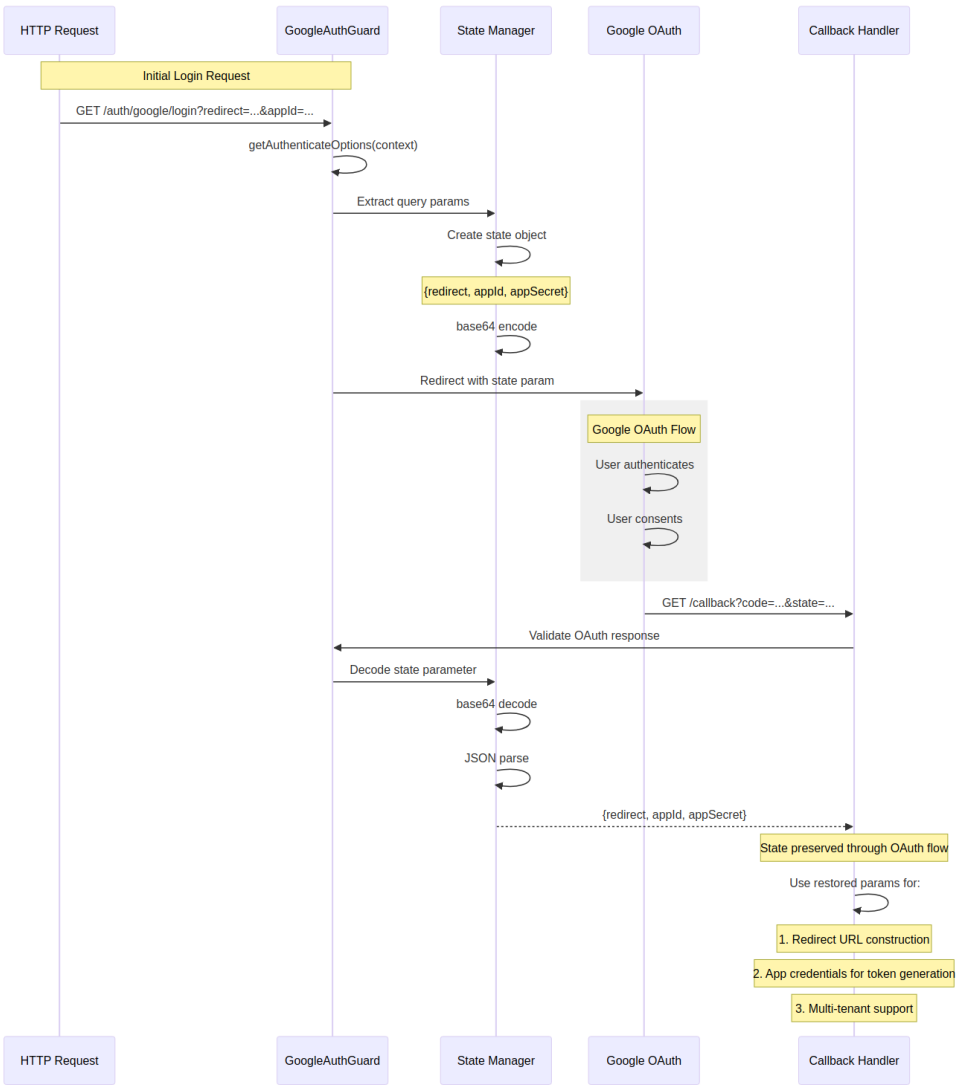


Figure 4.4: State Parameter Encoding Flow

The `GoogleAuthGuard` encodes query parameters into a base64 state parameter:

```
@Injectable()
export class GoogleAuthGuard extends AuthGuard('google') {
  getAuthenticateOptions(context: ExecutionContext) {
    const request = context.switchToHttp().getRequest()
    const { redirect, appId, appSecret, prompt } = request.query

    // Encode query parameters into state to preserve them through OAuth flow
    const state = JSON.stringify({
      redirect: redirect || null,
      appId: appId || null,
      appSecret: appSecret || null,
    })

    return {
      state: Buffer.from(state).toString('base64'),
      prompt: prompt || 'select_account', // Force account selection
      access_type: 'offline', // For refresh tokens
    }
```

```
  }
}
```

Listing 4.5: Google Auth Guard with State Encoding

4.1.1.2 Apple SSO Module

The Apple SSO Module (`apple-sso/`) implements Sign in with Apple using POST callback and JWT token verification.

4.1.1.2.1 Module Structure Figure 4.5 shows the Apple SSO module organization.

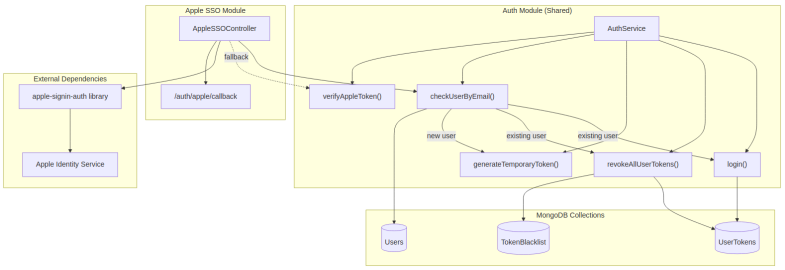


Figure 4.5: Apple SSO Module Structure

The module consists of:

- `AppleSSOController`: Handles `/auth/apple/*` endpoints
- `AppleSSOModule`: NestJS module importing `AuthModule`
- Uses `apple-signin-auth` library for token verification

4.1.1.2.2 API Endpoints Table 4.2 lists the Apple SSO endpoints.

Table 4.2: Apple SSO API Endpoints

Method	Endpoint	Description
POST	/auth/apple/callback	Handle Apple Sign-In callback with identity token

4.1.1.2.3 Callback Endpoint Handles the POST callback from Apple Sign-In with the identity token.

Request:

```
POST /auth/apple/callback
Content-Type: application/json

{
  "id_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...",
  "code": "authorization-code-from-apple"
}
```

Listing 4.6: Apple Callback Request

Response (New User): Returns HTML that stores temporary token and redirects to complete signup:

```
<html>
  <head>
    <script>
      localStorage.setItem('temporaryToken', '<temp-jwt>');
      window.location.href = 'https://app.example.com/complete-signup';
    </script>
  </head>
</html>
```

Listing 4.7: Apple New User Response

Response (Existing User): Returns HTML that stores access token and redirects to sign-in:

```
<html>
  <head>
    <script>
      localStorage.setItem('accessToken', '<access-token>');
      localStorage.setItem('jwt', '<access-token>');
      localStorage.setItem('subscriptionEnd', '1735689600000');
      localStorage.setItem('isVerified', 'true');
      window.location.href = 'https://app.example.com/sign-in';
    </script>
  </head>
</html>
```

Listing 4.8: Apple Existing User Response

Response (Error - JSON):

```
{
  "statusCode": 401,
  "success": false,
  "message": "Apple authentication failed: No id_token received"
}
```

Listing 4.9: Apple Error Response

4.1.1.2.4 Authentication Flow

Figure 4.6 shows the Apple Sign-In flow which uses POST callback instead of redirect-based OAuth.

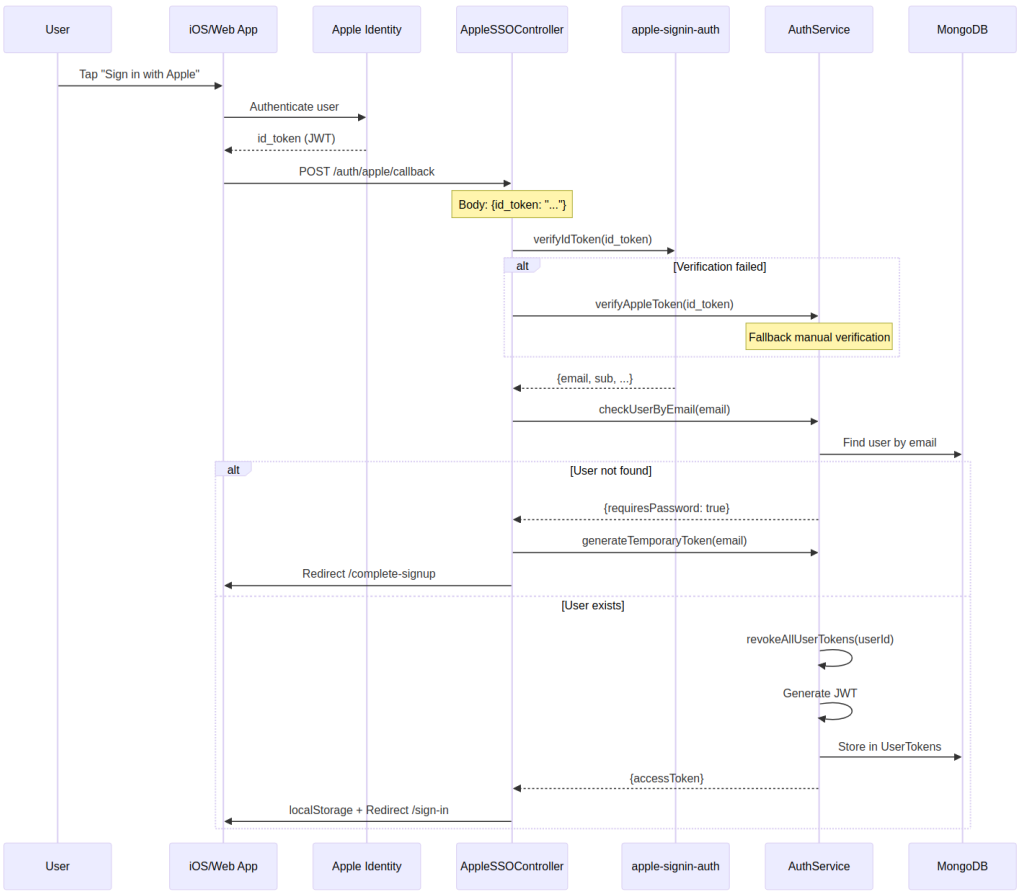


Figure 4.6: Apple Sign-In Authentication Flow

4.1.1.2.5 Token Verification Figure 4.7 shows the token verification process with fallback mechanism.

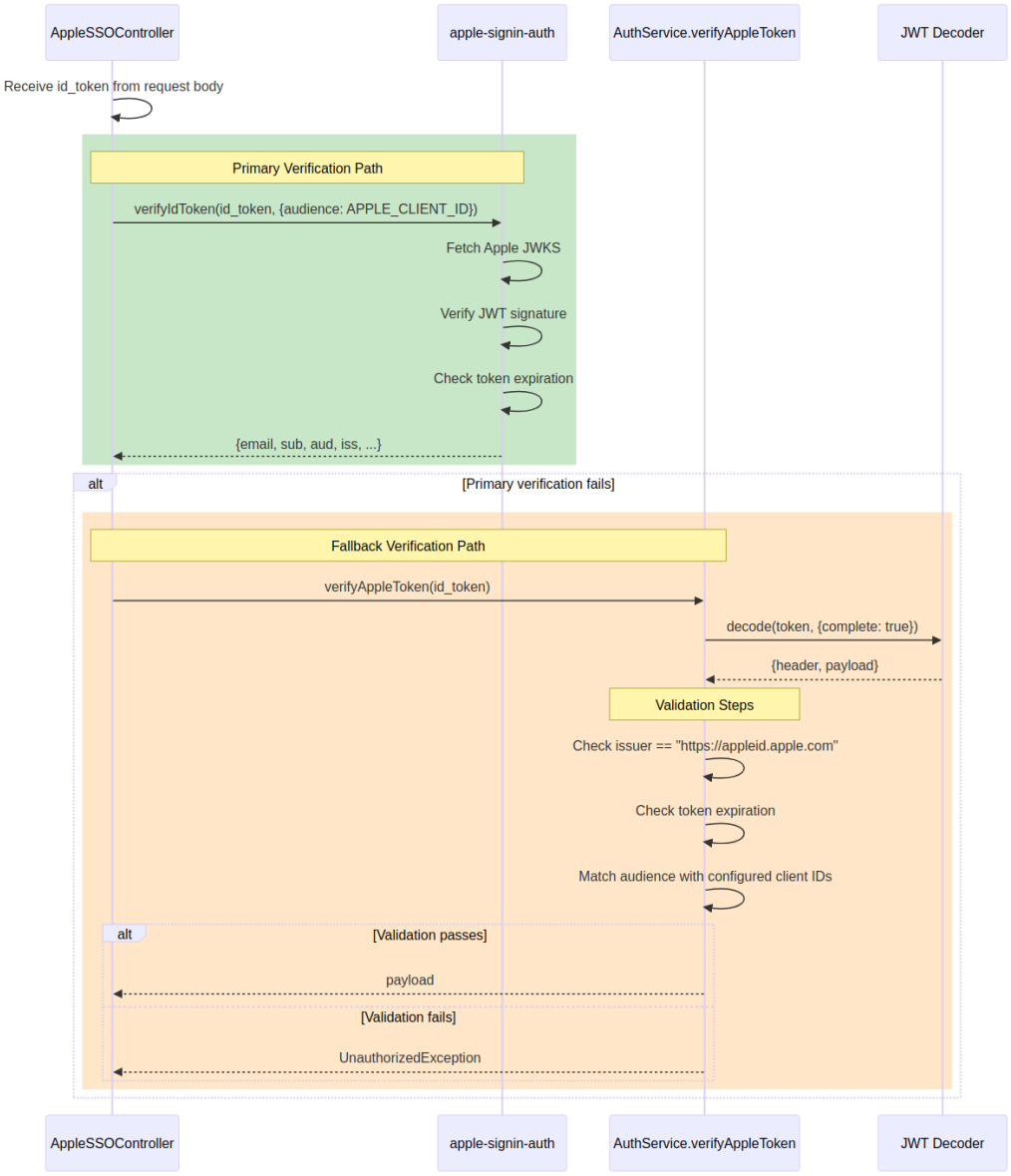


Figure 4.7: Apple Token Verification Flow

Apple tokens are verified using the `apple-signin-auth` library with fallback to manual verification:

```
@ApiTags('Apple SSO')
@Controller('auth/apple')
export class AppleSSOController {
  constructor(private readonly authService: AuthService) {}

  @Post('callback')
  async appleAuthCallback(@Body() body: AppleAuthDto, @Res() res: Response) {
    if (!body.code && !body.id_token) {
      return res.status(HttpStatus.UNAUTHORIZED).json({
        statusCode: HttpStatus.UNAUTHORIZED,
        success: false,
        message: 'Apple authentication failed: No id_token received',
      })
    }
```

```
15
16      let appleResponse
17      try {
18        // Primary verification using apple-signin-auth library
19        appleResponse = await appleSignin.verifyIdToken(body.id_token, {
20          audience: process.env.APPLE_CLIENT_ID,
21          ignoreExpiration: false,
22        })
23      } catch (verifyError) {
24        // Fallback to manual verification
25        appleResponse = await this.authService.verifyAppleToken(body.id_token)
26      }
27
28      const result = await this.authService.checkUserByEmail(appleResponse.email)
29      // Handle new/existing user and redirect accordingly
30    }
31  }
```

Listing 4.10: Apple SSO Controller Implementation

4.1.1.2.6 Manual Token Verification

The fallback verification in `AuthService` validates Apple tokens manually:

```
1  async verifyAppleToken(token: string): Promise<any> {
2    // Get all possible Apple Client IDs
3    const appleClientIds = [
4      this.configService.get('auth.appleClientId'),
5      this.configService.get('auth.appleClientIdAndroid'),
6    ].filter(Boolean)
7
8    // Decode JWT token to get header and payload
9    const decoded = jwt.decode(token, { complete: true })
10   if (!decoded) {
11     throw new UnauthorizedException('Invalid Apple token format')
12   }
13
14   const payload = decoded.payload as any
15
16   // Verify issuer
17   if (payload.iss !== 'https://appleid.apple.com') {
18     throw new UnauthorizedException('Invalid Apple token issuer')
19   }
20
21   // Check expiration
22   const now = Math.floor(Date.now() / 1000)
23   if (payload.exp && payload.exp < now) {
24     throw new UnauthorizedException('Apple token has expired')
25   }
26
27   // Verify audience matches configured client IDs
28   for (const clientId of appleClientIds) {
29     if (payload.aud === clientId) {
30       return payload
```

```
31      }
32    }
33
34    throw new UnauthorizedException('Apple token audience mismatch')
35  }
```

Listing 4.11: Apple Token Manual Verification

4.1.1.3 Auth Module

The Auth Module (`auth/`) serves as the core authentication component, providing traditional email/password authentication, token management, and shared services used by SSO modules.

4.1.1.3.1 Module Architecture Figure 4.8 shows the Auth module's internal structure.

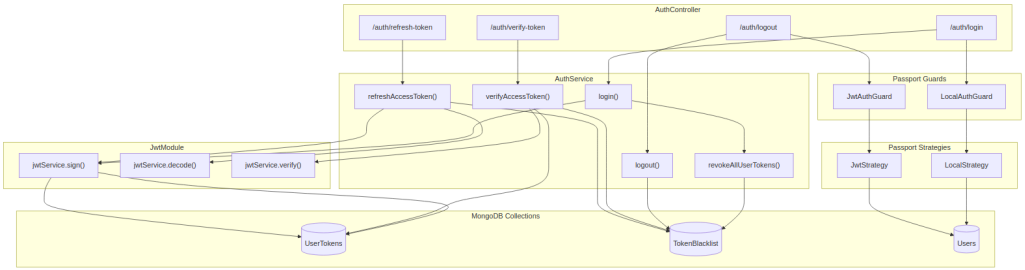


Figure 4.8: Auth Module Architecture

The module consists of:

- `AuthController`: Handles all `/auth/*` HTTP endpoints
- `AuthService`: Business logic for authentication, token management, and user verification
- `LocalAuthGuard` & `LocalStrategy`: Passport.js strategy for email/password authentication
- `JwtStrategy`: JWT token validation strategy
- `TokenBlacklist` & `UserTokens`: MongoDB schemas for session management

4.1.1.3.2 API Endpoints Figure 4.9 provides an overview of all Auth module endpoints.

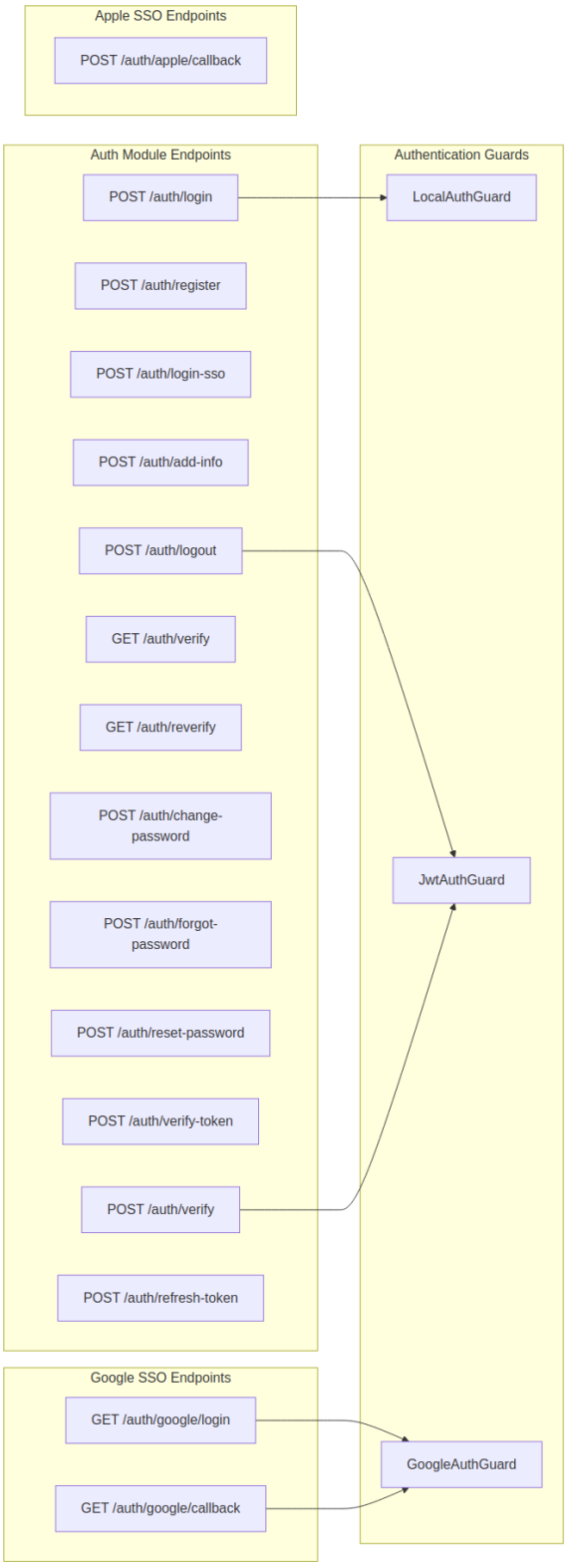


Figure 4.9: Auth Module API Endpoints Overview

Table 4.3 provides a complete list of Auth module endpoints with their descriptions.

Table 4.3: Auth Module API Endpoints

Method	Endpoint	Auth	Description
POST	/auth/login	None	Traditional email/password login
POST	/auth/register	None	Register new user account
POST	/auth/login-sso	None	SSO login with provider token
POST	/auth/add-info	None	Complete OAuth signup with password
POST	/auth/verify	JWT	Send verification email
GET	/auth/verify	None	Verify email with token
GET	/auth/reverify	None	Resend verification email
POST	/auth/change-password	None	Change user password
POST	/auth/forgot-password	None	Request password reset code
POST	/auth/reset-password	None	Reset password with code
POST	/auth/verify-token	None	Validate access token
POST	/auth/logout	JWT	Logout and blacklist token
POST	/auth/refresh-token	None	Refresh expired access token

4.1.1.3.3 Login Endpoint The primary login endpoint authenticates users with email and password credentials.

Request:

```
POST /auth/login
Content-Type: application/json

{
  "email": "user@example.com",
  "password": "securePassword123",
  "appId": "optional-app-id",
  "appSecret": "optional-app-secret"
}
```

Listing 4.12: Login Request

Response (Success):

```
{
  "statusCode": 200,
  "success": true,
  "message": "Login successful",
  "accessToken": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...",
  "subscriptionEnd": 1735689600000,
  "isVerified": true
}
```

Listing 4.13: Login Success Response

Figure 4.10 illustrates the complete login flow including single-session enforcement.

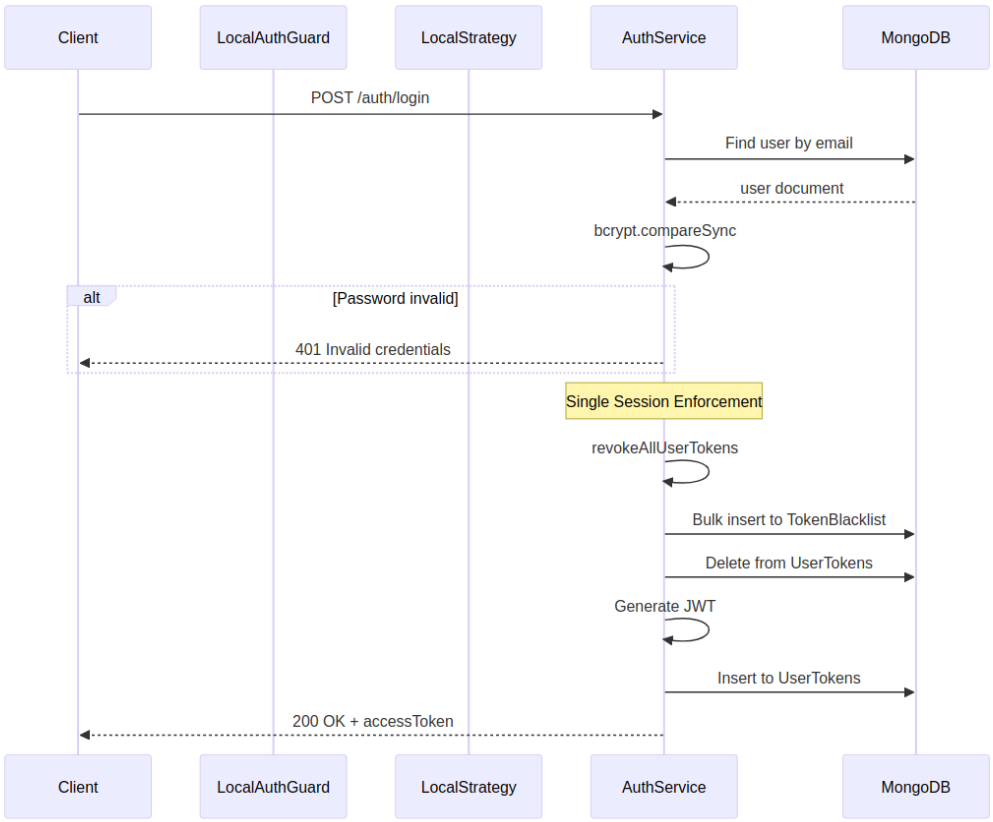


Figure 4.10: Login Authentication Flow

4.1.1.3.4 Registration Endpoint Request:

```
POST /auth/register
Content-Type: application/json

{
  "name": "John Doe",
  "email": "john@example.com",
  "password": "securePassword123",
  "appId": "optional-app-id",
  "appSecret": "optional-app-secret"
}
```

Listing 4.14: Registration Request

Response (Success):

```
{
  "statusCode": 201,
  "message": "User registered successfully",
  "success": true,
  "user": {
    "_id": "64abc123def456789",
    "name": "John Doe",
    "email": "john@example.com",
    "isVerified": false,
    "role": "user"
  }
```

```
}
```

Listing 4.15: Registration Success Response

Figure 4.11 shows the registration flow.

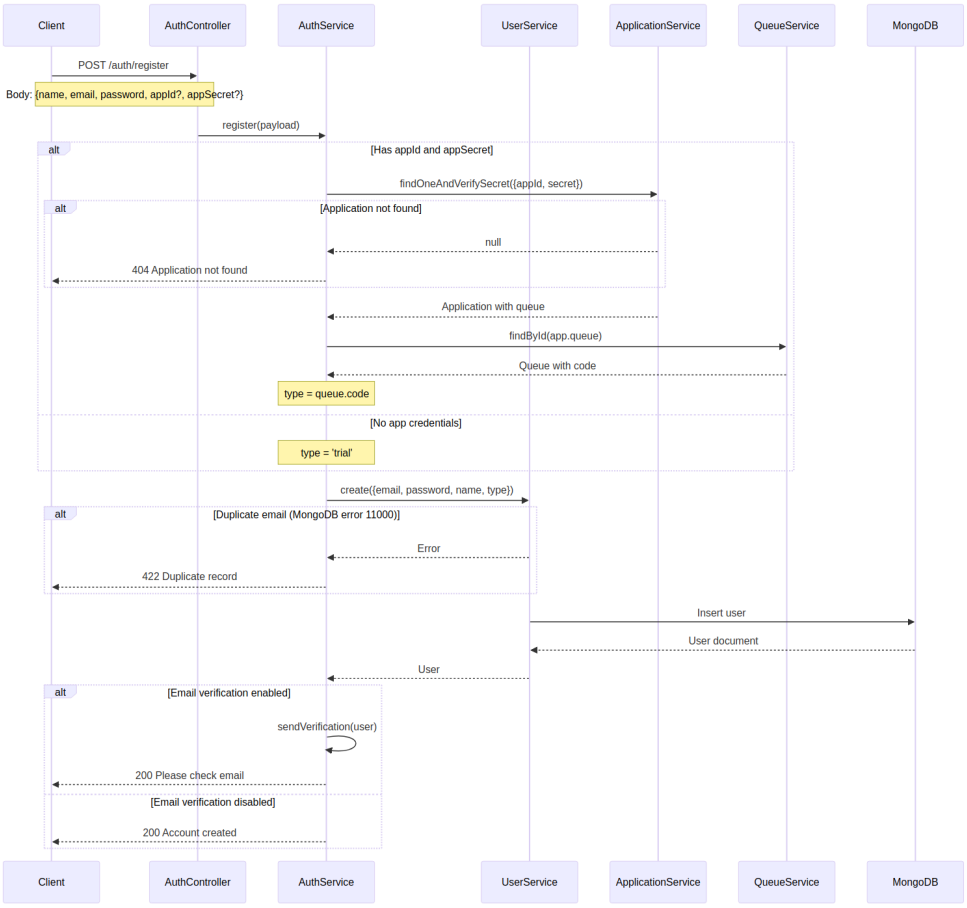


Figure 4.11: User Registration Flow

4.1.1.3.5 SSO Login Endpoint This endpoint handles SSO authentication using tokens from OAuth providers.

Request:

```
POST /auth/login-sso
Content-Type: application/json

{
  "token": "google-or-apple-id-token",
  "oauthClient": "google|apple|microsoft",
  "appId": "optional-app-id",
  "appSecret": "optional-app-secret"
}
```

Listing 4.16: SSO Login Request

Response (Success):

```
{
  "statusCode": 200,
```

```
  "message": "Login successful",
  "data": {
    "accessToken": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...",
    "refreshToken": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...",
    "user": {
      "id": "user_123",
      "email": "user@example.com",
      "name": "John Doe",
      "oauthProvider": "google"
    }
  }
}
```

Listing 4.17: SSO Login Success Response

4.1.1.3.6 Password Management Endpoints Change Password Request:

```
POST /auth/change-password
Content-Type: application/json

{
  "email": "user@example.com",
  "currentPassword": "oldPassword123",
  "newPassword": "newSecurePassword456",
  "confirmNewPassword": "newSecurePassword456"
}
```

Listing 4.18: Change Password Request

Response (Success):

```
{
  "statusCode": 200,
  "message": "Password changed successfully"
}
```

Listing 4.19: Change Password Success Response

Forgot Password Request:

```
POST /auth/forgot-password
Content-Type: application/json

{
  "email": "user@example.com"
}
```

Listing 4.20: Forgot Password Request

Response (Success):

```
{
  "statusCode": 200,
  "message": "Password reset code sent to email"
}
```

Listing 4.21: Forgot Password Success Response

Reset Password Request:

```
POST /auth/reset-password
Content-Type: application/json

{
  "email": "user@example.com",
  "code": "123456",
  "newPassword": "newSecurePassword456",
  "confirmNewPassword": "newSecurePassword456"
}
```

Listing 4.22: Reset Password Request

Response (Success):

```
{
  "statusCode": 200,
  "message": "Password reset successfully"
}
```

Listing 4.23: Reset Password Success Response

Figure 4.12 shows the password reset flow.

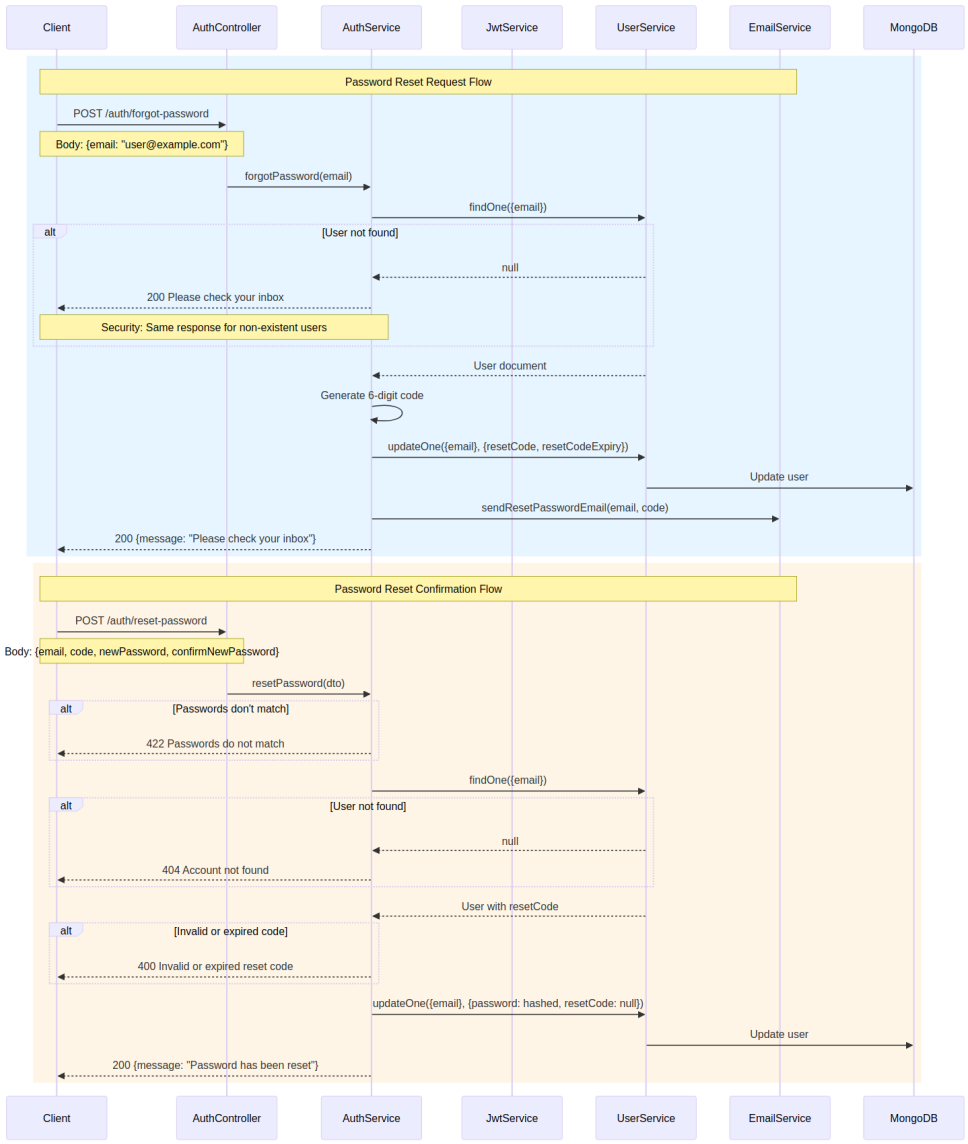


Figure 4.12: Password Reset Flow

4.1.1.3.7 Token Management Figure 4.13 illustrates the complete token lifecycle from generation to revocation.

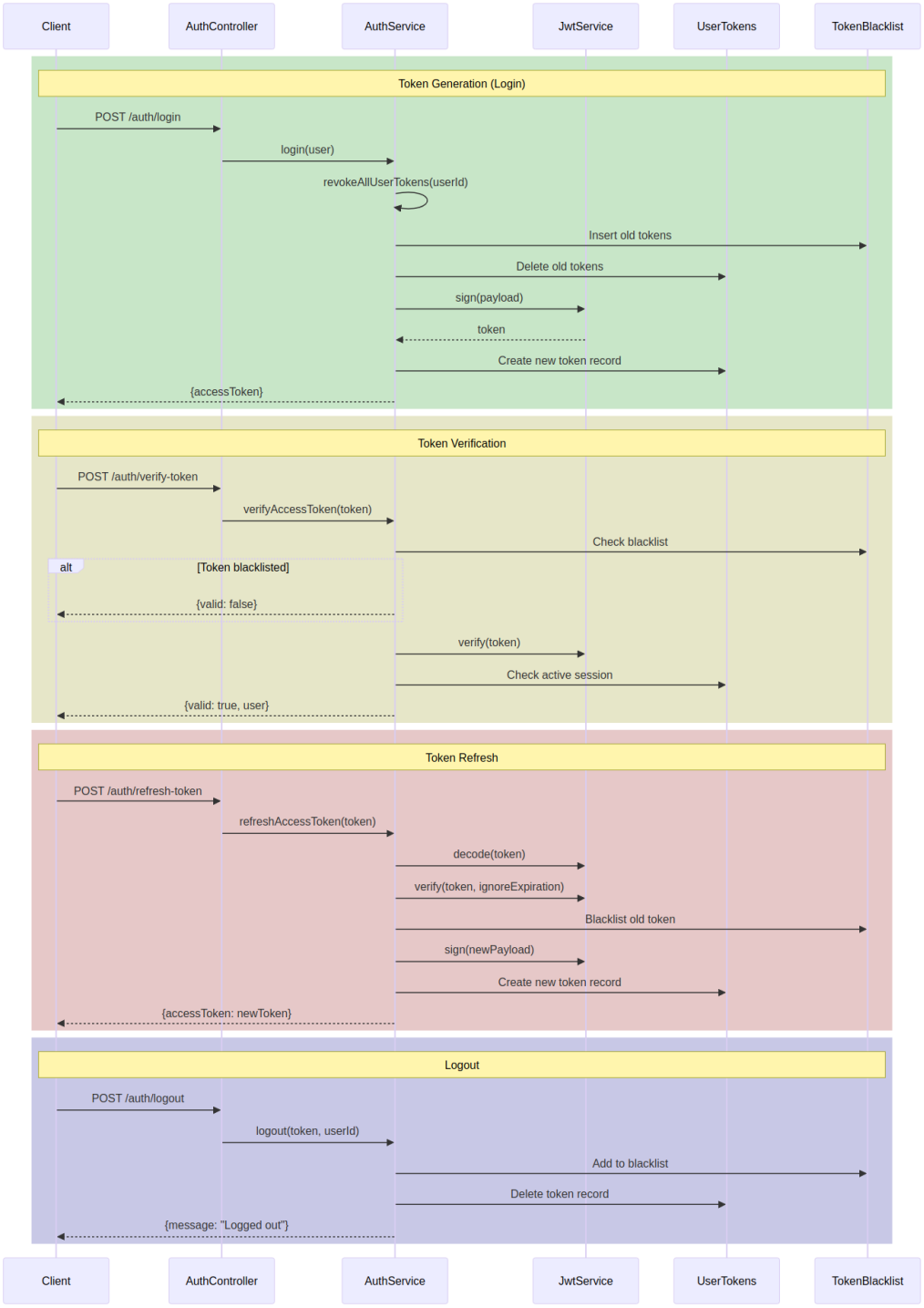


Figure 4.13: JWT Token Lifecycle Management

Verify Token Request:

```
POST /auth/verify-token
Content-Type: application/json

{
  "token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9..."
}
```

Listing 4.24: Verify Token Request

Verify Token Response:

```
{
  "statusCode": 200,
  "valid": true,
  "expired": false,
  "user": {
    "id": "64abc123def456789",
    "email": "user@example.com",
    "role": "user",
    "name": "John Doe",
    "type": "user",
    "isVerified": true
  },
  "expiresAt": 1735689600,
  "message": "Token is valid"
}
```

Listing 4.25: Verify Token Response

Figure 4.14 shows the token verification flow.

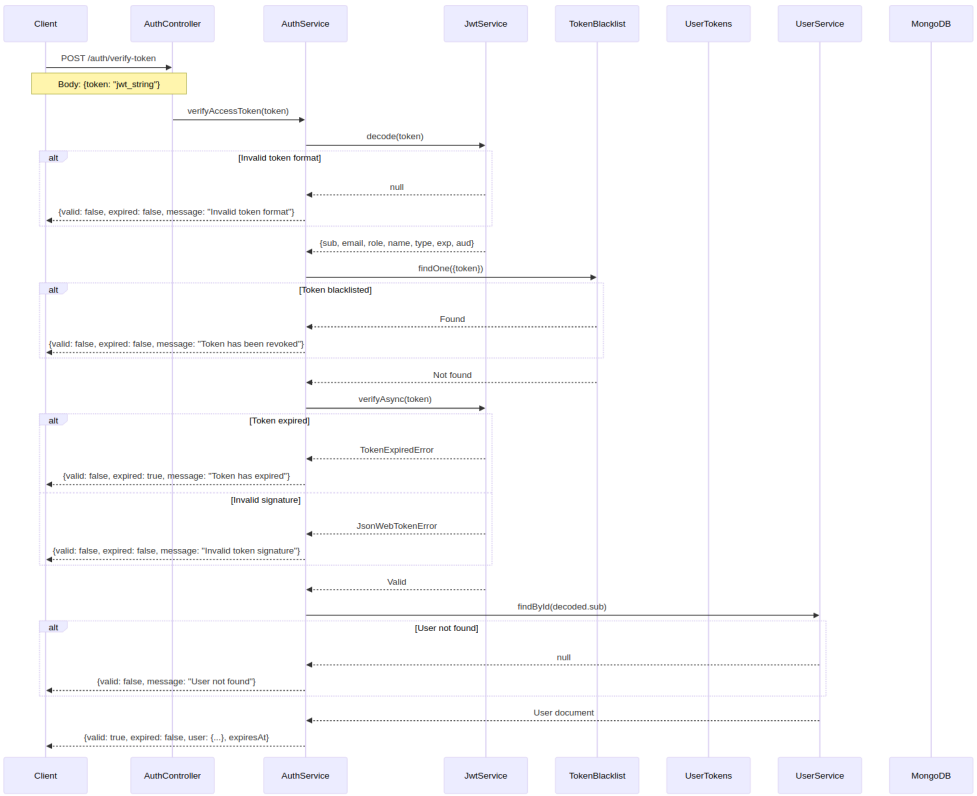


Figure 4.14: Token Verification Flow

Refresh Token Request:

```
POST /auth/refresh-token
Content-Type: application/json

{
  "token": "expired-access-token"
```

```
}
```

Listing 4.26: Refresh Token Request

Refresh Token Response:

```
{
  "accessToken": "new-access-token",
  "subscriptionEnd": 1735689600000,
  "isVerified": true
}
```

Listing 4.27: Refresh Token Response

Logout Request:

```
POST /auth/logout
Authorization: Bearer <access-token>
```

Listing 4.28: Logout Request

Logout Response:

```
{
  "statusCode": 200,
  "message": "Logged out successfully. Your access token has been revoked."
}
```

Listing 4.29: Logout Response

Figure 4.15 shows the logout flow with token blacklisting.

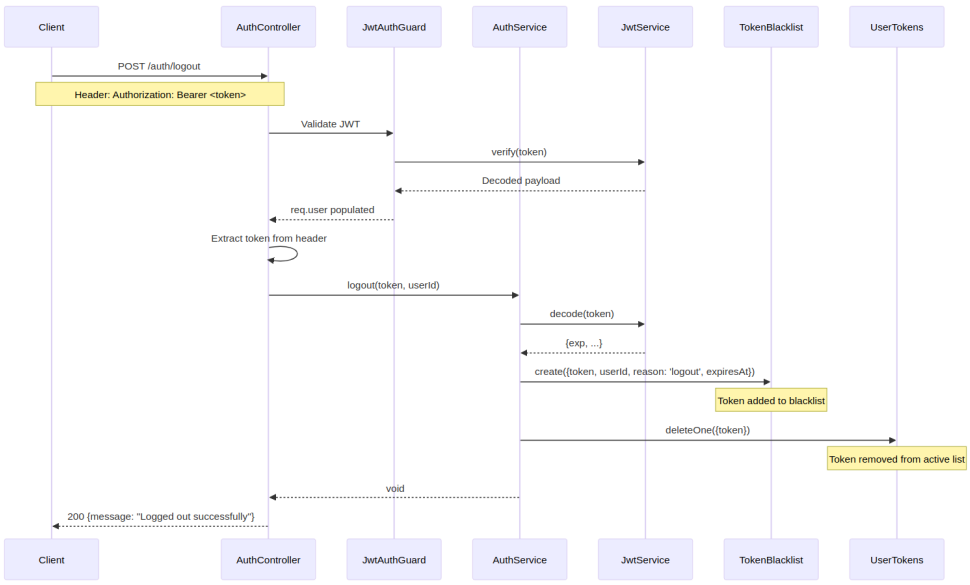


Figure 4.15: Logout Flow

4.1.1.3.8 Token Blacklisting The system implements token blacklisting for security, as shown in Figure 4.16.

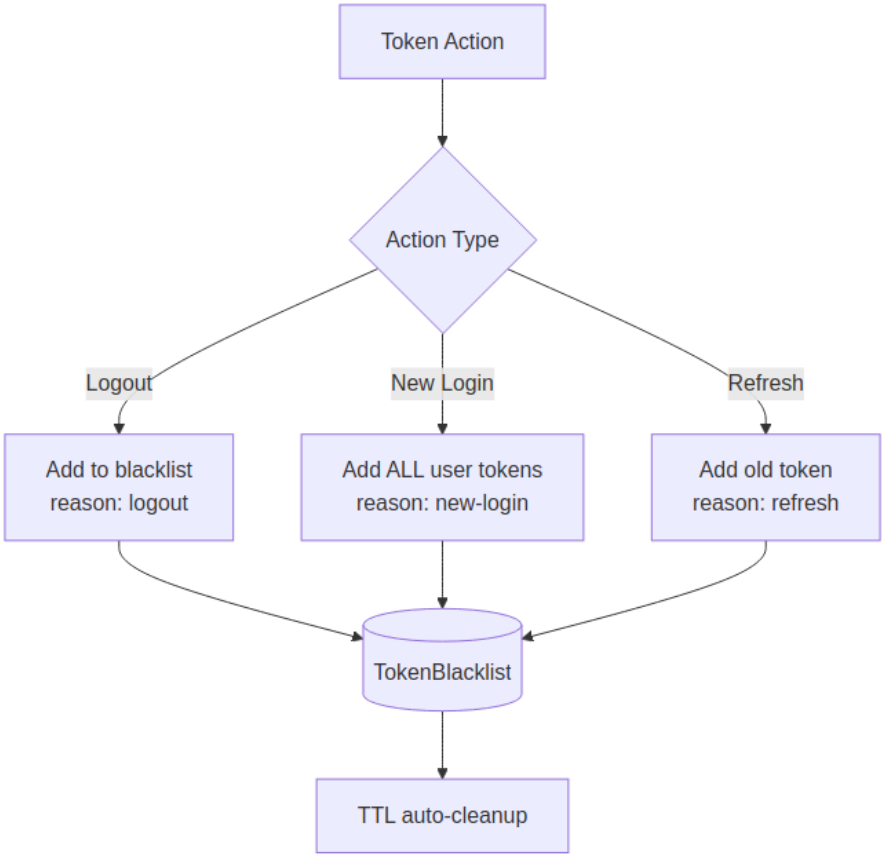


Figure 4.16: Token Blacklisting Flow

Tokens are blacklisted in three scenarios:

1. **Logout**: Current token is added with reason `logout`
2. **New Login**: All user tokens are blacklisted with reason `new-login` (single-session enforcement)
3. **Token Refresh**: Old token is blacklisted with reason `refresh`

4.1.1.3.9 Local Authentication Strategy The `LocalStrategy` validates email/-password credentials using bcrypt:

```
@Injectable()
export class LocalStrategy extends PassportStrategy(Strategy) {
  constructor(private authService: AuthService) {
    super({ usernameField: 'email' })
  }

  async validate(email: string, password: string): Promise<User> {
    const user = await this.authService.validateUser(email, password)
    if (!user) {
      throw new UnauthorizedException('Invalid credentials')
    }
    return user
  }
}
```

Listing 4.30: Local Strategy Implementation

4.1.1.4 Shared SSO Behavior

Both Google and Apple SSO modules share common behavior through the `AuthService`.

4.1.1.4.1 User Flow After SSO Authentication

1. **New Users**: A 15-minute temporary JWT is generated with `need_password: true`. The frontend redirects to `/complete-signup` where the user sets a password.
2. **Existing Users**: All previous tokens are revoked (single-session policy), a new JWT is issued, and stored in the `usertokens` collection.

4.1.1.4.2 Temporary Token Generation The `AuthService` generates a temporary JWT token for new SSO users that expires in 15 minutes:

```
async generateTemporaryToken(
  email: string,
  appId?: string,
  appSecret?: string
): Promise<string> {
  const payload = {
    email: email.toLowerCase(),
    need_password: true,
    appId: appId || null,
    appSecret: appSecret || null,
  }
  return this.jwtService.sign(payload, { expiresIn: '15m' })
}
```

Listing 4.31: Temporary Token Generation for SSO

4.1.1.4.3 Complete Signup Endpoint New OAuth users complete registration by setting a password:

Request:

```
POST /auth/add-info
Content-Type: application/json

{
  "token": "temporary-jwt-token",
  "password": "newSecurePassword123",
  "fullname": "John Doe"
}
```

Listing 4.32: Complete Signup Request

Response:

```
{
  "statusCode": 201,
  "success": true,
  "message": "Account created successfully",
  "accessToken": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...",
  "subscriptionEnd": 1735689600000,
  "isVerified": true
}
```

Listing 4.33: Complete Signup Response

4.1.2 Payment & Subscription Management

This subsection documents the payment processing and subscription management system built with Stripe integration, Redis caching, and MongoDB persistence. The system is organized into three distinct backend modules that handle different aspects of the payment workflow.

4.1.2.1 Architecture Overview

Figure 4.17 illustrates the payment system architecture showing the interaction between modules.

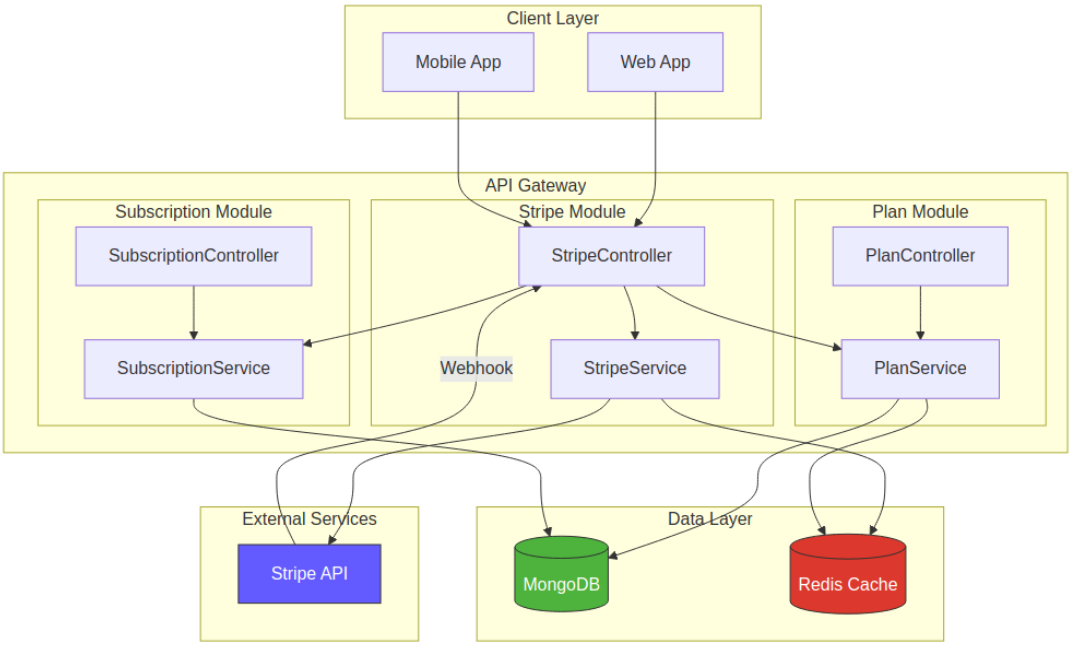


Figure 4.17: Payment System Architecture

The payment system consists of three main modules:

- **Stripe Module**: Handles Stripe API interactions, checkout sessions, webhooks, and billing operations
- **Subscription Module**: Tracks user subscriptions, quotas, and usage
- **Plan Module**: Manages subscription plans with versioning and Redis caching

4.1.2.1.1 Subscription Lifecycle Events

Figure 4.18 illustrates the complete subscription lifecycle from initial purchase through renewal, cancellation, and expiration.

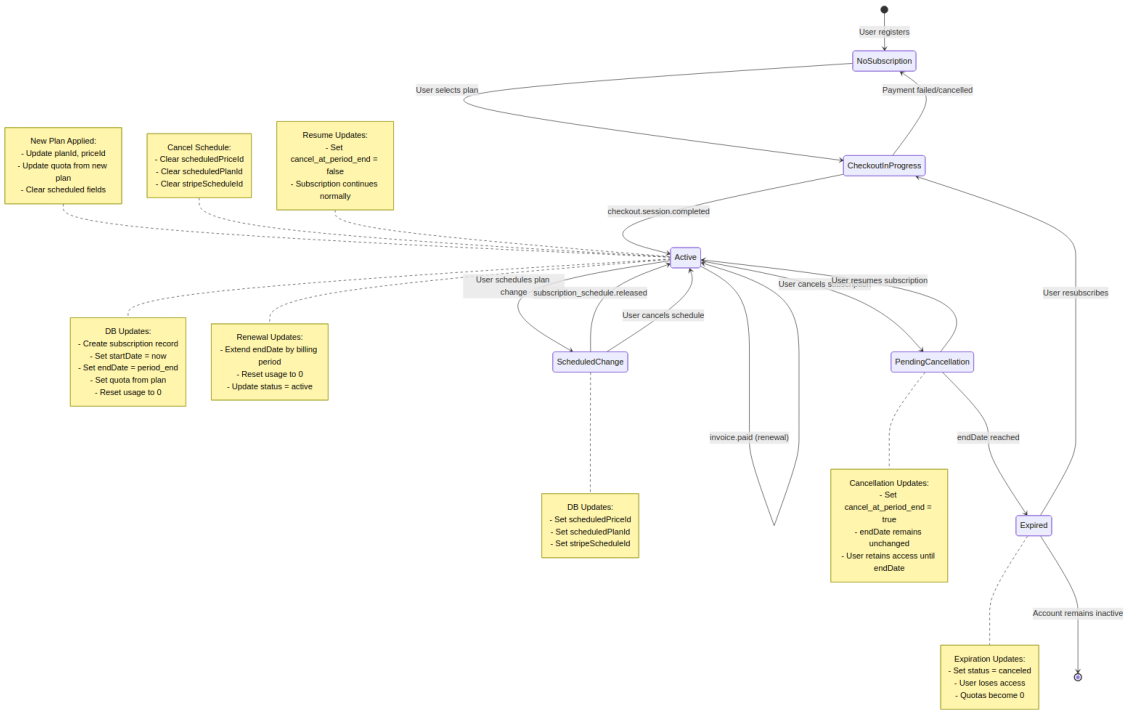


Figure 4.18: Subscription Lifecycle State Diagram

The subscription lifecycle consists of the following key events and their corresponding backend actions:

1. **New Subscription (checkout.session.completed)**:
   - Creates new subscription record in MongoDB
   - Sets `startDate` to current timestamp
   - Sets `endDate` to Stripe's `current_period_end`
   - Assigns quota based on selected plan (`batchDuration`, `liveDuration`)
   - Initializes usage counters to 0
   - Sets status to `active`
2. **Subscription Renewal (invoice.paid)**:
   - Extends `endDate` by the billing period (e.g., +30 days for monthly)
   - Resets `usage.batchDuration` and `usage.liveDuration` to 0
   - Maintains existing quota allocation
   - Updates status to `active` if previously past_due
3. **Subscription Cancellation**:
   - Sets `cancel_at_period_end = true` in Stripe
   - `endDate` remains unchanged (user retains access until period end)

- No immediate database status change
- User can resume subscription before `endDate`

4. **Subscription Expiration**:
   - Triggered when current time exceeds `endDate`
   - Sets status to `canceled`
   - User loses access to premium features
   - Quota checks return failure
5. **Plan Change Scheduling**:
   - Stores `scheduledPriceId` and `scheduledPlanId`
   - New plan takes effect at next billing cycle
   - Quota updates when `subscription_schedule.released` webhook fires

4.1.2.1.2 Post-Payment Database Updates

When Stripe fires the `checkout.session.compl` webhook, the backend performs the following database operations:

```
async handleCheckoutCompleted(session: Stripe.Checkout.Session) {
  const subscription = await this.stripe.subscriptions.retrieve(
    session.subscription as string
  );

  // Get plan details from line items
  const planDetails = await this.getPlanFromPriceId(
    subscription.items.data[0].price.id
  );

  // Create or update subscription in MongoDB
  await this.subscriptionModel.findOneAndUpdate(
    { user: session.metadata.userId },
    {
      user: session.metadata.userId,
      stripeSubscriptionId: subscription.id,
      stripeCustomerId: session.customer as string,
      planId: planDetails.id,
      planName: planDetails.name,
      priceId: subscription.items.data[0].price.id,
      status: 'active',
      startDate: new Date(subscription.current_period_start * 1000),
      endDate: new Date(subscription.current_period_end * 1000),
      quota: {
        batchDuration: planDetails.batchDuration,
        liveDuration: planDetails.liveDuration,
      },
      usage: {
        batchDuration: 0,  // Reset usage on new subscription
        liveDuration: 0,
      },
    },
```

```
    { upsert: true, new: true }
  );
}
```

Listing 4.34: Subscription Creation After Payment

4.1.2.1.3 Renewal and Usage Reset

When a subscription renews, the `invoice.paid` webhook triggers quota reset:

```
async handleInvoicePaid(invoice: Stripe.Invoice) {
  if (invoice.billing_reason !== 'subscription_cycle') return;

  const subscription = await this.stripe.subscriptions.retrieve(
    invoice.subscription as string
  );

  // Extend endDate and reset usage
  await this.subscriptionModel.findOneAndUpdate(
    { stripeSubscriptionId: subscription.id },
    {
      endDate: new Date(subscription.current_period_end * 1000),
      status: 'active',
      'usage.batchDuration': 0,  // Reset usage for new billing period
      'usage.liveDuration': 0,
    }
  );
}
```

Listing 4.35: Subscription Renewal Handler

4.1.2.1.4 Cancellation Expiry Handling

When a user cancels, the subscription remains active until the billing period ends:

```
async cancelSubscription(subscriptionId: string) {
  // Cancel at period end (not immediately)
  const subscription = await this.stripe.subscriptions.update(
    subscriptionId,
    { cancel_at_period_end: true }
  );

  // Update local record - endDate stays the same
  await this.subscriptionModel.findOneAndUpdate(
    { stripeSubscriptionId: subscriptionId },
    {
      // endDate remains unchanged - user keeps access until then
      cancelAtPeriodEnd: true,
      canceledAt: new Date(),
    }
  );

  return {
    status: subscription.status,
    cancelAt: subscription.cancel_at,
```

```
    endDate: new Date(subscription.current_period_end * 1000),
    message: 'Subscription will remain active until ' +
             new Date(subscription.current_period_end * 1000).toISOString(),
  };
}
```

Listing 4.36: Cancellation with Period-End Expiry

4.1.2.2 Stripe Module

The Stripe Module handles all payment-related operations including checkout sessions, subscription lifecycle management, plan scheduling, and webhook processing. It integrates with both the Plan and Subscription modules.

4.1.2.2.1 Module Structure The Stripe Module imports both Subscription and Plan schemas to manage cross-module operations. It configures the Stripe SDK through a dedicated configuration module and provides services for all payment-related functionality.

```
@Module({
  imports: [
    StripeConfigurationModule,
    MongooseModule.forFeature([
      { name: Subscription.name, schema: SubscriptionSchema },
      { name: Plan.name, schema: PlanSchema },
    ]),
  ],
  controllers: [StripeController],
  providers: [StripeService, SubscriptionService, PlanService],
  exports: [StripeService],
})
export class StripeModule {}
```

Listing 4.37: Stripe Module Configuration

4.1.2.2.2 Checkout Flow Figure 4.19 shows the complete Stripe Checkout flow from user request to subscription creation.

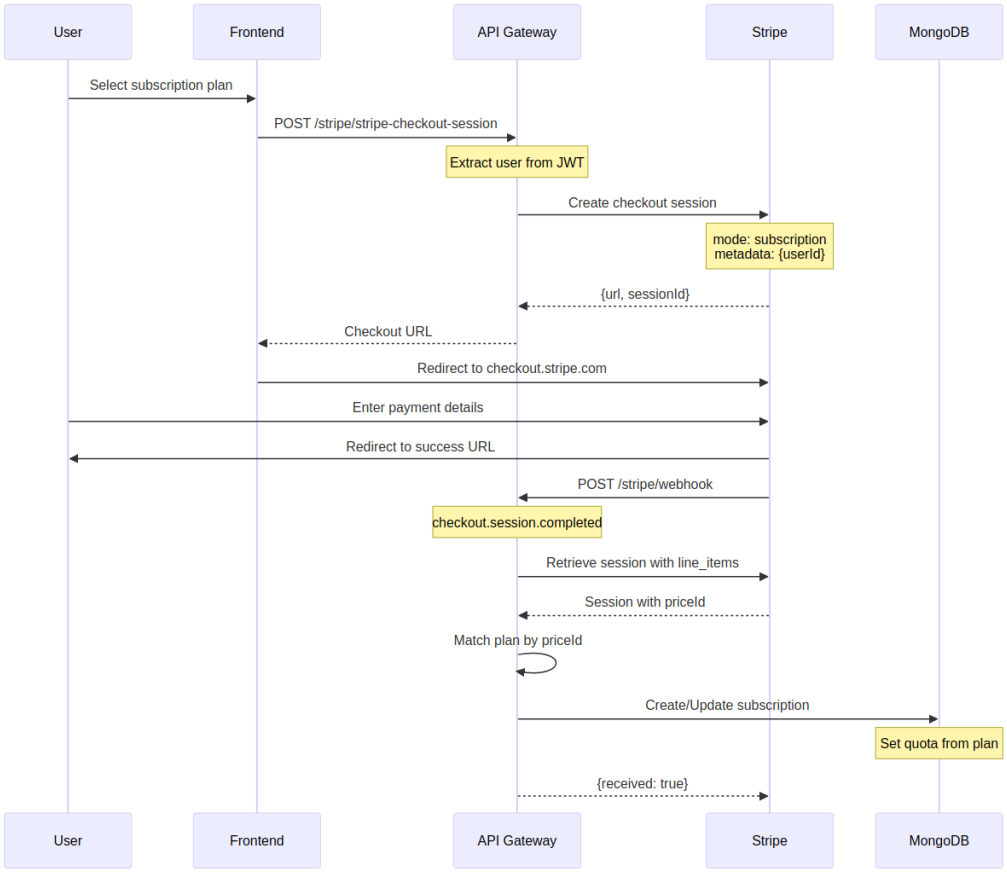


Figure 4.19: Stripe Checkout Session Flow

4.1.2.2.3 API Endpoints Table 4.4 provides a complete list of Stripe module endpoints.

Table 4.4: Stripe Module API Endpoints

Method	Endpoint	Auth	Description
POST	/stripe/stripe-checkout-session	JWT	Create checkout session
POST	/stripe/check-subscription-status	JWT	Check subscription status
POST	/stripe/cancel-subscription	JWT	Cancel subscription
POST	/stripe/resume-subscription	JWT	Resume cancelled subscription
POST	/stripe/get-price	None	Get price information
POST	/stripe/schedule-plan-change	JWT	Schedule plan change
POST	/stripe/get-schedule-info	JWT	Get schedule info
POST	/stripe/cancel-scheduled-change	JWT	Cancel scheduled change
POST	/stripe/webhook	Stripe	Handle Stripe webhooks

Create Checkout Session Creates a Stripe Checkout session for subscription purchase. User metadata is attached for webhook processing.

Request:

```
POST /stripe/stripe-checkout-session
Authorization: Bearer <access-token>
```

```
Content-Type: application/json

{
  "priceId": "price_1234567890"
}
```

Listing 4.38: Create Checkout Session Request

Response:

```
{
  "url": "https://checkout.stripe.com/c/pay/cs_test_...",
  "sessionId": "cs_test_a1b2c3d4e5f6"
}
```

Listing 4.39: Create Checkout Session Response

Check Subscription Status Retrieves detailed subscription status including cancellation information.

Request:

```
POST /stripe/check-subscription-status
Authorization: Bearer <access-token>
Content-Type: application/json

{
  "subscriptionId": "sub_1234567890"
}
```

Listing 4.40: Check Subscription Status Request

Response:

```
{
  "subscriptionId": "sub_1234567890",
  "status": "active",
  "isCancelled": false,
  "canceledAt": null,
  "cancelAtPeriodEnd": false,
  "currentPeriodEnd": 1704067199
}
```

Listing 4.41: Check Subscription Status Response

Cancel Subscription Cancels a subscription at the end of the current billing period.

Request:

```
POST /stripe/cancel-subscription
Authorization: Bearer <access-token>
Content-Type: application/json

{
  "subscriptionId": "sub_1234567890"
}
```

Listing 4.42: Cancel Subscription Request

Response:

```
{
  "subscriptionId": "sub_1234567890",
  "status": "active",
  "cancelAt": 1704067199,
  "cancelAtPeriodEnd": true,
  "message": "Subscription will be cancelled at the end of the current billing
   period"
}
```

Listing 4.43: Cancel Subscription Response

Resume Subscription Reverts a pending cancellation, keeping the subscription active.

Request:

```
POST /stripe/resume-subscription
Authorization: Bearer <access-token>
Content-Type: application/json

{
  "subscriptionId": "sub_1234567890"
}
```

Listing 4.44: Resume Subscription Request

Response:

```
{
  "subscriptionId": "sub_1234567890",
  "status": "active",
  "cancelAt": null,
  "cancelAtPeriodEnd": false
}
```

Listing 4.45: Resume Subscription Response

Get Price Information Retrieves Stripe price details with Redis caching.

Request:

```
POST /stripe/get-price
Content-Type: application/json

{
  "price_id": "price_1234567890"
}
```

Listing 4.46: Get Price Request

Response:

```
{
  "id": "price_1234567890",
  "currency": "usd",
  "unit_amount": 1999,
```

```
  "recurring": {
    "interval": "month",
    "interval_count": 1
  }
}
```

Listing 4.47: Get Price Response

4.1.2.2.4 Subscription Scheduling To support plan upgrades and downgrades without immediate billing impact, the system implements a scheduling mechanism using Stripe Subscription Schedules.

Scheduling Flow Figure 4.20 details the sequence for scheduling a plan change.

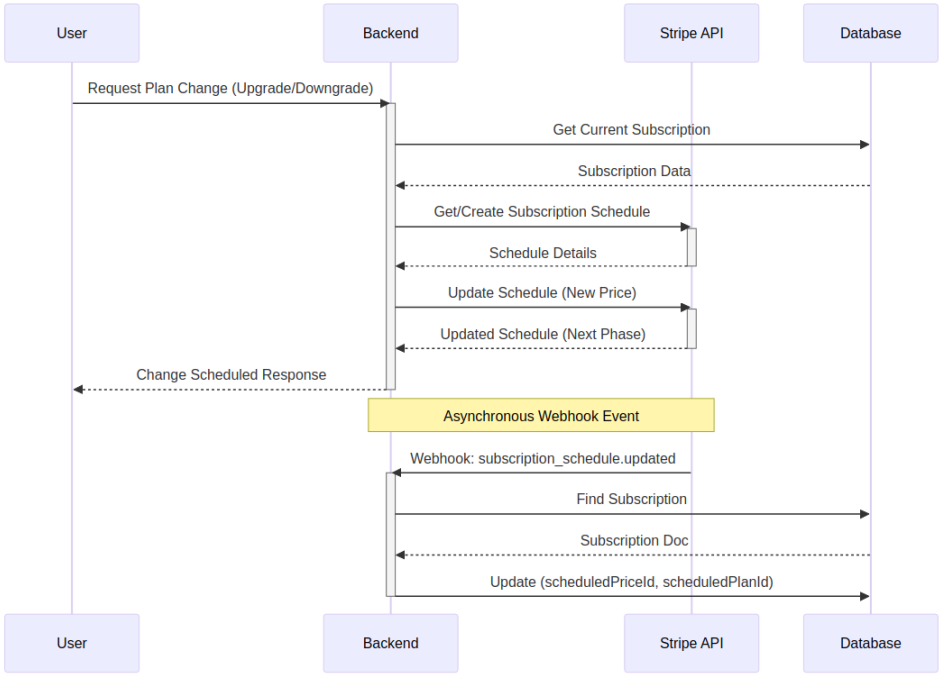


Figure 4.20: Subscription Plan Change Scheduling Sequence

The scheduling process follows these steps:

1. Retrieves current subscription details to identify the `current_period_end`.
2. Checks for any existing subscription schedules to prevent conflicts.
3. If a schedule exists, updates the subsequent phase with the new price.
4. If no schedule exists, creates a new one from the current subscription.

Advanced Scheduling Logic The implementation handles complex scenarios such as modifying an already scheduled change:

```
async schedulePlanChange(subscriptionId: string, newPriceId: string) {
  // 1. Get current subscription
  const subscription = await this.stripe.subscriptions.retrieve(
```

```
    subscriptionId, { expand: ['items'] }
  );

  // 2. Check for existing schedules
  const schedules = await this.stripe.subscriptionSchedules.list({
    customer: subscription.customer as string,
  });
  const activeSchedule = schedules.data.find(s =>
    s.subscription === subscriptionId &&
    (s.status === 'active' || s.status === 'not_started')
  );

  if (activeSchedule) {
    // 3a. Update existing schedule
    const currentPhase = activeSchedule.phases[0];
    return this.stripe.subscriptionSchedules.update(activeSchedule.id, {
      phases: [
        { ...currentPhase },
        { items: [{ price: newPriceId, quantity: 1 }] }
      ]
    });
  }

  // 3b. Create new schedule from subscription
  const schedule = await this.stripe.subscriptionSchedules.create({
    from_subscription: subscriptionId,
  });

  return this.stripe.subscriptionSchedules.update(schedule.id, {
    phases: [
      { ...schedule.phases[0] },
      { items: [{ price: newPriceId, quantity: 1 }] }
    ]
  });
}
```

Listing 4.48: Smart Scheduling Logic in StripeService

Schedule Plan Change Schedules a plan change to take effect at the next billing period.

Request:

```
POST /stripe/schedule-plan-change
Authorization: Bearer <access-token>
Content-Type: application/json

{
  "subscriptionId": "sub_1234567890",
  "newPriceId": "price_pro_monthly"
}
```

Listing 4.49: Schedule Plan Change Request

Response:

```
{
  "scheduleId": "sub_sched_abc123",
  "subscriptionId": "sub_1234567890",
  "status": "active",
  "currentPhaseEnd": 1704067199,
  "nextPriceId": "price_pro_monthly",
  "message": "Plan change scheduled for next billing period"
}
```

Listing 4.50: Schedule Plan Change Response

Get Schedule Info Retrieves information about a pending scheduled plan change.

Request:

```
POST /stripe/get-schedule-info
Authorization: Bearer <access-token>
Content-Type: application/json

{
  "subscriptionId": "sub_1234567890"
}
```

Listing 4.51: Get Schedule Info Request

Response:

```
{
  "hasSchedule": true,
  "scheduleId": "sub_sched_abc123",
  "status": "active",
  "currentPhaseEnd": 1704067199,
  "nextPriceId": "price_pro_monthly"
}
```

Listing 4.52: Get Schedule Info Response

Cancel Scheduled Change Cancels a pending scheduled plan change.

Request:

```
POST /stripe/cancel-scheduled-change
Authorization: Bearer <access-token>
Content-Type: application/json

{
  "scheduleId": "sub_sched_abc123"
}
```

Listing 4.53: Cancel Scheduled Change Request

Response:

```
{
  "scheduleId": "sub_sched_abc123",
  "status": "released",
```

```
  "message": "Scheduled plan change cancelled successfully"
}
```

Listing 4.54: Cancel Scheduled Change Response

4.1.2.2.5 Webhook Processing Webhooks are central to maintaining synchronization between Stripe and the local database.

Webhook Flowchart Figure 4.21 illustrates the decision logic for handling various Stripe events.

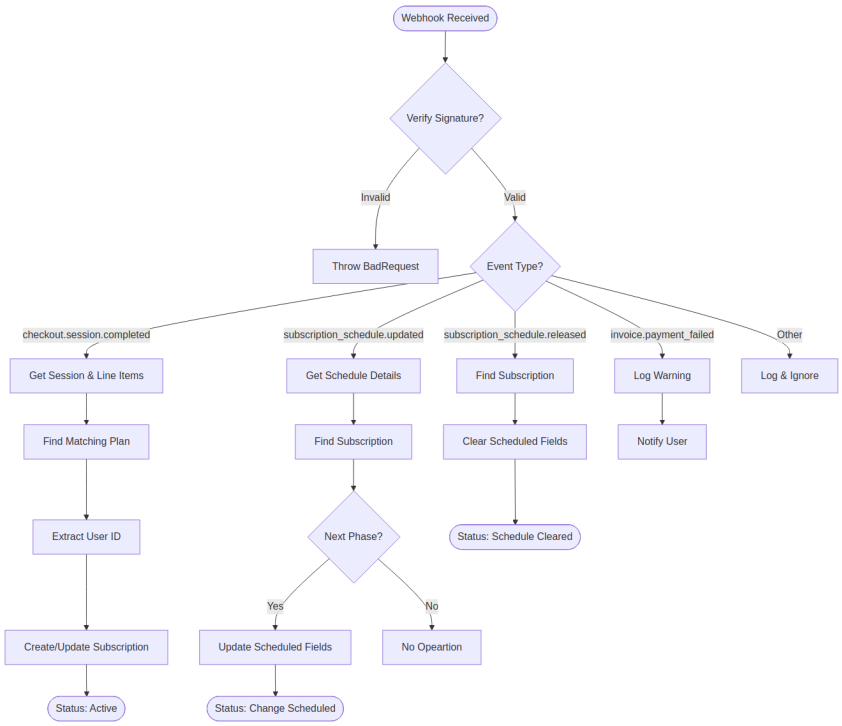


Figure 4.21: Stripe Webhook Processing Logic

The webhook handler processes the following event types:

- `checkout.session.completed`: Activates new subscriptions
- `subscription_schedule.updated`: Updates local DB with scheduled plan details
- `subscription_schedule.released`: Clears scheduled fields when changes are cancelled

Webhook Endpoint Receives and processes Stripe webhook events.
Request:

```
POST /stripe/webhook
stripe-signature: t=1234567890,v1=abc123...
Content-Type: application/json

{
  "id": "evt_1234567890",
```

```
  "type": "checkout.session.completed",
  "data": {
    "object": {
      "id": "cs_test_abc123",
      "customer": "cus_abc123",
      "subscription": "sub_abc123"
    }
  }
}
```

Listing 4.55: Stripe Webhook Request

Response:

```
{
  "received": true
}
```

Listing 4.56: Stripe Webhook Response

Webhook Handler Implementation The webhook handler implementation processes different event types:

```
@Post('webhook')
async handleWebhook(
  @Headers('stripe-signature') signature: string,
  @Req() req: Request
) {
  const payload = req.body as Buffer;
  const event = await this.stripeService.handleWebhook(payload, signature);

  switch (event.type) {
    case 'checkout.session.completed':
      const session = event.data.object;
      const fullSession = await this.stripeService.getSessionWithLineItems(
        session.id
      );
      // Create subscription record with plan details
      break;

    case 'subscription_schedule.updated':
      const schedule = event.data.object;
      const nextPhase = schedule.phases[1];
      if (nextPhase) {
        await this.subscriptionService.updateById(subId, {
          scheduledPriceId: nextPhase.items[0].price,
          scheduledPlanId: nextPhase.plan,
        });
      }
      break;

    case 'subscription_schedule.released':
      // Clear scheduled fields
      await this.subscriptionService.updateById(subscription._id, {
```

```
        scheduledPriceId: null,
        scheduledPlanId: null,
        stripeScheduleId: null,
      });
      break;
  }
  return { received: true };
}
```

Listing 4.57: Stripe Webhook Handler

4.1.2.2.6 Price Caching Figure 4.22 shows the price caching mechanism with cache penetration protection.

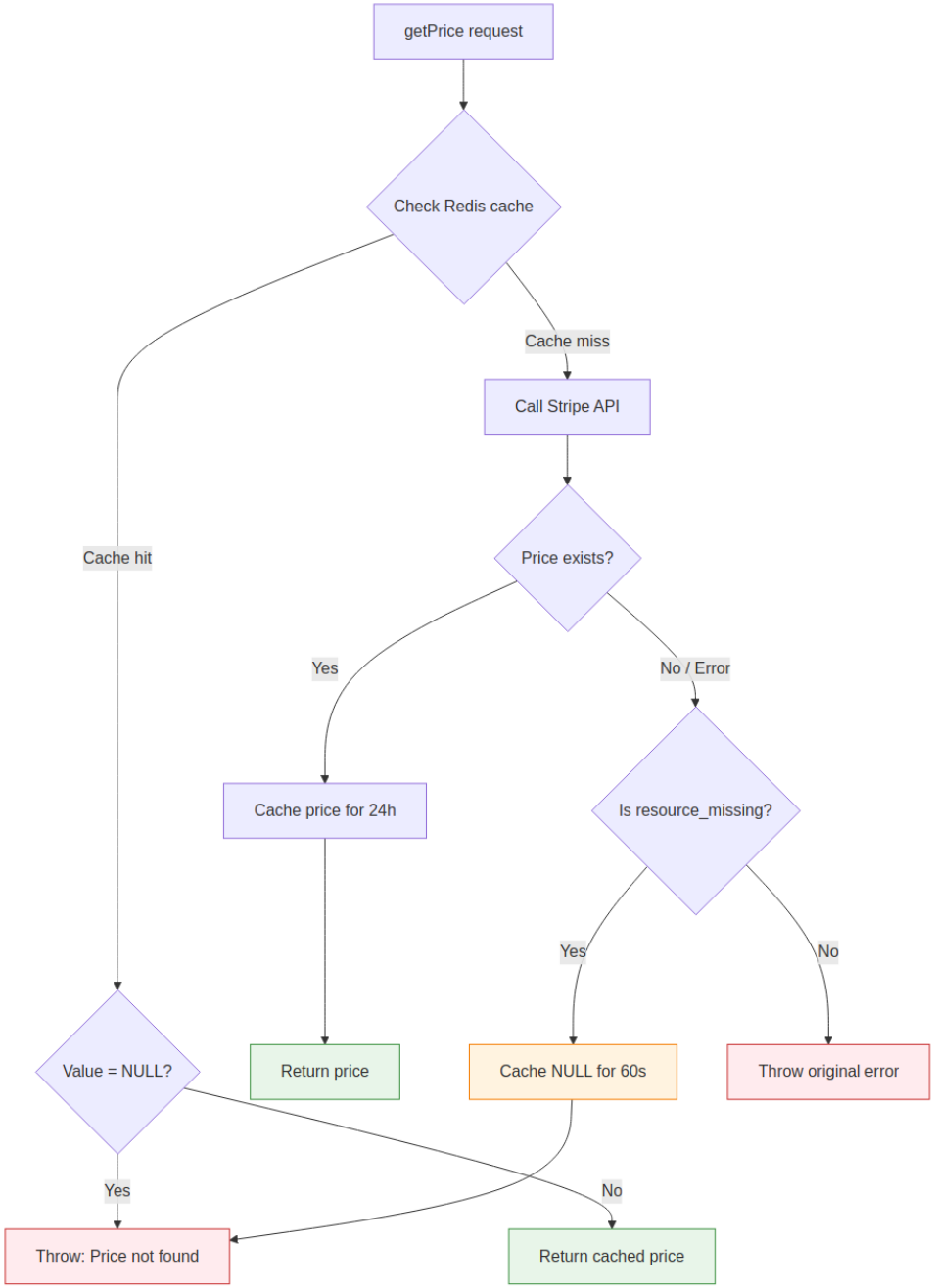


Figure 4.22: Stripe Price Caching with Penetration Protection

```
async getPrice(priceId: string): Promise<Stripe.Price> {
  const cacheKey = `stripe:price:${priceId}`;
  const cached = await this.redisClient.get(cacheKey);

  if (cached) {
    if (cached === 'NULL') throw new Error('Price not found (cached)');
    return JSON.parse(cached) as Stripe.Price;
  }

  try {
    const price = await this.stripe.prices.retrieve(priceId);
    await this.redisClient.set(cacheKey, JSON.stringify(price), {
```

```
      EX: 86400, // 24 hours
    });
    return price;
  } catch (error) {
    if (error?.code === 'resource_missing') {
      await this.redisClient.set(cacheKey, 'NULL', { EX: 60 });
    }
    throw error;
  }
}
```

Listing 4.58: Stripe Price Caching Implementation

4.1.2.3 Subscription Module

The Subscription Module tracks user subscriptions, manages quotas, monitors usage, and handles scheduled plan changes. It serves as the central source of truth for user subscription state.

4.1.2.3.1 Module Structure The Subscription Module follows a similar architecture pattern, providing dedicated endpoints for subscription management and exporting the service for use by other modules.

```
@Module({
  imports: [
    MongooseModule.forFeature([
      { name: Subscription.name, schema: SubscriptionSchema },
    ]),
  ],
  controllers: [SubscriptionController],
  providers: [SubscriptionService],
  exports: [SubscriptionService],
})
export class SubscriptionModule {}
```

Listing 4.59: Subscription Module Structure

4.1.2.3.2 Data Model The Subscription schema captures comprehensive subscription state including Stripe integration fields, quota management, usage tracking, and scheduled plan changes. The schema uses MongoDB's embedded document pattern for quota and usage objects.

```
@Schema({ timestamps: true })
export class Subscription extends Document {
  @Prop({ required: true, ref: User.name, type: ObjectId })
  user: string | User;

  // Stripe integration fields
  @Prop({ type: String }) stripeSubscriptionId?: string;
  @Prop({ type: String }) stripeCustomerId?: string;
  @Prop({ type: String }) planName?: string;
  @Prop({ type: String }) priceId?: string;
```

```
  @Prop({ type: String }) planId?: string;

  // Scheduled change fields (for plan upgrades/downgrades)
  @Prop({ type: String }) scheduledPriceId?: string;
  @Prop({ type: String }) scheduledPlanId?: string;
  @Prop({ type: String }) scheduledPlanName?: string;
  @Prop({ type: String }) stripeScheduleId?: string;

  // Subscription status
  @Prop({
    type: String,
    enum: ['active', 'canceled', 'incomplete', 'incomplete_expired',
           'past_due', 'trialing', 'unpaid', 'paused'],
    default: 'active'
  })
  status: string;

  // Quota and usage tracking
  @Prop(raw({ batchDuration: Number, liveDuration: Number }))
  quota: SubscriptionUnit;

  @Prop(raw({ batchDuration: Number, liveDuration: Number }))
  usage: SubscriptionUnit;

  // Validity period
  @Prop({ type: Date }) startDate: Date;
  @Prop({ type: Date }) endDate: Date;
}
```

Listing 4.60: Subscription Schema

4.1.2.3.3 Quota Checking Figure 4.23 illustrates the subscription and quota validation flow used before processing requests.

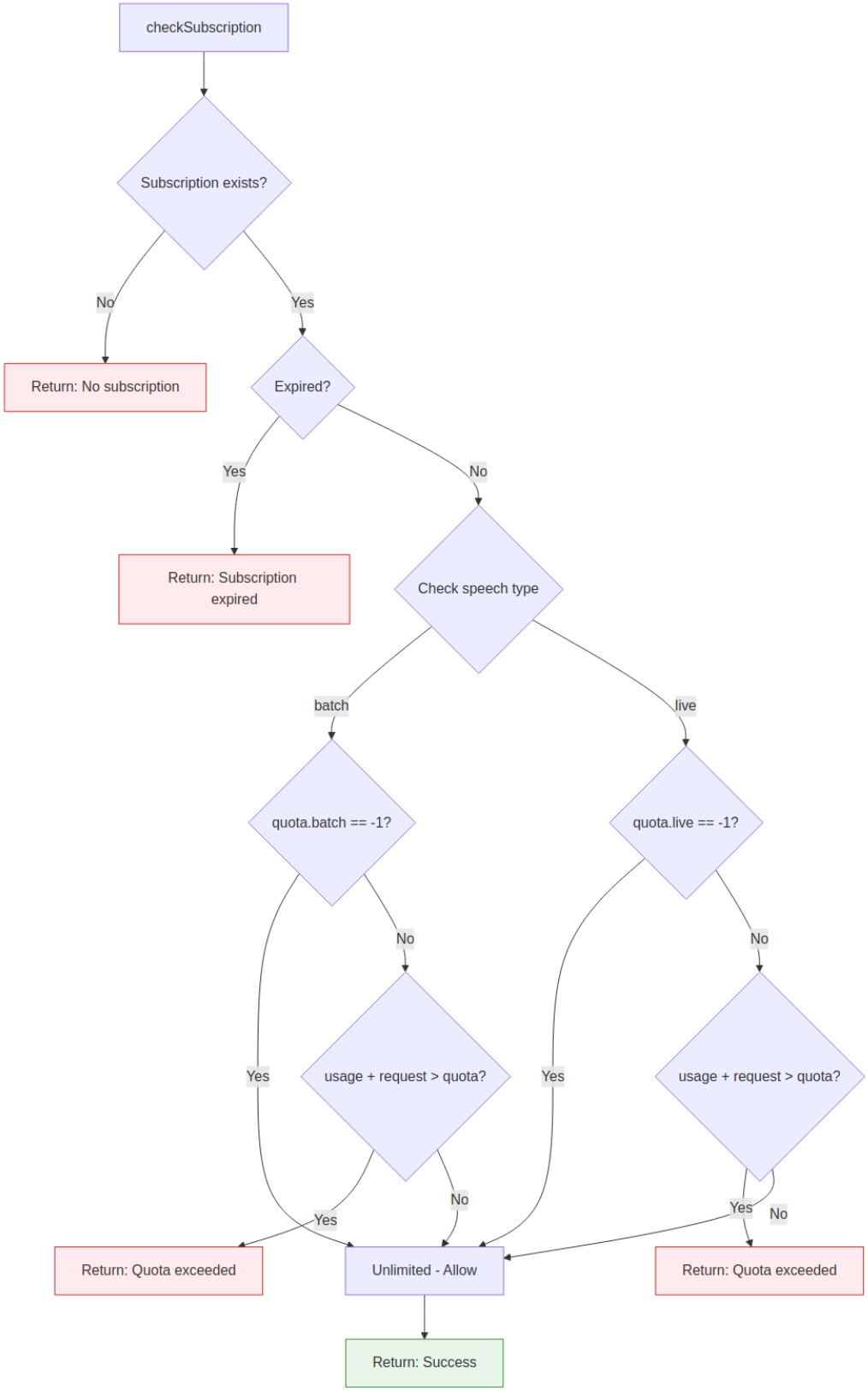


Figure 4.23: Subscription Quota Checking Flow

The quota checking logic validates user subscriptions and usage limits:

```
async checkSubscription(userId: string, speechType?: 'live' | 'batch') {
  const sub = await this.subscriptionModel.findOne({ user: userId }).exec();

```

```
  if (!sub) {
    return { success: false, message: 'User has no subscription' };
  }

  // Check batch quota (-1 means unlimited)
  if (speechType === 'batch' && sub.quota.batchDuration !== -1 &&
      sub.usage.batchDuration > sub.quota.batchDuration) {
    return { success: false, message: 'Batch quota exceeded' };
  }

  // Check live quota
  if (speechType === 'live' && sub.quota.liveDuration !== -1 &&
      sub.usage.liveDuration > sub.quota.liveDuration) {
    return { success: false, message: 'Live quota exceeded' };
  }

  return { success: true, subscription: sub };
}
```

Listing 4.61: Subscription Quota Validation

4.1.2.3.4 API Endpoints Table 4.5 provides a complete list of Subscription module endpoints.

Table 4.5: Subscription Module API Endpoints

Method	Endpoint	Auth	Description
GET	/subscriptions/user/:id	Admin	Get user subscription
PUT	/subscriptions/user/:id	Admin	Update user subscription

Get User Subscription (Admin) Retrieves subscription details for a specific user.

Request:

```
GET /subscriptions/user/6507a1b2c3d4e5f6a7b8c9d0
Authorization: Bearer <admin-token>
```

Listing 4.62: Get User Subscription Request

Response:

```
{
  "_id": "sub_123456",
  "user": "6507a1b2c3d4e5f6a7b8c9d0",
  "status": "active",
  "planName": "Pro",
  "quota": {
    "batchDuration": 3600,
    "liveDuration": 1800
  },
  "usage": {
    "batchDuration": 1200,
    "liveDuration": 600
  },
```

```
  "startDate": "2024-01-01T00:00:00Z",
  "endDate": "2024-12-31T23:59:59Z"
}
```

Listing 4.63: Get User Subscription Response

Update User Subscription (Admin) Allows administrators to manually update subscription quotas or extend validity periods.

Request:

```
PUT /subscriptions/user/6507a1b2c3d4e5f6a7b8c9d0
Authorization: Bearer <admin-token>
Content-Type: application/json

{
  "endDate": "2025-12-31",
  "batchDuration": 7200,
  "liveDuration": 3600
}
```

Listing 4.64: Update User Subscription Request

Response:

```
{
  "_id": "sub_123456",
  "user": "6507a1b2c3d4e5f6a7b8c9d0",
  "quota": {
    "batchDuration": 7200,
    "liveDuration": 3600
  },
  "endDate": "2025-12-31T23:59:59Z"
}
```

Listing 4.65: Update User Subscription Response

4.1.2.4 Plan Module

The Plan Module manages subscription plan definitions with versioning support and high-performance Redis caching. Plans are stored as versioned JSON documents, allowing for plan updates without affecting existing subscribers.

4.1.2.4.1 Module Structure The Plan Module follows the standard NestJS module architecture with a controller for handling HTTP requests, a service for business logic, and MongoDB integration via Mongoose.

```
@Module({
  imports: [
    MongooseModule.forFeature([{ name: Plan.name, schema: PlanSchema }]),
  ],
  controllers: [PlanController],
  providers: [PlanService],
  exports: [PlanService],
})
```

```
export class PlanModule {}
```

Listing 4.66: Plan Module Structure

4.1.2.4.2 Data Model

Plans are stored with automatic versioning to support historical tracking and rollback capabilities. Each plan document contains a unique identifier, a JSON string with plan details, and an auto-incrementing version number.

```
@Schema({ timestamps: { createdAt: true, updatedAt: false } })
export class Plan extends Document {
  @Prop({ required: true, type: String, unique: true })
  id: string;

  @Prop({ required: true, type: String })
  plans: string; // JSON string containing plan details

  @Prop({ required: true, type: Number, unique: true })
  version: number;

  createdAt?: Date;
}
```

Listing 4.67: Plan Schema Definition

4.1.2.4.3 Plan JSON Format

The `plans` field stores a stringified JSON object with the following structure:

```
{
  "plans": [
    {
      "id": string,           // Unique plan identifier
      "name": string,         // Display name
      "priceId": string,      // Stripe price ID
      "features": string[],   // List of feature descriptions
      "recommended": boolean, // Optional: highlight flag
      "batchDuration": number,// Quota in seconds (-1 for unlimited)
      "liveDuration": number  // Quota in seconds (-1 for unlimited)
    }
  ]
}
```

Listing 4.68: Plan JSON Structure

4.1.2.4.4 Plan JSON Example

A typical plan configuration includes Free, Basic, and Pro tiers:

```
{
  "plans": [
    {
      "id": "basic",
      "name": "Basic",
      "priceId": "price_1NqKx2LkdIwHu7ixsKR4L2nS",
```

```
      "features": [
        "60 minutes batch processing per month",
        "30 minutes live transcription per month",
        "Priority support",
        "High quality audio processing"
      ],
      "recommended": false,
      "batchDuration": 3600,
      "liveDuration": 1800
    },
    {
      "id": "pro",
      "name": "Pro",
      "priceId": "price_1NqKxBLkdIwHu7ixCkcQHZgD",
      "features": [
        "Unlimited batch processing",
        "Unlimited live transcription",
        "24/7 Premium support",
        "Highest quality audio processing",
        "Custom integrations"
      ],
      "recommended": true,
      "batchDuration": -1,
      "liveDuration": -1
    }
  ]
}
```

Listing 4.69: Plan JSON Example

4.1.2.4.5 Plan Versioning Figure 4.24 shows how plan versions are managed over time, enabling seamless plan updates.

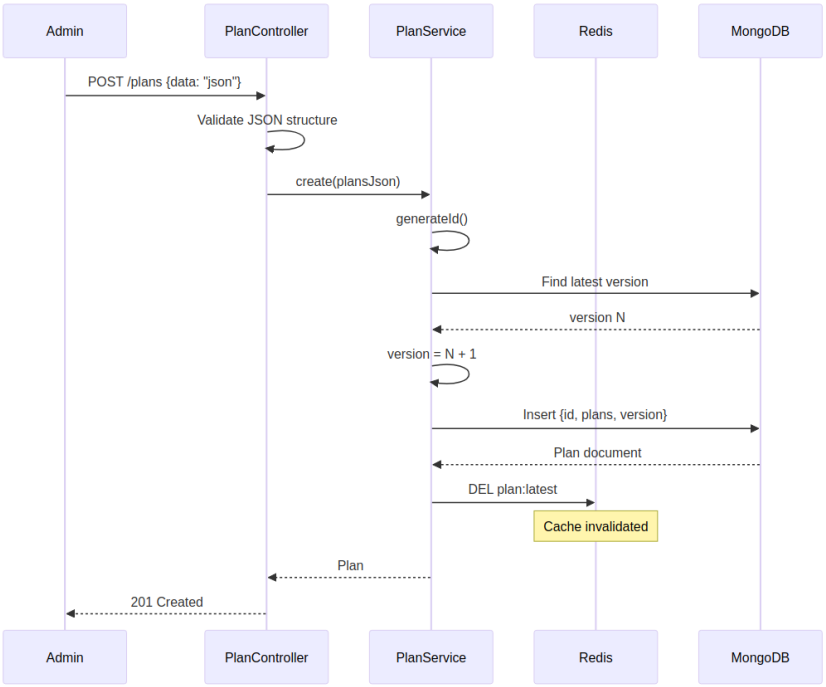


Figure 4.24: Plan Versioning Flow

4.1.2.4.6 Redis Caching with Cache Penetration Protection The service implements a cache-aside pattern with NULL value caching to prevent cache penetration attacks. Figure 3.2 illustrates this caching strategy.

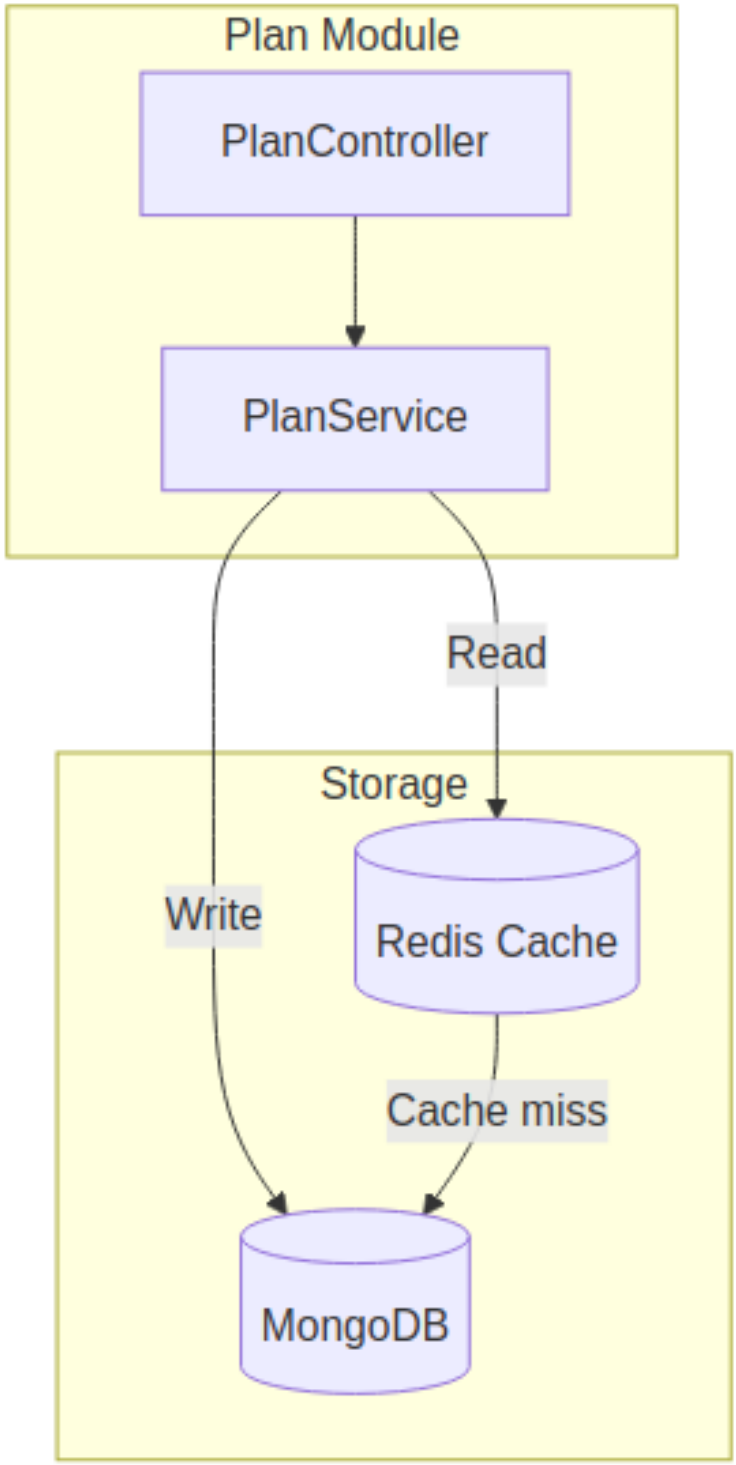


Figure 4.25: Plan Caching with Penetration Protection

```
@Injectable()
export class PlanService implements OnModuleInit, OnModuleDestroy {
  private readonly CACHE_KEY = 'plan:latest';
  private readonly CACHE_TTL = 3600; // 1 hour
  private readonly NULL_TTL = 60; // 60 seconds for null cache

  async getLatestVersion(): Promise<Plan | null> {
```

```
    // Check cache first
    const cached = await this.redisClient.get(this.CACHE_KEY);
    if (cached) {
      if (cached === 'NULL') return null; // Cache penetration protection
      return JSON.parse(cached) as Plan;
    }

    // Query database
    const latestPlan = await this.planModel
      .findOne().sort({ version: -1 }).exec();

    // Cache result (including null to prevent penetration)
    if (latestPlan) {
      await this.redisClient.set(this.CACHE_KEY, JSON.stringify(latestPlan), {
        EX: this.CACHE_TTL,
      });
    } else {
      await this.redisClient.set(this.CACHE_KEY, 'NULL', {
        EX: this.NULL_TTL,
      });
    }
    return latestPlan;
  }
}
```

Listing 4.70: Plan Caching Implementation

4.1.2.4.7 API Endpoints Table 4.6 provides a complete list of Plan module endpoints.

Table 4.6: Plan Module API Endpoints

Method	Endpoint	Auth	Description
GET	/plans/latest	None	Get latest plan version
POST	/plans	Admin	Create new plan version

Get Latest Plans Retrieves the most recent version of subscription plans. This endpoint is publicly accessible.

Request:

```
GET /plans/latest
```

Listing 4.71: Get Latest Plans Request

Response:

```
{
  "id": "plan_abc123",
  "plans": "{\"free\": {...}, \"pro\": {...}}",
  "version": 3,
  "createdAt": "2024-01-15T10:30:00Z"
}
```

Listing 4.72: Get Latest Plans Response

Create Plans (Admin) Creates a new plan version. Requires admin privileges. The system automatically increments the version number and invalidates the Redis cache.
Request:

```
POST /plans
Authorization: Bearer <admin-token>
Content-Type: application/json

{
  "data": "{\"plans\": [{\"name\": \"Free\", ...}, {\"name\": \"Pro\", ...}]}"
}
```

Listing 4.73: Create Plans Request

Response:

```
{
  "id": "plan_def456",
  "plans": "{...}",
  "version": 4,
  "createdAt": "2024-01-20T14:00:00Z"
}
```

Listing 4.74: Create Plans Response

4.2 Frontend Implementation

This section details the frontend implementation of the Gateway Dashboard, including the Vue.js architecture, component structure, routing implementation, and subscription management interface.

4.2.1 Overview

The Gateway Dashboard is a Vue.js 3 web application built with the Composition API and Vite, providing a modern management interface for the Speech Gateway API with real-time subscription management and authentication.

4.2.1.1 Frontend Architecture

Figure 4.26 illustrates the layered architecture of the frontend application.

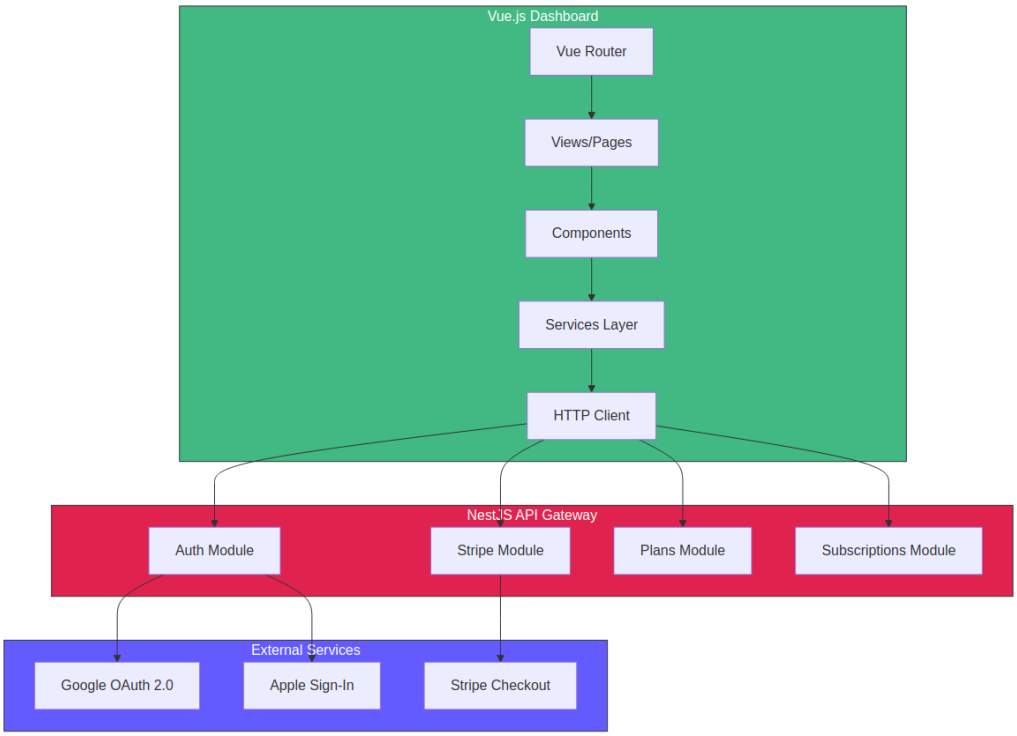


Figure 4.26: Frontend Application Architecture

The frontend follows a modern layered architecture with Vue 3 Composition API:

- **Vue Router 4**: Client-side routing with navigation guards and beforeEach hooks
- **Views/Pages**: Page-level components using Composition API and composables
- **Composables**: Reusable composition functions (useAuth, useGlobalState, useSnackbar)
- **Components**: Reusable UI components like PlanCard with reactive properties
- **Services Layer**: API communication with fetch and Axios for authenticated requests
- **HTTP Interceptors**: Automatic token refresh and error handling middleware
- **Global State**: Reactive state management using Vue's provide/inject pattern

4.2.1.2 Technology Stack

Layer	Technology	Purpose
Framework	Vue.js 3.5.13	Reactive UI with Composition API
Build Tool	Vite 6.x	Fast development and HMR
UI Library	Vuetify 3.8.0	Material Design 3 components
HTTP Client	Axios 1.7.9	Authenticated API requests
Router	Vue Router 4.5.0	SPA navigation with guards
State	Provide/Inject	Global reactive state
JWT	jwt-decode 4.0.0	Token decoding and validation
Payment	Stripe Checkout	Subscription processing
Charts	Chart.js 4.4.7	Data visualization
Editor	JSONEditor 10.4.2	Admin JSON plan editing

Table 4.7: Frontend Technology Stack

4.2.1.3 Project Structure

```
gateway-dashboard/
  src/
    components/
      PlanCard.vue              # Subscription plan card component
      HelloWorld.vue            # Welcome component
    composables/
      useAuth.js                # Authentication composable
      useGlobalState.js         # Global state management
    layouts/
      Main.vue                  # Main application layout
    router/
      index.js                  # Route definitions
      hooks.js                  # Navigation guards (ensureSession,
    ensureSignedIn)
    services/
      auth.service.js           # Authentication API (fetch-based)
      plans.service.js          # Plans API
      stripe.service.js         # Stripe integration API
      user.service.js           # User/subscription API
    views/
      SignIn.vue                # Login page with OAuth
      SignUp.vue                # Registration page
      CompleteSignup.vue        # OAuth password completion
      Subscription.vue          # Subscription management
      Speeches.vue              # Speech processing view
      Files.vue                 # File management
      Applications.vue          # Application management
      Users.vue                 # User management (admin)
      RequestLog.vue            # Request logging (admin)
      Settings.vue              # System settings (admin)
```

```
      Success.vue               # Payment success page
      Cancel.vue                # Payment cancel page
      dashboard/
        Dashboard.vue           # Admin dashboard
    plugins/
      vuetify.js                # Vuetify configuration
    http.js                     # Axios instance with interceptors
    main.js                     # Application entry point
    App.vue                     # Root component
    filters.js                  # Global filters
  vite.config.js                # Vite configuration
  package.json                  # Dependencies
```

Listing 4.75: Frontend Project Structure

4.2.2 Authentication Frontend

The frontend authentication system is built with Vue 3 Composition API, supporting three login methods: email/password, Google OAuth, and Apple Sign-In. The system uses a fetch-based Auth Service for unauthenticated requests and Axios for authenticated API calls.

4.2.2.1 Auth Service Architecture

The `AuthService` is a standalone service module using native fetch API to avoid circular dependencies with Axios interceptors. It manages all authentication operations and localStorage token persistence.

Method	Description
`login(data)`	Email/password authentication via POST /auth/login
`register(data)`	New account registration via POST /auth/register
`refreshToken(data)`	Refresh expired JWT via POST /auth/refresh-token
`verifyToken(data)`	Validate JWT via POST /auth/verify-token
`logout(token)`	Revoke token via POST /auth/logout
`setPassword()`	Complete OAuth signup via POST /auth/add-info
`initiateGoogleLogin()`	Redirect to /auth/google/login
`initiateAppleLogin()`	Trigger Apple SDK sign-in
`initializeAppleSignIn()`	Configure Apple SDK with clientId
`storeAuthData()`	Store tokens in localStorage
`clearAuthData()`	Clear all authentication data
`getAccessToken()`	Retrieve stored access token
`isAuthenticated()`	Check if user has valid token
`handleApiError()`	Standardize error handling

Table 4.8: Auth Service Methods

4.2.2.1.1 Auth Service Implementation

The service uses native fetch with manual error handling to prevent interceptor loops:

```
const API_URL = import.meta.env.VITE_GATEWAY_URL

export const authService = {
  async login(data) {
    try {
      const response = await fetch(`${API_URL}/auth/login`, {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(data)
      })

      const result = await response.json()
      if (!response.ok) {
        const err = new Error('API Error')
        err.response = result
        throw err
      }
      return result
    } catch (error) {
      return this.handleApiError(error)
    }
  },

```

```
  async refreshToken(data) {
    try {
      const response = await fetch(`${API_URL}/auth/refresh-token`, {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(data)
      })

      const result = await response.json()
      if (!response.ok) {
        const err = new Error('API Error')
        err.response = result
        throw err
      }
      return result
    } catch (error) {
      return this.handleApiError(error)
    }
  },

  storeAuthData(accessToken, subscriptionEnd, isVerified) {
    localStorage.setItem('accessToken', accessToken)
    localStorage.setItem('jwt', accessToken) // Backward compatibility

    if (subscriptionEnd !== undefined && subscriptionEnd !== null) {
      localStorage.setItem('subscriptionEnd', subscriptionEnd.toString())
    }

    if (isVerified !== undefined && isVerified !== null) {
      localStorage.setItem('isVerified', isVerified.toString())
    }
  },

  getAccessToken() {
    return localStorage.getItem('accessToken') ||
           localStorage.getItem('jwt')
  },

  clearAuthData() {
    localStorage.removeItem('accessToken')
    localStorage.removeItem('jwt')
    localStorage.removeItem('subscriptionEnd')
    localStorage.removeItem('isVerified')
  },

  isAuthenticated() {
    return !!this.getAccessToken()
  }
}
```

Listing 4.76: Auth Service Core Methods

4.2.2.2 useAuth Composable

The `useAuth` composable replaces Vue 2 mixins, providing reactive authentication state and methods to components using the Composition API.

```
import { computed, ref, onBeforeUnmount } from 'vue'
import { useRouter } from 'vue-router'
import authService from '@/services/auth.service'

export function useAuth() {
  const router = useRouter()
  const authTokenCheckInterval = ref(null)

  // Reactive computed properties
  const isAuthenticated = computed(() => authService.isAuthenticated())
  const accessToken = computed(() => authService.getAccessToken())
  const subscriptionEnd = computed(() => authService.getSubscriptionEnd())
  const isVerified = computed(() => authService.getIsVerified())

  // Logout method
  async function logout() {
    try {
      const token = authService.getAccessToken()
      if (token) {
        await authService.logout(token)
      }
    } catch (error) {
      console.error('Logout error:', error)
    } finally {
      authService.clearAuthData()
      router.push('/sign-in')
    }
  }

  // Token verification and refresh
  async function verifyAndRefreshToken() {
    const token = authService.getAccessToken()
    if (!token) return

    try {
      const result = await authService.verifyToken({ token })

      if (!result.valid) {
        if (result.expired) {
          // Attempt token refresh
          try {
            const refreshResult = await authService.refreshToken({ token })
            authService.storeAuthData(
              refreshResult.accessToken,
              refreshResult.subscriptionEnd,
              refreshResult.isVerified
            )
          } catch (refreshError) {
```

```
            console.error('Token refresh failed:', refreshError)
            authService.clearAuthData()
            router.push('/sign-in')
          }
        } else {
          console.error('Token is invalid:', result.message)
          authService.clearAuthData()
          router.push('/sign-in')
        }
      }
    } catch (error) {
      console.error('Token verification error:', error)
    }
  }

  // Start automatic token verification (every 5 minutes)
  function startTokenVerification() {
    verifyAndRefreshToken()
    authTokenCheckInterval.value = setInterval(() => {
      verifyAndRefreshToken()
    }, 5 * 60 * 1000)
  }

  // Stop automatic verification
  function stopTokenVerification() {
    if (authTokenCheckInterval.value) {
      clearInterval(authTokenCheckInterval.value)
      authTokenCheckInterval.value = null
    }
  }

  // Cleanup on component unmount
  onBeforeUnmount(() => {
    stopTokenVerification()
  })

  return {
    isAuthenticated,
    accessToken,
    subscriptionEnd,
    isVerified,
    logout,
    verifyAndRefreshToken,
    startTokenVerification,
    stopTokenVerification
  }
}
```

Listing 4.77: useAuth Composable Implementation

4.2.2.2.1 Component Usage Example Components can use the composable to access authentication state:

```vue
<script setup>
import { useAuth } from '@/composables/useAuth'

const {
  isAuthenticated,
  accessToken,
  logout,
  startTokenVerification
} = useAuth()

// Start token verification when component mounts
startTokenVerification()
</script>
```

Listing 4.78: Using useAuth in Components

4.2.2.3 Token Storage

Authentication data is stored in localStorage for persistence across page reloads:

```json
{
  "accessToken": "eyJhbGciOiJIUzI1NiIs...",
  "jwt": "eyJhbGciOiJIUzI1NiIs...",      // Backward compatibility
  "subscriptionEnd": "1737244800000",     // Unix timestamp (ms)
  "isVerified": "true"                    // String boolean
}
```

Listing 4.79: LocalStorage Token Schema

4.2.2.4 OAuth Integration

4.2.2.4.1 Google OAuth Google OAuth uses redirect-based authentication:

```javascript
// Initiate Google login (in AuthService)
initiateGoogleLogin() {
  const frontendUrl = window.location.origin
  const url = `${API_URL}/auth/google/login?redirect=${
    encodeURIComponent(frontendUrl)
  }`
  window.location.href = url
}

// After OAuth callback, backend redirects to frontend with:
// <script>
//   localStorage.setItem('accessToken', '<token>')
//   localStorage.setItem('subscriptionEnd', '<timestamp>')
//   localStorage.setItem('isVerified', 'true')
//   window.location.href = '/sign-in'
// </script>
```

Listing 4.80: Google OAuth Flow

4.2.2.4.2 Apple Sign-In

Apple Sign-In uses the Apple SDK with initialization:

```
// Initialize Apple SDK (call on app load)
initializeAppleSignIn(clientId) {
  if (!window.AppleID) {
    console.warn('Apple Sign-In SDK not loaded yet')
    return false
  }

  try {
    const appleClientId = clientId ||
      import.meta.env.VITE_APPLE_CLIENT_ID

    window.AppleID.auth.init({
      clientId: appleClientId,
      scope: 'name email',
      redirectURI: `${API_URL}/auth/apple/callback`,
      usePopup: false
    })
    return true
  } catch (error) {
    console.error('Failed to initialize Apple Sign-In:', error)
    return false
  }
}

// Trigger Apple Sign-In
async initiateAppleLogin() {
  return new Promise((resolve, reject) => {
    if (!window.AppleID) {
      reject(new Error('Apple Sign-In SDK not loaded'))
      return
    }

    try {
      window.AppleID.auth.signIn()
      resolve()
    } catch (error) {
      reject(error)
    }
  })
}
```

Listing 4.81: Apple Sign-In Integration

4.2.2.5 Complete Signup Flow

New OAuth users must complete signup by setting a password:

```
async setPassword(token, password, fullname) {
  try {
    const response = await fetch(`${API_URL}/auth/add-info`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
```

```
      body: JSON.stringify({
        token,
        password,
        fullname
      })
    })

    const result = await response.json()
    if (!response.ok) {
      const err = new Error('API Error')
      err.response = result
      throw err
    }

    // Store new authentication data
    const { accessToken, subscriptionEnd, isVerified } = result
    this.storeAuthData(accessToken, subscriptionEnd, isVerified)

    // Clear temporary token
    localStorage.removeItem('temporaryToken')

    return result
  } catch (error) {
    return this.handleApiError(error)
  }
}
```

Listing 4.82: Complete Signup Implementation

4.2.3 Routing and Navigation

Vue Router 4 manages navigation with global navigation guards that enforce authentication and role-based access control. The routing system uses the Composition API pattern with dependency injection for state management.

4.2.3.1 Route Configuration

Routes are defined with meta fields that specify authentication and authorization requirements:

Path	Auth	Admin	Component
/	Yes	Yes	Dashboard.vue
/sign-in	No	No	SignIn.vue
/sign-up	No	No	SignUp.vue
/complete-signup	No	No	CompleteSignup.vue
/success	No	No	Success.vue
/cancel	No	No	Cancel.vue
/speeches	Yes	No	Speeches.vue
/speeches/:speechId	Yes	No	SpeechDetail.vue
/files	Yes	No	Files.vue
/files/:fileId	Yes	No	FileDetail.vue
/applications	Yes	No	Applications.vue
/applications/:appId	Yes	No	ApplicationDetail.vue
/subscription	Yes	No	Subscription.vue
/users	Yes	Yes	Users.vue
/users/:userId	Yes	Yes	UserDetail.vue
/request-log	Yes	Yes	RequestLog.vue
/request-log/:reqId	Yes	Yes	RequestLogDetail.vue
/settings	Yes	Yes	Settings.vue

Table 4.9: Route Definitions

4.2.3.1.1 Router Instance The router is created with Vue Router 4 syntax using `createRouter` and `createWebHistory`:

```
import { createRouter, createWebHistory } from 'vue-router'
import { ensureSession, ensureSignedIn } from './hooks'

const routes = [
  {
    path: '/',
    name: 'Dashboard',
    component: () => import('@/views/dashboard/Dashboard.vue'),
    meta: {
      requiresAuth: true,
      requiresAdmin: true
    }
  },
  {
    path: '/speeches',
    name: 'Speeches',
    component: () => import('@/views/Speeches.vue'),
    meta: {
      requiresAuth: true
    }
  },
  {
    path: '/subscription',
    name: 'Subscription',
    component: () => import('@/views/Subscription.vue'),
    meta: {
```

```
      requiresAuth: true
    }
  },
  {
    path: '/sign-in',
    name: 'SignIn',
    component: () => import('@/views/SignIn.vue'),
    meta: {
      requiresAuth: false
    }
  },
  // Vue Router 4 catch-all syntax
  {
    path: '/:pathMatch(.*)*',
    name: 'NotFound',
    redirect: '/'
  }
]

const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes
})

// Global navigation guards
router.beforeEach(ensureSession)
router.beforeEach(ensureSignedIn)

export default router
```

Listing 4.83: Router Configuration

4.2.3.2 Navigation Guards Flow

Figure 4.27 illustrates the navigation guard logic.

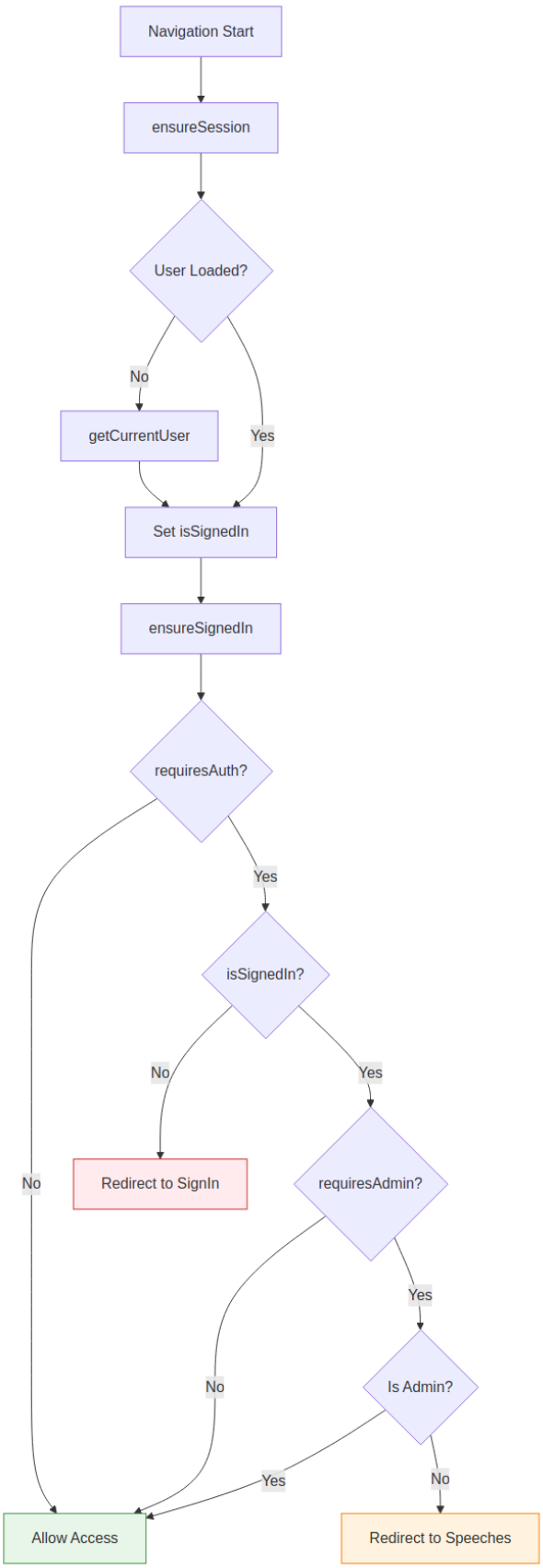


Figure 4.27: Navigation Guards Flow

4.2.3.3 Guard Implementation

The navigation guards use dependency injection to access global state and methods. Dependencies are set during app initialization in `main.js`.

4.2.3.3.1 Dependency Injection Pattern Guards receive dependencies from the main app:

```
// In main.js
import { setRouterDependencies } from './router/hooks'

const globalState = createGlobalState()
const globalMethods = { getCurrentUser, logout, ... }

// Inject dependencies into router hooks
setRouterDependencies(globalState, globalMethods)
```

Listing 4.84: Setting Router Dependencies

4.2.3.3.2 Session Guard The `ensureSession` guard loads user data before navigation:

```
import { nextTick } from 'vue'

let globalState = null
let globalMethods = null

export function setRouterDependencies(state, methods) {
  globalState = state
  globalMethods = methods
}

export const ensureSession = async (to, from, next) => {
  await nextTick()

  if (!globalState || !globalMethods) {
    next()
    return
  }

  if (!globalState.user && to.name !== 'SignIn') {
    // Load user data from JWT
    const user = await globalMethods.getCurrentUser()
    if (user) {
      to.meta.isSignedIn = true
      next()
    } else {
      to.meta.isSignedIn = false
      next()
    }
  } else {
    to.meta.isSignedIn = globalState.user !== null
    next()
```

```
  }
}
```

Listing 4.85: Session Guard Implementation

4.2.3.3.3 Authentication Guard

The `ensureSignedIn` guard enforces authentication and role requirements:

```
function anyTrue(to, prop) {
  return to.matched.findIndex(item => item.meta[prop]) !== -1
}

export const ensureSignedIn = (to, from, next) => {
  if (!globalState) {
    next()
    return
  }

  const requiresAdmin = anyTrue(to, 'requiresAdmin')
  const requiresAuth = requiresAdmin ? true : anyTrue(to, 'requiresAuth')

  if (requiresAuth) {
    const { isSignedIn } = to.meta

    if (isSignedIn) {
      if (requiresAdmin) {
        const isAdmin = globalState.user?.role === 'admin'

        if (isAdmin) {
          next()
        } else {
          // Redirect non-admin users to Speeches page
          next({ name: 'Speeches' })
        }
      } else {
        next()
      }
    } else {
      // Redirect unauthenticated users to sign-in
      next({ name: 'SignIn' })
    }
  } else {
    next()
  }
}
```

Listing 4.86: Auth Guard Implementation

4.2.3.4 HTTP Interceptors

Figure 4.28 shows the HTTP interceptor flow for automatic token handling.

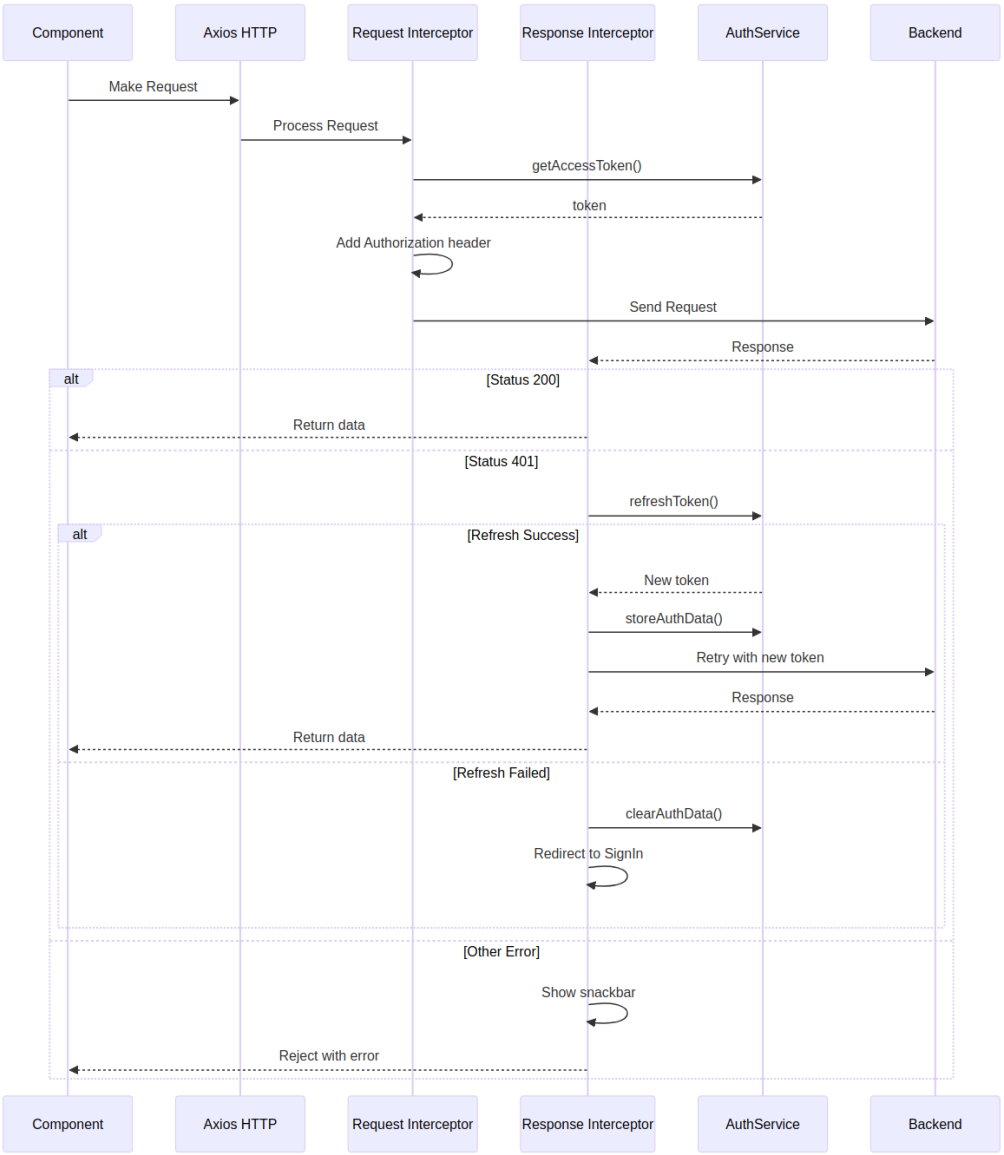


Figure 4.28: HTTP Interceptor Flow with Token Refresh

4.2.3.4.1 Axios Instance Configuration The Axios instance is configured with base URL and default headers:

```
import axios from 'axios'
import router from './router'
import authService from './services/auth.service'

const baseURL = import.meta.env.VITE_GATEWAY_URL

const http = axios.create({
  baseURL,
  headers: {
    Authorization: `Bearer ${authService.getAccessToken()}`,
    'Content-Type': 'application/json'
  }
})

```

```
// Global state reference (set from main.js)
let globalState = null
let globalMethods = null

export function setGlobalState(state, methods) {
  globalState = state
  globalMethods = methods
}
```

Listing 4.87: Axios HTTP Client Setup

4.2.3.4.2 Request Interceptor Automatically attaches fresh JWT token to every request:

```
http.interceptors.request.use((config) => {
  const token = authService.getAccessToken()
  if (token) {
    config.headers.Authorization = `Bearer ${token}`
  }
  return config
}, (error) => {
  return Promise.reject(error)
})
```

Listing 4.88: Request Interceptor

4.2.3.4.3 Response Interceptor Handles 401 errors with automatic token refresh:

```
http.interceptors.response.use((response) => {
  return response
}, async (error) => {
  const originalRequest = error.config

  if (error.response) {
    if (error.response.status === 401 &&
        router.currentRoute.value.name !== 'SignIn') {

      // Try to refresh token (only once per request)
      if (!originalRequest._retry) {
        originalRequest._retry = true

        try {
          const token = authService.getAccessToken()
          if (token) {
            const refreshResult = await authService.refreshToken({ token })

            authService.storeAuthData(
              refreshResult.accessToken,
              refreshResult.subscriptionEnd,
              refreshResult.isVerified
            )

            // Retry original request with new token
```

```
            originalRequest.headers.Authorization =
              `Bearer ${refreshResult.accessToken}`
            return http(originalRequest)
          }
        } catch (refreshError) {
          // Refresh failed, logout user
          if (globalState) {
            globalState.snackbar.message =
              'Session expired. Please login again.'
            globalState.snackbar.isVisible = true
            globalState.snackbar.color = 'error'
          }
          authService.clearAuthData()
          router.push({ name: 'SignIn' })
          return Promise.reject(refreshError)
        }
      }

      // If retry failed
      if (globalState) {
        globalState.snackbar.message =
          error.response.data.message || 'Authentication failed'
        globalState.snackbar.isVisible = true
        globalState.snackbar.color = 'error'
      }
      authService.clearAuthData()
      router.push({ name: 'SignIn' })
      return Promise.reject(error)
    }

    // Handle error messages
    if (Array.isArray(error.response.data.message)) {
      if (globalState) {
        globalState.snackbar.message = error.response.data.message[0]
      }
    } else {
      if (globalState) {
        globalState.snackbar.message = error.response.data.message
      }
    }
  } else {
    if (globalState) {
      globalState.snackbar.message = error.message
    }
  }

  if (globalState) {
    globalState.snackbar.isVisible = true
    globalState.snackbar.color = 'error'
  }

  return Promise.reject(error)
```

```
78 })
79
80 export default http
```

Listing 4.89: Response Interceptor with Token Refresh

The interceptors provide:

- **Automatic Token Injection**: Attaches Bearer token to Authorization header
- **Token Refresh on 401**: Attempts to refresh expired tokens transparently
- **Retry Mechanism**: Retries original request after successful refresh
- **Error Handling**: Displays user-friendly error messages in snackbar
- **Session Management**: Redirects to sign-in when refresh fails

4.2.4 Subscription Management Interface

The subscription interface provides plan selection, payment processing with Stripe Checkout, and real-time quota monitoring using Vue 3 Composition API components.

4.2.4.1 Component Architecture

The subscription system consists of interconnected components built with the Composition API:

- **Subscription.vue**: Main subscription management page with conditional rendering for users and admins
- **PlanCard.vue**: Reusable card component displaying plan details with dynamic Stripe pricing
- **JsonPlansEditor.vue**: Admin-only JSON editor for plan configuration management

4.2.4.2 User Subscription View

For users with active subscriptions, the view displays:

- Current plan name and subscription period (start/end dates)
- Quota allocation for batch and live processing (seconds or unlimited)
- Real-time usage progress bars with color-coded indicators
- Cancel/Resume subscription controls
- Scheduled plan changes with next billing cycle information

4.2.4.2.1 Usage Calculation Logic

Usage percentage is calculated with special handling for unlimited quotas:

```
function calculateUsagePercent(used, quota) {
  if (quota === -1) return 0      // Unlimited quota
  if (quota === 0) return 100     // No quota
  return Math.min((used / quota) * 100, 100)
}

function getUsageColor(used, quota) {
  if (quota === -1) return 'success'  // Unlimited is always green
  const percent = (used / quota) * 100
  if (percent >= 90) return 'error'   // Red for 90%+
  if (percent >= 75) return 'warning' // Orange for 75-89%
  return 'success'                    // Green for <75%
}
```

Listing 4.90: Usage Percentage Calculation

4.2.4.3 PlanCard Component

The PlanCard component is a self-contained Vue 3 Composition API component that fetches Stripe pricing data and displays plan features with interactive UI elements.

Prop/Emit	Type	Description
plan	Object	Plan data (id, name, features, priceId, batchDuration, liveDuration)
disabled	Boolean	Disable subscribe button (for admins viewing plans)
currentPlanId	String	Current user plan ID for badge display
nextPlanId	String	Scheduled next plan ID
hasActiveSubscription	Boolean	User has active subscription
@subscribe	Event	Emitted when user clicks subscribe (payload: {priceId, planId})
@cancel-schedule	Event	Emitted when user cancels scheduled plan change

Table 4.10: PlanCard Component Interface

4.2.4.3.1 Component Props and Emits

4.2.4.3.2 PlanCard Implementation

The component uses the Composition API with reactive state management:

```
import { ref, computed, onMounted, watch } from 'vue'
import { stripeService } from '../services/stripe.service'

```

```
export default {
  name: 'PlanCard',
  props: {
    plan: {
      type: Object,
      required: true,
      validator(value) {
        return value.id && value.name &&
               value.features && value.priceId
      }
    },
    disabled: { type: Boolean, default: false },
    currentPlanId: { type: String, default: null },
    nextPlanId: { type: String, default: null },
    hasActiveSubscription: { type: Boolean, default: false }
  },
  emits: ['subscribe', 'cancel-schedule'],
  setup(props, { emit }) {
    const priceInfo = ref(null)
    const loadingPrice = ref(false)
    const priceError = ref(null)

    // Computed properties
    const formattedPrice = computed(() => {
      if (!priceInfo.value) return 'N/A'

      const amount = priceInfo.value.unit_amount_decimal
        ? (parseInt(priceInfo.value.unit_amount_decimal) / 100).toFixed(2)
        : (priceInfo.value.unit_amount / 100).toFixed(2)

      const currency = priceInfo.value.currency.toUpperCase()
      return `${currency} $${amount}`
    })

    const billingInterval = computed(() => {
      if (!priceInfo.value || !priceInfo.value.recurring) {
        return 'one-time'
      }
      return priceInfo.value.recurring.interval
    })

    const isCurrentPlan = computed(() => {
      return props.currentPlanId &&
             props.plan.id === props.currentPlanId
    })

    const isNextPlan = computed(() => {
      return props.nextPlanId &&
             props.plan.id === props.nextPlanId
    })

    const buttonText = computed(() => {
```

```
    if (isCurrentPlan.value) return 'Current Plan'
    if (isNextPlan.value) return 'Cancel Schedule'
    if (props.hasActiveSubscription) return 'Schedule Change'
    return 'Subscribe Now'
  })

  // Methods
  const fetchPriceInfo = async () => {
    if (!props.plan.priceId) {
      priceError.value = 'No price ID provided'
      return
    }

    loadingPrice.value = true
    priceError.value = null

    try {
      priceInfo.value = await stripeService.getPriceInfo(
        props.plan.priceId
      )
    } catch (error) {
      console.error('Error fetching price info:', error)
      priceError.value = 'Price unavailable'
    } finally {
      loadingPrice.value = false
    }
  }

  const handleSubscribe = () => {
    if (isNextPlan.value) {
      emit('cancel-schedule')
      return
    }
    emit('subscribe', {
      priceId: props.plan.priceId,
      planId: props.plan.id
    })
  }

  const formatDuration = (seconds) => {
    if (seconds === -1) return 'Unlimited'  // Unlimited

    const hours = Math.floor(seconds / 3600)
    const minutes = Math.floor((seconds % 3600) / 60)
    const secs = seconds % 60

    const parts = []
    if (hours > 0) parts.push(`${hours}h`)
    if (minutes > 0) parts.push(`${minutes}m`)
    if (secs > 0 || parts.length === 0) parts.push(`${secs}s`)

    return parts.join(' ')
```

```
    }

    // Lifecycle hooks
    onMounted(() => {
      fetchPriceInfo()
    })

    // Watch for priceId changes
    watch(() => props.plan.priceId, () => {
      fetchPriceInfo()
    })

    return {
      priceInfo,
      loadingPrice,
      priceError,
      formattedPrice,
      billingInterval,
      isCurrentPlan,
      isNextPlan,
      buttonText,
      handleSubscribe,
      formatDuration
    }
  }
}
```

Listing 4.91: PlanCard Component Setup

4.2.4.3.3 PlanCard Template The template uses Vuetify 3 components with conditional badges and dynamic styling:

```
<template>
  <v-card
    class="plan-card elevation-8"
    :class="{ 'recommended-card': plan.recommended }"
  >
    <!-- Recommended Badge -->
    <v-chip
      v-if="plan.recommended && !isCurrentPlan && !isNextPlan"
      class="recommended-badge"
      color="primary"
      size="small"
      label
    >
      RECOMMENDED
    </v-chip>

    <!-- Current Plan Badge -->
    <v-chip
      v-if="isCurrentPlan"
      class="current-plan-badge"
      color="success"
```

```
      size="small"
      label
    >
      <v-icon start size="small">mdi-check-circle</v-icon>
      CURRENT PLAN
    </v-chip>

    <!-- Next Subscription Badge -->
    <v-chip
      v-if="isNextPlan && !isCurrentPlan"
      class="next-plan-badge"
      color="info"
      size="small"
      label
    >
      <v-icon start size="small">mdi-calendar-clock</v-icon>
      NEXT SUBSCRIPTION
    </v-chip>

    <v-card-text class="pa-6">
      <!-- Plan Name -->
      <h2 class="text-h4 font-weight-semibold mb-4">
        {{ plan.name }}
      </h2>

      <!-- Price with Loading State -->
      <div class="mb-6">
        <div v-if="loadingPrice" class="text-center">
          <v-progress-circular
            indeterminate
            color="primary"
            size="24"
          ></v-progress-circular>
          <span class="text-body-2 text-medium-emphasis ml-2">
            Loading price...
          </span>
        </div>
        <div v-else-if="priceError" class="text-caption text-error">
          {{ priceError }}
        </div>
        <div v-else-if="priceInfo">
          <span class="text-h3 font-weight-black">
            {{ formattedPrice }}
          </span>
          <span class="text-body-1 text-medium-emphasis">
            /{{ billingInterval }}
          </span>
        </div>
        <div v-else class="text-caption text-medium-emphasis">
          Price not available
        </div>
      </div>
```

```
74
75        <!-- Duration Info -->
76        <div v-if="plan.batchDuration !== undefined ||
77                   plan.liveDuration !== undefined"
78             class="mb-4">
79          <v-divider class="mb-4"></v-divider>
80          <div v-if="plan.batchDuration !== undefined"
81               class="d-flex align-center mb-2">
82            <v-icon size="small" color="primary" class="mr-2">
83              mdi-clock-outline
84            </v-icon>
85            <span class="text-body-2">
86              <strong>Batch Duration:</strong>
87              {{ formatDuration(plan.batchDuration) }}
88            </span>
89          </div>
90          <div v-if="plan.liveDuration !== undefined"
91               class="d-flex align-center">
92            <v-icon size="small" color="primary" class="mr-2">
93              mdi-clock-fast
94            </v-icon>
95            <span class="text-body-2">
96              <strong>Live Duration:</strong>
97              {{ formatDuration(plan.liveDuration) }}
98            </span>
99          </div>
100       </div>
101
102       <!-- Features List -->
103       <v-list class="mt-6 bg-transparent">
104         <v-list-item
105           v-for="(feature, index) in plan.features"
106           :key="index"
107           class="px-0"
108         >
109           <template v-slot:prepend>
110             <v-icon color="success">mdi-check</v-icon>
111           </template>
112           <v-list-item-title class="text-body-1">
113             {{ feature }}
114           </v-list-item-title>
115         </v-list-item>
116       </v-list>
117
118       <!-- Subscribe Button -->
119       <v-btn
120         color="primary"
121         size="x-large"
122         block
123         class="mt-6 elevation-4"
124         :disabled="disabled || isCurrentPlan"
125         @click="handleSubscribe"
```

```
      >
        {{ buttonText }}
      </v-btn>
    </v-card-text>
  </v-card>
</template>
```

Listing 4.92: PlanCard Template

4.2.4.4 Stripe Checkout Integration

When a user subscribes, the frontend initiates Stripe Checkout:

```
async handleSubscribe(data) {
  try {
    const { priceId, planId } = data
    const result = await stripeService.createCheckoutSession(priceId)

    if (result?.url) {
      // Redirect to Stripe Checkout
      window.location.href = result.url
    } else {
      this.showError('Failed to create checkout session. Please try again.')
    }
  } catch (err) {
    console.error('Subscription error:', err)
    this.showError('An error occurred during checkout.')
  }
}
```

Listing 4.93: Checkout Session Creation

4.2.4.5 Cancel and Resume Subscription

Users can manage their subscription lifecycle with confirmation dialogs:

```
async handleCancelSubscription() {
  const confirmed = confirm(
    'Are you sure you want to cancel your subscription? ' +
    'It will remain active until the end of the current billing period.'
  )

  if (!confirmed) return

  try {
    this.cancellingSubscription = true
    await stripeService.cancelSubscription(
      this.subscription.stripeSubscriptionId
    )

    // Refresh subscription status
    await this.checkStripeSubscriptionStatus(
      this.subscription.stripeSubscriptionId
```

```
    )

    this.showSuccess(
      'Subscription scheduled for cancellation at period end.'
    )
  } catch (err) {
    console.error('Cancel error:', err)
    this.showError('Failed to cancel subscription.')
  } finally {
    this.cancellingSubscription = false
  }
}

async handleResumeSubscription() {
  try {
    this.resumingSubscription = true
    await stripeService.resumeSubscription(
      this.subscription.stripeSubscriptionId
    )

    await this.checkStripeSubscriptionStatus(
      this.subscription.stripeSubscriptionId
    )

    this.showSuccess('Subscription resumption successful.')
  } catch (err) {
    console.error('Resume error:', err)
    this.showError('Failed to resume subscription.')
  } finally {
    this.resumingSubscription = false
  }
}
```

Listing 4.94: Cancel Subscription with Confirmation

4.2.4.6 Admin Plans Editor

Administrators access a JSON editor for managing plan configurations:

- Live JSON validation with syntax highlighting via JSONEditor library
- Stripe price ID validation before publishing to prevent invalid configurations
- Diff comparison with current version to review changes
- Version history tracking with rollback capability
- Auto-formatting and error highlighting

The editor validates each plan's `priceId` against Stripe's API before allowing publication, ensuring only valid pricing configurations are saved to the database.

Chapter 5

Conclusion and Future Work

This chapter summarizes the project achievements, draws conclusions from the implementation, and outlines directions for future work.

5.1 Project Summary

This project successfully designed and implemented a comprehensive **Gateway Dashboard** system to address the challenges faced by the Speech Gateway API ecosystem. The project delivered the following key components:

5.1.1 Authentication System

A robust authentication module was implemented supporting three authentication methods:

- **Email/Password Authentication**: Traditional credential-based login with password hashing using bcrypt.
- **Google OAuth 2.0**: Redirect-based OAuth flow for seamless Google account integration.
- **Apple Sign-In**: SDK-based authentication supporting Apple's privacy-focused identity system.

The authentication system includes advanced security features such as JWT token management, automatic token refresh, token blacklisting using Redis, and session security with device fingerprinting.

5.1.2 Payment and Subscription System

A complete payment processing infrastructure was developed using Stripe, consisting of:

- **Plan Module**: Versioned pricing plans with Redis caching and cache penetration protection.
- **Stripe Integration**: Checkout session creation, webhook handling, and subscription lifecycle management.
- **Subscription Module**: Quota tracking for batch and live processing with automatic access control.

5.1.3 Frontend Dashboard

A Vue.js 2 web application was developed with Vuetify providing:

- Intuitive user interface for subscription and service management.
- Admin portal with JSON-based plans editor and validation.
- Secure navigation guards and HTTP interceptors for token management.
- Responsive design supporting multiple device form factors.

5.2 Conclusions

The Gateway Dashboard project has successfully achieved all stated objectives:

1. **Web-based Dashboard**: A fully functional Vue.js application provides an intuitive interface for users and administrators, eliminating the need for direct API interaction.
2. **Secure Authentication**: The multi-provider SSO system enables seamless user onboarding while maintaining security through industry-standard protocols (OAuth 2.0, JWT, HTTPS).
3. **Subscription Infrastructure**: The Stripe integration enables automated billing, subscription management, and tiered access control, establishing the commercial viability of the Speech Gateway services.
4. **Administrative Tools**: The admin portal with RBAC provides efficient management of users, plans, and system monitoring without manual database intervention.
5. **Enhanced Security**: Token blacklisting, automatic refresh, and session management using Redis provide robust protection against unauthorized access.

The modular architecture adopted in this project promotes maintainability and extensibility. The separation between frontend and backend, combined with the service-oriented design in NestJS, allows for independent scaling and future enhancements.

5.3 Future Work

While the current implementation addresses the core requirements, several areas present opportunities for future development:

5.3.1 Technical Enhancements

- **Multi-language Support**: Implement internationalization (i18n) in the frontend to support multiple languages.
- **Real-time Usage Monitoring**: Add WebSocket-based real-time updates for quota usage and system status.
- **Enhanced Analytics**: Implement comprehensive usage analytics dashboard for administrators.

- **Mobile Application**: Develop native mobile applications for iOS and Android to complement the web dashboard.

5.3.2 Business Features

- **Team/Organization Accounts**: Support for enterprise customers with multiple users under a shared subscription.
- **Usage-based Billing**: Implement pay-as-you-go pricing model in addition to subscription plans.
- **Invoice Management**: Generate downloadable PDF invoices for accounting purposes.
- **Promotional Codes**: Support for discount codes and promotional campaigns.

5.3.3 Infrastructure Improvements

- **Containerization**: Docker containerization for easier deployment and scaling.
- **CI/CD Pipeline**: Automated testing and deployment pipelines for continuous integration.
- **Monitoring and Alerting**: Integration with tools like Prometheus and Grafana for production monitoring.

5.4 Final Remarks

The Gateway Dashboard project demonstrates how modern web technologies can be leveraged to create a comprehensive management system for API-based services. The combination of Vue.js, NestJS, MongoDB, and Redis provides a solid foundation for building scalable, maintainable, and secure web applications.

The project not only addresses the immediate needs of the Speech Gateway ecosystem but also establishes patterns and practices that can be applied to similar enterprise applications. The modular design ensures that the system can evolve to meet future requirements while maintaining backward compatibility.

Bibliography

[1] Apple Inc. *Apple Platform Security*. https://support.apple.com/guide/security. 2023.

[2] Apple Inc. *Sign in with Apple REST API*. https://developer.apple.com/documentation/sign_in_with_apple. 2023.

[3] Apple Inc. *Sign in with Apple: Technical Overview*. https://developer.apple.com/sign-in-with-apple. 2019.

[4] Kyle Banker. *MongoDB in action*. Manning Publications, 2011.

[5] Adam Barth, Collin Jackson, and John C Mitchell. "Securing frame communication in browsers". In: *17th USENIX Security Symposium*. 2008, pp. 17–30.

[6] Baymard Institute. *E-Commerce Checkout Usability: An Original Research Study*. https://baymard.com/checkout-usability. 2020.

[7] Gavin Bierman, Martín Abadi, and Mads Torgersen. "Understanding TypeScript". In: *European Conference on Object-Oriented Programming*. 2014, pp. 257–281.

[8] Josiah L Carlson. *Redis in Action*. Manning Publications, 2013.

[9] Kristina Chodorow. *MongoDB: the definitive guide*. O'Reilly Media, 2013.

[10] Chee Yong Chong and Sai Peck Lee. "Component-based software engineering: Technologies, development frameworks, and quality assurance schemes". In: *Information and Software Technology* 107 (2019), pp. 70–86.

[11] Roy Thomas Fielding. "Architectural styles and the design of network-based software architectures". PhD thesis. University of California, Irvine, 2000.

[12] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley, 2003.

[13] Zheng Gao, Christian Bird, and Earl T Barr. "Type systems for JavaScript: A survey". In: *Software Engineering at Google* (2017), pp. 1–27.

[14] Gartner Inc. *Market Guide for Cloud Workspace Productivity Suites*. 2023.

[15] Annabelle Gawer and Michael A Cusumano. *Platform leadership: How Intel, Microsoft, and Cisco drive industry innovation*. Harvard Business School Press, 2014.

[16] GitHub Inc. *GitHub Annual Report 2023*. https://github.com/about. 2023.

[17] Google Inc. *Google Account Security Features*. https://safety.google/authentication. 2023.

[18] Google Inc. *Google Services: Active User Statistics 2023*. https://about.google. 2023.

[19] Google Inc. *Material Design Guidelines*. https://material.io/design. 2014.

[20] Google Inc. *Using OAuth 2.0 to Access Google APIs*. https://developers.google.com/identity/protocols/oauth2. 2023.

[21] Wenjuan Guo et al. “Freemium as an optimal strategy for market dominant firms”. In: *Marketing Science* 38.1 (2019), pp. 150–169.

[22] Rahul Gupta. *Stripe integration: A comprehensive guide for developers*. 2020.

[23] Dick Hardt. *The OAuth 2.0 authorization framework*. Tech. rep. RFC 6749, 2012.

[24] Amir Herzberg and Ahmad Jbara. “Payment system integration: A comparative study of Stripe and PayPal”. In: *Journal of Electronic Commerce Research* 20.3 (2019), pp. 145–162.

[25] Jim Isaak and Mina J Hanna. “User data privacy: Facebook, Cambridge Analytica, and privacy protection”. In: *Computer* 51.8 (2018), pp. 56–59.

[26] Michael Jones, John Bradley, and Nat Sakimura. *JSON web token (JWT)*. Tech. rep. RFC 7519, 2015.

[27] Neal Leavitt. “Will NoSQL databases live up to their promise?” In: *Computer* 43.2 (2010), pp. 12–14.

[28] Cheng Li et al. “Building reliable distributed systems using message queues”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. 2015, pp. 239–254.

[29] Torsten Lodderstedt, Mark McGloin, and Phil Hunt. “OAuth 2.0 threat model and security considerations”. In: *RFC 6819* (2013).

[30] Callum Macrae. *Vue.js: Up and Running*. O’Reilly Media, 2018.

[31] Mark Masse. *REST API design rulebook*. O’Reilly Media, 2011.

[32] Microsoft Corporation. *Microsoft Identity Platform Documentation*. `https://docs.microsoft.com/azure/active-directory`. 2023.

[33] Michael S Mikowski and Josh C Powell. *Single Page Web Applications: JavaScript end-to-end*. Manning Publications, 2013.

[34] Sam Newman. *Building microservices: designing fine-grained systems*. O’Reilly Media, 2015.

[35] Addy Osmani. *A comparison of JavaScript frameworks*. 2017.

[36] Addy Osmani. *Web performance optimization*. 2017.

[37] OWASP Foundation. *OWASP Top 10 - 2021: The Ten Most Critical Web Application Security Risks*. Tech. rep. OWASP, 2021.

[38] Suhas Pai et al. “SoK: Systematization of knowledge on web single sign-on systems”. In: *2011 IEEE Symposium on Security and Privacy*. IEEE. 2011, pp. 232–246.

[39] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. “RESTful web services vs. ”big” web services: making the right architectural decision”. In: *Proceedings of the 17th international conference on World Wide Web* (2008), pp. 805–814.

[40] PayPal Holdings Inc. *PayPal Q4 2023 Earnings Report*. `https://investor.paypal-corp.com`. 2023.

[41] Ken Peffers, Charles E Gengler, and Tuure Tuunanen. “Electronic payment systems: an analysis and comparison of types”. In: *IEEE Transactions on Engineering Management* 50.3 (2003), pp. 296–304.

[42] Tomas Petricek, Gustavo Guerra, and Don Syme. “Types from data: Making structured data first-class citizens in F#”. In: *ACM SIGPLAN Notices* 51.6 (2016), pp. 477–490.

[43] Chris Richardson. *Microservices patterns: with examples in Java*. Manning Publications, 2018.

[44] Carlos Rodriguez et al. "RESTful web services: A study and comparison of tools". In: *Service Oriented Computing and Applications* 10 (2016), pp. 435–450.

[45] Nat Sakimura, John Bradley, and Naveen Agarwal. *Proof Key for Code Exchange by OAuth Public Clients*. Tech. rep. RFC 7636, 2015.

[46] Günther Schuh, Michael Riesener, and Christof Mattern. "Subscription business models: Growth and profitability". In: *Production Engineering* 7 (2013), pp. 385–392.

[47] Prabath Siriwardena. *Advanced API security: OAuth 2.0 and beyond*. Apress, 2020.

[48] S Mahbub Sohan. "A pattern language for webhooks". In: *PLoP'17: 24th Conference on Pattern Languages of Programs* (2017).

[49] StatCounter Global Stats. *Mobile Operating System Market Share Worldwide*. `https://gs.statcounter.com`. 2023.

[50] Statista Research Department. *Most popular online accounts worldwide 2023*. `https://www.statista.com`. 2023.

[51] Stripe Inc. *Stripe API Documentation*. `https://stripe.com/docs/api`. 2023.

[52] Stripe Inc. *Stripe Billing Documentation*. `https://stripe.com/docs/billing`. 2023.

[53] Stripe Inc. *Stripe Security and Compliance*. `https://stripe.com/docs/security`. 2023.

[54] Yi Sun et al. "Understanding user adoption of single sign-on systems". In: *Computers & Security* 83 (2019), pp. 143–154.

[55] Yuhui Sun, Yan Zhang, and Wei Peng. "Security and privacy for web databases and services". In: *Web Database Systems: Technologies and Applications* (2010), pp. 15–37.

Appendix A

This section will be completed in the final report, once everything has been finalized.