NANYANG TECHNOLOGICAL UNIVERSITY
COLLEGE OF COMPUTING AND DATA SCIENCE

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. The system addresses the need for a robust, user-friendly platform to manage access and subscriptions for speech processing services.

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 System** 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 mentors **Kyaw Zin Tun**, **Vu Thi Ly**, **Surana Tanmay** and my supervisor, **Chng Eng Siong**, for their invaluable guidance, continuous encouragement, and expert advice throughout the course of this Final Year Project. Their insights were instrumental in shaping the direction and reliability of this system.

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, and Frontend Dashboard Development. Each component involves both frontend and backend implementation as detailed below.

1.4.1 Single Sign-On (SSO) Authentication

This component enables users to authenticate using their existing Google or Apple accounts, eliminating the need for separate credentials.

Backend Implementation (NestJS)

- **Google SSO Module:** OAuth 2.0 integration with Google Sign-In using Passport.js strategies, handling authorization code flow with state management for CSRF protection.
- **Apple SSO Module:** Integration with Apple Sign In, including token verification and privacy-preserving email relay support.
- **JWT Token Management:** Generation and validation of access tokens (15-minute expiry) and refresh tokens with secure cookie storage (HttpOnly, Secure, SameSite attributes).
- **Token Blacklisting:** Redis-based blacklist for immediate token revocation on user logout or security events.
- **Traditional Auth:** Email/password authentication with bcrypt hashing as fallback option.

Frontend Implementation (Vue.js)

- **Authentication Pages:** Sign In, Sign Up, and OAuth callback handling pages.
- **SSO Integration:** "Sign in with Google" and "Sign in with Apple" buttons with proper OAuth flow initiation.
- **Token Management:** Automatic token refresh using interceptors, handling token expiration gracefully.
- **Navigation Guards:** Route protection based on authentication status, redirecting unauthenticated users to login.

1.4.2 Payment & Subscription Management

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

Backend Implementation (NestJS)

- **Stripe Integration Module:** Checkout session creation, webhook handling for payment events (`checkout.session.completed`, `customer.subscription.updated/deleted`) webhook signature verification.
- **Plans Module:** JSON-based plan configuration stored in MongoDB with versioning support, Redis caching for fast retrieval, admin API for plan updates with cache invalidation.
- **Subscription Module:** User subscription tracking synchronized with Stripe, usage quota management (batch and live speech processing durations), access control middleware enforcing plan limits.
- **Webhook Processing:** Idempotent event handling, automatic subscription creation/updates from Stripe events, cancellation and resumption logic.

Frontend Implementation (Vue.js)

- **Subscription Page:** Display available plans with pricing fetched from Stripe API, plan comparison cards showing features and quotas, checkout button triggering Stripe Checkout session.
- **User Dashboard:** Current subscription status and active plan display, usage quota tracking with visual progress indicators, cancel/resume subscription functionality.
- **Admin Plans Management:** JSON editor for creating and updating service plans with real-time validation, version history tracking, preview of plan changes before publishing.
- **Payment Success/Cancel Pages:** Post-checkout handling and user feedback.

1.4.3 Frontend Dashboard Development

This component provides the complete user interface built as a Single Page Application (SPA) using **Vue.js 2** and **Vuetify**.

Core Frontend Architecture

- **Component Structure:** Reusable components (PlanCard, SubscriptionStatus, UserProfile), page components (Home, Subscription, Dashboard, Admin), layout components (AppBar, NavigationDrawer, Footer).
- **Routing System:** Vue Router with nested routes, role-based route guards (user vs admin), lazy loading for code splitting and performance.
- **State Management:** Vuex store modules for auth, user, subscription, and plans state, centralized API error handling.

- **HTTP Client:** Axios configuration with request/response interceptors, automatic JWT token injection, retry logic for failed requests.
- **UI/UX Implementation:** Vuetify Material Design components (data tables, forms, dialogs, cards), responsive layout for mobile and desktop, loading states and error feedback.

User-Facing Features

- Authentication flow pages (Sign In, Sign Up, OAuth callbacks).
- Subscription management interface with plan selection and upgrade/downgrade.
- Usage statistics dashboard showing quota consumption.
- Profile management and account settings.

Admin-Facing Features

- Plans management interface with JSON editor and validation.
- System monitoring and user subscription overview.
- Configuration management tools.

Chapter 2

Literature Review

This chapter examines the existing research and technologies relevant to developing a comprehensive API management dashboard with authentication, subscription management, and payment integration. The review covers web application architectures, authentication mechanisms, payment systems, and modern web development frameworks.

2.1 Web Application Architecture Patterns

2.1.1 Single Page Applications (SPAs)

Single Page Applications load once and dynamically update content, providing fluid user experiences without page reloads [33]. The Gateway Dashboard implements this pattern using Vue.js, separating frontend and backend concerns through a RESTful API architecture [39]. This separation enables independent development cycles and better scalability.

2.1.2 API Gateway Pattern

The API Gateway pattern provides a single entry point for clients, centralizing authentication, rate limiting, and request routing [43]. The Speech Gateway API implements this pattern, acting as an intermediary between client applications and speech processing services. This simplifies client implementations while centralizing security enforcement [34].

2.2 Authentication and Authorization

2.2.1 JWT Authentication

The Gateway Dashboard implements token-based authentication using JSON Web Tokens (JWT) [26]. JWTs are self-contained tokens that encode user claims and can be verified cryptographically without database lookups, enabling stateless authentication that scales in distributed systems. The implementation addresses security considerations through token expiration, refresh token rotation, and blacklisting mechanisms for revocation [47].

2.2.2 Single Sign-On and OAuth 2.0

The Gateway Dashboard implements OAuth 2.0 [23] for delegated authentication, allowing users to sign in without creating new credentials. The implementation uses OAuth

2.0's authorization code flow with proper state management to prevent CSRF attacks [38] and PKCE extension [45] for enhanced security in the single-page application.

SSO Provider Comparison

Major SSO providers offer different advantages:

- **Google Sign-In:** 2+ billion active accounts, mature OAuth 2.0 implementation, excellent documentation [18]
- **Apple Sign In:** Privacy-focused with email hiding, mandatory for iOS apps with third-party SSO [3]
- **Facebook Login:** Large user base (2.9B) but declining developer trust due to privacy scandals [25]
- **Microsoft/GitHub:** Strong in enterprise and developer contexts respectively [32, 16]

Implementation: Why Google and Apple

The Gateway Dashboard implements Google Sign-In and Apple Sign In for the following reasons:

1. **Market Coverage:** Combined coverage of 95%+ internet users [50], with Google dominating Android (71% market share) and Apple mandatory for iOS (28%) [49]
2. **Technical Maturity:** Both offer well-documented OAuth 2.0 implementations with 99.9%+ uptime and official SDKs for JavaScript/TypeScript [20, 2]
3. **Privacy and Security:** Apple's email hiding and Google's granular permissions address user privacy concerns, while both support 2FA and fraud detection [3, 17]

Other providers were not implemented: Facebook (privacy concerns [25]), Microsoft (implemented by another developer in the project), GitHub (developers already have Google/Apple accounts), Twitter/LinkedIn (limited value for API management).

2.2.3 Token Management and Security

The Gateway Dashboard implements token management best practices [29]: short-lived access tokens (15 minutes), refresh token rotation, and Redis-based token blacklisting for immediate revocation on logout. Secure cookie attributes (HttpOnly, Secure, SameSite) and proper CORS configuration protect against common browser-based attacks [5].

2.3 Payment Processing and Subscription Management

2.3.1 Payment Platform Comparison: Stripe vs PayPal

Modern payment processing requires balancing security, developer experience, and subscription management capabilities [41]. The two dominant platforms are:

PayPal (400M users [40]): Strong brand recognition and buyer protection, but has complex multi-generation APIs (NVP, SOAP, REST), redirect-based flows that reduce conversions by 10-15% [6], and limited subscription flexibility [24].

Stripe: Developer-focused with superior API documentation [51], embedded checkout (no redirects) [22], advanced subscription management (prorated upgrades, metered billing) [52], and reliable webhook architecture [48]. Both charge 2.9% + $0.30 per transaction.

Implementation: Why Stripe

The Gateway Dashboard implements Stripe for the following reasons:

1. **Subscription Requirements:** The project requires plan-based access control, usage quotas, prorated upgrades/downgrades, and automated renewals. Stripe's Billing API provides native support; PayPal requires extensive custom development [52]

2. **Integration Speed:** Stripe's RESTful API and documentation enable 40% faster integration than PayPal [24], with official NestJS SDK

3. **User Experience:** Stripe Checkout embeds in the application (no redirects), improving conversion vs PayPal's redirect flows [6]

4. **Webhook Reliability:** Automatic retries, signature verification, and idempotency keys maintain payment-database consistency [48]

5. **Future Scalability:** Metered billing support for potential usage-based pricing (charging per speech processing duration)

For the technical audience (developers integrating speech services), Stripe's functionality and integration quality outweigh PayPal's consumer brand recognition.

2.3.2 Subscription Management Implementation

The Gateway Dashboard implements subscription lifecycle management [46] with: flexible JSON-based plan configuration, usage quota tracking (batch/live speech durations), automated renewal handling via Stripe webhooks, and self-service cancel/resume functionality. The system processes Stripe webhook events (`checkout.session.completed`, `customer.subscription.updated/deleted`) [48] with signature verification and idempotency to maintain payment-database synchronization [28].

2.4 Caching Implementation

The Gateway Dashboard implements Redis caching [8] for plan configurations, reducing database load by caching read-heavy data with sub-millisecond access times. The system uses cache-aside pattern [12]: check cache first, load from MongoDB on miss, then populate cache. Plan cache is warmed on application startup and invalidated when administrators update plans, optimizing the 99% read vs 1% write access pattern.

2.5 Summary

This literature review establishes the foundation for the Gateway Dashboard implementation, covering:

- **Architecture:** SPA pattern with Vue.js frontend and NestJS API Gateway backend
- **Authentication:** JWT tokens, OAuth 2.0 with Google/Apple SSO (95%+ user coverage), token blacklisting
- **Payment:** Stripe over PayPal for subscription management, embedded checkout, and webhook reliability
- **Database:** MongoDB with query-driven schema design and Redis caching for plans
- **Development:** TypeScript full-stack for type safety and shared definitions

The implementation choices prioritize developer experience, user convenience, and scalability for the speech processing 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.

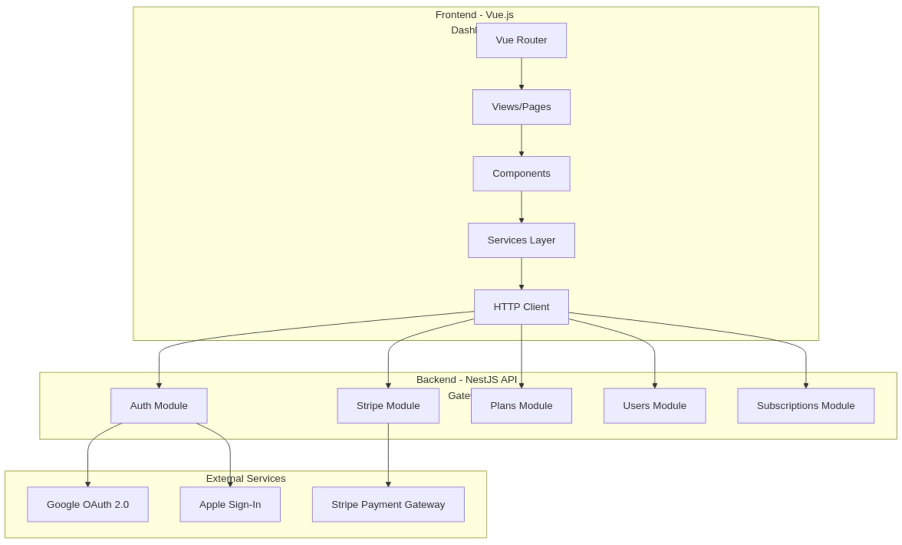


Figure 3.1: High-Level System Architecture

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 2 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	2.x	Progressive JavaScript framework for building user interfaces
Vuetify	2.x	Material Design component framework for Vue.js
Vue Router	3.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 2 was selected for its gentle learning curve, excellent documentation, and reactive data binding capabilities. The framework's component-based architecture facilitates code reuse and maintainability. **Vuetify** 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	9.x	Progressive Node.js framework for server-side applications
TypeScript	4.x	Typed superset of JavaScript for enhanced code quality
Passport	0.6.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	5.x	NoSQL document database for flexible schema design
Redis	6.x	In-memory data store for caching and session management

Table 3.3: Storage Technologies

MongoDB was chosen for its schema flexibility, horizontal scalability, and native JSON support. **Redis** serves dual purposes: caching frequently accessed data (plans, prices) and managing token blacklists for security.

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.

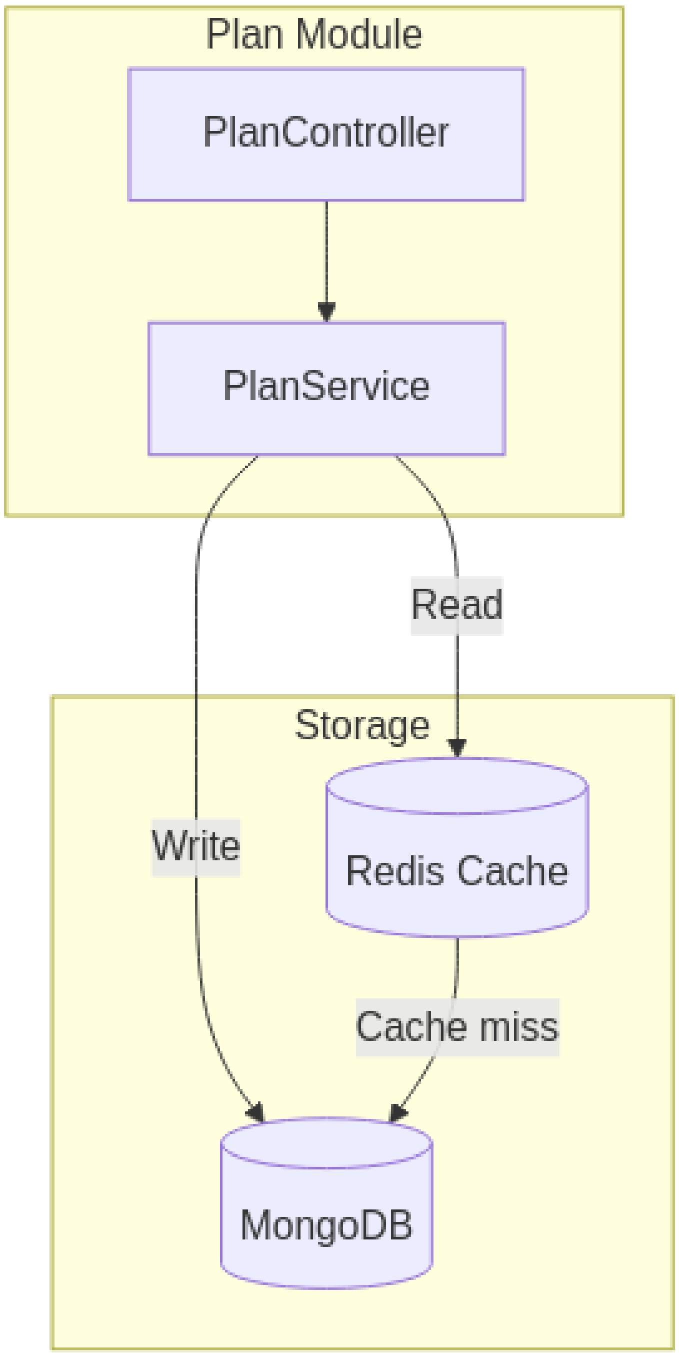


Figure 3.2: Plan Module Caching Architecture

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.

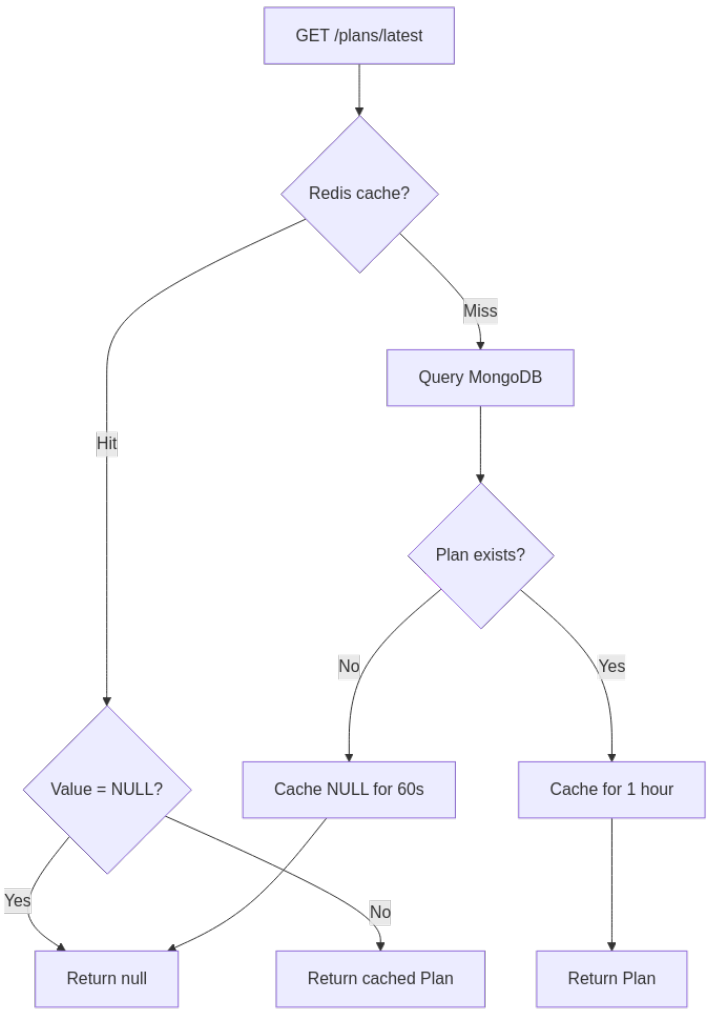


Figure 3.3: Cache Flow for Latest Plans Retrieval

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:

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

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
status	String	Subscription status
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.

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:

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.

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`.

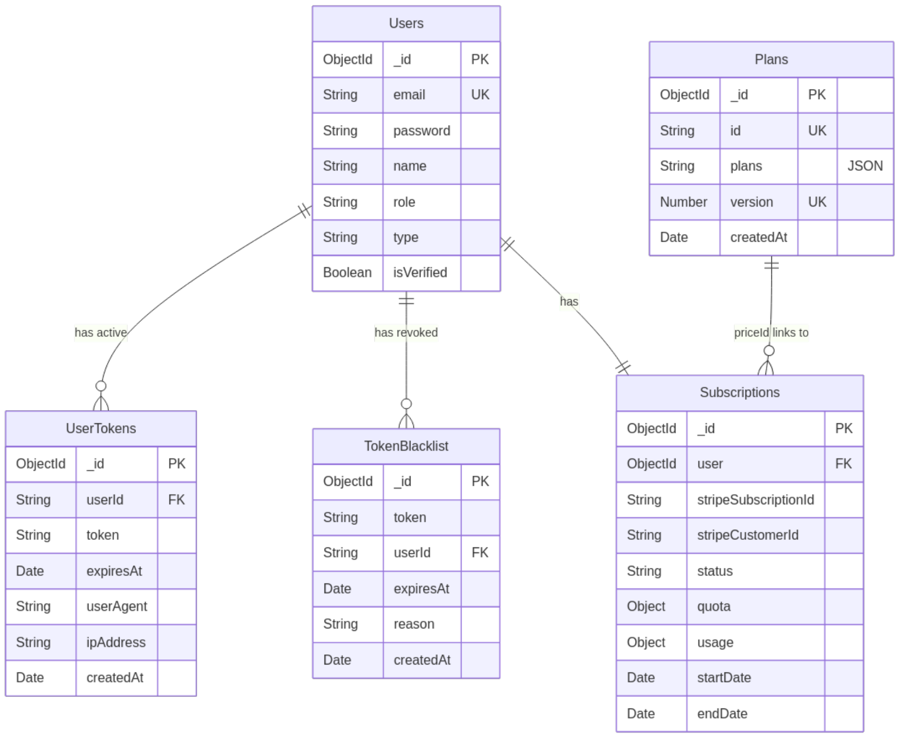


Figure 3.4: Entity Relationship Diagram

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

Authentication and Security

This chapter provides a comprehensive examination of the authentication system implemented in the Gateway Dashboard. It covers the supported authentication methods, detailed flow diagrams, token lifecycle management, and session security mechanisms.

4.1 Overview

The Gateway Dashboard implements a comprehensive authentication system supporting multiple identity providers with robust session management. This section provides an overview of the authentication architecture.

4.1.1 Supported Authentication Methods

The system supports three authentication methods:

Method	**Provider**	**Description**
Email/Password	Internal	Traditional credential-based login with bcrypt password hashing
Google OAuth 2.0	Google	Social login using Passport's Google strategy
Apple Sign-In	Apple	Authentication via identity token verification

Table 4.1: Supported Authentication Methods

4.1.2 Module Architecture

Figure 4.1 illustrates the overall architecture of the authentication modules.

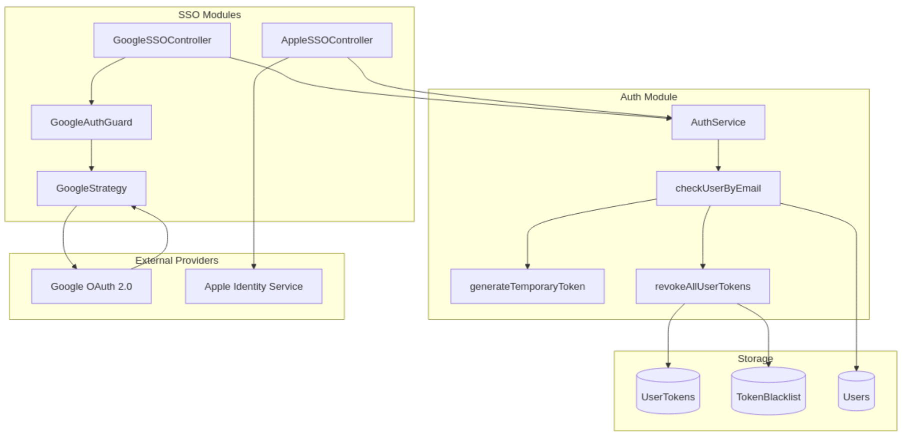


Figure 4.1: Authentication Module Architecture

The architecture consists of three main components:

- **SSO Modules**: GoogleSSOController and AppleSSOController handle OAuth flows with their respective providers.
- **Auth Module**: Central AuthService provides shared functionality including:
  - `checkUserByEmail()`
  - `generateTemporaryToken()`
  - `revokeAllUserTokens()`
- **Storage Layer**: UserTokens collection stores active sessions; TokenBlacklist stores revoked tokens; Users collection stores account data.

Both SSO controllers delegate user verification to the shared AuthService, ensuring consistent handling of new users (requiring password setup) and existing users (issuing access tokens).

4.1.3 Chapter Organization

This chapter is organized as follows:

- **Section 3.2**: Detailed authentication flows for each method
- **Section 3.3**: JWT token lifecycle and API endpoints
- **Section 3.4**: Session security mechanisms and database schemas

4.2 Authentication Flows

This section describes the detailed authentication flows for each supported method.

4.2.1 Email/Password Authentication

The traditional login flow uses Passport's LocalStrategy for credential validation. Figure 4.2 shows the sequence.

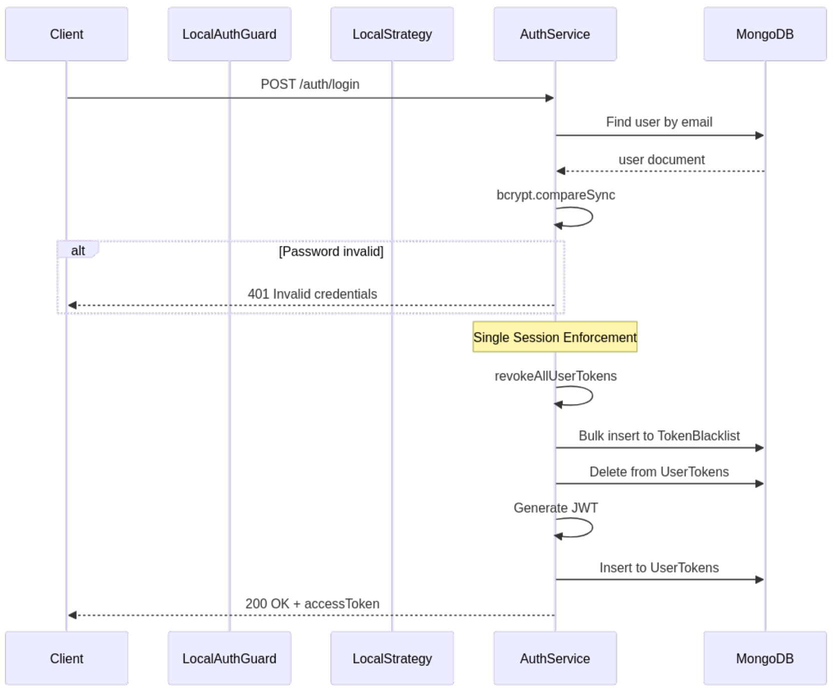


Figure 4.2: Email/Password Login Flow

The login process:

1. Client sends credentials to `POST /auth/login`.
2. LocalAuthGuard triggers LocalStrategy validation.
3. AuthService queries MongoDB and compares password hashes using bcrypt.
4. If valid, existing tokens are revoked (single session enforcement).
5. New JWT is generated and stored in UserTokens collection.
6. Response includes access token, subscription end date, and verification status.

4.2.2 Google OAuth 2.0

Google SSO uses Passport's OAuth 2.0 strategy with redirect-based authentication. This subsection covers the module architecture, authentication flow, and implementation details.

Google SSO Module Structure

Figure 4.3 shows the architecture of the Google SSO module, including its integration with Passport and the shared Auth module.

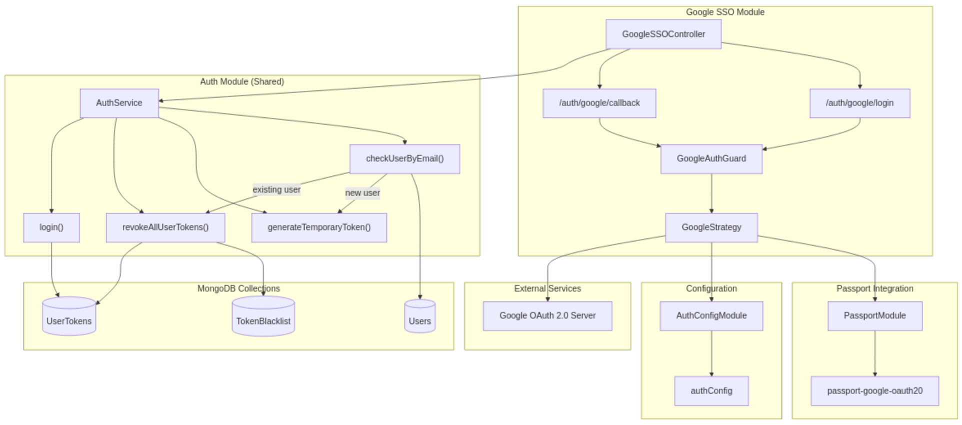


Figure 4.3: Google SSO Module Architecture

Key components include:

- **GoogleSSOController**: Exposes `/auth/google/login` and `/auth/google/callback` endpoints.
- **GoogleAuthGuard**: Extends Passport's AuthGuard with custom state parameter handling.
- **GoogleStrategy**: Implements Passport strategy with OAuth 2.0 configuration.
- **AuthService**: Shared service for user verification and token management.

Authentication Flow Overview

Figure 4.4 illustrates the high-level Google OAuth 2.0 flow.

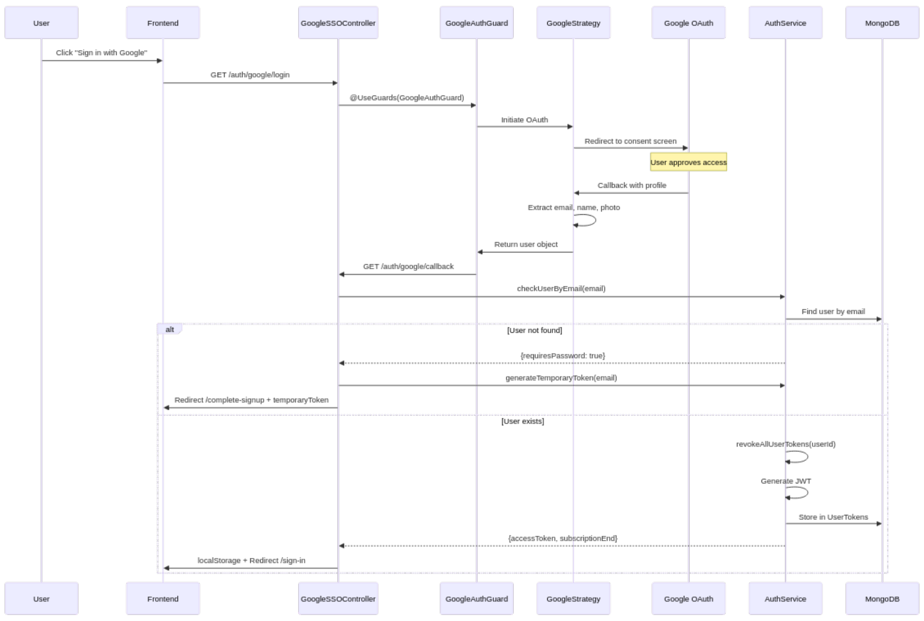


Figure 4.4: Google OAuth 2.0 Flow

Complete User Flow

Figure 4.5 provides a comprehensive flowchart of the entire Google Sign-In process, from initial click to session establishment.

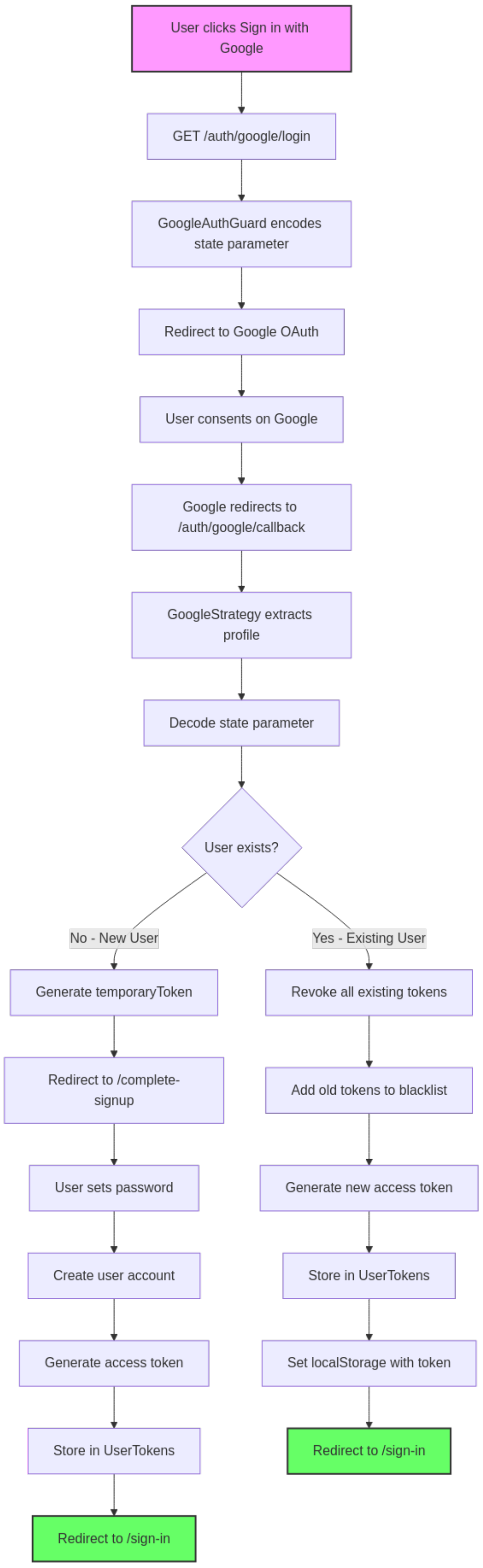


Figure 4.5: Complete Google SSO User Journey

The flow handles two scenarios:

- **New Users**: After Google authentication, the `checkUserByEmail` method indicates a new user by returning a result where `requiresPassword` is `true`. A temporary token is generated and stored in localStorage, after which the user is redirected to the complete-signup page to set their password.
- **Existing Users**: All previous tokens are revoked, a new access token is generated and stored in UserTokens collection and localStorage, then the user is redirected to sign-in.

Passport Authentication Flow

Figure 4.6 shows the detailed OAuth 2.0 authentication flow using Passport's Google strategy, including configuration and token exchange.

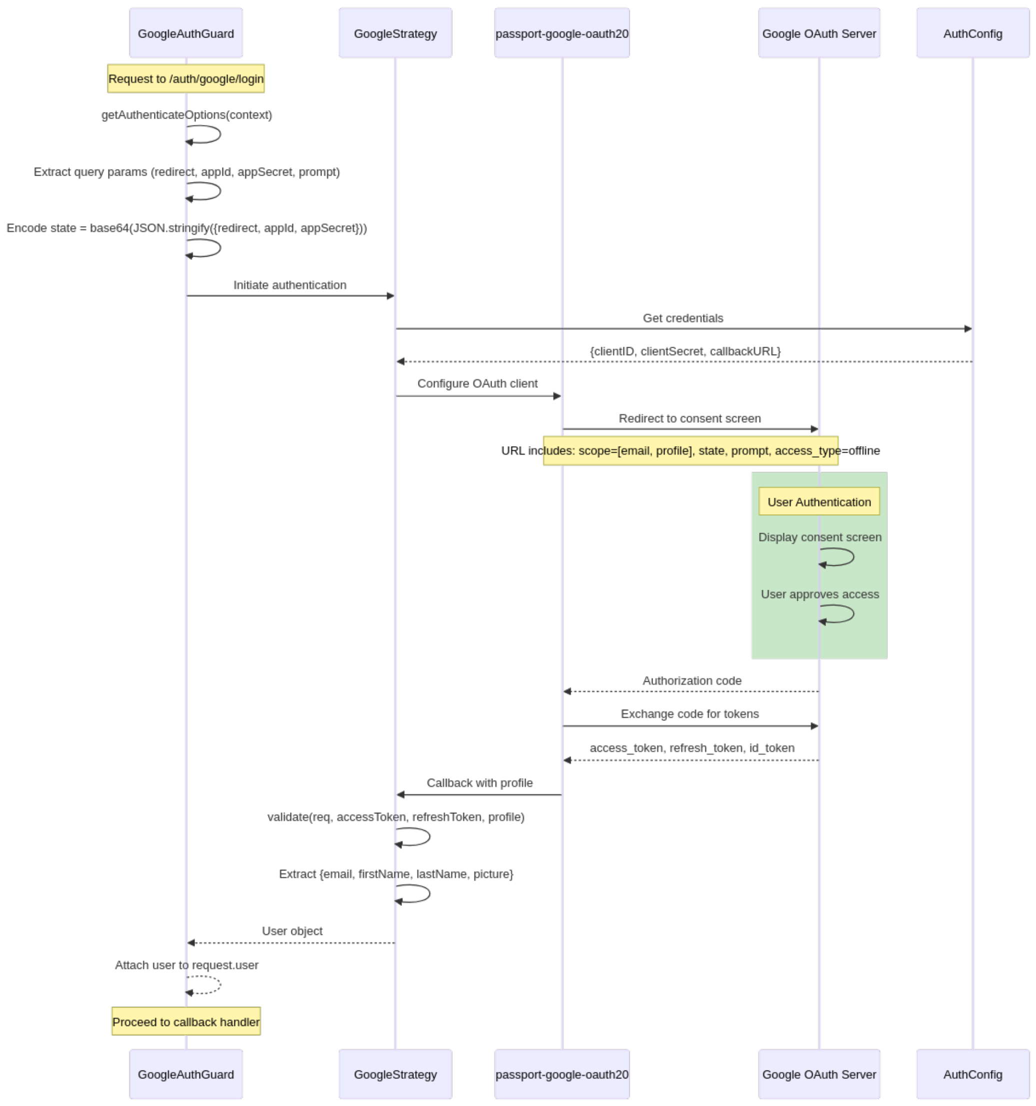

GoogleAuthGuard
GoogleStrategy
passport-google-oauth20
Google OAuth Server
AuthConfig
Request to /auth/google/login
getAuthenticateOptions(context)
Extract query params (redirect, appId, appSecret, prompt)
Encode state = base64(JSON.stringify({redirect, appId, appSecret}))
Initiate authentication
Get credentials
{clientID, clientSecret, callbackURL}
Configure OAuth client
Redirect to consent screen
URL includes: scope=[email, profile], state, prompt, access_type=offline
User Authentication
Display consent screen
User approves access
Authorization code
Exchange code for tokens
access_token, refresh_token, id_token
Callback with profile
validate(req, accessToken, refreshToken, profile)
Extract {email, firstName, lastName, picture}
User object
Attach user to request.user
Proceed to callback handler

Figure 4.6: Google OAuth Passport Authentication Flow

The authentication process involves:

1. **Guard Initialization**: GoogleAuthGuard extracts query parameters and encodes them into a base64 state parameter.

2. **Strategy Configuration**: GoogleStrategy configures OAuth client with credentials from AuthConfig.

3. **Token Exchange**: After user consent, Google returns an authorization code which is exchanged for access tokens.

4. **Profile Extraction**: The strategy validates and extracts user information (email, name, photo) from the profile.

State Parameter for Context Preservation

A key feature of the Google SSO implementation is preserving query parameters through the OAuth redirect cycle using the state parameter. Figure 4.7 illustrates this mechanism.

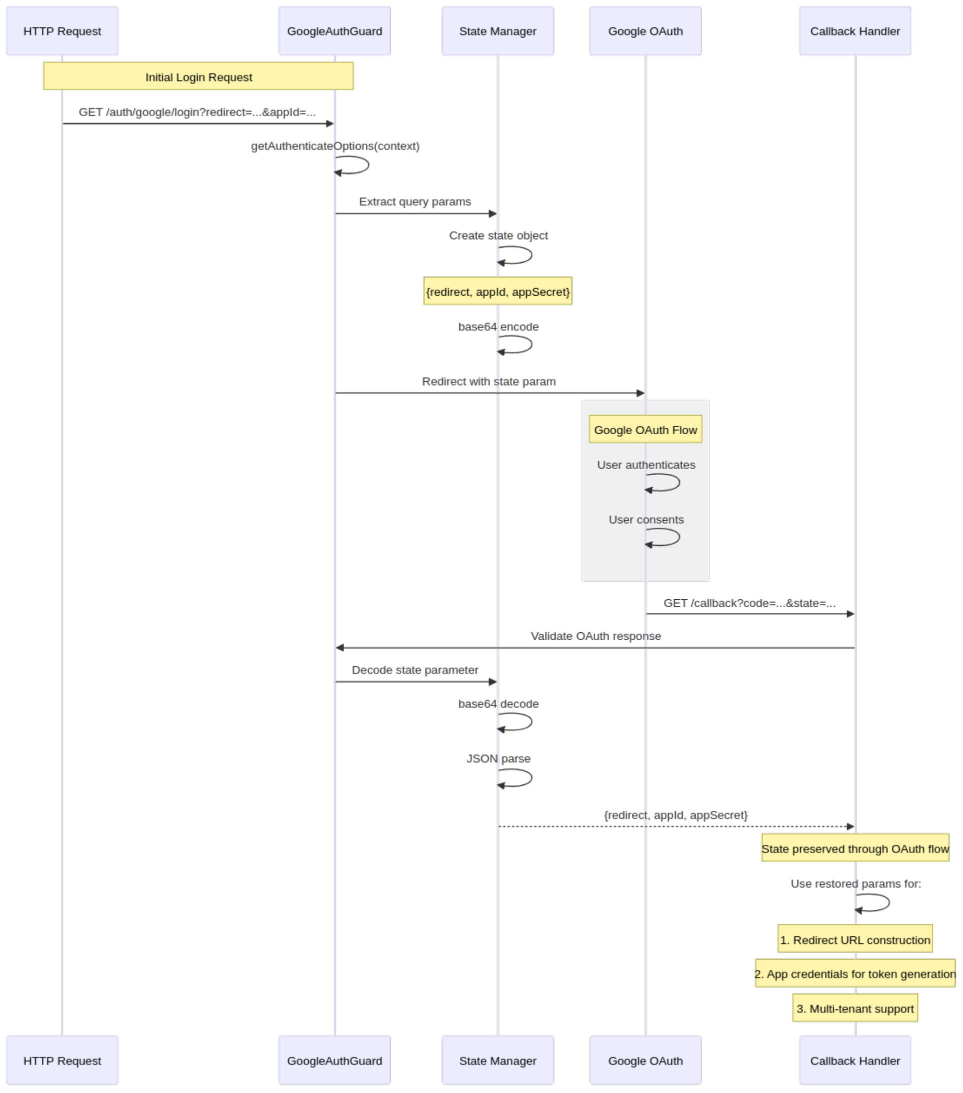


Figure 4.7: State Parameter Flow for Context Preservation

The state parameter enables:

- **Redirect URL Preservation**: Frontend callback URL is maintained across OAuth redirects.

- **Multi-tenant Support**: Application credentials (appId, appSecret) are preserved for token generation.
- **CSRF Protection**: The encoded state provides implicit protection against cross-site request forgery.

Implementation Details

Google Strategy Configuration. The GoogleStrategy extends Passport's base strategy with OAuth 2.0 configuration:

```
@Injectable()
export class GoogleStrategy extends
  PassportStrategy(Strategy, 'google') {
  constructor(@Inject(authConfig.KEY) private auth) {
    super({
      clientID: auth.googleClientId,
      clientSecret: auth.googleClientSecret,
      callbackURL: auth.googleCallbackURL,
      scope: ['email', 'profile'],
    });
  }

  async validate(req, accessToken, _refreshToken,
    profile, done) {
    const user = {
      email: profile.emails[0].value,
      firstName: profile.name.givenName,
      lastName: profile.name.familyName,
    };
    done(null, user);
  }
}
```

Listing 4.1: GoogleStrategy Implementation

Callback Handler Logic. The callback determines whether to redirect to complete-signup (new user) or sign-in (existing user):

```
@Get('callback')
@UseGuards(GoogleAuthGuard)
async googleAuthRedirect(@Req() req, @Res() res) {
  const result = await this.authService.checkUserByEmail(
    req.user.email, userAgent, ipAddress
  );

  if (result.requiresPassword) {
    // New user - generate temporary token
    const tempToken = await this.authService
      .generateTemporaryToken(result.email);
    // Redirect to /complete-signup
  } else if (result.accessToken) {
    // Existing user - store token and redirect
    }
}
```

Listing 4.2: Google Callback Handler

4.2.3 Apple Sign-In

Apple Sign-In uses a POST callback with the identity token, differing from Google's redirect-based OAuth flow. This subsection covers the module architecture, authentication flow, and implementation details.

Apple SSO Module Structure

Figure 4.8 shows the architecture of the Apple SSO module and its integration with the shared Auth module.

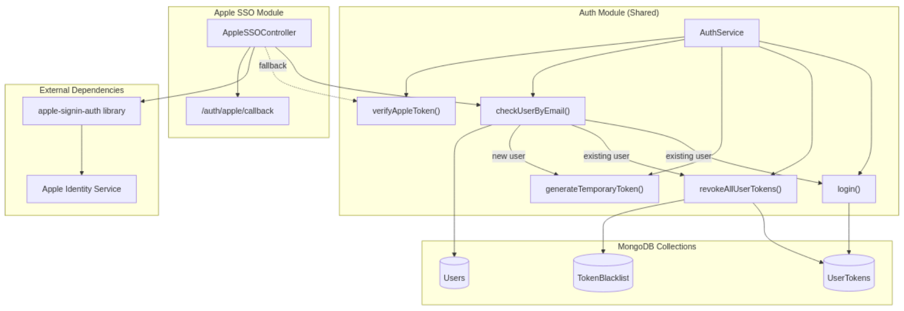


Figure 4.8: Apple SSO Module Architecture

Key components include:

- **AppleSSOController**: Handles the `POST /auth/apple/callback` endpoint.
- **apple-signin-auth library**: External dependency for token verification.
- **AuthService**: Shared service providing core functionality:
  - `verifyAppleToken()`
  - `checkUserByEmail()`
  - `generateTemporaryToken()`
  - `revokeAllUserTokens()`

Authentication Flow Overview

Figure 4.9 shows the high-level Apple Sign-In flow.

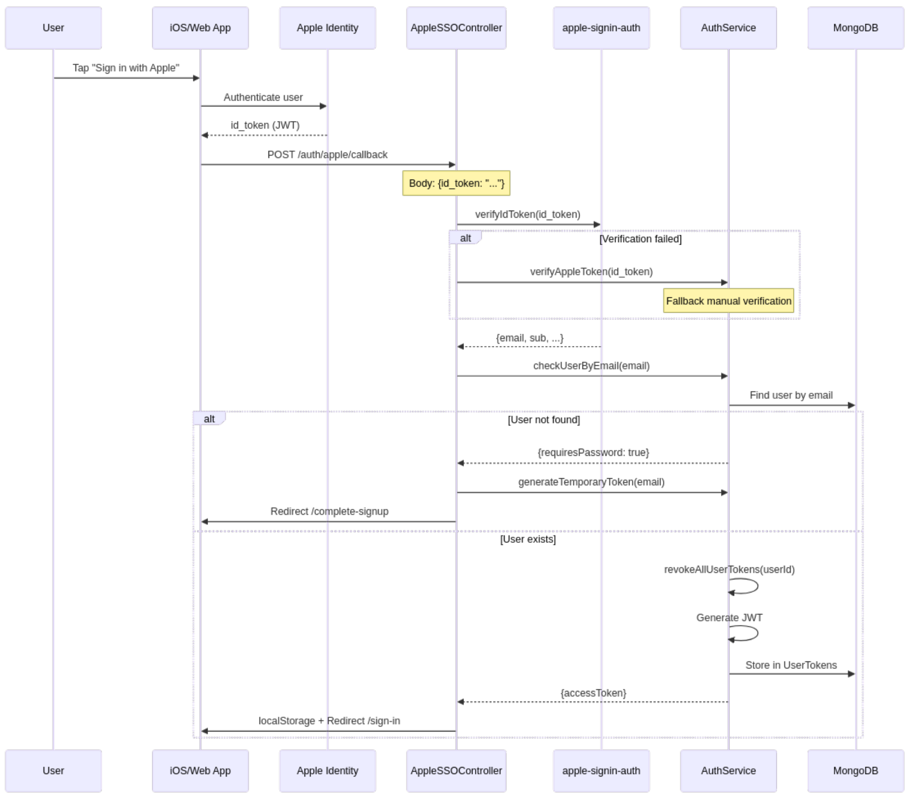


Figure 4.9: Apple Sign-In Flow

Complete User Flow

Figure 4.10 provides a comprehensive view of the entire Apple Sign-In process, from initial authentication to session establishment.

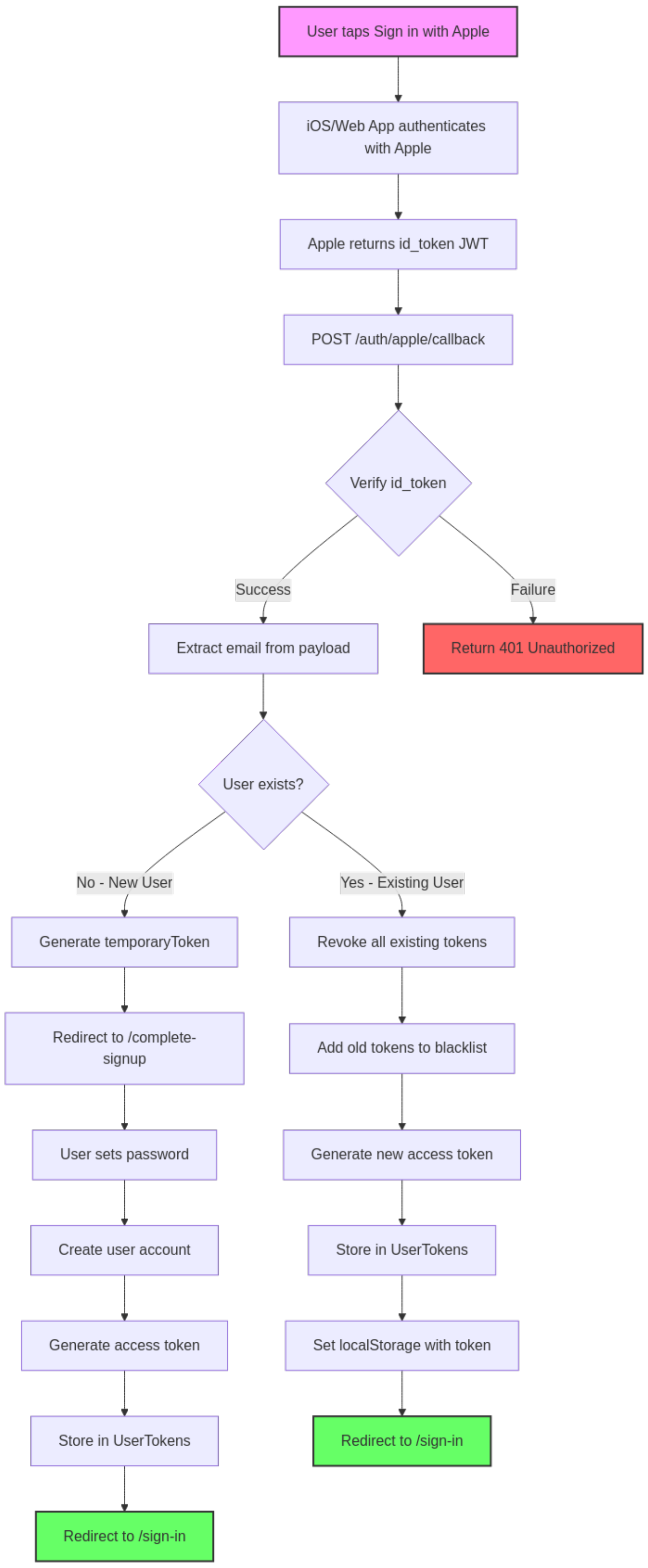

User taps Sign in with Apple
iOS/Web App authenticates with Apple
Apple returns id_token JWT
POST /auth/apple/callback
Verify id_token
Success
Failure
Extract email from payload
Return 401 Unauthorized
User exists?
No - New User
Yes - Existing User
Generate temporaryToken
Redirect to /complete-signup
User sets password
Create user account
Generate access token
Store in UserTokens
Redirect to /sign-in
Revoke all existing tokens
Add old tokens to blacklist
Generate new access token
Store in UserTokens
Set localStorage with token
Redirect to /sign-in

Figure 4.10: Complete Apple SSO User Journey

The flow branches based on whether the user exists:

- **New Users**: Receive a temporary token and are redirected to complete signup, where they set a password before account creation.
- **Existing Users**: Have their previous tokens revoked (single session enforcement),

receive a new access token stored in `UserTokens`, and are redirected with the token in localStorage.

Token Verification Architecture

The token verification process in Apple SSO is dual-layered. It primarily utilizes the `apple-signin-auth` library, but retains a manual fallback mechanism. Figure 4.11 illustrates this verification strategy.

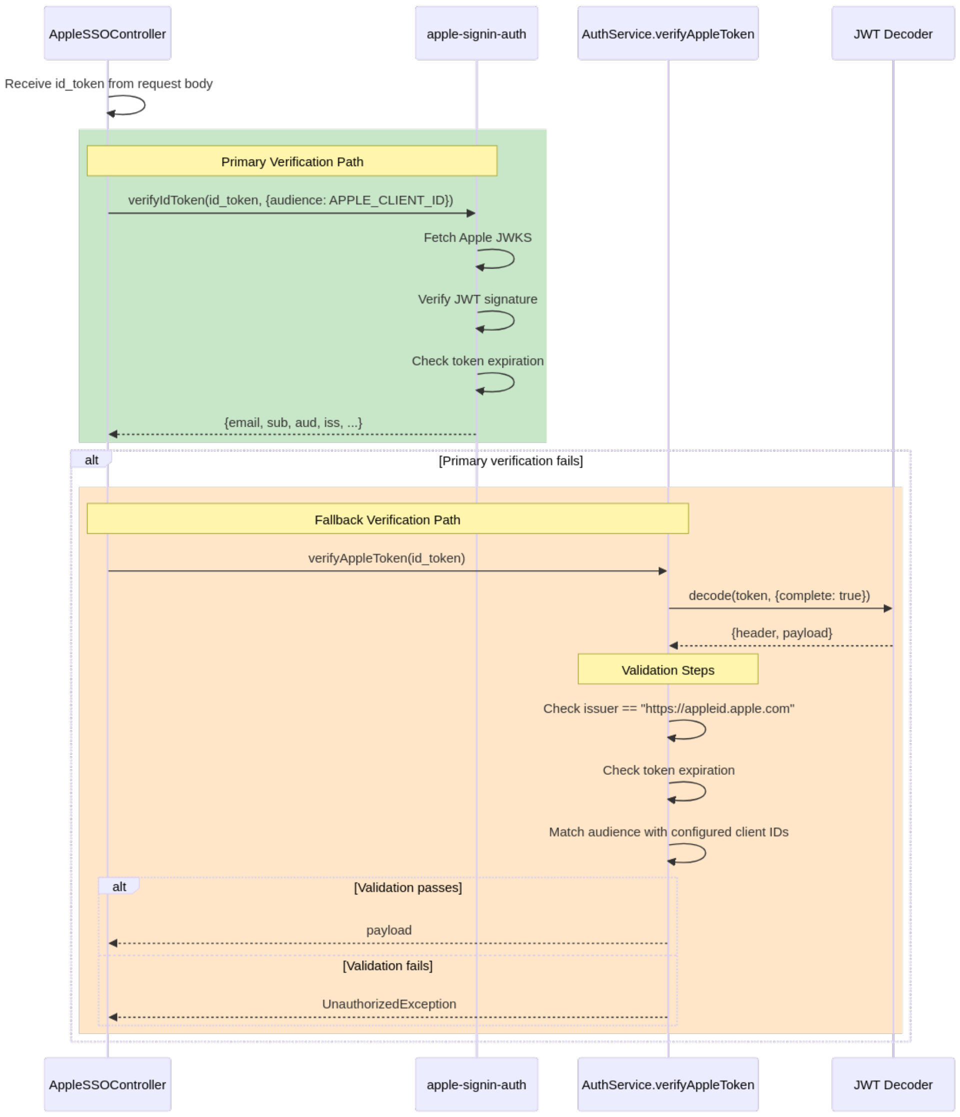

AppleSSOController
apple-signin-auth
AuthService.verifyAppleToken
JWT Decoder
Receive id_token from request body
Primary Verification Path
verifyIdToken(id_token, {audience: APPLE_CLIENT_ID})
Fetch Apple JWKS
Verify JWT signature
Check token expiration
{email, sub, aud, iss, ...}
alt
[Primary verification fails]
Fallback Verification Path
verifyAppleToken(id_token)
decode(token, {complete: true})
{header, payload}
Validation Steps
Check issuer == "https://appleid.apple.com"
Check token expiration
Match audience with configured client IDs
alt
[Validation passes]
payload
[Validation fails]
UnauthorizedException
AppleSSOController
apple-signin-auth
AuthService.verifyAppleToken
JWT Decoder

Figure 4.11: Apple SSO Token Verification Flow

The verification process:

1. **Primary Path**: The controller uses the `apple-signin-auth` library to verify the id_token. The library fetches Apple's JWKS, verifies the JWT signature, and checks token expiration.

2. **Fallback Path**: If the primary verification fails, the controller falls back to the `verifyAppleToken()` method of the `AuthService`, which manually decodes and validates the token by checking the issuer, expiration, and audience against configured Apple Client IDs.

Implementation Details

```
@Post('callback')
async appleAuthCallback(@Body() body, @Res() res) {
  let appleResponse;
  try {
    // Primary: use apple-signin-auth library
    appleResponse = await appleSignin.verifyIdToken(
      body.id_token,
      { audience: process.env.APPLE_CLIENT_ID }
    );
  } catch (error) {
    // Fallback: manual verification
    appleResponse = await this.authService
      .verifyAppleToken(body.id_token);
  }

  const result = await this.authService
    .checkUserByEmail(appleResponse.email);
  // Handle new user or existing user...
}
```

Listing 4.3: Apple Token Verification

4.2.4 Comparison of SSO Methods

Table 4.2 compares the key differences between Google OAuth and Apple Sign-In implementations.

Aspect	**Google OAuth**	**Apple Sign-In**
Flow Type	OAuth 2.0 redirect	POST with id_token
Guard	GoogleAuthGuard (Passport)	None (manual verification)
Verification	Passport strategy	apple-signin-auth library
Email Privacy	Direct email provided	May use relay email
State Management	Base64-encoded state parameter	Not applicable

Table 4.2: Google vs Apple SSO Comparison

4.3 Token Management

This section covers the JWT token lifecycle, including the API endpoints for token operations and the Auth module architecture.

4.3.1 Auth Module Architecture

Figure 4.12 shows the architecture of the Auth module, including the relationships between controllers, guards, strategies, services, and storage.

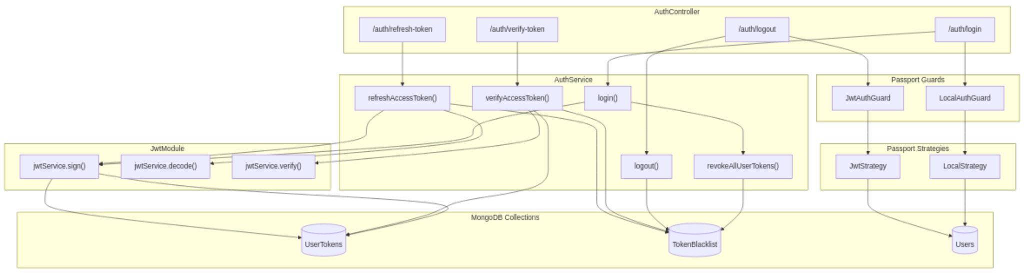


Figure 4.12: Auth Module Architecture

The module consists of:

- **AuthController**: Exposes endpoints for login, logout, token verification, and token refresh.
- **Guards**: LocalAuthGuard for credential-based login, JwtAuthGuard for protected endpoints.
- **Strategies**: LocalStrategy validates email/password, JwtStrategy validates JWT tokens with dynamic key selection.
- **AuthService**: Core business logic for token operations.
- **Storage**: UserTokens tracks active sessions, TokenBlacklist stores revoked tokens.

4.3.2 Token Lifecycle Overview

Stage	**API Endpoint**	**Description**
Generation	POST /auth/login	Creates JWT on successful authentication
Verification	POST /auth/verify-token	Validates token against blacklist and expiry
Refresh	POST /auth/refresh-token	Issues new token, blacklists old one
Revocation	POST /auth/logout	Adds token to blacklist immediately

Table 4.3: Token Lifecycle Stages

Figure 4.13 illustrates the complete token lifecycle sequence.

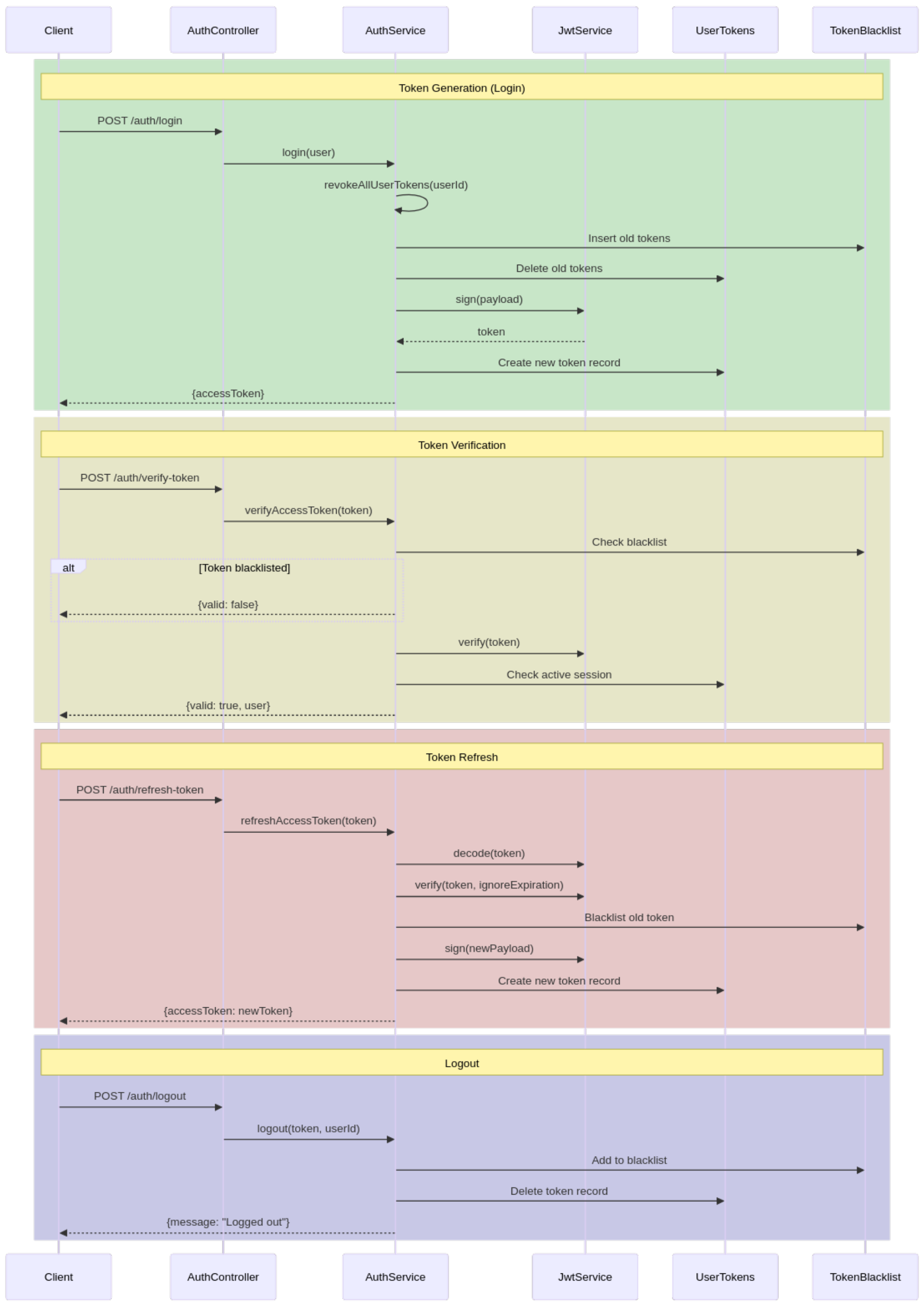


Figure 4.13: Token Lifecycle Sequence

4.3.3 Token Generation

Upon successful authentication, the AuthService generates a JWT containing user claims:

```
async login(user, userAgent, ipAddress) {
  // Revoke existing tokens (single session)
  await this.revokeAllUserTokens(user._id.toString());

  const payload = {
    email: user.email,
    role: user.role,
    name: user.name,
    type: user.type,
  };

  const token = this.jwtService.sign(payload, {
    subject: user._id.toString()
  });
  const decoded = this.jwtService.decode(token);

  // Store active token
  await this.userTokenModel.create({
    userId: user._id.toString(),
    token,
    expiresAt: new Date(decoded.exp * 1000),
    userAgent,
    ipAddress,
  });

  return { accessToken: token, ... };
}
```

Listing 4.4: JWT Token Generation

4.3.4 Token Verification

The verification process involves multiple validation steps. Figure 4.14 shows the complete decision tree.

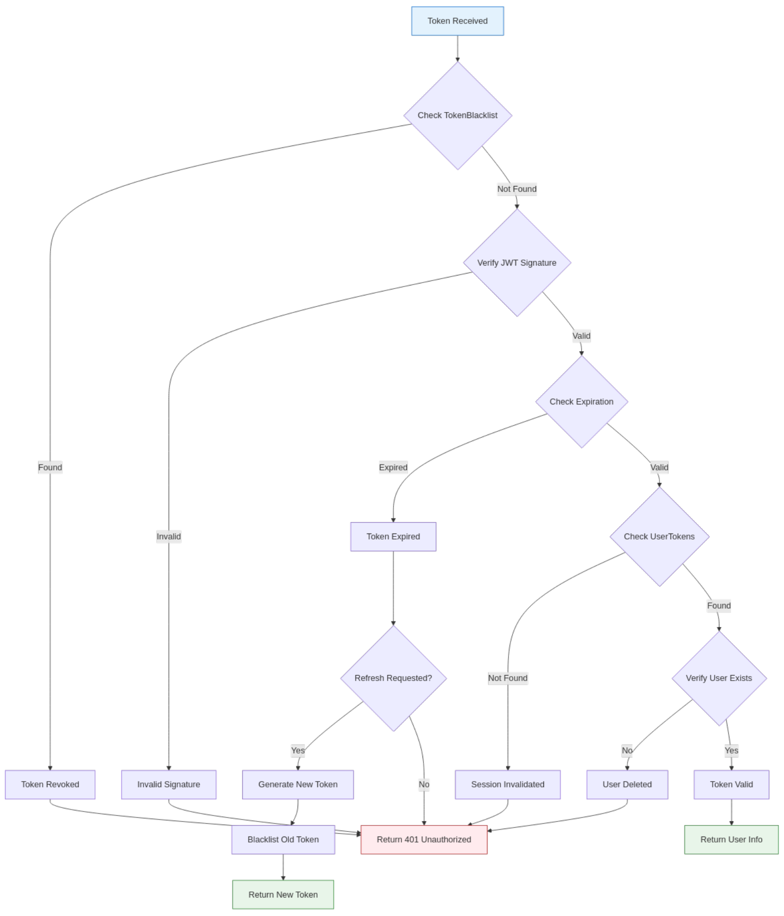


Figure 4.14: Token Verification Decision Flow

The verification endpoint checks blacklist status, signature validity, and active session existence:

```
async verifyToken(token: string) {
  // 1. Check blacklist
  const isBlacklisted = await this.tokenBlacklistModel
    .findOne({ token });
  if (isBlacklisted) {
    throw new UnauthorizedException('Token revoked');
  }

  // 2. Verify signature and expiration
  const decoded = await this.jwtService.verifyAsync(token);

  // 3. Check active session
  const activeToken = await this.userTokenModel
    .findOne({ token });
  if (!activeToken) {
    throw new UnauthorizedException('Session expired');
  }

  return { user: decoded };
}
```

Listing 4.5: Token Verification

4.3.5 JWT Validation with Dynamic Keys

The JwtStrategy supports multi-tenant authentication with dynamic key selection. Figure 4.15 shows this process.

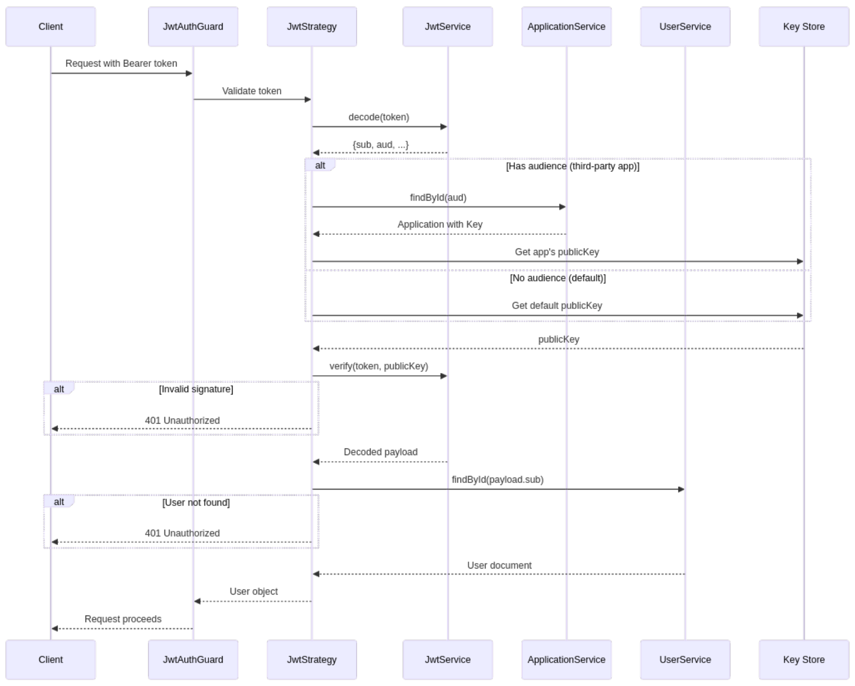


Figure 4.15: JWT Validation with Dynamic Key Selection

The strategy determines which public key to use based on the token's audience claim:

- **Default tokens**: Use the platform's default public key.
- **Third-party app tokens**: Use the application's specific public key for verification.

4.3.6 Token Refresh

The token refresh process allows clients to obtain a new access token using an expired token, provided the signature is valid. Figure 4.16 illustrates this flow.

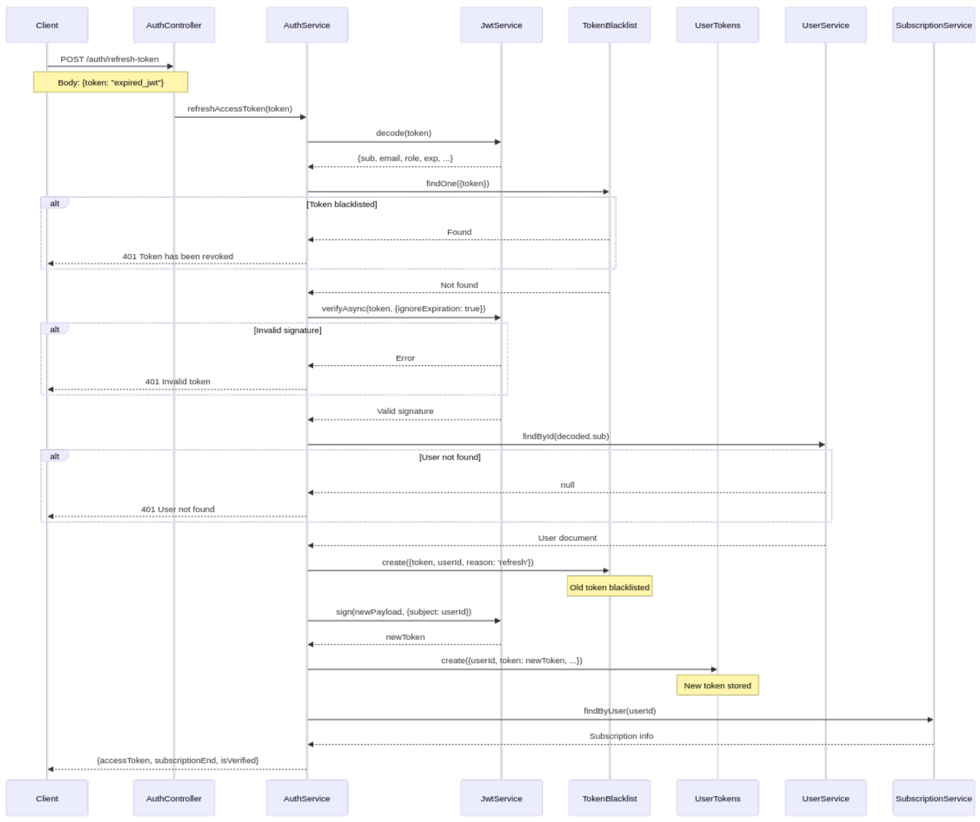


Figure 4.16: Token Refresh Flow

Refresh enables session continuation with expired tokens (signature must still be valid):

```
async refreshAccessToken(expiredToken: string) {
  // Decode without expiration check
  const decoded = this.jwtService.decode(expiredToken);

  // Verify signature only
  await this.jwtService.verifyAsync(expiredToken, {
    ignoreExpiration: true
  });

  // Blacklist old token
  await this.tokenBlacklistModel.create({
    token: expiredToken,
    userId: decoded.sub,
    expiresAt: new Date(decoded.exp * 1000),
    reason: 'refresh',
  });

  // Generate new token
  const newToken = this.jwtService.sign({...}, {
    subject: decoded.sub
  });

  return { accessToken: newToken };
}
```

Listing 4.6: Token Refresh

4.3.7 Token Revocation (Logout)

The logout process involves removing the active session and blacklisting the current token. Figure 4.17 details this sequence.

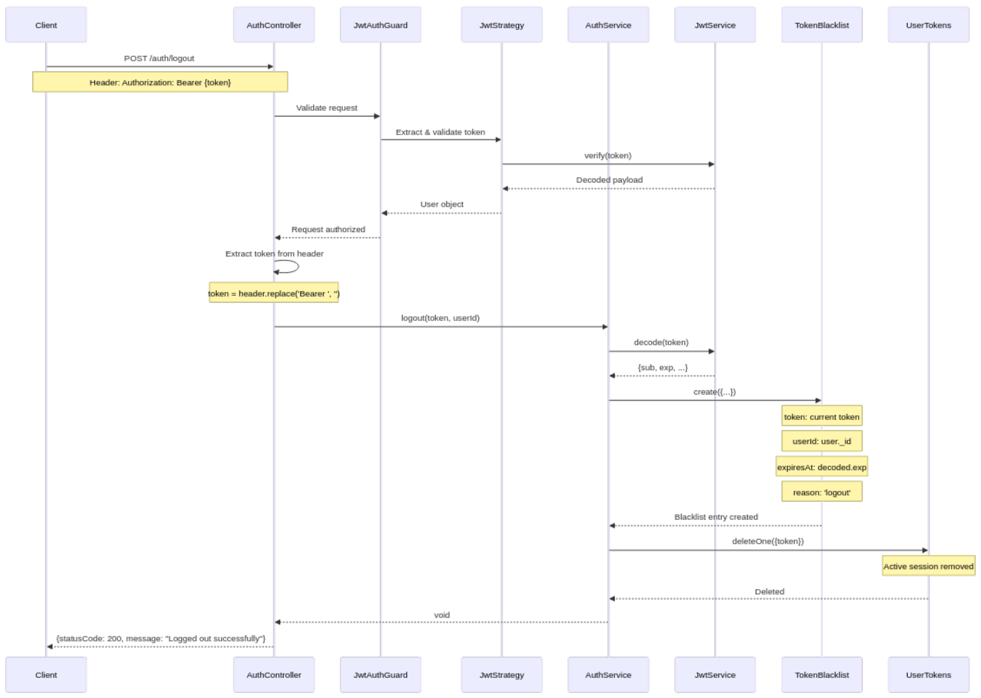


Figure 4.17: Token Revocation (Logout) Flow

```
async logout(token: string, userId: string) {
  const decoded = this.jwtService.decode(token);

  // Add to blacklist
  await this.tokenBlacklistModel.create({
    token,
    userId,
    expiresAt: new Date(decoded.exp * 1000),
    reason: 'logout',
  });

  // Remove from active tokens
  await this.userTokenModel.deleteOne({ token });
}
```

Listing 4.7: Logout Implementation

4.4 Session Security

This section details the security mechanisms including database schemas, single session enforcement, and token blacklisting.

4.4.1 Database Schema Design

Two MongoDB collections support the session security system.

UserTokens Collection

Tracks all active sessions with metadata for auditing:

```
@Schema({ timestamps: true })
export class UserToken extends Document {
  @Prop({ required: true, index: true })
  userId: string;

  @Prop({ required: true })
  token: string;

  @Prop({ required: true, type: Date })
  expiresAt: Date;

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

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

// TTL index - auto delete when expired
UserTokenSchema.index(
  { expiresAt: 1 },
  { expireAfterSeconds: 0 }
);
```

Listing 4.8: UserToken Schema

TokenBlacklist Collection

Stores revoked tokens to prevent reuse:

```
@Schema({ timestamps: true })
export class TokenBlacklist extends Document {
  @Prop({ required: true, index: true })
  token: string;

  @Prop({ required: true, index: true })
  userId: string;

  @Prop({ required: true, type: Date })
  expiresAt: Date;

  @Prop({ enum: ['logout', 'new-login', 'refresh'] })
  reason: string;
}

// TTL index for automatic cleanup
```

```
TokenBlacklistSchema.index(
  { expiresAt: 1 },
  { expireAfterSeconds: 0 }
);
```

Listing 4.9: TokenBlacklist Schema

4.4.2 Single Session Enforcement

When a user logs in, all existing sessions are invalidated:

```
async revokeAllUserTokens(userId: string) {
  // Find all active tokens
  const activeTokens = await this.userTokenModel.find({
    userId,
    expiresAt: { $gt: new Date() },
  });

  if (activeTokens.length === 0) return;

  // Bulk insert to blacklist
  const entries = activeTokens.map(t => ({
    token: t.token,
    userId: t.userId,
    expiresAt: t.expiresAt,
    reason: 'new-login',
  }));

  await this.tokenBlacklistModel.insertMany(entries);
  await this.userTokenModel.deleteMany({ userId });
}
```

Listing 4.10: Single Session Enforcement

Figure 4.18 illustrates the session security flow.

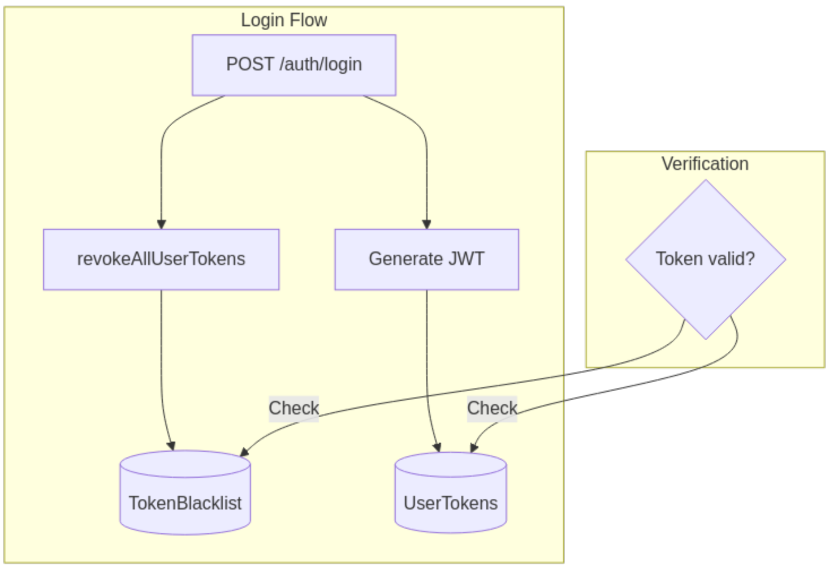


Figure 4.18: Session Security Flow

4.4.3 Token Blacklisting Flow

Figure 4.19 shows how tokens are blacklisted under different scenarios.

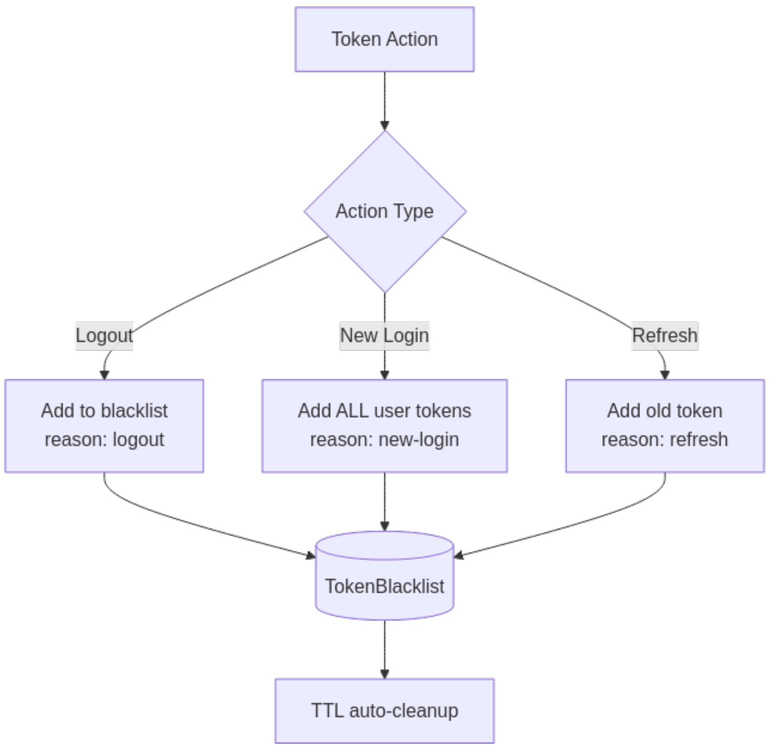

Token Action
Action Type
Logout
New Login
Refresh
Add to blacklist reason: logout
Add ALL user tokens reason: new-login
Add old token reason: refresh
TokenBlacklist
TTL auto-cleanup

Figure 4.19: Token Blacklisting Flow

Reason	**Trigger**	**Method**
logout	User logout	`logout()`
new-login	New login revokes old	`revokeAllUserTokens()`
refresh	Token refresh	`refreshAccessToken()`

Table 4.4: Token Blacklist Reasons

4.4.4 Security Benefits

Feature	**Benefit**
Single Session	Stolen tokens invalidated on new login
IP/UserAgent Tracking	Audit trail for security review
Token Blacklisting	Immediate revocation capability
TTL Auto-cleanup	MongoDB automatically removes expired entries

Table 4.5: Security Features and Benefits

Chapter 5

Payment and Subscription System

5.1 Overview

The Gateway Dashboard implements a comprehensive payment and subscription system using Stripe as the payment processor. This chapter covers the three interconnected modules that handle plan management, payment processing, and subscription tracking.

5.1.1 Payment System Architecture

Figure 5.1 illustrates the overall architecture of the payment system, showing the relationships between the Plan, Subscription, and Stripe modules.

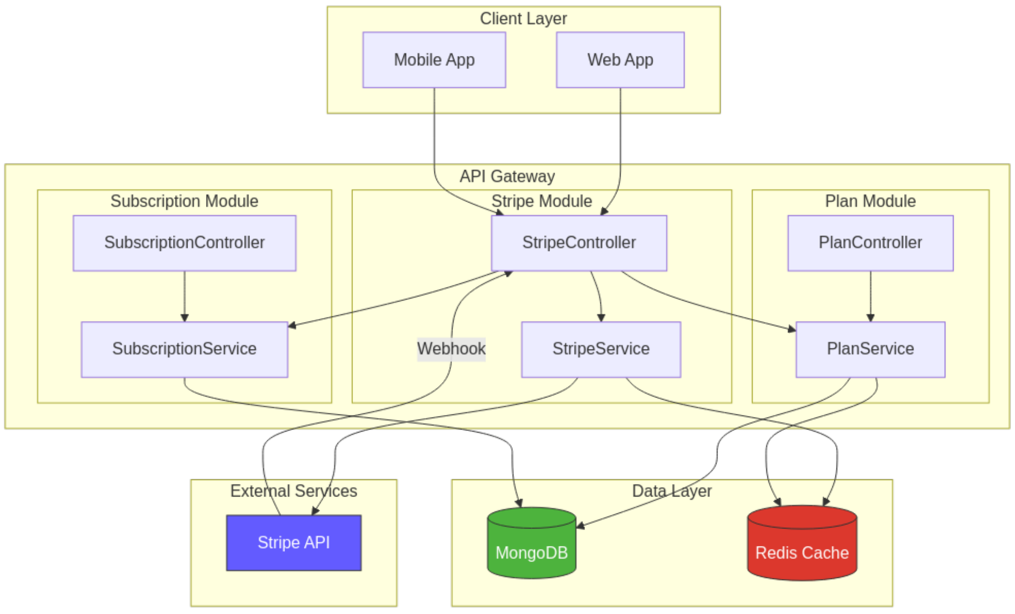


Figure 5.1: Payment System Architecture

The architecture consists of three primary modules:

- **Plan Module**: Manages versioned pricing plans with Redis caching for performance optimization.
- **Stripe Module**: Handles payment processing, checkout sessions, and webhook events from Stripe.

- **Subscription Module**: Tracks user subscriptions with quota and usage monitoring.

5.1.2 Module Relationships

Module	Dependencies	Responsibility
Plan	MongoDB, Redis	Plan versioning and caching
Stripe	PlanService, SubscriptionService, Redis	Payment processing and webhooks
Subscription	MongoDB	Quota tracking and access control

Table 5.1: Payment Module Dependencies

5.1.3 Payment Data Flow

The payment flow follows a clear sequence:

1. **Plan Selection**: User browses available plans retrieved via the Plan module.
2. **Checkout Initiation**: Frontend requests a Stripe checkout session from the API.
3. **Payment Processing**: User completes payment on Stripe's hosted checkout page.
4. **Webhook Processing**: Stripe sends a webhook notification; API matches payment to plan.
5. **Subscription Creation**: Subscription module creates/updates user subscription with quota.
6. **Access Control**: Subsequent API requests are validated against subscription quota.

5.1.4 Database Schema Overview

The payment system uses two primary collections:

Collection	Purpose
`plans`	Versioned pricing plans with features and Stripe price IDs
`subscriptions`	User subscription status, quota limits, and usage tracking

Table 5.2: Payment System Collections

5.1.5 Chapter Organization

This chapter is organized as follows:

- **Section 4.2**: Plan management with versioning and Redis caching
- **Section 4.3**: Subscription management and quota tracking
- **Section 4.4**: Stripe integration for payment processing

5.2 Plan Management

The Plan module manages pricing plans with versioning support and Redis caching for high-performance retrieval.

5.2.1 Module Architecture

The Plan module consists of three components organized with clear separation of concerns:

- **PlanController**: Exposes REST endpoints for plan management.
- **PlanService**: Contains business logic for versioning and caching.
- **Plan Schema**: Defines the MongoDB document structure.

5.2.2 Database Schema

Plans are stored as versioned documents, allowing for plan updates without affecting existing subscriptions.

Field	**Type**	**Description**
_id	ObjectId	Primary key
id	String	Unique random hex identifier
plans	String	JSON string containing plan array
version	Number	Auto-incremented version number
createdAt	Date	Creation timestamp

Table 5.3: Plans Collection Schema

Each plan within the JSON structure contains:

- `id`: Unique plan identifier (e.g., "basic", "pro")
- `name`: Display name for the plan
- `features`: Array of feature descriptions
- `priceId`: Stripe price ID for billing
- `batchDuration`: Allowed batch processing seconds (-1 for unlimited)
- `liveDuration`: Allowed live processing seconds (-1 for unlimited)

5.2.3 Plan Versioning Flow

Figure 5.2 illustrates the plan creation process with automatic versioning and cache invalidation.

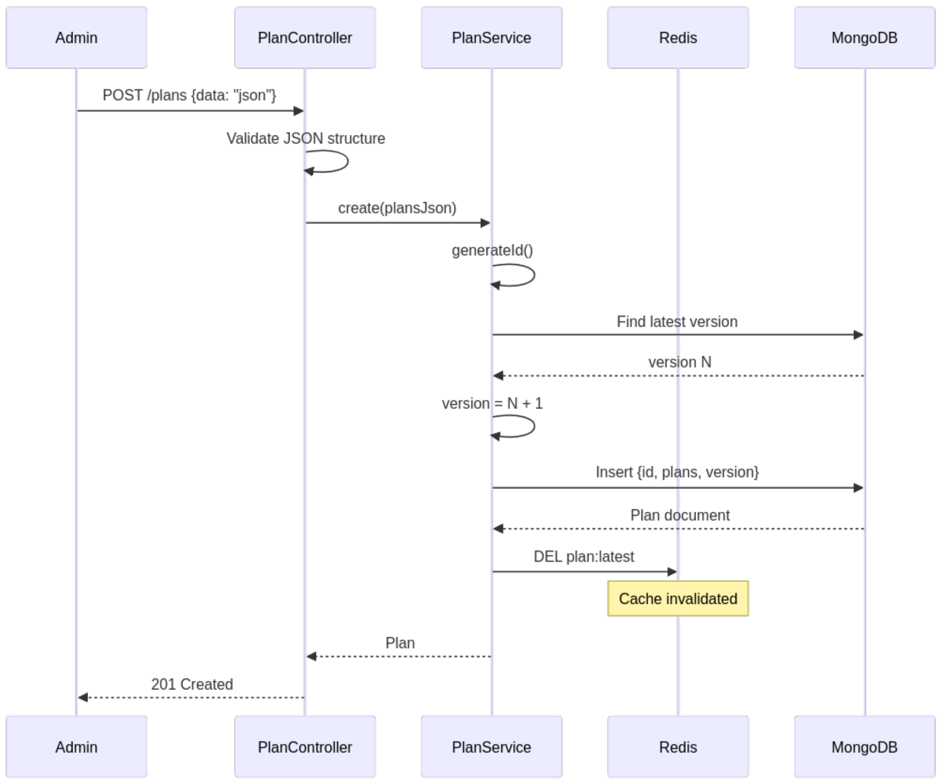


Figure 5.2: Plan Versioning Flow with Cache Invalidation

The versioning process ensures:

1. Each plan version is immutable once created.
2. New versions automatically receive an incremented version number.
3. Existing subscriptions reference their original plan version.
4. Cache is invalidated immediately upon new version creation.

5.2.4 Redis Caching Implementation

The Plan module implements a read-through caching strategy with cache penetration protection.

Caching Strategy

Cache Key	TTL	Purpose
`plan:latest`	1 hour	Cache latest plan version
`plan:latest` (NULL)	60 seconds	Prevent cache penetration

Table 5.4: Plan Cache Configuration

Implementation Details

```
private readonly CACHE_KEY = 'plan:latest'
private readonly CACHE_TTL = 3600     // 1 hour
private readonly NULL_TTL = 60        // 60 seconds

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

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

  if (latestPlan) {
    await this.redisClient.set(
      this.CACHE_KEY, JSON.stringify(latestPlan),
      { EX: this.CACHE_TTL }
    )
  } else {
    // Cache NULL to prevent repeated DB queries
    await this.redisClient.set(
      this.CACHE_KEY, 'NULL', { EX: this.NULL_TTL }
    )
  }

  return latestPlan
}
```

Listing 5.1: Plan Caching Implementation

5.2.5 API Endpoints

Create Plan (Admin Only)

```
@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(Role.Admin)
@Post()
async create(@Body() dto: CreatePlansDto) {
  await dto.validateDataStructure()
  const result = await this.planService.create(dto.data)
  return {
    statusCode: HttpStatus.CREATED,
    message: 'Plans created successfully',
    data: { plans: result.plans, version: result.version },
  }
}
```

Listing 5.2: Plan Creation Endpoint

Get Latest Plans (Public)

```
@Get('latest')
async getLatestVersion() {
  const plans = await this.planService.getLatestVersion()
  return {
    statusCode: HttpStatus.OK,
    message: 'Latest plans retrieved successfully',
    data: plans,
  }
}
```

Listing 5.3: Get Latest Plans Endpoint

5.3 Subscription Management

The Subscription module tracks user subscriptions, including quota limits, usage tracking, and access control.

5.3.1 Module Architecture

The Subscription module provides the core access control mechanism:

- **SubscriptionController**: Exposes endpoints for viewing and managing subscriptions.
- **SubscriptionService**: Contains quota checking and usage tracking logic.
- **Subscription Schema**: Defines the MongoDB document with quota and usage fields.

5.3.2 Database Schema

Field	Type	Description
_id	ObjectId	Primary key
user	ObjectId	Reference to Users collection
stripeSubscriptionId	String	Stripe subscription identifier
stripeCustomerId	String	Stripe customer identifier
status	String	Subscription status (active, canceled, etc.)
quota	Object	Allowed limits (batchDuration, liveDuration)
usage	Object	Current usage (batchDuration, liveDuration)
startDate	Date	Subscription start date
endDate	Date	Subscription end date

Table 5.5: Subscriptions Collection Schema

5.3.3 Quota System

The quota system uses two embedded objects to track limits and usage:

```
@Prop(raw({
  batchDuration: { type: Number, default: -1 },
  liveDuration: { type: Number, default: -1 },
}))
quota: SubscriptionUnit

@Prop(raw({
  batchDuration: { type: Number, default: 0 },
  liveDuration: { type: Number, default: 0 },
}))
usage: SubscriptionUnit
```

Listing 5.4: Quota and Usage Schema Definition

Value	Meaning
-1	Unlimited access
0	No access to this feature
> 0	Number of seconds allowed

Table 5.6: Quota Value Interpretation

5.3.4 Quota Checking Flow

Figure 5.3 shows the decision tree for validating user access.

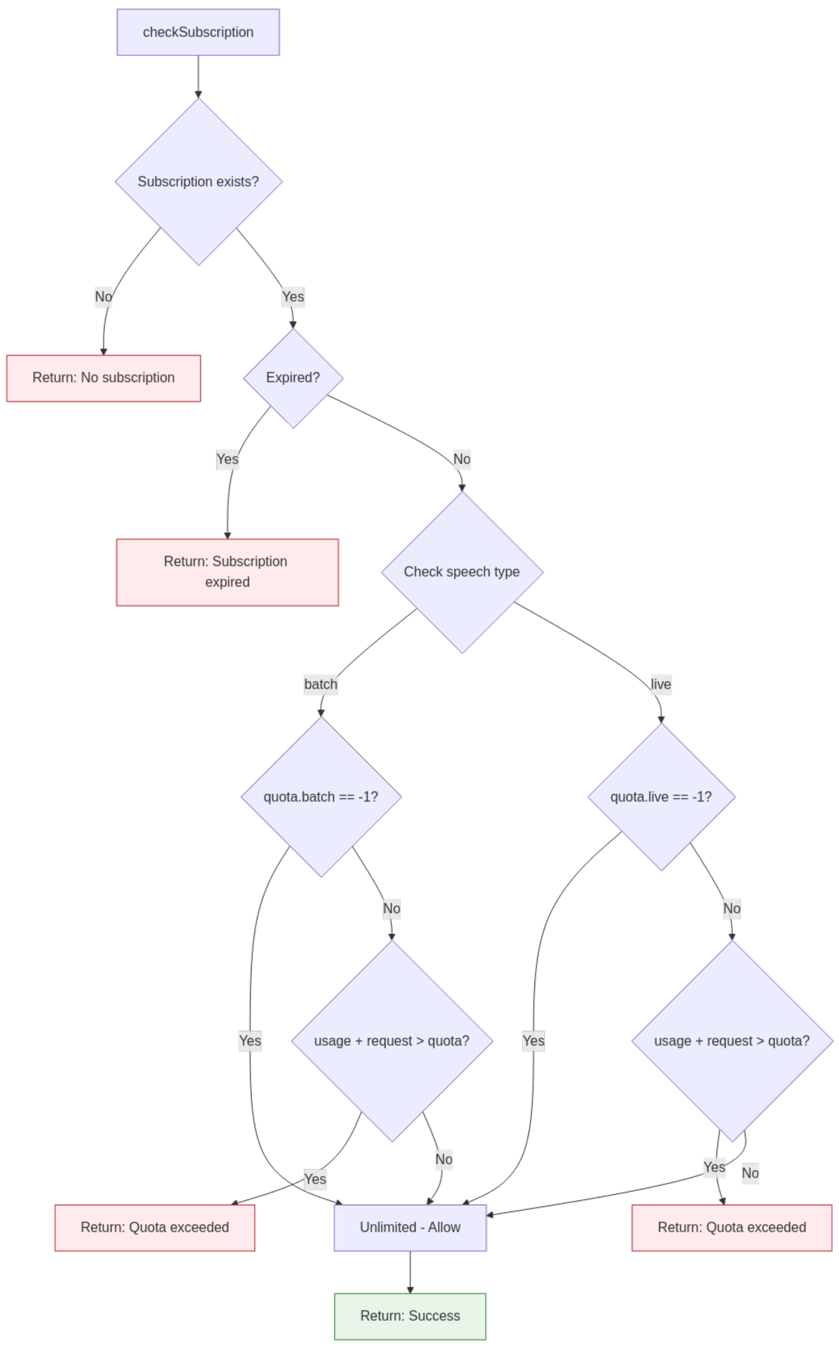


Figure 5.3: Subscription Quota Check Flow

5.3.5 Implementation Details

Quota Check Logic

```
async checkSubscription(
  userId: string,
  speechType?: 'live' | 'batch',
  payload?: SubscriptionUnit,
): Promise<{ success: boolean; message: string }> {
  const sub = await this.subscriptionModel
    .findOne({ user: userId }).exec()

  if (!sub) {
    return { success: false, message: 'No subscription' }
  }

  if (new Date() > sub.endDate) {
    return { success: false, message: 'Subscription expired' }
  }

  // Check batch quota (if not unlimited)
  if (speechType === 'batch' && sub.quota.batchDuration !== -1) {
    const wouldExceed = payload
      ? sub.usage.batchDuration + payload.batchDuration
        > sub.quota.batchDuration
      : sub.usage.batchDuration > sub.quota.batchDuration

    if (wouldExceed) {
      return { success: false, message: 'Quota exceeded' }
    }
  }

  // Similar check for live quota...
  return { success: true, message: null }
}
```

Listing 5.5: Quota Check Implementation

Authentication Integration

The subscription status is included in the login response:

```
// In AuthService.login()
const subscription = await this.subscriptionService
  .findByUser(user._id.toString())

return {
  accessToken: token,
  subscriptionEnd: subscription?.endDate?.getTime() || null,
  isVerified: user.isVerified,
}
```

Listing 5.6: Login Response with Subscription

5.3.6 API Endpoints

Get Subscription

Users can view their own subscription; admins can view any:

```
@UseGuards(JwtAuthGuard)
@Get('/user/:id')
getSubscriptionByUser(
  @Param() params: GetSubscriptionByUserDto,
  @Req() req: IRequest,
): Promise<Subscription> {
  if (req.user._id.toString() === params.id
      || req.user.role === 'admin') {
    return this.subscriptionService.findByUser(params.id)
  }
  throw new ForbiddenException()
}
```

Listing 5.7: Get Subscription Endpoint

Update Subscription (Admin Only)

```
@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(Role.Admin)
@Put('/user/:id')
updateSubscriptionByUser(
  @Body() body: UpdateSubscriptionByUserDto,
  @Param() params: GetSubscriptionByUserDto,
): Promise<Subscription> {
  const endDate = new Date(body.endDate)

  if (endDate < new Date()) {
    throw new UnprocessableEntityException(
      'End date must not be before today'
    )
  }

  const updatePayload = {
    endDate: new Date(body.endDate),
    'quota.batchDuration': body.batchDuration,
    'quota.liveDuration': body.liveDuration,
  }

  return this.subscriptionService.updateByUser(
    params.id, updatePayload
  )
}
```

Listing 5.8: Update Subscription Endpoint

5.4 Stripe Integration

The Stripe module handles payment processing through Stripe's API, including checkout session creation, webhook handling, and subscription lifecycle management.

5.4.1 Module Architecture

The Stripe module integrates with both Plan and Subscription modules:

- **StripeController**: Exposes payment endpoints and receives webhooks.

- **StripeService**: Wraps Stripe SDK with caching layer.
- **PlanService**: Provides plan data for webhook processing.
- **SubscriptionService**: Creates subscriptions from successful payments.

5.4.2 Checkout Session Flow

Figure 5.4 illustrates the complete checkout process from plan selection to subscription creation.

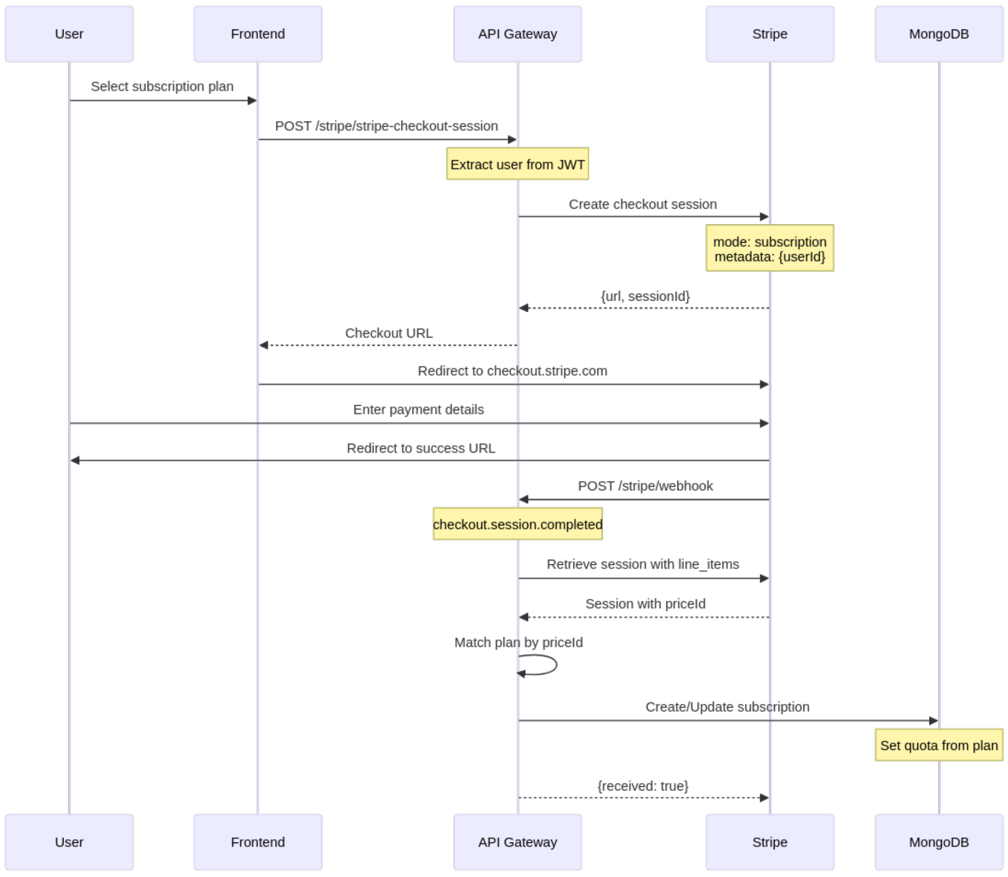


Figure 5.4: Stripe Checkout Session Flow

The checkout flow involves three parties:

1. **Frontend**: Initiates checkout and handles redirects.
2. **API Gateway**: Creates sessions and processes webhooks.
3. **Stripe**: Hosts the secure checkout page and sends events.

5.4.3 Implementation Details

Creating Checkout Sessions

```
async createCheckoutSession(
  priceId: string,
  customerEmail?: string,
  userId?: string,
): Promise<{ url: string; sessionId: string }> {
  const sessionParams: Stripe.Checkout.SessionCreateParams = {
    payment_method_types: ['card'],
    mode: 'subscription',
    line_items: [{ price: priceId, quantity: 1 }],
    success_url: this.configService.get('stripe.successUrl'),
    cancel_url: this.configService.get('stripe.cancelUrl'),
    billing_address_collection: 'required',
  }

  if (customerEmail) {
    sessionParams.customer_email = customerEmail
  }

  // Store userId for webhook processing
  if (userId) {
    sessionParams.metadata = { userId }
  }

  const session = await this.stripe.checkout.sessions.create(
    sessionParams
  )
  return { url: session.url, sessionId: session.id }
}
```

Listing 5.9: Checkout Session Creation

Key configuration parameters:

- `mode: 'subscription'`: Enables recurring billing.
- `metadata.userId`: Links the payment to the user's account.
- `success_url / cancel_url`: Redirect destinations after checkout.

Webhook Handler

```
@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)

      const priceId = fullSession.line_items?.data?.[0]
        ?.price?.id
      const userId = fullSession.metadata?.userId

      // Match plan and create subscription
      const latestPlans = await this.planService
        .getLatestVersion()
      const plansData = JSON.parse(latestPlans.plans)
      const matchingPlan = plansData.plans
        .find(p => p.priceId === priceId)

      await this.createOrUpdateSubscription(
        userId, matchingPlan, fullSession
      )
      break
  }

  return { received: true }
}
```

Listing 5.10: Webhook Processing

Event	Action
`checkout.session.completed`	Create or update subscription with quota
`invoice.payment_failed`	Log warning for monitoring

Table 5.7: Handled Webhook Events

5.4.4 Price Caching

Stripe price information is cached to reduce external API calls. Figure 5.5 illustrates the caching workflow with cache penetration protection.

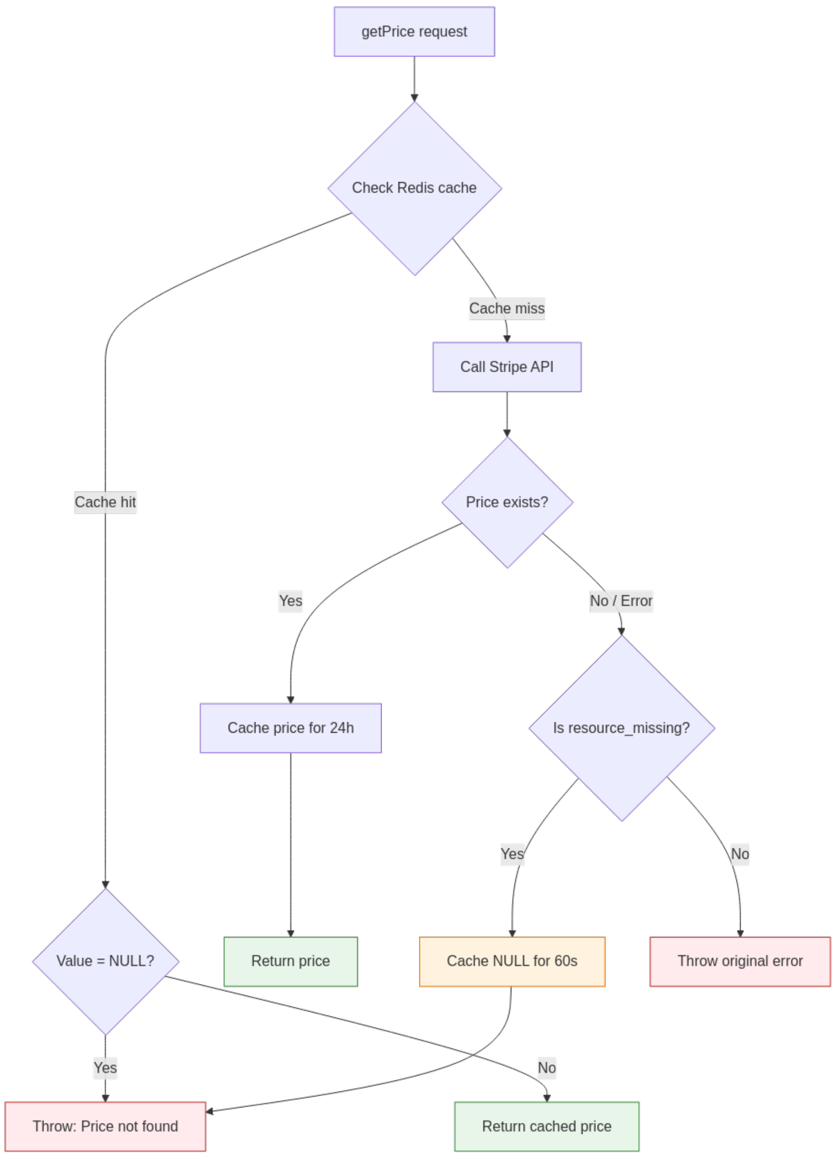


Figure 5.5: Stripe Price Caching Workflow

The caching strategy includes:

```
private readonly CACHE_TTL = 86400  // 24 hours

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

Listing 5.11: Price Caching with Penetration Protection

5.4.5 Subscription Lifecycle

The module provides endpoints for managing subscription status:

Cancel Subscription

Cancellation is scheduled for the end of the current billing period:

```
async cancelSubscription(subscriptionId: string) {
  return this.stripe.subscriptions.update(subscriptionId, {
    cancel_at_period_end: true,
  })
}
```

Listing 5.12: Cancel Subscription

Resume Subscription

A cancelled subscription can be resumed before the period ends:

```
async resumeSubscription(subscriptionId: string) {
  return this.stripe.subscriptions.update(subscriptionId, {
    cancel_at_period_end: false,
  })
}
```

Listing 5.13: Resume Subscription

5.4.6 Security Considerations

- **Webhook Signature Verification**: All webhooks are verified using the Stripe SDK's `constructEvent()` method with the webhook secret.
- **Raw Body Handling**: The webhook endpoint uses raw body parsing to preserve the payload for signature verification.
- **Idempotency**: The handler checks for existing subscriptions to ensure duplicate webhooks don't create duplicate records.

Chapter 6

Frontend Development

6.1 Overview

The Gateway Dashboard is a Vue.js 2 web application that provides a management interface for the Speech Gateway API. This chapter covers the frontend architecture, component design, and integration with the backend API.

6.1.1 Frontend Architecture

Figure 6.1 illustrates the layered architecture of the frontend application.

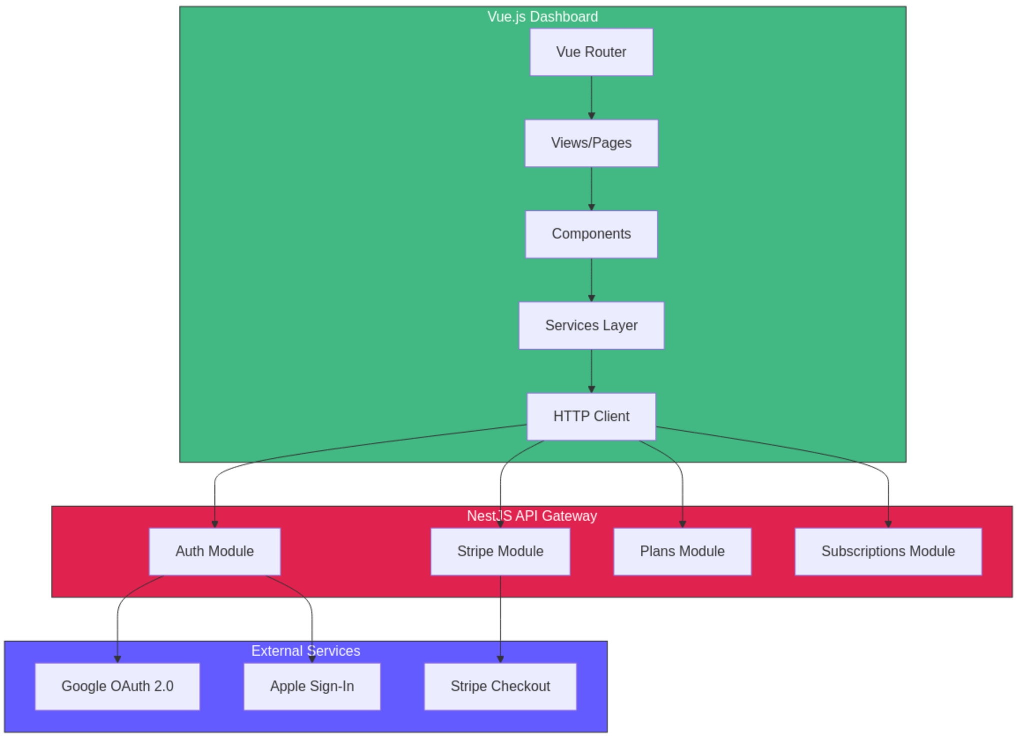


Figure 6.1: Frontend Application Architecture

The frontend follows a layered architecture:

- **Vue Router**: Manages navigation and route guards.
- **Views/Pages**: Page-level components for each route.

- **Components**: Reusable UI components like PlanCard.
- **Services Layer**: API communication logic.
- **HTTP Client**: Axios with interceptors for token handling.

6.1.2 Technology Stack

Layer	Technology	Purpose
Framework	Vue.js 2.x	Reactive UI framework
UI Library	Vuetify 2.x	Material design components
HTTP Client	Axios	API communication
Router	Vue Router	SPA navigation
State	Vue Instance + Mixins	Application state
Payment	Stripe Checkout	Subscription processing

Table 6.1: Frontend Technology Stack

6.1.3 Project Structure

```
gateway-dashboard/
  src/
    components/
      PlanCard.vue            # Subscription plan display
    layouts/
      Main.vue                # Main app layout
    mixins/
      auth.mixin.js           # Auth functionality mixin
    router/
      index.js                # Route definitions
      hooks.js                # Navigation guards
    services/
      auth.service.js         # Authentication API
      plans.service.js        # Plans API
      stripe.service.js       # Stripe integration
      user.service.js         # User/subscription API
    views/
      SignIn.vue              # Login page
      SignUp.vue              # Registration page
      CompleteSignup.vue      # OAuth completion
      Subscription.vue        # Subscription management
      admin/
        JsonPlansEditor.vue # Admin plans editor
    http.js                   # Axios HTTP client
    main.js                   # App entry point
```

Listing 6.1: Frontend Project Structure

6.1.4 Chapter Organization

This chapter is organized as follows:

- **Section 5.2**: Authentication frontend with OAuth integration
- **Section 5.3**: Routing and navigation guards
- **Section 5.4**: Subscription management interface

6.2 Authentication Frontend

The frontend authentication system supports three login methods: email/password, Google OAuth, and Apple Sign-In.

6.2.1 Auth Service Architecture

The `AuthService` manages all authentication operations and token storage.

Method	Description
`login()`	Email/password authentication
`register()`	New account registration
`initiateGoogleLogin()`	Redirect to Google OAuth
`initiateAppleLogin()`	Apple SDK sign-in flow
`refreshToken()`	Refresh expired JWT
`storeAuthData()`	Store token in localStorage
`clearAuthData()`	Clear token on logout

Table 6.2: Auth Service Methods

6.2.2 Token Storage

Authentication data is stored in localStorage for persistence:

```
{
  accessToken: "eyJhbGciOiJIUzI1NiIs...",
  jwt: "eyJhbGciOiJIUzI1NiIs...",   // backward compat
  subscriptionEnd: "1737244800000",  // Unix timestamp
  isVerified: "true"
}
```

Listing 6.2: LocalStorage Token Schema

6.2.3 Email/Password Login

```
async login({ email, password }) {
  const response = await http.post('/auth/login', {
    email, password
  })

  const { accessToken, subscriptionEnd, isVerified } =
    response.data

  this.storeAuthData(accessToken, subscriptionEnd, isVerified)
  return response.data
}
```

Listing 6.3: Login Implementation

6.2.4 OAuth Integration

Google OAuth

Google OAuth uses redirect-based authentication:

```
initiateGoogleLogin() {
  const frontendUrl = encodeURIComponent(window.location.origin)
  const redirectUrl = `${API_URL}/auth/google/login` +
    `?redirect=${frontendUrl}/sign-in`

  window.location.href = redirectUrl
}
```

Listing 6.4: Google OAuth Initiation

Apple Sign-In

Apple Sign-In uses the Apple SDK:

```
initializeAppleSignIn(clientId) {
  if (!window.AppleID) {
    console.warn('Apple SDK not loaded')
    return false
  }

  window.AppleID.auth.init({
    clientId: clientId || process.env.VUE_APP_APPLE_CLIENT_ID,
    scope: 'name email',
    redirectURI: `${API_URL}/auth/apple/callback`,
    usePopup: false
  })
  return true
}

async initiateAppleLogin() {
  await window.AppleID.auth.signIn()
}
```

Listing 6.5: Apple Sign-In Initialization

6.2.5 Complete Signup Flow

New OAuth users must complete signup by setting a password:

```
async setPassword(temporaryToken, password, fullname) {
  const response = await http.post('/auth/add-info', {
    token: temporaryToken,
    password,
    fullname
  })

  const { accessToken, subscriptionEnd, isVerified } =
    response.data
  this.storeAuthData(accessToken, subscriptionEnd, isVerified)

  localStorage.removeItem('temporaryToken')
  return response.data
}
```

Listing 6.6: Complete Signup Implementation

6.2.6 Auth Mixin

The auth mixin provides authentication state to components:

```
import { authMixin } from '@/mixins/auth.mixin'

export default {
  mixins: [authMixin],
  // Component now has access to:
  // - this.isAuthenticated
  // - this.accessToken
  // - this.subscriptionEnd
  // - this.logout()
  // - this.verifyAndRefreshToken()
}
```

Listing 6.7: Auth Mixin Usage

The mixin automatically starts token verification on mount and cleans up on destroy.

6.3 Routing and Navigation

Vue Router manages navigation with guards that enforce authentication and role-based access control.

6.3.1 Route Configuration

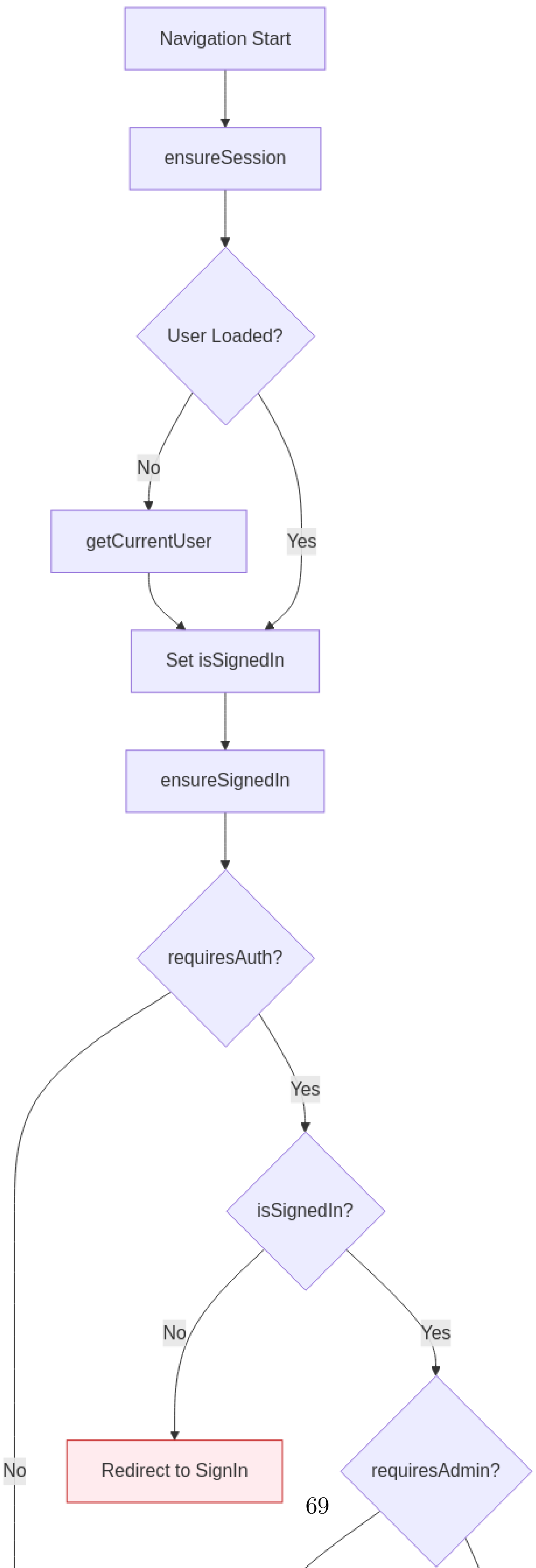
Routes are categorized by authentication requirements:

Path	**Auth**	**Admin**	**Component**
/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
/files	Yes	No	Files.vue
/applications	Yes	No	Applications.vue
/subscription	Yes	No	Subscription.vue
/dashboard	Yes	Yes	Dashboard.vue
/users	Yes	Yes	Users.vue
/request-log	Yes	Yes	RequestLog.vue
/settings	Yes	Yes	Settings.vue

Table 6.3: Route Definitions

6.3.2 Navigation Guards Flow

Figure 6.2 illustrates the navigation guard logic.


Navigation Start
ensureSession
User Loaded?
No
Yes
getCurrentUser
Set isSignedIn
ensureSignedIn
requiresAuth?
Yes
No
isSignedIn?
No
Yes
Redirect to SignIn
requiresAdmin?

6.3.3 Guard Implementation

Two navigation guards work together:

Session Guard

Ensures user data is loaded before navigation:

```
export const ensureSession = router => async (to, from, next) =>
 {
  await Vue.nextTick()

  if (!router.app.user && to.name !== 'SignIn') {
    const user = await router.app.getCurrentUser()
    to.meta.isSignedIn = !!user
  } else {
    to.meta.isSignedIn = router.app.user !== null
  }
  next()
}
```

Listing 6.8: Session Guard

Auth Guard

Enforces authentication and admin requirements:

```
export const ensureSignedIn = router => (to, from, next) => {
  const requiresAdmin = anyTrue(to, 'requiresAdmin')
  const requiresAuth = requiresAdmin || anyTrue(to, 'requiresAuth
     ')

  if (requiresAuth) {
    if (!to.meta.isSignedIn) {
      next({ name: 'SignIn' })
    } else if (requiresAdmin &&
               router.app.user.role !== 'admin') {
      next({ name: 'Speeches' })
    } else {
      next()
    }
  } else {
    next()
  }
}
```

Listing 6.9: Auth Guard

6.3.4 HTTP Interceptors

Figure 6.3 shows the HTTP interceptor flow for automatic token handling.

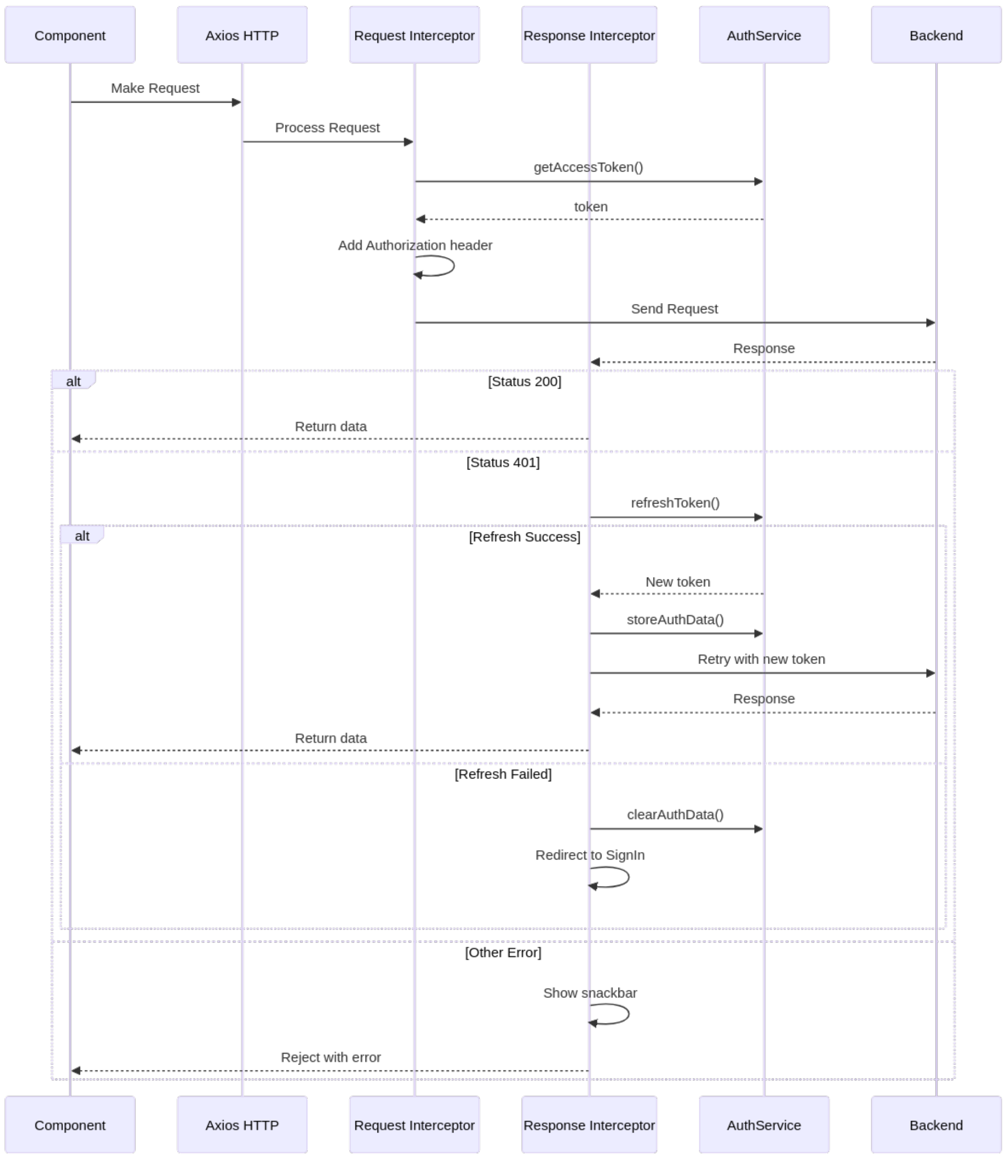


Figure 6.3: HTTP Interceptor Flow with Token Refresh

The interceptors handle:

- **Request**: Automatically attach Authorization header with JWT token.
- **Response 401**: Attempt token refresh; if failed, redirect to login.
- **Other Errors**: Display error message in snackbar.

6.4 Subscription Management Interface

The subscription interface provides plan selection, payment processing, and quota monitoring.

6.4.1 Component Architecture

The subscription system consists of interconnected components:

- **Subscription.vue**: Main page with conditional views for users and admins.
- **PlanCard.vue**: Displays plan details with dynamic pricing from Stripe.
- **JsonPlansEditor.vue**: Admin-only JSON editor for plan management.

6.4.2 User Subscription View

For users with active subscriptions, the view displays:

- Subscription period (start/end dates)
- Quota allocation (batch and live processing)
- Usage progress with color-coded indicators
- Cancel/Resume subscription controls

```
calculateUsagePercent(used, quota) {
  if (quota === -1) return 0      // Unlimited
  if (quota === 0) return 100
  return Math.min((used / quota) * 100, 100)
}

getUsageColor(used, quota) {
  if (quota === -1) return 'success'
  const percent = (used / quota) * 100
  if (percent >= 90) return 'error'
  if (percent >= 75) return 'warning'
  return 'success'
}
```

Listing 6.10: Usage Percentage Calculation

6.4.3 PlanCard Component

The PlanCard fetches pricing from Stripe and displays plan features:

Prop	Type	Description
plan	Object	Plan data (id, name, features, priceId)
disabled	Boolean	Disable subscribe button (for admins)

Table 6.4: PlanCard Component Props

```
async fetchPriceInfo() {
  if (!this.plan.priceId) return

  try {
    this.loadingPrice = true
    this.priceInfo = await stripeService.getPriceInfo(
      this.plan.priceId
    )
  } catch (error) {
    this.priceError = 'Failed to load price'
  } finally {
    this.loadingPrice = false
  }
}

// Computed property for display
get formattedPrice() {
  if (!this.priceInfo) return 'Loading...'
  const amount = this.priceInfo.unit_amount / 100
  return `$${amount.toFixed(2)}`
}
```

Listing 6.11: PlanCard Price Fetching

6.4.4 Stripe Checkout Integration

When a user subscribes, the frontend initiates Stripe Checkout:

```
async handleSubscribe(priceId) {
  try {
    const data = await stripeService.createCheckoutSession(
      priceId
    )

    if (data?.url) {
      window.location.href = data.url
    } else {
      this.showError('Failed to redirect. Please try again.')
    }
  } catch (err) {
    console.error('Subscription error', err)
    this.showError('An error occurred during checkout.')
  }
}
```

Listing 6.12: Checkout Session Creation

6.4.5 Cancel and Resume Subscription

Users can manage their subscription lifecycle:

```
async handleCancelSubscription() {
  if (!confirm('Are you sure you want to cancel?')) {
    return
  }

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

    this.showSuccess('Subscription scheduled for cancellation.')
  } catch (err) {
    this.showError('Failed to cancel subscription.')
  } finally {
    this.cancellingSubscription = false
  }
}
```

Listing 6.13: Cancel Subscription

6.4.6 Admin Plans Editor

Administrators access a JSON editor for managing plans:

- Live JSON validation with syntax highlighting
- Stripe price ID validation before publishing
- Diff comparison with current version
- Version history tracking

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

Chapter 7

Conclusion

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

7.1 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:

7.1.1 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.

7.1.2 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.

7.1.3 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.

7.2 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.

7.3 Future Work

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

7.3.1 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.

7.3.2 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.

7.3.3 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.

7.4 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

Proof

A long proof that you expect no one will read