GETA PRIME – DOCUMENTATION

Overview

Geta Prime is a subscription-based platform designed to manage subscriptions and deliver features according to the description plan. It offers tiered plans (Free, Standard, Premium) where each tier unlocks specific features dynamically through a database-driven access control system. Built with Next.js (frontend) and Node.js along Express.js with TypeORM (backend), it ensures scalability and flexibility for businesses.

Feature Planning

- Subscription Tiers
  1. Free Tier
     - 5 core features available
     - Acts as the entry point for new users
     - Example: Dashboard access, Profile management, Notifications
  2. Standard Tier
     - 18 features (includes all Free tier features)
     - Mid-level plan for regular users
     - Example: Advanced analytics, File uploads, Export reports, API access
  3. Premium Tier
     - 33 features (includes all Standard tier features)
     - Full access to the platform for power/business users
     - Example: Custom integrations, Priority support, Unlimited storage, Team managemen

  Feature Categorization

  - **Core** → Always visible (some locked for lower tiers)
  - **Analytics** → Reports, charts, insights
  - **Collaboration** → Team features, sharing
  - **Integrations** → APIs, third-party apps
  - **Communication** → APIs, Live Chat

    And many more categories

Project Architecture

The architecture of **Geta Prime** follows a **modular, service-oriented design** that ensures scalability, security, and maintainability as the platform grows. The system is primarily divided into three layers: **Frontend (React.js/Next.js), Backend (Node.js + Express), and Database (PostgreSQL with TypeORM)**, with external integrations for payments and notifications.

1. System Design & Components

- **Frontend (React.js + Next.js)**
  - Handles user interaction and provides a seamless subscription management UI.
  - Uses **Axios interceptors** to manage API calls, automatically refreshing JWT tokens when expired (5-minute lifetime for access tokens).
  - Implements tier-based feature accessibility so that users only access features allowed by their subscription.
- **Backend (Node.js + Express)**
  - Exposes REST APIs for authentication, subscription management, and feature access.
  - Uses **JWT for authentication** with **short-lived access tokens (5 minutes)** and **refresh tokens** for reissuing new access tokens.
  - Includes **middlewares**:
    - **Authentication Middleware** → Validates tokens.
    - **Feature Access Middleware** → Checks if the user's tier allows access to the requested feature and Verifies `subscription_end` date, updates DB if expired, and auto-downgrades the user to the **Free Tier**.
- **Database (PostgreSQL with TypeORM)**
  - Stores users, tiers, features, and tier-feature mappings.
  - Enforces relationships through foreign keys (e.g., `tier_id` in users).
  - Enables scalability with indexing and optimized queries.
- **External Integrations**
  - **Razorpay** → Used for secure subscription payments and tier upgrades/downgrades. Payment events trigger backend APIs to update the database.

- **Nodemailer** → Sends **email notifications** to users upon successful subscription/tier change.

2. Data Flow

1. **User Authentication & Session Management**
    a. User logs in → Backend validates credentials (hashed with bcrypt) → Issues **JWT access token (5 min)** + **refresh token**.
    b. Frontend stores tokens securely → Axios interceptors automatically refresh access tokens on `401 Unauthorized`.
2. **Feature Access & Validation**
    a. User requests a feature → Backend middleware checks:
        i. If JWT is valid.
        ii. If user's subscription tier allows access (via `tier_features` mapping).
        iii. If `subscription_end` date is valid; if expired → DB is updated, tier auto-changes to Free, and access is denied.
3. **Subscription & Payment Handling**
    a. User selects a tier → Razorpay payment gateway is triggered.
    b. On successful payment → Razorpay webhook updates backend → Database updates `tier_id`, `subscription_start`, and `subscription_end`.
    c. Nodemailer sends a confirmation email notifying the user about the new tier.
4. **Automatic Downgrade**
    a. If a subscription expires, the **Subscription Validation Middleware** ensures the tier is downgraded to Free during the next request automatically, keeping data consistent.

Technology Stack Justification

Frontend

- **Next.js with TypeScript**: Chosen for its **server-side rendering (SSR)** and **static site generation (SSG)** capabilities, which improve SEO and performance for dynamic content like subscription plans and leads. TypeScript ensures **type safety, scalability, and maintainability**, reducing runtime errors in a growing codebase.

- **Tailwind CSS**: A utility-first CSS framework that enables **fast, consistent, and responsive UI development** without writing repetitive CSS. It ensures a clean design system across components and accelerates front-end development.
- **Razorpay (Frontend Integration)**: Used for handling subscription payments with a **seamless checkout experience**. Razorpay provides ready-to-use SDKs for the frontend, ensuring **secure, PCI-compliant transactions** without the burden of managing sensitive payment data.

Backend

- **Node.js + Express (without TypeScript)**: Selected for its **non-blocking, event-driven architecture**, ideal for handling concurrent requests like user subscriptions, email notification, etc. Express is lightweight and flexible, making it easy to build and scale APIs quickly.
- **PostgreSQL**: A powerful relational database chosen for its **ACID compliance, reliability, and support for complex queries**. It ensures data integrity across critical modules like users, subscriptions, tiers, and activities.
- **TypeORM**: Provides an **Object-Relational Mapping (ORM)** layer, simplifying database interaction by allowing developers to work with entities instead of raw SQL. It enhances productivity and maintains schema consistency while still supporting advanced PostgreSQL features.
- **Nodemailer**: Integrated for **backend email notifications**, ensuring user gets an email notification on successful tier change. It allows SMTP and OAuth2 support, making it versatile for Gmail and other providers.

This stack balances **scalability, maintainability, and developer productivity**. The frontend leverages Next.js and Tailwind for performance and clean UI, while the backend relies on Node.js with Express for flexibility and PostgreSQL with TypeORM for robust data handling. The addition of **Nodemailer** for email notifications and **Razorpay** for secure payments makes Geta Prime a **full-featured, production-ready SaaS application**.

Database Architecture

The database for **Geta Prime** is designed using **PostgreSQL** with **TypeORM** to ensure relational integrity, scalability, and ease of querying. The schema follows a modular design, separating users, subscription tiers, and features into distinct entities, while using a join table for flexible many-to-many relationships between tiers and features. This

ensures that subscription plans and feature access remain dynamic and easily extendable.

Schema Design

1. **Users Table**
    a. **Columns**:
        i. `id` (Primary Key)
        ii. `username` (Unique identifier for each user)
        iii. `email` (Unique, for login/communication)
        iv. `password` (Securely hashed for authentication)
        v. `tier_id` (Foreign Key → `tiers.id`, defines the user's active subscription tier)
        vi. `subscription_start` (Timestamp, when subscription begins)
        vii. `subscription_end` (Timestamp, when subscription ends)
        viii. `created_at` (Timestamp, when user is created)
        ix. `updated_at` (Timestamp, when user is last updated)

    b. **Justification**: Stores user information and directly links each user to a subscription tier, enabling feature access validation.
2. **Tiers Table**
    a. **Columns**:
        i. `id` (Primary Key)
        ii. `name` (e.g., Free, Standard, Premium)
        iii. `price` (Monetary value for subscription)
        iv. `description` (Tier-specific details)
        v. `created_at` timestamp tier created
    b. **Justification**: Defines the available subscription tiers. A separate table allows adding new tiers or modifying pricing without affecting other entities.
3. **Features Table**
    a. **Columns**:
        i. `id` (Primary Key)
        ii. `name` (Feature title, e.g., "Email Integration")
        iii. `description` (Details of what the feature does)
        iv. `category` (Logical grouping, e.g., Communication, Analytics)
        v. `created_at` timestamp when feature created

    b. **Justification**: Keeps features decoupled from tiers, ensuring they can be reused across multiple subscription levels and grouped logically for better organization.
4. **Tier_Features (Join Table)**
    a. **Columns**:
        i. `tier_id` (Foreign Key → `tiers.id`)
        ii. `feature_id` (Foreign Key → `features.id`)
    b. **Justification**: Implements a many-to-many relationship, allowing multiple features per tier and multiple tiers to share the same feature. This provides maximum flexibility in configuring subscription plans without restructuring the schema.

Indexing Strategy

- Index on `users.tier_id` for fast tier lookup
- Index on `tier_features.tier_id` + `tier_features.feature_id` composite key
- Index on `features.category` for fast feature grouping

Scalability Strategy

The **Geta Prime** system is designed with scalability in mind, ensuring that it can handle increasing numbers of users, features, and overall system load as the platform grows.

1. **Database Scalability**
    a. The schema is normalized with clear relationships between **Users**, **Tiers**, and **Features**, reducing redundancy and improving query performance.
    b. Indexing on frequently queried fields (e.g., `email`, `tier_id`) ensures faster lookups.
    c. Horizontal scaling can be achieved using **read replicas** in PostgreSQL, while vertical scaling ensures performance improvements when required.
2. **Application Scalability**
    a. The backend (Node.js + Express) follows a modular service architecture, allowing new features or modules to be added without affecting existing functionality.
    b. **TypeORM** ensures database operations remain abstracted, making it easier to migrate or expand the database structure as features grow.

    c. Load balancing with multiple Node.js instances can be introduced to handle concurrent user traffic.
3. **Frontend Scalability**
    a. The frontend, built with **Next.js** and **TypeScript**, supports **server-side rendering (SSR)** and **static site generation (SSG)** for performance optimization under high traffic.
    NOTE: For the sake of simplicity, most of the components are CSR.
    b. Tailwind CSS provides a utility-first approach, allowing scalable UI development as more features and pages are added.
4. **Feature and Subscription Growth**
    a. The **tier-feature mapping** ensures that new features can be added dynamically without restructuring existing tiers.
    b. This flexibility allows the system to expand offerings without downtime or major schema changes.
5. **Infrastructure and Load Handling**
    a. Caching mechanisms (e.g., Redis) can be integrated to reduce database load for frequently accessed data such as user tiers and feature availability.
    b. As user volume increases, the system can be containerized using **Docker** and orchestrated with **Kubernetes** for elastic scaling.
    c. CDN (Content Delivery Network) integration ensures faster content delivery to global users.

In summary, the system is designed to handle growth gracefully by separating concerns at the database, application, and infrastructure levels, ensuring reliability and performance even as the number of users, features, and requests increase

Cost Optimization

To ensure **Geta Prime** remains cost-effective while scaling, the system incorporates several strategies to optimize resource usage:

1. **Efficient Database Design**
    a. Normalized schema with join tables (e.g., `tier_features`) avoids data duplication.
    b. Indexing on frequently queried fields (e.g., `email`, `tier_id`) improves query performance, reducing unnecessary compute usage.
2. **Dynamic Resource Allocation**

    a. Horizontal scaling on demand (e.g., auto-scaling backend servers during peak loads).
    b. Serverless components for background tasks (e.g., email notifications, subscription renewal reminders) to reduce always-on server costs.
3. **Tiered Feature Access**
    a. Resource-intensive features are only available for higher-tier users, ensuring free-tier users do not disproportionately consume server resources.
4. **Third-Party Service Optimization**
    a. Nodemailer is configured with transactional email services (e.g., Gmail API or SendGrid) to minimize infrastructure overhead.
    b. Razorpay ensures cost-effective payment processing with minimal overhead and reliable scaling.

Security Considerations

Ensuring security in a **subscription-based feature access system** is critical for protecting sensitive data, enforcing correct access control, and maintaining trust. The following strategies and implementations are applied in **Geta Prime**:

1. Authentication

- **JWT (JSON Web Tokens):**
    - Access tokens are issued upon successful login for stateless authentication.
    - Tokens are signed with a secure secret key and have short expiry times to reduce risks.
- **Refresh Tokens:**
    - Long-lived refresh tokens are used to generate new access tokens without re-authentication.
    - Stored securely (e.g., in HttpOnly cookies) to minimize exposure to XSS attacks.
- **Password Hashing:**
    - User passwords are hashed using **bcrypt** before storage.
    - Salting ensures uniqueness and prevents rainbow table attacks.

2. Authorization

- **Tier-Based Access Control:**
  - Each user is assigned a **subscription tier** (`Free`, `Standard`, `Premium`) which determines feature accessibility.
  - Authorization middleware verifies if the user's tier allows access to a requested feature.
- **Middleware for Feature Access:**
  - A centralized middleware checks if the feature being accessed is included in the user's tier.
  - Denies access with a `403 Forbidden` response if not allowed.
- **Subscription Expiry Validation:**
  - Middleware verifies the **subscription end date** before granting access.
  - Users with expired subscriptions are restricted and prompted to renew.

3. Data Protection

- **Input Validation:**
  - Client-side validation ensures only properly formatted data is submitted.
  - Prevents malformed requests from reaching the server.
- **Backend Validation (using Joi):**
  - All incoming API requests are validated against strict Joi schemas.
  - Helps prevent injection attacks, malformed payloads, and logic abuse.
- **Sensitive Data Handling:**
  - Personally Identifiable Information (PII) like email addresses is encrypted in transit (via HTTPS/TLS).
  - Tokens and secrets are stored securely using environment variables.

4. Best Practices Implemented/Futuristic

- **Rate Limiting & Brute Force Protection:** Limits excessive login attempts.
- **CORS Policies:** Restricts requests only from trusted domains.
- **Secure Session Management:** Uses HttpOnly and Secure flags for cookies.
- **Audit Logging:** Tracks subscription changes, logins, and failed access attempts.

Deployment Strategy (Planning)

The deployment strategy for **Geta Prime** is designed to ensure smooth delivery, scalability, and maintainability of the subscription-based platform as user demand grows.

1. Deployment Environment

- **Frontend (Next.js + TypeScript + Tailwind CSS)**
    - Deployed on **Vercel** for optimized Next.js hosting with automatic scaling, CDN-backed asset delivery, and built-in CI/CD pipeline.
    - This ensures low latency for global users and frictionless deployments.
- **Backend (Node.js + Express + PostgreSQL + TypeORM)**
    - Containerized using **Docker** to maintain consistency across environments.
    - Deployed on **AWS Elastic Beanstalk** or **AWS ECS (Elastic Container Service)** for easy scaling and load balancing.
    - PostgreSQL hosted on **AWS RDS** (managed database service) for reliability, backups, and automated scaling.
- **Payment & Notifications**
    - **Razorpay** integrated in the frontend for secure subscription payments.
    - **Nodemailer** in backend for transactional email notifications, configured with a reliable SMTP service (e.g., AWS SES or Gmail API).

2. CI/CD Pipeline

- **GitHub Actions** integrated with Vercel (for frontend) and AWS (for backend).
- Automatic builds and deployments triggered on code merges to the main branch.
- Linting, testing, and TypeORM migrations run in the pipeline before deployment to maintain system stability.

3. Scaling Strategy

- **Frontend**: Vercel auto-scales with incoming traffic. Edge network caching ensures minimal latency.
- **Backend**:
    - Horizontal scaling with AWS ECS or Elastic Beanstalk, managed by **Auto Scaling Groups**.
    - **Load Balancer** distributes requests efficiently.
- **Database**:

    - PostgreSQL on AWS RDS with read replicas for scaling read-heavy operations.
    - Connection pooling with **pgbouncer** to optimize database connections under load.

4. Monitoring & Logging

- **AWS CloudWatch** for backend performance monitoring, logs, and alerts.
- **Vercel Analytics** for frontend performance metrics.
- **Sentry** or **LogRocket** for error tracking and debugging.

5. Maintenance

- Regular **database migrations** managed by TypeORM.
- Scheduled **backups** via AWS RDS snapshot automation.
- Rolling updates for backend services to avoid downtime.
- Dependency updates and vulnerability scanning using **Dependabot** and **npm audit**.