ChatGPT

Web Application System Design

The proposed web app will have a React-based frontend and a Python-based backend, connected via RESTful APIs. It will allow users to create an account (via email/password or social OAuth), then save and manage multiple "plans." Unauthenticated visitors will be rate-limited to 10 free requests per month, after which they must sign up. We'll use free/freemium external APIs where needed (e.g. mapping, data lookup). This document outlines a comprehensive architecture, database schema, and detailed design for both frontend and backend, with an initial focus on implementing a secure login system. All technical decisions prioritize free tiers and open-source solutions wherever possible [1] [2].

Architecture Overview

The system follows a classic client-server model [3]. The **React frontend** (client) runs as a single-page application (potentially a Progressive Web App) that performs all user interactions. It communicates with the **Python backend** (server) through HTTP(S) API calls [4] [5]. The backend exposes endpoints for authentication, plan management, and other business logic. A reverse proxy or web server (e.g. Nginx) can sit in front of the backend for SSL termination, load balancing, and rate limiting [6]. Data is stored in a relational database (e.g. PostgreSQL) and optionally cached (e.g. Redis) for sessions or frequent queries [7] [8]. This separation of concerns (client/UI vs. server/API vs. database) simplifies development and maintenance [3].

Figure: Example client-server request flow. A React client (browser) issues an HTTP request (after DNS lookup) and receives a response from the Python server, following a typical web browsing sequence. Our app uses this pattern: the frontend sends REST API calls to the backend and processes the JSON responses.

Key Components

- **Frontend (React):** A JavaScript-based single-page app (SPA). We can use React (with state management like Redux or Context) to build interactive UIs. React is ideal for building reusable UI components and managing client-side routing [9]. All API calls (e.g. login, create plan, fetch plans) will use HTTP libraries like Axios or Fetch [5]. The UI will handle login/signup flows and display the user's saved plans.

- **Backend (Python):** A RESTful API server. Options include Django (with Django REST Framework) or Flask/FastAPI. Django provides a robust ecosystem with built-in ORM, authentication, and admin interfaces [10], while Flask/Flask-RESTful offers lightweight flexibility [4]. The backend will implement endpoints for user management (sign-up, login, profile), social OAuth callbacks, and plan CRUD (create, read, update, delete). Business logic (like usage limits) and external API calls will also reside here.

- **Database:** A SQL database (e.g. PostgreSQL [11]) to persist user accounts and plans. For example, a `users` table (id, email, password_hash, etc.) and a `plans` table (id, user_id foreign key, plan_data, timestamps). Each user can have many plans (one-to-many relation). We will define clear models/

schemas and run migrations when they change. We may also use Redis or similar to cache session data or rate-limit counters [8] [6].

- **Authentication & Sessions:** Passwords are stored hashed (e.g. bcrypt) [12]. We will use token-based authentication: on login the server issues a JSON Web Token (JWT) [13] or session token. The frontend stores this token (e.g. in an HttpOnly cookie or memory) and includes it in subsequent API requests. JWTs are signed to prevent tampering [13]. For social login, we implement OAuth2 flows (e.g. Google, Facebook). Python libraries (such as **drf-social-oauth2** or **python-social-auth**) can be used to integrate major social providers [14].

- **Rate Limiting:** To enforce "10 free requests/month" for anonymous users, we'll track requests per IP or per session in a fast store (e.g. in-memory Redis or a simple database counter). IP-based rate limiting is simple and can be enforced at the web server level (using Nginx's `limit_req` module) [6]. Alternatively, services like Cloudflare (free tier) offer basic DDoS and rate-limit features [15]. After the limit is reached, anonymous users must authenticate to continue using the service.

Frontend (React) Details

- **Application Structure:** Use a standard React setup (Create React App, Vite, or Next.js in SPA mode). Implement pages/components for:
- **Home/Landing:** Explains the app, with login/signup buttons.
- **Login/Signup Screens:** Forms for email/password signup and login, plus "Login with Google" (OAuth) button.
- **Plan Dashboard:** After login, list of saved plans (fetched via API), with options to create/edit/delete plans.
- **Create/Edit Plan Screen:** Form to specify plan details (text, options, etc.).
- **State Management:** We can use React Context or Redux. Actions (e.g. `login`, `fetchPlans`, `createPlan`) will make API calls using Axios [5] and update state. For example, on successful login the action saves the JWT in context; on fetching plans the response is stored in state.
- **Communication:** All operations use REST endpoints. For example, `POST /api/login` with credentials, `GET /api/plans` to get a user's plans, etc. Axios calls are straightforward:

```
axios.post('/api/login/', {email, password})
     .then(res => {/* store token and user info */})
axios.get('/api/plans/', { headers: { Authorization: `Bearer ${token}` } })
```

The sequence of Redux actions (REQUEST, SUCCESS, FAIL) manages loading states and errors [5].
- **PWA & Routing:** We can configure the React app as a Progressive Web App (as a bonus) so it installs on mobile. React Router will handle client-side navigation between screens.
- **UI/UX Libraries:** Use free UI frameworks (Bootstrap, Material UI, Tailwind CSS) to speed up development. Ensure forms validate input (e.g. email format, password strength) before sending to the backend.

Backend (Python) Details

- **Framework Choice:** Django REST Framework (DRF) or Flask/Flask-RESTful. Django/DRF is recommended for rapid development with built-in features [10] . For example:
- **Django**: Has a built-in user model, admin console, and middleware. We can use *Django REST Framework* to create serializers and viewsets for our models (User, Plan).
- **Flask**: We'd use Flask-RESTful or FastAPI to declare endpoints. Flask requires more setup (e.g. manual auth implementation) [4] .
- **Database Models:** For Django, define a `User` model (with email as username, password hash, etc.) and a `Plan` model with a ForeignKey to User. For Flask, use SQLAlchemy to model these tables. Enforce one-to-many (one user → many plans). Each plan record might store JSON data or structured fields describing the plan.
- **Authentication Endpoints:**
- `POST /api/signup` – create new user (hash password, save to DB).
- `POST /api/login` – verify credentials, return JWT.
- `GET /api/user` – (authenticated) return user profile info.
- `POST /api/logout` – (if using server-side sessions, else front-end just discards token). Use libraries like **Djoser** or **django-rest-auth** to quickly scaffold these endpoints if on Django. These libraries handle signup, login, password reset, etc.
- **Social Login (OAuth2):** Use a package such as **django-oauth-toolkit** for OAuth2 support [16] and **drf-social-oauth2** for social providers [14] . The flow: the frontend hits a login URL (e.g. `GET /api/login/google`), the user authenticates with Google, and Google redirects back to our backend with a code. The backend exchanges the code for a token and creates/updates the corresponding User in our database, then returns our own JWT to the client.
- **Token Handling:** We'll use JWTs (JSON Web Tokens) for stateless auth. Upon successful login (email or social), the backend issues a signed JWT containing the user's ID and other claims [13] . On each protected API call, the frontend sends this token in the Authorization header. The backend verifies the signature and extracts the user ID. Because JWTs are signed (not encrypted), the backend must use a secure secret key to sign tokens [13] . We must also guard against common JWT pitfalls (use HTTPS, short token expiry, refresh tokens if needed [13]).
- **Authorization:** Only authenticated users can access their own plans. Implement permission checks so that, for example, `GET /api/plans/{id}` ensures the requested plan belongs to the logged-in user. In DRF, this can be done with object-level permissions. For Flask, manually check `plan.user_id == current_user.id`.
- **Input Validation & Security:** All user inputs (including plan data) must be validated and sanitized. Use HTTPS for all traffic to protect credentials and tokens. Hash passwords (bcrypt/argon2) before storage [12] . Store tokens securely (cookies with HttpOnly or secure local storage). Implement rate-limiting (see below) to prevent brute-force attacks.

Database Schema Design

A simple relational design can suffice. Key tables:

- **users**: `(id, email, password_hash, name, created_at, ...)`. Email is unique. The password is stored in hashed form [12] . We'll use user `id` as the primary key.

- **plans**: `(id, user_id, title, details, created_at, updated_at, ...)`. Each plan row references `users.id` (one-to-many). It can store arbitrary plan data (text, JSON blob, etc.). We index on `user_id` to query a user's plans.
- **sessions / rate_limits** (optional): If using server-side sessions or tracking anonymous usage, we might have a small table like `anon_usage(ip_address, requests_this_month, last_reset)` or use Redis.

By normalizing, we ensure each plan belongs to exactly one user. To let users have multiple plans, we simply insert multiple rows in `plans` for the same `user_id`. This is a standard one-to-many relationship in SQL. We should also design for scalability: for large data, we can partition tables or use read replicas. But for now, one database (e.g. PostgreSQL) is fine [7] [11].

Authentication & Authorization Flow

1. **Signup/Login (Email):** Frontend posts credentials to the backend. The backend checks the credentials, creates a session or returns a JWT. The password in the database is hashed for security [12]. Example flow: user enters email/password on the signup form → `POST /api/signup` → backend saves new user → user is automatically logged in or redirected to login page. On login, `POST /api/login` returns a JWT.
2. **Social OAuth:** The user clicks "Login with Google/Facebook." The browser is redirected to the provider; after authorization, the provider redirects back to our backend with a code. We use a library (e.g. **drf-social-oauth2** [14]) to handle this exchange. We either create a new user or match an existing user by email, then issue our JWT to the frontend.
3. **Storing the Token:** The React app stores the received JWT (in memory or secure storage) and attaches it to all subsequent API requests. For convenience and security, we might use HTTP-only cookies (set by the backend) to avoid manual token handling in JS.
4. **Protected Endpoints:** The backend middleware checks incoming requests for a valid JWT (or token). If missing/invalid, return HTTP 401. Otherwise, allow the request. Thus endpoints like `/api/plans` can identify the user and only return that user's data.
5. **Rate-Limiting Anonymous Users:** For endpoints accessible without login (if any are open), we enforce a 10-requests/month limit per user/IP. After the 10th request, the backend should return a 403 or a message prompting login. We can implement this by tracking counts (e.g. in Redis with key = IP address) and resetting monthly. As a simpler solution, we could require login before any usage beyond 10 requests.

Throughout, standard security measures apply: use SSL/TLS for all transport, set CORS to allow requests only from our frontend domain, and log all auth events. We may also implement CSRF protection for session-based flows.

Free-Tier APIs and External Integrations

Any external service we use must offer a free tier. Examples by category:

- **Geolocation/Maps:** Use **OpenStreetMap** data and libraries like Leaflet (open-source) or Mapbox (free tier limited to ~50k requests/month) for maps and geocoding [17]. OSM is free with no usage limits [18].

- **Directions/Routes:** Free APIs include GraphHopper or OpenRouteService which have free tiers. These allow routing or distance calculations without cost for low usage.
- **Weather:** OpenWeatherMap and Weatherbit.io offer free tiers (a few thousand calls per month). They require API keys but have free limits.
- **Data/APIs:** A variety of APIs exist on the Public APIs list [2]. For example, if the app needs currency conversion, use Exchange Rates API (free tier), or for location info use Geonames or Nominatim (free OSM-based geocoder) [2].
- **Authentication (Social):** Google and Facebook OAuth have free usage for basic auth flows. We only use OAuth, which is free.
- **Others (as needed):** If we need AI or text-generation, options like OpenAI's free trial are limited, so we might skip heavy AI features or use smaller open models (e.g. via Hugging Face, which may offer a free API for research).

We will carefully check each API's pricing to ensure the expected usage (likely low scale) stays within the free tier. For example, Mapbox's free tier is enough for small apps (50k map loads) [19]. All chosen APIs should have generous free quotas.

Data Flow and System Design Principles

By following client-server separation, we adhere to best practices: each layer (client UI, server logic, database) is modular [3]. REST (HTTP/HTTPS) is our communication protocol [20]. The database schema and API contract define clear interfaces. React's virtual DOM and state updates ensure an efficient frontend, while the backend can scale independently. We'll use common tools: Git for version control, Docker for environment consistency, and possibly Kubernetes or cloud services for deployment. Caching (Redis or CDN) can improve performance, and load balancers (Nginx) or cloud WAFs (e.g. Cloudflare, AWS WAF) can protect against heavy loads or abuse [15].

Throughout, security (auth+encryption) and maintainability (clean code, docs) are priorities. Regular backups of the database and automated tests for auth flows and APIs will ensure reliability. The app will initially be a web application, but by cleanly separating frontend and API layers, the backend could later support a mobile app without major changes.

In summary, this design uses React for the UI and Python (Django/Flask) for the API, with a PostgreSQL database. It includes a robust authentication system (email/password + social OAuth) and limits anonymous usage to 10 requests/month via rate limiting [21] [6]. Users can save multiple plans in the database. All external services (maps, weather, etc.) will be free-tier APIs [18] [2]. This architecture is modular and scalable, enabling parallel frontend/backend development and a clear path to future expansion (e.g. mobile apps).

Sources: Authoritative guides on full-stack architecture, REST design, and authentication were used to inform this plan [22] [23] [5] [21] [13] [14]. Each cited source supports a core decision (e.g. using REST APIs [4], using JWT auth [13], using free/open APIs [18] [2]).

---

[1] Build an Authentication System Using Django, React and Tailwind - GeeksforGeeks
https://www.geeksforgeeks.org/python/build-an-authentication-system-using-django-react-and-tailwind/

2 GitHub - public-apis/public-apis: A collective list of free APIs
https://github.com/public-apis/public-apis

3 9 10 11 20 23 Client-Server Architecture - System Design - GeeksforGeeks
https://www.geeksforgeeks.org/system-design/client-server-architecture-system-design/

4 7 8 12 22 Full System Architecture of my React-Flask App
https://amlanscloud.com/apparchitecture/

5 Understanding the data flow for full stack development with Django REST and React | by Damian Stone | Medium
https://damianztone.medium.com/understanding-the-data-flow-for-full-stack-development-with-django-rest-and-react-3d27cf362fc4

6 15 21 load balancing - Rate limiting *un*-authenticated requests - Software Engineering Stack Exchange
https://softwareengineering.stackexchange.com/questions/384494/rate-limiting-un-authenticated-requests

13 JWT authentication: Best practices and when to use it - LogRocket Blog
https://blog.logrocket.com/jwt-authentication-best-practices/

14 16 Authentication - Django REST framework
https://www.django-rest-framework.org/api-guide/authentication/

17 18 19 10 Free Google Map API Alternatives | Webzstore Solutions
https://www.webzstore.com/10-free-google-map-api-alternatives/