Event Manager

Table of Contents

[1 Purpose 2](#_Toc189486867)

[2 Operating environment/High Level Design 2](#_Toc189486868)

[3 High level plan 3](#_Toc189486869)

[4 Phase Summary 3](#_Toc189486870)

[4.1 Phase 1: Core Environment Setup & Basic Scraper (One Site) 4](#_Toc189486871)

[4.2 Phase 2: Multi-Site Config & Basic Multi-User Data Layer 4](#_Toc189486872)

[4.3 Phase 3: ChatGPT Web UI Experiments & Partial Integration 5](#_Toc189486873)

[4.4 Phase 4: Move to OpenAI API & Automated Parsing 6](#_Toc189486874)

[4.5 Phase 5: Feedback Loop, Dedup, & Basic Preference Handling 7](#_Toc189486875)

[4.6 Phase 6: RESTful API & Multi-User Security 8](#_Toc189486876)

[4.7 Phase 7: Subscription & Billing (Future) 8](#_Toc189486877)

[4.8 Phase 8: Front-End (Mobile or Web) 9](#_Toc189486878)

[5 Refined Requirements & Architecture Planning 9](#_Toc189486879)

[5.1 Authentication Models & Their Trade-offs 10](#_Toc189486880)

[5.1.1 1. Session-Based Authentication 10](#_Toc189486881)

[5.1.2 2. Token-Based Authentication (JWT) 11](#_Toc189486882)

[5.1.3 3. OAuth 2.0 or External Identity Providers 11](#_Toc189486883)

[5.1.4 4. Which Auth Model to Choose? 12](#_Toc189486884)

[5.2 Putting It All Together 13](#_Toc189486885)

[5.2.1 Next Steps 13](#_Toc189486886)

[6 Status 13](#_Toc189486887)

[6.1 Server 13](#_Toc189486888)

[6.1.1 ServerGen enhancements 14](#_Toc189486889)

[7 Workflows 14](#_Toc189486890)

[7.1 create a user 14](#_Toc189486891)

[7.2 Create a profile (system level) 14](#_Toc189486892)

[7.3 Create a crawl 14](#_Toc189486893)

[7.4 User/profile event search 14](#_Toc189486894)

# Purpose

The Event Manager system is designed to make it easy for people for find events of interest. Using user profiles and event tagging, the system will leverage AI technologies such as a recommender system to help users quickly and easily find events of interest while hiding events that are uninteresting.

# Operating environment/High Level Design

EventMan will use a client/server model. All data will be stored in a Mongo database which will be accessible via a ReST API. This will support both web and mobile clients.

EventMan will have a set of URLs to crawl. The crawl frequency is TBD. It is expected that almost all crawls will be http based and may need instructions on how to extract events. These instructions will also be stored. ReST API based crawls will also be supported.

A user may have multiple profiles. Each profile will define a set of events that are of interest (kid-friendly, concerts, festivals, …) and will be defined via a set of tags. Events will also have tags. Tag matching is where the AI component will come into play. The goal will be to minimize the amount of time and effort users need to spend to find events of interest.

The primary user may create other users. This is called an account. The ability of users to share and/or edit profiles across an account is TBD. Additionally, EventMan will have a set of built-in profiles that users may access. EventMan profiles are available on a read-only basis within an account.

# High level plan

* Start small (Phase 1: one tough site) and ensure your advanced scraping approach (Playwright + stealth, etc.) works.
* Build out multi-site support and your multi-user data model (Phase 2).
* Experiment with ChatGPT’s web UI (Phase 3) before fully integrating the API for automated data parsing (Phase 4).
* Add feedback loops, dedup, and initial preference logic (Phase 5).
* Then formalize a REST API with user accounts (Phase 6), paving the way for subscription billing (Phase 7) and front-end apps (Phase 8).

Each phase remains **4–10 hours** of core development/testing (except for subscription/billing or a new front-end, which likely take more time). The biggest wildcard is **site complexity** (anti-scraping measures, dynamic rendering, captcha, etc.)—you may need more specialized solutions there.

With that blueprint, you can incrementally build a robust, subscription-ready scraping + event discovery service for multiple users, each with isolated data and preferences—backed by Mongo, a Python-based scraping pipeline, and eventually a ChatGPT-based classification/tagging engine.

# Phase Summary

Before coding:

1. **Identify Anti-Scraping Tactics**
   * Do you need **rotating proxies** or a service like [Bright Data](https://brightdata.com/), ScraperAPI, or custom solutions (e.g., Tor, purchased proxies)?
   * Will you use **stealth** or **incognito** modes in **Playwright** or **Puppeteer** (e.g., playwright-stealth or puppeteer-extra-plugin-stealth) to mimic a real browser?
2. **Multi-User Data Model**
   * **Users** collection in Mongo: \_id, username, hashed\_password, preferences, usage\_stats.
   * **Events** stored per user or as a shared pool with user-specific filters/feedback references.
   * Possibly a **ScrapeJob** or **SiteConfig** collection if each user can define custom scraping tasks.
3. **Subscription Billing** (Future)
   * High-level plan: measure each user’s **API calls** (to ChatGPT, to scraping services) and store usage in usage\_stats.
   * Later integrate a payment gateway (Stripe, PayPal, etc.).
4. **Security / Auth**
   * Even if it’s just you initially, plan to eventually add JWT or session-based auth for a multi-user REST API.

**Outcome**: You’ll confirm how you’ll handle advanced anti-scraping setups and finalize the user data model. This is primarily planning, so still ~2–4 hours of discussion, research, and setup design.

## **Phase 1: Core Environment Setup & Basic Scraper (One Site)**

**Goal**: Prove out the environment with the toughest website you can find (heavy JS, infinite scroll). Show you can scrape at least some data.

1. **Local Setup**
   * Python 3.10+ (virtualenv), MongoDB (local or Atlas).
   * Install **Playwright** (recommended for Python) or **Puppeteer** (with a Python wrapper).
   * Optional: integrate a **stealth plugin** or set custom headers / user agents if you suspect bot detection.
2. **One Tough Site**
   * Script the infinite scrolling or “Load More” approach.
   * If they block scraping aggressively, test rotating proxies or at least handle random user agents, random delays, etc.
3. **Mongo Integration**
   * Create minimal schema: Events collection (fields: title, date, etc.).
   * Insert or update data; check dedup by title + date.
4. **CLI Command**
   * python main.py scrape --site=mysite runs the script, logs progress.

**Outcome**

* A working pipeline (on your Mac) that **handles a complex site** and stores data in Mongo.
* You’ve got the skeleton for advanced scraping (stealth, proxies, etc.).
* ~4–10 hours depending on site complexity.

## **Phase 2: Multi-Site Config & Basic Multi-User Data Layer**

**Goal**: Make the scraper **configurable** for multiple sites and introduce the **user** data model in Mongo (even if you’re the only user at first).

1. **Config-Driven Scraping**
   * A JSON or Python config that says:

json

CopyEdit

{

"site\_name": "Site A",

"start\_url": "https://example.com/events",

"pagination": "infinite\_scroll",

"selectors": {

"event\_container": ".event-card",

"title": ".title",

"date": ".date",

"location": ".location"

},

...

}

* + A script that loops over these configs, so you can add more sites over time without rewriting the scraping logic completely.

1. **Multi-User Schema**
   * users collection: \_id, email, password\_hash, prefs (might be empty for now).
   * events collection can have a user\_id field or a separate user-event linking approach if each user has a private set of events. If events are huge, you might store them once and store user references in a user\_events collection.
2. **CLI Enhancements**
   * python main.py scrape all --user=someUserId or something similar.
   * Possibly store ScrapeJob records: which user ran it, for which site, when, etc.

**Outcome**

* You can handle **multiple sites** in a structured way.
* You have a **foundation for multi-user** with a simple schema.
* ~4–8 hours, depending on how fancy you get with site configs.

## **Phase 3: ChatGPT Web UI Experiments & Partial Integration**

**Goal**: Use ChatGPT’s web interface to prototype how to parse or tag complex data. Then start partial automation.

1. **Manual Prompting**
   * Copy-paste sample event texts or messy HTML into ChatGPT’s web UI to figure out an effective prompt:

“Given this text, extract the event title, date, location, and music genre. Return JSON.”

* + Refine until you get consistent results.

1. **Store or Document** the Best Prompt
   * Save your final prompt or prompt templates.
   * Possibly do a few different example inputs to see if ChatGPT handles variety well.
2. **Partial Integration** (Optional)
   * For a small subset of events, you might do a manual or semi-automated approach:
     + Script prints the raw data. You copy it to ChatGPT. Then you paste the improved output back into the script or Mongo.
   * This is not fully automated, but it helps you confirm feasibility before paying for the API usage.

**Outcome**

* You have validated the approach for how ChatGPT will parse or tag events.
* You have a refined prompt that you can later embed into the code.
* ~4 hours or so of experimentation and iterating.

## **Phase 4: Move to OpenAI API & Automated Parsing**

**Goal**: Build an actual pipeline that sends each event’s text to the ChatGPT API, receives structured data, and saves it in Mongo.

1. **OpenAI API Setup**
   * Store OPENAI\_API\_KEY in .env or environment variables.
   * Install openai Python library.
2. **Prompting & Parsing**
   * In your scraping flow, once you have raw text or partial fields, you call:

python

CopyEdit

response = openai.ChatCompletion.create(

model="gpt-3.5-turbo",

messages=[

{"role": "system", "content": "You are an expert event parser..."},

{"role": "user", "content": event\_text}

]

)

* + Parse JSON from response.choices[0].message.content.

1. **Batching & Rate Limit Handling**
   * If you’re processing thousands of events, do it in batches, sleep if you hit rate limits.
   * Log usage for potential subscription billing later.
2. **Tagging & Genre**
   * Add a field tags or genre to the event record. Let ChatGPT guess it.
   * Optionally store user feedback (if user disagrees with a tag).

**Outcome**

* Fully **automated** extraction and categorization (title, date, genre, etc.).
* You now have usage metrics (how many events are processed → how many API calls).
* ~6–10 hours for integration, error handling, etc.

## **Phase 5: Feedback Loop, Dedup, & Basic Preference Handling**

**Goal**: Let each user review their events, mark them “liked / disliked,” refine tags, etc. Implement a better dedup mechanism.

1. **Feedback & Review**
   * A simple CLI or minimal web form: “Show me events from the last scrape for user X. Let me mark them as relevant or not.”
   * Store that feedback in feedback collection or a feedback field on each event record (liked: true/false, tags\_overridden: [...]).
2. **Deduplication**
   * Compare (title, date, location) or do a text similarity (like hashing or partial embeddings).
   * If an event is flagged as a duplicate, merge or skip.
3. **Basic Preferences**
   * If a user consistently “dislikes” Rap events, store a preference like prefs.disliked\_genres = ["Rap"].
   * Next time you parse or tag events as Rap, you can reduce their priority or hide them (depending on the user’s setting).

**Outcome**

* Users can **review** and **train** the system.
* Preferences are recognized to do minimal recommendation filtering (e.g., hide disliked genres).
* ~6–10 hours if you build a small web UI or advanced CLI.

## **Phase 6: RESTful API & Multi-User Security**

**Goal**: Expose the system via a standardized API with user-based isolation, set the stage for a subscription-based service.

1. **FastAPI/Flask Setup**
   * GET /events → returns events for the authenticated user.
   * POST /feedback/{event\_id} → user can submit feedback.
   * POST /scrape → (optionally) triggers a scrape job for the user’s configured sites.
2. **User Authentication**
   * Use JSON Web Tokens (JWT) or session tokens.
   * Each user logs in, you store or generate a token.
   * All subsequent requests verify the token and scope the data to that user.
3. **Scalability & Rate Limiting**
   * If you envision many users, you might add a simple rate limit to API calls or scraping requests.
4. **Usage Tracking**
   * For each user, track how many ChatGPT API calls are made, how many scrapes, etc.
   * Store in usage\_stats to eventually do usage-based billing.

**Outcome**

* A multi-user, secure REST API that supports different user logins and usage tracking.
* ~6–10 hours, more if you add robust security or admin dashboards.

## **Phase 7: Subscription & Billing (Future)**

**Goal**: Turn this into a subscription-based service. Could be a big step depending on your revenue model.

1. **Integrate Payment Processor**
   * **Stripe** is common. You create products (monthly subscription tiers, pay-per-usage, etc.).
2. **Usage Metering**
   * If using usage-based billing (e.g., X events scraped or Y ChatGPT calls), you must measure usage accurately and feed that into Stripe’s subscription usage.
3. **User Portal**
   * A page or section where users see how many events they’ve scraped this month, how many ChatGPT calls, next billing date, etc.

**Outcome**

* Fully monetized subscription service with usage-based or tier-based pricing.
* This can take **10–20 hours** or more (integrating Stripe + building user dashboards + handling upgrades/downgrades).

## **Phase 8: Front-End (Mobile or Web)**

Whether you want an iOS app or a web SPA (React/Vue) or both:

1. **iOS (SwiftUI or UIKit)**
   * Integrate with your **REST API**.
   * Show login screen, list events, filters, user feedback (like or dislike).
   * Time investment: **~20+ hours** for a basic but polished app if you’re experienced with SwiftUI. More if you’re new to iOS.
2. **Web Front-End** (React/Vue/Angular)
   * Same idea: call your REST endpoints to list events, let the user authenticate, etc.
   * Time: **~10–20 hours** for a minimal but functional UI, more if you want a polished design.

# Refined Requirements & Architecture Planning

1. **Anti-Scraping Scope**
   * **Decision**: Postpone sophisticated anti-blocking techniques (rotating proxies, captcha solving, stealth fingerprinting) until a later phase.
   * **Impact**: Your initial code design (scraper logic, site config) should be modular enough to drop in advanced methods later without a major rewrite.
   * **Implementation Note**: A typical approach is to keep the “fetch page” function or class flexible. Later, you can swap in a “proxy or stealth” version.
2. **User Model**
   * **Required Fields**: \_id, username/email, hashed\_password (for login).
   * **Demographic Fields** (as you requested): age, location, sex.
     + e.g., sex could be “M”/“F” or any string to handle broader gender.
   * **Preferences**: Not heavily used at first, but you can store them in a prefs object if needed.
3. **Data Storage**
   * **Events Collection**: Events are stored globally, referencing no specific user.
   * **Filtering**: Each user will see only the events that match their interest (or you rank them differently).
   * **Feedback** or **User–Event Relationship**: If you want to track “liked” or “dismissed” events per user, you can store them in a separate user\_events or feedback collection.
4. **Subscription Billing Model**
   * **Monthly or Yearly** subscription.
   * If external services (like ChatGPT API) impose usage-based costs, you can later decide how to upcharge or limit usage per user.
   * You’ll store each user’s subscription status (e.g., “active,” “trial,” “expired”) in the user record or a separate billing table.
5. **Authentication Model**
   * You want to design it so that multi-user is straightforward.
   * For Phase 0, deciding **which** auth approach is key. You’ll implement it more fully in a later phase.

## Authentication Models & Their Trade-offs

Below is a quick breakdown of the most common auth patterns used in web services. You’ll likely choose either **session-based** or **JWT-based** for a subscription service, but here’s the bigger picture.

### 1. **Session-Based Authentication**

1. **How It Works**
   * User logs in with username/password → server validates → server creates a session on the backend (often stored in memory or Redis) and sends the user a **session ID** (often in a cookie).
   * On subsequent requests, the browser includes this session ID cookie, and the server looks it up to see if the user is authenticated.
2. **Pros**
   * Very traditional, well-documented, easy to implement with frameworks (e.g., Flask, Django sessions).
   * Straightforward to revoke or invalidate a session (just remove it from the server store).
   * Good for smaller-scale or single-server setups; also easy if you manage sticky sessions or a shared session store for load balancing.
3. **Cons**
   * **Not stateless**—the server must maintain session state. If you scale horizontally, you need a distributed session store (Redis, Memcached).
   * Less ideal if you plan a microservices architecture or want purely token-based logic on the frontend (like mobile apps or multiple services).
4. **Typical Use Cases**
   * Traditional web apps, single-server or small cluster deployments, simpler user flows.

### 2. **Token-Based Authentication (JWT)**

1. **How It Works**
   * User logs in → server validates → server generates a **JWT** (JSON Web Token) with user claims (userID, roles, etc.) and signs it with a secret key.
   * Client stores the token (in memory, localStorage, or a secure cookie).
   * On each request, the client sends the token in the Authorization header (e.g., Bearer <token>).
   * The server verifies the token signature; if valid, it trusts the claims inside (like user ID).
2. **Pros**
   * **Stateless** on the server side: no session store is needed.
   * Very flexible for microservices or if you want a single sign-on across multiple services.
   * Commonly used in mobile apps or SPA front-ends (React, Vue) that talk to an API.
3. **Cons**
   * Token **revocation** is trickier. If you want to forcibly log out a user, you can’t just kill the session in the DB. You might need a **revocation list** or short token lifespans with frequent refresh.
   * If the token is stolen, it’s valid until it expires (unless you implement additional checks or store a “blacklist” of tokens).
4. **Typical Use Cases**
   * Modern SPA or mobile apps that call a REST API.
   * Multi-server or microservices environment with minimal shared state.

### 3. **OAuth 2.0 or External Identity Providers**

1. **How It Works**
   * Rely on Google, Facebook, GitHub, etc., for identity.
   * You redirect users to the external provider; they log in; you get an **access token** or **ID token** verifying user identity.
2. **Pros**
   * You don’t manage user passwords or sign-ups yourself.
   * The provider handles security.
   * Can provide easier sign-up if you expect users from those ecosystems.
3. **Cons**
   * More complex to implement if you just need “simple” user accounts.
   * If you want subscription billing, you still need your own logic for associating the user’s external identity with your billing data.
4. **Typical Use Cases**
   * If you want frictionless sign-up or for enterprise SSO scenarios.

### 4. **Which Auth Model to Choose?**

Given your scenario:

* **You want a subscription service** that might scale beyond a single server or local environment.
* **You may have a web front end** or a mobile front end (in the future), both of which will talk to your REST API.
* **You want multi-user** with distinct data for each user.

A **JWT-based** approach is often the most straightforward for modern REST APIs, especially if you plan to scale or support multiple client types (mobile, web, etc.):

* **Pro**: No server-side session store; each request includes the token → easy to handle in stateless manner.
* **Con**: You’ll need a strategy for logging out users or rotating tokens (e.g., refresh tokens, short-lifespan tokens, or a revocation list if you must forcibly invalidate tokens).

A **session-based** approach can still work:

* Potentially simpler if you’re building a single web application and not planning extensive scaling.
* But once you want a separate mobile app or a more decoupled front end, you’ll have to adapt your session model or use cookies with cross-origin support.

**Recommendation** (most common in modern subscription APIs): Start with **JWT** (access + refresh tokens). That’s widely supported, easy to handle from mobile apps or SPAs, and suits a future microservice or load-balanced environment.

## Putting It All Together

* **Phase 0** (planning) will now:
  1. **Confirm** you’ll only do **basic** headless scraping (no rotating proxies or advanced anti-scraping) for the MVP.
  2. **Define** your user collection fields (age, location, sex, plus subscription status).
  3. **Decide** on a minimal set of event fields for your events collection.
  4. **Choose** an auth strategy. We recommend **JWT** for a subscription-based REST service, but session-based is also viable if you prefer.
  5. **Outline** the steps to add subscription billing in a future phase (Stripe or similar).

After finalizing these decisions, you’ll proceed to **Phase 1** with environment setup, implementing the first site scraper, and storing events in Mongo. Because you’ll have a plan for how to eventually integrate advanced anti-scraping solutions and user subscription billing, you can code the current system in a **modular** way that won’t require a total redesign down the line.

### Next Steps

* Confirm your **auth** choice (JWT or session-based).
* Lock down your **data model** for users, events, and any bridging (user\_events or feedback) if you want.
* Then begin **Phase 1** with a minimal, working prototype—scrape one site, store events, and let a single user see them.

That sets you up to expand iteratively into multi-site scraping, ChatGPT integration, feedback loops, multi-user accounts, and eventually subscription billing.

##### You said:

ok - JWT to start. Can I create an ER diagram that you read for the data model. How best should I supply the model to you.

# Status

## Server

The base server is auto-generated. This builds ReST based CRUD operations on all entities and enforces single fleid data constraints.

### ServerGen enhancements

* Base Class - Utiliize a base class for all objects that includes id, dateCreated and dateUpdated. The logic for storing/retrieving these fields will be auto generated.
* Authentication/Security – The generator shall also create all logic for handling auth and secutity:
* Authorization via oAuth
* Authentication
* crawl prevention

# Workflows

## create a user

* 1. Create an account
  2. Create a user attached to the account

## Create a profile (system level)

## Create a crawl

* 1. Find an event web site
  2. Find URLs to crawl
  3. Crawl and extract events
  4. Tag the events

## User/profile event search