Technical Report

COMP3011 Technical Report: EventHub – Event & RSVP API

Module: COMP3011 – Web Services and Web Data
Coursework: CW1 – Individual Web Services API Development
Student: Nathaniel Sebastian (sc232ns)
Date: 4th February 2026

GitHub Repository: github.com/NathS04/comp3011-cw1-api
Live API: comp3011-cw1-api.onrender.com
API Documentation (PDF): See `docs/API_DOCUMENTATION.pdf`

Abstract

EventHub is a RESTful API for managing event registrations, designed for student societies and community organisations. Beyond standard CRUD functionality, this project implements **novel data integration** through an idempotent dataset import pipeline with full provenance tracking, and provides **analytics endpoints** for seasonality analysis, trending detection, and personalised recommendations. The system demonstrates clean architectural separation, comprehensive test coverage (25 tests), and thoughtful use of Generative AI as a development partner. This report documents the design decisions, implementation challenges, and critical reflections on AI-assisted development.

---

1. Problem & Scope

1.1 Problem Domain

Event registration is a universal need—from university societies to corporate conferences. Organisers need to create events, track capacity, and understand attendance patterns. Attendees want to discover relevant events and RSVP easily.

This project builds a backend API for this domain, prioritising: - Clean data modelling with proper constraints - Secure authentication - Extensibility for analytics and external data

1.2 Functional Requirements

Requirement	Implementation
Event CRUD	Create, read, update, delete events with validation
Attendee management	Register attendees with unique emails
RSVP tracking	Link attendees to events with status (going/maybe/not_going)
Capacity statistics	Calculate remaining spots and RSVP breakdown
Authentication	JWT-based auth for write operations
Novel data integration	Import external datasets with provenance
Analytics	Seasonality, trending, and recommendations

1.3 Non-Goals (Explicit Exclusions)

To maintain scope, the following are **not** implemented: - Real-time notifications or WebSocket support - Payment processing or ticketing - Mobile app or frontend (API-only) - Machine learning models (recommendations use deterministic scoring) - Role-based access control (all authenticated users have equal permissions)

---

2. Architecture

2.1 Technology Stack

Component	Technology	Justification
Framework	FastAPI	Modern async support, automatic OpenAPI docs, excellent Pydantic integration
Database	SQLite (dev)	Zero-config, file-based. Prod-ready with PostgreSQL via `DATABASE_URL`
ORM	SQLAlchemy 2.x	Industry-standard, supports relationships and migrations
Migrations	Alembic	Version-controlled schema changes
Auth	JWT (python-jose)	Stateless tokens, fits REST principles

Component	Technology	Justification
Testing	pytest + TestClient	Isolated in-memory DB with StaticPool

2.2 Layered Architecture

```
┌──────────────────────────────────────────────────────┐
│                     HTTP Client                        │
└──────────────────────────────────────────────────────┘
                          │
                          ▼
┌──────────────────────────────────────────────────────┐
│   app/main.py (FastAPI app, middleware, CORS)        │
└──────────────────────────────────────────────────────┘
                          │
                          ▼
┌──────────────────────────────────────────────────────┐
│   app/api/routes.py + analytics.py (HTTP handlers)   │
└──────────────────────────────────────────────────────┘
                          │
                          ▼
┌──────────────────────────────────────────────────────┐
│   app/crud.py (Business logic, DB operations)        │
└──────────────────────────────────────────────────────┘
                          │
                          ▼
┌──────────────────────────────────────────────────────┐
│   app/models.py (SQLAlchemy ORM models)              │
└──────────────────────────────────────────────────────┘
                          │
                          ▼
┌──────────────────────────────────────────────────────┐
│   Database (SQLite / PostgreSQL)                     │
└──────────────────────────────────────────────────────┘
```

Key Principle: Routes are thin handlers. All business logic lives in `crud.py`, making it testable and reusable.

---

3. Data Model

3.1 Entity-Relationship Diagram

```
┌──────────────┐        ┌──────────────┐        ┌──────────────┐
│    users     │        │    events    │        │  attendees   │
├──────────────┤        ├──────────────┤        ├──────────────┤
│ id (PK)      │        │ id (PK)      │        │ id (PK)      │
│ username     │        │ title        │        │ name         │
│ email (UQ)   │        │ description  │        │ email (UQ)   │
│ hashed_pw    │        │ location     │        └──────────────┘
│ created_at   │        │ start_time   │               │
└──────────────┘        │ end_time     │               │
                        │ capacity     │               │
```

```
                       │ source_id(FK)│◄───┐         │
                       │ is_seeded    │    │         │
                       └──────┬───────┘    │         │
                              │            │         │
                              │            │         │
┌──────────────┐              │    ┌───────┴────────┐│
│ data_sources │              │    │  import_runs   ││
├──────────────┤              │    ├────────────────┤│
│ id (PK)      │◄─────────────┘    │ id (PK)        ││
│ name (UQ)    │                   │ source_id (FK) ││
│ url          │                   │ status         ││
│ license      │                   │ rows_inserted  ││
│ retrieved_at │                   │ errors_json    ││
└──────────────┘                   └────────────────┘│
                                                     │
                                                     │
                       ┌─────────────┐               │
                       │    rsvps    │◄──────────────┘
                       ├─────────────┤
                       │ id (PK)     │
                       │ event_id(FK)│
                       │ attendee_id(FK)│
                       │ status      │
                       │ created_at  │
                       └─────────────┘
                      UNIQUE(event_id, attendee_id)
```

3.2 Novel Tables: Data Provenance

To support external dataset integration, I added two tables:

Table	Purpose
`data_sources`	Tracks where external data came from (name, URL, license)
`import_runs`	Logs each import execution (rows processed, errors, status)

Events can optionally link to a `data_source` via `source_id`, enabling provenance queries.

---

4. Key Design Decisions

Decision 1: RSVP as a First-Class Entity

Choice: Separate `rsvps` table with its own attributes (status, created_at).

Alternative considered: Store RSVPs as a list field embedded in Event.

Trade-off: The separate table adds a JOIN for queries but enables: - Tracking when someone RSVP'd - Querying RSVPs independently (e.g., "all events user X is attending") - Database-level uniqueness constraint

I explored the embedded approach with AI assistance but found it created data integrity risks and made analytics harder.

Decision 2: JWT vs Server-Side Sessions

Choice: Stateless JWT tokens.

Alternative considered: Server-side session storage with Redis.

Trade-off: JWT is simpler for a pure API (no session store needed) but: - Tokens can't be revoked until expiry - Token size grows with claims

For this coursework scope, JWT simplicity outweighs revocation limitations.

Decision 3: SQLite with PostgreSQL Path

Choice: SQLite for development, designed for PostgreSQL in production.

Alternative considered: PostgreSQL from the start.

Trade-off: SQLite enables zero-configuration local development. The code uses standard SQLAlchemy patterns that work with both databases. Alembic migrations use `batch_alter_table` for SQLite compatibility.

---

5. Novel Data Integration

5.1 Dataset Source

The system supports importing external event data. For demonstration, I created a sample CSV representing events from a fictional "Leeds Public Events API":

Field	Example
id	EXT_001
title	Advanced AI Workshop
location	Leeds Tech Hub
start_time	2026-05-10T10:00:00
capacity	50

License: Sample data is CC-BY-4.0 compliant.

5.2 Import Pipeline Design

The `scripts/import_dataset.py` script implements:

1. **Source registration:** Creates or retrieves `DataSource` record
2. **Run logging:** Creates `ImportRun` with status “running”
3. **Idempotent import:** Uses `source_record_id` to detect existing records
4. **Error collection:** Continues on row errors, logs to `errors_json`
5. **Summary output:** Reports rows read, inserted, updated, errors

```
# Idempotency check
existing = db.query(Event).filter(
    Event.source_id == source.id,
    Event.source_record_id == record_id
).first()

if existing:
    # Update existing record
    rows_updated += 1
else:
    # Insert new record
    rows_inserted += 1
```

5.3 Verification Output

```
Created new data source: Leeds Public Events API
Started Import Run ID: 1
Import Finished. Status: success
Read: 3, Inserted: 3, Updated: 0, Errors: 0
```

---

6. Analytics & Recommendations

6.1 Seasonality Endpoint

Purpose: Aggregate events by month to identify peak periods.

Endpoint: `GET /analytics/events/seasonality`

Response: Monthly counts with top categories (placeholder for future enhancement).

6.2 Trending Events

Purpose: Surface events gaining momentum.

Endpoint: `GET /analytics/events/trending?window_days=30`

Scoring Formula:

```
trending_score = (recent_rsvps × 1.5) + (total_rsvps × 0.5)
```

The 1.5× multiplier for recent RSVPs rewards events with growing interest.

6.3 Personalised Recommendations

Purpose: Suggest relevant events based on user history.

Endpoint: `GET /events/recommendations`

Algorithm: 1. Find attendee record for authenticated user (by email) 2. Extract locations from past RSVPs 3. Recommend future events at those locations 4. Cold start: Return top upcoming events if no history

This is deterministic scoring, not machine learning—appropriate for coursework scope while demonstrating the concept.

---

7. Testing Strategy

7.1 Approach

- **Isolated database:** In-memory SQLite with `StaticPool`
- **Fresh state:** Tables recreated per test function
- **Both paths tested:** Success cases and error handling (401, 404, 409, 422)

7.2 Coverage Summary

Test File	Tests	What's Covered
`test_auth.py`	6	Register, login, wrong password, protected routes
`test_events.py`	5	CRUD operations, pagination, filtering
`test_attendees.py`	4	Create, get, list events for attendee
`test_rsvps.py`	4	Create, list, delete, duplicate rejection
`test_analytics.py`	4	Seasonality, trending, recommendations
`test_import.py`	1	Data provenance model constraints
`test_health.py`	1	Health endpoint

Total: 25 tests, all passing.

---

8. Deployment

8.1 Platform

Deployed on Render.com with GitHub integration for automatic deploys.

8.2 Environment Variables

Variable	Purpose
`DATABASE_URL`	PostgreSQL connection (Render-provided)
`SECRET_KEY`	JWT signing key
`ENVIRONMENT`	`production`

8.3 Verification

- Health check: `GET /health` returns `{"ok": true}`
- Swagger UI: `/docs` accessible
- Sample requests via curl verified

---

9. Generative AI Usage Declaration

9.1 Tools Used

Tool	Purpose
Google Gemini (Antigravity)	Primary assistant—architecture, code generation, debugging
ChatGPT (GPT-4)	Secondary—marking feedback interpretation

9.2 How AI Was Used

My workflow integrated AI throughout:

1. **Architecture planning:** Described requirements, AI suggested layered structure
2. **Scaffolding:** Generated initial models, schemas, CRUD functions
3. **Debugging:** Pasted tracebacks, received explanations and fixes
4. **Documentation:** Drafted sections, edited for accuracy and my voice

9.3 Exploring Alternatives (Outstanding-Level Usage)

Rather than just accepting AI suggestions, I used it to explore design alternatives:

Example 1: RSVP Storage

> **Me:** Should I store RSVPs as a list in the Event model or as a separate table?
>
> **AI:** A separate table enables uniqueness constraints, independent queries, and timestamps. Embedded lists are simpler but create data integrity risks.
>
> **My decision:** Separate table—alignment with relational design principles.

Example 2: Auth Approach

> **Me:** JWT vs session-based auth for a REST API?
>
> **AI:** JWT is stateless and simpler for APIs. Sessions require storage but enable revocation.
>
> **My decision:** JWT for simplicity, with documented limitation about revocation.

Example 3: SQLite Compatibility

> **AI initially suggested:** Standard `op.add_column` for migrations.
>
> **Problem:** SQLite doesn't support ALTER TABLE for foreign keys.
>
> **Solution found with AI:** Use Alembic's `batch_alter_table` which recreates the table.

9.4 Bugs from AI Suggestions

1. **bcrypt compatibility:** AI suggested bcrypt, but my environment had issues. Switched to pbkdf2_sha256.
2. **Circular imports:** AI-generated auth code caused import cycles. Fixed with local imports.
3. **Test fixtures:** Initial fixture didn't isolate state properly. Redesigned with per-test table recreation.

9.5 Critical Reflection

AI accelerated development significantly—especially for boilerplate and debugging. However:

- I always ran generated code before committing
- I caught security issues (hardcoded secrets) in review
- I ensured I understood every line before accepting it

AI was a partner, not an author. The architectural decisions and critical evaluation were mine.

---

10. Limitations & Future Work

Current Limitations

- Single-tenant: All authenticated users can modify all events
- No email verification for registration
- Token expiry (30 min) with no refresh mechanism
- SQLite concurrency limitations

Future Roadmap

- □ PostgreSQL for production scalability
- □ Role-based access (admin vs. user)
- □ Capacity enforcement (reject RSVPs when full)
- □ Email notifications
- □ Rate limiting
- □ Event categories for richer analytics

Appendix A: GenAI Conversation Logs

See attached `docs/appendix_genai_logs.md` *for selected excerpts.*

Word Count: ~2,100 words (excluding diagrams and code)

Report generated for COMP3011 CW1 submission, University of Leeds