COMP3011 Coursework 1 — Technical Report

Project: Habit & Productivity Analytics API (REST)
Student: Nicolas Issa
Repository: https://github.com/NicolasIssa1/COMP3011-CW1-HABIT-API
Deployed URL: https://comp3011-cw1-habit-api.onrender.com

---

1. Overview

This project delivers a RESTful Habit & Productivity Analytics API that enables clients to track habits, record habit completions, and generate productivity analytics. The API supports full CRUD operations for habits, "done-only" completion logs (one log represents completion of a habit on a given date), and analytics endpoints to compute habit streaks and weekly summaries.

Core capabilities:

- Habits CRUD: create, list, retrieve, update (partial), delete
- Completion logs: add completion records, list/filter logs, delete a log
- Analytics: current and longest streak for a habit; weekly completion summary across habits

The API is deployed and accessible via Swagger UI for interactive testing and demonstration.

---

2. Technology Stack and Justification

FastAPI (Python)

FastAPI was chosen for building the API because it supports rapid development while maintaining strong structure through type hints and Pydantic validation. It also automatically generates OpenAPI

documentation, which directly supports the coursework deliverables and makes the oral demonstration easier (interactive testing via Swagger UI).

SQLAlchemy (ORM) + Alembic (Migrations)

SQLAlchemy provides a clear and maintainable way to model data and interact with the database using an ORM. It helps keep database access consistent and testable across endpoints.
Alembic was used to manage database migrations so schema changes are versioned and reproducible. This improves reliability between local development and deployment environments.

Database: SQLite locally + Render deployment database configuration

SQLite is used locally for simplicity and quick iteration. The local database file (`dev.db`) is ignored from version control to keep the repository clean and reproducible.
In deployment, configuration is driven through environment variables (e.g., `DATABASE_URL`) so the same codebase can run in different environments. This separation supports professional deployment practice and avoids hardcoding deployment settings.

---

3. Architecture and Design

Repository structure

The code is organised to separate HTTP concerns, validation, data models, and business logic:

- `app/routers/` — API routes (habits, logs, analytics)
- `app/schemas/` — Pydantic request/response models
- `app/models/` — SQLAlchemy database models
- `app/db/` — database engine + session configuration
- `app/services/` — business logic (streaks, weekly summary)
- `migrations/` — Alembic migration history
- `tests/` — pytest test suite
- `docs/` — exported API documentation PDF

- `report/` — technical report (this document exported as PDF)
- `ai-logs/` — GenAI declaration and exported conversation logs

Database session dependency

Routes use a database session dependency (`get_db`) so request handlers remain clean and consistent. This pattern ensures session lifecycle management is controlled and supports testability.

Key design choices

1. Habits vs Logs separation
   Habits represent "what the user wants to do," while logs represent "when the user completed it." This normalised design avoids duplicating habit data and enables clean analytics queries.
2. Done-only logs model
   A completion log row exists only when the habit is completed. If a date is missing for a habit, the habit was not completed on that date. This reduces storage complexity and makes streak computation straightforward.
3. Uniqueness constraint and conflict handling
   A database uniqueness constraint ensures there can be at most one completion per habit per date (`habit_id, date`). If a client attempts to create a duplicate completion log, the API returns a `409 Conflict`. This preserves integrity and prevents inflated analytics.
4. Analytics computed from logs (no redundant storage)
   Streaks and summaries are computed from the existing logs rather than stored as fields in the database. This avoids inconsistency and ensures analytics always reflect the true underlying completion data.

---

4. API Functionality

Habits (CRUD)

- `POST /habits` — create a habit (201 Created)
- `GET /habits` — list all habits (200 OK)

- `GET /habits/{id}` — retrieve a habit by id (200 OK / 404 Not Found)
- `PATCH /habits/{id}` — partial update (200 OK / 404 Not Found)
- `DELETE /habits/{id}` — delete habit (204 No Content / 404 Not Found)

Logs (done-only completion records)

- `POST /habits/{id}/logs` — create a completion log (201 Created)
  - returns `404 Not Found` if the habit does not exist
  - returns `409 Conflict` if a log already exists for the same habit + date
- `GET /habits/{id}/logs` — list logs for a habit (200 OK / 404 Not Found)
  - supports date filtering (e.g., from/to dates depending on implementation)
- `DELETE /habits/{id}/logs/{log_id}` — delete a specific log (204 No Content / 404 Not Found)

Analytics

- `GET /habits/{id}/streak` — returns streak statistics (200 OK / 404 Not Found)
  Typical output includes:
  - current streak
  - longest streak
  - total completions (if included)
- `GET /analytics/weekly-summary?week=YYYY-WW` — weekly completion summary (200 OK / 400 Bad Request)
  This endpoint aggregates completions in the specified ISO week and returns:
  - total completions across all habits
  - breakdown per habit (counts and/or names depending on implementation)
    If the `week` parameter format is invalid, a `400 Bad Request` is returned.

Error handling and status codes

The API uses clear HTTP status codes to support predictable client behaviour:

- `404 Not Found` for missing resources (e.g., habit id does not exist)
- `409 Conflict` for duplicate completion logs (same habit + same date)
- `400 Bad Request` for invalid query parameters (e.g., incorrect week format)
- `201 Created` for successful creation
- `204 No Content` for successful deletion

---

5. Testing Approach

Testing is implemented using `pytest` with FastAPI's test client. The test suite focuses on correctness for core functionality and key edge cases:

- Habits CRUD:
  - create habit, list habits, retrieve by id, update, delete
  - confirm `404` behaviour for non-existent ids
- Logs:
  - create log for an existing habit
  - verify duplicate creation returns `409 Conflict`
  - list logs (including filtering where applicable)
  - delete a log and confirm it is removed
- Analytics:
  - verify streak calculations on known sets of log dates
  - verify weekly summary returns expected totals/breakdowns

These tests reduce regression risk and provide confidence for the oral examination because behaviour can be demonstrated and justified with automated evidence.

---

6. Deployment

Deployment environment

The API is deployed on Render and is accessible at: https://comp3011-cw1-habit-api.onrender.com

Useful URLs for demonstration

- Health check: `/health`
  https://comp3011-cw1-habit-api.onrender.com/health
- Swagger UI: `/docs`
  https://comp3011-cw1-habit-api.onrender.com/docs
- OpenAPI schema: `/openapi.json`
  https://comp3011-cw1-habit-api.onrender.com/openapi.json

Local run instructions (summary)

Typical local workflow:

- install dependencies from `requirements.txt`
- run migrations using Alembic
- start the server with Uvicorn
- run `pytest` for automated tests

The repository README contains the exact commands for setup and execution.

---

7. Challenges, Limitations, and Future Work

Challenge: Alembic autogenerate issue

A key challenge occurred early with Alembic migration autogeneration producing an empty migration file. This was resolved by creating the initial migration manually and ensuring migrations correctly reflected the SQLAlchemy models. This reinforced the importance of validating migration output rather than assuming it is correct automatically.

Current limitations

- No authentication / user accounts (API is open)
- Analytics are currently limited to streak + weekly summary
- Error formatting may rely on FastAPI defaults rather than a fully standardised error schema

Future work

- Add authentication (e.g., API key or JWT) if scope permits
- Add additional analytics such as monthly summaries, completion rate, or trends
- Improve validation further (e.g., allowed frequency values, stricter date rules)
- Optional "wow" feature: working-day streak using a bank holidays dataset (with correct licensing + references)

---

8. Generative AI Declaration and Usage Analysis

This is a GREEN assessment and GenAI usage has been declared transparently.

Tools used

- ChatGPT (OpenAI)

How GenAI was used

GenAI support was used to accelerate planning and improve correctness, including:

- designing endpoint structure and route conventions
- exploring alternatives for data modelling and error handling
- guidance for SQLAlchemy/Alembic patterns and debugging
- drafting documentation wording and report structure

What was verified manually

All GenAI suggestions were manually reviewed before being committed:

- API behaviour was validated through Swagger UI testing
- automated tests (`pytest`) were run to confirm correctness
- migrations were checked and applied to ensure schema consistency

Exported logs

GenAI declaration and exported conversation logs are stored in: `ai-logs/` (including `ai-logs/logs/`)

References

- FastAPI Documentation: https://fastapi.tiangolo.com/
- SQLAlchemy Documentation: https://docs.sqlalchemy.org/
- Alembic Documentation: https://alembic.sqlalchemy.org/
- Render Documentation: https://render.com/docs
- Pytest Documentation: https://docs.pytest.org/