



HABIT TRACKING APPLICATION

Development Phase

Marthinus Maree 92122115

Habit Tracking Application

Table of contents

Introduction	2
Overview	2
Definitions	2
Project Overview.....	2
User Interaction	2
Command line interface (CLI)	2
Swagger UI	4
Project Structure	7
Technology choices.....	9
Bibliography	9

Introduction

Most people want to stop unhealthy habits and create good habits in its place. To achieve this, they turn to technology for assistance. I want to create a habit tracking application to assist them to achieve their goals.

Overview

I used Python version 3.10.3 with **FastAPI** and **click** to create a backend for my habit tracking application.

Definitions

1. **Habit:** Regular activity or practice that you want to create e.g. *Go running every day*
2. **Completed Habit:** An activity or practice that you have completed at a specific time e.g.,
Went for a run at 1pm today.
3. **Tracked Habit:** A habit that was completed at least once is considered a tracked habit.

Project Overview

I created a backend with a command line interface and a Swagger UI to simplify user interaction. I have limited data input and some other features to the Swagger UI for simplicity. One could also use tools like Postman to interact with the REST API.

For testing I have included functions to seed (loading data) and delete (purge all data excluding frequencies) testing data.

User Interaction

All the commands listed below should be run from the command line in the **habits_backend** folder of the cloned project.

Command line interface (CLI)

Help Menu

To load a list of all available commands, run the following command:

```
python main.py -help
```

Output:

```
Usage: main.py [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  analyse-equal-periodicity  Return a list of habits with equal periodicity.
  analyse-longest-streak     Return the longest streak for any habit...
  analyse-streak-habit       Return the start and end date for a streak...
  analyse-tracked-habits     Return a list of tracked habits.
  data-clear                 Clear habit data from the database.
  data-seed                  Load testing data into the database.
  start-rest-api             Start the REST API for the habit tracking...
```

Load Data

To load testing data into the database run the following command:

```
python main.py data-seed
```

Output:

```
Seeded data
```

Clear Data

To clear the database of all habits and completed habits run the following command:

```
python main.py data-clear
```

Output:

```
Database Cleared
```

Get all tracked habits

The following command will return a list of all tracked habits:

```
python main.py analyse-tracked-habits
```

Output:

```
Tracked Habits:
```

```
[{'id': 1, 'name': 'Running', 'repeated': 'Daily', 'count': 15}, {'id': 2, 'name': 'Meditation', 'repeated': 'Daily', 'count': 8}, {'id': 3, 'name': 'Family', 'repeated': 'Weekly', 'count': 4}]
```

Get habits with the same period (equal periodicity)

To get habits with the same period (equal periodicity), run the following command:

```
python analyse-equal-periodicity --frequency daily
```

Output:

```
Equal Periodicity:
```

```
[{'id': 1, 'name': 'Running', 'repeated': 'Daily', 'count': 15}, {'id': 2, 'name': 'Meditation', 'repeated': 'Daily', 'count': 8}]
```

Habit with the longest run streak

Run the following command to display which habit has the longest run streak:

```
python main.py analyse-longest-streak
```

Output:

```
Longest Streak any habit:
```

```
{'start': datetime.datetime(2022, 4, 27, 14, 17, 45), 'end': datetime.datetime(2022, 5, 3, 15, 57, 21), 'cnt': 8, 'habit_id': 2}
```

Longest streak for a habit

To display the longest streak for a habit:

```
python main.py analyse-streak-habit --habit_id 1
```

Output:

```
Streak Details:
```

```
{'start': datetime.datetime(2022, 4, 27, 14, 17, 45), 'end': datetime.datetime(2022, 5, 2, 15, 57, 21), 'cnt': 6}
```

Swagger UI

To start the Swagger UI for a visual and interactive interface run the following command:

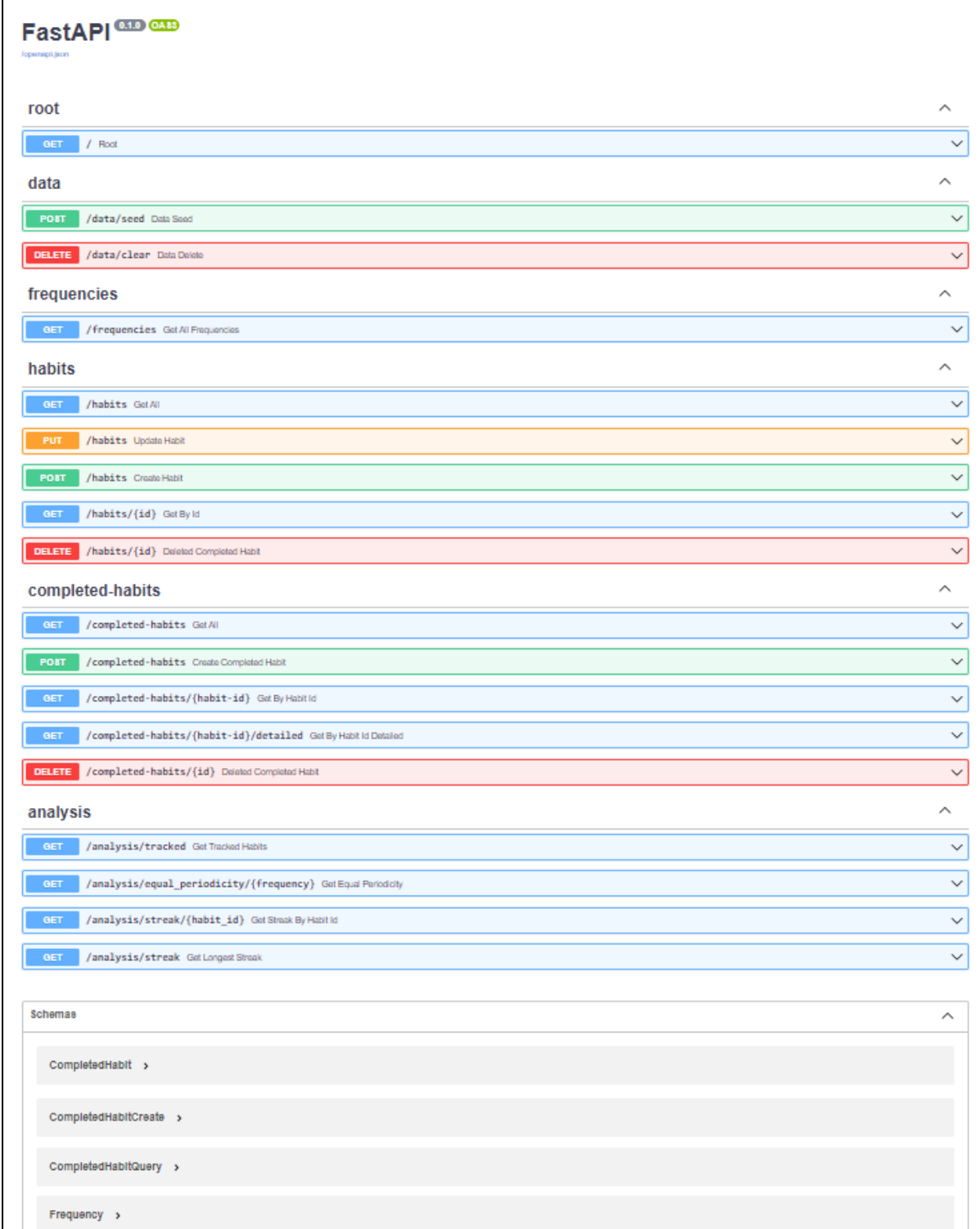
```
python main.py start-rest-api
```

Output:

```
INFO:      Started server process [13812]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
INFO:      Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```

Then if you browse to the <http://localhost:8000/> URL in your browser, you should see the following Swagger UI interface which will allow you to interact with the API's resources, inspect the paths and see a color-coded representation of the HTTP verbs. This is an interactive interface which

allows you to call the endpoints:



The screenshot displays the FastAPI Swagger UI for an application. The interface is organized into sections for different API endpoints, each with a color-coded header and a list of endpoints with their methods and descriptions.

- root**:
 - GET / Root
- data**:
 - POST /data/seed Data Seed
 - DELETE /data/clear Data Delete
- frequencies**:
 - GET /frequencies Get All Frequencies
- habits**:
 - GET /habits Get All
 - PUT /habits Update Habit
 - POST /habits Create Habit
 - GET /habits/{id} Get By Id
 - DELETE /habits/{id} Deleted Completed Habit
- completed-habits**:
 - GET /completed-habits Get All
 - POST /completed-habits Create Completed Habit
 - GET /completed-habits/{habit-id} Get By Habit Id
 - GET /completed-habits/{habit-id}/detailed Get By Habit Id Detailed
 - DELETE /completed-habits/{id} Deleted Completed Habit
- analysis**:
 - GET /analysis/tracked Get Tracked Habits
 - GET /analysis/equal_periodicity/{frequency} Get Equal Periodicity
 - GET /analysis/streak/{habit_id} Get Streak By Habit Id
 - GET /analysis/streak Get Longest Streak
- Schemas**:
 - CompletedHabit >
 - CompletedHabitCreate >
 - CompletedHabitQuery >
 - Frequency >

Data (Assist with testing)

We have URLs for loading and clearing the data. Normally this would not be part of the REST API but have been added here to assist with testing. This allows for loading test data or clearing the database. It will not delete the default frequencies. See the example below:

The screenshot shows a REST client interface for a POST request to `/data/seed`. The status bar indicates a 200 response. The response body is a JSON object: `{ "message": "Loaded the data" }`. The interface includes a 'Parameters' section (empty), an 'Execute' button, and a 'Responses' section with a 'Curl' command and a 'Request URL' field.

```
curl -X 'POST' \
  'http://localhost:8000/data/seed' \
  -H 'accept: application/json' \
  -d ''
```

Request URL: `http://localhost:8000/data/seed`

Server response

Code	Details
200	<pre>{ "message": "Loaded the data" }</pre>

Frequencies

We have pre-loaded three default frequencies to indicate how often a habit is repeated. For example, a habit must be repeated daily:

The screenshot shows a JSON schema for the `Frequency` model. The `repeat` field is an enum with values `day`, `week`, and `month`. The `id` field is an integer.

```
{
  "name": "Frequency",
  "properties": {
    "name": {
      "type": "string",
      "title": "Name"
    },
    "repeat": {
      "type": "string",
      "title": "TimeCode",
      "enum": [
        "day",
        "week",
        "month"
      ]
    },
    "id": {
      "type": "integer",
      "title": "Id"
    }
  }
}
```

The screenshot shows a REST client interface for a GET request to `/frequencies`. The status bar indicates a 200 response. The response body is a JSON array of three frequency objects: `[{"name": "Daily", "repeat": "day", "id": 1}, {"name": "Weekly", "repeat": "week", "id": 2}, {"name": "Monthly", "repeat": "month", "id": 3}]`. The interface includes a 'Parameters' section (empty), an 'Execute' button, and a 'Responses' section with a 'Curl' command and a 'Request URL' field.

```
curl -X 'GET' \
  'http://localhost:8000/frequencies' \
  -H 'accept: application/json'
```

Request URL: `http://localhost:8000/frequencies`

Server response

Code	Details
200	<pre>[{ "name": "Daily", "repeat": "day", "id": 1 }, { "name": "Weekly", "repeat": "week", "id": 2 }, { "name": "Monthly", "repeat": "month", "id": 3 }]</pre>

Habits

The habit URLs are used to query, create, and delete habits:

habits	
GET	/habits Get All
POST	/habits Create Habit
GET	/habits/{id} Get By Id
DELETE	/habits/{id} Deleted Completed Habit

Completed Habits

The completed habit URLs are used to query and track when a habit has been completed. The completed habits are used for the analysis and to calculate the running streaks:

completed-habits	
GET	/completed-habits Get All
POST	/completed-habits Create Completed Habit
GET	/completed-habits/{habit-id} Get By Habit Id
GET	/completed-habits/{habit-id}/detailed Get By Habit Id Detailed
DELETE	/completed-habits/{id} Deleted Completed Habit

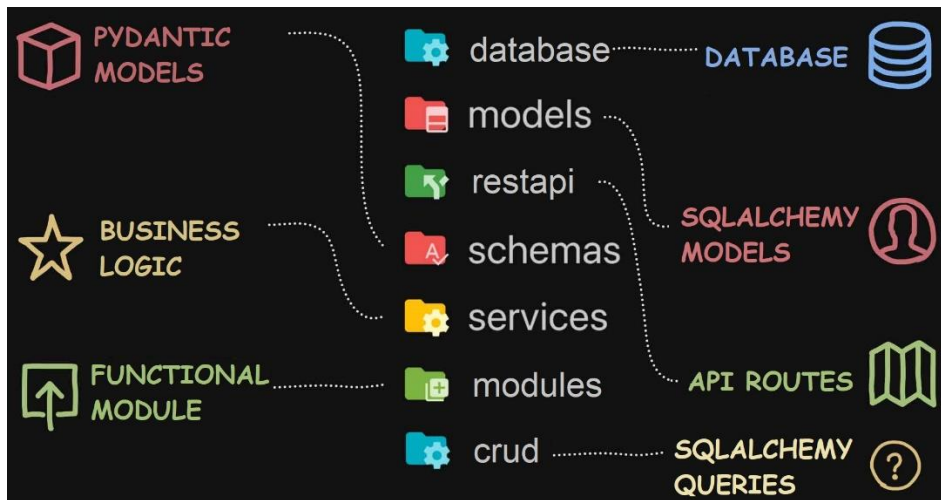
Analysis

The analysis URLs are used to query information like which habits are tracked, the longest running streak, etc.:

analysis	
GET	/analysis/tracked Get Tracked Habits
GET	/analysis/equal_periodicity/{frequency} Get Equal Periodicity
GET	/analysis/streak/{habit_id} Get Streak By Habit Id
GET	/analysis/streak Get Longest Streak
<div>Parameters Cancel</div> <div>No parameters</div> <div><div>Execute</div><div>Clear</div></div> <div>Responses</div> <div><div>Curl</div><div><pre>curl -X 'GET' \ 'http://localhost:8000/analysis/streak' \ -H 'accept: application/json'</pre></div><div>Request URL</div><div><pre>http://localhost:8000/analysis/streak</pre></div><div>Server response</div><div><div>Code</div><div>Details</div></div><div><div>200</div><div><div>Response body</div><div><pre>{ "start": "2022-04-27T14:17:45", "end": "2022-05-03T15:57:21", "cnt": 9, "habit_id": 2 }</pre></div><div><div>Download</div></div></div></div></div>	

Project Structure

The main entry point for the application is through main.py module. This module uses the [click](#) package to create a command line interface:



The user will interact with the CLI or the REST API. The interface will call the relevant service which in turn would call the crud functions. This is based on an [Clean Architecture approach](#).

The **pydantic** schemas are used for data parsing and validation. In turn the **SQLAlchemy** models are used for parsing the data for the database. I have added the functional programming module to the modules folder.

I used docstrings and inline comments to document my code:

```
"""
    This module contains the Habit analysis.
    We used functional programming concepts in this module.
"""

def is_tracked(habit):
    """Check if a habit is tracked. This means completed at least once."""
    return True if habit["count"] > 0 else False
```

I used **pytest** to test the functionality of my backend:

The screenshot shows the PyCharm IDE interface. The top pane displays Python code for testing a REST API. The bottom pane shows the test results for the same code.

```

19 def test_rest_api():
20     """Test the REST API startup."""
21     result = runner.invoke(start_rest_api)
22     assert result.exit_code == 0
23
24
25 def test_analyse_streak_habit():
26     """Test the analysis of a streak."""
27     result = runner.invoke(analyse_streak_habit, ['--habit_id', 1])
28     assert result.exit_code == 0
29

```

The test results pane shows the following output:

```

Run: pytest for test_main.test_rest_api x pytest in test_main.py x
Tests failed: 2, passed: 1 of 3 tests - 85 ms
Test Results
test_main 85 ms
  test_seed_data 12 ms
  test_rest_api 61 ms
  test_analyse_streak_habit 12 ms
E + where 1 = <Result SystemExit(1)>.exit_code
test_main.py:22: AssertionError
test_main.py::test_analyse_streak_habit PASSED [100%]
===== 2 failed, 1 passed in 0.65s =====
Process finished with exit code 1

```

Technology choices

- **Python version 3.10.3** – Project requirement to use 3.7 or later.
- **PyCharm 2022.1.1** - Popular IDE / source-code editor that runs on Windows, Linux and macOS. I found that it works better than Visual Studio Code for python development.
- **sqlite3** – It is a library that provides lightweight disk-based database to persist the data.
- **pytest** – Framework for writing tests.
- **FastAPI** – Framework for building APIs with python. This will provide an alternative for the CLI.
- **click** – Python library for creating command line interfaces.
- **Pylint** – Linting tool that checks for coding errors and enforce coding standards.
- **Swagger UI** – Interactive exploration to call and test your API from the browser.
- **SQLAlchemy** – Accessing data stored in the database.
- **pydantic** - Data parsing and validation.

Bibliography

1. n.a. (n.d). Google Python Style Guide – Naming
<https://google.github.io/styleguide/pyguide.html#316-naming>
2. Lutz, M (2013). OOP: The Big Picture, Learning Python 5th Edition. O'Reilly
3. Marcus, S (2021, July 16). Test Driven Development with pytest
<https://stackabuse.com/test-driven-development-with-pytest/>
4. n.a. (n.d). Zalando RESTful API and Event Guidelines
<https://opensource.zalando.com/restful-api-guidelines/>
5. n.a. (n.d). FastAPI - SQL (Relational) Databases
<https://fastapi.tiangolo.com/tutorial/sql-databases/>
6. n.a. (n.d). click
<https://click.palletsprojects.com/en/8.1.x/>
7. Martin, Robert C (2012, August 13) – The Clean Architecture
<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>