Mathematical Operations

The archive includes three versions of the requested Python project that implements three mathematical operations: power, n-th Fibonacci and n factorial: one that can be run from the command line, one synchronous FastAPI version and finally, the optimal solution, one asynchronous FastAPI version.

All three versions use SQLite to handle the database aspect and are comprised of four main .py files:

- services.py, containing the mathematical logic for each operation
- models.py, modeling the input and output formats using Pydantic
- store.py, for initializing, logging operations and interrogating the history of operations
- cli.py / main.py, to be started from the command line using the commands in the *commands.txt* files in each directory

A fifth file is present for the FastAPI versions, highlighting the improvements made by the asynchronous implementation when multiple requests are sent to the server, each with increasing degrees of complexity.

Requirements can be installed with *pip install -r requirements.txt* for each project. Pydantic and Flake8 are common for all versions, while the other packages are specific to each version:

- click for the command line implementation
- fastapi, uvicorn – for web server implementations, and requests for the synchronous implementation
- fastapi, uvicorn, aiosqlite – for asynchronous access to databases, aiohttp – for asynchronous requests and asyncio for the asynchronous implementation

The API versions use a REST API built with FastAPI, with POST methods for the operations and GET for retrieving their history.

The application can be tested by going to the address mentioned in the *commands.txt* file after starting the server. It is under the basic FastAPI UI and each method can be tested by opening its respective drop down menu, clicking on *"Try it out"*, changing the parameters (if necessary) and then pressing the *"Execute"* button. Then, below, the result will be visible.

Finally, despite the fact that asynchronous behavior intuitively shouldn't bring improvements in this case, since it was meant for I/O tasks like downloading files and these are just CPU operations, it does bring improvements of a few seconds in the Fibonacci case (between 1-4 seconds down from the 30 seconds needed in the synchronous behavior for the first 40 Fibonacci numbers). This is possibly due to the fact that each operation is slow due to the recursive implementation, so when one is finished, the large number is written to the database and another coroutine can start, whereas the other operations are instantaneous computations.