Exercise: Building a Rate-Limited Asynchronous Task Manager API with Polling

Overview

In this exercise, you'll build a RESTful API using FastAPI for managing asynchronous tasks. The API should allow users to submit tasks, poll for their status, and handle task processing in the background using an in-memory queue. All data must be stored in-memory (no databases), and the application should include rate limiting to prevent abuse.

Use Python 3.10 or later, leveraging features like structural pattern matching where appropriate.

This tests your skills in asynchronous programming, API design, concurrency, and error handling. Aim to complete it in 4-8 hours. Provide clean, well-documented code with tests.

Requirements

- **Libraries**: Use FastAPI for the API, `asyncio` for queues and async operations, and standard libraries like `uuid` and `collections`. No external dependencies beyond these.
- **In-Memory Storage**: Use a dictionary to store task details and an `asyncio.Queue` for pending tasks. Data is volatile and resets on restart.

1. **API Endpoints**:
    - `POST /tasks`: Submit a task.
        - Body: JSON with `task_type` (e.g., "compute_sum", "generate_report"), `parameters` (dict, e.g., {"numbers": [1, 2, 3]}). **Note:** The above examples are perfectly fine, but you are not required to use these.
        - Response: 202 Accepted with `task_id` (UUID) and status "queued". Validate inputs; return 400 on errors.
    - `GET /tasks/{task_id}`: Poll task status.
        - Returns: JSON with `status` ("queued", "processing", "completed", "failed"), `result` (if completed), `error` (if failed).
        - Optional: ?wait=true for long-polling (wait up to 10 seconds for status change).
    - `GET /tasks`: List tasks.
        - Optional filters: ?status=queued&limit=10.
        - Stream response if many tasks (use async generator).
    - `DELETE /tasks/{task_id}`: Cancel a task (set to "cancelled" if possible).
2. **Core Functionality**:
    - **Task Queuing and Processing**: Use `asyncio.Queue` for task submission. Run a background worker loop (via `asyncio.create_task`) that dequeues tasks, processes them asynchronously (simulate work with `asyncio.sleep` for 5-30 seconds based on type, using `match` for logic: e.g., sum numbers, generate fake report string, or raise errors for simulation). Update status in an in-memory dict (key: task_id, value: dict with status, parameters, result).

    - **Polling Mechanism**: For `/tasks/{task_id}`, if ?wait=true, use `asyncio.Event` per task to notify on status changes, awaiting with timeout. This demonstrates efficient long-polling without busy-waiting.
    - **Rate Limiting**: Limit to 10 requests/minute per IP. Track in an in-memory dict with `collections.deque` for timestamps. Use `asyncio.Lock` for thread-safety. Return 429 on exceedance.
    - **In-Memory Management**: Dict for task storage, queue for pending tasks. Implement auto-cleanup: remove completed/failed tasks after 10 minutes using a separate async timer task.
    - **Concurrency**: Process up to 5 tasks in parallel using `asyncio.gather`. Handle cancellations gracefully (e.g., via `asyncio.CancelledError`).
3. **Additional Features**:
    - Use Python 3.10+ specifics: `match` for task_type processing (e.g., match task_type: case "compute_sum": ...), parenthesized context managers for locks, or type aliases for task dicts.
    - Error Handling: Custom async exceptions (e.g., TaskFailedError), logging, and detailed JSON responses. Use `match` for error classification.
    - Testing: Async tests covering submission, polling (with mocks for sleeps), rate limiting, cancellations, and concurrency (e.g., simulate multiple submissions).
    - Documentation: README.md with setup instructions, API docs (using FastAPI's Swagger).
4. **Constraints and Best Practices**:
    - No DB or external queues; all in-memory.
    - Code modular, PEP 8 compliant, with type hints.
    - Handle edges: Invalid parameters, queue full (e.g., max 100 tasks, reject with 503), concurrent polls.
    - Bonus: Add progress tracking (e.g., percentage in status for long tasks) or integrate a simple priority system to the queue.

Submission Guidelines

- Full codebase in Git repo or ZIP.
- Include requirements.txt, setup: `pip install -r requirements.txt`, run with `uvicorn main:app --reload`.

Explain assumptions and anything relevant in a README.md