Hiring Challenge - HTTP Metadata Inventory Service:

Technical Requirements

Component	Technology	Role
Language	Python 3.11+	Core logic and scripting.
Web Framework	FastAPI	API development and documentation.
Database	MongoDB	Document storage for metadata and page sources.
Orchestration	Docker Compose	Local development and environment isolation.
Testing	Pytest	Unit and integration testing.

NOTE: Docker Compose is mandatory. The service must be able to run `docker-compose up` to start both the API and the database.

Coding Challenge

1. **POST Endpoint:** Create a metadata record for a given URL.
    - **Input:** A URL (e.g., `https://example.com`).
    - **Action:** Collect the headers, cookies, and page source of the URL.
    - **Storage:** Store the collected data in MongoDB.

2. **GET Endpoint:** Retrieve metadata for a given URL.
    - **Input:** A URL (e.g., `https://example.com`).
    - **Workflow:**
        - **Inventory Check:** Determine if the requested URL's metadata is currently present in the database.
        - **Immediate Resolution:** If the record exists, return the full dataset (headers, cookies, and page source).
        - **Conditional Inventory Update:** If the record is missing, the system must prioritise request responsiveness. Provide an immediate acknowledgement (e.g., `202 Accepted`) to the user that the request has been logged for future availability.

3. **Background Worker Logic:** For records not found during a GET request, the service must handle metadata collection asynchronously:
    - **Trigger:** The collection must be initiated internally by the GET endpoint when a cache miss occurs.
    - **Asynchronous Execution:** Initiate an internal process to update the inventory for that specific URL. This process must run independently of the request-response cycle and must not delay the API's response.
    - **Architectural Constraint:** The inventory update must be handled through internal logic orchestration. Avoid loops or external service-to-self HTTP calls.
    - **Result Persistence:** Ensure that once the background process concludes, the data is available for all subsequent retrieval attempts by the GET endpoint.

Implementation Guidelines

- **System Resilience:** Ensure the application remains stable during external service fluctuations or database startup delays.
- **Configuration:** Follow modern practices for managing application settings and external resource identifiers (e.g., environment variables).
- **Resource Management:** The solution should demonstrate efficient handling of I/O-bound tasks and system resources.
- **Code Architecture:** Organise the codebase to support future growth, prioritising modularity and a clear separation between data, logic, and transport layers.
- **Scope:** Focus on retrieving static content and metadata; advanced client-side rendering (JavaScript execution) is out of scope for this evaluation.

Deliverables

- A link to the GitHub Repository.
- A functional `docker-compose.yml` (API + MongoDB).
- A comprehensive test suite using `pytest`.

Candidate Evaluation Rubric

We evaluate submissions based on technical proficiency, code quality, and architectural thinking appropriate for the seniority level. Below are the criteria we use to review your challenge:

1. Functionality & Correctness

- Does the `docker-compose up` command result in a fully working environment?
- Are all endpoints (POST and GET) implemented according to the specifications?
- Does the background collection logic work seamlessly without blocking the API response?

2. Code Quality & Standards

- **Readability:** Is the code clean, well-commented, and following PEP8 standards?

- **Modern Python:** Are you utilising type hints and Pydantic models for data validation?
- **Error Handling:** How does the application handle invalid URLs, timeouts, or database connection issues?

3. Database & Performance

- **Schema Design:** Is the metadata structured logically in MongoDB?
- **Efficiency:** Are lookups optimised (e.g., indexing) to ensure the system remains fast as the dataset grows?

4. System Design & Scalability

- **Separation of Concerns:** Is there a clear distinction between the API layer and the business logic?
- **Containerization:** Is the Docker setup efficient, secure, and easy to maintain?
- **Extensibility:** How easy would it be to extend this service or move components into a distributed architecture in the future?

5. Documentation

- Does the README clearly explain how to run, test, and interact with the API?
- Is the API documented (e.g., via FastAPI's automatic Swagger/OpenAPI UI)?