Technical Challenge: Backend Developer

Develop an HTTP API using Python that allows consumers to interact with a database, providing use domain abstractions, treated as HTTP resources. The code should be hosted in a **GitHub as a public repo**, and the dataset should be included in the repository or accessible via a provided link.

The application to be developed should read data from a database, perform some transformation on the data based on a business requirement and make the transformed data conveniently accessible through an HTTP API. The API should allow for all CRUD operations.

Regarding the DB, a from-scratch design could be used as well as using some public DB. The pre-requisite is that the DB has at least 4 tables and that the structure allows the completion of the mandatory features.

Before the technical interview, the candidate must provide a URL for the public GitHub repository, and the dataset should be included in the repo. The repo's README should contain relevant context and necessary information to understand how a typical consumer would use the service and the steps taken during the development process. GitHub is not meant to be used as a hosting solution but as source control, meaning that a clean commit history reflecting the progress is expected.

Clarification: the usage of GenAI tools (e.g., ChatGPT, Cursor, TabNine, Github Copilot) is allowed and it is expected to streamline repetitive and/or easy to automate processes such as the following:

- Generate descriptive git commit messages.
- Generate docstrings for functions.
- Generate drafts for tests and boilerplate code.
- Use as a peer review tool to identify potential flaws.
- Identify which design patters will be most suitable for a particular functionality

Feel free to experiment and leverage these tools to enhance your work.

Mandatory Features:

- Use FastAPI as the framework for the application
- All endpoints should:
  - Return JSON objects
  - Use proper HTTP Status codes
  - Define all CRUD operations
  - Have a Swagger example
- Document the DB used with suitable diagrams using Mermaid
- Document a potential integration with another system using a Sequence diagram in Mermeid
- Design the application following a Layered Architecture. Having at least the following layers: Presentation (*Routes/Views*), Application (*Controller/Services*), Domain (*Entities*) and Persistence (*Data*).
- The Service layer should perform some business transformation on the data. Write the requirement that justifies that transformation.

- The resources exposed through the HTTP API should not be the same as the one in the DB. I.e. there should be a data transformation from the entities in the DB to the ones returned in the API. For example: API returns JSON in a denormalized way with and the DB stores data in a normalized way.
- The code should use Type-Hints throughout.
- Interactions with the database should be done through an ORM (e.g., SQLModel, SQLAlchemy)
- The API should be stateless and prepared to handle multiple (hundreds) of users concurrently.
- In addition to the CRUD endpoints, there should be one to use as a healthcheck and one for the version. Both unprotected.

Optional Features (include at least 1):

1. Make the CRUD operations work for single entities and for batches of entities.
2. Lock the application dependencies (e.g., pipenv, poetry, pdm).
3. Implement logging throught the codebase to document each step of the process.
4. Use the Problem Details standard (RFC 9457) to show errors.
5. Add a middleware to include a autogenerated UUID for correlation id.
6. Add an endpoint that processes a CSV file,
7. Implement pagination for some endpoint, possible pagination strategies are:
    a. Fixed size pagination i.e., using offset and limit
    b. Cursor based pagination. I.e., using page size and page token
8. Add testing with Pytest:
    a. Add unit tests, leveraging fixtures, mocks, and patches
    b. Add integration tests by using a TestClient
    c. Add tests with the database by leveraging Fakes
9. Use dependency injection to deal with external systems like the DB.
10. Use the repository pattern with Generics to manage queries to the database.
11. Use the inversion of control pattern.
12. Separate Database entities from Domain entities by leveraging Domain Driven Design.
13. Add quality gates using pre-commits to the codebase:
    a. Formatting and Linting with Ruff
    b. Linting with Pylint
    c. Type checking with Mypy
14. Make a docker container to easily deploy the application
15. Use a hosted DB (e.g Supabase)
16. Enhance the retrieve endpoints by adding the possibility to filter records by attributes
17. Implement rate limitting.
18. Add a caching methodology to an expensive operation.
19. Protect the service by using Authentication and Authorization
20. Make the HTTP API RESTFUL following Richardson levels:
    a. Use HTTP Resources
    b. Use HTTP Verbs
    c. Implement HATEOAS
21. Analyse metrics about the application performance (e.g., Apache ab, locust, hey, break, vegeta, siege).

22. Add versioning to the API, that is, having different endpoints for different versions. E.g., /api/v1/ and /api/v2/
23. Add a simple UI showcasing the service functionality and value. This could be done by using any of the following:
    a. Python tools that do not require HTML/CSS/JS - e.g., Streamlit.
    b. Pure HTML / CSS / JS.
    c. Backend-centered libraries for building frontends – e.g., HTMX.
    d. Lightweight component libraries – e.g., Lit.
    e. Fully Fledge Front-end frameworks/libraries - Angular, React or Vue.
24. In addition to HTTP REST, add a GraphQL endpoint for querying the data. Provide examples of useful queries.
25. Implement the CQRS pattern
26. Use a database migration tool (e.g Alembic)

Evaluation criteria:

The project will be evaluated in four dimensions:

- **Best practices**: a code review to evaluate best practices, patterns, and overall design.
- **Complexity**: the level of optional features implemented and how they were incorporated into the development process.
- **Time-Management**: the most important aspect is to have something functional by the deadline; the candidate should be able to organize themselves and deliver something valuable.
- **Innovation**: new features outside of what has been asked are a plus.

References

Here are some useful links and tutorials to assist you in completing the technical challenge:

1. FastAPI:
   a. Sebastián Ramírez: Building REST APIs with FastAPI | Real Python
   b. First Steps
2. Designing REST APIs and HATEOAS:
   a. Derek Comartin - How to (and how not to) design REST APIs – Code Opinion
   b. Derek Comartin - Want to build a good API? Here's 5 Tips for API Design. - Code Opinion
   c. Jason Desrosiers - Turning Up The Good On REST APIs – 8th Light
   d. Jason Desrosiers - Decoupling the Client and Server with Hypermedia – 8th Light
   e. Jason Desrosiers - The Hypermedia Maturity Model – 8th Light
3. Testing:
   a. John Leeman, Ryan May - Testing your Python Code with PyTest - Scipy 2019
   b. María Andrea Vignau - Patch, Stub y Mock - PyCon US 2022
   c. Harry Percival - Stop Using Mocks (for a while) - PyCon US 2020
4. Domain Driven Design:
   a. Arjan: Deep Dive Into the Repository Design Pattern in Python
5. Hosted Database:
   a. Patrick Loeber: Supabase Crash Course For Python Developers
6. Dependency Injection:
   a. Arjan: Dependency Injection Explained in 7 Minutes
   b. Arjan: Dependency INVERSION vs Dependency INJECTION in Python
7. Inversion of Control:
   a. Arjan: Should You Use Dependency Injection Frameworks?
8. API Versioning:
   a. Stanislav Zmiev - Versioning APIs - Talk Python to Me Ep.450