Prueba Técnica - Desarrollador Backend Python (FastAPI)

Propósito: construir una API REST funcional, segura y mantenible. La prueba está diseñada para evaluar tu capacidad de entregar algo que corre end-to-end, y también tu criterio para tomar decisiones técnicas.

Nota: *buscamos una solución* ***robusta****, pero no esperamos perfección. Algunas decisiones quedan a tu criterio; lo importante es que las hagas explícitas en el README.*

1) Alcance y reglas

- Tiempo recomendado: **2 a 4 horas**. Si no alcanzas todo, prioriza: **auth + CRUD + paginación + README**.
- Entrega en **repositorio público de GitHub**.
- Tu solución debe poder ejecutarse en un equipo nuevo siguiendo tu README (sin pasos “mágicos”).
- Solo entorno local para la base de datos (PostgreSQL en Docker).

2) Stack requerido

- Python 3.11.8
- FastAPI
- SQLAlchemy
- PostgreSQL
- Autenticación JWT
- Hash seguro de contraseñas

3) Base de datos (local con Docker)

Configura PostgreSQL exclusivamente en entorno local usando **docker-compose**.

Variables de entorno (ejemplo):

```
DB_HOST=localhost
DB_PORT=5432
DB_NAME=technical_test
DB_USER=postgres
DB_PASSWORD=postgres
```

4) Autenticación

Implementa un endpoint mínimo requerido:

POST /auth/login

- Autenticación basada en **JWT**.
- Contraseñas almacenadas con **hash seguro** (ej. bcrypt/argon2).
- Expiración del token **configurable** (variable de entorno o config).
- Debe existir un **usuario inicial** creado automáticamente (no manual).

A tu criterio: define si el usuario se identifica por email/username y el formato del payload del login. En el README deja las credenciales del usuario inicial y cómo se crea (migración/seed).

5) Entidad principal: Task

La entidad principal del sistema es **Task**. Define al menos estos campos (puedes agregar más):

- id
- title (requerido)
- description (opcional)
- status (pending/in_progress/done o equivalente)
- created_at

Se valorará que definas **índices** en campos relevantes. Elige cuáles y justifica brevemente en el README.

6) Funcionalidades requeridas

- CRUD completo de tareas.
- Listado con paginación.
- Manejo adecuado de errores HTTP (por ejemplo: 400/401/404/422).
- Migraciones para crear tablas necesarias.

A tu criterio: define nombres exactos de endpoints y parámetros (por ejemplo, page/page_size). Lo importante es que haya paginación real y sea consistente.

7) Migraciones y datos iniciales

Usa migraciones (por ejemplo Alembic) para:

- Crear las tablas necesarias.
- Crear el usuario inicial (obligatorio).
- Insertar seed de datos (opcional, pero suma puntos).

8) Arquitectura sugerida (no obligatoria)

No exigimos una estructura específica, pero evaluamos positivamente una arquitectura clara, modular y escalable.

```
app/
api/ # routers / endpoints
core/ # configuración, seguridad, auth
db/ # sesión, conexión, migraciones
models/ # modelos SQLAlchemy
schemas/ # esquemas Pydantic
services/ # lógica de negocio
main.py
```

9) Entregables

- Repositorio público en GitHub con el código fuente completo.
- README.md con: descripción, tecnologías, instrucciones de ejecución, variables de entorno, cómo levantar PostgreSQL con Docker, y ejemplos de uso (curl o Postman).
- Incluir docker-compose.yml para PostgreSQL.

10) Criterios de evaluación (resumen)

Criterio	Qué buscamos
Funciona end-to-end	Se puede correr siguiendo el README, sin ajustes manuales.
Seguridad básica	JWT bien aplicado, hash seguro, endpoints protegidos.
Calidad técnica	Código legible, manejo de errores, consistencia en la API.
Persistencia	SQLAlchemy + Postgres funcionando, migraciones claras.
Criterio y comunicación	Decisiones explicadas (trade-offs) en README.