Fase 1: Fundamentos de Python 🐍

Esta es la base sobre la que construirás toda tu carrera. Un dominio profundo aquí te diferenciará enormemente. No tengas prisa.

Sintaxis y Tipos de Datos

- **Concepto Clave y Relevancia Profesional:** Es el vocabulario y la gramática de Python. Sin esto, no puedes comunicarle al ordenador la lógica de negocio más básica. Las empresas necesitan que escribas código claro, eficiente y libre de errores sintácticos obvios desde el primer día.
- **Objetivos de Aprendizaje Concretos:**
  - Implementar un script que utilice al menos tres tipos de bucles (for, for anidado, while) para procesar una lista de diccionarios.
  - Escribir funciones que reciban diferentes tipos de datos (números, strings, booleanos) y usen operadores lógicos y de comparación para devolver un resultado.
  - Refactorizar un bloque de `if/elif/else` anidado en una versión más legible y eficiente, posiblemente usando diccionarios para el despacho.
- **Temas a Dominar (El "Qué"):**
  - Variables y asignación.
  - Tipos de datos primitivos: `int`, `float`, `str`, `bool`, `None`.
  - Operadores: aritméticos (`+`, `*`, `%`), de comparación (`==`, `!=`, `>`), lógicos (`and`, `or`, `not`).
  - Control de flujo: `if`, `elif`, `else`.
  - Bucles: `for` (con `range`, sobre iterables), `while`.
  - Palabras clave de control de bucles: `break`, `continue`, `pass`.
  - F-strings para formateo de cadenas.
- **Mejores Prácticas y Errores Comunes:**
  - **Mejor Práctica:** Usa siempre f-strings (`f"Hola, {variable}"`) para formatear cadenas en Python 3.6+. Es más rápido y legible que `str.format()` o el viejo operador `%`.
  - **Error Común:** Comparar un booleano con `== True` o `== False` (ej. `if mi_variable == True:`). Es redundante. Simplemente escribe `if mi_variable:` o `if not mi_variable:`.
- **Mini-Proyecto o Tarea Práctica:**
  - Escribe un script simple de "Adivina el número". El programa elige un número aleatorio entre 1 y 100, y el usuario tiene 10 intentos para adivinarlo. El programa debe dar pistas ("más alto", "más bajo") después de cada intento.

Estructuras de Datos a fondo

- **Concepto Clave y Relevancia Profesional:** Son las herramientas para organizar y almacenar colecciones de datos de manera eficiente. La elección de la estructura de datos incorrecta puede degradar el rendimiento de una aplicación de O(1) a O(n), lo cual es catastrófico a escala. Las empresas valoran a los ingenieros que entienden estas implicaciones.
- **Objetivos de Aprendizaje Concretos:**
  - Resolver un problema que requiera el uso de un `set` para encontrar elementos únicos o comunes entre dos listas.
  - Implementar una función que transforme una lista de tuplas en un diccionario usando una comprensión de diccionario.

    - Explicar la diferencia de rendimiento entre buscar un elemento en una lista y en un diccionario/set.
- **Temas a Dominar (El "Qué"):**
    - **Listas:** Métodos (`append`, `pop`, `remove`, `sort`), slicing, mutabilidad.
    - **Diccionarios:** Claves y valores, métodos (`get`, `keys`, `values`, `items`), cómo manejar `KeyError`.
    - **Tuplas:** Inmutabilidad, casos de uso (claves de diccionario, devolver múltiples valores desde una función).
    - **Sets:** Operaciones de conjuntos (`union`, `intersection`), casos de uso (eliminar duplicados, pruebas de membresía rápidas).
    - **Comprensiones:** `List comprehensions`, `dict comprehensions` para crear estructuras de datos de forma concisa y performante.
- **Mejores Prácticas y Errores Comunes:**
    - **Mejor Práctica:** Utiliza el método `diccionario.get('clave', valor_por_defecto)` en lugar de acceso directo `diccionario['clave']` cuando no estés seguro de si una clave existe, para evitar un `KeyError`.
    - **Error Común:** Modificar una lista mientras se itera sobre ella. Esto puede llevar a comportamientos inesperados y errores. Si necesitas hacerlo, itera sobre una copia: `for item in mi_lista[:]:`.
- **Mini-Proyecto o Tarea Práctica:**
    - Crea una función que reciba un texto largo como string y devuelva un diccionario con las 10 palabras más frecuentes y su conteo. Usa listas, diccionarios y sets para lograrlo eficientemente.

Programación Orientada a Objetos (POO)

- **Concepto Clave y Relevancia Profesional:** POO es un paradigma para estructurar el código en objetos reutilizables que contienen tanto datos (atributos) como comportamientos (métodos). Es la base de frameworks como Django y es fundamental para construir sistemas complejos, mantenibles y escalables.
- **Objetivos de Aprendizaje Concretos:**
    - Implementar una clase base `Vehiculo` y dos clases derivadas (`Coche`, `Bicicleta`) que hereden de ella, sobrescribiendo un método (`mover()`) para demostrar polimorfismo.
    - Crear una clase con atributos de instancia, un método de clase para llevar la cuenta de cuántas instancias se han creado, y un método estático de utilidad que no dependa del estado de la clase o la instancia.
    - Implementar los métodos mágicos `__str__` y `__repr__` en una clase y explicar cuándo se invoca cada uno.
- **Temas a Dominar (El "Qué"):**
    - Clases y objetos (instanciación).
    - Atributos de instancia vs. atributos de clase.
    - Métodos de instancia (`self`), métodos de clase (`@classmethod`), métodos estáticos (`@staticmethod`).
    - Pilares: Encapsulamiento (atributos `_privados` y `__muy_privados`), Herencia (simple y múltiple), Polimorfismo.
    - Métodos Mágicos (Dunders): `__init__`, `__str__`, `__repr__`, `__eq__`.
- **Mejores Prácticas y Errores Comunes:**
    - **Mejor Práctica:** Sigue el principio de Responsabilidad Única (SRP). Cada clase debe tener una única y bien definida responsabilidad. Clases que hacen "de todo" son una pesadilla para

mantener.
    - **Error Común:** Usar un tipo mutable (como una lista o diccionario) como valor por defecto en la definición de un método: `def __init__(self, items=[])`. Este valor por defecto se comparte entre TODAS las instancias de la clase. El valor por defecto correcto es `None` y se inicializa dentro del método: `self.items = items or []`.
- **Mini-Proyecto o Tarea Práctica:**
    - Modela un sistema simple de una biblioteca. Crea clases para `Libro`, `Autor` y `Biblioteca`. La clase `Biblioteca` debe tener métodos para añadir libros, prestarlos y ver el catálogo disponible.

Manejo de Errores y Excepciones

- **Concepto Clave y Relevancia Profesional:** Es el mecanismo para manejar condiciones inesperadas o errores durante la ejecución de un programa sin que este se caiga. Un backend robusto no crashea; anticipa fallos (conexiones de red, datos inválidos) y los maneja elegantemente, logueando el error y devolviendo una respuesta apropiada al cliente.
- **Objetivos de Aprendizaje Concretos:**
    - Escribir una función que lea un archivo y use `try/except` para manejar el `FileNotFoundError` y `try/finally` para asegurar que el archivo se cierre.
    - Crear una excepción personalizada, como `SaldoInsuficienteError`, que herede de `Exception` y lanzarla en una clase `CuentaBancaria` cuando se intente retirar más dinero del disponible.
    - Refactorizar un código que usa múltiples `if` para chequear errores y en su lugar usar un bloque `try/except` para un flujo más limpio.
- **Temas a Dominar (El "Qué"):**
    - La diferencia entre un error de sintaxis y una excepción.
    - Bloques `try`, `except Exception as e`.
    - Manejar excepciones específicas (`ValueError`, `TypeError`, `KeyError`).
    - Uso de los bloques `else` (se ejecuta si no hay excepción) y `finally` (se ejecuta siempre).
    - La palabra clave `raise` para lanzar excepciones.
    - Creación de excepciones personalizadas heredando de `Exception`.
- **Mejores Prácticas y Errores Comunes:**
    - **Mejor Práctica:** Sé específico en tus bloques `except`. Evita usar un `except Exception:` genérico que pueda ocultar errores que no esperabas. Atrapa solo las excepciones que sabes cómo manejar.
    - **Error Común:** Usar excepciones para control de flujo normal. Si puedes usar un `if` para chequear una condición, es casi siempre preferible y más performante que usar un `try/except`. Las excepciones son para casos excepcionales.
- **Mini-Proyecto o Tarea Práctica:**
    - Crea una función que reciba un diccionario y una clave. La función debe intentar acceder al valor de esa clave. Si la clave no existe (`KeyError`), debe loguear un warning y devolver `None`. Si el valor asociado a la clave no es un número (`TypeError`), debe lanzar una excepción personalizada `InvalidDataError`.

Fase 2: Entornos Virtuales y Gestión de Proyectos con Poetry

Aquí es donde pasas de escribir scripts a construir aplicaciones profesionales. Es un paso no negociable en la industria.

Poetry: Gestión Moderna de Dependencias

- **Concepto Clave y Relevancia Profesional:** Poetry es una herramienta que gestiona las librerías de terceros (dependencias) de tu proyecto y crea entornos aislados para cada uno. Resuelve el "en mi máquina funciona" al asegurar que todos los desarrolladores y el servidor de producción usen exactamente las mismas versiones de las librerías, evitando conflictos y asegurando reproducibilidad.
- **Objetivos de Aprendizaje Concretos:**
  - Inicializar un nuevo proyecto usando `poetry new`.
  - Añadir dependencias de producción (`fastapi`, `uvicorn`) y de desarrollo (`pytest`, `ruff`) usando `poetry add`.
  - Explicar el propósito del archivo `poetry.lock` y cómo regenerarlo si es necesario.
  - Ejecutar un script del proyecto usando `poetry run python mi_script.py`.
- **Temas a Dominar (El "Qué"):**
  - El problema que resuelven los entornos virtuales (`venv`, `virtualenv`).
  - Comandos esenciales: `poetry new`, `poetry init`, `poetry install`, `poetry add`, `poetry remove`, `poetry run`, `poetry shell`, `poetry update`.
  - El archivo `pyproject.toml`: su estructura, la sección `[tool.poetry]`, dependencias (`dependencies`) y dependencias de desarrollo (`group.dev.dependencies`).
  - El archivo `poetry.lock`: su rol como "la única fuente de la verdad" para las versiones exactas de las dependencias.
- **Mejores Prácticas y Errores Comunes:**
  - **Mejor Práctica:** Siempre haz "commit" de tu archivo `poetry.lock` a tu repositorio de Git. Esto garantiza que cualquier persona que clone tu proyecto (incluyendo el sistema de CI/CD) instale las mismas versiones exactas de las dependencias que tú.
  - **Error Común:** Instalar dependencias manualmente con `pip` dentro de un entorno gestionado por Poetry. Esto rompe el propósito de Poetry y puede llevar a inconsistencias. Usa siempre `poetry add`.
- **Mini-Proyecto o Tarea Práctica:**
  - Crea un nuevo proyecto con Poetry. Añade `requests` como dependencia. Escribe un script simple que use `requests` para hacer una petición a la API de `https://pokeapi.co/api/v2/pokemon/ditto` y muestre el nombre del Pokémon en la terminal. Ejecútalo con `poetry run`.

Fase 3: APIs de Alto Rendimiento con FastAPI y Pydantic ⚡

Este es el núcleo del stack moderno. Dominar FastAPI te posiciona en la vanguardia del desarrollo backend con Python.

Importancia de FastAPI

- **Concepto Clave y Relevancia Profesional:** FastAPI es un framework web de alto rendimiento para construir APIs, basado en las modernas declaraciones de tipo de Python. Las empresas lo adoptan por su increíble velocidad (comparable a NodeJS y Go), su robustez gracias a la validación de datos automática, y la reducción drástica en el tiempo de desarrollo.
- **Objetivos de Aprendizaje Concretos:**
  - Explicar qué es ASGI y cómo contribuye al alto rendimiento de FastAPI.

    - Crear una API simple y observar cómo los "type hints" en una función se convierten en validación de datos y documentación automática.
    - Comparar la cantidad de código necesario para un endpoint simple en FastAPI versus Flask o Django REST Framework.
- **Temas a Dominar (El "Qué"):**
    - ASGI vs WSGI: La diferencia fundamental (asíncrono vs síncrono).
    - Starlette: El microframework ASGI sobre el que se construye FastAPI.
    - Pydantic: La librería de validación de datos que potencia a FastAPI.
    - Declaraciones de tipo (Type Hints) y su impacto directo.
- **Mejores Prácticas y Errores Comunes:**
    - **Mejor Práctica:** Aprovecha al máximo los `type hints`. No solo para los modelos de Pydantic, sino para todas tus funciones. Mejora la legibilidad, permite el análisis estático y reduce errores.
    - **Error Común:** Mezclar código síncrono bloqueante (como `requests.get()`) en un endpoint `async def` sin el debido cuidado. Esto anula los beneficios del asincronismo. Para operaciones de I/O, usa librerías asíncronas (como `httpx`).
- **Mini-Proyecto o Tarea Práctica:**
    - Instala `fastapi` y `uvicorn`. Crea un archivo `main.py` con el "Hello World" de FastAPI y lánzalo con `uvicorn main:app --reload`. Accede a `http://127.0.0.1:8000` en tu navegador.

Path Operations

- **Concepto Clave y Relevancia Profesional:** Son las funciones que se ejecutan cuando un cliente hace una petición a una URL específica (`/users`, `/items/{item_id}`) con un método HTTP concreto (`GET`, `POST`, `DELETE`). Es la forma fundamental de definir la funcionalidad y los recursos de tu API.
- **Objetivos de Aprendizaje Concretos:**
    - Crear un endpoint `GET` que acepte un parámetro de ruta (`/items/{item_id}`).
    - Crear un endpoint `GET` que acepte parámetros de consulta opcionales (`/search?q=query`).
    - Implementar un endpoint `POST` que reciba un cuerpo JSON y lo valide.
    - Diferenciar cuándo usar un parámetro de ruta, de consulta o el cuerpo de la petición.
- **Temas a Dominar (El "Qué"):**
    - Decoradores: `@app.get()`, `@app.post()`, `@app.put()`, `@app.delete()`.
    - Parámetros de Ruta: `path: str`, `item_id: int`.
    - Parámetros de Consulta (Query): `skip: int = 0`, `limit: int = 100`.
    - Cuerpo de la Petición (Body): Usando modelos de Pydantic.
    - Combinación de parámetros en una sola función.
    - Códigos de estado HTTP (`status_code=201`).
- **Mejores Prácticas y Errores Comunes:**
    - **Mejor Práctica:** Usa sustantivos en plural para tus rutas de colección (`/users`, `/items`) y un identificador para el recurso específico (`/users/{user_id}`). Sigue las convenciones REST.
    - **Error Común:** Poner un parámetro de ruta opcional o con valor por defecto. Los parámetros de ruta, por definición, son parte de la identidad de la URL y no pueden ser opcionales. Los parámetros de consulta sí son ideales para filtros opcionales.
- **Mini-Proyecto o Tarea Práctica:**
    - Crea una API de "lista de tareas" en memoria (usando una lista de diccionarios). Implementa endpoints para:
        - `GET /tasks`: Devolver todas las tareas.
        - `POST /tasks`: Añadir una nueva tarea.

    - `GET /tasks/{task_id}`: Devolver una tarea específica.

Pydantic para Validación de Datos

- **Concepto Clave y Relevancia Profesional:** Pydantic usa los `type hints` de Python para validar, serializar y deserializar datos de forma automática y estricta. Esto elimina una enorme cantidad de código "boilerplate" de validación manual, previene incontables bugs por datos malformados y sirve como la "fuente de la verdad" para la estructura de tus datos.
- **Objetivos de Aprendizaje Concretos:**
  - Definir un modelo `BaseModel` de Pydantic para representar los datos de entrada de un endpoint `POST`.
  - Utilizar tipos de datos avanzados de Pydantic como `EmailStr`, `HttpUrl` o `Field` para validaciones más complejas (ej. `max_length`).
  - Crear un modelo con campos opcionales (`Optional[str] = None`) y valores por defecto.
  - Interpretar los errores de validación que FastAPI devuelve automáticamente cuando se envían datos incorrectos.
- **Temas a Dominar (El "Qué"):**
  - Herencia de `pydantic.BaseModel`.
  - Tipos de datos básicos (`str`, `int`, `float`, `bool`).
  - Tipos complejos: `List`, `Dict`, `Optional`.
  - Validadores personalizados con `@validator`.
  - Uso de `Field` para añadir metadatos (ejemplos, descripciones, restricciones de longitud).
  - Modelos anidados (un modelo de Pydantic dentro de otro).
  - Control de la salida del modelo con `response_model`.
- **Mejores Prácticas y Errores Comunes:**
  - **Mejor Práctica:** Crea modelos de Pydantic específicos para la entrada (`ItemCreate`) y la salida (`ItemRead`). El modelo de salida puede tener campos adicionales que no quieres que el cliente envíe (como `id` o `created_at`).
  - **Error Común:** Devolver directamente un modelo de la base de datos (como un objeto SQLAlchemy) que contiene campos que no deberían ser expuestos. Usar un `response_model` de Pydantic te fuerza a ser explícito sobre qué datos se envían al cliente.
- **Mini-Proyecto o Tarea Práctica:**
  - En tu API de "lista de tareas", reemplaza los diccionarios con modelos de Pydantic. Crea un `TaskCreate` para el endpoint `POST` y un `TaskRead` (que incluya un `id`) para los endpoints `GET`.

Dependency Injection

- **Concepto Clave y Relevancia Profesional:** Es un patrón de diseño donde los componentes (dependencias) que una función necesita para operar son "inyectados" desde fuera en lugar de ser creados dentro de ella. En FastAPI, esto permite una increíble reutilización de código para lógica como conexiones a bases de datos, autenticación de usuarios o recuperación de parámetros comunes.
- **Objetivos de Aprendizaje Concretos:**
  - Crear una función "dependable" que extraiga parámetros de consulta comunes (paginación: `skip`, `limit`) y usarla en múltiples endpoints.
  - Implementar una dependencia para obtener una sesión de base de datos y asegurar que se cierre correctamente después de cada petición.

    - Crear una dependencia que verifique una cabecera `X-API-Key` y la use para proteger un endpoint.
- **Temas a Dominar (El "Qué"):**
    - La función `Depends` de FastAPI.
    - Dependencias como funciones normales.
    - Dependencias basadas en clases.
    - Dependencias con `yield` para ejecutar código de "limpieza" (ej. cerrar conexión a BD).
    - Sub-dependencias (una dependencia que depende de otra).
    - Dependencias a nivel de router (`APIRouter`).
- **Mejores Prácticas y Errores Comunes:**
    - **Mejor Práctica:** Mantén tus dependencias pequeñas y enfocadas en una sola tarea (Principio de Responsabilidad Única). Una dependencia para la sesión de BD, otra para el usuario actual, etc.
    - **Error Común:** Poner lógica de negocio compleja dentro de una dependencia. Las dependencias son para "plomería" (obtener cosas, verificar permisos), no para ejecutar la lógica central de la aplicación. La lógica de negocio debe permanecer en la función del path operation.
- **Mini-Proyecto o Tarea Práctica:**
    - Crea una dependencia `async def get_db_session()` (por ahora puede ser una función falsa que no haga nada). Inyéctala en tus endpoints de la API de tareas. Más adelante, la conectarás a una base de datos real.

Documentación Automática (Swagger/ReDoc)

- **Concepto Clave y Relevancia Profesional:** FastAPI genera automáticamente una documentación interactiva de tu API basada en tus `path operations`, modelos de Pydantic y `type hints`. Esto ahorra incontables horas de documentación manual, sirve como un contrato claro para los desarrolladores de frontend y permite probar los endpoints directamente desde el navegador.
- **Objetivos de Aprendizaje Concretos:**
    - Navegar a los endpoints `/docs` (Swagger UI) y `/redoc` (ReDoc) de tu aplicación.
    - Utilizar la interfaz de Swagger para "probar" un endpoint `POST`, introduciendo un cuerpo JSON y ejecutando la petición.
    - Añadir metadatos como `title`, `description` y `tags` a tus `path operations` para enriquecer la documentación.
    - Explicar cómo un modelo de Pydantic se traduce en el "Schema" de la documentación.
- **Temas a Dominar (El "Qué"):**
    - OpenAPI (anteriormente Swagger Specification): El estándar detrás de la documentación.
    - Swagger UI: La interfaz interactiva (`/docs`).
    - ReDoc: La interfaz de documentación alternativa, más limpia (`/redoc`).
    - Añadir metadatos (`summary`, `description`, `response_description`, `tags`) a los decoradores de `path operation`.
    - Documentar respuestas de error con el parámetro `responses`.
- **Mejores Prácticas y Errores Comunes:**
    - **Mejor Práctica:** Sé prolijo con las descripciones y ejemplos en tus modelos de Pydantic (`Field(..., example="John Doe")`) y en tus endpoints. Un buen `description` puede ahorrarle a un colega (o a tu yo del futuro) mucho tiempo.
    - **Error Común:** No documentar los posibles errores que un endpoint puede devolver. Usa el parámetro `responses` para especificar otros códigos de estado además del 200 (ej. 404, 403) con sus respectivos modelos de respuesta.

- **Mini-Proyecto o Tarea Práctica:**
  - En tu API de tareas, añade `tags=["tasks"]` a todos tus endpoints. Añade descripciones claras a cada `path operation` y a los campos de tus modelos Pydantic. Explora la documentación generada en `/docs` y `/redoc`.

Fase 4: Bases de Datos con SQLModel y PostgreSQL 🗄️

Aquí es donde tu API cobra vida, persistiendo datos más allá del reinicio del servidor.

SQLModel: El ORM para FastAPI

- **Concepto Clave y Relevancia Profesional:** SQLModel es un ORM (Object-Relational Mapper) que combina Pydantic y SQLAlchemy, permitiéndote definir tus modelos de base de datos, API y validación en una única clase de Python. Esto reduce drásticamente la duplicación de código ("Don't Repeat Yourself"), minimiza errores de sincronización entre el modelo de la BD y el de la API, y mantiene la robustez de ambos mundos.
- **Objetivos de Aprendizaje Concretos:**
  - Definir un modelo de tabla de base de datos usando una clase que herede de `sqlmodel.SQLModel`.
  - Crear el "motor" (`engine`) de SQLAlchemy para conectar la aplicación a una base de datos PostgreSQL.
  - Escribir un servicio o repositorio que implemente las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) de forma asíncrona usando sesiones.
  - Explicar la diferencia entre el modelo `Task` (tabla de BD) y los modelos `TaskCreate`, `TaskRead` (API).
- **Temas a Dominar (El "Qué"):**
  - Definición de modelos y campos (`Field(default=None, primary_key=True)`).
  - Creación del `engine` (síncrono y asíncrono con `AsyncEngine`).
  - El patrón de Sesión (`Session`) como unidad de trabajo transaccional.
  - Consultas con `select()` y ejecución con `session.exec()`.
  - Operaciones CRUD: `session.add()`, `session.get()`, `session.delete()`, `session.commit()`, `session.refresh()`.
  - Manejo de relaciones (uno a muchos, muchos a muchos) con `Relationship` y `Mapped`.
- **Mejores Prácticas y Errores Comunes:**
  - **Mejor Práctica:** Abstrae la lógica de la base de datos en una capa de "repositorio" o "servicio". Tus `path operations` no deben contener código de SQLAlchemy; deben llamar a funciones como `crud.get_user()` o `user_service.create()`. Esto desacopla tu lógica de API de la de datos.
  - **Error Común:** Olvidar hacer `session.commit()` al final de una transacción que modifica datos (crear, actualizar, eliminar). Los cambios no se guardarán en la base de datos. Igualmente peligroso es olvidar el `session.rollback()` en caso de error.
- **Mini-Proyecto o Tarea Práctica:**
  - Refactoriza tu API de tareas para usar SQLModel y PostgreSQL. Crea una tabla `Task`. Implementa los endpoints CRUD para que interactúen con la base de datos real. Usa `docker-compose` para levantar la base de datos de PostgreSQL fácilmente.

PostgreSQL (Relacional)

- **Concepto Clave y Relevancia Profesional:** PostgreSQL es una de las bases de datos relacionales de código abierto más potentes, confiables y utilizadas en el mundo. Su soporte para transacciones ACID, tipos de datos avanzados y escalabilidad la hacen ideal para aplicaciones críticas donde la integridad de los datos es primordial.
- **Objetivos de Aprendizaje Concretos:**
  - Escribir consultas SQL básicas (`SELECT` con `WHERE` y `JOIN`, `INSERT`, `UPDATE`) directamente en un cliente de PostgreSQL (como DBeaver o psql).
  - Explicar qué significa cada letra en ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad) y por qué son cruciales para una transacción de transferencia bancaria.
  - Inicializar y ejecutar una migración de base de datos usando Alembic para crear tu primera tabla.
- **Temas a Dominar (El "Qué"):**
  - Lenguaje SQL fundamental: `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `ORDER BY`, `JOIN` (principalmente `INNER` y `LEFT`).
  - Comandos DML: `INSERT`, `UPDATE`, `DELETE`.
  - Comandos DDL: `CREATE TABLE`, `ALTER TABLE`.
  - Claves primarias y foráneas.
  - Índices y su impacto en el rendimiento de las consultas.
  - Transacciones ACID.
  - Migraciones de esquema con Alembic: `alembic revision --autogenerate`, `alembic upgrade head`.
- **Mejores Prácticas y Errores Comunes:**
  - **Mejor Práctica:** Nunca construyas consultas SQL concatenando strings (`f"SELECT * FROM users WHERE name = '{user_input}'"`). Esto es una invitación a la inyección de SQL. Siempre usa los mecanismos de parametrización de tu librería (SQLAlchemy/SQLModel lo hacen por ti) para pasar datos a las consultas.
  - **Error Común:** No poner índices en las columnas que se usan frecuentemente en cláusulas `WHERE` o para `JOINs` (como las claves foráneas). A medida que la tabla crece, la falta de índices hará que las consultas se vuelvan extremadamente lentas.
- **Mini-Proyecto o Tarea Práctica:**
  - Conecta un cliente de base de datos a tu PostgreSQL. Examina la tabla `Task` creada por Alembic. Intenta insertar una nueva tarea y luego leerla usando SQL puro. Esto te ayudará a entender lo que el ORM hace por debajo.

Redis (NoSQL - Caché)

- **Concepto Clave y Relevancia Profesional:** Redis es una base de datos en memoria extremadamente rápida, usada comúnmente como caché para reducir la carga sobre la base de datos principal (PostgreSQL) y acelerar drásticamente los tiempos de respuesta de la API. Para operaciones de lectura muy frecuentes, consultar Redis (microsegundos) es órdenes de magnitud más rápido que consultar el disco (milisegundos).
- **Objetivos de Aprendizaje Concretos:**
  - Implementar una estrategia de caché "cache-aside" para un endpoint `GET` que lee datos de la base de datos.
  - Invalidar (eliminar) la clave de caché correspondiente cuando un endpoint `PUT` o `DELETE` modifica los datos originales.
  - Configurar un tiempo de expiración (TTL - Time To Live) en las claves de Redis para que el caché se refresque automáticamente.

- **Temas a Dominar (El "Qué"):**
  - Estructuras de datos de Redis (principalmente `Strings` y `Hashes` para caché).
  - Comandos básicos: `SET`, `GET`, `DEL`, `EXPIRE`.
  - Patrones de caché: Cache-Aside, Write-Through.
  - Uso de una librería de cliente de Redis para Python (ej. `redis-py`).
  - Serialización de datos antes de guardarlos en Redis (ej. a JSON).
- **Mejores Prácticas y Errores Comunes:**
  - **Mejor Práctica:** No intentes cachear todo. Identifica los "puntos calientes" de tu aplicación: las consultas más lentas y más frecuentes. Esos son los candidatos perfectos para el caché.
  - **Error Común:** Olvidar invalidar el caché. Si actualizas un dato en PostgreSQL pero no eliminas su versión antigua en Redis, tu API servirá datos obsoletos. La invalidación del caché es una de las partes más difíciles de hacer bien.
- **Mini-Proyecto o Tarea Práctica:**
  - Añade Redis a tu `docker-compose.yml`. En tu API de tareas, modifica el endpoint `GET /tasks/{task_id}`. Antes de consultar PostgreSQL, comprueba si la tarea está en Redis. Si lo está, devuélvela desde ahí. Si no, consúltala de la base de datos, guárdala en Redis con un TTL de 1 minuto, y luego devuélvela.

Fase 5: Seguridad en APIs 🔐

Una API sin seguridad es una puerta abierta. Esta fase es crítica para ser un desarrollador responsable.

Autenticación y Autorización con JWT

- **Concepto Clave y Relevancia Profesional:** JWT (JSON Web Tokens) es un estándar para crear tokens de acceso que permiten la autenticación y autorización de forma segura y sin estado. El flujo OAuth2 es el estándar de la industria para que un usuario obtenga estos tokens (login), y FastAPI proporciona herramientas para proteger endpoints y asegurar que solo los usuarios autenticados y autorizados puedan acceder a ellos.
- **Objetivos de Aprendizaje Concretos:**
  - Implementar un endpoint `/token` que, dado un usuario y contraseña, devuelva un `access_token` JWT.
  - Crear una dependencia de seguridad que extraiga el token de la cabecera, lo valide (firma y expiración) y devuelva el payload del usuario.
  - Proteger un endpoint usando esta dependencia para que solo sea accesible con un token válido.
  - Explicar la diferencia entre Autenticación ("quién eres") y Autorización ("qué puedes hacer").
- **Temas a Dominar (El "Qué"):**
  - Estructura de un JWT: Header, Payload (con claims como `sub`, `exp`), Signature.
  - OAuth2 y el flujo "Password Flow" (`OAuth2PasswordBearer`).
  - Hashing de contraseñas (nunca guardarlas en texto plano) usando `passlib`.
  - Creación (encode) y validación (decode) de tokens usando una librería como `python-jose`.
  - El concepto de "Bearer Token".
  - Scopes de OAuth2 para autorización granular.
- **Mejores Prácticas y Errores Comunes:**
  - **Mejor Práctica:** Almacena el JWT en una cookie `HttpOnly` en el navegador del cliente. Esto evita que sea accesible por scripts de JavaScript maliciosos (ataques XSS), que es un riesgo si se almacena en `localStorage`.

    - **Error Común:** Poner información sensible en el payload del JWT. El payload es visible para cualquiera que tenga el token (solo la firma lo protege de manipulaciones). Nunca pongas contraseñas, claves de API u otros secretos ahí. El `user_id` o el `username` son suficientes.
- **Mini-Proyecto o Tarea Práctica:**
    - Añade un modelo de `User` a tu API (con `username` y `hashed_password`). Implementa el endpoint `/token`. Crea un endpoint `/users/me` que esté protegido y devuelva los datos del usuario que está autenticado, extrayéndolos del token.

OWASP Top 10 para APIs

- **Concepto Clave y Relevancia Profesional:** El OWASP Top 10 es una lista de referencia de los riesgos de seguridad más críticos para las aplicaciones web (y sus APIs). Conocerlos y saber cómo mitigarlos en FastAPI es una señal de madurez profesional y es fundamental para construir aplicaciones que protejan los datos de la empresa y de sus usuarios.
- **Objetivos de Aprendizaje Concretos:**
    - Explicar cómo el uso de un ORM como SQLModel/SQLAlchemy previene la Inyección de SQL (A03:2021).
    - Describir cómo la validación estricta de Pydantic mitiga los problemas de Asignación Masiva (A01:2021 - Broken Object Level Authorization).
    - Implementar limitación de tasa (rate limiting) en los endpoints más sensibles (como el de login) para prevenir ataques de fuerza bruta (A07:2021 - Identification and Authentication Failures).
- **Temas a Dominar (El "Qué"):**
    - Inyección (SQL, NoSQL, OS).
    - Autenticación Rota.
    - Exposición de Datos Sensibles.
    - Autorización a Nivel de Objeto Rota (ej. poder ver `/users/123` cuando eres el usuario 456).
    - Falsificación de Peticiones del Lado del Servidor (SSRF).
    - Uso de CORS (Cross-Origin Resource Sharing) en FastAPI para controlar qué dominios pueden acceder a tu API.
- **Mejores Prácticas y Errores Comunes:**
    - **Mejor Práctica:** Adopta un enfoque de "defensa en profundidad". No confíes en una sola medida de seguridad. Combina validación de entrada (Pydantic), consultas parametrizadas (ORM), autorización a nivel de endpoint y de objeto, y logs de seguridad.
    - **Error Común:** Configurar CORS con un comodín (`allow_origins=["*"]`) en producción. Esto permite que cualquier sitio web en internet haga peticiones a tu API, lo cual puede ser un riesgo de seguridad. Sé explícito y lista solo los dominios de tu frontend.
- **Mini-Proyecto o Tarea Práctica:**
    - En tu API de tareas, asegúrate de que un usuario solo pueda ver, actualizar o eliminar sus propias tareas. Implementa una lógica en tus endpoints que verifique que el `user_id` de la tarea a modificar coincida con el `user_id` del token del usuario autenticado.

Gestión de Secretos

- **Concepto Clave y Relevancia Profesional:** Los secretos son credenciales sensibles como claves de API, contraseñas de bases de datos o la clave secreta para firmar JWTs. Nunca deben estar escritos directamente en el código ("hardcodeados"). Usar variables de entorno y cargarlas a través de Pydantic BaseSettings es la forma moderna y segura de gestionar estas configuraciones.

- **Objetivos de Aprendizaje Concretos:**
  - Crear una clase de `Settings` que herede de `pydantic.BaseSettings` para cargar la URL de la base de datos y la clave secreta del JWT.
  - Crear un archivo `.env` para almacenar los secretos durante el desarrollo local y asegurarse de que esté incluido en el `.gitignore`.
  - Instanciar los `Settings` como una dependencia global y usarla en la aplicación.
- **Temas a Dominar (El "Qué"):**
  - Por qué nunca hacer "commit" de secretos a Git.
  - Variables de entorno.
  - El uso del archivo `.env`.
  - `pydantic_settings.BaseSettings` para la carga y validación automática de variables de entorno.
  - Diferencia entre configuración de desarrollo, testing y producción.
- **Mejores Prácticas y Errores Comunes:**
  - **Mejor Práctica:** Crea un archivo `.env.example` o `env.template` en tu repositorio. Este archivo debe tener la misma estructura que tu `.env` pero con valores vacíos o de ejemplo. Sirve como documentación para que otros desarrolladores sepan qué variables de entorno necesitan configurar.
  - **Error Común:** Cargar el archivo `.env` en producción. En entornos de producción (como Docker en un servidor), las variables de entorno se deben inyectar directamente en el contenedor, no a través de un archivo `.env` que podría ser expuesto accidentalmente. Pydantic manejará esto transparentemente.
- **Mini-Proyecto o Tarea Práctica:**
  - Refactoriza tu API para que todas las configuraciones (URL de la BD, `JWT_SECRET_KEY`, `ALGORITHM`) se carguen desde variables de entorno a través de una clase `Settings` de Pydantic.

Fase 6: Automatización y Despliegue con GitHub ☁

Escribir código es solo la mitad del trabajo. Entregarlo de forma fiable y automatizada es lo que te hace un ingeniero de ciclo completo.

Git y GitHub

- **Concepto Clave y Relevancia Profesional:** Git es el sistema de control de versiones estándar de la industria, y GitHub es la plataforma para alojar repositorios Git y colaborar. GitFlow (o una versión simplificada) es un flujo de trabajo que organiza cómo se usan las ramas para desarrollar nuevas funcionalidades, corregir errores y lanzar versiones de forma ordenada.
- **Objetivos de Aprendizaje Concretos:**
  - Crear una nueva "feature branch" a partir de `develop` para trabajar en una nueva funcionalidad.
  - Hacer commits atómicos y descriptivos.
  - Abrir una "Pull Request" (PR) en GitHub para fusionar tu rama de funcionalidad en `develop`.
  - Resolver un conflicto de fusión simple.
- **Temas a Dominar (El "Qué"):**
  - Ramas principales: `main` (o `master`) para producción, `develop` para integración.
  - Ramas de soporte: `feature/*`, `bugfix/*`, `hotfix/*`.
  - El ciclo: `git checkout -b feature/mi-feature`, trabajar, `git add`, `git commit`, `git push`, crear Pull Request.

    - Revisión de código en Pull Requests.
- **Mejores Prácticas y Errores Comunes:**
    - **Mejor Práctica:** Escribe buenos mensajes de commit. Sigue la convención "Conventional Commits" (ej. `feat: add user login endpoint`). Facilita la generación automática de changelogs y la comprensión del historial del proyecto.
    - **Error Común:** Hacer "push" directamente a `main` o `develop`. Estas ramas deben estar protegidas, y todos los cambios deben pasar por una Pull Request y una revisión de código.
- **Mini-Proyecto o Tarea Práctica:**
    - Sube tu proyecto de API a un nuevo repositorio en GitHub. Practica el flujo creando una rama para una nueva funcionalidad (ej. `feature/rate-limiting`), haciendo algunos commits y abriendo una PR para fusionarla a tu rama principal.

Docker

- **Concepto Clave y Relevancia Profesional:** Docker "empaqueta" tu aplicación con todas sus dependencias (incluyendo el sistema operativo base) en una unidad ligera y portable llamada "contenedor". Esto resuelve el problema de "en mi máquina funciona" a un nivel superior, garantizando que la aplicación se ejecute exactamente igual en desarrollo, testing y producción.
- **Objetivos de Aprendizaje Concretos:**
    - Escribir un `Dockerfile` que construya una imagen para tu aplicación FastAPI.
    - Crear un archivo `docker-compose.yml` que defina y orqueste los servicios de tu aplicación: la API de FastAPI, la base de datos PostgreSQL y el caché de Redis.
    - Levantar todo el entorno de desarrollo local con un solo comando: `docker-compose up`.
- **Temas a Dominar (El "Qué"):**
    - Imágenes vs. Contenedores.
    - El `Dockerfile`: `FROM`, `WORKDIR`, `COPY`, `RUN`, `CMD`.
    - Construcción de imágenes en múltiples etapas (multi-stage builds) para reducir el tamaño final.
    - `docker-compose.yml`: `services`, `image`, `build`, `ports`, `volumes`, `environment`.
    - Redes y volúmenes de Docker.
- **Mejores Prácticas y Errores Comunes:**
    - **Mejor Práctica:** Usa imágenes base oficiales y específicas (ej. `python:3.11-slim`) en lugar de `python:latest`. Usa `multi-stage builds` para mantener la imagen de producción final lo más pequeña y segura posible.
    - **Error Común:** Incluir secretos, archivos `.env` o el directorio `.git` en el contexto de construcción de la imagen de Docker. Usa un archivo `.dockerignore` (similar a `.gitignore`) para excluir estos archivos y optimizar el proceso de build.
- **Mini-Proyecto o Tarea Práctica:**
    - "Dockeriza" tu aplicación de API de tareas. Crea el `Dockerfile` y el `docker-compose.yml` para levantar la API, PostgreSQL y Redis. Asegúrate de que puedan comunicarse entre sí a través de la red de Docker.

CI/CD con GitHub Actions

- **Concepto Clave y Relevancia Profesional:** CI/CD (Integración Continua / Despliegue Continuo) es la práctica de automatizar las fases de construcción, prueba y despliegue de tu código. GitHub Actions permite definir flujos de trabajo que se ejecutan automáticamente en respuesta a eventos (como un

push o una PR), garantizando que cada cambio sea probado y sea desplegable, aumentando la calidad y velocidad de entrega.

- **Objetivos de Aprendizaje Concretos:**
  - Crear un workflow de GitHub Actions que se dispare en cada push a una PR.
  - Configurar un "job" que instale las dependencias del proyecto (con Poetry).
  - Añadir pasos para ejecutar linting (con `ruff` o `flake8`) y pruebas automatizadas (con `pytest`).
  - (https://www.google.com/search?q=Avanzado) Añadir un paso que construya la imagen de Docker para validar que el `Dockerfile` es correcto.
- **Temas a Dominar (El "Qué"):**
  - Sintaxis de los archivos YAML de workflows.
  - Eventos que disparan workflows (`on: [push, pull_request]`).
  - Jobs, steps y runners.
  - Uso de "actions" de la comunidad (ej. `actions/checkout@v3`, `actions/setup-python@v4`).
  - Matrices de prueba (ej. probar en diferentes versiones de Python).
  - Uso de cachés para acelerar los workflows (ej. caché de dependencias de Poetry).
- **Mejores Prácticas y Errores Comunes:**
  - **Mejor Práctica:** Haz que tus pipelines fallen rápido. Ejecuta los pasos más rápidos (como el linting) primero. Si fallan, el workflow se detiene inmediatamente, ahorrando tiempo y recursos del runner.
  - **Error Común:** No ejecutar las pruebas contra servicios reales (o dobles de prueba). Un pipeline que solo prueba el código en aislamiento puede pasar, pero la aplicación puede fallar al intentar conectar con la base de datos. Usa `services` en GitHub Actions para levantar una BD o Redis durante la ejecución de las pruebas.
- **Mini-Proyecto o Tarea Práctica:**
  - En tu repositorio de GitHub, crea un directorio `.github/workflows` y añade un archivo `ci.yml`. Configúralo para que en cada PR ejecute `poetry install`, un linter y tus pruebas de `pytest`. Observa cómo se ejecuta automáticamente cuando abres una nueva PR.

GitHub Hooks (Webhooks)

- **Concepto Clave y Relevancia Profesional:** Un webhook es una notificación automática vía HTTP que GitHub envía a un servicio externo cuando ocurre un evento (ej. un push, un comentario en una PR). Mientras que GitHub Actions es la herramienta *nativa* para reaccionar a estos eventos, los webhooks permiten integrar GitHub con un ecosistema infinito de herramientas externas (servidores de despliegue, sistemas de notificación como Slack, etc.).
- **Objetivos de Aprendizaje Concretos:**
  - Explicar la diferencia fundamental entre un webhook y una GitHub Action.
  - Configurar un webhook en un repositorio de GitHub que apunte a una URL pública (puedes usar un servicio como `webhook.site` para probar).
  - Observar el payload JSON que GitHub envía en el cuerpo de la petición del webhook.
- **Temas a Dominar (El "Qué"):**
  - Concepto de evento y payload.
  - Configuración en la UI de GitHub: URL del payload, tipo de contenido, secreto.
  - El rol del "secreto" para verificar que la petición viene realmente de GitHub.
  - Casos de uso comunes: notificaciones en Slack, triggers para despliegues personalizados, etc.
- **Mejores Prácticas y Errores Comunes:**

    - **Mejor Práctica:** Siempre configura un "secreto" para tu webhook y valida la firma de la petición en tu servicio receptor. Esto previene que un atacante pueda enviar peticiones falsas a tu endpoint y disparar acciones no deseadas.
    - **Error Común:** Crear un servicio receptor de webhooks que realice una tarea larga y bloqueante de forma síncrona. El servicio debe responder a GitHub lo más rápido posible (con un `200 OK`) para confirmar la recepción y luego procesar la tarea en segundo plano (usando una cola de tareas como Celery).
- **Mini-Proyecto o Tarea Práctica:**
    - Crea un endpoint simple `/webhooks/github` en tu API de FastAPI. Configura un webhook en tu repositorio para que apunte a esa URL (necesitarás exponer tu API local a internet temporalmente con una herramienta como `ngrok`). Haz que el endpoint simplemente imprima en la consola el cuerpo de la petición que recibe de GitHub. Haz un push y observa el resultado.

Fase 7: Ampliando el Stack: Django 🏛️

Entender un framework alternativo te da perspectiva y te hace un ingeniero más completo.

Concepto General y Componentes Clave

- **Concepto Clave y Relevancia Profesional:** Django es un framework web "con baterías incluidas", lo que significa que viene con soluciones integradas para la mayoría de las necesidades de una aplicación web grande y monolítica: ORM, panel de administración, autenticación, etc. Aunque FastAPI es ideal para microservicios y APIs, conocer Django es valioso para trabajar en proyectos existentes o cuando se necesita un desarrollo rápido de una aplicación completa con un panel de administración robusto.
- **Objetivos de Aprendizaje Concretos:**
    - Describir la arquitectura Modelo-Vista-Template (MVT) de Django y compararla con la aproximación de FastAPI.
    - Generar una aplicación simple y explorar el panel de administración autogenerado de Django.
    - Explicar las principales diferencias filosóficas entre el ORM de Django y SQLAlchemy/SQLModel.
    - Entender el rol de Django REST Framework (DRF) como la herramienta para construir APIs sobre Django.
- **Temas a Dominar (El "Qué"):**
    - Filosofía "Baterías Incluidas" vs. "Micro" framework.
    - Arquitectura MVT (Modelo, Vista, Template).
    - El ORM de Django: `models.Model`, `Manager`, `QuerySet`.
    - Migraciones (`manage.py makemigrations`, `manage.py migrate`).
    - El Panel de Administración (`admin.site.register`).
    - Django REST Framework (DRF): `Serializers`, `ViewSets`, `Routers`.
- **Mejores Prácticas y Errores Comunes:**
    - **Mejor Práctica:** Si decides usar Django, aprovecha sus "baterías". Usa su sistema de autenticación, su ORM y su panel de administración. Intentar reemplazar estos componentes centrales suele ser más complicado que usar un framework más ligero como FastAPI desde el principio.
    - **Error Común:** Intentar hacer que Django sea asíncrono en todas partes. Aunque Django está añadiendo soporte asíncrono, su ecosistema y su diseño principal siguen siendo predominantemente síncronos. Forzar el asincronismo en partes que no están diseñadas para ello puede ser complejo y contraintuitivo.

- **Mini-Proyecto o Tarea Práctica:**
  - Sigue el tutorial oficial de Django para crear la aplicación de encuestas. Presta especial atención a cómo se definen los modelos, cómo se registran en el panel de administración y cómo se generan las vistas. No necesitas ser un experto, solo entender la filosofía.

Fase 8: Introducción al Frontend: React ⚛

Un desarrollador backend que entiende el frontend es 10 veces más efectivo.

Fundamentos de React y Hooks Esenciales

- **Concepto Clave y Relevancia Profesional:** React es una librería de JavaScript para construir interfaces de usuario interactivas a partir de piezas reutilizables llamadas "componentes". Los "Hooks" (`useState`, `useEffect`) son funciones que te permiten "enganchar" el estado y el ciclo de vida de un componente, permitiendo una lógica compleja dentro de componentes funcionales y simples.
- **Objetivos de Aprendizaje Concretos:**
  - Crear un componente funcional de React que reciba datos a través de `props` y los renderice.
  - Utilizar el hook `useState` para manejar un estado simple, como el valor de un campo de entrada de texto.
  - Utilizar el hook `useEffect` para ejecutar una llamada a tu API de FastAPI cuando el componente se monta por primera vez.
- **Temas a Dominar (El "Qué"):**
  - JSX: La sintaxis que mezcla HTML y JavaScript.
  - Componentes Funcionales.
  - Props (para pasar datos de padre a hijo) vs. State (para manejar datos internos del componente).
  - Hook `useState`: Para declarar y actualizar variables de estado.
  - Hook `useEffect`: Para manejar efectos secundarios (llamadas a API, suscripciones, etc.).
  - Renderizado condicional y de listas (`.map()`).
- **Mejores Prácticas y Errores Comunes:**
  - **Mejor Práctica:** Descompón tu UI en componentes pequeños y reutilizables. Un componente `Button` es mejor que repetir el código del botón en diez sitios diferentes.
  - **Error Común:** Modificar el estado directamente (`miEstado = 'nuevo valor'`). Siempre debes usar la función seteadora que te devuelve `useState` (`setMiEstado('nuevo valor')`) para que React sepa que debe volver a renderizar el componente.
- **Mini-Proyecto o Tarea Práctica:**
  - Crea una aplicación de React simple con `create-react-app`. Crea un componente `TaskList` que, por ahora, muestre una lista de tareas "hardcodeada".

Comunicación con el Backend y Gestión de Estado

- **Concepto Clave y Relevancia Profesional:** Una aplicación de frontend moderna (Single Page Application) es inútil sin datos. Saber cómo conectar tu aplicación React a tu API de FastAPI usando herramientas como `fetch` o `axios` es el puente que une los dos mundos. La gestión de estado con herramientas como `useContext` permite compartir datos (como la información del usuario autenticado) a través de toda la aplicación sin tener que pasarlos por `props` manualmente.
- **Objetivos de Aprendizaje Concretos:**
  - Usar `fetch` o `axios` dentro de un `useEffect` para obtener la lista de tareas de tu API de FastAPI y guardarla en el estado del componente.

    - Renderizar la lista de tareas obtenida de la API en lugar de la lista "hardcodeada".
    - Crear un formulario en React que, al enviarse, realice una petición `POST` a tu API para crear una nueva tarea.
    - Envolver tu aplicación con un `Context.Provider` para hacer que los datos del usuario autenticado estén disponibles globalmente.
- **Temas a Dominar (El "Qué"):**
    - Llamadas a API con `fetch` o `axios`.
    - Manejo de promesas con `.then().catch()` o `async/await`.
    - Gestión de estados de carga (`loading`) y error (`error`).
    - Envío de cabeceras (`Authorization` con el token JWT).
    - El Hook `useContext` para compartir estado sin "prop drilling".
- **Mejores Prácticas y Errores Comunes:**
    - **Mejor Práctica:** Centraliza tu lógica de llamadas a la API. Crea funciones como `api.getTasks()` o `api.createTask(taskData)` en un archivo separado. Tus componentes no deberían construir las URLs o las cabeceras, solo llamar a estas funciones de servicio.
    - **Error Común:** Olvidar el array de dependencias en `useEffect`. Si haces una llamada a la API en un `useEffect` sin un array de dependencias vacío (`[]`) al final, la llamada se ejecutará en un bucle infinito después de cada renderizado.
- **Mini-Proyecto o Tarea Práctica:**
    - Conecta tu frontend de React con tu backend de FastAPI. Implementa la funcionalidad completa:
        1. Mostrar la lista de tareas.
        2. Añadir una nueva tarea a través de un formulario.
        3. Poder eliminar una tarea.
        4. (Bonus) Implementa un formulario de login que guarde el JWT y lo use en las peticiones subsiguientes.