Módulo 0: Preparando el Terreno (Setup y Contexto)

0.1. ¿Por qué necesitamos Servicios Web?

El Problema de la "Productivización"

En el ámbito del Data Science (y en el desarrollo de software en general), nos enfrentamos a un desafío crítico: la **"productivización"**.
Tú, como investigador, puedes desarrollar un modelo de predicción de accidentes de tráfico (basado en tus redes neuronales causales) con una precisión asombrosa. Este modelo puede existir como un *script* de Python, un *notebook* de Jupyter o un conjunto de archivos .py que importas como una librería.
El problema es: **¿Cómo consume este modelo el resto del mundo?**

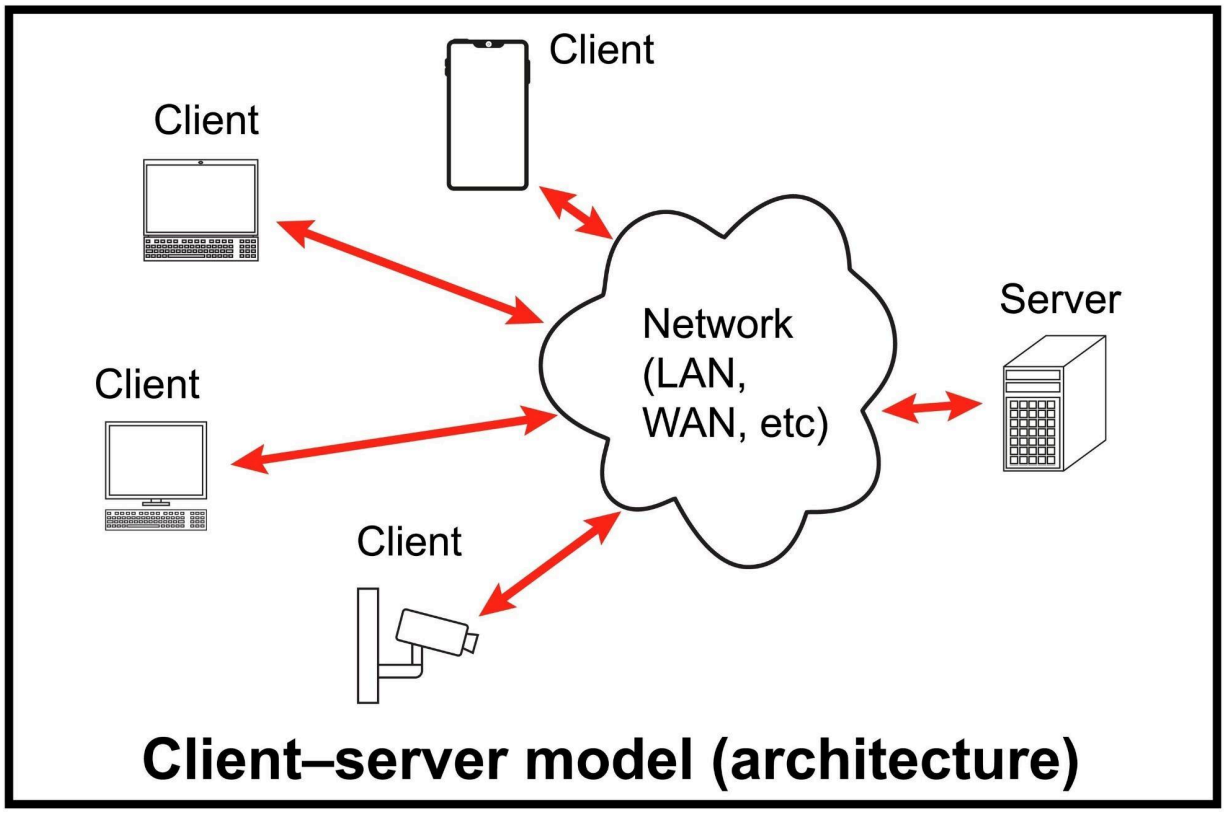
- ¿Cómo lo integra una aplicación móvil que quiere alertar a un conductor?
- ¿Cómo lo utiliza un dashboard de Business Intelligence para mostrar zonas de riesgo?
- ¿Cómo lo consume otro equipo de analistas para validar sus propios datos?

La respuesta es: **No pueden**. Al menos, no de forma eficiente, escalable o segura. El modelo vive aislado en tu máquina.
La "productivización" es el proceso de tomar ese *artefacto* de software (un modelo, un algoritmo, un conjunto de datos) y exponerlo de una manera robusta, fiable y accesible para que otras aplicaciones (clientes) puedan interactuar con él.

Diferencia entre Librería, Script y Servicio

Es vital diferenciar estos conceptos:

1. **Script (.py):**
    - **Qué es:** Un archivo ejecutado de principio a fin para realizar una tarea concreta.
    - **Ejemplo:** python train_model.py.
    - **Interacción:** Se ejecuta manualmente (o por un cron). No está "vivo" esperando peticiones.
2. **Librería (Módulo de Python):**
    - **Qué es:** Código diseñado para ser importado (import my_utils) y utilizado por *otro* código.
    - **Ejemplo:** pandas, scikit-learn, o tu propio módulo causal_nn.
    - **Interacción:** El acoplamiento es total. Quien lo usa debe programar en Python, tener la librería instalada y gestionar las mismas dependencias que tú.
3. **Servicio Web (API):**
    - **Qué es:** Un programa que se ejecuta de forma persistente en un servidor, escuchando peticiones en un puerto de red (ej. puerto 8000).



Shutterstock

- **Ejemplo:** Una API de FastAPI ejecutándose.
- **Interacción:** El desacoplamiento es máximo. Un cliente (una app móvil, una web, otro *backend*) le envía una petición a través de un protocolo estándar (HTTP) y recibe una respuesta (generalmente en formato JSON).
- **Ventaja clave:** El cliente no necesita saber nada sobre cómo está implementado el servicio. Tu modelo de predicción puede estar escrito en Python/PyTorch, pero el cliente puede ser una app programada en Swift (iOS) o Kotlin (Android).

El Rol del Data Scientist en la Exposición de Datos

Tradicionalmente, el Data Scientist entregaba un modelo (un archivo .pkl, por ejemplo) y un equipo de "Ingeniería de Software" o "MLOps" se encargaba de crear el servicio web.
Este paradigma está cambiando. Herramientas como FastAPI han reducido tanto la fricción para crear servicios web de alto rendimiento que **ahora se espera que el Data Scientist sea capaz de exponer su propio trabajo**.
Tu rol no termina al entrenar el modelo, sino al hacerlo *útil*. Un servicio web es el "contrato" o la "interfaz" que permite que tu investigación tenga un impacto real en un producto.

0.2. ¿Por qué FastAPI?

El Ecosistema Python: De Flask/Django a FastAPI

En Python, los dos *frameworks* web tradicionales han sido:

- **Django:** Un *framework* "full-stack" y "opinado". Incluye su propio ORM, panel de administración, sistema de autenticación, etc. Es excelente para construir aplicaciones monolíticas complejas (como un CMS o un e-commerce).
- **Flask:** Un "micro-framework". Proporciona lo mínimo indispensable (rutas y plantillas) y deja que tú elijas el resto (ORM, validación, etc.). Es más ligero y flexible, ideal para servicios pequeños.

Ambos son excelentes, pero nacieron en la era **WSGI (Web Server Gateway Interface)**. WSGI es un estándar síncrono: el servidor maneja una petición, la procesa y devuelve una respuesta. Si el procesamiento tarda (ej. una consulta a BBDD o una llamada a otra API), el hilo del servidor se queda *bloqueado* esperando.

Rendimiento (ASGI vs WSGI) y Asincronismo

Aquí es donde entra **ASGI (Asynchronous Server Gateway Interface)**. ASGI es la evolución de WSGI, diseñado para el mundo asíncrono.

El **asincronismo** (usando async y await en Python) permite que, mientras un hilo está esperando una operación de entrada/salida (I/O) —como esperar datos de la red o leer un archivo—, pueda dedicarse a procesar *otra* petición. No se bloquea.

FastAPI está construido desde cero sobre ASGI. Esto le da un rendimiento comparable al de Node.js o Go (que son asíncronos por naturaleza), pero permitiéndote programar en Python.

Ejemplo conceptual (Sin código aún):

- **Síncrono (Flask/WSGI):**
  1. Llega Petición A.
  2. El servidor empieza a procesar A.
  3. El proceso A necesita 3 segundos para consultar una BBDD. El servidor se bloquea 3 segundos.
  4. Llega Petición B. Tiene que esperar.
  5. Termina Petición A.
  6. El servidor empieza a procesar B.
- **Asíncrono (FastAPI/ASGI):**
  1. Llega Petición A.
  2. El servidor empieza a procesar A.
  3. El proceso A necesita 3 segundos de I/O (await db.query(...)). El servidor "aparca" la Petición A y se libera.
  4. Llega Petición B. El servidor está libre y empieza a procesar B.
  5. ...
  6. Llegan los datos de la BBDD para A. El servidor retoma Petición A donde la dejó.

Este manejo concurrente de peticiones es lo que da a FastAPI su altísimo rendimiento.

Validación de Datos Nativa (Pydantic)

En cualquier API, recibes datos (normalmente JSON). ¿Cómo te aseguras de que esos datos son correctos? ¿Que el user_id es un entero y no un *string*? ¿Que el email tiene un formato válido?

En Flask, tendrías que validar esto manualmente (con ifs) o usar una librería externa.
FastAPI usa **Pydantic** de forma nativa. Pydantic es una librería que te permite definir "schemas" (esquemas) de datos usando *type hints* de Python.
Ejemplo conceptual (El código real lo veremos en el Módulo 3):

```
# Esto no es código de FastAPI aún, es Pydantic puro
from pydantic import BaseModel, EmailStr

class User(BaseModel):
    username: str
    user_id: int
    email: EmailStr  # Validación de email gratis

# Si alguien nos envía JSON malformado...
json_malo = {"username": "Pepe", "user_id": "123", "email": "pepe@dominio"}
# "123" es un string, no un int. "pepe@dominio" no es un email válido.

try:
    User.model_validate(json_malo)
except Exception as e:
    print(e) # Pydantic lanzará un error de validación clarísimo.
```

FastAPI integra esto: si defines que tu *endpoint* espera un User, FastAPI automáticamente valida el JSON entrante y, si falla, devuelve un error 422 (Unprocessable Entity) al cliente con los detalles del fallo.

Documentación Automática (El tema de Swagger)

Este es, quizás, el punto más valorado de FastAPI.
Como tu API es un "contrato", necesitas *documentar* ese contrato. ¿Qué *endpoints* existen? ¿Qué métodos HTTP usan (GET, POST)? ¿Qué datos esperan? ¿Qué datos devuelven? Tradicionalmente, esto se hacía a mano (o con herramientas tediosas).
FastAPI se adhiere a un estándar llamado **OpenAPI** (antes conocido como Swagger). Como usas Pydantic para definir tus modelos de datos, FastAPI tiene toda la información que necesita para **generar automáticamente una documentación interactiva**.
Esta documentación es **Swagger UI**. Es una página web (normalmente en /docs) donde puedes ver todos tus *endpoints* e incluso **probarlos en vivo** desde el navegador. Es gratis, automático y siempre está sincronizado con tu código.

0.3. Instalación del Entorno

Vamos a preparar nuestro entorno de trabajo. Es **altamente recomendable** usar un entorno virtual (venv) para aislar las dependencias de este proyecto.

```
# 1. Crear un directorio para el proyecto
mkdir curso_fastapi
```

```
cd curso_fastapi

# 2. Crear un entorno virtual
python -m venv venv

# 3. Activar el entorno virtual
# En macOS / Linux:
source venv/bin/activate
# En Windows (cmd):
.\venv\Scripts\activate
# En Windows (PowerShell):
.\venv\Scripts\Activate.ps1

# (Verás (venv) al principio de tu línea de comandos)
```

Ahora, instalamos las dos librerías necesarias:

```
# 1. Instalar FastAPI
pip install fastapi

# 2. Instalar Uvicorn, el servidor ASGI
# "standard" incluye optimizaciones recomendadas
pip install "uvicorn[standard]"
```

¡Eso es todo! Con estas dos librerías, ya tienes todo lo necesario para construir y ejecutar tu API.

Estructura de Proyecto Recomendada

Para empezar, solo necesitaremos un archivo. Pero a medida que el proyecto crezca (Módulo 4), querremos separar las cosas. Una estructura inicial simple sería:

```
curso_fastapi/
├── venv/           # El entorno virtual
├── main.py         # El archivo principal de nuestra app FastAPI
└── db.json         # Nuestra futura "base de datos"
```