Módulo 1: Fundamentos Teóricos de APIs

En el Módulo 0 establecimos la *necesidad* de exponer nuestro trabajo. En este módulo, definiremos formalmente *qué* vamos a construir. No se trata solo de "hacer que funcione", sino de adherirnos a un paradigma arquitectónico que ha demostrado ser la base de la web escalable moderna.

1.1. ¿Qué es un Servicio Web?

Definición Formal: Un **Servicio Web** es un sistema de software diseñado para soportar la interoperabilidad **máquina-a-máquina (M2M)** a través de una red.
Dicho de otra forma, es un programa que expone una interfaz (la API, *Application Programming Interface*) que no está pensada para ser consumida por un humano, sino por *otro programa* (un "cliente").
Su objetivo es el **desacoplamiento**: el cliente y el servidor pueden estar escritos en lenguajes distintos, ejecutarse en sistemas operativos diferentes y estar en ubicaciones geográficas opuestas. Lo único que comparten es un *protocolo* y un *contrato* (la API) común.

Evolución: De RPC a REST

Comprender la evolución nos da contexto sobre por qué REST (el paradigma que usaremos) es tan dominante hoy:

1. **RPC (Remote Procedure Call):** El primer intento. La idea era hacer que una llamada a una función en una máquina remota (remote_calculate(a, b)) fuera tan simple como una llamada a una función local.
    - **Problema:** Alto acoplamiento. El cliente necesitaba saber el *nombre exacto* de la función, sus parámetros y tipo de retorno. Era frágil ante los cambios.
2. **SOAP (Simple Object Access Protocol):** El intento de estandarización. Se basaba en XML para todo. Definía un sobre (<Envelope>) para el mensaje y usaba un archivo WSDL (Web Services Description Language) para definir rigurosamente el "contrato".
    - **Problema:** Extremadamente verboso, complejo y "pesado". Exigía un procesamiento XML considerable en ambos extremos.
3. **APIs Web (Basadas en HTTP):** La comunidad se dio cuenta de que ya existía un protocolo perfectamente funcional, robusto y universal: **HTTP** (el protocolo de la propia web). En lugar de inventar nuevos protocolos, ¿por qué no usar HTTP directamente?
    - **Ventaja:** Ligero, universalmente soportado (cualquier lenguaje puede hacer una petición HTTP) y utiliza formatos de datos simples como **JSON (JavaScript Object Notation)**, que es nativo para la web y trivial de *parsear* en cualquier lenguaje (incluido Python).

FastAPI se sitúa en esta última categoría, implementando el estilo arquitectónico más exitoso para construir APIs Web: **REST**.

1.2. El Paradigma REST (Representational State

Transfer)

Este es el concepto más importante del módulo.

- **REST NO es un protocolo.** (HTTP es el protocolo).
- **REST NO es un estándar.** (SOAP es un estándar).
- **REST es un ESTILO ARQUITECTÓNICO.**

Fue definido por **Roy Fielding** en su disertación doctoral (2000), "Architectural Styles and the Design of Network-based Software Architectures". Fielding fue uno de los autores principales de la especificación HTTP.
Su tesis analizaba *por qué* la World Wide Web (el sistema de hipermedia distribuido más grande del mundo) funcionaba y escalaba tan bien. REST es, en esencia, la codificación de los principios (restricciones) que hacen que la Web funcione.
El nombre "Representational State Transfer" (Transferencia de Estado Representacional) es clave:

- **Recurso (Resource):** El concepto central. Es cualquier "cosa" a la que queramos dar un nombre: un usuario, un producto, una predicción de tu modelo, un resultado de un cálculo.
- **Estado (State):** El estado actual de ese recurso (ej. el usuario "Pepe" tiene 30 años y vive en Alicante).
- **Representación (Representation):** Una "foto" o versión de ese estado en un formato concreto (ej. un JSON: {"nombre": "Pepe", "edad": 30}).
- **Transferencia (Transfer):** Cuando un cliente pide un recurso, el servidor le *transfiere* una *representación* de su *estado*. Cuando el cliente quiere modificar el recurso, envía una *representación* modificada de vuelta al servidor.

1.3. Los 6 Principios (Restricciones) de REST

Para que una arquitectura sea considerada "RESTful", debe adherirse a las siguientes 6 restricciones.

1. Cliente-Servidor (Separation of Concerns)

La arquitectura debe estar compuesta por dos entidades desacopladas:

- **Servidor:** Gestiona los datos (lógica de negocio, base de datos, modelos de ML). No sabe nada de la interfaz de usuario.
- **Cliente:** Gestiona la presentación (una app móvil, una web en React). No sabe nada de la lógica de negocio interna del servidor.

Se comunican *exclusivamente* a través de la interfaz (la API). Esto permite que ambos evolucionen de forma independiente.

2. *Stateless* (Sin Estado)

Esta es la restricción más crítica.
Significa que el servidor **no debe almacenar ningún estado de la sesión del cliente**. Cada petición (request) enviada por el cliente debe ser **autocontenida** y contener *toda* la

información que el servidor necesita para procesarla.

- **Mal (Stateful):**
    1. Cliente: POST /login?user=pepe
    2. Servidor: (Guarda en memoria: "El cliente 123 es Pepe") -> Respuesta: 200 OK
    3. Cliente: GET /my-profile
    4. Servidor: (Busca en memoria: "El cliente 123 es Pepe") -> Respuesta: Datos de Pepe
- **Bien (Stateless):**
    1. Cliente: POST /login?user=pepe
    2. Servidor: (Valida a Pepe, crea un "Token") -> Respuesta: 200 OK, {"token": "xyz123"}
    3. Cliente: GET /my-profile (Adjunta el token en la cabecera: Authorization: Bearer xyz123)
    4. Servidor: (Decodifica el token, ve que es Pepe) -> Respuesta: Datos de Pepe

En el segundo caso, el servidor no tuvo que *recordar* nada. La petición 3 contenía toda la información necesaria (el token) para ser procesada.
Ventajas: Simplifica el diseño del servidor, mejora la **escalabilidad** (cualquier instancia del servidor puede atender cualquier petición, ya que no hay estado de sesión local) y la **fiabilidad**.

3. *Cacheable*

Las respuestas del servidor deben indicar explícitamente si pueden ser cacheadas (almacenadas temporalmente) por el cliente o por nodos intermedios (Proxies, CDNs).
Esto se hace con cabeceras HTTP (ej. Cache-Control: max-age=3600). Si los datos de una petición GET no cambian frecuentemente, permitir el caché reduce la latencia para el cliente y la carga en el servidor.

4. Interfaz Uniforme

Este es el núcleo de REST y lo que lo diferencia de RPC. La interfaz para interactuar con *todos* los recursos del servidor debe ser la misma. Esta restricción se divide en 4 sub-principios (ver 1.4).

5. Sistema de Capas (Layered System)

El cliente no debe saber (ni importarle) si está hablando directamente con el servidor de la aplicación, o si hay capas intermedias (ej. un Balanceador de Carga, un Proxy de seguridad, una CDN de caché).
Esto permite introducir nuevas capas en la infraestructura (para mejorar la seguridad, el rendimiento o la escalabilidad) sin tener que modificar el cliente.

6. Código bajo demanda (Opcional)

Es la única restricción opcional. El servidor puede, si lo desea, enviar código ejecutable (ej. JavaScript) al cliente para extender su funcionalidad. Esto es, de hecho, cómo funciona la

mayoría de la web moderna (React, Angular, etc.).

1.4. Desglosando la Interfaz Uniforme

Para que la interfaz sea "uniforme", debe cumplir 4 reglas:

1. Identificación de Recursos (URIs)

Cada "recurso" debe tener un identificador único, la **URI (Uniform Resource Identifier)**. En la web, usamos **URLs** (Localizadores).
La clave es que la URI identifica al *recurso* (el concepto, la "cosa"), no a la *acción*.

- **Mal (RPC):** /getUser?id=123, /createUser
- **Bien (REST):** /users/123 (el recurso), /users (la colección de recursos)

2. Verbos (Métodos HTTP): Las Acciones

¿Cómo le decimos al servidor qué *acción* queremos realizar sobre el recurso? Usamos los **métodos (o verbos) HTTP** estándar.
El protocolo HTTP define varias acciones semánticas. Las 4 principales para un CRUD son:

- **GET (Leer):** Pide una representación del recurso. Es **seguro** (no modifica datos) e **idempotente** (múltiples llamadas idénticas tienen el mismo efecto que una sola).
    - GET /users/123 -> Devuelve el usuario 123.
    - GET /users -> Devuelve la lista de usuarios.
- **POST (Crear):** Envía datos a un recurso (normalmente una colección) para crear un *nuevo* recurso subordinado. **No es idempotente** (dos llamadas POST idénticas crearán *dos* recursos nuevos).
    - POST /users (con datos JSON en el *body*) -> Crea un nuevo usuario.
- **PUT (Actualizar/Reemplazar):** Envía una representación completa de un recurso para reemplazar el estado actual del recurso en esa URI. Es **idempotente** (múltiples PUT idénticos del mismo recurso resultan en el mismo estado final).
    - PUT /users/123 (con datos JSON en el *body*) -> Reemplaza totalmente los datos del usuario 123.
- **DELETE (Borrar):** Elimina el recurso en la URI especificada. Es **idempotente** (borrar algo que ya está borrado sigue teniendo el mismo resultado: está borrado).
    - DELETE /users/123 -> Borra el usuario 123.

3. Representaciones (JSON, XML...)

El cliente nunca interactúa con el recurso *directo* (la fila en la BBDD), sino con una *representación* (ej. un JSON).
Los mensajes deben ser **autodescriptivos**. El cliente debe decirle al servidor qué formato le está enviando (ej. Content-Type: application/json) y el servidor debe responder qué formato está devolviendo (ej. Content-Type: application/json).

4. HATEOAS (Hypermedia as the Engine of Application State)

Este es el principio más avanzado y menos implementado, pero fundamental para un REST

"puro".
Significa que la respuesta del servidor no solo debe contener los datos (la representación), sino también **hipervínculos (hypermedia)** que le digan al cliente qué *otras acciones* puede realizar o a qué *otros recursos* puede navegar.
El cliente "descubre" la API navegando por los enlaces, no teniendo las rutas (URIs) *hardcodeadas*.
Ejemplo de respuesta HATEOAS:
Petición: GET /users/123
Respuesta (JSON):

```
{
  "id": 123,
  "name": "Alice",
  "email": "alice@example.com",
  "links": [
    {
      "rel": "self",
      "href": "/users/123",
      "method": "GET"
    },
    {
      "rel": "edit",
      "href": "/users/123",
      "method": "PUT"
    },
    {
      "rel": "delete",
      "href": "/users/123",
      "method": "DELETE"
    },
    {
      "rel": "orders",
      "href": "/users/123/orders",
      "method": "GET"
    }
  ]
}
```

Al recibir esto, el cliente no solo tiene los datos de Alice, sino que *sabe* (sin tenerlo programado de antemano) que puede hacer un PUT o un DELETE a /users/123 o un GET a /users/123/orders para ver sus pedidos.