Programación en Backend

Servidor Web

- Aplicación que se encuentra esperando (ejecución continua) por peticiones de clientes.
- Una vez que recibe una petición, la procesa y luego devuelve una respuesta al cliente que realizó la petición.
- Se pueden utilizar distintos frameworks:

Arquitectura de comunicación

Fastapi

https://fastapi.tiangolo.com/

- FastAPI es un framework web moderno de alto rendimiento para construir APIs con Python basado en los estándares abiertos OpenAPI (antes Swagger) y JSON Schema.
- Beneficios:
  - Velocidad (Performance), al nivel de Node.js
  - Rapidez de desarrollo: Reduce el tiempo de escritura de código entre un 200% y un 300%.
  - Menos errores: Minimiza casi el 40% de los errores inducidos por el desarrollador gracias al tipado.
  - Documentación automática: Solo por escribir el código, ya tienes una interfaz para probar tu API.

Asincronía en Fastapi

- Sigue la misma explicación que el uso de async/await en javascript.

- Se declarar funciones asíncronas con **`async def`** y **`await`** para esperar procesos largos sin detener el servidor.

Ecosistema de Fastapi

- **Pydantic:** Se encarga de la validación de datos. Si dices que un campo es un **int** y te envían un **string**, Pydantic lanza el error antes de que llegue a tu lógica.

- **Uvicorn:** Es el servidor web (ASGI) que permite que FastAPI sea tan rápido y maneje la asincronía.

Instalación

Desde una línea de comandos, creamos el entorno virtual:

```
$ python -m venv env
```

Activamos el entorno virtual

```
$ .\env\Scripts\activate
```

Instalamos FastAPI y Uvicorn:

```
$ pip install "fastapi[all]"
```

El primer API

- **app = FastAPI():** Creamos la instancia principal de la aplicación. Es el punto de entrada de todas las peticiones.
- **@app.get("/"):** Esto es un decorador de ruta. Le dice a FastAPI que la función que está justo debajo se encargará de las peticiones que lleguen a la URL raíz (/) usando el método GET.
- **async def root():** Definimos una función asíncrona. Aunque para un "Hello World" no es estrictamente necesario, es la convención estándar en FastAPI.
- **return {...}:** FastAPI convierte automáticamente los diccionarios de Python a formato JSON, que es el estándar de comunicación en la web.

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    return {"message" : "Hola mundo FastAPI"}
```

Ejecutando API

Iniciando servidor

- main: El nombre del archivo (main.py).
- app: El nombre del objeto que creamos dentro del archivo (app = FastAPI()).
- --reload: Modo desarrollo. Cada vez que guardes un cambio en el código, el servidor se reiniciará automáticamente.
-

```
$ uvicorn main:app --reload
```

¡No olvidarse de tener activo el entorno virtual!

Documentación del API

- Sin escribir una sola línea de código adicional, FastAPI genera documentación profesional.
- Swagger UI: http://127.0.0.1:8000/docs
  - Permite ver todos los "endpoints" disponibles.
  - Permite probar la API directamente desde el navegador (botón "Try it out").
- Redoc: http://127.0.0.1:8000/redoc
  - Una documentación más limpia y orientada a lectura técnica.

Ejercicio 1

- Crear una nueva ruta que cumpla con lo siguiente:
  - Debe responder al método GET en la URL /bienvenida
  - Debe retornar un mensaje que diga: "¡Bienvenido a mi clase de Backend!”.

Envío de datos

Type hints

- A diferencia de Python, en FastAPI tenemos que definir explícitamente los tipos de los datos con los que trabajamos.
- Con esto:
  - FastAPI podrá validar los datos que nos entregue el cliente
  - Hará conversiones de tipo en el caso que lo pueda hacer

Formas de envío de data

- Se podrán enviar por:
  - URL:
    - Query parameters
    - Path parameters
  - *Payload* (cuerpo) de la petición.
    - Vía forms
    - Cruda (*raw*)

Path params

- Normalmente se utilizan para identificar un recurso específico.
- Al declarar user_id: int, FastAPI hace tres cosas:
  - Analiza: Lee el valor de la URL.
  - Valida: Se asegura de que sea un entero.
  - Documenta: En el Swagger (/docs), el parámetro aparecerá marcado como obligatorio y de tipo entero.

```
@app.get("/usuarios/{user_id}")
async def leer_usuario(user_id: int):
    # Se debería devolver un usuario con el id user_id
    return {"user_id": user_id, "mensaje": "Usuario encontrado"}
```

Query params

- Son parámetros opcionales que van después del signo ? en la URL. No se definen en la ruta del decorador, sino como argumentos de la función.
- Valores por defecto:
  - Al poner limit: int = 10, el parámetro se vuelve opcional.
- Optional: Usamos Optional[str] (del módulo typing) para indicar explícitamente que el valor puede ser un string o None.

```
from typing import Optional

@app.get("/items/")
async def listar_items(q: Optional[str] = None, limit: int = 10):
    return {"busqueda": q, "limite": limit}
```

Validaciones

Path() y Query()

- Utilizamos estas funciones para añadir reglas de negocio.
- Parámetros de validación comunes:
  - ge (Greater than or equal): Mayor o igual a.
  - le (Less than or equal): Menor o igual a.
  - min_length / max_length: Para limitar la extensión de strings.

```
@app.get("/productos/{prod_id}")
async def obtener_producto(
    prod_id: int = Path(..., title="ID del producto", ge=1, le=1000),
    tags: list[str] = Query(default=["nuevo", "oferta"])
):
    return {"id": prod_id, "tags": tags}
```

Validación de datos por *payload*

- También se pueden recibir los datos por el cuerpo de la petición.
- Con la librería pydantic, podemos también validar el formato de lo que nos están enviando.
  - Herencia de BaseModel: Es obligatorio para que Pydantic reconozca la clase como un modelo de datos.
  - Validación Automática: Si el cliente olvida enviar title o envía un texto en price, FastAPI responderá automáticamente con un error 422 Unprocessable Entity.
  - Conversión: El JSON entrante se convierte automáticamente en un objeto de Python accesible mediante puntos (ej. item.title).

```
from pydantic import BaseModel, Field
from typing import Optional

class Item(BaseModel):
    title: str
    description: Optional[str] = None
    price: float
    tax: float | None = None

@app.post("/items/")
async def create_item(item: Item):
    # FastAPI ya validó que 'item' cumple con el modelo
    return {"message": "Producto creado", "data": item}
```

Validación de datos por *payload*

Utilización de Field para validaciones extra

- Field para validar los atributos dentro de una clase.
- Ejemplo:
  - example: Es muy útil porque este valor aparecerá por defecto en la documentación de Swagger, facilitando las pruebas.
  - Reglas numéricas y de texto: Se usan las mismas siglas que vimos antes (gt, le, min_length, etc.).

```
class User(BaseModel):
    username: str = Field(..., min_length=3, max_length=20, example="juan_perez")
    email: str = Field(..., pattern=r"^\S+@\S+\.\S+$")
    age: int = Field(None, gt=0, lt=150, description="La edad debe ser un número real")
```

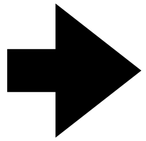
Validación de datos por *payload*

Modelos anidados

- Podemos también construir modelos anidados.

```python
class Image(BaseModel):
    url: str
    name: str

class Item(BaseModel):
    name: str
    tags: list[str] = []
    image: Image | None = None # Un modelo dentro de otro
```



```
{
    "name": "Laptop",
    "tags": ["tech", "work"],
    "image": {
        "url": "http://example.com/img.jpg",
        "name": "portatil-foto"
    }
}
```

Ejercicio 2

- Implementar un endpoint para login.
- Debe tener un username (string, mínimo 5 caracteres).
- Debe tener un password (string, mínimo 8 caracteres).
- Deben crear una ruta POST llamada /login/ que reciba este modelo y devuelva un mensaje de "Acceso concedido" junto al nombre de usuario.

Implementación de CRUD

Create, Retrieve, Update and Delete

- Son operaciones básicas que se hacen con una entidad del backend.
- Primero inicializamos el proyecto y creamos las entidades y su estructura.

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from uuid import uuid4 # Para generar IDs únicos

app = FastAPI()

class Task(BaseModel):
    id: str | None = None
    title: str
    description: str
    completed: bool = False

# Base de datos simulada local
db = []
```

Obtener recursos y crear un recurso

- Dos endpoints: uno que devuelve un grupo de tareas y otro que permite crear una tarea en base a lo que se va a recibir del cliente.
- Tener en cuenta que se devuelve un código de status custom (201).

```
@app.get("/tasks")
async def get_tasks():
    return db

@app.post("/tasks", status_code=201)
async def create_task(task: Task):
    task.id = str(uuid4()) # Generamos un ID único
    db.append(task)
    return task
```

Update y manejo de errores

- En caso de no encontrar un task, se devuelve un código personalizado (en este caso, 404).

```
@app.put("/tasks/{task_id}")
async def update_task(task_id: str, updated_task: Task):
    for index, task in enumerate(db):
        if task.id == task_id:
            updated_task.id = task_id # Mantener el mismo ID
            db[index] = updated_task
            return {"message": "Tarea actualizada"}

    # Si no se encuentra, lanzamos un error 404
    raise HTTPException(status_code=404, detail="Tarea no encontrada")
```

Delete

```
@app.delete("/tasks/{task_id}")
async def delete_task(task_id: str):
    for index, task in enumerate(db):
        if task.id == task_id:
            db.pop(index)
            return {"message": "Tarea eliminada satisfactoriamente"}

    raise HTTPException(status_code=404, detail="No se pudo eliminar: Tarea no encontrada")
```

Ejercicio 3

- Crear el siguiente endpoint:
  - `GET /tasks/{task_id}`: Que devuelva una sola tarea por su ID. Si no existe, debe lanzar un error 404.

Dependencias y Middlewares

Dependencias

- Permite inyectarle lógica antes de la ejecución de determinado endpoint.
- Casos de uso:
  - Verificar autentificación de usuarios
  - Validar parámetros que se repiten en varios endpoints

Dependencias

Para parámetros de paginación

```
from fastapi import Depends, FastAPI, HTTPException

app = FastAPI()

# Definimos la función de dependencia
async def pagination_params(q: str | None = None, skip: int = 0, limit: int = 10):
    return {"q": q, "skip": skip, "limit": limit}

@app.get("/items/")
async def read_items(params: dict = Depends(pagination_params)):
    return {"message": "Listando items", "pagination": params}

@app.get("/users/")
async def read_users(params: dict = Depends(pagination_params)):
    return {"message": "Listando usuarios", "pagination": params}
```

Dependencias

Seguridad de endpoints

```
from fastapi import Header

async def verify_token(x_token: str = Header(...)):
    if x_token != "super-secret-token":
        raise HTTPException(status_code=400, detail="X-Token header invalid")
    return x_token

@app.get("/secure-data/", dependencies=[Depends(verify_token)])
async def get_secure_data():
    return {"data": "Este contenido está protegido"}
```

Ejercicio 4

- Verificar que todos los endpoints se hayan autenticado con anterioridad.
- Para esto, pasar un token estático fijo que demuestre que es una conexión válida (este token debe de obtenerlo luego del login).

Middlewares

- Un middleware es una lógica que se ejecuta para todas las peticiones que lleguen al servidor (sin excepción).
- La diferencia es que las dependencias se pueden definir por endpoint, mientras que los middlewares es para todas las peticiones.
- Además, desde los middlewares no se puede tener acceso al framework de FastAPI (Path, Query, etc).

Middlewares

Devolver tiempo de procesamiento de petición

```
import time
from fastapi import Request

@app.middleware("http")
async def add_process_time_header(request: Request, call_next):
    start_time = time.time()
    response = await call_next(request) # Aquí sigue su camino a la ruta
    process_time = time.time() - start_time
    response.headers["X-Process-Time"] = str(process_time)
    return response
```

Ejercicio 5

- Mejorar la autenticación para realizarla con tokens dinámicos.
- La funcionalidad a implementar es la siguiente:
  - En caso de un login correcto, el endpoint genera un token que contenga el username del usuario y la fecha de login. Esta información tenerla en un string para luego hashearla. Luego el token almacenarlo en listado de accesos.
  - El frontend debe almacenar este token para futuras peticiones.
- Cada vez que se intente consumir otro endpoint, validar que el token exista en el listado. En caso que no exista, retornar status code 403.
- Extra: Definir endpoint de logout que elimine la entrada del token.