Desarrollo Backend

TB022 - Esteban - Riesgo - Kristal

Backend?

Un "backend" es un sistema que contiene data y lógica de negocio para que un producto funcione.

Generalmente está separado del "frontend": sistema que toma data de un backend y tiene lógica de cómo mostrar esa información y de cómo interactuar con el usuario.

Generalmente, proveen una *Application Programming Interface (API)* la cual sirve para definir cómo los clientes del servidor (sean backend o frontend) deben comunicarse para obtener la información que desean.

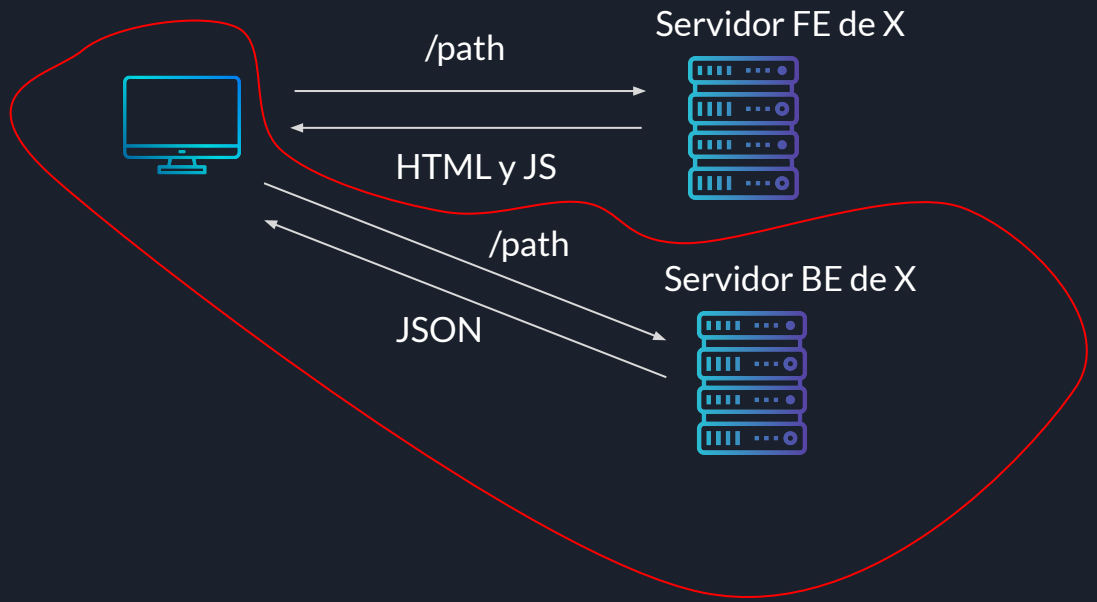
API

Una API provee información de:

- Qué recursos hay disponibles y qué interfaces/métodos existen para interactuar sobre ellos. Éstos se llaman *endpoints*.
- Qué formato e información debe proveer el cliente en su pedido al servidor por estos recursos. Éstos se llaman *requests*.
- Qué formato de información devuelve el servidor ante cada uno de los posibles pedidos al mismo. Éstos se llaman *responses*.

Comunicación con el servidor

Vamos a enfocarnos hoy en esta parte



Cómo crear un servidor

La mayoría de los lenguajes de programación provee en su librería standard mecanismos para poder levantar un servidor, definirle endpoints, sus tipos de responses, y todo lo que queramos.

También existen infinidad de frameworks y librerías que facilitan este asunto, ya que manejar todo manualmente es fastidioso y complicado.

Por qué? Porque el tener que comunicarse a través de Internet, genera que puedan haber una infinidad de situaciones de error que hasta ahora no teníamos en programas convencionales. Es recomendable entonces aprovechar los clientes ya establecidos en vez de hacer todo de cero nosotros (aunque es una buena práctica para aprender también, pero es muy avanzado para nosotros).

Hay muchas opciones para usar, nosotros elegimos usar puntualmente **FastAPI**.

FastAPI

Es un framework muy simple y sencillo.

Permite construir servidores con muy poco código.

Es muy rápido y consume pocos recursos.

Es ideal para aplicaciones pequeñas (como la que vamos a hacer nosotros).

Link oficial

Setup inicial de proyecto

Siempre nos conviene hacer el mismo setup para todos nuestros proyectos de Python.

1. Instalar la versión de Python que queremos con pyenv
2. Crear el virtual env con venv
3. Usar pip para instalar dependencias

1. `pyenv install 3.13.2`
2. `pyenv local 3.13.2`
3. `python -m venv .venv`
4. Crear un archivo `requirements.txt` y agregarle esta linea `fastapi[standard]`
5. `pip install -r requirements.txt`

FastAPI: Hello World

En un directorio exclusivo para nuestro servidor, crear un archivo main.py y colocar lo siguiente:

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def root():
    return {"Hello": "World"}
```

Dependencia con el módulo de FastAPI

Inicializa un servidor

Annotation para definir un endpoint. Todas las request a esta ruta van a ejecutar esta función

Lo que devuelva la función se va a convertir a JSON y va a ser la response a la request recibida

Luego, en una nueva terminal, correr el comando `python -m fastapi dev main.py` para levantar el servidor localmente

Correr comandos con python -m

Cuando corremos comandos que nos descargamos de paquetes (por ejemplo fastapi dev o pytest) estos no siempre detectan correctamente la versión de python a utilizar, por más que tengamos establecidos el virtual env correctamente.

Por lo tanto, recomendamos siempre correr los comandos poniendo primero python -m

Esto hace que se fuerce la versión de python a usar a la que tenemos establecida en el venv.

Por lo que siempre vamos a usarlo para pytest, pip y para fastapi también

```
python -m pip install ...
```

```
python -m pytest ...
```

```
python -m fastapi dev ...
```

Servidor local

http://127.0.0.1:8000

IP puerto

Los servidores montados localmente **no van a estar expuestos a Internet**. Sólo nosotros los podemos acceder.
Sólo van a poder ser accedidos por la máquina local.
127.0.0.1 es una IP ficticia, referencia a la máquina/computadora misma. El sistema sabe interpretarla y no intenta buscarla en Internet. Existe un alias para denominarla: *localhost*.

Cuando el servidor está montado localmente, debemos especificar el ***puerto*** a usar.
Los puertos son lo que permiten que una computadora se conecte a la web.
Hay una cantidad limitada, cada puerto sólo puede ser usado únicamente por una aplicación a la vez.
Existe un standard que todos los sistemas usan donde hay puertos reservados para ciertos usos específicos.
Por ejemplo HTTP utiliza el **puerto 80** siempre. Si hubiera una aplicación usando ese puerto, no podríamos usar un navegador web.

FastApi: trabajar sobre recursos

Un recurso, modelo o entidad es una forma de representar información que es almacenada en nuestro servidor.

Si en nuestro servidor vamos a mantener información de personas, probablemente tendremos una entidad Persona.

Vamos a representar a nuestras entidades como Clases (de Python).

Trabajar con entidades nos permite usar estándares que definen cómo acceder y modificar esa información.

Particularmente nos interesa intentar seguir *REST*. No vamos a adentrarnos en esto pero pueden leer más al respecto acá.

Clases de Python

Una clase es la definición de un nuevo tipo.

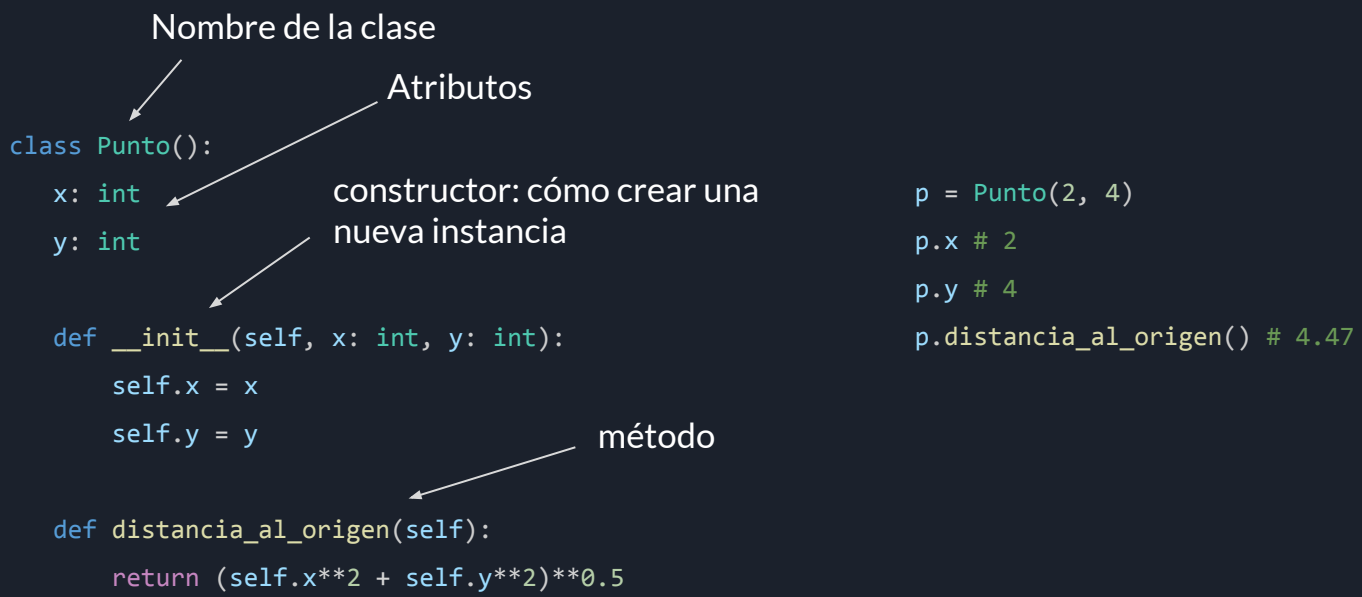
Cuando creamos un string con `variable = "string"` estamos creando una nueva **instancia** de un **objeto** cuya **clase** es `str` (string).

Estos tipos nuevos que nosotros definimos podemos agregarle dos cosas:

estado: definiendole **atributos**

comportamiento: definiéndole **métodos**

Clases de Python




```
p = Punto(2, 4)
p.x # 2
p.y # 4
p.distancia_al_origen() # 4.47
```

Definir entidad

Vamos a usar una herramienta que viene con FastAPI: Pydantic. Nos va a facilitar la validación de tipos y valores en nuestros modelos/entidades.

Definamos entonces un Alumno de la facultad.

```python
from pydantic import BaseModel

class Alumno(BaseModel):
    padron: int
    nombre: str
    apellido: str
    edad: float | None = None
```

Esta sintáxis es válida para Python >=3.10

El campo edad es opcional, puede ser nulo. Su valor default es `None`.
Los demás valores deben estar presentes. No se puede crear una nueva instancia de la entidad sin ellos.

Obtener todas las instancias de la entidad

Creemos un endpoint que nos devuelva todos los alumnos que existan en nuestro sistema.

Primero que nada: donde están todos estos alumnos?

Por ahora, vamos a definir una lista de alumnos *harcodeada* y después vemos más sobre ésto.

```
alumnos: list[Alumno] = [
    Alumno(padron=94557,nombre="Federico",apellido="Esteban"),
    Alumno(padron=95557,nombre="Daniela",apellido="Riesgo"),
    Alumno(padron=99779,nombre="Juan Ignacio",apellido="Kristal")
]
```

El nombre del método no importa, pero es convención que se llame *list*

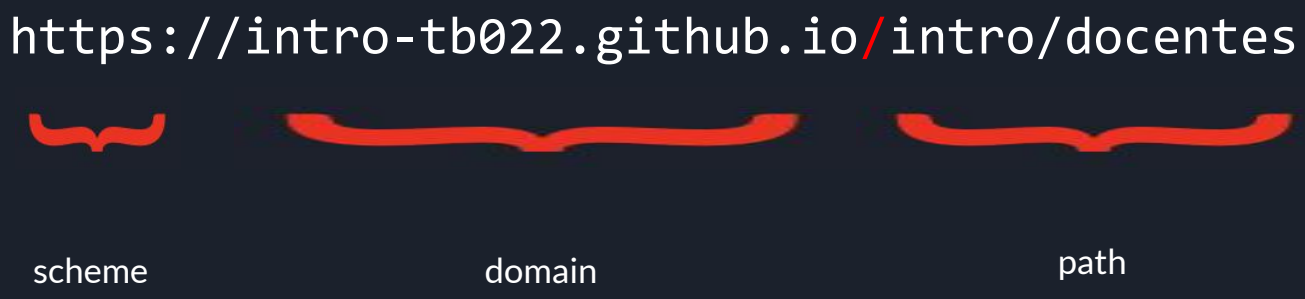
```
@app.get("/alumnos/")
def list() -> list[Alumno]:
    return alumnos
```

PATH del endpoint

FastAPI va a convertir automáticamente lo que devolvamos a formato JSON

URL: Uniform Resource Locator

Una URL es lo que normalmente conocemos y usamos en el día a día en un navegador web. Una URL permite identificar inequívocamente a un recurso en toda la web.



Al definir un servidor, tendremos un set de reglas que van a matchear un endpoint a partir del path de la request hecha.

Obtener todas las instancias de la entidad

Primero que nada: donde están todas estas instancias?

Por ahora, vamos a definir una lista de alumnos *harcodeada* y después vemos más sobre ésto.

```
alumnos: list[Alumno] = [
    Alumno(padron=94557,nombre="Federico",apellido="Esteban"),
    Alumno(padron=95557,nombre="Daniela",apellido="Riesgo"),
    Alumno(padron=99779,nombre="Juan Ignacio",apellido="Kristal")
]
```

```
@app.get("/alumnos/")
def list() -> list[Alumno]:
    return alumnos
```

Todas las requests a nuestro servicio que tengan este path, serán matcheadas con este endpoint

Y si el path de la request no matchea con ningún endpoint?

Status Codes y errores

HTTP define distintos códigos numéricos que pueden ser usados en las responses que genera un servidor.

Lista completa acá

Cada uno tiene su uso particular. Si bien nadie nos obliga a usarlos de ninguna manera en específico, hay ciertas convenciones a seguir y se espera que estos status codes se utilicen correctamente según su definición.

`2xx success` – la request fue recibida, entendida, aceptada y procesada correctamente (todo anduvo bien)

`3xx redirection` – se necesita hacer alguna acción adicional para completar la request

`4xx client error` – la request no está bien armada o no puede ser completada

`5xx server error` – hubo un error fatal en el servidor y no pudo procesar la request válida

Obtener una instancia en particular

Queremos obtener a un alumno puntual según su identificador primario

Qué es un identificador primario?

Generalmente toda entidad tiene algún identificador (ID) único que permite identificarlo inequívocamente de entre otras entidades del mismo tipo.

Todas las personas de un país tienen un número de identificación (DNI o Pasaporte) para identificarlas.

En nuestro modelo de Alumno fiuba, qué podríamos usar?

Obtener una instancia en particular

Queremos obtener a un alumno puntual según su padrón

```python
@app.get("/alumnos/{padron}")
def show(padron: int) -> Alumno:
    for alumno in alumnos:
        if alumno.padron == padron:
            return alumno
```

Misma sintáxis que cadena con formato.

Estamos diciendo que parte del PATH va a ser el ID (el padron) de la entidad Alumno.

Esto se llama *path parameter*.

El framework va a procesarlo por nosotros, validar que sea un entero (fallar si no lo es) y pasarlo como argumento a la función.

El nombre del método no importa, pero es convención que se llame *show*

Qué pasa si el alumno no está? Si no existe un alumno con ese padrón?

Manejo de errores

Cuando no se encuentra un recurso particular que se está buscando, en general se suele devolver un error al cliente.

Como no se encontró la entidad buscada, podemos devolver una respuesta con status code `Not Found (404)`.

Otra corriente de pensamiento podría ser devolver una respuesta exitosa (200) ya que técnicamente el servidor la pudo procesar normalmente, pero que en el cuerpo de la misma se indique un campo error y el motivo del fallo (con algún código puntual que indique en detalle cuál fue el error).

Si queremos ir por el primer camino, el framework nos da una facilidad: podemos levantar una excepción con el detalle del problema y ya nos arma la response por nosotros

```
raise HTTPException(status_code=404, detail="Alumno not found")
```

Obtener una instancia en particular

Queremos obtener a un alumno puntual según su padrón

```python
@app.get("/alumnos/{padron}")
def show(padron: int) -> Alumno:
    for alumno in alumnos:
        if alumno.padron == padron:
            return alumno
    raise HTTPException(
         status_code=status.HTTP_404_NOT_FOUND,
         detail="Alumno not found",
     )
```

Documentación

En desarrollo web, es fundamental documentar las API que exponen nuestros servicios, pues es la única forma que tienen los clientes de saber qué operaciones hay disponibles, entidades, etc.

Por suerte, FastAPI provee un mecanismo de generación automática de documentación siguiendo el standard OpenApi Specification v3

Se puede acceder en el path `/docs`

HTTP methods

Indican qué tipo de acción queremos realizar sobre el path especificado

GET: pedir una representación de uno o varios recursos

POST: "entregar" una instancia de un recurso

PUT: reemplazar todo el contenido de un recurso en su totalidad

PATCH: reemplazar parcialmente un recurso

DELETE: eliminar un recurso

Crear una nueva instancia

Necesitamos tener un endpoint que pueda recibir todos los campos necesarios para crear una nueva instancia de Alumno y que guarde esa información en nuestra "base de datos".

```python
@app.post("/alumnos/", status_code=status.HTTP_201_CREATED)
def create(alumno: Alumno) -> Alumno:
    alumnos.append(alumno)
    return alumno
```

Pydantic se va a encargar de las validaciones por nosotros

El nombre del método no importa, pero es convención que se llame *create*

Deberíamos chequear que no exista un alumno con ese padrón ya.

El path sigue siendo el mismo, pero cambia el verbo que usamos.

Es convención devolver el elemento creado. En este caso es el mismo que recibimos, pero podría tener campos extra que se generan en el servidor.

Cuando creamos una nueva instancia, devolvemos 201, para decir que fue creada correctamente

Un momento, cómo hago para mandar todo un Alumno en una request?

Request body

Las requests pueden enviar información adicional en su cuerpo (body).

Las requests de tipo GET **no** tienen body.

Las requests de tipo POST, PUT y PATCH sí *pueden* tener.

Como estamos haciendo una JSON API, el body de la request también va a tener este formato.

Siguiendo con nuestro ejemplo, un Alumno en formato JSON sería:

```
{
    "nombre": "Juani",
    "apellido": "Kristal",
    "padron": 99779,
}
```

Probar requests de tipo POST, PUT

El navegador web sólo nos permite hacer requests de tipo GET.

Para hacer otro tipo de requests necesitamos usar una herramienta específica.

En nuestro caso, podemos usar los ejemplos interactivos de /docs autogenerados por FastAPI, pero no todos los frameworks tienen algo así disponible.

Hay herramientas genéricas para usarlas que ya mencionamos:

- cURL: es por consola, CLI
- Postman: tiene una interfaz gráfica (GUI), es la herramientas más usada
- Bruno: alternativa a Postman que permite guardar las requests y colecciones en texto plano.

Bruno

Todo lo que hagamos vamos a agregarle también ejemplos de requests hechas en Bruno para el futuro ayudarnos a debuggear más fácilmente nuestro sistema.

Descargar: https://www.usebruno.com/downloads

Cada servidor va a tener una colección de Bruno con al menos un ejemplo de uso de cada endpoint. Esto nos permite probar los endpoints rápidamente y además, de requerirlo, poder tener a disposición tests de integración automáticos para todo el sistema.

Valores autogenerados

Antes vimos como crear un Alumno en nuestro sistema. Pero cuando un alumno se da de alta en la facultad, no tiene un padrón, es el sistema el que le asigna uno.

Podemos cambiar nuestra implementación para reflejar esto.

Por un lado, en la función del endpoint create, asignamos nosotros el padrón:

```
alumno.padron = len(alumnos) + 1
```

(asumiendo que el padrón es el orden de inscripción a la facultad).

Pero además, deberíamos tener un nuevo modelo que se use únicamente en el create y el update y que no contenga el padrón. Para hacer imposible que el usuario setee ese campo:

```
class AlumnoUpsert(BaseModel):
    nombre: str
    apellido: str
    edad: int | None = None
```

Y el endpoint ahora tiene una nueva firma:

```
def create(alumno: AlumnoUpsert) -> Alumno:
```

Update de una instancia

```python
@app.put("/alumnos/{padron}")
def update(padron: int, alumno_actualizado: AlumnoUpsert) -> Alumno:
    for alumno in alumnos:
        if alumno.padron == padron:
            alumno.nombre = alumno_actualizado.nombre
            alumno.apellido = alumno_actualizado.apellido
            alumno.edad = alumno_actualizado.edad
            return alumno
    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Alumno not found")
```

El path sigue siendo el mismo, pero cambia el verbo que usamos.

El padrón no es actualizable, ni debería serlo.

Si no encontramos al alumno por padrón, devolvemos 404.

Pydantic se va a encargar de las validaciones por nosotros

El nombre del método no importa, pero es convención que se llame *update*

Delete de una instancia

```python
@app.delete("/alumnos/{padron}")
def delete(padron: int) -> Alumno:
    for alumno in alumnos:
        if alumno.padron == padron:
            alumnos.remove(alumno)
            return alumno
    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Alumno not found")
```

El path sigue siendo el mismo, pero cambia el verbo que usamos.

Si no encontramos al alumno por padrón, devolvemos 404.

Es convención devolver el elemento borrado.

El nombre del método no importa, pero es convención que se llame *delete*

CRUD

Create

Retrieve (show)

Update

Delete

Custom endpoints

CRUD no son los únicos endpoints que podemos hacer, aunque sí son los más comunes cuando trabajamos con entidades.

Supongamos lo siguiente:

- Además de lo anterior, queremos guardar en cada alumno una lista con sus notas hasta el momento.
- Al ingresar, el alumno no tiene ninguna nota.
- A medida que va aprobando cada materia se agrega la nota de esa materia a la lista.
- Una vez que una nota se agregó, no se puede borrar.

Cómo lo haríamos?

- Podemos usar la función de *update* que me permite actualizar un alumno.
- Podemos crear un nuevo endpoint `/cargar_nota` que lo único que haga sea agregar esa nueva nota al alumno. Cómo debería ser el path de este endpoint?

Cuál idea es mejor?

Custom endpoint: cargar_nota

```python
@app.put("/alumnos/{padron}/cargar_nota")
def cargar_nota(padron: int, nota: int) -> Alumno:
    alumno = buscar_alumno(padron)
    alumno.notas.append(nota)
    return alumno
```

```python
class Alumno(BaseModel):
    padron: int
    nombre: str
    apellido: str
    edad: int | None = None
    notas: list[float] = []
```

`buscar_alumno` es una función auxiliar que hace lo que ya venimos haciendo

Un momento, cómo paso la nota en la request?

Query parameters

Además de todo lo ya existente, una request puede enviar información extra dentro del mismo path de la URL.

```
https://api.com/path/lo_que_sea?arg1=1&arg2=asd&...
```

```python
def lo_que_sea(arg1: int, arg2: str, ...)
```

Cualquier tipo de request puede tener query parameters, no importa el verbo.

Se suelen usar para campos simples, pero pueden incluir objetos complejos.

Testing

FastAPI nos provee muchas facilidades para hacer tests de nuestra aplicación.

Depende qué tan complejo nuestro servidor sea, nuestros tests serán más o menos complejos también.

Se recomienda lo siguiente:

- Cada archivo de código debe tener su contraparte `test_<archivo>.py` con los tests para el mismo.
- Si hay varias capas en nuestra aplicación, recomendamos mockear llamadas a capas inferiores (hacer mocks de las clases que sean llamadas por la clase que estamos probando).

En nuestra versión inicial de servidor no usaremos mocks porque la aplicación es muy sencilla.

Testing

Probemos que el endpoint de mostrar Alumnos devuelva la información que queremos

```python
from main import app

client = TestClient(app)

def test_get_alumnos() -> None:
    response = client.get(
        "/alumnos/",
    )
    assert response.status_code == 200
    content = response.json()
    assert len(content) == 3
```

Además deberíamos chequear que nos devuelva los alumnos que nosotros queremos, no sólamente 3 cualesquiera.

El mismo patrón lo podemos aplicar para los demás tests.

Setup (vacío)

Execute

Verify

Testing

Testeemos que podemos agregar un nuevo alumno

```python
def test_create_alumno() -> None:
    data = {"nombre": "Test", "apellido": "Alumno", "edad": 19}
    response = client.post(
        "/alumnos/",
        json=data
    )
    assert response.status_code == 201
    content = response.json()
    assert content["nombre"] == "Test"
    assert content["apellido"] == "Alumno"
    assert content["edad"] == 19
    assert content["padron"] == 4
```

Setup

Execute

Verify

Testing

Testeemos que podemos agregar un nuevo alumno

```python
def test_create_alumno() -> None:
    data = {"nombre": "Test", "apellido": "Alumno", "edad": 19}
    response = client.post(
        "/alumnos/",
        json=data
    )
    assert response.status_code == 201
    content = response.json()
    assert content["nombre"] == "Test"
    assert content["apellido"] == "Alumno"
    assert content["edad"] == 19
    assert content["padron"] == 1
```

IMPORTANTE

Como estamos guardando nuestros alumnos en una variable global, si modificamos esa variable en nuestros tests, los demás tests de van a ver afectados.

Eso no está nada bueno, porque el orden en que corremos los tests afecta el resultado, y nuestros tests dejan de ser realmente tests unitarios.

Podemos hacer un Monkey Patch es nuestros tests para solucionarlo, pero también podemos reordenar la estructura de nuestro proyecto para dejar mejor organizado el código, y hacer la menor cantidad de patches posibles.

Tests - qué hay que probar?

Los casos a testear van a depender del tipo de endpoint que estemos testeando (justamente), pero hay ciertas cosas generales que aplican para todos:

- todo flujo que pueda dar un error (por cada error distinto deberíamos tener un test)
- todo flujo que sea exitoso usando distintos parámetros (si al crear una nueva entidad pasan cosas distintas si le paso el parámetro A o el B; si al listar hay suficientes elementos, o ninguno, o uso filtros o paginado; etc.

Además, en cada caso de prueba necesitamos validar correctamente el resultado de la operación. Para endpoints, esto implica:

- ver que la response del servidor sea correcta (tenga el status correcto)
- ver que el contenido del body de la response sea el correcto (que tenga los elementos esperados, en el orden esperado, etc.)

Estructura de proyecto y buenas prácticas

Estructura de proyecto

Escribir toda una aplicación en un main no es una gran idea para cualquier aplicación que no sea super trivial.

Hay muchas cuestiones a tener en cuenta que podemos hacer para dividir el código que necesitemos tener.

Generalmente vamos a tener 3 partes (o también dicho capas, niveles) bien diferenciadas en nuestros proyectos que sean mínimamente grandes:

- Una primer capa que define los endpoints, valida las requests y devuelve las responses
- Una capa intermedia que se encarga de la "lógica de negocio" todo lo que es necesario para computar el resultado a devolver en la response a partir de los datos recibidos en la request
- Una tercer capa, la más inferior, que se encarga de leer y escribir datos en donde sea que los estemos almacenando.

Para endpoints CRUD o flujos muy simples la capa intermedia no es realmente necesaria. Cobra sentido cuando se hacen cosas más complejas.

Centrémonos por ahora entonces en hacer las otras dos.

Routers

Para separar y agrupar las rutas, podemos usar routers, una facilidad que nos provee FastAPI pero que es un concepto que se usa en muchos proyectos. Es simplemente una forma de organizar endpoints. Se suelen organizar según recurso (todos los endpoints de alumno irán a un router, los de otra cosa al suyo, etc.)

Por ejemplo:

```
api_router = APIRouter()

api_router.include_router(alumnos.router, prefix="/alumnos", tags=["alumnos"])

api_router.include_router(grupos.router, prefix="/grupos", tags=["grupos"])
```

Y en el main

```
app = FastAPI()
app.include_router(api_router)
```

Todos los endpoints de este router tendrán este prefijo, no es necesario aclararlo en el endpoint.

De manera que el directorio será algo como:

```
app/
     routes/
          alumnos.py
          grupos.py
          etc
     main.py
```

Datos: modelos

Primero que nada, podemos agrupar la definición de nuestros modelos en un directorio de modelos y organizarlos por módulos según la clase, etc. Igual a como hicimos con los Routers.

```
app/
     routes/
     modelos/
          alumno.py
          grupo.py
          materia.py
          etc
     main.py
```

Datos: manejo

Para manejar la lectura y actualización de los datos vamos a tener una capa que tenga esas acciones como su único fin. Un nombre que se le suele dar a esta capa es de Repositorio. Pero también le podemos decir Base De Datos, o lo que querramos.

Esta capa va a tener todas las acciones que querramos hacer con nuestros datos.

Creemos una clase Database

```
class Database:
    def __init__(self):
        self.alumnos = self.cargar_alumnos()

    def cargar_alumnos(self):
        return [
            Alumno(padron=94557, nombre="Federico", apellido="Esteban", edad=32),
            Alumno(padron=95557, nombre="Daniela", apellido="Riesgo", edad=30),
            Alumno(padron=98713, nombre="Juan Ignacio", apellido="Kristal", edad=27)
        ]
```

Por ahora la información la seguimos hardcodeando en memoria.
Igual, ya tenemos una ventaja: los datos están atados a una instancia puntual, no son globales. Ya no tendremos problemas con los tests.

Datos: Repository/Database

Agreguémosle algunos de los métodos que fuimos haciendo:

```python
class Database:
    def list(self) -> list[Alumno]:
        return self.alumnos

    def add(self, alumno_a_crear: AlumnoUpsert) -> Alumno:
        padron = len(self.alumnos) + 1
        alumno = Alumno(**alumno_a_crear.model_dump(), padron=padron)
        self.alumnos.append(alumno)
        return alumno

    def find(self, padron: int) -> Alumno:
        for alumno in self.alumnos:
            if alumno.padron == padron:
                return alumno
        raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail="Alumno not found")
```

Y el router nos va a quedar más escueto:

```python
@router.get("/")
def list() -> list[Alumno]:
    return db.list()

@router.get("/{padron}", responses={status.HTTP_404_NOT_FOUND: {"model":
Error}})
def show(padron: int) -> Alumno:
    return db.find(padron)

@router.post("/", status_code=status.HTTP_201_CREATED)
def create(alumno_a_crear: AlumnoUpsert) -> Alumno:
    alumno = db.add(alumno_a_crear)
    return alumno
```

Datos: Repository/Database

Datos: Repository/Database

Opción 1

Instancio la DB en el módulo database.py en una variable global

```
db = Database()
```

Y lo uso directamente en el router importándolo

```
from database.database import db

@router.get("/")
def list() -> list[Alumno]:
   return db.list()
```

Sigo usando la variable global. No gané nada.

Datos: Repository/Database

Opción 2 (Más recomendada)
Instancio la clase en algún módulo común a todas las rutas

```
db = Database()
```

Y para usarlo agrego esta "Dependency"

```
def get_database() -> Database:
    global database_instance
    if database_instance is None:
        raise RuntimeError("Database instance not initialized.")
    return database_instance


DatabaseDep = Annotated[Database, Depends(get_database)]
```

Cada endpoint tendrá un nuevo argumento a esto:

```
@router.get("/")
def list(db: DatabaseDep) -> list[Alumno]:
    return db.list()
```

Dependency

Me permite agregar o "inyectar" cosas en donde normalmente no tendría forma de accederlas

```python
def get_database() -> Database:
    global database_instance
    if database_instance is None:
        raise RuntimeError("Database instance not initialized.")
    return database_instance

DatabaseDep = Annotated[Database, Depends(get_database)]

@router.get("/")
def list(db: DatabaseDep) -> list[Alumno]:
    return db.list()
```

Ésta es la función que me va a devolver la instancia que necesito. En este caso es la que se crea en el módulo database

Esta definición me crea una dependencia hacia esta función que definí previamente.

Esta dependencia es la que puedo empezar a recibir como parámetro en cualquiera de las funciones que defina en los routers. Este argumento es "invisible". Se inyecta automáticamente. Quien llame a este método (que no somos nosotros, es interno al Framework de FastAPI), no tiene que pasar este valor.

Lo que ganamos con ésto es que ahora podemos tener tests de nuestros routers independientes entre sí porque dejamos de usar una variable global y dependemos de un objeto concreto.

Donde setupear las dependencias?

Por ahora tenemos una sola, pero podríamos tener más. Hagamos un módulo cerca del main que se encargue de eso.

Cambiemos main.py para que tenga algo asi:

```
def main():
    init_dep()

app = FastAPI()
app.include_router(api_router)

main()
```

Y todo el código que vimos antes lo podemos poner en un modulo dependencies.py o similar:

```
database_instance = None

def init_dep():
    global database_instance
    database_instance = Database()

def get_database() -> Database:
    global database_instance
    if database_instance is None:
        raise RuntimeError("Database instance not initialized.")
    return database_instance

DatabaseDep = Annotated[Database, Depends(get_database)]
```

main.py

Para inicializar valores que sólo se ejecuten cuando se inicia la aplicación y no cuando se importa el módulo hay que usar un par de cosas especiales de FastAPI.

```python
from contextlib import asynccontextmanager

from fastapi import FastAPI

from dependencies.dependencies import init_dep
from routes.routes import api_router
from seed import seed

@asynccontextmanager
async def lifespan(app: FastAPI):
    init_dep()
    seed()
    yield

app = FastAPI(lifespan=lifespan)
app.include_router(api_router)
```

Es parecido a como funciona `if __name__ == "__main__"` para cuando ejecutamos un servidor de FastAPI, de la forma que sea.

Todo lo que pongamos en la función lifespan antes del `yield` se va a ejecutar al iniciar la aplicación.
Todo lo que pongamos después del mismo, cuando la aplicación finalice.

> `yield` es bastante complejo de explicar, pueden intentar ver este post al respecto

Testing: ahora sí

Podemos mockear fácilmente la dependencia a la db desde cualquier test de un Router.

```
mock_db = MagicMock(Database)
app.dependency_overrides[get_database] = lambda: mock_db


def test_get_alumnos():
    mock_db.list.return_value = [
        Alumno(padron=1, nombre="Juan", apellido="Perez", edad=20),
        Alumno(padron=2, nombre="Maria", apellido="Lopez", edad=22)
    ]

    response = client.get(
        "/alumnos/",
    )

    assert response.status_code == 200
    content = response.json()
    assert len(content) == 2
```

Generamos un mock de una Database para devolver

Como tenemos una Dependency, podemos mockearla con este helper

Finalmente hacemos el test. Podría testear también que cada alumno devuelto sea el que yo esperaba (los devueltos por el mock).

Testing: otro ejemplo

Podemos mockear fácilmente la dependencia al repository desde cualquier test de un Router.

```
mock_db = MagicMock(Database)
app.dependency_overrides[get_database] = lambda: mock_db

def test_get_alumno():
   mock_db.find.return_value = Alumno(padron=1, nombre="Juan",
apellido="Perez", edad=20)

   response = client.get(
       "/alumnos/1",
   )

   assert response.status_code == 200
   content = response.json()
   assert content["padron"] == 1
   assert content["nombre"] == "Juan"
   assert content["apellido"] == "Perez"
   assert content["edad"] == 20
```

Estas dos lineas son comunes, no hay que repetirlas en cada test

El test de obtener un alumno se fija que el alumno devuelto sea el deseado con los campos esperados (si espero que el endpoint me devuelva por cada Alumno sus notas también, por ejemplo, tengo que agregarlo al test como parte del contenido a verificar.

Testing: database

Ahora veamos cómo serían los tests de la clase Database

Datos básicos que vamos a necesitar

```
juan =
       Alumno(padron=1, nombre="Juan", apellido="Perez", edad=20)
juan_upsert =
       AlumnoUpsert(nombre="Juan", apellido="Perez", edad=20)
pepito =
       Alumno(padron=2, nombre="Pepito", apellido="Equis", edad=21)
pepito_upsert =
       AlumnoUpsert(nombre="Pepito", apellido="Equis", edad=21)

@pytest.fixture
def db():
   return Database()
```

Testing: database

Ahora veamos cómo serían los tests de la clase Database

```python
class TestDatabase:
    def test_list_vacio(self, db):
        assert db.list() == []

    def test_list_no_vacio(self, db):
        db.cargar_alumnos([juan])

        assert db.list() == [juan]

    def test_add(self, db):
        db.add(juan_upsert)

        assert db.list() == [juan]

    def test_add_autoincrementa_padron(self, db):
        db.add(juan_upsert)
        db.add(pepito_upsert)
        db.add(AlumnoUpsert(nombre="Fake", apellido="Alumno", edad= 50))

        assert [a.padron for a in db.list()] == [1, 2, 3]
```

```python
def test_find_valido(self, db):
    db.add(juan_upsert)
    db.add(pepito_upsert)

    assert db.find(2) == pepito

def test_find_no_existente(self, db):
    db.add(juan)

    try:
        db.find(3)
        assert False
    except Exception as e:
        assert str(e) == "404: Alumno not found"
```

Datos: cargar la información

Tener los datos hardcodeados en el código no está bueno, aún para una aplicación de prueba.

De hecho, es muy común que un proyecto de la vida real, requiera tener ciertos datos cargados de antemano antes de ejecutarse por primera vez. Estos datos pueden estar en archivos de texto que podemos leer y cargar en nuestro sistema cuando se inicie la aplicación.

Esto se lo conoce como *seed*. Un seed es un conjunto de datos iniciales (que puede estar en cualquier lado y tener cualquier forma) que necesita un sistema para funcionar y que se pueden cargar en la aplicación de distintas maneras.

Seed: versión inicial

Carguemos los mismos datos que veníamos usando pero estructurando el código para usar un script de seed.

Agreguemos un módulo seed.py con nuestra lógica:

```
def seed():
    alumnos = [
        Alumno(padron=94557, nombre="Federico", apellido="Esteban", edad=32),
        Alumno(padron=95557, nombre="Daniela", apellido="Riesgo", edad=30),
        Alumno(padron=98713, nombre="Juan Ignacio", apellido="Kristal", edad=27)
    ]
    get_database().cargar_alumnos(alumnos)
```

Y agregamos la llamada en el main luego de cargar las dependencias:

```
def main():
    init_dep()
    seed()
```

Tenemos que exponer este método porque no podemos usar el add normalmente, porque no lleva padrón (es un detalle de nuestro caso, no siempre va a pasar esto).

Seed: desde archivo

En muchas situaciones nos vamos a encontrar con que tenemos mucha información para cargar, por lo que uno de los mejores caminos es usando archivos de texto y cargar la data de allí.

Teniendo un archivo `alumnos.csv` con este formato y encabezado:
Padron;Apellido;Nombre;Email

podemos tener un script muy sencillo para cargar esta información en nuestro sistema.
Simplemente vamos al modulo seed.py y cambiamos lo que tenemos por algo así:

```
def seed():
    cargar_alumnos(os.path.join(repository_root, "resources", "alumnos.csv"))

def cargar_alumnos(path):
    alumnos = []
    with open(path) as f:
        csvFile = csv.DictReader(f, delimiter=";")
        for linea in csvFile:
            alumnos.append(
                Alumno(
                    padron=int(linea["Padron"]),
                    nombre=linea["Nombre"],
                    apellido=linea["Apellido"],
                    edad=random.randint(18, 35)
                )
            )
    get_database().cargar_alumnos(sorted(alumnos, key= lambda x: x.padron))
```

Ejemplo completo

El ejemplo completo que fuimos iterando lo pueden encontrar en
https://github.com/intro-tb022/fastapi_example

Hay varias branches con distintas versiones.

La branch `basic-with-structure` sigue el ejemplo que vimos.
La branch `basic-with-multiple-models` agrega un nuevo modelo a la aplicación y sirve como ejemplo para ver cómo relacionar múltiples entidades entre sí.

Ejercicio

Partiendo del repositorio https://github.com/intro-tb022/fastapi_example y de la branch `basic-with-multiple-models`, hacer lo siguiente:

1. Hacer un fork del repositorio.
2. Agregar las siguientes features siguiendo las buenas prácticas de código (incluyendo tests correspondientes) en una branch distinta partiendo de `basic-with-multiple-models`
    a. Agregar el campo email al modelo Alumno. Cuando se inscribe un alumno a la facultad, se le asigna un email con formato inicial del nombre + apellido + "@fi.uba.ar". El campo no es editable.
    b. Agregar materias al sistema.
        i. El modelo `Materia` tiene un nombre y un código.
        ii. Cada Alumno puede inscribirse y desinscribirse a múltiples Materias según su código.
        iii. Cada Grupo de tp de ahora en más también es de una Materia puntual. Al crear un Grupo hay que especificar para qué Materia es. Guardar la información de manera relevante en el modelo.
        iv. Para cada Materia puedo consultar cuáles son sus alumnos inscriptos, y en un Alumno la recíproca.
3. Hacer un PR de su branch del fork contra la branch `basic-with-multiple-models` del repositorio original.

Bonus Tracks

Documentación de errores

Si revisamos la documentación auto generada de nuestros endpoints, veremos que en los que nosotros devolvemos errores manualmente (por ejemplo, al buscar un alumno por padrón y no encontrarlo) éstos no aparecen allí.

Para que se muestren, debemos agregar los distintos errores que devolvemos a la información de cada endpoint.

```
class Error(BaseModel):
    detail: str

@app.get("/alumnos/{padron}", responses={status.HTTP_404_NOT_FOUND: {"model": Error}})
def show(padron: int) -> Alumno:
```

Estamos especificando que este endpoint puede devolver 404 y el tipo de respuesta es la indicada por el modelo `Error`

Modelos parciales

No todos los endpoints relacionados con la misma entidad, van a querer devolver la misma información de esa entidad.

El listado de alumnos puede mostrar sólo la información básica de cada alumno, pero en el detalle de cada alumno quiero ver más información, por ejemplo, el listado de grupos al que pertenece un alumno.
Para limitar la información mostrada, puedo usar distintos modelos de Alumno.

```python
class AlumnoBase(BaseModel):
    padron: int
    nombre: str
    apellido: str
    edad: int

class AlumnoFull(AlumnoBase):
    grupos: list[GrupoBase] | None = []
```

```python
class GrupoBase(BaseModel):
    id: int
    nombre: str
    nota: int
```

```python
@router.get("/")
def list(...) -> list[AlumnoBase]:

@router.get("/{padron}")
def show(...) -> AlumnoFull:
```

Idempotencia

Idempotency is a mathematical and computer science concept that means an operation can be repeated multiple times without changing the outcome.

En en el contexto del desarrollo web:

Una operación (una request a un endpoint) es idempotente si hacer la operación múltiples veces tienen el mismo efecto que hacer una vez. El estado del servidor se verá modificado al menos una sola vez.

La idempotencia es importante para entender procesos que se pueden repetir (por ejemplo, en casos de falla parcial) y los que no.

Las requests de tipo GET son idempotentes. Alguna más? UPDATE también

Exponer un servidor local a la Internet

Necesitamos hacer un port forwarding en nuestro router con el puerto sobre el cual se monta el servidor (cómo hacerlo depende del router).

Una vez logrado esto cualquiera puede acceder a través de la Public IP de nuestra computadora (se puede consultar acá whatismyip).

Si no, hay herramientas que lo hacen por nosotros:

http://localtunnel.me/

https://ngrok.com/

http://localhost.run/

Ignorar archivos en VSCode

En proyectos de este estilo, suele pasar que tenemos archivos o directorios enteros que son auto generados, temporales, u otros que no queremos ver.

Por un lado los agregado al .gitignore pero seguimos viéndolos en nuestro repositorio local.

Podemos por lo menos ignorarlos desde la vista del IDE, o en VSCode por lo menos:

Code Manage > Settings > Workspace, search files:exclude

Redirects

TODO

Fin