TALENTOFUTURO

Docentes:
Roberto Arce
Pablo Cruz

Clase 5: REST y servicios FastAPI

Apoyado por:
CORFO

Módulo 4: Desarrollo Backend en Entornos Python

Clase de hoy

01

REST y FastAPI

Introducción a las arquitecturas REST y el uso de FastAPI.

02

Implementación de un caso

Implementación de un caso REST con FastAPI.

5A

Introducción a REST y FastAPI

Bloque A

Qué veremos en Bloque A

- Introducción a REST
- Introducción a FastAPI
- Creación de una aplicación/servicio con FastAPI

REST

REpresentational State Transfer

- REST es un estilo de arquitectura propuesto por Roy Fielding en su tesis doctoral en el año 2000.
- Como todo estilo de arquitectura de software, REST impone algunas reglas sobre los componentes y conexiones entre estos.
- Típicamente escuchamos la frase “API REST”, aunque normalmente esto es una generalización demasiado ambigua.
- NO existen las APIs REST: ¡existen las arquitecturas REST!

REST: https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

REST

REpresentational State Transfer

- Una de las principales características de REST es el uso de “recursos” para el flujo de información.
- Por lo tanto, si una arquitectura será REST, esta debe orientarse hacia el uso de recursos y no procedimientos.
- ¿Cómo determinamos estos recursos?
    - ¡En las historias de usuario!
    - ¡En los casos de uso!
- Por esto es de suma importancia la correcta escritura de historias de usuario y casos de uso.

REST

REpresentational State Transfer

- **¿Y si no son recursos? ¿Qué otra forma existe?**
  - Procedimientos: el llamar a un procedimiento o función, típicamente remota, que realiza algo y nos entrega un resultado computado.
- **Entonces, ¿cuál es la propuesta de la orientación a recursos?**
  - La orientación a recursos delega responsabilidad: si somos el cliente, el servicio nos dice: “tome el recurso, y usted haga lo que desee con él (incluso realizar cálculos con este)”.

REST

REpresentational State Transfer

- Además de la orientación a recursos, REST nos dice que también debemos trabajar con “representaciones” de recursos, y no con recursos directamente.
- Es decir, si existe el recurso Libro (título, isbn, autor, editorial, edición, año, rating[0,5]), la representación podría ser Libro (título, rating[verde, amarillo, rojo], autor).
- Nuestras operaciones se realizarán sobre la representación y NUNCA sobre el recurso directamente.

REST

REpresentational State Transfer

- Los recursos se identifican con una URI (uniform resource identifier) que típicamente consiste de una dirección IP o dominio más un conjunto de elementos separados por “/”.
- REST también obliga el uso de verbos de HTTP para dar la semántica completa: GET, POST, PUT, DELETE más el recurso.
- Por ejemplo, si la historia de usuario dice “deseo poder agregar un libro” entonces debemos usar un verbo HTTP adecuado (e.g., POST) más el recurso:

→ POST /Book

¡Endpoints en inglés!

REST

REpresentational State Transfer

- Finalmente, REST fomenta el principio HATEOAS: Hypermedia As The Engine Of The Application State.
- REST es en esencia una arquitectura “stateless”.
- Una ventaja importante de REST siendo “stateless” es que las llamadas ¡se pueden realizar en forma independiente!
- Pero todo siempre tiene su desventaja ☹: cada llamada debe caracterizar completamente la petición.

REST

Diseño de APIs

- Dominios:
  - usar subdominios para agrupar lógicamente recursos
    - http://sp.myserver.com/books
    - http://en.myserver.com/books
- En la ruta (o sea, el nombre del método):
  - Usar “/” para indicar jerarquía entre recursos.
    - /world/project/milestones

REST

Diseño de APIs

- **En la ruta (o sea, el nombre del método):**
  - Usar “,” y “;” para indicar elementos del mismo nivel:
    - /value;x=0,y=10
  - Usar “–” o “ ” para mejorar la lectura de nombres largos
    - /project/research-Project
  - Evitar uso de extensión de archivos
    - /image (en vez de /image.jpg)

REST

Diseño de APIs

- En la consulta (o sea, los parámetros del método):
  - Usar “&” para separar parámetros
    - /book?isbn=...&edition=..

FastAPI

Escribiendo servicios pequeños (¿microservicios?)

- FastAPI es un framework de Python que permite la escritura de servicios pequeños y altamente escalables.
- Aunque FastAPI es más bien agnóstico respecto del tipo de arquitectura, típicamente se utiliza para construir arquitecturas altamente escalables basadas en microservicios y REST.
- Lo primero es lo primero... debemos instalar FastAPI:

```
$ pip install fastapi
```

FastAPI

Escribiendo servicios pequeños (¿microservicios?)

- FastAPI es el framework, pero necesitamos un servidor que nos permita levantar la aplicación.
- Usaremos un servidor “production-ready” llamado “uvicorn”:

```
$ pip install uvicorn
```

FastAPI

Escribiendo servicios pequeños (¿microservicios?)

- Ya conocemos el concepto de modelo: también los usaremos en FastAPI.
- Los modelos se desarrollarán con la biblioteca pydantic (no es necesario instalarla ya que viene con FastAPI).

```
$ pip install pydantic
```

FastAPI

Levantemos un servicio básico

```
# myfirstapp.py
from fastapi import FastAPI
app = FastAPI()

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

Servimos con: `$ uvicorn myfirstapp:app --host 127.0.0.1 --port 8000 --reload`

FastAPI

Levantemos un servicio básico, pero un poco más elaborado...

```
# myfirstapp.py
from fastapi import FastAPI
from fastapi.responses import HTMLResponse
app = FastAPI()

@app.get("/")
def read_root():
    html_response = """
    <html>
        <title>Hello</title>
        <body>
            <p>Hello World!</p>
        </body>
    </html>
    """

    return HTMLResponse(html_response, status_code=200)
```

FastAPI

Operaciones GET con un parámetro...

```
# myfirstapp.py
@app.get("/profile/{name}")
def read_root(name: str):
    html_response = """
    <html>
        <title>Hello</title>
        <body>
            <p>Hello World, {name}!</p>
        </body>
    </html>
    """.format(name=name)

    return HTMLResponse(html_response, status_code=200)
```

FastAPI

Operaciones GET con dos parámetros...

```
# myfirstapp.py
@app.get("/profile/{name}/{lastname}")
def read_root(name: str, lastname: str):
    html_response = """
    <html>
        <title>Hello</title>
        <body>
            <p>Hello World, {name} {lastname}!</p>
        </body>
    </html>
    """.format(name=name, lastname=lastname)

    return HTMLResponse(html_response, status_code=200)
```

FastAPI

Respondiendo con distintos status codes...

```
# myfirstapp.py
from fastapi import FastAPI, HTTPException
…
@app.get("/profile/{name}/{lastname}")
def read_root(name: str, lastname: str):
    if len(name) < 3 or len(lastname) < 3:
            raise HTTPException(status_code=400, detail="Hey! I was supposed to
            see your name!")

    html_response = """
    <html>
        <title>Hello</title>
        <body>
            <p>Hello World, {name} {lastname}!</p>
        </body>
    </html>
    """.format(name=name, lastname=lastname)

    return HTMLResponse(html_response, status_code=200)
```

FastAPI

Veamos una operación POST...

- Las operaciones POST son algo más complejas.
- FastAPI nos pide que manejemos los datos de la consulta por medio de un modelo.
- El modelo será siempre subclase de la clase BaseModel de pydantic.

```
from pydantic import BaseModel
```

FastAPI

Veamos una operación POST... (Ejemplo de un modelo)

```
from pydantic import BaseModel
class Profile(BaseModel):
    name: str
    lastname: str
    age: int
```

FastAPI

Veamos una operación POST... (Ejemplo de un modelo)

- Luego, el endpoint POST queda:

```
@app.post("/")
def read_profile(profile: Profile):
    return {"name": profile.name, "lastname": profile.lastname,
            "age": profile.age}
```

FastAPI formatea automáticamente en JSON válido una respuesta que es transformable a JSON. En casos más específicos es posible utilizar JSONResponse.

Trabajo grupal – Bloque A

Paralelo 2

G1	G2	G3	G4
Nicolas Mardones	Víctor Meza Herrera	Daniela Méndez Gándara	Claudia Blanco
Manuel Denis	Estefania Manriquez	Álvaro Pérez	Ariel Inostroza
Bryan Castillo	Patricio Vera	Pedro Nahum	Héctor Aguayo
GERALDY SUAREZ	Oscar Torres	Javier Gajardo	Ruben Sanhueza Ramirez
Scarlett Espinoza	Braulio Quiroz	Angela Proboste Neira	Félix González
Ulises Campodónico	Yerko Gallardo	Nicolás Guzmán	Ariel Mora
Carol Leiva	Rodrigo Araya		

G5	G6	G7	G8
Fabian Díaz	Camila Oyarzún	Mayerlyn Rodriguez	Daniela Porto
Natalia Rivera	Stefanya Pulgar	Sebastian Vega	Cristian Chavez Jara
Juan Salinas	Carlos Emilio Azócar Riquelme	Efrain Duarte Campos	Juan Rodrigo Vega
Rodrigo Pastén Cortés	Nicolas Rojas	Bianel Bianchini	Rodolfo Cantillana
Flavio Jara R.	luis.paillan.cnc@gmail.com	Bastián Gamboa Labbé	Abraham Ruiz
Daniel García	Cristóbal Gajardo	Pablo Uribe	Rodrigo Álvarez

Trabajo grupal – Ejercicio #1

Crear un endpoint simple (GET)

1. Escriba el código para implementar un endpoint de tipo GET que reciba tres parámetros (nombre del libro, autor, fecha de publicación) de tipo “path”.

2. Su endpoint debe responder con con HTML-compatible presentando el nombre del libro, autor, fecha de publicación.

Trabajo grupal – Ejercicio #2

Crear un endpoint simple (GET)

1. Modifique el código anterior para que la respuesta sea JSON compatible.

Trabajo grupal – Ejercicio #2

Crear un endpoint simple (POST)

1. Escriba el código para implementar un endpoint POST que permita recibir un objeto “búsqueda” con los siguientes elementos: fecha inicial de publicación, fecha final de publicación.

2. Por ahora, su código debe crear una lista de libros con fechas de publicación.

3. El endpoint debe responder con el primer libro que se encuentre entre las fechas de inicio y fecha final.

4. Si no se encuentra un libro, debe responder con status code 404: not found.

5. Discuta: ¿por qué la búsquesa se implementa como POST y no GET?

Break!

5B Implementación de un caso REST con FastAPI

Caso

- Implementaremos un servicio “Paciente” para una clínica.
- Inicialmente, el caso será muy sencillo.
- El servicio es “stand-alone”, es decir, similar a un microservicio.
- Utiliza una base de datos sqlite.
- Provee endpoints para Pacientes y Exámenes.

Caso

Estructura del servicio

- Archivo models.py contiene todos nuestros modelos.
- Archivos *_service.py: contienen los servicios de storage (comunicación con la base de datos).
- Archivo request_handler.py implementa los endpoints.
- Archivo domain.py contiene las clases del dominio (distintas a las clases de modelos).
- Archivo tf_backend_api.db contiene la base de datos sqlite.

Caso

Estructura del servicio

- Archivo models.py contiene todos nuestros modelos.
- Archivos *_service.py: contienen los servicios de storage (comunicación con la base de datos).
- Archivo request_handler.py implementa los endpoints.
- Archivo domain.py contiene las clases del dominio (distintas a las clases de modelos).
- Archivo tf_backend_api.db contiene la base de datos sqlite.

Caso

¡Vamos a codificar!

Continuará...

- Esta clase era una introducción a REST y su implementacion con FastAPI.
- Seguiremos desarrollando el caso REST en otra clase.
- ¡No perdamos el código desarrollado hasta ahora!

Trabajo grupal – Bloque B

Paralelo 2

G1	G2	G3	G4
Nicolas Mardones	Víctor Meza Herrera	Daniela Méndez Gándara	Claudia Blanco
Manuel Denis	Estefania Manriquez	Álvaro Pérez	Ariel Inostroza
Bryan Castillo	Patricio Vera	Pedro Nahum	Héctor Aguayo
GERALDY SUAREZ	Oscar Torres	Javier Gajardo	Ruben Sanhueza Ramirez
Scarlett Espinoza	Braulio Quiroz	Angela Proboste Neira	Félix González
Ulises Campodónico	Yerko Gallardo	Nicolás Guzmán	Ariel Mora
Carol Leiva	Rodrigo Araya		

G5	G6	G7	G8
Fabian Díaz	Camila Oyarzún	Mayerlyn Rodriguez	Daniela Porto
Natalia Rivera	Stefanya Pulgar	Sebastian Vega	Cristian Chavez Jara
Juan Salinas	Carlos Emilio Azócar Riquelme	Efrain Duarte Campos	Juan Rodrigo Vega
Rodrigo Pastén Cortés	Nicolas Rojas	Bianel Bianchini	Rodolfo Cantillana
Flavio Jara R.	luis.paillan.cnc@gmail.com	Bastián Gamboa Labbé	Abraham Ruiz
Daniel García	Cristóbal Gajardo	Pablo Uribe	Rodrigo Álvarez

Trabajo grupal – Ejercicio #1

Endpoint GET para los datos de un paciente.

1. Con el código ya revisado, cree un nuevo endpoint de tipo GET para responder con un HTML con los datos de un paciente.

Trabajo grupal – Ejercicio #2

Endpoint GET para los datos de un paciente.

1. Modifique el endpoint anterior para que responda con un enlace hacia la representación en XML.

2. ¡Usted debe probar que su enlace responda con un XML!

Referencias

***1. Tesis doctoral de Roy Fielding*:**

URL: https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

2. Libro:

"API Design Patterns", JJ Geewax.

Manning, 2021.

3. Documentación official FastAPI:

URL: https://fastapi.tiangolo.com/reference/

¿Preguntas?

¡Hemos llegado al final de la clase!

TALENTOFUTURO