Parámetros de path

Puedes declarar los "parámetros" o "variables" con la misma sintaxis que usan los format strings de Python:

```
{!../../../docs_src/path_params/tutorial001.py!}
```

El valor del parámetro de path `item_id` será pasado a tu función como el argumento `item_id`.

Entonces, si corres este ejemplo y vas a http://127.0.0.1:8000/items/foo, verás una respuesta de:

```
{"item_id":"foo"}
```

Parámetros de path con tipos

Puedes declarar el tipo de un parámetro de path en la función usando las anotaciones de tipos estándar de Python:

```
{!../../../docs_src/path_params/tutorial002.py!}
```

En este caso, `item_id` es declarado como un `int`.

!!! check "Revisa" Esto te dará soporte en el editor dentro de tu función, con chequeos de errores, autocompletado, etc.

Conversión de datos

Si corres este ejemplo y abres tu navegador en http://127.0.0.1:8000/items/3 verás una respuesta de:

```
{"item_id":3}
```

!!! check "Revisa" Observa que el valor que recibió (y devolvió) tu función es `3`, como un Python `int`, y no un string `"3"`.

```
Entonces, con esa declaración de tipos **FastAPI** te da <abbr title="convertir el string que viene de un HTTP request a datos de Python">"parsing"</abbr> automático del request.
```

Validación de datos

Pero si abres tu navegador en http://127.0.0.1:8000/items/foo verás este lindo error de HTTP:

```
{
    "detail": [
        {
            "loc": [
                "path",
                "item_id"
            ],
            "msg": "value is not a valid integer",
```

```
            "type": "type_error.integer"
        }
    ]
}
```

debido a que el parámetro de path `item_id` tenía el valor `"foo"`, que no es un `int`.

El mismo error aparecería si pasaras un `float` en vez de un `int` como en: http://127.0.0.1:8000/items/4.2

!!! check "Revisa" Así, con la misma declaración de tipo de Python, **FastAPI** te da validación de datos.

```
Observa que el error también muestra claramente el punto exacto en el que no pasó la validación.

Esto es increíblemente útil cuando estás desarrollando y debugging código que interactúa con tu API.
```

Documentación

Cuando abras tu navegador en http://127.0.0.1:8000/docs verás la documentación automática e interactiva del API como:

!!! check "Revisa" Nuevamente, con la misma declaración de tipo de Python, **FastAPI** te da documentación automática e interactiva (integrándose con Swagger UI)

```
Observa que el parámetro de path está declarado como un integer.
```

Beneficios basados en estándares, documentación alternativa

Debido a que el schema generado es del estándar OpenAPI hay muchas herramientas compatibles.

Es por esto que **FastAPI** mismo provee una documentación alternativa de la API (usando ReDoc), a la que puedes acceder en http://127.0.0.1:8000/redoc:

De la misma manera hay muchas herramientas compatibles. Incluyendo herramientas de generación de código para muchos lenguajes.

Pydantic

Toda la validación de datos es realizada tras bastidores por Pydantic, así que obtienes todos sus beneficios. Así sabes que estás en buenas manos.

Puedes usar las mismas declaraciones de tipos con `str`, `float`, `bool` y otros tipos de datos más complejos.

Exploraremos varios de estos tipos en los próximos capítulos del tutorial.

El orden importa

Cuando creas *operaciones de path* puedes encontrarte con situaciones en las que tengas un path fijo.

Digamos algo como `/users/me` que sea para obtener datos del usuario actual.

... y luego puedes tener el path `/users/{user_id}` para obtener los datos sobre un usuario específico asociados a un ID de usuario.

Porque las *operaciones de path* son evaluadas en orden, tienes que asegurarte de que el path para `/users/me` sea declarado antes que el path para `/users/{user_id}` :

```
{!../../../docs_src/path_params/tutorial003.py!}
```

De otra manera el path para `/users/{user_id}` coincidiría también con `/users/me` "pensando" que está recibiendo el parámetro `user_id` con el valor `"me"` .

Valores predefinidos

Si tienes una *operación de path* que recibe un *parámetro de path* pero quieres que los valores posibles del *parámetro de path* sean predefinidos puedes usar un `Enum` estándar de Python.

Crea una clase `Enum`

Importa `Enum` y crea una sub-clase que herede desde `str` y desde `Enum` .

Al heredar desde `str` la documentación de la API podrá saber que los valores deben ser de tipo `string` y podrá mostrarlos correctamente.

Luego crea atributos de clase con valores fijos, que serán los valores disponibles válidos:

```
{!../../../docs_src/path_params/tutorial005.py!}
```

!!! info "Información" Las Enumerations (o enums) están disponibles en Python desde la versión 3.4.

!!! tip "Consejo" Si lo estás dudando, "AlexNet", "ResNet", y "LeNet" son solo nombres de modelos de Machine Learning.

Declara un *parámetro de path*

Luego, crea un *parámetro de path* con anotaciones de tipos usando la clase enum que creaste (`ModelName`):

```
{!../../../docs_src/path_params/tutorial005.py!}
```

Revisa la documentación

Debido a que los valores disponibles para el *parámetro de path* están predefinidos, la documentación interactiva los puede mostrar bien:

Trabajando con los *enumerations* de Python

El valor del *parámetro de path* será un *enumeration member*.

Compara *enumeration members*

Puedes compararlo con el *enumeration member* en el enum (`ModelName`) que creaste:

```
{!../../../docs_src/path_params/tutorial005.py!}
```

Obtén el *enumeration value*

Puedes obtener el valor exacto (un `str` en este caso) usando `model_name.value` , o en general, `your_enum_member.value` :

```
{!../../../docs_src/path_params/tutorial005.py!}
```

!!! tip "Consejo" También podrías obtener el valor `"lenet"` con `ModelName.lenet.value` .

Devuelve *enumeration members*

Puedes devolver *enum members* desde tu *operación de path* inclusive en un body de JSON anidado (por ejemplo, un `dict`).

Ellos serán convertidos a sus valores correspondientes (strings en este caso) antes de devolverlos al cliente:

```
{!../../../docs_src/path_params/tutorial005.py!}
```

En tu cliente obtendrás una respuesta en JSON como:

```
{
  "model_name": "alexnet",
  "message": "Deep Learning FTW!"
}
```

Parámetros de path parameters que contienen paths

Digamos que tienes una *operación de path* con un path `/files/{file_path}` .

Pero necesitas que el mismo `file_path` contenga un path como `home/johndoe/myfile.txt` .

Entonces, la URL para ese archivo sería algo como: `/files/home/johndoe/myfile.txt` .

Soporte de OpenAPI

OpenAPI no soporta una manera de declarar un *parámetro de path* que contenga un path, dado que esto podría llevar a escenarios que son difíciles de probar y definir.

Sin embargo, lo puedes hacer en **FastAPI** usando una de las herramientas internas de Starlette.

La documentación seguirá funcionando, aunque no añadirá ninguna información diciendo que el parámetro debería contener un path.

Convertidor de path

Usando una opción directamente desde Starlette puedes declarar un *parámetro de path* que contenga un path usando una URL como:

```
/files/{file_path:path}
```

En este caso el nombre del parámetro es `file_path` y la última parte, `:path`, le dice que el parámetro debería coincidir con cualquier path.

Entonces lo puedes usar con:

```
{!../../../docs_src/path_params/tutorial004.py!}
```

!!! tip "Consejo" Podrías necesitar que el parámetro contenga `/home/johndoe/myfile.txt` con un slash inicial (`/`).

```
En este caso la URL sería `/files//home/johndoe/myfile.txt` con un slash doble (`//`) entre `files` y `home`.
```

Repaso

Con **FastAPI**, usando declaraciones de tipo de Python intuitivas y estándares, obtienes:

- Soporte en el editor: chequeos de errores, auto-completado, etc.
- "Parsing" de datos
- Validación de datos
- Anotación de la API y documentación automática

Solo tienes que declararlos una vez.

Esa es probablemente la principal ventaja visible de **FastAPI** sobre otros frameworks alternativos (aparte del rendimiento puro).