Path Parameters

You can declare path "parameters" or "variables" with the same syntax used by Python format strings:

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

The value of the path parameter `item_id` will be passed to your function as the argument `item_id`.

So, if you run this example and go to http://127.0.0.1:8000/items/foo, you will see a response of:

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

Path parameters with types

You can declare the type of a path parameter in the function, using standard Python type annotations:

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

In this case, `item_id` is declared to be an `int`.

!!! check This will give you editor support inside of your function, with error checks, completion, etc.

Data conversion

If you run this example and open your browser at http://127.0.0.1:8000/items/3, you will see a response of:

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

!!! check Notice that the value your function received (and returned) is `3`, as a Python `int`, not a string `"3"`.

```
So, with that type declaration, **FastAPI** gives you automatic request <abbr
title="converting the string that comes from an HTTP request into Python
data">"parsing"</abbr>.
```

Data validation

But if you go to the browser at http://127.0.0.1:8000/items/foo, you will see a nice HTTP error of:

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

```
    ]
}
```

because the path parameter `item_id` had a value of `"foo"`, which is not an `int`.

The same error would appear if you provided a `float` instead of an `int`, as in: http://127.0.0.1:8000/items/4.2

!!! check So, with the same Python type declaration, **FastAPI** gives you data validation.

```
Notice that the error also clearly states exactly the point where the validation
didn't pass.

This is incredibly helpful while developing and debugging code that interacts with
your API.
```

Documentation

And when you open your browser at http://127.0.0.1:8000/docs, you will see an automatic, interactive, API documentation like:

!!! check Again, just with that same Python type declaration, **FastAPI** gives you automatic, interactive documentation (integrating Swagger UI).

```
Notice that the path parameter is declared to be an integer.
```

Standards-based benefits, alternative documentation

And because the generated schema is from the OpenAPI standard, there are many compatible tools.

Because of this, **FastAPI** itself provides an alternative API documentation (using ReDoc), which you can access at http://127.0.0.1:8000/redoc:

The same way, there are many compatible tools. Including code generation tools for many languages.

Pydantic

All the data validation is performed under the hood by Pydantic, so you get all the benefits from it. And you know you are in good hands.

You can use the same type declarations with `str`, `float`, `bool` and many other complex data types.

Several of these are explored in the next chapters of the tutorial.

Order matters

When creating *path operations*, you can find situations where you have a fixed path.

Like `/users/me`, let's say that it's to get data about the current user.

And then you can also have a path `/users/{user_id}` to get data about a specific user by some user ID.

Because *path operations* are evaluated in order, you need to make sure that the path for `/users/me` is declared before the one for `/users/{user_id}`:

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

Otherwise, the path for `/users/{user_id}` would match also for `/users/me`, "thinking" that it's receiving a parameter `user_id` with a value of `"me"`.

Predefined values

If you have a *path operation* that receives a *path parameter*, but you want the possible valid *path parameter* values to be predefined, you can use a standard Python `Enum`.

Create an `Enum` class

Import `Enum` and create a sub-class that inherits from `str` and from `Enum`.

By inheriting from `str` the API docs will be able to know that the values must be of type `string` and will be able to render correctly.

Then create class attributes with fixed values, which will be the available valid values:

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

!!! info Enumerations (or enums) are available in Python since version 3.4.

!!! tip If you are wondering, "AlexNet", "ResNet", and "LeNet" are just names of Machine Learning models.

Declare a *path parameter*

Then create a *path parameter* with a type annotation using the enum class you created (`ModelName`):

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

Check the docs

Because the available values for the *path parameter* are predefined, the interactive docs can show them nicely:

Working with Python *enumerations*

The value of the *path parameter* will be an *enumeration member*.

Compare *enumeration members*

You can compare it with the *enumeration member* in your created enum `ModelName`:

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

Get the *enumeration value*

You can get the actual value (a `str` in this case) using `model_name.value`, or in general, `your_enum_member.value`:

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

!!! tip You could also access the value `"lenet"` with `ModelName.lenet.value`.

Return *enumeration members*

You can return *enum members* from your *path operation*, even nested in a JSON body (e.g. a `dict`).

They will be converted to their corresponding values (strings in this case) before returning them to the client:

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

In your client you will get a JSON response like:

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

Path parameters containing paths

Let's say you have a *path operation* with a path `/files/{file_path}`.

But you need `file_path` itself to contain a *path*, like `home/johndoe/myfile.txt`.

So, the URL for that file would be something like: `/files/home/johndoe/myfile.txt`.

OpenAPI support

OpenAPI doesn't support a way to declare a *path parameter* to contain a *path* inside, as that could lead to scenarios that are difficult to test and define.

Nevertheless, you can still do it in **FastAPI**, using one of the internal tools from Starlette.

And the docs would still work, although not adding any documentation telling that the parameter should contain a path.

Path convertor

Using an option directly from Starlette you can declare a *path parameter* containing a *path* using a URL like:

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

In this case, the name of the parameter is `file_path`, and the last part, `:path`, tells it that the parameter should match any *path*.

So, you can use it with:

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

!!! tip You could need the parameter to contain `/home/johndoe/myfile.txt`, with a leading slash (`/`).

```
In that case, the URL would be: `/files//home/johndoe/myfile.txt`, with a double slash (`//`) between `files` and `home`.
```

Recap

With **FastAPI**, by using short, intuitive and standard Python type declarations, you get:

- Editor support: error checks, autocompletion, etc.
- Data "parsing"
- Data validation
- API annotation and automatic documentation

And you only have to declare them once.

That's probably the main visible advantage of **FastAPI** compared to alternative frameworks (apart from the raw performance).