Path Parameters and Numeric Validations

The same way you can declare more validations and metadata for query parameters with `Query`, you can declare the same type of validations and metadata for path parameters with `Path`.

Import Path

First, import `Path` from `fastapi`:

Python 3.6 and above

```python
from typing import Union

from fastapi import FastAPI, Path, Query

app = FastAPI()


@app.get("/items/{item_id}")
async def read_items(
    item_id: int = Path(title="The ID of the item to get"),
    q: Union[str, None] = Query(default=None, alias="item-query"),
):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results
```

Python 3.10 and above

```python
from fastapi import FastAPI, Path, Query

app = FastAPI()


@app.get("/items/{item_id}")
async def read_items(
    item_id: int = Path(title="The ID of the item to get"),
    q: str | None = Query(default=None, alias="item-query"),
):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results
```

Declare metadata

You can declare all the same parameters as for `Query`.

For example, to declare a `title` metadata value for the path parameter `item_id` you can type:

Python 3.6 and above

```python
from typing import Union

from fastapi import FastAPI, Path, Query

app = FastAPI()


@app.get("/items/{item_id}")
async def read_items(
    item_id: int = Path(title="The ID of the item to get"),
    q: Union[str, None] = Query(default=None, alias="item-query"),
):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results
```

Python 3.10 and above

```python
from fastapi import FastAPI, Path, Query

app = FastAPI()


@app.get("/items/{item_id}")
async def read_items(
    item_id: int = Path(title="The ID of the item to get"),
    q: str | None = Query(default=None, alias="item-query"),
):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results
```

Note

A path parameter is always required as it has to be part of the path.

So, you should declare it with `...` to mark it as required.

Nevertheless, even if you declared it with `None` or set a default value, it would not affect anything, it would still be always required.

Order the parameters as you need

Let's say that you want to declare the query parameter `q` as a required `str`.

And you don't need to declare anything else for that parameter, so you don't really need to use `Query`.

But you still need to use `Path` for the `item_id` path parameter.

Python will complain if you put a value with a "default" before a value that doesn't have a "default".

But you can re-order them, and have the value without a default (the query parameter `q`) first.

It doesn't matter for **FastAPI**. It will detect the parameters by their names, types and default declarations (`Query`, `Path`, etc), it doesn't care about the order.

So, you can declare your function as:

```
from fastapi import FastAPI, Path

app = FastAPI()


@app.get("/items/{item_id}")
async def read_items(q: str, item_id: int = Path(title="The ID of the item to get")):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results
```

Order the parameters as you need, tricks

If you want to declare the `q` query parameter without a `Query` nor any default value, and the path parameter `item_id` using `Path`, and have them in a different order, Python has a little special syntax for that.

Pass `*`, as the first parameter of the function.

Python won't do anything with that `*`, but it will know that all the following parameters should be called as keyword arguments (key-value pairs), also known as `kwargs`. Even if they don't have a default value.

```
from fastapi import FastAPI, Path

app = FastAPI()
```

```python
@app.get("/items/{item_id}")
async def read_items(*, item_id: int = Path(title="The ID of the item to
get"), q: str):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results
```

Number validations: greater than or equal

With `Query` and `Path` (and other's you'll see later) you can declare string constraints, but also number constraints.

Here, with `ge=1`, `item_id` will need to be an integer number "`g`reater than or `e`qual" to `1`.

```python
from fastapi import FastAPI, Path

app = FastAPI()


@app.get("/items/{item_id}")
async def read_items(
    *, item_id: int = Path(title="The ID of the item to get", ge=1), q: str
):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results
```

Number validations: greater than and less than or equal

The same applies for:

- `gt`: `g`reater `t`han
- `le`: `l`ess than or `e`qual

```python
from fastapi import FastAPI, Path

app = FastAPI()


@app.get("/items/{item_id}")
async def read_items(
    *,
    item_id: int = Path(title="The ID of the item to get", gt=0, le=1000),
    q: str,
):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results
```

Number validations: floats, greater than and less than

Number validations also work for `float` values.

Here's where it becomes important to be able to declare `gt` and not just `ge`. As with it you can require, for example, that a value must be greater than `0`, even if it is less than `1`.

So, `0.5` would be a valid value. But `0.0` or `0` would not.

And the same for `lt`.

```
from fastapi import FastAPI, Path, Query

app = FastAPI()


@app.get("/items/{item_id}")
async def read_items(
    *,
    item_id: int = Path(title="The ID of the item to get", ge=0, le=1000),
    q: str,
    size: float = Query(gt=0, lt=10.5)
):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results
```

Recap

With `Query`, `Path` (and others you haven't seen yet) you can declare metadata and string validations in the same ways as with Query Parameters and String Validations ↪.

And you can also declare numeric validations:

- `gt`: `g` reater `t` han
- `ge`: `g` reater than or `e` qual
- `lt`: `l` ess `t` han
- `le`: `l` ess than or `e` qual

Info

`Query`, `Path`, and others you will see later are subclasses of a common `Param` class (that you don't need to use).

And all of them share the same all these same parameters of additional validation and metadata you have seen.

Technical Details

When you import `Query`, `Path` and others from `fastapi`, they are actually functions.

That when called, return instances of classes of the same name.

So, you import `Query`, which is a function. And when you call it, it returns an instance of a class also named `Query`.

These functions are there (instead of just using the classes directly) so that your editor doesn't mark errors about their types.

That way you can use your normal editor and coding tools without having to add custom configurations to disregard those errors.