Body - Nested Models

With **FastAPI**, you can define, validate, document, and use arbitrarily deeply nested models (thanks to Pydantic).

List fields

You can define an attribute to be a subtype. For example, a Python `list`:

=== "Python 3.6 and above"

```
```Python hl_lines="14"
{!> ../../../docs_src/body_nested_models/tutorial001.py!}
```
```

=== "Python 3.10 and above"

```
```Python hl_lines="12"
{!> ../../../docs_src/body_nested_models/tutorial001_py310.py!}
```
```

This will make `tags` be a list of items. Although it doesn't declare the type of each of the items.

List fields with type parameter

But Python has a specific way to declare lists with internal types, or "type parameters":

Import typing's `List`

In Python 3.9 and above you can use the standard `list` to declare these type annotations as we'll see below. 💡

But in Python versions before 3.9 (3.6 and above), you first need to import `List` from standard Python's `typing` module:

```
{!> ../../../docs_src/body_nested_models/tutorial002.py!}
```

Declare a `list` with a type parameter

To declare types that have type parameters (internal types), like `list`, `dict`, `tuple`:

- If you are in a Python version lower than 3.9, import their equivalent version from the `typing` module
- Pass the internal type(s) as "type parameters" using square brackets: `[` and `]`

In Python 3.9 it would be:

```
my_list: list[str]
```

In versions of Python before 3.9, it would be:

```
from typing import List

my_list: List[str]
```

That's all standard Python syntax for type declarations.

Use that same standard syntax for model attributes with internal types.

So, in our example, we can make `tags` be specifically a "list of strings":

=== "Python 3.6 and above"

```
```Python hl_lines="14"
{!> ../../../docs_src/body_nested_models/tutorial002.py!}
```
```

=== "Python 3.9 and above"

```
```Python hl_lines="14"
{!> ../../../docs_src/body_nested_models/tutorial002_py39.py!}
```
```

=== "Python 3.10 and above"

```
```Python hl_lines="12"
{!> ../../../docs_src/body_nested_models/tutorial002_py310.py!}
```
```

Set types

But then we think about it, and realize that tags shouldn't repeat, they would probably be unique strings.

And Python has a special data type for sets of unique items, the `set`.

Then we can declare `tags` as a set of strings:

=== "Python 3.6 and above"

```
```Python hl_lines="1  14"
{!> ../../../docs_src/body_nested_models/tutorial003.py!}
```
```

=== "Python 3.9 and above"

```
```Python hl_lines="14"
{!> ../../../docs_src/body_nested_models/tutorial003_py39.py!}
```
```

=== "Python 3.10 and above"

```
```Python hl_lines="12"
{!> ../../../docs_src/body_nested_models/tutorial003_py310.py!}
```
```

With this, even if you receive a request with duplicate data, it will be converted to a set of unique items.

And whenever you output that data, even if the source had duplicates, it will be output as a set of unique items.

And it will be annotated / documented accordingly too.

Nested Models

Each attribute of a Pydantic model has a type.

But that type can itself be another Pydantic model.

So, you can declare deeply nested JSON "objects" with specific attribute names, types and validations.

All that, arbitrarily nested.

Define a submodel

For example, we can define an `Image` model:

=== "Python 3.6 and above"

```
```Python hl_lines="9-11"
{!> ../../../docs_src/body_nested_models/tutorial004.py!}
```
```

=== "Python 3.9 and above"

```
```Python hl_lines="9-11"
{!> ../../../docs_src/body_nested_models/tutorial004_py39.py!}
```
```

=== "Python 3.10 and above"

```
```Python hl_lines="7-9"
{!> ../../../docs_src/body_nested_models/tutorial004_py310.py!}
```
```

Use the submodel as a type

And then we can use it as the type of an attribute:

=== "Python 3.6 and above"

```
```Python hl_lines="20"
{!> ../../../docs_src/body_nested_models/tutorial004.py!}
```
```

=== "Python 3.9 and above"

```
```Python hl_lines="20"
{!> ../../../docs_src/body_nested_models/tutorial004_py39.py!}
```
```

=== "Python 3.10 and above"

```
```Python hl_lines="18"
{!> ../../../docs_src/body_nested_models/tutorial004_py310.py!}
```
```

```
```

This would mean that **FastAPI** would expect a body similar to:

```
{
    "name": "Foo",
    "description": "The pretender",
    "price": 42.0,
    "tax": 3.2,
    "tags": ["rock", "metal", "bar"],
    "image": {
        "url": "http://example.com/baz.jpg",
        "name": "The Foo live"
    }
}
```

Again, doing just that declaration, with **FastAPI** you get:

- Editor support (completion, etc), even for nested models
- Data conversion
- Data validation
- Automatic documentation

Special types and validation

Apart from normal singular types like `str`, `int`, `float`, etc. You can use more complex singular types that inherit from `str`.

To see all the options you have, checkout the docs for Pydantic's exotic types. You will see some examples in the next chapter.

For example, as in the `Image` model we have a `url` field, we can declare it to be instead of a `str`, a Pydantic's `HttpUrl`:

=== "Python 3.6 and above"

```
```Python hl_lines="4  10"
{!> ../../../docs_src/body_nested_models/tutorial005.py!}
```
```

=== "Python 3.9 and above"

```
```Python hl_lines="4  10"
{!> ../../../docs_src/body_nested_models/tutorial005_py39.py!}
```
```

=== "Python 3.10 and above"

```
```Python hl_lines="2  8"
{!> ../../../docs_src/body_nested_models/tutorial005_py310.py!}
```
```

The string will be checked to be a valid URL, and documented in JSON Schema / OpenAPI as such.

Attributes with lists of submodels

You can also use Pydantic models as subtypes of `list`, `set`, etc:

=== "Python 3.6 and above"

```
```Python hl_lines="20"
{!> ../../../docs_src/body_nested_models/tutorial006.py!}
```
```

=== "Python 3.9 and above"

```
```Python hl_lines="20"
{!> ../../../docs_src/body_nested_models/tutorial006_py39.py!}
```
```

=== "Python 3.10 and above"

```
```Python hl_lines="18"
{!> ../../../docs_src/body_nested_models/tutorial006_py310.py!}
```
```

This will expect (convert, validate, document, etc) a JSON body like:

```
{
    "name": "Foo",
    "description": "The pretender",
    "price": 42.0,
    "tax": 3.2,
    "tags": [
        "rock",
        "metal",
        "bar"
    ],
    "images": [
        {
            "url": "http://example.com/baz.jpg",
            "name": "The Foo live"
        },
        {
            "url": "http://example.com/dave.jpg",
            "name": "The Baz"
        }
    ]
}
```

!!! info Notice how the `images` key now has a list of image objects.

Deeply nested models

You can define arbitrarily deeply nested models:

=== "Python 3.6 and above"

```Python hl_lines="9  14  20  23  27"
{!> ../../../docs_src/body_nested_models/tutorial007.py!}
```

=== "Python 3.9 and above"

```Python hl_lines="9  14  20  23  27"
{!> ../../../docs_src/body_nested_models/tutorial007_py39.py!}
```

=== "Python 3.10 and above"

```Python hl_lines="7  12  18  21  25"
{!> ../../../docs_src/body_nested_models/tutorial007_py310.py!}
```

!!! info Notice how `Offer` has a list of `Item` s, which in turn have an optional list of `Image` s

Bodies of pure lists

If the top level value of the JSON body you expect is a JSON `array` (a Python `list`), you can declare the type in the parameter of the function, the same as in Pydantic models:

```
images: List[Image]
```

or in Python 3.9 and above:

```
images: list[Image]
```

as in:

=== "Python 3.6 and above"

```Python hl_lines="15"
{!> ../../../docs_src/body_nested_models/tutorial008.py!}
```

=== "Python 3.9 and above"

```Python hl_lines="13"
{!> ../../../docs_src/body_nested_models/tutorial008_py39.py!}
```

Editor support everywhere

And you get editor support everywhere.

Even for items inside of lists:

You couldn't get this kind of editor support if you were working directly with `dict` instead of Pydantic models.

But you don't have to worry about them either, incoming dicts are converted automatically and your output is converted automatically to JSON too.

Bodies of arbitrary `dict`s

You can also declare a body as a `dict` with keys of some type and values of other type.

Without having to know beforehand what are the valid field/attribute names (as would be the case with Pydantic models).

This would be useful if you want to receive keys that you don't already know.

Other useful case is when you want to have keys of other type, e.g. `int`.

That's what we are going to see here.

In this case, you would accept any `dict` as long as it has `int` keys with `float` values:

=== "Python 3.6 and above"

```
```Python hl_lines="9"
{!> ../../../docs_src/body_nested_models/tutorial009.py!}
```
```

=== "Python 3.9 and above"

```
```Python hl_lines="7"
{!> ../../../docs_src/body_nested_models/tutorial009_py39.py!}
```
```

!!! tip Have in mind that JSON only supports `str` as keys.

```
But Pydantic has automatic data conversion.

This means that, even though your API clients can only send strings as keys, as long
as those strings contain pure integers, Pydantic will convert them and validate them.

And the `dict` you receive as `weights` will actually have `int` keys and `float`
values.
```

Recap

With **FastAPI** you have the maximum flexibility provided by Pydantic models, while keeping your code simple, short and elegant.

But with all the benefits:

- Editor support (completion everywhere!)
- Data conversion (a.k.a. parsing / serialization)

- Data validation
- Schema documentation
- Automatic docs