Response Model - Return Type

You can declare the type used for the response by annotating the *path operation function* **return type**.

You can use **type annotations** the same way you would for input data in function **parameters**, you can use Pydantic models, lists, dictionaries, scalar values like integers, booleans, etc.

Python 3.10+

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None
    tags: list[str] = []


@app.post("/items/")
async def create_item(item: Item) -> Item:
    return item


@app.get("/items/")
async def read_items() -> list[Item]:
    return [
        Item(name="Portal Gun", price=42.0),
        Item(name="Plumbus", price=32.0),
    ]
```

▸ 🤓 Other versions and variants

Python 3.9+

```
from typing import Union

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: Union[str, None] = None
    price: float
    tax: Union[float, None] = None
    tags: list[str] = []


@app.post("/items/")
async def create_item(item: Item) -> Item:
    return item


@app.get("/items/")
async def read_items() -> list[Item]:
    return [
        Item(name="Portal Gun", price=42.0),
        Item(name="Plumbus", price=32.0),
    ]
```

Python 3.8+

```
from typing import List, Union

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: Union[str, None] = None
    price: float
    tax: Union[float, None] = None
    tags: List[str] = []


@app.post("/items/")
async def create_item(item: Item) -> Item:
    return item


@app.get("/items/")
async def read_items() -> List[Item]:
```

```
    return [
        Item(name="Portal Gun", price=42.0),
        Item(name="Plumbus", price=32.0),
    ]
```

FastAPI will use this return type to:

- **Validate** the returned data.
  - If the data is invalid (e.g. you are missing a field), it means that *your* app code is broken, not returning what it should, and it will return a server error instead of returning incorrect data. This way you and your clients can be certain that they will receive the data and the data shape expected.
- Add a **JSON Schema** for the response, in the OpenAPI *path operation*.
  - This will be used by the **automatic docs**.
  - It will also be used by automatic client code generation tools.

But most importantly:

- It will **limit and filter** the output data to what is defined in the return type.
  - This is particularly important for **security**, we'll see more of that below.

`response_model` Parameter

There are some cases where you need or want to return some data that is not exactly what the type declares.

For example, you could want to **return a dictionary** or a database object, but **declare it as a Pydantic model**. This way the Pydantic model would do all the data documentation, validation, etc. for the object that you returned (e.g. a dictionary or database object).

If you added the return type annotation, tools and editors would complain with a (correct) error telling you that your function is returning a type (e.g. a dict) that is different from what you declared (e.g. a Pydantic model).

In those cases, you can use the *path operation decorator* parameter `response_model` instead of the return type.

You can use the `response_model` parameter in any of the *path operations*:

- `@app.get()`
- `@app.post()`
- `@app.put()`
- `@app.delete()`
- etc.

Python 3.10+

```
from typing import Any

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None
    tags: list[str] = []


@app.post("/items/", response_model=Item)
async def create_item(item: Item) -> Any:
    return item


@app.get("/items/", response_model=list[Item])
async def read_items() -> Any:
    return [
        {"name": "Portal Gun", "price": 42.0},
        {"name": "Plumbus", "price": 32.0},
    ]
```

▸ 🤓 Other versions and variants

Python 3.9+

```
from typing import Any, Union

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
```

```
    name: str
    description: Union[str, None] = None
    price: float
    tax: Union[float, None] = None
    tags: list[str] = []


@app.post("/items/", response_model=Item)
async def create_item(item: Item) -> Any:
    return item


@app.get("/items/", response_model=list[Item])
async def read_items() -> Any:
    return [
        {"name": "Portal Gun", "price": 42.0},
        {"name": "Plumbus", "price": 32.0},
    ]
```

Python 3.8+

```
from typing import Any, List, Union

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: Union[str, None] = None
    price: float
    tax: Union[float, None] = None
    tags: List[str] = []


@app.post("/items/", response_model=Item)
async def create_item(item: Item) -> Any:
    return item


@app.get("/items/", response_model=List[Item])
async def read_items() -> Any:
    return [
        {"name": "Portal Gun", "price": 42.0},
        {"name": "Plumbus", "price": 32.0},
    ]
```

Note

Notice that `response_model` is a parameter of the "decorator" method (`get` , `post` , etc). Not of your *path operation function*, like all the parameters and body.

`response_model` receives the same type you would declare for a Pydantic model field, so, it can be a Pydantic model, but it can also be, e.g. a `list` of Pydantic models, like `List[Item]` .

FastAPI will use this `response_model` to do all the data documentation, validation, etc. and also to **convert and filter the output data** to its type declaration.

Tip

If you have strict type checks in your editor, mypy, etc, you can declare the function return type as `Any` .

That way you tell the editor that you are intentionally returning anything. But FastAPI will still do the data documentation, validation, filtering, etc. with the `response_model` .

`response_model` Priority

If you declare both a return type and a `response_model` , the `response_model` will take priority and be used by FastAPI.

This way you can add correct type annotations to your functions even when you are returning a type different than the response model, to be used by the editor and tools like mypy. And still you can have FastAPI do the data validation, documentation, etc. using the `response_model` .

You can also use `response_model=None` to disable creating a response model for that *path operation*, you might need to do it if you are adding type annotations for things that are not valid Pydantic fields, you will see an example of that in one of the sections below.

Return the same input data

Here we are declaring a `UserIn` model, it will contain a plaintext password:

Python 3.10+

```
from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()
```

```
class UserIn(BaseModel):
    username: str
    password: str
    email: EmailStr
    full_name: str | None = None


# Don't do this in production!
@app.post("/user/")
async def create_user(user: UserIn) -> UserIn:
    return user
```

▸ 🤓 Other versions and variants

Python 3.8+

```
from typing import Union

from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()


class UserIn(BaseModel):
    username: str
    password: str
    email: EmailStr
    full_name: Union[str, None] = None


# Don't do this in production!
@app.post("/user/")
async def create_user(user: UserIn) -> UserIn:
    return user
```

Info

To use `EmailStr`, first install `email-validator`.

Make sure you create a virtual environment, activate it, and then install it, for example:

```
$ pip install email-validator
```

or with:

```
$ pip install "pydantic[email]"
```

And we are using this model to declare our input and the same model to declare our output:

Python 3.10+

```
from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()


class UserIn(BaseModel):
    username: str
    password: str
    email: EmailStr
    full_name: str | None = None


# Don't do this in production!
@app.post("/user/")
async def create_user(user: UserIn) -> UserIn:
    return user
```

▸ 🤓 Other versions and variants

Python 3.8+

```
from typing import Union

from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()


class UserIn(BaseModel):
    username: str
    password: str
    email: EmailStr
```

```
    full_name: Union[str, None] = None


# Don't do this in production!
@app.post("/user/")
async def create_user(user: UserIn) -> UserIn:
    return user
```

Now, whenever a browser is creating a user with a password, the API will return the same password in the response.

In this case, it might not be a problem, because it's the same user sending the password.

But if we use the same model for another *path operation*, we could be sending our user's passwords to every client.

Danger

Never store the plain password of a user or send it in a response like this, unless you know all the caveats and you know what you are doing.

Add an output model

We can instead create an input model with the plaintext password and an output model without it:

Python 3.10+

```
from typing import Any

from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()


class UserIn(BaseModel):
    username: str
    password: str
    email: EmailStr
    full_name: str | None = None


class UserOut(BaseModel):
    username: str
    email: EmailStr
    full_name: str | None = None


@app.post("/user/", response_model=UserOut)
async def create_user(user: UserIn) -> Any:
    return user
```

▸ 🤓 Other versions and variants

Python 3.8+

```
from typing import Any, Union

from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()


class UserIn(BaseModel):
    username: str
    password: str
    email: EmailStr
    full_name: Union[str, None] = None


class UserOut(BaseModel):
    username: str
    email: EmailStr
    full_name: Union[str, None] = None


@app.post("/user/", response_model=UserOut)
async def create_user(user: UserIn) -> Any:
    return user
```

Here, even though our *path operation function* is returning the same input user that contains the password:

Python 3.10+

```
from typing import Any

from fastapi import FastAPI
from pydantic import BaseModel, EmailStr
```

```
app = FastAPI()


class UserIn(BaseModel):
    username: str
    password: str
    email: EmailStr
    full_name: str | None = None


class UserOut(BaseModel):
    username: str
    email: EmailStr
    full_name: str | None = None


@app.post("/user/", response_model=UserOut)
async def create_user(user: UserIn) -> Any:
    return user
```

▸ 🤓 Other versions and variants

Python 3.8+

```
from typing import Any, Union

from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()


class UserIn(BaseModel):
    username: str
    password: str
    email: EmailStr
    full_name: Union[str, None] = None


class UserOut(BaseModel):
    username: str
    email: EmailStr
    full_name: Union[str, None] = None


@app.post("/user/", response_model=UserOut)
async def create_user(user: UserIn) -> Any:
    return user
```

...we declared the `response_model` to be our model `UserOut`, that doesn't include the password:

Python 3.10+

```
from typing import Any

from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()


class UserIn(BaseModel):
    username: str
    password: str
    email: EmailStr
    full_name: str | None = None


class UserOut(BaseModel):
    username: str
    email: EmailStr
    full_name: str | None = None


@app.post("/user/", response_model=UserOut)
async def create_user(user: UserIn) -> Any:
    return user
```

▸ 🤓 Other versions and variants

Python 3.8+

```
from typing import Any, Union

from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()


class UserIn(BaseModel):
```

```
    username: str
    password: str
    email: EmailStr
    full_name: Union[str, None] = None


class UserOut(BaseModel):
    username: str
    email: EmailStr
    full_name: Union[str, None] = None


@app.post("/user/", response_model=UserOut)
async def create_user(user: UserIn) -> Any:
    return user
```

So, **FastAPI** will take care of filtering out all the data that is not declared in the output model (using Pydantic).

`response_model` **or Return Type**

In this case, because the two models are different, if we annotated the function return type as `UserOut`, the editor and tools would complain that we are returning an invalid type, as those are different classes.

That's why in this example we have to declare it in the `response_model` parameter.

...but continue reading below to see how to overcome that.

Return Type and Data Filtering

Let's continue from the previous example. We wanted to **annotate the function with one type**, but we wanted to be able to return from the function something that actually includes **more data**.

We want FastAPI to keep **filtering** the data using the response model. So that even though the function returns more data, the response will only include the fields declared in the response model.

In the previous example, because the classes were different, we had to use the `response_model` parameter. But that also means that we don't get the support from the editor and tools checking the function return type.

But in most of the cases where we need to do something like this, we want the model just to **filter/remove** some of the data as in this example.

And in those cases, we can use classes and inheritance to take advantage of function **type annotations** to get better support in the editor and tools, and still get the FastAPI **data filtering**.

Python 3.10+

```
from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()


class BaseUser(BaseModel):
    username: str
    email: EmailStr
    full_name: str | None = None


class UserIn(BaseUser):
    password: str


@app.post("/user/")
async def create_user(user: UserIn) -> BaseUser:
    return user
```

▸ Other versions and variants

Python 3.8+

```
from typing import Union

from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()


class BaseUser(BaseModel):
    username: str
    email: EmailStr
    full_name: Union[str, None] = None


class UserIn(BaseUser):
    password: str


@app.post("/user/")
async def create_user(user: UserIn) -> BaseUser:
    return user
```

With this, we get tooling support, from editors and mypy as this code is correct in terms of types, but we also get the data filtering from FastAPI.

How does this work? Let's check that out. 🤓

Type Annotations and Tooling

First let's see how editors, mypy and other tools would see this.

`BaseUser` has the base fields. Then `UserIn` inherits from `BaseUser` and adds the `password` field, so, it will include all the fields from both models.

We annotate the function return type as `BaseUser`, but we are actually returning a `UserIn` instance.

The editor, mypy, and other tools won't complain about this because, in typing terms, `UserIn` is a subclass of `BaseUser`, which means it's a *valid* type when what is expected is anything that is a `BaseUser`.

FastAPI Data Filtering

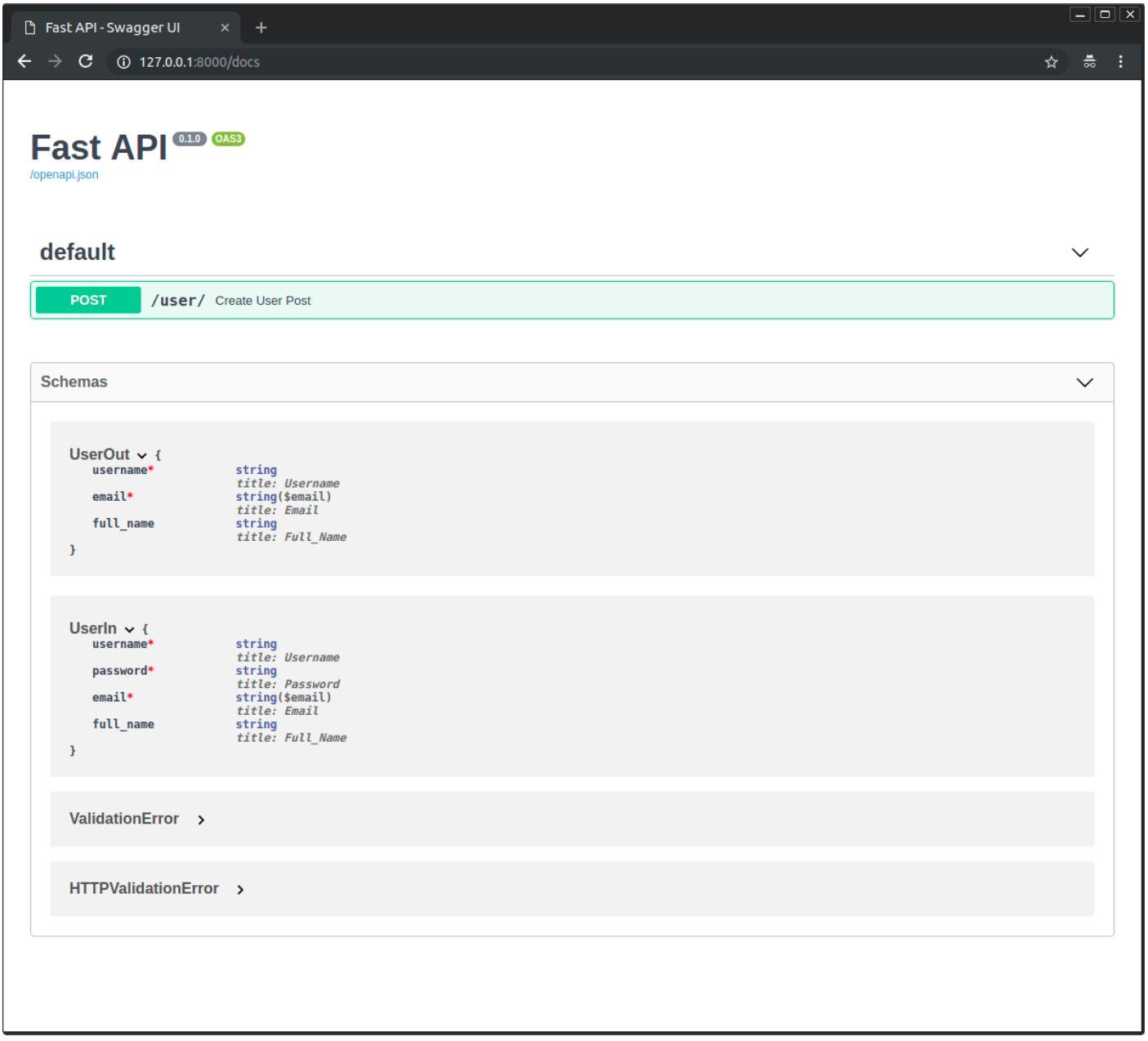
Now, for FastAPI, it will see the return type and make sure that what you return includes **only** the fields that are declared in the type.

FastAPI does several things internally with Pydantic to make sure that those same rules of class inheritance are not used for the returned data filtering, otherwise you could end up returning much more data than what you expected.

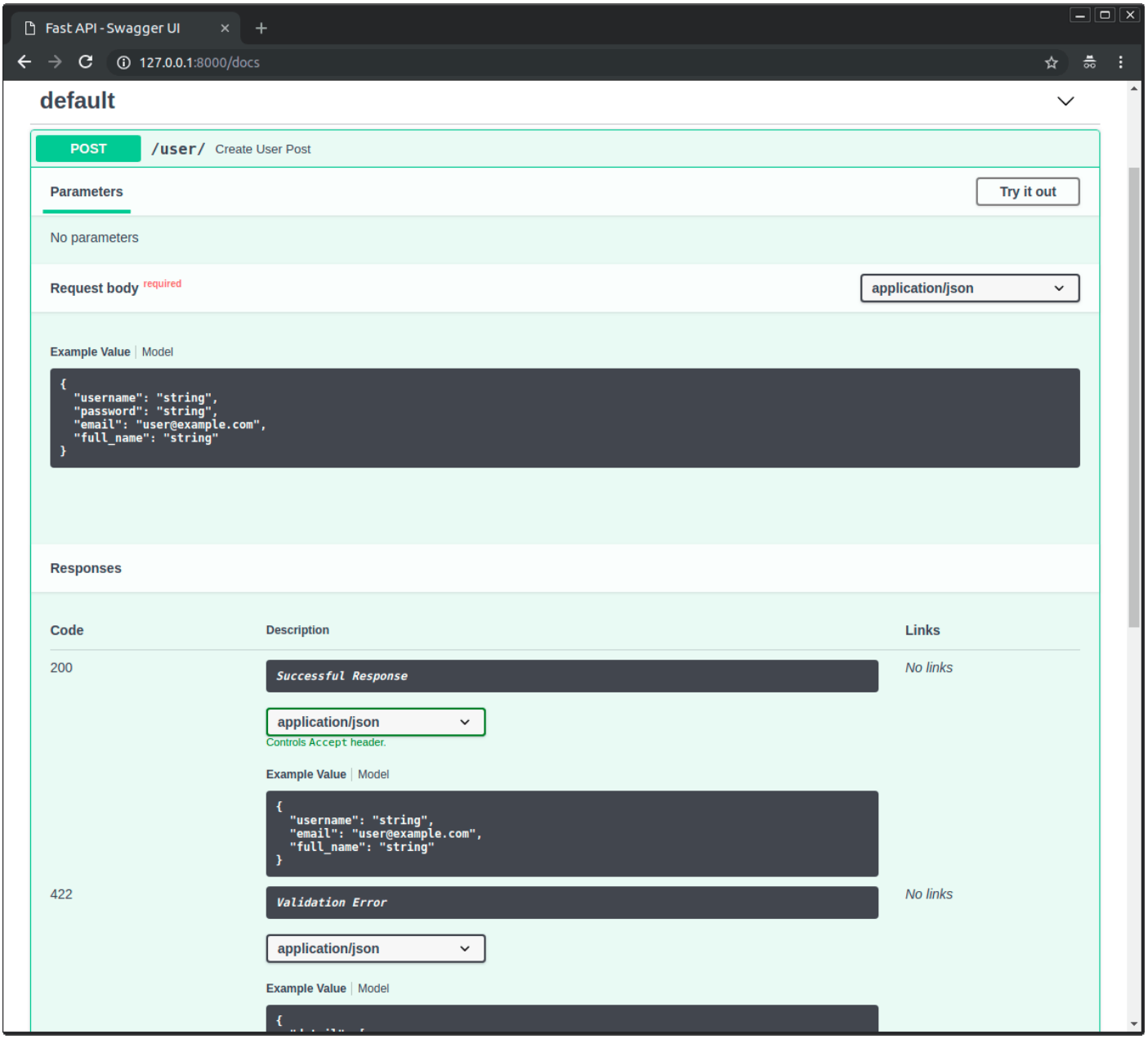
This way, you can get the best of both worlds: type annotations with **tooling support** and **data filtering**.

See it in the docs

When you see the automatic docs, you can check that the input model and output model will both have their own JSON Schema:



And both models will be used for the interactive API documentation:



Other Return Type Annotations

There might be cases where you return something that is not a valid Pydantic field and you annotate it in the function, only to get the support provided by tooling (the editor, mypy, etc).

Return a Response Directly

The most common case would be returning a Response directly as explained later in the advanced docs.

Python 3.8+

```
from fastapi import FastAPI, Response
from fastapi.responses import JSONResponse, RedirectResponse

app = FastAPI()


@app.get("/portal")
async def get_portal(teleport: bool = False) -> Response:
    if teleport:
        return RedirectResponse(url="https://www.youtube.com/watch?v=dQw4w9WgXcQ")
    return JSONResponse(content={"message": "Here's your interdimensional portal."})
```

This simple case is handled automatically by FastAPI because the return type annotation is the class (or a subclass of) `Response`.

And tools will also be happy because both `RedirectResponse` and `JSONResponse` are subclasses of `Response`, so the type annotation is correct.

Annotate a Response Subclass

You can also use a subclass of `Response` in the type annotation:

Python 3.8+

```
from fastapi import FastAPI
from fastapi.responses import RedirectResponse

app = FastAPI()


@app.get("/teleport")
async def get_teleport() -> RedirectResponse:
    return RedirectResponse(url="https://www.youtube.com/watch?v=dQw4w9WgXcQ")
```

This will also work because `RedirectResponse` is a subclass of `Response`, and FastAPI will automatically handle this simple case.

Invalid Return Type Annotations

But when you return some other arbitrary object that is not a valid Pydantic type (e.g. a database object) and you annotate it like that in the function, FastAPI will try to create a Pydantic response model from that type annotation, and will fail.

The same would happen if you had something like a union between different types where one or more of them are not valid Pydantic types, for example this would fail 💥:

Python 3.10+

```
from fastapi import FastAPI, Response
from fastapi.responses import RedirectResponse

app = FastAPI()


@app.get("/portal")
async def get_portal(teleport: bool = False) -> Response | dict:
    if teleport:
        return RedirectResponse(url="https://www.youtube.com/watch?v=dQw4w9WgXcQ")
    return {"message": "Here's your interdimensional portal."}
```

▸ 🤓 Other versions and variants

Python 3.8+

```
from typing import Union

from fastapi import FastAPI, Response
from fastapi.responses import RedirectResponse

app = FastAPI()


@app.get("/portal")
async def get_portal(teleport: bool = False) -> Union[Response, dict]:
    if teleport:
        return RedirectResponse(url="https://www.youtube.com/watch?v=dQw4w9WgXcQ")
    return {"message": "Here's your interdimensional portal."}
```

...this fails because the type annotation is not a Pydantic type and is not just a single `Response` class or subclass, it's a union (any of the two) between a `Response` and a `dict`.

Disable Response Model

Continuing from the example above, you might not want to have the default data validation, documentation, filtering, etc. that is performed by FastAPI.

But you might want to still keep the return type annotation in the function to get the support from tools like editors and type checkers (e.g. mypy).

In this case, you can disable the response model generation by setting `response_model=None`:

Python 3.10+

```
from fastapi import FastAPI, Response
from fastapi.responses import RedirectResponse

app = FastAPI()


@app.get("/portal", response_model=None)
async def get_portal(teleport: bool = False) -> Response | dict:
    if teleport:
        return RedirectResponse(url="https://www.youtube.com/watch?v=dQw4w9WgXcQ")
    return {"message": "Here's your interdimensional portal."}
```

▸ 🤓 Other versions and variants

Python 3.8+

```
from typing import Union

from fastapi import FastAPI, Response
from fastapi.responses import RedirectResponse
```

```
app = FastAPI()


@app.get("/portal", response_model=None)
async def get_portal(teleport: bool = False) -> Union[Response, dict]:
    if teleport:
        return RedirectResponse(url="https://www.youtube.com/watch?v=dQw4w9WgXcQ")
    return {"message": "Here's your interdimensional portal."}
```

This will make FastAPI skip the response model generation and that way you can have any return type annotations you need without it affecting your FastAPI application. 🤓

Response Model encoding parameters

Your response model could have default values, like:

Python 3.10+

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float = 10.5
    tags: list[str] = []


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}


@app.get("/items/{item_id}", response_model=Item, response_model_exclude_unset=True)
async def read_item(item_id: str):
    return items[item_id]
```

▸ 🤓 Other versions and variants

Python 3.9+

```
from typing import Union

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: Union[str, None] = None
    price: float
    tax: float = 10.5
    tags: list[str] = []


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}


@app.get("/items/{item_id}", response_model=Item, response_model_exclude_unset=True)
async def read_item(item_id: str):
    return items[item_id]
```

Python 3.8+

```
from typing import List, Union

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: Union[str, None] = None
    price: float
```

```
    tax: float = 10.5
    tags: List[str] = []


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}


@app.get("/items/{item_id}", response_model=Item, response_model_exclude_unset=True)
async def read_item(item_id: str):
    return items[item_id]
```

- `description: Union[str, None] = None` (or `str | None = None` in Python 3.10) has a default of `None`.
- `tax: float = 10.5` has a default of `10.5`.
- `tags: List[str] = []` has a default of an empty list: `[]`.

but you might want to omit them from the result if they were not actually stored.

For example, if you have models with many optional attributes in a NoSQL database, but you don't want to send very long JSON responses full of default values.

Use the `response_model_exclude_unset` parameter

You can set the *path operation decorator* parameter `response_model_exclude_unset=True`:

Python 3.10+

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float = 10.5
    tags: list[str] = []


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}


@app.get("/items/{item_id}", response_model=Item, response_model_exclude_unset=True)
async def read_item(item_id: str):
    return items[item_id]
```

▸ 🤓 Other versions and variants

Python 3.9+

```
from typing import Union

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: Union[str, None] = None
    price: float
    tax: float = 10.5
    tags: list[str] = []


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}


@app.get("/items/{item_id}", response_model=Item, response_model_exclude_unset=True)
async def read_item(item_id: str):
    return items[item_id]
```

Python 3.8+

```python
from typing import List, Union

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: Union[str, None] = None
    price: float
    tax: float = 10.5
    tags: List[str] = []


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}


@app.get("/items/{item_id}", response_model=Item, response_model_exclude_unset=True)
async def read_item(item_id: str):
    return items[item_id]
```

and those default values won't be included in the response, only the values actually set.

So, if you send a request to that *path operation* for the item with ID `foo`, the response (not including default values) will be:

```
{
    "name": "Foo",
    "price": 50.2
}
```

Info

In Pydantic v1 the method was called `.dict()`, it was deprecated (but still supported) in Pydantic v2, and renamed to `.model_dump()`.

The examples here use `.dict()` for compatibility with Pydantic v1, but you should use `.model_dump()` instead if you can use Pydantic v2.

Info

FastAPI uses Pydantic model's `.dict()` with its `exclude_unset` parameter to achieve this.

Info

You can also use:

- `response_model_exclude_defaults=True`
- `response_model_exclude_none=True`

as described in the Pydantic docs for `exclude_defaults` and `exclude_none`.

Data with values for fields with defaults

But if your data has values for the model's fields with default values, like the item with ID `bar`:

```
{
    "name": "Bar",
    "description": "The bartenders",
    "price": 62,
    "tax": 20.2
}
```

they will be included in the response.

Data with the same values as the defaults

If the data has the same values as the default ones, like the item with ID `baz`:

```
{
    "name": "Baz",
    "description": None,
    "price": 50.2,
    "tax": 10.5,
    "tags": []
}
```

FastAPI is smart enough (actually, Pydantic is smart enough) to realize that, even though `description`, `tax`, and `tags` have the same values as the defaults, they were set explicitly (instead of taken from the defaults).

So, they will be included in the JSON response.

Tip

Notice that the default values can be anything, not only `None`.

They can be a list (`[]`), a `float` of `10.5`, etc.

`response_model_include` and `response_model_exclude`

You can also use the *path operation decorator* parameters `response_model_include` and `response_model_exclude`.

They take a `set` of `str` with the name of the attributes to include (omitting the rest) or to exclude (including the rest).

This can be used as a quick shortcut if you have only one Pydantic model and want to remove some data from the output.

Tip

But it is still recommended to use the ideas above, using multiple classes, instead of these parameters.

This is because the JSON Schema generated in your app's OpenAPI (and the docs) will still be the one for the complete model, even if you use `response_model_include` or `response_model_exclude` to omit some attributes.

This also applies to `response_model_by_alias` that works similarly.

Python 3.10+

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float = 10.5


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The Bar fighters", "price": 62, "tax": 20.2},
    "baz": {
        "name": "Baz",
        "description": "There goes my baz",
        "price": 50.2,
        "tax": 10.5,
    },
}


@app.get(
    "/items/{item_id}/name",
    response_model=Item,
    response_model_include={"name", "description"},
)
async def read_item_name(item_id: str):
    return items[item_id]


@app.get("/items/{item_id}/public", response_model=Item, response_model_exclude={"tax"})
async def read_item_public_data(item_id: str):
    return items[item_id]
```

▸ Other versions and variants

Python 3.8+

```
from typing import Union

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: Union[str, None] = None
    price: float
    tax: float = 10.5


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The Bar fighters", "price": 62, "tax": 20.2},
    "baz": {
        "name": "Baz",
        "description": "There goes my baz",
        "price": 50.2,
```

```
        "tax": 10.5,
    },
}


@app.get(
    "/items/{item_id}/name",
    response_model=Item,
    response_model_include={"name", "description"},
)
async def read_item_name(item_id: str):
    return items[item_id]


@app.get("/items/{item_id}/public", response_model=Item, response_model_exclude={"tax"})
async def read_item_public_data(item_id: str):
    return items[item_id]
```

Tip

The syntax `{"name", "description"}` creates a `set` with those two values.

It is equivalent to `set(["name", "description"])`.

Using `list`s instead of `set`s

If you forget to use a `set` and use a `list` or `tuple` instead, FastAPI will still convert it to a `set` and it will work correctly:

Python 3.10+

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float = 10.5


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The Bar fighters", "price": 62, "tax": 20.2},
    "baz": {
        "name": "Baz",
        "description": "There goes my baz",
        "price": 50.2,
        "tax": 10.5,
    },
}


@app.get(
    "/items/{item_id}/name",
    response_model=Item,
    response_model_include=["name", "description"],
)
async def read_item_name(item_id: str):
    return items[item_id]


@app.get("/items/{item_id}/public", response_model=Item, response_model_exclude=["tax"])
async def read_item_public_data(item_id: str):
    return items[item_id]
```

▸ Other versions and variants

Python 3.8+

```
from typing import Union

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: Union[str, None] = None
    price: float
    tax: float = 10.5


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The Bar fighters", "price": 62, "tax": 20.2},
```

```
    "baz": {
        "name": "Baz",
        "description": "There goes my baz",
        "price": 50.2,
        "tax": 10.5,
    },
}


@app.get(
    "/items/{item_id}/name",
    response_model=Item,
    response_model_include=["name", "description"],
)
async def read_item_name(item_id: str):
    return items[item_id]


@app.get("/items/{item_id}/public", response_model=Item, response_model_exclude=["tax"])
async def read_item_public_data(item_id: str):
    return items[item_id]
```

Recap

Use the *path operation decorator*'s parameter `response_model` to define response models and especially to ensure private data is filtered out.

Use `response_model_exclude_unset` to return only the values explicitly set.