Extra Models

Continuing with the previous example, it will be common to have more than one related model.

This is especially the case for user models, because:

- The **input model** needs to be able to have a password.
- The **output model** should not have a password.
- The **database model** would probably need to have a hashed password.

> **Danger**
>
> Never store user's plaintext passwords. Always store a "secure hash" that you can then verify.
>
> If you don't know, you will learn what a "password hash" is in the security chapters.

Multiple models

Here's a general idea of how the models could look like with their password fields and the places where they are used:

Python 3.10+

```python
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


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


def fake_password_hasher(raw_password: str):
    return "supersecret" + raw_password


def fake_save_user(user_in: UserIn):
    hashed_password = fake_password_hasher(user_in.password)
    user_in_db = UserInDB(**user_in.dict(), hashed_password=hashed_password)
    print("User saved! ..not really")
    return user_in_db


@app.post("/user/", response_model=UserOut)
async def create_user(user_in: UserIn):
    user_saved = fake_save_user(user_in)
    return user_saved
```

▶ 😎 Other versions and variants

Python 3.8+

```python
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


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

```python
class UserInDB(BaseModel):
    username: str
    hashed_password: str
    email: EmailStr
    full_name: Union[str, None] = None


def fake_password_hasher(raw_password: str):
    return "supersecret" + raw_password


def fake_save_user(user_in: UserIn):
    hashed_password = fake_password_hasher(user_in.password)
    user_in_db = UserInDB(**user_in.dict(), hashed_password=hashed_password)
    print("User saved! ..not really")
    return user_in_db


@app.post("/user/", response_model=UserOut)
async def create_user(user_in: UserIn):
    user_saved = fake_save_user(user_in)
    return user_saved
```

> **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.

About `**user_in.dict()`

Pydantic's `.dict()`

`user_in` is a Pydantic model of class `UserIn`.

Pydantic models have a `.dict()` method that returns a `dict` with the model's data.

So, if we create a Pydantic object `user_in` like:

```python
user_in = UserIn(username="john", password="secret", email="john.doe@example.com")
```

and then we call:

```python
user_dict = user_in.dict()
```

we now have a `dict` with the data in the variable `user_dict` (it's a `dict` instead of a Pydantic model object).

And if we call:

```python
print(user_dict)
```

we would get a Python `dict` with:

```python
{
    'username': 'john',
    'password': 'secret',
    'email': 'john.doe@example.com',
    'full_name': None,
}
```

Unwrapping a `dict`

If we take a `dict` like `user_dict` and pass it to a function (or class) with `**user_dict`, Python will "unwrap" it. It will pass the keys and values of the `user_dict` directly as key-value arguments.

So, continuing with the `user_dict` from above, writing:

```python
UserInDB(**user_dict)
```

would result in something equivalent to:

```python
UserInDB(
    username="john",
    password="secret",
    email="john.doe@example.com",
    full_name=None,
)
```

Or more exactly, using `user_dict` directly, with whatever contents it might have in the future:

```python
UserInDB(
    username = user_dict["username"],
    password = user_dict["password"],
    email = user_dict["email"],
    full_name = user_dict["full_name"],
```

```
)
```

A Pydantic model from the contents of another

As in the example above we got `user_dict` from `user_in.dict()`, this code:

```
user_dict = user_in.dict()
UserInDB(**user_dict)
```

would be equivalent to:

```
UserInDB(**user_in.dict())
```

...because `user_in.dict()` is a `dict`, and then we make Python "unwrap" it by passing it to `UserInDB` prefixed with `**`.

So, we get a Pydantic model from the data in another Pydantic model.

Unwrapping a `dict` and extra keywords

And then adding the extra keyword argument `hashed_password=hashed_password`, like in:

```
UserInDB(**user_in.dict(), hashed_password=hashed_password)
```

...ends up being like:

```
UserInDB(
    username = user_dict["username"],
    password = user_dict["password"],
    email = user_dict["email"],
    full_name = user_dict["full_name"],
    hashed_password = hashed_password,
)
```

> **Warning**
>
> The supporting additional functions `fake_password_hasher` and `fake_save_user` are just to demo a possible flow of the data, but they of course are not providing any real security.

Reduce duplication

Reducing code duplication is one of the core ideas in FastAPI.

As code duplication increments the chances of bugs, security issues, code desynchronization issues (when you update in one place but not in the others), etc.

And these models are all sharing a lot of the data and duplicating attribute names and types.

We could do better.

We can declare a `UserBase` model that serves as a base for our other models. And then we can make subclasses of that model that inherit its attributes (type declarations, validation, etc).

All the data conversion, validation, documentation, etc. will still work as normally.

That way, we can declare just the differences between the models (with plaintext `password`, with `hashed_password` and without password):

Python 3.10+

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

app = FastAPI()


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


class UserIn(UserBase):
    password: str


class UserOut(UserBase):
    pass


class UserInDB(UserBase):
    hashed_password: str


def fake_password_hasher(raw_password: str):
    return "supersecret" + raw_password


def fake_save_user(user_in: UserIn):
    hashed_password = fake_password_hasher(user_in.password)
    user_in_db = UserInDB(**user_in.dict(), hashed_password=hashed_password)
    print("User saved! ..not really")
```

```
        return user_in_db


@app.post("/user/", response_model=UserOut)
async def create_user(user_in: UserIn):
    user_saved = fake_save_user(user_in)
    return user_saved
```

▸ 🤓 Other versions and variants

Python 3.8+

```python
from typing import Union

from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()


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


class UserIn(UserBase):
    password: str


class UserOut(UserBase):
    pass


class UserInDB(UserBase):
    hashed_password: str


def fake_password_hasher(raw_password: str):
    return "supersecret" + raw_password


def fake_save_user(user_in: UserIn):
    hashed_password = fake_password_hasher(user_in.password)
    user_in_db = UserInDB(**user_in.dict(), hashed_password=hashed_password)
    print("User saved! ..not really")
    return user_in_db


@app.post("/user/", response_model=UserOut)
async def create_user(user_in: UserIn):
    user_saved = fake_save_user(user_in)
    return user_saved
```

Union **or** anyOf

You can declare a response to be the `Union` of two or more types, that means, that the response would be any of them.

It will be defined in OpenAPI with `anyOf`.

To do that, use the standard Python type hint `typing.Union`:

> **Note**
>
> When defining a `Union`, include the most specific type first, followed by the less specific type. In the example below, the more specific `PlaneItem` comes before `CarItem` in `Union[PlaneItem, CarItem]`.

Python 3.10+

```python
from typing import Union

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class BaseItem(BaseModel):
    description: str
    type: str


class CarItem(BaseItem):
    type: str = "car"


class PlaneItem(BaseItem):
    type: str = "plane"
```

```
        size: int


items = {
    "item1": {"description": "All my friends drive a low rider", "type": "car"},
    "item2": {
        "description": "Music is my aeroplane, it's my aeroplane",
        "type": "plane",
        "size": 5,
    },
}


@app.get("/items/{item_id}", response_model=Union[PlaneItem, CarItem])
async def read_item(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 BaseItem(BaseModel):
    description: str
    type: str


class CarItem(BaseItem):
    type: str = "car"


class PlaneItem(BaseItem):
    type: str = "plane"
    size: int


items = {
    "item1": {"description": "All my friends drive a low rider", "type": "car"},
    "item2": {
        "description": "Music is my aeroplane, it's my aeroplane",
        "type": "plane",
        "size": 5,
    },
}


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

`Union` **in Python 3.10**

In this example we pass `Union[PlaneItem, CarItem]` as the value of the argument `response_model`.

Because we are passing it as a **value to an argument** instead of putting it in a **type annotation**, we have to use `Union` even in Python 3.10.

If it was in a type annotation we could have used the vertical bar, as:

```
some_variable: PlaneItem | CarItem
```

But if we put that in the assignment `response_model=PlaneItem | CarItem` we would get an error, because Python would try to perform an **invalid operation** between `PlaneItem` and `CarItem` instead of interpreting that as a type annotation.

List of models

The same way, you can declare responses of lists of objects.

For that, use the standard Python `typing.List` (or just `list` in Python 3.9 and above):

Python 3.9+

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: str
```

```
items = [
    {"name": "Foo", "description": "There comes my hero"},
    {"name": "Red", "description": "It's my aeroplane"},
]


@app.get("/items/", response_model=list[Item])
async def read_items():
    return items
```

▶ 🤓 Other versions and variants

Python 3.8+

```
from typing import List

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: str


items = [
    {"name": "Foo", "description": "There comes my hero"},
    {"name": "Red", "description": "It's my aeroplane"},
]


@app.get("/items/", response_model=List[Item])
async def read_items():
    return items
```

Response with arbitrary `dict`

You can also declare a response using a plain arbitrary `dict`, declaring just the type of the keys and values, without using a Pydantic model.

This is useful if you don't know the valid field/attribute names (that would be needed for a Pydantic model) beforehand.

In this case, you can use `typing.Dict` (or just `dict` in Python 3.9 and above):

Python 3.9+

```
from fastapi import FastAPI

app = FastAPI()


@app.get("/keyword-weights/", response_model=dict[str, float])
async def read_keyword_weights():
    return {"foo": 2.3, "bar": 3.4}
```

▶ 🤓 Other versions and variants

Python 3.8+

```
from typing import Dict

from fastapi import FastAPI

app = FastAPI()


@app.get("/keyword-weights/", response_model=Dict[str, float])
async def read_keyword_weights():
    return {"foo": 2.3, "bar": 3.4}
```

Recap

Use multiple Pydantic models and inherit freely for each case.

You don't need to have a single data model per entity if that entity must be able to have different "states". As the case with the user "entity" with a state including `password`, `password_hash` and no password.

Exported from DevDocs — https://devdocs.io