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](secu
```

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 3.6 and above"

~~~~
```Python hl_lines="9  11  16  22  24  29-30  33-35  40-41"
{!> ../../../docs_src/extra_models/tutorial001.py!}
```
~~~~

=== "Python 3.10 and above"

~~~~
```Python hl_lines="7  9  14  20  22  27-28  31-33  38-39"
{!> ../../../docs_src/extra_models/tutorial001_py310.py!}
```
~~~~

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:

```
{
    '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:

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

Would result in something equivalent to:

```
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:

```
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` prepended 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 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 3.6 and above”

~~~
```Python hl_lines="9  15-16  19-20  23-24"
{!> ../../../docs_src/extra_models/tutorial002.py!}
```
~~~

=== “Python 3.10 and above”

~~~
```Python hl_lines="7  13-14  17-18  21-22"
{!> ../../../docs_src/extra_models/tutorial002_py310.py!}
```
~~~

`Union` or `anyOf`

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

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 3.6 and above”

~~~~
```Python hl_lines="1  14-15  18-20  33"
{!> ../../../docs_src/extra_models/tutorial003.py!}
```
~~~~

=== “Python 3.10 and above”

~~~~
```Python hl_lines="1  14-15  18-20  33"
{!> ../../../docs_src/extra_models/tutorial003_py310.py!}
```
~~~~

`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 `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 3.6 and above”

~~~~
```Python hl_lines="1  20"
{!> ../../../docs_src/extra_models/tutorial004.py!}
```
~~~~

=== “Python 3.9 and above”

~~~
```Python hl_lines="18"
{!> ../../../docs_src/extra_models/tutorial004_py39.py!}
```
~~~

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 3.6 and above”

~~~
```Python hl_lines="1  8"
{!> ../../../docs_src/extra_models/tutorial005.py!}
```
~~~

=== “Python 3.9 and above”

~~~
```Python hl_lines="6"
{!> ../../../docs_src/extra_models/tutorial005_py39.py!}
```
~~~

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.