额外的模型

我们从前面的示例继续，拥有多个相关的模型是很常见的。

对用户模型来说尤其如此，因为：

- **输入模型**需要拥有密码属性。
- **输出模型**不应该包含密码。
- **数据库模型**很可能需要保存密码的哈希值。

!!! danger 永远不要存储用户的明文密码。始终存储一个可以用于验证的「安全哈希值」。

如果你尚未了解该知识，你可以在[安全章节](security/simple-oauth2.md#password-hashing){.internal-link target=_blank}中学习何为「密码哈希值」。

多个模型

下面是应该如何根据它们的密码字段以及使用位置去定义模型的大概思路：

```
{!../../../docs_src/extra_models/tutorial001.py!}
```

关于 `**user_in.dict()`

Pydantic 的 `.dict()`

`user_in` 是一个 `UserIn` 类的 Pydantic 模型.

Pydantic 模型具有 `.dict()` 方法，该方法返回一个拥有模型数据的 `dict`。

因此，如果我们像下面这样创建一个 Pydantic 对象 `user_in`：

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

然后我们调用：

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

现在我们有了一个数据位于变量 `user_dict` 中的 `dict`（它是一个 `dict` 而不是 Pydantic 模型对象）。

如果我们调用：

```
print(user_dict)
```

我们将获得一个这样的 Python `dict`：

```
{
    'username': 'john',
    'password': 'secret',
    'email': 'john.doe@example.com',
```

```
    'full_name': None,
}
```

解包 `dict`

如果我们将 `user_dict` 这样的 `dict` 以 `**user_dict` 形式传递给一个函数（或类），Python将对其进行「解包」。它会将 `user_dict` 的键和值作为关键字参数直接传递。

因此，从上面的 `user_dict` 继续，编写：

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

会产生类似于以下的结果：

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

或者更确切地，直接使用 `user_dict` 来表示将来可能包含的任何内容：

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

来自于其他模型内容的 Pydantic 模型

如上例所示，我们从 `user_in.dict()` 中获得了 `user_dict`，此代码：

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

等同于：

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

...因为 `user_in.dict()` 是一个 `dict`，然后我们通过以 `**` 开头传递给 `UserInDB` 来使 Python「解包」它。

这样，我们获得了一个来自于其他 Pydantic 模型中的数据的 Pydantic 模型。

解包 `dict` 和额外关键字

然后添加额外的关键字参数 `hashed_password=hashed_password`，例如：

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

...最终的结果如下：

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

!!! warning 辅助性的额外函数只是为了演示可能的数据流，但它们显然不能提供任何真正的安全性。

减少重复

减少代码重复是 **FastAPI** 的核心思想之一。

因为代码重复会增加出现 bug、安全性问题、代码失步问题（当你在一个位置更新了代码但没有在其他位置更新）等的可能性。

上面的这些模型都共享了大量数据，并拥有重复的属性名称和类型。

我们可以做得更好。

我们可以声明一个 `UserBase` 模型作为其他模型的基类。然后我们可以创建继承该模型属性（类型声明，校验等）的子类。

所有的数据转换、校验、文档生成等仍将正常运行。

这样，我们可以仅声明模型之间的差异部分（具有明文的 `password`、具有 `hashed_password` 以及不包括密码）。

```
{!../../../docs_src/extra_models/tutorial002.py!}
```

`Union` 或者 `anyOf`

你可以将一个响应声明为两种类型的 `Union`，这意味着该响应将是两种类型中的任何一种。

这将在 OpenAPI 中使用 `anyOf` 进行定义。

为此，请使用标准的 Python 类型提示 `typing.Union`：

!!! note 定义一个 `Union` 类型时，首先包括最详细的类型，然后是不太详细的类型。在下面的示例中，更详细的 `PlaneItem` 位于 `Union[PlaneItem, CarItem]` 中的 `CarItem` 之前。

```
{!../../../docs_src/extra_models/tutorial003.py!}
```

模型列表

你可以用同样的方式声明由对象列表构成的响应。

为此，请使用标准的 Python `typing.List`：

```
{!../../../docs_src/extra_models/tutorial004.py!}
```

任意 `dict` 构成的响应

你还可以使用一个任意的普通 `dict` 声明响应，仅声明键和值的类型，而不使用 Pydantic 模型。

如果你事先不知道有效的字段/属性名称（对于 Pydantic 模型是必需的），这将很有用。

在这种情况下，你可以使用 `typing.Dict`：

```
{!../../../docs_src/extra_models/tutorial005.py!}
```

总结

使用多个 Pydantic 模型，并针对不同场景自由地继承。

如果一个实体必须能够具有不同的「状态」，你无需为每个状态的实体定义单独的数据模型。以用户「实体」为例，其状态有包含 `password`、包含 `password_hash` 以及不含密码。