获取当前用户

在上一章节中，（基于依赖项注入系统的）安全系统向*路径操作函数*提供了一个 `str` 类型的 `token`：

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

但这还不是很实用。

让我们来使它返回当前用户给我们。

创建一个用户模型

首先，让我们来创建一个用户 Pydantic 模型。

与使用 Pydantic 声明请求体的方式相同，我们可以在其他任何地方使用它：

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

创建一个 `get_current_user` 依赖项

让我们来创建一个 `get_current_user` 依赖项。

还记得依赖项可以有子依赖项吗？

`get_current_user` 将具有一个我们之前所创建的同一个 `oauth2_scheme` 作为依赖项。

与我们之前直接在路径操作中所做的相同，我们新的依赖项 `get_current_user` 将从子依赖项 `oauth2_scheme` 中接收一个 `str` 类型的 `token`：

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

获取用户

`get_current_user` 将使用我们创建的（伪）工具函数，该函数接收 `str` 类型的令牌并返回我们的 Pydantic `User` 模型：

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

注入当前用户

因此现在我们可以在*路径操作*中使用 `get_current_user` 作为 `Depends` 了：

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

注意我们将 `current_user` 的类型声明为 Pydantic 模型 `User`。

这将帮助我们在函数内部使用所有的代码补全和类型检查。

!!! tip 你可能还记得请求体也是使用 Pydantic 模型来声明的。

在这里 `**FastAPI**` 不会搞混，因为你正在使用的是 `` `Depends` ``。

!!! check 这种依赖系统的设计方式使我们可以拥有不同的依赖项（不同的「可依赖类型」），并且它们都返回一个 `User` 模型。

我们并未被局限于只能有一个返回该类型数据的依赖项。

其他模型

现在你可以直接在*路径操作函数*中获取当前用户，并使用 `Depends` 在**依赖注入**级别处理安全性机制。

你可以使用任何模型或数据来满足安全性要求（在这个示例中，使用的是 Pydantic 模型 `User`）。

但是你并未被限制只能使用某些特定的数据模型，类或类型。

你想要在模型中使用 `id` 和 `email` 而不使用任何的 `username`？当然可以。你可以同样地使用这些工具。

你只想要一个 `str`？或者仅仅一个 `dict`？还是直接一个数据库模型类的实例？它们的工作方式都是一样的。

实际上你没有用户登录到你的应用程序，而是只拥有访问令牌的机器人，程序或其他系统？再一次，它们的工作方式也是一样的。

尽管去使用你的应用程序所需要的任何模型，任何类，任何数据库。**FastAPI** 通过依赖项注入系统都帮你搞定。

代码体积

这个示例似乎看起来很冗长。考虑到我们在同一文件中混合了安全性，数据模型工具函数和路径操作等代码。

但关键的是。

安全性和依赖项注入内容只需要编写一次。

你可以根据需要使其变得很复杂。而且只需要在一个地方写一次。但仍然具备所有的灵活性。

但是，你可以有无数个使用同一安全系统的端点（*路径操作*）。

所有（或所需的任何部分）的端点，都可以利用对这些或你创建的其他依赖项进行复用所带来的优势。

所有的这无数个*路径操作*甚至可以小到只需 3 行代码：

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

总结

现在你可以直接在*路径操作函数*中获取当前用户。

我们已经进行到一半了。

我们只需要再为用户/客户端添加一个真正发送 `username` 和 `password` 的*路径操作*。

这些内容在下一章节。