Dependencies - First Steps

FastAPI has a very powerful but intuitive **Dependency Injection** system.

It is designed to be very simple to use, and to make it very easy for any developer to integrate other components with **FastAPI**.

What is "Dependency Injection"

"Dependency Injection" means, in programming, that there is a way for your code (in this case, your *path operation functions*) to declare things that it requires to work and use: "dependencies".

And then, that system (in this case **FastAPI**) will take care of doing whatever is needed to provide your code with those needed dependencies ("inject" the dependencies).

This is very useful when you need to:

- Have shared logic (the same code logic again and again).
- Share database connections.
- Enforce security, authentication, role requirements, etc.
- And many other things...

All these, while minimizing code repetition.

First Steps

Let's see a very simple example. It will be so simple that it is not very useful, for now.

But this way we can focus on how the **Dependency Injection** system works.

Create a dependency, or "dependable"

Let's first focus on the dependency.

It is just a function that can take all the same parameters that a *path operation function* can take:

=== "Python 3.6 and above"

```Python hl_lines="8-9"
{!> ../../../docs_src/dependencies/tutorial001.py!}
```

=== "Python 3.10 and above"

```Python hl_lines="6-7"
{!> ../../../docs_src/dependencies/tutorial001_py310.py!}
```

That's it.

2 lines.

And it has the same shape and structure that all your *path operation functions* have.

You can think of it as a *path operation function* without the "decorator" (without the `@app.get("/some-path")`).

And it can return anything you want.

In this case, this dependency expects:

- An optional query parameter `q` that is a `str`.
- An optional query parameter `skip` that is an `int`, and by default is `0`.
- An optional query parameter `limit` that is an `int`, and by default is `100`.

And then it just returns a `dict` containing those values.

Import `Depends`

=== "Python 3.6 and above"

```Python hl_lines="3"
{!> ../../../docs_src/dependencies/tutorial001.py!}
```

=== "Python 3.10 and above"

```Python hl_lines="1"
{!> ../../../docs_src/dependencies/tutorial001_py310.py!}
```

Declare the dependency, in the "dependant"

The same way you use `Body`, `Query`, etc. with your *path operation function* parameters, use `Depends` with a new parameter:

=== "Python 3.6 and above"

```Python hl_lines="13  18"
{!> ../../../docs_src/dependencies/tutorial001.py!}
```

=== "Python 3.10 and above"

```Python hl_lines="11  16"
{!> ../../../docs_src/dependencies/tutorial001_py310.py!}
```

Although you use `Depends` in the parameters of your function the same way you use `Body`, `Query`, etc, `Depends` works a bit differently.

You only give `Depends` a single parameter.

This parameter must be something like a function.

And that function takes parameters in the same way that *path operation functions* do.

!!! tip You'll see what other "things", apart from functions, can be used as dependencies in the next chapter.

Whenever a new request arrives, **FastAPI** will take care of:

- Calling your dependency ("dependable") function with the correct parameters.
- Get the result from your function.
- Assign that result to the parameter in your *path operation function*.

```
graph TB

common_parameters(["common_parameters"])
read_items["/items/"]
read_users["/users/"]

common_parameters --> read_items
common_parameters --> read_users
```

This way you write shared code once and **FastAPI** takes care of calling it for your *path operations*.

!!! check Notice that you don't have to create a special class and pass it somewhere to **FastAPI** to "register" it or anything similar.

```
You just pass it to `Depends` and **FastAPI** knows how to do the rest.
```

To `async` or not to `async`

As dependencies will also be called by **FastAPI** (the same as your *path operation functions*), the same rules apply while defining your functions.

You can use `async def` or normal `def`.

And you can declare dependencies with `async def` inside of normal `def` *path operation functions*, or `def` dependencies inside of `async def` *path operation functions*, etc.

It doesn't matter. **FastAPI** will know what to do.

!!! note If you don't know, check the Async: *"In a hurry?"* section about `async` and `await` in the docs.

Integrated with OpenAPI

All the request declarations, validations and requirements of your dependencies (and sub-dependencies) will be integrated in the same OpenAPI schema.

So, the interactive docs will have all the information from these dependencies too:

Simple usage

If you look at it, *path operation functions* are declared to be used whenever a *path* and *operation* matches, and then **FastAPI** takes care of calling the function with the correct parameters, extracting the data from the request.

Actually, all (or most) of the web frameworks work in this same way.

You never call those functions directly. They are called by your framework (in this case, **FastAPI**).

With the Dependency Injection system, you can also tell **FastAPI** that your *path operation function* also "depends" on something else that should be executed before your *path operation function*, and **FastAPI** will take care of executing it and "injecting" the results.

Other common terms for this same idea of "dependency injection" are:

- resources
- providers
- services
- injectables
- components

FastAPI plug-ins

Integrations and "plug-in"s can be built using the **Dependency Injection** system. But in fact, there is actually **no need to create "plug-ins"**, as by using dependencies it's possible to declare an infinite number of integrations and interactions that become available to your *path operation functions*.

And dependencies can be created in a very simple and intuitive way that allow you to just import the Python packages you need, and integrate them with your API functions in a couple of lines of code, *literally*.

You will see examples of this in the next chapters, about relational and NoSQL databases, security, etc.

FastAPI compatibility

The simplicity of the dependency injection system makes **FastAPI** compatible with:

- all the relational databases
- NoSQL databases
- external packages
- external APIs
- authentication and authorization systems
- API usage monitoring systems
- response data injection systems
- etc.

Simple and Powerful

Although the hierarchical dependency injection system is very simple to define and use, it's still very powerful.

You can define dependencies that in turn can define dependencies themselves.

In the end, a hierarchical tree of dependencies is built, and the **Dependency Injection** system takes care of solving all these dependencies for you (and their sub-dependencies) and providing (injecting) the results at each step.

For example, let's say you have 4 API endpoints (*path operations*):

- `/items/public/`
- `/items/private/`
- `/users/{user_id}/activate`
- `/items/pro/`

then you could add different permission requirements for each of them just with dependencies and sub-dependencies:

```
graph TB

current_user(["current_user"])
active_user(["active_user"])
admin_user(["admin_user"])
paying_user(["paying_user"])

public["/items/public/"]
private["/items/private/"]
activate_user["/users/{user_id}/activate"]
pro_items["/items/pro/"]

current_user --> active_user
active_user --> admin_user
active_user --> paying_user

current_user --> public
active_user --> private
```

```
admin_user --> activate_user
paying_user --> pro_items
```

Integrated with OpenAPI

All these dependencies, while declaring their requirements, also add parameters, validations, etc. to your *path operations*.

FastAPI will take care of adding it all to the OpenAPI schema, so that it is shown in the interactive documentation systems.