OAuth2 scopes

You can use OAuth2 scopes directly with **FastAPI**, they are integrated to work seamlessly.

This would allow you to have a more fine-grained permission system, following the OAuth2 standard, integrated into your OpenAPI application (and the API docs).

OAuth2 with scopes is the mechanism used by many big authentication providers, like Facebook, Google, GitHub, Microsoft, Twitter, etc. They use it to provide specific permissions to users and applications.

Every time you "log in with" Facebook, Google, GitHub, Microsoft, Twitter, that application is using OAuth2 with scopes.

In this section you will see how to manage authentication and authorization with the same OAuth2 with scopes in your **FastAPI** application.

!!! warning This is a more or less advanced section. If you are just starting, you can skip it.

```
You don't necessarily need OAuth2 scopes, and you can handle authentication and
authorization however you want.

But OAuth2 with scopes can be nicely integrated into your API (with OpenAPI) and your
API docs.

Nevertheless, you still enforce those scopes, or any other security/authorization
requirement, however you need, in your code.

In many cases, OAuth2 with scopes can be an overkill.

But if you know you need it, or you are curious, keep reading.
```

OAuth2 scopes and OpenAPI

The OAuth2 specification defines "scopes" as a list of strings separated by spaces.

The content of each of these strings can have any format, but should not contain spaces.

These scopes represent "permissions".

In OpenAPI (e.g. the API docs), you can define "security schemes".

When one of these security schemes uses OAuth2, you can also declare and use scopes.

Each "scope" is just a string (without spaces).

They are normally used to declare specific security permissions, for example:

- `users:read` or `users:write` are common examples.
- `instagram_basic` is used by Facebook / Instagram.
- `https://www.googleapis.com/auth/drive` is used by Google.

!!! info In OAuth2 a "scope" is just a string that declares a specific permission required.

```
It doesn't matter if it has other characters like `:` or if it is a URL.
```

```
Those details are implementation specific.

For OAuth2 they are just strings.
```

Global view

First, let's quickly see the parts that change from the examples in the main **Tutorial - User Guide** for OAuth2 with Password (and hashing), Bearer with JWT tokens{.internal-link target=_blank}. Now using OAuth2 scopes:

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

Now let's review those changes step by step.

OAuth2 Security scheme

The first change is that now we are declaring the OAuth2 security scheme with two available scopes, `me` and `items`.

The `scopes` parameter receives a `dict` with each scope as a key and the description as the value:

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

Because we are now declaring those scopes, they will show up in the API docs when you log-in/authorize.

And you will be able to select which scopes you want to give access to: `me` and `items`.

This is the same mechanism used when you give permissions while logging in with Facebook, Google, GitHub, etc:

JWT token with scopes

Now, modify the token *path operation* to return the scopes requested.

We are still using the same `OAuth2PasswordRequestForm`. It includes a property `scopes` with a `list` of `str`, with each scope it received in the request.

And we return the scopes as part of the JWT token.

!!! danger For simplicity, here we are just adding the scopes received directly to the token.

```
But in your application, for security, you should make sure you only add the scopes
that the user is actually able to have, or the ones you have predefined.
```

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

Declare scopes in *path operations* and dependencies

Now we declare that the *path operation* for `/users/me/items/` requires the scope `items`.

For this, we import and use `Security` from `fastapi`.

You can use `Security` to declare dependencies (just like `Depends`), but `Security` also receives a parameter `scopes` with a list of scopes (strings).

In this case, we pass a dependency function `get_current_active_user` to `Security` (the same way we would do with `Depends`).

But we also pass a `list` of scopes, in this case with just one scope: `items` (it could have more).

And the dependency function `get_current_active_user` can also declare sub-dependencies, not only with `Depends` but also with `Security`. Declaring its own sub-dependency function (`get_current_user`), and more scope requirements.

In this case, it requires the scope `me` (it could require more than one scope).

!!! note You don't necessarily need to add different scopes in different places.

```
We are doing it here to demonstrate how **FastAPI** handles scopes declared at
different levels.
```

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

!!! info "Technical Details" `Security` is actually a subclass of `Depends`, and it has just one extra parameter that we'll see later.

```
But by using `Security` instead of `Depends`, **FastAPI** will know that it can
declare security scopes, use them internally, and document the API with OpenAPI.

But when you import `Query`, `Path`, `Depends`, `Security` and others from `fastapi`,
those are actually functions that return special classes.
```

Use `SecurityScopes`

Now update the dependency `get_current_user`.

This is the one used by the dependencies above.

Here's were we are using the same OAuth2 scheme we created before, declaring it as a dependency: `oauth2_scheme`.

Because this dependency function doesn't have any scope requirements itself, we can use `Depends` with `oauth2_scheme`, we don't have to use `Security` when we don't need to specify security scopes.

We also declare a special parameter of type `SecurityScopes`, imported from `fastapi.security`.

This `SecurityScopes` class is similar to `Request` (`Request` was used to get the request object directly).

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

Use the `scopes`

The parameter `security_scopes` will be of type `SecurityScopes`.

It will have a property `scopes` with a list containing all the scopes required by itself and all the dependencies that use this as a sub-dependency. That means, all the "dependants"... this might sound confusing, it is explained again later below.

The `security_scopes` object (of class `SecurityScopes`) also provides a `scope_str` attribute with a single string, containing those scopes separated by spaces (we are going to use it).

We create an `HTTPException` that we can re-use (`raise`) later at several points.

In this exception, we include the scopes required (if any) as a string separated by spaces (using `scope_str`). We put that string containing the scopes in the `WWW-Authenticate` header (this is part of the spec).

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

Verify the `username` and data shape

We verify that we get a `username`, and extract the scopes.

And then we validate that data with the Pydantic model (catching the `ValidationError` exception), and if we get an error reading the JWT token or validating the data with Pydantic, we raise the `HTTPException` we created before.

For that, we update the Pydantic model `TokenData` with a new property `scopes`.

By validating the data with Pydantic we can make sure that we have, for example, exactly a `list` of `str` with the scopes and a `str` with the `username`.

Instead of, for example, a `dict`, or something else, as it could break the application at some point later, making it a security risk.

We also verify that we have a user with that username, and if not, we raise that same exception we created before.

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

Verify the `scopes`

We now verify that all the scopes required, by this dependency and all the dependants (including *path operations*), are included in the scopes provided in the token received, otherwise raise an `HTTPException`.

For this, we use `security_scopes.scopes`, that contains a `list` with all these scopes as `str`.

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

Dependency tree and scopes

Let's review again this dependency tree and the scopes.

As the `get_current_active_user` dependency has as a sub-dependency on `get_current_user`, the scope `"me"` declared at `get_current_active_user` will be included in the list of required scopes in the `security_scopes.scopes` passed to `get_current_user`.

The *path operation* itself also declares a scope, `"items"`, so this will also be in the list of `security_scopes.scopes` passed to `get_current_user`.

Here's how the hierarchy of dependencies and scopes looks like:

- The *path operation* `read_own_items` has:
  - Required scopes `["items"]` with the dependency:
  - `get_current_active_user`:
    - The dependency function `get_current_active_user` has:
      - Required scopes `["me"]` with the dependency:
      - `get_current_user`:
        - The dependency function `get_current_user` has:
          - No scopes required by itself.
          - A dependency using `oauth2_scheme`.
          - A `security_scopes` parameter of type `SecurityScopes`:
            - This `security_scopes` parameter has a property `scopes` with a `list` containing all these scopes declared above, so:
              - `security_scopes.scopes` will contain `["me", "items"]` for the *path operation* `read_own_items`.
              - `security_scopes.scopes` will contain `["me"]` for the *path operation* `read_users_me`, because it is declared in the dependency `get_current_active_user`.
              - `security_scopes.scopes` will contain `[]` (nothing) for the *path operation* `read_system_status`, because it didn't declare any `Security` with `scopes`, and its dependency, `get_current_user`, doesn't declare any `scope` either.

!!! tip The important and "magic" thing here is that `get_current_user` will have a different list of `scopes` to check for each *path operation*.

```
All depending on the `scopes` declared in each *path operation* and each dependency in
the dependency tree for that specific *path operation*.
```

More details about `SecurityScopes`

You can use `SecurityScopes` at any point, and in multiple places, it doesn't have to be at the "root" dependency.

It will always have the security scopes declared in the current `Security` dependencies and all the dependants for **that specific** *path operation* and **that specific** dependency tree.

Because the `SecurityScopes` will have all the scopes declared by dependants, you can use it to verify that a token has the required scopes in a central dependency function, and then declare different scope requirements in different *path operations*.

They will be checked independently for each *path operation*.

Check it

If you open the API docs, you can authenticate and specify which scopes you want to authorize.

If you don't select any scope, you will be "authenticated", but when you try to access `/users/me/` or `/users/me/items/` you will get an error saying that you don't have enough permissions. You will still be able to access `/status/` .

And if you select the scope `me` but not the scope `items` , you will be able to access `/users/me/` but not `/users/me/items/` .

That's what would happen to a third party application that tried to access one of these *path operations* with a token provided by a user, depending on how many permissions the user gave the application.

About third party integrations

In this example we are using the OAuth2 "password" flow.

This is appropriate when we are logging in to our own application, probably with our own frontend.

Because we can trust it to receive the `username` and `password` , as we control it.

But if you are building an OAuth2 application that others would connect to (i.e., if you are building an authentication provider equivalent to Facebook, Google, GitHub, etc.) you should use one of the other flows.

The most common is the implicit flow.

The most secure is the code flow, but is more complex to implement as it requires more steps. As it is more complex, many providers end up suggesting the implicit flow.

!!! note It's common that each authentication provider names their flows in a different way, to make it part of their brand.

```
But in the end, they are implementing the same OAuth2 standard.
```

FastAPI includes utilities for all these OAuth2 authentication flows in `fastapi.security.oauth2` .

`Security` in decorator `dependencies`

The same way you can define a `list` of `Depends` in the decorator's `dependencies` parameter (as explained in Dependencies in path operation decorators{.internal-link target=_blank}), you could also use `Security` with `scopes` there.