Understanding Dependencies in FastAPI

Jun 7, 2025 • 5 min read
Tags: Python, FastAPI, Dependency

What are dependencies?

We can think of dependencies as helpers for our routes without cluttering up the main logic. Need to check if a user is logged in? Grab a database connection? Instead of cramming all of that into your route function, you offload it to a dependency—a separate function (or class) that handles one specific job.

You just say, *"Hey, I need X"* with `Depends(X)`, and FastAPI will make sure `X` is ready and available when your route runs.

Sample dependency in action

Now lets see how dependency can make our life easier.

Let's say that you have a database of users, and you want to protect your endpoints from unauthorized users. Below is the sample code.

To make things simple, we will use a Python dict to store the users.

```
from fastapi import FastAPI

app = FastAPI()

USERS = [
    {
        "id": 1,
        "name": "John Doe",
        "role": "admin",
        "access_token": "token111"
    },
    {
        "id": 2,
        "name": "Jane Doe",
        "role": "user",
        "access_token": "token222"
    }
]
```

```
@app.get('/protected1')
async def protected1():
    return {"message": "You are accessing our protected 1"}

@app.get('/protected2')
async def protected2():
    return {"message": "You are accessing our protected 2"}
```

We want the users to pass their access token when making the HTTP requests to our endpoints. This access token will be checked against the database, and only users with valid access token are allowed to access the endpoint.

To avoid repeating the same token validation logic in every endpoint, we can use **FastAPI's dependency injection system**.

Create a simple dependency to handle the access token check:

```
# New imports: Header, HTTPException, Depends
from fastapi import FastAPI, Header, HTTPException, Depends

def get_current_user(access_token: str = Header(...)):
    for user in USERS:
        if user["access_token"] == access_token:
            return user
    raise HTTPException(status_code=401, detail="Invalid or missing token")
...
```

Let's break down the code above:

1. We define a function `get_current_user`.
2. It expects the `Access-Token` header in the HTTP request.
3. It loops through the `USERS` list to find a match.
4. Return the user with the matches access token.
5. If no match is found, raise a HTTP 401 error.

Now we can apply this dependency to our endpoints like this:

```
from typing import Annotated
```

```
...

@app.get('/protected1')
async def protected1(user: Annotated[dict, Depends(get_current_user)]):
    return {"message": f"Hello, {user['name']}! You are on protected 1"}

@app.get('/protected2')
async def protected2(user: Annotated[dict, Depends(get_current_user)]):
    return {"message": f"Hello, {user['name']}! You are accessing our
protected 2"}
```

Boom! Now both endpoints are protected. FastAPI will run the `get_current_user` function before hitting our endpoint, and if it passes, the user object is injected into the endpoint.

If user John Doe want to access the protected endpoints, he needs to pass his access token when making the requests:

```
$ curl http://localhost:8000/protected1 -H "Access-Token: token111"
{"message":"Hello, John Doe! You are accessing our protected 1"}
```

Can I write dependency that depends on other dependency?

Yes, you can!

In FastAPI, a **nested dependency** happens when a dependency function itself depends on another dependency. This allows you to build modular and reusable logic without cluttering your endpoints.

Lets say we have another endpoint, `/admin-only`, that requires not only user with valid access token, but they should also have the “admin” role.

We can write the code like this:

```
...

def has_role(required_role: str):
    def role_checker(user: Annotated[dict, Depends(get_current_user)]):
        if user['role'] == required_role:
            return user
        raise HTTPException(status_code=403, detail="Forbidden")
    return role_checker
```

```
@app.get('/admin-only')
async def admin_only(user: Annotated[dict, Depends(has_role('admin'))]):
    return {"message": f"Hello, {user['name']! You are accessing our
admin-only endpoint"}
```

Here, `role_checker` depends on `get_current_user`. When a request hits the `/admin-only` endpoint, FastAPI will first resolve `get_current_user` to get the current user from the `Access-Token` header. If succeed, that user object is then passed to `role_checker`, which checks if the user's role matches the required role.

This chain continues until all dependencies are resolved, and finally, the endpoint function runs with the validated user object.

The cool part is — you can stack or nest as many dependencies as you like, keeping your endpoint functions clean and focused only on the business logic.

Class-based dependencies

You are not limited to use function-based dependencies only. You can also use class-based dependencies for your route functions.

For example, we can rewrite the `get_current_user` and `has_role` dependencies above into class-based dependencies like the following:

```
class CurrentUser(object):
    def __call__(self, access_token: str = Header(...)):
        for user in USERS:
            if user["access_token"] == access_token:
                return user
        raise HTTPException(
            status_code=401,
            detail="Invalid or missing token"
        )


class RoleChecker(object):
    def __init__(self, required_role: str):
        self.required_role = required_role

    def __call__(self, user: Annotated[dict, Depends(CurrentUser())]):
        if user['role'] == self.required_role:
            return user
        raise HTTPException(status_code=403, detail="Forbidden")
```

Apply the dependencies to the routes:

```
@app.get('/protected1')
async def protected1(user: Annotated[dict, Depends(CurrentUser())]):
...

@app.get('/protected2')
async def protected2(user: Annotated[dict, Depends(CurrentUser())]):
...

@app.get('/admin-only')
async def admin_only(user: Annotated[dict, Depends(RoleChecker('admin'))]):
...
```

Just like before, the `/protected1` and `/protected2` endpoints only require a valid token, while `/admin-only` checks both the token and that the user has the "admin" role.

Also note that it is still a nested dependencies like before–`RoleChecker` depends on `CurrentUser` to get the authenticated user before checking their role.

The complete code

Below is the complete code for your reference:

```
from typing import Annotated
from fastapi import FastAPI, Header, HTTPException, Depends


app = FastAPI()

USERS = [
    {
        "id": 1,
        "name": "John Doe",
        "role": "admin",
        "access_token": "token111"
    },
    {
        "id": 2,
        "name": "Jane Doe",
        "role": "user",
        "access_token": "token222"
    }
]


def get_current_user(access_token: str = Header(...)):
```

```
    for user in USERS:
        if user["access_token"] == access_token:
            return user
    raise HTTPException(status_code=401, detail="Invalid or missing token")


def has_role(required_role: str):
    def role_checker(user: Annotated[dict, Depends(get_current_user)]):
        if user['role'] == required_role:
            return user
        raise HTTPException(status_code=403, detail="Forbidden")
    return role_checker


class CurrentUser(object):
    """
    Class-based version of the `get_current_user` dependency
    """
    def __call__(self, access_token: str = Header(...)):
        for user in USERS:
            if user["access_token"] == access_token:
                return user
        raise HTTPException(status_code=401, detail="Invalid or missing
token")


class RoleChecker(object):
    """
    Class-based version of the `has_role` dependency
    """
    def __init__(self, required_role: str):
        self.required_role = required_role

    def __call__(self, user: Annotated[dict, Depends(CurrentUser())]):
        if user['role'] == self.required_role:
            return user
        raise HTTPException(status_code=403, detail="Forbidden")


@app.get('/protected1')
async def protected1(user: Annotated[dict, Depends(get_current_user)]):
    return {"message": f"Hello, {user['name']}! You are accessing our
protected 1"}


@app.get('/protected2')
async def protected2(user: Annotated[dict, Depends(get_current_user)]):
    return {"message": f"Hello, {user['name']}! You are accessing our
protected 2"}


@app.get('/admin-only')
```

```
async def admin_only(user: Annotated[dict, Depends(has_role('admin'))]):
    return {"message": f"Hello, {user['name']}! You are accessing our admin
endpoint"}
```