Database

The most natural way for storing tokens is of course the very same database you're using for your application. In this strategy, we set up a table (or collection) for storing those tokens with the associated user id. On each request, we try to retrieve this token from the database to get the corresponding user id.

Configuration

The configuration of this strategy is a bit more complex than the others as it requires you to configure models and a database adapter, exactly like we did for users.

Database adapters

An access token will be structured like this in your database:

- `token` (`str`) – Unique identifier of the token. It's generated automatically upon login by the strategy.
- `user_id` (`ID`) – User id. of the user associated to this token.
- `created_at` (`datetime`) – Date and time of creation of the token. It's used to determine if the token is expired or not.

We are providing a base model with those fields for each database we are supporting.

SQLAlchemy

We'll expand from the basic SQLAlchemy configuration.

```
from collections.abc import AsyncGenerator

from fastapi import Depends
from fastapi_users.db import SQLAlchemyBaseUserTableUUID, SQLAlchemyUserDatabase
from fastapi_users_db_sqlalchemy.access_token import (
    SQLAlchemyAccessTokenDatabase,
    SQLAlchemyBaseAccessTokenTableUUID,
)
from sqlalchemy.ext.asyncio import AsyncSession, async_sessionmaker,
create_async_engine
from sqlalchemy.orm import DeclarativeBase
```

```python
DATABASE_URL = "sqlite+aiosqlite:///./test.db"


class Base(DeclarativeBase):
    pass


class User(SQLAlchemyBaseUserTableUUID, Base):
    pass


class AccessToken(SQLAlchemyBaseAccessTokenTableUUID, Base):  # (1)
    pass


engine = create_async_engine(DATABASE_URL)
async_session_maker = async_sessionmaker(engine, expire_on_commit=False)


async def create_db_and_tables():
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.create_all)


async def get_async_session() -> AsyncGenerator[AsyncSession, None]:
    async with async_session_maker() as session:
        yield session


async def get_user_db(session: AsyncSession = Depends(get_async_session)):
    yield SQLAlchemyUserDatabase(session, User)


async def get_access_token_db(
    session: AsyncSession = Depends(get_async_session),
):  # (2)
    yield SQLAlchemyAccessTokenDatabase(session, AccessToken)
```

1. We define an `AccessToken` ORM model inheriting from `SQLAlchemyBaseAccessTokenTableUUID`.
2. We define a dependency to instantiate the `SQLAlchemyAccessTokenDatabase` class. Just like the user database adapter, it expects a fresh SQLAlchemy session and the `AccessToken` model class we defined above.

`user_id` foreign key is defined as UUID

By default, we use UUID as a primary key ID for your user, so we follow the same convention to define the foreign key pointing to the user.

If you want to use another type, like an auto-incremented integer, you can use `SQLAlchemyBaseAccessTokenTable` as base class and define your own `user_id` column.

```
class AccessToken(SQLAlchemyBaseAccessTokenTable[int], Base):
    @declared_attr
    def user_id(cls) -> Mapped[int]:
        return mapped_column(Integer, ForeignKey("user.id", ondelete="cascade"),
nullable=False)
```

Notice that `SQLAlchemyBaseAccessTokenTable` expects a generic type to define the actual type of ID you use.

Beanie

We'll expand from the basic Beanie configuration.

```
import motor.motor_asyncio
from beanie import Document
from fastapi_users.db import BeanieBaseUser, BeanieUserDatabase
from fastapi_users_db_beanie.access_token import (
    BeanieAccessTokenDatabase,
    BeanieBaseAccessToken,
)

DATABASE_URL = "mongodb://localhost:27017"
client = motor.motor_asyncio.AsyncIOMotorClient(
    DATABASE_URL, uuidRepresentation="standard"
)
db = client["database_name"]


class User(BeanieBaseUser, Document):
    pass


class AccessToken(BeanieBaseAccessToken, Document):  (1)
    pass


async def get_user_db():
    yield BeanieUserDatabase(User)
```

```
async def get_access_token_db():  (2)
    yield BeanieAccessTokenDatabase(AccessToken)
```

(1) We define an `AccessToken` ODM model inheriting from `BeanieBaseAccessToken`. Notice that we set a generic type to define the type of the `user_id` reference. By default, it's a standard MongoDB ObjectID.

(2) We define a dependency to instantiate the `BeanieAccessTokenDatabase` class. Just like the user database adapter, it expects the `AccessToken` model class we defined above.

Don't forget to add the `AccessToken` ODM model to the `document_models` array in your Beanie initialization, just like you did with the `User` model!

> **Info**
>
> If you want to add your own custom settings to your `AccessToken` document model - like changing the collection name - don't forget to let your inner `Settings` class inherit the pre-defined settings from `BeanieBaseAccessToken` like this: `Settings(BeanieBaseAccessToken.Settings): # ...`! See Beanie's documentation on `Settings` for details.

Strategy

```
import uuid

from fastapi import Depends
from fastapi_users.authentication.strategy.db import AccessTokenDatabase,
DatabaseStrategy

from .db import AccessToken, User


def get_database_strategy(
    access_token_db: AccessTokenDatabase[AccessToken] =
Depends(get_access_token_db),
) -> DatabaseStrategy:
    return DatabaseStrategy(access_token_db, lifetime_seconds=3600)
```

As you can see, instantiation is quite simple. It accepts the following arguments:

- `database` (`AccessTokenDatabase`): A database adapter instance for `AccessToken` table, like we defined above.
- `lifetime_seconds` (`int`): The lifetime of the token in seconds.

Why it's inside a function?

To allow strategies to be instantiated dynamically with other dependencies, they have to be provided as a callable to the authentication backend.

As you can see here, this pattern allows us to dynamically inject a connection to the database.

Logout

On logout, this strategy will delete the token from the database.