UserManager

The `UserManager` class is the core logic of FastAPI Users. We provide the `BaseUserManager` class which you should extend to set some parameters and define logic, for example when a user just registered or forgot its password.

It's designed to be easily extensible and customizable so that you can integrate your very own logic.

Create your `UserManager` class

You should define your own version of the `UserManager` class to set various parameters.

```
import uuid
from typing import Optional

from fastapi import Depends, Request
from fastapi_users import BaseUserManager, UUIDIDMixin

from .db import User, get_user_db

SECRET = "SECRET"


class UserManager(UUIDIDMixin, BaseUserManager[User, uuid.UUID]):
    reset_password_token_secret = SECRET
    verification_token_secret = SECRET

    async def on_after_register(self, user: User, request: Optional[Request] =
None):
        print(f"User {user.id} has registered.")

    async def on_after_forgot_password(
        self, user: User, token: str, request: Optional[Request] = None
    ):
        print(f"User {user.id} has forgot their password. Reset token: {token}")

    async def on_after_request_verify(
        self, user: User, token: str, request: Optional[Request] = None
    ):
        print(f"Verification requested for user {user.id}. Verification token:
{token}")
```

```
async def get_user_manager(user_db=Depends(get_user_db)):
    yield UserManager(user_db)
```

As you can see, you have to define here various attributes and methods. You can find the complete list of those below.

Typing: User and ID generic types are expected

You can see that we define two generic types when extending the base class:

- `User`, which is the user model we defined in the database part
- The ID, which should correspond to the type of ID you use on your model. Here, we chose UUID, but it can be anything, like an integer or a MongoDB ObjectID.

It'll help you to have **good type-checking and auto-completion** when implementing the custom methods.

The ID parser mixin

Since the user ID is fully generic, we need a way to **parse it reliably when it'll come from API requests**, typically as URL path attributes.

That's why we added the `UUIDIDMixin` in the example above. It implements the `parse_id` method, ensuring UUID are valid and correctly parsed.

Of course, it's important that this logic **matches the type of your ID**. To help you with this, we provide mixins for the most common cases:

- `UUIDIDMixin`, for UUID ID.
- `IntegerIDMixin`, for integer ID.
- `ObjectIDIDMixin` (provided by `fastapi_users_db_beanie`), for MongoDB ObjectID.

Inheritance order matters

Notice in your example that **the mixin comes first in our `UserManager` inheritance**. Because of the Method-Resolution-Order (MRO) of Python, the left-most element takes precedence.

If you need another type of ID, you can simply overload the `parse_id` method on your `UserManager` class:

```python
from fastapi_users import BaseUserManager, InvalidID


class UserManager(BaseUserManager[User, MyCustomID]):
    def parse_id(self, value: Any) -> MyCustomID:
        try:
            return MyCustomID(value)
        except ValueError as e:
            raise InvalidID() from e   (1)
```

(1) If the ID can't be parsed into the desired type, you'll need to raise an `InvalidID` exception.

Create `get_user_manager` dependency

The `UserManager` class will be injected at runtime using a FastAPI dependency. This way, you can run it in a database session or swap it with a mock during testing.

```python
import uuid
from typing import Optional

from fastapi import Depends, Request
from fastapi_users import BaseUserManager, UUIDIDMixin

from .db import User, get_user_db

SECRET = "SECRET"


class UserManager(UUIDIDMixin, BaseUserManager[User, uuid.UUID]):
    reset_password_token_secret = SECRET
    verification_token_secret = SECRET

    async def on_after_register(self, user: User, request: Optional[Request] =
None):
        print(f"User {user.id} has registered.")

    async def on_after_forgot_password(
        self, user: User, token: str, request: Optional[Request] = None
    ):
        print(f"User {user.id} has forgot their password. Reset token: {token}")

    async def on_after_request_verify(
        self, user: User, token: str, request: Optional[Request] = None
    ):
        print(f"Verification requested for user {user.id}. Verification token:
{token}")


async def get_user_manager(user_db=Depends(get_user_db)):
```

```
    yield UserManager(user_db)
```

Notice that we use the `get_user_db` dependency we defined earlier to inject the database instance.

Customize attributes and methods

Attributes

- `reset_password_token_secret`: Secret to encode reset password token. **Use a strong passphrase and keep it secure.**
- `reset_password_token_lifetime_seconds`: Lifetime of reset password token. Defaults to 3600.
- `reset_password_token_audience`: JWT audience of reset password token. Defaults to `fastapi-users:reset`.
- `verification_token_secret`: Secret to encode verification token. **Use a strong passphrase and keep it secure.**
- `verification_token_lifetime_seconds`: Lifetime of verification token. Defaults to 3600.
- `verification_token_audience`: JWT audience of verification token. Defaults to `fastapi-users:verify`.

Methods

`validate_password`

Validate a password.

Arguments

- `password` (`str`): the password to validate.
- `user` (`Union[UserCreate, User]`): user model which we are currently validating the password. Useful if you want to check that the password doesn't contain the name or the birthdate of the user for example.

Output

This function should return `None` if the password is valid or raise `InvalidPasswordException` if not. This exception expects an argument `reason` telling why the password is invalid. It'll be part of the error response.

Example

```
from fastapi_users import BaseUserManager, InvalidPasswordException, UUIDIDMixin


class UserManager(UUIDIDMixin, BaseUserManager[User, uuid.UUID]):
    # ...
    async def validate_password(
        self,
        password: str,
        user: Union[UserCreate, User],
    ) -> None:
        if len(password) < 8:
            raise InvalidPasswordException(
                reason="Password should be at least 8 characters"
            )
        if user.email in password:
            raise InvalidPasswordException(
                reason="Password should not contain e-mail"
            )
```

`on_after_register`

Perform logic after successful user registration.

Typically, you'll want to **send a welcome e-mail** or add it to your marketing analytics pipeline.

Arguments

- `user` (`User`): the registered user.
- `request` (`Optional[Request]`): optional FastAPI request object that triggered the operation. Defaults to None.

Example

```
from fastapi_users import BaseUserManager, UUIDIDMixin


class UserManager(UUIDIDMixin, BaseUserManager[User, uuid.UUID]):
    # ...
    async def on_after_register(self, user: User, request: Optional[Request] =
None):
        print(f"User {user.id} has registered.")
```

`on_after_update`

Perform logic after successful user update.

It may be useful, for example, if you wish to update your user in a data analytics or customer success platform.

Arguments

- `user` (`User`): the updated user.
- `update_dict` (`Dict[str, Any]`): dictionary with the updated user fields.
- `request` (`Optional[Request]`): optional FastAPI request object that triggered the operation. Defaults to None.

Example

```
from fastapi_users import BaseUserManager, UUIDIDMixin


class UserManager(UUIDIDMixin, BaseUserManager[User, uuid.UUID]):
    # ...
    async def on_after_update(
        self,
        user: User,
        update_dict: Dict[str, Any],
        request: Optional[Request] = None,
    ):
        print(f"User {user.id} has been updated with {update_dict}.")
```

`on_after_login`

Perform logic after a successful user login.

It may be useful for custom logic or processes triggered by new logins, for example a daily login reward or for analytics.

Arguments

- `user` (`User`): the updated user.
- `request` (`Optional[Request]`): optional FastAPI request object that triggered the operation. Defaults to None.
- `response` (`Optional[Response]`): Optional response built by the transport. Defaults to None.

Example

```
from fastapi_users import BaseUserManager, UUIDIDMixin


class UserManager(UUIDIDMixin, BaseUserManager[User, uuid.UUID]):
```

```
    # ...
    async def on_after_login(
        self,
        user: User,
        request: Optional[Request] = None,
        response: Optional[Response] = None,
    ):
        print(f"User {user.id} logged in.")
```

`on_after_request_verify`

Perform logic after successful verification request.

Typically, you'll want to **send an e-mail** with the link (and the token) that allows the user to verify their e-mail.

Arguments

- `user` (`User`): the user to verify.
- `token` (`str`): the verification token.
- `request` (`Optional[Request]`): optional FastAPI request object that triggered the operation. Defaults to None.

Example

```
from fastapi_users import BaseUserManager, UUIDIDMixin


class UserManager(UUIDIDMixin, BaseUserManager[User, uuid.UUID]):
    # ...
    async def on_after_request_verify(
        self, user: User, token: str, request: Optional[Request] = None
    ):
        print(f"Verification requested for user {user.id}. Verification token:
{token}")
```

`on_after_verify`

Perform logic after successful user verification.

This may be useful if you wish to send another e-mail or store this information in a data analytics or customer success platform.

Arguments

- `user` (`User`): the verified user.

- `request` (`Optional[Request]`): optional FastAPI request object that triggered the operation. Defaults to None.

Example

```
from fastapi_users import BaseUserManager, UUIDIDMixin


class UserManager(UUIDIDMixin, BaseUserManager[User, uuid.UUID]):
    # ...
    async def on_after_verify(
        self, user: User, request: Optional[Request] = None
    ):
        print(f"User {user.id} has been verified")
```

`on_after_forgot_password`

Perform logic after successful forgot password request.

Typically, you'll want to **send an e-mail** with the link (and the token) that allows the user to reset their password.

Arguments

- `user` (`User`): the user that forgot its password.
- `token` (`str`): the forgot password token
- `request` (`Optional[Request]`): optional FastAPI request object that triggered the operation. Defaults to None.

Example

```
from fastapi_users import BaseUserManager, UUIDIDMixin


class UserManager(UUIDIDMixin, BaseUserManager[User, uuid.UUID]):
    # ...
    async def on_after_forgot_password(
        self, user: User, token: str, request: Optional[Request] = None
    ):
        print(f"User {user.id} has forgot their password. Reset token: {token}")
```

`on_after_reset_password`

Perform logic after successful password reset.

For example, you may want to **send an e-mail** to the concerned user to warn him that their password has been changed and that they should take action if they think they have been hacked.

Arguments

- `user` (`User`): the user that reset its password.
- `request` (`Optional[Request]`): optional FastAPI request object that triggered the operation. Defaults to None.

Example

```
from fastapi_users import BaseUserManager, UUIDIDMixin


class UserManager(UUIDIDMixin, BaseUserManager[User, uuid.UUID]):
    # ...
    async def on_after_reset_password(self, user: User, request: Optional[Request]
= None):
        print(f"User {user.id} has reset their password.")
```

`on_before_delete`

Perform logic before user delete.

For example, you may want to **valide user resource integrity** to see if any related user resource need to be marked inactive, or delete them recursively.

Arguments

- `user` (`User`): the user to be deleted.
- `request` (`Optional[Request]`): optional FastAPI request object that triggered the operation. Defaults to None.

Example

```
from fastapi_users import BaseUserManager, UUIDIDMixin


class UserManager(UUIDIDMixin, BaseUserManager[User, uuid.UUID]):
    # ...
    async def on_before_delete(self, user: User, request: Optional[Request] =
None):
        print(f"User {user.id} is going to be deleted")
```

`on_after_delete`

Perform logic after user delete.

For example, you may want to **send an email** to the administrator about the event.

Arguments

- `user` (`User`): the user to be deleted.
- `request` (`Optional[Request]`): optional FastAPI request object that triggered the operation. Defaults to None.

Example

```
from fastapi_users import BaseUserManager, UUIDIDMixin


class UserManager(UUIDIDMixin, BaseUserManager[User, uuid.UUID]):
    # ...
    async def on_after_delete(self, user: User, request: Optional[Request] =
None):
        print(f"User {user.id} is successfully deleted")
```