Валидация

`Валидация` — это процесс проверки входных данных на соответствие определённым правилам или критериям. В программировании валидация используется для того, чтобы убедиться, что данные, которые вводит пользователь или передаются в систему, корректны и соответствуют ожидаемому формату перед тем, как они будут обработаны или сохранены.

Зачем нужна валидация?

- **Предотвращение ошибок**: некорректные данные могут вызвать ошибки при обработке, сохранить неправильную информацию или привести к краху программы.
- **Безопасность**: некорректные данные могут быть использованы для атак на систему, таких как SQL-инъекции или XSS.
- **Улучшение пользовательского опыта**: пользователь получает обратную связь, если его данные не соответствуют ожиданиям, и может исправить их сразу.

Создание валидации

В папке `auth` создадим файл `validators.py` . Добавим следующие импорты:

```
import re
from fastapi import HTTPException, status
from sqlalchemy.ext.asyncio import AsyncSession
```

Но перед продолжением нам понадобится функция, которая будет проверять, существует ли объект в БД. В теории для этого мы могли бы попытаться получить объект из БД, сделать проверку на то, вернулось `None` или нет. Но мы создадим `exists` , он просто проверит, есть ли объект с некими параметрами, но не будет возвращать его, а вернет лишь `True` или `False` соответственно.

В папке `user` создадим файл `utils.py` , а в него добавьте следующий код:

```python
from sqlalchemy.sql import select, exists
from sqlalchemy.ext.asyncio import AsyncSession

from .models import User

async def user_exists_by_username(session: AsyncSession, username: str) -> bool:
    result = await session.execute(select(exists().where(User.username == username))) #where как
    return result.scalar() #Все результаты нужно проводить через scalar
    # Если это список значений, то scalars().all()
    # Также есть возможность сделать scalar_one_or_none() или scalar().first()
```

В целом эта функция похожа на получение значений из таблицы, только там на месте `exists()` обычно пишут название модели. Импортируем её в файл `validators.py` и создадим класс для обработки ошибок валидации

```python
import re
from fastapi import HTTPException, status
from sqlalchemy.ext.asyncio import AsyncSession

from ..user.utils import user_exists_by_username

class ValidateError(Exception):
    def __init__(
        self, detail: str, status_code: int = status.HTTP_400_BAD_REQUEST
    ) -> None:
        self.detail = detail
        self.status_code = status_code
        super().__init__(detail)
```

Этот класс наследуется от `Exception(Исключение)` и принимает `status(Код, возвращаемый API)` и `detail(Подробности возникшей ошибки)`

> **Tip**
>
> Виды статусов
>
> - **1xx** — Информационные коды (Informational responses)
> - **2xx** — Успешные коды (Success)
> - **3xx** — Перенаправления (Redirection)
> - **4xx** — Ошибки клиента (Client errors)

- **5xx** — Ошибки сервера (Server errors)

Давайте же создадим класс для валидации регистрации, в этот же файл добавьте следующий код:

```python
class RegistrationValidator:
    def __init__(
        self,
        username: str,
        password: str,
        confirm_password: str,
        full_name: str,
        session: AsyncSession,
    ) -> None:

        self.username = username
        self.password = password
        self.confirm_password = confirm_password
        self.full_name = full_name
        self.session = session

    async def validate(self):
        try:
            await self.validate_username()
            await self.validate_password()
            await self.validate_full_name()

        except ValidateError as e:
            raise HTTPException(status_code=e.status_code, detail=e.detail)

    async def validate_username(self):
        exists = await user_exists_by_username(self.session, self.username)
        if exists:
            raise ValidateError(
                "A user with this username already exists", status.HTTP_409_CONFLICT
            )
        if not self.username or self.username == "":
            raise ValidateError(
                "Username cannot be empty", status.HTTP_422_UNPROCESSABLE_ENTITY
            )
        if not (4 <= len(self.username) <= 20):
            raise ValidateError(
                "Username must be between 4 and 20 characters long",
                status.HTTP_422_UNPROCESSABLE_ENTITY,
            )
        if not re.match(r"^[A-Za-z0-9 ]+$", self.username):
            raise ValidateError(
                "Username must consist only of English letters, digits, and spaces",
```

```
            status.HTTP_422_UNPROCESSABLE_ENTITY,
        )

async def validate_password(self):
    if not self.password or self.password == "":
        raise ValidateError(
            "Password cannot be empty", status.HTTP_422_UNPROCESSABLE_ENTITY
        )
    if not self.confirm_password or self.confirm_password == "":
        raise ValidateError(
            "Confirm your password", status.HTTP_422_UNPROCESSABLE_ENTITY
        )
    if not (8 <= len(self.password) <= 20):
        raise ValidateError(
            "Password must be between 8 and 20 characters long",
            status.HTTP_422_UNPROCESSABLE_ENTITY,
        )
    if not re.search(r"^[A-Za-z0-9!@#$%&_|?]*$", self.password):
        raise ValidateError(
            "The password must consist only of Latin letters, numbers and the following spec
            status.HTTP_400_BAD_REQUEST,
        )
    if not re.search("[a-z]", self.password):
        raise ValidateError(
            "The password must contain at least one lowercase letter.",
            status.HTTP_422_UNPROCESSABLE_ENTITY,
        )
    if not re.search("[A-Z]", self.password):
        raise ValidateError(
            "The password must contain at least one uppercase letter.",
            status.HTTP_422_UNPROCESSABLE_ENTITY,
        )
    if not re.search("[0-9]", self.password):
        raise ValidateError(
            "The password must contain at least one digit.",
            status.HTTP_422_UNPROCESSABLE_ENTITY,
        )
    if not re.search("[!@#$%&_|?]", self.password):
        raise ValidateError(
            "The password must contain at least one special character.",
            status.HTTP_422_UNPROCESSABLE_ENTITY,
        )
    if self.password != self.confirm_password:
```

```
            raise ValidateError("Passwords do not match", status.HTTP_400_BAD_REQUEST)

    async def validate_full_name(self):
        if not self.full_name or self.full_name == "":
            raise ValidateError(
                "Fullname cannot be empty", status.HTTP_422_UNPROCESSABLE_ENTITY
            )
        if not (15 <= len(self.full_name) <= 50):
            raise ValidateError(
                "Fullname must be between 15 and 50 characters long",
                status.HTTP_422_UNPROCESSABLE_ENTITY,
            )
        if not re.match(r"^[а-яА-ЯёЁ]+\s[а-яА-ЯёЁ]+\s[а-яА-ЯёЁ]+$", self.full_name):
            raise ValidateError(
                "The full name must consist of three words written in Russian letters only",
                status.HTTP_422_UNPROCESSABLE_ENTITY,
            )
```

Tip

Библиотека `re` в Python — это модуль для работы с регулярными выражениями (regular expressions). Регулярные выражения позволяют эффективно искать, сравнивать и заменять текстовые данные, используя шаблоны для поиска подстрок в строках.

Этот класс принимает данные из `UserCreate` и `AsyncSession` и проводит многочисленные проверки над входными данными, в случае возникновения ошибки он возвращает ее через `HTTPException`. Вернемся же к роутеру для регистрации и добавим нашу валидацию

```python
#Прошлые импорты
from .validators import RegistrationValidator

@router.post("/registration", response_class=JSONResponse)
async def create_user(
    user_create: UserCreate, session: AsyncSession = Depends(get_session)
):
    validator = RegistrationValidator(
        user_create.username,
        user_create.password,
        user_create.confirm_password,
        user_create.full_name,
        session,
    )
    await validator.validate()

    user = await qr.registration_user(session, user_create)
    return JSONResponse(
        content=UserResponse(message="Пользователь зарегестрирован", user=user).dict(),
        status_code=status.HTTP_201_CREATED,
    )
```

Теперь при регистрации данные из `UserCreate` будут проходить через валидацию, что отсеет случайные ошибки, конфликты и приведет данные к похожему формату