Реализация различных методов

Создание моделей

Сейчас мы с вами научимся реализовывать основные методы: POST, GET, PUT, DELETE, PATCH, чтобы вы уже сами могли попробовать сделать что - то своё. Создадим в `api` новую папку и назовём её `film` . В ней создадим файл `models.py`

```
#Импорты
from sqlalchemy import Column, Integer, String, ForeignKey, Enum, Table
from sqlalchemy.orm import relationship

from ..user.models import Base, BaseEnum
```

Для реализации фильмов нам необходимо, чтобы у него могло быть множество жанров, для этого мы реализуем связь ManyToMany. Для нее нам понадобится создать смежную таблицу под названием `film_genre` , связывающую `film` и `genre` . Возрастные ограничения реализуем при помощи `Enum` . И не забываем про `relationship` .

```python
class AgeRating(BaseEnum):
    G = "Для всех возрастов"
    PG = "Рекомендуется присутствие родителей"
    PG_13 = "Детям до 13 лет просмотр не желателен"
    R = "Лица до 17 лет допускаются с родителями"
    NC_17 = "Лица до 17 лет не допускаются"
    TV_MA = "Только для взрослых"


film_genre = Table(
    "film_genre",
    Base.metadata,
    Column("film_id", Integer, ForeignKey("films.id"), primary_key=True),
    Column("genre_id", Integer, ForeignKey("genres.id"), primary_key=True),
)


class Film(Base):
    __tablename__ = "films"

    id = Column(Integer, primary_key=True)
    title = Column(String(30), unique=True, nullable=False)
    description = Column(String(200), nullable=False)
    age_rating = Column(Enum(AgeRating), default=AgeRating.PG_13, nullable=False)
    duration = Column(Integer, default=120, nullable=False)
    release_year = Column(Integer, default=2024, nullable=False)

    #secondary указывает на смежную таблицу
    genres = relationship("Genre", secondary="film_genre", back_populates="films")


class Genre(Base):
    __tablename__ = "genres"

    id = Column(Integer, primary_key=True)
    title = Column(String(30), unique=True, nullable=False)

    #secondary указывает на смежную таблицу
    films = relationship("Film", secondary="film_genre", back_populates="genres")
```

> **Warning**

> Не забудьте заменить в migrations/env.py
>
> ```
> #from api.user.models import Base
> from api.film.models import Base
> ```
>
> Теперь можете создавать и применять миграции

Создание схем

В папке `film` создадим 2 файла `schemes` и `validators`. В первом мы создадим схемы для фильмов, а во втором мы создадим функции, для перехода объекта модели к этой схеме, ведь мы их будем использоваться дляч валидации данных в роутерах.

```python
from pydantic import BaseModel, Field
from typing import List, Optional, Union #Различные типы данных

from .models import AgeRating


class GenreCreate(BaseModel):
    title: str = Field(..., title="Название жанра", max_length=30)


class GenreResponse(BaseModel):
    id: int
    title: str


class FilmCreate(BaseModel):
    title: str = Field(..., min_length=1, max_length=30)
    description: str = Field(..., max_length=200)
    age_rating: AgeRating = AgeRating.PG_13
    duration: int = Field(..., ge=120, description="Duration in minutes")
    release_year: int = Field(..., ge=1888, description="Year of film release")
    genre_ids: List[int]

#Схема для обновления объекта
class FilmUpdate(BaseModel):
    title: Optional[str] = None
    description: Optional[str] = None
    age_rating: Optional[AgeRating] = None
    duration: Optional[int] = None
    release_year: Optional[int] = None
    genre_ids: Optional[List[int]] = None


class FilmResponse(BaseModel):
    id: int
    title: str
    description: str
    age_rating: Union[AgeRating, str] #Означает Или AgeRating, или str
    duration: int
    release_year: int
    genres: Optional[List[str]] = []
```

Тут в схеме `FilmResponse` мы отошли от прошлого концепта в UserResponse, который возвращал сообщение, вместо этого мы будем возвращать просто объект. Но как вы могли заметить `genres` у нас список из `str` , а не `Enum` , этот переход мы реализуем в функции, которую будем использовать везде, по этому проблем с валидацией данных не будет

Создание валидации схем

Теперь в уже созданном файле `validators.py` вставьте следующие функции. Они будут приводить объекты к нашим Pydantic моделям, т.е схемам и только после этого мы будем возвращать их

```
from typing import List
from .models import Film
from .schemes import FilmResponse


async def film_to_pydantic(film: Film) -> FilmResponse:
    return FilmResponse(
        id=film.id,
        title=film.title,
        description=film.description,
        age_rating=film.age_rating,
        duration=film.duration,
        release_year=film.release_year,
        genres=[genre.title for genre in film.genres],
    )

async def list_film_to_pydantic(films: List[Film]) -> List[FilmResponse]:
    return [await film_to_pydantic(film) for film in films]
```

Создание queries

Создадим в папке film файл queries.py и сразу реализуем все функции, которые нам могут понадобиться в роутере

```python
from typing import List
from fastapi import HTTPException, status
from sqlalchemy import select
from sqlalchemy.orm import selectinload
from sqlalchemy.ext.asyncio import AsyncSession

from .models import Film, Genre
from .schemes import GenreCreate, GenreResponse, FilmCreate, FilmUpdate
from .utils import existing_film_by_title

#Создание жанра
async def create_genre(
    session: AsyncSession, genre_create: GenreCreate
) -> GenreResponse:
    #Проверка на наличие в БД
    existing_genre = await session.execute(
        select(Genre).where(Genre.title == genre_create.title)
    )

    if existing_genre.scalar() is not None:
        raise HTTPException(
            status.HTTP_409_CONFLICT,
            "Жанр с таким названием уже существует",
        )

    new_genre = Genre(title=genre_create.title)
    session.add(new_genre)
    await session.commit()
    await session.refresh(new_genre)

    return GenreResponse(id=new_genre.id, title=new_genre.title)

#Получение жанров
async def get_all_genres(session: AsyncSession) -> List[Genre]:
    result = await session.execute(select(Genre))
    genres = result.scalars().all()
    if not genres:
        raise HTTPException(status.HTTP_204_NO_CONTENT)
    return genres

#Получение жанра по id
async def get_genre_by_id(session: AsyncSession, genre_id: int) -> Genre:
    result = await session.execute(select(Genre).where(Genre.id == genre_id))
```

```
    genre = result.scalar_one_or_none()
    if not genre:
        raise HTTPException(status.HTTP_404_NOT_FOUND, "Жанр не найден")
    return genre

#Получение жанра по title
async def get_genre_by_title(session: AsyncSession, genre_title: str) -> Genre:
    result = await session.execute(select(Genre).where(Genre.title == genre_title))
    genre = result.scalar_one_or_none()
    if not genre:
        raise HTTPException(status.HTTP_404_NOT_FOUND, "Жанр не найден")
    return genre

#Обновление жанра
async def update_genre(
    session: AsyncSession, genre_id: int, genre_update: GenreCreate
) -> Genre:
    genre = await get_genre_by_id(session, genre_id)
    if not genre:
        raise HTTPException(status.HTTP_404_NOT_FOUND, "Жанр не найден")

    genre.title = genre_update.title
    await session.commit()
    await session.refresh(genre)

    return genre

#Удаление жанра
async def delete_genre(session: AsyncSession, genre_id: int) -> None:
    genre = await session.get(Genre, genre_id)
    if not genre:
        raise HTTPException(status.HTTP_404_NOT_FOUND, "Жанр не найден")

    await session.delete(genre)
    await session.commit()

#Создание фильма
async def create_film(session: AsyncSession, film_create: FilmCreate) -> Film:
    existing_film = await existing_film_by_title(session, film_create.title)

    if existing_film:
        raise HTTPException(
            status.HTTP_409_CONFLICT, "Фильм с таким названием уже существует"
```

```
        )

    new_film = Film(
        title=film_create.title,
        description=film_create.description,
        age_rating=film_create.age_rating,
        duration=film_create.duration,
        release_year=film_create.release_year,
    )

    genres = await session.execute(
        select(Genre).where(Genre.id.in_(film_create.genre_ids))
    )
    genre_list = genres.scalars().all()

    if len(genre_list) != len(film_create.genre_ids):
        raise HTTPException(
            status.HTTP_404_NOT_FOUND,
            "Один или несколько жанров не найдены",
        )

    new_film.genres.extend(genre_list)
    session.add(new_film)
    await session.commit()

    return new_film

#Получение фильмов
async def get_all_films(session: AsyncSession) -> List[Film]:
    result = await session.execute(select(Film).options(selectinload(Film.genres)))
    films = result.scalars().all()
    if not films:
        raise HTTPException(status.HTTP_204_NO_CONTENT)

    return films

#Получение фильма по id
async def get_film_by_id(session: AsyncSession, film_id: int) -> Film:
    result = await session.execute(
        select(Film).where(Film.id == film_id).options(selectinload(Film.genres))
    )
    film = result.scalar_one_or_none()
    if not film:
```

```
        raise HTTPException(status.HTTP_404_NOT_FOUND, "Фильм не найден")
    return film

#Получение фильма по title
async def get_film_by_title(session: AsyncSession, film_title: str) -> Film:
    result = await session.execute(
        select(Film).where(Film.title == film_title).options(selectinload(Film.genres))
    )
    film = result.scalar_one_or_none()
    if not film:
        raise HTTPException(status.HTTP_404_NOT_FOUND, "Фильм не найден")
    return film

#Обновление фильма
async def update_film(
    session: AsyncSession, film_id: int, film_update: FilmUpdate
) -> Film:
    film = await get_film_by_id(session, film_id)
    if not film:
        raise HTTPException(status.HTTP_404_NOT_FOUND, "Фильм не найден")

    #Обновление обновляются только те поля, которые не Null
    #Т.е когда в Swagger вводите данные, не нужные строки просто удаляйте
    if film_update.title is not None:
        film.title = film_update.title
    if film_update.description is not None:
        film.description = film_update.description
    if film_update.age_rating is not None:
        film.age_rating = film_update.age_rating
    if film_update.duration is not None:
        film.duration = film_update.duration
    if film_update.release_year is not None:
        film.release_year = film_update.release_year

    if film_update.genre_ids is not None:
        genres = await session.execute(
            select(Genre).where(Genre.id.in_(film_update.genre_ids))
        )
        genre_list = genres.scalars().all()

        if len(genre_list) != len(film_update.genre_ids):
            raise HTTPException(
                status.HTTP_404_NOT_FOUND, "Один или несколько жанров не найдены"
```

```
            )

        film.genres.clear()
        film.genres.extend(genre_list)

    await session.commit()
    await session.refresh(film)

    print(film.age_rating)

    return film

#Удаление фильма
async def delete_film(session: AsyncSession, film_id: int) -> None:
    film = await session.get(Film, film_id)
    if not film:
        raise HTTPException(status.HTTP_404_NOT_FOUND, "Фильм не найден")

    await session.delete(film)
    await session.commit()
```

> **Warning**
>
> Также нам необходимо реализовать `existing_film_by_title` , для этого создадим файл `utils` в папке `film` . Мы сразу на будущее создадим `existing_film_by_id`

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

from .models import Film


async def existing_film_by_title(session: AsyncSession, title: str) -> bool:
    result = await session.execute(select(exists().where(Film.title == title)))
    return result.scalar()


async def existing_film_by_id(session: AsyncSession, id: int) -> None:
    result = await session.execute(select(exists().where(Film.id == id)))
    film_exists = result.scalar()
    if not film_exists:
        raise HTTPException(status.HTTP_404_NOT_FOUND, "Фильм не найден")
```

Создание роутеров

Warning

Перед созданием роутера нам необходимо реализовать функционал получения current_user(Текущий пользователь) для ограничения не авторизованных и провекри ролей.

В файле auth/queries.py добавим следующие функции

```python
#Получение объекта токена по токену
async def get_token(session: AsyncSession, token: str) -> Token:
result = await session.execute(select(Token).where(Token.token == token))

return result.scalar_one_or_none()

#Верификация токена
async def verify_token_and_get_user(session: AsyncSession, token: str) -> User:
    token = await get_token(session, token)

    if token is None:
        raise HTTPException(status.HTTP_401_UNAUTHORIZED, "Токен не найден")

    await token.verify_token(session, None)
    user = await get_user_by_id(session, token.user_id)

    return user

#Получение текущего пользователя
async def get_current_user(
    session: AsyncSession = Depends(get_session),
    token: str = Depends(OAuth2PasswordBearer(tokenUrl="/auth/login")),
) -> User:
    print(token)
    user = await verify_token_and_get_user(session, token)
    return user
```

Для проверки ролей в файле user/utils.py добавим

```python
from .models import User, Role

#Проверка доступа только для админов
async def admin_check(user: User):
if user.role is not Role.ADMIN:
    raise HTTPException(
        status.HTTP_403_FORBIDDEN,
        "Только администратор имет доступ к этому ендпоинту",
    )

#Проверка ограничения пользователя
async def cashier_check(user: User):
    if user.role is Role.USER:
        raise HTTPException(
            status.HTTP_403_FORBIDDEN,
            "Пользователи не имеют достпуп к этому ендпоинту",
        )
```

Теперь в папке `film` создадим файл `router.py`

№Импорты

```python
from typing import List, Optional, Union
from fastapi import Depends, APIRouter, status
from sqlalchemy.ext.asyncio import AsyncSession

from ..database import get_session
from ..auth.queries import get_current_user
from ..user.models import User
from ..user import utils as ut

from .schemes import GenreCreate, GenreResponse, FilmCreate, FilmUpdate, FilmResponse
from . import queries as qr
from . import validators as validator

router = APIRouter(prefix="/films")

#Создание жанра
@router.post("/genres/", response_model=GenreResponse, status_code=status.HTTP_201_CREATED)
async def create_genre(
    genre_create: GenreCreate,
    session: AsyncSession = Depends(get_session),
    current_user: User = Depends(get_current_user),
):
    await ut.admin_check(current_user)
    return await qr.create_genre(session, genre_create)

#Получение жанров (Если title не none, то получит по title)
@router.get("/genres/", response_model=Union[List[GenreResponse], GenreResponse])
async def get_genres_or_genre(
    title: Optional[str] = None,
    session: AsyncSession = Depends(get_session),
    # current_user: User = Depends(get_current_user)
):
    if not title:
        genres = await qr.get_all_genres(session)
        return genres

    genre = await qr.get_genre_by_title(session, title)
    return genre

#Создание жанра по id
@router.get("/genres/{genre_id}", response_model=GenreResponse)
async def get_genre_by_id(
```

```
    genre_id: int,
    session: AsyncSession = Depends(get_session),
    # current_user: User = Depends(get_current_user)
):
    genre = await qr.get_genre_by_id(session, genre_id)
    return genre

#Изменение жанра
@router.put("/genres/{genre_id}", response_model=GenreResponse)
async def update_genre(
    genre_id: int,
    genre_update: GenreCreate,
    session: AsyncSession = Depends(get_session),
    current_user: User = Depends(get_current_user),
):
    await ut.admin_check(current_user)
    updated_genre = await qr.update_genre(session, genre_id, genre_update)
    return updated_genre

#Удаление жанра
@router.delete("/genres/{genre_id}", status_code=status.HTTP_200_OK)
async def delete_genre(
    genre_id: int,
    session: AsyncSession = Depends(get_session),
    current_user: User = Depends(get_current_user),
):
    await ut.admin_check(current_user)
    await qr.delete_genre(session, genre_id)
    return "Жанр удалён"

#Создание фильма
@router.post("/", response_model=FilmResponse, status_code=status.HTTP_201_CREATED)
async def create_film(
    film_create: FilmCreate,
    session: AsyncSession = Depends(get_session),
    current_user: User = Depends(get_current_user),
):
    await ut.admin_check(current_user)
    created_film = await qr.create_film(session, film_create)
    return await validator.film_to_pydantic(created_film)

#Получение фильмов (Также как и у жанров)
@router.get("/", response_model=Union[List[FilmResponse], FilmResponse])
```

```python
async def get_films_or_film(
    title: Optional[str] = None,
    session: AsyncSession = Depends(get_session),
    # current_user: User = Depends(get_current_user)
):
    if not title:
        films = await qr.get_all_films(session)
        return await validator.list_film_to_pydantic(films)

    film = await qr.get_film_by_title(session, title)
    return await validator.film_to_pydantic(film)

#Получение фильма по id
@router.get("/{film_id}", response_model=FilmResponse)
async def get_film_by_id(
    film_id: int,
    session: AsyncSession = Depends(get_session),
    # current_user: User = Depends(get_current_user)
):
    film = await qr.get_film_by_id(session, film_id)
    return await validator.film_to_pydantic(film)

#Изменение фильма (Тут patch, м.к изменяется выборочно, put изменяет весь объект)
@router.patch("/films/{film_id}")
async def update_film(
    film_id: int,
    film_update: FilmUpdate,
    session: AsyncSession = Depends(get_session),
    current_user: User = Depends(get_current_user),
):
    await ut.admin_check(current_user)
    updated_film = await qr.update_film(session, film_id, film_update)
    return await validator.film_to_pydantic(updated_film)

#Удаление фильма
@router.delete("/{film_id}", status_code=status.HTTP_200_OK)
async def delete_film(
    film_id: int,
    session: AsyncSession = Depends(get_session),
    current_user: User = Depends(get_current_user),
):
    await ut.admin_check(current_user)
```

```
    await qr.delete_film(session, film_id)
    return "Фильм удалён"
```

Оставлось только импортировать роутер в проложение в файле `main.py` и можно тестировать добавленные эндпоинты. На этом конец, дальше вы сами сможете доделать апи или создать что - то свое. Всем удачи.