# Список Python-файлов

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\main.py

from fastapi import FastAPI  
from app.api.routers import routers # Предполагается, что это ваш роутер  
from app.core.models import Base  
from app.core.database.helper import engine, SessionLocal  
from app.core.models.city import City  
from app.core.models.category import Category  
from sqlalchemy.orm import Session  
import logging  
  
app = FastAPI()  
  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(\_\_name\_\_)  
  
def init\_db():  
 Base.metadata.create\_all(bind=engine)  
 with SessionLocal() as session:  
 if not session.query(City).first():  
 default\_city = City(name="Кокшетау")  
 session.add(default\_city)  
 session.commit()  
 logger.info("Добавлен город по умолчанию: Кокшетау")  
 if not session.query(Category).first():  
 default\_category = Category(name="Общие услуги")  
 session.add(default\_category)  
 session.commit()  
 logger.info("Добавлена категория по умолчанию: Общие услуги")  
  
@app.on\_event("startup")  
async def startup():  
 init\_db()  
 logger.info("API запущен, роутеры подключены")  
  
# Проверяем подключение роутера  
logger.info(f"Подключение роутера: {routers}")  
app.include\_router(routers)  
  
# Для отладки: выводим все маршруты  
@app.get("/debug/routes")  
async def debug\_routes():  
 routes = [{"path": route.path, "methods": list(route.methods)} for route in app.routes]  
 logger.info(f"Зарегистрированные маршруты: {routes}")  
 return routes

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\alembic\env.py

from logging.config import fileConfig  
from sqlalchemy import engine\_from\_config, pool  
from alembic import context  
from app.core.config import DB\_URL  
from app.core.models.base import Base  
from app.core.models.user import User  
from app.core.models.association import user\_categories  
from app.core.models.category import Category  
from app.core.models.city import City  
from app.core.models.offer import Offer  
from app.core.models.order import Order  
from app.core.models.review import Review  
  
# Настраиваем логирование  
if context.config.config\_file\_name is not None:  
 fileConfig(context.config.config\_file\_name)  
  
# Указываем метаданные для автогенерации  
target\_metadata = Base.metadata  
  
def run\_migrations\_offline() -> None:  
 """Запуск миграций в оффлайн-режиме."""  
 url = DB\_URL # Используем DB\_URL из config.py  
 context.configure(  
 url=url,  
 target\_metadata=target\_metadata,  
 literal\_binds=True,  
 dialect\_opts={"paramstyle": "named"},  
 )  
 with context.begin\_transaction():  
 context.run\_migrations()  
  
def run\_migrations\_online() -> None:  
 """Запуск миграций в онлайн-режиме."""  
 connectable = engine\_from\_config(  
 {"sqlalchemy.url": DB\_URL}, # Используем DB\_URL напрямую  
 prefix="sqlalchemy.",  
 poolclass=pool.NullPool,  
 )  
 with connectable.connect() as connection:  
 context.configure(  
 connection=connection,  
 target\_metadata=target\_metadata,  
 )  
 with context.begin\_transaction():  
 context.run\_migrations()  
  
if context.is\_offline\_mode():  
 run\_migrations\_offline()  
else:  
 run\_migrations\_online() # Добавлен вызов функции с правильным отступом

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\alembic\versions\02487aa00d87\_add\_start\_date\_to\_offer.py

"""Add start\_date to Offer  
  
Revision ID: 02487aa00d87  
Revises: ec634b458093  
Create Date: 2025-03-12 12:31:35.912972  
  
"""  
from typing import Sequence, Union  
  
from alembic import op  
import sqlalchemy as sa  
  
  
# revision identifiers, used by Alembic.  
revision: str = '02487aa00d87'  
down\_revision: Union[str, None] = 'ec634b458093'  
branch\_labels: Union[str, Sequence[str], None] = None  
depends\_on: Union[str, Sequence[str], None] = None  
  
  
def upgrade() -> None:  
 """Upgrade schema."""  
 # ### commands auto generated by Alembic - please adjust! ###  
 op.add\_column('offers', sa.Column('start\_date', sa.DateTime(), nullable=True))  
 # ### end Alembic commands ###  
  
  
def downgrade() -> None:  
 """Downgrade schema."""  
 # ### commands auto generated by Alembic - please adjust! ###  
 op.drop\_column('offers', 'start\_date')  
 # ### end Alembic commands ###

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\alembic\versions\2025\_03\_15\_2145-4b861d6e9719\_add\_status\_to\_orders\_and\_remove\_is\_.py

"""add status to orders and remove is\_completed  
  
Revision ID: 4b861d6e9719  
Revises: 02487aa00d87  
Create Date: 2025-03-15 21:45:30.776605  
"""  
  
from typing import Sequence, Union  
  
from alembic import op  
import sqlalchemy as sa  
  
# revision identifiers, used by Alembic.  
revision: str = "4b861d6e9719"  
down\_revision: Union[str, None] = "02487aa00d87"  
branch\_labels: Union[str, Sequence[str], None] = None  
depends\_on: Union[str, Sequence[str], None] = None  
  
# Определение ENUM-типа с значениями в верхнем регистре  
orderstatus = sa.Enum("PENDING", "IN\_PROGRESS", "COMPLETED", "CANCELED", name="orderstatus")  
  
def upgrade() -> None:  
 """Upgrade schema."""  
 # Создание типа orderstatus перед добавлением колонки  
 orderstatus.create(op.get\_bind(), checkfirst=True)  
  
 # Добавление колонки status с типом orderstatus и значением по умолчанию  
 op.add\_column(  
 "orders",  
 sa.Column(  
 "status",  
 orderstatus,  
 nullable=False,  
 server\_default="PENDING" # Значение по умолчанию в верхнем регистре  
 ),  
 )  
  
 # Удаление колонки is\_completed  
 op.drop\_column("orders", "is\_completed")  
  
def downgrade() -> None:  
 """Downgrade schema."""  
 # Добавление обратно колонки is\_completed  
 op.add\_column(  
 "orders",  
 sa.Column(  
 "is\_completed", sa.BOOLEAN(), autoincrement=False, nullable=False  
 ),  
 )  
  
 # Удаление колонки status  
 op.drop\_column("orders", "status")  
  
 # Удаление типа orderstatus  
 orderstatus.drop(op.get\_bind(), checkfirst=True)

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\alembic\versions\2025\_03\_15\_2214-92fdddd0c03c\_create\_otder\_nullable\_true.py

"""Create otder nullable true  
  
Revision ID: 92fdddd0c03c  
Revises: 4b861d6e9719  
Create Date: 2025-03-15 22:14:44.580445  
  
"""  
  
from typing import Sequence, Union  
  
from alembic import op  
import sqlalchemy as sa  
from sqlalchemy.dialects import postgresql  
  
# revision identifiers, used by Alembic.  
revision: str = "92fdddd0c03c"  
down\_revision: Union[str, None] = "4b861d6e9719"  
branch\_labels: Union[str, Sequence[str], None] = None  
depends\_on: Union[str, Sequence[str], None] = None  
  
  
def upgrade() -> None:  
 """Upgrade schema."""  
 # ### commands auto generated by Alembic - please adjust! ###  
 op.alter\_column(  
 "orders",  
 "status",  
 existing\_type=postgresql.ENUM(  
 "pending",  
 "in\_progress",  
 "completed",  
 "canceled",  
 name="orderstatus",  
 ),  
 nullable=True,  
 existing\_server\_default=sa.text("'pending'::orderstatus"),  
 )  
 # ### end Alembic commands ###  
  
  
def downgrade() -> None:  
 """Downgrade schema."""  
 # ### commands auto generated by Alembic - please adjust! ###  
 op.alter\_column(  
 "orders",  
 "status",  
 existing\_type=postgresql.ENUM(  
 "pending",  
 "in\_progress",  
 "completed",  
 "canceled",  
 name="orderstatus",  
 ),  
 nullable=False,  
 existing\_server\_default=sa.text("'pending'::orderstatus"),  
 )  
 # ### end Alembic commands ###

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\alembic\versions\2025\_03\_15\_2254-b343e50c4c1a\_еще\_одна\_попытка\_перемиграции\_после\_.py

"""Еще одна попытка перемиграции после изменений  
  
Revision ID: b343e50c4c1a  
Revises: 92fdddd0c03c  
Create Date: 2025-03-15 22:54:41.125060  
  
"""  
  
from typing import Sequence, Union  
  
from alembic import op  
import sqlalchemy as sa  
  
  
# revision identifiers, used by Alembic.  
revision: str = "b343e50c4c1a"  
down\_revision: Union[str, None] = "92fdddd0c03c"  
branch\_labels: Union[str, Sequence[str], None] = None  
depends\_on: Union[str, Sequence[str], None] = None  
  
  
def upgrade() -> None:  
 """Upgrade schema."""  
 # ### commands auto generated by Alembic - please adjust! ###  
 pass  
 # ### end Alembic commands ###  
  
  
def downgrade() -> None:  
 """Downgrade schema."""  
 # ### commands auto generated by Alembic - please adjust! ###  
 pass  
 # ### end Alembic commands ###

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\alembic\versions\2025\_03\_15\_2327-0ed9e05ec864\_.py

"""empty message  
  
Revision ID: 0ed9e05ec864  
Revises: b343e50c4c1a  
Create Date: 2025-03-15 23:27:28.254177  
  
"""  
  
from typing import Sequence, Union  
  
from alembic import op  
import sqlalchemy as sa  
  
  
# revision identifiers, used by Alembic.  
revision: str = "0ed9e05ec864"  
down\_revision: Union[str, None] = "b343e50c4c1a"  
branch\_labels: Union[str, Sequence[str], None] = None  
depends\_on: Union[str, Sequence[str], None] = None  
  
  
def upgrade() -> None:  
 """Upgrade schema."""  
 pass  
  
  
def downgrade() -> None:  
 """Downgrade schema."""  
 pass

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\alembic\versions\2025\_03\_15\_2328-423455fd3281\_изменены\_статусы\_на\_русский\_язык.py

"""Изменены статусы на русский язык  
  
Revision ID: 423455fd3281  
Revises: 0ed9e05ec864  
Create Date: 2025-03-15 23:28:31.488773  
  
"""  
  
from typing import Sequence, Union  
  
from alembic import op  
import sqlalchemy as sa  
  
  
# revision identifiers, used by Alembic.  
revision: str = "423455fd3281"  
down\_revision: Union[str, None] = "0ed9e05ec864"  
branch\_labels: Union[str, Sequence[str], None] = None  
depends\_on: Union[str, Sequence[str], None] = None  
  
  
def upgrade() -> None:  
 """Upgrade schema."""  
 pass  
  
  
def downgrade() -> None:  
 """Downgrade schema."""  
 pass

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\alembic\versions\ec634b458093\_initial\_migration.py

"""Initial migration  
  
Revision ID: ec634b458093  
Revises:   
Create Date: 2025-03-10 07:46:03.885036  
  
"""  
from typing import Sequence, Union  
  
from alembic import op  
import sqlalchemy as sa  
  
  
# revision identifiers, used by Alembic.  
revision: str = 'ec634b458093'  
down\_revision: Union[str, None] = None  
branch\_labels: Union[str, Sequence[str], None] = None  
depends\_on: Union[str, Sequence[str], None] = None  
  
  
def upgrade() -> None:  
 """Upgrade schema."""  
 # ### commands auto generated by Alembic - please adjust! ###  
 op.create\_table('categories',  
 sa.Column('name', sa.String(), nullable=False),  
 sa.Column('id', sa.Integer(), nullable=False),  
 sa.PrimaryKeyConstraint('id'),  
 sa.UniqueConstraint('name')  
 )  
 op.create\_index(op.f('ix\_categories\_id'), 'categories', ['id'], unique=False)  
 op.create\_table('cities',  
 sa.Column('name', sa.String(), nullable=False),  
 sa.Column('id', sa.Integer(), nullable=False),  
 sa.PrimaryKeyConstraint('id'),  
 sa.UniqueConstraint('name')  
 )  
 op.create\_index(op.f('ix\_cities\_id'), 'cities', ['id'], unique=False)  
 op.create\_table('users',  
 sa.Column('telegram\_id', sa.Integer(), nullable=False),  
 sa.Column('name', sa.String(), nullable=False),  
 sa.Column('username', sa.String(), nullable=True),  
 sa.Column('is\_customer', sa.Boolean(), nullable=False),  
 sa.Column('is\_executor', sa.Boolean(), nullable=False),  
 sa.Column('is\_admin', sa.Boolean(), nullable=False),  
 sa.Column('city\_id', sa.Integer(), nullable=False),  
 sa.Column('rating', sa.Numeric(precision=2, scale=1), nullable=False),  
 sa.Column('completed\_orders', sa.Integer(), nullable=False),  
 sa.Column('id', sa.Integer(), nullable=False),  
 sa.ForeignKeyConstraint(['city\_id'], ['cities.id'], ondelete='CASCADE'),  
 sa.PrimaryKeyConstraint('id'),  
 sa.UniqueConstraint('telegram\_id'),  
 sa.UniqueConstraint('username')  
 )  
 op.create\_index(op.f('ix\_users\_id'), 'users', ['id'], unique=False)  
 op.create\_table('orders',  
 sa.Column('customer\_id', sa.Integer(), nullable=False),  
 sa.Column('executor\_id', sa.Integer(), nullable=True),  
 sa.Column('category\_id', sa.Integer(), nullable=False),  
 sa.Column('title', sa.String(), nullable=False),  
 sa.Column('description', sa.String(), nullable=True),  
 sa.Column('desired\_price', sa.Numeric(precision=10, scale=2), nullable=False),  
 sa.Column('due\_date', sa.DateTime(), nullable=False),  
 sa.Column('created\_at', sa.DateTime(), nullable=False),  
 sa.Column('is\_completed', sa.Boolean(), nullable=False),  
 sa.Column('id', sa.Integer(), nullable=False),  
 sa.ForeignKeyConstraint(['category\_id'], ['categories.id'], ondelete='CASCADE'),  
 sa.ForeignKeyConstraint(['customer\_id'], ['users.id'], ondelete='CASCADE'),  
 sa.ForeignKeyConstraint(['executor\_id'], ['users.id'], ondelete='SET NULL'),  
 sa.PrimaryKeyConstraint('id')  
 )  
 op.create\_index(op.f('ix\_orders\_id'), 'orders', ['id'], unique=False)  
 op.create\_table('user\_categories',  
 sa.Column('user\_id', sa.Integer(), nullable=False),  
 sa.Column('category\_id', sa.Integer(), nullable=False),  
 sa.ForeignKeyConstraint(['category\_id'], ['categories.id'], ondelete='CASCADE'),  
 sa.ForeignKeyConstraint(['user\_id'], ['users.id'], ondelete='CASCADE'),  
 sa.PrimaryKeyConstraint('user\_id', 'category\_id')  
 )  
 op.create\_table('offers',  
 sa.Column('order\_id', sa.Integer(), nullable=False),  
 sa.Column('executor\_id', sa.Integer(), nullable=False),  
 sa.Column('price', sa.Numeric(precision=10, scale=2), nullable=False),  
 sa.Column('estimated\_time', sa.Integer(), nullable=False),  
 sa.Column('status', sa.Enum('PENDING', 'ACCEPTED', 'REJECTED', name='offerstatus'), nullable=False),  
 sa.Column('created\_at', sa.DateTime(), nullable=False),  
 sa.Column('id', sa.Integer(), nullable=False),  
 sa.ForeignKeyConstraint(['executor\_id'], ['users.id'], ondelete='CASCADE'),  
 sa.ForeignKeyConstraint(['order\_id'], ['orders.id'], ondelete='CASCADE'),  
 sa.PrimaryKeyConstraint('id')  
 )  
 op.create\_index(op.f('ix\_offers\_id'), 'offers', ['id'], unique=False)  
 op.create\_table('reviews',  
 sa.Column('order\_id', sa.Integer(), nullable=False),  
 sa.Column('author\_id', sa.Integer(), nullable=False),  
 sa.Column('target\_id', sa.Integer(), nullable=False),  
 sa.Column('rating', sa.Integer(), nullable=False),  
 sa.Column('comment', sa.String(), nullable=True),  
 sa.Column('created\_at', sa.DateTime(), nullable=False),  
 sa.Column('id', sa.Integer(), nullable=False),  
 sa.ForeignKeyConstraint(['author\_id'], ['users.id'], ondelete='CASCADE'),  
 sa.ForeignKeyConstraint(['order\_id'], ['orders.id'], ondelete='CASCADE'),  
 sa.ForeignKeyConstraint(['target\_id'], ['users.id'], ondelete='CASCADE'),  
 sa.PrimaryKeyConstraint('id')  
 )  
 op.create\_index(op.f('ix\_reviews\_id'), 'reviews', ['id'], unique=False)  
 # ### end Alembic commands ###  
  
  
def downgrade() -> None:  
 """Downgrade schema."""  
 # ### commands auto generated by Alembic - please adjust! ###  
 op.drop\_index(op.f('ix\_reviews\_id'), table\_name='reviews')  
 op.drop\_table('reviews')  
 op.drop\_index(op.f('ix\_offers\_id'), table\_name='offers')  
 op.drop\_table('offers')  
 op.drop\_table('user\_categories')  
 op.drop\_index(op.f('ix\_orders\_id'), table\_name='orders')  
 op.drop\_table('orders')  
 op.drop\_index(op.f('ix\_users\_id'), table\_name='users')  
 op.drop\_table('users')  
 op.drop\_index(op.f('ix\_cities\_id'), table\_name='cities')  
 op.drop\_table('cities')  
 op.drop\_index(op.f('ix\_categories\_id'), table\_name='categories')  
 op.drop\_table('categories')  
 # ### end Alembic commands ###

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\api\category.py

from typing import Annotated, List  
from fastapi import APIRouter, Depends, status, HTTPException  
from sqlalchemy.orm import Session  
from app.core.database.helper import get\_session  
from app.core.models.user import User  
from app.core.services import category as category\_service  
from app.core.schemas.category import CategoryRead, CategoryCreate, CategoryUpdate  
from app.api.depends.user import get\_current\_user, get\_admin\_user  
  
router = APIRouter(prefix="/category", tags=["Category"]) # Маршруты для категорий  
  
@router.get("/", response\_model=List[CategoryRead])  
def get\_categories(session: Annotated[Session, Depends(get\_session)]):  
 """Получить список всех категорий."""  
 return category\_service.get\_all\_categories(session)  
  
@router.get("/{id}", response\_model=CategoryRead)  
def get\_category(  
 id: int,  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Получить категорию по ID."""  
 return category\_service.get\_category\_by\_id(session, id)  
  
@router.post("/", response\_model=CategoryRead, status\_code=status.HTTP\_201\_CREATED)  
def create\_category(  
 data: CategoryCreate,  
 admin: Annotated[User, Depends(get\_admin\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Создать новую категорию (доступно только администратору)."""  
 return category\_service.create\_category(session, data)  
  
@router.patch("/{id}", response\_model=CategoryRead)  
def update\_category(  
 id: int,  
 data: CategoryUpdate,  
 admin: Annotated[User, Depends(get\_admin\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Обновить данные категории (доступно только администратору)."""  
 return category\_service.update\_category\_by\_id(session, data, id)  
  
@router.delete("/{id}", status\_code=status.HTTP\_204\_NO\_CONTENT)  
def delete\_category(  
 id: int,  
 admin: Annotated[User, Depends(get\_admin\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Удалить категорию (доступно только администратору)."""  
 category\_service.delete\_category\_by\_id(session, id)

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\api\city.py

from typing import Annotated, List  
from fastapi import APIRouter, Depends, status, HTTPException  
from sqlalchemy.orm import Session  
from app.core.database.helper import get\_session  
from app.core.models.user import User  
from app.core.services import city as city\_service  
from app.core.schemas.city import CityRead, CityCreate, CityUpdate  
from app.api.depends.user import get\_current\_user, get\_admin\_user  
  
router = APIRouter(prefix="/city", tags=["City"]) # Маршруты для городов  
  
@router.get("/", response\_model=List[CityRead])  
def get\_cities(session: Annotated[Session, Depends(get\_session)]):  
 """Получить список всех городов."""  
 return city\_service.get\_all\_cities(session)  
  
@router.get("/{id}", response\_model=CityRead)  
def get\_city(  
 id: int,  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Получить город по ID."""  
 return city\_service.get\_city\_by\_id(session, id)  
  
@router.post("/", response\_model=CityRead, status\_code=status.HTTP\_201\_CREATED)  
def create\_city(  
 data: CityCreate,  
 admin: Annotated[User, Depends(get\_admin\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Создать новый город (доступно только администратору)."""  
 return city\_service.create\_city(session, data)  
  
@router.patch("/{id}", response\_model=CityRead)  
def update\_city(  
 id: int,  
 data: CityUpdate,  
 admin: Annotated[User, Depends(get\_admin\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Обновить данные города (доступно только администратору)."""  
 return city\_service.update\_city\_by\_id(session, data, id)  
  
@router.delete("/{id}", status\_code=status.HTTP\_204\_NO\_CONTENT)  
def delete\_city(  
 id: int,  
 admin: Annotated[User, Depends(get\_admin\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Удалить город (доступно только администратору)."""  
 city\_service.delete\_city\_by\_id(session, id)

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\api\offer.py

from typing import Annotated, List  
from fastapi import APIRouter, Depends, status, HTTPException  
from sqlalchemy.orm import Session  
from app.core.database.helper import get\_session  
from app.core.models import Order  
from app.core.models.user import User  
from app.core.schemas.order import OrderUpdate, OrderRead  
from app.core.services import offer as offer\_service  
from app.core.schemas.offer import OfferRead, OfferCreate, OfferUpdate  
from app.api.depends.user import get\_current\_user  
import aiohttp  
from app.bot.config import BOT\_TOKEN # Импортируем токен бота  
  
router = APIRouter(prefix="/offer", tags=["Offer"])  
  
  
async def send\_telegram\_message(chat\_id: int, text: str):  
 """Отправка сообщения через Telegram API."""  
 url = f"https://api.telegram.org/bot{BOT\_TOKEN}/sendMessage"  
 payload = {"chat\_id": chat\_id, "text": text}  
 async with aiohttp.ClientSession() as session:  
 async with session.post(url, json=payload) as response:  
 if response.status != 200:  
 raise Exception(f"Ошибка Telegram API: {await response.text()}")  
  
@router.post("/{id}/offers/{offer\_id}/accept", response\_model=OrderRead)  
async def accept\_offer(  
 id: int,  
 offer\_id: int,  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Принять предложение и назначить исполнителя (доступно только заказчику)."""  
 order = order\_service.get\_order\_by\_id(session, id)  
 if order.customer\_id != current\_user.id:  
 raise HTTPException(status\_code=403, detail="Только заказчик может принять предложение")  
 if order.status != "В\_ожидании": # Проверка на русский статус  
 raise HTTPException(status\_code=400, detail="Нельзя принять предложение для заказа не в статусе 'В\_ожидании'")  
  
 offer = offer\_service.get\_offer\_by\_id(session, offer\_id)  
 if offer.order\_id != id:  
 raise HTTPException(status\_code=400, detail="Предложение не относится к этому заказу")  
  
 order\_data = OrderUpdate(executor\_id=offer.executor\_id, status="В\_прогрессе")  
 updated\_order = order\_service.update\_order\_by\_id(session, order\_data, id)  
 offer\_service.update\_offer\_by\_id(session, OfferUpdate(status="accepted"), offer\_id)  
  
 executor = session.get(User, offer.executor\_id)  
 message = (  
 f"Ваше предложение по заказу '{order.title}' (ID: {order.id}) принято!\n"  
 f"Цена: {offer.price} тенге\n"  
 f"Время выполнения: {offer.estimated\_time} часов\n"  
 f"Свяжитесь с заказчиком: @{current\_user.username}"  
 )  
 try:  
 await send\_telegram\_message(executor.telegram\_id, message)  
 except Exception as e:  
 logger.error(f"Ошибка отправки уведомления исполнителю: {e}")  
  
 return updated\_order  
  
  
@router.post("/", response\_model=OfferRead, status\_code=status.HTTP\_201\_CREATED)  
async def create\_offer(  
 data: OfferCreate,  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Создать новое предложение (доступно только исполнителям)."""  
 if not current\_user.is\_executor:  
 raise HTTPException(status\_code=403, detail="Только исполнители могут создавать предложения")  
 offer = offer\_service.create\_offer(session, data, current\_user.id)  
  
 # Получаем заказ и заказчика  
 order = session.get(Order, offer.order\_id)  
 customer = session.get(User, order.customer\_id)  
  
 # Отправляем уведомление заказчику  
 message = (  
 f"Новое предложение по вашему заказу '{order.title}' (ID: {order.id}):\n"  
 f"Исполнитель: {current\_user.name}\n"  
 f"Цена: {offer.price} тенге\n"  
 f"Время выполнения: {offer.estimated\_time} часов"  
 )  
 try:  
 await send\_telegram\_message(customer.telegram\_id, message)  
 except Exception as e:  
 # Логируем ошибку, но не прерываем выполнение  
 print(f"Ошибка отправки уведомления: {e}")  
  
 return offer  
  
  
# Остальные эндпоинты остаются без изменений  
@router.get("/", response\_model=List[OfferRead])  
def get\_offers(  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Получить список предложений текущего пользователя."""  
 return offer\_service.get\_offers\_by\_user(session, current\_user.id)  
  
  
@router.get("/{id}", response\_model=OfferRead)  
def get\_offer(  
 id: int,  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Получить предложение по ID."""  
 offer = offer\_service.get\_offer\_by\_id(session, id)  
 if offer.executor\_id != current\_user.id:  
 raise HTTPException(status\_code=403, detail="Нет прав для просмотра этого предложения")  
 return offer  
  
  
@router.patch("/{id}", response\_model=OfferRead)  
def update\_offer(  
 id: int,  
 data: OfferUpdate,  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Обновить предложение (доступно только исполнителю)."""  
 offer = offer\_service.get\_offer\_by\_id(session, id)  
 if offer.executor\_id != current\_user.id:  
 raise HTTPException(status\_code=403, detail="Только исполнитель может обновлять это предложение")  
 return offer\_service.update\_offer\_by\_id(session, data, id)  
  
  
@router.delete("/{id}", status\_code=status.HTTP\_204\_NO\_CONTENT)  
def delete\_offer(  
 id: int,  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Удалить предложение (доступно только исполнителю)."""  
 offer = offer\_service.get\_offer\_by\_id(session, id)  
 if offer.executor\_id != current\_user.id:  
 raise HTTPException(status\_code=403, detail="Только исполнитель может удалить это предложение")  
 offer\_service.delete\_offer\_by\_id(session, id)  
  
@router.post("/{id}/offers/{offer\_id}/reject", response\_model=OfferRead)  
async def reject\_offer(  
 id: int,  
 offer\_id: int,  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Отклонить предложение (доступно только заказчику)."""  
 order = order\_service.get\_order\_by\_id(session, id)  
 if order.customer\_id != current\_user.id:  
 raise HTTPException(status\_code=403, detail="Только заказчик может отклонить предложение")  
 if order.status != "В\_ожидании": # Проверка на русский статус  
 raise HTTPException(status\_code=400, detail="Нельзя отклонить предложение для заказа не в статусе 'В\_ожидании'")  
  
 offer = offer\_service.get\_offer\_by\_id(session, offer\_id)  
 if offer.order\_id != id:  
 raise HTTPException(status\_code=400, detail="Предложение не относится к этому заказу")  
  
 updated\_offer = offer\_service.update\_offer\_by\_id(session, OfferUpdate(status="rejected"), offer\_id)  
  
 executor = session.get(User, offer.executor\_id)  
 message = (  
 f"Ваше предложение по заказу '{order.title}' (ID: {order.id}) было отклонено заказчиком.\n"  
 f"Цена: {offer.price} тенге\n"  
 f"Время выполнения: {offer.estimated\_time} часов"  
 )  
 try:  
 await send\_telegram\_message(executor.telegram\_id, message)  
 except Exception as e:  
 logger.error(f"Ошибка отправки уведомления исполнителю: {e}")  
  
 return updated\_offer

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\api\order.py

from typing import Annotated, List  
from fastapi import APIRouter, Depends, status, HTTPException  
from sqlalchemy.orm import Session  
from app.api.offer import send\_telegram\_message  
from app.core.database.helper import get\_session  
from app.core.models.user import User  
from app.core.schemas.offer import OfferRead, OfferUpdate  
from app.core.services import order as order\_service  
from app.core.schemas.order import OrderRead, OrderCreate, OrderUpdate  
from app.api.depends.user import get\_current\_user, get\_admin\_user  
from datetime import datetime, timedelta  
import logging  
  
router = APIRouter(prefix="/order", tags=["Order"])  
logger = logging.getLogger(\_\_name\_\_)  
  
  
@router.post("/", response\_model=OrderRead, status\_code=status.HTTP\_201\_CREATED)  
def create\_order(  
 data: OrderCreate,  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Создать новый заказ (доступно только заказчикам)."""  
 logger.info(f"Создание заказа пользователем {current\_user.id}")  
 if not current\_user.is\_customer:  
 logger.warning(f"Попытка создания заказа не заказчиком: {current\_user.id}")  
 raise HTTPException(status\_code=403, detail="Только заказчики могут создавать заказы")  
 order = order\_service.create\_order(session, data, current\_user.id)  
 logger.info(f"Заказ создан: ID {order.id}")  
 return order  
  
  
@router.get("/", response\_model=List[OrderRead])  
def get\_orders(  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Получить список заказов текущего пользователя."""  
 logger.info(f"Получение заказов для пользователя {current\_user.id}")  
 orders = order\_service.get\_orders\_by\_user(session, current\_user.id)  
 logger.info(f"Найдено {len(orders)} заказов для пользователя {current\_user.id}")  
 return orders  
  
  
@router.get("/available", response\_model=List[OrderRead])  
def get\_available\_orders(  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 logger.info(f"Запрос доступных заказов от пользователя {current\_user.id}")  
 if not current\_user.is\_executor:  
 logger.warning(f"Попытка доступа к доступным заказам не исполнителем: {current\_user.id}")  
 raise HTTPException(status\_code=403, detail="Только исполнители могут видеть доступные заказы")  
 orders = order\_service.get\_available\_orders(session)  
 logger.info(f"Найдено {len(orders)} доступных заказов")  
 return orders  
  
  
@router.get("/{id}", response\_model=OrderRead)  
def get\_order(  
 id: int,  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Получить заказ по ID."""  
 logger.info(f"Запрос заказа ID {id} от пользователя {current\_user.id}")  
 order = order\_service.get\_order\_by\_id(session, id)  
 if order.customer\_id != current\_user.id and order.executor\_id != current\_user.id:  
 logger.warning(f"Попытка доступа к заказу ID {id} без прав: {current\_user.id}")  
 raise HTTPException(status\_code=403, detail="Нет прав для просмотра этого заказа")  
 return order  
  
  
@router.get("/{id}/offers", response\_model=List[OfferRead])  
def get\_order\_offers(  
 id: int,  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Получить список предложений по заказу (доступно только заказчику)."""  
 order = order\_service.get\_order\_by\_id(session, id)  
 if order.customer\_id != current\_user.id:  
 raise HTTPException(status\_code=403, detail="Только заказчик может видеть предложения по этому заказу")  
  
 # Добавляем информацию об исполнителе  
 offers = order.offers  
 for offer in offers:  
 executor = session.get(User, offer.executor\_id)  
 offer.executor\_rating = executor.rating # Добавляем рейтинг исполнителя  
 return offers  
  
  
@router.patch("/{id}", response\_model=OrderRead)  
def update\_order(  
 id: int,  
 data: OrderUpdate,  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Обновить заказ (доступно только заказчику)."""  
 logger.info(f"Обновление заказа ID {id} пользователем {current\_user.id}")  
 order = order\_service.get\_order\_by\_id(session, id)  
 if order.customer\_id != current\_user.id:  
 logger.warning(f"Попытка обновления заказа ID {id} не заказчиком: {current\_user.id}")  
 raise HTTPException(status\_code=403, detail="Только заказчик может обновлять этот заказ")  
 updated\_order = order\_service.update\_order\_by\_id(session, data, id)  
 logger.info(f"Заказ ID {id} обновлён")  
 return updated\_order  
  
  
@router.delete("/{id}", status\_code=status.HTTP\_204\_NO\_CONTENT)  
def delete\_order(  
 id: int,  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Удалить заказ (доступно заказчику только в статусе 'В\_ожидании' или администратору)."""  
 logger.info(f"Удаление заказа ID {id} пользователем {current\_user.id}")  
 order = order\_service.get\_order\_by\_id(session, id)  
 if current\_user.is\_admin:  
 order\_service.delete\_order\_by\_id(session, id)  
 logger.info(f"Заказ ID {id} удалён администратором {current\_user.id}")  
 elif order.customer\_id == current\_user.id:  
 if order.status != "В\_ожидании": # Проверяем на русский статус  
 logger.warning(f"Попытка удаления заказа ID {id} не в статусе 'В\_ожидании': {current\_user.id}")  
 raise HTTPException(status\_code=403, detail="Заказ можно удалить только в статусе 'В\_ожидании'")  
 order\_service.delete\_order\_by\_id(session, id)  
 logger.info(f"Заказ ID {id} удалён заказчиком {current\_user.id}")  
 else:  
 logger.warning(f"Попытка удаления заказа ID {id} без прав: {current\_user.id}")  
 raise HTTPException(status\_code=403, detail="Нет прав для удаления этого заказа")  
  
@router.post("/{id}/cancel", response\_model=OrderRead)  
def cancel\_order(  
 id: int,  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Отменить заказ (доступно заказчику в течение 5 минут после создания)."""  
 logger.info(f"Отмена заказа ID {id} пользователем {current\_user.id}")  
 order = order\_service.get\_order\_by\_id(session, id)  
 if order.customer\_id != current\_user.id:  
 logger.warning(f"Попытка отмены заказа ID {id} не заказчиком: {current\_user.id}")  
 raise HTTPException(status\_code=403, detail="Только заказчик может отменить этот заказ")  
 if datetime.utcnow() > order.created\_at + timedelta(minutes=5):  
 logger.warning(f"Попытка отмены заказа ID {id} после 5 минут: {current\_user.id}")  
 raise HTTPException(status\_code=400, detail="Заказ можно отменить только в течение 5 минут после создания")  
 canceled\_order = order\_service.update\_order\_by\_id(session, OrderUpdate(status="Отменен"), id)  
 logger.info(f"Заказ ID {id} отменён")  
 return canceled\_order  
  
@router.post("/{id}/offers/{offer\_id}/reject", response\_model=OfferRead)  
async def reject\_offer(  
 id: int,  
 offer\_id: int,  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Отклонить предложение (доступно только заказчику)."""  
 order = order\_service.get\_order\_by\_id(session, id)  
 if order.customer\_id != current\_user.id:  
 raise HTTPException(status\_code=403, detail="Только заказчик может отклонить предложение")  
 if order.status != "pending":  
 raise HTTPException(status\_code=400, detail="Нельзя отклонить предложение для заказа не в статусе 'pending'")  
  
 offer = offer\_service.get\_offer\_by\_id(session, offer\_id)  
 if offer.order\_id != id:  
 raise HTTPException(status\_code=400, detail="Предложение не относится к этому заказу")  
  
 # Обновляем статус предложения  
 updated\_offer = offer\_service.update\_offer\_by\_id(session, OfferUpdate(status="rejected"), offer\_id)  
  
 # Уведомляем исполнителя  
 executor = session.get(User, offer.executor\_id)  
 message = (  
 f"Ваше предложение по заказу '{order.title}' (ID: {order.id}) было отклонено заказчиком.\n"  
 f"Цена: {offer.price} тенге\n"  
 f"Время выполнения: {offer.estimated\_time} часов"  
 )  
 try:  
 await send\_telegram\_message(executor.telegram\_id, message)  
 except Exception as e:  
 logger.error(f"Ошибка отправки уведомления исполнителю: {e}")  
  
 return updated\_offer

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\api\review.py

from typing import Annotated, List  
from fastapi import APIRouter, Depends, status, HTTPException  
from sqlalchemy.orm import Session  
from app.core.database.helper import get\_session  
from app.core.models.user import User  
from app.core.services import review as review\_service  
from app.core.schemas.review import ReviewRead, ReviewCreate, ReviewUpdate  
from app.api.depends.user import get\_current\_user, get\_admin\_user  
  
router = APIRouter(prefix="/review", tags=["Review"]) # Маршруты для отзывов  
  
@router.post("/", response\_model=ReviewRead, status\_code=status.HTTP\_201\_CREATED)  
def create\_review(  
 data: ReviewCreate,  
 user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Создать новый отзыв (доступно только заказчикам)."""  
 if not user.is\_customer:  
 raise HTTPException(status\_code=403, detail="Только заказчики могут создавать отзывы")  
 return review\_service.create\_review(session, data, user.id)  
  
@router.get("/", response\_model=List[ReviewRead])  
def get\_reviews(  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Получить список отзывов текущего пользователя."""  
 return review\_service.get\_reviews\_by\_user(session, current\_user.id)  
  
@router.get("/{id}", response\_model=ReviewRead)  
def get\_review(  
 id: int,  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Получить отзыв по ID."""  
 review = review\_service.get\_review\_by\_id(session, id)  
 if review.author\_id != current\_user.id:  
 raise HTTPException(status\_code=403, detail="Нет прав для просмотра этого отзыва")  
 return review  
  
@router.patch("/{id}", response\_model=ReviewRead)  
def update\_review(  
 id: int,  
 data: ReviewUpdate,  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Обновить отзыв (доступно только автору)."""  
 review = review\_service.get\_review\_by\_id(session, id)  
 if review.author\_id != current\_user.id:  
 raise HTTPException(status\_code=403, detail="Только автор может обновлять этот отзыв")  
 return review\_service.update\_review\_by\_id(session, data, id)  
  
@router.delete("/{id}", status\_code=status.HTTP\_204\_NO\_CONTENT)  
def delete\_review(  
 id: int,  
 admin: Annotated[User, Depends(get\_admin\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Удалить отзыв (доступно только администратору)."""  
 review\_service.delete\_review\_by\_id(session, id)

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\api\routers.py

from fastapi import APIRouter  
from app.api.user import router as user\_router  
from app.api.offer import router as offer\_router  
from app.api.review import router as review\_router  
from app.api.city import router as city\_router  
from app.api.category import router as category\_router  
from app.api.order import router as order\_router  
  
routers = APIRouter(prefix="/api")  
routers.include\_router(user\_router)  
routers.include\_router(order\_router) # Убедитесь, что эта строка присутствует  
routers.include\_router(offer\_router)  
routers.include\_router(review\_router)  
routers.include\_router(city\_router)  
routers.include\_router(category\_router)

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\api\user.py

from typing import Annotated, List  
from fastapi import APIRouter, Depends, status  
from sqlalchemy.orm import Session  
from app.core.database.helper import get\_session  
from app.core.models.user import User  
from app.core.services import user as user\_service  
from app.core.schemas.user import UserRead, UserCreate, UserUpdate  
from app.api.depends.user import get\_current\_user, get\_admin\_user  
from fastapi.exceptions import HTTPException  
  
router = APIRouter(prefix="/user", tags=["User"]) # Маршруты для пользователей  
  
@router.get("/by\_telegram\_id/{telegram\_id}", response\_model=UserRead)  
def get\_user\_by\_telegram\_id(  
 telegram\_id: int,  
 session: Annotated[Session, Depends(get\_session)],  
 current\_user: Annotated[User, Depends(get\_current\_user)],  
):  
 """Получить пользователя по Telegram ID."""  
 user = session.query(User).filter(User.telegram\_id == telegram\_id).first()  
 if not user:  
 raise HTTPException(status\_code=404, detail="Пользователь не найден")  
 return user  
  
@router.get("/me", response\_model=UserRead)  
def get\_me(user: Annotated[User, Depends(get\_current\_user)]):  
 """Получить данные текущего пользователя."""  
 return user  
  
@router.get("/all", response\_model=List[UserRead])  
def get\_all\_users(  
 admin: Annotated[User, Depends(get\_admin\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Получить список всех пользователей (доступно только администратору)."""  
 return user\_service.get\_users(session)  
  
@router.post("/", response\_model=UserRead, status\_code=status.HTTP\_201\_CREATED)  
def create\_user(  
 data: UserCreate,  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Создать нового пользователя."""  
 return user\_service.create\_user(session, data)  
  
@router.patch("/me", response\_model=UserRead)  
def update\_me(  
 data: UserUpdate,  
 user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Обновить данные текущего пользователя."""  
 return user\_service.update\_user\_by\_id(session, data, user.id)  
  
@router.patch("/{id}", response\_model=UserRead)  
def update\_user\_by\_id(  
 id: int,  
 data: UserUpdate,  
 admin: Annotated[User, Depends(get\_admin\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Обновить данные пользователя по ID (доступно только администратору)."""  
 return user\_service.update\_user\_by\_id(session, data, id)  
  
@router.delete("/{id}", status\_code=status.HTTP\_204\_NO\_CONTENT)  
def delete\_user(  
 id: int,  
 admin: Annotated[User, Depends(get\_admin\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Удалить пользователя по ID (доступно только администратору)."""  
 user\_service.delete\_user\_by\_id(session, id)

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\api\depends\user.py

from typing import Annotated  
from sqlalchemy.orm import Session  
from fastapi.exceptions import HTTPException  
from fastapi import Depends, Header, status  
from app.core.models.user import User  
from app.core.database.helper import get\_session  
  
ADMIN\_TELEGRAM\_ID = 704342630 # Telegram ID администратора  
  
def get\_current\_user(  
 x\_telegram\_id: Annotated[str, Header()],  
 session: Annotated[Session, Depends(get\_session)],  
) -> User:  
 """Получить текущего пользователя по заголовку x-telegram-id."""  
 if not x\_telegram\_id:  
 raise HTTPException(status\_code=400, detail="Заголовок x-telegram-id отсутствует")  
 try:  
 telegram\_id = int(x\_telegram\_id)  
 except ValueError:  
 raise HTTPException(status\_code=400, detail="x-telegram-id должен быть целым числом")  
 user = session.query(User).filter(User.telegram\_id == telegram\_id).first()  
 if not user:  
 raise HTTPException(status\_code=404, detail="Пользователь не найден")  
 return user  
  
def get\_admin\_user(  
 current\_user: Annotated[User, Depends(get\_current\_user)]  
) -> User:  
 """Проверить, является ли текущий пользователь администратором."""  
 if current\_user.telegram\_id != ADMIN\_TELEGRAM\_ID:  
 raise HTTPException(status\_code=403, detail="Доступ запрещен")  
 return current\_user

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\bot\_runner.py

from aiogram import Bot, Dispatcher  
from app.bot.config import BOT\_TOKEN, ADMIN\_TELEGRAM\_ID  
from app.bot.handlers.start import router as start\_router  
from app.bot.handlers.switch\_role import router as switch\_role\_router  
from app.bot.handlers.admin import admin\_routers  
from app.bot.handlers.customer import customer\_routers  
from app.bot.handlers.executor import executor\_routers  
import asyncio  
import logging  
  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(\_\_name\_\_)  
  
if not BOT\_TOKEN:  
 raise ValueError("BOT\_TOKEN не задан")  
  
bot = Bot(token=BOT\_TOKEN)  
dp = Dispatcher()  
  
# Подключаем роутеры  
dp.include\_router(start\_router)  
dp.include\_router(switch\_role\_router)  
for admin\_router in admin\_routers:  
 dp.include\_router(admin\_router)  
for customer\_router in customer\_routers:  
 dp.include\_router(customer\_router)  
for executor\_router in executor\_routers:  
 dp.include\_router(executor\_router)  
  
async def main():  
 logger.info(f"Бот запущен с токеном: {BOT\_TOKEN[:10]}...")  
 try:  
 bot\_info = await bot.get\_me()  
 logger.info(f"Bot info: {bot\_info.username}, ID: {bot\_info.id}")  
 await dp.start\_polling(bot)  
 except Exception as e:  
 logger.error(f"Ошибка при запуске polling: {e}")  
 raise  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 asyncio.run(main())

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\config.py

BOT\_TOKEN = "7706031286:AAHKHOZKmrGInQ7aUOGJfOBRx28ao8ctOkk" # Ваш токен  
API\_URL = "http://localhost:8010/api/" # Для API  
ADMIN\_TELEGRAM\_ID = 704342630 # Ваш Telegram ID

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\handlers\common.py

import aiohttp  
from aiogram import Router  
from aiogram.types import Message, ReplyKeyboardMarkup, KeyboardButton, InlineKeyboardMarkup, InlineKeyboardButton  
from app.bot.config import ADMIN\_TELEGRAM\_ID, API\_URL  
import logging  
  
logger = logging.getLogger(\_\_name\_\_)  
  
# Общий роутер для утилит  
common\_router = Router()  
  
# Функция для выполнения API-запросов  
async def api\_request(method: str, url: str, telegram\_id: int, data: dict = None):  
 headers = {"x-telegram-id": str(telegram\_id)}  
 logger.info(f"Выполняется запрос: {method} {url} с headers={headers}")  
 async with aiohttp.ClientSession() as session:  
 try:  
 if method == "GET":  
 async with session.get(url, headers=headers) as response:  
 if response.status != 200:  
 error\_text = await response.text()  
 raise Exception(f"Ошибка {response.status}: {error\_text}")  
 return await response.json()  
 elif method == "POST":  
 async with session.post(url, headers=headers, json=data) as response:  
 if response.status not in (200, 201):  
 raise Exception(f"Ошибка {response.status}: {await response.text()}")  
 return await response.json()  
 elif method == "PATCH":  
 async with session.patch(url, headers=headers, json=data) as response:  
 if response.status != 200:  
 raise Exception(f"Ошибка {response.status}: {await response.text()}")  
 return await response.json()  
 elif method == "DELETE":  
 async with session.delete(url, headers=headers) as response:  
 if response.status != 204:  
 raise Exception(f"Ошибка {response.status}: {await response.text()}")  
 return None  
 except Exception as e:  
 logger.error(f"Ошибка при выполнении запроса {method} {url}: {e}")  
 raise  
  
# Функция для выполнения API-запросов без авторизации  
async def api\_request\_no\_auth(method: str, url: str):  
 async with aiohttp.ClientSession() as session:  
 if method == "GET":  
 async with session.get(url) as response:  
 if response.status != 200:  
 raise Exception(f"Ошибка {response.status}: {await response.text()}")  
 return await response.json()  
  
# Получение Telegram ID пользователя  
def get\_user\_telegram\_id(message: Message) -> int:  
 return message.from\_user.id  
  
# Основная клавиатура  
def get\_main\_keyboard(roles: dict = None) -> ReplyKeyboardMarkup:  
 roles = roles or {}  
 buttons = [  
 [KeyboardButton(text="Профиль"), KeyboardButton(text="Создать заказ")],  
 [KeyboardButton(text="Список заказов"), KeyboardButton(text="Сменить роль")]  
 ]  
 if roles.get("is\_executor"):  
 buttons.append([KeyboardButton(text="Создать предложение")])  
 if roles.get("is\_customer"):  
 buttons.append([KeyboardButton(text="Посмотреть предложения")])  
 if roles.get("is\_admin"):  
 buttons.append([KeyboardButton(text="Админ панель")])  
 return ReplyKeyboardMarkup(keyboard=buttons, resize\_keyboard=True, row\_width=2)  
  
# Функция для получения роли пользователя  
async def get\_user\_roles(telegram\_id: int) -> dict:  
 try:  
 user = await api\_request("GET", f"{API\_URL}user/by\_telegram\_id/{telegram\_id}", telegram\_id)  
 return {  
 "is\_admin": telegram\_id == ADMIN\_TELEGRAM\_ID,  
 "is\_executor": user["is\_executor"],  
 "is\_customer": user["is\_customer"]  
 }  
 except Exception as e:  
 logger.error(f"Ошибка получения ролей пользователя: {e}")  
 return {"is\_admin": False, "is\_executor": False, "is\_customer": False}

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\handlers\start.py

from aiogram import Router, F  
from aiogram.filters import Command  
from aiogram.types import Message, ReplyKeyboardMarkup, KeyboardButton, InlineKeyboardMarkup, InlineKeyboardButton, CallbackQuery  
from app.bot.config import ADMIN\_TELEGRAM\_ID, API\_URL  
from app.bot.handlers.common import api\_request, api\_request\_no\_auth, get\_main\_keyboard, get\_user\_roles  
import aiohttp  
import logging  
  
router = Router()  
logger = logging.getLogger(\_\_name\_\_)  
  
@router.message(Command("start"))  
async def start\_command(message: Message):  
 logger.info(f"Команда /start от пользователя {message.from\_user.id}")  
 telegram\_id = message.from\_user.id  
 try:  
 user = await api\_request("GET", f"{API\_URL}user/by\_telegram\_id/{telegram\_id}", telegram\_id)  
 roles = await get\_user\_roles(telegram\_id)  
 except Exception as e:  
 logger.error(f"Ошибка при проверке профиля: {e}")  
 if "404" in str(e):  
 try:  
 cities = await api\_request\_no\_auth("GET", f"{API\_URL}city/")  
 if not cities:  
 await message.answer("В системе нет городов. Обратитесь к администратору.")  
 return  
 city\_id = cities[0]["id"]  
 except Exception as city\_error:  
 logger.error(f"Ошибка с городами: {city\_error}")  
 await message.answer(f"Ошибка с городами: {city\_error}")  
 return  
  
 user\_data = {  
 "telegram\_id": telegram\_id,  
 "name": message.from\_user.full\_name or "Unnamed",  
 "username": message.from\_user.username,  
 "is\_customer": True,  
 "is\_executor": False,  
 "city\_id": city\_id  
 }  
 try:  
 await api\_request("POST", f"{API\_URL}user/", telegram\_id, data=user\_data)  
 roles = await get\_user\_roles(telegram\_id)  
 except Exception as create\_error:  
 logger.error(f"Ошибка создания профиля: {create\_error}")  
 await message.answer(f"Ошибка создания профиля: {create\_error}")  
 return  
 else:  
 await message.answer(f"Ошибка при проверке профиля: {e}")  
 return  
 await message.answer("Добро пожаловать!", reply\_markup=get\_main\_keyboard(roles))  
  
@router.message(F.text == "Профиль")  
async def show\_profile(message: Message):  
 logger.info(f"Команда 'Профиль' от пользователя {message.from\_user.id}")  
 telegram\_id = message.from\_user.id  
 try:  
 user = await api\_request("GET", f"{API\_URL}user/by\_telegram\_id/{telegram\_id}", telegram\_id)  
 role = "Заказчик" if user["is\_customer"] else "Исполнитель" if user["is\_executor"] else "Не определена"  
 city = await api\_request("GET", f"{API\_URL}city/{user['city\_id']}", telegram\_id)  
 text = f"Имя: {user['name']}\nРоль: {role}\nГород: {city['name']}\nРейтинг: {user['rating']}"  
 keyboard = InlineKeyboardMarkup(inline\_keyboard=[  
 [InlineKeyboardButton(text="Изменить имя", callback\_data="update\_name")],  
 [InlineKeyboardButton(text="Назад", callback\_data="back")]  
 ])  
 roles = await get\_user\_roles(telegram\_id)  
 await message.answer(text, reply\_markup=keyboard)  
 except Exception as e:  
 logger.error(f"Ошибка в show\_profile: {e}")  
 roles = await get\_user\_roles(telegram\_id)  
 await message.answer(f"Ошибка загрузки профиля: {e}", reply\_markup=get\_main\_keyboard(roles))  
  
@router.message(F.text == "Список заказов")  
async def show\_orders(message: Message):  
 logger.info(f"Команда 'Список заказов' от пользователя {message.from\_user.id}")  
 telegram\_id = message.from\_user.id  
 try:  
 user = await api\_request("GET", f"{API\_URL}user/by\_telegram\_id/{telegram\_id}", telegram\_id)  
 roles = await get\_user\_roles(telegram\_id)  
 if roles["is\_executor"]:  
 url = f"{API\_URL}order/available"  
 logger.info(f"Запрос для исполнителя: {url}")  
 orders = await api\_request("GET", url, telegram\_id)  
 title = "Доступные заказы:"  
 else:  
 url = f"{API\_URL}order/"  
 logger.info(f"Запрос для заказчика: {url}")  
 orders = await api\_request("GET", url, telegram\_id)  
 title = "Ваши заказы:"  
  
 if not orders:  
 await message.answer(f"{title.split(':')[0]} пока нет.", reply\_markup=get\_main\_keyboard(roles))  
 return  
  
 response = f"{title}\n\n"  
 status\_map = {  
 "В\_ожидании": "Ожидает",  
 "В\_прогрессе": "В процессе",  
 "Выполнен": "Завершён",  
 "Отменен": "Отменён"  
 }  
 for order in orders:  
 status = status\_map.get(order["status"], order["status"])  
 due\_date = order["due\_date"].split("T")[0]  
 response += (  
 f"ID: {order['id']}\n"  
 f"Название: {order['title']}\n"  
 f"Статус: {status}\n"  
 f"Желаемая цена: {order['desired\_price']} тенге\n"  
 f"Срок: {due\_date}\n\n"  
 )  
 await message.answer(response.strip(), reply\_markup=get\_main\_keyboard(roles))  
 except Exception as e:  
 logger.error(f"Ошибка в show\_orders: {e}")  
 roles = await get\_user\_roles(telegram\_id)  
 error\_msg = f"Ошибка загрузки заказов: {e}"  
 if "500" in str(e):  
 error\_msg += "\nПроблема на сервере. Обратитесь к администратору."  
 await message.answer(error\_msg, reply\_markup=get\_main\_keyboard(roles))  
  
@router.callback\_query(F.data == "update\_name")  
async def update\_name(callback: CallbackQuery):  
 logger.info(f"Обработчик update\_name вызван для telegram\_id={callback.from\_user.id}")  
 await callback.message.answer("Функция изменения имени пока не реализована.")  
 await callback.answer()  
  
@router.callback\_query(F.data == "back")  
async def back\_to\_main(callback: CallbackQuery):  
 logger.info(f"Обработчик back\_to\_main вызван для telegram\_id={callback.from\_user.id}")  
 telegram\_id = callback.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 await callback.message.answer("Главное меню:", reply\_markup=get\_main\_keyboard(roles))  
 await callback.answer()

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\handlers\switch\_role.py

from aiogram import Router, F  
from aiogram.types import Message, InlineKeyboardMarkup, InlineKeyboardButton, CallbackQuery  
from app.bot.handlers.common import api\_request, get\_main\_keyboard, get\_user\_roles  
from app.bot.config import ADMIN\_TELEGRAM\_ID, API\_URL  
  
router = Router()  
  
@router.message(F.text == "Сменить роль")  
async def switch\_role(message: Message):  
 telegram\_id = message.from\_user.id  
 try:  
 user = await api\_request("GET", f"{API\_URL}user/me", telegram\_id)  
 current\_role = "Заказчик" if user["is\_customer"] else "Исполнитель" if user["is\_executor"] else "Не определена"  
 keyboard = InlineKeyboardMarkup(inline\_keyboard=[  
 [InlineKeyboardButton(text="Заказчик", callback\_data="role\_customer")],  
 [InlineKeyboardButton(text="Исполнитель", callback\_data="role\_executor")],  
 [InlineKeyboardButton(text="Назад", callback\_data="back")]  
 ])  
 await message.answer(f"Текущая роль: {current\_role}\nВыберите новую роль:", reply\_markup=keyboard)  
 except Exception as e:  
 roles = await get\_user\_roles(telegram\_id)  
 await message.answer(f"Ошибка загрузки текущей роли: {e}", reply\_markup=get\_main\_keyboard(roles))  
  
@router.callback\_query(F.data.startswith("role\_"))  
async def change\_role(callback: CallbackQuery):  
 telegram\_id = callback.from\_user.id  
 role = callback.data.split("\_")[1]  
 role\_name = "Заказчик" if role == "customer" else "Исполнитель"  
 try:  
 update\_data = {  
 "is\_customer": role == "customer",  
 "is\_executor": role == "executor"  
 }  
 await api\_request("PATCH", f"{API\_URL}user/me", telegram\_id, data=update\_data)  
 user = await api\_request("GET", f"{API\_URL}user/me", telegram\_id)  
 roles = await get\_user\_roles(telegram\_id)  
 await callback.message.answer(f"Роль успешно изменена на: {role\_name}", reply\_markup=get\_main\_keyboard(roles))  
 except Exception as e:  
 roles = await get\_user\_roles(telegram\_id)  
 await callback.message.answer(f"Ошибка смены роли: {e}", reply\_markup=get\_main\_keyboard(roles))  
 await callback.answer()  
  
@router.callback\_query(F.data == "back")  
async def back\_to\_main(callback: CallbackQuery):  
 telegram\_id = callback.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 await callback.message.answer("Главное меню:", reply\_markup=get\_main\_keyboard(roles))  
 await callback.answer()

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\handlers\admin\category\_management.py

from aiogram import Router, F  
from aiogram.types import Message, CallbackQuery  
from aiogram.fsm.state import State, StatesGroup  
from aiogram.fsm.context import FSMContext  
from app.bot.handlers.common import api\_request, get\_main\_keyboard  
from app.bot.config import API\_URL  
import logging  
  
router = Router()  
logger = logging.getLogger(\_\_name\_\_)  
  
class AdminCategoryStates(StatesGroup):  
 add\_category = State()  
 edit\_category\_select = State()  
 edit\_category\_name = State()  
 delete\_category = State()  
  
@router.callback\_query(F.data == "add\_category")  
async def start\_add\_category(callback: CallbackQuery, state: FSMContext):  
 await callback.message.answer("Введите название новой категории:")  
 await state.set\_state(AdminCategoryStates.add\_category)  
 await callback.answer()  
  
@router.message(AdminCategoryStates.add\_category)  
async def process\_add\_category(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 category\_name = message.text.strip()  
 try:  
 data = {"name": category\_name}  
 await api\_request("POST", f"{API\_URL}category/", telegram\_id, data=data)  
 await message.answer(f"Категория '{category\_name}' добавлена.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 except Exception as e:  
 logger.error(f"Ошибка в process\_add\_category: {e}")  
 await message.answer(f"Ошибка добавления категории: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await state.clear()  
  
@router.callback\_query(F.data == "edit\_category")  
async def start\_edit\_category(callback: CallbackQuery, state: FSMContext):  
 telegram\_id = callback.from\_user.id  
 try:  
 categories = await api\_request("GET", f"{API\_URL}category/", telegram\_id)  
 if not categories:  
 await callback.message.answer("Категорий нет.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
 return  
 response = "Список категорий:\n\n"  
 for category in categories:  
 response += f"ID: {category['id']} - {category['name']}\n"  
 await callback.message.answer(response.strip() + "\n\nВведите ID категории для изменения:")  
 await state.set\_state(AdminCategoryStates.edit\_category\_select)  
 except Exception as e:  
 logger.error(f"Ошибка в start\_edit\_category: {e}")  
 await callback.message.answer(f"Ошибка загрузки категорий: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
  
@router.message(AdminCategoryStates.edit\_category\_select)  
async def process\_edit\_category\_select(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 category\_id = int(message.text)  
 category = await api\_request("GET", f"{API\_URL}category/{category\_id}", telegram\_id)  
 await state.update\_data(category\_id=category\_id)  
 await message.answer(f"Текущее название: {category['name']}\nВведите новое название категории:")  
 await state.set\_state(AdminCategoryStates.edit\_category\_name)  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID категории.")  
 except Exception as e:  
 logger.error(f"Ошибка в process\_edit\_category\_select: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await state.clear()  
  
@router.message(AdminCategoryStates.edit\_category\_name)  
async def process\_edit\_category\_name(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 data = await state.get\_data()  
 category\_id = data["category\_id"]  
 new\_name = message.text.strip()  
 update\_data = {"name": new\_name}  
 await api\_request("PATCH", f"{API\_URL}category/{category\_id}", telegram\_id, data=update\_data)  
 await message.answer(f"Категория с ID {category\_id} изменена на '{new\_name}'.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 except Exception as e:  
 logger.error(f"Ошибка в process\_edit\_category\_name: {e}")  
 await message.answer(f"Ошибка изменения категории: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await state.clear()  
  
@router.callback\_query(F.data == "delete\_category")  
async def start\_delete\_category(callback: CallbackQuery, state: FSMContext):  
 telegram\_id = callback.from\_user.id  
 try:  
 categories = await api\_request("GET", f"{API\_URL}category/", telegram\_id)  
 if not categories:  
 await callback.message.answer("Категорий нет.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
 return  
 response = "Список категорий:\n\n"  
 for category in categories:  
 response += f"ID: {category['id']} - {category['name']}\n"  
 await callback.message.answer(response.strip() + "\n\nВведите ID категории для удаления:")  
 await state.set\_state(AdminCategoryStates.delete\_category)  
 except Exception as e:  
 logger.error(f"Ошибка в start\_delete\_category: {e}")  
 await callback.message.answer(f"Ошибка загрузки категорий: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
  
@router.message(AdminCategoryStates.delete\_category)  
async def process\_delete\_category(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 category\_id = int(message.text)  
 await api\_request("DELETE", f"{API\_URL}category/{category\_id}", telegram\_id)  
 await message.answer(f"Категория с ID {category\_id} удалена.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID категории.")  
 except Exception as e:  
 logger.error(f"Ошибка в process\_delete\_category: {e}")  
 await message.answer(f"Ошибка удаления категории: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await state.clear()

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\handlers\admin\city\_management.py

from aiogram import Router, F  
from aiogram.types import Message, CallbackQuery  
from aiogram.fsm.state import State, StatesGroup  
from aiogram.fsm.context import FSMContext  
from app.bot.handlers.common import api\_request, get\_main\_keyboard  
from app.bot.config import API\_URL  
import logging  
  
router = Router()  
logger = logging.getLogger(\_\_name\_\_)  
  
class AdminCityStates(StatesGroup):  
 add\_city = State()  
 edit\_city\_select = State()  
 edit\_city\_name = State()  
 delete\_city = State()  
  
@router.callback\_query(F.data == "list\_cities")  
async def list\_cities(callback: CallbackQuery):  
 telegram\_id = callback.from\_user.id  
 try:  
 cities = await api\_request("GET", f"{API\_URL}city/", telegram\_id)  
 if not cities:  
 await callback.message.answer("Городов нет.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
 return  
 response = "Список городов:\n\n"  
 for city in cities:  
 response += f"ID: {city['id']} - {city['name']}\n"  
 await callback.message.answer(response.strip(), reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 except Exception as e:  
 logger.error(f"Ошибка в list\_cities: {e}")  
 await callback.message.answer(f"Ошибка загрузки городов: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
  
@router.callback\_query(F.data == "add\_city")  
async def start\_add\_city(callback: CallbackQuery, state: FSMContext):  
 await callback.message.answer("Введите название нового города:")  
 await state.set\_state(AdminCityStates.add\_city)  
 await callback.answer()  
  
@router.message(AdminCityStates.add\_city)  
async def process\_add\_city(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 city\_name = message.text.strip()  
 try:  
 data = {"name": city\_name}  
 await api\_request("POST", f"{API\_URL}city/", telegram\_id, data=data)  
 await message.answer(f"Город '{city\_name}' добавлен.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 except Exception as e:  
 logger.error(f"Ошибка в process\_add\_city: {e}")  
 await message.answer(f"Ошибка добавления города: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await state.clear()  
  
@router.callback\_query(F.data == "edit\_city")  
async def start\_edit\_city(callback: CallbackQuery, state: FSMContext):  
 telegram\_id = callback.from\_user.id  
 try:  
 cities = await api\_request("GET", f"{API\_URL}city/", telegram\_id)  
 if not cities:  
 await callback.message.answer("Городов нет.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
 return  
 response = "Список городов:\n\n"  
 for city in cities:  
 response += f"ID: {city['id']} - {city['name']}\n"  
 await callback.message.answer(response.strip() + "\n\nВведите ID города для изменения:")  
 await state.set\_state(AdminCityStates.edit\_city\_select)  
 except Exception as e:  
 logger.error(f"Ошибка в start\_edit\_city: {e}")  
 await callback.message.answer(f"Ошибка загрузки городов: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
  
@router.message(AdminCityStates.edit\_city\_select)  
async def process\_edit\_city\_select(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 city\_id = int(message.text)  
 city = await api\_request("GET", f"{API\_URL}city/{city\_id}", telegram\_id)  
 await state.update\_data(city\_id=city\_id)  
 await message.answer(f"Текущее название: {city['name']}\nВведите новое название города:")  
 await state.set\_state(AdminCityStates.edit\_city\_name)  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID города.")  
 except Exception as e:  
 logger.error(f"Ошибка в process\_edit\_city\_select: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await state.clear()  
  
@router.message(AdminCityStates.edit\_city\_name)  
async def process\_edit\_city\_name(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 data = await state.get\_data()  
 city\_id = data["city\_id"]  
 new\_name = message.text.strip()  
 update\_data = {"name": new\_name}  
 await api\_request("PATCH", f"{API\_URL}city/{city\_id}", telegram\_id, data=update\_data)  
 await message.answer(f"Город с ID {city\_id} изменён на '{new\_name}'.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 except Exception as e:  
 logger.error(f"Ошибка в process\_edit\_city\_name: {e}")  
 await message.answer(f"Ошибка изменения города: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await state.clear()  
  
@router.callback\_query(F.data == "delete\_city")  
async def start\_delete\_city(callback: CallbackQuery, state: FSMContext):  
 telegram\_id = callback.from\_user.id  
 try:  
 cities = await api\_request("GET", f"{API\_URL}city/", telegram\_id)  
 if not cities:  
 await callback.message.answer("Городов нет.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
 return  
 response = "Список городов:\n\n"  
 for city in cities:  
 response += f"ID: {city['id']} - {city['name']}\n"  
 await callback.message.answer(response.strip() + "\n\nВведите ID города для удаления:")  
 await state.set\_state(AdminCityStates.delete\_city)  
 except Exception as e:  
 logger.error(f"Ошибка в start\_delete\_city: {e}")  
 await callback.message.answer(f"Ошибка загрузки городов: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
  
@router.message(AdminCityStates.delete\_city)  
async def process\_delete\_city(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 city\_id = int(message.text)  
 await api\_request("DELETE", f"{API\_URL}city/{city\_id}", telegram\_id)  
 await message.answer(f"Город с ID {city\_id} удалён.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID города.")  
 except Exception as e:  
 logger.error(f"Ошибка в process\_delete\_city: {e}")  
 await message.answer(f"Ошибка удаления города: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await state.clear()

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\handlers\admin\main.py

from aiogram import Router, F  
from aiogram.types import Message, InlineKeyboardMarkup, InlineKeyboardButton  
from app.bot.config import ADMIN\_TELEGRAM\_ID  
from app.bot.handlers.common import get\_main\_keyboard, get\_user\_roles  
import logging  
  
router = Router()  
logger = logging.getLogger(\_\_name\_\_)  
  
@router.message(F.text == "Админ панель")  
async def admin\_panel(message: Message):  
 logger.info(f"Попытка доступа к админ-панели от пользователя {message.from\_user.id}")  
 if message.from\_user.id != ADMIN\_TELEGRAM\_ID:  
 logger.warning(f"Доступ запрещен для пользователя {message.from\_user.id}")  
 await message.answer("Доступ запрещен!")  
 return  
 keyboard = InlineKeyboardMarkup(inline\_keyboard=[  
 [InlineKeyboardButton(text="Список пользователей", callback\_data="list\_users"),  
 InlineKeyboardButton(text="Список заказов", callback\_data="list\_orders")],  
 [InlineKeyboardButton(text="Удалить пользователя", callback\_data="delete\_user"),  
 InlineKeyboardButton(text="Удалить заказ", callback\_data="delete\_order")],  
 [InlineKeyboardButton(text="Список городов", callback\_data="list\_cities"),  
 InlineKeyboardButton(text="Добавить город", callback\_data="add\_city")],  
 [InlineKeyboardButton(text="Изменить город", callback\_data="edit\_city"),  
 InlineKeyboardButton(text="Удалить город", callback\_data="delete\_city")],  
 [InlineKeyboardButton(text="Добавить категорию", callback\_data="add\_category"),  
 InlineKeyboardButton(text="Изменить категорию", callback\_data="edit\_category")],  
 [InlineKeyboardButton(text="Удалить категорию", callback\_data="delete\_category"),  
 InlineKeyboardButton(text="Назад", callback\_data="back")]  
 ])  
 await message.answer("Админ-панель:", reply\_markup=keyboard)

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\handlers\admin\order\_management.py

from aiogram import Router, F  
from aiogram.types import Message, CallbackQuery  
from aiogram.fsm.state import State, StatesGroup  
from aiogram.fsm.context import FSMContext  
from app.bot.handlers.common import api\_request, get\_main\_keyboard, get\_user\_roles  
from app.bot.config import API\_URL  
import logging  
  
router = Router()  
logger = logging.getLogger(\_\_name\_\_)  
  
class AdminOrderStates(StatesGroup):  
 delete\_order = State()  
  
@router.callback\_query(F.data == "list\_orders")  
async def list\_orders(callback: CallbackQuery):  
 logger.info(f"Обработчик list\_orders вызван для telegram\_id={callback.from\_user.id}")  
 telegram\_id = callback.from\_user.id  
 try:  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 if not orders:  
 await callback.message.answer("Заказов нет.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
 return  
  
 response = "Список заказов:\n\n"  
 status\_map = {  
 "В\_ожидании": "Ожидает",  
 "В\_прогрессе": "В процессе",  
 "Выполнен": "Завершён",  
 "Отменен": "Отменён"  
 }  
 for order in orders:  
 status = status\_map.get(order["status"], order["status"])  
 due\_date = order["due\_date"].split("T")[0]  
 response += (  
 f"ID: {order['id']}\n"  
 f"Название: {order['title']}\n"  
 f"Статус: {status}\n"  
 f"Желаемая цена: {order['desired\_price']} тенге\n"  
 f"Срок: {due\_date}\n\n"  
 )  
 await callback.message.answer(response.strip(), reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 except Exception as e:  
 logger.error(f"Ошибка в list\_orders: {e}")  
 await callback.message.answer(f"Ошибка загрузки заказов: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
  
@router.callback\_query(F.data == "delete\_order")  
async def start\_delete\_order(callback: CallbackQuery, state: FSMContext):  
 logger.info(f"Обработчик start\_delete\_order вызван для telegram\_id={callback.from\_user.id}")  
 telegram\_id = callback.from\_user.id  
 try:  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 if not orders:  
 await callback.message.answer("Заказов нет.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
 return  
 response = "Список заказов:\n\n"  
 for order in orders:  
 response += f"ID: {order['id']} - {order['title']}\n"  
 await callback.message.answer(response.strip() + "\n\nВведите ID заказа для удаления:")  
 await state.set\_state(AdminOrderStates.delete\_order)  
 except Exception as e:  
 logger.error(f"Ошибка в start\_delete\_order: {e}")  
 await callback.message.answer(f"Ошибка загрузки заказов: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
  
@router.message(AdminOrderStates.delete\_order)  
async def process\_delete\_order(message: Message, state: FSMContext):  
 logger.info(f"Обработчик process\_delete\_order вызван для telegram\_id={message.from\_user.id}")  
 telegram\_id = message.from\_user.id  
 try:  
 order\_id = int(message.text)  
 await api\_request("DELETE", f"{API\_URL}order/{order\_id}", telegram\_id)  
 await message.answer(f"Заказ с ID {order\_id} удалён.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID заказа.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 except Exception as e:  
 logger.error(f"Ошибка в process\_delete\_order: {e}")  
 await message.answer(f"Ошибка удаления заказа: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await state.clear()

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\handlers\admin\user\_management.py

from aiogram import Router, F  
from aiogram.types import Message, CallbackQuery  
from aiogram.fsm.state import State, StatesGroup  
from aiogram.fsm.context import FSMContext  
from app.bot.handlers.common import api\_request, get\_main\_keyboard, get\_user\_roles  
from app.bot.config import API\_URL, ADMIN\_TELEGRAM\_ID  
import logging  
  
router = Router()  
logger = logging.getLogger(\_\_name\_\_)  
  
class AdminUserStates(StatesGroup):  
 delete\_user = State()  
  
@router.callback\_query(F.data == "list\_users")  
async def list\_users(callback: CallbackQuery):  
 logger.info(f"Обработчик list\_users вызван для telegram\_id={callback.from\_user.id}")  
 telegram\_id = callback.from\_user.id  
 try:  
 users = await api\_request("GET", f"{API\_URL}user/all", telegram\_id)  
 if not users:  
 await callback.message.answer("Пользователей нет.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
 return  
  
 response = "Список пользователей:\n\n"  
 for user in users:  
 role = "Заказчик" if user["is\_customer"] else "Исполнитель" if user["is\_executor"] else "Не определена"  
 response += (  
 f"ID: {user['id']}\n"  
 f"Telegram ID: {user['telegram\_id']}\n"  
 f"Имя: {user['name']}\n"  
 f"Роль: {role}\n"  
 f"Рейтинг: {user['rating']}\n\n"  
 )  
 await callback.message.answer(response.strip(), reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 except Exception as e:  
 logger.error(f"Ошибка в list\_users: {e}")  
 await callback.message.answer(f"Ошибка загрузки пользователей: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
  
@router.callback\_query(F.data == "delete\_user")  
async def start\_delete\_user(callback: CallbackQuery, state: FSMContext):  
 logger.info(f"Обработчик start\_delete\_user вызван для telegram\_id={callback.from\_user.id}")  
 telegram\_id = callback.from\_user.id  
 try:  
 users = await api\_request("GET", f"{API\_URL}user/all", telegram\_id)  
 if not users:  
 await callback.message.answer("Пользователей нет.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
 return  
 response = "Список пользователей:\n\n"  
 for user in users:  
 role = "Заказчик" if user["is\_customer"] else "Исполнитель" if user["is\_executor"] else "Не определена"  
 response += f"ID: {user['id']} - {user['name']} ({role})\n"  
 await callback.message.answer(response.strip() + "\n\nВведите ID пользователя для удаления:")  
 await state.set\_state(AdminUserStates.delete\_user)  
 except Exception as e:  
 logger.error(f"Ошибка в start\_delete\_user: {e}")  
 await callback.message.answer(f"Ошибка загрузки пользователей: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
  
@router.message(AdminUserStates.delete\_user)  
async def process\_delete\_user(message: Message, state: FSMContext):  
 logger.info(f"Обработчик process\_delete\_user вызван для telegram\_id={message.from\_user.id}")  
 telegram\_id = message.from\_user.id  
 try:  
 user\_id = int(message.text)  
 await api\_request("DELETE", f"{API\_URL}user/{user\_id}", telegram\_id)  
 await message.answer(f"Пользователь с ID {user\_id} удалён.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID пользователя.", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 except Exception as e:  
 logger.error(f"Ошибка в process\_delete\_user: {e}")  
 await message.answer(f"Ошибка удаления пользователя: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await state.clear()

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\handlers\admin\\_\_init\_\_.py

from .main import router as admin\_main\_router  
from .user\_management import router as user\_management\_router  
from .order\_management import router as order\_management\_router  
from .city\_management import router as city\_management\_router  
from .category\_management import router as category\_management\_router  
  
admin\_routers = [  
 admin\_main\_router,  
 user\_management\_router,  
 order\_management\_router,  
 city\_management\_router,  
 category\_management\_router  
]

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\handlers\customer\main.py

from aiogram import Router, F  
from aiogram.types import Message  
from app.bot.handlers.common import get\_main\_keyboard, get\_user\_roles  
import logging  
  
router = Router()  
logger = logging.getLogger(\_\_name\_\_)  
  
@router.message(F.text == "Создать заказ")  
async def create\_order\_entrypoint(message: Message):  
 telegram\_id = message.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 if not roles["is\_customer"]:  
 await message.answer("Только заказчики могут создавать заказы.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 # Передаем управление в orders.py  
 from .orders import start\_create\_order  
 await start\_create\_order(message)

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\handlers\customer\offers.py

from aiogram import Router, F  
from aiogram.types import Message, CallbackQuery, InlineKeyboardMarkup, InlineKeyboardButton  
from aiogram.fsm.state import State, StatesGroup  
from aiogram.fsm.context import FSMContext  
from app.bot.handlers.common import api\_request, get\_main\_keyboard, get\_user\_roles  
from app.bot.config import API\_URL  
import logging  
  
router = Router()  
logger = logging.getLogger(\_\_name\_\_)  
  
class ManageOffersStates(StatesGroup):  
 select\_order = State()  
  
@router.message(F.text == "Посмотреть предложения")  
async def manage\_offers(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 user = await api\_request("GET", f"{API\_URL}user/me", telegram\_id)  
 if not user["is\_customer"]:  
 roles = await get\_user\_roles(telegram\_id)  
 await message.answer("Только заказчики могут просматривать предложения.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 if not orders:  
 roles = await get\_user\_roles(telegram\_id)  
 await message.answer("У вас нет заказов.", reply\_markup=get\_main\_keyboard(roles))  
 return  
  
 keyboard = InlineKeyboardMarkup(inline\_keyboard=[  
 [InlineKeyboardButton(text=f"ID {order['id']} - {order['title']}", callback\_data=f"view\_offers\_{order['id']}")]  
 for order in orders  
 ] + [[InlineKeyboardButton(text="Отмена", callback\_data="cancel")]])  
 await message.answer("Выберите заказ для просмотра предложений:", reply\_markup=keyboard)  
 await state.set\_state(ManageOffersStates.select\_order)  
 except Exception as e:  
 roles = await get\_user\_roles(telegram\_id)  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(roles))

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\handlers\customer\orders.py

from aiogram import Router, F  
from aiogram.types import Message  
from aiogram.fsm.state import State, StatesGroup  
from aiogram.fsm.context import FSMContext  
from app.bot.handlers.common import api\_request, get\_main\_keyboard, get\_user\_roles  
from app.bot.config import API\_URL  
import logging  
  
router = Router()  
logger = logging.getLogger(\_\_name\_\_)  
  
class CreateOrderStates(StatesGroup):  
 title = State()  
 description = State()  
 price = State()  
 due\_date = State()  
 category = State()  
  
async def start\_create\_order(message: Message):  
 await message.answer("Введите название заказа:")  
 await message.bot.get\_state(message.from\_user.id, message.chat.id).set\_state(CreateOrderStates.title)  
  
@router.message(CreateOrderStates.title)  
async def process\_title(message: Message, state: FSMContext):  
 await state.update\_data(title=message.text.strip())  
 await message.answer("Введите описание заказа (или пропустите, нажав Enter):")  
 await state.set\_state(CreateOrderStates.description)  
  
# Добавьте остальные состояния для создания заказа...

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\handlers\customer\\_\_init\_\_.py

from .main import router as customer\_main\_router  
from .orders import router as customer\_orders\_router  
from .offers import router as customer\_offers\_router  
  
customer\_routers = [  
 customer\_main\_router,  
 customer\_orders\_router,  
 customer\_offers\_router  
]

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\handlers\executor\main.py

from aiogram import Router, F  
from aiogram.types import Message  
from app.bot.handlers.common import get\_main\_keyboard, get\_user\_roles  
import logging  
  
router = Router()  
logger = logging.getLogger(\_\_name\_\_)  
  
@router.message(F.text == "Создать предложение")  
async def create\_offer\_entrypoint(message: Message):  
 telegram\_id = message.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 if not roles["is\_executor"]:  
 await message.answer("Только исполнители могут создавать предложения.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 # Передаем управление в offers.py  
 from .offers import start\_create\_offer  
 await start\_create\_offer(message)

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\handlers\executor\offers.py

from aiogram import Router  
from aiogram.types import Message  
from aiogram.fsm.state import State, StatesGroup  
from aiogram.fsm.context import FSMContext  
from app.bot.handlers.common import api\_request, get\_main\_keyboard  
from app.bot.config import API\_URL  
import logging  
  
router = Router()  
logger = logging.getLogger(\_\_name\_\_)  
  
class CreateOfferStates(StatesGroup):  
 order\_id = State()  
 price = State()  
 estimated\_time = State()  
  
async def start\_create\_offer(message: Message):  
 telegram\_id = message.from\_user.id  
 orders = await api\_request("GET", f"{API\_URL}order/available", telegram\_id)  
 if not orders:  
 await message.answer("Нет доступных заказов.", reply\_markup=get\_main\_keyboard({"is\_executor": True}))  
 return  
 response = "Доступные заказы:\n\n"  
 for order in orders:  
 response += f"ID: {order['id']} - {order['title']}\n"  
 await message.answer(response + "\nВведите ID заказа для создания предложения:")  
 await message.bot.get\_state(message.from\_user.id, message.chat.id).set\_state(CreateOfferStates.order\_id)

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\handlers\executor\\_\_init\_\_.py

from .main import router as executor\_main\_router  
from .offers import router as executor\_offers\_router  
  
executor\_routers = [  
 executor\_main\_router,  
 executor\_offers\_router  
]

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\config.py

import os  
from dotenv import load\_dotenv  
  
load\_dotenv()  
  
DB\_URL = os.getenv("DB\_URL")  
DB\_ECHO = False

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\init\_db.py

from app.core.database.helper import engine  
from app.core.models import Base  
Base.metadata.create\_all(bind=engine)  
# если база пустая то запустить отдельно

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\database\helper.py

from sqlalchemy import create\_engine  
from sqlalchemy.orm import sessionmaker  
from app.core.config import DB\_URL, DB\_ECHO  
  
engine = create\_engine(url=DB\_URL, echo=DB\_ECHO)  
SessionLocal = sessionmaker(bind=engine)  
  
def get\_session():  
 with SessionLocal() as session:  
 yield session

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\models\association.py

from sqlalchemy import Table, Column, Integer, ForeignKey  
from app.core.models.base import Base  
  
user\_categories = Table(  
 "user\_categories",  
 Base.metadata,  
 Column("user\_id", Integer, ForeignKey("users.id", ondelete="CASCADE"), primary\_key=True), # ID пользователя  
 Column("category\_id", Integer, ForeignKey("categories.id", ondelete="CASCADE"), primary\_key=True), # ID категории  
) # Таблица связи многие-ко-многим между пользователями и категориями

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\models\base.py

from sqlalchemy.orm import DeclarativeBase, Mapped, mapped\_column  
  
class Base(DeclarativeBase):  
 """Базовый класс для всех моделей базы данных."""  
 id: Mapped[int] = mapped\_column(primary\_key=True, index=True)

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\models\category.py

from sqlalchemy.orm import Mapped, mapped\_column, relationship  
from app.core.models.base import Base  
  
class Category(Base):  
 """Модель категории услуг."""  
 \_\_tablename\_\_ = "categories"  
  
 name: Mapped[str] = mapped\_column(unique=True, nullable=False) # Название категории (уникальное)  
 users: Mapped[list["User"]] = relationship(  
 "User", secondary="user\_categories", back\_populates="categories" # Пользователи, связанные с категорией  
 )

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\models\city.py

from sqlalchemy.orm import Mapped, mapped\_column, relationship  
from app.core.models.base import Base  
  
class City(Base):  
 """Модель города."""  
 \_\_tablename\_\_ = "cities"  
  
 name: Mapped[str] = mapped\_column(unique=True, nullable=False) # Название города (уникальное)  
 users: Mapped[list["User"]] = relationship("User", back\_populates="city") # Пользователи, привязанные к городу

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\models\offer.py

from sqlalchemy.orm import Mapped, mapped\_column, relationship  
from datetime import datetime  
from app.core.models.base import Base  
from decimal import Decimal  
from sqlalchemy import Enum, ForeignKey, Numeric  
import enum  
  
class OfferStatus(str, enum.Enum):  
 PENDING = "pending" # Ожидает  
 ACCEPTED = "accepted" # Принято  
 REJECTED = "rejected" # Отклонено  
  
class Offer(Base):  
 """Модель предложения."""  
 \_\_tablename\_\_ = "offers"  
  
 order\_id: Mapped[int] = mapped\_column(ForeignKey("orders.id", ondelete="CASCADE"), nullable=False) # ID заказа  
 executor\_id: Mapped[int] = mapped\_column(ForeignKey("users.id", ondelete="CASCADE"), nullable=False) # ID исполнителя  
 price: Mapped[Decimal] = mapped\_column(Numeric(10, 2), nullable=False) # Цена предложения  
 estimated\_time: Mapped[int] = mapped\_column(nullable=False) # Оценочное время выполнения (в часах)  
 status: Mapped[OfferStatus] = mapped\_column(Enum(OfferStatus), default=OfferStatus.PENDING, nullable=False) # Статус предложения  
 created\_at: Mapped[datetime] = mapped\_column(default=datetime.utcnow, nullable=False) # Дата создания  
 start\_date: Mapped[datetime | None] = mapped\_column(nullable=True) # Добавлено поле для даты начала  
  
 # Связи с другими моделями  
 order: Mapped["Order"] = relationship("Order", back\_populates="offers") # Заказ  
 executor: Mapped["User"] = relationship("User", back\_populates="offers") # Исполнитель

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\models\order.py

from sqlalchemy.orm import Mapped, mapped\_column, relationship  
from datetime import datetime  
from app.core.models.base import Base  
from decimal import Decimal  
from sqlalchemy import ForeignKey, Numeric, Enum  
import enum  
  
class OrderStatus(str, enum.Enum):  
 PENDING = "В\_ожидании" # Приводим к верхнему регистру  
 IN\_PROGRESS = "В\_прогрессе"  
 COMPLETED = "Выполнен"  
 CANCELED = "Отменен"  
  
class Order(Base):  
 """Модель заказа."""  
 \_\_tablename\_\_ = "orders"  
  
 customer\_id: Mapped[int] = mapped\_column(ForeignKey("users.id", ondelete="CASCADE"), nullable=False) # ID заказчика  
 executor\_id: Mapped[int | None] = mapped\_column(ForeignKey("users.id", ondelete="SET NULL"), nullable=True) # ID исполнителя (опционально)  
 category\_id: Mapped[int] = mapped\_column(ForeignKey("categories.id", ondelete="CASCADE"), nullable=False) # ID категории  
 title: Mapped[str] = mapped\_column(nullable=False) # Название заказа  
 description: Mapped[str | None] = mapped\_column(nullable=True) # Описание заказа (опционально)  
 desired\_price: Mapped[Decimal] = mapped\_column(Numeric(10, 2), nullable=False) # Желаемая цена  
 due\_date: Mapped[datetime] = mapped\_column(nullable=False) # Срок выполнения  
 created\_at: Mapped[datetime] = mapped\_column(default=datetime.utcnow, nullable=False) # Дата создания  
 status: Mapped[OrderStatus] = mapped\_column(Enum(OrderStatus, name="orderstatus"), default=OrderStatus.PENDING,  
 nullable=True)  
 # Связи с другими моделями  
 customer: Mapped["User"] = relationship("User", foreign\_keys="Order.customer\_id", back\_populates="orders\_created") # Заказчик  
 executor: Mapped["User"] = relationship("User", foreign\_keys="Order.executor\_id", back\_populates="orders\_executed") # Исполнитель  
 category: Mapped["Category"] = relationship("Category") # Категория  
 offers: Mapped[list["Offer"]] = relationship("Offer", back\_populates="order") # Предложения по заказу  
 review: Mapped["Review"] = relationship("Review", back\_populates="order", uselist=False) # Отзыв по заказу

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\models\review.py

from sqlalchemy.orm import Mapped, mapped\_column, relationship  
from datetime import datetime  
from app.core.models.base import Base  
from sqlalchemy import ForeignKey, CheckConstraint  
  
class Review(Base):  
 """Модель отзыва."""  
 \_\_tablename\_\_ = "reviews"  
  
 order\_id: Mapped[int] = mapped\_column(ForeignKey("orders.id", ondelete="CASCADE"), nullable=False) # ID заказа  
 author\_id: Mapped[int] = mapped\_column(ForeignKey("users.id", ondelete="CASCADE"), nullable=False) # ID автора отзыва  
 target\_id: Mapped[int] = mapped\_column(ForeignKey("users.id", ondelete="CASCADE"), nullable=False) # ID получателя отзыва  
 rating: Mapped[int] = mapped\_column(CheckConstraint("rating BETWEEN 1 AND 5"), nullable=False) # Рейтинг (от 1 до 5)  
 comment: Mapped[str | None] = mapped\_column(nullable=True) # Комментарий (опционально)  
 created\_at: Mapped[datetime] = mapped\_column(default=datetime.utcnow, nullable=False) # Дата создания  
  
 # Связи с другими моделями  
 order: Mapped["Order"] = relationship("Order", back\_populates="review") # Заказ  
 author: Mapped["User"] = relationship("User", foreign\_keys="Review.author\_id", back\_populates="reviews\_written") # Автор  
 target: Mapped["User"] = relationship("User", foreign\_keys="Review.target\_id", back\_populates="reviews\_received") # Получатель

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\models\user.py

from sqlalchemy.orm import Mapped, mapped\_column, relationship  
from app.core.models.base import Base  
from decimal import Decimal  
from sqlalchemy import ForeignKey, Numeric, CheckConstraint  
  
class User(Base):  
 """Модель пользователя."""  
 \_\_tablename\_\_ = "users"  
  
 telegram\_id: Mapped[int] = mapped\_column(unique=True, nullable=False) # Уникальный Telegram ID  
 name: Mapped[str] = mapped\_column(nullable=False) # Имя пользователя  
 username: Mapped[str | None] = mapped\_column(unique=True, nullable=True) # Уникальное имя в Telegram (опционально)  
 is\_customer: Mapped[bool] = mapped\_column(default=False) # Является ли заказчиком  
 is\_executor: Mapped[bool] = mapped\_column(default=False) # Является ли исполнителем  
 is\_admin: Mapped[bool] = mapped\_column(default=False) # Является ли администратором  
 city\_id: Mapped[int] = mapped\_column(ForeignKey("cities.id", ondelete="CASCADE"), nullable=False) # ID города  
 rating: Mapped[Decimal] = mapped\_column(Numeric(2, 1), default=0.0, nullable=False) # Рейтинг пользователя  
 completed\_orders: Mapped[int] = mapped\_column(default=0, nullable=False) # Количество завершенных заказов  
  
 # Связи с другими моделями  
 city: Mapped["City"] = relationship("City", back\_populates="users") # Связь с городом  
 categories: Mapped[list["Category"]] = relationship(  
 "Category", secondary="user\_categories", back\_populates="users" # Связь с категориями через таблицу user\_categories  
 )  
 orders\_created: Mapped[list["Order"]] = relationship(  
 "Order", foreign\_keys="Order.customer\_id", back\_populates="customer" # Заказы, созданные пользователем  
 )  
 orders\_executed: Mapped[list["Order"]] = relationship(  
 "Order", foreign\_keys="Order.executor\_id", back\_populates="executor" # Заказы, выполненные пользователем  
 )  
 offers: Mapped[list["Offer"]] = relationship("Offer", back\_populates="executor") # Предложения пользователя  
 reviews\_received: Mapped[list["Review"]] = relationship(  
 "Review", foreign\_keys="Review.target\_id", back\_populates="target" # Полученные отзывы  
 )  
 reviews\_written: Mapped[list["Review"]] = relationship(  
 "Review", foreign\_keys="Review.author\_id", back\_populates="author" # Написанные отзывы  
 )  
  
 \_\_table\_args\_\_ = (  
 CheckConstraint("NOT (is\_customer AND is\_executor)", name="check\_role\_exclusivity"), # Проверка: нельзя быть заказчиком и исполнителем одновременно  
 )

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\models\\_\_init\_\_.py

from .base import Base  
from .user import User  
from .city import City  
from .category import Category  
from .association import user\_categories  
from .order import Order  
from .offer import Offer  
from .review import Review

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\schemas\base.py

from pydantic import BaseModel  
  
class BaseSchema(BaseModel):  
 model\_config = {  
 "from\_attributes": True,  
 "populate\_by\_name": True,  
 # Убираем alias\_generator=camelize  
 }

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\schemas\category.py

from app.core.schemas.base import BaseSchema  
from typing import Optional  
  
  
class CategoryRead(BaseSchema):  
 id: int  
 name: str  
  
  
class CategoryCreate(BaseSchema):  
 name: str  
  
 model\_config = {"str\_strip\_whitespace": True}  
  
  
class CategoryUpdate(BaseSchema):  
 name: Optional[str] = None  
  
 model\_config = {"str\_strip\_whitespace": True}

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\schemas\city.py

from app.core.schemas.base import BaseSchema  
from typing import Optional  
  
  
class CityRead(BaseSchema):  
 id: int  
 name: str  
  
  
class CityCreate(BaseSchema):  
 name: str  
  
 model\_config = {"str\_strip\_whitespace": True}  
  
  
class CityUpdate(BaseSchema):  
 name: Optional[str] = None  
  
 model\_config = {"str\_strip\_whitespace": True}

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\schemas\offer.py

from typing import Optional  
from datetime import datetime  
from app.core.schemas.base import BaseSchema  
import enum  
  
class OfferStatus(str, enum.Enum):  
 PENDING = "pending"  
 ACCEPTED = "accepted"  
 REJECTED = "rejected"  
  
class OfferRead(BaseSchema):  
 id: int  
 order\_id: int  
 executor\_id: int  
 price: float  
 estimated\_time: int  
 status: OfferStatus  
 created\_at: datetime  
 start\_date: Optional[datetime] # Добавлено  
  
class OfferCreate(BaseSchema):  
 order\_id: int  
 price: float  
 estimated\_time: int  
 start\_date: Optional[datetime] # Добавлено  
  
 model\_config = {"str\_strip\_whitespace": True}  
  
class OfferUpdate(BaseSchema):  
 price: Optional[float] = None  
 estimated\_time: Optional[int] = None  
 status: Optional[OfferStatus] = None  
 start\_date: Optional[datetime] = None # Добавлено

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\schemas\order.py

from typing import Optional  
from datetime import datetime  
from app.core.schemas.base import BaseSchema  
import enum  
  
class OrderStatus(str, enum.Enum):  
 PENDING = "В\_ожидании" # Приводим к верхнему регистру  
 IN\_PROGRESS = "В\_прогрессе"  
 COMPLETED = "Выполнен"  
 CANCELED = "Отменен"  
  
class OrderRead(BaseSchema):  
 id: int  
 customer\_id: int  
 executor\_id: Optional[int]  
 category\_id: int  
 title: str  
 description: Optional[str]  
 desired\_price: float  
 due\_date: datetime  
 created\_at: datetime  
 status: OrderStatus  
  
class OrderCreate(BaseSchema):  
 category\_id: int  
 title: str  
 description: Optional[str] = None  
 desired\_price: float  
 due\_date: datetime  
  
 model\_config = {"str\_strip\_whitespace": True}  
  
class OrderUpdate(BaseSchema):  
 executor\_id: Optional[int] = None  
 status: Optional[OrderStatus] = None

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\schemas\review.py

from typing import Optional  
from datetime import datetime  
from app.core.schemas.base import BaseSchema  
  
class ReviewRead(BaseSchema):  
 id: int  
 order\_id: int  
 author\_id: int  
 target\_id: int  
 rating: int  
 comment: Optional[str]  
 created\_at: datetime  
  
class ReviewCreate(BaseSchema):  
 order\_id: int  
 target\_id: int  
 rating: int  
 comment: Optional[str] = None  
  
 model\_config = {"str\_strip\_whitespace": True}  
  
class ReviewUpdate(BaseSchema):  
 rating: Optional[int] = None  
 comment: Optional[str] = None  
  
 model\_config = {"str\_strip\_whitespace": True}

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\schemas\user.py

from typing import Optional, List  
from app.core.schemas.base import BaseSchema  
  
class UserRead(BaseSchema):  
 id: int  
 telegram\_id: int  
 name: str  
 username: Optional[str]  
 is\_customer: bool  
 is\_executor: bool  
 is\_admin: bool  
 city\_id: int  
 rating: float  
 completed\_orders: int  
  
class UserCreate(BaseSchema):  
 telegram\_id: int  
 name: str  
 username: Optional[str]  
 is\_customer: bool = False  
 is\_executor: bool = False  
 city\_id: int  
 category\_ids: Optional[List[int]] = None  
  
 model\_config = {"str\_strip\_whitespace": True}  
  
class UserUpdate(BaseSchema):  
 name: Optional[str] = None  
 username: Optional[str] = None  
 is\_customer: Optional[bool] = None  
 is\_executor: Optional[bool] = None  
 city\_id: Optional[int] = None  
 category\_ids: Optional[List[int]] = None  
  
 model\_config = {"str\_strip\_whitespace": True}

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\services\category.py

from fastapi.exceptions import HTTPException  
from sqlalchemy import select  
from sqlalchemy.orm import Session  
from sqlalchemy.exc import SQLAlchemyError, IntegrityError  
from app.core.models.category import Category  
from app.core.schemas.category import CategoryCreate, CategoryUpdate  
  
def create\_category(session: Session, data: CategoryCreate) -> Category:  
 """Создать новую категорию."""  
 category = Category(\*\*data.model\_dump())  
 session.add(category)  
 try:  
 session.commit()  
 session.refresh(category)  
 except IntegrityError:  
 session.rollback()  
 raise HTTPException(status\_code=400, detail="Категория с таким названием уже существует")  
 except SQLAlchemyError as e:  
 session.rollback()  
 raise HTTPException(status\_code=500, detail=f"Ошибка при создании категории: {e}")  
 return category  
  
def get\_category\_by\_id(session: Session, id: int) -> Category:  
 """Получить категорию по ID."""  
 category = session.get(Category, id)  
 if not category:  
 raise HTTPException(status\_code=404, detail="Категория не найдена")  
 return category  
  
def get\_all\_categories(session: Session) -> list[Category]:  
 """Получить список всех категорий."""  
 return session.scalars(select(Category)).all()  
  
def update\_category\_by\_id(session: Session, data: CategoryUpdate, id: int) -> Category:  
 """Обновить данные категории по ID."""  
 category = get\_category\_by\_id(session, id)  
 update\_data = data.model\_dump(exclude\_unset=True)  
 for key, value in update\_data.items():  
 setattr(category, key, value)  
 try:  
 session.commit()  
 session.refresh(category)  
 except IntegrityError:  
 session.rollback()  
 raise HTTPException(status\_code=400, detail="Категория с таким названием уже существует")  
 except SQLAlchemyError as e:  
 session.rollback()  
 raise HTTPException(status\_code=500, detail=f"Ошибка при обновлении категории: {e}")  
 return category  
  
def delete\_category\_by\_id(session: Session, id: int):  
 """Удалить категорию по ID."""  
 category = get\_category\_by\_id(session, id)  
 session.delete(category)  
 try:  
 session.commit()  
 except SQLAlchemyError as e:  
 session.rollback()  
 raise HTTPException(status\_code=500, detail=f"Ошибка при удалении категории: {e}")

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\services\city.py

from fastapi.exceptions import HTTPException  
from sqlalchemy import select  
from sqlalchemy.orm import Session  
from sqlalchemy.exc import SQLAlchemyError, IntegrityError  
from app.core.models.city import City  
from app.core.schemas.city import CityCreate, CityUpdate  
  
def create\_city(session: Session, data: CityCreate) -> City:  
 """Создать новый город."""  
 city = City(\*\*data.model\_dump())  
 session.add(city)  
 try:  
 session.commit()  
 session.refresh(city)  
 except IntegrityError:  
 session.rollback()  
 raise HTTPException(status\_code=400, detail="Город с таким названием уже существует")  
 except SQLAlchemyError as e:  
 session.rollback()  
 raise HTTPException(status\_code=500, detail=f"Ошибка при создании города: {e}")  
 return city  
  
def get\_city\_by\_id(session: Session, id: int) -> City:  
 """Получить город по ID."""  
 city = session.get(City, id)  
 if not city:  
 raise HTTPException(status\_code=404, detail="Город не найден")  
 return city  
  
def get\_all\_cities(session: Session) -> list[City]:  
 """Получить список всех городов."""  
 return session.scalars(select(City)).all()  
  
def update\_city\_by\_id(session: Session, data: CityUpdate, id: int) -> City:  
 """Обновить данные города по ID."""  
 city = get\_city\_by\_id(session, id)  
 update\_data = data.model\_dump(exclude\_unset=True)  
 for key, value in update\_data.items():  
 setattr(city, key, value)  
 try:  
 session.commit()  
 session.refresh(city)  
 except IntegrityError:  
 session.rollback()  
 raise HTTPException(status\_code=400, detail="Город с таким названием уже существует")  
 except SQLAlchemyError as e:  
 session.rollback()  
 raise HTTPException(status\_code=500, detail=f"Ошибка при обновлении города: {e}")  
 return city  
  
def delete\_city\_by\_id(session: Session, id: int):  
 """Удалить город по ID."""  
 city = get\_city\_by\_id(session, id)  
 session.delete(city)  
 try:  
 session.commit()  
 except SQLAlchemyError as e:  
 session.rollback()  
 raise HTTPException(status\_code=500, detail=f"Ошибка при удалении города: {e}")

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\services\offer.py

from fastapi.exceptions import HTTPException  
from sqlalchemy import select  
from sqlalchemy.orm import Session  
from sqlalchemy.exc import SQLAlchemyError  
from app.core.models.offer import Offer  
from app.core.models.order import Order  
from app.core.schemas.offer import OfferCreate, OfferUpdate  
  
def create\_offer(session: Session, data: OfferCreate, executor\_id: int) -> Offer:  
 """Создать новое предложение."""  
 order = session.get(Order, data.order\_id)  
 if not order:  
 raise HTTPException(status\_code=400, detail="Заказ не найден")  
 executor = session.get(User, executor\_id)  
 if order.customer\_id == executor\_id and not executor.is\_admin:  
 raise HTTPException(status\_code=400, detail="Самопредложение запрещено для не-администраторов")  
 offer\_data = data.model\_dump()  
 offer = Offer(\*\*offer\_data, executor\_id=executor\_id)  
 session.add(offer)  
 try:  
 session.commit()  
 session.refresh(offer)  
 except SQLAlchemyError as e:  
 session.rollback()  
 raise HTTPException(status\_code=500, detail=f"Ошибка при создании предложения: {e}")  
 return offer  
  
def get\_offer\_by\_id(session: Session, id: int) -> Offer:  
 """Получить предложение по ID."""  
 offer = session.get(Offer, id)  
 if not offer:  
 raise HTTPException(status\_code=404, detail="Предложение не найдено")  
 return offer  
  
def get\_offers\_by\_user(session: Session, user\_id: int) -> list[Offer]:  
 """Получить список предложений пользователя."""  
 stmt = select(Offer).where(Offer.executor\_id == user\_id)  
 return session.scalars(stmt).all()  
  
def update\_offer\_by\_id(session: Session, data: OfferUpdate, id: int) -> Offer:  
 """Обновить данные предложения по ID."""  
 offer = get\_offer\_by\_id(session, id)  
 update\_data = data.model\_dump(exclude\_unset=True)  
 for key, value in update\_data.items():  
 setattr(offer, key, value)  
 try:  
 session.commit()  
 session.refresh(offer)  
 except SQLAlchemyError as e:  
 session.rollback()  
 raise HTTPException(status\_code=500, detail=f"Ошибка при обновлении предложения: {e}")  
 return offer  
  
def delete\_offer\_by\_id(session: Session, id: int):  
 """Удалить предложение по ID."""  
 offer = get\_offer\_by\_id(session, id)  
 session.delete(offer)  
 try:  
 session.commit()  
 except SQLAlchemyError as e:  
 session.rollback()  
 raise HTTPException(status\_code=500, detail=f"Ошибка при удалении предложения: {e}")

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\services\order.py

from fastapi.exceptions import HTTPException  
from sqlalchemy import select  
from sqlalchemy.orm import Session  
from sqlalchemy.exc import SQLAlchemyError  
from app.core.models.order import Order  
from app.core.schemas.order import OrderCreate, OrderUpdate  
from app.core.services.category import get\_category\_by\_id  
  
def create\_order(session: Session, data: OrderCreate, customer\_id: int) -> Order:  
 """Создать новый заказ."""  
 get\_category\_by\_id(session, data.category\_id) # Проверка существования категории  
 order\_data = data.model\_dump()  
 order = Order(\*\*order\_data, customer\_id=customer\_id)  
 session.add(order)  
 try:  
 session.commit()  
 session.refresh(order)  
 except SQLAlchemyError as e:  
 session.rollback()  
 raise HTTPException(status\_code=500, detail=f"Ошибка при создании заказа: {e}")  
 return order  
  
def get\_order\_by\_id(session: Session, id: int) -> Order:  
 """Получить заказ по ID."""  
 order = session.get(Order, id)  
 if not order:  
 raise HTTPException(status\_code=404, detail="Заказ не найден")  
 return order  
  
def get\_orders\_by\_user(session: Session, user\_id: int) -> list[Order]:  
 """Получить список заказов пользователя."""  
 stmt = select(Order).where((Order.customer\_id == user\_id) | (Order.executor\_id == user\_id))  
 return session.scalars(stmt).all()  
  
def get\_available\_orders(session: Session) -> list[Order]:  
 """Получить список всех заказов со статусом 'В\_ожидании'."""  
 stmt = select(Order).where(Order.status == "В\_ожидании", Order.executor\_id.is\_(None))  
 return session.scalars(stmt).all()  
  
def update\_order\_by\_id(session: Session, data: OrderUpdate, id: int) -> Order:  
 """Обновить данные заказа по ID."""  
 order = get\_order\_by\_id(session, id)  
 update\_data = data.model\_dump(exclude\_unset=True)  
 for key, value in update\_data.items():  
 setattr(order, key, value)  
 try:  
 session.commit()  
 session.refresh(order)  
 except SQLAlchemyError as e:  
 session.rollback()  
 raise HTTPException(status\_code=500, detail=f"Ошибка при обновлении заказа: {e}")  
 return order  
  
def delete\_order\_by\_id(session: Session, id: int):  
 """Удалить заказ по ID."""  
 order = get\_order\_by\_id(session, id)  
 session.delete(order)  
 try:  
 session.commit()  
 except SQLAlchemyError as e:  
 session.rollback()  
 raise HTTPException(status\_code=500, detail=f"Ошибка при удалении заказа: {e}")

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\services\review.py

from fastapi.exceptions import HTTPException  
from sqlalchemy import select  
from sqlalchemy.orm import Session  
from sqlalchemy.exc import SQLAlchemyError  
from app.core.models.review import Review  
from app.core.models.order import Order  
from app.core.schemas.review import ReviewCreate, ReviewUpdate  
  
def create\_review(session: Session, data: ReviewCreate, author\_id: int) -> Review:  
 """Создать новый отзыв."""  
 order = session.get(Order, data.order\_id)  
 if not order or order.customer\_id != author\_id or order.status != "completed":  
 raise HTTPException(status\_code=400, detail="Недопустимый или незавершенный заказ")  
 review\_data = data.model\_dump()  
 review = Review(\*\*review\_data, author\_id=author\_id)  
 session.add(review)  
 try:  
 session.commit()  
 session.refresh(review)  
 except SQLAlchemyError as e:  
 session.rollback()  
 raise HTTPException(status\_code=500, detail=f"Ошибка при создании отзыва: {e}")  
 return review  
  
def get\_review\_by\_id(session: Session, id: int) -> Review:  
 """Получить отзыв по ID."""  
 review = session.get(Review, id)  
 if not review:  
 raise HTTPException(status\_code=404, detail="Отзыв не найден")  
 return review  
  
def get\_reviews\_by\_user(session: Session, user\_id: int) -> list[Review]:  
 """Получить список отзывов пользователя."""  
 stmt = select(Review).where(Review.author\_id == user\_id)  
 return session.scalars(stmt).all()  
  
def update\_review\_by\_id(session: Session, data: ReviewUpdate, id: int) -> Review:  
 """Обновить данные отзыва по ID."""  
 review = get\_review\_by\_id(session, id)  
 update\_data = data.model\_dump(exclude\_unset=True)  
 for key, value in update\_data.items():  
 setattr(review, key, value)  
 try:  
 session.commit()  
 session.refresh(review)  
 except SQLAlchemyError as e:  
 session.rollback()  
 raise HTTPException(status\_code=500, detail=f"Ошибка при обновлении отзыва: {e}")  
 return review  
  
def delete\_review\_by\_id(session: Session, id: int):  
 """Удалить отзыв по ID."""  
 review = get\_review\_by\_id(session, id)  
 session.delete(review)  
 try:  
 session.commit()  
 except SQLAlchemyError as e:  
 session.rollback()  
 raise HTTPException(status\_code=500, detail=f"Ошибка при удалении отзыва: {e}")

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\core\services\user.py

from fastapi.exceptions import HTTPException  
from sqlalchemy import select  
from sqlalchemy.orm import Session  
from sqlalchemy.exc import SQLAlchemyError, IntegrityError  
from app.core.models.user import User  
from app.core.models.category import Category  
from app.core.schemas.user import UserCreate, UserUpdate  
from app.core.services.city import get\_city\_by\_id  
  
def create\_user(session: Session, data: UserCreate) -> User:  
 """Создать нового пользователя."""  
 from app.api.depends.user import ADMIN\_TELEGRAM\_ID # Импортируем здесь  
 get\_city\_by\_id(session, data.city\_id) # Проверка существования города  
 user\_data = data.model\_dump(exclude={"category\_ids"}) # Исключаем category\_ids из данных  
 # Устанавливаем is\_admin=True, если telegram\_id совпадает с ADMIN\_TELEGRAM\_ID  
 if user\_data["telegram\_id"] == ADMIN\_TELEGRAM\_ID:  
 user\_data["is\_admin"] = True  
 user = User(\*\*user\_data)  
 if data.category\_ids: # Если указаны категории  
 categories = session.query(Category).filter(Category.id.in\_(data.category\_ids)).all()  
 if len(categories) != len(data.category\_ids):  
 raise HTTPException(status\_code=404, detail="Одна или несколько категорий не найдены")  
 user.categories = categories  
 session.add(user)  
 try:  
 session.commit()  
 session.refresh(user)  
 except IntegrityError:  
 session.rollback()  
 raise HTTPException(status\_code=400, detail="Пользователь с таким telegram\_id или username уже существует")  
 except SQLAlchemyError as e:  
 session.rollback()  
 raise HTTPException(status\_code=500, detail=f"Ошибка при создании пользователя: {e}")  
 return user  
  
def get\_user\_by\_id(session: Session, id: int) -> User:  
 """Получить пользователя по ID."""  
 user = session.get(User, id)  
 if not user:  
 raise HTTPException(status\_code=404, detail="Пользователь не найден")  
 return user  
  
def get\_users(session: Session) -> list[User]:  
 """Получить список всех пользователей."""  
 return session.scalars(select(User)).all()  
  
def update\_user\_by\_id(session: Session, data: UserUpdate, id: int) -> User:  
 """Обновить данные пользователя по ID."""  
 from app.api.depends.user import ADMIN\_TELEGRAM\_ID # Импортируем здесь  
 user = get\_user\_by\_id(session, id)  
 update\_data = data.model\_dump(exclude\_unset=True, exclude\_none=True)  
 if "city\_id" in update\_data:  
 get\_city\_by\_id(session, data.city\_id) # Проверка существования города  
 if "category\_ids" in update\_data and data.category\_ids is not None:  
 categories = session.query(Category).filter(Category.id.in\_(data.category\_ids)).all()  
 if len(categories) != len(data.category\_ids):  
 raise HTTPException(status\_code=404, detail="Одна или несколько категорий не найдены")  
 user.categories = categories  
 del update\_data["category\_ids"]  
 for key, value in update\_data.items():  
 setattr(user, key, value)  
 # Синхронизируем is\_admin с ADMIN\_TELEGRAM\_ID  
 if user.telegram\_id == ADMIN\_TELEGRAM\_ID:  
 user.is\_admin = True  
 try:  
 session.commit()  
 session.refresh(user)  
 except SQLAlchemyError as e:  
 session.rollback()  
 raise HTTPException(status\_code=500, detail=f"Ошибка при обновлении пользователя: {e}")  
 return user  
  
def delete\_user\_by\_id(session: Session, id: int):  
 """Удалить пользователя по ID."""  
 user = get\_user\_by\_id(session, id)  
 session.delete(user)  
 try:  
 session.commit()  
 except SQLAlchemyError as e:  
 session.rollback()  
 raise HTTPException(status\_code=500, detail=f"Ошибка при удалении пользователя: {e}")