# 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\2025\_03\_18\_1309-21f870535022\_initial\_migration\_from\_current\_models.py

"""Initial migration from current models  
  
Revision ID: 21f870535022  
Revises:  
Create Date: 2025-03-18 13:09:44.028638  
  
"""  
  
from typing import Sequence, Union  
  
from alembic import op  
import sqlalchemy as sa  
  
  
# revision identifiers, used by Alembic.  
revision: str = "21f870535022"  
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."""  
 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.CheckConstraint(  
 "NOT (is\_customer AND is\_executor)", name="check\_role\_exclusivity"  
 ),  
 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(  
 "status",  
 sa.Enum(  
 "PENDING",  
 "IN\_PROGRESS",  
 "COMPLETED",  
 "CANCELED",  
 name="orderstatus",  
 ),  
 nullable=True,  
 ),  
 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("start\_date", sa.DateTime(), nullable=True),  
 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)  
  
  
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")

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\alembic\versions\2025\_03\_18\_1424-b2677e3291ae\_initial\_migration\_from\_current\_models2.py

"""Initial migration from current models2  
  
Revision ID: b2677e3291ae  
Revises: 21f870535022  
Create Date: 2025-03-18 14:24:18.465539  
  
"""  
  
from typing import Sequence, Union  
  
from alembic import op  
import sqlalchemy as sa  
  
  
# revision identifiers, used by Alembic.  
revision: str = "b2677e3291ae"  
down\_revision: Union[str, None] = "21f870535022"  
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\_18\_1502-3057d1616d62\_update\_order\_status\_enum.py

"""Update order status enum  
  
Revision ID: 3057d1616d62  
Revises: b2677e3291ae  
Create Date: 2025-03-18 15:02:00  
  
"""  
from alembic import op  
import sqlalchemy as sa  
  
revision = "3057d1616d62"  
down\_revision = "b2677e3291ae"  
branch\_labels = None  
depends\_on = None  
  
def upgrade():  
 # Создаем временный новый enum  
 new\_enum = sa.Enum("В\_ожидании", "В\_прогрессе", "Выполнен", "Отменен", name="orderstatus\_new")  
 new\_enum.create(op.get\_bind(), checkfirst=True)  
  
 # Добавляем временную колонку с новым enum  
 op.add\_column("orders", sa.Column("status\_new", new\_enum, nullable=True))  
  
 # Переносим данные со старого статуса на новый с явным приведением типов  
 op.execute("""  
 UPDATE orders  
 SET status\_new = CASE  
 WHEN status = 'PENDING' THEN 'В\_ожидании'::orderstatus\_new  
 WHEN status = 'IN\_PROGRESS' THEN 'В\_прогрессе'::orderstatus\_new  
 WHEN status = 'COMPLETED' THEN 'Выполнен'::orderstatus\_new  
 WHEN status = 'CANCELED' THEN 'Отменен'::orderstatus\_new  
 ELSE 'В\_ожидании'::orderstatus\_new  
 END  
 """)  
  
 # Удаляем старую колонку  
 op.drop\_column("orders", "status")  
  
 # Переименовываем новую колонку и делаем её NOT NULL  
 op.alter\_column("orders", "status\_new", new\_name="status", nullable=False, server\_default="В\_ожидании")  
  
 # Переименовываем enum  
 op.execute("ALTER TYPE orderstatus RENAME TO orderstatus\_old")  
 op.execute("ALTER TYPE orderstatus\_new RENAME TO orderstatus")  
 op.execute("DROP TYPE orderstatus\_old")  
  
def downgrade():  
 # Создаем старый enum  
 old\_enum = sa.Enum("PENDING", "IN\_PROGRESS", "COMPLETED", "CANCELED", name="orderstatus\_old")  
 old\_enum.create(op.get\_bind(), checkfirst=True)  
  
 # Добавляем временную колонку со старым enum  
 op.add\_column("orders", sa.Column("status\_old", old\_enum, nullable=True))  
  
 # Переносим данные обратно с явным приведением типов  
 op.execute("""  
 UPDATE orders  
 SET status\_old = CASE  
 WHEN status = 'В\_ожидании' THEN 'PENDING'::orderstatus\_old  
 WHEN status = 'В\_прогрессе' THEN 'IN\_PROGRESS'::orderstatus\_old  
 WHEN status = 'Выполнен' THEN 'COMPLETED'::orderstatus\_old  
 WHEN status = 'Отменен' THEN 'CANCELED'::orderstatus\_old  
 ELSE 'PENDING'::orderstatus\_old  
 END  
 """)  
  
 # Удаляем новую колонку  
 op.drop\_column("orders", "status")  
  
 # Переименовываем старую колонку и делаем её nullable  
 op.alter\_column("orders", "status\_old", new\_name="status", nullable=True, server\_default="PENDING")  
  
 # Переименовываем enum обратно  
 op.execute("ALTER TYPE orderstatus RENAME TO orderstatus\_new")  
 op.execute("ALTER TYPE orderstatus\_old RENAME TO orderstatus")  
 op.execute("DROP TYPE orderstatus\_new")

## 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  
import aiohttp  
from app.bot.config import BOT\_TOKEN  
  
router = APIRouter(prefix="/order", tags=["Order"])  
logger = logging.getLogger(\_\_name\_\_)  
  
  
@router.post("/", response\_model=OrderRead, status\_code=status.HTTP\_201\_CREATED)  
async 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}")  
  
 # Уведомление исполнителям (всем с подходящей категорией)  
 executors = session.query(User).filter(  
 User.is\_executor == True,  
 User.categories.any(id=data.category\_id)  
 ).all()  
 message = (  
 f"Новый заказ '{order.title}' (ID: {order.id}):\n"  
 f"Категория: {order.category.name}\n"  
 f"Желаемая цена: {order.desired\_price} тенге\n"  
 f"Срок: {order.due\_date.strftime('%Y-%m-%d %H:%M')}"  
 )  
 for executor in executors:  
 try:  
 await send\_telegram\_message(executor.telegram\_id, message)  
 except Exception as e:  
 logger.error(f"Ошибка отправки уведомления исполнителю {executor.id}: {e}")  
  
 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")  
  
 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)  
async 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, или исполнителю для завершения)."""  
 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 and order.executor\_id != current\_user.id:  
 logger.warning(f"Попытка обновления заказа ID {id} без прав: {current\_user.id}")  
 raise HTTPException(status\_code=403, detail="Нет прав для обновления этого заказа")  
  
 if data.status == "Выполнен" and order.executor\_id == current\_user.id:  
 # Завершение заказа исполнителем  
 if order.status != "В\_прогрессе":  
 raise HTTPException(status\_code=400, detail="Заказ можно завершить только из статуса 'В\_прогрессе'")  
 updated\_order = order\_service.update\_order\_by\_id(session, data, id)  
 customer = session.get(User, order.customer\_id)  
 message = (  
 f"Заказ '{order.title}' (ID: {id}) завершён исполнителем.\n"  
 f"Пожалуйста, оставьте отзыв."  
 )  
 try:  
 await send\_telegram\_message(customer.telegram\_id, message)  
 except Exception as e:  
 logger.error(f"Ошибка отправки уведомления заказчику {customer.id}: {e}")  
 return updated\_order  
 elif order.customer\_id == current\_user.id:  
 if data.status and data.status not in ["В\_ожидании", "Отменен"]:  
 raise HTTPException(status\_code=403,  
 detail="Заказчик может менять статус только на 'В\_ожидании' или 'Отменен'")  
 updated\_order = order\_service.update\_order\_by\_id(session, data, id)  
 if data.status == "Отменен" and order.executor\_id:  
 executor = session.get(User, order.executor\_id)  
 message = (  
 f"Заказ '{order.title}' (ID: {id}) был отменён заказчиком."  
 )  
 try:  
 await send\_telegram\_message(executor.telegram\_id, message)  
 except Exception as e:  
 logger.error(f"Ошибка отправки уведомления исполнителю {executor.id}: {e}")  
 logger.info(f"Заказ ID {id} обновлён")  
 return updated\_order  
 else:  
 raise HTTPException(status\_code=403, detail="Нет прав для обновления этого заказа")  
  
  
@router.delete("/{id}", status\_code=status.HTTP\_204\_NO\_CONTENT)  
async 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":  
 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)  
 if order.executor\_id:  
 executor = session.get(User, order.executor\_id)  
 message = f"Заказ '{order.title}' (ID: {id}) был удалён заказчиком."  
 try:  
 await send\_telegram\_message(executor.telegram\_id, message)  
 except Exception as e:  
 logger.error(f"Ошибка отправки уведомления исполнителю {executor.id}: {e}")  
 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)  
async 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)  
 if order.executor\_id:  
 executor = session.get(User, order.executor\_id)  
 message = f"Заказ '{order.title}' (ID: {id}) был отменён заказчиком в течение 5 минут."  
 try:  
 await send\_telegram\_message(executor.telegram\_id, message)  
 except Exception as e:  
 logger.error(f"Ошибка отправки уведомления исполнителю {executor.id}: {e}")  
 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 != "В\_ожидании":  
 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\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  
from app.api.offer import send\_telegram\_message  
import logging  
  
router = APIRouter(prefix="/review", tags=["Review"])  
logger = logging.getLogger(\_\_name\_\_)  
  
  
@router.post("/", response\_model=ReviewRead, status\_code=status.HTTP\_201\_CREATED)  
async 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")  
 review = review\_service.create\_review(session, data, user.id)  
  
 # Уведомление получателю отзыва  
 target = session.get(User, data.target\_id)  
 message = (  
 f"Вы получили новый отзыв по заказу ID {data.order\_id}:\n"  
 f"Рейтинг: {data.rating}/5\n"  
 f"Комментарий: {data.comment or 'Без комментария'}"  
 )  
 try:  
 await send\_telegram\_message(target.telegram\_id, message)  
 except Exception as e:  
 logger.error(f"Ошибка отправки уведомления пользователю {target.id}: {e}")  
  
 return review  
  
  
@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)  
async 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")  
 updated\_review = review\_service.update\_review\_by\_id(session, data, id)  
  
 # Уведомление получателю об изменении отзыва  
 target = session.get(User, review.target\_id)  
 message = (  
 f"Отзыв по заказу ID {review.order\_id} был обновлён:\n"  
 f"Рейтинг: {updated\_review.rating}/5\n"  
 f"Комментарий: {updated\_review.comment or 'Без комментария'}"  
 )  
 try:  
 await send\_telegram\_message(target.telegram\_id, message)  
 except Exception as e:  
 logger.error(f"Ошибка отправки уведомления пользователю {target.id}: {e}")  
  
 return updated\_review  
  
  
@router.delete("/{id}", status\_code=status.HTTP\_204\_NO\_CONTENT)  
async def delete\_review(  
 id: int,  
 admin: Annotated[User, Depends(get\_admin\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 """Удалить отзыв (доступно только администратору)."""  
 review = review\_service.get\_review\_by\_id(session, id)  
 target = session.get(User, review.target\_id)  
 review\_service.delete\_review\_by\_id(session, id)  
  
 # Уведомление получателю об удалении отзыва  
 message = f"Ваш отзыв по заказу ID {review.order\_id} был удалён администратором."  
 try:  
 await send\_telegram\_message(target.telegram\_id, message)  
 except Exception as e:  
 logger.error(f"Ошибка отправки уведомления пользователю {target.id}: {e}")

## 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 UserRead.from\_orm(user) # Используем from\_orm для корректной сериализации  
  
@router.get("/me", response\_model=UserRead)  
def get\_me(user: Annotated[User, Depends(get\_current\_user)]):  
 """Получить данные текущего пользователя."""  
 return UserRead.from\_orm(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 [UserRead.from\_orm(user) for user in 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."""  
 user = user\_service.create\_user(session, data)  
 return UserRead.from\_orm(user)  
  
@router.patch("/me", response\_model=UserRead)  
def update\_me(  
 data: UserUpdate,  
 user: Annotated[User, Depends(get\_current\_user)],  
 session: Annotated[Session, Depends(get\_session)],  
):  
 updated\_user = user\_service.update\_user\_by\_id(session, data, user.id)  
 return UserRead.from\_orm(updated\_user)  
  
@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)."""  
 updated\_user = user\_service.update\_user\_by\_id(session, data, id)  
 return UserRead.from\_orm(updated\_user)  
  
@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:8019/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()}")  
 text = await response.text()  
 if not text:  
 raise Exception("Сервер вернул пустой ответ")  
 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  
from aiogram.types import ReplyKeyboardMarkup, KeyboardButton  
  
def get\_main\_keyboard(roles: dict = None) -> ReplyKeyboardMarkup:  
 roles = roles or {}  
 buttons = [  
 [KeyboardButton(text="Профиль"), KeyboardButton(text="Список заказов")],  
 [KeyboardButton(text="Сменить роль")]  
 ]  
 # Кнопки для заказчиков  
 if roles.get("is\_customer"):  
 buttons[0].insert(1, KeyboardButton(text="Создать заказ"))  
 buttons.append([KeyboardButton(text="Отменить заказ"), KeyboardButton(text="Редактировать заказ")])  
 buttons.append([KeyboardButton(text="Удалить заказ")])  
 buttons.append([KeyboardButton(text="Посмотреть предложения")])  
 buttons.append([KeyboardButton(text="Оставить отзыв")])  
 # Кнопки для исполнителей  
 if roles.get("is\_executor"):  
 buttons.append([KeyboardButton(text="Создать предложение")])  
 buttons.append([KeyboardButton(text="Список доступных заказов")])  
 if roles.get("is\_admin"):  
 buttons.append([KeyboardButton(text="Админ панель")])  
 return ReplyKeyboardMarkup(keyboard=buttons, resize\_keyboard=True)  
  
# 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 aiogram.fsm.state import State, StatesGroup  
from aiogram.fsm.context import FSMContext  
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\_\_)  
  
# Определяем состояния для редактирования профиля  
class ProfileEditStates(StatesGroup):  
 waiting\_for\_field = State()  
 waiting\_for\_name = State()  
 waiting\_for\_city = State()  
 waiting\_for\_category = State()  
  
@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! Выберите действие в меню ниже:", reply\_markup=get\_main\_keyboard(roles))  
  
@router.message(F.text == "Профиль")  
async def show\_profile(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 user = await api\_request("GET", f"{API\_URL}user/by\_telegram\_id/{telegram\_id}", telegram\_id)  
 city = await api\_request("GET", f"{API\_URL}city/{user['city\_id']}", telegram\_id)  
 all\_categories = await api\_request("GET", f"{API\_URL}category/", telegram\_id)  
 category\_ids = user.get('category\_ids', []) or []  
 categories = [cat['name'] for cat in all\_categories if cat['id'] in category\_ids]  
  
 # Формируем строку роли  
 roles = []  
 if user['is\_admin']:  
 roles.append("Администратор")  
 if user['is\_customer']:  
 roles.append("Заказчик")  
 if user['is\_executor']:  
 roles.append("Исполнитель")  
 role\_text = ", ".join(roles) if roles else "Не определена"  
  
 profile\_text = (  
 f"Ваш профиль:\n\n"  
 f"Telegram ID: {user['telegram\_id']}\n"  
 f"Имя: {user['name']}\n"  
 f"Username: @{user['username'] if user['username'] else 'Не указан'}\n"  
 f"Город: {city['name']}\n"  
 f"Категории: {', '.join(categories) if categories else 'Не указаны'}\n"  
 f"Роль: {role\_text}\n"  
 f"Рейтинг: {user['rating']}\n"  
 f"Завершенные заказы: {user['completed\_orders']}"  
 )  
  
 edit\_keyboard = InlineKeyboardMarkup(inline\_keyboard=[  
 [  
 InlineKeyboardButton(text="Изменить имя", callback\_data="edit\_name"),  
 InlineKeyboardButton(text="Изменить город", callback\_data="edit\_city")  
 ],  
 [  
 InlineKeyboardButton(text="Изменить категории", callback\_data="edit\_categories"),  
 InlineKeyboardButton(text="Назад", callback\_data="back\_to\_main")  
 ]  
 ])  
  
 await message.answer(profile\_text, reply\_markup=edit\_keyboard)  
 except Exception as e:  
 logger.error(f"Ошибка при отображении профиля: {e}")  
 roles = await get\_user\_roles(telegram\_id)  
 await message.answer(  
 f"Ошибка при загрузке профиля: {e}",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
  
@router.callback\_query(F.data == "edit\_name")  
async def start\_edit\_name(callback: CallbackQuery, state: FSMContext):  
 await callback.message.answer("Введите новое имя:")  
 await state.set\_state(ProfileEditStates.waiting\_for\_name)  
 await callback.answer()  
  
@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("В системе нет городов. Обратитесь к администратору.")  
 return  
 cities\_list = "\n".join([f"ID: {city['id']} - {city['name']}" for city in cities])  
 await callback.message.answer(f"Доступные города:\n{cities\_list}\n\nВведите ID нового города:")  
 await state.set\_state(ProfileEditStates.waiting\_for\_city)  
 except Exception as e:  
 logger.error(f"Ошибка при загрузке городов: {e}")  
 await callback.message.answer(f"Ошибка: {e}")  
 await callback.answer()  
  
@router.callback\_query(F.data == "edit\_categories")  
async def start\_edit\_categories(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("В системе нет категорий. Обратитесь к администратору.")  
 return  
 categories\_list = "\n".join([f"ID: {cat['id']} - {cat['name']}" for cat in categories])  
 await callback.message.answer(  
 f"Доступные категории:\n{categories\_list}\n\n"  
 f"Введите ID категорий через запятую (например: 1, 2, 3):"  
 )  
 await state.set\_state(ProfileEditStates.waiting\_for\_category)  
 except Exception as e:  
 logger.error(f"Ошибка при загрузке категорий: {e}")  
 await callback.message.answer(f"Ошибка: {e}")  
 await callback.answer()  
  
@router.message(ProfileEditStates.waiting\_for\_name)  
async def process\_name\_change(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 new\_name = message.text.strip()  
 try:  
 update\_data = {"name": new\_name}  
 await api\_request("PATCH", f"{API\_URL}user/me", telegram\_id, data=update\_data)  
 await message.answer(  
 f"Имя успешно изменено на '{new\_name}'. Выберите действие в меню ниже:",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 await show\_profile(message, state) # Показываем обновленный профиль  
 except Exception as e:  
 logger.error(f"Ошибка при изменении имени: {e}")  
 roles = await get\_user\_roles(telegram\_id)  
 await message.answer(  
 f"Ошибка при изменении имени: {e}. Выберите действие в меню ниже:",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 await state.clear()  
  
@router.message(ProfileEditStates.waiting\_for\_city)  
async def process\_city\_change(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 city\_id = int(message.text.strip())  
 update\_data = {"city\_id": city\_id}  
 await api\_request("PATCH", f"{API\_URL}user/me", telegram\_id, data=update\_data)  
 await message.answer(  
 "Город успешно изменён. Выберите действие в меню ниже:",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 await show\_profile(message, state) # Показываем обновленный профиль  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID города.")  
 except Exception as e:  
 logger.error(f"Ошибка при изменении города: {e}")  
 roles = await get\_user\_roles(telegram\_id)  
 await message.answer(  
 f"Ошибка: {e}. Выберите действие в меню ниже:",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 await state.clear()  
  
@router.message(ProfileEditStates.waiting\_for\_category)  
async def process\_category\_change(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 category\_ids = [int(cat\_id.strip()) for cat\_id in message.text.split(",")]  
 update\_data = {"category\_ids": category\_ids}  
 await api\_request("PATCH", f"{API\_URL}user/me", telegram\_id, data=update\_data)  
 await message.answer(  
 "Категории успешно изменены. Выберите действие в меню ниже:",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 await show\_profile(message, state) # Показываем обновленный профиль  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректные ID категорий через запятую.")  
 except Exception as e:  
 logger.error(f"Ошибка при изменении категорий: {e}")  
 roles = await get\_user\_roles(telegram\_id)  
 await message.answer(  
 f"Ошибка: {e}. Выберите действие в меню ниже:",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 await state.clear()  
  
@router.callback\_query(F.data == "back\_to\_main")  
async def back\_to\_main(callback: CallbackQuery):  
 telegram\_id = callback.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 await callback.message.edit\_text(  
 "Выберите действие в меню ниже:",  
 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, CallbackQuery, InlineKeyboardMarkup, InlineKeyboardButton, ReplyKeyboardRemove  
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\_\_)  
  
@router.message(F.text == "Сменить роль")  
async def switch\_role\_start(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 keyboard = InlineKeyboardMarkup(inline\_keyboard=[  
 [  
 InlineKeyboardButton(text="Заказчик", callback\_data="role\_customer"),  
 InlineKeyboardButton(text="Исполнитель", callback\_data="role\_executor")  
 ],  
 [InlineKeyboardButton(text="Отмена", callback\_data="cancel")]  
 ])  
 await message.answer("Выберите новую роль:", reply\_markup=keyboard)  
  
@router.callback\_query(F.data.startswith("role\_"))  
async def switch\_role\_process(callback: CallbackQuery, state: FSMContext):  
 telegram\_id = callback.from\_user.id  
 role = callback.data.split("\_")[1]  
 try:  
 if role == "customer":  
 user\_data = {"is\_customer": True, "is\_executor": False}  
 elif role == "executor":  
 user\_data = {"is\_customer": False, "is\_executor": True}  
 else:  
 raise ValueError("Недопустимая роль")  
 await api\_request("PATCH", f"{API\_URL}user/me", telegram\_id, data=user\_data)  
 roles = await get\_user\_roles(telegram\_id)  
 # Удаляем inline-кнопки и возвращаем нижнюю панель  
 await callback.message.edit\_text(  
 f"Роль изменена на {'Заказчик' if role == 'customer' else 'Исполнитель'}!"  
 )  
 await callback.message.answer(  
 "Выберите действие в меню ниже:",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 except Exception as e:  
 logger.error(f"Ошибка смены роли: {e}")  
 roles = await get\_user\_roles(telegram\_id)  
 await callback.message.edit\_text(f"Ошибка: {e}")  
 await callback.message.answer(  
 "Выберите действие в меню ниже:",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 await callback.answer()  
  
@router.callback\_query(F.data == "cancel")  
async def switch\_role\_cancel(callback: CallbackQuery, state: FSMContext):  
 roles = await get\_user\_roles(callback.from\_user.id)  
 await callback.message.edit\_text("Действие отменено.")  
 await callback.message.answer(  
 "Выберите действие в меню ниже:",  
 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 == "list\_categories") # Новый обработчик  
async def list\_categories(callback: CallbackQuery):  
 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(), reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 except Exception as e:  
 logger.error(f"Ошибка в list\_categories: {e}")  
 await callback.message.answer(f"Ошибка загрузки категорий: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 await callback.answer()  
  
@router.callback\_query(F.data == "add\_category")  
async def start\_add\_category(callback: CallbackQuery, state: FSMContext):  
 telegram\_id = callback.from\_user.id  
 try:  
 # Показываем текущий список категорий перед добавлением  
 categories = await api\_request("GET", f"{API\_URL}category/", telegram\_id)  
 if not categories:  
 await callback.message.answer("Категорий нет.\n\nВведите название новой категории:")  
 else:  
 response = "Текущие категории:\n\n"  
 for category in categories:  
 response += f"ID: {category['id']} - {category['name']}\n"  
 response += "\nВведите название новой категории:"  
 await callback.message.answer(response.strip())  
 await state.set\_state(AdminCategoryStates.add\_category)  
 except Exception as e:  
 logger.error(f"Ошибка в start\_add\_category: {e}")  
 await callback.message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard({"is\_admin": True}))  
 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, get\_user\_roles  
from app.bot.config import API\_URL, ADMIN\_TELEGRAM\_ID  
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  
 roles = await get\_user\_roles(telegram\_id)  
 if not roles["is\_admin"] or telegram\_id != ADMIN\_TELEGRAM\_ID:  
 await callback.message.answer(  
 "Доступ только для администраторов.",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 return  
 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(roles)  
 )  
 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(roles))  
 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(roles)  
 )  
  
@router.callback\_query(F.data == "add\_city")  
async def start\_add\_city(callback: CallbackQuery, state: FSMContext):  
 telegram\_id = callback.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 if not roles["is\_admin"] or telegram\_id != ADMIN\_TELEGRAM\_ID:  
 await callback.message.answer(  
 "Доступ только для администраторов.",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 return  
 await callback.message.answer("Введите название нового города:")  
 await state.set\_state(AdminCityStates.add\_city)  
  
@router.message(AdminCityStates.add\_city)  
async def process\_add\_city(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 city\_name = message.text.strip()  
 if not city\_name:  
 await message.answer("Название города не может быть пустым.")  
 return  
 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(roles)  
 )  
 except Exception as e:  
 logger.error(f"Ошибка в process\_add\_city: {e}")  
 error\_msg = "Ошибка добавления города. Возможно, город с таким названием уже существует." if "уже существует" in str(e) else f"Ошибка: {e}"  
 await message.answer(error\_msg, reply\_markup=get\_main\_keyboard(roles))  
 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  
 roles = await get\_user\_roles(telegram\_id)  
 if not roles["is\_admin"] or telegram\_id != ADMIN\_TELEGRAM\_ID:  
 await callback.message.answer(  
 "Доступ только для администраторов.",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 return  
 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(roles)  
 )  
 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:",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 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(roles)  
 )  
  
@router.message(AdminCityStates.edit\_city\_select)  
async def process\_edit\_city\_select(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 try:  
 city\_id = int(message.text.strip())  
 city = await api\_request("GET", f"{API\_URL}city/{city\_id}", telegram\_id)  
 await state.update\_data(city\_id=city\_id, old\_name=city["name"])  
 await message.answer(  
 f"Текущее название: {city['name']}\nВведите новое название города:",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 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}")  
 error\_msg = "Город не найден." if "404" in str(e) else f"Ошибка: {e}"  
 await message.answer(error\_msg, reply\_markup=get\_main\_keyboard(roles))  
 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  
 roles = await get\_user\_roles(telegram\_id)  
 new\_name = message.text.strip()  
 if not new\_name:  
 await message.answer("Название города не может быть пустым.")  
 return  
 try:  
 data = await state.get\_data()  
 city\_id = data["city\_id"]  
 old\_name = data["old\_name"]  
 update\_data = {"name": new\_name}  
 updated\_city = 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 с '{old\_name}' на '{updated\_city['name']}'.",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 except Exception as e:  
 logger.error(f"Ошибка в process\_edit\_city\_name: {e}")  
 error\_msg = "Ошибка изменения города. Возможно, такое название уже существует." if "уже существует" in str(e) else f"Ошибка: {e}"  
 await message.answer(error\_msg, reply\_markup=get\_main\_keyboard(roles))  
 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  
 roles = await get\_user\_roles(telegram\_id)  
 if not roles["is\_admin"] or telegram\_id != ADMIN\_TELEGRAM\_ID:  
 await callback.message.answer(  
 "Доступ только для администраторов.",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 return  
 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(roles)  
 )  
 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:",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 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(roles)  
 )  
  
@router.message(AdminCityStates.delete\_city)  
async def process\_delete\_city(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 try:  
 city\_id = int(message.text.strip())  
 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(roles)  
 )  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID города.")  
 except Exception as e:  
 logger.error(f"Ошибка в process\_delete\_city: {e}")  
 error\_msg = "Город не найден или не может быть удалён (возможно, к нему привязаны пользователи)." if "404" in str(e) else f"Ошибка: {e}"  
 await message.answer(error\_msg, reply\_markup=get\_main\_keyboard(roles))  
 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="Список категорий", callback\_data="list\_categories"), # Новая кнопка  
 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, InlineKeyboardMarkup, InlineKeyboardButton, CallbackQuery  
from aiogram.fsm.context import FSMContext  
from aiogram.fsm.state import State, StatesGroup  
from app.bot.handlers.common import api\_request, get\_main\_keyboard, get\_user\_roles  
from app.bot.config import API\_URL, BOT\_TOKEN # Предполагается, что BOT\_TOKEN доступен в config  
import aiohttp  
from datetime import datetime, timedelta  
import logging  
  
router = Router()  
logger = logging.getLogger(\_\_name\_\_)  
  
  
# Состояния для управления заказами  
class OrderStates(StatesGroup):  
 select\_order\_edit = State() # Выбор заказа для редактирования  
 edit\_title = State() # Ввод нового названия  
 edit\_description = State() # Ввод нового описания  
 edit\_price = State() # Ввод новой цены  
 edit\_due\_date = State() # Ввод нового дедлайна  
 select\_order\_delete = State() # Выбор заказа для удаления  
 select\_order\_offers = State() # Выбор заказа для просмотра предложений  
 select\_order\_cancel = State() # Выбор заказа для отмены  
 select\_order\_review = State() # Выбор заказа для отзыва  
 input\_rating = State() # Ввод рейтинга для отзыва  
 input\_comment = State() # Ввод комментария для отзыва  
 offers\_filter = State() # Ввод фильтра для предложений  
 offers\_sort = State() # Ввод сортировки для предложений  
 chat\_with\_executor = State() # Ввод сообщения для исполнителя  
  
  
# Асинхронная функция для отправки сообщения через Telegram API  
async def send\_telegram\_message(chat\_id: int, text: str):  
 async with aiohttp.ClientSession() as session:  
 url = f"https://api.telegram.org/bot{BOT\_TOKEN}/sendMessage"  
 payload = {  
 "chat\_id": chat\_id,  
 "text": text  
 }  
 async with session.post(url, json=payload) as response:  
 if response.status != 200:  
 logger.error(f"Ошибка отправки сообщения в Telegram: {await response.text()}")  
 return False  
 return True  
  
  
# Команда "Создать заказ"  
@router.message(F.text == "Создать заказ")  
async def start\_create\_order(message: Message, state: FSMContext):  
 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  
 try:  
 cities = await api\_request("GET", f"{API\_URL}city/", telegram\_id)  
 categories = await api\_request("GET", f"{API\_URL}category/", telegram\_id)  
 if not cities or not categories:  
 await message.answer("Нет доступных городов или категорий.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 city\_list = "\n".join([f"{city['id']}: {city['name']}" for city in cities])  
 category\_list = "\n".join([f"{cat['id']}: {cat['name']}" for cat in categories])  
 await message.answer(  
 f"Доступные города:\n{city\_list}\n\nДоступные категории:\n{category\_list}\n\nВведите ID категории:"  
 )  
 await state.set\_state(OrderStates.edit\_title)  
 await state.update\_data(cities=cities, categories=categories, step="category")  
 except Exception as e:  
 logger.error(f"Ошибка при запуске создания заказа: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(roles))  
  
  
@router.message(OrderStates.edit\_title)  
async def process\_create\_order(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 data = await state.get\_data()  
 step = data.get("step", "category")  
  
 if step == "category":  
 try:  
 category\_id = int(message.text.strip())  
 if not any(cat["id"] == category\_id for cat in data["categories"]):  
 await message.answer("Неверный ID категории. Попробуйте снова:")  
 return  
 await state.update\_data(category\_id=category\_id)  
 await message.answer("Введите название заказа:")  
 await state.update\_data(step="title")  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID категории.")  
 elif step == "title":  
 await state.update\_data(title=message.text.strip())  
 await message.answer("Введите описание заказа (или /skip для пропуска):")  
 await state.update\_data(step="description")  
 elif step == "description":  
 if message.text.strip() != "/skip":  
 await state.update\_data(description=message.text.strip())  
 await message.answer("Введите желаемую цену (в тенге):")  
 await state.update\_data(step="price")  
 elif step == "price":  
 try:  
 price = float(message.text.strip())  
 if price <= 0:  
 await message.answer("Цена должна быть больше 0.")  
 return  
 await state.update\_data(desired\_price=price)  
 await message.answer("Введите дедлайн (гггг-мм-дд чч:мм):")  
 await state.update\_data(step="due\_date")  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректное число для цены.")  
 elif step == "due\_date":  
 try:  
 due\_date = datetime.strptime(message.text.strip(), "%Y-%m-%d %H:%M")  
 if due\_date < datetime.utcnow():  
 await message.answer("Дедлайн не может быть в прошлом.")  
 return  
 data = await state.get\_data()  
 order\_data = {  
 "category\_id": data["category\_id"],  
 "title": data["title"],  
 "description": data.get("description"),  
 "desired\_price": data["desired\_price"],  
 "due\_date": due\_date.isoformat()  
 }  
 order = await api\_request("POST", f"{API\_URL}order/", telegram\_id, data=order\_data)  
 await message.answer(  
 f"Заказ успешно создан!\nID: {order['id']}\nНазвание: {order['title']}",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 except ValueError:  
 await message.answer("Пожалуйста, введите дату в формате гггг-мм-дд чч:мм.")  
 except Exception as e:  
 logger.error(f"Ошибка при создании заказа: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
  
  
# Редактирование заказа  
@router.message(F.text == "Редактировать заказ")  
async def start\_edit\_order(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 if not roles["is\_customer"]:  
 await message.answer("Только заказчики могут редактировать заказы.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 try:  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 editable\_orders = [o for o in orders if o["status"] == "В\_ожидании" and o["customer\_id"] == telegram\_id]  
 if not editable\_orders:  
 await message.answer("У вас нет заказов, доступных для редактирования.",  
 reply\_markup=get\_main\_keyboard(roles))  
 return  
 response = "Ваши заказы в ожидании:\n\n"  
 for order in editable\_orders:  
 response += f"ID: {order['id']} - {order['title']}\n"  
 await message.answer(response + "\nВведите ID заказа для редактирования:")  
 await state.set\_state(OrderStates.select\_order\_edit)  
 except Exception as e:  
 logger.error(f"Ошибка загрузки заказов для редактирования: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(roles))  
  
  
@router.message(OrderStates.select\_order\_edit)  
async def process\_edit\_order\_selection(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 order\_id = int(message.text.strip())  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 order = next((o for o in orders if  
 o["id"] == order\_id and o["status"] == "В\_ожидании" and o["customer\_id"] == telegram\_id), None)  
 if not order:  
 await message.answer("Заказ не найден или недоступен для редактирования.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
 return  
 await state.update\_data(order\_id=order\_id, current\_order=order)  
 await message.answer(f"Текущее название: {order['title']}\nВведите новое название (или /skip для пропуска):")  
 await state.set\_state(OrderStates.edit\_title)  
 await state.update\_data(step="edit\_title")  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID заказа.")  
 except Exception as e:  
 logger.error(f"Ошибка при выборе заказа для редактирования: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
  
  
@router.message(OrderStates.edit\_title, F.text.regexp(r"^(?!/skip).\*$"))  
async def process\_edit\_title(message: Message, state: FSMContext):  
 data = await state.get\_data()  
 step = data.get("step")  
 if step == "edit\_title":  
 await state.update\_data(title=message.text.strip())  
 await message.answer("Введите новое описание (или /skip для пропуска):")  
 await state.update\_data(step="edit\_description")  
 await state.set\_state(OrderStates.edit\_description)  
  
  
@router.message(OrderStates.edit\_description)  
async def process\_edit\_description(message: Message, state: FSMContext):  
 if message.text.strip() != "/skip":  
 await state.update\_data(description=message.text.strip())  
 await message.answer("Введите новую желаемую цену (в тенге, или /skip для пропуска):")  
 await state.set\_state(OrderStates.edit\_price)  
  
  
@router.message(OrderStates.edit\_price)  
async def process\_edit\_price(message: Message, state: FSMContext):  
 text = message.text.strip()  
 if text != "/skip":  
 try:  
 price = float(text)  
 if price <= 0:  
 await message.answer("Цена должна быть больше 0.")  
 return  
 await state.update\_data(desired\_price=price)  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректное число для цены.")  
 return  
 await message.answer("Введите новый дедлайн (гггг-мм-дд чч:мм, или /skip для пропуска):")  
 await state.set\_state(OrderStates.edit\_due\_date)  
  
  
@router.message(OrderStates.edit\_due\_date)  
async def process\_edit\_due\_date(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 text = message.text.strip()  
 data = await state.get\_data()  
 order\_id = data["order\_id"]  
 update\_data = {}  
 if text != "/skip":  
 try:  
 due\_date = datetime.strptime(text, "%Y-%m-%d %H:%M")  
 if due\_date < datetime.utcnow():  
 await message.answer("Дедлайн не может быть в прошлом.")  
 return  
 update\_data["due\_date"] = due\_date.isoformat()  
 except ValueError:  
 await message.answer("Пожалуйста, введите дату в формате гггг-мм-дд чч:мм.")  
 return  
 if "title" in data:  
 update\_data["title"] = data["title"]  
 if "description" in data:  
 update\_data["description"] = data["description"]  
 if "desired\_price" in data:  
 update\_data["desired\_price"] = data["desired\_price"]  
  
 if update\_data:  
 try:  
 updated\_order = await api\_request("PATCH", f"{API\_URL}order/{order\_id}", telegram\_id, data=update\_data)  
 await message.answer(  
 f"Заказ ID {order\_id} успешно обновлен!\n"  
 f"Название: {updated\_order['title']}\n"  
 f"Описание: {updated\_order['description'] or 'Нет'}\n"  
 f"Цена: {updated\_order['desired\_price']} тенге\n"  
 f"Дедлайн: {updated\_order['due\_date']}",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 except Exception as e:  
 logger.error(f"Ошибка при обновлении заказа: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 else:  
 await message.answer("Ничего не изменено.", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
  
  
# Удаление заказа  
@router.message(F.text == "Удалить заказ")  
async def start\_delete\_order(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 if not roles["is\_customer"]:  
 await message.answer("Только заказчики могут удалять заказы.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 try:  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 deletable\_orders = [o for o in orders if o["status"] == "В\_ожидании" and o["customer\_id"] == telegram\_id]  
 if not deletable\_orders:  
 await message.answer("У вас нет заказов, доступных для удаления.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 response = "Ваши заказы в ожидании:\n\n"  
 for order in deletable\_orders:  
 response += f"ID: {order['id']} - {order['title']}\n"  
 await message.answer(response + "\nВведите ID заказа для удаления:")  
 await state.set\_state(OrderStates.select\_order\_delete)  
 except Exception as e:  
 logger.error(f"Ошибка загрузки заказов для удаления: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(roles))  
  
  
@router.message(OrderStates.select\_order\_delete)  
async def process\_delete\_order(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 order\_id = int(message.text.strip())  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 order = next((o for o in orders if  
 o["id"] == order\_id and o["status"] == "В\_ожидании" and o["customer\_id"] == telegram\_id), None)  
 if not order:  
 await message.answer("Заказ не найден или недоступен для удаления.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
 return  
 await api\_request("DELETE", f"{API\_URL}order/{order\_id}", telegram\_id)  
 await message.answer(f"Заказ ID {order\_id} успешно удален!",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID заказа.")  
 except Exception as e:  
 logger.error(f"Ошибка при удалении заказа: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
  
  
# Просмотр предложений  
@router.message(F.text == "Посмотреть предложения")  
async def start\_view\_offers(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 if not roles["is\_customer"]:  
 await message.answer("Только заказчики могут просматривать предложения.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 try:  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 pending\_orders = [o for o in orders if o["status"] == "В\_ожидании" and o["customer\_id"] == telegram\_id]  
 if not pending\_orders:  
 await message.answer("У вас нет заказов в ожидании с предложениями.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 response = "Ваши заказы в ожидании:\n\n"  
 for order in pending\_orders:  
 response += f"ID: {order['id']} - {order['title']}\n"  
 await message.answer(response + "\nВведите ID заказа для просмотра предложений:")  
 await state.set\_state(OrderStates.select\_order\_offers)  
 except Exception as e:  
 logger.error(f"Ошибка загрузки заказов для просмотра предложений: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(roles))  
  
  
@router.message(OrderStates.select\_order\_offers)  
async def process\_select\_order\_offers(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 order\_id = int(message.text.strip())  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 order = next((o for o in orders if  
 o["id"] == order\_id and o["status"] == "В\_ожидании" and o["customer\_id"] == telegram\_id), None)  
 if not order:  
 await message.answer("Заказ не найден или недоступен.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
 return  
 await state.update\_data(order\_id=order\_id)  
 await message.answer(  
 "Выберите фильтр для предложений:\n"  
 "1. Все\n"  
 "2. Только pending\n"  
 "3. Только accepted\n"  
 "4. Только rejected\n"  
 "Введите номер (или /skip для всех):"  
 )  
 await state.set\_state(OrderStates.offers\_filter)  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID заказа.")  
 except Exception as e:  
 logger.error(f"Ошибка при выборе заказа для предложений: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
  
  
@router.message(OrderStates.offers\_filter)  
async def process\_offers\_filter(message: Message, state: FSMContext):  
 text = message.text.strip()  
 filter\_map = {"1": None, "2": "pending", "3": "accepted", "4": "rejected"}  
 if text == "/skip":  
 await state.update\_data(filter\_status=None)  
 elif text in filter\_map:  
 await state.update\_data(filter\_status=filter\_map[text])  
 else:  
 await message.answer("Пожалуйста, выберите корректный номер фильтра (1-4) или /skip.")  
 return  
 await message.answer(  
 "Выберите сортировку:\n"  
 "1. Без сортировки\n"  
 "2. По цене (возрастание)\n"  
 "3. По рейтингу исполнителя (убывание)\n"  
 "Введите номер (или /skip для без сортировки):"  
 )  
 await state.set\_state(OrderStates.offers\_sort)  
  
  
@router.message(OrderStates.offers\_sort)  
async def process\_offers\_sort(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 text = message.text.strip()  
 sort\_map = {"1": None, "2": "price", "3": "executor\_rating"}  
 if text == "/skip":  
 await state.update\_data(sort\_by=None)  
 elif text in sort\_map:  
 await state.update\_data(sort\_by=sort\_map[text])  
 else:  
 await message.answer("Пожалуйста, выберите корректный номер сортировки (1-3) или /skip.")  
 return  
  
 data = await state.get\_data()  
 order\_id = data["order\_id"]  
 filter\_status = data.get("filter\_status")  
 sort\_by = data.get("sort\_by")  
  
 try:  
 url = f"{API\_URL}order/{order\_id}/offers"  
 params = {}  
 if filter\_status:  
 params["status"] = filter\_status  
 if sort\_by:  
 params["sort\_by"] = sort\_by  
 offers = await api\_request("GET", url, telegram\_id, params=params if params else None)  
  
 if not offers:  
 await message.answer(f"По заказу ID {order\_id} нет предложений с выбранными фильтрами.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
 return  
  
 response = f"Предложения по заказу ID {order\_id}:\n\n"  
 inline\_buttons = []  
 for offer in offers:  
 response += (  
 f"ID: {offer['id']}\n"  
 f"Исполнитель: {offer['executor\_id']} (Рейтинг: {offer['executor\_rating']})\n"  
 f"Цена: {offer['price']} тенге\n"  
 f"Время: {offer['estimated\_time']} часов\n"  
 f"Статус: {offer['status']}\n\n"  
 )  
 buttons = []  
 if offer["status"] == "pending":  
 buttons.extend([  
 InlineKeyboardButton(text=f"Принять {offer['id']}",  
 callback\_data=f"accept\_offer\_{order\_id}\_{offer['id']}"),  
 InlineKeyboardButton(text=f"Отклонить {offer['id']}",  
 callback\_data=f"reject\_offer\_{order\_id}\_{offer['id']}")  
 ])  
 buttons.extend([  
 InlineKeyboardButton(text=f"Подробнее об исполнителе {offer['executor\_id']}",  
 callback\_data=f"executor\_info\_{offer['executor\_id']}"),  
 InlineKeyboardButton(text=f"Связаться с исполнителем {offer['executor\_id']}",  
 callback\_data=f"chat\_executor\_{order\_id}\_{offer['executor\_id']}")  
 ])  
 inline\_buttons.append(buttons)  
  
 inline\_kb = InlineKeyboardMarkup(inline\_keyboard=inline\_buttons)  
 await message.answer(response, reply\_markup=inline\_kb)  
 await state.clear()  
 except Exception as e:  
 logger.error(f"Ошибка при просмотре предложений: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
  
  
@router.callback\_query(lambda c: c.data.startswith("accept\_offer\_"))  
async def process\_accept\_offer(callback: CallbackQuery):  
 telegram\_id = callback.from\_user.id  
 \_, order\_id, offer\_id = callback.data.split("\_")  
 try:  
 order = await api\_request("GET", f"{API\_URL}order/{order\_id}", telegram\_id)  
 if order["customer\_id"] != telegram\_id:  
 await callback.message.edit\_text("Вы не можете принять предложение для этого заказа.")  
 await callback.answer()  
 return  
 updated\_order = await api\_request("POST", f"{API\_URL}offer/{order\_id}/offers/{offer\_id}/accept", telegram\_id)  
 await callback.message.edit\_text(  
 f"Предложение ID {offer\_id} принято! Заказ ID {order\_id} теперь в статусе 'В\_прогрессе'.",  
 reply\_markup=None  
 )  
 await callback.answer()  
 except Exception as e:  
 logger.error(f"Ошибка при принятии предложения: {e}")  
 await callback.message.edit\_text(f"Ошибка: {e}", reply\_markup=None)  
 await callback.answer()  
  
  
@router.callback\_query(lambda c: c.data.startswith("reject\_offer\_"))  
async def process\_reject\_offer(callback: CallbackQuery):  
 telegram\_id = callback.from\_user.id  
 \_, order\_id, offer\_id = callback.data.split("\_")  
 try:  
 order = await api\_request("GET", f"{API\_URL}order/{order\_id}", telegram\_id)  
 if order["customer\_id"] != telegram\_id:  
 await callback.message.edit\_text("Вы не можете отклонить предложение для этого заказа.")  
 await callback.answer()  
 return  
 updated\_offer = await api\_request("POST", f"{API\_URL}offer/{order\_id}/offers/{offer\_id}/reject", telegram\_id)  
 await callback.message.edit\_text(  
 f"Предложение ID {offer\_id} отклонено!",  
 reply\_markup=None  
 )  
 await callback.answer()  
 except Exception as e:  
 logger.error(f"Ошибка при отклонении предложения: {e}")  
 await callback.message.edit\_text(f"Ошибка: {e}", reply\_markup=None)  
 await callback.answer()  
  
  
@router.callback\_query(lambda c: c.data.startswith("executor\_info\_"))  
async def process\_executor\_info(callback: CallbackQuery):  
 telegram\_id = callback.from\_user.id  
 \_, executor\_id = callback.data.split("\_")  
 try:  
 executor = await api\_request("GET", f"{API\_URL}user/{executor\_id}", telegram\_id)  
 response = (  
 f"Информация об исполнителе ID {executor\_id}:\n"  
 f"Имя: {executor.get('name', 'Не указано')}\n"  
 f"Рейтинг: {executor.get('rating', 'Нет рейтинга')}\n"  
 f"Город: {executor.get('city\_id', 'Не указан')}\n"  
 f"Категории: {', '.join(str(cat) for cat in executor.get('category\_ids', [])) or 'Не указаны'}\n"  
 )  
 await callback.message.answer(response)  
 await callback.answer()  
 except Exception as e:  
 logger.error(f"Ошибка при запросе информации об исполнителе: {e}")  
 await callback.message.answer(f"Ошибка: {e}")  
 await callback.answer()  
  
  
@router.callback\_query(lambda c: c.data.startswith("chat\_executor\_"))  
async def start\_chat\_with\_executor(callback: CallbackQuery, state: FSMContext):  
 telegram\_id = callback.from\_user.id  
 \_, order\_id, executor\_id = callback.data.split("\_")  
 try:  
 order = await api\_request("GET", f"{API\_URL}order/{order\_id}", telegram\_id)  
 if order["customer\_id"] != telegram\_id:  
 await callback.message.edit\_text("Вы не можете связаться с исполнителем для этого заказа.")  
 await callback.answer()  
 return  
 await state.update\_data(order\_id=order\_id, executor\_id=executor\_id)  
 await callback.message.answer("Введите сообщение для исполнителя:")  
 await state.set\_state(OrderStates.chat\_with\_executor)  
 await callback.answer()  
 except Exception as e:  
 logger.error(f"Ошибка при запуске чата с исполнителем: {e}")  
 await callback.message.edit\_text(f"Ошибка: {e}", reply\_markup=None)  
 await callback.answer()  
  
  
@router.message(OrderStates.chat\_with\_executor)  
async def process\_chat\_message(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 data = await state.get\_data()  
 order\_id = data["order\_id"]  
 executor\_id = data["executor\_id"]  
  
 try:  
 executor = await api\_request("GET", f"{API\_URL}user/{executor\_id}", telegram\_id)  
 executor\_telegram\_id = executor.get("telegram\_id")  
 if not executor\_telegram\_id:  
 await message.answer("У исполнителя не указан Telegram ID.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
 return  
  
 customer = await api\_request("GET", f"{API\_URL}user/me", telegram\_id)  
 customer\_name = customer.get("name", "Заказчик")  
 message\_text = f"Сообщение от {customer\_name} (Заказ ID {order\_id}):\n{message.text.strip()}"  
  
 success = await send\_telegram\_message(executor\_telegram\_id, message\_text)  
 if success:  
 await message.answer(  
 f"Сообщение успешно отправлено исполнителю ID {executor\_id}!",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 else:  
 await message.answer(  
 "Не удалось отправить сообщение исполнителю.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 except Exception as e:  
 logger.error(f"Ошибка при отправке сообщения исполнителю: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
  
  
# Список заказов  
@router.message(F.text == "Список заказов")  
async def list\_orders(message: Message):  
 telegram\_id = message.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 try:  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 if not orders:  
 await message.answer("У вас нет заказов.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 response = "Ваши заказы:\n\n"  
 for order in orders:  
 status = {"В\_ожидании": "Ожидает", "В\_прогрессе": "В процессе", "Выполнен": "Завершён",  
 "Отменен": "Отменён"}.get(order["status"], order["status"])  
 response += f"ID: {order['id']} - {order['title']} ({status})\n"  
 await message.answer(response, reply\_markup=get\_main\_keyboard(roles))  
 except Exception as e:  
 logger.error(f"Ошибка загрузки заказов: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(roles))  
  
  
# Отмена заказа  
@router.message(F.text == "Отменить заказ")  
async def cancel\_order\_start(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 if not roles["is\_customer"]:  
 await message.answer("Только заказчики могут отменять заказы.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 try:  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 cancellable\_orders = [  
 o for o in orders  
 if o["status"] == "В\_ожидании" and o["customer\_id"] == telegram\_id and  
 (datetime.utcnow() - datetime.fromisoformat(o["created\_at"].replace("Z", "+00:00"))) < timedelta(  
 minutes=5)  
 ]  
 if not cancellable\_orders:  
 await message.answer("У вас нет заказов, доступных для отмены.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 response = "Ваши заказы, доступные для отмены:\n\n"  
 for order in cancellable\_orders:  
 response += f"ID: {order['id']} - {order['title']}\n"  
 await message.answer(response + "\nВведите ID заказа для отмены:")  
 await state.set\_state(OrderStates.select\_order\_cancel)  
 except Exception as e:  
 logger.error(f"Ошибка загрузки заказов для отмены: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(roles))  
  
  
@router.message(OrderStates.select\_order\_cancel)  
async def process\_cancel\_order(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 order\_id = int(message.text.strip())  
 canceled\_order = await api\_request("POST", f"{API\_URL}order/{order\_id}/cancel", telegram\_id)  
 await message.answer(f"Заказ ID {order\_id} успешно отменен!",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID заказа.")  
 except Exception as e:  
 logger.error(f"Ошибка при отмене заказа: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
  
  
# Оставить отзыв  
@router.message(F.text == "Оставить отзыв")  
async def start\_review(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 if not roles["is\_customer"]:  
 await message.answer("Только заказчики могут оставлять отзывы.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 try:  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 completed\_orders = [o for o in orders if o["status"] == "Выполнен" and o["customer\_id"] == telegram\_id]  
 if not completed\_orders:  
 await message.answer("У вас нет завершенных заказов для отзыва.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 response = "Ваши завершенные заказы:\n\n"  
 for order in completed\_orders:  
 response += f"ID: {order['id']} - {order['title']}\n"  
 await message.answer(response + "\nВведите ID заказа для отзыва:")  
 await state.set\_state(OrderStates.select\_order\_review)  
 except Exception as e:  
 logger.error(f"Ошибка загрузки заказов для отзыва: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(roles))  
  
  
@router.message(OrderStates.select\_order\_review)  
async def process\_review\_order\_selection(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 order\_id = int(message.text.strip())  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 order = next((o for o in orders if  
 o["id"] == order\_id and o["status"] == "Выполнен" and o["customer\_id"] == telegram\_id), None)  
 if not order:  
 await message.answer("Заказ не найден или не завершен.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
 return  
 await state.update\_data(order\_id=order\_id, executor\_id=order["executor\_id"])  
 await message.answer("Введите рейтинг (от 1 до 5):")  
 await state.set\_state(OrderStates.input\_rating)  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID заказа.")  
 except Exception as e:  
 logger.error(f"Ошибка при выборе заказа для отзыва: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
  
  
@router.message(OrderStates.input\_rating)  
async def process\_review\_rating(message: Message, state: FSMContext):  
 try:  
 rating = int(message.text.strip())  
 if not 1 <= rating <= 5:  
 await message.answer("Рейтинг должен быть от 1 до 5.")  
 return  
 await state.update\_data(rating=rating)  
 await message.answer("Введите комментарий (или /skip для пропуска):")  
 await state.set\_state(OrderStates.input\_comment)  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректное число для рейтинга.")  
 except Exception as e:  
 logger.error(f"Ошибка при вводе рейтинга: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(message.from\_user.id)))  
 await state.clear()  
  
  
@router.message(OrderStates.input\_comment)  
async def process\_review\_comment(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 data = await state.get\_data()  
 comment = message.text.strip() if message.text.strip() != "/skip" else None  
 review\_data = {  
 "order\_id": data["order\_id"],  
 "target\_id": data["executor\_id"],  
 "rating": data["rating"],  
 "comment": comment  
 }  
 try:  
 review = await api\_request("POST", f"{API\_URL}review/", telegram\_id, data=review\_data)  
 await message.answer(  
 f"Отзыв успешно оставлен!\nРейтинг: {review['rating']}\nКомментарий: {review['comment'] or 'Нет'}",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 except Exception as e:  
 logger.error(f"Ошибка при создании отзыва: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()

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

from aiogram import Router, F  
from aiogram.types import Message, InlineKeyboardMarkup, InlineKeyboardButton, CallbackQuery  
from aiogram.fsm.context import FSMContext  
from aiogram.fsm.state import State, StatesGroup  
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\_\_)  
  
# Определяем состояния для принятия/отклонения предложений  
class OfferActionStates(StatesGroup):  
 select\_order = State() # Выбор заказа  
 select\_offer = State() # Выбор действия с предложением  
  
# Точка входа для просмотра предложений  
@router.message(F.text == "Посмотреть предложения")  
async def start\_view\_offers(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 if not roles["is\_customer"]:  
 await message.answer(  
 "Только заказчики могут просматривать предложения.",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 return  
  
 try:  
 # Получаем список заказов пользователя  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 pending\_orders = [o for o in orders if o["status"] == "В\_ожидании"]  
  
 if not pending\_orders:  
 await message.answer(  
 "У вас нет заказов со статусом 'В\_ожидании', для которых можно просмотреть предложения.",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 return  
  
 orders\_list = "\n".join([f"ID: {order['id']} - {order['title']}" for order in pending\_orders])  
 await message.answer(  
 f"Ваши заказы в ожидании:\n{orders\_list}\n\nВведите ID заказа, чтобы увидеть предложения:",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 await state.set\_state(OfferActionStates.select\_order)  
 except Exception as e:  
 logger.error(f"Ошибка при загрузке заказов: {e}")  
 await message.answer(  
 f"Ошибка: {e}",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 await state.clear()  
  
# Обработка выбора заказа  
@router.message(OfferActionStates.select\_order)  
async def process\_order\_selection(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 order\_id = int(message.text.strip())  
 # Проверяем, принадлежит ли заказ пользователю и имеет ли статус "В\_ожидании"  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 selected\_order = next(  
 (order for order in orders if order["id"] == order\_id and order["status"] == "В\_ожидании"),  
 None  
 )  
 if not selected\_order:  
 await message.answer(  
 "Заказ не найден или не находится в статусе 'В\_ожидании'.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 return  
  
 # Получаем предложения по заказу  
 offers = await api\_request("GET", f"{API\_URL}order/{order\_id}/offers", telegram\_id)  
 if not offers:  
 await message.answer(  
 f"По заказу ID {order\_id} нет предложений.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 return  
  
 # Формируем список предложений с inline-кнопками  
 offers\_list = "\n\n".join([  
 f"ID: {offer['id']}\n"  
 f"Исполнитель: {offer['executor\_id']}\n"  
 f"Цена: {offer['price']} тенге\n"  
 f"Время выполнения: {offer['estimated\_time']} часов\n"  
 f"Статус: {offer['status']}"  
 for offer in offers if offer["status"] == "pending" # Показываем только "pending"  
 ])  
 if not offers\_list:  
 await message.answer(  
 f"По заказу ID {order\_id} нет активных предложений.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 return  
  
 # Создаём inline-клавиатуру для каждого предложения  
 inline\_kb = InlineKeyboardMarkup(inline\_keyboard=[])  
 for offer in offers:  
 if offer["status"] == "pending":  
 inline\_kb.inline\_keyboard.append([  
 InlineKeyboardButton(  
 text=f"Принять (ID: {offer['id']})",  
 callback\_data=f"accept\_offer\_{order\_id}\_{offer['id']}"  
 ),  
 InlineKeyboardButton(  
 text=f"Отклонить (ID: {offer['id']})",  
 callback\_data=f"reject\_offer\_{order\_id}\_{offer['id']}"  
 )  
 ])  
  
 await message.answer(  
 f"Предложения по заказу ID {order\_id}:\n{offers\_list}",  
 reply\_markup=inline\_kb  
 )  
 await state.update\_data(order\_id=order\_id)  
 await state.set\_state(OfferActionStates.select\_offer)  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID заказа.")  
 except Exception as e:  
 logger.error(f"Ошибка при выборе заказа: {e}")  
 await message.answer(  
 f"Ошибка: {e}",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
  
# Обработка callback-запросов для принятия/отклонения предложений  
@router.callback\_query(F.data.startswith("accept\_offer\_"))  
async def process\_accept\_offer(callback: CallbackQuery, state: FSMContext):  
 telegram\_id = callback.from\_user.id  
 try:  
 \_, order\_id, offer\_id = callback.data.split("\_")  
 order\_id = int(order\_id)  
 offer\_id = int(offer\_id)  
  
 # Отправляем запрос на принятие предложения  
 updated\_order = await api\_request(  
 "POST",  
 f"{API\_URL}order/{order\_id}/offers/{offer\_id}/accept",  
 telegram\_id  
 )  
 await callback.message.edit\_text(  
 f"Предложение ID {offer\_id} по заказу ID {order\_id} успешно принято!\n"  
 f"Статус заказа: {updated\_order['status']}",  
 reply\_markup=None  
 )  
 await callback.answer("Предложение принято!")  
 await state.clear()  
 except Exception as e:  
 logger.error(f"Ошибка при принятии предложения: {e}")  
 await callback.message.edit\_text(  
 f"Ошибка при принятии предложения: {e}",  
 reply\_markup=None  
 )  
 await callback.answer("Ошибка!")  
 await state.clear()  
  
@router.callback\_query(F.data.startswith("reject\_offer\_"))  
async def process\_reject\_offer(callback: CallbackQuery, state: FSMContext):  
 telegram\_id = callback.from\_user.id  
 try:  
 \_, order\_id, offer\_id = callback.data.split("\_")  
 order\_id = int(order\_id)  
 offer\_id = int(offer\_id)  
  
 # Отправляем запрос на отклонение предложения  
 updated\_offer = await api\_request(  
 "POST",  
 f"{API\_URL}order/{order\_id}/offers/{offer\_id}/reject",  
 telegram\_id  
 )  
 await callback.message.edit\_text(  
 f"Предложение ID {offer\_id} по заказу ID {order\_id} успешно отклонено!\n"  
 f"Статус предложения: {updated\_offer['status']}",  
 reply\_markup=None  
 )  
 await callback.answer("Предложение отклонено!")  
 await state.clear()  
 except Exception as e:  
 logger.error(f"Ошибка при отклонении предложения: {e}")  
 await callback.message.edit\_text(  
 f"Ошибка при отклонении предложения: {e}",  
 reply\_markup=None  
 )  
 await callback.answer("Ошибка!")  
 await state.clear()

## 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.context import FSMContext  
from aiogram.fsm.state import State, StatesGroup  
from app.bot.handlers.common import api\_request, get\_main\_keyboard, get\_user\_roles  
from app.bot.config import API\_URL  
import logging  
  
from app.bot.handlers.customer.main import OrderStates  
  
router = Router()  
logger = logging.getLogger(\_\_name\_\_)  
  
# Определяем состояния для создания заказа  
class CreateOrderStates(StatesGroup):  
 title = State()  
 description = State()  
 desired\_price = State() # Используем desired\_price вместо price  
 due\_date = State()  
 category = State()  
  
async def start\_create\_order(message: Message, state: FSMContext):  
 await message.answer("Введите название заказа:")  
 await state.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:")  
 await state.set\_state(CreateOrderStates.description)  
  
@router.message(CreateOrderStates.description)  
async def process\_description(message: Message, state: FSMContext):  
 await state.update\_data(description=message.text.strip())  
 await message.answer("Введите желаемую цену (в рублях):")  
 await state.set\_state(CreateOrderStates.desired\_price)  
  
@router.message(CreateOrderStates.desired\_price) # Исправлено с price на desired\_price  
async def process\_price(message: Message, state: FSMContext):  
 try:  
 price = float(message.text.strip())  
 await state.update\_data(desired\_price=price)  
 await message.answer("Введите дедлайн (в формате ДД.ММ.ГГГГ):")  
 await state.set\_state(CreateOrderStates.due\_date)  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректное число для цены.")  
  
@router.message(CreateOrderStates.due\_date)  
async def process\_due\_date(message: Message, state: FSMContext):  
 try:  
 due\_date = message.text.strip()  
 # Проверка формата даты (можно усилить валидацию)  
 datetime.strptime(due\_date, "%d.%m.%Y")  
 await state.update\_data(due\_date=due\_date)  
 telegram\_id = message.from\_user.id  
 categories = await api\_request("GET", f"{API\_URL}category/", telegram\_id)  
 if not categories:  
 await message.answer("В системе нет категорий. Обратитесь к администратору.")  
 await state.clear()  
 return  
 categories\_list = "\n".join([f"ID: {cat['id']} - {cat['name']}" for cat in categories])  
 await message.answer(f"Выберите категорию:\n{categories\_list}\n\nВведите ID категории:")  
 await state.set\_state(CreateOrderStates.category)  
 except ValueError:  
 await message.answer("Пожалуйста, введите дату в формате ДД.ММ.ГГГГ.")  
  
@router.message(CreateOrderStates.category)  
async def process\_category(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 category\_id = int(message.text.strip())  
 data = await state.get\_data()  
 order\_data = {  
 "category\_id": category\_id,  
 "title": data["title"],  
 "description": data["description"],  
 "desired\_price": data["desired\_price"],  
 "due\_date": data["due\_date"]  
 }  
 order = await api\_request("POST", f"{API\_URL}order/", telegram\_id, data=order\_data)  
 await message.answer(  
 f"Заказ '{order['title']}' (ID: {order['id']}) успешно создан!\nВыберите действие в меню ниже:",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID категории.")  
 except Exception as e:  
 logger.error(f"Ошибка при создании заказа: {e}")  
 await message.answer(  
 f"Ошибка при создании заказа: {e}\nВыберите действие в меню ниже:",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
  
@router.message(F.text == "Редактировать заказ")  
async def edit\_order\_start(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 if not roles["is\_customer"]:  
 await message.answer("Только заказчики могут редактировать заказы.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 editable\_orders = [o for o in orders if o["status"] == "В\_ожидании"]  
 if not editable\_orders:  
 await message.answer("Нет заказов для редактирования.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 response = "Выберите заказ:\n" + "\n".join([f"ID: {o['id']} - {o['title']}" for o in editable\_orders])  
 await message.answer(response + "\nВведите ID заказа:")  
 await state.set\_state(OrderStates.select\_order)

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

from aiogram import Router, F  
from aiogram.types import Message  
from aiogram.fsm.context import FSMContext  
from aiogram.fsm.state import State, StatesGroup  
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\_\_)  
  
# Определяем состояния для управления отзывами  
class ReviewStates(StatesGroup):  
 select\_action = State() # Выбор действия (создать, просмотреть, редактировать)  
 select\_order\_create = State() # Выбор заказа для создания отзыва  
 rating\_create = State() # Ввод рейтинга для создания отзыва  
 comment\_create = State() # Ввод комментария для создания отзыва  
 select\_review\_view = State() # Выбор отзыва для просмотра  
 select\_review\_edit = State() # Выбор отзыва для редактирования  
 rating\_edit = State() # Ввод нового рейтинга для редактирования  
 comment\_edit = State() # Ввод нового комментария для редактирования  
  
# Главная точка входа для отзывов  
@router.message(F.text == "Оставить отзыв")  
async def review\_menu(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 if not roles["is\_customer"]:  
 await message.answer("Только заказчики могут работать с отзывами.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 await message.answer(  
 "Выберите действие с отзывами:\n1. Оставить новый отзыв\n2. Посмотреть мои отзывы\n3. Редактировать отзыв",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 await state.set\_state(ReviewStates.select\_action)  
  
# Обработка выбора действия  
@router.message(ReviewStates.select\_action)  
async def process\_review\_action(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 action = message.text.strip().lower()  
 try:  
 if "оставить" in action or "1" in action:  
 await start\_create\_review(message, state)  
 elif "посмотреть" in action or "2" in action:  
 await view\_reviews(message, state)  
 elif "редактировать" in action or "3" in action:  
 await start\_edit\_review(message, state)  
 else:  
 await message.answer("Пожалуйста, выберите действие: 1, 2 или 3.")  
 except Exception as e:  
 logger.error(f"Ошибка при выборе действия с отзывами: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
  
# Создание нового отзыва  
async def start\_create\_review(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 completed\_orders = [o for o in orders if o["status"] == "Выполнен" and "review" not in o]  
 if not completed\_orders:  
 await message.answer(  
 "У вас нет завершённых заказов без отзывов.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 return  
 orders\_list = "\n".join([f"ID: {o['id']} - {o['title']}" for o in completed\_orders])  
 await message.answer(  
 f"Выберите заказ для отзыва:\n{orders\_list}\n\nВведите ID заказа:",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.set\_state(ReviewStates.select\_order\_create)  
 except Exception as e:  
 logger.error(f"Ошибка при загрузке завершённых заказов: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
  
@router.message(ReviewStates.select\_order\_create)  
async def process\_order\_create(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 order\_id = int(message.text.strip())  
 order = await api\_request("GET", f"{API\_URL}order/{order\_id}", telegram\_id)  
 if order["status"] != "Выполнен" or order["customer\_id"] != telegram\_id:  
 await message.answer(  
 "Этот заказ нельзя оценить.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 return  
 if "review" in order and order["review"]:  
 await message.answer(  
 "На этот заказ уже оставлен отзыв.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 return  
 await state.update\_data(order\_id=order\_id, target\_id=order["executor\_id"])  
 await message.answer("Введите рейтинг (от 1 до 5):")  
 await state.set\_state(ReviewStates.rating\_create)  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID заказа.")  
 except Exception as e:  
 logger.error(f"Ошибка при выборе заказа для отзыва: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
  
@router.message(ReviewStates.rating\_create)  
async def process\_rating\_create(message: Message, state: FSMContext):  
 try:  
 rating = int(message.text.strip())  
 if not 1 <= rating <= 5:  
 await message.answer("Пожалуйста, введите рейтинг от 1 до 5.")  
 return  
 await state.update\_data(rating=rating)  
 await message.answer("Введите комментарий (или отправьте пустое сообщение для пропуска):")  
 await state.set\_state(ReviewStates.comment\_create)  
 except ValueError:  
 await message.answer("Пожалуйста, введите число от 1 до 5.")  
 except Exception as e:  
 logger.error(f"Ошибка при вводе рейтинга: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(message.from\_user.id)))  
 await state.clear()  
  
@router.message(ReviewStates.comment\_create)  
async def process\_comment\_create(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 comment = message.text.strip() if message.text.strip() else None  
 data = await state.get\_data()  
 review\_data = {  
 "order\_id": data["order\_id"],  
 "target\_id": data["target\_id"],  
 "rating": data["rating"],  
 "comment": comment  
 }  
 review = await api\_request("POST", f"{API\_URL}review/", telegram\_id, data=review\_data)  
 await message.answer(  
 f"Отзыв для заказа ID {review['order\_id']} успешно оставлен!\n"  
 f"Рейтинг: {review['rating']}/5\n"  
 f"Комментарий: {review['comment'] or 'Без комментария'}",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 except Exception as e:  
 logger.error(f"Ошибка при создании отзыва: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
  
# Просмотр отзывов  
async def view\_reviews(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 reviews = await api\_request("GET", f"{API\_URL}review/", telegram\_id)  
 if not reviews:  
 await message.answer(  
 "У вас нет оставленных отзывов.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 return  
 reviews\_list = "\n\n".join([  
 f"ID: {r['id']} - Заказ ID: {r['order\_id']}\n"  
 f"Рейтинг: {r['rating']}/5\n"  
 f"Комментарий: {r['comment'] or 'Без комментария'}"  
 for r in reviews  
 ])  
 await message.answer(f"Ваши отзывы:\n{reviews\_list}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
 except Exception as e:  
 logger.error(f"Ошибка при загрузке отзывов: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
  
# Редактирование отзыва  
async def start\_edit\_review(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 reviews = await api\_request("GET", f"{API\_URL}review/", telegram\_id)  
 if not reviews:  
 await message.answer(  
 "У вас нет отзывов для редактирования.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 return  
 reviews\_list = "\n".join([f"ID: {r['id']} - Заказ ID: {r['order\_id']}" for r in reviews])  
 await message.answer(  
 f"Выберите отзыв для редактирования:\n{reviews\_list}\n\nВведите ID отзыва:",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.set\_state(ReviewStates.select\_review\_edit)  
 except Exception as e:  
 logger.error(f"Ошибка при загрузке отзывов для редактирования: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
  
@router.message(ReviewStates.select\_review\_edit)  
async def process\_review\_edit(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 review\_id = int(message.text.strip())  
 review = await api\_request("GET", f"{API\_URL}review/{review\_id}", telegram\_id)  
 if review["author\_id"] != telegram\_id:  
 await message.answer(  
 "Вы не можете редактировать этот отзыв.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 return  
 await state.update\_data(review\_id=review\_id)  
 await message.answer(  
 f"Текущий рейтинг: {review['rating']}/5\nВведите новый рейтинг (от 1 до 5):"  
 )  
 await state.set\_state(ReviewStates.rating\_edit)  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID отзыва.")  
 except Exception as e:  
 logger.error(f"Ошибка при выборе отзыва для редактирования: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
  
@router.message(ReviewStates.rating\_edit)  
async def process\_rating\_edit(message: Message, state: FSMContext):  
 try:  
 rating = int(message.text.strip())  
 if not 1 <= rating <= 5:  
 await message.answer("Пожалуйста, введите рейтинг от 1 до 5.")  
 return  
 await state.update\_data(rating=rating)  
 await message.answer("Введите новый комментарий (или отправьте пустое сообщение для пропуска):")  
 await state.set\_state(ReviewStates.comment\_edit)  
 except ValueError:  
 await message.answer("Пожалуйста, введите число от 1 до 5.")  
 except Exception as e:  
 logger.error(f"Ошибка при редактировании рейтинга: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(message.from\_user.id)))  
 await state.clear()  
  
@router.message(ReviewStates.comment\_edit)  
async def process\_comment\_edit(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 comment = message.text.strip() if message.text.strip() else None  
 data = await state.get\_data()  
 review\_data = {  
 "rating": data["rating"],  
 "comment": comment  
 }  
 review = await api\_request("PATCH", f"{API\_URL}review/{data['review\_id']}", telegram\_id, data=review\_data)  
 await message.answer(  
 f"Отзыв ID {review['id']} успешно обновлён!\n"  
 f"Рейтинг: {review['rating']}/5\n"  
 f"Комментарий: {review['comment'] or 'Без комментария'}",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 except Exception as e:  
 logger.error(f"Ошибка при обновлении отзыва: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()

## 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  
from .reviews import router as customer\_reviews\_router # Добавляем новый роутер  
  
customer\_routers = [  
 customer\_main\_router,  
 customer\_orders\_router,  
 customer\_offers\_router,  
 customer\_reviews\_router # Добавляем в список  
]

## Файл: C:/Users/User/PycharmProjects/telegram\_service/app\bot\handlers\executor\main.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  
  
from app.bot.handlers.executor.offers import start\_create\_offer  
  
router = Router()  
logger = logging.getLogger(\_\_name\_\_)  
  
class CompleteOrderStates(StatesGroup):  
 select\_order = State()  
  
@router.message(F.text == "Создать предложение")  
async def create\_offer\_entrypoint(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 if not roles.get("is\_executor"):  
 await message.answer(  
 "Эта функция доступна только для исполнителей.",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 return  
 try:  
 await start\_create\_offer(message, state)  
 except Exception as e:  
 logger.error(f"Ошибка при создании предложения: {e}")  
 await message.answer(  
 f"Не удалось начать создание предложения: {e}",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
  
@router.message(F.text == "Список заказов")  
async def list\_orders(message: Message):  
 telegram\_id = message.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 try:  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 if not orders:  
 await message.answer("У вас нет заказов.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 response = "Ваши заказы:\n\n"  
 for order in orders:  
 status = {"В\_ожидании": "Ожидает", "В\_прогрессе": "В процессе", "Выполнен": "Завершён", "Отменен": "Отменён"}.get(order["status"], order["status"])  
 response += f"ID: {order['id']} - {order['title']} ({status})\n"  
 await message.answer(response, reply\_markup=get\_main\_keyboard(roles))  
 except Exception as e:  
 logger.error(f"Ошибка загрузки заказов: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(roles))  
  
@router.message(F.text == "Завершить заказ")  
async def complete\_order\_start(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 if not roles["is\_executor"]:  
 await message.answer("Только исполнители могут завершать заказы.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 try:  
 orders = await api\_request("GET", f"{API\_URL}order/", telegram\_id)  
 active\_orders = [o for o in orders if o["status"] == "В\_прогрессе" and o["executor\_id"] == telegram\_id]  
 if not active\_orders:  
 await message.answer("У вас нет активных заказов для завершения.", reply\_markup=get\_main\_keyboard(roles))  
 return  
 response = "Ваши активные заказы:\n\n"  
 for order in active\_orders:  
 response += f"ID: {order['id']} - {order['title']}\n"  
 await message.answer(response + "\nВведите ID заказа для завершения:")  
 await state.set\_state(CompleteOrderStates.select\_order)  
 except Exception as e:  
 logger.error(f"Ошибка загрузки заказов: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(roles))  
  
@router.message(CompleteOrderStates.select\_order)  
async def complete\_order\_process(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 order\_id = int(message.text.strip())  
 order\_data = {"status": "Выполнен"}  
 await api\_request("PATCH", f"{API\_URL}order/{order\_id}", telegram\_id, data=order\_data)  
 await message.answer(f"Заказ ID {order\_id} завершён!", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()  
 except ValueError:  
 await message.answer("Ошибка: Введите корректный ID заказа")  
 except Exception as e:  
 logger.error(f"Ошибка завершения заказа: {e}")  
 await message.answer(f"Ошибка: {e}", reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id)))  
 await state.clear()

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

from aiogram import Router, F  
from aiogram.types import Message, InlineKeyboardMarkup, InlineKeyboardButton, CallbackQuery  
from aiogram.fsm.context import FSMContext  
from aiogram.fsm.state import State, StatesGroup  
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\_\_)  
  
# Определяем состояния для создания, редактирования и удаления предложений  
class OfferStates(StatesGroup):  
 select\_order = State() # Выбор заказа для создания предложения  
 price = State() # Ввод цены для создания  
 estimated\_time = State() # Ввод времени выполнения для создания  
 select\_offer\_edit = State() # Выбор предложения для редактирования  
 price\_edit = State() # Ввод новой цены для редактирования  
 estimated\_time\_edit = State() # Ввод нового времени для редактирования  
 select\_offer\_delete = State() # Выбор предложения для удаления  
  
# Главная точка входа для создания предложения  
@router.message(F.text == "Создать предложение")  
async def start\_create\_offer(message: Message, state: FSMContext):  
 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  
  
 try:  
 user\_profile = await api\_request("GET", f"{API\_URL}user/me", telegram\_id)  
 executor\_categories = set(user\_profile.get("category\_ids", []))  
 executor\_city = user\_profile.get("city\_id")  
  
 if not executor\_categories or not executor\_city:  
 await message.answer(  
 "Пожалуйста, обновите профиль, указав категории и город, чтобы видеть доступные заказы.",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 return  
  
 available\_orders = await api\_request("GET", f"{API\_URL}order/available", telegram\_id)  
 filtered\_orders = [  
 order for order in available\_orders  
 if order["category\_id"] in executor\_categories and order["city\_id"] == executor\_city  
 ]  
  
 if not filtered\_orders:  
 await message.answer(  
 "Нет доступных заказов в вашем городе и категориях.",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 return  
  
 orders\_list = "\n\n".join([  
 f"ID: {order['id']}\n"  
 f"Название: {order['title']}\n"  
 f"Описание: {order['description'] or 'Нет описания'}\n"  
 f"Цена: {order['desired\_price']} тенге\n"  
 f"Дедлайн: {order['due\_date']}"  
 for order in filtered\_orders  
 ])  
 await message.answer(  
 f"Доступные заказы:\n{orders\_list}\n\nВведите ID заказа, на который хотите подать предложение:",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 await state.set\_state(OfferStates.select\_order)  
 except Exception as e:  
 logger.error(f"Ошибка при загрузке доступных заказов: {e}")  
 await message.answer(  
 f"Ошибка: {e}",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 await state.clear()  
  
# Обработка выбора заказа для создания предложения  
@router.message(OfferStates.select\_order)  
async def process\_order\_selection(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 order\_id = int(message.text.strip())  
 available\_orders = await api\_request("GET", f"{API\_URL}order/available", telegram\_id)  
 selected\_order = next((order for order in available\_orders if order["id"] == order\_id), None)  
 if not selected\_order:  
 await message.answer(  
 "Заказ не найден или недоступен.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 return  
  
 await state.update\_data(order\_id=order\_id)  
 await message.answer("Введите вашу цену за выполнение заказа (в тенге):")  
 await state.set\_state(OfferStates.price)  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID заказа.")  
 except Exception as e:  
 logger.error(f"Ошибка при выборе заказа: {e}")  
 await message.answer(  
 f"Ошибка: {e}",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
  
# Обработка ввода цены для создания  
@router.message(OfferStates.price)  
async def process\_price(message: Message, state: FSMContext):  
 try:  
 price = float(message.text.strip())  
 if price <= 0:  
 await message.answer("Цена должна быть больше 0.")  
 return  
 await state.update\_data(price=price)  
 await message.answer("Введите предполагаемое время выполнения (в часах):")  
 await state.set\_state(OfferStates.estimated\_time)  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректное число для цены.")  
 except Exception as e:  
 logger.error(f"Ошибка при вводе цены: {e}")  
 await message.answer(  
 f"Ошибка: {e}",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(message.from\_user.id))  
 )  
 await state.clear()  
  
# Обработка ввода времени и отправка предложения  
@router.message(OfferStates.estimated\_time)  
async def process\_estimated\_time(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 estimated\_time = int(message.text.strip())  
 if estimated\_time <= 0:  
 await message.answer("Время выполнения должно быть больше 0.")  
 return  
 data = await state.get\_data()  
 offer\_data = {  
 "order\_id": data["order\_id"],  
 "price": data["price"],  
 "estimated\_time": estimated\_time  
 }  
 offer = await api\_request("POST", f"{API\_URL}offer/", telegram\_id, data=offer\_data)  
 await message.answer(  
 f"Предложение для заказа ID {offer['order\_id']} успешно создано!\n"  
 f"Цена: {offer['price']} тенге\n"  
 f"Время выполнения: {offer['estimated\_time']} часов",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректное число для времени выполнения.")  
 except Exception as e:  
 logger.error(f"Ошибка при создании предложения: {e}")  
 await message.answer(  
 f"Ошибка: {e}",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
  
# Команда "Мои предложения" для просмотра списка предложений  
@router.message(F.text == "Мои предложения")  
async def list\_offers(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 roles = await get\_user\_roles(telegram\_id)  
 if not roles["is\_executor"]:  
 await message.answer(  
 "Только исполнители могут просматривать свои предложения.",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 return  
  
 try:  
 offers = await api\_request("GET", f"{API\_URL}offer/", telegram\_id)  
 if not offers:  
 await message.answer(  
 "У вас нет предложений.",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 return  
  
 offers\_list = "\n\n".join([  
 f"ID: {offer['id']}\n"  
 f"Заказ ID: {offer['order\_id']}\n"  
 f"Цена: {offer['price']} тенге\n"  
 f"Время выполнения: {offer['estimated\_time']} часов\n"  
 f"Статус: {offer['status']}"  
 for offer in offers  
 ])  
 inline\_kb = InlineKeyboardMarkup(inline\_keyboard=[  
 [InlineKeyboardButton(text="Редактировать", callback\_data="edit\_offer"),  
 InlineKeyboardButton(text="Удалить", callback\_data="delete\_offer")]  
 ])  
 await message.answer(  
 f"Ваши предложения:\n{offers\_list}",  
 reply\_markup=inline\_kb  
 )  
 except Exception as e:  
 logger.error(f"Ошибка при загрузке предложений: {e}")  
 await message.answer(  
 f"Ошибка: {e}",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
  
# Обработка редактирования предложений  
@router.callback\_query(F.data == "edit\_offer")  
async def start\_edit\_offer(callback: CallbackQuery, state: FSMContext):  
 telegram\_id = callback.from\_user.id  
 try:  
 offers = await api\_request("GET", f"{API\_URL}offer/", telegram\_id)  
 editable\_offers = [o for o in offers if o["status"] == "pending"] # Только "pending" можно редактировать  
 if not editable\_offers:  
 await callback.message.edit\_text(  
 "У вас нет предложений, доступных для редактирования.",  
 reply\_markup=None  
 )  
 await callback.answer()  
 return  
  
 offers\_list = "\n".join([  
 f"ID: {offer['id']} - Заказ ID: {offer['order\_id']}"  
 for offer in editable\_offers  
 ])  
 await callback.message.edit\_text(  
 f"Выберите предложение для редактирования:\n{offers\_list}\n\nВведите ID предложения:",  
 reply\_markup=None  
 )  
 await state.set\_state(OfferStates.select\_offer\_edit)  
 await callback.answer()  
 except Exception as e:  
 logger.error(f"Ошибка при загрузке предложений для редактирования: {e}")  
 await callback.message.edit\_text(  
 f"Ошибка: {e}",  
 reply\_markup=None  
 )  
 await state.clear()  
 await callback.answer()  
  
@router.message(OfferStates.select\_offer\_edit)  
async def process\_offer\_edit\_selection(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 offer\_id = int(message.text.strip())  
 offer = await api\_request("GET", f"{API\_URL}offer/{offer\_id}", telegram\_id)  
 if offer["status"] != "pending":  
 await message.answer(  
 "Это предложение нельзя редактировать, так как оно уже принято или отклонено.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 return  
  
 await state.update\_data(offer\_id=offer\_id)  
 await message.answer(  
 f"Текущая цена: {offer['price']} тенге\nВведите новую цену (в тенге):"  
 )  
 await state.set\_state(OfferStates.price\_edit)  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID предложения.")  
 except Exception as e:  
 logger.error(f"Ошибка при выборе предложения для редактирования: {e}")  
 await message.answer(  
 f"Ошибка: {e}",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
  
@router.message(OfferStates.price\_edit)  
async def process\_price\_edit(message: Message, state: FSMContext):  
 try:  
 price = float(message.text.strip())  
 if price <= 0:  
 await message.answer("Цена должна быть больше 0.")  
 return  
 await state.update\_data(price=price)  
 await message.answer("Введите новое предполагаемое время выполнения (в часах):")  
 await state.set\_state(OfferStates.estimated\_time\_edit)  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректное число для цены.")  
 except Exception as e:  
 logger.error(f"Ошибка при редактировании цены: {e}")  
 await message.answer(  
 f"Ошибка: {e}",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(message.from\_user.id))  
 )  
 await state.clear()  
  
@router.message(OfferStates.estimated\_time\_edit)  
async def process\_estimated\_time\_edit(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 estimated\_time = int(message.text.strip())  
 if estimated\_time <= 0:  
 await message.answer("Время выполнения должно быть больше 0.")  
 return  
 data = await state.get\_data()  
 offer\_data = {  
 "price": data["price"],  
 "estimated\_time": estimated\_time  
 }  
 updated\_offer = await api\_request(  
 "PATCH",  
 f"{API\_URL}offer/{data['offer\_id']}",  
 telegram\_id,  
 data=offer\_data  
 )  
 await message.answer(  
 f"Предложение ID {updated\_offer['id']} успешно обновлено!\n"  
 f"Цена: {updated\_offer['price']} тенге\n"  
 f"Время выполнения: {updated\_offer['estimated\_time']} часов",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректное число для времени выполнения.")  
 except Exception as e:  
 logger.error(f"Ошибка при обновлении предложения: {e}")  
 await message.answer(  
 f"Ошибка: {e}",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
  
# Обработка удаления предложений  
@router.callback\_query(F.data == "delete\_offer")  
async def start\_delete\_offer(callback: CallbackQuery, state: FSMContext):  
 telegram\_id = callback.from\_user.id  
 try:  
 offers = await api\_request("GET", f"{API\_URL}offer/", telegram\_id)  
 deletable\_offers = [o for o in offers if o["status"] == "pending"] # Только "pending" можно удалять  
 if not deletable\_offers:  
 await callback.message.edit\_text(  
 "У вас нет предложений, доступных для удаления.",  
 reply\_markup=None  
 )  
 await callback.answer()  
 return  
  
 offers\_list = "\n".join([  
 f"ID: {offer['id']} - Заказ ID: {offer['order\_id']}"  
 for offer in deletable\_offers  
 ])  
 inline\_kb = InlineKeyboardMarkup(inline\_keyboard=[  
 [InlineKeyboardButton(text="Отмена", callback\_data="cancel\_delete")]  
 ])  
 await callback.message.edit\_text(  
 f"Выберите предложение для удаления:\n{offers\_list}\n\nВведите ID предложения:",  
 reply\_markup=inline\_kb  
 )  
 await state.set\_state(OfferStates.select\_offer\_delete)  
 await callback.answer()  
 except Exception as e:  
 logger.error(f"Ошибка при загрузке предложений для удаления: {e}")  
 await callback.message.edit\_text(  
 f"Ошибка: {e}",  
 reply\_markup=None  
 )  
 await state.clear()  
 await callback.answer()  
  
@router.message(OfferStates.select\_offer\_delete)  
async def process\_offer\_delete\_selection(message: Message, state: FSMContext):  
 telegram\_id = message.from\_user.id  
 try:  
 offer\_id = int(message.text.strip())  
 offer = await api\_request("GET", f"{API\_URL}offer/{offer\_id}", telegram\_id)  
 if offer["status"] != "pending":  
 await message.answer(  
 "Это предложение нельзя удалить, так как оно уже принято или отклонено.",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 return  
  
 await api\_request("DELETE", f"{API\_URL}offer/{offer\_id}", telegram\_id)  
 await message.answer(  
 f"Предложение ID {offer\_id} успешно удалено!",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
 except ValueError:  
 await message.answer("Пожалуйста, введите корректный ID предложения.")  
 except Exception as e:  
 logger.error(f"Ошибка при удалении предложения: {e}")  
 await message.answer(  
 f"Ошибка: {e}",  
 reply\_markup=get\_main\_keyboard(await get\_user\_roles(telegram\_id))  
 )  
 await state.clear()  
  
@router.callback\_query(F.data == "cancel\_delete")  
async def cancel\_delete\_offer(callback: CallbackQuery, state: FSMContext):  
 await callback.message.edit\_text(  
 "Удаление предложения отменено.",  
 reply\_markup=None  
 )  
 await state.clear()  
 await callback.answer()  
  
# Просмотр списка доступных заказов (без изменений)  
@router.message(F.text == "Список доступных заказов")  
async def list\_available\_orders(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 просматривать доступные заказы.",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 return  
  
 try:  
 user\_profile = await api\_request("GET", f"{API\_URL}user/me", telegram\_id)  
 executor\_categories = set(user\_profile.get("category\_ids", []))  
 executor\_city = user\_profile.get("city\_id")  
  
 if not executor\_categories or not executor\_city:  
 await message.answer(  
 "Пожалуйста, обновите профиль, указав категории и город.",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 return  
  
 available\_orders = await api\_request("GET", f"{API\_URL}order/available", telegram\_id)  
 filtered\_orders = [  
 order for order in available\_orders  
 if order["category\_id"] in executor\_categories and order["city\_id"] == executor\_city  
 ]  
  
 if not filtered\_orders:  
 await message.answer(  
 "Нет доступных заказов в вашем городе и категориях.",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 return  
  
 orders\_list = "\n\n".join([  
 f"ID: {order['id']}\n"  
 f"Название: {order['title']}\n"  
 f"Описание: {order['description'] or 'Нет описания'}\n"  
 f"Цена: {order['desired\_price']} тенге\n"  
 f"Дедлайн: {order['due\_date']}"  
 for order in filtered\_orders  
 ])  
 await message.answer(  
 f"Доступные заказы:\n{orders\_list}",  
 reply\_markup=get\_main\_keyboard(roles)  
 )  
 except Exception as e:  
 logger.error(f"Ошибка при загрузке списка доступных заказов: {e}")  
 await message.answer(  
 f"Ошибка: {e}",  
 reply\_markup=get\_main\_keyboard(roles)  
 )

## 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"  
 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)  
 executor\_id: Mapped[int | None] = mapped\_column(ForeignKey("users.id", ondelete="SET NULL"), nullable=True)  
 category\_id: Mapped[int] = mapped\_column(ForeignKey("categories.id", ondelete="CASCADE"), nullable=False)  
 title: Mapped[str] = mapped\_column(nullable=False)  
 description: Mapped[str | None] = mapped\_column(nullable=True)  
 desired\_price: Mapped[Decimal] = mapped\_column(Numeric(10, 2), nullable=False)  
 due\_date: Mapped[datetime] = mapped\_column(nullable=False)  
 created\_at: Mapped[datetime] = mapped\_column(default=datetime.utcnow, nullable=False)  
 status: Mapped[OrderStatus] = mapped\_column(Enum(OrderStatus), default=OrderStatus.PENDING, nullable=False)  
  
 customer: Mapped["User"] = relationship("User", foreign\_keys="Order.customer\_id", back\_populates="orders\_created")  
 executor: Mapped["User"] = relationship("User", foreign\_keys="Order.executor\_id", back\_populates="orders\_executed")  
 category: Mapped["Category"] = relationship("Category")  
 offers: Mapped[list["Offer"]] = relationship("Offer", back\_populates="order")  
 review: Mapped["Review"] = relationship("Review", back\_populates="order", uselist=False)

## 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  
  
# 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  
 category\_ids: Optional[List[int]] = None # Добавляем поле для категорий  
  
 @classmethod  
 def from\_orm(cls, obj):  
 data = super().from\_orm(obj).\_\_dict\_\_  
 data["category\_ids"] = [cat.id for cat in obj.categories] if obj.categories else []  
 return cls(\*\*data)  
  
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()  
  
  
# app\core\services\user.py  
def update\_user\_by\_id(session: Session, data: UserUpdate, id: int) -> User:  
 from app.api.depends.user import ADMIN\_TELEGRAM\_ID  
 import logging  
 logger = logging.getLogger(\_\_name\_\_)  
  
 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)  
 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  
 logger.info(f"Обновлены категории для пользователя {id}: {[cat.id for cat in categories]}")  
 del update\_data["category\_ids"]  
 for key, value in update\_data.items():  
 setattr(user, key, value)  
 if user.telegram\_id == ADMIN\_TELEGRAM\_ID:  
 user.is\_admin = True  
 try:  
 session.commit()  
 session.refresh(user)  
 logger.info(f"Пользователь {id} после обновления: category\_ids={[cat.id for cat in user.categories]}")  
 except SQLAlchemyError as e:  
 session.rollback()  
 logger.error(f"Ошибка SQL при обновлении пользователя {id}: {e}")  
 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}")