Отчет по домашней работе по диспиплине “Парадигмы и конструкции языков программирования”

app/db.py

```python
import os
import uuid
import logging


from tortoise.contrib.fastapi import register_tortoise
from aerich import Command
from fastapi import FastAPI


from app import settings
from app.utils.redis import r, ping_redis_connection


# logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


def get_tortoise_config() -> dict:
    app_list = ["app.models", "aerich.models"]
    config = {
        "connections": settings.DB_CONNECTIONS,
        "apps": {
            "models": {
                "models": app_list,
```

```
                "default_connection": "default"
            }
        }
    }
    return config


TORTOISE_ORM = get_tortoise_config()


def register_db(app: FastAPI, db_url: str = None) -> None:
    db_url = db_url or settings.DB_URL
    app_list = ["app.models", "aerich.models"]
    register_tortoise(
        app,
        db_url=db_url,
        modules={"models": app_list},
        generate_schemas=False,
        add_exception_handlers=True
    )


async def upgrade_db(app: FastAPI, db_url: str = None):
    command = Command(tortoise_config=TORTOISE_ORM, app="models",
location="./migrations")
    print(TORTOISE_ORM)
    if not os.path.exists("./migrations/models"):
        await command.init_db(safe=True)
```

```
    await command.init()
    await command.migrate(str(uuid.uuid4()))
    await command.upgrade(run_in_transaction=True)



async def init(app: FastAPI):
    # await upgrade_db(app)
    # register_db(app)
    # logger.debug("Connected to db")
    await ping_redis_connection(r)
    logger.debug("Connected to redis")
```

app/models.py

```
from typing import Optional
from datetime import date



from tortoise import fields
from tortoise.models import Model
from tortoise.exceptions import DoesNotExist
from pydantic import UUID4


from app.schemas import UserCreate
from app.utils import password
```

```python
class User(Model):
    uuid = fields.UUIDField(unique=True, pk=True)
    username = fields.CharField(max_length=64, null=True)
    email = fields.CharField(max_length=64, unique=True)
    password_hash = fields.CharField(max_length=255, unique=True)
    registration_date = fields.DateField(auto_now_add=True)
    is_admin = fields.BooleanField(default=False)
    is_confirmed = fields.BooleanField(default=False)


    @classmethod
    async def get_by_email(cls, email: str) -> Optional["User"]:
        try:
            query = cls.get_or_none(email=email)
            user = await query
            return user
        except DoesNotExist:
            return None

    @classmethod
    async def get_by_uuid(cls, uuid: UUID4) -> Optional["User"]:
        try:
            query = cls.get_or_none(uuid=uuid)
            user = await query
            return user
```

```
        except DoesNotExist:
            return None


    @classmethod
    async def create(cls, user: UserCreate) -> "User":
        user_dict = user.model_dump(exclude=["password"])
        password_hash = password.get_password_hash(password=user.password)
        model = cls(**user_dict, password_hash=password_hash,
registration_date=date.today())
        await model.save()
        return model


    async def to_dict(self):
        d = {}
        for field in self._meta.db_fields:
            d[field] = getattr(self, field)
        for field in self._meta.backward_fk_fields:
            d[field] = await getattr(self, field).all().values()
        return d


    class Meta:
        table = "users"
```

app/routes.py

```python
from typing import Optional

from uuid import uuid4

import asyncio

import uuid



from fastapi import APIRouter, WebSocket, WebSocketDisconnect, Request

from fastapi.responses import JSONResponse, HTMLResponse

from fastapi.exceptions import WebSocketException, HTTPException

from pydantic import UUID4



from app.utils.clients import rooms, clients

from app.utils.actions import Actions



router = APIRouter()



@router.post("/create_room/{room_id}")

async def create_room(room_id: str, user_id: str, is_group: bool = False):

    room = rooms.create_room(room_id=room_id, owner_id=user_id,
is_group=is_group)

    return JSONResponse({"room_id": str(room.id), "is_group":
room.is_group})
```

```python
@router.post("/join_room")

async def join_room(room_id: str, client_id: str, client_name:
Optional[str] = None):



    client = clients.get_client(client_id=client_id)

    if client is None:

        client = clients.create_client(client_id=client_id,
name=client_name)

    room = client.get_room()



    if room is not None:

        await room.remove_client(client_id=client.id)



    room = rooms.get_room(room_id=room_id)

    if room is None:

        return HTTPException(

            status_code=404,

            detail="The room with this id does not exist"

        )



    room.add_client(client=client)

    client.set_room(room=room)

    return JSONResponse({"room_id": str(room.id), "client_id":
str(client.id), "is_group": room.is_group})
```

```python
@router.websocket("/lobby")
async def websocket_lobby_endpoint(websocket: WebSocket):
    await websocket.accept()

    try:
        while True:
            await websocket.send_json({"type": Actions.SHARE_ROOMS.value, "rooms": rooms.get_rooms_info()})
            await asyncio.sleep(5)
    except WebSocketDisconnect:
        return



# TODO make messages receiving
@router.websocket("/ws/{client_id}")
async def websocket_room_endpoint(websocket: WebSocket, client_id: str):
    await websocket.accept()

    client = clients.get_client(client_id=client_id)
    if client is None:
        raise WebSocketException(
            code=1007,
            reason="The client with this uuid does not exist"
        )
    room = client.get_room()
```

```
    if room is None:
        raise WebSocketException(
            code=1007,
            reason="The room with this uuid does not exist"
        )

    client.open(websocket=websocket)


    try:
        while True:
            try:
                data = await websocket.receive()
                if data is not None:
                    await client.handle_message(message=data, room=room)
            except RuntimeError:
                break
    except WebSocketDisconnect:
        await client.handle_disconnect(room=room)
        await clients.delete_client(client_id=client_id)
```

app/schemas.py

```
import uuid
from typing import List, Optional
```

```python
from datetime import date


from pydantic import BaseModel, UUID4, field_validator, EmailStr



class BaseProperties(BaseModel):
    @field_validator("uuid", pre=True, always=True, check_fields=False)
    def default_hashed_id(cls, v):
        return v or uuid.uuid4()



class BaseUser(BaseProperties):
    uuid: Optional[UUID4] = None
    username: Optional[str] = None
    email: Optional[EmailStr] = None
    registration_date: Optional[date] = None
    is_admin: Optional[bool] = None



class UserCreate(BaseProperties):
    username: str
    email: EmailStr
    password: str
```

```python
class UserOut(BaseUser):
    uuid: UUID4
    username: str
    email: EmailStr
    registration_date: date
    is_admin: bool
```

app/settings.py

```python
import random
import string
import os
from dotenv import load_dotenv
load_dotenv()


API_HOST = os.getenv("API_HOST")
API_PORT = os.getenv("API_PORT")


REDIS_HOST = os.getenv("REDIS_HOST")
REDIS_PORT = os.getenv("REDIS_PORT")


DB_USER = os.getenv("DB_USER")
DB_PASSWORD = os.getenv("DB_PASSWORD")
DB_HOST = os.getenv("DB_HOST")
DB_PORT = os.getenv("DB_PORT")
DB_NAME = os.getenv("DB_NAME")
```

```
DB_URL =
f"postgres://{DB_USER}:{DB_PASSWORD}@{DB_HOST}:{DB_PORT}/{DB_NAME}"



DB_CONNECTIONS = {

        "default": DB_URL,

    }



SECRET_KEY = os.getenv("SECRET_KEY",
default="".join([random.choice(string.ascii_letters) for _ in range(32)]))

CLIENT_ID = os.getenv("CLIENT_ID",
default="".join([random.choice(string.ascii_letters) for _ in range(32)]))



LOGIN_URL = f"http://auth:8083/login/access-token"



CORS_ORIGINS = ["*"]

CORS_ALLOW_CREDENTIALS = True

CORS_ALLOW_METHODS = ["*"]

CORS_ALLOW_HEADERS = ["*"]
```

app/utils/actions.py

```
from enum import Enum
```

```
class Actions(Enum):
    JOIN = "join"
    LEAVE = "leave"
    YOUR_NAME = "your-name"
    SHARE_ROOMS = "share-rooms"
    ADD_PEER = "add-peer"
    REMOVE_PEER = "remove-peer"
    RELAY_SDP = "relay-sdp"
    RELAY_ICE = "relay-ice"
    ICE_CANDIDATE = "ice-candidate"
    SESSION_DESCRIPTION = "session-description"
    DISCONNECT = "websocket.disconnect"
    NEW_MSG = "new_msg"
    CHAT_HISTORY = "chat_history"
    STARTED_SCREEN_SHARING = "started-screen-sharing"
    STOPPED_SCREEN_SHARING = "stopped-screen-sharing"
    END_CALL = "end-call"
    OWNER = "owner"
    ENABLE_CAMERA = "enable-camera"
    DISABLE_CAMERA = "disable-camera"
    ENABLE_MICROPHONE = "enable-microphone"
    DISABLE_MICROPHONE = "disable-microphone"
    ROTATE = "rotate"
    PING = "ping"
```

```
    PONG = "pong"
    SERVER_OFFER = "server-offer"
    SERVER_ANSWER = "server-answer"
    SERVER_ICE = "server-ice"



class DeviceTypes(Enum):
    DESKTOP = "desktop"
    MOBILE = "mobile"
```

app/utils/chat.py

```
import json
from datetime import datetime
from random import randint
from typing import List, Optional
from operator import is_not
from functools import partial
from enum import Enum
from uuid import uuid4, UUID
from copy import deepcopy


from pydantic import UUID4, BaseModel


from app.utils.redis import r
```

```
from app import settings




class UserMessage(BaseModel):

    client_id: str

    username: str

    message: str

    timestamp: datetime = datetime.now()




class Message:

    def __init__(self, client_id: str, username: str, message: str,
timestamp: datetime = datetime.now()) -> None:

        self.client_id = client_id

        self.username = username

        self.message = message

        self.timestamp = timestamp



    def __str__(self) -> str:

        return
f"{self.timestamp}::{self.client_id}::{self.username}::{self.message}"




class Chat:

    def __init__(self, session_id: UUID4) -> None:

        self.session_id = session_id
```

```
    async def send_message(self, message: UserMessage):
        message_data = message.__str__()
        async with r.pipeline(transaction=True) as pipe:
            (await (pipe.rpush(f"message:{self.session_id}",
message_data).execute()))



    async def get_all_messages(self):
        async with r.pipeline(transaction=True) as pipe:
            messages = (await (pipe.lrange(f"message:{self.session_id}",
0, -1).execute()))[0]
            message_objects = []
            for message in messages:
                message_str = message.split("::")
                message_obj = Message(client_id=message_str[1],
                                      username=message_str[2],
                                      message=message_str[3],

timestamp=datetime.fromisoformat(message_str[0]))
                message_objects.append(message_obj)
            return message_objects



    async def list(self) -> dict:
        messages = await self.get_all_messages()
        return [message.__str__() for message in messages]
```

app/utils/clients.py

```
from datetime import datetime

from typing import List, Optional

from uuid import uuid4, UUID

import json



from fastapi import WebSocket

from pydantic import UUID4

from aiortc import RTCPeerConnection, RTCSessionDescription,
RTCIceCandidate, RTCIceServer, RTCConfiguration



from app.utils.actions import Actions, DeviceTypes

from app.utils.chat import Message, Chat

from app.utils.recording import Recorder




class Client:

    def __init__(self, client_id: str, name: Optional[str] = None, room:
Optional["Room"] = None) -> None:

        self.id = client_id

        self.name: str = name or f"Undefined{self.id}"

        self.websocket: Optional[WebSocket] = None

        self.room: Optional[Room] = room

        self.camera_toggle: Optional[bool] = False
```

```
        self.microphone_toggle: Optional[bool] = False

        self.screen_sharing_toggle: Optional[bool] = False

        self.device_type: Optional[DeviceTypes] =
DeviceTypes.DESKTOP.value

        self.pc: Optional[RTCPeerConnection] = None

        self.recorder: Optional[Recorder] = None



    async def setup_rtc(self):

        """Set up RTC connection and attach a recorder."""

        ice_server = RTCIceServer(urls=['stun:stun.l.google.com:19302'])

        config = RTCConfiguration(iceServers=[ice_server])



        self.pc = RTCPeerConnection(configuration=config)



        if self.recorder is None:

            self.recorder = Recorder(self.room.id, self.id)

            self.recorder.setup()



        @self.pc.on("track")

        async def on_track(track):

            print("Got track:", track, track)



            self.recorder.add_track(track)
```

```
            await self.recorder.start()
            print("Recording started for the first track.")



        @self.pc.on("iceconnectionstatechange")
        async def on_ice_connection_state_change():
            print(f"ICE connection state changed:
{self.pc.iceConnectionState}")
            if self.pc.iceConnectionState == "failed":
                print("ICE connection failed. Closing connection.")
                await self.close()
            else:
                print("ICE connection state changed:",
self.pc.iceConnectionState)



        @self.pc.on("icecandidate")
        async def on_ice_candidate(event):
            print("ICE CANDIDATE", event)



        @self.pc.on("datachannel")
        async def on_datachannel(channel):
            print(f"changed datachannel to {channel}")



        @self.pc.on("signalingstatechange")
        async def on_signalingstatechange():
            print(f"changed signalingstatechange
{self.pc.signalingState}")
```

```
        @self.pc.on("icegatheringstatechange")
        async def on_icegatheringstatechange():
            print(f"changed icegatheringstatechange
{self.pc.iceGatheringState}")


    # async def start_recording(self):
    #     """Start the recorder."""
    #     if self.recorder:
    #         await self.recorder.start()


    # async def stop_recording(self):
    #     """Stop the recorder."""
    #     if self.recorder:
    #         await self.recorder.stop()


    def get_room(self) -> Optional["Room"]:
        return self.room


    def set_room(self, room: "Room"):
        self.room = room


    def open(self, websocket: Optional[WebSocket] = None) -> None:
        self.websocket = websocket
```

```
    def to_dict(self) -> dict:
        return {
            "id": str(self.id)
        }

    def get_toggles(self) -> dict:
        return {
            "camera_toggle": self.camera_toggle,
            "microphone_toggle": self.microphone_toggle,
            "screen_sharing_toggle": self.screen_sharing_toggle
        }


    async def handle_join(self, data, room: "Room"):
        client_id = data["client_id"]
        is_viewer = data.get("is_viewer", False)
        toggles = data.get("toggles", {})
        camera_toggle = toggles.get("camera_toggle", False)
        microphone_toggle = toggles.get("microphone_toggle", False)
        screen_sharing_toggle = toggles.get("screen_sharing_toggle",
False)
        device_type = data.get("device_type", DeviceTypes.DESKTOP.value)
        orientation = data.get("orientation", "landscape")


        self.camera_toggle = camera_toggle
```

```
        self.microphone_toggle = microphone_toggle
        self.screen_sharing_toggle = screen_sharing_toggle
        self.device_type = device_type
        self.orientation = orientation



        clients = room.get_clients()
        toggles = self.get_toggles()



        if is_viewer:
            for client in clients:
                if client.websocket is not None:
                    await client.websocket.send_json({"type":
Actions.ADD_PEER.value, "payload": { "peerID": client_id,

"client_name": self.name,

"createOffer": False,

"is_viewer": is_viewer,

"toggles": toggles,

"device_type": self.device_type,

"orientation": self.orientation}})
                if self.websocket is not None:
                    await self.websocket.send_json({"type":
Actions.ADD_PEER.value, "payload": { "peerID": str(client.id),
```

```
"client_name": client.name,

"createOffer": True,

"is_viewer": is_viewer,

"toggles": client.get_toggles(),

"device_type": client.device_type,

"orientation": client.orientation}})


        else:

            for client in clients:

                if client.websocket is not None:

                    await client.websocket.send_json({"type":
Actions.ADD_PEER.value, "payload": { "peerID": client_id,

"client_name": self.name,

"createOffer": True,

"is_viewer": is_viewer,

"toggles": toggles,

"device_type": self.device_type,

"orientation": self.orientation}})

                if self.websocket is not None:
```

```
                    await self.websocket.send_json({"type":
Actions.ADD_PEER.value, "payload": { "peerID": str(client.id),

"client_name": client.name,

"createOffer": False,

"is_viewer": is_viewer,

"toggles": client.get_toggles(),

"device_type": client.device_type,

"orientation": client.orientation}})



        data = await room.get_all_messages()
        await self.websocket.send_json(data)



        owner_id = room.get_owner()
        await self.websocket.send_json(
            {
                "type": Actions.OWNER.value,
                "payload": owner_id
            }
        )



        client_name = self.name
        await self.websocket.send_json(
```

```
            {
                "type": Actions.YOUR_NAME.value,
                "payload": client_name
            }
        )


        room.add_client(self)


        await self.setup_rtc()
        # await self.start_recording()


    async def handle_leave(self, data, room: "Room"):
        client_id = data["client_id"]


        clients = room.get_clients()


        await room.remove_client(self.id)


        for client in clients:
            if client.websocket is not None:
                await client.websocket.send_json({"type":
Actions.REMOVE_PEER.value, "payload": { "peerID": client_id }})
            if self.websocket is not None:
```

```
                await self.websocket.send_json({"type":
Actions.REMOVE_PEER.value, "payload": { "peerID": str(client.id) }})



        await self.websocket.close()

        self.websocket = None



        await self.stop_recording()  # Stop recording

        if self.pc:

            await self.pc.close()  # Close RTC connection

            self.pc = None



    async def handle_disconnect(self, room: "Room"):

        clients = room.get_clients()



        await room.remove_client(self.id)



        for client in clients:

            if client.websocket is not None:

                await client.websocket.send_json({"type":
Actions.REMOVE_PEER.value, "payload": { "peerID": str(self.id) }})



        if self.recorder:

            await self.recorder.stop()  # Stop the recording
```

```
        # Close WebSocket connection
        if self.websocket is not None:
            await self.websocket.close()
        self.websocket = None


        # Close RTC connection if exists
        if self.pc:
            await self.pc.close()
            self.pc = None


    async def handle_relay_sdp(self, data, room: "Room"):
        peer_id = data["peerID"]
        session_description = data["sessionDescription"]
        type = data["type"]


        client = room.get_client(client_id=peer_id)


        await client.websocket.send_json({"type":
Actions.SESSION_DESCRIPTION.value, "payload": { "peerID": str(self.id),
"sessionDescription": session_description, "type": type}})


    async def handle_relay_ice(self, data, room: "Room"):
        peer_id = data["peerID"]
```

```
        ice_candidate = data["iceCandidate"]



        client = room.get_client(client_id=peer_id)



        if client.websocket is not None:

            await client.websocket.send_json({"type":
Actions.ICE_CANDIDATE.value, "payload": { "peerID": str(self.id),
"iceCandidate": ice_candidate}})



    async def handle_new_message(self, data, room: "Room") -> None:

        clients = room.get_clients()

        data = await room.send_chat_message(client_id=self.id,
message=data)



        for client in clients:

            if client.websocket is not None:

                await client.websocket.send_json(data)



    async def handle_started_screen_sharing(self, data, room: "Room"):

        clients = room.get_clients()



        for client in clients:

            if client.id != self.id and client.websocket is not None:
```

```
                await client.websocket.send_json({"type":
Actions.STARTED_SCREEN_SHARING.value, "payload": { "peerID":
str(self.id) }})



        self.screen_sharing_toggle = True



    async def handle_stopped_screen_sharing(self, data, room: "Room"):
        clients = room.get_clients()



        for client in clients:
            if client.id != self.id and client.websocket is not None:
                await client.websocket.send_json({"type":
Actions.STOPPED_SCREEN_SHARING.value, "payload": { "peerID":
str(self.id) }})



        self.screen_sharing_toggle = False



    async def handle_enable_camera(self, data, room: "Room"):
        clients = room.get_clients()



        for client in clients:
            if client.id != self.id and client.websocket is not None:
                await client.websocket.send_json({"type":
Actions.ENABLE_CAMERA.value, "payload": { "peerID": str(self.id) }})
```

```
        self.camera_toggle = True



    async def handle_disable_camera(self, data, room: "Room"):
        clients = room.get_clients()



        for client in clients:
            if client.id != self.id and client.websocket is not None:
                await client.websocket.send_json({"type":
Actions.DISABLE_CAMERA.value, "payload": { "peerID": str(self.id) }})



        self.camera_toggle = False



    async def handle_enable_microphone(self, data, room: "Room"):
        clients = room.get_clients()



        for client in clients:
            if client.id != self.id and client.websocket is not None:
                await client.websocket.send_json({"type":
Actions.ENABLE_MICROPHONE.value, "payload": { "peerID": str(self.id) }})



        self.microphone_toggle = True



    async def handle_disable_microphone(self, data, room: "Room"):
```

```
        clients = room.get_clients()



        for client in clients:

            if client.id != self.id and client.websocket is not None:

                await client.websocket.send_json({"type":
Actions.DISABLE_MICROPHONE.value, "payload": { "peerID": str(self.id) }})



        self.microphone_toggle = False



    async def handle_rotate(self, data, room: "Room"):

        clients = room.get_clients()



        for client in clients:

            if client.id != self.id and client.websocket is not None:

                await client.websocket.send_json({"type":
Actions.ROTATE.value, "payload": { "peerID": str(self.id), "orientation":
self.orientation }})



        self.orientation = data["orientation"]



    async def handle_end_call(self, data, room: "Room"):

        clients = room.get_clients()



        for client in clients:
```

```
            await client.websocket.send_json({"type":
Actions.END_CALL.value})



    async def handle_ping(self):
        if self.websocket is not None:
            await self.websocket.send_json({"type": Actions.PONG.value})



    async def handle_offer(self, data, room: "Room"):
        offer = RTCSessionDescription(sdp=data["sdp"], type=data["type"])
        # offer = RTCSessionDescription(data)



        await self.pc.setRemoteDescription(offer)
        answer = await self.pc.createAnswer()
        await self.pc.setLocalDescription(answer)



        if self.websocket is not None:
            await self.websocket.send_json({"type":
Actions.SERVER_ANSWER.value,
                                            "payload": { "sdp":
self.pc.localDescription.sdp,
                                                         "type":
self.pc.localDescription.type }})



    async def handle_server_ice(self, data, room: "Room"):
        candidate = data["iceCandidate"]
```

```
        if candidate["candidate"] == "":
            return
        ip = candidate["candidate"].split(" ")[4]
        port = candidate["candidate"].split(" ")[5]
        protocol = candidate["candidate"].split(" ")[7]
        priority = candidate["candidate"].split(" ")[3]
        foundation = candidate["candidate"].split(" ")[0]
        component = candidate["candidate"].split(" ")[1]
        type = candidate["candidate"].split(" ")[7]
        ice_candidate = RTCIceCandidate(
            ip=ip,
            port=port,
            protocol=protocol,
            priority=priority,
            foundation=foundation,
            component=component,
            type=type,
            sdpMid=candidate["sdpMid"],
            sdpMLineIndex=candidate["sdpMLineIndex"]
        )


        await self.pc.addIceCandidate(ice_candidate)


    async def handle_server_answer(self, data, room: "Room"):
        answer = RTCSessionDescription(data)
```

```
        await self.pc.setRemoteDescription(answer)



    async def handle_message(self, message, room: "Room"):
        type = message.get("type")
        data = message.get("text")
        if type == Actions.DISCONNECT.value:
            await self.handle_disconnect(room)
            return
        data = json.loads(data)
        if data["type"] == Actions.JOIN.value:
            await self.handle_join(data["payload"], room)
        elif data["type"] == Actions.LEAVE.value:
            await self.handle_leave(data["payload"], room)
        elif data["type"] == Actions.RELAY_SDP.value:
            await self.handle_relay_sdp(data["payload"], room)
        elif data["type"] == Actions.RELAY_ICE.value:
            await self.handle_relay_ice(data["payload"], room)
        elif data["type"] == Actions.NEW_MSG.value:
            await self.handle_new_message(data["payload"], room)
        elif data["type"] == Actions.STARTED_SCREEN_SHARING.value:
            await self.handle_started_screen_sharing(data["payload"],
room)
        elif data["type"] == Actions.STOPPED_SCREEN_SHARING.value:
            await self.handle_stopped_screen_sharing(data["payload"],
room)
        elif data["type"] == Actions.END_CALL.value:
            await self.handle_end_call(data["payload"], room)
```

```
        elif data["type"] == Actions.ENABLE_CAMERA.value:
            await self.handle_enable_camera(data["payload"], room)
        elif data["type"] == Actions.DISABLE_CAMERA.value:
            await self.handle_disable_camera(data["payload"], room)
        elif data["type"] == Actions.ENABLE_MICROPHONE.value:
            await self.handle_enable_microphone(data["payload"], room)
        elif data["type"] == Actions.DISABLE_MICROPHONE.value:
            await self.handle_disable_microphone(data["payload"], room)
        elif data["type"] == Actions.ROTATE.value:
            await self.handle_rotate(data["payload"], room)
        elif data["type"] == Actions.PING.value:
            await self.handle_ping()
        elif data["type"] == Actions.SERVER_OFFER.value:
            await self.handle_offer(data["payload"], room)
        elif data["type"] == Actions.SERVER_ICE.value:
            await self.handle_server_ice(data["payload"], room)
        elif data["type"] == Actions.SERVER_ANSWER.value:
            await self.handle_server_answer(data["payload"], room)


    async def close(self) -> None:
        try:
            await self.websocket.close()
        except RuntimeError:
            pass
```

```python
class Room:

    def __init__(self, room_id: str | None, owner_id: str | None,
is_group: bool = False) -> None:

        self.id = room_id or uuid4()

        self.clients = {}

        self.chat: Chat = Chat(session_id=self.id)

        self.owner_id = owner_id

        self.is_group = is_group



    def add_client(self, client: Client) -> None:

        self.clients[client.id] = client



    async def remove_client(self, client_id) -> None:

        if client_id in self.clients:

            del self.clients[client_id]

        #if len(self.clients.keys()) == 0:

        #   await rooms.delete_room(self.id)



    def get_clients(self) -> List[Client]:

        return self.clients.values()


    def get_client(self, client_id: str) -> Client:

        return self.clients[client_id]


    def get_client_by_uuid(self, client_id: str) -> Client:
```

```
        return self.clients[client_id]


    def get_owner(self) -> str | None:
        return self.owner_id


    async def send_chat_message(self, client_id: str, message: str) ->
None:
        client = self.get_client_by_uuid(client_id=client_id)
        if client is None:
            return
        send_time = datetime.now()
        message_obj = Message(client_id=client_id, username=client.name,
message=message, timestamp=send_time)
        await self.chat.send_message(message=message_obj)
        data = {
            "type": Actions.NEW_MSG.value,
            "payload": message_obj.__str__()
        }
        return data


    async def get_all_messages(self) -> None:
        messages = await self.chat.get_all_messages()
        data = {
            "type": Actions.CHAT_HISTORY.value,
            "payload": [message.__str__() for message in messages]
        }
        return data
```

```
    def to_dict(self) -> dict:
        return {
            "id": str(self.id),
            "clients": [client.to_dict() for client in
self.clients.values()]
        }


    async def close(self) -> None:
        for client in list(self.clients.values()):
            await client.close()
        self.clients.clear()



class RoomsContainer:
    def __init__(self) -> None:
        self.rooms = {}


    def get_room(self, room_id: str) -> Optional[Room]:
        return self.rooms.get(room_id, None)


    def create_room(self, room_id: int, owner_id: str, is_group: bool =
False) -> Room:
        room = self.get_room(room_id=room_id)
        if room is None:
            room = Room(room_id, owner_id, is_group)
```

```
            self.rooms[room.id] = room
        return room


    async def delete_room(self, room_id: str) -> None:
        room = self.get_room(room_id=room_id)
        if room is not None:
            await room.close()
            del self.rooms[room_id]


    def get_rooms_info(self) -> dict:
        return [room.to_dict() for room in self.rooms.values()]


    async def clear(self) -> None:
        for room in self.rooms.values():
            await self.delete_room(room_id=room.id)
        self.rooms.clear()



class ClientsContainer:
    def __init__(self) -> None:
        self.clients = {}


    def get_client(self, client_id: str) -> Optional[Client]:
        return self.clients.get(client_id)
```

```python
    def create_client(self, client_id: str, name: Optional[str] = None,
room_id: Optional[str] = None) -> Client:
        if room_id is not None:
            room = rooms.get_room(room_id=room_id)
        else:
            room = None
        client = Client(client_id=client_id, name=name, room=room)
        self.clients[client.id] = client
        return client


    async def delete_client(self, client_id: str) -> None:
        client = self.get_client(client_id=client_id)
        await client.close()
        del self.clients[client_id]


    async def clear(self) -> None:
        for client in self.clients.values():
            await self.delete_client(client.id)
        self.clients.clear()



rooms = RoomsContainer()
clients = ClientsContainer()
```

app/utils/parser.py

```
def parse_ice_candidate(ice_candidate_dict):
    """
    Parses an ICE candidate JSON object and returns an instance of
aiortc.RTCIceCandidate.



    :param ice_candidate_json: The ICE candidate JSON object as a string.
    :return: An instance of aiortc.RTCIceCandidate.
    """
    # try:
    #     ice_candidate_dict = json.loads(ice_candidate_json)
    # except json.JSONDecodeError as e:
    #     raise ValueError(f"Invalid JSON: {e}")



    candidate = ice_candidate_dict.get('candidate')
    sdp_mid = ice_candidate_dict.get('sdpMid')
    sdp_mline_index = ice_candidate_dict.get('sdpMLineIndex')
    username_fragment = ice_candidate_dict.get('usernameFragment')



    if not candidate:
        raise ValueError("Missing 'candidate' field in the ICE candidate
JSON.")



    data = candidate.split()
```

```
related_address = None
related_port = None
tcp_type = None


if len(data) > 8:
    if data[8].startswith('raddr'):
        related_address = data[9]
        try:
            related_port = int(data[10])
        except ValueError:
            related_port = data[10]
    elif data[8].startswith('tcptype'):
        tcp_type = data[9]


out = {
    "component": int(data[1]),
    "foundation": data[0],
    "ip": data[4],
    "port": int(data[5]),
    "priority": int(data[3]),
    "protocol": data[2],
    "type": data[7],
    "relatedAddress": related_address,
    "relatedPort": related_port,
```

```
        "sdpMid": sdp_mid,
        "sdpMLineIndex": sdp_mline_index,
        "tcpType": tcp_type,
    }


    return out
```

app/utils/password.py

```
from typing import Tuple



from passlib import pwd
from passlib.context import CryptContext



pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")




def verify_and_update_password(plain_password: str, hashed_passord: str)
-> Tuple[bool, str]:
    return pwd_context.verify_and_update(plain_password, hashed_passord)



def get_password_hash(password: str) -> str:
    return pwd_context.hash(password)
```

```
def generate_password() -> str:
    return pwd.genword()
```

app/utils/recording.py

```
import os
import asyncio
from datetime import datetime
from enum import Enum
from aiortc import RTCPeerConnection, MediaStreamTrack
from aiortc.contrib.media import MediaRecorder
import moviepy



class RecorderTypes(Enum):
    WEBCAM = "webcam"
    SCREEN = "screen"


class Recorder:
    def __init__(self, room_id: str, client_id: str, base_dir: str =
"./recordings"):
        """
        Initialize the Recorder.
```

```
        :param room_id: ID of the room where the recording is taking
place.
        :param client_id: ID of the client associated with the recording.
        :param base_dir: Base directory to store recording files.
        """
        self.room_id = room_id
        self.client_id = client_id
        self.base_dir = base_dir
        self.recorders = []  # List to hold track-specific MediaRecorder
instances


        # Ensure the base directory exists
        self._ensure_directory()


    def _ensure_directory(self):
        """Ensure the recording directory exists."""
        room_path = os.path.join(self.base_dir, self.room_id)
        if not os.path.exists(room_path):
            os.makedirs(room_path)


    def setup(self):
        """Prepare the recorder (additional setup if needed)."""
        print(f"Recorder setup complete for room {self.room_id}, client
{self.client_id}")


    def add_track(self, track):
```

```
        """
        Add a track to the recorder.

        :param track: The media track to be recorded.
        """
        if track.kind == "audio":
            file_path = os.path.join(self.base_dir, self.room_id, f"{self.client_id}_audio.wav")
        elif track.kind == "video":
            file_path = os.path.join(self.base_dir, self.room_id, f"{self.client_id}_video.mp4")
        else:
            print(f"Unsupported track type: {track.kind}")
            return

        recorder = MediaRecorder(file_path)
        recorder.addTrack(track)
        self.recorders.append(recorder)

        print(f"Added {track.kind} track to recording: {file_path}")

    async def start(self):
        """Start recording all tracks."""
        for recorder in self.recorders:
            await recorder.start()
        print("Recording started.")
```

```
    async def stop(self):
        """Stop recording all tracks."""
        for recorder in self.recorders:
            await recorder.stop()
        print("Recording stopped and files saved.")


    def cleanup(self):
        """Clean up resources and remove incomplete files if needed."""
        print("Recorder cleanup complete.")
```

app/utils/redis.py

```
from typing import Optional


from fastapi.exceptions import HTTPException
from redis import Redis
from redis.asyncio import from_url
from redis.exceptions import ConnectionError


from app import settings
```

```
connection_url =
f"redis://{settings.REDIS_HOST}:{settings.REDIS_PORT}?decode_responses=Tru
e"



r = from_url(connection_url)



async def ping_redis_connection(r: Redis):
    try:
        await r.ping()
        print("Redis pinged. Successfully connected")
    except ConnectionError:
        raise HTTPException(
            status_code=500,
            detail=f"Redis error: failed to connect to redis database with
url {connection_url}"
        )
```

main.py

```
import uvicorn



from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
```

```python
from contextlib import asynccontextmanager



from app.db import init
from app import settings
from app.routes import router as users_router



def init_middlewares(app: FastAPI):
    app.add_middleware(
        CORSMiddleware,
        allow_origins=settings.CORS_ORIGINS,
        allow_credentials=settings.CORS_ALLOW_CREDENTIALS,
        allow_methods=settings.CORS_ALLOW_METHODS,
        allow_headers=settings.CORS_ALLOW_HEADERS
    )



app = FastAPI()



main_app_lifespan = app.router.lifespan_context
@asynccontextmanager
async def lifespan_wrapper(app):
    await init(app)
    async with main_app_lifespan(app) as maybe_state:
        yield maybe_state
```

```
app.router.lifespan_context = lifespan_wrapper



init_middlewares(app)

app.include_router(users_router)



# if __name__ == "__main__":

#     uvicorn.run(app, host=settings.API_HOST,
port=int(settings.API_PORT))
```