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

app/db.py

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

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

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})

@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})

@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

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

class UserOut(BaseUser):

uuid: UUID4

username: str

email: EmailStr

registration\_date: date

is\_admin: bool

app/settings.py

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

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)

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=True"

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

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))*