File: /Users/kh.kim/Documents/AIChat/.DS_Store

[binary]

File: /Users/kh.kim/Documents/AIChat/app/config.py

```python
import os
from redis import Redis
from urllib.parse import urlparse
import logging

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

redis_url = os.getenv('UPSTASH_REDIS_URL')  # UPSTASH_REDIS_URL

if redis_url:
    url = urlparse(redis_url)
    logger.debug(f"Parsed Redis URL: {url}")
    redis_client = Redis(
        host=url.hostname,
        port=url.port,
        password=url.password,
        ssl=url.scheme == 'rediss',
        ssl_cert_reqs=None,
        decode_responses=True
    )
else:
    redis_client = None
    logger.warning("No Redis URL provided")

def test_redis_connection():
    if redis_client is None:
        print("No Redis URL provided")
        return False
    try:
        redis_client.ping()
        print("Successfully connected to Redis")
        logger.info("Successfully connected to Redis")
        return True
    except Exception as e:
        print(f"Failed to connect to Redis: {e}")
        logger.error(f"Failed to connect to Redis: {e}")
        return False
```

File: /Users/kh.kim/Documents/AIChat/app/__init__.py

```python
from .main import app

def create_app():
    return app
```

File: /Users/kh.kim/Documents/AIChat/app/utils/__init__.py

```
[binary]
```

File: /Users/kh.kim/Documents/AIChat/app/utils/helpers.py

```
[binary]
```

File: /Users/kh.kim/Documents/AIChat/app/models/user.py

```python
from typing import Literal, Optional

from pydantic import BaseModel, EmailStr, Field


class UserBase(BaseModel):
    email: EmailStr
    nickname: str = Field(..., min_length=2, max_length=50)
    is_admin: bool = False

class UserCreate(UserBase):
    password: str = Field(..., min_length=8)

class UserUpdate(BaseModel):
    nickname: Optional[str] = Field(None, min_length=2, max_length=50)
    profile_image_url: Optional[str] = None
    is_admin: Optional[bool] = None

class SocialLoginData(BaseModel):
    provider: Literal["google", "kakao"]

class UserInDB(UserBase):
    id: str
    hashed_password: str
    login_type: str = "email"
    profile_image_url: Optional[str] = None

class UserProfile(UserBase):
    id: str
    profile_image_url: Optional[str]
    login_type: str

    class Config:
        from_attributes = True

class Token(BaseModel):
    access_token: str
    token_type: str

class TokenData(BaseModel):
    email: Optional[str] = None
```

File: /Users/kh.kim/Documents/AIChat/app/models/relationship.py

```python
from datetime import datetime, timezone
from enum import Enum
from typing import Optional

from pydantic import BaseModel


class RelationshipType(Enum):
    ENEMY = "enemy"
    RIVAL = "rival"
    STRANGER = "stranger"
    ACQUAINTANCE = "acquaintance"
    FRIEND = "friend"
    CLOSE_FRIEND = "close_friend"
    LOVER = "lover"
    SPOUSE = "spouse"

class UserCharacterInteractionBase(BaseModel):
    user_id: str
    character_id: str
    affinity: float = 0
    relationship_type: RelationshipType = RelationshipType.STRANGER
    nickname: Optional[str] = None
    last_interaction: datetime = datetime.now(timezone.utc)
    interaction_count: int = 0
    conversation_memory: int = 0
    learning_rate: float = 0.0
    custom_traits: dict = {}
    conversation_history: dict = {}

class UserCharacterInteractionCreate(UserCharacterInteractionBase):
    pass

class UserCharacterInteractionUpdate(BaseModel):
    affinity: Optional[float] = None
    relationship_type: Optional[RelationshipType] = None
    nickname: Optional[str] = None
    last_interaction: Optional[datetime] = None
    interaction_count: Optional[int] = None
    conversation_memory: Optional[int] = None
    learning_rate: Optional[float] = None
    custom_traits: Optional[dict] = None
    conversation_history: Optional[dict] = None

class UserCharacterInteractionInDB(UserCharacterInteractionBase):
    id: str

    class Config:
        orm_mode = True

```

File: /Users/kh.kim/Documents/AIChat/app/models/conversation.py

```python
import uuid
from datetime import datetime
from typing import Dict, List, Literal, Optional

from pydantic import BaseModel, Field


class ConversationBase(BaseModel):
    # user_id: uuid.UUID
    # character_id: uuid.UUID
    user_id: str = Field(..., description="User ID as string")
    character_id: str = Field(..., description="Character ID as string")
    context: Optional[Dict] = Field(default_factory=dict)
    state: Optional[Dict] = Field(default_factory=dict)

class ConversationCreate(ConversationBase):
    pass

class ConversationUpdate(BaseModel):
    context: Optional[Dict] = None
    state: Optional[Dict] = None

class ConversationInDB(ConversationBase):
    id: uuid.UUID
    created_at: datetime
    updated_at: datetime

    class Config:
        from_attributes = True

class ConversationProfile(ConversationInDB):
    pass

class MessageBase(BaseModel):
    conversation_id: uuid.UUID
    sender_type: Literal['user', 'character']
    content: List[Dict]
    metadata: Optional[Dict] = Field(default_factory=dict)

class MessageCreate(MessageBase):
    pass

class MessageUpdate(BaseModel):
    content: Optional[List[Dict]] = None
    metadata: Optional[Dict] = None

class MessageInDB(MessageBase):
    id: uuid.UUID
    created_at: datetime
    embedding: Optional[List[float]] = None

    class Config:
        from_attributes = True

class MessageProfile(MessageBase):
    id: uuid.UUID
    created_at: datetime

    class Config:
        from_attributes = True
```

File: /Users/kh.kim/Documents/AIChat/app/models/__init__.py

```
[binary]
```

File: /Users/kh.kim/Documents/AIChat/app/models/chat.py

```
[binary]
```

File: /Users/kh.kim/Documents/AIChat/app/models/scenario.py

```
# # models/scenario.py

# from pydantic import BaseModel
# from typing import List, Optional
# from enum import Enum
# from datetime import datetime

# class ScenarioTriggerType(Enum):
#     """     """
#     AFFINITY = "affinity"  #
#     TIME = "time"  #
#     EVENT = "event"  #

# class ScenarioStep(BaseModel):
#     """    """
#     step_id: str
#     content: str
#     image_url: Optional[str] = None  #     URL ()

# class Scenario(BaseModel):
#     """    """
#     id: str
#     character_id: str
#     title: str
#     description: str
#     trigger_type: ScenarioTriggerType
#     trigger_value: float  # :  60  7
#     steps: List[ScenarioStep]
#     created_at: datetime
#     updated_at: datetime

# class ScenarioProgress(BaseModel):
#     """     """
#     id: str
#     user_id: str
#     scenario_id: str
#     current_step: int
#     started_at: datetime
#     completed_at: Optional[datetime] = None
#     is_completed: bool = False

# class ScenarioCreate(BaseModel):
#     """    """
#     character_id: str
#     title: str
#     description: str
#     trigger_type: ScenarioTriggerType
#     trigger_value: float
#     steps: List[ScenarioStep]

# class ScenarioUpdate(BaseModel):
#     """    """
#     title: Optional[str] = None
#     description: Optional[str] = None
#     trigger_type: Optional[ScenarioTriggerType] = None
#     trigger_value: Optional[float] = None
#     steps: Optional[List[ScenarioStep]] = None
```

File: /Users/kh.kim/Documents/AIChat/app/models/character.py

```python
import uuid
from datetime import datetime
from typing import Dict, List, Optional

from pydantic import BaseModel, Field


class LocalizedContent(BaseModel):
    ko: Optional[str] = Field(None, description="Korean content")
    en: Optional[str] = Field(None, description="English content")
    ja: Optional[str] = Field(None, description="Japanese content")

    class Config:
        extra = 'forbid'  # This prevents additional fields


class LanguageProficiency(BaseModel):
    language_code: str
    proficiency: str
    preference_order: int = Field(ge=1)

class PersonalityTrait(BaseModel):
    trait: str
    score: float = Field(ge=0.0, le=1.0)

class Interest(BaseModel):
    topic: str
    level: str

class CharacterBase(BaseModel):
    version: str
    names: LocalizedContent
    gender: str
    age: int = Field(ge=0, le=150)
    personality_traits: List[PersonalityTrait]
    interests: List[Interest]
    occupation: LocalizedContent
    background: LocalizedContent
    appearance_seed: str
    appearance_description: LocalizedContent
    relationship_status: Optional[str] = None
    languages: List[LanguageProficiency]
    conversation_style: LocalizedContent
    communication_preferences: Dict[str, str]
    backstory: LocalizedContent
    goals: LocalizedContent
    quirks: LocalizedContent
    emotional_intelligence: float = Field(ge=0.0, le=1.0)
    cultural_sensitivity: float = Field(ge=0.0, le=1.0)
    relationship_progression_pace: str
    conflict_resolution_style: str
    interaction_prompts: Dict[str, LocalizedContent]
    character_prompt: str
    response_generation_parameters: Dict[str, float]
    is_public: bool = False

class CharacterCreate(CharacterBase):
    pass

class CharacterUpdate(BaseModel):
    version: Optional[str] = None
    names: Optional[LocalizedContent] = None
    gender: Optional[str] = None
    age: Optional[int] = Field(None, ge=0, le=150)
    personality_traits: Optional[List[PersonalityTrait]] = None
    interests: Optional[List[Interest]] = None
    occupation: Optional[LocalizedContent] = None
    background: Optional[LocalizedContent] = None
    appearance_seed: Optional[str] = None
    appearance_description: Optional[LocalizedContent] = None
    relationship_status: Optional[str] = None
    languages: Optional[List[LanguageProficiency]] = None
    conversation_style: Optional[LocalizedContent] = None
    communication_preferences: Optional[Dict[str, str]] = None
    backstory: Optional[LocalizedContent] = None
    goals: Optional[LocalizedContent] = None
    quirks: Optional[LocalizedContent] = None
    emotional_intelligence: Optional[float] = Field(None, ge=0.0, le=1.0)
    cultural_sensitivity: Optional[float] = Field(None, ge=0.0, le=1.0)
    relationship_progression_pace: Optional[str] = None
    conflict_resolution_style: Optional[str] = None
    interaction_prompts: Optional[Dict[str, LocalizedContent]] = None
    character_prompt: Optional[str] = None
    response_generation_parameters: Optional[Dict[str, float]] = None
    is_public: Optional[bool] = None

class CharacterInDB(CharacterBase):
    id: uuid.UUID
    creator_id: str
    created_at: datetime
    updated_at: datetime

    class Config:
        from_attributes = True

class CharacterProfile(CharacterBase):
    id: uuid.UUID
    creator_id: str

    class Config:
        from_attributes = True
```

File: /Users/kh.kim/Documents/AIChat/app/main.py

```python
import logging
import os
import sys
from contextlib import asynccontextmanager

from fastapi import FastAPI, Request
from fastapi.middleware.cors import CORSMiddleware
from supabase import Client, create_client

import redis

from app.config import redis_client
from app.routes.characters import router as characters_router
from app.routes.conversations import router as conversations_router
from app.routes.users import router as users_router
# from app.routes.scenarios import router as scenarios_router
from app.services.auth_service import router as auth_router

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

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

@asynccontextmanager
async def lifespan(app: FastAPI):
    logger.info("Application is starting up")
    from app.config import test_redis_connection, redis_client

    if test_redis_connection():
        logger.info("Successfully connected to Redis")
    else:
        logger.warning("Continuing without Redis connection")

    yield

    if redis_client:
        redis_client.close()
        logger.info("Redis connection closed")
    logger.info("Application is shutting down")

app = FastAPI(debug=True, lifespan=lifespan)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  #
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

supabase_url = os.getenv("SUPABASE_URL")
supabase_key = os.getenv("SUPABASE_KEY")
if not supabase_url or not supabase_key:
    raise ValueError("SUPABASE_URL and SUPABASE_KEY must be set in environment variables")

supabase: Client = create_client(supabase_url, supabase_key)
app.state.supabase = supabase

@app.middleware("http")
async def log_requests(request: Request, call_next):
    logger.info(f"Request path: {request.url.path}")
    response = await call_next(request)
    logger.info(f"Response status code: {response.status_code}")
    return response

app.include_router(users_router)
app.include_router(characters_router)
app.include_router(conversations_router)
# app.include_router(scenarios_router)
app.include_router(auth_router, prefix="/auth")

@app.get("/")
async def root():
    return {"message": "Hello World"}

if __name__ == "__main__":
    import uvicorn
    port = int(os.environ.get("PORT", 8000))
    uvicorn.run("app.main:app", host="0.0.0.0", port=port, reload=True)


```

File: /Users/kh.kim/Documents/AIChat/app/routes/scenarios.py

```
# # routes/scenarios.py

# from fastapi import APIRouter, HTTPException, Depends
# from typing import List
# from app.models.scenario import ScenarioCreate, ScenarioUpdate, Scenario, ScenarioProgress
# from app.services import scenario_service
# from app.models.user import UserProfile as User
# from app.dependencies import get_current_user

# router = APIRouter()

# @router.post("/scenarios", response_model=Scenario)
# async def create_scenario(scenario: ScenarioCreate, current_user: User = Depends(get_current_user)):
#     """
#       .
#     """
#     return await scenario_service.create_scenario(scenario)

# @router.get("/scenarios/{scenario_id}", response_model=Scenario)
# async def get_scenario(scenario_id: str, current_user: User = Depends(get_current_user)):
#     """
#        .
#     """
#     scenario = await scenario_service.get_scenario(scenario_id)
#     if not scenario:
#         raise HTTPException(status_code=404, detail="Scenario not found")
#     return scenario

# @router.put("/scenarios/{scenario_id}", response_model=Scenario)
# async def update_scenario(scenario_id: str, scenario_update: ScenarioUpdate, current_user: User = Depends(get_current_user)):
#     """
#       .
#     """
#     updated_scenario = await scenario_service.update_scenario(scenario_id, scenario_update)
#     if not updated_scenario:
#         raise HTTPException(status_code=404, detail="Scenario not found")
#     return updated_scenario

# @router.delete("/scenarios/{scenario_id}", response_model=bool)
# async def delete_scenario(scenario_id: str, current_user: User = Depends(get_current_user)):
#     """
#      .
#     """
#     deleted = await scenario_service.delete_scenario(scenario_id)
#     if not deleted:
#         raise HTTPException(status_code=404, detail="Scenario not found")
#     return True

# @router.post("/scenarios/{scenario_id}/start", response_model=ScenarioProgress)
# async def start_scenario(scenario_id: str, current_user: User = Depends(get_current_user)):
#     """
#         .
#     """
#     try:
#         progress = await scenario_service.start_scenario(str(current_user.id), scenario_id)
#         return progress
#     except ValueError as e:
#         raise HTTPException(status_code=400, detail=str(e))

# @router.post("/scenarios/{scenario_id}/progress", response_model=ScenarioProgress)
# async def progress_scenario(scenario_id: str, current_user: User = Depends(get_current_user)):
#     """
#        .
#     """
#     try:
#         progress = await scenario_service.progress_scenario(str(current_user.id), scenario_id)
#         return progress
#     except ValueError as e:
#         raise HTTPException(status_code=400, detail=str(e))

# @router.get("/scenarios/check_trigger", response_model=Scenario)
# async def check_scenario_trigger(character_id: str, current_user: User = Depends(get_current_user)):
#     """
#           .
#     """
#     scenario = await scenario_service.check_scenario_trigger(str(current_user.id), character_id)
#     if not scenario:
#         raise HTTPException(status_code=404, detail="No scenario triggered")
#     return scenario

# @router.get("/scenarios", response_model=List[Scenario])
# async def list_scenarios(character_id: str, current_user: User = Depends(get_current_user)):
#     """
#         .
#     """
#     scenarios = await scenario_service.get_all_scenarios_for_character(character_id)
#     return scenarios

# @router.get("/scenarios/{scenario_id}/progress", response_model=ScenarioProgress)
# async def get_scenario_progress(scenario_id: str, current_user: User = Depends(get_current_user)):
#     """
#          .
#     """
#     progress = await scenario_service.get_scenario_progress(str(current_user.id), scenario_id)
#     if not progress:
#         raise HTTPException(status_code=404, detail="Scenario progress not found")
#     return progress
```

File: /Users/kh.kim/Documents/AIChat/app/routes/users.py

```python
from typing import List

from fastapi import APIRouter, Depends, HTTPException, Request
from fastapi.responses import JSONResponse
from pydantic import BaseModel

from app.models.user import UserBase, UserCreate, UserProfile, UserUpdate
from app.services.auth_service import (auth_callback, get_current_user,
                                       get_linked_accounts, get_user_profile,
                                       login_user, logout_user, process_token,
                                       register_user, social_login,
                                       update_user_profile)

router = APIRouter()

@router.post("/register")
async def register(user: UserCreate):
    try:
        result = register_user(user.email, user.password, user.nickname)
        return result
    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))

class UserBase(BaseModel):
    email: str
    password: str

@router.post("/login")
async def login(user: UserBase):
    try:
        result = login_user(user.email, user.password)
        return result
    except Exception as e:
        raise HTTPException(status_code=401, detail=str(e))

@router.post("/social-login/{provider}")
async def social_login_route(provider: str, request: Request):
    try:
        response = social_login(provider, request)
        return JSONResponse(content=response)
    except HTTPException as e:
        return JSONResponse(content={"detail": e.detail}, status_code=e.status_code)

@router.get("/auth/callback")
async def auth_callback_route(request: Request):
    return await auth_callback(request)

@router.post("/process_token")
async def process_token_route(token_data: dict):
    try:
        response = await process_token(token_data)
        return JSONResponse(content=response)
    except HTTPException as e:
        return JSONResponse(content={"detail": e.detail}, status_code=e.status_code)

@router.get("/profile", response_model=UserProfile)
async def get_profile(user=Depends(get_current_user)):
    return await get_user_profile(user)

@router.put("/profile", response_model=UserProfile)
async def update_profile(user_update: UserUpdate, user=Depends(get_current_user)):
    return await update_user_profile(user_update, user)

@router.post("/logout")
async def logout(user=Depends(get_current_user)):
    return await logout_user()

@router.get("/linked-accounts", response_model=List[str])
async def linked_accounts(user=Depends(get_current_user)):
    return await get_linked_accounts(user)
```

File: /Users/kh.kim/Documents/AIChat/app/routes/__init__.py

```
[binary]
```

File: /Users/kh.kim/Documents/AIChat/app/routes/characters.py

```python
from typing import List

from fastapi import APIRouter, Depends

from app.models.character import (CharacterCreate, CharacterProfile,
                                  CharacterUpdate)
from app.models.user import UserProfile as User
from app.services.auth_service import get_current_user
from app.services.character_service import (create_character, delete_character,
                                            get_character, list_characters,
                                            update_character)

router = APIRouter()

@router.post("/characters", response_model=CharacterProfile)
async def create_character_route(character: CharacterCreate, current_user: User = Depends(get_current_user)):
    return await create_character(character, current_user)

@router.get("/characters/{character_id}", response_model=CharacterProfile)
async def get_character_route(character_id: str, current_user: User = Depends(get_current_user)):
    return await get_character(character_id, current_user)

@router.put("/characters/{character_id}", response_model=CharacterProfile)
async def update_character_route(character_id: str, character: CharacterUpdate, current_user: User = Depends(get_current_user)):
    return await update_character(character_id, character, current_user)

@router.delete("/characters/{character_id}")
async def delete_character_route(character_id: str, current_user: User = Depends(get_current_user)):
    await delete_character(character_id, current_user)
    return {"message": "Character deleted successfully"}

@router.get("/characters", response_model=List[CharacterProfile])
async def list_characters_route(current_user: User = Depends(get_current_user)):
    return await list_characters(current_user)
```

File: /Users/kh.kim/Documents/AIChat/app/routes/conversations.py

```python
from typing import List

from fastapi import APIRouter, Depends, Query

from app.models.conversation import (ConversationCreate, ConversationProfile,
                                     ConversationUpdate, MessageCreate,
                                     MessageProfile)
from app.models.relationship import (RelationshipType,
                                     UserCharacterInteractionUpdate)
from app.models.user import UserProfile as User
from app.services.auth_service import get_current_user
from app.services.conversation_service import ConversationService

router = APIRouter()
conversation_service = ConversationService()

@router.post("/conversations", response_model=ConversationProfile)
async def create_conversation_route(conversation: ConversationCreate, current_user: User = Depends(get_current_user)):
    return await conversation_service.create_conversation(conversation, current_user)

@router.get("/conversations/{conversation_id}", response_model=ConversationProfile)
async def get_conversation_route(conversation_id: str, current_user: User = Depends(get_current_user)):
    return await conversation_service.get_conversation(conversation_id, current_user)

@router.put("/conversations/{conversation_id}", response_model=ConversationProfile)
async def update_conversation_route(conversation_id: str, conversation: ConversationUpdate, current_user: User = Depends(get_current_user)):
    return await conversation_service.update_conversation(conversation_id, conversation, current_user)

@router.delete("/conversations/{conversation_id}")
async def delete_conversation_route(conversation_id: str, current_user: User = Depends(get_current_user)):
    await conversation_service.delete_conversation(conversation_id, current_user)
    return {"message": "Conversation deleted successfully"}

@router.get("/conversations", response_model=List[ConversationProfile])
async def list_conversations_route(current_user: User = Depends(get_current_user)):
    return await conversation_service.list_conversations(current_user)

@router.post("/conversations/{conversation_id}/messages", response_model=MessageProfile)
async def create_message_route(conversation_id: str, message: MessageCreate, current_user: User = Depends(get_current_user)):
    return await conversation_service.create_message_and_respond(conversation_id, message, current_user)

@router.get("/conversations/{conversation_id}/messages", response_model=List[MessageProfile])
async def list_messages_route(conversation_id: str, current_user: User = Depends(get_current_user)):
    return await conversation_service.list_messages(conversation_id, current_user)

@router.get("/conversations/{conversation_id}/messages/{message_id}", response_model=MessageProfile)
async def get_message_route(conversation_id: str, message_id: str, current_user: User = Depends(get_current_user)):
    return await conversation_service.get_message(message_id, current_user)

@router.post("/conversations/{conversation_id}/summarize")
async def summarize_conversation_route(conversation_id: str, current_user: User = Depends(get_current_user)):
    summary = await conversation_service.summarize_conversation(conversation_id, current_user)
    return {"summary": summary}

@router.get("/conversations/{conversation_id}/similar-messages")
async def get_similar_messages_route(
    conversation_id: str,
    message_content: str = Query(..., description="Content of the message to find similar ones"),
    top_k: int = Query(5, description="Number of similar messages to return"),
    current_user: User = Depends(get_current_user)
):
    similar_messages = await conversation_service.get_similar_messages(conversation_id, message_content, top_k)
    return {"similar_messages": similar_messages}

@router.get("/conversations/{conversation_id}/message-count")
async def get_message_count_route(conversation_id: str, current_user: User = Depends(get_current_user)):
    count = await conversation_service.get_message_count(conversation_id)
    return {"message_count": count}

@router.put("/conversations/{conversation_id}/nickname")
async def update_nickname_route(
    conversation_id: str,
    nickname: str = Query(..., description="New nickname for the user"),
    current_user: User = Depends(get_current_user)
):
    conversation = await conversation_service.get_conversation(conversation_id, current_user)
    await conversation_service.relationship_service.update_interaction(
        str(conversation.character_id),
        str(current_user.id),
        UserCharacterInteractionUpdate(nickname=nickname)
    )
    return {"message": "Nickname updated successfully"}

@router.put("/conversations/{conversation_id}/relationship-type")
async def update_relationship_type_route(
    conversation_id: str,
    relationship_type: RelationshipType,
    current_user: User = Depends(get_current_user)
):
    conversation = await conversation_service.get_conversation(conversation_id, current_user)
    await conversation_service.relationship_service.update_interaction(
        str(conversation.character_id),
        str(current_user.id),
        relationship_type
    )
    return {"message": "Relationship type updated successfully"}
```

File: /Users/kh.kim/Documents/AIChat/app/services/ai_service.py

```python
import asyncio
import os
from typing import Any, Dict, List
from dotenv import load_dotenv

# .env
load_dotenv()

import openai
from pinecone import Pinecone, ServerlessSpec


class AIService:
    def __init__(self):
        openai.api_key = os.environ.get('OPENAI_API_KEY')

        # Pinecone
        self.pc = Pinecone(api_key=os.environ.get("PINECONE_API_KEY"))

        #
        index_name = os.environ.get("PINECONE_INDEX_NAME")

        #
        if index_name not in self.pc.list_indexes().names():
            self.pc.create_index(
                name=index_name,
                dimension=1536,  # OpenAI text-embedding-ada-002
                metric='cosine',
                spec=ServerlessSpec(cloud=os.environ.get("PINECONE_CLOUD", "aws"),
                                    region=os.environ.get("PINECONE_REGION", "us-west-2"))
            )

        #
        self.index = self.pc.Index(index_name)

    def vectorize_text(self, text: str) -> List[float]:
        """   """
        response = openai.Embedding.create(
            input=text,
            model="text-embedding-ada-002"
        )
        embedding = response['data'][0]['embedding']
        return embedding

    def store_vector(self, id: str, vector: List[float], metadata: Dict[str, Any]):
        """ Pinecone  """
        self.index.upsert(vectors=[(id, vector, metadata)])

    async def store_vector_async(self, id: str, vector: List[float], metadata: Dict[str, Any]):
        """ Pinecone   """
        #    run_in_executor
        await asyncio.get_event_loop().run_in_executor(
            None, self.store_vector, id, vector, metadata
        )


    def search_similar_vectors(self, vector: List[float], conversation_id: str, top_k: int = 5) -> List[Dict[str, Any]]:
        """


        :param vector:
        :param conversation_id:    ID
        :param top_k:
        :return:    (ID, , )
        """
        results = self.index.query(
            vector=vector,
            top_k=top_k,
            include_metadata=True,
            filter={"conversation_id": conversation_id}
        )
        return results['matches']

    async def generate_response(self, context: str) -> str:
        """
           AI

        :param context:
        :return:  AI
        """
        # TODO:  AI
        #    AI  API    .
        return f"AI response based on context: {context[:50]}..."  #
```

File: /Users/kh.kim/Documents/AIChat/app/services/auth_service.py

```python
import logging
import os

from fastapi import APIRouter, Depends, HTTPException, Request
from fastapi.responses import HTMLResponse
from fastapi.security import HTTPAuthorizationCredentials, HTTPBearer
from pydantic import BaseModel
from supabase import Client, create_client

router = APIRouter()

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

supabase_url = os.getenv("SUPABASE_URL")
supabase_key = os.getenv("SUPABASE_KEY")
if not supabase_url or not supabase_key:
    raise ValueError("SUPABASE_URL and SUPABASE_KEY must be set in .env file")

supabase: Client = create_client(supabase_url, supabase_key)

security = HTTPBearer()

class User(BaseModel):
    id: str
    email: str
    is_admin: bool = False  # is_admin


def get_supabase_token(email: str, password: str) -> str:
    try:
        response = supabase.auth.sign_in_with_password({"email": email, "password": password})
        if response and response.session:
            return response.session.access_token
        else:
            raise HTTPException(status_code=401, detail="Unable to authenticate with Supabase")
    except Exception as e:
        print(f"Supabase Auth Error: {str(e)}")
        raise HTTPException(status_code=401, detail="Supabase authentication error")

```

```python
async def get_current_user(credentials: HTTPAuthorizationCredentials = Depends(security)):
    token = credentials.credentials
    try:
        response = supabase.auth.get_user(token)
        if response and response.user:
            # is_admin
            user_data = supabase.table("users").select("is_admin").eq("id", response.user.id).single().execute()
            is_admin = user_data.data.get('is_admin', False) if user_data.data else False
            return User(id=response.user.id, email=response.user.email, is_admin=is_admin)  # is_admin
        raise HTTPException(status_code=401, detail="Invalid authentication credentials")
    except Exception as e:
        raise HTTPException(status_code=401, detail=f"Invalid authentication credentials: {str(e)}")


def register_user(email: str, password: str, nickname: str):
    try:
        auth_response = supabase.auth.sign_up({
            "email": email,
            "password": password
        })

        if auth_response.user:
            user_data = supabase.table("users").insert({
                "id": auth_response.user.id,
                "email": email,
                "nickname": nickname,
                "login_type": "email",
                "is_admin": False  #
            }).execute()

            logger.info(f"User data insert response: {user_data}")

            return {
                "message": "User registered successfully",
                "user_id": auth_response.user.id
            }
        else:
            raise HTTPException(status_code=400, detail="Registration failed")
    except Exception as e:
        logger.error(f"Registration error: {str(e)}")
        raise HTTPException(status_code=400, detail=str(e))



def login_user(email: str, password: str):
    try:
        logger.info(f"Attempting login for email: {email}")
        auth_response = supabase.auth.sign_in_with_password({"email": email, "password": password})
        logger.info(f"Auth response: {auth_response}")

        if auth_response.user and auth_response.session:
            return {
                "message": "Login successful",
                "access_token": auth_response.session.access_token,
                "user_id": auth_response.user.id
            }
        else:
            logger.error("Login failed: User or session not found in auth response")
            raise HTTPException(status_code=401, detail="Invalid credentials")
    except Exception as e:
        logger.error(f"Login error: {str(e)}")
        if "Invalid login credentials" in str(e):
            raise HTTPException(status_code=401, detail="Invalid email or password")
        elif "Email not confirmed" in str(e):
            raise HTTPException(status_code=401, detail="Email not confirmed. Please check your email for verification link.")
        else:
            raise HTTPException(status_code=401, detail="Login failed. Please try again.")

def social_login(provider: str, request: Request):
    try:
        callback_url = "http://localhost:8000/auth/callback"
        logger.info(f"Callback URL: {callback_url}")
        auth_response = supabase.auth.sign_in_with_oauth({
            "provider": provider,
            "options": {
                "redirect_to": callback_url
            }
        })

        logger.info(f"Auth response: {auth_response}")
        if hasattr(auth_response, 'url'):
            return {"url": auth_response.url}
        else:
            raise HTTPException(status_code=400, detail="OAuth initialization failed")
    except Exception as e:
        logger.error(f"Social Login Error: {str(e)}")
        raise HTTPException(status_code=400, detail=f"Social login error: {str(e)}")


async def auth_callback(request: Request):
    try:
        logger.debug("Auth callback hit")
        code = request.query_params.get('code')
        error = request.query_params.get('error')
        logger.debug(f"Received code: {code}, error: {error}")

        # URL    HTML
        html_content = """
        <html>
        <body>
            <script>
                function sendTokenToFrontend() {
                    var hash = window.location.hash.substring(1);
                    var params = new URLSearchParams(hash);
                    var access_token = params.get('access_token');

                    if (!access_token) {
                        window.location.href = 'https://localhost:3000/auth-error?error=no_token';
                        return;
                    }

                    //
                    window.location.href = 'https://localhost:3000/auth-success?token=' + access_token;
                }
                window.onload = sendTokenToFrontend;
            </script>
            <h1>Processing authentication...</h1>
        </body>
        </html>
        """
        return HTMLResponse(content=html_content)
    except Exception as e:
        logger.exception(f"Error in auth_callback: {str(e)}")
        raise HTTPException(status_code=400, detail=f"Error processing authentication: {str(e)}")

# async def auth_callback(request: Request):
#     try:
#         logger.debug("Auth callback hit")
#         code = request.query_params.get('code')
#         error = request.query_params.get('error')
#         logger.debug(f"Received code: {code}, error: {error}")


#         # URL    HTML
#         html_content = """
#         <html>
#         <body>
#             <script>
#                 function sendTokenToServer() {
#                     var hash = window.location.hash.substring(1);
#                     var params = new URLSearchParams(hash);
#                     var access_token = params.get('access_token');

#                     if (!access_token) {
#                         document.body.innerHTML = '<h1>Error: No access token found</h1>';
#                         return;
#                     }

#                     fetch('/process_token', {
#                         method: 'POST',
#                         headers: {
#                             'Content-Type': 'application/json',
#                         },
#                         body: JSON.stringify({access_token: access_token}),
#                     })
#                     .then(response => {
#                         if (!response.ok) {
#                             return response.json().then(err => {
#                                 throw new Error(err.detail || 'Unknown error occurred');
#                             });
#                         }
#                         return response.json();
#                     })
#                     .then(data => {
#                         if (data.message && data.user_id) {
#                             document.body.innerHTML = `<h1>${data.message}</h1><p>User ID: ${data.user_id}</p>`;
#                             setTimeout(() => {
#                                 window.location.href = '/';
#                             }, 2000);  // Redirect back to the original page after 3 seconds
#                         } else {
#                             throw new Error('Invalid response data');
#                         }
#                     })
#                     .catch((error) => {
#                         console.error('Error:', error);
#                         document.body.innerHTML = '<h1>Error occurred during authentication</h1><p>' + error.message + '</p>';
#                     });
#                 }
#                 window.onload = sendTokenToServer;
#             </script>
#             <h1>Processing authentication...</h1>
#         </body>
#         </html>
#         """
#         return HTMLResponse(content=html_content)
#     except Exception as e:
#         logger.exception(f"Error in auth_callback: {str(e)}")
#         raise HTTPException(status_code=400, detail=f"Error processing authentication: {str(e)}")


async def process_token(token_data: dict):
    try:
        access_token = token_data.get('access_token')
        if not access_token:
            raise HTTPException(status_code=400, detail="Access token not provided")

        #
        user = supabase.auth.get_user(access_token)

        if not user or not user.user:
            raise HTTPException(status_code=400, detail="User information not found")

        user_id = user.user.id
        user_email = user.user.email

        #
        user_data = supabase.table("users").select("*").eq("id", user_id).execute()
        logger.debug(f"User data: {user_data}")

        if user_data.data:
            #
            message = f"Successfully logged in."
            logger.debug(f"Existing user logged in: {user_id}")
        else:
            #
            new_user = {
                "id": user_id,
                "email": user_email,
                "nickname": f"User_{user_id[:8]}",
                "login_type": "social",
                "is_admin": False  #
            }
            insert_result = supabase.table("users").insert(new_user).execute()
            logger.debug(f"Insert result: {insert_result}")
            if not insert_result.data:
                raise HTTPException(status_code=500, detail="Failed to create new user")
            message = f"New user successfully created."
            logger.debug(f"New user created: {user_id}")

        response_data = {
            "message": message,
            "user_id": user_id
        }
        logger.debug(f"Returning response: {response_data}")
        return response_data

    except Exception as e:
        logger.exception(f"Detailed error in process_token: {str(e)}")
        raise HTTPException(status_code=400, detail=f"Error processing authentication: {str(e)}")

async def get_user_profile(user: User = Depends(get_current_user)):
    try:
        logger.debug(f"Fetching profile for user ID: {user.id}")
        user_data = supabase.table("users").select("*").eq("id", user.id).single().execute()
        if user_data.data:
            logger.debug(f"User data retrieved: {user_data.data}")
            # is_admin
            return {**user_data.data, "is_admin": user.is_admin}
        else:
            logger.error("User profile not found")
            raise HTTPException(status_code=404, detail="User profile not found")
    except Exception as e:
        logger.error(f"Error fetching user profile: {str(e)}")
        raise HTTPException(status_code=400, detail=str(e))

async def update_user_profile(user_update, user):
    try:
        update_data = user_update.dict(exclude_unset=True)
        user_data = supabase.table("users").update(update_data).eq("id", user.id).execute()
        if user_data and user_data.get("data"):
            return user_data["data"][0]
        else:
            raise HTTPException(status_code=404, detail="User profile not found")
    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))

async def logout_user():
    try:
        supabase.auth.sign_out()
        return {"message": "Logout successful"}
    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))

async def get_linked_accounts(user):
    try:
        user_data = supabase.table("users").select("login_type").eq("id", user.id).execute()
        if user_data and user_data.get("data"):
            return [user_data["data"][0]['login_type']]
        else:
            return []
    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))
```

File: /Users/kh.kim/Documents/AIChat/app/services/chat_service.py

```
[binary]
```

File: /Users/kh.kim/Documents/AIChat/app/services/conversation_service.py

```
import json
import logging
import os
from datetime import datetime, timezone
from typing import Dict, List

import tiktoken
from fastapi import APIRouter, HTTPException
from langchain_core.prompts import PromptTemplate
from langchain_core.runnables import RunnableSequence
from langchain_openai import OpenAI, ChatOpenAI
from langchain_core.documents import Document
from langchain.memory import ConversationBufferWindowMemory
from langchain_core.messages import AIMessage, HumanMessage
from langchain.chains.summarize import load_summarize_chain
from langchain.chains import LLMChain
from supabase import Client, create_client

from app.config import redis_client
from app.models.conversation import (ConversationCreate, ConversationProfile,
                                     ConversationUpdate, MessageCreate,
                                     MessageProfile)
from app.models.relationship import (UserCharacterInteractionCreate,
                                     UserCharacterInteractionInDB,
                                     UserCharacterInteractionUpdate)
from app.models.user import UserProfile as User
from app.services.ai_service import AIService
from app.services.relationship_service import RelationshipService


router = APIRouter()

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

supabase_url = os.getenv("SUPABASE_URL")
supabase_key = os.getenv("SUPABASE_KEY")
if not supabase_url or not supabase_key:
    raise ValueError("SUPABASE_URL and SUPABASE_KEY must be set in .env file")

supabase: Client = create_client(supabase_url, supabase_key)

class ConversationContextManager: #
    def __init__(self, window_size: int = 10): # window_size:
        self.memory = ConversationBufferWindowMemory(k=window_size) #
        self.relationship_info = {} #
        self.current_scenario = None #


    def add_message(self, role: str, content: str): # role:   , content:
        if role == 'human':
            self.memory.chat_memory.add_message(HumanMessage(content=content))
        elif role == 'ai':
            self.memory.chat_memory.add_message(AIMessage(content=content))

    def get_conversation_history(self) -> List[Dict[str, str]]:
        return [{"role": msg.type, "content": msg.content} for msg in self.memory.chat_memory.messages]

    def update_relationship_info(self, affinity: float, interaction_count: int):
        self.relationship_info.update({
            "affinity": affinity,
            "interaction_count": interaction_count
        })

    def set_current_scenario(self, scenario: Dict[str, any]):
        self.current_scenario = scenario

    def get_formatted_context(self) -> str:
        context = {
            "conversation_history": self.get_conversation_history(),
            "relationship_info": self.relationship_info,
            "current_scenario": self.current_scenario
        }
        return json.dumps(context, ensure_ascii=False, indent=2)

    def clear_context(self):
        self.memory.clear()
        self.relationship_info = {}
        self.current_scenario = None

class ConversationService:
    def __init__(self):
        self.context_manager = ConversationContextManager()
        self.llm = OpenAI(temperature=0)  # OpenAI
        self.summarize_chain = load_summarize_chain(self.llm, chain_type="map_reduce")
        self.ai_service = AIService()

        self.relationship_service = RelationshipService()
        self.llm = ChatOpenAI(temperature=0.7)
        self.prompt_template = PromptTemplate(
            input_variables=["context", "recent_messages", "summary", "similar_messages", "affinity", "nickname"],
            template="""
             AI .     :

             : {context}
             : {recent_messages}
             : {summary}
             : {similar_messages}
             ( ): {affinity}
              : {nickname}

                   .     .
            """
        )
        self.llm_chain = LLMChain(llm=self.llm, prompt=self.prompt_template)

        self.relationship_service = RelationshipService()
        self.llm = ChatOpenAI(temperature=0.7)
        self.affinity_prompt = PromptTemplate(
            input_variables=["summary"],
            template="""
              :
            {summary}

               , AI      .
              -5 5   .
            -5 AI       , 0  , 5     .

              :
            """
        )
        self.affinity_chain = LLMChain(llm=self.llm, prompt=self.affinity_prompt)

        self.redis_client = redis_client



    async def summarize_conversation(self, conversation_id: str, current_user: User) -> str:
        #
        messages = await self.list_messages(conversation_id, current_user)

        #  10
        recent_messages = messages[-10:]

        #  Document
        docs = [Document(page_content=msg.content) for msg in recent_messages]

        #
        summary = self.summarize_chain.run(docs)

        #
        await self.save_summary(conversation_id, summary)

        # AI
        affinity_change = await self.calculate_affinity_change(summary)

        #
        conversation = await self.get_conversation(conversation_id, current_user)
        await self.relationship_service.update_affinity(str(conversation.character_id), str(current_user.id), affinity_change)

        return summary

    async def save_summary(self, conversation_id: str, summary: str):
        """


        :param conversation_id:  ID
        :param summary:
        """
        try:
            # Supabase
            response = supabase.table("conversation_summaries").insert({
                "conversation_id": conversation_id,
                "summary": summary,
                "created_at": datetime.now(timezone.utc).isoformat()
            }).execute()

            if not response.data:
                raise HTTPException(status_code=400, detail="Failed to save summary")
        except Exception as e:
            logger.error(f"Error saving summary: {str(e)}")
            raise HTTPException(status_code=400, detail=str(e))

    async def create_message_and_respond(self, conversation_id: str, message: MessageCreate, current_user: User) -> MessageProfile:
        created_message = await self.create_message(conversation_id, message, current_user)

        if message.sender == "user":
            conversation = await self.get_conversation(conversation_id, current_user)
            ai_response = await self.generate_ai_response(conversation_id, str(current_user.id), str(conversation.character_id))

            ai_message = MessageCreate(sender="character", content=ai_response)
            await self.create_message(conversation_id, ai_message, current_user)

        return created_message

    async def create_conversation(self, conversation: ConversationCreate, current_user: User) -> ConversationProfile:
        try:
            conversation_data = conversation.model_dump()
            conversation_data['user_id'] = str(current_user.id)
            conversation_data['character_id'] = str(conversation.character_id)

            response = supabase.table("conversations").insert(conversation_data).execute()

            if response.data:
                self.context_manager.clear_context()
                return ConversationProfile(**response.data[0])
            else:
                raise HTTPException(status_code=400, detail="Failed to create conversation")
        except Exception as e:
            logger.error(f"Error creating conversation: {str(e)}")
            raise HTTPException(status_code=400, detail=str(e))


    async def get_conversation(self, conversation_id: str, current_user: User) -> ConversationProfile:
        # Redis
        cached_conversation = self.redis_client.get(f"conversation:{conversation_id}")
        if cached_conversation:
            conversation = json.loads(cached_conversation)
            if conversation['user_id'] == current_user.id:
                return ConversationProfile(**conversation)

        # Redis
        try:
            response = supabase.table("conversations").select("*").eq("id", conversation_id).execute()
            if response.data:
                conversation = response.data[0]
                if conversation['user_id'] == current_user.id:
                    # Redis
                    self.redis_client.setex(f"conversation:{conversation_id}", 3600, json.dumps(conversation))  # 1
                    return ConversationProfile(**conversation)
                else:
                    raise HTTPException(status_code=403, detail="You don't have permission to access this conversation")
            else:
                raise HTTPException(status_code=404, detail="Conversation not found")
        except Exception as e:
```

```python
            logger.error(f"Error getting conversation: {str(e)}")
            raise HTTPException(status_code=400, detail=str(e))

    async def update_conversation(self, conversation_id: str, conversation: ConversationUpdate, current_user: User) -> ConversationProfile:
        try:
            existing_conversation = await self.get_conversation(conversation_id, current_user)
            if existing_conversation.user_id != current_user.id:
                raise HTTPException(status_code=403, detail="You don't have permission to update this conversation")

            update_data = conversation.model_dump(exclude_unset=True)

            response = supabase.table("conversations").update(update_data).eq("id", conversation_id).execute()
            if response.data:
                return ConversationProfile(**response.data[0])
            else:
                raise HTTPException(status_code=400, detail="Failed to update conversation")
        except Exception as e:
            logger.error(f"Error updating conversation: {str(e)}")
            raise HTTPException(status_code=400, detail=str(e))

    async def delete_conversation(self, conversation_id: str, current_user: User):
        try:
            existing_conversation = await self.get_conversation(conversation_id, current_user)
            if existing_conversation.user_id != current_user.id:
                raise HTTPException(status_code=403, detail="You don't have permission to delete this conversation")

            response = supabase.table("conversations").delete().eq("id", conversation_id).execute()
            if not response.data:
                raise HTTPException(status_code=400, detail="Failed to delete conversation")
        except Exception as e:
            logger.error(f"Error deleting conversation: {str(e)}")
            raise HTTPException(status_code=400, detail=str(e))

    async def list_conversations(self, current_user: User) -> List[ConversationProfile]:
        try:
            response = supabase.table("conversations").select("*").eq("user_id", current_user.id).execute()

            if response.data:
                return [ConversationProfile(**conversation) for conversation in response.data]
            else:
                return []
        except Exception as e:
            logger.error(f"Error listing conversations: {str(e)}")
            raise HTTPException(status_code=400, detail=str(e))

    async def create_message(self, conversation_id: str, message: MessageCreate, current_user: User) -> MessageProfile:
        try:
            await self.get_conversation(conversation_id, current_user)

            message_data = message.model_dump()
            message_data['conversation_id'] = conversation_id

            self.context_manager.add_message("human" if message.sender == "user" else "ai", message.content)

            #
            message_count = await self.get_message_count(conversation_id)
            #   10
            if message_count % 10 == 0:
                await self.summarize_conversation(conversation_id, current_user)

            #
            vector = self.ai_service.vectorize_text(message.content)
            message_data['embedding'] = vector


            # Supabase   ( )
            response = supabase.table("messages").insert(message_data).execute()

            if response.data:
                created_message = MessageProfile(**response.data[0])

                # Redis
                self.redis_client.lpush(f"recent_messages:{conversation_id}", json.dumps(created_message.dict()))
                self.redis_client.ltrim(f"recent_messages:{conversation_id}", 0, 9)  #  10

                # Pinecone
                await self.ai_service.store_vector_async(
                    id=str(created_message.id),
                    vector=vector,
                    metadata={
                        "conversation_id": conversation_id,
                        "content": message.content,
                        "created_at": created_message.created_at.isoformat()
                    }
                )

                return created_message
            else:
                raise HTTPException(status_code=400, detail="Failed to create message")
        except Exception as e:
            logger.error(f"Error creating message: {str(e)}")
            raise HTTPException(status_code=400, detail=str(e))


    async def get_similar_messages(self, conversation_id: str, message_content: str, top_k: int = 5) -> List[MessageProfile]:
        """


        :param conversation_id:    ID
        :param message_content:
        :param top_k:
        :return:
        """
        vector = self.ai_service.vectorize_text(message_content)
        similar_vectors = self.ai_service.search_similar_vectors(vector, conversation_id, top_k)

        similar_messages = []
        for match in similar_vectors:
            message_id = match['id']
            message = await self.get_message(message_id, None)  #  None  .     .
            similar_messages.append(message)

        return similar_messages

    async def get_message_count(self, conversation_id: str) -> int:
        """


        :param conversation_id:  ID
        :return:
        """
        try:
            response = supabase.table("messages").select("id", count="exact").eq("conversation_id", conversation_id).execute()
            return response.count
        except Exception as e:
            logger.error(f"Error getting message count: {str(e)}")
            raise HTTPException(status_code=400, detail=str(e))
    async def list_messages(self, conversation_id: str, current_user: User) -> List[MessageProfile]:
        try:
            await self.get_conversation(conversation_id, current_user)

            response = supabase.table("messages").select("*").eq("conversation_id", conversation_id).order("created_at").execute()

            if response.data:
                return [MessageProfile(**message) for message in response.data]
            else:
                return []
        except Exception as e:
            logger.error(f"Error listing messages: {str(e)}")
            raise HTTPException(status_code=400, detail=str(e))

    async def get_message(self, message_id: str, current_user: User) -> MessageProfile:
        try:
            response = supabase.table("messages").select("*").eq("id", message_id).execute()
            if response.data:
                message = response.data[0]
                conversation = await self.get_conversation(message['conversation_id'], current_user)
                if conversation.user_id != current_user.id:
                    raise HTTPException(status_code=403, detail="You don't have permission to access this message")
                return MessageProfile(**message)
            else:
                raise HTTPException(status_code=404, detail="Message not found")
        except Exception as e:
            logger.error(f"Error getting message: {str(e)}")
            raise HTTPException(status_code=400, detail=str(e))

    async def update_relationship(self, user_id: str, character_id: str, affinity: float, interaction_count: int):
        try:
            self.context_manager.update_relationship_info(affinity, interaction_count)
            #
        except Exception as e:
            logger.error(f"Error updating relationship: {str(e)}")
            raise HTTPException(status_code=400, detail=str(e))

    async def set_scenario(self, scenario_id: str):
        try:
            scenario = await self.get_scenario_from_db(scenario_id)
            self.context_manager.set_current_scenario(scenario)
        except Exception as e:
            logger.error(f"Error setting scenario: {str(e)}")
            raise HTTPException(status_code=400, detail=str(e))

    async def get_scenario_from_db(self, scenario_id: str):
        #
        # :
        response = supabase.table("scenarios").select("*").eq("id", scenario_id).execute()
        if response.data:
            return response.data[0]
        else:
            raise HTTPException(status_code=404, detail="Scenario not found")

    async def get_conversation_summary(self, conversation_id: str) -> str:
        # Redis
        cached_summary = self.redis_client.get(f"conversation_summary:{conversation_id}")
        if cached_summary:
            return cached_summary

        # Redis
        try:
            response = supabase.table("conversation_summaries").select("summary").eq("conversation_id", conversation_id).order("created_at", ascending=False).lim
            if response.data:
                summary = response.data[0]['summary']
                # Redis
                self.redis_client.setex(f"conversation_summary:{conversation_id}", 3600, summary)  # 1
                return summary
            return "  ."
        except Exception as e:
            logger.error(f"Error getting conversation summary: {str(e)}")
            return "   ."

    async def get_recent_messages(self, conversation_id: str, limit: int) -> List[MessageProfile]:
        # Redis
        cached_messages = self.redis_client.lrange(f"recent_messages:{conversation_id}", 0, limit - 1)
        if cached_messages and len(cached_messages) == limit:
            return [MessageProfile(**json.loads(msg)) for msg in cached_messages]

        # Redis
        try:
            response = supabase.table("messages").select("*").eq("conversation_id", conversation_id).order("created_at", ascending=False).limit(limit).execute()
            messages = [MessageProfile(**msg) for msg in response.data][::-1]

            # Redis
            for msg in messages:
                self.redis_client.lpush(f"recent_messages:{conversation_id}", json.dumps(msg.model_dump()))
            self.redis_client.ltrim(f"recent_messages:{conversation_id}", 0, limit - 1)

            return messages
        except Exception as e:
            logger.error(f"Error getting recent messages: {str(e)}")
            return []


    async def get_relationship(self, character_id: str, user_id: str) -> UserCharacterInteractionInDB:
        try:
            return await self.relationship_service.get_interaction(character_id, user_id)
        except HTTPException:
            #
            new_relationship = UserCharacterInteractionCreate(character_id=character_id, user_id=user_id)
            return await self.relationship_service.create_interaction(new_relationship)

    async def generate_ai_response(self, conversation_id: str, user_id: str, character_id: str) -> str:
        try:
            recent_messages = await self.get_recent_messages(conversation_id, 10)
            summary = await self.get_conversation_summary(conversation_id)
            similar_messages = await self.get_similar_messages(conversation_id, recent_messages[-1].content, 3)
            relationship = await self.relationship_service.get_interaction(user_id, character_id)
            affinity_level = self.relationship_service.get_affinity_level(relationship.affinity)

            #
            self.context_manager.update_relationship_info(relationship.affinity, relationship.interaction_count)
            self.context_manager.add_message("human", recent_messages[-1].content)

            context = self.context_manager.get_formatted_context()

            #
            encoding = tiktoken.encoding_for_model("gpt-3.5-turbo")
            max_tokens = 4096  # GPT-3.5-turbo
            prompt_tokens = len(encoding.encode(context))
            available_tokens = max_tokens - prompt_tokens - 100  #

            prompt_template = PromptTemplate(
                input_variables=["recent_messages", "summary", "similar_messages", "affinity_level", "relationship_type", "nickname"],
                template="""
                 AI .      :

                : {context}
                 : {recent_messages}
                 : {summary}
                  : {similar_messages}
                  : {affinity_level}
                 : {relationship_type}
                 : {nickname}

                       .
                       .
                    ,    .
                """
            )

            llm = ChatOpenAI(temperature=0.7, max_tokens=available_tokens)
            llm_chain = LLMChain(llm=llm, prompt=prompt_template)

            ai_response = llm_chain.run(
                context=context,
                recent_messages=self.format_messages(recent_messages),
                summary=summary,
                similar_messages=self.format_messages(similar_messages),
                affinity_level=affinity_level,
                relationship_type=relationship.relationship_type.value,
                nickname=relationship.nickname or ""
            )

            # AI
            self.context_manager.add_message("ai", ai_response)

            return ai_response
        except Exception as e:
            logger.error(f"Error generating AI response: {str(e)}")
            return ".     .   ."


    def format_messages(self, messages: List[MessageProfile]) -> str:
        return "\n".join([f"{msg.sender}: {msg.content}" for msg in messages])


    async def update_affinity(self, conversation_id: str, user_id: str, character_id: str, message_content: str):
        #
        affinity_change = self.calculate_affinity_change(message_content)

        relationship = await self.relationship_service.get_relationship(user_id, character_id)
        new_affinity = max(-100, min(100, relationship.affinity + affinity_change))

        await self.relationship_service.update_relationship(
            user_id,
            character_id,
            UserCharacterInteractionUpdate(affinity=new_affinity, last_interaction=datetime.datetime.now(timezone.utc))
        )

    async def calculate_affinity_change(self, summary: str) -> float:
        affinity_change_str = await self.affinity_chain.arun(summary=summary)
        try:
            affinity_change = float(affinity_change_str.strip())
            return max(-5, min(5, affinity_change))  #  -5 5
        except ValueError:
            return 0  #

#   ConversationService
```

File: /Users/kh.kim/Documents/AIChat/app/services/__init__.py

```
[binary]
```

File: /Users/kh.kim/Documents/AIChat/app/services/scenario_service.py

```
# # services/scenario_service.py

# import asyncio
# from app.models.scenario import Scenario, ScenarioProgress, ScenarioCreate, ScenarioUpdate, ScenarioStep
# from app.services.relationship_service import RelationshipService
# from datetime import datetime
# from typing import List, Optional
# from app.config import supabase_client  # Supabase

# async def db_get_scenario(scenario_id: str) -> Optional[Scenario]:
#     """
#        .
#         .
#     """
#     result = await supabase_client.table('scenarios').select('*, scenario_steps(*)').eq('id', scenario_id).execute()
#     if result.data:
#         scenario_data = result.data[0]
#         steps = [ScenarioStep(**step) for step in scenario_data.pop('scenario_steps')]
#         return Scenario(**scenario_data, steps=steps)
#     return None

# async def db_create_scenario(scenario: ScenarioCreate) -> Scenario:
#     """
#        .
#            .
#     """
#     scenario_data = scenario.model_dump(exclude={'steps'})
#     result = await supabase_client.table('scenarios').insert(scenario_data).execute()
#     created_scenario = result.data[0]

#     steps_data = [{"scenario_id": created_scenario['id'], "step_order": i, **step.model_dump()}
#                   for i, step in enumerate(scenario.steps)]
#     await supabase_client.table('scenario_steps').insert(steps_data).execute()

#     return await db_get_scenario(created_scenario['id'])

# async def db_update_scenario(scenario_id: str, scenario_update: ScenarioUpdate) -> Scenario:
#     """
#       .
#          .
#     """
#     update_data = scenario_update.model_dump(exclude_unset=True)
#     steps = update_data.pop('steps', None)

#     result = await supabase_client.table('scenarios').update(update_data).eq('id', scenario_id).execute()

#     if steps is not None:
#         #      .
#         await supabase_client.table('scenario_steps').delete().eq('scenario_id', scenario_id).execute()
#         steps_data = [{"scenario_id": scenario_id, "step_order": i, **step.model_dump()}
#                       for i, step in enumerate(steps)]
#         await supabase_client.table('scenario_steps').insert(steps_data).execute()

#     return await db_get_scenario(scenario_id)

# async def db_delete_scenario(scenario_id: str) -> bool:
```

```
#     """
#        .
#          (CASCADE  ).
#     """
#     result = await supabase_client.table('scenarios').delete().eq('id', scenario_id).execute()
#     return len(result.data) > 0

# async def db_get_scenario_progress(user_id: str, scenario_id: str) -> Optional[ScenarioProgress]:
#     """
#           .
#     """
#     result = await supabase_client.table('scenario_progress').select('*').eq('user_id', user_id).eq('scenario_id', scenario_id).execute()
#     if result.data:
#         return ScenarioProgress(**result.data[0])
#     return None

# async def db_update_scenario_progress(progress: ScenarioProgress) -> ScenarioProgress:
#     """
#          .
#     """
#     progress_data = progress.model_dump()
#     result = await supabase_client.table('scenario_progress').upsert(progress_data).execute()
#     return ScenarioProgress(**result.data[0])

# async def create_scenario(scenario: ScenarioCreate) -> Scenario:
#     """   ."""
#     return await db_create_scenario(scenario)

# async def get_scenario(scenario_id: str) -> Optional[Scenario]:
#     """ ID  ."""
#     return await db_get_scenario(scenario_id)

# async def update_scenario(scenario_id: str, scenario_update: ScenarioUpdate) -> Scenario:
#     """  ."""
#     return await db_update_scenario(scenario_id, scenario_update)

# async def delete_scenario(scenario_id: str) -> bool:
#     """ ."""
#     return await db_delete_scenario(scenario_id)

# async def check_scenario_trigger(user_id: str, character_id: str) -> Optional[Scenario]:
#     """
#          ,
#          .
#     """
#     relationship_service = RelationshipService()
#     relationship = await relationship_service.get_interaction(user_id, character_id)


#     result = await supabase_client.table('scenarios').select('*').eq('character_id', character_id).execute()
#     scenarios = [Scenario(**scenario) for scenario in result.data]

#     #     .
#     async def check_scenario(scenario):
#         if scenario.trigger_type == "affinity" and relationship.affinity >= scenario.trigger_value:
#             return scenario
#         return None

#     checked_scenarios = await asyncio.gather(*[check_scenario(scenario) for scenario in scenarios])
#     triggered_scenarios = [s for s in checked_scenarios if s is not None]

#     return triggered_scenarios[0] if triggered_scenarios else None

# async def start_scenario(user_id: str, scenario_id: str) -> ScenarioProgress:
#     """
#         .
#         .
#     """
#     scenario = await get_scenario(scenario_id)
#     if not scenario:
#         raise ValueError("Scenario not found")

#     progress = ScenarioProgress(
#         id=f"{user_id}_{scenario_id}",
#         user_id=user_id,
#         scenario_id=scenario_id,
#         current_step=0,
#         started_at=datetime.now(),
#         is_completed=False
#     )

#     return await db_update_scenario_progress(progress)

# async def progress_scenario(user_id: str, scenario_id: str) -> ScenarioProgress:
#     """
#        .
#           .
#     """
#     progress = await db_get_scenario_progress(user_id, scenario_id)
#     if not progress:
#         raise ValueError("Scenario progress not found")

#     scenario = await get_scenario(scenario_id)
#     if not scenario:
#         raise ValueError("Scenario not found")

#     if progress.current_step < len(scenario.steps) - 1:
#         progress.current_step += 1
#     else:
#         progress.is_completed = True
#         progress.completed_at = datetime.now()

#     return await db_update_scenario_progress(progress)

# async def get_scenario_message(scenario_id: str, step: int) -> str:
#     """
#          .
#     """
#     scenario = await get_scenario(scenario_id)
#     if not scenario or step >= len(scenario.steps):
#         raise ValueError("Invalid scenario or step")

#     return scenario.steps[step].content

# #
# async def get_all_scenarios_for_character(character_id: str) -> List[Scenario]:
#     """
#         .
#     """
#     result = await supabase_client.table('scenarios').select('*, scenario_steps(*)').eq('character_id', character_id).execute()
#     scenarios = []
#     for scenario_data in result.data:
#         steps = [ScenarioStep(**step) for step in scenario_data.pop('scenario_steps')]
#         scenarios.append(Scenario(**scenario_data, steps=steps))
#     return scenarios

# async def bulk_update_scenario_progress(progresses: List[ScenarioProgress]) -> List[ScenarioProgress]:
#     """
#           .
#     """
#     progress_data = [progress.model_dump() for progress in progresses]
#     result = await supabase_client.table('scenario_progress').upsert(progress_data).execute()
#     return [ScenarioProgress(**data) for data in result.data]
```

File: /Users/kh.kim/Documents/AIChat/app/services/relationship_service.py

```python
import json
import os
from datetime import datetime, timezone

from fastapi import HTTPException
from supabase import Client, create_client

from app.config import redis_client
from app.models.relationship import (RelationshipType,
                                     UserCharacterInteractionCreate,
                                     UserCharacterInteractionInDB,
                                     UserCharacterInteractionUpdate)

supabase_url = os.getenv("SUPABASE_URL")
supabase_key = os.getenv("SUPABASE_KEY")
if not supabase_url or not supabase_key:
    raise ValueError("SUPABASE_URL and SUPABASE_KEY must be set in .env file")


supabase: Client = create_client(supabase_url, supabase_key)


class RelationshipService:
    def __init__(self):
        self.redis_client = redis_client

    async def get_interaction(self, character_id: str, user_id: str) -> UserCharacterInteractionInDB:
        # Redis
        cached_interaction = self.redis_client.get(f"interaction:{character_id}:{user_id}")
        if cached_interaction:
            return UserCharacterInteractionInDB(**json.loads(cached_interaction))

        # Redis
        response = supabase.table("user_character_interactions").select("*").eq("character_id", character_id).eq("user_id", user_id).execute()
        if response.data:
            interaction = UserCharacterInteractionInDB(**response.data[0])
            # Redis
            self.redis_client.setex(f"interaction:{character_id}:{user_id}", 3600, json.dumps(interaction.model_dump()))  # 1
            return interaction
        raise HTTPException(status_code=404, detail="Interaction not found")


    async def create_interaction(self, interaction: UserCharacterInteractionCreate) -> UserCharacterInteractionInDB:
        response = supabase.table("user_character_interactions").insert(interaction.model_dump()).execute()
        if response.data:
            created_interaction = UserCharacterInteractionInDB(**response.data[0])
            # Redis
            self.redis_client.setex(
                f"interaction:{created_interaction.character_id}:{created_interaction.user_id}",
                3600,
                json.dumps(created_interaction.model_dump())
            )
            return created_interaction
        raise HTTPException(status_code=400, detail="Failed to create interaction")

    async def update_interaction(self, character_id: str, user_id: str, interaction: UserCharacterInteractionUpdate) -> UserCharacterInteractionInDB:
        update_data = interaction.model_dump(exclude_unset=True)
        update_data['last_interaction'] = datetime.now(timezone.utc)
        response = supabase.table("user_character_interactions").update(update_data).eq("character_id", character_id).eq("user_id", user_id).execute()
        if response.data:
            updated_interaction = UserCharacterInteractionInDB(**response.data[0])
            # Redis
            self.redis_client.setex(f"interaction:{character_id}:{user_id}", 3600, json.dumps(updated_interaction.model_dump()))
            return updated_interaction
        raise HTTPException(status_code=400, detail="Failed to update interaction")


    async def update_affinity(self, character_id: str, user_id: str, affinity_change: float):
        interaction = await self.get_interaction(character_id, user_id)
        new_affinity = max(-100, min(100, interaction.affinity + affinity_change))
        new_relationship_type = self.get_relationship_type(new_affinity)
        updated_interaction = await self.update_interaction(
            character_id,
            user_id,
            UserCharacterInteractionUpdate(
                affinity=new_affinity,
                relationship_type=new_relationship_type,
                interaction_count=interaction.interaction_count + 1
            )
        )
        # Redis
        self.redis_client.setex(f"interaction:{character_id}:{user_id}", 3600, json.dumps(updated_interaction.model_dump()))

    def get_affinity_level(self, affinity: float) -> str:
        if affinity <= -91:
            return " "
        elif affinity <= -71:
            return " "
        elif affinity <= -41:
            return ""
        elif affinity <= -11:
            return " "
        elif affinity <= 10:
            return ""
        elif affinity <= 40:
            return ""
        elif affinity <= 60:
            return " "
        elif affinity <= 70:
            return " "
        elif affinity <= 90:
            return "  "
        else:
            return " "

    def get_relationship_type(self, affinity: float) -> RelationshipType:
        if affinity <= -91:
            return RelationshipType.ENEMY
        elif affinity <= -51:
            return RelationshipType.RIVAL
        elif affinity <= 10:
            return RelationshipType.STRANGER
        elif affinity <= 20:
            return RelationshipType.ACQUAINTANCE
        elif affinity <= 30:
            return RelationshipType.FRIEND
        elif affinity <= 50:
            return RelationshipType.CLOSE_FRIEND
        elif affinity <= 70:
            return RelationshipType.LOVER
        elif affinity <= 100:
            return RelationshipType.SPOUSE
        else:
            raise ValueError("Affinity value out of bounds")

    async def update_custom_traits(self, character_id: str, user_id: str, custom_traits: dict):
        updated_interaction = await self.update_interaction(
            character_id,
            user_id,
            UserCharacterInteractionUpdate(custom_traits=custom_traits)
        )
        # Redis
        self.redis_client.setex(f"interaction:{character_id}:{user_id}", 3600, json.dumps(updated_interaction.model_dump()))

    async def update_conversation_history(self, character_id: str, user_id: str, conversation_history: dict):
        updated_interaction = await self.update_interaction(
            character_id,
            user_id,
            UserCharacterInteractionUpdate(conversation_history=conversation_history)
        )
        # Redis
        self.redis_client.setex(f"interaction:{character_id}:{user_id}", 3600, json.dumps(updated_interaction.model_dump()))
```

File: /Users/kh.kim/Documents/AIChat/app/services/character_service.py

```python
# services/character_service.py

import logging
import os
from typing import List

from fastapi import APIRouter, HTTPException
from supabase import Client, create_client

from app.models.character import (CharacterCreate, CharacterProfile,
                                  CharacterUpdate)
from app.models.user import UserProfile as User

router = APIRouter()


logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

supabase_url = os.getenv("SUPABASE_URL")
supabase_key = os.getenv("SUPABASE_KEY")
if not supabase_url or not supabase_key:
    raise ValueError("SUPABASE_URL and SUPABASE_KEY must be set in .env file")


supabase: Client = create_client(supabase_url, supabase_key)
def is_admin(user: User) -> bool:
    return user.is_admin




async def create_character(character: CharacterCreate, current_user: User) -> CharacterProfile:
    try:
        character_data = character.model_dump()
        character_data['creator_id'] = current_user.id

        # Convert LocalizedContent fields to JSON
        for field in ['names', 'occupation', 'background', 'appearance_description']:
            if field in character_data:
                character_data[field] = character_data[field].dict()

        response = supabase.table("characters").insert(character_data).execute()
        if response.data:
            return CharacterProfile(**response.data[0])
        else:
            raise HTTPException(status_code=400, detail="Failed to create character")
    except Exception as e:
        logger.error(f"Error creating character: {str(e)}")
        raise HTTPException(status_code=400, detail=str(e))

async def get_character(character_id: str, current_user: User) -> CharacterProfile:
    try:
        response = supabase.table("characters").select("*").eq("id", character_id).execute()
        if response.data:
            character = response.data[0]
            if character['creator_id'] == current_user.id or is_admin(current_user):
                return CharacterProfile(**character)
            else:
                raise HTTPException(status_code=403, detail="You don't have permission to access this character")
        else:
            raise HTTPException(status_code=404, detail="Character not found")
    except Exception as e:
        logger.error(f"Error getting character: {str(e)}")
        raise HTTPException(status_code=400, detail=str(e))

async def update_character(character_id: str, character: CharacterUpdate, current_user: User) -> CharacterProfile:
    try:
        existing_character = await get_character(character_id, current_user)
        if existing_character.creator_id != current_user.id and not is_admin(current_user):
            raise HTTPException(status_code=403, detail="You don't have permission to update this character")

        update_data = character.model_dump(exclude_unset=True)

        # Convert LocalizedContent fields to JSON
        for field in ['names', 'occupation', 'background', 'appearance_description']:
            if field in update_data:
                update_data[field] = update_data[field].dict()

        response = supabase.table("characters").update(update_data).eq("id", character_id).execute()
        if response.data:
            return CharacterProfile(**response.data[0])
        else:
            raise HTTPException(status_code=400, detail="Failed to update character")
    except Exception as e:
        logger.error(f"Error updating character: {str(e)}")
        raise HTTPException(status_code=400, detail=str(e))

async def delete_character(character_id: str, current_user: User):
    try:
        existing_character = await get_character(character_id, current_user)
        if existing_character.creator_id != current_user.id and not is_admin(current_user):
            raise HTTPException(status_code=403, detail="You don't have permission to delete this character")

        response = supabase.table("characters").delete().eq("id", character_id).execute()
        if not response.data:
            raise HTTPException(status_code=400, detail="Failed to delete character")
    except Exception as e:
        logger.error(f"Error deleting character: {str(e)}")
        raise HTTPException(status_code=400, detail=str(e))

async def list_characters(current_user: User) -> List[CharacterProfile]:
    try:
        if is_admin(current_user):
            response = supabase.table("characters").select("*").execute()
        else:
            response = supabase.table("characters").select("*").or_(f"creator_id.eq.{current_user.id},is_public.eq.true").execute()

        if response.data:
            return [CharacterProfile(**character) for character in response.data]
        else:
            return []
    except Exception as e:
        logger.error(f"Error listing characters: {str(e)}")
        raise HTTPException(status_code=400, detail=str(e))
```

File: /Users/kh.kim/Documents/AIChat/requirements.txt

```
aiohappyeyeballs==2.3.4
aiohttp==3.10.1
aiosignal==1.3.1
annotated-types==0.7.0
anyio==4.4.0
async-timeout==4.0.3
attrs==24.1.0
```

```
certifi==2024.7.4
charset-normalizer==3.3.2
click==8.1.7
deprecation==2.1.0
distro==1.9.0
dnspython==2.6.1
email_validator==2.2.0
exceptiongroup==1.2.2
fastapi==0.111.1
fastapi-cli==0.0.5
frozenlist==1.4.1
gotrue==2.6.1
fastapi==0.111.1
gunicorn==22.0.0
h11==0.14.0
h2==4.1.0
hpack==4.0.0
httpcore==1.0.5
httptools==0.6.1
httpx==0.27.0
hyperframe==6.0.1
idna==3.7
Jinja2==3.1.4
jsonpatch==1.33
jsonpointer==3.0.0
langchain==0.2.12
langchain-core==0.2.28
langchain-openai==0.1.20
langchain-text-splitters==0.2.2
langsmith==0.1.98
markdown-it-py==3.0.0
MarkupSafe==2.1.5
mdurl==0.1.2
multidict==6.0.5
numpy==1.26.4
openai==1.37.1
orjson==3.10.6
packaging==23.2
pinecone==4.0.0
postgrest==0.16.9
pydantic==2.8.2
pydantic_core==2.20.1
Pygments==2.18.0
python-dateutil==2.9.0.post0
python-dotenv==1.0.1
python-multipart==0.0.9
PyYAML==6.0.1
realtime==1.0.6
redis==5.0.7
regex==2024.7.24
requests==2.32.3
rich==13.7.1
shellingham==1.5.4
six==1.16.0
sniffio==1.3.1
SQLAlchemy==2.0.32
starlette==0.37.2
storage3==0.7.7
StrEnum==0.4.15
supabase==2.6.0
supafunc==0.5.1
tenacity==8.5.0
tiktoken==0.7.0
tqdm==4.66.5
typer==0.12.3
typing_extensions==4.12.2
ujson==5.10.0
urllib3==2.2.2
uvicorn==0.27.1
uvloop==0.19.0
watchfiles==0.22.0
websockets==12.0
yarl==1.9.4
```

File: /Users/kh.kim/Documents/AIChat/Dockerfile

```
# Dockerfile
FROM python:3.9-slim

# 작업 디렉토리 설정
WORKDIR /app

# 종속성 파일 복사 및 설치
COPY requirements.txt requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# 애플리케이션 코드 복사
COPY . .

# 포트 설정 (Heroku는 기본적으로 $PORT 환경 변수를 사용)
ENV PORT=8000

# 애플리케이션 시작 명령어
CMD uvicorn app.main:app --host 0.0.0.0 --port $PORT
```

File: /Users/kh.kim/Documents/AIChat/test.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Auth Service Test</title>
</head>
<body>
    <h1>Auth Service Test Page</h1>

    <!-- Registration Form -->
    <h2>Register</h2>
    <form id="register-form">
        <input type="text" id="register-email" placeholder="Email" required>
        <input type="password" id="register-password" placeholder="Password" required>
        <input type="text" id="register-nickname" placeholder="Nickname" required>
        <button type="submit">Register</button>
    </form>

    <!-- Login Form -->
    <h2>Login</h2>
    <form id="login-form">
        <input type="text" id="login-email" placeholder="Email" required>
        <input type="password" id="login-password" placeholder="Password" required>
        <button type="submit">Login</button>
    </form>

    <!-- Social Login -->
    <h2>Social Login</h2>
    <button id="google-login">Login with Google</button>
    <button id="github-login">Login with GitHub</button>
    <button id="facebook-login">Login with Facebook</button>

    <!-- Logout Button -->
    <h2>Logout</h2>
    <button id="logout-button">Logout</button>

    <!-- Profile Form -->
    <h2>Profile</h2>
    <form id="profile-form">
        <input type="text" id="profile-username" placeholder="Username" required>
        <input type="text" id="profile-email" placeholder="Email">
        <button type="submit">Update Profile</button>
    </form>
    <button id="get-profile-button">Get Profile</button>

    <!-- Output -->
    <h2>Output</h2>
    <pre id="output"></pre>

    <script>
        const apiUrl = 'http://localhost:8000'; // Update with your API base URL
        let token = localStorage.getItem('token') || '';

        document.getElementById('register-form').addEventListener('submit', async (event) => {
            event.preventDefault();
            const email = document.getElementById('register-email').value;
            const password = document.getElementById('register-password').value;
            const nickname = document.getElementById('register-nickname').value;
            const response = await fetch(`${apiUrl}/register`, {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify({ email, password, nickname })
            });
            const data = await response.json();
            document.getElementById('output').innerText = JSON.stringify(data, null, 2);
        });

        document.getElementById('login-form').addEventListener('submit', async (event) => {
            event.preventDefault();
            const email = document.getElementById('login-email').value;
            const password = document.getElementById('login-password').value;
            const response = await fetch(`${apiUrl}/login`, {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify({ email, password })
            });
            const data = await response.json();
            if (response.ok) {
                token = data.access_token;
                localStorage.setItem('token', token);
            }
            document.getElementById('output').innerText = JSON.stringify(data, null, 2);
        });

        document.getElementById('logout-button').addEventListener('click', async () => {
            const response = await fetch(`${apiUrl}/logout`, {
                method: 'POST',
                headers: {
                    'Authorization': `Bearer ${token}`
                }
            });
            const data = await response.json();
            token = '';
            localStorage.removeItem('token');
            document.getElementById('output').innerText = JSON.stringify(data, null, 2);
        });

        document.getElementById('profile-form').addEventListener('submit', async (event) => {
            event.preventDefault();
            const username = document.getElementById('profile-username').value;
            const email = document.getElementById('profile-email').value;
            const response = await fetch(`${apiUrl}/profile`, {
                method: 'PUT',
                headers: {
                    'Content-Type': 'application/json',
                    'Authorization': `Bearer ${token}`
                },
                body: JSON.stringify({ username, email })
            });
            const data = await response.json();
            document.getElementById('output').innerText = JSON.stringify(data, null, 2);
        });

        document.getElementById('get-profile-button').addEventListener('click', async () => {
            console.log("Sending GET request to /profile", `Bearer ${token}`);
            const response = await fetch(`${apiUrl}/profile`, {
                method: 'GET',
                headers: {
                    'Authorization': `Bearer ${token}`
                }
            });
            const data = await response.json();
            document.getElementById('output').innerText = JSON.stringify(data, null, 2);
        });

        document.getElementById('google-login').addEventListener('click', async () => {
            localStorage.setItem('original_url', window.location.href);
            const response = await fetch(`${apiUrl}/social-login/google`, {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                }
            });
            const data = await response.json();
            if (data.url) {
                window.location.href = data.url;
            } else {
                document.getElementById('output').innerText = JSON.stringify(data, null, 2);
            }
        });

        document.getElementById('github-login').addEventListener('click', async () => {
            localStorage.setItem('original_url', window.location.href);
            const response = await fetch(`${apiUrl}/social-login/github`, {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                }
            });
            const data = await response.json();
            if (data.url) {
                window.location.href = data.url;
            } else {
                document.getElementById('output').innerText = JSON.stringify(data, null, 2);
            }
        });

        document.getElementById('facebook-login').addEventListener('click', async () => {
            localStorage.setItem('original_url', window.location.href);
            const response = await fetch(`${apiUrl}/social-login/facebook`, {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                }
            });
            const data = await response.json();
            if (data.url) {
                window.location.href = data.url;
            } else {
                document.getElementById('output').innerText = JSON.stringify(data, null, 2);
            }
        });
    </script>
</body>
</html>
```

File: /Users/kh.kim/Documents/AIChat/requirements-dev.txt

```
-r requirements.txt
awsebcli==3.20.10
eb==0.1.5
h2==4.1.0
importlib-resources==6.4.0
jaraco.text==3.12.1
keyring==24.3.1
ordered-set==4.1.0
pip-chill==1.0.3
pipreqs==0.5.0
rapidfuzz==3.9.4
tomli==2.0.1
```

File: /Users/kh.kim/Documents/AIChat/.env

```
SUPABASE_URL=https://ufcesffieoelerxmgekv.supabase.co
SUPABASE_KEY=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzdXBhYmFzZSIsInJlZiI6InVmY2VzZmZpZW9lbGVyeG1nZWt2Iiwicm9sZSI6ImFub24iLCJpYXQiOjE3MjAxODA2M
Mk-uTDUmgW4KEVSU
TESTING=True pytest
PINECONE_API_KEY=c0ebd048-48f1-47b5-ae28-dcb89ef1f8b5
PINECONE_ENVIRONMENT=us-east-1
PINECONE_INDEX_NAME=ai-dating-simulator
OPENAI_API_KEY=sk-proj-lM3W5QMEkH9vHMWfaa1ET3BlbkFJU4BUmhw53qQRFjrdkJ2x
UPSTASH_REDIS_REST_TOKEN=AaFXAQIncDFhYTM5OTkzMjg3YzM0ZjYyYTg0ZTg3MzQ4ZThmZDEyOXAxNDEzMDM
UPSTASH_REDIS_REST_URL=https://cool-toad-41303.upstash.io
UPSTASH_REDIS_URL=rediss://:AaFXAQIncDFhYTM5OTkzMjg3YzM0ZjYyYTg0ZTg3MzQ4ZThmZDEyOXAxNDEzMDM@cool-toad-41303.upstash.io:6379
```

File: /Users/kh.kim/Documents/AIChat/.code2pdf

```
:directories:
- .vscode
- static
- tests
- venv
- .venv
- .git
:files:
- .env.example
- .gitignore
- README.md
- test_redis_connection.py
- test.html.code2pdf.yaml
```

File: /Users/kh.kim/Documents/AIChat/Procfile

```
web: uvicorn app.main:app --host 0.0.0.0 --port $PORT
```