

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

[binary]

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

```
1 import os
2 from redis import Redis
3 from urllib.parse import urlparse
4 import logging
5
6 #
7 logging.basicConfig(level=logging.DEBUG)
8 logger = logging.getLogger(__name__)
9
10 redis_url = os.getenv('UPSTASH_REDIS_URL') # UPSTASH_REDIS_URL
11
12 if redis_url:
13     url = urlparse(redis_url)
14     logger.debug(f"Parsed Redis URL: {url}")
15     redis_client = Redis(
16         host=url.hostname,
17         port=url.port,
18         password=url.password,
19         ssl_url_scheme='rediss',
20         ssl_cert_reqs=None,
21         decode_responses=True
22     )
23 else:
24     redis_client = None
25     logger.warning("No Redis URL provided")
26
27 def test_redis_connection():
28     if redis_client is None:
29         print("No Redis URL provided")
30         return False
31     try:
32         redis_client.ping()
33         print("Successfully connected to Redis")
34         logger.info("Successfully connected to Redis")
35         return True
36     except Exception as e:
37         print(f"Failed to connect to Redis: {e}")
38         logger.error(f"Failed to connect to Redis: {e}")
39         return False
```

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

```
1 from .main import app
2
3 def create_app():
4     return app
```

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

1 [binary]

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

1 [binary]

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

```
1 from typing import Literal, Optional
2
3 from pydantic import BaseModel, EmailStr, Field
4
5
6 class UserBase(BaseModel):
7     email: EmailStr
8     nickname: str = Field(..., min_length=2, max_length=50)
9     is_admin: bool = False
10
11 class UserCreate(UserBase):
12     password: str = Field(..., min_length=8)
13
14 class UserUpdate(BaseModel):
15     nickname: Optional[str] = Field(None, min_length=2, max_length=50)
16     profile_image_url: Optional[str] = None
17     is_admin: Optional[bool] = None
18
19 class SocialLoginData(BaseModel):
20     provider: Literal["google", "kakao"]
21
22 class UserInDB(UserBase):
23     id: str
24     hashed_password: str
25     login_type: str = "email"
26     profile_image_url: Optional[str] = None
27
28 class UserProfile(UserBase):
29     id: str
30     profile_image_url: Optional[str]
31     login_type: str
32
33     class Config:
34         from_attributes = True
35
36 class Token(BaseModel):
37     access_token: str
38     token_type: str
39
40 class TokenData(BaseModel):
41     email: Optional[str] = None
```

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

```

1 from datetime import datetime, timezone
2 from enum import Enum
3 from typing import Optional
4
5 from pydantic import BaseModel
6
7
8 class RelationshipType(Enum):
9     ENEMY = "enemy"
10    RIVAL = "rival"
11    STRANGER = "stranger"
12    ACQUAINTANCE = "acquaintance"
13    FRIEND = "friend"
14    CLOSE_FRIEND = "close_friend"
15    LOVER = "lover"
16    SPOUSE = "spouse"
17
18 class UserCharacterInteractionBase(BaseModel):
19     user_id: str
20     character_id: str
21     affinity: float = 0
22     relationship_type: RelationshipType = RelationshipType.STRANGER
23     nickname: Optional[str] = None
24     last_interaction: datetime = datetime.now(timezone.utc)
25     interaction_count: int = 0
26     conversation_memory: int = 0
27     learning_rate: float = 0.0
28     custom_traits: dict = {}
29     conversation_history: dict = {}
30
31 class UserCharacterInteractionCreate(UserCharacterInteractionBase):
32     pass
33
34 class UserCharacterInteractionUpdate(BaseModel):
35     affinity: Optional[float] = None
36     relationship_type: Optional[RelationshipType] = None
37     nickname: Optional[str] = None
38     last_interaction: Optional[datetime] = None
39     interaction_count: Optional[int] = None
40     conversation_memory: Optional[int] = None
41     learning_rate: Optional[float] = None
42     custom_traits: Optional[dict] = None
43     conversation_history: Optional[dict] = None
44
45 class UserCharacterInteractionInDB(UserCharacterInteractionBase):
46     id: str
47
48     class Config:
49         orm_mode = True
50

```

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

```

1 import uuid
2 from datetime import datetime
3 from typing import Dict, List, Literal, Optional
4
5 from pydantic import BaseModel, Field
6
7
8 class ConversationBase(BaseModel):
9     # user_id: uuid.UUID
10    # character_id: uuid.UUID
11    user_id: str = Field(..., description="User ID as string")
12    character_id: str = Field(..., description="Character ID as string")
13    context: Optional[Dict] = Field(default_factory=dict)
14    state: Optional[Dict] = Field(default_factory=dict)
15
16 class ConversationCreate(ConversationBase):
17     pass
18
19 class ConversationUpdate(BaseModel):
20     context: Optional[Dict] = None
21     state: Optional[Dict] = None
22
23 class ConversationInDB(ConversationBase):
24     id: uuid.UUID
25     created_at: datetime
26     updated_at: datetime
27
28     class Config:
29         from_attributes = True
30
31 class ConversationProfile(ConversationInDB):
32     pass
33
34 class MessageBase(BaseModel):
35     conversation_id: uuid.UUID
36     sender_type: Literal['user', 'character']
37     content: List[Dict]
38     metadata: Optional[Dict] = Field(default_factory=dict)
39
40 class MessageCreate(MessageBase):
41     pass
42
43 class MessageUpdate(BaseModel):
44     content: Optional[List[Dict]] = None
45     metadata: Optional[Dict] = None
46
47 class MessageInDB(MessageBase):
48     id: uuid.UUID
49     created_at: datetime
50     embedding: Optional[List[float]] = None
51
52     class Config:
53         from_attributes = True
54
55 class MessageProfile(MessageBase):
56     id: uuid.UUID
57     created_at: datetime
58
59     class Config:
60         from_attributes = True
61

```

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

```

1 [binary]

```

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

1 [binary]

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

```
1 # # models/scenario.py
2
3 # from pydantic import BaseModel
4 # from typing import List, Optional
5 # from enum import Enum
6 # from datetime import datetime
7
8 # class ScenarioTriggerType(Enum):
9 #     """ """
10 #     AFFINITY = "affinity" #
11 #     TIME = "time" #
12 #     EVENT = "event" #
13
14 # class ScenarioStep(BaseModel):
15 #     """ """
16 #     step_id: str
17 #     content: str
18 #     image_url: Optional[str] = None # URL ()
19
20 # class Scenario(BaseModel):
21 #     """ """
22 #     id: str
23 #     character_id: str
24 #     title: str
25 #     description: str
26 #     trigger_type: ScenarioTriggerType
27 #     trigger_value: float #: 60 7
28 #     steps: List[ScenarioStep]
29 #     created_at: datetime
30 #     updated_at: datetime
31
32 # class ScenarioProgress(BaseModel):
33 #     """ """
34 #     id: str
35 #     user_id: str
36 #     scenario_id: str
37 #     current_step: int
38 #     started_at: datetime
39 #     completed_at: Optional[datetime] = None
40 #     is_completed: bool = False
41
42 # class ScenarioCreate(BaseModel):
43 #     """ """
44 #     character_id: str
45 #     title: str
46 #     description: str
47 #     trigger_type: ScenarioTriggerType
48 #     trigger_value: float
49 #     steps: List[ScenarioStep]
50
51 # class ScenarioUpdate(BaseModel):
52 #     """ """
53 #     title: Optional[str] = None
54 #     description: Optional[str] = None
55 #     trigger_type: Optional[ScenarioTriggerType] = None
56 #     trigger_value: Optional[float] = None
57 #     steps: Optional[List[ScenarioStep]] = None
```

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

```

1  import uuid
2  from datetime import datetime
3  from typing import Dict, List, Optional
4
5  from pydantic import BaseModel, Field
6
7
8  class LocalizedContent(BaseModel):
9      ko: Optional[str] = Field(None, description="Korean content")
10     en: Optional[str] = Field(None, description="English content")
11     ja: Optional[str] = Field(None, description="Japanese content")
12
13     class Config:
14         extra = 'forbid' # This prevents additional fields
15
16
17 class LanguageProficiency(BaseModel):
18     language_code: str
19     proficiency: str
20     preference_order: int = Field(ge=1)
21
22 class PersonalityTrait(BaseModel):
23     trait: str
24     score: float = Field(ge=0.0, le=1.0)
25
26 class Interest(BaseModel):
27     topic: str
28     level: str
29
30 class CharacterBase(BaseModel):
31     version: str
32     names: LocalizedContent
33     gender: str
34     age: int = Field(ge=0, le=150)
35     personality_traits: List[PersonalityTrait]
36     interests: List[Interest]
37     occupation: LocalizedContent
38     background: LocalizedContent
39     appearance_seed: str
40     appearance_description: LocalizedContent
41     relationship_status: Optional[str] = None
42     languages: List[LanguageProficiency]
43     conversation_style: LocalizedContent
44     communication_preferences: Dict[str, str]
45     backstory: LocalizedContent
46     goals: LocalizedContent
47     quirks: LocalizedContent
48     emotional_intelligence: float = Field(ge=0.0, le=1.0)
49     cultural_sensitivity: float = Field(ge=0.0, le=1.0)
50     relationship_progression_pace: str
51     conflict_resolution_style: str
52     interaction_prompts: Dict[str, LocalizedContent]
53     character_prompt: str
54     response_generation_parameters: Dict[str, float]
55     is_public: bool = False
56
57 class CharacterCreate(CharacterBase):
58     pass
59
60 class CharacterUpdate(BaseModel):
61     version: Optional[str] = None
62     names: Optional[LocalizedContent] = None
63     gender: Optional[str] = None
64     age: Optional[int] = Field(None, ge=0, le=150)
65     personality_traits: Optional[List[PersonalityTrait]] = None
66     interests: Optional[List[Interest]] = None
67     occupation: Optional[LocalizedContent] = None
68     background: Optional[LocalizedContent] = None
69     appearance_seed: Optional[str] = None
70     appearance_description: Optional[LocalizedContent] = None
71     relationship_status: Optional[str] = None
72     languages: Optional[List[LanguageProficiency]] = None
73     conversation_style: Optional[LocalizedContent] = None
74     communication_preferences: Optional[Dict[str, str]] = None
75     backstory: Optional[LocalizedContent] = None
76     goals: Optional[LocalizedContent] = None
77     quirks: Optional[LocalizedContent] = None
78     emotional_intelligence: Optional[float] = Field(None, ge=0.0, le=1.0)
79     cultural_sensitivity: Optional[float] = Field(None, ge=0.0, le=1.0)
80     relationship_progression_pace: Optional[str] = None
81     conflict_resolution_style: Optional[str] = None
82     interaction_prompts: Optional[Dict[str, LocalizedContent]] = None
83     character_prompt: Optional[str] = None
84     response_generation_parameters: Optional[Dict[str, float]] = None
85     is_public: Optional[bool] = None
86
87 class CharacterInDB(CharacterBase):
88     id: uuid.UUID
89     creator_id: str
90     created_at: datetime
91     updated_at: datetime
92
93     class Config:
94         from_attributes = True
95
96 class CharacterProfile(CharacterBase):
97     id: uuid.UUID
98     creator_id: str
99
100     class Config:
101         from_attributes = True

```

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

```

1 import logging
2 import os
3 import sys
4 from contextlib import asynccontextmanager
5
6 from fastapi import FastAPI, Request
7 from fastapi.middleware.cors import CORSMiddleware
8 from supabase import Client, create_client
9
10 import redis
11
12 from app.config import redis_client
13 from app.routes.characters import router as characters_router
14 from app.routes.conversations import router as conversations_router
15 from app.routes.users import router as users_router
16 # from app.routes.scenarios import router as scenarios_router
17 from app.services.auth_service import router as auth_router
18
19 sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
20
21 logging.basicConfig(level=logging.INFO)
22 logger = logging.getLogger(__name__)
23
24 @asynccontextmanager
25 async def lifespan(app: FastAPI):
26     logger.info("Application is starting up")
27     from app.config import test_redis_connection, redis_client
28
29     if test_redis_connection():
30         logger.info("Successfully connected to Redis")
31     else:
32         logger.warning("Continuing without Redis connection")
33
34     yield
35
36     if redis_client:
37         redis_client.close()
38         logger.info("Redis connection closed")
39     logger.info("Application is shutting down")
40
41 app = FastAPI(debug=True, lifespan=lifespan)
42
43 app.add_middleware(
44     CORSMiddleware,
45     allow_origins=["*"], #
46     allow_credentials=True,
47     allow_methods=["*"],
48     allow_headers=["*"],
49 )
50
51 supabase_url = os.getenv("SUPABASE_URL")
52 supabase_key = os.getenv("SUPABASE_KEY")
53 if not supabase_url or not supabase_key:
54     raise ValueError("SUPABASE_URL and SUPABASE_KEY must be set in environment variables")
55
56 supabase: Client = create_client(supabase_url, supabase_key)
57 app.state.supabase = supabase
58
59 @app.middleware("http")
60 async def log_requests(request: Request, call_next):
61     logger.info(f"Request path: {request.url.path}")
62     response = await call_next(request)
63     logger.info(f"Response status code: {response.status_code}")
64     return response
65
66 app.include_router(users_router)
67 app.include_router(characters_router)
68 app.include_router(conversations_router)
69 # app.include_router(scenarios_router)
70 app.include_router(auth_router, prefix="/auth")
71
72 @app.get("/")
73 async def root():
74     return {"message": "Hello World"}
75
76 if __name__ == "__main__":
77     import uvicorn
78     port = int(os.environ.get("PORT", 8000))
79     uvicorn.run("app.main:app", host="0.0.0.0", port=port, reload=True)
80
81

```

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

```

1 # # routes/scenarios.py
2
3 # from fastapi import APIRouter, HTTPException, Depends
4 # from typing import List
5 # from app.models.scenario import ScenarioCreate, ScenarioUpdate, Scenario, ScenarioProgress
6 # from app.services import scenario_service
7 # from app.models.user import UserProfile as User
8 # from app.dependencies import get_current_user
9
10 # router = APIRouter()
11
12 # @router.post("/scenarios", response_model=Scenario)
13 # async def create_scenario(scenario: ScenarioCreate, current_user: User = Depends(get_current_user)):
14 #     """
15 #     .
16 #     """
17 #     return await scenario_service.create_scenario(scenario)
18
19 # @router.get("/scenarios/{scenario_id}", response_model=Scenario)
20 # async def get_scenario(scenario_id: str, current_user: User = Depends(get_current_user)):
21 #     """
22 #     .
23 #     """
24 #     scenario = await scenario_service.get_scenario(scenario_id)
25 #     if not scenario:
26 #         raise HTTPException(status_code=404, detail="Scenario not found")
27 #     return scenario
28
29 # @router.put("/scenarios/{scenario_id}", response_model=Scenario)
30 # async def update_scenario(scenario_id: str, scenario_update: ScenarioUpdate, current_user: User = Depends(get_current_user)):
31 #     """
32 #     .
33 #     """
34 #     updated_scenario = await scenario_service.update_scenario(scenario_id, scenario_update)
35 #     if not updated_scenario:
36 #         raise HTTPException(status_code=404, detail="Scenario not found")
37 #     return updated_scenario
38
39 # @router.delete("/scenarios/{scenario_id}", response_model=bool)
40 # async def delete_scenario(scenario_id: str, current_user: User = Depends(get_current_user)):
41 #     """
42 #     .
43 #     """
44 #     deleted = await scenario_service.delete_scenario(scenario_id)
45 #     if not deleted:
46 #         raise HTTPException(status_code=404, detail="Scenario not found")
47 #     return True
48
49 # @router.post("/scenarios/{scenario_id}/start", response_model=ScenarioProgress)
50 # async def start_scenario(scenario_id: str, current_user: User = Depends(get_current_user)):
51 #     """
52 #     .
53 #     """
54 #     try:
55 #         progress = await scenario_service.start_scenario(str(current_user.id), scenario_id)
56 #         return progress
57 #     except ValueError as e:
58 #         raise HTTPException(status_code=400, detail=str(e))
59
60 # @router.post("/scenarios/{scenario_id}/progress", response_model=ScenarioProgress)
61 # async def progress_scenario(scenario_id: str, current_user: User = Depends(get_current_user)):
62 #     """
63 #     .
64 #     """
65 #     try:
66 #         progress = await scenario_service.progress_scenario(str(current_user.id), scenario_id)
67 #         return progress
68 #     except ValueError as e:
69 #         raise HTTPException(status_code=400, detail=str(e))
70
71 # @router.get("/scenarios/check_trigger", response_model=Scenario)
72 # async def check_scenario_trigger(character_id: str, current_user: User = Depends(get_current_user)):
73 #     """
74 #     .
75 #     """
76 #     scenario = await scenario_service.check_scenario_trigger(str(current_user.id), character_id)
77 #     if not scenario:
78 #         raise HTTPException(status_code=404, detail="No scenario triggered")
79 #     return scenario
80
81 # @router.get("/scenarios", response_model=List[Scenario])
82 # async def list_scenarios(character_id: str, current_user: User = Depends(get_current_user)):
83 #     """
84 #     .
85 #     """
86 #     scenarios = await scenario_service.get_all_scenarios_for_character(character_id)
87 #     return scenarios
88
89 # @router.get("/scenarios/{scenario_id}/progress", response_model=ScenarioProgress)
90 # async def get_scenario_progress(scenario_id: str, current_user: User = Depends(get_current_user)):
91 #     """
92 #     .
93 #     """
94 #     progress = await scenario_service.get_scenario_progress(str(current_user.id), scenario_id)
95 #     if not progress:
96 #         raise HTTPException(status_code=404, detail="Scenario progress not found")
97 #     return progress

```

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

```

1 from typing import List
2
3 from fastapi import APIRouter, Depends, HTTPException, Request
4 from fastapi.responses import JSONResponse
5 from pydantic import BaseModel
6
7 from app.models.user import UserBase, UserCreate, UserProfile, UserUpdate
8 from app.services.auth_service import (auth_callback, get_current_user,
9                                         get_linked_accounts, get_user_profile,
10                                        login_user, logout_user, process_token,
11                                        register_user, social_login,
12                                        update_user_profile)
13
14 router = APIRouter()
15
16 @router.post("/register")
17 async def register(user: UserCreate):
18     try:
19         result = register_user(user.email, user.password, user.nickname)
20         return result
21     except Exception as e:
22         raise HTTPException(status_code=400, detail=str(e))
23
24 class UserBase(BaseModel):
25     email: str
26     password: str
27
28 @router.post("/login")
29 async def login(user: UserBase):
30     try:
31         result = login_user(user.email, user.password)
32         return result
33     except Exception as e:
34         raise HTTPException(status_code=401, detail=str(e))
35
36 @router.post("/social-login/{provider}")
37 async def social_login_route(provider: str, request: Request):
38     try:
39         response = social_login(provider, request)
40         return JSONResponse(content=response)
41     except HTTPException as e:
42         return JSONResponse(content={"detail": e.detail}, status_code=e.status_code)
43
44 @router.get("/auth/callback")
45 async def auth_callback_route(request: Request):
46     return await auth_callback(request)
47
48 @router.post("/process_token")
49 async def process_token_route(token_data: dict):
50     try:
51         response = await process_token(token_data)
52         return JSONResponse(content=response)
53     except HTTPException as e:
54         return JSONResponse(content={"detail": e.detail}, status_code=e.status_code)
55
56 @router.get("/profile", response_model=UserProfile)
57 async def get_profile(user=Depends(get_current_user)):
58     return await get_user_profile(user)
59
60 @router.put("/profile", response_model=UserProfile)
61 async def update_profile(user_update: UserUpdate, user=Depends(get_current_user)):
62     return await update_user_profile(user_update, user)
63
64 @router.post("/logout")
65 async def logout(user=Depends(get_current_user)):
66     return await logout_user()
67
68 @router.get("/linked-accounts", response_model=List[str])
69 async def linked_accounts(user=Depends(get_current_user)):
70     return await get_linked_accounts(user)

```

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

```

1 [binary]

```

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

```

1 from typing import List
2
3 from fastapi import APIRouter, Depends
4
5 from app.models.character import (CharacterCreate, CharacterProfile,
6                                   CharacterUpdate)
7 from app.models.user import UserProfile as User
8 from app.services.auth_service import get_current_user
9 from app.services.character_service import (create_character, delete_character,
10                                             get_character, list_characters,
11                                             update_character)
12
13 router = APIRouter()
14
15 @router.post("/characters", response_model=CharacterProfile)
16 async def create_character_route(character: CharacterCreate, current_user: User = Depends(get_current_user)):
17     return await create_character(character, current_user)
18
19 @router.get("/characters/{character_id}", response_model=CharacterProfile)
20 async def get_character_route(character_id: str, current_user: User = Depends(get_current_user)):
21     return await get_character(character_id, current_user)
22
23 @router.put("/characters/{character_id}", response_model=CharacterProfile)
24 async def update_character_route(character_id: str, character: CharacterUpdate, current_user: User = Depends(get_current_user)):
25     return await update_character(character_id, character, current_user)
26
27 @router.delete("/characters/{character_id}")
28 async def delete_character_route(character_id: str, current_user: User = Depends(get_current_user)):
29     await delete_character(character_id, current_user)
30     return {"message": "Character deleted successfully"}
31
32 @router.get("/characters", response_model=List[CharacterProfile])
33 async def list_characters_route(current_user: User = Depends(get_current_user)):
34     return await list_characters(current_user)

```

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

```

1 from typing import List
2
3 from fastapi import APIRouter, Depends, Query
4
5 from app.models.conversation import (ConversationCreate, ConversationProfile,
6                                     ConversationUpdate, MessageCreate,
7                                     MessageProfile)
8 from app.models.relationship import RelationshipType,
9                                     UserCharacterInteractionUpdate)
10 from app.models.user import UserProfile as User
11 from app.services.auth_service import get_current_user
12 from app.services.conversation_service import ConversationService
13
14 router = APIRouter()
15 conversation_service = ConversationService()
16
17 @router.post("/conversations", response_model=ConversationProfile)
18 async def create_conversation_route(conversation: ConversationCreate, current_user: User = Depends(get_current_user)):
19     return await conversation_service.create_conversation(conversation, current_user)
20
21 @router.get("/conversations/{conversation_id}", response_model=ConversationProfile)
22 async def get_conversation_route(conversation_id: str, current_user: User = Depends(get_current_user)):
23     return await conversation_service.get_conversation(conversation_id, current_user)
24
25 @router.put("/conversations/{conversation_id}", response_model=ConversationProfile)
26 async def update_conversation_route(conversation_id: str, conversation: ConversationUpdate, current_user: User = Depends(get_current_user)):
27     return await conversation_service.update_conversation(conversation_id, conversation, current_user)
28
29 @router.delete("/conversations/{conversation_id}")
30 async def delete_conversation_route(conversation_id: str, current_user: User = Depends(get_current_user)):
31     await conversation_service.delete_conversation(conversation_id, current_user)
32     return {"message": "Conversation deleted successfully"}
33
34 @router.get("/conversations", response_model=List[ConversationProfile])
35 async def list_conversations_route(current_user: User = Depends(get_current_user)):
36     return await conversation_service.list_conversations(current_user)
37
38 @router.post("/conversations/{conversation_id}/messages", response_model=MessageProfile)
39 async def create_message_route(conversation_id: str, message: MessageCreate, current_user: User = Depends(get_current_user)):
40     return await conversation_service.create_message_and_respond(conversation_id, message, current_user)
41
42 @router.get("/conversations/{conversation_id}/messages", response_model=List[MessageProfile])
43 async def list_messages_route(conversation_id: str, current_user: User = Depends(get_current_user)):
44     return await conversation_service.list_messages(conversation_id, current_user)
45
46 @router.get("/conversations/{conversation_id}/messages/{message_id}", response_model=MessageProfile)
47 async def get_message_route(conversation_id: str, message_id: str, current_user: User = Depends(get_current_user)):
48     return await conversation_service.get_message(message_id, current_user)
49
50 @router.post("/conversations/{conversation_id}/summarize")
51 async def summarize_conversation_route(conversation_id: str, current_user: User = Depends(get_current_user)):
52     summary = await conversation_service.summarize_conversation(conversation_id, current_user)
53     return {"summary": summary}
54
55 @router.get("/conversations/{conversation_id}/similar-messages")
56 async def get_similar_messages_route(
57     conversation_id: str,
58     message_content: str = Query(..., description="Content of the message to find similar ones"),
59     top_k: int = Query(5, description="Number of similar messages to return"),
60     current_user: User = Depends(get_current_user)
61 ):
62     similar_messages = await conversation_service.get_similar_messages(conversation_id, message_content, top_k)
63     return {"similar_messages": similar_messages}
64
65 @router.get("/conversations/{conversation_id}/message-count")
66 async def get_message_count_route(conversation_id: str, current_user: User = Depends(get_current_user)):
67     count = await conversation_service.get_message_count(conversation_id)
68     return {"message_count": count}
69
70 @router.put("/conversations/{conversation_id}/nickname")
71 async def update_nickname_route(
72     conversation_id: str,
73     nickname: str = Query(..., description="New nickname for the user"),
74     current_user: User = Depends(get_current_user)
75 ):
76     conversation = await conversation_service.get_conversation(conversation_id, current_user)
77     await conversation_service.relationship_service.update_interaction(
78         str(conversation.character_id),
79         str(current_user.id),
80         UserCharacterInteractionUpdate(nickname=nickname)
81     )
82     return {"message": "Nickname updated successfully"}
83
84 @router.put("/conversations/{conversation_id}/relationship-type")
85 async def update_relationship_type_route(
86     conversation_id: str,
87     relationship_type: RelationshipType,
88     current_user: User = Depends(get_current_user)
89 ):
90     conversation = await conversation_service.get_conversation(conversation_id, current_user)
91     await conversation_service.relationship_service.update_interaction(
92         str(conversation.character_id),
93         str(current_user.id),
94         relationship_type
95     )
96     return {"message": "Relationship type updated successfully"}

```

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


```

1 import asyncio
2 import os
3 from typing import Any, Dict, List
4 from dotenv import load_dotenv
5
6 # .env
7 load_dotenv()
8
9 import openai
10 from pinecone import Pinecone, ServerlessSpec
11
12
13 class AIService:
14     def __init__(self):
15         openai.api_key = os.environ.get('OPENAI_API_KEY')
16
17         # Pinecone
18         self.pc = Pinecone(api_key=os.environ.get("PINECONE_API_KEY"))
19
20         #
21         index_name = os.environ.get("PINECONE_INDEX_NAME")
22
23         #
24         if index_name not in self.pc.list_indexes().names():
25             self.pc.create_index(
26                 name=index_name,
27                 dimension=1536, # OpenAI text-embedding-ada-002
28                 metric='cosine',
29                 spec=ServerlessSpec(cloud=os.environ.get("PINECONE_CLOUD", "aws"),
30                                     region=os.environ.get("PINECONE_REGION", "us-west-2"))
31             )
32
33         #
34         self.index = self.pc.Index(index_name)
35
36     def vectorize_text(self, text: str) -> List[float]:
37         """ """
38         response = openai.Embedding.create(
39             input=text,
40             model="text-embedding-ada-002"
41         )
42         embedding = response['data'][0]['embedding']
43         return embedding
44
45     def store_vector(self, id: str, vector: List[float], metadata: Dict[str, Any]):
46         """ Pinecone """
47         self.index.upsert(vectors=[(id, vector, metadata)])
48
49     async def store_vector_async(self, id: str, vector: List[float], metadata: Dict[str, Any]):
50         """ Pinecone """
51         # run_in_executor
52         await asyncio.get_event_loop().run_in_executor(
53             None, self.store_vector, id, vector, metadata
54         )
55
56
57     def search_similar_vectors(self, vector: List[float], conversation_id: str, top_k: int = 5) -> List[Dict[str, Any]]:
58         """
59
60         :param vector:
61         :param conversation_id: ID
62         :param top_k:
63         :return: (ID, , )
64         """
65
66         results = self.index.query(
67             vector=vector,
68             top_k=top_k,
69             include_metadata=True,
70             filter={"conversation_id": conversation_id}
71         )
72         return results['matches']
73
74     async def generate_response(self, context: str) -> str:
75         """
76         AI
77
78         :param context:
79         :return: AI
80         """
81         # TODO: AI
82         # AI API
83         return f"AI response based on context: {context[:50]}..." #

```

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

```

1 import logging
2 import os
3
4 from fastapi import APIRouter, Depends, HTTPException, Request
5 from fastapi.responses import HTMLResponse
6 from fastapi.security import HTTPAuthorizationCredentials, HTTPBearer
7 from pydantic import BaseModel
8 from supabase import Client, create_client
9
10 router = APIRouter()
11
12 logging.basicConfig(level=logging.DEBUG)
13 logger = logging.getLogger(__name__)
14
15 supabase_url = os.getenv("SUPABASE_URL")
16 supabase_key = os.getenv("SUPABASE_KEY")
17 if not supabase_url or not supabase_key:
18     raise ValueError("SUPABASE_URL and SUPABASE_KEY must be set in .env file")
19
20 supabase: Client = create_client(supabase_url, supabase_key)
21
22 security = HTTPBearer()
23
24 class User(BaseModel):
25     id: str
26     email: str
27     is_admin: bool = False # is_admin
28
29
30 def get_supabase_token(email: str, password: str) -> str:
31     try:
32         response = supabase.auth.sign_in_with_password({"email": email, "password": password})
33         if response and response.session:
34             return response.session.access_token
35         else:
36             raise HTTPException(status_code=401, detail="Unable to authenticate with Supabase")
37     except Exception as e:
38         print(f"Supabase Auth Error: {str(e)}")
39         raise HTTPException(status_code=401, detail="Supabase authentication error")
40

```

```

41 async def get_current_user(credentials: HTTPAuthorizationCredentials = Depends(security)):
42     token = credentials.credentials
43     try:
44         response = supabase.auth.get_user(token)
45         if response and response.user:
46             # is_admin
47             user_data = supabase.table("users").select("is_admin").eq("id", response.user.id).single().execute()
48             is_admin = user_data.data.get('is_admin', False) if user_data.data else False
49             return User(id=response.user.id, email=response.user.email, is_admin=is_admin) # is_admin
50             raise HTTPException(status_code=401, detail="Invalid authentication credentials")
51     except Exception as e:
52         raise HTTPException(status_code=401, detail=f"Invalid authentication credentials: {str(e)}")
53
54
55 def register_user(email: str, password: str, nickname: str):
56     try:
57         auth_response = supabase.auth.sign_up({
58             "email": email,
59             "password": password
60         })
61
62         if auth_response.user:
63             user_data = supabase.table("users").insert({
64                 "id": auth_response.user.id,
65                 "email": email,
66                 "nickname": nickname,
67                 "login_type": "email",
68                 "is_admin": False #
69             }).execute()
70
71             logger.info(f"User data insert response: {user_data}")
72
73             return {
74                 "message": "User registered successfully",
75                 "user_id": auth_response.user.id
76             }
77         else:
78             raise HTTPException(status_code=400, detail="Registration failed")
79     except Exception as e:
80         logger.error(f"Registration error: {str(e)}")
81         raise HTTPException(status_code=400, detail=str(e))
82
83
84
85 def login_user(email: str, password: str):
86     try:
87         logger.info(f"Attempting login for email: {email}")
88         auth_response = supabase.auth.sign_in_with_password({"email": email, "password": password})
89         logger.info(f"Auth response: {auth_response}")
90
91         if auth_response.user and auth_response.session:
92             return {
93                 "message": "Login successful",
94                 "access_token": auth_response.session.access_token,
95                 "user_id": auth_response.user.id
96             }
97         else:
98             logger.error("Login failed: User or session not found in auth response")
99             raise HTTPException(status_code=401, detail="Invalid credentials")
100     except Exception as e:
101         logger.error(f"Login error: {str(e)}")
102         if "Invalid login credentials" in str(e):
103             raise HTTPException(status_code=401, detail="Invalid email or password")
104         elif "Email not confirmed" in str(e):
105             raise HTTPException(status_code=401, detail="Email not confirmed. Please check your email for verification link.")
106         else:
107             raise HTTPException(status_code=401, detail="Login failed. Please try again.")
108
109 def social_login(provider: str, request: Request):
110     try:
111         callback_url = "http://localhost:8000/auth/callback"
112         logger.info(f"Callback URL: {callback_url}")
113         auth_response = supabase.auth.sign_in_with_oauth({
114             "provider": provider,
115             "options": {
116                 "redirect_to": callback_url
117             }
118         })
119
120         logger.info(f"Auth response: {auth_response}")
121         if hasattr(auth_response, 'url'):
122             return {"url": auth_response.url}
123         else:
124             raise HTTPException(status_code=400, detail="OAuth initialization failed")
125     except Exception as e:
126         logger.error(f"Social Login Error: {str(e)}")
127         raise HTTPException(status_code=400, detail=f"Social login error: {str(e)}")
128
129
130 async def auth_callback(request: Request):
131     try:
132         logger.debug("Auth callback hit")
133         code = request.query_params.get('code')
134         error = request.query_params.get('error')
135         logger.debug(f"Received code: {code}, error: {error}")
136
137         # URL HTML
138         html_content = """
139         <html>
140         <body>
141             <script>
142                 function sendTokenToFrontend() {
143                     var hash = window.location.hash.substring(1);
144                     var params = new URLSearchParams(hash);
145                     var access_token = params.get('access_token');
146
147                     if (!access_token) {
148                         window.location.href = 'https://localhost:3000/auth-error?error=no_token';
149                         return;
150                     }
151
152                     //
153                     window.location.href = 'https://localhost:3000/auth-success?token=' + access_token;
154                 }
155                 window.onload = sendTokenToFrontend;
156             </script>
157             <h1>Processing authentication...</h1>
158         </body>
159         </html>
160         """
161         return HTMLResponse(content=html_content)
162     except Exception as e:
163         logger.exception(f"Error in auth_callback: {str(e)}")
164         raise HTTPException(status_code=400, detail=f"Error processing authentication: {str(e)}")
165
166 # async def auth_callback(request: Request):
167 #     try:
168 #         logger.debug("Auth callback hit")
169 #         code = request.query_params.get('code')
170 #         error = request.query_params.get('error')

```

```

171 #         logger.debug(f"Received code: {code}, error: {error}")
172
173
174 #         # URL      HTML
175 #         html_content = ""
176 #         <html>
177 #         <body>
178 #             <script>
179 #                 function sendTokenToServer() {
180 #                     var hash = window.location.hash.substring(1);
181 #                     var params = new URLSearchParams(hash);
182 #                     var access_token = params.get('access_token');
183
184 #                     if (!access_token) {
185 #                         document.body.innerHTML = '<h1>Error: No access token found</h1>';
186 #                         return;
187 #                     }
188
189 #                     fetch('/process_token', {
190 #                         method: 'POST',
191 #                         headers: {
192 #                             'Content-Type': 'application/json',
193 #                         },
194 #                         body: JSON.stringify({access_token: access_token}),
195 #                     })
196 #                     .then(response => {
197 #                         if (!response.ok) {
198 #                             return response.json().then(err => {
199 #                                 throw new Error(err.detail || 'Unknown error occurred');
200 #                             });
201 #                         }
202 #                         return response.json();
203 #                     })
204 #                     .then(data => {
205 #                         if (data.message && data.user_id) {
206 #                             document.body.innerHTML = '<h1>${data.message}</h1><p>User ID: ${data.user_id}</p>';
207 #                             setTimeout(() => {
208 #                                 window.location.href = '/';
209 #                                 // Redirect back to the original page after 3 seconds
210 #                             }, 2000);
211 #                         } else {
212 #                             throw new Error('Invalid response data');
213 #                         }
214 #                     })
215 #                     .catch((error) => {
216 #                         console.error('Error:', error);
217 #                         document.body.innerHTML = '<h1>Error occurred during authentication</h1><p>' + error.message + '</p>';
218 #                     });
219 #                 }
220 #             </script>
221 #             <h1>Processing authentication...</h1>
222 #         </body>
223 #         </html>
224 #         ""
225 #         return HTMLResponse(content=html_content)
226 #     except Exception as e:
227 #         logger.exception(f"Error in auth_callback: {str(e)}")
228 #         raise HTTPException(status_code=400, detail=f"Error processing authentication: {str(e)}")
229
230
231 async def process_token(token_data: dict):
232     try:
233         access_token = token_data.get('access_token')
234         if not access_token:
235             raise HTTPException(status_code=400, detail="Access token not provided")
236
237         #
238         user = supabase.auth.get_user(access_token)
239
240         if not user or not user.user:
241             raise HTTPException(status_code=400, detail="User information not found")
242
243         user_id = user.user.id
244         user_email = user.user.email
245
246         #
247         user_data = supabase.table("users").select("*,").eq("id", user_id).execute()
248         logger.debug(f"User data: {user_data}")
249
250         if user_data.data:
251             #
252             message = f"Successfully logged in."
253             logger.debug(f"Existing user logged in: {user_id}")
254         else:
255             #
256             new_user = {
257                 "id": user_id,
258                 "email": user_email,
259                 "nickname": f"User_{user_id[:8]}",
260                 "login_type": "social",
261                 "is_admin": False
262             }
263             insert_result = supabase.table("users").insert(new_user).execute()
264             logger.debug(f"Insert result: {insert_result}")
265             if not insert_result.data:
266                 raise HTTPException(status_code=500, detail="Failed to create new user")
267             message = f"New user successfully created."
268             logger.debug(f"New user created: {user_id}")
269
270         response_data = {
271             "message": message,
272             "user_id": user_id
273         }
274         logger.debug(f"Returning response: {response_data}")
275         return response_data
276
277     except Exception as e:
278         logger.exception(f"Detailed error in process_token: {str(e)}")
279         raise HTTPException(status_code=400, detail=f"Error processing authentication: {str(e)}")
280
281 async def get_user_profile(user: User = Depends(get_current_user)):
282     try:
283         logger.debug(f"Fetching profile for user ID: {user.id}")
284         user_data = supabase.table("users").select("*,").eq("id", user.id).single().execute()
285         if user_data.data:
286             logger.debug(f"User data retrieved: {user_data.data}")
287             # is admin
288             return {"user_data": user_data.data, "is_admin": user_data.data.is_admin}
289         else:
290             logger.error("User profile not found")
291             raise HTTPException(status_code=404, detail="User profile not found")
292     except Exception as e:
293         logger.error(f"Error fetching user profile: {str(e)}")
294         raise HTTPException(status_code=400, detail=str(e))
295
296 async def update_user_profile(user_update: User):
297     try:
298         update_data = user_update.dict(exclude_unset=True)
299         user_data = supabase.table("users").update(update_data).eq("id", user_update.id).execute()
300         if user_data and user_data.get("data"):

```

```

301         return user_data["data"][0]
302     else:
303         raise HTTPException(status_code=404, detail="User profile not found")
304 except Exception as e:
305     raise HTTPException(status_code=400, detail=str(e))
306
307 async def logout_user():
308     try:
309         supabase.auth.sign_out()
310         return {"message": "Logout successful"}
311     except Exception as e:
312         raise HTTPException(status_code=400, detail=str(e))
313
314 async def get_linked_accounts(user):
315     try:
316         user_data = supabase.table("users").select("login_type").eq("id", user.id).execute()
317         if user_data and user_data.get("data"):
318             return [user_data["data"][0]['login_type']]
319         else:
320             return []
321     except Exception as e:
322         raise HTTPException(status_code=400, detail=str(e))

```

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

```
1 [binary]
```

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

```

1  import json
2  import logging
3  import os
4  from datetime import datetime, timezone
5  from typing import Dict, List
6
7  import tiktoken
8  from fastapi import APIRouter, HTTPException
9  from langchain_core.prompts import PromptTemplate
10 from langchain_core.runnables import RunnableSequence
11 from langchain_openai import OpenAI, ChatOpenAI
12 from langchain_core.documents import Document
13 from langchain.memory import ConversationBufferWindowMemory
14 from langchain_core.messages import AIMessage, HumanMessage
15 from langchain.chains.summarize import load_summarize_chain
16 from langchain.chains import LLMChain
17 from supabase import Client, create_client
18
19 from app.config import redis_client
20 from app.models.conversation import (ConversationCreate, ConversationProfile,
21                                     ConversationUpdate, MessageCreate,
22                                     MessageProfile)
23 from app.models.relationship import (UserCharacterInteractionCreate,
24                                     UserCharacterInteractionInDB,
25                                     UserCharacterInteractionUpdate)
26 from app.models.user import UserProfile as User
27 from app.services.ai_service import AIService
28 from app.services.relationship_service import RelationshipService
29
30
31 router = APIRouter()
32
33 logging.basicConfig(level=logging.DEBUG)
34 logger = logging.getLogger(__name__)
35
36 supabase_url = os.getenv("SUPABASE_URL")
37 supabase_key = os.getenv("SUPABASE_KEY")
38 if not supabase_url or not supabase_key:
39     raise ValueError("SUPABASE_URL and SUPABASE_KEY must be set in .env file")
40
41 supabase: Client = create_client(supabase_url, supabase_key)
42
43 class ConversationContextManager:
44     def __init__(self, window_size: int = 10): # window_size:
45         self.memory = ConversationBufferWindowMemory(k=window_size) #
46         self.relationship_info = {} #
47         self.current_scenario = None #
48
49
50     def add_message(self, role: str, content: str): # role: , content:
51         if role == 'human':
52             self.memory.chat_memory.add_message(HumanMessage(content=content))
53         elif role == 'ai':
54             self.memory.chat_memory.add_message(AIMessage(content=content))
55
56     def get_conversation_history(self) -> List[Dict[str, str]]:
57         return [{"role": msg.type, "content": msg.content} for msg in self.memory.chat_memory.messages]
58
59     def update_relationship_info(self, affinity: float, interaction_count: int):
60         self.relationship_info.update({
61             "affinity": affinity,
62             "interaction_count": interaction_count
63         })
64
65     def set_current_scenario(self, scenario: Dict[str, any]):
66         self.current_scenario = scenario
67
68     def get_formatted_context(self) -> str:
69         context = {
70             "conversation_history": self.get_conversation_history(),
71             "relationship_info": self.relationship_info,
72             "current_scenario": self.current_scenario
73         }
74         return json.dumps(context, ensure_ascii=False, indent=2)
75
76     def clear_context(self):
77         self.memory.clear()
78         self.relationship_info = {}
79         self.current_scenario = None
80
81 class ConversationService:
82     def __init__(self):
83         self.context_manager = ConversationContextManager()
84         self.llm = OpenAI(temperature=0) # OpenAI
85         self.summarize_chain = load_summarize_chain(self.llm, chain_type="map_reduce")
86         self.ai_service = AIService()
87
88         self.relationship_service = RelationshipService()
89         self.llm = ChatOpenAI(temperature=0.7)
90         self.prompt_template = PromptTemplate(
91             input_variables=["context", "recent_messages", "summary", "similar_messages", "affinity", "nickname"],
92             template="""
93             AI .
94             :
95             : {context}

```

```

96         : {recent_messages}
97         : {summary}
98         : {similar_messages}
99         ( ): {affinity}
100        : {nickname}
101
102        """
103
104    )
105    self.llm_chain = LLMChain(llm=self.llm, prompt=self.prompt_template)
106
107    self.relationship_service = RelationshipService()
108    self.llm = ChatOpenAI(temperature=0.7)
109    self.affinity_prompt = PromptTemplate(
110        input_variables=["summary"],
111        template="""
112        :
113        {summary}
114
115        , AI
116        -5 5
117        -5 AI , 0 , 5
118
119        :
120        """
121    )
122    self.affinity_chain = LLMChain(llm=self.llm, prompt=self.affinity_prompt)
123
124    self.redis_client = redis_client
125
126
127
128    async def summarize_conversation(self, conversation_id: str, current_user: User) -> str:
129        #
130        messages = await self.list_messages(conversation_id, current_user)
131
132        # 10
133        recent_messages = messages[-10:]
134
135        # Document
136        docs = [Document(page_content=msg.content) for msg in recent_messages]
137
138        #
139        summary = self.summarize_chain.run(docs)
140
141        #
142        await self.save_summary(conversation_id, summary)
143
144        # AI
145        affinity_change = await self.calculate_affinity_change(summary)
146
147        #
148        conversation = await self.get_conversation(conversation_id, current_user)
149        await self.relationship_service.update_affinity(str(conversation.character_id), str(current_user.id), affinity_change)
150
151        return summary
152
153    async def save_summary(self, conversation_id: str, summary: str):
154        """
155
156        :param conversation_id: ID
157        :param summary:
158        """
159        try:
160            # Supabase
161            response = supabase.table("conversation_summaries").insert((
162                "conversation_id": conversation_id,
163                "summary": summary,
164                "created_at": datetime.now(timezone.utc).isoformat()
165            )).execute()
166
167            if not response.data:
168                raise HTTPException(status_code=400, detail="Failed to save summary")
169        except Exception as e:
170            logger.error(f"Error saving summary: {str(e)}")
171            raise HTTPException(status_code=400, detail=str(e))
172
173
174    async def create_message_and_respond(self, conversation_id: str, message: MessageCreate, current_user: User) -> MessageProfile:
175        created_message = await self.create_message(conversation_id, message, current_user)
176
177        if message.sender == "user":
178            conversation = await self.get_conversation(conversation_id, current_user)
179            ai_response = await self.generate_ai_response(conversation_id, str(current_user.id), str(conversation.character_id))
180
181            ai_message = MessageCreate(sender="character", content=ai_response)
182            await self.create_message(conversation_id, ai_message, current_user)
183
184        return created_message
185
186    async def create_conversation(self, conversation: ConversationCreate, current_user: User) -> ConversationProfile:
187        try:
188            conversation_data = conversation.model_dump()
189            conversation_data['user_id'] = str(current_user.id)
190            conversation_data['character_id'] = str(conversation.character_id)
191
192            response = supabase.table("conversations").insert(conversation_data).execute()
193
194            if response.data:
195                self.context_manager.clear_context()
196                return ConversationProfile(**response.data[0])
197            else:
198                raise HTTPException(status_code=400, detail="Failed to create conversation")
199        except Exception as e:
200            logger.error(f"Error creating conversation: {str(e)}")
201            raise HTTPException(status_code=400, detail=str(e))
202
203
204    async def get_conversation(self, conversation_id: str, current_user: User) -> ConversationProfile:
205        # Redis
206        cached_conversation = self.redis_client.get(f"conversation:{conversation_id}")
207        if cached_conversation:
208            conversation = json.loads(cached_conversation)
209            if conversation['user_id'] == current_user.id:
210                return ConversationProfile(**conversation)
211
212        # Redis
213        try:
214            response = supabase.table("conversations").select("*").eq("id", conversation_id).execute()
215            if response.data:
216                conversation = response.data[0]
217                if conversation['user_id'] == current_user.id:
218                    # Redis
219                    self.redis_client.setex(f"conversation:{conversation_id}", 3600, json.dumps(conversation)) # 1
220                    return ConversationProfile(**conversation)
221                else:
222                    raise HTTPException(status_code=403, detail="You don't have permission to access this conversation")
223            else:
224                raise HTTPException(status_code=404, detail="Conversation not found")
225        except Exception as e:

```

```

226         logger.error(f"Error getting conversation: {str(e)}")
227         raise HTTPException(status_code=400, detail=str(e))
228
229     async def update_conversation(self, conversation_id: str, conversation: ConversationUpdate, current_user: User) -> ConversationProfile:
230     try:
231         existing_conversation = await self.get_conversation(conversation_id, current_user)
232         if existing_conversation.user_id != current_user.id:
233             raise HTTPException(status_code=403, detail="You don't have permission to update this conversation")
234
235         update_data = conversation.model_dump(exclude_unset=True)
236
237         response = supabase.table("conversations").update(update_data).eq("id", conversation_id).execute()
238         if response.data:
239             return ConversationProfile(**response.data[0])
240         else:
241             raise HTTPException(status_code=400, detail="Failed to update conversation")
242     except Exception as e:
243         logger.error(f"Error updating conversation: {str(e)}")
244         raise HTTPException(status_code=400, detail=str(e))
245
246     async def delete_conversation(self, conversation_id: str, current_user: User):
247     try:
248         existing_conversation = await self.get_conversation(conversation_id, current_user)
249         if existing_conversation.user_id != current_user.id:
250             raise HTTPException(status_code=403, detail="You don't have permission to delete this conversation")
251
252         response = supabase.table("conversations").delete().eq("id", conversation_id).execute()
253         if not response.data:
254             raise HTTPException(status_code=400, detail="Failed to delete conversation")
255     except Exception as e:
256         logger.error(f"Error deleting conversation: {str(e)}")
257         raise HTTPException(status_code=400, detail=str(e))
258
259     async def list_conversations(self, current_user: User) -> List[ConversationProfile]:
260     try:
261         response = supabase.table("conversations").select("*").eq("user_id", current_user.id).execute()
262         if response.data:
263             return [ConversationProfile(**conversation) for conversation in response.data]
264         else:
265             return []
266     except Exception as e:
267         logger.error(f"Error listing conversations: {str(e)}")
268         raise HTTPException(status_code=400, detail=str(e))
269
270     async def create_message(self, conversation_id: str, message: MessageCreate, current_user: User) -> MessageProfile:
271     try:
272         await self.get_conversation(conversation_id, current_user)
273
274         message_data = message.model_dump()
275         message_data["conversation_id"] = conversation_id
276
277         self.context_manager.add_message("human" if message.sender == "user" else "ai", message.content)
278
279         #
280         message_count = await self.get_message_count(conversation_id)
281         # 10
282         if message_count > 10 == 0:
283             await self.summarize_conversation(conversation_id, current_user)
284
285         #
286         vector = self.ai_service.vectorize_text(message.content)
287         message_data["embedding"] = vector
288
289         # Supabase
290         response = supabase.table("messages").insert(message_data).execute()
291
292         if response.data:
293             created_message = MessageProfile(**response.data[0])
294
295             # Redis
296             self.redis_client.lpush(f"recent_messages:{conversation_id}", json.dumps(created_message.dict()))
297             self.redis_client.ltrim(f"recent_messages:{conversation_id}", 0, 9) # 10
298
299             # Pinecone
300             await self.ai_service.store_vector_async(
301                 id=str(created_message.id),
302                 vector=vector,
303                 metadata={
304                     "conversation_id": conversation_id,
305                     "content": message.content,
306                     "created_at": created_message.created_at.isoformat()
307                 }
308             )
309
310             return created_message
311         else:
312             raise HTTPException(status_code=400, detail="Failed to create message")
313     except Exception as e:
314         logger.error(f"Error creating message: {str(e)}")
315         raise HTTPException(status_code=400, detail=str(e))
316
317     async def get_similar_messages(self, conversation_id: str, message_content: str, top_k: int = 5) -> List[MessageProfile]:
318     """
319     """
320
321     :param conversation_id: ID
322     :param message_content:
323     :param top_k:
324     :return:
325     """
326     vector = self.ai_service.vectorize_text(message_content)
327     similar_vectors = self.ai_service.search_similar_vectors(vector, conversation_id, top_k)
328
329     similar_messages = []
330     for match in similar_vectors:
331         message_id = match['id']
332         message = await self.get_message(message_id, None) # None .
333         similar_messages.append(message)
334
335     return similar_messages
336
337     async def get_message_count(self, conversation_id: str) -> int:
338     """
339     """
340
341     :param conversation_id: ID
342     :return:
343     """
344     try:
345         response = supabase.table("messages").select("id", count="exact").eq("conversation_id", conversation_id).execute()
346         return response.count
347     except Exception as e:
348         logger.error(f"Error getting message count: {str(e)}")
349         raise HTTPException(status_code=400, detail=str(e))
350
351
352
353
354
355

```

```

356 async def list_messages(self, conversation_id: str, current_user: User) -> List[MessageProfile]:
357     try:
358         await self.get_conversation(conversation_id, current_user)
359
360         response = supabase.table("messages").select(".*").eq("conversation_id", conversation_id).order("created_at").execute()
361
362         if response.data:
363             return [MessageProfile(**message) for message in response.data]
364         else:
365             return []
366     except Exception as e:
367         logger.error(f"Error listing messages: {str(e)}")
368         raise HTTPException(status_code=400, detail=str(e))
369
370 async def get_message(self, message_id: str, current_user: User) -> MessageProfile:
371     try:
372         response = supabase.table("messages").select(".*").eq("id", message_id).execute()
373         if response.data:
374             message = response.data[0]
375             conversation = await self.get_conversation(message['conversation_id'], current_user)
376             if conversation.user_id != current_user.id:
377                 raise HTTPException(status_code=403, detail="You don't have permission to access this message")
378             return MessageProfile(**message)
379         else:
380             raise HTTPException(status_code=404, detail="Message not found")
381     except Exception as e:
382         logger.error(f"Error getting message: {str(e)}")
383         raise HTTPException(status_code=400, detail=str(e))
384
385 async def update_relationship(self, user_id: str, character_id: str, affinity: float, interaction_count: int):
386     try:
387         self.context_manager.update_relationship_info(affinity, interaction_count)
388     except Exception as e:
389         logger.error(f"Error updating relationship: {str(e)}")
390         raise HTTPException(status_code=400, detail=str(e))
391
392 async def set_scenario(self, scenario_id: str):
393     try:
394         scenario = await self.get_scenario_from_db(scenario_id)
395         self.context_manager.set_current_scenario(scenario)
396     except Exception as e:
397         logger.error(f"Error setting scenario: {str(e)}")
398         raise HTTPException(status_code=400, detail=str(e))
399
400 async def get_scenario_from_db(self, scenario_id: str):
401     # :
402     response = supabase.table("scenarios").select(".*").eq("id", scenario_id).execute()
403     if response.data:
404         return response.data[0]
405     else:
406         raise HTTPException(status_code=404, detail="Scenario not found")
407
408 async def get_conversation_summary(self, conversation_id: str) -> str:
409     # Redis
410     cached_summary = self.redis_client.get(f"conversation_summary:{conversation_id}")
411     if cached_summary:
412         return cached_summary
413
414     # Redis
415     try:
416         response = supabase.table("conversation_summaries").select("summary").eq("conversation_id", conversation_id).order("created_at", ascending=False).limit(1)
417         if response.data:
418             summary = response.data[0]['summary']
419             # Redis
420             self.redis_client.setex(f"conversation_summary:{conversation_id}", 3600, summary) # 1
421             return summary
422         return ""
423     except Exception as e:
424         logger.error(f"Error getting conversation summary: {str(e)}")
425         return ""
426
427 async def get_recent_messages(self, conversation_id: str, limit: int) -> List[MessageProfile]:
428     # Redis
429     cached_messages = self.redis_client.lrange(f"recent_messages:{conversation_id}", 0, limit - 1)
430     if cached_messages and len(cached_messages) == limit:
431         return [MessageProfile(**json.loads(msg)) for msg in cached_messages]
432
433     # Redis
434     try:
435         response = supabase.table("messages").select(".*").eq("conversation_id", conversation_id).order("created_at", ascending=False).limit(limit).execute()
436         messages = [MessageProfile(**msg) for msg in response.data][::-1]
437
438         # Redis
439         for msg in messages:
440             self.redis_client.lpush(f"recent_messages:{conversation_id}", json.dumps(msg.model_dump()))
441         self.redis_client.ltrim(f"recent_messages:{conversation_id}", 0, limit - 1)
442
443         return messages
444     except Exception as e:
445         logger.error(f"Error getting recent messages: {str(e)}")
446         return []
447
448 async def get_relationship(self, character_id: str, user_id: str) -> UserCharacterInteractionInDB:
449     try:
450         return await self.relationship_service.get_interaction(character_id, user_id)
451     except HTTPException:
452         #
453         new_relationship = UserCharacterInteractionCreate(character_id=character_id, user_id=user_id)
454         return await self.relationship_service.create_interaction(new_relationship)
455
456 async def generate_ai_response(self, conversation_id: str, user_id: str, character_id: str) -> str:
457     try:
458         recent_messages = await self.get_recent_messages(conversation_id, 10)
459         summary = await self.get_conversation_summary(conversation_id)
460         similar_messages = await self.get_similar_messages(conversation_id, recent_messages[-1].content, 3)
461         relationship = await self.relationship_service.get_interaction(user_id, character_id)
462         affinity_level = self.relationship_service.get_affinity_level(relationship.affinity)
463
464         #
465         self.context_manager.update_relationship_info(relationship.affinity, relationship.interaction_count)
466         self.context_manager.add_message("human", recent_messages[-1].content)
467
468         context = self.context_manager.get_formatted_context()
469
470         #
471         encoding = tiktoken.encoding_for_model("gpt-3.5-turbo")
472         max_tokens = 4096 # GPT-3.5-turbo
473         prompt_tokens = len(encoding.encode(context))
474         available_tokens = max_tokens - prompt_tokens - 100 #
475
476         prompt_template = PromptTemplate(
477             input_variables=["recent_messages", "summary", "similar_messages", "affinity_level", "relationship_type", "nickname"],
478             template="""
479             AI .
480             :
481             {context}
482             : {recent_messages}
483
484
485

```

```

486         : {summary}
487         : {similar_messages}
488         : {affinity_level}
489         : {relationship_type}
490         : {nickname}
491     """
492     .
493     .
494     .
495     """
496 )
497
498 llm = ChatOpenAI(temperature=0.7, max_tokens=available_tokens)
499 llm_chain = LLMChain(llm=llm, prompt=prompt_template)
500
501 ai_response = llm_chain.run(
502     context=context,
503     recent_messages=self.format_messages(recent_messages),
504     summary=summary,
505     similar_messages=self.format_messages(similar_messages),
506     affinity_level=affinity_level,
507     relationship_type=relationship.relationship_type.value,
508     nickname=relationship.nickname or ""
509 )
510
511 # AI
512 self.context_manager.add_message("ai", ai_response)
513
514 return ai_response
515 except Exception as e:
516     logger.error(f"Error generating AI response: {str(e)}")
517     return ""
518
519
520 def format_messages(self, messages: List[MessageProfile]) -> str:
521     return "\n".join([f"{msg.sender}: {msg.content}" for msg in messages])
522
523
524 async def update_affinity(self, conversation_id: str, user_id: str, character_id: str, message_content: str):
525     #
526     affinity_change = self.calculate_affinity_change(message_content)
527
528     relationship = await self.relationship_service.get_relationship(user_id, character_id)
529     new_affinity = max(-100, min(100, relationship.affinity + affinity_change))
530
531     await self.relationship_service.update_relationship(
532         user_id,
533         character_id,
534         UserCharacterInteractionUpdate(affinity=new_affinity, last_interaction=datetime.datetime.now(timezone.utc))
535     )
536
537 async def calculate_affinity_change(self, summary: str) -> float:
538     affinity_change_str = await self.affinity_chain.arun(summary=summary)
539     try:
540         affinity_change = float(affinity_change_str.strip())
541         return max(-5, min(5, affinity_change)) # -5 5
542     except ValueError:
543         return 0 #
544
545 # ConversationService

```

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

```
1 [binary]
```

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

```

1 # # services/scenario_service.py
2
3 # import asyncio
4 # from app.models.scenario import Scenario, ScenarioProgress, ScenarioCreate, ScenarioUpdate, ScenarioStep
5 # from app.services.relationship_service import RelationshipService
6 # from datetime import datetime
7 # from typing import List, Optional
8 # from app.config import supabase_client # Supabase
9
10 # async def db_get_scenario(scenario_id: str) -> Optional[Scenario]:
11 #     """
12 #     .
13 #     .
14 #     """
15 #     result = await supabase_client.table('scenarios').select('*', scenario_steps('*')).eq('id', scenario_id).execute()
16 #     if result.data:
17 #         scenario_data = result.data[0]
18 #         steps = [ScenarioStep(**step) for step in scenario_data.pop('scenario_steps')]
19 #         return Scenario(**scenario_data, steps=steps)
20 #     return None
21
22 # async def db_create_scenario(scenario: ScenarioCreate) -> Scenario:
23 #     """
24 #     .
25 #     .
26 #     """
27 #     scenario_data = scenario.model_dump(exclude='steps')
28 #     result = await supabase_client.table('scenarios').insert(scenario_data).execute()
29 #     created_scenario = result.data[0]
30
31 #     steps_data = [{"scenario_id": created_scenario['id'], "step_order": i, **step.model_dump()}
32 #                   for i, step in enumerate(scenario.steps)]
33 #     await supabase_client.table('scenario_steps').insert(steps_data).execute()
34 #
35 #     return await db_get_scenario(created_scenario['id'])
36
37 # async def db_update_scenario(scenario_id: str, scenario_update: ScenarioUpdate) -> Scenario:
38 #     """
39 #     .
40 #     .
41 #     """
42 #     update_data = scenario_update.model_dump(exclude_unset=True)
43 #     steps = update_data.pop('steps', None)
44
45 #     result = await supabase_client.table('scenarios').update(update_data).eq('id', scenario_id).execute()
46
47 #     if steps is not None:
48 #         #
49 #         await supabase_client.table('scenario_steps').delete().eq('scenario_id', scenario_id).execute()
50 #         steps_data = [{"scenario_id": scenario_id, "step_order": i, **step.model_dump()}
51 #                       for i, step in enumerate(steps)]
52 #         await supabase_client.table('scenario_steps').insert(steps_data).execute()
53
54 #     return await db_get_scenario(scenario_id)
55
56 # async def db_delete_scenario(scenario_id: str) -> bool:

```



```

57 # """
58 # .
59 # (CASCADE ).
60 # """
61 # result = await supabase_client.table('scenarios').delete().eq('id', scenario_id).execute()
62 # return len(result.data) > 0
63
64 # async def db_get_scenario_progress(user_id: str, scenario_id: str) -> Optional[ScenarioProgress]:
65 # """
66 # .
67 # """
68 # result = await supabase_client.table('scenario_progress').select('').eq('user_id', user_id).eq('scenario_id', scenario_id).execute()
69 # if result.data:
70 #     return ScenarioProgress(**result.data[0])
71 # return None
72
73 # async def db_update_scenario_progress(progress: ScenarioProgress) -> ScenarioProgress:
74 # """
75 # .
76 # """
77 # progress_data = progress.model_dump()
78 # result = await supabase_client.table('scenario_progress').upsert(progress_data).execute()
79 # return ScenarioProgress(**result.data[0])
80
81 # async def create_scenario(scenario: ScenarioCreate) -> Scenario:
82 # """
83 # ."""
84 # return await db_create_scenario(scenario)
85
86 # async def get_scenario(scenario_id: str) -> Optional[Scenario]:
87 # """ ID """
88 # return await db_get_scenario(scenario_id)
89
90 # async def update_scenario(scenario_id: str, scenario_update: ScenarioUpdate) -> Scenario:
91 # """ """
92 # return await db_update_scenario(scenario_id, scenario_update)
93
94 # async def delete_scenario(scenario_id: str) -> bool:
95 # """ """
96 # return await db_delete_scenario(scenario_id)
97
98 # async def check_scenario_trigger(user_id: str, character_id: str) -> Optional[Scenario]:
99 # """
100 # .
101 # """
102 # relationship_service = RelationshipService()
103 # relationship = await relationship_service.get_interaction(user_id, character_id)
104
105 # result = await supabase_client.table('scenarios').select('').eq('character_id', character_id).execute()
106 # scenarios = [Scenario(**scenario) for scenario in result.data]
107
108 #
109 # async def check_scenario(scenario):
110 #     if scenario.trigger_type == "affinity" and relationship.affinity >= scenario.trigger_value:
111 #         return scenario
112 #     return None
113
114 # checked_scenarios = await asyncio.gather(*[check_scenario(scenario) for scenario in scenarios])
115 # triggered_scenarios = [s for s in checked_scenarios if s is not None]
116
117 # return triggered_scenarios[0] if triggered_scenarios else None
118
119 # async def start_scenario(user_id: str, scenario_id: str) -> ScenarioProgress:
120 # """
121 # .
122 # """
123 # .
124 # """
125 # scenario = await get_scenario(scenario_id)
126 # if not scenario:
127 #     raise ValueError("Scenario not found")
128
129 # progress = ScenarioProgress(
130 #     id=f"{user_id}_{scenario_id}",
131 #     user_id=user_id,
132 #     scenario_id=scenario_id,
133 #     current_step=0,
134 #     started_at=datetime.now(),
135 #     is_completed=False
136 # )
137
138 # return await db_update_scenario_progress(progress)
139
140 # async def progress_scenario(user_id: str, scenario_id: str) -> ScenarioProgress:
141 # """
142 # .
143 # """
144 # .
145 # progress = await db_get_scenario_progress(user_id, scenario_id)
146 # if not progress:
147 #     raise ValueError("Scenario progress not found")
148
149 # scenario = await get_scenario(scenario_id)
150 # if not scenario:
151 #     raise ValueError("Scenario not found")
152
153 # if progress.current_step < len(scenario.steps) - 1:
154 #     progress.current_step += 1
155 # else:
156 #     progress.is_completed = True
157 #     progress.completed_at = datetime.now()
158
159 # return await db_update_scenario_progress(progress)
160
161 # async def get_scenario_message(scenario_id: str, step: int) -> str:
162 # """
163 # .
164 # """
165 # scenario = await get_scenario(scenario_id)
166 # if not scenario or step >= len(scenario.steps):
167 #     raise ValueError("Invalid scenario or step")
168
169 # return scenario.steps[step].content
170
171 #
172 # async def get_all_scenarios_for_character(character_id: str) -> List[Scenario]:
173 # """
174 # .
175 # """
176 # result = await supabase_client.table('scenarios').select('*', scenario_steps('*')).eq('character_id', character_id).execute()
177 # scenarios = []
178 # for scenario_data in result.data:
179 #     steps = [ScenarioStep(**step) for step in scenario_data.pop('scenario_steps')]
180 #     scenarios.append(Scenario(**scenario_data, steps=steps))
181 # return scenarios
182
183 # async def bulk_update_scenario_progress(progresses: List[ScenarioProgress]) -> List[ScenarioProgress]:
184 # """
185 # .
186 # """

```

```

187 #     progress_data = [progress.model_dump() for progress in progresses]
188 #     result = await supabase_client.table('scenario_progress').upsert(progress_data).execute()
189 #     return [ScenarioProgress(**data) for data in result.data]

```

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

```

1  import json
2  import os
3  from datetime import datetime, timezone
4
5  from fastapi import HTTPException
6  from supabase import Client, create_client
7
8  from app.config import redis_client
9  from app.models.relationship import (RelationshipType,
10                                     UserCharacterInteractionCreate,
11                                     UserCharacterInteractionInDB,
12                                     UserCharacterInteractionUpdate)
13
14  supabase_url = os.getenv("SUPABASE_URL")
15  supabase_key = os.getenv("SUPABASE_KEY")
16  if not supabase_url or not supabase_key:
17      raise ValueError("SUPABASE_URL and SUPABASE_KEY must be set in .env file")
18
19
20  supabase: Client = create_client(supabase_url, supabase_key)
21
22
23  class RelationshipService:
24      def __init__(self):
25          self.redis_client = redis_client
26
27      async def get_interaction(self, character_id: str, user_id: str) -> UserCharacterInteractionInDB:
28          # Redis
29          cached_interaction = self.redis_client.get(f"interaction:{character_id}:{user_id}")
30          if cached_interaction:
31              return UserCharacterInteractionInDB(**json.loads(cached_interaction))
32
33          # Redis
34          response = supabase.table("user_character_interactions").select("*, ").eq("character_id", character_id).eq("user_id", user_id).execute()
35          if response.data:
36              interaction = UserCharacterInteractionInDB(**response.data[0])
37              # Redis
38              self.redis_client.setex(f"interaction:{character_id}:{user_id}", 3600, json.dumps(interaction.model_dump())) # 1
39              return interaction
40          raise HTTPException(status_code=404, detail="Interaction not found")
41
42
43      async def create_interaction(self, interaction: UserCharacterInteractionCreate) -> UserCharacterInteractionInDB:
44          response = supabase.table("user_character_interactions").insert(interaction.model_dump()).execute()
45          if response.data:
46              created_interaction = UserCharacterInteractionInDB(**response.data[0])
47              # Redis
48              self.redis_client.setex(
49                  f"interaction:{created_interaction.character_id}:{created_interaction.user_id}",
50                  3600,
51                  json.dumps(created_interaction.model_dump())
52              )
53              return created_interaction
54          raise HTTPException(status_code=400, detail="Failed to create interaction")
55
56      async def update_interaction(self, character_id: str, user_id: str, interaction: UserCharacterInteractionUpdate) -> UserCharacterInteractionInDB:
57          update_data = interaction.model_dump(exclude_unset=True)
58          update_data['last_interaction'] = datetime.now(timezone.utc)
59          response = supabase.table("user_character_interactions").update(update_data).eq("character_id", character_id).eq("user_id", user_id).execute()
60          if response.data:
61              updated_interaction = UserCharacterInteractionInDB(**response.data[0])
62              # Redis
63              self.redis_client.setex(f"interaction:{character_id}:{user_id}", 3600, json.dumps(updated_interaction.model_dump()))
64              return updated_interaction
65          raise HTTPException(status_code=400, detail="Failed to update interaction")
66
67
68      async def update_affinity(self, character_id: str, user_id: str, affinity_change: float):
69          interaction = await self.get_interaction(character_id, user_id)
70          new_affinity = max(-100, min(100, interaction.affinity + affinity_change))
71          new_relationship_type = self.get_relationship_type(new_affinity)
72          updated_interaction = await self.update_interaction(
73              character_id,
74              user_id,
75              UserCharacterInteractionUpdate(
76                  affinity=new_affinity,
77                  relationship_type=new_relationship_type,
78                  interaction_count=interaction.interaction_count + 1
79              )
80          )
81          # Redis
82          self.redis_client.setex(f"interaction:{character_id}:{user_id}", 3600, json.dumps(updated_interaction.model_dump()))
83
84      def get_affinity_level(self, affinity: float) -> str:
85          if affinity <= -91:
86              return " "
87          elif affinity <= -71:
88              return " "
89          elif affinity <= -41:
90              return ""
91          elif affinity <= -11:
92              return " "
93          elif affinity <= 10:
94              return ""
95          elif affinity <= 40:
96              return ""
97          elif affinity <= 60:
98              return " "
99          elif affinity <= 70:
100             return " "
101          elif affinity <= 90:
102             return " "
103          else:
104             return " "
105
106      def get_relationship_type(self, affinity: float) -> RelationshipType:
107          if affinity <= -91:
108              return RelationshipType.ENEMY
109          elif affinity <= -51:
110              return RelationshipType.RIVAL
111          elif affinity <= -10:
112              return RelationshipType.STRANGER
113          elif affinity <= 20:
114              return RelationshipType.ACQUAINTANCE
115          elif affinity <= 30:
116              return RelationshipType.FRIEND
117          elif affinity <= 50:
118              return RelationshipType.CLOSE_FRIEND
119          elif affinity <= 70:
120              return RelationshipType.LOVER
121          elif affinity <= 100:

```

```

122         return RelationshipType.SPOUSE
123     else:
124         raise ValueError("Affinity value out of bounds")
125
126     async def update_custom_traits(self, character_id: str, user_id: str, custom_traits: dict):
127         updated_interaction = await self.update_interaction(
128             character_id,
129             user_id,
130             UserCharacterInteractionUpdate(custom_traits=custom_traits)
131         )
132         # Redis
133         self.redis_client.setex(f"interaction:{character_id}:{user_id}", 3600, json.dumps(updated_interaction.model_dump()))
134
135     async def update_conversation_history(self, character_id: str, user_id: str, conversation_history: dict):
136         updated_interaction = await self.update_interaction(
137             character_id,
138             user_id,
139             UserCharacterInteractionUpdate(conversation_history=conversation_history)
140         )
141         # Redis
142         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

```

1 # services/character_service.py
2
3 import logging
4 import os
5 from typing import List
6
7 from fastapi import APIRouter, HTTPException
8 from supabase import Client, create_client
9
10 from app.models.character import (CharacterCreate, CharacterProfile,
11                                  CharacterUpdate)
12 from app.models.user import UserProfile as User
13
14 router = APIRouter()
15
16
17 logging.basicConfig(level=logging.DEBUG)
18 logger = logging.getLogger(__name__)
19
20 supabase_url = os.getenv("SUPABASE_URL")
21 supabase_key = os.getenv("SUPABASE_KEY")
22 if not supabase_url or not supabase_key:
23     raise ValueError("SUPABASE_URL and SUPABASE_KEY must be set in .env file")
24
25
26 supabase: Client = create_client(supabase_url, supabase_key)
27 def is_admin(user: User) -> bool:
28     return user.is_admin
29
30
31
32
33 async def create_character(character: CharacterCreate, current_user: User) -> CharacterProfile:
34     try:
35         character_data = character.model_dump()
36         character_data['creator_id'] = current_user.id
37
38         # Convert LocalizedContent fields to JSON
39         for field in ['names', 'occupation', 'background', 'appearance_description']:
40             if field in character_data:
41                 character_data[field] = character_data[field].dict()
42
43         response = supabase.table("characters").insert(character_data).execute()
44         if response.data:
45             return CharacterProfile(**response.data[0])
46         else:
47             raise HTTPException(status_code=400, detail="Failed to create character")
48     except Exception as e:
49         logger.error(f"Error creating character: {str(e)}")
50         raise HTTPException(status_code=400, detail=str(e))
51
52 async def get_character(character_id: str, current_user: User) -> CharacterProfile:
53     try:
54         response = supabase.table("characters").select("*").eq("id", character_id).execute()
55         if response.data:
56             character = response.data[0]
57             if character['creator_id'] == current_user.id or is_admin(current_user):
58                 return CharacterProfile(**character)
59             else:
60                 raise HTTPException(status_code=403, detail="You don't have permission to access this character")
61         else:
62             raise HTTPException(status_code=404, detail="Character not found")
63     except Exception as e:
64         logger.error(f"Error getting character: {str(e)}")
65         raise HTTPException(status_code=400, detail=str(e))
66
67 async def update_character(character_id: str, character: CharacterUpdate, current_user: User) -> CharacterProfile:
68     try:
69         existing_character = await get_character(character_id, current_user)
70         if existing_character.creator_id != current_user.id and not is_admin(current_user):
71             raise HTTPException(status_code=403, detail="You don't have permission to update this character")
72
73         update_data = character.model_dump(exclude_unset=True)
74
75         # Convert LocalizedContent fields to JSON
76         for field in ['names', 'occupation', 'background', 'appearance_description']:
77             if field in update_data:
78                 update_data[field] = update_data[field].dict()
79
80         response = supabase.table("characters").update(update_data).eq("id", character_id).execute()
81         if response.data:
82             return CharacterProfile(**response.data[0])
83         else:
84             raise HTTPException(status_code=400, detail="Failed to update character")
85     except Exception as e:
86         logger.error(f"Error updating character: {str(e)}")
87         raise HTTPException(status_code=400, detail=str(e))
88
89 async def delete_character(character_id: str, current_user: User):
90     try:
91         existing_character = await get_character(character_id, current_user)
92         if existing_character.creator_id != current_user.id and not is_admin(current_user):
93             raise HTTPException(status_code=403, detail="You don't have permission to delete this character")
94
95         response = supabase.table("characters").delete().eq("id", character_id).execute()
96         if not response.data:
97             raise HTTPException(status_code=400, detail="Failed to delete character")
98     except Exception as e:
99         logger.error(f"Error deleting character: {str(e)}")
100         raise HTTPException(status_code=400, detail=str(e))
101
102 async def list_characters(current_user: User) -> List[CharacterProfile]:
103     try:
104         if is_admin(current_user):
105             response = supabase.table("characters").select("*").execute()
106         else:
107             response = supabase.table("characters").select("*").or_(f"creator_id.eq.{current_user.id},is_public.eq.true").execute()
108
109         if response.data:
110             return [CharacterProfile(**character) for character in response.data]
111         else:
112             return []
113     except Exception as e:
114         logger.error(f"Error listing characters: {str(e)}")
115         raise HTTPException(status_code=400, detail=str(e))

```

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

```

aiohappyeyeballs==2.3.4
aiohhttp==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
frozenset==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

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

```

10 <!-- Registration Form -->
11 <h2>Register</h2>
12 <form id="register-form">
13   <input type="text" id="register-email" placeholder="Email" required>
14   <input type="password" id="register-password" placeholder="Password" required>
15   <input type="text" id="register-nickname" placeholder="Nickname" required>
16   <button type="submit">Register</button>
17 </form>
18
19 <!-- Login Form -->
20 <h2>Login</h2>
21 <form id="login-form">
22   <input type="text" id="login-email" placeholder="Email" required>
23   <input type="password" id="login-password" placeholder="Password" required>
24   <button type="submit">Login</button>
25 </form>
26
27 <!-- Social Login -->
28 <h2>Social Login</h2>
29 <button id="google-login">Login with Google</button>
30 <button id="github-login">Login with GitHub</button>
31 <button id="facebook-login">Login with Facebook</button>
32
33 <!-- Logout Button -->
34 <h2>Logout</h2>
35 <button id="logout-button">Logout</button>
36
37 <!-- Profile Form -->
38 <h2>Profile</h2>
39 <form id="profile-form">
40   <input type="text" id="profile-username" placeholder="Username" required>
41   <input type="text" id="profile-email" placeholder="Email">
42   <button type="submit">Update Profile</button>
43 </form>
44 <button id="get-profile-button">Get Profile</button>
45
46 <!-- Output -->
47 <h2>Output</h2>
48 <pre id="output"></pre>
49
50 <script>
51   const apiUrl = 'http://localhost:8000'; // Update with your API base URL
52   let token = localStorage.getItem('token') || '';
53
54   document.getElementById('register-form').addEventListener('submit', async (event) => {
55     event.preventDefault();
56     const email = document.getElementById('register-email').value;
57     const password = document.getElementById('register-password').value;
58     const nickname = document.getElementById('register-nickname').value;
59     const response = await fetch(`${apiUrl}/register`, {
60       method: 'POST',
61       headers: {
62         'Content-Type': 'application/json'
63       },
64       body: JSON.stringify({ email, password, nickname })
65     });
66     const data = await response.json();
67     document.getElementById('output').innerText = JSON.stringify(data, null, 2);
68   });
69
70   document.getElementById('login-form').addEventListener('submit', async (event) => {
71     event.preventDefault();
72     const email = document.getElementById('login-email').value;
73     const password = document.getElementById('login-password').value;
74     const response = await fetch(`${apiUrl}/login`, {
75       method: 'POST',
76       headers: {
77         'Content-Type': 'application/json'
78       },
79       body: JSON.stringify({ email, password })
80     });
81     const data = await response.json();
82     if (response.ok) {
83       token = data.access_token;
84       localStorage.setItem('token', token);
85     }
86     document.getElementById('output').innerText = JSON.stringify(data, null, 2);
87   });
88
89   document.getElementById('logout-button').addEventListener('click', async () => {
90     const response = await fetch(`${apiUrl}/logout`, {
91       method: 'POST',
92       headers: {
93         'Authorization': 'Bearer ${token}'
94       }
95     });
96     const data = await response.json();
97     token = '';
98     localStorage.removeItem('token');
99     document.getElementById('output').innerText = JSON.stringify(data, null, 2);
100   });
101
102   document.getElementById('profile-form').addEventListener('submit', async (event) => {
103     event.preventDefault();
104     const username = document.getElementById('profile-username').value;
105     const email = document.getElementById('profile-email').value;
106     const response = await fetch(`${apiUrl}/profile`, {
107       method: 'PUT',
108       headers: {
109         'Content-Type': 'application/json',
110         'Authorization': 'Bearer ${token}'
111       },
112       body: JSON.stringify({ username, email })
113     });
114     const data = await response.json();
115     document.getElementById('output').innerText = JSON.stringify(data, null, 2);
116   });
117
118   document.getElementById('get-profile-button').addEventListener('click', async () => {
119     console.log("Sending GET request to /profile", 'Bearer ${token}');
120     const response = await fetch(`${apiUrl}/profile`, {
121       method: 'GET',
122       headers: {
123         'Authorization': 'Bearer ${token}'
124       }
125     });
126     const data = await response.json();
127     document.getElementById('output').innerText = JSON.stringify(data, null, 2);
128   });
129
130   document.getElementById('google-login').addEventListener('click', async () => {
131     localStorage.setItem('original_url', window.location.href);
132     const response = await fetch(`${apiUrl}/social-login/google`, {
133       method: 'POST',
134       headers: {
135         'Content-Type': 'application/json'
136       }
137     });
138     const data = await response.json();
139

```

