# main.py (сервер)

import sys

import os

# main.py (сервер)

import sysimport os

import uuid

from datetime import datetime, timedelta

from typing import Dict, Optional, List

from shared.logger import setup\_logger

class SessionManager:

def \_\_init\_\_(self):

self.active\_sessions: Dict[str, Dict] = {} # file\_path:username -> session\_data

self.closed\_sessions: Dict[str, List[Dict]] = {} # История закрытых сессий

self.logger = setup\_logger(\_\_name\_\_)

self.config = {}

def set\_config(self, config: dict):

"""Устанавливает конфигурацию"""

self.config = config

timeout = self.config.get('session\_timeout\_minutes', 1)

self.logger.info(f"⚙️ Session config: timeout={timeout}min, max\_age={self.config.get('max\_session\_hours', 3)}h")

def \_get\_session\_key(self, file\_path: str, username: str) -> str:

"""Генерирует ключ сессии"""

return f"{file\_path}:{username}"

def \_find\_recently\_closed(self, session\_key: str, hours: int = 1) -> Optional[Dict]:

"""Находит недавно закрытую сессию для возможного возобновления"""

if session\_key not in self.closed\_sessions:

return None

closed\_sessions = self.closed\_sessions[session\_key]

if not closed\_sessions:

return None

# Берем последнюю закрытую сессию

last\_session = closed\_sessions[-1]

# Проверяем, закрыта ли она в пределах указанного времени

if 'ended\_at' in last\_session and last\_session['ended\_at']:

time\_since\_close = datetime.now() - last\_session['ended\_at']

if time\_since\_close <= timedelta(hours=hours):

return last\_session

return None

def \_resume\_session(self, session\_data: Dict, file\_hash: str = None) -> Dict:

"""Возобновляет существующую сессию"""

session\_key = self.\_get\_session\_key(session\_data['file\_path'], session\_data['username'])

# Обновляем данные сессии

resumed\_session = session\_data.copy()

resumed\_session['last\_activity'] = datetime.now()

resumed\_session['resumed\_at'] = datetime.now()

resumed\_session['resume\_count'] = resumed\_session.get('resume\_count', 0) + 1

resumed\_session['hash\_before'] = file\_hash

# Убираем поля окончания, т.к. сессия снова активна

resumed\_session.pop('ended\_at', None)

resumed\_session.pop('hash\_after', None)

# ДОБАВЛЕНО: Сохраняем информацию о со-редакторах при возобновлении

if 'co\_editors' in session\_data:

resumed\_session['co\_editors'] = session\_data['co\_editors']

if 'is\_multi\_user' in session\_data:

resumed\_session['is\_multi\_user'] = session\_data['is\_multi\_user']

# Возвращаем в активные сессии

self.active\_sessions[session\_key] = resumed\_session

# Удаляем из истории закрытых, если она там есть

if session\_key in self.closed\_sessions and session\_data in self.closed\_sessions[session\_key]:

self.closed\_sessions[session\_key].remove(session\_data)

self.logger.info(f"🔄 Resumed session for {resumed\_session['file\_path']}")

return resumed\_session

def get\_active\_session(self, file\_path: str, username: str) -> Optional[Dict]:

"""Возвращает активную сессию для файла и пользователя"""

session\_key = self.\_get\_session\_key(file\_path, username)

session\_data = self.active\_sessions.get(session\_key)

if session\_data:

# Проверяем не истекла ли сессия

if self.\_is\_session\_expired(session\_data):

# Сессия истекла - закрываем ее

self.logger.info(f"🕒 Session expired, closing: {file\_path}")

self.close\_session(file\_path, username)

return None

# Обновляем время последней активности

session\_data['last\_activity'] = datetime.now()

return session\_data

def \_is\_session\_expired(self, session\_data: Dict) -> bool:

"""Проверяет истекла ли сессия"""

timeout\_minutes = self.config.get('session\_timeout\_minutes', 1)

max\_age\_hours = self.config.get('max\_session\_hours', 3)

last\_activity = session\_data['last\_activity']

session\_age = datetime.now() - session\_data['started\_at']

time\_since\_activity = datetime.now() - last\_activity

timeout\_seconds = timeout\_minutes \* 60

# Проверяем таймаут активности

if time\_since\_activity.total\_seconds() > timeout\_seconds:

self.logger.info(f"🕒 Session expired by timeout: {session\_data['file\_path']}, inactive for {time\_since\_activity.total\_seconds():.1f}s > {timeout\_seconds}s")

return True

# Проверяем максимальный возраст сессии

if session\_age.total\_seconds() > (max\_age\_hours \* 3600):

self.logger.info(f"📅 Session expired by max age: {session\_data['file\_path']}, age: {session\_age.total\_seconds()/3600:.1f}h")

return True

return False

def check\_and\_close\_expired\_sessions(self) -> List[Dict]:

"""Проверяет и закрывает все просроченные сессии"""

expired\_sessions = []

total\_sessions = len(self.active\_sessions)

if total\_sessions == 0:

return expired\_sessions

self.logger.info(f"🔍 Checking {total\_sessions} active sessions for expiration...")

# Создаем копию списка для безопасной итерации

sessions\_to\_check = list(self.active\_sessions.items())

for session\_key, session\_data in sessions\_to\_check:

try:

if self.\_is\_session\_expired(session\_data):

file\_path = session\_data['file\_path']

username = session\_data['username']

time\_since\_activity = datetime.now() - session\_data['last\_activity']

self.logger.info(f"🔒 Closing expired session: {file\_path} (inactive: {time\_since\_activity.total\_seconds():.1f}s)")

# Закрываем сессию и получаем данные с ended\_at

closed\_data = self.close\_session(file\_path, username)

if closed\_data:

expired\_sessions.append(closed\_data)

self.logger.info(f"✅ Session closed with ended\_at: {closed\_data['ended\_at']}")

except Exception as e:

self.logger.error(f"❌ Error checking session {session\_key}: {e}")

if expired\_sessions:

self.logger.info(f"✅ Closed {len(expired\_sessions)} expired sessions")

else:

self.logger.debug(f"📊 All {total\_sessions} sessions are active")

return expired\_sessions

def smart\_create\_session(self, file\_path: str, username: str, file\_hash: str = None, resume\_window\_hours: int = 1) -> Dict:

"""Умное создание сессии с возможностью возобновления недавно закрытой сессии"""

# Сначала проверяем активную сессию

active\_session = self.get\_active\_session(file\_path, username)

if active\_session:

return active\_session

# Пытаемся найти недавно закрытую сессию для возобновления

session\_key = self.\_get\_session\_key(file\_path, username)

recently\_closed = self.\_find\_recently\_closed(session\_key, resume\_window\_hours)

if recently\_closed:

return self.\_resume\_session(recently\_closed, file\_hash)

else:

return self.create\_session(file\_path, username, file\_hash)

def create\_session(self, file\_path: str, username: str, file\_hash: str = None) -> Dict:

"""Создает новую сессию"""

session\_key = self.\_get\_session\_key(file\_path, username)

session\_data = {

'session\_id': str(uuid.uuid4()),

'file\_path': file\_path,

'file\_name': os.path.basename(file\_path),

'username': username,

'started\_at': datetime.now(),

'last\_activity': datetime.now(),

'hash\_before': file\_hash,

'events': [],

'resume\_count': 0

}

self.active\_sessions[session\_key] = session\_data

self.logger.info(f"✅ Created session for {file\_path}")

return session\_data

def update\_session(self, file\_path: str, username: str, file\_hash: str = None) -> Dict:

"""Обновляет существующую сессию или создает новую"""

session\_data = self.get\_active\_session(file\_path, username)

if session\_data:

# Обновляем существующую сессию

if file\_hash:

session\_data['hash\_after'] = file\_hash

session\_data['last\_activity'] = datetime.now()

self.logger.debug(f"📝 Updated session for {file\_path}")

else:

# Создаем новую сессию

session\_data = self.create\_session(file\_path, username, file\_hash)

return session\_data

def close\_session(self, file\_path: str, username: str, file\_hash: str = None) -> Optional[Dict]:

"""Закрывает сессию"""

session\_key = self.\_get\_session\_key(file\_path, username)

session\_data = self.active\_sessions.pop(session\_key, None)

if session\_data:

# Устанавливаем время окончания

ended\_at = datetime.now()

session\_data['ended\_at'] = ended\_at

if file\_hash:

session\_data['hash\_after'] = file\_hash

# Сохраняем в историю закрытых сессий

if session\_key not in self.closed\_sessions:

self.closed\_sessions[session\_key] = []

self.closed\_sessions[session\_key].append(session\_data)

# Ограничиваем историю

if len(self.closed\_sessions[session\_key]) > 10:

self.closed\_sessions[session\_key] = self.closed\_sessions[session\_key][-10:]

# Логируем с информацией о времени сессии

session\_duration = ended\_at - session\_data['started\_at']

self.logger.info(f"🔒 Closed session for {file\_path} (duration: {session\_duration.total\_seconds():.1f}s, ended\_at: {ended\_at})")

return session\_data

else:

self.logger.debug(f"❌ No active session found for: {file\_path} (user: {username})")

return None

def close\_all\_sessions\_for\_file(self, file\_path: str) -> List[Dict]:

"""Принудительно закрывает все сессии для указанного файла"""

closed\_sessions = []

sessions\_to\_close = []

for session\_key, session\_data in list(self.active\_sessions.items()):

if session\_data['file\_path'] == file\_path:

sessions\_to\_close.append((session\_data['file\_path'], session\_data['username']))

self.logger.info(f"🔍 Found {len(sessions\_to\_close)} sessions to close for: {file\_path}")

for file\_path, username in sessions\_to\_close:

session\_data = self.close\_session(file\_path, username)

if session\_data:

closed\_sessions.append(session\_data)

return closed\_sessions

def transfer\_session(self, old\_file\_path: str, new\_file\_path: str, username: str, file\_hash: str = None) -> Optional[Dict]:

"""Переносит сессию со старого пути на новый"""

session\_key\_old = self.\_get\_session\_key(old\_file\_path, username)

session\_data = self.active\_sessions.pop(session\_key\_old, None)

if session\_data:

# Обновляем путь файла в данных сессии

session\_data['file\_path'] = new\_file\_path

session\_data['file\_name'] = os.path.basename(new\_file\_path)

session\_data['last\_activity'] = datetime.now()

if file\_hash:

session\_data['hash\_after'] = file\_hash

# Сохраняем с новым ключом

session\_key\_new = self.\_get\_session\_key(new\_file\_path, username)

self.active\_sessions[session\_key\_new] = session\_data

# Также обновляем в истории закрытых сессий, если нужно

if session\_key\_old in self.closed\_sessions:

self.closed\_sessions[session\_key\_new] = self.closed\_sessions.pop(session\_key\_old)

self.logger.info(f"🔄 Transferred session from {old\_file\_path} to {new\_file\_path}")

return session\_data

return None

def cleanup\_expired\_sessions(self, event\_handler) -> list:

"""Очищает просроченные сессии"""

return self.check\_and\_close\_expired\_sessions()

def get\_session\_stats(self) -> Dict:

"""Возвращает статистику по сессиям"""

total\_resumes = sum(session.get('resume\_count', 0) for session in self.active\_sessions.values())

# ДОБАВЛЕНО: Статистика по многопользовательским сессиям

multi\_user\_sessions = sum(1 for session in self.active\_sessions.values() if session.get('is\_multi\_user', False))

return {

'active\_sessions': len(self.active\_sessions),

'session\_keys': list(self.active\_sessions.keys()),

'total\_resumes': total\_resumes,

'closed\_sessions\_count': sum(len(sessions) for sessions in self.closed\_sessions.values()),

'multi\_user\_sessions': multi\_user\_sessions

}

def get\_session\_history(self, file\_path: str, username: str) -> List[Dict]:

"""Возвращает историю сессий для файла и пользователя"""

session\_key = self.\_get\_session\_key(file\_path, username)

return self.closed\_sessions.get(session\_key, [])

import os

# Добавляем корень проекта в PYTHONPATH

current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

project\_root = os.path.dirname(os.path.dirname(current\_dir)) # Поднимаемся на уровень выше

sys.path.insert(0, project\_root)

from fastapi import FastAPI, Depends, HTTPException

from sqlalchemy.orm import Session

from shared.config\_loader import get\_api\_config

from . import database, models, crud, schemas

from typing import List, Optional

import uuid

from datetime import datetime

# Загружаем конфигурацию API

api\_config = get\_api\_config()

app = FastAPI(

title="File Monitoring API",

description="API для отслеживания изменений файлов с системой комментариев",

version="1.0.0"

)

# Dependency

def get\_db():

db = database.SessionLocal()

try:

yield db

finally:

db.close()

@app.get("/")

async def root():

return {

"message": "File Monitoring System API",

"status": "running",

"config\_source": "YAML"

}

@app.get("/health")

async def health\_check():

return {"status": "healthy", "service": "session-service"}

@app.get("/config")

async def show\_config():

"""Показывает текущую конфигурацию (без паролей)"""

return {

"api\_host": api\_config.get('host'),

"api\_port": api\_config.get('port'),

"environment": "development"

}

# Events endpoint для мониторинга

@app.post("/api/events")

async def create\_event(event\_data: dict, db: Session = Depends(get\_db)):

"""Эндпоинт для приема событий от мониторинга"""

print(f"📨 Received event: {event\_data}")

try:

# Обрабатываем событие в зависимости от типа

result = await process\_file\_event(db, event\_data)

return {"status": "processed", "event\_type": event\_data.get('event\_type'), "result": result}

except Exception as e:

print(f"❌ Error processing event: {e}")

raise HTTPException(status\_code=500, detail=f"Error processing event: {str(e)}")

async def process\_file\_event(db: Session, event\_data: dict):

"""Обрабатывает файловое событие и создает/обновляет сессии"""

event\_type = event\_data.get('event\_type')

file\_path = event\_data.get('file\_path')

file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

file\_hash = event\_data.get('file\_hash')

session\_id = event\_data.get('session\_id')

resume\_count = event\_data.get('resume\_count', 0)

print(f"🔧 Processing {event\_type} for {file\_path} (user: {username}, session: {session\_id}, resume: {resume\_count})")

# Обрабатываем перемещение отдельно

if event\_type == 'moved':

return await handle\_file\_moved(db, event\_data, file\_hash)

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# Обрабатываем в зависимости от типа события

if event\_type == 'created':

return await handle\_file\_created(db, user.id, file.id, event\_data, file\_hash, resume\_count)

elif event\_type == 'modified':

return await handle\_file\_modified(db, user.id, file.id, event\_data, file\_hash, session\_id, resume\_count)

elif event\_type == 'deleted':

return await handle\_file\_deleted(db, user.id, file.id, event\_data, session\_id)

elif event\_type == 'closed':

return await handle\_file\_closed(db, session\_id, file\_hash, event\_data)

else:

raise ValueError(f"Unknown event type: {event\_type}")

async def handle\_file\_moved(db: Session, event\_data: dict, file\_hash: str = None):

"""Обрабатывает перемещение/переименование файла"""

old\_file\_path = event\_data.get('old\_file\_path')

new\_file\_path = event\_data.get('file\_path')

new\_file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

print(f"🔄 Processing file move: {old\_file\_path} -> {new\_file\_path}")

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Ищем активную сессию для старого пути

old\_file = crud.get\_file\_by\_path(db, old\_file\_path)

if old\_file:

# Находим активные сессии для этого файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, old\_file.id)

if active\_sessions:

# Берем последнюю активную сессию

session = active\_sessions[0]

# Обновляем файл с новым путем

old\_file.file\_path = new\_file\_path

old\_file.file\_name = new\_file\_name

db.commit()

# Обновляем время активности сессии

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Moved session {session.id} from {old\_file\_path} to {new\_file\_path}")

return {"action": "session\_moved", "session\_id": str(session.id)}

# Если не нашли активную сессию, создаем новую

file\_data = schemas.FileCreate(file\_path=new\_file\_path, file\_name=new\_file\_name)

file = crud.create\_file(db, file\_data)

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"✅ Created new session {session.id} for moved file {new\_file\_path}")

return {"action": "session\_created", "session\_id": str(session.id)}

async def handle\_file\_created(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, resume\_count: int = 0):

"""Обрабатывает создание файла - создает новую сессию"""

# ПРОВЕРКА ДУБЛИРОВАНИЯ СЕССИЙ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

print(f"⚠️ Active session already exists for file, updating instead of creating new")

session = active\_sessions[0]

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_before = file\_hash

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

# Создаем событие created

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Updated existing session {session.id} for created file")

return {"action": "session\_updated", "session\_id": str(session.id), "resumed": False}

# Проверяем, есть ли недавно закрытая сессия для возобновления

recent\_session = None

if resume\_count > 0:

recent\_session = crud.get\_recent\_closed\_session(db, user\_id, file\_id)

if recent\_session:

# Возобновляем существующую сессию

recent\_session.ended\_at = None

recent\_session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

recent\_session.resume\_count = resume\_count

recent\_session.hash\_before = file\_hash

db.commit()

db.refresh(recent\_session)

print(f"🔄 Resumed session {recent\_session.id} (resume count: {resume\_count})")

session = recent\_session

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for file {event\_data.get('file\_path')}")

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": "session\_created", "session\_id": str(session.id), "resumed": recent\_session is not None}

async def handle\_file\_modified(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, session\_id: str = None, resume\_count: int = 0):

"""Обрабатывает изменение файла - обновляет существующую сессию или создает новую"""

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

if session:

# Обновляем существующую сессию

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

if resume\_count > session.resume\_count:

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

print(f"📝 Updated session {session.id} for file {event\_data.get('file\_path')} (resume: {resume\_count})")

action = "session\_updated"

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created new session {session.id} for modified file {event\_data.get('file\_path')}")

action = "session\_created"

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='modified',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": action, "session\_id": str(session.id)}

async def handle\_file\_deleted(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, session\_id: str = None):

"""Обрабатывает удаление файла - закрывает сессию и создает событие deleted"""

print(f"🗑️ Processing file deletion for file\_id: {file\_id}, session\_id: {session\_id}")

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

print(f"🔍 Found active session for deletion: {session.id}")

if session:

# Закрываем сессию с временем из события

session.ended\_at = datetime.fromisoformat(event\_data.get('event\_timestamp'))

db.commit()

db.refresh(session)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None, # Для удаленного файла хеша нет

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Closed session {session.id} for deleted file and created deleted event")

return {"action": "session\_closed", "session\_id": str(session.id)}

else:

print(f"⚠️ No active session found for deleted file, creating standalone deleted event")

# Если сессии нет, все равно создаем событие deleted

# Создаем временную сессию для привязки события

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

ended\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')), # Сразу закрываем

hash\_before=None,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Created standalone deleted event with session {session.id}")

return {"action": "deleted\_event\_created", "session\_id": str(session.id)}

async def handle\_file\_closed(db: Session, session\_id: str, file\_hash: str = None, event\_data: dict = None):

"""Обрабатывает закрытие файла - закрывает сессию с правильным ended\_at"""

if session\_id:

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

if session:

# ОБНОВЛЯЕМ СЕССИЮ С ПРАВИЛЬНЫМ ended\_at

if event\_data and 'event\_timestamp' in event\_data:

# Используем время из события закрытия

ended\_at = datetime.fromisoformat(event\_data['event\_timestamp'])

else:

# Используем текущее время

ended\_at = datetime.utcnow()

session.ended\_at = ended\_at

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие закрытия

event\_timestamp = ended\_at # Используем то же время что и для ended\_at

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='closed',

file\_hash=file\_hash,

event\_timestamp=event\_timestamp

)

crud.create\_file\_event(db, event\_record)

# Получаем информацию о файле для логов

file = crud.get\_file(db, session.file\_id)

user = crud.get\_user(db, session.user\_id)

duration = session.ended\_at - session.started\_at

print(f"🔒 Closed session {session.id} for file {file.file\_path} (user: {user.username}, duration: {duration}, ended\_at: {session.ended\_at})")

return {

"action": "session\_closed",

"session\_id": str(session.id),

"duration": str(duration),

"ended\_at": session.ended\_at.isoformat(),

"resume\_count": session.resume\_count

}

except ValueError:

pass

return {"action": "no\_session\_found"}

# Новый эндпоинт для создания сессий (для совместимости с мониторингом)

@app.post("/api/sessions")

async def create\_session(session\_data: dict, db: Session = Depends(get\_db)):

"""Создает сессию на сервере и возвращает session\_id"""

try:

print(f"📝 Creating session: {session\_data}")

# Получаем или создаем пользователя

username = session\_data.get('username')

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file\_path = session\_data.get('file\_path')

file\_name = session\_data.get('file\_name', os.path.basename(file\_path))

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# ПРОВЕРКА ДУБЛИРОВАНИЯ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, file.id)

if active\_sessions:

print(f"⚠️ Active session already exists, returning existing: {active\_sessions[0].id}")

return {"id": str(active\_sessions[0].id), "status": "existing"}

# Создаем сессию

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.utcnow(),

last\_activity=datetime.utcnow(),

hash\_before=session\_data.get('file\_hash'),

resume\_count=session\_data.get('resume\_count', 0)

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for {file\_path}")

return {"id": str(session.id), "status": "created"}

except Exception as e:

print(f"❌ Error creating session: {e}")

raise HTTPException(status\_code=500, detail=f"Error creating session: {str(e)}")

# Простые эндпоинты для проверки данных

@app.get("/api/users")

async def get\_users(db: Session = Depends(get\_db)):

"""Получает список пользователей"""

users = db.query(models.User).all()

return {"users": [{"id": str(user.id), "username": user.username} for user in users]}

@app.get("/api/files")

async def get\_files(db: Session = Depends(get\_db)):

"""Получает список файлов"""

files = db.query(models.File).all()

return {"files": [{"id": str(file.id), "path": file.file\_path, "name": file.file\_name} for file in files]}

@app.get("/api/sessions")

async def get\_sessions(db: Session = Depends(get\_db)):

"""Получает список сессий"""

sessions = db.query(models.FileSession).all()

return {"sessions": [

{

"id": str(session.id),

"user\_id": str(session.user\_id),

"file\_id": str(session.file\_id),

"started\_at": session.started\_at.isoformat(),

"ended\_at": session.ended\_at.isoformat() if session.ended\_at else None,

"last\_activity": session.last\_activity.isoformat(),

"resume\_count": session.resume\_count,

"is\_commented": session.is\_commented

} for session in sessions

]}

@app.get("/api/events")

async def get\_events(db: Session = Depends(get\_db)):

"""Получает список событий"""

events = db.query(models.FileEvent).all()

return {"events": [

{

"id": str(event.id),

"session\_id": str(event.session\_id),

"event\_type": event.event\_type,

"file\_hash": event.file\_hash,

"timestamp": event.event\_timestamp.isoformat()

} for event in events

]}

# НОВЫЕ ЭНДПОИНТЫ ДЛЯ КОММЕНТАРИЕВ

@app.post("/api/comments", response\_model=schemas.Comment)

async def create\_comment(comment: schemas.CommentCreate, db: Session = Depends(get\_db)):

"""Создает комментарий для сессии"""

try:

# Проверяем существование сессии

session = crud.get\_file\_session(db, comment.session\_id)

if not session:

raise HTTPException(status\_code=404, detail="Session not found")

# Проверяем существование пользователя

user = crud.get\_user(db, comment.user\_id)

if not user:

raise HTTPException(status\_code=404, detail="User not found")

# Проверяем, нет ли уже комментария для этой сессии

existing\_comment = crud.get\_comment\_by\_session(db, comment.session\_id)

if existing\_comment:

raise HTTPException(status\_code=400, detail="Comment already exists for this session")

session.ended\_at = datetime.utcnow()

session.is\_commented = True

# Создаем комментарий

db\_comment = crud.create\_comment(db, comment)

db.commit()

return db\_comment

except HTTPException:

raise

except Exception as e:

db.rollback()

raise HTTPException(status\_code=500, detail=f"Error creating comment: {str(e)}")

@app.get("/api/comments", response\_model=List[schemas.CommentWithUser])

async def get\_comments(

skip: int = 0,

limit: int = 100,

change\_type: Optional[str] = None,

db: Session = Depends(get\_db)

):

"""Получает список комментариев с возможностью фильтрации по типу изменения"""

try:

# Базовый запрос с join пользователя

query = db.query(

models.Comment,

models.User.username

).join(

models.User, models.Comment.user\_id == models.User.id

)

# Фильтрация по типу изменения если указан

if change\_type:

query = query.filter(models.Comment.change\_type == change\_type)

comments = query.offset(skip).limit(limit).all()

# Преобразуем в схему

result = []

for comment, username in comments:

result.append(schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

))

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comments: {str(e)}")

@app.get("/api/comments/{session\_id}", response\_model=schemas.CommentWithUser)

async def get\_comment\_by\_session(session\_id: str, db: Session = Depends(get\_db)):

"""Получает комментарий по ID сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_comment\_with\_user(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Comment not found")

comment, username = result

return schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comment: {str(e)}")

@app.get("/api/sessions-with-comments", response\_model=List[schemas.SessionWithDetails])

async def get\_sessions\_with\_comments(

skip: int = 0,

limit: int = 100,

db: Session = Depends(get\_db)

):

"""Получает сессии с комментариями"""

try:

sessions = crud.get\_sessions\_with\_comments(db, skip=skip, limit=limit)

result = []

for session, file\_path, file\_name, username, comment in sessions:

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

result.append(session\_details)

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting sessions: {str(e)}")

@app.get("/api/change-types")

async def get\_change\_types():

"""Возвращает доступные типы изменений"""

return {

"change\_types": [

"technical\_changes", # Изменение технических решений

"design\_changes", # Изменение оформления/дизайна

"content\_changes", # Изменение содержания

"bug\_fixes", # Исправление ошибок

"optimization", # Оптимизация

"refactoring", # Рефакторинг

"new\_feature", # Новая функциональность

"documentation", # Изменение документации

"other" # Другое

]

}

@app.get("/api/sessions/{session\_id}/details", response\_model=schemas.SessionWithDetails)

async def get\_session\_details(session\_id: str, db: Session = Depends(get\_db)):

"""Получает детальную информацию о сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_session\_with\_details(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Session not found")

session, file\_path, file\_name, username, comment = result

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

return session\_details

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting session details: {str(e)}")

if \_\_name\_\_ == "\_\_main\_\_":

import uvicorn

uvicorn.run(app, host="0.0.0.0", port=8000)

# Добавляем корень проекта в PYTHONPATH

current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

project\_root = os.path.dirname(os.path.dirname(current\_dir)) # Поднимаемся на уровень выше

sys.path.insert(0, project\_root)

from fastapi import FastAPI, Depends, HTTPException

from sqlalchemy.orm import Session

from shared.config\_loader import get\_api\_config

from . import database, models, crud, schemas

from typing import List, Optional

import uuid

from datetime import datetime

# Загружаем конфигурацию API

api\_config = get\_api\_config()

app = FastAPI(

title="File Monitoring API",

description="API для отслеживания изменений файлов с системой комментариев",

version="1.0.0"

)

# Dependency

def get\_db():

db = database.SessionLocal()

try:

yield db

finally:

db.close()

@app.get("/")

async def root():

return {

"message": "File Monitoring System API",

"status": "running",

"config\_source": "YAML"

}

@app.get("/health")

async def health\_check():

return {"status": "healthy", "service": "session-service"}

@app.get("/config")

async def show\_config():

"""Показывает текущую конфигурацию (без паролей)"""

return {

"api\_host": api\_config.get('host'),

"api\_port": api\_config.get('port'),

"environment": "development"

}

# Events endpoint для мониторинга

@app.post("/api/events")

async def create\_event(event\_data: dict, db: Session = Depends(get\_db)):

"""Эндпоинт для приема событий от мониторинга"""

print(f"📨 Received event: {event\_data}")

try:

# Обрабатываем событие в зависимости от типа

result = await process\_file\_event(db, event\_data)

return {"status": "processed", "event\_type": event\_data.get('event\_type'), "result": result}

except Exception as e:

print(f"❌ Error processing event: {e}")

raise HTTPException(status\_code=500, detail=f"Error processing event: {str(e)}")

async def process\_file\_event(db: Session, event\_data: dict):

"""Обрабатывает файловое событие и создает/обновляет сессии"""

event\_type = event\_data.get('event\_type')

file\_path = event\_data.get('file\_path')

file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

file\_hash = event\_data.get('file\_hash')

session\_id = event\_data.get('session\_id')

resume\_count = event\_data.get('resume\_count', 0)

print(f"🔧 Processing {event\_type} for {file\_path} (user: {username}, session: {session\_id}, resume: {resume\_count})")

# Обрабатываем перемещение отдельно

if event\_type == 'moved':

return await handle\_file\_moved(db, event\_data, file\_hash)

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# Обрабатываем в зависимости от типа события

if event\_type == 'created':

return await handle\_file\_created(db, user.id, file.id, event\_data, file\_hash, resume\_count)

elif event\_type == 'modified':

return await handle\_file\_modified(db, user.id, file.id, event\_data, file\_hash, session\_id, resume\_count)

elif event\_type == 'deleted':

return await handle\_file\_deleted(db, user.id, file.id, event\_data, session\_id)

elif event\_type == 'closed':

return await handle\_file\_closed(db, session\_id, file\_hash, event\_data)

else:

raise ValueError(f"Unknown event type: {event\_type}")

async def handle\_file\_moved(db: Session, event\_data: dict, file\_hash: str = None):

"""Обрабатывает перемещение/переименование файла"""

old\_file\_path = event\_data.get('old\_file\_path')

new\_file\_path = event\_data.get('file\_path')

new\_file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

print(f"🔄 Processing file move: {old\_file\_path} -> {new\_file\_path}")

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Ищем активную сессию для старого пути

old\_file = crud.get\_file\_by\_path(db, old\_file\_path)

if old\_file:

# Находим активные сессии для этого файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, old\_file.id)

if active\_sessions:

# Берем последнюю активную сессию

session = active\_sessions[0]

# Обновляем файл с новым путем

old\_file.file\_path = new\_file\_path

old\_file.file\_name = new\_file\_name

db.commit()

# Обновляем время активности сессии

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Moved session {session.id} from {old\_file\_path} to {new\_file\_path}")

return {"action": "session\_moved", "session\_id": str(session.id)}

# Если не нашли активную сессию, создаем новую

file\_data = schemas.FileCreate(file\_path=new\_file\_path, file\_name=new\_file\_name)

file = crud.create\_file(db, file\_data)

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"✅ Created new session {session.id} for moved file {new\_file\_path}")

return {"action": "session\_created", "session\_id": str(session.id)}

async def handle\_file\_created(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, resume\_count: int = 0):

"""Обрабатывает создание файла - создает новую сессию"""

# ПРОВЕРКА ДУБЛИРОВАНИЯ СЕССИЙ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

print(f"⚠️ Active session already exists for file, updating instead of creating new")

session = active\_sessions[0]

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_before = file\_hash

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

# Создаем событие created

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Updated existing session {session.id} for created file")

return {"action": "session\_updated", "session\_id": str(session.id), "resumed": False}

# Проверяем, есть ли недавно закрытая сессия для возобновления

recent\_session = None

if resume\_count > 0:

recent\_session = crud.get\_recent\_closed\_session(db, user\_id, file\_id)

if recent\_session:

# Возобновляем существующую сессию

recent\_session.ended\_at = None

recent\_session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

recent\_session.resume\_count = resume\_count

recent\_session.hash\_before = file\_hash

db.commit()

db.refresh(recent\_session)

print(f"🔄 Resumed session {recent\_session.id} (resume count: {resume\_count})")

session = recent\_session

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for file {event\_data.get('file\_path')}")

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": "session\_created", "session\_id": str(session.id), "resumed": recent\_session is not None}

async def handle\_file\_modified(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, session\_id: str = None, resume\_count: int = 0):

"""Обрабатывает изменение файла - обновляет существующую сессию или создает новую"""

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

if session:

# Обновляем существующую сессию

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

if resume\_count > session.resume\_count:

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

print(f"📝 Updated session {session.id} for file {event\_data.get('file\_path')} (resume: {resume\_count})")

action = "session\_updated"

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created new session {session.id} for modified file {event\_data.get('file\_path')}")

action = "session\_created"

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='modified',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": action, "session\_id": str(session.id)}

async def handle\_file\_deleted(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, session\_id: str = None):

"""Обрабатывает удаление файла - закрывает сессию и создает событие deleted"""

print(f"🗑️ Processing file deletion for file\_id: {file\_id}, session\_id: {session\_id}")

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

print(f"🔍 Found active session for deletion: {session.id}")

if session:

# Закрываем сессию с временем из события

session.ended\_at = datetime.fromisoformat(event\_data.get('event\_timestamp'))

db.commit()

db.refresh(session)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None, # Для удаленного файла хеша нет

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Closed session {session.id} for deleted file and created deleted event")

return {"action": "session\_closed", "session\_id": str(session.id)}

else:

print(f"⚠️ No active session found for deleted file, creating standalone deleted event")

# Если сессии нет, все равно создаем событие deleted

# Создаем временную сессию для привязки события

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

ended\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')), # Сразу закрываем

hash\_before=None,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Created standalone deleted event with session {session.id}")

return {"action": "deleted\_event\_created", "session\_id": str(session.id)}

async def handle\_file\_closed(db: Session, session\_id: str, file\_hash: str = None, event\_data: dict = None):

"""Обрабатывает закрытие файла - закрывает сессию с правильным ended\_at"""

if session\_id:

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

if session:

# ОБНОВЛЯЕМ СЕССИЮ С ПРАВИЛЬНЫМ ended\_at

if event\_data and 'event\_timestamp' in event\_data:

# Используем время из события закрытия

ended\_at = datetime.fromisoformat(event\_data['event\_timestamp'])

else:

# Используем текущее время

ended\_at = datetime.utcnow()

session.ended\_at = ended\_at

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие закрытия

event\_timestamp = ended\_at # Используем то же время что и для ended\_at

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='closed',

file\_hash=file\_hash,

event\_timestamp=event\_timestamp

)

crud.create\_file\_event(db, event\_record)

# Получаем информацию о файле для логов

file = crud.get\_file(db, session.file\_id)

user = crud.get\_user(db, session.user\_id)

duration = session.ended\_at - session.started\_at

print(f"🔒 Closed session {session.id} for file {file.file\_path} (user: {user.username}, duration: {duration}, ended\_at: {session.ended\_at})")

return {

"action": "session\_closed",

"session\_id": str(session.id),

"duration": str(duration),

"ended\_at": session.ended\_at.isoformat(),

"resume\_count": session.resume\_count

}

except ValueError:

pass

return {"action": "no\_session\_found"}

# Новый эндпоинт для создания сессий (для совместимости с мониторингом)

@app.post("/api/sessions")

async def create\_session(session\_data: dict, db: Session = Depends(get\_db)):

"""Создает сессию на сервере и возвращает session\_id"""

try:

print(f"📝 Creating session: {session\_data}")

# Получаем или создаем пользователя

username = session\_data.get('username')

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file\_path = session\_data.get('file\_path')

file\_name = session\_data.get('file\_name', os.path.basename(file\_path))

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# ПРОВЕРКА ДУБЛИРОВАНИЯ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, file.id)

if active\_sessions:

print(f"⚠️ Active session already exists, returning existing: {active\_sessions[0].id}")

return {"id": str(active\_sessions[0].id), "status": "existing"}

# Создаем сессию

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.utcnow(),

last\_activity=datetime.utcnow(),

hash\_before=session\_data.get('file\_hash'),

resume\_count=session\_data.get('resume\_count', 0)

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for {file\_path}")

return {"id": str(session.id), "status": "created"}

except Exception as e:

print(f"❌ Error creating session: {e}")

raise HTTPException(status\_code=500, detail=f"Error creating session: {str(e)}")

# Простые эндпоинты для проверки данных

@app.get("/api/users")

async def get\_users(db: Session = Depends(get\_db)):

"""Получает список пользователей"""

users = db.query(models.User).all()

return {"users": [{"id": str(user.id), "username": user.username} for user in users]}

@app.get("/api/files")

async def get\_files(db: Session = Depends(get\_db)):

"""Получает список файлов"""

files = db.query(models.File).all()

return {"files": [{"id": str(file.id), "path": file.file\_path, "name": file.file\_name} for file in files]}

@app.get("/api/sessions")

async def get\_sessions(db: Session = Depends(get\_db)):

"""Получает список сессий"""

sessions = db.query(models.FileSession).all()

return {"sessions": [

{

"id": str(session.id),

"user\_id": str(session.user\_id),

"file\_id": str(session.file\_id),

"started\_at": session.started\_at.isoformat(),

"ended\_at": session.ended\_at.isoformat() if session.ended\_at else None,

"last\_activity": session.last\_activity.isoformat(),

"resume\_count": session.resume\_count,

"is\_commented": session.is\_commented

} for session in sessions

]}

@app.get("/api/events")

async def get\_events(db: Session = Depends(get\_db)):

"""Получает список событий"""

events = db.query(models.FileEvent).all()

return {"events": [

{

"id": str(event.id),

"session\_id": str(event.session\_id),

"event\_type": event.event\_type,

"file\_hash": event.file\_hash,

"timestamp": event.event\_timestamp.isoformat()

} for event in events

]}

# НОВЫЕ ЭНДПОИНТЫ ДЛЯ КОММЕНТАРИЕВ

@app.post("/api/comments", response\_model=schemas.Comment)

async def create\_comment(comment: schemas.CommentCreate, db: Session = Depends(get\_db)):

"""Создает комментарий для сессии"""

try:

# Проверяем существование сессии

session = crud.get\_file\_session(db, comment.session\_id)

if not session:

raise HTTPException(status\_code=404, detail="Session not found")

# Проверяем существование пользователя

user = crud.get\_user(db, comment.user\_id)

if not user:

raise HTTPException(status\_code=404, detail="User not found")

# Проверяем, нет ли уже комментария для этой сессии

existing\_comment = crud.get\_comment\_by\_session(db, comment.session\_id)

if existing\_comment:

raise HTTPException(status\_code=400, detail="Comment already exists for this session")

session.ended\_at = datetime.utcnow()

session.is\_commented = True

# Создаем комментарий

db\_comment = crud.create\_comment(db, comment)

db.commit()

return db\_comment

except HTTPException:

raise

except Exception as e:

db.rollback()

raise HTTPException(status\_code=500, detail=f"Error creating comment: {str(e)}")

@app.get("/api/comments", response\_model=List[schemas.CommentWithUser])

async def get\_comments(

skip: int = 0,

limit: int = 100,

change\_type: Optional[str] = None,

db: Session = Depends(get\_db)

):

"""Получает список комментариев с возможностью фильтрации по типу изменения"""

try:

# Базовый запрос с join пользователя

query = db.query(

models.Comment,

models.User.username

).join(

models.User, models.Comment.user\_id == models.User.id

)

# Фильтрация по типу изменения если указан

if change\_type:

query = query.filter(models.Comment.change\_type == change\_type)

comments = query.offset(skip).limit(limit).all()

# Преобразуем в схему

result = []

for comment, username in comments:

result.append(schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

))

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comments: {str(e)}")

@app.get("/api/comments/{session\_id}", response\_model=schemas.CommentWithUser)

async def get\_comment\_by\_session(session\_id: str, db: Session = Depends(get\_db)):

"""Получает комментарий по ID сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_comment\_with\_user(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Comment not found")

comment, username = result

return schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comment: {str(e)}")

@app.get("/api/sessions-with-comments", response\_model=List[schemas.SessionWithDetails])

async def get\_sessions\_with\_comments(

skip: int = 0,

limit: int = 100,

db: Session = Depends(get\_db)

):

"""Получает сессии с комментариями"""

try:

sessions = crud.get\_sessions\_with\_comments(db, skip=skip, limit=limit)

result = []

for session, file\_path, file\_name, username, comment in sessions:

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

result.append(session\_details)

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting sessions: {str(e)}")

@app.get("/api/change-types")

async def get\_change\_types():

"""Возвращает доступные типы изменений"""

return {

"change\_types": [

"technical\_changes", # Изменение технических решений

"design\_changes", # Изменение оформления/дизайна

"content\_changes", # Изменение содержания

"bug\_fixes", # Исправление ошибок

"optimization", # Оптимизация

"refactoring", # Рефакторинг

"new\_feature", # Новая функциональность

"documentation", # Изменение документации

"other" # Другое

]

}

@app.get("/api/sessions/{session\_id}/details", response\_model=schemas.SessionWithDetails)

async def get\_session\_details(session\_id: str, db: Session = Depends(get\_db)):

"""Получает детальную информацию о сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_session\_with\_details(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Session not found")

session, file\_path, file\_name, username, comment = result

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

return session\_details

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting session details: {str(e)}")

if \_\_name\_\_ == "\_\_main\_\_":

import uvicorn

uvicorn.run(app, host="0.0.0.0", port=8000) # main.py (сервер)

import sys

import os

# Добавляем корень проекта в PYTHONPATH

current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

project\_root = os.path.dirname(os.path.dirname(current\_dir)) # Поднимаемся на уровень выше

sys.path.insert(0, project\_root)

from fastapi import FastAPI, Depends, HTTPException

from sqlalchemy.orm import Session

from shared.config\_loader import get\_api\_config

from . import database, models, crud, schemas

from typing import List, Optional

import uuid

from datetime import datetime

# Загружаем конфигурацию API

api\_config = get\_api\_config()

app = FastAPI(

title="File Monitoring API",

description="API для отслеживания изменений файлов с системой комментариев",

version="1.0.0"

)

# Dependency

def get\_db():

db = database.SessionLocal()

try:

yield db

finally:

db.close()

@app.get("/")

async def root():

return {

"message": "File Monitoring System API",

"status": "running",

"config\_source": "YAML"

}

@app.get("/health")

async def health\_check():

return {"status": "healthy", "service": "session-service"}

@app.get("/config")

async def show\_config():

"""Показывает текущую конфигурацию (без паролей)"""

return {

"api\_host": api\_config.get('host'),

"api\_port": api\_config.get('port'),

"environment": "development"

}

# Events endpoint для мониторинга

@app.post("/api/events")

async def create\_event(event\_data: dict, db: Session = Depends(get\_db)):

"""Эндпоинт для приема событий от мониторинга"""

print(f"📨 Received event: {event\_data}")

try:

# Обрабатываем событие в зависимости от типа

result = await process\_file\_event(db, event\_data)

return {"status": "processed", "event\_type": event\_data.get('event\_type'), "result": result}

except Exception as e:

print(f"❌ Error processing event: {e}")

raise HTTPException(status\_code=500, detail=f"Error processing event: {str(e)}")

async def process\_file\_event(db: Session, event\_data: dict):

"""Обрабатывает файловое событие и создает/обновляет сессии"""

event\_type = event\_data.get('event\_type')

file\_path = event\_data.get('file\_path')

file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

file\_hash = event\_data.get('file\_hash')

session\_id = event\_data.get('session\_id')

resume\_count = event\_data.get('resume\_count', 0)

print(f"🔧 Processing {event\_type} for {file\_path} (user: {username}, session: {session\_id}, resume: {resume\_count})")

# Обрабатываем перемещение отдельно

if event\_type == 'moved':

return await handle\_file\_moved(db, event\_data, file\_hash)

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# Обрабатываем в зависимости от типа события

if event\_type == 'created':

return await handle\_file\_created(db, user.id, file.id, event\_data, file\_hash, resume\_count)

elif event\_type == 'modified':

return await handle\_file\_modified(db, user.id, file.id, event\_data, file\_hash, session\_id, resume\_count)

elif event\_type == 'deleted':

return await handle\_file\_deleted(db, user.id, file.id, event\_data, session\_id)

elif event\_type == 'closed':

return await handle\_file\_closed(db, session\_id, file\_hash, event\_data)

else:

raise ValueError(f"Unknown event type: {event\_type}")

async def handle\_file\_moved(db: Session, event\_data: dict, file\_hash: str = None):

"""Обрабатывает перемещение/переименование файла"""

old\_file\_path = event\_data.get('old\_file\_path')

new\_file\_path = event\_data.get('file\_path')

new\_file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

print(f"🔄 Processing file move: {old\_file\_path} -> {new\_file\_path}")

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Ищем активную сессию для старого пути

old\_file = crud.get\_file\_by\_path(db, old\_file\_path)

if old\_file:

# Находим активные сессии для этого файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, old\_file.id)

if active\_sessions:

# Берем последнюю активную сессию

session = active\_sessions[0]

# Обновляем файл с новым путем

old\_file.file\_path = new\_file\_path

old\_file.file\_name = new\_file\_name

db.commit()

# Обновляем время активности сессии

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Moved session {session.id} from {old\_file\_path} to {new\_file\_path}")

return {"action": "session\_moved", "session\_id": str(session.id)}

# Если не нашли активную сессию, создаем новую

file\_data = schemas.FileCreate(file\_path=new\_file\_path, file\_name=new\_file\_name)

file = crud.create\_file(db, file\_data)

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"✅ Created new session {session.id} for moved file {new\_file\_path}")

return {"action": "session\_created", "session\_id": str(session.id)}

async def handle\_file\_created(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, resume\_count: int = 0):

"""Обрабатывает создание файла - создает новую сессию"""

# ПРОВЕРКА ДУБЛИРОВАНИЯ СЕССИЙ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

print(f"⚠️ Active session already exists for file, updating instead of creating new")

session = active\_sessions[0]

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_before = file\_hash

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

# Создаем событие created

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Updated existing session {session.id} for created file")

return {"action": "session\_updated", "session\_id": str(session.id), "resumed": False}

# Проверяем, есть ли недавно закрытая сессия для возобновления

recent\_session = None

if resume\_count > 0:

recent\_session = crud.get\_recent\_closed\_session(db, user\_id, file\_id)

if recent\_session:

# Возобновляем существующую сессию

recent\_session.ended\_at = None

recent\_session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

recent\_session.resume\_count = resume\_count

recent\_session.hash\_before = file\_hash

db.commit()

db.refresh(recent\_session)

print(f"🔄 Resumed session {recent\_session.id} (resume count: {resume\_count})")

session = recent\_session

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for file {event\_data.get('file\_path')}")

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": "session\_created", "session\_id": str(session.id), "resumed": recent\_session is not None}

async def handle\_file\_modified(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, session\_id: str = None, resume\_count: int = 0):

"""Обрабатывает изменение файла - обновляет существующую сессию или создает новую"""

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

if session:

# Обновляем существующую сессию

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

if resume\_count > session.resume\_count:

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

print(f"📝 Updated session {session.id} for file {event\_data.get('file\_path')} (resume: {resume\_count})")

action = "session\_updated"

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created new session {session.id} for modified file {event\_data.get('file\_path')}")

action = "session\_created"

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='modified',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": action, "session\_id": str(session.id)}

async def handle\_file\_deleted(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, session\_id: str = None):

"""Обрабатывает удаление файла - закрывает сессию и создает событие deleted"""

print(f"🗑️ Processing file deletion for file\_id: {file\_id}, session\_id: {session\_id}")

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

print(f"🔍 Found active session for deletion: {session.id}")

if session:

# Закрываем сессию с временем из события

session.ended\_at = datetime.fromisoformat(event\_data.get('event\_timestamp'))

db.commit()

db.refresh(session)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None, # Для удаленного файла хеша нет

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Closed session {session.id} for deleted file and created deleted event")

return {"action": "session\_closed", "session\_id": str(session.id)}

else:

print(f"⚠️ No active session found for deleted file, creating standalone deleted event")

# Если сессии нет, все равно создаем событие deleted

# Создаем временную сессию для привязки события

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

ended\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')), # Сразу закрываем

hash\_before=None,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Created standalone deleted event with session {session.id}")

return {"action": "deleted\_event\_created", "session\_id": str(session.id)}

async def handle\_file\_closed(db: Session, session\_id: str, file\_hash: str = None, event\_data: dict = None):

"""Обрабатывает закрытие файла - закрывает сессию с правильным ended\_at"""

if session\_id:

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

if session:

# ОБНОВЛЯЕМ СЕССИЮ С ПРАВИЛЬНЫМ ended\_at

if event\_data and 'event\_timestamp' in event\_data:

# Используем время из события закрытия

ended\_at = datetime.fromisoformat(event\_data['event\_timestamp'])

else:

# Используем текущее время

ended\_at = datetime.utcnow()

session.ended\_at = ended\_at

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие закрытия

event\_timestamp = ended\_at # Используем то же время что и для ended\_at

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='closed',

file\_hash=file\_hash,

event\_timestamp=event\_timestamp

)

crud.create\_file\_event(db, event\_record)

# Получаем информацию о файле для логов

file = crud.get\_file(db, session.file\_id)

user = crud.get\_user(db, session.user\_id)

duration = session.ended\_at - session.started\_at

print(f"🔒 Closed session {session.id} for file {file.file\_path} (user: {user.username}, duration: {duration}, ended\_at: {session.ended\_at})")

return {

"action": "session\_closed",

"session\_id": str(session.id),

"duration": str(duration),

"ended\_at": session.ended\_at.isoformat(),

"resume\_count": session.resume\_count

}

except ValueError:

pass

return {"action": "no\_session\_found"}

# Новый эндпоинт для создания сессий (для совместимости с мониторингом)

@app.post("/api/sessions")

async def create\_session(session\_data: dict, db: Session = Depends(get\_db)):

"""Создает сессию на сервере и возвращает session\_id"""

try:

print(f"📝 Creating session: {session\_data}")

# Получаем или создаем пользователя

username = session\_data.get('username')

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file\_path = session\_data.get('file\_path')

file\_name = session\_data.get('file\_name', os.path.basename(file\_path))

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# ПРОВЕРКА ДУБЛИРОВАНИЯ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, file.id)

if active\_sessions:

print(f"⚠️ Active session already exists, returning existing: {active\_sessions[0].id}")

return {"id": str(active\_sessions[0].id), "status": "existing"}

# Создаем сессию

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.utcnow(),

last\_activity=datetime.utcnow(),

hash\_before=session\_data.get('file\_hash'),

resume\_count=session\_data.get('resume\_count', 0)

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for {file\_path}")

return {"id": str(session.id), "status": "created"}

except Exception as e:

print(f"❌ Error creating session: {e}")

raise HTTPException(status\_code=500, detail=f"Error creating session: {str(e)}")

# Простые эндпоинты для проверки данных

@app.get("/api/users")

async def get\_users(db: Session = Depends(get\_db)):

"""Получает список пользователей"""

users = db.query(models.User).all()

return {"users": [{"id": str(user.id), "username": user.username} for user in users]}

@app.get("/api/files")

async def get\_files(db: Session = Depends(get\_db)):

"""Получает список файлов"""

files = db.query(models.File).all()

return {"files": [{"id": str(file.id), "path": file.file\_path, "name": file.file\_name} for file in files]}

@app.get("/api/sessions")

async def get\_sessions(db: Session = Depends(get\_db)):

"""Получает список сессий"""

sessions = db.query(models.FileSession).all()

return {"sessions": [

{

"id": str(session.id),

"user\_id": str(session.user\_id),

"file\_id": str(session.file\_id),

"started\_at": session.started\_at.isoformat(),

"ended\_at": session.ended\_at.isoformat() if session.ended\_at else None,

"last\_activity": session.last\_activity.isoformat(),

"resume\_count": session.resume\_count,

"is\_commented": session.is\_commented

} for session in sessions

]}

@app.get("/api/events")

async def get\_events(db: Session = Depends(get\_db)):

"""Получает список событий"""

events = db.query(models.FileEvent).all()

return {"events": [

{

"id": str(event.id),

"session\_id": str(event.session\_id),

"event\_type": event.event\_type,

"file\_hash": event.file\_hash,

"timestamp": event.event\_timestamp.isoformat()

} for event in events

]}

# НОВЫЕ ЭНДПОИНТЫ ДЛЯ КОММЕНТАРИЕВ

@app.post("/api/comments", response\_model=schemas.Comment)

async def create\_comment(comment: schemas.CommentCreate, db: Session = Depends(get\_db)):

"""Создает комментарий для сессии"""

try:

# Проверяем существование сессии

session = crud.get\_file\_session(db, comment.session\_id)

if not session:

raise HTTPException(status\_code=404, detail="Session not found")

# Проверяем существование пользователя

user = crud.get\_user(db, comment.user\_id)

if not user:

raise HTTPException(status\_code=404, detail="User not found")

# Проверяем, нет ли уже комментария для этой сессии

existing\_comment = crud.get\_comment\_by\_session(db, comment.session\_id)

if existing\_comment:

raise HTTPException(status\_code=400, detail="Comment already exists for this session")

session.ended\_at = datetime.utcnow()

session.is\_commented = True

# Создаем комментарий

db\_comment = crud.create\_comment(db, comment)

db.commit()

return db\_comment

except HTTPException:

raise

except Exception as e:

db.rollback()

raise HTTPException(status\_code=500, detail=f"Error creating comment: {str(e)}")

@app.get("/api/comments", response\_model=List[schemas.CommentWithUser])

async def get\_comments(

skip: int = 0,

limit: int = 100,

change\_type: Optional[str] = None,

db: Session = Depends(get\_db)

):

"""Получает список комментариев с возможностью фильтрации по типу изменения"""

try:

# Базовый запрос с join пользователя

query = db.query(

models.Comment,

models.User.username

).join(

models.User, models.Comment.user\_id == models.User.id

)

# Фильтрация по типу изменения если указан

if change\_type:

query = query.filter(models.Comment.change\_type == change\_type)

comments = query.offset(skip).limit(limit).all()

# Преобразуем в схему

result = []

for comment, username in comments:

result.append(schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

))

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comments: {str(e)}")

@app.get("/api/comments/{session\_id}", response\_model=schemas.CommentWithUser)

async def get\_comment\_by\_session(session\_id: str, db: Session = Depends(get\_db)):

"""Получает комментарий по ID сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_comment\_with\_user(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Comment not found")

comment, username = result

return schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comment: {str(e)}")

@app.get("/api/sessions-with-comments", response\_model=List[schemas.SessionWithDetails])

async def get\_sessions\_with\_comments(

skip: int = 0,

limit: int = 100,

db: Session = Depends(get\_db)

):

"""Получает сессии с комментариями"""

try:

sessions = crud.get\_sessions\_with\_comments(db, skip=skip, limit=limit)

result = []

for session, file\_path, file\_name, username, comment in sessions:

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

result.append(session\_details)

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting sessions: {str(e)}")

@app.get("/api/change-types")

async def get\_change\_types():

"""Возвращает доступные типы изменений"""

return {

"change\_types": [

"technical\_changes", # Изменение технических решений

"design\_changes", # Изменение оформления/дизайна

"content\_changes", # Изменение содержания

"bug\_fixes", # Исправление ошибок

"optimization", # Оптимизация

"refactoring", # Рефакторинг

"new\_feature", # Новая функциональность

"documentation", # Изменение документации

"other" # Другое

]

}

@app.get("/api/sessions/{session\_id}/details", response\_model=schemas.SessionWithDetails)

async def get\_session\_details(session\_id: str, db: Session = Depends(get\_db)):

"""Получает детальную информацию о сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_session\_with\_details(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Session not found")

session, file\_path, file\_name, username, comment = result

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

return session\_details

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting session details: {str(e)}")

if \_\_name\_\_ == "\_\_main\_\_":

import uvicorn

uvicorn.run(app, host="0.0.0.0", port=8000) # main.py (сервер)

import sys

import os

# Добавляем корень проекта в PYTHONPATH

current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

project\_root = os.path.dirname(os.path.dirname(current\_dir)) # Поднимаемся на уровень выше

sys.path.insert(0, project\_root)

from fastapi import FastAPI, Depends, HTTPException

from sqlalchemy.orm import Session

from shared.config\_loader import get\_api\_config

from . import database, models, crud, schemas

from typing import List, Optional

import uuid

from datetime import datetime

# Загружаем конфигурацию API

api\_config = get\_api\_config()

app = FastAPI(

title="File Monitoring API",

description="API для отслеживания изменений файлов с системой комментариев",

version="1.0.0"

)

# Dependency

def get\_db():

db = database.SessionLocal()

try:

yield db

finally:

db.close()

@app.get("/")

async def root():

return {

"message": "File Monitoring System API",

"status": "running",

"config\_source": "YAML"

}

@app.get("/health")

async def health\_check():

return {"status": "healthy", "service": "session-service"}

@app.get("/config")

async def show\_config():

"""Показывает текущую конфигурацию (без паролей)"""

return {

"api\_host": api\_config.get('host'),

"api\_port": api\_config.get('port'),

"environment": "development"

}

# Events endpoint для мониторинга

@app.post("/api/events")

async def create\_event(event\_data: dict, db: Session = Depends(get\_db)):

"""Эндпоинт для приема событий от мониторинга"""

print(f"📨 Received event: {event\_data}")

try:

# Обрабатываем событие в зависимости от типа

result = await process\_file\_event(db, event\_data)

return {"status": "processed", "event\_type": event\_data.get('event\_type'), "result": result}

except Exception as e:

print(f"❌ Error processing event: {e}")

raise HTTPException(status\_code=500, detail=f"Error processing event: {str(e)}")

async def process\_file\_event(db: Session, event\_data: dict):

"""Обрабатывает файловое событие и создает/обновляет сессии"""

event\_type = event\_data.get('event\_type')

file\_path = event\_data.get('file\_path')

file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

file\_hash = event\_data.get('file\_hash')

session\_id = event\_data.get('session\_id')

resume\_count = event\_data.get('resume\_count', 0)

print(f"🔧 Processing {event\_type} for {file\_path} (user: {username}, session: {session\_id}, resume: {resume\_count})")

# Обрабатываем перемещение отдельно

if event\_type == 'moved':

return await handle\_file\_moved(db, event\_data, file\_hash)

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# Обрабатываем в зависимости от типа события

if event\_type == 'created':

return await handle\_file\_created(db, user.id, file.id, event\_data, file\_hash, resume\_count)

elif event\_type == 'modified':

return await handle\_file\_modified(db, user.id, file.id, event\_data, file\_hash, session\_id, resume\_count)

elif event\_type == 'deleted':

return await handle\_file\_deleted(db, user.id, file.id, event\_data, session\_id)

elif event\_type == 'closed':

return await handle\_file\_closed(db, session\_id, file\_hash, event\_data)

else:

raise ValueError(f"Unknown event type: {event\_type}")

async def handle\_file\_moved(db: Session, event\_data: dict, file\_hash: str = None):

"""Обрабатывает перемещение/переименование файла"""

old\_file\_path = event\_data.get('old\_file\_path')

new\_file\_path = event\_data.get('file\_path')

new\_file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

print(f"🔄 Processing file move: {old\_file\_path} -> {new\_file\_path}")

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Ищем активную сессию для старого пути

old\_file = crud.get\_file\_by\_path(db, old\_file\_path)

if old\_file:

# Находим активные сессии для этого файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, old\_file.id)

if active\_sessions:

# Берем последнюю активную сессию

session = active\_sessions[0]

# Обновляем файл с новым путем

old\_file.file\_path = new\_file\_path

old\_file.file\_name = new\_file\_name

db.commit()

# Обновляем время активности сессии

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Moved session {session.id} from {old\_file\_path} to {new\_file\_path}")

return {"action": "session\_moved", "session\_id": str(session.id)}

# Если не нашли активную сессию, создаем новую

file\_data = schemas.FileCreate(file\_path=new\_file\_path, file\_name=new\_file\_name)

file = crud.create\_file(db, file\_data)

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"✅ Created new session {session.id} for moved file {new\_file\_path}")

return {"action": "session\_created", "session\_id": str(session.id)}

async def handle\_file\_created(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, resume\_count: int = 0):

"""Обрабатывает создание файла - создает новую сессию"""

# ПРОВЕРКА ДУБЛИРОВАНИЯ СЕССИЙ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

print(f"⚠️ Active session already exists for file, updating instead of creating new")

session = active\_sessions[0]

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_before = file\_hash

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

# Создаем событие created

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Updated existing session {session.id} for created file")

return {"action": "session\_updated", "session\_id": str(session.id), "resumed": False}

# Проверяем, есть ли недавно закрытая сессия для возобновления

recent\_session = None

if resume\_count > 0:

recent\_session = crud.get\_recent\_closed\_session(db, user\_id, file\_id)

if recent\_session:

# Возобновляем существующую сессию

recent\_session.ended\_at = None

recent\_session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

recent\_session.resume\_count = resume\_count

recent\_session.hash\_before = file\_hash

db.commit()

db.refresh(recent\_session)

print(f"🔄 Resumed session {recent\_session.id} (resume count: {resume\_count})")

session = recent\_session

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for file {event\_data.get('file\_path')}")

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": "session\_created", "session\_id": str(session.id), "resumed": recent\_session is not None}

async def handle\_file\_modified(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, session\_id: str = None, resume\_count: int = 0):

"""Обрабатывает изменение файла - обновляет существующую сессию или создает новую"""

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

if session:

# Обновляем существующую сессию

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

if resume\_count > session.resume\_count:

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

print(f"📝 Updated session {session.id} for file {event\_data.get('file\_path')} (resume: {resume\_count})")

action = "session\_updated"

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created new session {session.id} for modified file {event\_data.get('file\_path')}")

action = "session\_created"

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='modified',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": action, "session\_id": str(session.id)}

async def handle\_file\_deleted(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, session\_id: str = None):

"""Обрабатывает удаление файла - закрывает сессию и создает событие deleted"""

print(f"🗑️ Processing file deletion for file\_id: {file\_id}, session\_id: {session\_id}")

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

print(f"🔍 Found active session for deletion: {session.id}")

if session:

# Закрываем сессию с временем из события

session.ended\_at = datetime.fromisoformat(event\_data.get('event\_timestamp'))

db.commit()

db.refresh(session)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None, # Для удаленного файла хеша нет

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Closed session {session.id} for deleted file and created deleted event")

return {"action": "session\_closed", "session\_id": str(session.id)}

else:

print(f"⚠️ No active session found for deleted file, creating standalone deleted event")

# Если сессии нет, все равно создаем событие deleted

# Создаем временную сессию для привязки события

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

ended\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')), # Сразу закрываем

hash\_before=None,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Created standalone deleted event with session {session.id}")

return {"action": "deleted\_event\_created", "session\_id": str(session.id)}

async def handle\_file\_closed(db: Session, session\_id: str, file\_hash: str = None, event\_data: dict = None):

"""Обрабатывает закрытие файла - закрывает сессию с правильным ended\_at"""

if session\_id:

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

if session:

# ОБНОВЛЯЕМ СЕССИЮ С ПРАВИЛЬНЫМ ended\_at

if event\_data and 'event\_timestamp' in event\_data:

# Используем время из события закрытия

ended\_at = datetime.fromisoformat(event\_data['event\_timestamp'])

else:

# Используем текущее время

ended\_at = datetime.utcnow()

session.ended\_at = ended\_at

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие закрытия

event\_timestamp = ended\_at # Используем то же время что и для ended\_at

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='closed',

file\_hash=file\_hash,

event\_timestamp=event\_timestamp

)

crud.create\_file\_event(db, event\_record)

# Получаем информацию о файле для логов

file = crud.get\_file(db, session.file\_id)

user = crud.get\_user(db, session.user\_id)

duration = session.ended\_at - session.started\_at

print(f"🔒 Closed session {session.id} for file {file.file\_path} (user: {user.username}, duration: {duration}, ended\_at: {session.ended\_at})")

return {

"action": "session\_closed",

"session\_id": str(session.id),

"duration": str(duration),

"ended\_at": session.ended\_at.isoformat(),

"resume\_count": session.resume\_count

}

except ValueError:

pass

return {"action": "no\_session\_found"}

# Новый эндпоинт для создания сессий (для совместимости с мониторингом)

@app.post("/api/sessions")

async def create\_session(session\_data: dict, db: Session = Depends(get\_db)):

"""Создает сессию на сервере и возвращает session\_id"""

try:

print(f"📝 Creating session: {session\_data}")

# Получаем или создаем пользователя

username = session\_data.get('username')

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file\_path = session\_data.get('file\_path')

file\_name = session\_data.get('file\_name', os.path.basename(file\_path))

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# ПРОВЕРКА ДУБЛИРОВАНИЯ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, file.id)

if active\_sessions:

print(f"⚠️ Active session already exists, returning existing: {active\_sessions[0].id}")

return {"id": str(active\_sessions[0].id), "status": "existing"}

# Создаем сессию

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.utcnow(),

last\_activity=datetime.utcnow(),

hash\_before=session\_data.get('file\_hash'),

resume\_count=session\_data.get('resume\_count', 0)

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for {file\_path}")

return {"id": str(session.id), "status": "created"}

except Exception as e:

print(f"❌ Error creating session: {e}")

raise HTTPException(status\_code=500, detail=f"Error creating session: {str(e)}")

# Простые эндпоинты для проверки данных

@app.get("/api/users")

async def get\_users(db: Session = Depends(get\_db)):

"""Получает список пользователей"""

users = db.query(models.User).all()

return {"users": [{"id": str(user.id), "username": user.username} for user in users]}

@app.get("/api/files")

async def get\_files(db: Session = Depends(get\_db)):

"""Получает список файлов"""

files = db.query(models.File).all()

return {"files": [{"id": str(file.id), "path": file.file\_path, "name": file.file\_name} for file in files]}

@app.get("/api/sessions")

async def get\_sessions(db: Session = Depends(get\_db)):

"""Получает список сессий"""

sessions = db.query(models.FileSession).all()

return {"sessions": [

{

"id": str(session.id),

"user\_id": str(session.user\_id),

"file\_id": str(session.file\_id),

"started\_at": session.started\_at.isoformat(),

"ended\_at": session.ended\_at.isoformat() if session.ended\_at else None,

"last\_activity": session.last\_activity.isoformat(),

"resume\_count": session.resume\_count,

"is\_commented": session.is\_commented

} for session in sessions

]}

@app.get("/api/events")

async def get\_events(db: Session = Depends(get\_db)):

"""Получает список событий"""

events = db.query(models.FileEvent).all()

return {"events": [

{

"id": str(event.id),

"session\_id": str(event.session\_id),

"event\_type": event.event\_type,

"file\_hash": event.file\_hash,

"timestamp": event.event\_timestamp.isoformat()

} for event in events

]}

# НОВЫЕ ЭНДПОИНТЫ ДЛЯ КОММЕНТАРИЕВ

@app.post("/api/comments", response\_model=schemas.Comment)

async def create\_comment(comment: schemas.CommentCreate, db: Session = Depends(get\_db)):

"""Создает комментарий для сессии"""

try:

# Проверяем существование сессии

session = crud.get\_file\_session(db, comment.session\_id)

if not session:

raise HTTPException(status\_code=404, detail="Session not found")

# Проверяем существование пользователя

user = crud.get\_user(db, comment.user\_id)

if not user:

raise HTTPException(status\_code=404, detail="User not found")

# Проверяем, нет ли уже комментария для этой сессии

existing\_comment = crud.get\_comment\_by\_session(db, comment.session\_id)

if existing\_comment:

raise HTTPException(status\_code=400, detail="Comment already exists for this session")

session.ended\_at = datetime.utcnow()

session.is\_commented = True

# Создаем комментарий

db\_comment = crud.create\_comment(db, comment)

db.commit()

return db\_comment

except HTTPException:

raise

except Exception as e:

db.rollback()

raise HTTPException(status\_code=500, detail=f"Error creating comment: {str(e)}")

@app.get("/api/comments", response\_model=List[schemas.CommentWithUser])

async def get\_comments(

skip: int = 0,

limit: int = 100,

change\_type: Optional[str] = None,

db: Session = Depends(get\_db)

):

"""Получает список комментариев с возможностью фильтрации по типу изменения"""

try:

# Базовый запрос с join пользователя

query = db.query(

models.Comment,

models.User.username

).join(

models.User, models.Comment.user\_id == models.User.id

)

# Фильтрация по типу изменения если указан

if change\_type:

query = query.filter(models.Comment.change\_type == change\_type)

comments = query.offset(skip).limit(limit).all()

# Преобразуем в схему

result = []

for comment, username in comments:

result.append(schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

))

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comments: {str(e)}")

@app.get("/api/comments/{session\_id}", response\_model=schemas.CommentWithUser)

async def get\_comment\_by\_session(session\_id: str, db: Session = Depends(get\_db)):

"""Получает комментарий по ID сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_comment\_with\_user(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Comment not found")

comment, username = result

return schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comment: {str(e)}")

@app.get("/api/sessions-with-comments", response\_model=List[schemas.SessionWithDetails])

async def get\_sessions\_with\_comments(

skip: int = 0,

limit: int = 100,

db: Session = Depends(get\_db)

):

"""Получает сессии с комментариями"""

try:

sessions = crud.get\_sessions\_with\_comments(db, skip=skip, limit=limit)

result = []

for session, file\_path, file\_name, username, comment in sessions:

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

result.append(session\_details)

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting sessions: {str(e)}")

@app.get("/api/change-types")

async def get\_change\_types():

"""Возвращает доступные типы изменений"""

return {

"change\_types": [

"technical\_changes", # Изменение технических решений

"design\_changes", # Изменение оформления/дизайна

"content\_changes", # Изменение содержания

"bug\_fixes", # Исправление ошибок

"optimization", # Оптимизация

"refactoring", # Рефакторинг

"new\_feature", # Новая функциональность

"documentation", # Изменение документации

"other" # Другое

]

}

@app.get("/api/sessions/{session\_id}/details", response\_model=schemas.SessionWithDetails)

async def get\_session\_details(session\_id: str, db: Session = Depends(get\_db)):

"""Получает детальную информацию о сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_session\_with\_details(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Session not found")

session, file\_path, file\_name, username, comment = result

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

return session\_details

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting session details: {str(e)}")

if \_\_name\_\_ == "\_\_main\_\_":

import uvicorn

uvicorn.run(app, host="0.0.0.0", port=8000) # main.py (сервер)

import sys

import os

# Добавляем корень проекта в PYTHONPATH

current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

project\_root = os.path.dirname(os.path.dirname(current\_dir)) # Поднимаемся на уровень выше

sys.path.insert(0, project\_root)

from fastapi import FastAPI, Depends, HTTPException

from sqlalchemy.orm import Session

from shared.config\_loader import get\_api\_config

from . import database, models, crud, schemas

from typing import List, Optional

import uuid

from datetime import datetime

# Загружаем конфигурацию API

api\_config = get\_api\_config()

app = FastAPI(

title="File Monitoring API",

description="API для отслеживания изменений файлов с системой комментариев",

version="1.0.0"

)

# Dependency

def get\_db():

db = database.SessionLocal()

try:

yield db

finally:

db.close()

@app.get("/")

async def root():

return {

"message": "File Monitoring System API",

"status": "running",

"config\_source": "YAML"

}

@app.get("/health")

async def health\_check():

return {"status": "healthy", "service": "session-service"}

@app.get("/config")

async def show\_config():

"""Показывает текущую конфигурацию (без паролей)"""

return {

"api\_host": api\_config.get('host'),

"api\_port": api\_config.get('port'),

"environment": "development"

}

# Events endpoint для мониторинга

@app.post("/api/events")

async def create\_event(event\_data: dict, db: Session = Depends(get\_db)):

"""Эндпоинт для приема событий от мониторинга"""

print(f"📨 Received event: {event\_data}")

try:

# Обрабатываем событие в зависимости от типа

result = await process\_file\_event(db, event\_data)

return {"status": "processed", "event\_type": event\_data.get('event\_type'), "result": result}

except Exception as e:

print(f"❌ Error processing event: {e}")

raise HTTPException(status\_code=500, detail=f"Error processing event: {str(e)}")

async def process\_file\_event(db: Session, event\_data: dict):

"""Обрабатывает файловое событие и создает/обновляет сессии"""

event\_type = event\_data.get('event\_type')

file\_path = event\_data.get('file\_path')

file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

file\_hash = event\_data.get('file\_hash')

session\_id = event\_data.get('session\_id')

resume\_count = event\_data.get('resume\_count', 0)

print(f"🔧 Processing {event\_type} for {file\_path} (user: {username}, session: {session\_id}, resume: {resume\_count})")

# Обрабатываем перемещение отдельно

if event\_type == 'moved':

return await handle\_file\_moved(db, event\_data, file\_hash)

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# Обрабатываем в зависимости от типа события

if event\_type == 'created':

return await handle\_file\_created(db, user.id, file.id, event\_data, file\_hash, resume\_count)

elif event\_type == 'modified':

return await handle\_file\_modified(db, user.id, file.id, event\_data, file\_hash, session\_id, resume\_count)

elif event\_type == 'deleted':

return await handle\_file\_deleted(db, user.id, file.id, event\_data, session\_id)

elif event\_type == 'closed':

return await handle\_file\_closed(db, session\_id, file\_hash, event\_data)

else:

raise ValueError(f"Unknown event type: {event\_type}")

async def handle\_file\_moved(db: Session, event\_data: dict, file\_hash: str = None):

"""Обрабатывает перемещение/переименование файла"""

old\_file\_path = event\_data.get('old\_file\_path')

new\_file\_path = event\_data.get('file\_path')

new\_file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

print(f"🔄 Processing file move: {old\_file\_path} -> {new\_file\_path}")

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Ищем активную сессию для старого пути

old\_file = crud.get\_file\_by\_path(db, old\_file\_path)

if old\_file:

# Находим активные сессии для этого файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, old\_file.id)

if active\_sessions:

# Берем последнюю активную сессию

session = active\_sessions[0]

# Обновляем файл с новым путем

old\_file.file\_path = new\_file\_path

old\_file.file\_name = new\_file\_name

db.commit()

# Обновляем время активности сессии

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Moved session {session.id} from {old\_file\_path} to {new\_file\_path}")

return {"action": "session\_moved", "session\_id": str(session.id)}

# Если не нашли активную сессию, создаем новую

file\_data = schemas.FileCreate(file\_path=new\_file\_path, file\_name=new\_file\_name)

file = crud.create\_file(db, file\_data)

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"✅ Created new session {session.id} for moved file {new\_file\_path}")

return {"action": "session\_created", "session\_id": str(session.id)}

async def handle\_file\_created(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, resume\_count: int = 0):

"""Обрабатывает создание файла - создает новую сессию"""

# ПРОВЕРКА ДУБЛИРОВАНИЯ СЕССИЙ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

print(f"⚠️ Active session already exists for file, updating instead of creating new")

session = active\_sessions[0]

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_before = file\_hash

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

# Создаем событие created

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Updated existing session {session.id} for created file")

return {"action": "session\_updated", "session\_id": str(session.id), "resumed": False}

# Проверяем, есть ли недавно закрытая сессия для возобновления

recent\_session = None

if resume\_count > 0:

recent\_session = crud.get\_recent\_closed\_session(db, user\_id, file\_id)

if recent\_session:

# Возобновляем существующую сессию

recent\_session.ended\_at = None

recent\_session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

recent\_session.resume\_count = resume\_count

recent\_session.hash\_before = file\_hash

db.commit()

db.refresh(recent\_session)

print(f"🔄 Resumed session {recent\_session.id} (resume count: {resume\_count})")

session = recent\_session

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for file {event\_data.get('file\_path')}")

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": "session\_created", "session\_id": str(session.id), "resumed": recent\_session is not None}

async def handle\_file\_modified(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, session\_id: str = None, resume\_count: int = 0):

"""Обрабатывает изменение файла - обновляет существующую сессию или создает новую"""

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

if session:

# Обновляем существующую сессию

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

if resume\_count > session.resume\_count:

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

print(f"📝 Updated session {session.id} for file {event\_data.get('file\_path')} (resume: {resume\_count})")

action = "session\_updated"

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created new session {session.id} for modified file {event\_data.get('file\_path')}")

action = "session\_created"

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='modified',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": action, "session\_id": str(session.id)}

async def handle\_file\_deleted(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, session\_id: str = None):

"""Обрабатывает удаление файла - закрывает сессию и создает событие deleted"""

print(f"🗑️ Processing file deletion for file\_id: {file\_id}, session\_id: {session\_id}")

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

print(f"🔍 Found active session for deletion: {session.id}")

if session:

# Закрываем сессию с временем из события

session.ended\_at = datetime.fromisoformat(event\_data.get('event\_timestamp'))

db.commit()

db.refresh(session)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None, # Для удаленного файла хеша нет

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Closed session {session.id} for deleted file and created deleted event")

return {"action": "session\_closed", "session\_id": str(session.id)}

else:

print(f"⚠️ No active session found for deleted file, creating standalone deleted event")

# Если сессии нет, все равно создаем событие deleted

# Создаем временную сессию для привязки события

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

ended\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')), # Сразу закрываем

hash\_before=None,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Created standalone deleted event with session {session.id}")

return {"action": "deleted\_event\_created", "session\_id": str(session.id)}

async def handle\_file\_closed(db: Session, session\_id: str, file\_hash: str = None, event\_data: dict = None):

"""Обрабатывает закрытие файла - закрывает сессию с правильным ended\_at"""

if session\_id:

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

if session:

# ОБНОВЛЯЕМ СЕССИЮ С ПРАВИЛЬНЫМ ended\_at

if event\_data and 'event\_timestamp' in event\_data:

# Используем время из события закрытия

ended\_at = datetime.fromisoformat(event\_data['event\_timestamp'])

else:

# Используем текущее время

ended\_at = datetime.utcnow()

session.ended\_at = ended\_at

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие закрытия

event\_timestamp = ended\_at # Используем то же время что и для ended\_at

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='closed',

file\_hash=file\_hash,

event\_timestamp=event\_timestamp

)

crud.create\_file\_event(db, event\_record)

# Получаем информацию о файле для логов

file = crud.get\_file(db, session.file\_id)

user = crud.get\_user(db, session.user\_id)

duration = session.ended\_at - session.started\_at

print(f"🔒 Closed session {session.id} for file {file.file\_path} (user: {user.username}, duration: {duration}, ended\_at: {session.ended\_at})")

return {

"action": "session\_closed",

"session\_id": str(session.id),

"duration": str(duration),

"ended\_at": session.ended\_at.isoformat(),

"resume\_count": session.resume\_count

}

except ValueError:

pass

return {"action": "no\_session\_found"}

# Новый эндпоинт для создания сессий (для совместимости с мониторингом)

@app.post("/api/sessions")

async def create\_session(session\_data: dict, db: Session = Depends(get\_db)):

"""Создает сессию на сервере и возвращает session\_id"""

try:

print(f"📝 Creating session: {session\_data}")

# Получаем или создаем пользователя

username = session\_data.get('username')

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file\_path = session\_data.get('file\_path')

file\_name = session\_data.get('file\_name', os.path.basename(file\_path))

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# ПРОВЕРКА ДУБЛИРОВАНИЯ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, file.id)

if active\_sessions:

print(f"⚠️ Active session already exists, returning existing: {active\_sessions[0].id}")

return {"id": str(active\_sessions[0].id), "status": "existing"}

# Создаем сессию

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.utcnow(),

last\_activity=datetime.utcnow(),

hash\_before=session\_data.get('file\_hash'),

resume\_count=session\_data.get('resume\_count', 0)

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for {file\_path}")

return {"id": str(session.id), "status": "created"}

except Exception as e:

print(f"❌ Error creating session: {e}")

raise HTTPException(status\_code=500, detail=f"Error creating session: {str(e)}")

# Простые эндпоинты для проверки данных

@app.get("/api/users")

async def get\_users(db: Session = Depends(get\_db)):

"""Получает список пользователей"""

users = db.query(models.User).all()

return {"users": [{"id": str(user.id), "username": user.username} for user in users]}

@app.get("/api/files")

async def get\_files(db: Session = Depends(get\_db)):

"""Получает список файлов"""

files = db.query(models.File).all()

return {"files": [{"id": str(file.id), "path": file.file\_path, "name": file.file\_name} for file in files]}

@app.get("/api/sessions")

async def get\_sessions(db: Session = Depends(get\_db)):

"""Получает список сессий"""

sessions = db.query(models.FileSession).all()

return {"sessions": [

{

"id": str(session.id),

"user\_id": str(session.user\_id),

"file\_id": str(session.file\_id),

"started\_at": session.started\_at.isoformat(),

"ended\_at": session.ended\_at.isoformat() if session.ended\_at else None,

"last\_activity": session.last\_activity.isoformat(),

"resume\_count": session.resume\_count,

"is\_commented": session.is\_commented

} for session in sessions

]}

@app.get("/api/events")

async def get\_events(db: Session = Depends(get\_db)):

"""Получает список событий"""

events = db.query(models.FileEvent).all()

return {"events": [

{

"id": str(event.id),

"session\_id": str(event.session\_id),

"event\_type": event.event\_type,

"file\_hash": event.file\_hash,

"timestamp": event.event\_timestamp.isoformat()

} for event in events

]}

# НОВЫЕ ЭНДПОИНТЫ ДЛЯ КОММЕНТАРИЕВ

@app.post("/api/comments", response\_model=schemas.Comment)

async def create\_comment(comment: schemas.CommentCreate, db: Session = Depends(get\_db)):

"""Создает комментарий для сессии"""

try:

# Проверяем существование сессии

session = crud.get\_file\_session(db, comment.session\_id)

if not session:

raise HTTPException(status\_code=404, detail="Session not found")

# Проверяем существование пользователя

user = crud.get\_user(db, comment.user\_id)

if not user:

raise HTTPException(status\_code=404, detail="User not found")

# Проверяем, нет ли уже комментария для этой сессии

existing\_comment = crud.get\_comment\_by\_session(db, comment.session\_id)

if existing\_comment:

raise HTTPException(status\_code=400, detail="Comment already exists for this session")

session.ended\_at = datetime.utcnow()

session.is\_commented = True

# Создаем комментарий

db\_comment = crud.create\_comment(db, comment)

db.commit()

return db\_comment

except HTTPException:

raise

except Exception as e:

db.rollback()

raise HTTPException(status\_code=500, detail=f"Error creating comment: {str(e)}")

@app.get("/api/comments", response\_model=List[schemas.CommentWithUser])

async def get\_comments(

skip: int = 0,

limit: int = 100,

change\_type: Optional[str] = None,

db: Session = Depends(get\_db)

):

"""Получает список комментариев с возможностью фильтрации по типу изменения"""

try:

# Базовый запрос с join пользователя

query = db.query(

models.Comment,

models.User.username

).join(

models.User, models.Comment.user\_id == models.User.id

)

# Фильтрация по типу изменения если указан

if change\_type:

query = query.filter(models.Comment.change\_type == change\_type)

comments = query.offset(skip).limit(limit).all()

# Преобразуем в схему

result = []

for comment, username in comments:

result.append(schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

))

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comments: {str(e)}")

@app.get("/api/comments/{session\_id}", response\_model=schemas.CommentWithUser)

async def get\_comment\_by\_session(session\_id: str, db: Session = Depends(get\_db)):

"""Получает комментарий по ID сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_comment\_with\_user(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Comment not found")

comment, username = result

return schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comment: {str(e)}")

@app.get("/api/sessions-with-comments", response\_model=List[schemas.SessionWithDetails])

async def get\_sessions\_with\_comments(

skip: int = 0,

limit: int = 100,

db: Session = Depends(get\_db)

):

"""Получает сессии с комментариями"""

try:

sessions = crud.get\_sessions\_with\_comments(db, skip=skip, limit=limit)

result = []

for session, file\_path, file\_name, username, comment in sessions:

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

result.append(session\_details)

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting sessions: {str(e)}")

@app.get("/api/change-types")

async def get\_change\_types():

"""Возвращает доступные типы изменений"""

return {

"change\_types": [

"technical\_changes", # Изменение технических решений

"design\_changes", # Изменение оформления/дизайна

"content\_changes", # Изменение содержания

"bug\_fixes", # Исправление ошибок

"optimization", # Оптимизация

"refactoring", # Рефакторинг

"new\_feature", # Новая функциональность

"documentation", # Изменение документации

"other" # Другое

]

}

@app.get("/api/sessions/{session\_id}/details", response\_model=schemas.SessionWithDetails)

async def get\_session\_details(session\_id: str, db: Session = Depends(get\_db)):

"""Получает детальную информацию о сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_session\_with\_details(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Session not found")

session, file\_path, file\_name, username, comment = result

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

return session\_details

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting session details: {str(e)}")

if \_\_name\_\_ == "\_\_main\_\_":

import uvicorn

uvicorn.run(app, host="0.0.0.0", port=8000) # main.py (сервер)

import sys

import os

# Добавляем корень проекта в PYTHONPATH

current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

project\_root = os.path.dirname(os.path.dirname(current\_dir)) # Поднимаемся на уровень выше

sys.path.insert(0, project\_root)

from fastapi import FastAPI, Depends, HTTPException

from sqlalchemy.orm import Session

from shared.config\_loader import get\_api\_config

from . import database, models, crud, schemas

from typing import List, Optional

import uuid

from datetime import datetime

# Загружаем конфигурацию API

api\_config = get\_api\_config()

app = FastAPI(

title="File Monitoring API",

description="API для отслеживания изменений файлов с системой комментариев",

version="1.0.0"

)

# Dependency

def get\_db():

db = database.SessionLocal()

try:

yield db

finally:

db.close()

@app.get("/")

async def root():

return {

"message": "File Monitoring System API",

"status": "running",

"config\_source": "YAML"

}

@app.get("/health")

async def health\_check():

return {"status": "healthy", "service": "session-service"}

@app.get("/config")

async def show\_config():

"""Показывает текущую конфигурацию (без паролей)"""

return {

"api\_host": api\_config.get('host'),

"api\_port": api\_config.get('port'),

"environment": "development"

}

# Events endpoint для мониторинга

@app.post("/api/events")

async def create\_event(event\_data: dict, db: Session = Depends(get\_db)):

"""Эндпоинт для приема событий от мониторинга"""

print(f"📨 Received event: {event\_data}")

try:

# Обрабатываем событие в зависимости от типа

result = await process\_file\_event(db, event\_data)

return {"status": "processed", "event\_type": event\_data.get('event\_type'), "result": result}

except Exception as e:

print(f"❌ Error processing event: {e}")

raise HTTPException(status\_code=500, detail=f"Error processing event: {str(e)}")

async def process\_file\_event(db: Session, event\_data: dict):

"""Обрабатывает файловое событие и создает/обновляет сессии"""

event\_type = event\_data.get('event\_type')

file\_path = event\_data.get('file\_path')

file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

file\_hash = event\_data.get('file\_hash')

session\_id = event\_data.get('session\_id')

resume\_count = event\_data.get('resume\_count', 0)

print(f"🔧 Processing {event\_type} for {file\_path} (user: {username}, session: {session\_id}, resume: {resume\_count})")

# Обрабатываем перемещение отдельно

if event\_type == 'moved':

return await handle\_file\_moved(db, event\_data, file\_hash)

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# Обрабатываем в зависимости от типа события

if event\_type == 'created':

return await handle\_file\_created(db, user.id, file.id, event\_data, file\_hash, resume\_count)

elif event\_type == 'modified':

return await handle\_file\_modified(db, user.id, file.id, event\_data, file\_hash, session\_id, resume\_count)

elif event\_type == 'deleted':

return await handle\_file\_deleted(db, user.id, file.id, event\_data, session\_id)

elif event\_type == 'closed':

return await handle\_file\_closed(db, session\_id, file\_hash, event\_data)

else:

raise ValueError(f"Unknown event type: {event\_type}")

async def handle\_file\_moved(db: Session, event\_data: dict, file\_hash: str = None):

"""Обрабатывает перемещение/переименование файла"""

old\_file\_path = event\_data.get('old\_file\_path')

new\_file\_path = event\_data.get('file\_path')

new\_file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

print(f"🔄 Processing file move: {old\_file\_path} -> {new\_file\_path}")

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Ищем активную сессию для старого пути

old\_file = crud.get\_file\_by\_path(db, old\_file\_path)

if old\_file:

# Находим активные сессии для этого файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, old\_file.id)

if active\_sessions:

# Берем последнюю активную сессию

session = active\_sessions[0]

# Обновляем файл с новым путем

old\_file.file\_path = new\_file\_path

old\_file.file\_name = new\_file\_name

db.commit()

# Обновляем время активности сессии

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Moved session {session.id} from {old\_file\_path} to {new\_file\_path}")

return {"action": "session\_moved", "session\_id": str(session.id)}

# Если не нашли активную сессию, создаем новую

file\_data = schemas.FileCreate(file\_path=new\_file\_path, file\_name=new\_file\_name)

file = crud.create\_file(db, file\_data)

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"✅ Created new session {session.id} for moved file {new\_file\_path}")

return {"action": "session\_created", "session\_id": str(session.id)}

async def handle\_file\_created(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, resume\_count: int = 0):

"""Обрабатывает создание файла - создает новую сессию"""

# ПРОВЕРКА ДУБЛИРОВАНИЯ СЕССИЙ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

print(f"⚠️ Active session already exists for file, updating instead of creating new")

session = active\_sessions[0]

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_before = file\_hash

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

# Создаем событие created

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Updated existing session {session.id} for created file")

return {"action": "session\_updated", "session\_id": str(session.id), "resumed": False}

# Проверяем, есть ли недавно закрытая сессия для возобновления

recent\_session = None

if resume\_count > 0:

recent\_session = crud.get\_recent\_closed\_session(db, user\_id, file\_id)

if recent\_session:

# Возобновляем существующую сессию

recent\_session.ended\_at = None

recent\_session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

recent\_session.resume\_count = resume\_count

recent\_session.hash\_before = file\_hash

db.commit()

db.refresh(recent\_session)

print(f"🔄 Resumed session {recent\_session.id} (resume count: {resume\_count})")

session = recent\_session

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for file {event\_data.get('file\_path')}")

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": "session\_created", "session\_id": str(session.id), "resumed": recent\_session is not None}

async def handle\_file\_modified(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, session\_id: str = None, resume\_count: int = 0):

"""Обрабатывает изменение файла - обновляет существующую сессию или создает новую"""

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

if session:

# Обновляем существующую сессию

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

if resume\_count > session.resume\_count:

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

print(f"📝 Updated session {session.id} for file {event\_data.get('file\_path')} (resume: {resume\_count})")

action = "session\_updated"

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created new session {session.id} for modified file {event\_data.get('file\_path')}")

action = "session\_created"

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='modified',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": action, "session\_id": str(session.id)}

async def handle\_file\_deleted(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, session\_id: str = None):

"""Обрабатывает удаление файла - закрывает сессию и создает событие deleted"""

print(f"🗑️ Processing file deletion for file\_id: {file\_id}, session\_id: {session\_id}")

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

print(f"🔍 Found active session for deletion: {session.id}")

if session:

# Закрываем сессию с временем из события

session.ended\_at = datetime.fromisoformat(event\_data.get('event\_timestamp'))

db.commit()

db.refresh(session)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None, # Для удаленного файла хеша нет

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Closed session {session.id} for deleted file and created deleted event")

return {"action": "session\_closed", "session\_id": str(session.id)}

else:

print(f"⚠️ No active session found for deleted file, creating standalone deleted event")

# Если сессии нет, все равно создаем событие deleted

# Создаем временную сессию для привязки события

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

ended\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')), # Сразу закрываем

hash\_before=None,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Created standalone deleted event with session {session.id}")

return {"action": "deleted\_event\_created", "session\_id": str(session.id)}

async def handle\_file\_closed(db: Session, session\_id: str, file\_hash: str = None, event\_data: dict = None):

"""Обрабатывает закрытие файла - закрывает сессию с правильным ended\_at"""

if session\_id:

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

if session:

# ОБНОВЛЯЕМ СЕССИЮ С ПРАВИЛЬНЫМ ended\_at

if event\_data and 'event\_timestamp' in event\_data:

# Используем время из события закрытия

ended\_at = datetime.fromisoformat(event\_data['event\_timestamp'])

else:

# Используем текущее время

ended\_at = datetime.utcnow()

session.ended\_at = ended\_at

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие закрытия

event\_timestamp = ended\_at # Используем то же время что и для ended\_at

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='closed',

file\_hash=file\_hash,

event\_timestamp=event\_timestamp

)

crud.create\_file\_event(db, event\_record)

# Получаем информацию о файле для логов

file = crud.get\_file(db, session.file\_id)

user = crud.get\_user(db, session.user\_id)

duration = session.ended\_at - session.started\_at

print(f"🔒 Closed session {session.id} for file {file.file\_path} (user: {user.username}, duration: {duration}, ended\_at: {session.ended\_at})")

return {

"action": "session\_closed",

"session\_id": str(session.id),

"duration": str(duration),

"ended\_at": session.ended\_at.isoformat(),

"resume\_count": session.resume\_count

}

except ValueError:

pass

return {"action": "no\_session\_found"}

# Новый эндпоинт для создания сессий (для совместимости с мониторингом)

@app.post("/api/sessions")

async def create\_session(session\_data: dict, db: Session = Depends(get\_db)):

"""Создает сессию на сервере и возвращает session\_id"""

try:

print(f"📝 Creating session: {session\_data}")

# Получаем или создаем пользователя

username = session\_data.get('username')

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file\_path = session\_data.get('file\_path')

file\_name = session\_data.get('file\_name', os.path.basename(file\_path))

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# ПРОВЕРКА ДУБЛИРОВАНИЯ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, file.id)

if active\_sessions:

print(f"⚠️ Active session already exists, returning existing: {active\_sessions[0].id}")

return {"id": str(active\_sessions[0].id), "status": "existing"}

# Создаем сессию

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.utcnow(),

last\_activity=datetime.utcnow(),

hash\_before=session\_data.get('file\_hash'),

resume\_count=session\_data.get('resume\_count', 0)

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for {file\_path}")

return {"id": str(session.id), "status": "created"}

except Exception as e:

print(f"❌ Error creating session: {e}")

raise HTTPException(status\_code=500, detail=f"Error creating session: {str(e)}")

# Простые эндпоинты для проверки данных

@app.get("/api/users")

async def get\_users(db: Session = Depends(get\_db)):

"""Получает список пользователей"""

users = db.query(models.User).all()

return {"users": [{"id": str(user.id), "username": user.username} for user in users]}

@app.get("/api/files")

async def get\_files(db: Session = Depends(get\_db)):

"""Получает список файлов"""

files = db.query(models.File).all()

return {"files": [{"id": str(file.id), "path": file.file\_path, "name": file.file\_name} for file in files]}

@app.get("/api/sessions")

async def get\_sessions(db: Session = Depends(get\_db)):

"""Получает список сессий"""

sessions = db.query(models.FileSession).all()

return {"sessions": [

{

"id": str(session.id),

"user\_id": str(session.user\_id),

"file\_id": str(session.file\_id),

"started\_at": session.started\_at.isoformat(),

"ended\_at": session.ended\_at.isoformat() if session.ended\_at else None,

"last\_activity": session.last\_activity.isoformat(),

"resume\_count": session.resume\_count,

"is\_commented": session.is\_commented

} for session in sessions

]}

@app.get("/api/events")

async def get\_events(db: Session = Depends(get\_db)):

"""Получает список событий"""

events = db.query(models.FileEvent).all()

return {"events": [

{

"id": str(event.id),

"session\_id": str(event.session\_id),

"event\_type": event.event\_type,

"file\_hash": event.file\_hash,

"timestamp": event.event\_timestamp.isoformat()

} for event in events

]}

# НОВЫЕ ЭНДПОИНТЫ ДЛЯ КОММЕНТАРИЕВ

@app.post("/api/comments", response\_model=schemas.Comment)

async def create\_comment(comment: schemas.CommentCreate, db: Session = Depends(get\_db)):

"""Создает комментарий для сессии"""

try:

# Проверяем существование сессии

session = crud.get\_file\_session(db, comment.session\_id)

if not session:

raise HTTPException(status\_code=404, detail="Session not found")

# Проверяем существование пользователя

user = crud.get\_user(db, comment.user\_id)

if not user:

raise HTTPException(status\_code=404, detail="User not found")

# Проверяем, нет ли уже комментария для этой сессии

existing\_comment = crud.get\_comment\_by\_session(db, comment.session\_id)

if existing\_comment:

raise HTTPException(status\_code=400, detail="Comment already exists for this session")

session.ended\_at = datetime.utcnow()

session.is\_commented = True

# Создаем комментарий

db\_comment = crud.create\_comment(db, comment)

db.commit()

return db\_comment

except HTTPException:

raise

except Exception as e:

db.rollback()

raise HTTPException(status\_code=500, detail=f"Error creating comment: {str(e)}")

@app.get("/api/comments", response\_model=List[schemas.CommentWithUser])

async def get\_comments(

skip: int = 0,

limit: int = 100,

change\_type: Optional[str] = None,

db: Session = Depends(get\_db)

):

"""Получает список комментариев с возможностью фильтрации по типу изменения"""

try:

# Базовый запрос с join пользователя

query = db.query(

models.Comment,

models.User.username

).join(

models.User, models.Comment.user\_id == models.User.id

)

# Фильтрация по типу изменения если указан

if change\_type:

query = query.filter(models.Comment.change\_type == change\_type)

comments = query.offset(skip).limit(limit).all()

# Преобразуем в схему

result = []

for comment, username in comments:

result.append(schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

))

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comments: {str(e)}")

@app.get("/api/comments/{session\_id}", response\_model=schemas.CommentWithUser)

async def get\_comment\_by\_session(session\_id: str, db: Session = Depends(get\_db)):

"""Получает комментарий по ID сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_comment\_with\_user(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Comment not found")

comment, username = result

return schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comment: {str(e)}")

@app.get("/api/sessions-with-comments", response\_model=List[schemas.SessionWithDetails])

async def get\_sessions\_with\_comments(

skip: int = 0,

limit: int = 100,

db: Session = Depends(get\_db)

):

"""Получает сессии с комментариями"""

try:

sessions = crud.get\_sessions\_with\_comments(db, skip=skip, limit=limit)

result = []

for session, file\_path, file\_name, username, comment in sessions:

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

result.append(session\_details)

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting sessions: {str(e)}")

@app.get("/api/change-types")

async def get\_change\_types():

"""Возвращает доступные типы изменений"""

return {

"change\_types": [

"technical\_changes", # Изменение технических решений

"design\_changes", # Изменение оформления/дизайна

"content\_changes", # Изменение содержания

"bug\_fixes", # Исправление ошибок

"optimization", # Оптимизация

"refactoring", # Рефакторинг

"new\_feature", # Новая функциональность

"documentation", # Изменение документации

"other" # Другое

]

}

@app.get("/api/sessions/{session\_id}/details", response\_model=schemas.SessionWithDetails)

async def get\_session\_details(session\_id: str, db: Session = Depends(get\_db)):

"""Получает детальную информацию о сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_session\_with\_details(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Session not found")

session, file\_path, file\_name, username, comment = result

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

return session\_details

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting session details: {str(e)}")

if \_\_name\_\_ == "\_\_main\_\_":

import uvicorn

uvicorn.run(app, host="0.0.0.0", port=8000) # main.py (сервер)

import sys

import os

# Добавляем корень проекта в PYTHONPATH

current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

project\_root = os.path.dirname(os.path.dirname(current\_dir)) # Поднимаемся на уровень выше

sys.path.insert(0, project\_root)

from fastapi import FastAPI, Depends, HTTPException

from sqlalchemy.orm import Session

from shared.config\_loader import get\_api\_config

from . import database, models, crud, schemas

from typing import List, Optional

import uuid

from datetime import datetime

# Загружаем конфигурацию API

api\_config = get\_api\_config()

app = FastAPI(

title="File Monitoring API",

description="API для отслеживания изменений файлов с системой комментариев",

version="1.0.0"

)

# Dependency

def get\_db():

db = database.SessionLocal()

try:

yield db

finally:

db.close()

@app.get("/")

async def root():

return {

"message": "File Monitoring System API",

"status": "running",

"config\_source": "YAML"

}

@app.get("/health")

async def health\_check():

return {"status": "healthy", "service": "session-service"}

@app.get("/config")

async def show\_config():

"""Показывает текущую конфигурацию (без паролей)"""

return {

"api\_host": api\_config.get('host'),

"api\_port": api\_config.get('port'),

"environment": "development"

}

# Events endpoint для мониторинга

@app.post("/api/events")

async def create\_event(event\_data: dict, db: Session = Depends(get\_db)):

"""Эндпоинт для приема событий от мониторинга"""

print(f"📨 Received event: {event\_data}")

try:

# Обрабатываем событие в зависимости от типа

result = await process\_file\_event(db, event\_data)

return {"status": "processed", "event\_type": event\_data.get('event\_type'), "result": result}

except Exception as e:

print(f"❌ Error processing event: {e}")

raise HTTPException(status\_code=500, detail=f"Error processing event: {str(e)}")

async def process\_file\_event(db: Session, event\_data: dict):

"""Обрабатывает файловое событие и создает/обновляет сессии"""

event\_type = event\_data.get('event\_type')

file\_path = event\_data.get('file\_path')

file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

file\_hash = event\_data.get('file\_hash')

session\_id = event\_data.get('session\_id')

resume\_count = event\_data.get('resume\_count', 0)

print(f"🔧 Processing {event\_type} for {file\_path} (user: {username}, session: {session\_id}, resume: {resume\_count})")

# Обрабатываем перемещение отдельно

if event\_type == 'moved':

return await handle\_file\_moved(db, event\_data, file\_hash)

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# Обрабатываем в зависимости от типа события

if event\_type == 'created':

return await handle\_file\_created(db, user.id, file.id, event\_data, file\_hash, resume\_count)

elif event\_type == 'modified':

return await handle\_file\_modified(db, user.id, file.id, event\_data, file\_hash, session\_id, resume\_count)

elif event\_type == 'deleted':

return await handle\_file\_deleted(db, user.id, file.id, event\_data, session\_id)

elif event\_type == 'closed':

return await handle\_file\_closed(db, session\_id, file\_hash, event\_data)

else:

raise ValueError(f"Unknown event type: {event\_type}")

async def handle\_file\_moved(db: Session, event\_data: dict, file\_hash: str = None):

"""Обрабатывает перемещение/переименование файла"""

old\_file\_path = event\_data.get('old\_file\_path')

new\_file\_path = event\_data.get('file\_path')

new\_file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

print(f"🔄 Processing file move: {old\_file\_path} -> {new\_file\_path}")

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Ищем активную сессию для старого пути

old\_file = crud.get\_file\_by\_path(db, old\_file\_path)

if old\_file:

# Находим активные сессии для этого файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, old\_file.id)

if active\_sessions:

# Берем последнюю активную сессию

session = active\_sessions[0]

# Обновляем файл с новым путем

old\_file.file\_path = new\_file\_path

old\_file.file\_name = new\_file\_name

db.commit()

# Обновляем время активности сессии

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Moved session {session.id} from {old\_file\_path} to {new\_file\_path}")

return {"action": "session\_moved", "session\_id": str(session.id)}

# Если не нашли активную сессию, создаем новую

file\_data = schemas.FileCreate(file\_path=new\_file\_path, file\_name=new\_file\_name)

file = crud.create\_file(db, file\_data)

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"✅ Created new session {session.id} for moved file {new\_file\_path}")

return {"action": "session\_created", "session\_id": str(session.id)}

async def handle\_file\_created(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, resume\_count: int = 0):

"""Обрабатывает создание файла - создает новую сессию"""

# ПРОВЕРКА ДУБЛИРОВАНИЯ СЕССИЙ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

print(f"⚠️ Active session already exists for file, updating instead of creating new")

session = active\_sessions[0]

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_before = file\_hash

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

# Создаем событие created

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Updated existing session {session.id} for created file")

return {"action": "session\_updated", "session\_id": str(session.id), "resumed": False}

# Проверяем, есть ли недавно закрытая сессия для возобновления

recent\_session = None

if resume\_count > 0:

recent\_session = crud.get\_recent\_closed\_session(db, user\_id, file\_id)

if recent\_session:

# Возобновляем существующую сессию

recent\_session.ended\_at = None

recent\_session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

recent\_session.resume\_count = resume\_count

recent\_session.hash\_before = file\_hash

db.commit()

db.refresh(recent\_session)

print(f"🔄 Resumed session {recent\_session.id} (resume count: {resume\_count})")

session = recent\_session

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for file {event\_data.get('file\_path')}")

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": "session\_created", "session\_id": str(session.id), "resumed": recent\_session is not None}

async def handle\_file\_modified(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, session\_id: str = None, resume\_count: int = 0):

"""Обрабатывает изменение файла - обновляет существующую сессию или создает новую"""

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

if session:

# Обновляем существующую сессию

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

if resume\_count > session.resume\_count:

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

print(f"📝 Updated session {session.id} for file {event\_data.get('file\_path')} (resume: {resume\_count})")

action = "session\_updated"

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created new session {session.id} for modified file {event\_data.get('file\_path')}")

action = "session\_created"

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='modified',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": action, "session\_id": str(session.id)}

async def handle\_file\_deleted(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, session\_id: str = None):

"""Обрабатывает удаление файла - закрывает сессию и создает событие deleted"""

print(f"🗑️ Processing file deletion for file\_id: {file\_id}, session\_id: {session\_id}")

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

print(f"🔍 Found active session for deletion: {session.id}")

if session:

# Закрываем сессию с временем из события

session.ended\_at = datetime.fromisoformat(event\_data.get('event\_timestamp'))

db.commit()

db.refresh(session)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None, # Для удаленного файла хеша нет

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Closed session {session.id} for deleted file and created deleted event")

return {"action": "session\_closed", "session\_id": str(session.id)}

else:

print(f"⚠️ No active session found for deleted file, creating standalone deleted event")

# Если сессии нет, все равно создаем событие deleted

# Создаем временную сессию для привязки события

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

ended\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')), # Сразу закрываем

hash\_before=None,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Created standalone deleted event with session {session.id}")

return {"action": "deleted\_event\_created", "session\_id": str(session.id)}

async def handle\_file\_closed(db: Session, session\_id: str, file\_hash: str = None, event\_data: dict = None):

"""Обрабатывает закрытие файла - закрывает сессию с правильным ended\_at"""

if session\_id:

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

if session:

# ОБНОВЛЯЕМ СЕССИЮ С ПРАВИЛЬНЫМ ended\_at

if event\_data and 'event\_timestamp' in event\_data:

# Используем время из события закрытия

ended\_at = datetime.fromisoformat(event\_data['event\_timestamp'])

else:

# Используем текущее время

ended\_at = datetime.utcnow()

session.ended\_at = ended\_at

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие закрытия

event\_timestamp = ended\_at # Используем то же время что и для ended\_at

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='closed',

file\_hash=file\_hash,

event\_timestamp=event\_timestamp

)

crud.create\_file\_event(db, event\_record)

# Получаем информацию о файле для логов

file = crud.get\_file(db, session.file\_id)

user = crud.get\_user(db, session.user\_id)

duration = session.ended\_at - session.started\_at

print(f"🔒 Closed session {session.id} for file {file.file\_path} (user: {user.username}, duration: {duration}, ended\_at: {session.ended\_at})")

return {

"action": "session\_closed",

"session\_id": str(session.id),

"duration": str(duration),

"ended\_at": session.ended\_at.isoformat(),

"resume\_count": session.resume\_count

}

except ValueError:

pass

return {"action": "no\_session\_found"}

# Новый эндпоинт для создания сессий (для совместимости с мониторингом)

@app.post("/api/sessions")

async def create\_session(session\_data: dict, db: Session = Depends(get\_db)):

"""Создает сессию на сервере и возвращает session\_id"""

try:

print(f"📝 Creating session: {session\_data}")

# Получаем или создаем пользователя

username = session\_data.get('username')

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file\_path = session\_data.get('file\_path')

file\_name = session\_data.get('file\_name', os.path.basename(file\_path))

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# ПРОВЕРКА ДУБЛИРОВАНИЯ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, file.id)

if active\_sessions:

print(f"⚠️ Active session already exists, returning existing: {active\_sessions[0].id}")

return {"id": str(active\_sessions[0].id), "status": "existing"}

# Создаем сессию

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.utcnow(),

last\_activity=datetime.utcnow(),

hash\_before=session\_data.get('file\_hash'),

resume\_count=session\_data.get('resume\_count', 0)

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for {file\_path}")

return {"id": str(session.id), "status": "created"}

except Exception as e:

print(f"❌ Error creating session: {e}")

raise HTTPException(status\_code=500, detail=f"Error creating session: {str(e)}")

# Простые эндпоинты для проверки данных

@app.get("/api/users")

async def get\_users(db: Session = Depends(get\_db)):

"""Получает список пользователей"""

users = db.query(models.User).all()

return {"users": [{"id": str(user.id), "username": user.username} for user in users]}

@app.get("/api/files")

async def get\_files(db: Session = Depends(get\_db)):

"""Получает список файлов"""

files = db.query(models.File).all()

return {"files": [{"id": str(file.id), "path": file.file\_path, "name": file.file\_name} for file in files]}

@app.get("/api/sessions")

async def get\_sessions(db: Session = Depends(get\_db)):

"""Получает список сессий"""

sessions = db.query(models.FileSession).all()

return {"sessions": [

{

"id": str(session.id),

"user\_id": str(session.user\_id),

"file\_id": str(session.file\_id),

"started\_at": session.started\_at.isoformat(),

"ended\_at": session.ended\_at.isoformat() if session.ended\_at else None,

"last\_activity": session.last\_activity.isoformat(),

"resume\_count": session.resume\_count,

"is\_commented": session.is\_commented

} for session in sessions

]}

@app.get("/api/events")

async def get\_events(db: Session = Depends(get\_db)):

"""Получает список событий"""

events = db.query(models.FileEvent).all()

return {"events": [

{

"id": str(event.id),

"session\_id": str(event.session\_id),

"event\_type": event.event\_type,

"file\_hash": event.file\_hash,

"timestamp": event.event\_timestamp.isoformat()

} for event in events

]}

# НОВЫЕ ЭНДПОИНТЫ ДЛЯ КОММЕНТАРИЕВ

@app.post("/api/comments", response\_model=schemas.Comment)

async def create\_comment(comment: schemas.CommentCreate, db: Session = Depends(get\_db)):

"""Создает комментарий для сессии"""

try:

# Проверяем существование сессии

session = crud.get\_file\_session(db, comment.session\_id)

if not session:

raise HTTPException(status\_code=404, detail="Session not found")

# Проверяем существование пользователя

user = crud.get\_user(db, comment.user\_id)

if not user:

raise HTTPException(status\_code=404, detail="User not found")

# Проверяем, нет ли уже комментария для этой сессии

existing\_comment = crud.get\_comment\_by\_session(db, comment.session\_id)

if existing\_comment:

raise HTTPException(status\_code=400, detail="Comment already exists for this session")

session.ended\_at = datetime.utcnow()

session.is\_commented = True

# Создаем комментарий

db\_comment = crud.create\_comment(db, comment)

db.commit()

return db\_comment

except HTTPException:

raise

except Exception as e:

db.rollback()

raise HTTPException(status\_code=500, detail=f"Error creating comment: {str(e)}")

@app.get("/api/comments", response\_model=List[schemas.CommentWithUser])

async def get\_comments(

skip: int = 0,

limit: int = 100,

change\_type: Optional[str] = None,

db: Session = Depends(get\_db)

):

"""Получает список комментариев с возможностью фильтрации по типу изменения"""

try:

# Базовый запрос с join пользователя

query = db.query(

models.Comment,

models.User.username

).join(

models.User, models.Comment.user\_id == models.User.id

)

# Фильтрация по типу изменения если указан

if change\_type:

query = query.filter(models.Comment.change\_type == change\_type)

comments = query.offset(skip).limit(limit).all()

# Преобразуем в схему

result = []

for comment, username in comments:

result.append(schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

))

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comments: {str(e)}")

@app.get("/api/comments/{session\_id}", response\_model=schemas.CommentWithUser)

async def get\_comment\_by\_session(session\_id: str, db: Session = Depends(get\_db)):

"""Получает комментарий по ID сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_comment\_with\_user(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Comment not found")

comment, username = result

return schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comment: {str(e)}")

@app.get("/api/sessions-with-comments", response\_model=List[schemas.SessionWithDetails])

async def get\_sessions\_with\_comments(

skip: int = 0,

limit: int = 100,

db: Session = Depends(get\_db)

):

"""Получает сессии с комментариями"""

try:

sessions = crud.get\_sessions\_with\_comments(db, skip=skip, limit=limit)

result = []

for session, file\_path, file\_name, username, comment in sessions:

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

result.append(session\_details)

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting sessions: {str(e)}")

@app.get("/api/change-types")

async def get\_change\_types():

"""Возвращает доступные типы изменений"""

return {

"change\_types": [

"technical\_changes", # Изменение технических решений

"design\_changes", # Изменение оформления/дизайна

"content\_changes", # Изменение содержания

"bug\_fixes", # Исправление ошибок

"optimization", # Оптимизация

"refactoring", # Рефакторинг

"new\_feature", # Новая функциональность

"documentation", # Изменение документации

"other" # Другое

]

}

@app.get("/api/sessions/{session\_id}/details", response\_model=schemas.SessionWithDetails)

async def get\_session\_details(session\_id: str, db: Session = Depends(get\_db)):

"""Получает детальную информацию о сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_session\_with\_details(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Session not found")

session, file\_path, file\_name, username, comment = result

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

return session\_details

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting session details: {str(e)}")

if \_\_name\_\_ == "\_\_main\_\_":

import uvicorn

uvicorn.run(app, host="0.0.0.0", port=8000) # main.py (сервер)

import sys

import os

# Добавляем корень проекта в PYTHONPATH

current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

project\_root = os.path.dirname(os.path.dirname(current\_dir)) # Поднимаемся на уровень выше

sys.path.insert(0, project\_root)

from fastapi import FastAPI, Depends, HTTPException

from sqlalchemy.orm import Session

from shared.config\_loader import get\_api\_config

from . import database, models, crud, schemas

from typing import List, Optional

import uuid

from datetime import datetime

# Загружаем конфигурацию API

api\_config = get\_api\_config()

app = FastAPI(

title="File Monitoring API",

description="API для отслеживания изменений файлов с системой комментариев",

version="1.0.0"

)

# Dependency

def get\_db():

db = database.SessionLocal()

try:

yield db

finally:

db.close()

@app.get("/")

async def root():

return {

"message": "File Monitoring System API",

"status": "running",

"config\_source": "YAML"

}

@app.get("/health")

async def health\_check():

return {"status": "healthy", "service": "session-service"}

@app.get("/config")

async def show\_config():

"""Показывает текущую конфигурацию (без паролей)"""

return {

"api\_host": api\_config.get('host'),

"api\_port": api\_config.get('port'),

"environment": "development"

}

# Events endpoint для мониторинга

@app.post("/api/events")

async def create\_event(event\_data: dict, db: Session = Depends(get\_db)):

"""Эндпоинт для приема событий от мониторинга"""

print(f"📨 Received event: {event\_data}")

try:

# Обрабатываем событие в зависимости от типа

result = await process\_file\_event(db, event\_data)

return {"status": "processed", "event\_type": event\_data.get('event\_type'), "result": result}

except Exception as e:

print(f"❌ Error processing event: {e}")

raise HTTPException(status\_code=500, detail=f"Error processing event: {str(e)}")

async def process\_file\_event(db: Session, event\_data: dict):

"""Обрабатывает файловое событие и создает/обновляет сессии"""

event\_type = event\_data.get('event\_type')

file\_path = event\_data.get('file\_path')

file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

file\_hash = event\_data.get('file\_hash')

session\_id = event\_data.get('session\_id')

resume\_count = event\_data.get('resume\_count', 0)

print(f"🔧 Processing {event\_type} for {file\_path} (user: {username}, session: {session\_id}, resume: {resume\_count})")

# Обрабатываем перемещение отдельно

if event\_type == 'moved':

return await handle\_file\_moved(db, event\_data, file\_hash)

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# Обрабатываем в зависимости от типа события

if event\_type == 'created':

return await handle\_file\_created(db, user.id, file.id, event\_data, file\_hash, resume\_count)

elif event\_type == 'modified':

return await handle\_file\_modified(db, user.id, file.id, event\_data, file\_hash, session\_id, resume\_count)

elif event\_type == 'deleted':

return await handle\_file\_deleted(db, user.id, file.id, event\_data, session\_id)

elif event\_type == 'closed':

return await handle\_file\_closed(db, session\_id, file\_hash, event\_data)

else:

raise ValueError(f"Unknown event type: {event\_type}")

async def handle\_file\_moved(db: Session, event\_data: dict, file\_hash: str = None):

"""Обрабатывает перемещение/переименование файла"""

old\_file\_path = event\_data.get('old\_file\_path')

new\_file\_path = event\_data.get('file\_path')

new\_file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

print(f"🔄 Processing file move: {old\_file\_path} -> {new\_file\_path}")

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Ищем активную сессию для старого пути

old\_file = crud.get\_file\_by\_path(db, old\_file\_path)

if old\_file:

# Находим активные сессии для этого файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, old\_file.id)

if active\_sessions:

# Берем последнюю активную сессию

session = active\_sessions[0]

# Обновляем файл с новым путем

old\_file.file\_path = new\_file\_path

old\_file.file\_name = new\_file\_name

db.commit()

# Обновляем время активности сессии

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Moved session {session.id} from {old\_file\_path} to {new\_file\_path}")

return {"action": "session\_moved", "session\_id": str(session.id)}

# Если не нашли активную сессию, создаем новую

file\_data = schemas.FileCreate(file\_path=new\_file\_path, file\_name=new\_file\_name)

file = crud.create\_file(db, file\_data)

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"✅ Created new session {session.id} for moved file {new\_file\_path}")

return {"action": "session\_created", "session\_id": str(session.id)}

async def handle\_file\_created(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, resume\_count: int = 0):

"""Обрабатывает создание файла - создает новую сессию"""

# ПРОВЕРКА ДУБЛИРОВАНИЯ СЕССИЙ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

print(f"⚠️ Active session already exists for file, updating instead of creating new")

session = active\_sessions[0]

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_before = file\_hash

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

# Создаем событие created

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Updated existing session {session.id} for created file")

return {"action": "session\_updated", "session\_id": str(session.id), "resumed": False}

# Проверяем, есть ли недавно закрытая сессия для возобновления

recent\_session = None

if resume\_count > 0:

recent\_session = crud.get\_recent\_closed\_session(db, user\_id, file\_id)

if recent\_session:

# Возобновляем существующую сессию

recent\_session.ended\_at = None

recent\_session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

recent\_session.resume\_count = resume\_count

recent\_session.hash\_before = file\_hash

db.commit()

db.refresh(recent\_session)

print(f"🔄 Resumed session {recent\_session.id} (resume count: {resume\_count})")

session = recent\_session

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for file {event\_data.get('file\_path')}")

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": "session\_created", "session\_id": str(session.id), "resumed": recent\_session is not None}

async def handle\_file\_modified(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, session\_id: str = None, resume\_count: int = 0):

"""Обрабатывает изменение файла - обновляет существующую сессию или создает новую"""

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

if session:

# Обновляем существующую сессию

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

if resume\_count > session.resume\_count:

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

print(f"📝 Updated session {session.id} for file {event\_data.get('file\_path')} (resume: {resume\_count})")

action = "session\_updated"

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created new session {session.id} for modified file {event\_data.get('file\_path')}")

action = "session\_created"

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='modified',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": action, "session\_id": str(session.id)}

async def handle\_file\_deleted(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, session\_id: str = None):

"""Обрабатывает удаление файла - закрывает сессию и создает событие deleted"""

print(f"🗑️ Processing file deletion for file\_id: {file\_id}, session\_id: {session\_id}")

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

print(f"🔍 Found active session for deletion: {session.id}")

if session:

# Закрываем сессию с временем из события

session.ended\_at = datetime.fromisoformat(event\_data.get('event\_timestamp'))

db.commit()

db.refresh(session)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None, # Для удаленного файла хеша нет

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Closed session {session.id} for deleted file and created deleted event")

return {"action": "session\_closed", "session\_id": str(session.id)}

else:

print(f"⚠️ No active session found for deleted file, creating standalone deleted event")

# Если сессии нет, все равно создаем событие deleted

# Создаем временную сессию для привязки события

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

ended\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')), # Сразу закрываем

hash\_before=None,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Created standalone deleted event with session {session.id}")

return {"action": "deleted\_event\_created", "session\_id": str(session.id)}

async def handle\_file\_closed(db: Session, session\_id: str, file\_hash: str = None, event\_data: dict = None):

"""Обрабатывает закрытие файла - закрывает сессию с правильным ended\_at"""

if session\_id:

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

if session:

# ОБНОВЛЯЕМ СЕССИЮ С ПРАВИЛЬНЫМ ended\_at

if event\_data and 'event\_timestamp' in event\_data:

# Используем время из события закрытия

ended\_at = datetime.fromisoformat(event\_data['event\_timestamp'])

else:

# Используем текущее время

ended\_at = datetime.utcnow()

session.ended\_at = ended\_at

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие закрытия

event\_timestamp = ended\_at # Используем то же время что и для ended\_at

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='closed',

file\_hash=file\_hash,

event\_timestamp=event\_timestamp

)

crud.create\_file\_event(db, event\_record)

# Получаем информацию о файле для логов

file = crud.get\_file(db, session.file\_id)

user = crud.get\_user(db, session.user\_id)

duration = session.ended\_at - session.started\_at

print(f"🔒 Closed session {session.id} for file {file.file\_path} (user: {user.username}, duration: {duration}, ended\_at: {session.ended\_at})")

return {

"action": "session\_closed",

"session\_id": str(session.id),

"duration": str(duration),

"ended\_at": session.ended\_at.isoformat(),

"resume\_count": session.resume\_count

}

except ValueError:

pass

return {"action": "no\_session\_found"}

# Новый эндпоинт для создания сессий (для совместимости с мониторингом)

@app.post("/api/sessions")

async def create\_session(session\_data: dict, db: Session = Depends(get\_db)):

"""Создает сессию на сервере и возвращает session\_id"""

try:

print(f"📝 Creating session: {session\_data}")

# Получаем или создаем пользователя

username = session\_data.get('username')

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file\_path = session\_data.get('file\_path')

file\_name = session\_data.get('file\_name', os.path.basename(file\_path))

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# ПРОВЕРКА ДУБЛИРОВАНИЯ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, file.id)

if active\_sessions:

print(f"⚠️ Active session already exists, returning existing: {active\_sessions[0].id}")

return {"id": str(active\_sessions[0].id), "status": "existing"}

# Создаем сессию

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.utcnow(),

last\_activity=datetime.utcnow(),

hash\_before=session\_data.get('file\_hash'),

resume\_count=session\_data.get('resume\_count', 0)

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for {file\_path}")

return {"id": str(session.id), "status": "created"}

except Exception as e:

print(f"❌ Error creating session: {e}")

raise HTTPException(status\_code=500, detail=f"Error creating session: {str(e)}")

# Простые эндпоинты для проверки данных

@app.get("/api/users")

async def get\_users(db: Session = Depends(get\_db)):

"""Получает список пользователей"""

users = db.query(models.User).all()

return {"users": [{"id": str(user.id), "username": user.username} for user in users]}

@app.get("/api/files")

async def get\_files(db: Session = Depends(get\_db)):

"""Получает список файлов"""

files = db.query(models.File).all()

return {"files": [{"id": str(file.id), "path": file.file\_path, "name": file.file\_name} for file in files]}

@app.get("/api/sessions")

async def get\_sessions(db: Session = Depends(get\_db)):

"""Получает список сессий"""

sessions = db.query(models.FileSession).all()

return {"sessions": [

{

"id": str(session.id),

"user\_id": str(session.user\_id),

"file\_id": str(session.file\_id),

"started\_at": session.started\_at.isoformat(),

"ended\_at": session.ended\_at.isoformat() if session.ended\_at else None,

"last\_activity": session.last\_activity.isoformat(),

"resume\_count": session.resume\_count,

"is\_commented": session.is\_commented

} for session in sessions

]}

@app.get("/api/events")

async def get\_events(db: Session = Depends(get\_db)):

"""Получает список событий"""

events = db.query(models.FileEvent).all()

return {"events": [

{

"id": str(event.id),

"session\_id": str(event.session\_id),

"event\_type": event.event\_type,

"file\_hash": event.file\_hash,

"timestamp": event.event\_timestamp.isoformat()

} for event in events

]}

# НОВЫЕ ЭНДПОИНТЫ ДЛЯ КОММЕНТАРИЕВ

@app.post("/api/comments", response\_model=schemas.Comment)

async def create\_comment(comment: schemas.CommentCreate, db: Session = Depends(get\_db)):

"""Создает комментарий для сессии"""

try:

# Проверяем существование сессии

session = crud.get\_file\_session(db, comment.session\_id)

if not session:

raise HTTPException(status\_code=404, detail="Session not found")

# Проверяем существование пользователя

user = crud.get\_user(db, comment.user\_id)

if not user:

raise HTTPException(status\_code=404, detail="User not found")

# Проверяем, нет ли уже комментария для этой сессии

existing\_comment = crud.get\_comment\_by\_session(db, comment.session\_id)

if existing\_comment:

raise HTTPException(status\_code=400, detail="Comment already exists for this session")

session.ended\_at = datetime.utcnow()

session.is\_commented = True

# Создаем комментарий

db\_comment = crud.create\_comment(db, comment)

db.commit()

return db\_comment

except HTTPException:

raise

except Exception as e:

db.rollback()

raise HTTPException(status\_code=500, detail=f"Error creating comment: {str(e)}")

@app.get("/api/comments", response\_model=List[schemas.CommentWithUser])

async def get\_comments(

skip: int = 0,

limit: int = 100,

change\_type: Optional[str] = None,

db: Session = Depends(get\_db)

):

"""Получает список комментариев с возможностью фильтрации по типу изменения"""

try:

# Базовый запрос с join пользователя

query = db.query(

models.Comment,

models.User.username

).join(

models.User, models.Comment.user\_id == models.User.id

)

# Фильтрация по типу изменения если указан

if change\_type:

query = query.filter(models.Comment.change\_type == change\_type)

comments = query.offset(skip).limit(limit).all()

# Преобразуем в схему

result = []

for comment, username in comments:

result.append(schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

))

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comments: {str(e)}")

@app.get("/api/comments/{session\_id}", response\_model=schemas.CommentWithUser)

async def get\_comment\_by\_session(session\_id: str, db: Session = Depends(get\_db)):

"""Получает комментарий по ID сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_comment\_with\_user(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Comment not found")

comment, username = result

return schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comment: {str(e)}")

@app.get("/api/sessions-with-comments", response\_model=List[schemas.SessionWithDetails])

async def get\_sessions\_with\_comments(

skip: int = 0,

limit: int = 100,

db: Session = Depends(get\_db)

):

"""Получает сессии с комментариями"""

try:

sessions = crud.get\_sessions\_with\_comments(db, skip=skip, limit=limit)

result = []

for session, file\_path, file\_name, username, comment in sessions:

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

result.append(session\_details)

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting sessions: {str(e)}")

@app.get("/api/change-types")

async def get\_change\_types():

"""Возвращает доступные типы изменений"""

return {

"change\_types": [

"technical\_changes", # Изменение технических решений

"design\_changes", # Изменение оформления/дизайна

"content\_changes", # Изменение содержания

"bug\_fixes", # Исправление ошибок

"optimization", # Оптимизация

"refactoring", # Рефакторинг

"new\_feature", # Новая функциональность

"documentation", # Изменение документации

"other" # Другое

]

}

@app.get("/api/sessions/{session\_id}/details", response\_model=schemas.SessionWithDetails)

async def get\_session\_details(session\_id: str, db: Session = Depends(get\_db)):

"""Получает детальную информацию о сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_session\_with\_details(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Session not found")

session, file\_path, file\_name, username, comment = result

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

return session\_details

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting session details: {str(e)}")

if \_\_name\_\_ == "\_\_main\_\_":

import uvicorn

uvicorn.run(app, host="0.0.0.0", port=8000) # main.py (сервер)

import sys

import os

# Добавляем корень проекта в PYTHONPATH

current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

project\_root = os.path.dirname(os.path.dirname(current\_dir)) # Поднимаемся на уровень выше

sys.path.insert(0, project\_root)

from fastapi import FastAPI, Depends, HTTPException

from sqlalchemy.orm import Session

from shared.config\_loader import get\_api\_config

from . import database, models, crud, schemas

from typing import List, Optional

import uuid

from datetime import datetime

# Загружаем конфигурацию API

api\_config = get\_api\_config()

app = FastAPI(

title="File Monitoring API",

description="API для отслеживания изменений файлов с системой комментариев",

version="1.0.0"

)

# Dependency

def get\_db():

db = database.SessionLocal()

try:

yield db

finally:

db.close()

@app.get("/")

async def root():

return {

"message": "File Monitoring System API",

"status": "running",

"config\_source": "YAML"

}

@app.get("/health")

async def health\_check():

return {"status": "healthy", "service": "session-service"}

@app.get("/config")

async def show\_config():

"""Показывает текущую конфигурацию (без паролей)"""

return {

"api\_host": api\_config.get('host'),

"api\_port": api\_config.get('port'),

"environment": "development"

}

# Events endpoint для мониторинга

@app.post("/api/events")

async def create\_event(event\_data: dict, db: Session = Depends(get\_db)):

"""Эндпоинт для приема событий от мониторинга"""

print(f"📨 Received event: {event\_data}")

try:

# Обрабатываем событие в зависимости от типа

result = await process\_file\_event(db, event\_data)

return {"status": "processed", "event\_type": event\_data.get('event\_type'), "result": result}

except Exception as e:

print(f"❌ Error processing event: {e}")

raise HTTPException(status\_code=500, detail=f"Error processing event: {str(e)}")

async def process\_file\_event(db: Session, event\_data: dict):

"""Обрабатывает файловое событие и создает/обновляет сессии"""

event\_type = event\_data.get('event\_type')

file\_path = event\_data.get('file\_path')

file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

file\_hash = event\_data.get('file\_hash')

session\_id = event\_data.get('session\_id')

resume\_count = event\_data.get('resume\_count', 0)

print(f"🔧 Processing {event\_type} for {file\_path} (user: {username}, session: {session\_id}, resume: {resume\_count})")

# Обрабатываем перемещение отдельно

if event\_type == 'moved':

return await handle\_file\_moved(db, event\_data, file\_hash)

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# Обрабатываем в зависимости от типа события

if event\_type == 'created':

return await handle\_file\_created(db, user.id, file.id, event\_data, file\_hash, resume\_count)

elif event\_type == 'modified':

return await handle\_file\_modified(db, user.id, file.id, event\_data, file\_hash, session\_id, resume\_count)

elif event\_type == 'deleted':

return await handle\_file\_deleted(db, user.id, file.id, event\_data, session\_id)

elif event\_type == 'closed':

return await handle\_file\_closed(db, session\_id, file\_hash, event\_data)

else:

raise ValueError(f"Unknown event type: {event\_type}")

async def handle\_file\_moved(db: Session, event\_data: dict, file\_hash: str = None):

"""Обрабатывает перемещение/переименование файла"""

old\_file\_path = event\_data.get('old\_file\_path')

new\_file\_path = event\_data.get('file\_path')

new\_file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

print(f"🔄 Processing file move: {old\_file\_path} -> {new\_file\_path}")

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Ищем активную сессию для старого пути

old\_file = crud.get\_file\_by\_path(db, old\_file\_path)

if old\_file:

# Находим активные сессии для этого файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, old\_file.id)

if active\_sessions:

# Берем последнюю активную сессию

session = active\_sessions[0]

# Обновляем файл с новым путем

old\_file.file\_path = new\_file\_path

old\_file.file\_name = new\_file\_name

db.commit()

# Обновляем время активности сессии

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Moved session {session.id} from {old\_file\_path} to {new\_file\_path}")

return {"action": "session\_moved", "session\_id": str(session.id)}

# Если не нашли активную сессию, создаем новую

file\_data = schemas.FileCreate(file\_path=new\_file\_path, file\_name=new\_file\_name)

file = crud.create\_file(db, file\_data)

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"✅ Created new session {session.id} for moved file {new\_file\_path}")

return {"action": "session\_created", "session\_id": str(session.id)}

async def handle\_file\_created(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, resume\_count: int = 0):

"""Обрабатывает создание файла - создает новую сессию"""

# ПРОВЕРКА ДУБЛИРОВАНИЯ СЕССИЙ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

print(f"⚠️ Active session already exists for file, updating instead of creating new")

session = active\_sessions[0]

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_before = file\_hash

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

# Создаем событие created

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Updated existing session {session.id} for created file")

return {"action": "session\_updated", "session\_id": str(session.id), "resumed": False}

# Проверяем, есть ли недавно закрытая сессия для возобновления

recent\_session = None

if resume\_count > 0:

recent\_session = crud.get\_recent\_closed\_session(db, user\_id, file\_id)

if recent\_session:

# Возобновляем существующую сессию

recent\_session.ended\_at = None

recent\_session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

recent\_session.resume\_count = resume\_count

recent\_session.hash\_before = file\_hash

db.commit()

db.refresh(recent\_session)

print(f"🔄 Resumed session {recent\_session.id} (resume count: {resume\_count})")

session = recent\_session

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for file {event\_data.get('file\_path')}")

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": "session\_created", "session\_id": str(session.id), "resumed": recent\_session is not None}

async def handle\_file\_modified(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, session\_id: str = None, resume\_count: int = 0):

"""Обрабатывает изменение файла - обновляет существующую сессию или создает новую"""

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

if session:

# Обновляем существующую сессию

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

if resume\_count > session.resume\_count:

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

print(f"📝 Updated session {session.id} for file {event\_data.get('file\_path')} (resume: {resume\_count})")

action = "session\_updated"

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created new session {session.id} for modified file {event\_data.get('file\_path')}")

action = "session\_created"

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='modified',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": action, "session\_id": str(session.id)}

async def handle\_file\_deleted(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, session\_id: str = None):

"""Обрабатывает удаление файла - закрывает сессию и создает событие deleted"""

print(f"🗑️ Processing file deletion for file\_id: {file\_id}, session\_id: {session\_id}")

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

print(f"🔍 Found active session for deletion: {session.id}")

if session:

# Закрываем сессию с временем из события

session.ended\_at = datetime.fromisoformat(event\_data.get('event\_timestamp'))

db.commit()

db.refresh(session)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None, # Для удаленного файла хеша нет

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Closed session {session.id} for deleted file and created deleted event")

return {"action": "session\_closed", "session\_id": str(session.id)}

else:

print(f"⚠️ No active session found for deleted file, creating standalone deleted event")

# Если сессии нет, все равно создаем событие deleted

# Создаем временную сессию для привязки события

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

ended\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')), # Сразу закрываем

hash\_before=None,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Created standalone deleted event with session {session.id}")

return {"action": "deleted\_event\_created", "session\_id": str(session.id)}

async def handle\_file\_closed(db: Session, session\_id: str, file\_hash: str = None, event\_data: dict = None):

"""Обрабатывает закрытие файла - закрывает сессию с правильным ended\_at"""

if session\_id:

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

if session:

# ОБНОВЛЯЕМ СЕССИЮ С ПРАВИЛЬНЫМ ended\_at

if event\_data and 'event\_timestamp' in event\_data:

# Используем время из события закрытия

ended\_at = datetime.fromisoformat(event\_data['event\_timestamp'])

else:

# Используем текущее время

ended\_at = datetime.utcnow()

session.ended\_at = ended\_at

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие закрытия

event\_timestamp = ended\_at # Используем то же время что и для ended\_at

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='closed',

file\_hash=file\_hash,

event\_timestamp=event\_timestamp

)

crud.create\_file\_event(db, event\_record)

# Получаем информацию о файле для логов

file = crud.get\_file(db, session.file\_id)

user = crud.get\_user(db, session.user\_id)

duration = session.ended\_at - session.started\_at

print(f"🔒 Closed session {session.id} for file {file.file\_path} (user: {user.username}, duration: {duration}, ended\_at: {session.ended\_at})")

return {

"action": "session\_closed",

"session\_id": str(session.id),

"duration": str(duration),

"ended\_at": session.ended\_at.isoformat(),

"resume\_count": session.resume\_count

}

except ValueError:

pass

return {"action": "no\_session\_found"}

# Новый эндпоинт для создания сессий (для совместимости с мониторингом)

@app.post("/api/sessions")

async def create\_session(session\_data: dict, db: Session = Depends(get\_db)):

"""Создает сессию на сервере и возвращает session\_id"""

try:

print(f"📝 Creating session: {session\_data}")

# Получаем или создаем пользователя

username = session\_data.get('username')

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file\_path = session\_data.get('file\_path')

file\_name = session\_data.get('file\_name', os.path.basename(file\_path))

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# ПРОВЕРКА ДУБЛИРОВАНИЯ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, file.id)

if active\_sessions:

print(f"⚠️ Active session already exists, returning existing: {active\_sessions[0].id}")

return {"id": str(active\_sessions[0].id), "status": "existing"}

# Создаем сессию

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.utcnow(),

last\_activity=datetime.utcnow(),

hash\_before=session\_data.get('file\_hash'),

resume\_count=session\_data.get('resume\_count', 0)

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for {file\_path}")

return {"id": str(session.id), "status": "created"}

except Exception as e:

print(f"❌ Error creating session: {e}")

raise HTTPException(status\_code=500, detail=f"Error creating session: {str(e)}")

# Простые эндпоинты для проверки данных

@app.get("/api/users")

async def get\_users(db: Session = Depends(get\_db)):

"""Получает список пользователей"""

users = db.query(models.User).all()

return {"users": [{"id": str(user.id), "username": user.username} for user in users]}

@app.get("/api/files")

async def get\_files(db: Session = Depends(get\_db)):

"""Получает список файлов"""

files = db.query(models.File).all()

return {"files": [{"id": str(file.id), "path": file.file\_path, "name": file.file\_name} for file in files]}

@app.get("/api/sessions")

async def get\_sessions(db: Session = Depends(get\_db)):

"""Получает список сессий"""

sessions = db.query(models.FileSession).all()

return {"sessions": [

{

"id": str(session.id),

"user\_id": str(session.user\_id),

"file\_id": str(session.file\_id),

"started\_at": session.started\_at.isoformat(),

"ended\_at": session.ended\_at.isoformat() if session.ended\_at else None,

"last\_activity": session.last\_activity.isoformat(),

"resume\_count": session.resume\_count,

"is\_commented": session.is\_commented

} for session in sessions

]}

@app.get("/api/events")

async def get\_events(db: Session = Depends(get\_db)):

"""Получает список событий"""

events = db.query(models.FileEvent).all()

return {"events": [

{

"id": str(event.id),

"session\_id": str(event.session\_id),

"event\_type": event.event\_type,

"file\_hash": event.file\_hash,

"timestamp": event.event\_timestamp.isoformat()

} for event in events

]}

# НОВЫЕ ЭНДПОИНТЫ ДЛЯ КОММЕНТАРИЕВ

@app.post("/api/comments", response\_model=schemas.Comment)

async def create\_comment(comment: schemas.CommentCreate, db: Session = Depends(get\_db)):

"""Создает комментарий для сессии"""

try:

# Проверяем существование сессии

session = crud.get\_file\_session(db, comment.session\_id)

if not session:

raise HTTPException(status\_code=404, detail="Session not found")

# Проверяем существование пользователя

user = crud.get\_user(db, comment.user\_id)

if not user:

raise HTTPException(status\_code=404, detail="User not found")

# Проверяем, нет ли уже комментария для этой сессии

existing\_comment = crud.get\_comment\_by\_session(db, comment.session\_id)

if existing\_comment:

raise HTTPException(status\_code=400, detail="Comment already exists for this session")

session.ended\_at = datetime.utcnow()

session.is\_commented = True

# Создаем комментарий

db\_comment = crud.create\_comment(db, comment)

db.commit()

return db\_comment

except HTTPException:

raise

except Exception as e:

db.rollback()

raise HTTPException(status\_code=500, detail=f"Error creating comment: {str(e)}")

@app.get("/api/comments", response\_model=List[schemas.CommentWithUser])

async def get\_comments(

skip: int = 0,

limit: int = 100,

change\_type: Optional[str] = None,

db: Session = Depends(get\_db)

):

"""Получает список комментариев с возможностью фильтрации по типу изменения"""

try:

# Базовый запрос с join пользователя

query = db.query(

models.Comment,

models.User.username

).join(

models.User, models.Comment.user\_id == models.User.id

)

# Фильтрация по типу изменения если указан

if change\_type:

query = query.filter(models.Comment.change\_type == change\_type)

comments = query.offset(skip).limit(limit).all()

# Преобразуем в схему

result = []

for comment, username in comments:

result.append(schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

))

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comments: {str(e)}")

@app.get("/api/comments/{session\_id}", response\_model=schemas.CommentWithUser)

async def get\_comment\_by\_session(session\_id: str, db: Session = Depends(get\_db)):

"""Получает комментарий по ID сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_comment\_with\_user(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Comment not found")

comment, username = result

return schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comment: {str(e)}")

@app.get("/api/sessions-with-comments", response\_model=List[schemas.SessionWithDetails])

async def get\_sessions\_with\_comments(

skip: int = 0,

limit: int = 100,

db: Session = Depends(get\_db)

):

"""Получает сессии с комментариями"""

try:

sessions = crud.get\_sessions\_with\_comments(db, skip=skip, limit=limit)

result = []

for session, file\_path, file\_name, username, comment in sessions:

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

result.append(session\_details)

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting sessions: {str(e)}")

@app.get("/api/change-types")

async def get\_change\_types():

"""Возвращает доступные типы изменений"""

return {

"change\_types": [

"technical\_changes", # Изменение технических решений

"design\_changes", # Изменение оформления/дизайна

"content\_changes", # Изменение содержания

"bug\_fixes", # Исправление ошибок

"optimization", # Оптимизация

"refactoring", # Рефакторинг

"new\_feature", # Новая функциональность

"documentation", # Изменение документации

"other" # Другое

]

}

@app.get("/api/sessions/{session\_id}/details", response\_model=schemas.SessionWithDetails)

async def get\_session\_details(session\_id: str, db: Session = Depends(get\_db)):

"""Получает детальную информацию о сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_session\_with\_details(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Session not found")

session, file\_path, file\_name, username, comment = result

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

return session\_details

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting session details: {str(e)}")

if \_\_name\_\_ == "\_\_main\_\_":

import uvicorn

uvicorn.run(app, host="0.0.0.0", port=8000) # main.py (сервер)

import sys

import os

# Добавляем корень проекта в PYTHONPATH

current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

project\_root = os.path.dirname(os.path.dirname(current\_dir)) # Поднимаемся на уровень выше

sys.path.insert(0, project\_root)

from fastapi import FastAPI, Depends, HTTPException

from sqlalchemy.orm import Session

from shared.config\_loader import get\_api\_config

from . import database, models, crud, schemas

from typing import List, Optional

import uuid

from datetime import datetime

# Загружаем конфигурацию API

api\_config = get\_api\_config()

app = FastAPI(

title="File Monitoring API",

description="API для отслеживания изменений файлов с системой комментариев",

version="1.0.0"

)

# Dependency

def get\_db():

db = database.SessionLocal()

try:

yield db

finally:

db.close()

@app.get("/")

async def root():

return {

"message": "File Monitoring System API",

"status": "running",

"config\_source": "YAML"

}

@app.get("/health")

async def health\_check():

return {"status": "healthy", "service": "session-service"}

@app.get("/config")

async def show\_config():

"""Показывает текущую конфигурацию (без паролей)"""

return {

"api\_host": api\_config.get('host'),

"api\_port": api\_config.get('port'),

"environment": "development"

}

# Events endpoint для мониторинга

@app.post("/api/events")

async def create\_event(event\_data: dict, db: Session = Depends(get\_db)):

"""Эндпоинт для приема событий от мониторинга"""

print(f"📨 Received event: {event\_data}")

try:

# Обрабатываем событие в зависимости от типа

result = await process\_file\_event(db, event\_data)

return {"status": "processed", "event\_type": event\_data.get('event\_type'), "result": result}

except Exception as e:

print(f"❌ Error processing event: {e}")

raise HTTPException(status\_code=500, detail=f"Error processing event: {str(e)}")

async def process\_file\_event(db: Session, event\_data: dict):

"""Обрабатывает файловое событие и создает/обновляет сессии"""

event\_type = event\_data.get('event\_type')

file\_path = event\_data.get('file\_path')

file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

file\_hash = event\_data.get('file\_hash')

session\_id = event\_data.get('session\_id')

resume\_count = event\_data.get('resume\_count', 0)

print(f"🔧 Processing {event\_type} for {file\_path} (user: {username}, session: {session\_id}, resume: {resume\_count})")

# Обрабатываем перемещение отдельно

if event\_type == 'moved':

return await handle\_file\_moved(db, event\_data, file\_hash)

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# Обрабатываем в зависимости от типа события

if event\_type == 'created':

return await handle\_file\_created(db, user.id, file.id, event\_data, file\_hash, resume\_count)

elif event\_type == 'modified':

return await handle\_file\_modified(db, user.id, file.id, event\_data, file\_hash, session\_id, resume\_count)

elif event\_type == 'deleted':

return await handle\_file\_deleted(db, user.id, file.id, event\_data, session\_id)

elif event\_type == 'closed':

return await handle\_file\_closed(db, session\_id, file\_hash, event\_data)

else:

raise ValueError(f"Unknown event type: {event\_type}")

async def handle\_file\_moved(db: Session, event\_data: dict, file\_hash: str = None):

"""Обрабатывает перемещение/переименование файла"""

old\_file\_path = event\_data.get('old\_file\_path')

new\_file\_path = event\_data.get('file\_path')

new\_file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

print(f"🔄 Processing file move: {old\_file\_path} -> {new\_file\_path}")

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Ищем активную сессию для старого пути

old\_file = crud.get\_file\_by\_path(db, old\_file\_path)

if old\_file:

# Находим активные сессии для этого файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, old\_file.id)

if active\_sessions:

# Берем последнюю активную сессию

session = active\_sessions[0]

# Обновляем файл с новым путем

old\_file.file\_path = new\_file\_path

old\_file.file\_name = new\_file\_name

db.commit()

# Обновляем время активности сессии

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Moved session {session.id} from {old\_file\_path} to {new\_file\_path}")

return {"action": "session\_moved", "session\_id": str(session.id)}

# Если не нашли активную сессию, создаем новую

file\_data = schemas.FileCreate(file\_path=new\_file\_path, file\_name=new\_file\_name)

file = crud.create\_file(db, file\_data)

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"✅ Created new session {session.id} for moved file {new\_file\_path}")

return {"action": "session\_created", "session\_id": str(session.id)}

async def handle\_file\_created(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, resume\_count: int = 0):

"""Обрабатывает создание файла - создает новую сессию"""

# ПРОВЕРКА ДУБЛИРОВАНИЯ СЕССИЙ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

print(f"⚠️ Active session already exists for file, updating instead of creating new")

session = active\_sessions[0]

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_before = file\_hash

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

# Создаем событие created

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Updated existing session {session.id} for created file")

return {"action": "session\_updated", "session\_id": str(session.id), "resumed": False}

# Проверяем, есть ли недавно закрытая сессия для возобновления

recent\_session = None

if resume\_count > 0:

recent\_session = crud.get\_recent\_closed\_session(db, user\_id, file\_id)

if recent\_session:

# Возобновляем существующую сессию

recent\_session.ended\_at = None

recent\_session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

recent\_session.resume\_count = resume\_count

recent\_session.hash\_before = file\_hash

db.commit()

db.refresh(recent\_session)

print(f"🔄 Resumed session {recent\_session.id} (resume count: {resume\_count})")

session = recent\_session

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for file {event\_data.get('file\_path')}")

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": "session\_created", "session\_id": str(session.id), "resumed": recent\_session is not None}

async def handle\_file\_modified(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, session\_id: str = None, resume\_count: int = 0):

"""Обрабатывает изменение файла - обновляет существующую сессию или создает новую"""

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

if session:

# Обновляем существующую сессию

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

if resume\_count > session.resume\_count:

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

print(f"📝 Updated session {session.id} for file {event\_data.get('file\_path')} (resume: {resume\_count})")

action = "session\_updated"

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created new session {session.id} for modified file {event\_data.get('file\_path')}")

action = "session\_created"

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='modified',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": action, "session\_id": str(session.id)}

async def handle\_file\_deleted(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, session\_id: str = None):

"""Обрабатывает удаление файла - закрывает сессию и создает событие deleted"""

print(f"🗑️ Processing file deletion for file\_id: {file\_id}, session\_id: {session\_id}")

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

print(f"🔍 Found active session for deletion: {session.id}")

if session:

# Закрываем сессию с временем из события

session.ended\_at = datetime.fromisoformat(event\_data.get('event\_timestamp'))

db.commit()

db.refresh(session)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None, # Для удаленного файла хеша нет

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Closed session {session.id} for deleted file and created deleted event")

return {"action": "session\_closed", "session\_id": str(session.id)}

else:

print(f"⚠️ No active session found for deleted file, creating standalone deleted event")

# Если сессии нет, все равно создаем событие deleted

# Создаем временную сессию для привязки события

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

ended\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')), # Сразу закрываем

hash\_before=None,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Created standalone deleted event with session {session.id}")

return {"action": "deleted\_event\_created", "session\_id": str(session.id)}

async def handle\_file\_closed(db: Session, session\_id: str, file\_hash: str = None, event\_data: dict = None):

"""Обрабатывает закрытие файла - закрывает сессию с правильным ended\_at"""

if session\_id:

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

if session:

# ОБНОВЛЯЕМ СЕССИЮ С ПРАВИЛЬНЫМ ended\_at

if event\_data and 'event\_timestamp' in event\_data:

# Используем время из события закрытия

ended\_at = datetime.fromisoformat(event\_data['event\_timestamp'])

else:

# Используем текущее время

ended\_at = datetime.utcnow()

session.ended\_at = ended\_at

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие закрытия

event\_timestamp = ended\_at # Используем то же время что и для ended\_at

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='closed',

file\_hash=file\_hash,

event\_timestamp=event\_timestamp

)

crud.create\_file\_event(db, event\_record)

# Получаем информацию о файле для логов

file = crud.get\_file(db, session.file\_id)

user = crud.get\_user(db, session.user\_id)

duration = session.ended\_at - session.started\_at

print(f"🔒 Closed session {session.id} for file {file.file\_path} (user: {user.username}, duration: {duration}, ended\_at: {session.ended\_at})")

return {

"action": "session\_closed",

"session\_id": str(session.id),

"duration": str(duration),

"ended\_at": session.ended\_at.isoformat(),

"resume\_count": session.resume\_count

}

except ValueError:

pass

return {"action": "no\_session\_found"}

# Новый эндпоинт для создания сессий (для совместимости с мониторингом)

@app.post("/api/sessions")

async def create\_session(session\_data: dict, db: Session = Depends(get\_db)):

"""Создает сессию на сервере и возвращает session\_id"""

try:

print(f"📝 Creating session: {session\_data}")

# Получаем или создаем пользователя

username = session\_data.get('username')

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file\_path = session\_data.get('file\_path')

file\_name = session\_data.get('file\_name', os.path.basename(file\_path))

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# ПРОВЕРКА ДУБЛИРОВАНИЯ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, file.id)

if active\_sessions:

print(f"⚠️ Active session already exists, returning existing: {active\_sessions[0].id}")

return {"id": str(active\_sessions[0].id), "status": "existing"}

# Создаем сессию

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.utcnow(),

last\_activity=datetime.utcnow(),

hash\_before=session\_data.get('file\_hash'),

resume\_count=session\_data.get('resume\_count', 0)

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for {file\_path}")

return {"id": str(session.id), "status": "created"}

except Exception as e:

print(f"❌ Error creating session: {e}")

raise HTTPException(status\_code=500, detail=f"Error creating session: {str(e)}")

# Простые эндпоинты для проверки данных

@app.get("/api/users")

async def get\_users(db: Session = Depends(get\_db)):

"""Получает список пользователей"""

users = db.query(models.User).all()

return {"users": [{"id": str(user.id), "username": user.username} for user in users]}

@app.get("/api/files")

async def get\_files(db: Session = Depends(get\_db)):

"""Получает список файлов"""

files = db.query(models.File).all()

return {"files": [{"id": str(file.id), "path": file.file\_path, "name": file.file\_name} for file in files]}

@app.get("/api/sessions")

async def get\_sessions(db: Session = Depends(get\_db)):

"""Получает список сессий"""

sessions = db.query(models.FileSession).all()

return {"sessions": [

{

"id": str(session.id),

"user\_id": str(session.user\_id),

"file\_id": str(session.file\_id),

"started\_at": session.started\_at.isoformat(),

"ended\_at": session.ended\_at.isoformat() if session.ended\_at else None,

"last\_activity": session.last\_activity.isoformat(),

"resume\_count": session.resume\_count,

"is\_commented": session.is\_commented

} for session in sessions

]}

@app.get("/api/events")

async def get\_events(db: Session = Depends(get\_db)):

"""Получает список событий"""

events = db.query(models.FileEvent).all()

return {"events": [

{

"id": str(event.id),

"session\_id": str(event.session\_id),

"event\_type": event.event\_type,

"file\_hash": event.file\_hash,

"timestamp": event.event\_timestamp.isoformat()

} for event in events

]}

# НОВЫЕ ЭНДПОИНТЫ ДЛЯ КОММЕНТАРИЕВ

@app.post("/api/comments", response\_model=schemas.Comment)

async def create\_comment(comment: schemas.CommentCreate, db: Session = Depends(get\_db)):

"""Создает комментарий для сессии"""

try:

# Проверяем существование сессии

session = crud.get\_file\_session(db, comment.session\_id)

if not session:

raise HTTPException(status\_code=404, detail="Session not found")

# Проверяем существование пользователя

user = crud.get\_user(db, comment.user\_id)

if not user:

raise HTTPException(status\_code=404, detail="User not found")

# Проверяем, нет ли уже комментария для этой сессии

existing\_comment = crud.get\_comment\_by\_session(db, comment.session\_id)

if existing\_comment:

raise HTTPException(status\_code=400, detail="Comment already exists for this session")

session.ended\_at = datetime.utcnow()

session.is\_commented = True

# Создаем комментарий

db\_comment = crud.create\_comment(db, comment)

db.commit()

return db\_comment

except HTTPException:

raise

except Exception as e:

db.rollback()

raise HTTPException(status\_code=500, detail=f"Error creating comment: {str(e)}")

@app.get("/api/comments", response\_model=List[schemas.CommentWithUser])

async def get\_comments(

skip: int = 0,

limit: int = 100,

change\_type: Optional[str] = None,

db: Session = Depends(get\_db)

):

"""Получает список комментариев с возможностью фильтрации по типу изменения"""

try:

# Базовый запрос с join пользователя

query = db.query(

models.Comment,

models.User.username

).join(

models.User, models.Comment.user\_id == models.User.id

)

# Фильтрация по типу изменения если указан

if change\_type:

query = query.filter(models.Comment.change\_type == change\_type)

comments = query.offset(skip).limit(limit).all()

# Преобразуем в схему

result = []

for comment, username in comments:

result.append(schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

))

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comments: {str(e)}")

@app.get("/api/comments/{session\_id}", response\_model=schemas.CommentWithUser)

async def get\_comment\_by\_session(session\_id: str, db: Session = Depends(get\_db)):

"""Получает комментарий по ID сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_comment\_with\_user(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Comment not found")

comment, username = result

return schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comment: {str(e)}")

@app.get("/api/sessions-with-comments", response\_model=List[schemas.SessionWithDetails])

async def get\_sessions\_with\_comments(

skip: int = 0,

limit: int = 100,

db: Session = Depends(get\_db)

):

"""Получает сессии с комментариями"""

try:

sessions = crud.get\_sessions\_with\_comments(db, skip=skip, limit=limit)

result = []

for session, file\_path, file\_name, username, comment in sessions:

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

result.append(session\_details)

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting sessions: {str(e)}")

@app.get("/api/change-types")

async def get\_change\_types():

"""Возвращает доступные типы изменений"""

return {

"change\_types": [

"technical\_changes", # Изменение технических решений

"design\_changes", # Изменение оформления/дизайна

"content\_changes", # Изменение содержания

"bug\_fixes", # Исправление ошибок

"optimization", # Оптимизация

"refactoring", # Рефакторинг

"new\_feature", # Новая функциональность

"documentation", # Изменение документации

"other" # Другое

]

}

@app.get("/api/sessions/{session\_id}/details", response\_model=schemas.SessionWithDetails)

async def get\_session\_details(session\_id: str, db: Session = Depends(get\_db)):

"""Получает детальную информацию о сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_session\_with\_details(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Session not found")

session, file\_path, file\_name, username, comment = result

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

return session\_details

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting session details: {str(e)}")

if \_\_name\_\_ == "\_\_main\_\_":

import uvicorn

uvicorn.run(app, host="0.0.0.0", port=8000) # main.py (сервер)

import sys

import os

# Добавляем корень проекта в PYTHONPATH

current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

project\_root = os.path.dirname(os.path.dirname(current\_dir)) # Поднимаемся на уровень выше

sys.path.insert(0, project\_root)

from fastapi import FastAPI, Depends, HTTPException

from sqlalchemy.orm import Session

from shared.config\_loader import get\_api\_config

from . import database, models, crud, schemas

from typing import List, Optional

import uuid

from datetime import datetime

# Загружаем конфигурацию API

api\_config = get\_api\_config()

app = FastAPI(

title="File Monitoring API",

description="API для отслеживания изменений файлов с системой комментариев",

version="1.0.0"

)

# Dependency

def get\_db():

db = database.SessionLocal()

try:

yield db

finally:

db.close()

@app.get("/")

async def root():

return {

"message": "File Monitoring System API",

"status": "running",

"config\_source": "YAML"

}

@app.get("/health")

async def health\_check():

return {"status": "healthy", "service": "session-service"}

@app.get("/config")

async def show\_config():

"""Показывает текущую конфигурацию (без паролей)"""

return {

"api\_host": api\_config.get('host'),

"api\_port": api\_config.get('port'),

"environment": "development"

}

# Events endpoint для мониторинга

@app.post("/api/events")

async def create\_event(event\_data: dict, db: Session = Depends(get\_db)):

"""Эндпоинт для приема событий от мониторинга"""

print(f"📨 Received event: {event\_data}")

try:

# Обрабатываем событие в зависимости от типа

result = await process\_file\_event(db, event\_data)

return {"status": "processed", "event\_type": event\_data.get('event\_type'), "result": result}

except Exception as e:

print(f"❌ Error processing event: {e}")

raise HTTPException(status\_code=500, detail=f"Error processing event: {str(e)}")

async def process\_file\_event(db: Session, event\_data: dict):

"""Обрабатывает файловое событие и создает/обновляет сессии"""

event\_type = event\_data.get('event\_type')

file\_path = event\_data.get('file\_path')

file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

file\_hash = event\_data.get('file\_hash')

session\_id = event\_data.get('session\_id')

resume\_count = event\_data.get('resume\_count', 0)

print(f"🔧 Processing {event\_type} for {file\_path} (user: {username}, session: {session\_id}, resume: {resume\_count})")

# Обрабатываем перемещение отдельно

if event\_type == 'moved':

return await handle\_file\_moved(db, event\_data, file\_hash)

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# Обрабатываем в зависимости от типа события

if event\_type == 'created':

return await handle\_file\_created(db, user.id, file.id, event\_data, file\_hash, resume\_count)

elif event\_type == 'modified':

return await handle\_file\_modified(db, user.id, file.id, event\_data, file\_hash, session\_id, resume\_count)

elif event\_type == 'deleted':

return await handle\_file\_deleted(db, user.id, file.id, event\_data, session\_id)

elif event\_type == 'closed':

return await handle\_file\_closed(db, session\_id, file\_hash, event\_data)

else:

raise ValueError(f"Unknown event type: {event\_type}")

async def handle\_file\_moved(db: Session, event\_data: dict, file\_hash: str = None):

"""Обрабатывает перемещение/переименование файла"""

old\_file\_path = event\_data.get('old\_file\_path')

new\_file\_path = event\_data.get('file\_path')

new\_file\_name = event\_data.get('file\_name')

username = event\_data.get('user\_id')

print(f"🔄 Processing file move: {old\_file\_path} -> {new\_file\_path}")

# Получаем или создаем пользователя

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Ищем активную сессию для старого пути

old\_file = crud.get\_file\_by\_path(db, old\_file\_path)

if old\_file:

# Находим активные сессии для этого файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, old\_file.id)

if active\_sessions:

# Берем последнюю активную сессию

session = active\_sessions[0]

# Обновляем файл с новым путем

old\_file.file\_path = new\_file\_path

old\_file.file\_name = new\_file\_name

db.commit()

# Обновляем время активности сессии

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Moved session {session.id} from {old\_file\_path} to {new\_file\_path}")

return {"action": "session\_moved", "session\_id": str(session.id)}

# Если не нашли активную сессию, создаем новую

file\_data = schemas.FileCreate(file\_path=new\_file\_path, file\_name=new\_file\_name)

file = crud.create\_file(db, file\_data)

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие перемещения

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='moved',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"✅ Created new session {session.id} for moved file {new\_file\_path}")

return {"action": "session\_created", "session\_id": str(session.id)}

async def handle\_file\_created(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, resume\_count: int = 0):

"""Обрабатывает создание файла - создает новую сессию"""

# ПРОВЕРКА ДУБЛИРОВАНИЯ СЕССИЙ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

print(f"⚠️ Active session already exists for file, updating instead of creating new")

session = active\_sessions[0]

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_before = file\_hash

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

# Создаем событие created

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🔄 Updated existing session {session.id} for created file")

return {"action": "session\_updated", "session\_id": str(session.id), "resumed": False}

# Проверяем, есть ли недавно закрытая сессия для возобновления

recent\_session = None

if resume\_count > 0:

recent\_session = crud.get\_recent\_closed\_session(db, user\_id, file\_id)

if recent\_session:

# Возобновляем существующую сессию

recent\_session.ended\_at = None

recent\_session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

recent\_session.resume\_count = resume\_count

recent\_session.hash\_before = file\_hash

db.commit()

db.refresh(recent\_session)

print(f"🔄 Resumed session {recent\_session.id} (resume count: {resume\_count})")

session = recent\_session

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for file {event\_data.get('file\_path')}")

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='created',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": "session\_created", "session\_id": str(session.id), "resumed": recent\_session is not None}

async def handle\_file\_modified(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, file\_hash: str = None, session\_id: str = None, resume\_count: int = 0):

"""Обрабатывает изменение файла - обновляет существующую сессию или создает новую"""

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

if session:

# Обновляем существующую сессию

session.last\_activity = datetime.fromisoformat(event\_data.get('event\_timestamp'))

if file\_hash:

session.hash\_after = file\_hash

if resume\_count > session.resume\_count:

session.resume\_count = resume\_count

db.commit()

db.refresh(session)

print(f"📝 Updated session {session.id} for file {event\_data.get('file\_path')} (resume: {resume\_count})")

action = "session\_updated"

else:

# Создаем новую сессию

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

hash\_before=file\_hash,

resume\_count=resume\_count

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created new session {session.id} for modified file {event\_data.get('file\_path')}")

action = "session\_created"

# Создаем событие файла

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='modified',

file\_hash=file\_hash,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

return {"action": action, "session\_id": str(session.id)}

async def handle\_file\_deleted(db: Session, user\_id: uuid.UUID, file\_id: uuid.UUID, event\_data: dict, session\_id: str = None):

"""Обрабатывает удаление файла - закрывает сессию и создает событие deleted"""

print(f"🗑️ Processing file deletion for file\_id: {file\_id}, session\_id: {session\_id}")

session = None

if session\_id:

# Пытаемся найти существующую сессию

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

except ValueError:

pass # Невалидный UUID

if not session:

# Ищем активную сессию для этого пользователя и файла

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user\_id, file\_id)

if active\_sessions:

session = active\_sessions[0]

print(f"🔍 Found active session for deletion: {session.id}")

if session:

# Закрываем сессию с временем из события

session.ended\_at = datetime.fromisoformat(event\_data.get('event\_timestamp'))

db.commit()

db.refresh(session)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None, # Для удаленного файла хеша нет

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Closed session {session.id} for deleted file and created deleted event")

return {"action": "session\_closed", "session\_id": str(session.id)}

else:

print(f"⚠️ No active session found for deleted file, creating standalone deleted event")

# Если сессии нет, все равно создаем событие deleted

# Создаем временную сессию для привязки события

session\_data = schemas.FileSessionCreate(

user\_id=user\_id,

file\_id=file\_id,

started\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')),

last\_activity=datetime.fromisoformat(event\_data.get('event\_timestamp')),

ended\_at=datetime.fromisoformat(event\_data.get('event\_timestamp')), # Сразу закрываем

hash\_before=None,

resume\_count=0

)

session = crud.create\_file\_session(db, session\_data)

# Создаем событие удаления

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='deleted',

file\_hash=None,

event\_timestamp=datetime.fromisoformat(event\_data.get('event\_timestamp'))

)

crud.create\_file\_event(db, event\_record)

print(f"🗑️ Created standalone deleted event with session {session.id}")

return {"action": "deleted\_event\_created", "session\_id": str(session.id)}

async def handle\_file\_closed(db: Session, session\_id: str, file\_hash: str = None, event\_data: dict = None):

"""Обрабатывает закрытие файла - закрывает сессию с правильным ended\_at"""

if session\_id:

try:

session\_uuid = uuid.UUID(session\_id)

session = crud.get\_file\_session(db, session\_uuid)

if session:

# ОБНОВЛЯЕМ СЕССИЮ С ПРАВИЛЬНЫМ ended\_at

if event\_data and 'event\_timestamp' in event\_data:

# Используем время из события закрытия

ended\_at = datetime.fromisoformat(event\_data['event\_timestamp'])

else:

# Используем текущее время

ended\_at = datetime.utcnow()

session.ended\_at = ended\_at

if file\_hash:

session.hash\_after = file\_hash

db.commit()

# Создаем событие закрытия

event\_timestamp = ended\_at # Используем то же время что и для ended\_at

event\_record = schemas.FileEventCreate(

session\_id=session.id,

event\_type='closed',

file\_hash=file\_hash,

event\_timestamp=event\_timestamp

)

crud.create\_file\_event(db, event\_record)

# Получаем информацию о файле для логов

file = crud.get\_file(db, session.file\_id)

user = crud.get\_user(db, session.user\_id)

duration = session.ended\_at - session.started\_at

print(f"🔒 Closed session {session.id} for file {file.file\_path} (user: {user.username}, duration: {duration}, ended\_at: {session.ended\_at})")

return {

"action": "session\_closed",

"session\_id": str(session.id),

"duration": str(duration),

"ended\_at": session.ended\_at.isoformat(),

"resume\_count": session.resume\_count

}

except ValueError:

pass

return {"action": "no\_session\_found"}

# Новый эндпоинт для создания сессий (для совместимости с мониторингом)

@app.post("/api/sessions")

async def create\_session(session\_data: dict, db: Session = Depends(get\_db)):

"""Создает сессию на сервере и возвращает session\_id"""

try:

print(f"📝 Creating session: {session\_data}")

# Получаем или создаем пользователя

username = session\_data.get('username')

user = crud.get\_user\_by\_username(db, username)

if not user:

user\_data = schemas.UserCreate(username=username, email=f"{username}@example.com")

user = crud.create\_user(db, user\_data)

# Получаем или создаем файл

file\_path = session\_data.get('file\_path')

file\_name = session\_data.get('file\_name', os.path.basename(file\_path))

file = crud.get\_file\_by\_path(db, file\_path)

if not file:

file\_data = schemas.FileCreate(file\_path=file\_path, file\_name=file\_name)

file = crud.create\_file(db, file\_data)

# ПРОВЕРКА ДУБЛИРОВАНИЯ - ДОБАВЛЕНО

active\_sessions = crud.get\_active\_sessions\_by\_user\_and\_file(db, user.id, file.id)

if active\_sessions:

print(f"⚠️ Active session already exists, returning existing: {active\_sessions[0].id}")

return {"id": str(active\_sessions[0].id), "status": "existing"}

# Создаем сессию

session\_data = schemas.FileSessionCreate(

user\_id=user.id,

file\_id=file.id,

started\_at=datetime.utcnow(),

last\_activity=datetime.utcnow(),

hash\_before=session\_data.get('file\_hash'),

resume\_count=session\_data.get('resume\_count', 0)

)

session = crud.create\_file\_session(db, session\_data)

print(f"✅ Created session {session.id} for {file\_path}")

return {"id": str(session.id), "status": "created"}

except Exception as e:

print(f"❌ Error creating session: {e}")

raise HTTPException(status\_code=500, detail=f"Error creating session: {str(e)}")

# Простые эндпоинты для проверки данных

@app.get("/api/users")

async def get\_users(db: Session = Depends(get\_db)):

"""Получает список пользователей"""

users = db.query(models.User).all()

return {"users": [{"id": str(user.id), "username": user.username} for user in users]}

@app.get("/api/files")

async def get\_files(db: Session = Depends(get\_db)):

"""Получает список файлов"""

files = db.query(models.File).all()

return {"files": [{"id": str(file.id), "path": file.file\_path, "name": file.file\_name} for file in files]}

@app.get("/api/sessions")

async def get\_sessions(db: Session = Depends(get\_db)):

"""Получает список сессий"""

sessions = db.query(models.FileSession).all()

return {"sessions": [

{

"id": str(session.id),

"user\_id": str(session.user\_id),

"file\_id": str(session.file\_id),

"started\_at": session.started\_at.isoformat(),

"ended\_at": session.ended\_at.isoformat() if session.ended\_at else None,

"last\_activity": session.last\_activity.isoformat(),

"resume\_count": session.resume\_count,

"is\_commented": session.is\_commented

} for session in sessions

]}

@app.get("/api/events")

async def get\_events(db: Session = Depends(get\_db)):

"""Получает список событий"""

events = db.query(models.FileEvent).all()

return {"events": [

{

"id": str(event.id),

"session\_id": str(event.session\_id),

"event\_type": event.event\_type,

"file\_hash": event.file\_hash,

"timestamp": event.event\_timestamp.isoformat()

} for event in events

]}

# НОВЫЕ ЭНДПОИНТЫ ДЛЯ КОММЕНТАРИЕВ

@app.post("/api/comments", response\_model=schemas.Comment)

async def create\_comment(comment: schemas.CommentCreate, db: Session = Depends(get\_db)):

"""Создает комментарий для сессии"""

try:

# Проверяем существование сессии

session = crud.get\_file\_session(db, comment.session\_id)

if not session:

raise HTTPException(status\_code=404, detail="Session not found")

# Проверяем существование пользователя

user = crud.get\_user(db, comment.user\_id)

if not user:

raise HTTPException(status\_code=404, detail="User not found")

# Проверяем, нет ли уже комментария для этой сессии

existing\_comment = crud.get\_comment\_by\_session(db, comment.session\_id)

if existing\_comment:

raise HTTPException(status\_code=400, detail="Comment already exists for this session")

session.ended\_at = datetime.utcnow()

session.is\_commented = True

# Создаем комментарий

db\_comment = crud.create\_comment(db, comment)

db.commit()

return db\_comment

except HTTPException:

raise

except Exception as e:

db.rollback()

raise HTTPException(status\_code=500, detail=f"Error creating comment: {str(e)}")

@app.get("/api/comments", response\_model=List[schemas.CommentWithUser])

async def get\_comments(

skip: int = 0,

limit: int = 100,

change\_type: Optional[str] = None,

db: Session = Depends(get\_db)

):

"""Получает список комментариев с возможностью фильтрации по типу изменения"""

try:

# Базовый запрос с join пользователя

query = db.query(

models.Comment,

models.User.username

).join(

models.User, models.Comment.user\_id == models.User.id

)

# Фильтрация по типу изменения если указан

if change\_type:

query = query.filter(models.Comment.change\_type == change\_type)

comments = query.offset(skip).limit(limit).all()

# Преобразуем в схему

result = []

for comment, username in comments:

result.append(schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

))

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comments: {str(e)}")

@app.get("/api/comments/{session\_id}", response\_model=schemas.CommentWithUser)

async def get\_comment\_by\_session(session\_id: str, db: Session = Depends(get\_db)):

"""Получает комментарий по ID сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_comment\_with\_user(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Comment not found")

comment, username = result

return schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting comment: {str(e)}")

@app.get("/api/sessions-with-comments", response\_model=List[schemas.SessionWithDetails])

async def get\_sessions\_with\_comments(

skip: int = 0,

limit: int = 100,

db: Session = Depends(get\_db)

):

"""Получает сессии с комментариями"""

try:

sessions = crud.get\_sessions\_with\_comments(db, skip=skip, limit=limit)

result = []

for session, file\_path, file\_name, username, comment in sessions:

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

result.append(session\_details)

return result

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting sessions: {str(e)}")

@app.get("/api/change-types")

async def get\_change\_types():

"""Возвращает доступные типы изменений"""

return {

"change\_types": [

"technical\_changes", # Изменение технических решений

"design\_changes", # Изменение оформления/дизайна

"content\_changes", # Изменение содержания

"bug\_fixes", # Исправление ошибок

"optimization", # Оптимизация

"refactoring", # Рефакторинг

"new\_feature", # Новая функциональность

"documentation", # Изменение документации

"other" # Другое

]

}

@app.get("/api/sessions/{session\_id}/details", response\_model=schemas.SessionWithDetails)

async def get\_session\_details(session\_id: str, db: Session = Depends(get\_db)):

"""Получает детальную информацию о сессии"""

try:

session\_uuid = uuid.UUID(session\_id)

result = crud.get\_session\_with\_details(db, session\_uuid)

if not result:

raise HTTPException(status\_code=404, detail="Session not found")

session, file\_path, file\_name, username, comment = result

session\_details = schemas.SessionWithDetails(

id=session.id,

started\_at=session.started\_at,

ended\_at=session.ended\_at,

last\_activity=session.last\_activity,

resume\_count=session.resume\_count,

file\_path=file\_path,

file\_name=file\_name,

username=username,

comment=None

)

if comment:

session\_details.comment = schemas.CommentWithUser(

id=comment.id,

content=comment.content,

change\_type=comment.change\_type,

created\_at=comment.created\_at,

username=username

)

return session\_details

except ValueError:

raise HTTPException(status\_code=400, detail="Invalid session ID format")

except HTTPException:

raise

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error getting session details: {str(e)}")

if \_\_name\_\_ == "\_\_main\_\_":

import uvicorn

uvicorn.run(app, host="0.0.0.0", port=8000)