# file\_validator.py

import os

import fnmatchimport os

import getpass

import platform

from datetime import datetime, timedelta

from typing import Dict, Any, Optional

from shareimport time

from datetime import datetime

from watchdog.observers import Observer

from watchdog.observers.polling import PollingObserver

from watchdog.events import FileSystemEventHandler

from shared.logger import setup\_logger

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

from .event\_handler import EventHandler

from .background\_checker import BackgroundSessionChecker

class FileMonitorHandler(FileSystemEventHandler):

def \_\_init\_\_(self, event\_handler: EventHandler):

self.event\_handler = event\_handler

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

def on\_created(self, event):

if not event.is\_directory:

self.event\_handler.handle\_file\_event('created', event.src\_path)

def on\_modified(self, event):

if not event.is\_directory:

self.event\_handler.handle\_file\_event('modified', event.src\_path)

def on\_deleted(self, event):

if not event.is\_directory:

self.event\_handler.handle\_file\_event('deleted', event.src\_path)

def on\_moved(self, event):

if not event.is\_directory:

self.event\_handler.handle\_file\_event('moved', event.src\_path, event.dest\_path)

class FileWatcher:

def \_\_init\_\_(self, monitoring\_config=None):

if monitoring\_config is None:

self.config = get\_monitoring\_config()

else:

self.config = monitoring\_config

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

self.use\_polling = self.config.get('use\_polling', False)

if self.use\_polling:

self.observer = PollingObserver()

else:

self.observer = Observer()

# Передаем конфигурацию в EventHandler

self.event\_handler = EventHandler(monitoring\_config=self.config)

self.monitor\_handler = FileMonitorHandler(self.event\_handler)

# Создаем фоновый проверщик сессий

check\_interval = self.config.get('background\_check\_interval', 15)

self.background\_checker = BackgroundSessionChecker(

self.event\_handler,

check\_interval=check\_interval

)

self.logger.info(f"🎯 FileWatcher initialized with background checking every {check\_interval}s")

def start(self):

"""Запускает мониторинг"""

watch\_paths = self.config.get('watch\_paths', ['./monitor'])

for path in watch\_paths:

try:

self.observer.schedule(

self.monitor\_handler,

path,

recursive=True

)

self.logger.info(f"📁 Started monitoring: {path}")

except Exception as e:

self.logger.error(f"❌ Failed to schedule watcher for {path}: {e}")

self.observer.start()

self.logger.info("✅ File monitoring started successfully")

# Проверяем соединение с API

if not self.event\_handler.api\_client.test\_connection():

self.logger.error("❌ Cannot connect to API server")

return

# ЗАПУСКАЕМ ФОНОВУЮ ПРОВЕРКУ

self.background\_checker.start()

# Основной цикл (только для статистики)

try:

stats\_interval = 30 # Логируем статистику каждые 30 секунд

while True:

time.sleep(stats\_interval)

# Логируем статистику

stats = self.event\_handler.get\_stats()

active\_sessions = stats['active\_sessions']

self.logger.info(f"📊 Stats: {active\_sessions} active sessions, {stats.get('expired\_sessions', 0)} expired")

# Детальная информация об активных сессиях

if active\_sessions > 0:

timeout\_minutes = self.event\_handler.session\_manager.config.get('session\_timeout\_minutes', 30)

for session\_key, session\_data in self.event\_handler.session\_manager.active\_sessions.items():

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

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

remaining = (timeout\_minutes \* 60) - time\_since\_activity.total\_seconds()

self.logger.debug(f"⏰ {session\_key}: inactive {time\_since\_activity.total\_seconds():.1f}s (expires in {remaining:.1f}s)")

except KeyboardInterrupt:

self.stop()

except Exception as e:

self.logger.error(f"❌ Unexpected error: {e}")

self.stop()

def stop(self):

"""Останавливает мониторинг"""

self.background\_checker.stop()

self.observer.stop()

self.observer.join()

self.logger.info("🛑 File monitoring stopped")d.logger import setup\_logger

from shared.config\_loimport 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)

# 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 - ИСПРАВЛЕННАЯ ВЕРСИЯ"""

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 - ИСПРАВЛЕННАЯ ВЕРСИЯ"""

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

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

if session\_data:

# Устанавливаем время окончания - ВАЖНО: ДОБАВЛЕНО ВРЕМЯ ЗАКРЫТИЯ

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 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())

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())

}

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, [])ader import get\_monitoring\_config, get\_api\_client\_config

try:

import win32security # For Windows user

except ImportError:

win32security = None

try:

import psutil # For tracking open files

except ImportError:

psutil = None

from .hash\_calculator import HashCalculator

from .session\_manager import SessionManager

from .api\_client import APIClient

from .file\_validator import FileValidator

class EventHandler:

def \_\_init\_\_(self, monitoring\_config=None):

if monitoring\_config is None:

self.config = get\_monitoring\_config()

else:

self.config = monitoring\_config

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

# Инициализация компонентов

self.hash\_calculator = HashCalculator(self.config.get('hashing', {}))

self.session\_manager = SessionManager()

# ПЕРЕДАЕМ КОНФИГУРАЦИЮ СЕССИЙ ПРАВИЛЬНО

session\_config = self.config.get('sessions', {})

self.session\_manager.set\_config(session\_config)

self.api\_client = APIClient()

self.file\_validator = FileValidator(self.config)

# Статистика

self.stats = {

'events\_processed': 0,

'events\_failed': 0,

'sessions\_created': 0,

'sessions\_resumed': 0,

'files\_closed': 0,

'files\_deleted': 0,

'expired\_sessions': 0

}

# Отслеживание открытых файлов

self.open\_files = {} # file\_path -> {username, processes, last\_activity}

# Трекер перемещений файлов

self.file\_renames = {}

# Время последней проверки открытых файлов

self.last\_open\_files\_check = datetime.now()

self.logger.info("EventHandler initialized with centralized config")

def \_normalize\_username(self, username: str) -> str:

"""Нормализует имя пользователя к единому формату"""

if not username:

return getpass.getuser()

# Если имя содержит домен (формат DOMAIN\\username), извлекаем только username

if '\\' in username:

# Разделяем по обратному слешу и берем последнюю часть

parts = username.split('\\')

normalized = parts[-1] # Берем последнюю часть после \\

self.logger.debug(f"Normalized username: {username} -> {normalized}")

return normalized

# Если имя уже в правильном формате, возвращаем как есть

return username

def handle\_file\_event(self, event\_type: str, file\_path: str, dest\_path: str = None) -> bool:

"""Обрабатывает событие файла"""

try:

self.stats['events\_processed'] += 1

self.logger.debug(f"Raw event: {event\_type} - {file\_path} -> {dest\_path}")

# Обрабатываем перемещение как специальный случай

if event\_type == 'moved' and dest\_path:

return self.\_handle\_file\_moved(file\_path, dest\_path)

# Проверяем нужно ли отслеживать файл

if not self.\_should\_process\_file(file\_path, event\_type):

self.logger.debug(f"Ignoring file: {file\_path} (event: {event\_type})")

return True

# Получаем пользователя ОС и НОРМАЛИЗУЕМ его имя

username = self.\_get\_file\_modifier\_safe(file\_path, event\_type)

normalized\_username = self.\_normalize\_username(username)

self.logger.debug(f"User: {username} -> normalized: {normalized\_username}")

# Обновляем отслеживание открытых файлов

if event\_type in ('created', 'modified'):

self.\_update\_open\_file\_tracking(file\_path, normalized\_username, event\_type)

elif event\_type == 'deleted':

# Удаляем из отслеживания открытых файлов

if file\_path in self.open\_files:

del self.open\_files[file\_path]

# Вычисляем хеш если нужно (только для существующих файлов)

file\_hash = None

if (event\_type != 'deleted' and

self.config.get('hashing', {}).get('enabled', True) and

not self.\_is\_temporary\_file(file\_path) and

os.path.exists(file\_path)):

file\_hash = self.hash\_calculator.calculate\_file\_hash\_with\_retry(file\_path)

# Обрабатываем в зависимости от типа события

if event\_type == 'created':

return self.\_handle\_file\_created(file\_path, normalized\_username, file\_hash)

elif event\_type == 'modified':

return self.\_handle\_file\_modified(file\_path, normalized\_username, file\_hash)

elif event\_type == 'deleted':

return self.\_handle\_file\_deleted(file\_path, normalized\_username)

else:

self.logger.warning(f"Unknown event type: {event\_type}")

return False

except Exception as e:

self.stats['events\_failed'] += 1

self.logger.error(f"Error handling {event\_type} event for {file\_path}: {e}")

return False

def check\_expired\_sessions(self):

"""Проверяет и закрывает просроченные сессии"""

try:

self.logger.debug("🔍 Starting expired sessions check...")

expired\_sessions = self.session\_manager.check\_and\_close\_expired\_sessions()

closed\_count = 0

for session\_data in expired\_sessions:

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

username = session\_data['username']

self.stats['expired\_sessions'] += 1

closed\_count += 1

# ПРОВЕРЯЕМ ЧТО ended\_at УСТАНОВЛЕНО

if 'ended\_at' not in session\_data or session\_data['ended\_at'] is None:

self.logger.error(f"❌ Session closed but ended\_at is None for: {file\_path}")

continue

# Вычисляем финальный хеш если файл существует

file\_hash = None

if os.path.exists(file\_path) and self.config.get('hashing', {}).get('enabled', True):

file\_hash = self.hash\_calculator.calculate\_file\_hash\_with\_retry(file\_path)

# Отправляем событие closed для expired сессии

event\_data = {

'file\_path': file\_path,

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

'event\_type': 'closed',

'file\_hash': file\_hash,

'user\_id': username,

'session\_id': session\_data['session\_id'],

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

'session\_duration': (session\_data['ended\_at'] - session\_data['started\_at']).total\_seconds(),

'event\_timestamp': session\_data['ended\_at'].isoformat() # ИСПОЛЬЗУЕМ ВРЕМЯ ЗАКРЫТИЯ СЕССИИ

}

success = self.api\_client.send\_event(event\_data)

if success:

self.logger.info(f"✅ Closed expired session: {file\_path} (ended\_at: {session\_data['ended\_at']})")

else:

self.logger.error(f"❌ Failed to send closed event for: {file\_path}")

return closed\_count

except Exception as e:

self.logger.error(f"❌ Error checking expired sessions: {e}")

return 0

def \_update\_open\_file\_tracking(self, file\_path: str, username: str, event\_type: str):

"""Обновляет информацию об открытых файлах"""

if not psutil:

return

try:

current\_processes = self.\_get\_processes\_using\_file(file\_path)

current\_time = datetime.now()

if current\_processes:

# Файл открыт - обновляем информацию

self.open\_files[file\_path] = {

'username': username,

'processes': current\_processes,

'last\_activity': current\_time,

'last\_checked': current\_time,

'event\_type': event\_type

}

self.logger.debug(f"File {file\_path} is open in {len(current\_processes)} processes")

else:

# Файл больше не открыт - проверяем нужно ли закрыть сессию

if file\_path in self.open\_files:

file\_info = self.open\_files[file\_path]

time\_since\_last\_activity = current\_time - file\_info['last\_activity']

# Закрываем сессию только если прошло достаточно времени с последней активности

if time\_since\_last\_activity > timedelta(seconds=5):

self.logger.info(f"File {file\_path} is no longer open, closing session")

# Вычисляем финальный хеш если файл существует

file\_hash = None

if os.path.exists(file\_path) and self.config.get('hashing', {}).get('enabled', True):

file\_hash = self.hash\_calculator.calculate\_file\_hash\_with\_retry(file\_path)

# Закрываем сессию

self.\_handle\_file\_closed(file\_path, file\_info['username'], file\_hash)

del self.open\_files[file\_path]

self.stats['files\_closed'] += 1

else:

# Обновляем время проверки, но не закрываем сессию

self.open\_files[file\_path]['last\_checked'] = current\_time

except Exception as e:

self.logger.error(f"Error updating open file tracking for {file\_path}: {e}")

def \_get\_processes\_using\_file(self, file\_path: str) -> list:

"""Возвращает список процессов, использующих файл"""

if not psutil:

return []

processes = []

try:

for proc in psutil.process\_iter(['pid', 'name', 'username', 'open\_files']):

try:

open\_files = proc.info.get('open\_files')

if open\_files is None:

continue

for file in open\_files:

if file.path.lower() == file\_path.lower():

# НОРМАЛИЗУЕМ имя пользователя процесса

process\_username = self.\_normalize\_username(proc.info.get('username', 'unknown'))

processes.append({

'pid': proc.pid,

'name': proc.info['name'],

'username': process\_username

})

break

except (psutil.NoSuchProcess, psutil.AccessDenied, FileNotFoundError):

continue

except Exception as e:

self.logger.debug(f"Error getting processes for {file\_path}: {e}")

return processes

def check\_open\_files(self):

"""Периодически проверяет состояние открытых файлов"""

if not psutil:

return

try:

current\_time = datetime.now()

check\_interval = timedelta(seconds=30)

if current\_time - self.last\_open\_files\_check < check\_interval:

return

self.last\_open\_files\_check = current\_time

files\_to\_close = []

for file\_path, file\_info in list(self.open\_files.items()):

# Проверяем, открыт ли файл все еще

current\_processes = self.\_get\_processes\_using\_file(file\_path)

if not current\_processes:

# Файл больше не открыт - проверяем время с последней активности

time\_since\_last\_activity = current\_time - file\_info['last\_activity']

# Закрываем сессию только если прошло достаточно времени

if time\_since\_last\_activity > timedelta(seconds=5):

files\_to\_close.append((file\_path, file\_info))

else:

# Обновляем время проверки

file\_info['last\_checked'] = current\_time

else:

# Файл все еще открыт - обновляем информацию

file\_info['processes'] = current\_processes

file\_info['last\_checked'] = current\_time

# Закрываем сессии для файлов, которые больше не открыты

for file\_path, file\_info in files\_to\_close:

self.logger.info(f"Detected file closure: {file\_path}")

# Вычисляем финальный хеш если файл существует

file\_hash = None

if os.path.exists(file\_path) and self.config.get('hashing', {}).get('enabled', True):

file\_hash = self.hash\_calculator.calculate\_file\_hash\_with\_retry(file\_path)

# Закрываем сессию

self.\_handle\_file\_closed(file\_path, file\_info['username'], file\_hash)

del self.open\_files[file\_path]

self.stats['files\_closed'] += 1

except Exception as e:

self.logger.error(f"Error checking open files: {e}")

def \_handle\_file\_closed(self, file\_path: str, username: str, file\_hash: str) -> bool:

"""Обрабатывает закрытие файла - ИСПРАВЛЕННАЯ ВЕРСИЯ"""

self.logger.info(f"File closed: {file\_path} by {username}")

# Закрываем сессию в SessionManager

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

if session\_data:

# ВАЖНО: проверяем что ended\_at установлен

if 'ended\_at' not in session\_data or session\_data['ended\_at'] is None:

self.logger.error(f"❌ Session closed but ended\_at is not set for {file\_path}")

# Устанавливаем ended\_at если его нет

session\_data['ended\_at'] = datetime.now()

self.logger.info(f"✅ Manually set ended\_at to: {session\_data['ended\_at']}")

session\_duration = (session\_data['ended\_at'] - session\_data['started\_at']).total\_seconds()

event\_data = {

'file\_path': file\_path,

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

'event\_type': 'closed',

'file\_hash': file\_hash,

'user\_id': username,

'session\_id': session\_data['session\_id'],

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

'session\_duration': session\_duration,

'event\_timestamp': session\_data['ended\_at'].isoformat() # ИСПОЛЬЗУЕМ ВРЕМЯ ЗАКРЫТИЯ СЕССИИ

}

success = self.api\_client.send\_event(event\_data)

if success:

self.logger.info(f"✅ Successfully closed session for {file\_path} (duration: {session\_duration:.1f}s, ended\_at: {session\_data['ended\_at']})")

else:

self.logger.error(f"❌ Failed to send closed event for {file\_path}")

return success

else:

self.logger.warning(f"No active session found for closed file: {file\_path}")

return True

def \_should\_process\_file(self, file\_path: str, event\_type: str) -> bool:

"""Определяет нужно ли обрабатывать файл"""

if event\_type in ('deleted', 'closed') and not os.path.exists(file\_path):

return self.file\_validator.should\_monitor\_file\_by\_name(file\_path)

return self.file\_validator.should\_monitor\_file(file\_path)

def \_get\_file\_modifier\_safe(self, file\_path: str, event\_type: str) -> str:

"""Безопасное получение модификатора файла"""

try:

if not os.path.exists(file\_path) or self.\_is\_temporary\_file(file\_path):

return getpass.getuser()

return self.\_get\_file\_modifier(file\_path)

except Exception as e:

self.logger.warning(f"Failed to get file modifier for {file\_path}, using current user: {e}")

return getpass.getuser()

def \_is\_temporary\_file(self, file\_path: str) -> bool:

"""Проверяет является ли файл временным"""

filename = os.path.basename(file\_path).lower()

temp\_patterns = ['~wr', '~$', '.tmp', '.temp']

return any(pattern in filename for pattern in temp\_patterns)

def \_is\_temporary\_operation(self, src\_path: str, dest\_path: str) -> bool:

"""Определяет является ли операция временной"""

src\_name = os.path.basename(src\_path).lower()

dest\_name = os.path.basename(dest\_path).lower()

temp\_patterns = ['~wr', '~$', '.tmp', '.temp']

src\_is\_temp = any(pattern in src\_name for pattern in temp\_patterns)

dest\_is\_temp = any(pattern in dest\_name for pattern in temp\_patterns)

return src\_is\_temp or dest\_is\_temp

def \_handle\_file\_moved(self, src\_path: str, dest\_path: str) -> bool:

"""Обрабатывает перемещение/переименование файла"""

self.logger.info(f"File moved: {src\_path} -> {dest\_path}")

is\_temp\_operation = self.\_is\_temporary\_operation(src\_path, dest\_path)

if is\_temp\_operation:

self.logger.debug(f"Detected temporary file operation: {src\_path} -> {dest\_path}")

return True

username = self.\_get\_file\_modifier\_safe(dest\_path, 'moved')

normalized\_username = self.\_normalize\_username(username) # НОРМАЛИЗУЕМ имя пользователя

self.file\_renames[src\_path] = dest\_path

file\_hash = None

if self.config.get('hashing', {}).get('enabled', True):

file\_hash = self.hash\_calculator.calculate\_file\_hash\_with\_retry(dest\_path)

# Обновляем сессию для нового пути файла

old\_session\_key = f"{src\_path}:{normalized\_username}"

if old\_session\_key in self.session\_manager.active\_sessions:

old\_session = self.session\_manager.close\_session(src\_path, normalized\_username)

if old\_session:

self.session\_manager.smart\_create\_session(dest\_path, normalized\_username, file\_hash)

event\_data = {

'file\_path': dest\_path,

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

'old\_file\_path': src\_path,

'old\_file\_name': os.path.basename(src\_path),

'event\_type': 'moved',

'file\_hash': file\_hash,

'user\_id': normalized\_username, # ИСПОЛЬЗУЕМ НОРМАЛИЗОВАННОЕ ИМЯ

'event\_timestamp': datetime.now().isoformat()

}

success = self.api\_client.send\_event(event\_data)

if not success:

self.logger.error(f"Failed to send moved event for {src\_path} -> {dest\_path}")

return success

def \_get\_file\_modifier(self, file\_path: str) -> str:

"""Получает пользователя, изменившего файл"""

try:

if platform.system() == 'Windows' and win32security is not None:

sd = win32security.GetFileSecurity(file\_path, win32security.OWNER\_SECURITY\_INFORMATION)

owner\_sid = sd.GetSecurityDescriptorOwner()

name, domain, \_ = win32security.LookupAccountSid(None, owner\_sid)

# Возвращаем полное имя с доменом (будет нормализовано позже)

return f"{domain}\\{name}"

else:

return getpass.getuser()

except Exception as e:

self.logger.error(f"Failed to get file modifier for {file\_path}: {e}")

return getpass.getuser()

def \_handle\_file\_created(self, file\_path: str, username: str, file\_hash: str) -> bool:

"""Обрабатывает создание файла"""

if file\_path in self.file\_renames.values():

self.logger.debug(f"Ignoring created event for moved file: {file\_path}")

return True

self.logger.info(f"File created: {file\_path} by {username}")

session\_data = self.session\_manager.smart\_create\_session(file\_path, username, file\_hash)

if session\_data.get('resume\_count', 0) > 0:

self.stats['sessions\_resumed'] += 1

self.logger.info(f"Session resumed for {file\_path} (resume count: {session\_data['resume\_count']})")

else:

self.stats['sessions\_created'] += 1

event\_data = {

'file\_path': file\_path,

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

'event\_type': 'created',

'file\_hash': file\_hash,

'user\_id': username, # УЖЕ НОРМАЛИЗОВАННОЕ ИМЯ

'session\_id': session\_data['session\_id'],

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

'event\_timestamp': datetime.now().isoformat()

}

success = self.api\_client.send\_event(event\_data)

if not success:

self.logger.error(f"Failed to send created event for {file\_path}: {event\_data}")

return success

def \_handle\_file\_modified(self, file\_path: str, username: str, file\_hash: str) -> bool:

"""Обрабатывает изменение файла"""

self.logger.debug(f"File modified: {file\_path} by {username}")

session\_data = self.session\_manager.smart\_create\_session(file\_path, username, file\_hash)

event\_data = {

'file\_path': file\_path,

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

'event\_type': 'modified',

'file\_hash': file\_hash,

'user\_id': username, # УЖЕ НОРМАЛИЗОВАННОЕ ИМЯ

'session\_id': session\_data['session\_id'],

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

'event\_timestamp': datetime.now().isoformat()

}

success = self.api\_client.send\_event(event\_data)

if not success:

self.logger.error(f"Failed to send modified event for {file\_path}: {event\_data}")

return success

def \_handle\_file\_deleted(self, file\_path: str, username: str) -> bool:

"""Обрабатывает удаление файла"""

# Проверяем, не является ли это частью перемещения

if file\_path in self.file\_renames:

self.logger.debug(f"📦 File moved, closing session for: {file\_path}")

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

if session\_data:

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

return True

self.logger.info(f"🗑️ File deleted: {file\_path} by {username}")

self.stats['files\_deleted'] += 1

# Удаляем из отслеживания открытых файлов

if file\_path in self.open\_files:

del self.open\_files[file\_path]

# Пытаемся закрыть сессию обычным способом

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

# Если не нашли, используем принудительное закрытие

if not session\_data:

self.logger.info(f"🔄 Using forced session close for: {file\_path}")

closed\_sessions = self.session\_manager.close\_all\_sessions\_for\_file(file\_path)

if closed\_sessions:

session\_data = closed\_sessions[0]

self.logger.info(f"✅ Forced close: closed {len(closed\_sessions)} sessions")

if session\_data:

self.logger.info(f"✅ Successfully closed session for deleted file: {file\_path}")

event\_data = {

'file\_path': file\_path,

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

'event\_type': 'deleted',

'user\_id': username, # УЖЕ НОРМАЛИЗОВАННОЕ ИМЯ

'session\_id': session\_data['session\_id'],

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

'event\_timestamp': datetime.now().isoformat()

}

success = self.api\_client.send\_event(event\_data)

if not success:

self.logger.error(f"❌ Failed to send deleted event for: {file\_path}")

return success

else:

self.logger.warning(f"⚠️ No session found for deleted file: {file\_path}")

# Все равно отправляем событие удаления

event\_data = {

'file\_path': file\_path,

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

'event\_type': 'deleted',

'user\_id': username, # УЖЕ НОРМАЛИЗОВАННОЕ ИМЯ

'event\_timestamp': datetime.now().isoformat()

}

success = self.api\_client.send\_event(event\_data)

return success

def cleanup\_orphaned\_sessions(self):

"""Очищает сессии для файлов, которые больше не существуют"""

expired\_sessions = []

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

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

username = session\_data['username']

if not os.path.exists(file\_path):

self.logger.info(f"Closing orphaned session for deleted file: {file\_path}")

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

if closed\_session:

expired\_sessions.append(closed\_session)

event\_data = {

'file\_path': file\_path,

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

'event\_type': 'deleted',

'user\_id': username,

'session\_id': closed\_session['session\_id'],

'resume\_count': closed\_session.get('resume\_count', 0),

'event\_timestamp': datetime.now().isoformat()

}

self.api\_client.send\_event(event\_data)

return expired\_sessions

def get\_stats(self) -> Dict[str, Any]:

"""Возвращает статистику обработки"""

session\_stats = self.session\_manager.get\_session\_stats()

return {

\*\*self.stats,

\*\*session\_stats,

'open\_files\_tracking': len(self.open\_files)

}

def cleanup(self):

"""Очищает ресурсы"""

# Очищаем expired сессии

expired\_sessions = self.session\_manager.cleanup\_expired\_sessions(self)

for session\_data in expired\_sessions:

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

username = session\_data['username']

file\_hash = None

if os.path.exists(file\_path) and self.config.get('hashing', {}).get('enabled', True):

file\_hash = self.hash\_calculator.calculate\_file\_hash\_with\_retry(file\_path)

self.\_handle\_file\_closed(file\_path, username, file\_hash)

# Проверяем открытые файлы

self.check\_open\_files()

# Очищаем orphaned сессии

self.cleanup\_orphaned\_sessions()

import re

from typing import Listimport os

import fnmatch

import re

from typing import List

from shared.logger import setup\_logger

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

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

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

# ДОБАВЛЕНО: Список расширений для мониторинга

self.monitored\_extensions = self.config.get('file\_extensions', [

'.doc', '.docx', '.xls', '.xlsx', '.ppt', '.pptx',

'.pdf', '.txt', '.rtf', '.odt', '.ods', '.odp'

])

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# ДОБАВЛЕНО: Проверяем разрешенные расширения (только целевые файлы)

if self.monitored\_extensions and file\_ext not in self.monitored\_extensions:

self.logger.debug(f"Ignoring file - extension not in monitored list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

# ДОБАВЛЕНО: Проверяем размер файла (игнорируем слишком маленькие файлы)

try:

file\_size = os.path.getsize(file\_path)

if file\_size < 10: # Игнорируем файлы меньше 10 байт

self.logger.debug(f"Ignoring too small file: {filename} ({file\_size} bytes)")

return False

except (OSError, ValueError):

pass

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# ДОБАВЛЕНО: Проверяем разрешенные расширения (только целевые файлы)

if self.monitored\_extensions and file\_ext not in self.monitored\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in monitored list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

for root, dirs, files in os.walk(directory):

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

file\_path = os.path.join(root, file)

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return False

from shared.logger import setup\_logger

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

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

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

# ПРОВЕРКА ВРЕМЕННЫХ ФАЙЛОВ OFFICE - ДОБАВЛЕНО

if self.\_is\_office\_temporary\_file(filename):

self.logger.debug(f"Ignoring Office temporary file: {filename}")

return False

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

return True

def \_is\_office\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Office"""

filename\_lower = filename.lower()

# Паттерны временных файлов Office

office\_temp\_patterns = [

'~$', # Начинающиеся с ~$

'~wr', # Word recovery

'~wl', # Word lock

'~rf', # Office temporary

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы начинающиеся с ~

]

# Проверяем основные паттерны

for pattern in office\_temp\_patterns:

if filename\_lower.startswith(pattern) or filename\_lower.endswith(pattern):

return True

# Специфичные паттерны для Office

office\_specific = [

lambda x: x.startswith('~$') and '.xlsx' in x, # Excel временные

lambda x: x.startswith('~$') and '.docx' in x, # Word временные

lambda x: x.startswith('~$') and '.pptx' in x, # PowerPoint временные

lambda x: '~wr' in x and '.xlsx' in x, # Excel recovery

lambda x: '~wr' in x and '.docx' in x, # Word recovery

]

return any(pattern(filename\_lower) for pattern in office\_specific)

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# ПРОВЕРКА ВРЕМЕННЫХ ФАЙЛОВ OFFICE - ДОБАВЛЕНО

if self.\_is\_office\_temporary\_file(filename):

self.logger.debug(f"Ignoring Office temporary file: {filename}")

return False

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

for root, dirs, files in os.walk(directory):

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

file\_path = os.path.join(root, file)

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return False